AI Powered Search

Trey Grainger
Doug Turnbull
Max Irwin

MANNING

MEAP Edition
Manning Early Access Program

AI-Powered Search

Version 13

welcome

Thanks for purchasing the MEAP for *AI-Powered Search*!

This book teaches you the knowledge and skills you need to deliver highly-intelligent search applications that are able to automatically learn from every content update and user interaction, delivering continuously more relevant search results.

As you can imagine given that goal, this is not an "introduction to search" book. In order to get the most out of this book, you should ideally already be familiar with the core capabilities of modern search engines (inverted indices, relevance ranking, faceting, query parsing, text analysis, and so on) through experience with a technology like Apache Solr, Elasticsearch/OpenSearch, Vespa, or Apache Lucene. If you need to come up to speed quickly, *Solr in Action* (which I also wrote) provides you with all the search background necessary to dive head-first into *AI-Powered Search*.

Additionally, the code examples in this book are written in Python (and delivered in pre-configured Jupyter notebooks) to appeal both to engineers and data scientists. You don't need to be an expert in Python, but you should have some programming experience to be able to read and understand the examples.

Over my career, I've had the opportunity to dive deep into search relevance, semantic search, recommendations, behavioral signals processing, learning to rank, dense vector search, and many other AI-powered search capabilities, publishing cutting-edge research in top journals and conferences and, more importantly, delivering working software at massive scale. As Founder of Searchkernel and as Lucidworks' former Chief Algorithms Officer and SVP of Engineering, I've also helped deliver many of these capabilities to hundreds of the most innovative companies in the world to help them power search experiences you probably use every single day.

I'm thrilled to also have Doug Turnbull (Shopify) and Max Irwin (OpenSource Connections) as contributing authors on this book, pulling from their many years of hands-on experience helping companies and clients with search and relevance engineering. Doug is contributing chapters 10-12 about building machine-learned ranking models (Learning to Rank) and automating their training using click models, and Max is contributing chapters 13-14 on dense vector search, question answering, and the search frontier.

In this book, we distill our decades of combined experience into a practical guide to help you take your search applications to the next level. You'll discover how to enable your applications to continually learn to better understand your content, users, and domain in order to deliver optimally-relevant experiences with each and every user interaction. We're working steadily on the book, and readers should expect a new chapter to arrive about every 1 to 2 months.

By purchasing the MEAP of *AI-Powered Search,* you gain early access to written chapters, and well as the ability to provide input into what goes into the book as it is being written. If you have any comments or questions along the way, please direct them to Manning's liveBook Discussion Forum for the book.

I would greatly appreciate your feedback and suggestions, as they will be invaluable toward making this book all it can be. Thanks again for purchasing the MEAP, thank you in advance for your input, and best wishes as you begin putting *AI-Powered Search* into practice!

—Trey Grainger

brief contents

1 Introducing AI-powered search

This chapter covers

- **The need for AI-powered search**
- **The dimensions of user intent**
- **Foundational technologies for building AI-powered Search**
- **How AI-powered search works**

The search box has rapidly become the default user interface for interacting with data in most modern applications. If you think of every major app or website you visit on a daily basis, one of the first things you likely do on each visit is type or speak a query in order to find the content or actions most relevant to you in that moment.

Even in scenarios where you are not explicitly searching, you may instead be consuming streams of content customized for your particular tastes and interests. Whether these be video recommendations, items for purchase, e-mails sorted by priority or recency, news articles, or other content, you are likely still looking at filtered or ranked results and given the option to either page through or explictly filter the content with your own query.

Whereas the phrase "search engine" to most people brings up thoughts of a website like Google, Baidu, or Bing, that enables queries based upon a crawl of the entire public internet, the reality is that search is now ubiquitous - it is a tool present and available in nearly all of our digital interactions every day across the numerous websites and applications we use.

Furthermore, while not too long ago the expected response from a search box may have been simply returning "ten blue links" - a list of ranked documents for a user to investigate to find further information in response to their query - expectations for the intelligence-level of search technologies have sky-rocketed in recent years.

Today’s search capabilities are expected to be:

- *Domain-aware*: understanding the entities, terminology, categories, and attributes of each specific use case and corpus of documents, not just leveraging generic statitistics on strings of text.
- *Contextual & Personalized*: able to take user context (location, last search, profile, previous interactions, user recommendations, and user classification), query context (other keywords, similar searches), and domain context (inventory, business rules, domain-specific terminology) in order to better understand user intent.
- *Conversational*: able to interact in natural language and guide users through a multi-step discovery process while learning and remembering relevant new information along the way.
- *Multi-modal*: able to resolve text queries, voice queries, search using images or video, or even monitor for events and send event-based pushed notifications.
- *Intelligent*: able to deliver predictive type-ahead, to understand what users mean (spelling correction, phrase and attribute detection, intent classification, conceptual searching, and so on) to deliver the right answers at the right time and to be constantly getting smarter.
- *Assistive*: moving beyond just delivery of links to delivering answers and available actions.

The goal of AI-powered search is to leverage automated learning techniques to deliver on these desired capabilities. While many organizations start with basic text search and spend many years trying to manually optimize synonyms lists, business rules, ontologies, field weights, and countless other aspects of their search configuration, some are beginning to realize that most of this process can actually be automated.

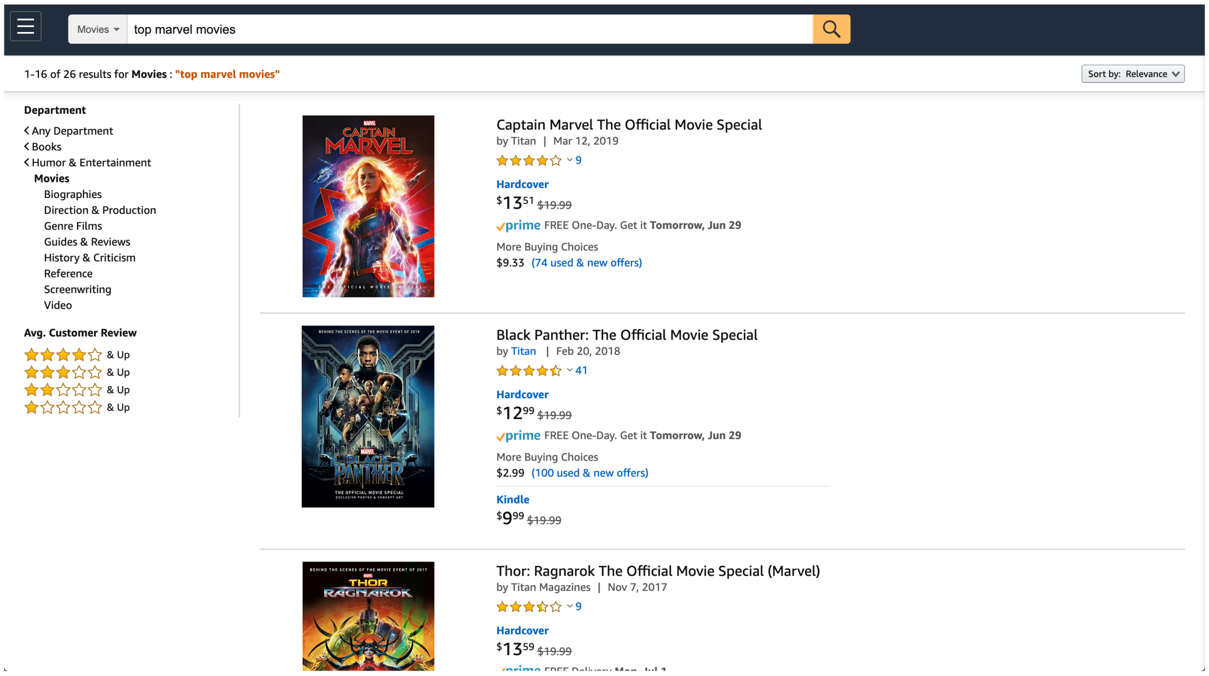
This book is an example-driven guide through the most applicable machine learning algorithms and techniques commonly leveraged to build intelligent search systems. We’ll not only walk through key concepts, but will also provide reusable code examples to cover data collection and processing techniques, as well as the self-learning query interpretation and relevance strategies employed to deliver AI-powered search capabilities across today’s leading organizations - hopefully soon to include your own!

1.1 Searching for User Intent

In order to deliver AI-powered search, it’s important that we establish up-front a cohesive understanding of the many dimensions involved in interpreting user intent and subsequently returning content matching that intent. Within the field of information retrieval, search engines and recommendation engines are the two most popular technologies employed in an attempt to deliver users the content needed to satisfy their information need. Many organizations think of search engines and recommendation engines as separate technologies solving different use cases, and indeed it is often the case that different teams within the same organization - often with different skillsets - work independently on separate search engines and recommendation engines. In this section, we’ll dive into why separating search and recommendations into separate silos (independent functions and teams) can often lead to less than ideal outcomes.

1.1.1 Search Engines

A search engine is typically thought of as a technology for explictly entering queries and receiving a response. It is usually exposed to end users through some sort of text box into which a user can enter keywords or full natural-language questions, and the results are usually returned in a list alongside additional filtering options that enable further refinement of the initial query. Using this mechanism, search is leveraged as a tool for directed discovery of relevant content. Whenever a user is finished with their search session, however, they can usually issue a new query and start with a blank slate, wiping away any context from the previous search.



top marvel movies

1-16 of 26 results for Movies : "top marvel movies"

Captain Marvel The Official Movie Special
by Titan | Mar 12, 2019
9
Hardcover
$13.51 $19.99
prime FREE One-Day. Get it Tomorrow, Jun 29
More Buying Choices
$9.33 (74 used & new offers)

Black Panther: The Official Movie Special
by Titan | Feb 20, 2018
41
Hardcover
$12.99 $19.99
prime FREE One-Day. Get it Tomorrow, Jun 29
More Buying Choices
$2.99 (100 used & new offers)
Kindle
$9.99 $19.99

Thor: Ragnarok The Official Movie Special (Marvel)
by Titan Magazines | Nov 7, 2017
9
Hardcover
$13.59 $19.99

Figure 1.1 A typical search experience, with a user entering a query and seeing search results with filtering options to support further refinement of search results

Within the software engineering world, a search engine is one of the most cross-functional kinds of systems in existence. Most underlying search engine technology is designed to operate in a massively scalable way - scaling to millions, billions, or in a select cases trillions of documents, handling enormous volumes of concurrent queries, and delivering search results in hundreds of milliseconds or often less. In many cases, real-time processing and near-real-time searching on newly ingested data is also required, and all of this must be massively parallelizable across numerous servers in order to scale out to meet such high performance requirements.

Needless to say, engineering the above kind of system requires strong back-end developers with a solid understanding of distributed systems, concurrency, data structures, operating systems, and high-performance computing.

Search engines also require substantial work building search specific data structures like an inverted index, an understanding of linear algebra and vector similarity scoring, experience with text analysis and natural language processing, and knowledge of numerous search-specific kinds of data models and capabilities (spell checking, autosuggest, faceting, text highlighting, and so on).

Further, as relevancy becomes more important, search teams often employ relevance engineers who live and breathe the domain of their company, and who think in terms of ranking models, A/B testing, click-models, feature engineering, and query interpretation and intent discovery.

Of course, there are product managers driving requirements, data analysts assessing search quality and improvements, DevOps engineers running and scaling the search platform, and numerous other supporting roles surrounding a production search engine implementation.

It is rare to find someone with multiple of these skillsets, let alone all of them, making search a highly cross-functional discipline. In order for a search engine to fully interpret user intent, however, being able to combine a thorough understanding of your content, your users, and your domain is critical. We’ll revisit why this is important after briefly discussing the related topic of recommendation engines.

1.1.2 Recommendation Engines

Whereas search engines are traditionally thought of as technology that requires explicit user-driven queries, most people in contrast think of recommendation engines as systems which don’t accept user input and only push out content to users based upon what it knows about them and believes best matches their computed interests. Both of these are oversimplifications, as we’ll see in section 1.1.3, but they are nevertheless the prevailing ways in which search engines and recommendation engines are commonly distinguished. If you routinely visit Amazon.com or any other major ecommerce website, you are no doubt familiar with recommendation engines sections stating that "based on your interest in this item, you may also like …" or otherwise just recommending a list of items based upon your collective browsing and purchase history, similar to Figure 1.2. These recommendations often drive significant revenue for companies, and they help customers discover relevant, personalized, and related content that often complements what they were looking for explicitly.

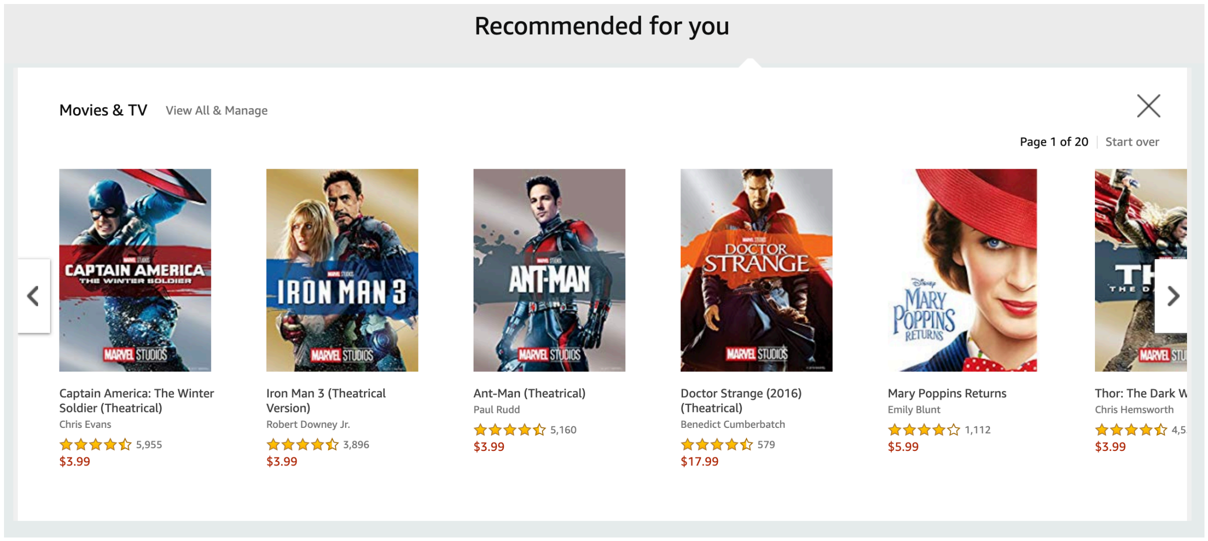


Figure 1.2 Recommendations based upon purchase patterns of user expressing interest in similar items

Recommendation engines, also commonly referred to as recommendation systems, come in many shapes and sizes. They employ algorithms which can take in inputs (user preferences, user behavior, content, and so on) and leverage those inputs to automatically match the most relevant content. Recommendation algorithms can roughly be divided into three categories: Content-based Recommenders, Behavior-based Recommenders, and Multi-modal Recommenders.

CONTENT-BASED RECOMMENDERS

These algorithms recommend new content based upon attributes shared between different entities (often between users and items, between items and items, or between users and users). For example, imagine a job search website. Jobs may have properties on them like "job title", "industry", "salary range", "years of experience", and "skills", and users will also have similar attributes on their profile or resume/CV. Based upon this, a content-based recommendation algorithm can figure out which of these features matter the most, and can then rank the best matching jobs for any given user based upon how well that job matches the user's desired attributes. This is what's known as an user-item (or "user to item") recommender.

Similarly, if a user likes a particular job, it is possible to leverage this same process to recommend similar other jobs based upon how well those jobs match the attributes for the first job that is known to be good. This type of recommendation is popular on product details pages, where a user is already looking at an item and it may be desirable to help them explore additional related items. This kind of a recommendation algorithm is known as an item-item (or "item to item") recommender.

Figure 1.3 demonstrates how a content-based recommender might leverage attributes about items with which a user has previously interacted in order to match similar items for that user. In this case, our user viewed the "detergent" product, and then the recommendation algorithm suggests "fabric softener" and "dryer sheets" products based upon them matching within the same category field (the "laundry" category) and containing similar text to the "detergent" product within their product descriptions.

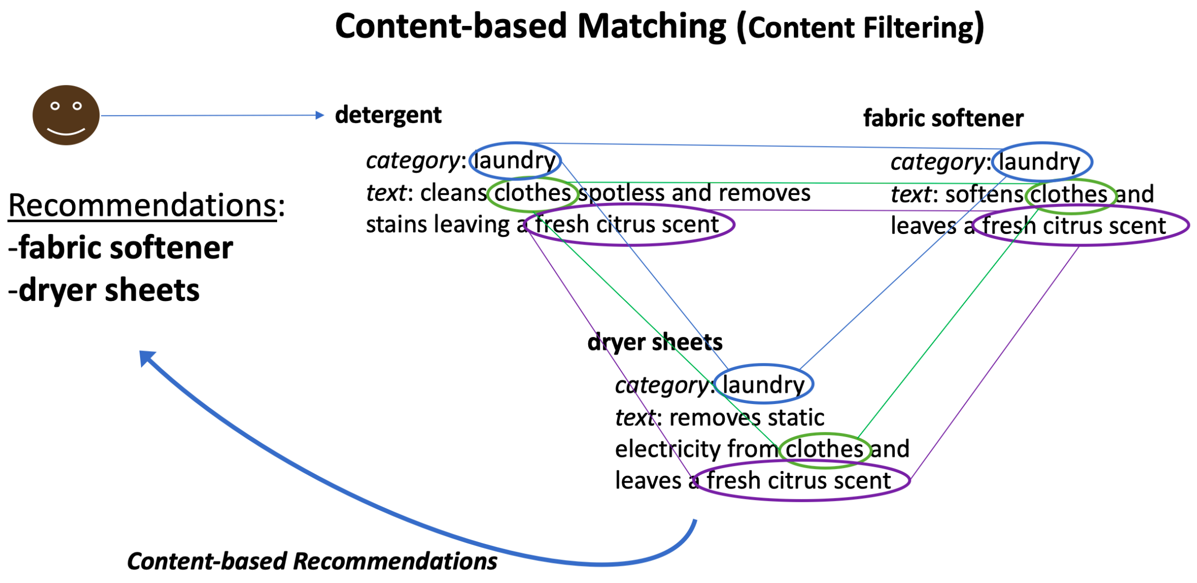

Content-based Matching (Content Filtering)
detergent
category: laundry
text: cleans clothes spotless and removes stains leaving a fresh citrus scent
fabric softener
category: laundry
text: softens clothes and leaves a fresh citrus scent
dryer sheets
category: laundry
text: removes static electricity from clothes and leaves a fresh citrus scent
Recommendations:
-fabric softener
-dryer sheets
Content-based Recommendations

Figure 1.3 Content-based recommendations based upon matching attributes of an item of interest to a user, such as categories and text keywords.

It is also possible to match users to other users or really any entity to any other entity. In the context of content-based recommenders, all recommendations can be seen as "item-item" recommendations, where each item is just an arbitrary entity that shares attributes with the other entities being recommended.

BEHAVIOR-BASED RECOMMENDERS

These algorithms leverage user interactions with items (documents) to discover similar patterns of interest among groups of items. This process is called *collaborative filtering*, referring to the use of a multiple-person (collaborative) voting process to filter matches to those demonstrating the highest similarity, as measured by how many overlapping users interacted with the same items. The intuition here is that similar users (i.e. those with the similar preferences) tend to interact with the same items, and that when users interact with multiple items they are more likely to be looking for and interacting with similar items as opposed to unrelated items.

One amazing characteristic of collaborative filtering algorithms is that they fully crowd-source the relevance scoring process to your end users. In fact, features of the items themselves (name, brand, colors, text, and so on) are not needed - all that is required is a unique ID for each item, and knowledge of which users interacted with which items. Further, the more users and interactions (known as "*behavior signals*" or just "*signals*") you have, the smarter these algorithms tend to get, because you have more people continually voting and informing your scoring algorithm. This often leads to collaborative filtering algorithms significantly outperforming content-based algorithms.

Figure 1.4 demonstrates how overlapping behavioral signals between multiple users can be used to drive collaborative recommendations

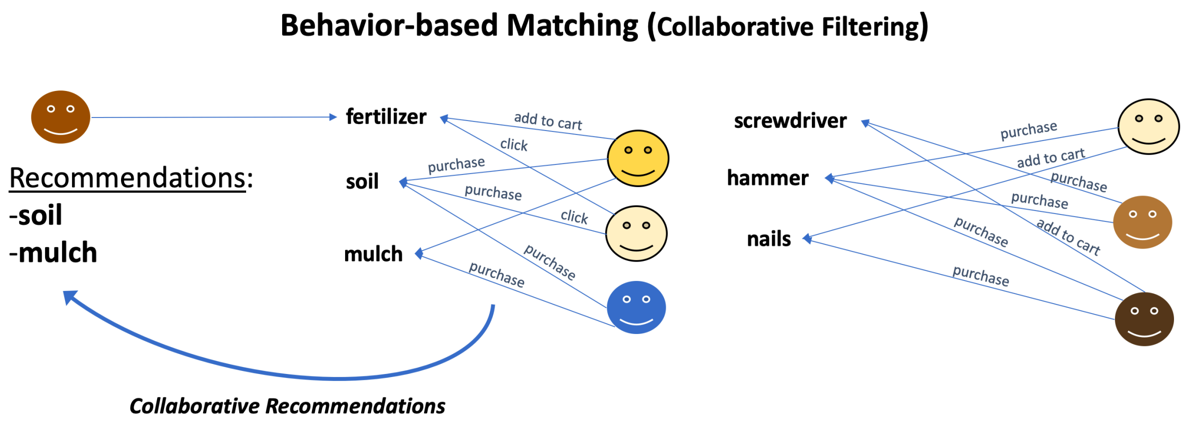


Figure 1.4 Recommendations based upon collaborative filtering, a technique leveraging the overlap between behavioral signals across multiple users.

Unfortunately, the same dependence upon user behavioral signals that makes collaborative filtering so powerful also turns out to be their Achilles' heel. What happens when there are only a few interactions with a particular item—or possibly none at all? The answer is that the item either never gets recommended (when there are no signals), or otherwise will likely to generate poor recommendations or show up as a bad match for other items (when there are few signals). This situation is known as the *cold start problem*, and is a major challenge for behavior-based recommenders.

MULTI-MODAL RECOMMENDERS

Sometimes, you can have your cake and eat it, too. Such is the case with multi-modal recommenders, which attempt to combine the best of both the content-based and behavior-based recommender approaches. Since collaborative filtering tends to work best for items with many signals, but poorly when few or no signals are present, it is most effective to use both content-based features as a baseline and then to layer a model based on behavioral signals on top. This way, if few signals are present, the content-based matcher will still return results, while if there are many signals, then the collaborative filtering matches will take greater prominence. We'll cover methods for combining approaches in chapter 6, but it is easy to infer already that including some results from both approaches can give you the best of both worlds: high quality crowd-sourced matching, while avoiding the cold start problem for newer and less-well-discovered content. Figure 1.5 demonstrates how this can work in practice.

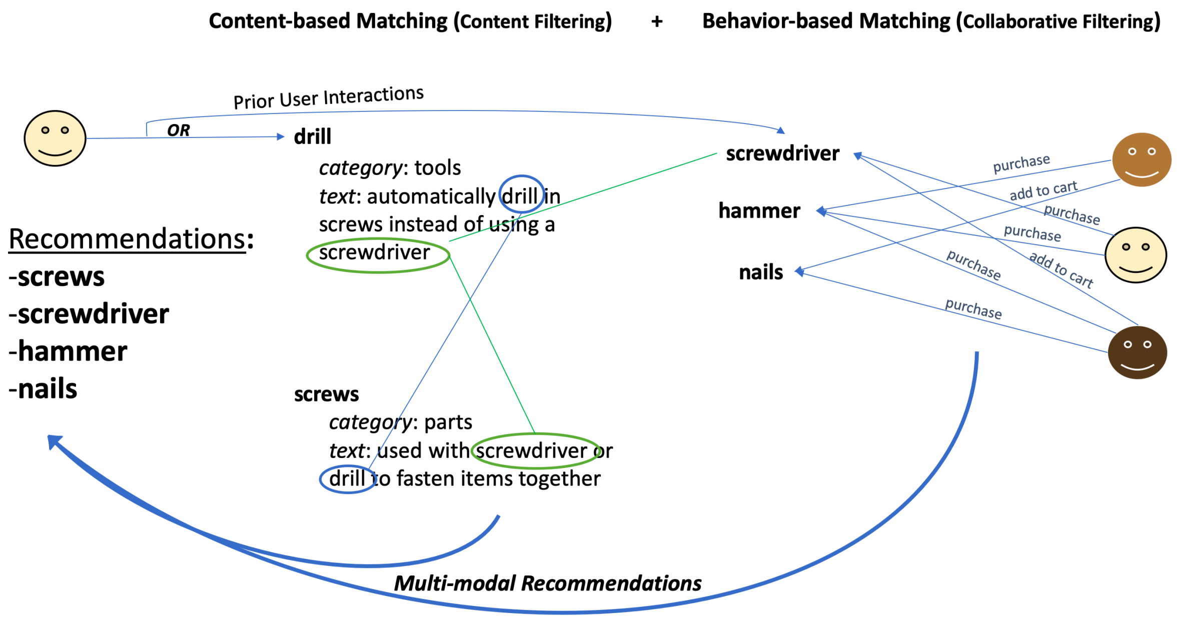

Content-based Matching (Content Filtering) + Behavior-based Matching (Collaborative Filtering)
Prior User Interactions
OR
drill
category: tools
text: automatically drill in screws instead of using a screwdriver
Recommendations:
-screws
-screwdriver
-hammer
-nails
screws
category: parts
text: used with screwdriver or drill to fasten items together
screwdriver
hammer
nails
purchase
add to cart
purchase
purchase
purchase
add to cart
purchase
Multi-modal Recommendations

Figure 1.5 Multi-modal recommendations, which combine both content-based matching and collaborative filtering into a hybrid matching algorithm.

You can see in Figure 1.5 that the user could interact with either the drill (which has no signals and can thus only be used in recommendations through a content-based matcher) or the screwdriver (which has previous signals from other users, as well as content), and the user would receive recommendations in both cases. This provides the benefit that signals-based collaborative filtering can be used, while also enabling content-based matching for items where insufficient signals are currently available, such as is the case with the drill in this example. This kind of multi-modal recommendation algorithm provides significant flexibility and advantage over only leveraging content-based or behavior-based recommendations alone.

1.1.3 The Information Retrieval Continuum

We just finished discussing two different kinds of information retrieval systems: Search Engines and Recommendation Engines. The key differentiating factor between the two was that search engines are typically guided by users and match their explicitly-entered queries, whereas recommendation engines typically accept no direct user input and instead recommend—based upon already known or inferred knowledge—what a user may want to see next.

The reality, however, is that these two kinds of systems are really just two sides of the same coin. The goal, in both cases, is to understand what a user is looking for, and to deliver relevant results to meet that user's information need. In this section, we'll discuss the broad spectrum of capabilities that lie within the information retrieval continuum between search and recommendation systems

PERSONALIZED SEARCH

Let's imagine running a restaurant search engine. Our user, Michelle, is on her phone in New York at lunch time and she types in a keyword search for "steamed bagels". She sees top-rated steamed bagel shops in Greenville, South Carolina (USA), Columbus, Ohio (USA), and London (UK).

What's wrong with these search results? Well, in this case, the answer is fairly obvious—Michelle is looking for lunch in New York, but the search engine is showing her results hundreds to thousands of miles away. But Michelle never *told* the search engine she only wanted to see results in New York, nor did she tell the search engine that she was looking for a lunch place close by because she wants to eat now…

Consider another scenario - Michelle is at the airport after a long flight and she searches her phone for "driver". The top results that come back are for a golf club for hitting the ball off a tee, followed by a link to printer drivers, followed by a screwdriver. If the search engine knows Michelle's location, shouldn't it be able to infer her intended meaning?

Using our job search example from earlier, let's assume Michelle goes to her favorite job search engine and types in "nursing jobs". Similar to our restaurant example earlier, wouldn't it be ideal if nursing jobs in New York showed up at the top of the list? What if she later types in "jobs in Seattle"? Wouldn't it be ideal if—instead of seeing random jobs in Seattle (doctor, engineer, chef, etc.)— nursing jobs now showed up at the top of the list, since the engine previously learned that she is a nurse?

Each of the above are examples of personalized search - the process of combining both explicit user input *and* an implicit understanding of each user's specific intent and preferences, into a query that can return a set of results specifically catering to that user. Doing this well is a tricky subject, as you have to carefully balance leveraging your understanding of the user without overriding anything for which they explicitly want to query. We'll dive into how to gracefully balance these concerns later in chapter 9.

USER-GUIDED RECOMMENDATIONS

Just as it is possible to sprinkle in implicit and user-specific attributes into a search to generate personalized search, it is also possible to enable user-guided recommendations.

It is becoming increasingly more common for recommendation engines to allow users to see and edit their recommendation preferences. These preferences usually include a list of items the user has interacted with before, such as watched movies and ratings for a movie recommendation engine, or a list of previously viewed or purchased items for an ecommerce website. Across a wide array of use cases, these preferences could also include things like clothing sizes, brand affinities, favorite colors, prefered local store, favorite menu items, desired job titles and skills, prefered salary ranges, etc. In essence, these recommendation preferences make up a user profile - they define what is known about a customer, and the more control you can give a user to see, adjust, and improve this profile, the better you'll be able to understand your users and the happier they'll likely be.

SEARCH VS. RECOMMENDATIONS: THE FALSE DICHOTOMY

We've seen that when trying to find content for your end users, that personalization profile information can either be ignored (traditional keyword search), used implicity along with other explicit user input (personalized search), used explicitly with the ability for a user to adjust (user-guided recommendations), or used explicitly with no ability for a user to adjust (traditional recommendations). Figure 1.6 shows this personalization spectrum.

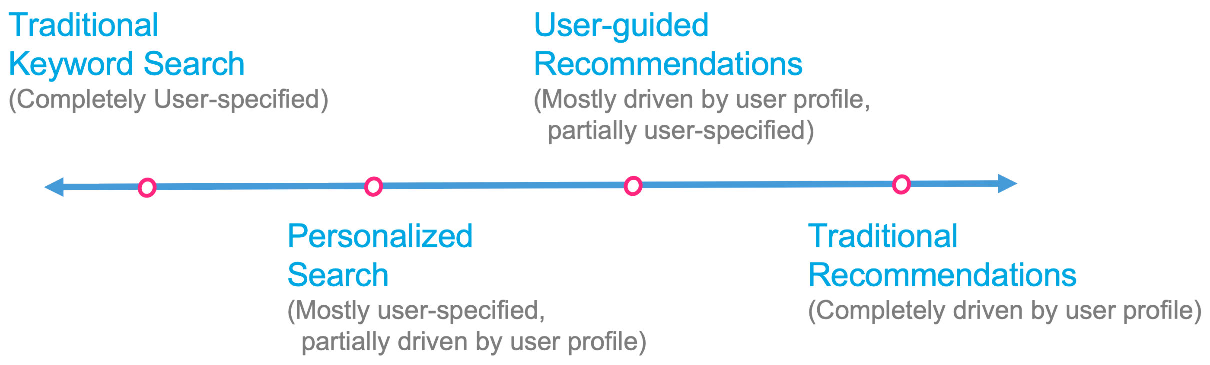

Traditional Keyword Search
(Completely User-specified)
User-guided Recommendations
(Mostly driven by user profile, partially user-specified)
Personalized Search
(Mostly user-specified, partially driven by user profile)
Traditional Recommendations
(Completely driven by user profile)

Figure 1.6 The personalization spectrum, showing traditional keyword search and traditional recommendations as simply two ends of a larger continuum

While the two ends of this personalization spectrum (traditional keyword search, and traditional recommendations) represent the extremes, they are also the two most common. Unfortunately, one of the biggest mistakes that I see in many organizations is the building of teams and organizations around the belief that search and recommendations are separate problems. This often leads to data science teams building complicated personalization and segmentation models that can only recommend content and can’t do search at all, and for engineering teams to build large-scale keyword matching engines that can’t take easy advantage of the robust personalization models built by the recommendations teams. More often than not, the recommendation teams are staffed by Data Scientists with minimal information retrieval background, and the search teams are often staffed by engineers with minimal data science background. Due to Conway’s Law ("organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations."), this ultimately results in challenges solving problems along this personalization spectrum (particularly in the middle) that need the best from both teams. In this scenario, more often than not the search and recommendation engines will naturally evolve into separate systems that can’t as easily maximize for relevance needs across the full personalization spectrum. In this book, we’ll focus on the shared techniques that enable search to become smarter and for recommendations to become more flexible through a unified approach, as an AI-powered search platform needs to be able to continuously learn from both your users and your content and enable your users to do the same.

1.1.4 Semantic Search and Knowledge Graphs

While we’ve presented search and recommendations as a personalization spectrum in Figure 1.6, with personalized search and user-guided recommendations in-between, there’s still one more dimension that is critical for building a good AI-powered search system—a deep understanding of the given domain. It’s not enough to match on keywords and to recommend content based upon how users collectively interact with different documents. To build a truly smart search system, it is important for the engine to learn as much as it can about the domain. This includes learning all of the important domain-specific phrases, synonyms, and related terms, as well as identifying entities in documents and queries, generating a knowledge graphs that relates those entities, disambiguating the many nuanced meanings represented by domain-specific terminology, and ultimately being able to effectively parse, interpret, and conceptually match the nuanced intent of users within your domain. Figure 1.7 shows an example of a semantic parsing of a query, with the goal being to search for "things" (known entities) instead of "strings" (just text matching).

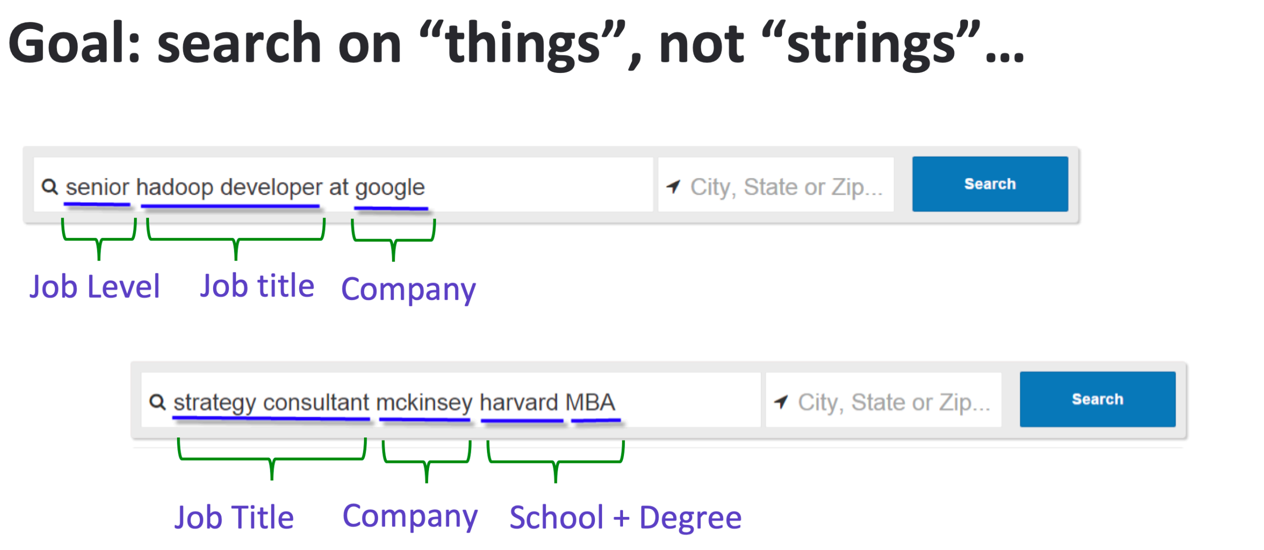

Goal: search on "things", not "strings"…
senior hadoop developer at google
City, State or Zip...
Search
Job Level
Job title
Company
strategy consultant mckinsey harvard MBA
City, State or Zip...
Search
Job Title
Company
School + Degree

Figure 1.7 Semantic parsing of a query, demonstrating an understanding of the entities ("things") represented by query terms

Many companies have spent considerable money building out large teams to manually create dictionaries and knowledge graphs in an attempt to understand the relationships between entities in their users' queries in order to make their search smarter. This book will focus on a more scalable approach, however - building an AI-powered search engine that can automatically learn all of the above on a continuous basis. We'll dive deep into the topic of automatically generating this domain-specific intelligence when we cover semantic search further in chapters 5, 6, and 7. We'll also dive into additional emerging techniques for conceptual search, such as dense vector search and neural search in chapters 13 and 14.

1.1.5 Understanding the Dimensions of User Intent

We've discussed the important role of traditional keyword search, of recommendations, and of the personalization spectrum in-between. We also discussed the need for Semantic Search (leveraging a Knowledge Graph) to be able to provide domain-specific understanding of your content and your user's queries. All of these are key pillars of a singular, larger goal, however: to fully understand user intent. Figure 1.8 provides a comprehensive model demonstrating the interplay between each of these key pillars of user intent.

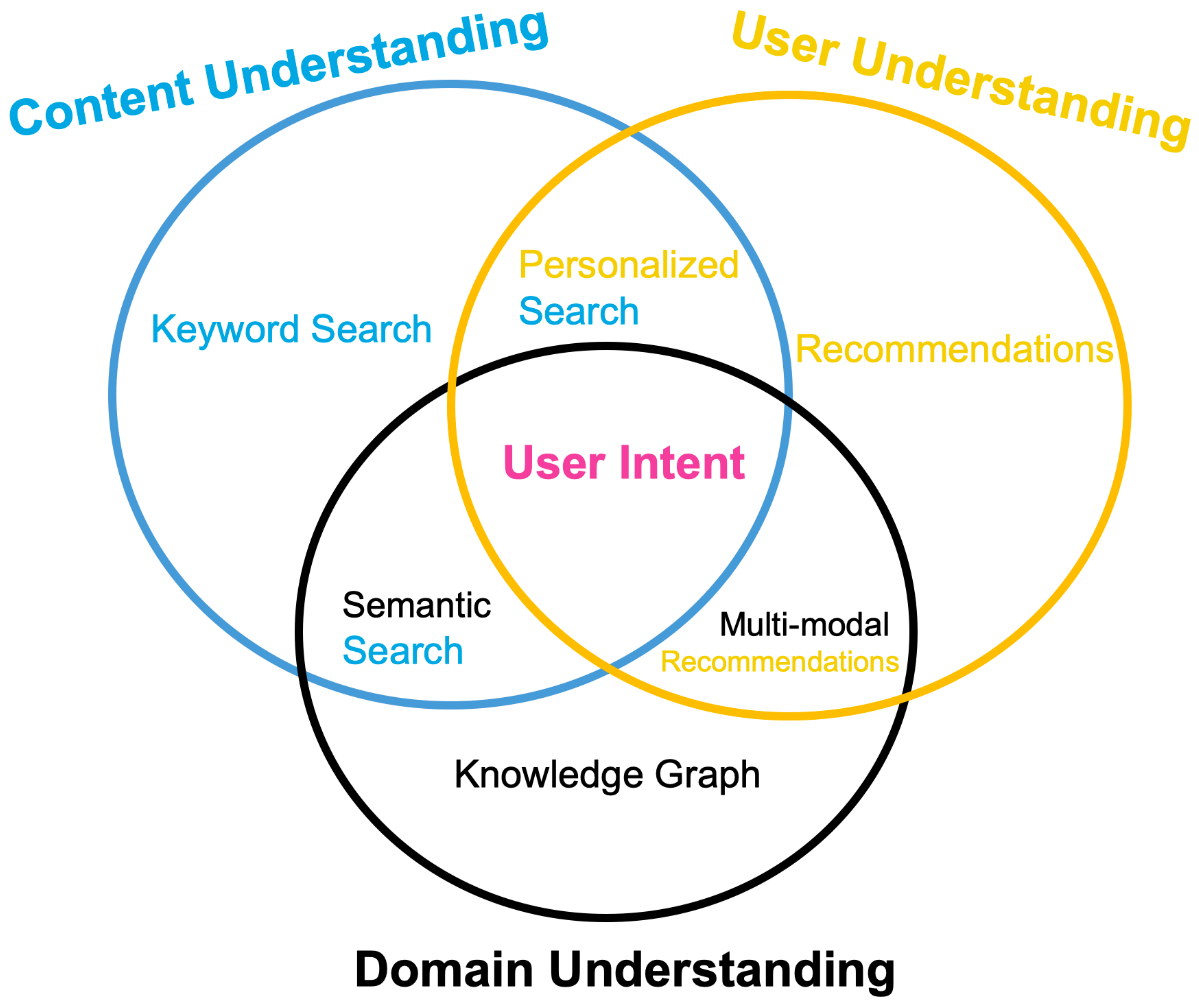


Figure 1.8 The dimensions of user intent, a combination of content understanding, user understanding, and domain understanding.

The top-left circle in Figure 1.8 represents "content understanding" - the ability to find any arbitrary content leveraging keywords and matching on attributes. The top-right circle represents "user understanding" - the ability to understand each user’s specific preferences and leverage those to return more personalized results. Finally, the lower circle represents "domain understanding" - the ability to interpret words, phrases, concepts, entities, and nuanced interpretations and relationships between each of these within your own domain-specific context.

A query only in the "content understanding" circle represents traditional *keyword search*, enabling matching on keywords but without leveraging any domain- or user-specific context. A query only in the "user understanding" circle would be generated recommendations from collaborative filtering, with no ability for the user to override the inputs and no understanding of the domain or content underlying documents. A query only in the "domain understanding" bucket might be a structured query only on known tags, categories, or entities, or even a browse-like interface that enabled exploration of a *knowledge graph* of these domain-specific entities and their relationships, but without any user specific personalization or ability to find arbitrary terms, phrases, and content.

When traditional keyword search and recommendations overlap, we get *personalized search* or guided recommendations (discussed earlier). When traditional keyword search and knowledge graphs overlap, we get *semantic search* (also discussed earlier in section 1.1.3): a smart, domain-specific search experience. Finally, when recommendations and knowledge graphs overlap we get smarter *multi-modal recommendations* that not only match on the crowd-sourced interactions of users across similar documents, but also on a domain-specific understanding of the important attributes of those documents.

The holy grail for AI-powered search, however, is to harness the intersection of all three categories: personalized search, semantic search, and multi-modal recommendations. That is to say, in order to truly understand user intent, we need an expert understanding of the domain the user is searching, an expert understanding of the user and their preferences, and an expert ability to match and rank arbitrary queries against any content. AI-powered search starts with the three pillars of user intent (content, domain, and user), and then leverages intelligent algorithms to constantly learn and improve in each of these areas automatically. This includes techniques like automatically learning ranking criteria (see chapters 10-11), automatically learning user preferences (see chapter 9), and automatically learning knowledge graphs of the represented domain (see chapters 5-6). At the end of the day, the combination of each of these approaches - and balancing them appropriately against each other - provides the key to optimal understanding of users and their query intent, which is the end goals of our AI-powered search system.

1.2 Key Technologies for AI-powered search

This book is a guide for readers to take their search applications to the next level and build highly-relevant, self-learning search systems. In order to get to the "next level", however, this assumes readers have prior experience with or are already familiar with key concepts behind traditional keyword search engines and modern data processing platforms.

While the techniques in this book are broadly applicable for use in most open source and commercial search engines, we have chosen to adopt the following key technologies for the examples in this book:

- *Programming Language*: Python
- *Data Processing Framework*: Spark (PySpark)
- *Delivery Mechanism*: Docker Containers
- *Code Setup and Walkthroughs*: Jupyter Notebooks
- *Search Engine*: Apache Solr

In this section, we'll introduce these technologies, as well the following other search engines besides Apache Solr which may also be good choices for applying the techniques we'll cover in this book: Lucene, Elasticsearch, Open Search, Lucidworks Fusion, Milvus, and Vespa.

1.2.1 Programming language: Python

Python was chosen as the programming language for this book for a few reasons. First, given this book targets both software engineers and data scientists, we needed to select a language popular and widely used among both professions. With Python being by far the most used programming language among data scientists and being a top-3 most used programming language among software engineerings (after C and Java), Python was the clear front-runner here.

Additionally, many of the most prominent data science frameworks applicable to the problems we'll tackle in this book are written in python or contain python bindings, making it a pragmatic choice.

Finally, Python is one of the most readible and approachable programming languages to those with no prior experience, making a great choice for teaching concepts even to those with no prior experience.

1.2.2 Data Processing Framework: Spark (PySpark)

Apache Spark is one of today's go-to technologies for large-scale data processing. It can be run easily on a single server, or scaled to large clusters for production distributed data processing jobs. Because search engines often deal with massive amounts of data, and AI-powered Search techniques in particular usually work better the most data you can provide, we wanted to show implementations which could be easily scaled out as needed, which building on Apache Spark provides.

Additionally, because we expect readers to be using many different search engines and other technologies, by demonstrating data processing using a technology not ties to any particular search engine, this enables the maximum amount of re-use of the techniques you'll learn in this book. Spark uses the generic idea of data frames, which can pull from any arbitrary data source, process data, and then save back to any data source. This means that even HDFS can be swapped out, as needed, with faster data platforms. Apache Solr and Elasticsearch, for example, can both be used as data sources and data targets of Spark jobs, enabling Spark to be run directly on top of your search platform to process and enhance your search data (content augmentation, user behavior analysis, machine learning model generation, and so on) and to take advantage of the speed of leveraging your search index when appropriate.

1.2.3 Delivery Mechanism: Docker Containers

The *AI-powered Search* contains hundreds of libraries and dependencies, and setting them all up seamlessly on a new computer or across different operating systems can be a very time-consuming and error-prone task. As such, we've chosen to ship all of the code for the book as Docker files and Docker containers. Docker is a platform for running virtual machine images with all operating system configuration and dependencies needed, enabling you to be up and running with no special configuration required and no opportunitiy for version or operating system conflicts.

1.2.4 Code Setup and Walkthroughs: Jupyter Notebooks

We want it to be as easy as possible for you to run and experiment with the code examples in this book. Jupyter notebooks enable all of the code to be shipped as walkthrough tutorials where you can view the code examples, make changes to the code, and just click "run" in your web browser to see the code running and in action.

The notebooks are all designed such that you can run them as many times as you like and get the same results, enabling you to focus on reading and understanding the each of the chapters, and following along and running all of the corresponding code without having to worry about configuration management or running system-level commands to get the code to run.

1.2.5 Choosing the right search engine technology

Though this book is a practical guide to delivering AI-powered search regardless of the underlying search engine, it is unfortunately not practical to create duplicate code examples targeting every available search engine technology. As such, we have chosen to base the book's code examples on the open source Apache Solr search engine.

Most of the data processing and machine learning code implemented in Python or Apache Spark can target virtually any search engine with minor configuration changes, and most direct search engine interactions (for queries, for example) can be implemented through using slightly different engine-specific configuration and query syntax.

With regard to search-specific features, the vast majority of capabilities needed to implement the techniques in this book exist across most major referenced search engines. That said, a handful of features needed for this book are only available in Solr - such as Solr's Text Tagger for entity extraction, Solr's Semantic Knowledge Graph capabilities, Solr's query-time graph traversals, and some of Solr's ranking and query parser capabilities, so users of other engines may have a bit of extra work when trying to implement those capabilities.

With Elasticsearch and Solr being the number one and number two top ranked full text search engines respectively, with Elasticsearch no longer being opensourced as of 2021, and with Elasticsearch missing a few key features needed to implement the techniques in this book, the choice to go with Solr was a clear one. Solr is based on Apache Lucene, which is the same underlying search engine library that powers Elasticsearch (not open source) and OpenSearch (an up and coming fork or Elasticsearch that is open source). Since we expect the large majority of readers to be using Lucene-based search engines (Elasticsearch, Solr, OpenSearch, or Lucene directly) due to their market dominance, we've chosen the most popular open sourced one and expect for the concepts to map well to practitioners using any of these engines.

Whether you choose one of the aforementioned search engines for your AI-powered search application, already have your own engine and are just looking to enhance it, or are potentially looking to try something new and cutting edge, it is good to know your options. The upcoming sections will provide a high-level summary of each of these engines, plus some others you may want to also consider depending on your use case.

APACHE LUCENE: THE CORE SEARCH LIBRARY POWERING APACHE SOLR, ELASTICSEARCH, AND OPENSEARCH

Apache Lucene is an open-source Java framework for building rich search applications. Lucene does not provide a search "engine", but instead provides extensive java libraries that you can integrate into your own software to build inverted indices, perform text analysis and tokenization, convert data into efficient data types for search, and manage both the low-level operations and the functions necessary to perform most search operations (query parsing, faceting, index updates and segment merges, indexing and querying, and so on).

Lucene is internally used in more search applications than any other search technology today, but is typically not used directly. That is to say, whereas Lucene is a search library, other software projects have evolved on top of Lucene to expose out fully-featured search servers that can be run as scalable distributed systems across large computing clusters (Lucene, in contrast, runs locally on each node only). These search servers expose APIs (REST and other protocols), as well as client libraries that are ready to deploy and use out of the box. Though some people still build applications directly on top of Lucene, most organizations have switched to using one of the popular production-ready search servers built on top of Lucene, such as Apache Solr, Elasticsearch, or OpenSearch.

APACHE SOLR: THE OPEN SOURCED, COMMUNITY-DRIVEN, RELEVANCE-FOCUSED SEARCH ENGINE

Solr is open source and was actually part of the Apache Lucene project until 2021, when became it's own top-level project at the Apache Software Foundation.

While we won't go deep into all the capabilities of Solr in this book (*Solr in Action* took 638 pages, and that was written back in 2014!), suffice it to say that Solr powers search for many of the most advanced search applications in the world, and it is very feature-rich and well-suited for building an AI-powered search application.

It is second in popularity only to Elasticsearch, but Solr has historically been leveraged moreso on information retrieval and ranking use cases over logs and data analytics (Elasticsearch's core focus), providing Solr with some unique capabilities on the relevance front which we'll make use of throughout our journey.

Regarding data processing, Solr also maintains a streaming expression framework that rivals Sparks distributed processing capabilities in many ways. While we could leverage this framework in our examples to reduce dependencies, we'll instead choose to leverage Spark for most of our offline data crunching and machine learning tasks in order to keep our examples more generally-applicable across different search platforms. This will make all of the examples easily reusable across all other engines.

Ultimately, our goal in this book is to teach you reusable techniques that you can apply regardless of your underlying search technology, so leveraging open source Solr for search-specific requirements and engine-agnostic technologies where possible will help us best achieve that goal of being engine-agnostic.

ELASTICSEARCH: THE MOST USED, ANAYTICS-FOCUSED, FULL TEXT SEARCH ENGINE

Elasticsearch is the most downloaded and most well-known search engine technology today by developers, apart from Apache Lucene, which internally powers Solr, Elasticsearch, and OpenSearch.

Due to Elasticsearch's popularity and mindshare and relative ease of use, it is often the go-to search application for new developers getting started with search.

Elasticsearch has overtaken Solr over time in terms of number of deployments and overall developer mind share, largely due to the popularity of the ELK stack (Elasticsearch + Logstash + Kibana), which has become the industry-standard for ingesting, searching, and analyzing log data. While log and event search is the main use case for Elasticsearch, it is also a very capable text search engine, providing a high level of feature parity with Solr (plus some additional features unavailable in Solr) for traditional search use cases.

Unfortunately Elasticsearch is no longer open sourced (as of 2021), making it now a much riskier choice for many organizations. That said, other than licensing issues, Elasticsearch is a perfectly great technology to use to apply the techniques in this book, and we expect Elasticsearch users to be able to easily follow along and apply most of the techniques in this book.

OPENSEARCH: THE OPEN SOURCE FORK OF ELASTICSEARCH

The OpenSearch project is an open source fork of Elasticsearch launched by Amazon in April 2021, after Elastic announced they would abandon their open source licensing for Elasticsearch in January 2021. It is yet to be seen whether OpenSearch will take over significant marketshare as an open source alternative to Elasticsearch and Solr, but OpenSearch provides most of the same capabilities as Elasticsearch, without the licensing issues. The Elasticsearch and OpenSearch projects will almost certainly diverge more and more over time, so the trajectory of OpenSearch under Amazon's management is yet to be determined.

LUCIDWORKS FUSION: THE OUT-OF-THE-BOX AI-POWERED SEARCH ENGINE

Sometimes it is more expedient to buy the capabilities you needs versus build them all in house. Whereas Apache Solr, Elasticsearch, and OpenSearch provide solid core search engine capabilities, neither of them provide many of the AI-powered search capabilities and pipelines found in this book.

Commercial vendors, like Lucidworks, can deliver end-to-end AI-powered search capabilities out of the box, whereas Solr and Elasticsearch are core matching engines but lack much of the extended infrastructure needed for AI-powered search. Lucidworks delivers Fusion using an "open core" model, where the central technologies Fusion relies upon (Apache Solr, Apache Spark) are completely open source, and customers thus have the full ability to see, debug, and modify their systems without being stuck with a black box system that restricts access to the core technologies, algorithms, and code.

NOTE **Disclaimer: Many of the techniques in this book are integrated out of the box in Fusion because the author previously worked at Lucidworks and helped integrate these techniques into Fusion. This book does not endorse any specific technology, so we recommend you do your own research before choosing a vendor.**

Lucidworks Fusion ships with most of the capabilities described in this book, giving you the opportunity to buy vs. build them all yourself. While this book is primarily focused on explaining how AI-powered search works under the covers and showing you how to implement it yourself, there's no doubt that leveraging a leading commercial vendor's product will get you better results faster. Regardless of which path you choose, this book will cover many of the techniques used by products like Fusion, and will help Fusion customers better understand, tune, and improve their Fusion-powered search applications.

MILVUS

Massive improvements in natural language models has given rise to new mechanisms for search and ranking based upon mapping queries and documents into dense vector spaces. This is sometimes referred to as "neural search" depending on the models being leveraged. We'll cover all of this in chapters 13-14, but several emerging search techniques rely entirely on these dense vector search approaches.

Milvus is an open source vector database, which is leading the way on this front. It enables efficient indexing of vectors, querying of vectors through a technique known as approximate nearest neighbors search, and scoring of vector similarities.

Some commercial vector databases are also beginning to emerge, such as the Pinecone vector similarity search engine. These engines open up new doors for high-performance natural language search and question answering systems, but they also lack most of the free-text search capabilities that power traditional search engines. As such, while these may be good for specialized use cases, they won't enable to full spectrum of AI-powered search techniques we'll cover in this book.

VESPA

The Vespa search engine is a traditional keyword search engine, but with first-class support tensors, which enable indexing of vectors, matricies, and other kinds of multi-dimensional data. The means that Vespa supports the best of both worlds, which rich text and attribute based search and ranking, as well as rich dense vector based search and ranking.

Vespa is the search engine technology acquired and developed by Yahoo to power Yahoo search, and later acquired by Verizon Media and open sourced. This means it has years of development, investment, and large-scale production hardening. Vespa was open sourced in September 2017, well after Solr and Elasticsearch had achieved the majority of their current marketshare. Vespa is not as widely known or used, but it contains a unique combination features that make it an interesting choice for someone implementing an AI-powered search engine. Solr, Elasticsearch, and OpenSearch are likely to catch up to Vespa over time on the dense vector search front, but as of now Vespa is positioned capability-wise in a very envious position possessing both competitive text search and ranking and also much more sophisticated dense vector search capabilities than any of the Lucene-based search engines.

JINA

Jina is an open source framework for building neural search systems. It provides a flexible and feature-rich framework for developing document and query processing pipelines. Jina AI's focus is specifically on integrating cutting edge AI models to enable next-generation search capabilities like semantic search, question answering, chatbots, multimodal search (images, audio, video, text combined), and so on. Jina is more of a python-based framework for building and running search workflows as opposed to a search engine in the traditional sense (so we won't be using it in this book), but it provides a very powerful toolkit of building blocks for assembling your own AI-powered search workflows, so it could be an interesting technology to consider for integrating many of the techniques in this book.

1.3 Target Audience for AI-Powered Search

This book assumes certain prerequisite knowledge of the mechanics of a search engine. If you do not have prior experience working with a technology like Solr or Elasticsearch, you can come up to speed quickly by reading through the books *Solr in Action* or *Elasticsearch in Action*.

1.3.1 Targeted Skillsets and Occupations

This book is primarily targeted at search engineers, software engineers, and data scientists. The book will also provide relevant conceptual understanding of AI-powered search for product managers and business leaders who do not possess these skills, but for a reader to get the most out of this book, they will need to be able to follow the Python code examples and will need a basic understanding how search and relevancy work leveraging a technology like Solr or Elasticsearch.

1.3.2 System Requirements for Running Code Examples

In order to comfortably run all the examples in this book, you will need a Mac, Linux, or Windows computer, and we recommend a minimum of 8GB of RAM to be able to run through some of the more heavy-duty Spark jobs. You will need to ensure you have Java 8+ installed on your computer, but we will walk you through configuration of all other dependencies on an as-needed basis. You will also need to install Docker and pull or build the Docker containers for the book (instruction in Appendix A). The Docker containers and datasets are quite large, so we recommend a minimum of 25GB of available disk space to pull and process all examples in the book.

1.4 When to consider AI-powered search

AI-powered search, as can be seen from Figure 1.9, provides a much more sophisticated alternative to manual relevance tuning. For organizations that benefit greatly from better search relevancy and either invest or are thinking about investing significantly in their search platform, deploying AI-powered search should lead to significantly higher quality with significantly less effort in the long run.

That being said, installing Elasticsearch/OpenSearch, Solr, or Vespa and ingesting documents is relatively easy, and if you only need basic keyword matching or are unable to invest the resources to build out and maintain an AI-powered search platform, then you might proceed cautiously with implementing too many of the techniques in this book, as it will definitely add more components and complexity to your search engine overall.

For organizations desparately needing to optimize their search relevance investment, however, the techniques in this book are an excellent guide in that endeavor.

1.5 How does AI-powered search work?

We've laid out our end goal of matching user intent through content understanding, user understanding, and domain understanding. We've also laid out the key technology platforms we'll leverage and prerequisite knowledge you'll need to do pull this off. With that background established, let's now wrap up this chapter with an overview of what components are needed to actually deliver an AI-powered search platform. Search intelligence is not black and white with search either being "basic" or "AI-powered". Instead, search intelligence typically matures along a predictable progression, as shown in Figure 1.9.

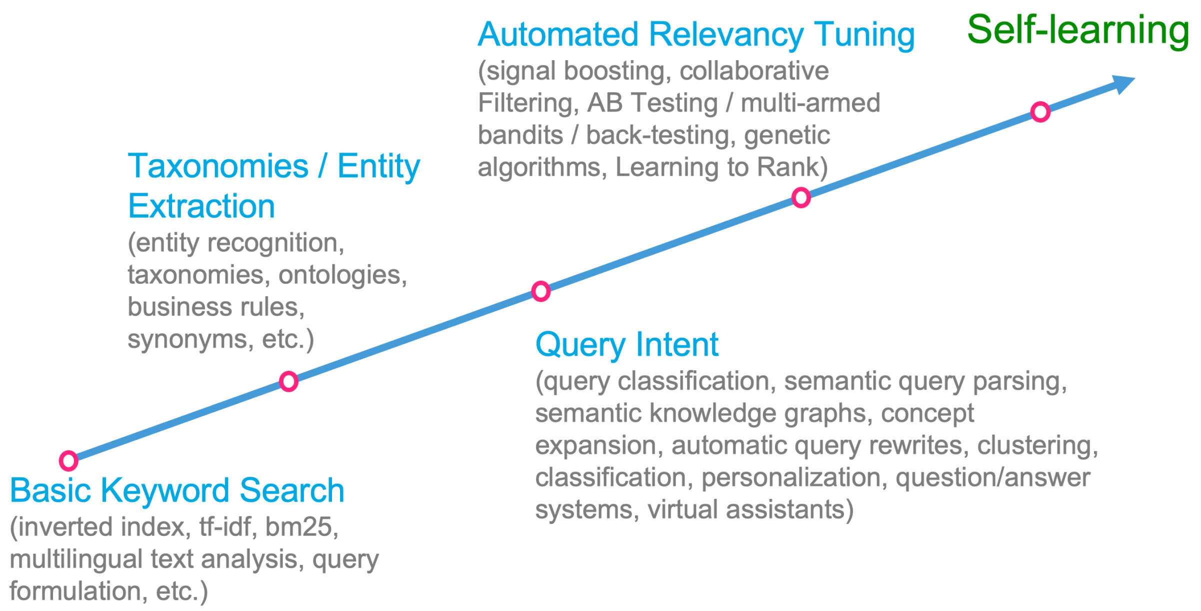

Basic Keyword Search
(inverted index, tf-idf, bm25, multilingual text analysis, query formulation, etc.)
Taxonomies / Entity Extraction
(entity recognition, taxonomies, ontologies, business rules, synonyms, etc.)
Query Intent
(query classification, semantic query parsing, semantic knowledge graphs, concept expansion, automatic query rewrites, clustering, classification, personalization, question/answer systems, virtual assistants)
Automated Relevancy Tuning
(signal boosting, collaborative Filtering, AB Testing / multi-armed bandits / back-testing, genetic algorithms, Learning to Rank)
Self-learning

Figure 1.9 The Typical Search Intelligence Progression, from basic keyword search to a full self-learning search platform.

In this progression, organizations almost always start with basic keyword search in order to get up and running. Once in production, they realize their search relevancy needs to be improved, at which point they start manually tuning field weights, boosts, text and language analysis, and introducing additional features and functions to improve the aggregate relevancy of search results.

Eventually, they realize they need to inject domain understanding into their search capabilities, at which point organizations begin to invest in synonyms lists, taxonomies, lists of known entities, and domain-specific business rules. While these all help, they eventually also discover that relevant search is very much dependent upon successfully interpreting user queries and understanding user intent, so they begin investing in techniques for query classification, semantic query parsing, knowledge graphs, personalization, and other attempts to correctly interpret user queries.

Because these tasks successfully yield improvement, this success often results in the creation of large teams investing significant time manually tuning lists and parameters, and eventually organizations may realize that it is possible (and more expedient) to automate as much of that process as possible through learning from user signals, user testing (A/B, multi-armed-bandit, and offline relevancy simulations), and building of machine-learned relevancy models. The end goal is to completely automate each of these steps along the search intelligence progression and enable the engine to be self-learning. It takes a while for most to get there, however, so it's important to start with a solid foundation.

1.5.1 The Core Search Foundation

The first step in building a search platform is almost always to get traditional keyword search working (the "content understanding" part back in Figure 1.8). Many people and teams have spent years or even a decade or more tuning and improving this step, and a whole discipline called *Relevance Engineering* has arisen that has historically focused significant efforts in understanding content, improving content for search, adjusting boosts, query parameters, and query functions, and otherwise trying to maximize the relevance of the traditional search experience. As relevance engineers become more sophisticated, their work often bleeds over into the realms of user understanding and recommendations, as well as into the domain-understanding and semantic search realm.

Our focus in *AI Powered Search* will be on automating the process of learning and optimizing search relevance to be continual and automatic, but a deep understanding of how to think like a relevance engineer and tune a search engine yourself would be quite helpful to you as background on that journey. For that background into the world of Relevance Engineering and tuning traditional keyword search relevance, I highly recommend the book *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016).

So what characteristics actually differentiate a well-tuned search engine and an AI-powered search engine? A well-tuned search engine clearly serves as the foundation upon which AI-powered search is built, but AI-powered search goes far beyond just being well-tuned and delivers on the ability to continuously learn and improve through Reflected Intelligence.

Reflected Intelligence is the idea of leveraging continual feedback loops of user inputs, content updates, and user interactions with content in order to continually learn and improve the quality of your search application.

1.5.2 Reflected Intelligence through Feedback Loops

Feedback loops are critical to building an AI-powered search solution. Imagine if your entire middle school, secondary school, and possible post-secondary school education had consisted of nothing more that you reading text books: no teachers to ask questions, no exams to test your knowledge and provide feedback, and no classmates or others with which to interact, study, or collaborate. You would have probably hit endless walls where you were unable to fully grasp certain concepts or even understand what was being read, and you would probably have understood many ideas incorrectly and never even had the opportunity to realize or adjust your assumptions.

Search engines often operate this same way. Smart engineers push data to the search engine and tune certain features and feature weights, but then the engine just reads those configurations and that content and statically acts upon it the same way every time for repeated user queries. Search engines are the perfect kind of system for interactive learning, however, through the use of feedback loops. Figure 1.10 shows the idea of search engine feedback loops.

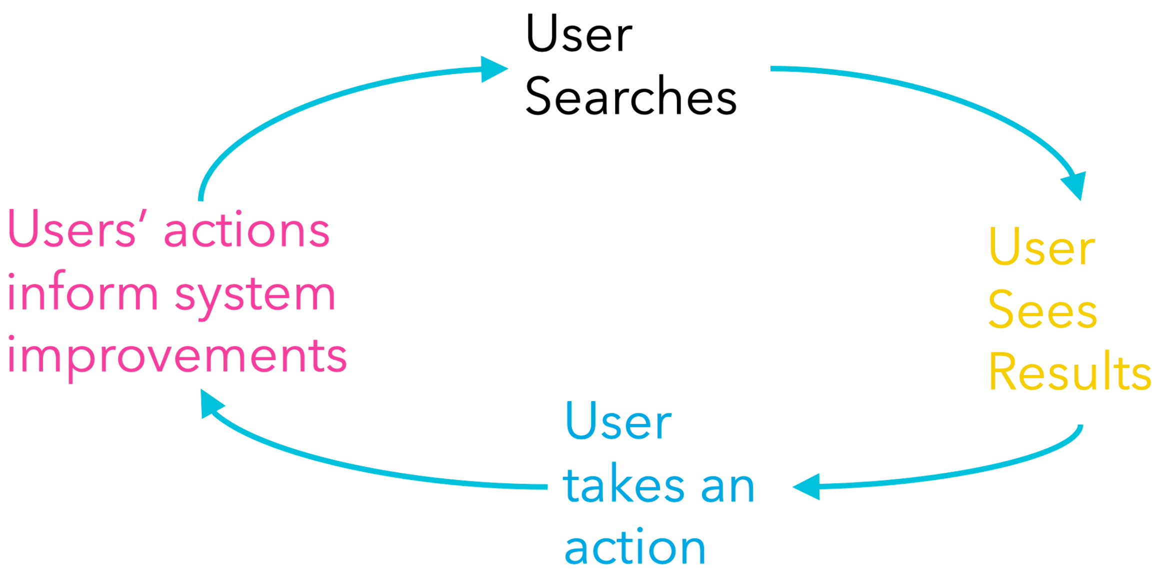


Figure 1.10 Reflected Intelligence through Feedback Loops

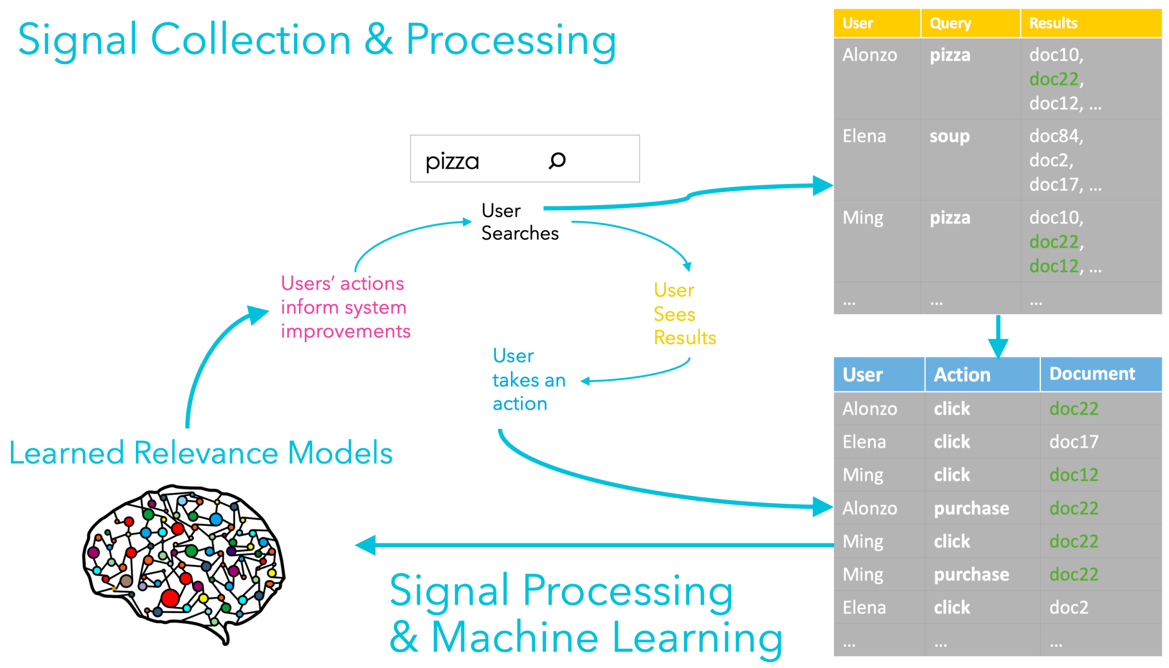
In Figure 1.10, you see the typical flow of information through a search feedback loop. First, a user issues a query. This query executes a search, which returns results to an end user, such as a specific answer or list of answers or a list of links to pages. Once presented with the list, the user then takes one or more actions. These actions usually start with clicks on documents, but those clicks can ultimately lead to adding an item to a shopping cart and purchasing it (ecommerce), giving the item a thumbs up or thumbs down (media consumption website), liking or commenting on the result (social media website), or any number of other context-specific actions.

Each of those actions taken by a user, however, is likely to provide clues as to the perceived relevance of the search results to that user. A thumbs up or thumbs down provides explicit positive and negative relevance judgements, as do add to cart, purchases, likes, and bookmarks. Clicks provide less clear signals, but usually indicate some perceived suitability as a search result.

These actions can then be leveraged to generate an improved relevance ranking model for future searches. For example, if the majority of users for a given query always click on result number three more often than on result one or two, this is a strong indicator that result number three is perceived as a better, more relevant result. Leveraging feedback loops, your search application can then automatically adjust the results ordering for that query in future searches, delivering an improved search experience for the next user's search. This feedback loop then continues again, ad infinitum, as the engine is constantly self-tuning.

BEHAVIORAL INTELLIGENCE FROM USER SIGNALS

The searches, clicks, likes, add to carts, purchases, comments, and any number of other interactions your users may have with your search application, are incredibly valuable data that your search application needs to capture. We collectively refer to these data points as *signals*. Signals provide a constant stream of feedback to your search application recording every meaningful interaction with your end users. These digital moments - whether a search, a list of returned search results, a click, a purchase, or anything in-between, can then be leveraged by machine learning algorithms to generate all kinds of models to power user understanding, content understanding, and domain understanding. Figure 1.11 shows the data flow for the collection and processing of signals in a typical AI-powered search application.


Signal Collection & Processing
pizza
User Searches
User Sees Results
User takes an action
Users' actions inform system improvements
Learned Relevance Models
Signal Processing & Machine Learning

User	Query	Results
Alonzo	pizza	doc10, doc22, doc12, …
Elena	soup	doc84, doc2, doc17, …
Ming	pizza	doc10, doc22, doc12, …
…	…	…

User	Action	Document
Alonzo	click	doc22
Elena	click	doc17
Ming	click	doc12
Alonzo	purchase	doc22
Ming	click	doc22
Ming	purchase	doc22
Elena	click	doc2
…	…	…

Figure 1.11 Signal collection & processing data flow

In Figure 1.11, you'll see signals being collected for each search, as well as resulting clicks and purchases, though unique signals can also be recorded for any other kind of user interaction (add to cart, facet click, bookmark, hover, or even page dwell time).

Signals are one of the two kinds of fuel that power the intelligence engine of your AI-powered search application, with the other type being your content.

CONTENT INTELLIGENCE FROM DOCUMENTS

While signals provide the constant stream of usage and feedback data to your search application, your content is also a rich source of information that can be leveraged in your feedback loops. For example, if someone searches for a particular keyword, the other keywords and top categories in the documents returned serve as a valuable datapoint that can be leveraged to tag or categorize the query. That data might be shown to end users (as facets, for example), and then users might interact with that data, which subsequently generates a signal from which the engine can then learn.

Further, the content of your documents contains a representative textual model of your domain. Entities, domain-specific terminology, and the sentences contained within your documents serve as a rich, semantic graph representing the domain of your data. That graph can be leveraged to drive powerful conceptual and semantic search that better understands your domain. We'll dive more deeply into understanding your content in chapter 2, and into these semantic search capabilities leveraging this rich semantic knowledge graph in chapter 5.

1.5.3 Curated vs. Black-box AI

Many modern AI techniques rely heavily on deep learning based on artificial neural networks. Unfortunately, it is very challenging in most cases for a human to understand specific factors that go into any particular prediction or output from the deep learning model due to the internal complexity of the learned model.

This results in a "black box AI" system, where the results may be correct or impressive, but they are not easy for someone to debug—or more importantly, correct—when the model makes an incorrect judgement. An entire field of *Explainable AI* (sometimes called *Interpretable AI* or *Transparent AI*) has arisen out of a need to be able to understand, curate, and trust these models.

In this book, while we will cover several deep learning approaches to search (see chapters 13 and 14 on dense vector search and question answering), we will largely focus our efforts on creating intelligence which can be expressed in human terms and then corrected and augmented by human intelligence. You can think of this as "AI-assisted human curation", or alternatively as "human-assisted AI", but either way the overriding philosophy of this book is to use AI to automate the process of search intelligence, while keeping the human in the loop and able to take control and inject their own subject matter expertise as needed.

1.5.4 Architecture for an AI-powered search engine

The architecture for an AI-powered search engine requires numerous building blocks to be assembled together to form a smart end-to-end system. You obviously need a core search engine like Solr, Elasticsearch/OpenSearch, or Vespa. You need to feed your searchable content into the engine, transforming it to make it more useful. These transformation might include changes like:

- Classifying the document, adding the classification as a field
- Normalizing field values
- Entity Extraction from text, adding entities in separate fields
- Sentiment Analysis
- Clustering content, adding clusters as a field
- Phrase detection and annotation
- Pulling in additional data from a knowledge graph, external API, or other data source
- Part of Speech Detection and other Natural Language Processing steps
- Fact extraction (such as RDF triples)
- Spam or NSFW (Not Safe For Work) content detection
- Application of other Machine Learning models or ETL rules to enrich the document

Once the data is in the engine, your goal is to make it available for searching. This requires query pipelines, which can parse incoming queries, idenfity phrases and entities, expand to related terms/synonyms and concepts, correct misspellings, and then rewrite the query so your core engine can find the most relevant results.

Much of this query intelligence requires a robust understanding of your domain, however. This requires running batch jobs on your content and user signals in order to learn patterns and derive domain-specific intelligence. What are the most common misspellings from your users, and what do they choose as the correct spelling among multiple candidates (covered in chapter 6)? When a user searches for specific queries, which documents should be boosted as the most popular (covered in chapter 8)? For unknown queries, what is the ideal ranking among all the attributes/features available for matching (covered in chapter 10)?

We need access to most of these answers in a pre-computed (or quickly computable) fashion at query time, because we expect queries to be real-time and often to return within milliseconds. This requires a job processing framework (we'll use Spark in this book) and a workflow scheduling mechanism to keep the jobs running in sequence.

You'll also have a constant stream of new data incoming in the form of user signals, so you'll need a mechanism for capturing and storing those (capturing on the front-end application, storing in your core engine or other back-end datastore).

The signals will then be used to generate all kinds of models—from signal boosting of specific items for popular queries, to generation of generalized learning to rank models which apply to all queries, to generation of user-specific recommendations and personalization preferences for each user or segment of users.

Ultimately, you'll end up with a system that receives constant streams of document changes and user signals, is constantly processing those updates to learn and improve the model, and is then constantly adjusting future search results and measuring the impact of changes to continually deliver more intelligent results. That is the key behind AI-powered search: the processes of continual learning and improvement based upon real user interactions and evolving content patterns, in order to fully understand user intent and deliver an ever improving search experience.

1.6 Summary

- Expectations for search sophistication are evolving, with end users expecting search to now be domain-aware, contextual and personalized, conversational, multi-modal (through text, voice, images, or even push-based events), intelligent, and assistive.
- Search and recommendations are the two extreme ends of a continuous personalization spectrum within information retrieval.
- Correctly interpreting user intent requires simultaneous understanding of your content, your user and their preferences, and the knowledge domain in which your platform operates.
- Optimal search relevancy lies at the intersection of Personalized Search (traditional keyword search + collaborative recommendations), Semantic Search (traditional keyword search + knowledge graphs), and Multi-modal Recommendations (collaborative recommendations + knowledge graphs).
- The techniques in this book should apply to most search platforms, but we'll primarily leverage Apache Solr and Apache Spark for most of our examples.
- AI-powered search operates on two kinds of fuel: content and user signals
- Reflected Intelligence - the use of feedback loops to continually collect signals, tune results, and measure improvements, is the engine the enables AI-powered search to learn and constantly improve.

2 Working with natural language

This chapter covers

- **Uncovering the hidden structure in unstructured data**
- **A search-centric philosophy of language and natural language understanding**
- **Exploring distributional semantics and word embeddings**
- **Modeling domain-specific knowledge**
- **Tackling challenges in natural language understanding and query interpretation**
- **Applying natural language learning techniques to both content and signals**

In the first chapter, we provided a high-level overview of what it means to build an AI-powered search system. Throughout the rest of the book, we'll explore and demonstrate the numerous ways your search application can continuously learn from your content and your user behavioral signals in order to better understand your content, your users, and your domain, and to ultimately deliver users the answers they need. We will get much more hands on in chapter three, firing up a search server (Apache Solr), a data processing layer (Apache Spark), and starting with the first of our Jupyter notebooks, which we'll use throughout the book to walk through many step-by-step examples.

Before we dive into those hands-on examples and specific implementations (the "what"), however, it is important in this chapter that we first establish a shared mental model for the higher level problems we're trying to solve. Specifically, when it comes to intelligent search, we have to deal with many complexities and nuances in natural language - both in the content we're searching and in our users' search queries. We have to deal with keywords, entities, concepts, misspellings, synonyms, acronyms, ambiguous terms, explicit and implied relationships between concepts, hierarchical relationships usually found in taxonomies, higher-level relationships usually found in ontologies, and specific instances of entity relationships usually found in higher-level knowledge graphs.

While it might be tempting to dive immediately into some specific problems like how to automatically learn misspellings from content or how to discover synonyms from mining user search sessions, it’s going to be more prudent to first establish a conceptual foundation that explains what *kinds* of problems we have to deal with in search and natural language understanding. Establishing that philosophical foundation will enable us to build better end-to-end solutions in our AI-powered search system, where all the parts work together in a cohesive and integrated way. This chapter will thus provide the philosophical underpinnings for how we tackle the problems of natural language understanding throughout this book and apply those solutions to make our search applications more intelligent. We’ll begin by discussing some common misconceptions about the nature of free text and other unstructured data sources.

2.1 The myth of unstructured data

The term "unstructured data" has been used for years to describe textual data, because it does not appear to be formatted in a way that can be readily interpreted and queried. The widely held idea that text, or any other data that doesn’t fit a pre-defined schema ("structure"), is actually "unstructured", however, is a myth that we’ll spend time reconsidering throughout this section.

If you look up *unstructured data* in Wikipedia, it is defined as "information that either does not have a pre-defined data model or is not organized in a pre-defined manner". The entry goes on to say that "unstructured information is typically text-heavy, but may contain data such as dates, numbers, and facts, as well".

The phrase "unstructured data" really is a poor term to describe textual content, however. In reality, the terms and phrases present in text encode an enormous amount of meaning, and the linguistic rules applied to the text to give it meaning serve as their own structure. Calling text unstructured is a bit like calling a song playing on the radio "arbitrary audio waves". Even though every song has unique characteristics, most typically exhibit common attributes (tempo, melodies, harmonies, lyrics, and so on). Though these attributes may differ or be absent from song to song, they nevertheless fit common expectations that then enable meaning to be conveyed by and extracted from each song. Textual information typically follows similar rules - sentence structure, grammar, punctuation, interaction between parts of speech, and so on. Figure 2.1 shows an example of text which we’ll explore a bit more in the upcoming sections as we investigate this structure further.

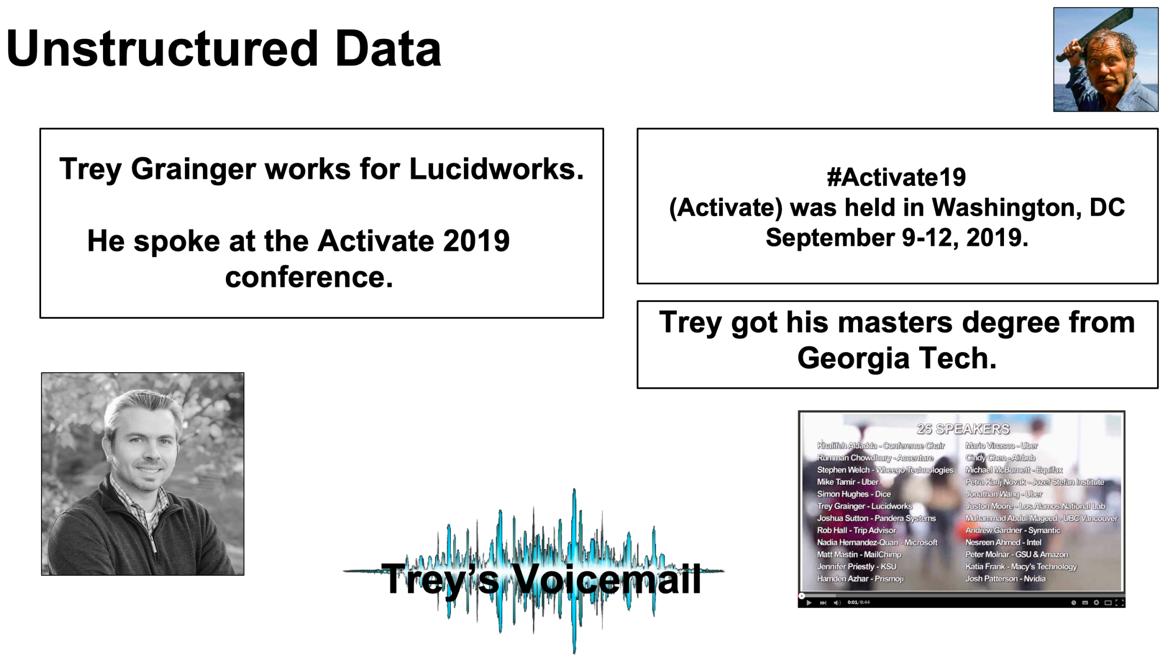
Trey Grainger works at Searchkernel.
He spoke at the Activate 2019 conference.
#Activate19 (Activate) was held in Washington, DC September 9-12, 2019. Trey got his masters from Georgia Tech.

Figure 2.1 Unstructured Data. This text represents typical unstructured data you may find in a search engine.

While text is the most commonly recognized kind of unstructured data, there are also several other kinds of unstructured data that share similar characteristics with textual data, as we'll see in the next section.

2.1.1 Types of unstructured data

Free text content is considered the primary type of unstructured data, but search engines are also commonly used to index many other kinds of data that similarly don't fit neatly into a structured database. Common examples include images, audio, video, and event logs. Figure 2.2 expands on our text example from Figure 2.1 and includes several other types of unstructured data, such as audio, images, and video.



Trey Grainger works for Lucidworks.

He spoke at the Activate 2019 conference.

#Activate19
(Activate) was held in Washington, DC
September 9-12, 2019.

Trey got his masters degree from Georgia Tech.

25 SPEAKERS

Khalifeh AlJadda - Conference Chair	Mario Vinasco - Uber
Rumman Chowdhury - Accenture	Cindy Chen - Airbnb
Stephen Welch - Wheego Technologies	Michael McBurnett - Equifax
Mike Tamir - Uber	Petra Karlj Novak - Jozef Stefan Institute
Simon Hughes - Dice	Jonathan Wang - Uber
Trey Grainger - Lucidworks	Juston Moore - Los Alamos National Lab
Joshua Sutton - Pandera Systems	Muhammad Abdul-Mageed - UBC Vancouver
Rob Hall - Trip Advisor	Andrew Gardner - Symantic
Nadia Hernandez-Quan - Microsoft	Nesreen Ahmed - Intel
Matt Mastin - MailChimp	Peter Molnar - GSU & Amazon
Jennifer Priestly - KSU	Katia Frank - Macy's Technology
Hamden Azhar - Prismoji	Josh Patterson - Nvidia

0:01/8:44

Figure 2.2 Multiple types of unstructured data. In addition to the text from the last example, we now see images, audio, and video, which Are other forms of unstructured data.

Audio is the most similar to text content, since it is often just another way to encode words and sentences. Of course, audio can include much more than just spoken words—it can include music and non-language sounds, and it can more effectively encode nuances such as emotion, tone of voice, and simultaneously overlapping communication.

Images are another kind of unstructured data. Just as words form sentences and paragraphs to express ideas, images form grids of colors that taken together form pictures.

Video, then, serves as yet another kind of unstructured data, as it is a combination of multiple images over time, as well as optional audio that coincides with the image progression.

Often times unstructured data may be found mixed with structured data, which we typically refer to as "semi-structured" data. Log data is a great example of such semi-structured data. Often logs end up being semi-structured, for example, having an event date, event type (such as warning vs. error or search vs. click), and some kind of log message or description in free text.

Technically speaking virtually any kind of file could be considered unstructured data, but we'll primarily deal with the aforementioned types throughout this book.

Search engines are often tasked with handling each of these kinds of unstructured data, so we'll discuss strategies for handling them throughout the book.

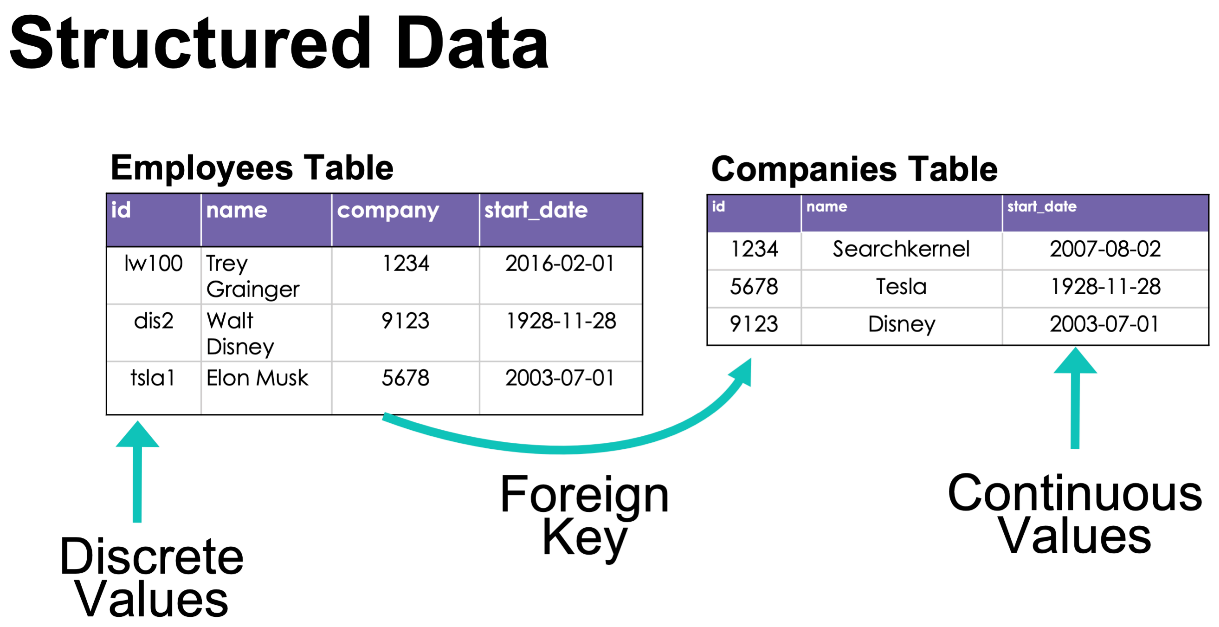
2.1.2 Data types in traditional structured databases

In order to better deal with our unstructured data, it may be useful to first contrast it with structured data in SQL database. This will allow us to later draw parallels between how we can query unstructured data representations versus structured ones.

A record (row) in a SQL database is segmented into columns, which can each be of a particular data type. Some of these data types represent discrete values - values that come from an enumerated list. IDs, names, and textual attributes are a few examples of discrete values. Other columns may hold continuous values, such as date/time ranges, numbers, and other column types which represent ranges without a finite number of possible values.

Generally speaking, when one wants to relate different rows together or to relate them to rows in other database tables, "joins" will be performed on the discrete values. Joins leverage a shared value (often an ID field) to link two or more records together in order to form a composite record that relates each of those records together.

For example, if someone had two tables of data, one representing "employees" and another representing "companies", then there would likely be an "ID" column on the "companies" table, and a corresponding "company" column on the employees table. The company field on the employees table is known as a *foreign key*, which is a value that is shared across the two tables and which is used to link the records together based upon a shared identifier. Figure 2.3 demonstrates this example, showing examples of discrete values, continuous values, and a join across tables using a foreign key.



id	name	company	start_date
lw100	Trey Grainger	1234	2016-02-01
dis2	Walt Disney	9123	1928-11-28
tsla1	Elon Musk	5678	2003-07-01

id	name	start_date
1234	Searchkernel	2007-08-02
5678	Tesla	1928-11-28
9123	Disney	2003-07-01

Figure 2.3 Structured data in a typical database. Discrete values represent identifiers and enumerated values, continuous values represent data that falls within ranges, and foreign keys exist when the same value exists across two tables and can thus be used as a shared attribute which creates a relationship between corresponding rows from each table.

This notion of joining different records together based upon known relationships (keys and foreign keys) is powerful way to work with relational data across explicitly modelled tables, but as we'll see in the next section, very similar techniques can also be applied even to free-form unstructured data.

2.1.3 Joins, fuzzy joins, and entity resolution in unstructured data

Whereas structured data in a database is already in an easily queryable form, the reality is that unstructured data suffers less from a lack of structure, and more just from having a large amount of information packed into a very flexible structure. In this section, we'll walk through a concrete example that uncovers this hidden structure in unstructured data and demonstrates the ways it can similarly be leveraged to find and join relationships between documents.

FOREIGN KEYS IN UNSTRUCTURED DATA

In the last section, we discussed how foreign keys can be used to join two rows together in a database based upon a shared identifier between the two records. In this section, we'll show how the same objective can actually also be achieved with text data.

For example, we can easily map the idea of "foreign keys" used in a SQL table into the same unstructured information we previously explored in Figure 2.2. Notice in Figure 2.4 that two different sections of text both contain the word "Activate", which refers to a technology conference.

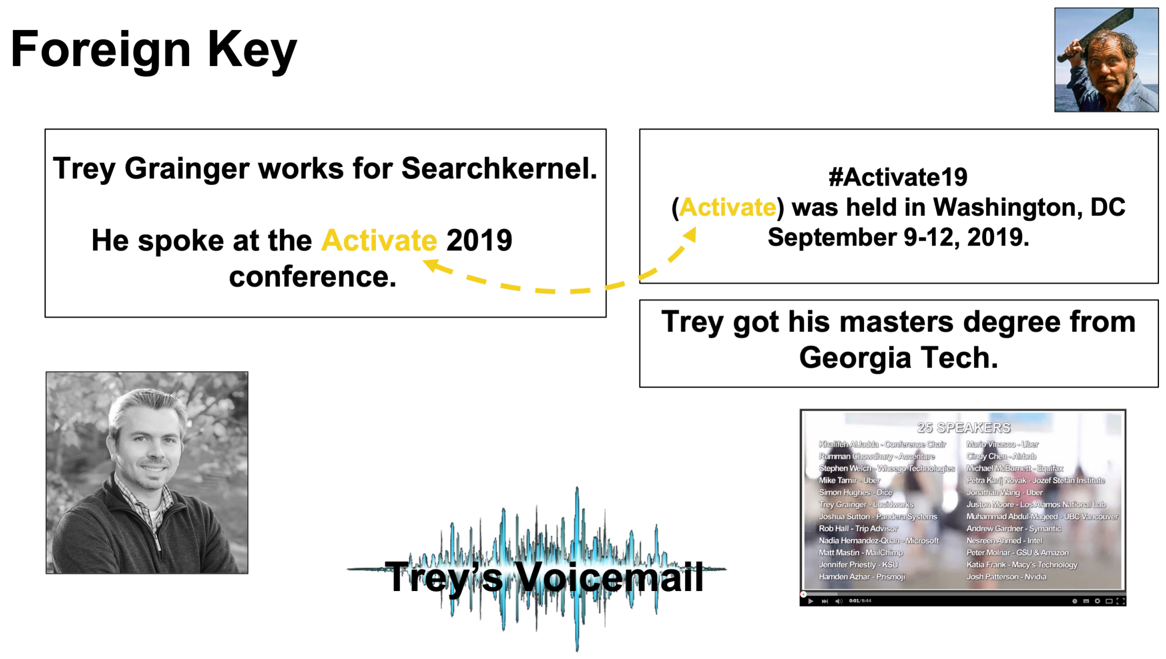
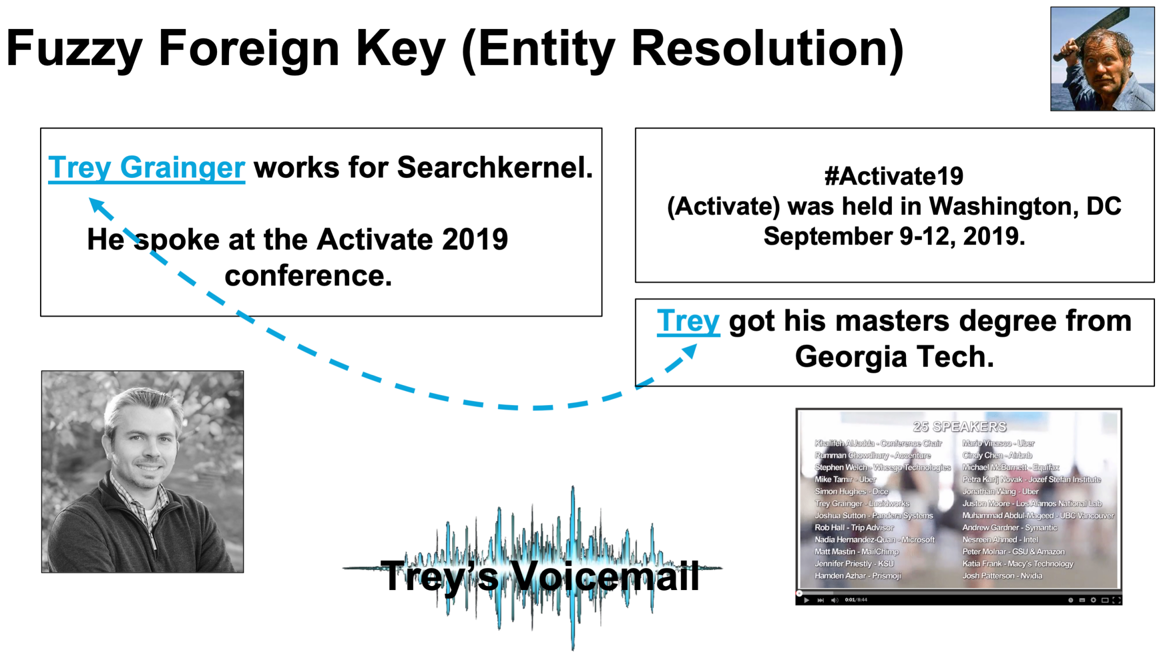

Foreign Key
Trey Grainger works for Searchkernel.
He spoke at the Activate 2019 conference.
#Activate19 (Activate) was held in Washington, DC September 9-12, 2019.
Trey got his masters degree from Georgia Tech.
Trey's Voicemail
25 SPEAKERS
Khalifeh AlJadda - Conference Chair
Rumman Chowdhury - Accenture
Stephen Welch - Wheego Technologies
Mike Tamir - Uber
Simon Hughes - Dice
Trey Grainger - Lucidworks
Joshua Sutton - Pandera Systems
Rob Hall - Trip Advisor
Nadia Hernandez-Quan - Microsoft
Matt Mastin - MailChimp
Jennifer Priestly - KSU
Hamden Azhar - Prismoji
Mario Vinasco - Uber
Cindy Chen - Airbnb
Michael McBurnett - Equifax
Petra Karlj Novak - Jozef Stefan Institute
Jonathan Wang - Uber
Juston Moore - Los Alamos National Lab
Muhammad Abdul-Mageed - UBC Vancouver
Andrew Gardner - Symantic
Nesreen Ahmed - Intel
Peter Molnar - GSU & Amazon
Katia Frank - Macy's Technology
Josh Patterson - Nvidia
0:01/8:44

Figure 2.4 Foreign keys in unstructured data. In this example, the same term is being used to join across two related text documents.

The first instance indicates a conference being spoken at, while the second block of text contains general information about the event. For purposes of our example, let's assume that every piece of information (block of text, image, video, and audio clip) is represented as a separate document in our search engine. As such, there is functionally very little difference between having two rows in a database table that each contain a column with the value of "Activate", and having separate documents in our search engine that each contain the value of "Activate". In both cases, we can think of these documents as related by a foreign key.

FUZZY FOREIGN KEYS IN UNSTRUCTURED DATA

With unstructured data, however, we have much more power than with traditional structured data modeling. In Figure 2.5, for example, notice that now two documents are linked that both refer to me - one using my full name of "Trey Grainger", and one simply using my first name of "Trey".



Fuzzy Foreign Key (Entity Resolution)

Trey Grainger works for Searchkernel.

He spoke at the Activate 2019 conference.

#Activate19
(Activate) was held in Washington, DC
September 9-12, 2019.

Trey got his masters degree from Georgia Tech.

25 SPEAKERS

Khalifeh AlJadda - Conference Chair
Rumman Chowdhury - Accenture
Stephen Welch - Wheego Technologies
Mike Tamir - Uber
Simon Hughes - Dice
Trey Grainger - Lucidworks
Joshua Sutton - Pandera Systems
Rob Hall - Trip Advisor
Nadia Hernandez-Quan - Microsoft
Matt Mastin - MailChimp
Jennifer Priestly - KSU
Hamden Azhar - Prismoji
Mario Vinasco - Uber
Cindy Chen - Airbnb
Michael McBurnett - Equifax
Petra Karlj Novak - Jozef Stefan Institute
Jonathan Wang - Uber
Juston Moore - Los Alamos National Lab
Muhammad Abdul-Mageed - UBC Vancouver
Andrew Gardner - Symantic
Nesreen Ahmed - Intel
Peter Molnar - GSU & Amazon
Katia Frank - Macy's Technology
Josh Patterson - Nvidia

Figure 2.5 Fuzzy foreign keys. In this example, the same entity is being referenced using different term sequences, and a join is occurring based upon both phrases resolving to the same entity.

This is an example of *entity resolution*, where there are two different representations of the entity, but they can still be resolved to the same meaning, and therefore can still be used to join information between two documents. you can think of this as a "fuzzy foreign key", since it's still a foreign key, but not in a strict token matching sense, as it requires additional Natural Language Processing and entity resolution techniques to resolve.

Once we've opened that door to advanced text processing for entity resolution, of course, now we can learn even more from our unstructured information.

For example, not only do the names `Trey` and `Trey Grainger` in these documents refer to the same entity, but so do the words `he` and `his`, as Figure 2.6 demonstrates.

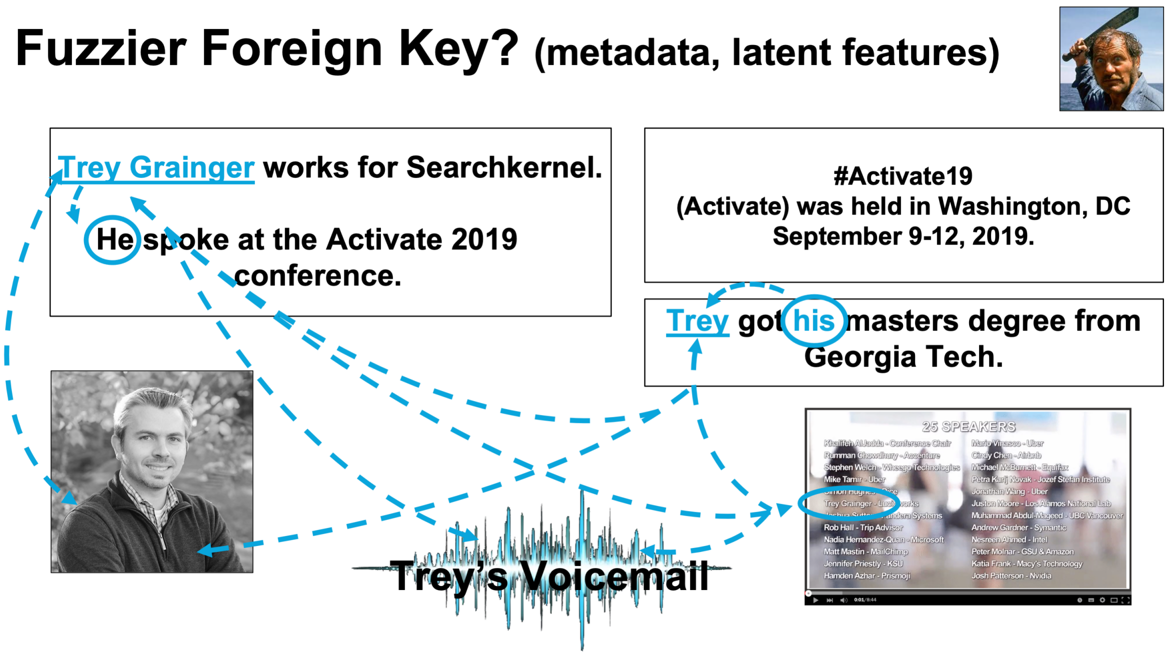

Fuzzier Foreign Key? (metadata, latent features)
Trey Grainger works for Searchkernel.
He spoke at the Activate 2019 conference.
#Activate19
(Activate) was held in Washington, DC
September 9-12, 2019.
Trey got his masters degree from Georgia Tech.
25 SPEAKERS
Trey's Voicemail

Figure 2.6 Fuzzier foreign keys. In this example, proper nouns, pronouns, images, and references within video are all resolved to the same entity, which can then be used to join across documents.

You'll also notice in Figure 2.6 that both an image of me (in the bottom-left corner, in case you have no idea what I look like) and a video containing a reference to my name are identified as related and joined back to the textual references. We're essentially relying on the hidden structure present in all of this unstructured information in order to understand the meaning, related the documents together, and learn even more about each of the referenced entities in those documents.

DEALING WITH AMBIGUOUS TERMS

So far, so good, but in real-world content it is not always appropriate to assume that the same term in multiple places carries the same meaning, or even that our entity resolution always resolves entities correctly. This problem of the same spelling of words and phrases having multiple potential meanings is called *polysemy*, and dealing with these ambiguous terms can be a huge problem in search applications.

You may have noticed an odd image in the upper-right-hand corner of the previous figures that seemed a bit out of place in our examples. This image is of a fairly terrifying man holding a machete. Apparently, if you go to Google and search for `Trey Grainger`, this image comes back. If you dig in further, you'll see in Figure 2.7 that there's a Twitter user also named `Trey Grainger`, and this image is his profile picture.

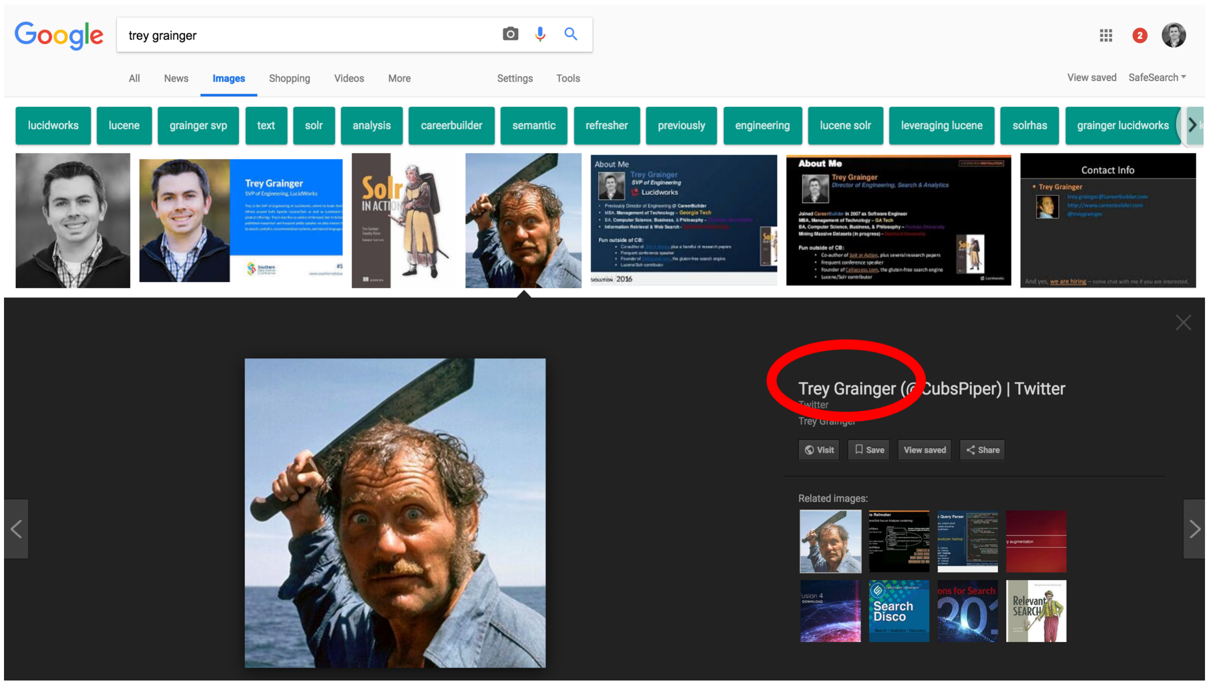


Figure 2.7 Polysemy. This image shows a Google search for the phrase "Trey Grainger". Pictures of multiple different people are returned because those people's names share the same spelling, making the term sequence "Trey Grainger" ambiguous.

The picture is apparently of Robert Shaw (who plays Quint in the 1975 movie Jaws), but it's definitely not the kind of thing you want people to first come across when they search for you online!

There are two key lessons to take away here. First, never Google yourself (you might be terrified at what you find!). Second, and on a more serious note, polysemy is a major problem in search and natural language understanding. It's not safe to assume a term has a single meaning, or even a consistent meaning in different contexts, and it's important that our AI-powered search engine is able to leverage context to differentiate these different meanings.

UNSTRUCTURED DATA AS A GIANT GRAPH OF RELATIONSHIPS

In the previous sections we've seen that unstructured data not only contains rich information (entities and their relationships), but also that it is possible to relate together different documents by joining them on shared entities, similarly to how foreign keys work in traditional databases. Typical unstructured data contains so many of these relationships, however, that instead of thinking in terms of rows and colums, it may be more useful to think of the collection of data as a giant graph of relationships, as we'll explore in this section.

At this point, it should be clear that there is much more structure hidden in unstructured data than most people appreciate. Unstructured information is really more like "hyper-structured" information - it is a graph that contains much more structure than typical "structured data".

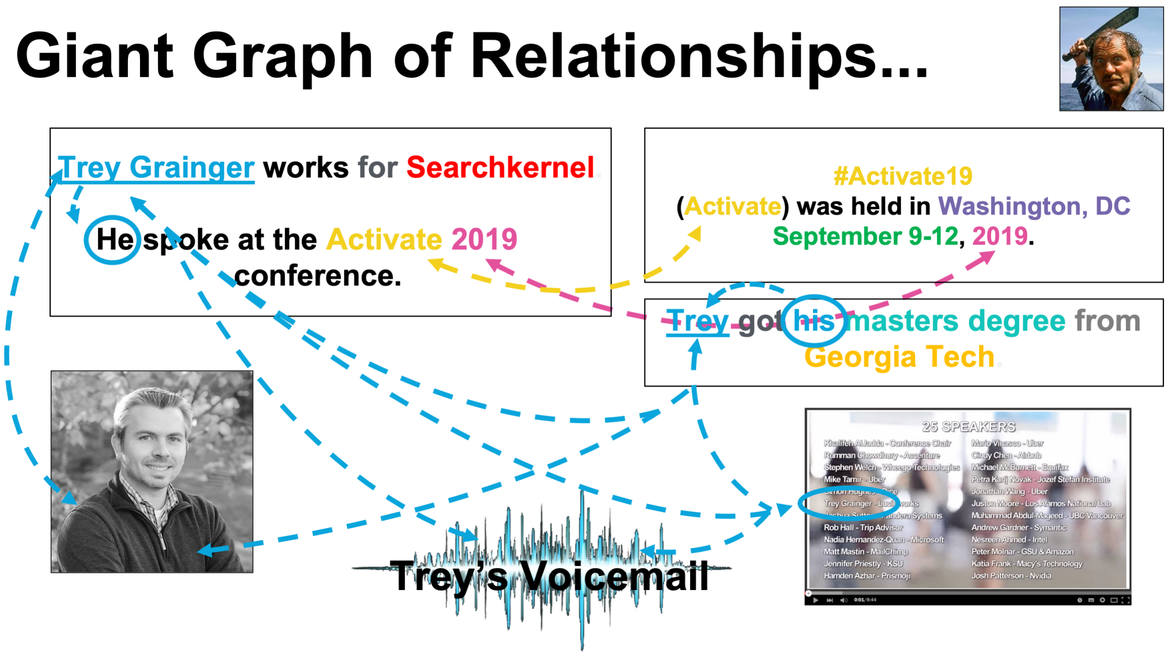

Giant Graph of Relationships...
Trey Grainger works for Searchkernel
He spoke at the Activate 2019 conference.
#Activate19 (Activate) was held in Washington, DC September 9-12, 2019.
Trey got his masters degree from Georgia Tech
25 SPEAKERS
Trey's Voicemail

Figure 2.8 Giant graph of relationships. A rich graph of relationships emerges from even just a few related documents.

Figure 2.8 demonstrates this giant graph of relationships that is present in even the small handful of documents from our example. You can see names, dates, events, locations, people, companies, and other entities, and you can infer relationships between them leveraging joins between the entities across documents. You'll also notice that the images have been correctly disambiguated so that the machete guy is now disconnected from the graph.

If all of this can be learned from just a few documents, imagine what can be learned from the thousands, or millions, or billions of documents you have within your own search engine.

Part of an AI-powered search platform is being able to learn insights like this from your data. The question is, how do you leverage this enormous graph of semantic knowledge in order to drive this intelligence?

Fortunately, the inherent structure of the inverted index in your search engine makes it very easy to traverse this graph without any additional explicit data modeling required. We will dive deep into how to harness this semantic knowledge graph hidden in your data in chapter 5.

2.2 The structure of natural language

In the last section we discussed how text and unstructured data typically contain a giant graph of relationships which can be derived by looking at shared terms between different records. If you've been building search engines for a while, you are used to thinking about your content in terms of "documents", "fields", and "terms" within those fields. When interpreting the semantic meaning of your content, however, there are a few more levels to consider.

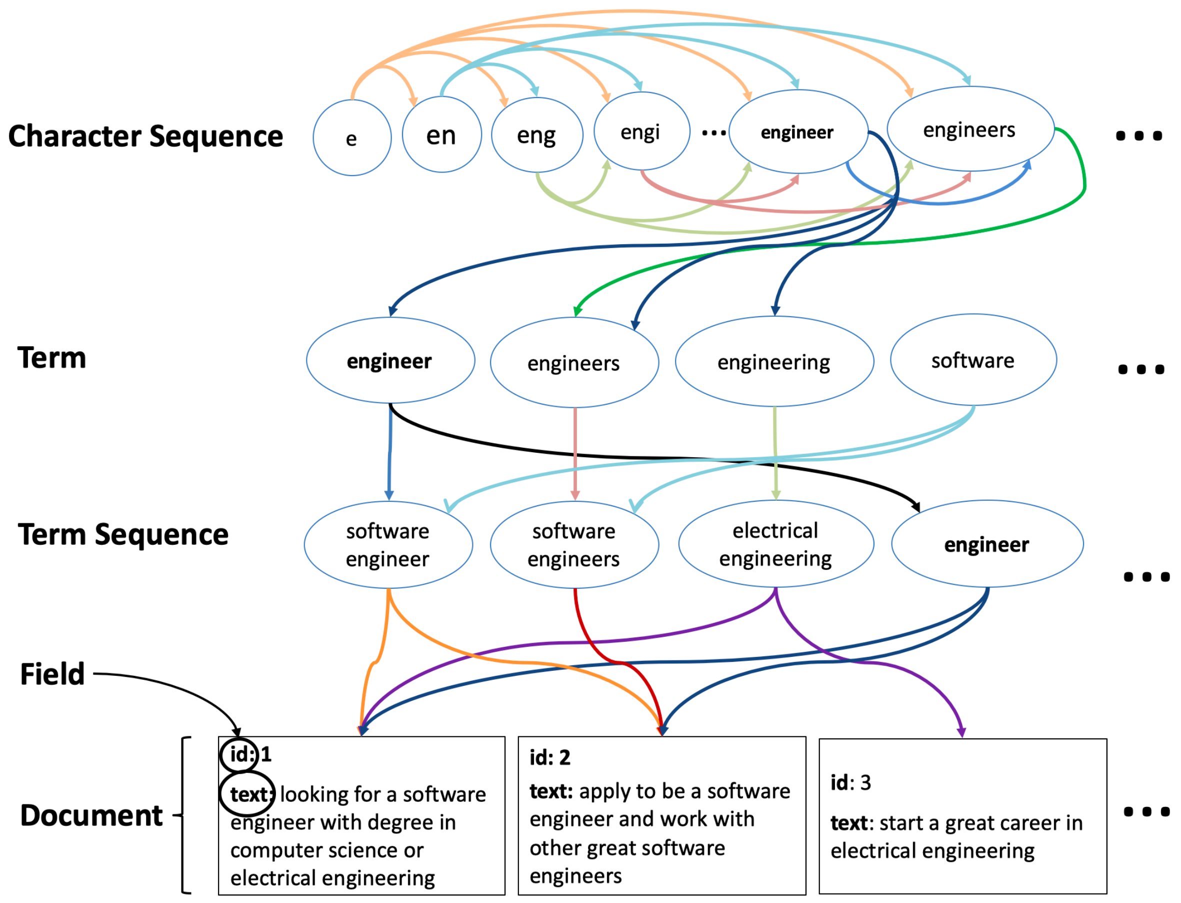

Character Sequence
e
en
eng
engi
engineer
engineers
Term
engineer
engineers
engineering
software
Term Sequence
software engineer
software engineers
electrical engineering
engineer
Field
Document
id: 1
text: looking for a software engineer with degree in computer science or electrical engineering
id: 2
text: apply to be a software engineer and work with other great software engineers
id: 3
text: start a great career in electrical engineering

Figure 2.9 Semantic data encoded into free text content. Characters form character sequences, which form terms, which form term sequences, which form fields, which form documents, which form a corpus.

Figure 2.9 walks through these additional levels of semantic meaning. At the most basic level, you have *characters*, which are single letters, numbers, or symbols, such as the letter "e" in the figure. One or more characters are then combined together to form *character sequences* such as "e", "en," eng", … "engineer", "engineers". Some character sequences form terms, which are completed words or tokens that carry an identifiable meaning, such as "engineer", "engineers", "engineering", or "software". One or more terms can then be combined together into *term sequences* - usually called "phrases" when the terms are all sequential . These include things like "software engineer", "software engineers", and "senior software engineer". For simplicity in this book, we also consider single terms to be "term sequences", and thus any time we refer to "phrases", this is also inclusive of single-term phrases.

SIDEBAR **Term Sequences vs. Phrases**

You may look at these definitions and be wondering what the difference is between a "term sequence" and a "phrase". Quite simply, a phrase is a term sequence where all of the terms appear sequentially. For example "chief executive officer" is both a phrase and a term sequence, whereas "chief officer"~2 (meaning "officer" within two positions, or edit distances, of "chief") is only a term sequence, since it specifies a sequence of terms that is not necessarily sequential. In the vast majority of cases, you will only be dealing with sequential term sequences, hence we'll mostly use the word "phrase" for simplicity throughout the book when referring inclusively to both single and multi-term sequential term sequences.

Of course, we know that multiple term sequences together can form sentences, multiple sentences can form paragraphs, and that paragraphs can then be rolled up into even larger groups of text. For the purposes of a search engine, though, the next level of grouping we'll typically focus on above term sequences is simply a *field*. Text fields can be analyzed in any number of ways using a text analyzer, which typically includes techniques like splitting on white space and punctuation, lowercasing all terms so they are case insensitive, stripping out noise (stopwords and certain characters), stemming or lemmatization to reduce terms down to a base form, and removal of accents. If the text analysis process is unfamiliar to you or you would like a refresher, I'd recommend checking out chapter 6 of *Solr in Action*.

One or more fields are then composed together into a *document*, and multiple documents form a *corpus* or collection of data. Whenever a query is executed against the search index, it filters the corpus into a *document set*, which is a subset of the corpus that specifically relates the query in question.

Each of these linguistic levels - character sequences, terms, term sequences, fields, documents, document sets, and the corpus, all provide unique insights into understanding your content and it's unique meaning within your specific domain.

2.3 Distributional semantics and word embeddings

Distributional semantics is a research area within the field Natural Language Processing that focuses upon the semantic relationships between terms and phrases based upon the distributional hypothesis. The distributional hypothesis is that words that occur in similar contexts tend to share similar meanings. It is summarized well by the popular quote: "You shall know a word by the company it keeps".[1]

TIP **"You shall know a word by the company it keeps."**
-John Rupert Firth

When applying machine learning approaches to your text, these distributional semantics become increasingly important, and the search engine makes it incredibly easy to derive the context for any linguistic representation present in your corpus. For example, if one wanted to find all documents about C-level executives, you could issue a query like:

```
c?o
```

This query would match "CEO", "CMO", "CFO, or any other CXO-style title, as it is asking for any character sequence starting with "c" and ending with "o" with a single character in-between.

The same kind of freedom exists to query for arbitrarily-complex term sequences, as well:

```
"VP Engineering"~2
```

This query would match "VP Engineering", "VP of Engineering", "Engineering VP", or even "VP of Software Engineering", as it is asking to find "VP" and "Engineering" within two positions (edit distances) of each other.

Of course, the nature of the inverted index also makes it trivial to support arbitary Boolean queries. For example, if someone searches for the term "Word", but we want to ensure any matched documents also contain either the term "Microsoft" or "MS" somewhere in the document, we could issue the following query:

```
(Microsoft OR MS) AND Word
```

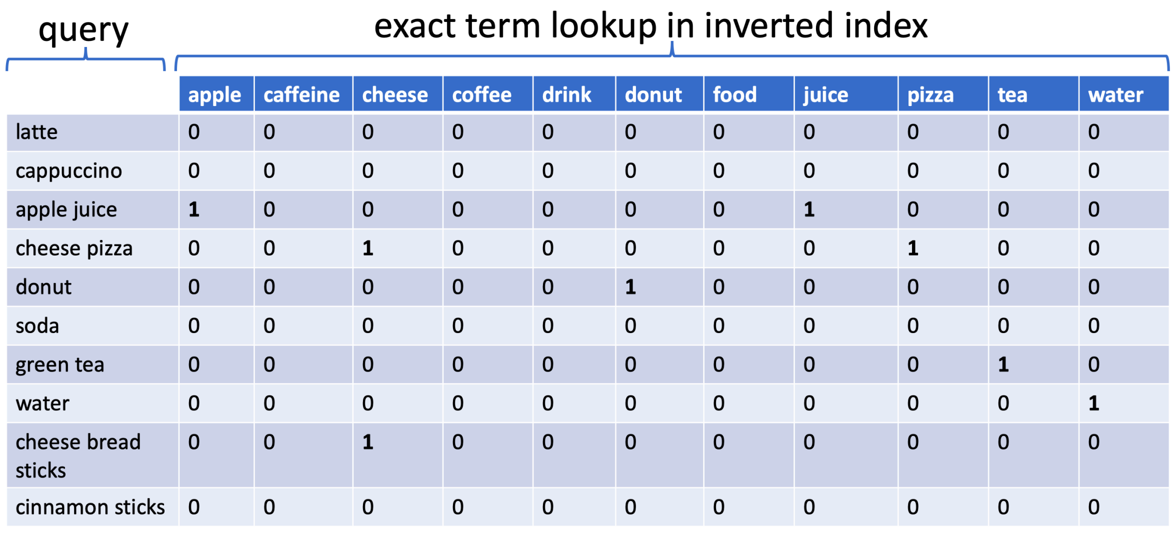
Search engines support arbitrarily complex combinations of queries for character sequences, terms, and term sequences throughout your corpus, returning document sets that supply a unique context of content matching that query. For example, if I query for the term "pizza", the documents returned are more likely going to be restaurants than car rental companies, and if I query for the term "machine learning", I'm more likely to see jobs for data scientists or software engineers than for accountants, food service workers, or pharmacists. This means that you can infer a strong relationship between "machine learning" and "software engineering", and a weak relationship between "machine learning" and "food service worker". If you dig deeper, you'll also be able to see what other terms and phrases most commonly co-occur within the machine learning document set relative to in the rest of your corpus, and thereby better understand the meaning and usage of the phrase "machine learning". We'll dive into hands-on examples of leveraging these relationships in chapter 5.

SIDEBAR **Introducing Vectors**

In this section, we first introduce the concept of a vector. A vector is essentially a list of values describing some attributes of an item. For example, if your items are houses, you may have a list of attributes like `price`, `size`, **and** `number of bedrooms`**. If you have a home costing $100,000 with 1000 square feet, and 2 bedrooms, this could be represented as the vector** `[100000, 1000, 2]`**. These attributes (price, size, and number of bedrooms) are often referred to as dimensions, and a specific collection of dimensions is called vector space. You can represent any other items (like other homes, apartments, or dwellings) within the same vector space if you can assign them values within the dimensions of the vector space. If we consider other vectors within the same vector space (for example, a house** `[1000000, 9850, 12]` **and another house** `[120000, 1400, 3]`**, we can perform mathematical operations on the vectors to learn trends and compare vectors. For example, you may intuitively look at these three vectors and determine that "home prices tend to incease as number of rooms increases" or that "number of rooms tends to increase as home size increases". We can also perform similarity calculations on vectors to determine, for example, that the $120,000 home with 1400 square feet and 3 bedrooms is more similar to the $100,000 home with 1000 square feet and 2 bedrooms than to the $1,000,000 home with 9850 square feet and 12 bedrooms. If you have not worked with vectors like this before or need a quick refresher on the math, we have included Appendix B to ensure you are up to speed comfortable with these concepts. A basic undertanding of vector operations will be important as you progress through this book.**

In recent years, the distributional hypothesis has been applied to create semantic understandings of terms and term sequences through what are known as *word embeddings*. A word embedding is a numerical vector (usually a list of floats) that is intended to represent the semantic meaning of a given term sequence (typically a word or phrase). For example The term sequence is encoded into a reduced-dimension vector which can be compared with the vectors for all of the other word embeddings within the corpus, in order to find the most semantically-related documents.

In order to understand this process, it may be useful to think of how a search engine works out of the box. Let’s imagine a vector exists for each term that contains a value (dimension) for every word in your corpus. It might look something like Figure 2.10.


query
exact term lookup in inverted index

	apple	caffeine	cheese	coffee	drink	donut	food	juice	pizza	tea	water
latte	0	0	0	0	0	0	0	0	0	0	0
cappuccino	0	0	0	0	0	0	0	0	0	0	0
apple juice	**1**	0	0	0	0	0	0	**1**	0	0	0
cheese pizza	0	0	**1**	0	0	0	0	0	**1**	0	0
donut	0	0	0	0	0	**1**	0	0	0	0	0
soda	0	0	0	0	0	0	0	0	0	0	0
green tea	0	0	0	0	0	0	0	0	0	**1**	0
water	0	0	0	0	0	0	0	0	0	0	**1**
cheese bread sticks	0	0	**1**	0	0	0	0	0	0	0	0
cinnamon sticks	0	0	0	0	0	0	0	0	0	0	0

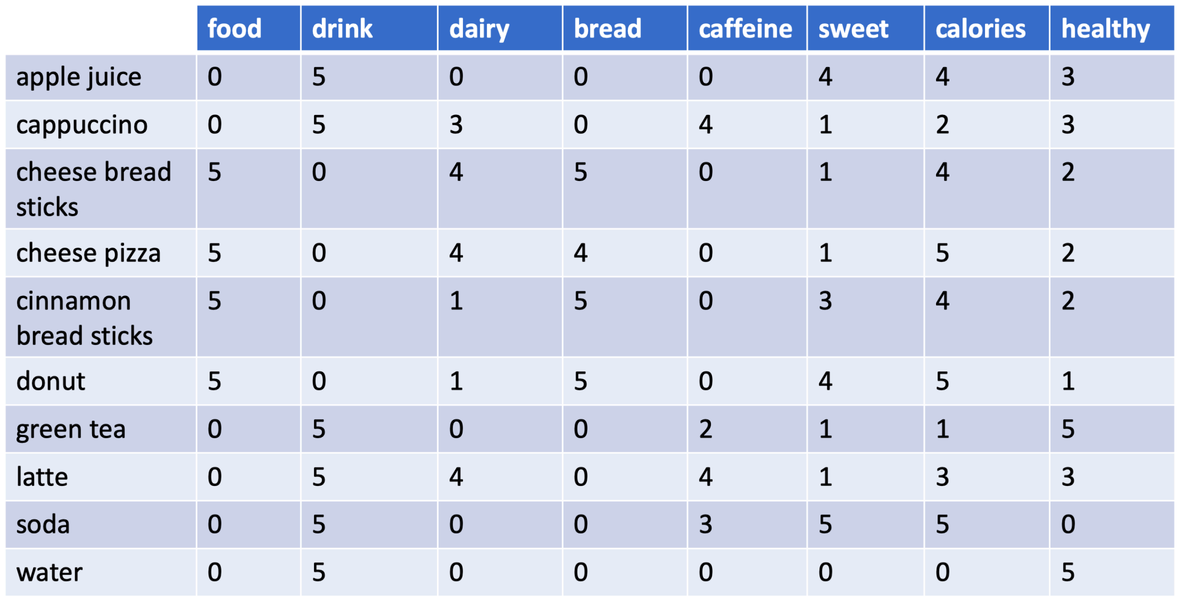
Figure 2.10 Vectors with one dimension per term in the inverted index. Every query on the left maps to a vector on the right, with a value of "1" for any term in the index that is also in the query, and a "0" for any term in the index that is not in the query.

Figure 2.10 demonstrates how document matching and similarity scoring typically works in most search engines by default. For every query, a vector exists which contains a dimension for every term that is in the inverted index. If that term exists in the query, the value in the vector is "1" for that dimension, and if that value does not exist in the query, then the value is "0" for that dimension. A similar vector exists for every document in the inverted index, with a "1" value for any term from the index that appears in the document, and a zero for all other terms.

When a query is executed, an exact lookup occurs in the index for any matched terms (post-text-analysis), and then a similarity score is calculated based on a comparison of the vector for the query and the vector for the document that is being scored relative to the query. We'll walk through the specific scoring calculation further in chapter 3, but that high-level understanding is sufficient for now.

There are obvious downsides to this approach. While it is great for finding documents with exact keywords matches, what happens when you want to find "related" things instead? For example, you'll notice in Figure 2.10 that the term "soda" appears in a query, but never in the index. Even though there are other kinds of drinks (apple juice, water, cappuccino, and latte), the search engine will always return zero results because it doesn't understand that the user is searching for a drink. Similarly, you'll notice that even though the term caffeine exists in the index, that queries for "latte", "cappuccino", and "green tea" will never match the term caffeine, even though they are related.

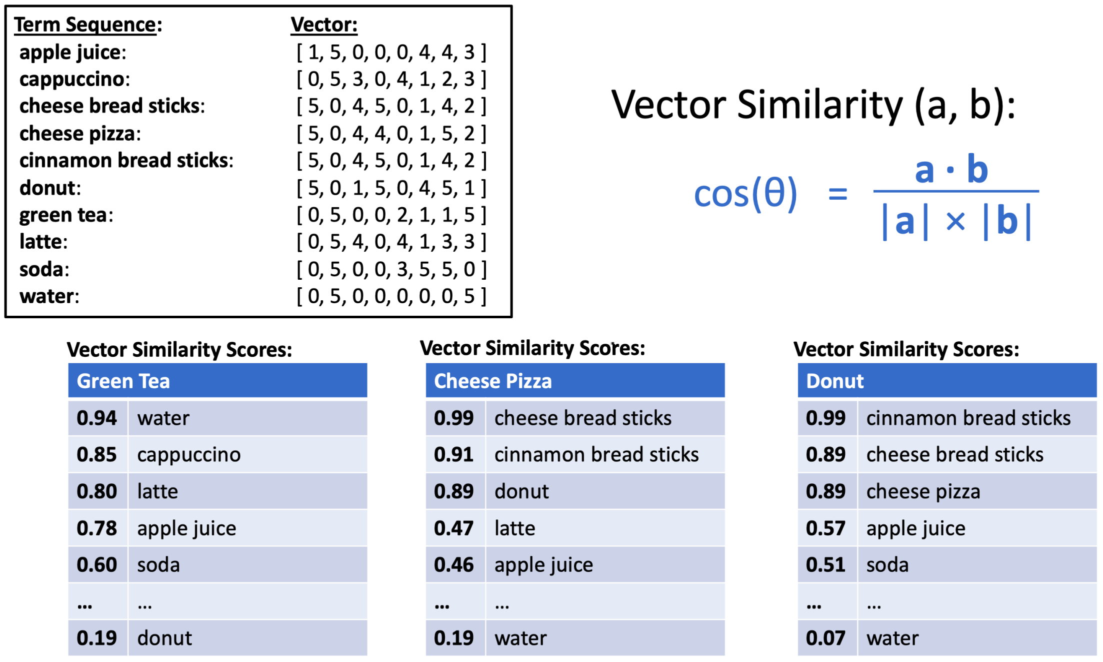
For these reasons, it is now a common practice to use something called word embeddings to model a semantic meaning for term sequences in your index and queries. A *word embedding* for a term is a vector of features which represents the term's conceptual meaning in a semantic space. Figure 2.11 demonstrates the terms now mapped to a dimensionally-reduced vector that can serve as a word embedding.



	food	drink	dairy	bread	caffeine	sweet	calories	healthy
apple juice	0	5	0	0	0	4	4	3
cappuccino	0	5	3	0	4	1	2	3
cheese bread sticks	5	0	4	5	0	1	4	2
cheese pizza	5	0	4	4	0	1	5	2
cinnamon bread sticks	5	0	1	5	0	3	4	2
donut	5	0	1	5	0	4	5	1
green tea	0	5	0	0	2	1	1	5
latte	0	5	4	0	4	1	3	3
soda	0	5	0	0	3	5	5	0
water	0	5	0	0	0	0	0	5

Figure 2.11 Word embeddings with reduced dimensions. In this case, instead of one dimension per term (exists or missing), now higher-level dimensions exist that score shared attributes across items such as "healthy", contains "caffeine" or "bread" or "dairy", or whether the item is "food" or a "drink".

With a new word embedding vector now available for each term sequence in the left-most column of Figure 2.11, we can now score the relationship between each pair of term sequences leveraging the similarity between their vectors. In Linear Algebra, we will use a cosine similarity function or other distance measure to score the relationship between two vectors, which is simply computed by performing a dot product between the two vectors and scaling it by the magnitudes (lengths) of each of the vectors. We'll visit the math in more detail in the next chapter, but for now, Figure 2.12 shows the results of scoring the similarity between several of these vectors.



Term Sequence:	Vector:
apple juice:	[1, 5, 0, 0, 0, 4, 4, 3]
cappuccino:	[0, 5, 3, 0, 4, 1, 2, 3]
cheese bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
cheese pizza:	[5, 0, 4, 4, 0, 1, 5, 2]
cinnamon bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
donut:	[5, 0, 1, 5, 0, 4, 5, 1]
green tea:	[0, 5, 0, 0, 2, 1, 1, 5]
latte:	[0, 5, 4, 0, 4, 1, 3, 3]
soda:	[0, 5, 0, 0, 3, 5, 5, 0]
water:	[0, 5, 0, 0, 0, 0, 0, 5]

Vector Similarity (a, b):

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \times |\mathbf{b}|}$$

Vector Similarity Scores:

Green Tea	
0.94	water
0.85	cappuccino
0.80	latte
0.78	apple juice
0.60	soda
...	...
0.19	donut

Vector Similarity Scores:

Cheese Pizza	
0.99	cheese bread sticks
0.91	cinnamon bread sticks
0.89	donut
0.47	latte
0.46	apple juice
...	...
0.19	water

Vector Similarity Scores:

Donut	
0.99	cinnamon bread sticks
0.89	cheese bread sticks
0.89	cheese pizza
0.57	apple juice
0.51	soda
...	...
0.07	water

Figure 2.12 Similarity between Word Embeddings. The dot product between vectors shows the items list sorted by similarity with "green tea", with "cheese pizza", and with "donut".

As you can see in Figure 2.12, since each term sequence is now encoded into a vector that represents its meaning in terms of higher-level features, this vector (or word embedding) can now be used to score the similarity of that term sequence with any other similar vector. You'll see three vector similarity lists at the bottom of the figure: one for "green tea", one for "cheese pizza", and one for "donut".

By comparing the vector similarity of "green tea" with all the other term sequences, we find that the top most related items are "water", "cappuccino", "latte", "apple juice", and "soda", with the least related being "donut". This makes intuitive sense, as "green tea" shares more attributes with the items higher in the list. For the "cheese pizza" vector, we see that the most similar other word embeddings are for "cheese bread sticks", "cinnamon breads sticks", and "donut", with "water" being at the bottom of the list. Finally, for the query "donut", we find the top items to be "cinnamon bread sticks", "cheese bread sticks", and "cheese pizza", with "water" once again being at the bottom of the list. These results do a great job of finding the most similar items to our original query.

It's worth noting that this vector scoring is only used in the calculation of similarity between items. In your search engine, there's usually a two-phase process whereby you first execute a keyword search and then score the resulting documents. Unless you're going to skip the first step and score all of your documents relative to your query vectors (which can be time and processing intensive), you'll still need some combination of initial keyword or document set filtering. We'll dive more into these mechanics for successfully implementing word embeddings and vector search in chapter 13.

These higher-level attribute vectors we’ve discussed might represent other term sequences in queries, or they could be term sequences within documents, or they could even be entire documents. It is commonplace to encode terms and term sequences into word embeddings, but *sentence embeddings* (encoding a vector for an entire sentence), *paragraph embeddings* (encoding a vector for an entire paragraph), and *document embeddings* (encoding a vector for an entire document) are also common techniques. It’s also very common that dimensions themselves are more abstract than our examples here. For example, deep learning models may be applied that pull out seemingly unitelligible features from character sequences and the way that documents cluster together within the corpus. We wouldn’t be able to easily label such dimensions in the embedding vector, but as long as it improves the predictive power of the model to increase relevance, this is often not a huge concern for most search teams.

Ultimately, combining multiple models for harnessing the power of distributional semantics and word embeddings tends to create the best outcomes, and we’ll dive further into numerous graph and vector-based approaches to leveraging these techniques throughout the rest of this book.

2.4 Modeling domain-specific knowledge

In chapter 1, we discussed the Search Intelligence Progression (refer to Figure 1.9), whereby organizations start with basic keyword search, and progress through several additional stages of improvement before they ultimately achieve a full self-learning system. The second stage in that search intelligence progression was that of building taxonomies and ontologies, and the third stage ("query intent") included the building and use of knowledge graphs. Unfortunately, there can sometimes be significant confusion among practitioners in the industry on proper definitions and use of key terminology like "ontology", "taxonomy", "synonym lists", "knowledge graphs", "alternative labels", and so on, so it will benefit us to provide some definitions for use in this book so as to prevent any ambiguity. Specifically, we’ll lay out definitions for the key terms of "knowledge graph", "ontology", "taxonomy", "synonyms", and "alternative labels". Figure 2.13 shows a high level diagram for how they relate.

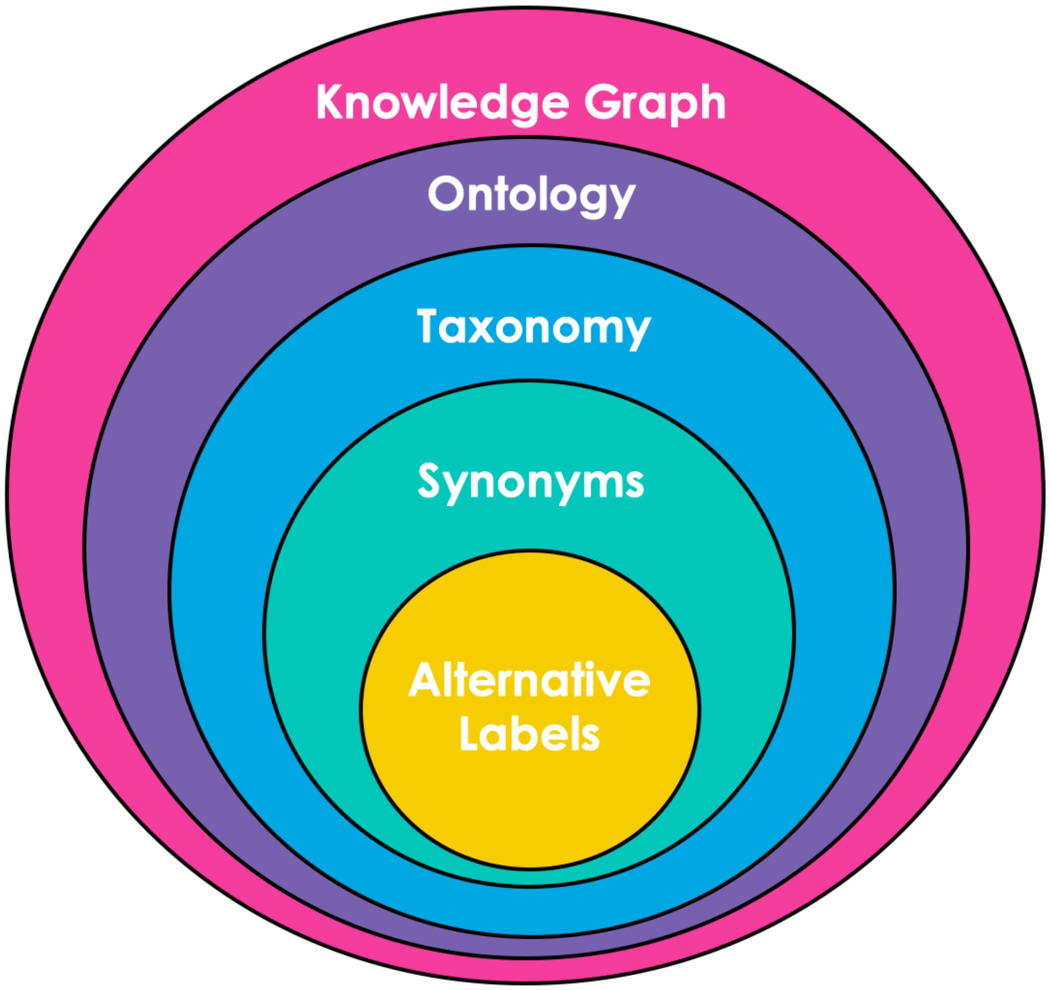


Figure 2.13 Levels of domain-specific knowledge modeling. Knowledge graphs extend ontologies, which extend taxonomies. Synonyms extend alternative labels and map to entries in taxonomies.

We define each of these knowledge modeling techniques as follows:

- **Alternative Labels** (or Alt. Labels): Substitute term sequences with identical meanings.

```
Examples:
  CTO => Chief Technology Officer
  specialise => specialize
```

- **Synonyms**: Substitute term sequences that can be used to represent the same or very similar things.

```
Examples:
  human => homo sapiens, mankind
  food => sustenance, meal
```

- **Taxonomy**: A classification of things into categories.

```
Examples:
  human is mammal
  mammal is animal
```

- **Ontology**: A mapping of relationships between types of things

```
Examples:
  animal eats food
  human is animal
```

- **Knowledge Graph**: An instantiation of an Ontology that also contains the things that are related

```
Examples:
  John is human
  John eats food
```

Creation of Alternative Labels is the most straight-forward of these techniques to understand. Acronyms (RN => Registered Nurse) virtually always serve as alternative labels, as do misspellings and alternative spellings. Sometimes it is useful to keep these mappings stored in separate lists, particularly if you’re using algorithms to determine them and you expect to allow for human modification of them and/or plan to re-run the algorithms later.

Synonyms are the next most common of the techniques, as virtually every search engine will have some implementation of a synonyms list. Alternative labels are a subset of a synonyms list and are the most obvious kind of synonym. Most people consider "highly related" term sequences to be synonyms, as well. For example, "software engineer" and "software developer" are often considered synonyms since they are usually used interchangably, even though there are some slight nuances in meaning between the two. Sometimes, you’ll even see translations of words between languages showing up in synonyms for bilingual search use cases.

One key difference between Alternative Labels and more general Synonyms is that alternative labels can be seen as "replacement" terms for the original, whereas synonyms are more often used as "expansion" terms to add alongside the original. Implementations can vary widely, but this ultimately boils down to whether you are confident two term sequences carry exactly the same meaning (and want to normalize it), or whether you’re just trying to include additional related term sequences so you don’t miss other relevant results.

Taxonomies are the next step up from synonyms. Taxonomies focus less on substitute or expansion words, and instead focus on categorizing your content into a hierarchy. Taxonomical information will often be used to drive website navigation, to change behavior with a subset of search results (for example, show different faceting or filtering options based upon a parent product category), or to apply dynamic filtering based upon a category to which a query maps. For example, if someone searches for "range" on a home improvement website, the site might automatically filter down to "appliances" to remove the noise of other products which contain phrases like "fall within the range" in their product description. Synonyms then map into a taxonomy as pointers to particular items within the taxonomy.

Whereas taxonomies tend to specify parent-child relationships between categories and then map things into those categories, ontologies provide the ability to define much richer relationships between things (term sequences, entities) within a domain. Ontologies typically define more abstract relationships, attempting to model the relationships between kinds of things in a domain - for example, "employee reports to boss", "CMO’s boss is CEO", "CMO is employee". This makes ontologies really useful for deriving new information from known facts by mapping the facts into the ontology and then drawing logical conclusions based upon relationships in the ontology that can be applied to those facts.

Knowledge Graphs are the relative newcomer to the knowledge management space. Whereas ontologies define high-level relationships which apply to types of things, knowledge graphs tend to be full instantiations of ontologies that also include each of the specific entities that fall within those types. Using our previous ontology example, a knowledge graph would additionally have "Michael is CMO", "Michael reports to Marcia", and "Marcia is CEO" as relationships in the graph. Before knowledge graphs came into the forefront, it was common for these more detailed relationships to be modeled into ontologies, and many people still do this today. As a result, you’ll often see the terms knowledge graph and ontology used interchangably, though this is becoming less common over time.

Throughout this book, we will mostly focus our discussions on alternative labels, synonyms, and knowledge graphs, since taxonomies and ontologies are mostly subsumed into knowledge graphs.

2.5 Challenges in natural language understanding for search

In the last few sections, we’ve discussed the rich graph of meaning embedded within unstructured data and text, as well as how distributional semantics and word embeddings can be leveraged to derive and score semantic relationships between term sequences in queries and documents. We also introduced key techniques for knowledge modeling and defined key terminology we’ll use throughout this book. In this section, we’ll discuss a few key challenges associated with Natural Language Understanding that we’ll seek to overcome in the coming chapters.

2.5.1 The challenge of ambiguity (polysemy)

In section 2.1.3, we introduced the idea of polysemy, or ambiguous terms. In that section, we were dealing with an image tagged with the name "Trey Grainger", but which was referring to a different person than the author of this book. In textual data, however, we have the same problem, and it can get very messy.

Consider a word like "driver". Driver can refer broadly to a "vehicle driver", a kind of golf club for hitting the ball off a tee, software that enables a hardware device to work, a kind of tool (screwdriver), or the impetus for pushing something forward ("a key driver of success"). Clearly there are many potential meaning for this word, but in reality you could dive in and explore many even more granular meanings. For example, within the "vehicle driver" category, it could mean taxi driver, or Uber driver, or Lyft driver, or it could mean professional trucker like a CDL driver (someone with a Commercial Drivers License), or it could mean bus driver. Within the subset of bus drivers, it could mean a school bus driver, a driver of a public city bus, a driver for a tour bus, and so on. This list could continue being broken down into dozens of additional categories at a minimum.

Often times when building search applications, engineers will naively create static synonyms lists and assume terms have a singular meaning that can be applied universally. The reality, however, is that every term (word or phrase) takes on a unique meaning that is based upon the specific context in which it is being used.

TIP **Every term takes on a unique meaning that is based upon the specific context in which it is being used.**

It’s not often practical to support an infinite number of potential meanings, though we will discuss techniques to approximate this with a semantic knowledge graph in chapter 5. Nevertheless, regardless of whether you support many meanings per phrase or just a few, it’s important to recognize the clear need to be able to generate an accurate (and often nuanced) interpretation for any given phrase your users may encounter.

2.5.2 The challenge of understanding context

I like to say that every term (word or phrase) you ever encounter is a "context-dependent cluster of meaning with an ambiguous label".

TIP **Every word or phrase is "a context-dependent cluster of meaning with an ambiguous label"**

That is to say, there is a label (the textual representation of the term) that is being applied to some concept (a cluster of meaning) that is dependent upon the context in which it is found. By this definition, it is impossible to ever interpret a term without an understanding of the context in which it is found, and as such, creating fixed synonyms lists that aren’t able to take context into account is likely to create suboptimal search experiences for your users.

As we discussed in chapter 1, the context for a query includes more than just the search keywords and the content within your documents, however. It also includes an understanding of your domain, as well as an understanding of your user. Queries can take on entirely different meaning based upon what you know about your user and any domain-specific understanding you may have. This context is necessary to both detect and to resolve the kinds of ambiguity we discussed in the last section, as well as to ensure your users are receiving the most intelligent search experience possible.

Throughout this book, our focus will be on techniques to automatically learn contextual interpretations of each query based upon the unique context in which it is being used.

2.5.3 The challenge of personalization

Just because context is important doesn't mean it is always easy to apply correctly. It is always necessary to have the ability to perform basic keywords search as a fallback for when your system doesn't understand a query, and it is almost always useful to have pre-built domain understanding that can similarly be relied upon to help interpret queries. This pre-built domain understanding then ends up overriding some of the default keyword-based matching behavior (such as joining individual keywords into phrases, injecting synonyms, and correcting misspellings).

Once you begin better understanding your users, however, it is not always obvious how to apply user-specific personalization on top of the pre-existing content and domain-specific scoring. For example, say you learn that a particular user really likes Apple as a brand because they keep searching for iPhones. Does this mean that Apple should also be boosted when they are searching for watches and computers and television streaming boxes and keyboards and headphones and music players? It could be that the user only likes Apple-branded phones and that by boosting the brand in other categories you may actually frustrate the user. For example, even if the user did search for iPhone previously, how do you know they weren't just trying to compare the iPhone with other phones they were considering?

Out of all of the dimensions of user intent (figure 1.8), personalization is the easiest one to trip up on, and subsequently it is the one that is least-commonly seen in modern AI-powered search applications (outside of recommendation engines, of course). We'll work through these problems in detail in chapter 9 in order to highlight how to strike the right balance when rolling out a personalized search experience.

2.5.4 Challenges interpreting queries vs. documents

One common problem I see when engineers and data scientists first get started with search is a propensity to apply standard Natural Language Processing techniques like language detection, part of speech detection, phrase detection, and sentiment analysis to queries. All of those techniques were designed to operate on longer blocks of text - usually at the document, paragraph, or at least sentence level.

Documents tend to be longer and to supply significantly more context to the surrounding text, whereas queries tend to be short (a few keywords only) in most use cases, and even when they are longer they tend to combine multiple ideas together as opposed to supplying more linguistic context.

As such, when trying to interpret queries, you need to leverage external context as much as possible to interpret the query. Instead of using a Natural Language Processing library that typically relies upon sentence structure to interpret the query, for example, you can try to lookup the phrases from your query in your corpus of documents to find the most common domain-specific interpretations of them. Likewise, you can leverage the co-occurrence of terms within your query across previous user search sessions by mining your user behavioral signals. This enables you to learn real intention from similar users, which would be very challenging to accurately derive from a standard Natural Language Processing library on a consistent basis.

In short, queries need special handling and interpretation due to their tendency to be short and to often imply more than they state explicitly, so fully leveraging search-centric data science approaches to queries is going to generate much better results than traditional Natural Language Processing approaches.

2.5.5 Challenges interpreting query intent

While the process of parsing a query to understand the terms and phrases it contains is important, there is often a higher-level intent behind the query—a query type, if you will. For example, lets consider the inherent differences between following queries:

```
who is the CEO?
support
iphone screen blacked out
iphone
verizon silver iphone 8 plus 64GB
sale
refrigerators
pay my bill
```

The intent of the first query for "who is the CEO?" is clearly to find a factual answer and not a list of documents. The second query for "support" is trying to navigate to the support section of a website, or to otherwise contact the support team. The third query for "iphone screen blacked out" is also looking for support, but it is for a specific problem, and the person is likely to want to find troubleshooting pages that may exist to help with that specific problem before reaching out to the actual support team.

The next two queries for "iphone" and for "verizon silver iphone 8 plus 64GB" are quite interesting. While they are both for iphones, the first search is a general search, indicating a likely browsing or product research intent, whereas the second query is a much more specific variant of the first search, indicating the user knows exactly what they are looking for, and may be closer to making a purchasing decision. As such, the general query for "iphone" may do better to return a landing page that provides an overview of iphones and the available options, while the more specific query may do better to go straight to the product page with a purchase button immediately available. As a general rule of thumb, the more general a query, the more likely the user is just browsing, whereas more specific queries—especially when they refer to specific items by name—often indicate a purchase intent or desire to find a particular known item.

The query for "sale" indicates that the user is looking for items which are available for purchase at a discounted rate, which will invoke some specially-implemented filter or redirect to a particular landing page for an ongoing sale event. The query for "refrigerators" indicates that the use wants to browse a particular category of product documents. Finally, the query for "pay my bill" indicates that the user wants to take an action—the best response to this query isn’t a set of search results or even an answer, but instead a redirect to a bill review and payment section of the application.

Each of these queries contains an intent beyond just a set of keywords to be matched. Whether the intent is to redirect to a particular page, to apply particular filters, to browse or to purchase items, or even to take domain-specific actions, the point is that there is domain-specific nuance to how users may express their goals to your search engine. Often times, it can be difficult to automatically derive these domain-specific user intents automatically. It is fairly common for businesses to implement specific business rules to handle these as one-off requests. Query Intent classifiers can certainly be built to handle subsets of this problem, but successfully interpreting every possible query intent still remains challenging when building out natural language query interpretation capabilities.

2.6 The fuel powering AI-powered search

In the first chapter, we introduced the idea of Reflected Intelligence - leveraging feedback loops to continually learn from both content and user interactions. This chapter has focused entirely on understanding the meaning and intelligence embedded within your content, but it’s important to recognize that many of the techniques we’ll apply to the "unstructured data" in your documents can also be just as readily applied to your user behavioral signals. For example, we discussed earlier in this chapter how the meaning of phrases can be derived from finding the other phrases that they appear with the most often within your corpus. We used the example that "machine learning" appears more commonly with "data scientist" and "software engineer" than it does with "accountants", "food service workers", or "pharmacists".

If you abstract this idea beyond your documents and to your users' behavior, you might also expect that the users querying your search engine are likely to exhibit similar query behavior that also falls inline with the distributional hypothesis. Specifically, people who are data scientists or who are searching for data scientists are far more likely to also search for or interact with documents about "machine learning", and the likelihood of a food service worker or accountant searching for machine learning content is much lower than the likelihood for a software engineering doing the same. We can thus apply these same techniques to learn related terms and term sequences from query logs, where instead of thinking of terms and term sequences mapping to fields in documents, we think of terms in queries and clicks on search results mapping to user sessions, which then map to users.

Some search applications will be content-rich, but have very few user signals. Other search applications will have an enormous number of signals, but will have very little content or have content which poses challenges from an automated learning perspective. In an ideal scenario, you have great content and an enormous quantity of user signals to learn from, which allows combining the best of both worlds into an even smarter AI-powered search application. Regardless of which scenario you're in, keep in mind that your content and your user signals can both serve as fuel for your application, and you should do your best to maximize the collection and quality of collection of each.

Now that we've covered all the background needed to begin extracting meaning from your natural language content, it's time to roll up your sleeves and get hands-on. In the next chapter, we'll dive into lots of examples as we begin to explore content-based relevancy in an AI-powered search application.

2.7 Summary

- Unstructured data is a misnomer - it is really more like hyper-structured data, as it represents a giant graph of domain-specific knowledge.
- Search engines can leverage distributional semantics - interpreting the semantic relationships between terms and phrases based upon the distributional hypothesis - to harness rich semantic meaning at the level of character sequences, terms, term sequences (typically phrases), fields, documents, document sets, and an entire corpus.
- Distributional semantics approaches enable us to learn the nuanced meaning of our queries and content from their larger surrounding context.
- Word embeddings are a powerful technique for modeling and scoring based upon the semantic meaning of phrases instead of just pure text matching statistics.
- Domain specific knowledge is commonly modeled through a combination of alternative labels, synonyms lists, taxonomies, ontologies, and knowledge graphs. Knowledge graphs typically model the output from each of the other approaches into a unified knowledge representation of a particular domain.
- Polysemy (ambiguous terms), context, personalization, and query-specific Natural Language Processing approaches represent some of the more interesting challenges in natural language search.
- Content and user signals are both important fuel for our AI-powered search applications to leverage when solving natural language challenges.

3 Ranking and content-based relevance

This chapter covers

- **Executing queries and returning matching search results**
- **Ranking search results based upon how relevant they are to an incoming query**
- **Controlling and specifying your own ranking functions with function queries**
- **Catering ranking functions to a specific domain**

Search engines fundamentally do three things: ingest content, return content matching incoming queries, and sort the returned content based upon some measure of how well it matches the query. *Relevance* is the term used to describe this notion of "how well the content matches the query". Most of the time the matched content is documents, and the returned and ranked content is those matched documents along with some corresponding metadata describing the documents.

In most search engines, the default relevance sorting is based upon a score indicating how well each keyword in a query matches the same keyword in each document, with the best matches yielding the highest relevance score and returned at the top of the search results. The relevance calculation is highly configurable, however, and can be easily adjusted on a per-query-basis in order to enable very sophisticated ranking behavior.

In this chapter, we will provide an overview of how relevance is calculated, how the relevance function can be easily controlled and adjusted through function queries, and how to implement popular domain-specific and user-specific relevance ranking features. We'll start by looking at how ranking actually works.

3.1 Scoring query and document vectors with cosine similarity

In section 2.3, we demonstrated the idea of measuring the similarity of two vectors by calculating the cosine between them. We created vectors (lists of numbers, where each number represents the strength of some feature) representing different food items, and we then calculated the cosine (the size of the angle between the vectors) in order to determine their similarity. We'll expand upon that technique in this section, discussing how text queries and documents can map into vectors for ranking purposes. We'll further get into some popular text-based feature weighting techniques and how they can be integrated to create an improved relevance ranking formula.

SIDEBAR **Running the Code Examples**

All code listings in the book are available in Jupyter notebooks, enabling you to run live versions of the code in real-time and easily follow along as you read through the chapters. See Appendix A: "Running the Examples" for instructions on how to launch the Jupyter notebooks and follow along in your web browser.

For your convenience, electronic copies of the book also contain hyperlinks for each listing, making it seamless for you to link directly to the live code examples from the text if you are running the Jupyter notebooks on the same device. For brevity, listings may leave out certain lines of code, such as imports ancillary code, so going to the notebook will also enable you to dive deeper into some of those implementation details when needed.

In this section, we will begin diving into our first code listings for the book, so it will be helpful to fire up our live code examples and open the chapter 3 Jupyter Notebooks so that you can follow along. Instructions for doing this are located in Appendix A: "Running the Code Examples".

3.1.1 Mapping text to vectors

In a typical search application, we start with a collection of documents and we then try to rank documents based upon how well they match some user's query. In this section, we'll walk through the process of mapping the text of queries and documents into vectors.

In the last chapter, we used the example of a search for food and beverage items, like `apple juice`, so let's reuse that example here.

Query: `apple juice`

Let's assume we have two different documents that we would like to sort based upon how well they match this query.

Document 1:

```
Lynn: ham and cheese sandwich, chocolate cookie, ice water.
Brian: turkey avocado sandwich, plain potato chips, apple juice
Mohammed: grilled chicken salad, fruit cup, lemonade
```

Document 2:

```
Orchard Farms apple juice is premium, organic apple juice made from the freshest apples, never
➥from concentrate. Its juice has received the regional award for best apple juice three
➥years in a row.
```

If we mapped both of these documents (containing a combined 48 words) to vectors, they would map to a 48-word vector-space with the following dimensions:

```
[a, and, apple, apples, avocado, award, best, brian, cheese, chicken, chips, chocolate,
➥concentrate, cookie, cup, farms, for, freshest, from, fruit, grilled, ham, has, ice, in, is,
➥its, juice, lemonade, lynn, made, mohammed, never, orchard, organic, plain, potato, premium,
➥received, regional, row, salad, sandwich, the, three, turkey, water, years]
```

If you recall in section 2.3, we proposed thinking of a query for the phrase `apple juice` as a vector containing a feature for every word in any of our documents, with a value of `1` for the terms `apple` and `juice`, and a value of `0` all other terms.

Since the word `apple` is in the third position and `juice` is in the 28th position of our 48-word vector-space, a query vector for the phrase `apple juice` would look as shown in Figure 3.1.

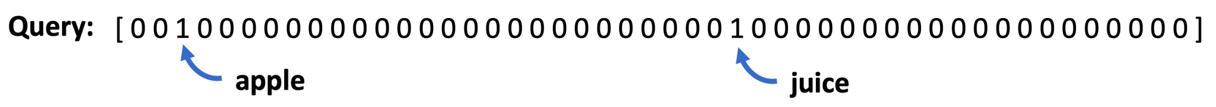


Figure 3.1 Query Vector. The query for `apple juice` is mapped to a vector containing one dimension for every known term, with a value of `1` for the terms `apple` and `juice` and a value of `0` for all other terms.

Note that even though the query vector only contains a non-zero value for two dimensions (representing the position of `apple` and `juice`), that it still contains values of `0` for all other possible dimensions. Representing a vector like this including every possible value is known as a *dense vector representation*.

Each of the documents also maps to the same vector space based upon each of the terms it contains:

```
Document 1: [0 1 1 0 1 0 0 1 1 1 1 1 0 1 1 0 0 0 0 1 1 1 0 1 0 0 0 1 1 1 0
➥1 0 0 1 1 1 0 0 0 0 1 1 0 0 1 1 0]
```

```
Document 2: [1 0 1 1 0 1 1 0 0 0 0 0 1 0 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 0 1
➥0 1 1 0 0 0 1 1 1 1 0 0 1 1 0 0 1]
```

With these dense vector representations of our query and documents, we can now use linear algebra to measure the similarity between our query vector and each of the document vectors.

3.1.2 Calculating similarity between dense vector representations

To rank our documents, we just need to follow the same process we used in chapter 2 to calculate the cosine between each document and the query. This cosine value will then become the relevance score for each document, and we'll be able to sort documents based upon that relevance score.

Listing 3.1 shows how we would represent the query and document vectors in code, and how we would calculate the cosine similarity between the query and each document.

Listing 3.1 Cosine Similarity calculation between query and document vectors

```
query_vector = np.array([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
➥0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
➥0, 0, 0, 0, 0, 0, 0])

doc1_vector = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
➥0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0,
➥1, 1, 0, 0, 1, 1, 0])

doc2_vector = np.array([1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1,
➥1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
➥0, 0, 1, 1, 0, 0, 1])

def cos_sim(vector1, vector2):
  return dot(vector1, vector2) / (norm(vector1) * norm(vector2))


doc1_score = cos_sim(query_vector, doc1_vector)
doc2_score = cos_sim(query_vector, doc2_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " +
➥num2str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 0.2828
 doc2: 0.2828
```

Interesting… Both documents received exactly the same relevance score, even though the documents contain lengthy vectors with very different content. It might not be immediately obvious what's going on, so let's simplify the calculation by focusing only on the features that matter.

3.1.3 Calculating similarity between sparse vector representations

The key to understanding the calculation in the last section is understanding that the only features that matter are the ones shared between both the query and a document. All other features (words appearing in documents that don’t match the query) have zero impact on whether one document is ranked higher than another. As a result, we can remove all of the other insignificant terms from our vector to simplify the example, converting from a dense vector representation to what is known as a sparse vector representation.

In the last section, we worked with dense vector representations, which are vector representations including a value for every possible dimension, even if many of those values are `0` (see Figure 3.1). It is also possible to create a much smaller vector representation that only contains useful values, however. A vector representation only containing the non-zero values for a calculation is known as a *sparse vector representation*, as shown in Figure 3.2.

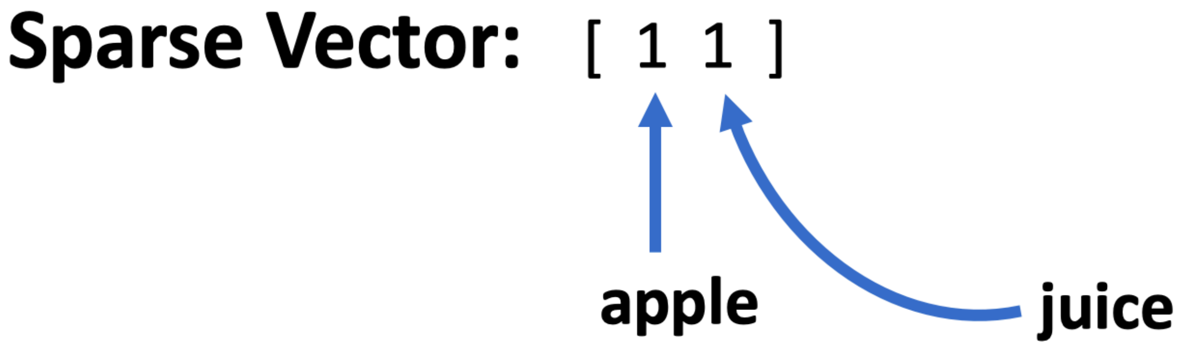


Figure 3.2 Sparse vector representation, which only contains the "present" features, unlike dense vector representations which also contain the `0` valued entries for every feature.

In most search engine scoring operations, we tend to deal with sparse vector representations because they are more efficient to work with when we are only dealing with scoring based upon the small number of features.

In addition, we can further simplify our calculations by creating sparse vectors that only include "meaningful entries" - the terms that are actually present in the query - as shown in Listing 3.2.

Listing 3.2 Cosine Similarity calculation between sparse query and document vectors

```
sparse_query_vector = [1, 1] #[apple, juice]
sparse_doc1_vector = [1, 1]
sparse_doc2_vector = [1, 1]

doc1_score = cos_sim(sparse_query_vector, sparse_doc1_vector)
doc2_score = cos_sim(sparse_query_vector, sparse_doc2_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " +
➥num2str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 1.0
 doc2: 1.0
```

Notice that `doc1` and `doc2` still yield the same relative score, but that now the actual score is `1.0`. If you remember, a `1.0` score from a cosine calculation means the vectors are perfect matches, and in fact, it should be obvious that since each of the sparse vectors contain the exact same values (`[1 1]`) that they are all equal and thus get a perfect score.

In fact, you'll notice several very interesting things:

- This simplified sparse vector calculation still shows both `doc1` and `doc2` returning equivalent relevance scores, since they both match all the words in the query.
- Even though the absolute score between the dense vector representation similarity (0.2828) and the sparse vector representation similarity (1.0) are different due to our new sparse vector only containing terms actually in the query, the scores are still the same relative to each other within each vector type.
- The feature weights for the two query terms (`apple`, `juice`) are exactly the same between the query and each of the documents, resulting in a cosine score of 1.0.

SIDEBAR **Vectors vs. Vector Representations**

We've been careful to refer to "dense vector representations" and "sparse vector representations" instead of "dense vectors" and "sparse "vectors". This is because there is an conceptual distinction between the idea of a vector and its representation.

Specifically, a dense vector is any vector that contains mostly non-zero values, whereas a sparse vector is any vector that contains mostly zero-values, regardless of how they are stored or represented. The vector representations, on the other hand, deal with the data structures we actually use to work with the vectors, and so since our query and document vectors are all sparse vectors, it makes sense to use a sparse vector representation to work with only the non-zero values.

In addition to these academic terminology distinctions, however, when we are working with our sparse vector representations, we are not actually preserving the original vector, but instead we are creating a brand new vector containing only the values that are present in the query. Since we've determined that values not in the query do not affect the relative difference in score between any two documents, we simplify our calculations by creating a new sparse vector that only contains the values present in the query and, of course, we use a sparse vector representation to work with this sparse new vector.

Search engines adjust for these issues by not just considering each feature in the vector as a `1` (exists) or a `0` (does not exist), but instead providing a score for each feature based upon how *well* the feature matches. We'll discuss a few ways to do this in the following sections.

3.1.4 Term Frequency (TF): measuring how well documents match a term

The problem we encountered in the last section is that the features in our term vectors only signify *if* the word `apple` or `juice` exists, not how well each document actually represents either of the terms. We can see that the oddity of representing each term from the query as a feature with a value of `1` indicating it exists, is that both `doc1` and `doc2` will always have the same cosine similarity score for the query, even though qualitatively `doc2` is a much better match since it talks about apple juice much more.

Instead of using a value of `1` for each existing term, we can try to correct for this notion of "how well" a document matches by using the *term frequency*, which is a measure of the number of times a term occurs within each document. The idea here is that the more frequently a term occurs within a specific document, the greater the liklihood that the document is more related to the query.

If we replace the feature weights in our vector with a count of the number of times each term occurs within the document or query, then we get the vectors in Listing 3.3

Listing 3.3 Cosine similarity of term frequency vectors based upon raw term counts

```
doc1_tf_vector = [1, 1] #[apple:1, juice:1]
doc2_tf_vector = [3, 4] #[apple:3, juice:4]

query_vector = [1, 1] #[apple:1, juice:1]

doc1_score = num2str(cos_sim(query_vector, doc1_tf_vector))
doc2_score = num2str(cos_sim(query_vector, doc2_tf_vector))

print("Relevance Scores:\n doc1: " + str(doc1_score) + "\n doc2: " +
➥str(doc2_score))
```

Results:

```
Relevance Scores:
 doc1: 1.0
 doc2: 0.9899
```

As you can see, `doc1` is considered a better cosine similarity match than `doc2`. This is because the terms `apple` and `juice` both occur "the same proportion of times" (one occurrence of each term for every occurrence of the other term) in both the query and in `doc1`, making them the most textually similar. Even though `doc2` is intuitively more "about" the query, mentioning the terms in the query significantly more. Since our goal is for documents like `doc2` with higher term frequency to score higher, we can overcome these by either:

1. Continuing to use cosine similarity, but modifying the query features to actually represent the "best" possible score for each query term, or
2. Switching from cosine similarity to another scoring function that increases as feature weights continue to increase.

Let's try option 1 for now (we'll visit option 2 in section 3.2).

SIDEBAR **Phrase matching and other relevance tricks**

By now, you may be wondering why we keep treating `apple` and `juice` as independent terms and why we don't just treat `apple juice` as a phrase to boost documents higher that match the exact phrase. If so, your intuition is great, and this is one of many easy relevance tuning trick we'll learn later in the chapter. For now, though, we'll keep our query processing simple and just deal with individual keywords in order to stay focused on our main goal - explaining vector-based relevance scoring and text-based keyword scoring features.

In Listing 3.4, we adjust the feature weights in the query vector to be based upon the "best" possible match for each term.

Listing 3.4 Cosine similarity of term frequency vectors, with query weights per term normalized for best match.

```
doc1_tf_vector = [1, 1] #[apple:1, juice:1]
doc2_tf_vector = [3, 4] #[apple:3, juice:4]

query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector])
➥#[apple:3, juice:4]   ❶

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " +
➥num2str(doc2_score))
```

❶ Query vector should represent the "best possible" match, so we include the top possible score for each term in the query vector.

Results:

```
Relevance Scores:
 doc1: 0.9899
 doc2: 1.0
```

As you can see, `doc2` now yields a higher cosine similarity with the query than `doc1`, an improvement that aligns better with our intuition.

While using the term frequency as the feature weight in our vectors certainly helps, textual queries exhibit additional challenges that also need to be considered. Thusfar, our documents have all contained every term from our queries, which does not match with most real-world scenarios. The following example will better demonstrate some of the limitations still present when using only term-frequency-based weighting for our text-based sparse vector similarity scoring. Let's start with the following three text documents:

Document 1:

```
In light of the big reveal in the interview, the interesting thing is that
➥the person in the wrong probably made the right decision in the end.
```

Document 2:

```
My favorite book is the cat in the hat, which is about a crazy cat in a hat
➥who breaks into a house and creates a crazy afternoon for two kids.
```

Document 3:

```
My careless neighbors apparently let a stray cat stay in their garage
➥unsupervised, which resulted in my favorite hat that I let them borrow
➥being ruined.
```

Let’s now map these documents into their corresponding (sparse) vector representations and calculate a similarity score. Listing 3.5 demonstrates a code example for ranking text similarity based upon term frequencies.

Listing 3.5 Ranking text similarity based upon Term Frequency.

```
doc1 = "In light of the big reveal in the interview, the interesting thing
➥is that the person in the wrong probably made the right decision in
➥the end."
doc2 = "My favorite book is the cat in the hat, which is about a crazy cat
➥in a hat who breaks into a house and creates a crazy afternoon for two
➥kids."
doc3 = "My careless neighbors apparently let a stray cat stay in their
➥garage unsupervised, which resulted in my favorite hat that I let them
➥borrow being ruined."

def tf(content, term):
    tokenized_content = tokenize(content)
    term_count = tokenized_content.count(term.lower())
    return float(term_count)

doc1_tf_vector = [ tf(doc1,"the"), tf(doc1,"cat"), tf(doc1,"in"),
➥tf(doc1,"the"), tf(doc1,"hat") ]
doc2_tf_vector = [ tf(doc2,"the"), tf(doc2,"cat"), tf(doc2,"in"),
➥tf(doc2,"the"), tf(doc2,"hat") ]
doc3_tf_vector = [ tf(doc3,"the"), tf(doc3,"cat"), tf(doc3,"in"),
➥tf(doc3,"the"), tf(doc3,"hat") ]

print ("labels: [the, cat, in, the, hat]")
print ("doc1_vector: [" + ", ".join(map(num2str,doc1_tf_vector)) + "]")
print ("doc2_vector: [" + ", ".join(map(num2str,doc2_tf_vector)) + "]")
print ("doc3_vector: [" + ", ".join(map(num2str,doc3_tf_vector)) + "]\n")

query = "the cat in the hat"
query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector,
➥doc3_tf_vector])
print ("query_vector: [" + ", ".join(map(num2str,query_vector)) + "]\n")

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)
doc3_score = cos_sim(query_vector, doc3_tf_vector)

print("Relevance Scores:\n doc1: " + num2str(doc1_score) + "\n doc2: " +
➥num2str(doc2_score)+ "\n doc3: " + num2str(doc3_score))
```

Results:

```
labels: [the, cat, in, the, hat]
doc1_vector: [6.0, 0.0, 4.0, 6.0, 0.0]
doc2_vector: [2.0, 2.0, 2.0, 2.0, 2.0]
doc3_vector: [0.0, 1.0, 2.0, 0.0, 1.0]

query_vector: [6.0, 2.0, 4.0, 6.0, 2.0]

Relevance Scores:
 doc1: 0.9574
 doc2: 0.9129
 doc3: 0.5
```

While we at least receive different relevance scores now for each document based upon the number of times each term matches, the ordering of the results doesn't necessarily match our intuition about which documents are the best matches.

Intuitively, we would instead expect the following ordering:

1. **doc2**: because it is actually about the book *The Cat in the Hat*
2. **doc3**: because it matches all of the words `the`, `cat`, `in`, and `hat`
3. **doc1**: because it only matches the words `the` and `in`, even though it contains them many times

The problem here, of course, is that since every occurrence of any word is considered just as important, the more times ANY term appears, the more relevant that document becomes. In this case, `doc1` is getting the highest score, because it contains 16 total term matches (the first `the` six times, `in` four times, and the second `the` six times), yielding more total term matches than any other document.

It doesn't really make sense that a document containing a word 16 times should actually be considered 16-times as relevant, though. Usually real-world TF calculations dampen the effect of each additional occurrence of a word by calculating TF as the square root of the number of occurrences of each term. Additionally, term frequency is often also normalized relative to document length by dividing that dampened TF by the total number of terms in each document. Since longer documents are naturally more likely to contain any given term and to contain terms more often, this helps ensure that the score is normalized to the document length so that shorter and longer documents are treated equally. This final, normalized TF calculation can be seen in Figure 3.3.

$$TF(t \in d) = \frac{\sqrt{d.count(t)}}{d.totalTerms}$$

Figure 3.3 Term Frequency Calculation. `t` represents a term and `d` represents a document. TF equals the square root of the number of times the term appears in the current document, divided by the number of terms in the document. The numerator dampens the additional relevance contribution of each additional occurrence of a term, while the denominator normalized that dampened frequency to the document length so that longer documents with more terms are comparable to shorter documents with less terms.

Going forward, we'll use this dampened TF calculation to ensure additional occurrences of the same term continue to improve relevance, but without having an outsized impact on the overall score, as it is generally better to match multiple different terms from a query than simply the same terms over and over.

With this improved TF calculation now in place, let's re-calculate our relevance ranking to see if there is any improvement in Listing 3.6.

Listing 3.6 Ranking text similarity based upon Term Frequency.

```
def tf(content, term):
    tokenized_content = tokenize(content)
    term_count = tokenized_content.count(term.lower())
    vector_length = len(tokenized_content)
    return float(np.sqrt(term_count)) / float(vector_length)

doc1_tf_vector = [ tf(doc1,"the"), tf(doc1,"cat"), tf(doc1,"in"),
➥tf(doc1,"the"), tf(doc1,"hat") ]
doc2_tf_vector = [ tf(doc2,"the"), tf(doc2,"cat"), tf(doc2,"in"),
➥tf(doc2,"the"), tf(doc2,"hat") ]
doc3_tf_vector = [ tf(doc3,"the"), tf(doc3,"cat"), tf(doc3,"in"),
➥tf(doc3,"the"), tf(doc3,"hat") ]

print ("labels: [the, cat, in, the, hat]")
print ("doc1_vector: [" + ", ".join(map(num2str,doc1_tf_vector)) + "]")
print ("doc2_vector: [" + ", ".join(map(num2str,doc2_tf_vector)) + "]")
print ("doc3_vector: [" + ", ".join(map(num2str,doc3_tf_vector)) + "]\n")

query = "the cat in the hat"
query_vector = np.maximum.reduce([doc1_tf_vector, doc2_tf_vector,
➥doc3_tf_vector])
print ("query_vector: [" + ", ".join(map(num2str,query_vector)) + "]\n")

doc1_score = cos_sim(query_vector, doc1_tf_vector)
doc2_score = cos_sim(query_vector, doc2_tf_vector)
doc3_score = cos_sim(query_vector, doc3_tf_vector)

print("Relevance Scores:\n doc2: " + num2str(doc2_score) + "\n doc1: "
                                   + num2str(doc2_score)+ "\n doc3: "
                                   + num2str(doc3_score))
```

Results:

```
labels: [the, cat, in, the, hat]
doc1_vector: [0.0942, 0.0, 0.0769, 0.0942, 0.0]
doc2_vector: [0.0456, 0.0456, 0.0456, 0.0456, 0.0456]
doc3_vector: [0.0, 0.0385, 0.0544, 0.0, 0.0385]

query_vector: [0.0942, 0.0456, 0.0769, 0.0942, 0.0456]

Relevance Scores:
 doc2: 0.9222
 doc1: 0.9559
 doc3: 0.5995
```

The normalized TF clearly helped, as `doc2` is now ranked the highest, as we would expect. This is mostly because of the dampening effect on number of term occurrences in `doc1` (which matched `the` and `in` so many times), such that each additional occurrrence contributes less to the feature weight than prior occurrences. Unfortunately, `doc1` is still ranked second highest, so even the improved `tf` function wasn't enough to get the better matching `doc3` to the top.

Your intuition is probably also screaming right now, "Yeah, but nobody really cares about the words `the` and `in`. It's obvious that the words `cat` and `hat` should be given the most weight here instead!" And you would be right. Let's modify our scoring calculation to fix this oversight by introducing a new variable that takes the importance of each term into consideration.

3.1.5 Inverse Document Frequency (IDF): measuring the importance of a term in the query

While Term Frequency has proven useful at measuring how well a document matches each term in a query, it unfortunately does little to differentiate between the importance of the terms in the query. In this section, we'll introduce a technique leveraging the significance of specific keywords based upon their frequency of occurrence across documents.

Document Frequency (DF) for a term is defined as the total number of documents in the search engine that contain the term, and it serves as a good measure for how important a term is. The intuition here is that more specific or rare words (like `cat` and `hat`) tend to be more important than common words (like `the` and `in`). The function used to calculate document frequency is shown in Figure 3.4.

$$DF(t) = \sum_{d=1}^{|D|} if t \in D_i : 1; if t \notin D_i : 0$$

Figure 3.4 Document Frequency Calculation. D is the set of all documents, and `t` is the input term. `DF` is simply the number of documents containing the input term, and the lower the number, the more specific and important the term is when seen in queries.

Since we would like words which are more important to get a higher score, we take an inverse of the document frequency (IDF), typically defined through function in Figure 3.5.

$$IDF(t) = 1 + \log\left(\frac{|D| + 1}{DF(t) + 1}\right)$$

Figure 3.5 Inverse Document Frequency. |D| is the total count of all documents, t is the term, and DF(t) is the count of all documents containing the term. The lower the number, the more insignificant a term, and the higher, the more a term in a query should count toward the relevance score.

Carrrying forward our `the cat in the hat` example from the last section, a vector of IDFs would thus look as shown in Listing 3.7.

Listing 3.7 Calculating Inverse Document Frequency (IDF)

```
df_map = {"the": 9500, "cat": 100, "in":9000, "hat":50}
totalDocs = 10000   ❶

def idf(term):
    return 1 + np.log(totalDocs / (df_map[term] + 1) )   ❷

idf_vector = np.array([idf("the"), idf("cat"), idf("in"), idf("the"),
➥idf("hat")])   ❸

print ("labels: [the, cat, in, the, hat]\nidf_vector: " +
➥vec2str(idf_vector))
```

❶ Simulating that we have a representative sample of docs with meaningful real-world statistics

❷ The IDF function, which dictates the importance of a term in the query

❸ IDF is term-dependent, not document dependent, so it is the same for both queries and documents

Results:

```
labels: [the, cat, in, the, hat]
idf_vector: [1.0512, 5.5952, 1.1052, 1.0512, 6.2785]
```

These results look encouraging. The terms are all now ranked based upon their relative descriptiveness or significance/importance to the query:

1. `hat: 6.2785,`
2. `cat: 5.5952,`
3. `in: 1.1052,`
4. `the: 1.0512`

With a way to differentiate which documents are better matches for specific terms (TF) and a way to determine which specific terms should matter the most in any given query (IDF), we can now combine these two features together to generate a much more balanced relevance-ranking feature called TF-IDF.

3.1.6 TF-IDF: a balanced weighting metric for text-based relevance

We now have the two principle components of text-based relevance ranking:

- TF (measures how well a term describes a document)
- IDF (measures how important each term is)

Most search engines, and many other data science applications, leverage a combination of each of these factors as the basis for textual similarity scoring, using a variation of the function in Figure 3.6.

Listing 3.8 TF-IDF score. Combines both the term frequency and inverse document frequency calculations together into a balanced text-ranking similarity score.

```
"TF-IDF" = TF * IDF^2
```

With this improved feature-weighting function in place, we can finally calculate a balanced relevance score (that weights both number of occurrences and usefulness of terms) for how well each of our documents match our query, as shown in Listing 3.8.

Listing 3.9 TF-IDF Ranking Code for the query `the cat in the hat`

```
def tf_idf(tf,idf):
    return tf * idf**2

query = "the cat in the hat"

print ("labels: [the, cat, in, the, hat]")
doc1_tfidf = [
               tf_idf(tf(doc1, "the"), idf("the")),
               tf_idf(tf(doc1, "cat"), idf("cat")),
               tf_idf(tf(doc1, "in"), idf("in")),
               tf_idf(tf(doc1, "the"), idf("the")),
               tf_idf(tf(doc1, "hat"), idf("hat"))
             ]
print("doc1_tfidf: " + vec2str(doc1_tfidf))

doc2_tfidf = [
               tf_idf(tf(doc2, "the"), idf("the")),
               tf_idf(tf(doc2, "cat"), idf("cat")),
               tf_idf(tf(doc2, "in"), idf("in")),
               tf_idf(tf(doc2, "the"), idf("the")),
               tf_idf(tf(doc2, "hat"), idf("hat"))
             ]
print("doc2_tfidf: " + vec2str(doc2_tfidf))

doc3_tfidf = [
               tf_idf(tf(doc3, "the"), idf("the")),
               tf_idf(tf(doc3, "cat"), idf("cat")),
               tf_idf(tf(doc3, "in"), idf("in")),
               tf_idf(tf(doc3, "the"), idf("the")),
               tf_idf(tf(doc3, "hat"), idf("hat"))
             ]
print("doc3_tfidf: " + vec2str(doc3_tfidf))

query_tfidf = np.maximum.reduce([doc1_tfidf, doc2_tfidf, doc3_tfidf])

doc1_relevance = cos_sim(query_tfidf,doc1_tfidf)
doc2_relevance = cos_sim(query_tfidf,doc2_tfidf)
doc3_relevance = cos_sim(query_tfidf,doc3_tfidf)

print("\nRelevance Scores:\n doc2: " + num2str(doc2_relevance)
                       + "\n doc3: " + num2str(doc3_relevance)
                       + "\n doc1: " + num2str(doc1_relevance))
```

Results:

```
labels: [the, cat, in, the, hat]
doc1_tfidf: [0.1041, 0.0, 0.094, 0.1041, 0.0]
doc2_tfidf: [0.0504, 1.4282, 0.0557, 0.0504, 1.7983]
doc3_tfidf: [0.0, 1.2041, 0.0664, 0.0, 1.5161]

Relevance Scores:
 doc2: 0.9993
 doc3: 0.9979
 doc1: 0.0758
```

Finally our search results make intuitive sense! `doc2` gets the highest score, since it matches the most important words the most, followed by `doc3`, which contains all the words, but not as many times, followed by `doc1`, which only contains an abundance of insignificant words.

SIDEBAR **Cosine similarity vs. TF-IDF matching score**

You may have noticed that for our query vectors, we've been using the "maximum" possible score for each feature, instead of calculating a TF-IDF value based upon the query. The reason for this is that we want documents which are better matches (i.e. more matches of each term) for the query to get higher cosine similarity scores, not documents which "contain the query words roughly the same number of times", which would occur since every word in the query occurs only once and cosine similarity only cares about the angle of vectors and not the magnitude. In practice, instead of normalizing the query to the maximum values from all documents like we have (which is challenging at scale), search engines will typically just sum the calculated values for all the feature weights to arrive at a final relevance score. This closely approximates a cosine similarity that is normalized for "best match" on each term, and it is a much easier calculation than trying to perform an actual cosine similarity calculation. The BM25 relevance calculation, which we'll introduce in the next section, will flip to this optimized method of calculating similarity.

This TF-IDF calculation is at the heart of many search engine relevance calculations, including the default similarity algorithm in Apache Lucene-based search engines - called BM25 - which we will introduce in the next section.

3.2 Controlling the relevance calculation

In the last section, we showed how queries and documents can be represented as vectors, how cosine similarity can be used as a relevance function to compare queries and documents, and how TF-IDF ranking can be used to create a feature weight that balances both the strength of occurrence (TF) and significance of a term (IDF) for each term in a term-based vector.

In this section, we'll show how a full relevance function can be specified and controlled in a search engine (Apache Solr), including common query capabilities, modeling queries as functions, ranking vs. filtering, and applying different kinds of boosting techniques.

In section 3.1, we used a cosine similarity calculation to determine the relevance ranking of documents relative to queries, ultimately arriving at TF-IDF as our balanced keyword-weighting metric. We discovered from Listing 3.3, however, that we can't simply use a value of `1` as the term frequency of each of our terms in the query vector, because then documents are largely ranked based upon how well they maintain a similar proportion of keywords. This is a side effect of using the cosine similarity, which only looks at the difference in *angle* between two vectors and ignores the magnitude of the distance. Specifically, for the query of `apple juice`, if the query vector was `[1 1]` then documents containing `[2 2]`, `[3 3]` … `[N N]` would all have a cosine of `1.0` because they all have the keywords in the same relative proportion as the query, even though a document containing more occurrences like `[3 4]` intuitively seems more relevant than a document only containing `apple` and `juice` once (`[1 1]`) or twice (`[2 2]`).

To fix this, search engines typically utilize different scoring functions than pure cosine similarity for text-based ranking. In our previous examples from Listing 3.4 to Listing 3.8, we modified the weights for each term in our query vector to represent the "best possible" TF ranking for each term in any document, such that an "ideal" document would be one containing the top possible score for each keyword. This approach works well for our simple examples, but doesn’t scale well in a larger production environment, so search engines instead typically take approaches which can iteratively calculate relevance one document at a time. One option is to use something like a *dot product*, which is a cosine calculation that is multiplied by the magnitude of each dimension in the vector. We won’t cover that technique here, but will instead dive straight into the more common technique that is used by default in most search engines - to calculate a relevance score per keyword and then simply sum up the weights for each keyword.

Let’s start by showing off the default similarity calculation (which uses this summing up approach) leveraged by all Lucene-based search engines: BM25.

3.2.1 BM25: Lucene’s default text-similarity algorithm

BM25 is the name of the default similarity algorithm in Apache Lucene, Apache Solr, Elasticsearch, Lucidworks Fusion, and other Lucene-based search engines. BM25 (short for Okapi "Best Matching" version 25) was first published in 1994, and it demonstrates improvements over standard TF-IDF cosine similarity ranking in many real-world, text-based ranking evaluations.

BM25 still uses TF-IDF at its core, but it also includes several other parameters which make it easier to control things like term frequency saturation point and document length normalization, and it sums up the weights for each matched keyword instead of calculating a cosine. The full BM25 calculation is shown in Figure 3.7.

$$Score(q,d) = \sum (t \in q) \frac{idf(t) \cdot (tf(t \in d) \cdot (k+1))}{tf(t \in d) + k \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

Where:

t = term; **d** = document; **q** = query; **i** = index

tf(t in d) = numTermOccurrencesInDocument $^{1/2}$

idf(t) = 1 + log (numDocs / (docFreq + 1))

$|d| = \sum_{t \text{ in } d} 1$

avgdl = $\left(\sum_{d \text{ in } i} |d|\right) / \left(\sum_{d \text{ in } i} 1\right)$

k = Free parameter. Usually ~1.2 to 2.0. Increases term frequency saturation point.

b = Free parameter. Usually ~0.75. Increases impact of document normalization.

Figure 3.6 BM25 Scoring Function. It still leverages TF and IDF prominently, but provides more control over how much each additional occurrence of a term contributes to the score (the k parameter), and how much scores are normalized based upon document length the b parameter).

Instead of reimplementing all of this math in Python to explain it, let's now switch over to using our search engine and see how it performs the calculation. Let's start by creating a collection in Solr (Listing 3.9) and adding some documents (using our previous `the cat in the hat` example), as shown in Listing 3.10.

Listing 3.10 Creating a collection. A collection contains a specific schema and configuration for holding a group of documents, and is the unit upon which we will add documents, search, rank, and retrieve search results.

```
import sys
sys.path.append('..')
from aips import *

collection = "cat_in_the_hat"
create_collection(collection)

#Ensure the fields we need are available
upsert_text_field(collection, "title")
upsert_text_field(collection, "description")
```

Response:

```
Wiping 'cat_in_the_hat' collection
Status: Success

Creating cat_in_the_hat' collection
Status: Success

Adding 'title' field to collection
Status: Success

Adding 'description' field to collection
Status: Success
```

Listing 3.11 Adding documents to a collection

```
docs = [
    {
        "id": "doc1",
        "title": "Worst",
        "description": "The interesting thing is that the person in the
        ➥wrong made the right decision in the end."
    },
    {
        "id": "doc2",
        "title": "Best",
        "description": "My favorite book is the cat in the hat, which is
        ➥about a crazy cat who breaks into a house and creates a crazy
        ➥afternoon for two kids."

    },
    {
        "id": "doc3",
        "title": "Okay",
        "description": "My neighbors let the stray cat stay in their garage,
        ➥which resulted in my favorite hat that I let them borrow being
        ➥ruined."
    }
]
print("\nAdding Documents to '" + collection + "' collection")
response = requests.post(solr_url + collection + "/update?commit=true",
➥json=docs).json()
print("Status: " "Success" if response["responseHeader"]["status"] == 0
➥else "Failure" )
```

Response:

```
Adding Documents to 'cat_in_the_hat' collection
Status: Success
```

With our documents added to the search engine, we can now issue our query and see the full BM25 scores. Listing 3.11 demonstrates how to run our search for the query `the cat in the hat` and to request the detailed relevance calculation back with each document.

Listing 3.12 Ranking by and inspecting the BM25 similarity score

```
query = "the cat in the hat"
request = {
  "query": query,
  "fields": ["id", "title", "description", "score", "[explain style=html]"],
  "params": {
    "qf": "description",
    "defType": "edismax",
    "indent": "true"
  }
}

from IPython.core.display import display,HTML
display(HTML(
    "<br/><strong>Query: </strong><i>" + query
    + "</i><br/><br/><strong>Ranked Docs:</strong>"))

response = str(requests.post(solr_url + collection + "/select",
➥json=request).json()
        ["response"]["docs"]).replace('\\n', '').replace(", '", ",<br/>'")
display(HTML(response))
```

Response:

```
Query: the cat in the hat

Ranked Docs:

[{'id': 'doc2',
'title': ['Best'],
'description': ['My favorite book is the cat in the hat, which is about a
➥crazy cat who breaks into a house and creates a crazy afternoon for
➥two kids.'],
'score': 0.6823196,
'[explain]': '
    0.6823196 = sum of:
        0.15655403 = weight(description:the in 1) [SchemaSimilarity], result of:
            0.15655403 = score(freq=2.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.58620685 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.19487953 = weight(description:hat in 1) [SchemaSimilarity], result of:
            0.19487953 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.4146341 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.27551934 = weight(description:cat in 1) [SchemaSimilarity], result of:
            0.27551934 = score(freq=2.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.58620685 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.05536667 = weight(description:in in 1) [SchemaSimilarity], result of:
            0.05536667 = score(freq=1.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.4146341 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    28.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}, {'id': 'doc3',
'title': ['Okay'],
'description': ['My neighbors let the stray cat stay in their garage,
➥which resulted in my favorite hat that I let them borrow being ruined.'],
'score': 0.62850046,
'[explain]': '
    0.62850046 = sum of:
        0.120666236 = weight(description:the in 2) [SchemaSimilarity], result of:
            0.120666236 = score(freq=1.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.21236044 = weight(description:hat in 2) [SchemaSimilarity], result of:
            0.21236044 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.21236044 = weight(description:cat in 2) [SchemaSimilarity], result of:
            0.21236044 = score(freq=1.0), product of:
                0.47000363 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    2 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.45182723 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    1.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.08311336 = weight(description:in in 2) [SchemaSimilarity], result of:
            0.08311336 = score(freq=2.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.6224256 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    23.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}, {'id': 'doc1',
'title': ['Worst'],
'description': ['The interesting thing is that the person in the wrong made
➥the right decision in the end.'],
'score': 0.3132525,
'[explain]': '
    0.3132525 = sum of:
        0.2234835 = weight(description:the in 0) [SchemaSimilarity], result of:
            0.2234835 = score(freq=5.0), product of:
                2.0 = boost
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.83682007 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    5.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    17.0 = dl, length of field
                    22.666666 = avgdl, average length of field
        0.089769006 = weight(description:in in 0) [SchemaSimilarity], result of:
            0.089769006 = score(freq=2.0), product of:
                0.13353139 = idf, computed as log(1 + (N - n + 0.5) / (
          ➥n + 0.5)) from:
                    3 = n, number of documents containing term
                    3 = N, total number of documents with field
                0.6722689 = tf, computed as freq / (freq + k1 * (
          ➥1 - b + b * dl / avgdl)) from:
                    2.0 = freq, occurrences of term within document
                    1.2 = k1, term saturation parameter
                    0.75 = b, length normalization parameter
                    17.0 = dl, length of field
                    22.666666 = avgdl, average length of field
'}]
```

While the BM25 calculation is more complex than the TF-IDF feature weight calculations we saw in the last section, it is fundamentally still leverages TF-IDF at its core. Therefore it should be no suprise that the ranked search results actually return in the same relative order as our TF-IDF calculations from the Listing 3.8:

```
Ranked Results (Listing 3.8: TF-IDF Cosine Similarity)
 doc2: 0.998
 doc3: 0.9907
 doc1: 0.0809

Ranked Results (Listing 3.9: BM25 Similarity)
 doc2: 0.6878265
 doc3: 0.62850046
 doc1: 0.3132525
```

Our query for `the cat in the hat` can still very much be thought of as a vector of the BM25 scores for each of the terms: ["the", "cat", "in", "the", "hat"].

What may not be obvious, however, is that the feature weights for each of these terms are actually just overridable functions. Instead of thinking of our query as simply a bunch of keywords, we can think of our query as a mathematical function composed of other functions, where some of those functions take keywords as inputs and return numerical values (scores) back to be used in the relevance calculation. For example, our query could alternatively be expressed as the vector:

```
[ query("the"), query("cat"), query("in"), query("the"), query("hat") ]
```

In Solr query syntax, this would be:

```
q={!func}query("the") {!func}query("cat") {!func}query("in")
➥{!func}query("the") {!func}query("hat")
```

If we execute this "functionized" version of the query, we will get the exact same relevance score as if we had just executed the query directly. Listing 3.12 shows the code to perform this version of the query.

Listing 3.13 Text Similarity using the Query Function

```
query = '{!func}query("the") {!func}query("cat") {!func}query("in")
➥{!func}query("the") {!func}query("hat")'
request = {
    "query": query,
    "fields": ["id", "title", "score"],
    "params": {
      "qf": "description",
      "defType": "edismax",
      "indent": "true"
    }
}
display(HTML("<strong>Query</strong>: <i>" + query + "</i><br/><br/>
➥<strong>Results:</strong>"))
response = str(requests.post(solr_url + collection + "/select",
➥json=request).json()["response"]["docs"]).replace('\\n', '').replace(
➥", ", ",<br/>'")
display(HTML(response))
```

Response:

```
Query: {!func}query("the") {!func}query("cat") {!func}query("in")
➥{!func}query("the") {!func}query("hat")

Results:
[{'id': 'doc2',
''title': ['Best'],
''score': 0.6823196},
'{'id': 'doc3',
''title': ['Okay'],
''score': 0.62850046},
'{'id': 'doc1',
''title': ['Worst'],
''score': 0.3132525}]
```

As expected, the scores are exactly the same as before - we’ve simply substituted in explicit functions where implicit functions were previously assumed. Once we realize that every term in a query to the search engine is actually just a configurable scoring function, it opens up tremendous possibilities for manipulating that scoring function!

3.2.2 Functions, functions, everywhere!

Now that we’ve seen that the relevance score for each term in our queries is simply a function operating on that term to generate a feature weight, the next logical question is "what *other* kinds of functions can I use in my queries?".

We’ve already encountered the `query` function (at the end of section 3.2.1), which is effectively the default calculation that executes whenever no explicit function is just specified, and which uses the BM25 similarity algorithm by default.

But what if we want to consider some other features in our scoring calculation, perhaps some that are not text-based?

Here is a partial list of common relevance techniques:

- *Geospatial Boosting*: Documents near the user running the query should rank higher.
- *Date Boosting*: Newer documents should get a higher relevancy boost
- *Popularity Boosting*: Documents which are more popular should get a higher relevancy boost.
- *Field Boosting*: Terms matching in certain fields should get a higher weight than in other fields
- *Category Boosting*: Documents in categories related to query terms should get a higher relevancy boost.
- *Phrase Boosting*: Documents matching multi-term phrases in the query should rank higher than those only matching the words separately.
- *Semantic Expansion*: Documents containing other words or concepts that are highly related to the query keywords and context should be boosted.

Many of these techniques are built into specific query parsers in Solr, either through query syntax or through query parser options. For example, field boosting can be accomplished through the `qf` parameter on the `edismax` query parser. The following query, for example, provides a 10X relevancy boost for matches in the title field, and a 2.5X relevancy boost for matches in the `description` field.

```
q={!type=edismax qf="title^10 description^2.5"}the cat in the hat
```

Boosting on full phrase matching, on two-word phrases, and on three-word phrases is also a native feature of the edismax query parser:

- Boost docs containing the exact phrase `"the cat in the hat"` in the `title` field:

```
q={!type=edismax qf="title description" pf=title}the cat in the hat
```

- Boost docs containing the two-word phrases `"the cat"`, `"cat in"`, `"in the"`, or `"the hat"` in the `title` or `description` field:

```
q={!type=edismax qf="title description" pf2="title description"}the
➥cat in the hat
```

- Boost docs containing the three-word phrases `"the cat in"` or `"in the hat"` in the `description` field:

```
q={!type=edismax qf="title description" pf3=description}the cat in the hat
```

Many of the relevancy boosting techniques will require constructing your own features leveraging function queries, however. For example, if we wanted to create a query that did nothing more than boost the relevance ranking of documents geographically closest to the user running the search (relevance based on distance away), we could issue the following query:

```
q=*:*&
 sort=geodist(location, $user_latitude, $user_longitude) asc&
 user_latitude=33.748&
 user_longitude=-84.39
```

That last query is using the `sort` parameter to strictly order documents by the calculated value from the `geodist` function. This works great if we want to order results by a single feature, but what if we want to construct a more nuanced sort based upon multiple features? To accomplish this, we would just update our query to include each of these function in each document's relevance calculation, and then sort by the relevance score:

```
q={!func}scale(query($keywords),0,25)
  {!func}recip(geodist($lat_long_field,$user_latitude,$user_longitude),
  ➥1,25,1)
  {!func}recip(ms(NOW/HOUR,modify_date),3.16e-11,25,1)
  {!func}scale(popularity,0,25)
  &keywords="basketball"&
  lat_long_field=location&
  user_latitude=33.748&
  user_longitude=-84.391
```

That query does a few interesting things:

- It constructs a query vector containing four features: BM25 Keywords relevance score (higher is better), geo distance (lower is better), publication date (newer is better), and popularity (higher is better).
- Each of the feature values is scaled between `0` and `25` so that they are all comparable, with the best keyword/geo/publication date/popularity score getting a score of `25`, and the worst getting a score close to `0`.
- Thus a "perfect score" would add up to `100` (25 + 25 + 25 + 25), and the worst score would be approximately `0`.
- Since the relative contribution of `25` is specified as part of the query for each function, we can easily change the weights of any feature on the fly to give preference to certain features in the final relevance calculation.

With the last query, we have now fully taken the relevance calculation into our own hands by modeling our relevance features and giving them weights. While this is very powerful, it still requires significant manual effort and testing to figure out which features matter for a given domain, and what their relative weights should be. In chapter 10 we will walk through building Machine-learned Ranking models to automatically make those decisions for us (a process known as "Learning to Rank"). For now, however, our goal was to ensure you understood the mechanics of modeling features in query vectors, and controlling their weights.

SIDEBAR **Deeper dives on function queries**

If you'd like to learn more about how to utilize function queries, we recommend reviewing chapter 15 of Solr in Action (Trey Grainger and Timothy Potter, Manning, 2014) on "Complex Query Operations", for a much fuller exposition. For a full list of available function queries in Solr, you can also check out the documentation in the function query section of the Solr Reference Guide.

While we've seen the power of utilizing functions as features in our queries, thusfar all of our examples have been what are called "additive" boosts, where the sum of the values of each function calculation comprise the final relevance score. It is also frequently useful to combine functions in a fuzzier, more flexible way through "multiplicative" boosts, which we'll cover in the next section.

3.2.3 Choosing multiplicative vs. additive boosting for relevance functions

One last topic to address concerning how we control our relevance functions is the idea of using multiplicative vs. additive boosting of relevance features.

In all of our examples to this point, we have added multiple features together into our query vector to contribute to the score. For example, the following queries will all yield equivalent relevance calculations assuming they are all filtered down to the same result set (i.e. `fq=the cat in the hat`):

```
Text query (score + filter)
  q=the cat in the hat

Function Query (score only, no filter)
  q={!func}query("the cat in the hat")

Multiple Function Queries (score only, no filter)
  q={!func}query("the")
    {!func}query("cat")
    {!func}query("in")
    {!func}query("the")
    {!func}query("hat")

Boost Query (score only, no filter)
  q=*:*&bq=the cat in the hat
```

The kind of relevance boosting in each of these examples is known as *additive boosting*, and maps well to our concept of a query as nothing more than a vector of features that needs to have its similarity compared across documents.

In many cases, however, we are likely to want to specify what are known as *multiplicative boosts* as part of our relevance calculations. Instead of inserting additional features into our vector, multiplicative boosts increase the relevance of an entire document by multiplying the document's full score by some calculated value.

For example, if we wanted to query for `the cat in the hat`, but wanted the popularity of documents (those with a higher number in the `popularity` field) to have a less constrained effect, we can’t easily do this by just adding another feature into our query vector - at least not without modifying the weights of all the other features, plus any additional normalization that may be applied by the BM25 ranking function. If we wanted to apply multiple boosts like this (for example, boosting both on popularity AND on publication date), then the option of modeling this as an additive boost becomes unreasonably complex and harder to control.

In section 3.2.2, we were able to successfully utilize additive boosting by explicitly constraining the minimum and maximum values for the features in our query vector so that each feature provides a known contribution to the overall relevance function.

Multiplicative boosting enables boosts to "pile up" on each other, however, because each of the boosts is multiplied against the overall relevance score for the document, resulting in a much fuzzier match and preventing the need for the kind of tight constraints we had to supply for our additive boost example.

To supply a multiplicative boost, you can either use the `boost` query parser (`{!boost}`) in your query vector or, if you are using the `edismax` query parser, the simplified `boost` query param. For example, to multiply a document’s relevance score by ten times the value in the `popularity` field, you would do either:

```
q=the cat in the hat&
  defType=edismax&
  boost=mul(popularity,10)
```

OR

```
`q={!boost b=mul(popularity,10)}the cat in the hat
```

In general, multiplicative boosts enable you greater flexibility to combine different relevance features without having to explicitly pre-define and exact relevance formula accounting for every potential contributing factor. On the other hand, this flexibility can lead to unexpected consequences if the multiplicative boost values for particular features get too high and overshadow other features. In contrast, additive boosts can be a pain to manage, because you have to explicitly scale them so that they can be combined together and maintain a predictible contribution to the overall score, but once you’ve done this you maintain tight control over the relevance scoring calculation and range of scores.

Both additive and multiplicative boosts can be useful in different scenarios, so it’s best to consider the problem at hand and experiment with what gets you the best results. We’ve now covered the major ways to control relevance ranking in the search engine, but matching and filtering of documents can often be just as important, so we'll cover them in the next section.

3.2.4 Differentiating matching (filtering) vs. ranking (scoring) of documents

We opened up this chapter by stating that search engines fundamentally do three things: ingest content, return content matching incoming queries, and sort the returned content based upon some measure of how well it matches the query. Thusfar, we've only talked about the third capability (relevance ranking), however.

We've only really spoken of queries as feature vectors, and we've only discussed relevance ranking as a process of either calculating a cosine similarity or of adding up document scores for each feature (keyword or function) in the query. This may seem a bit strange, since most search books start with coverage of matching keywords in the search engine's inverted index to filter result sets well before discussing relevance.

We've delayed the discussion of filtering results until this point on purpose, however, in order to focus on relevance and the idea of queries and documents as vectors of features to be compared and ranked based upon similarity.

A pre-requisite for comparing queries with documents, of course, is that the search engine has already ingested some content from which those features are derived. Once content is ingested, there are then two steps involved in executing a query:

- *Matching*: Filtering results to a known set of possible answers
- *Ranking*: Ordering all of the possible answers by relevance

In reality, we can often completely skip step 1 (matching/filtering) and still see the exact same results on page one (and for many pages), since the most relevant results should generally rank the highest and thus show up first. If you think back to chapter 2, we even saw some vector scoring calculations (comparing feature vectors for food items - i.e. "apple juice" vs. "donut") where we would have been unable to filter results at all, and we instead had to first score every document to determine which ones to return based upon relevance alone. In this scenario, we didn't even have any keywords or other attributes which could be leveraged as a filter.

So if the initial matching phase is effectively optional, then why do it at all? One obvious answer is that it provides a significant performance optimization. Instead of iterating through every single document and calculating a relevance score, we can greatly speed up both our relevance calculations and the overall response time of our search engine by first filtering the initial result set to a smaller set of documents which are logical matches.

Of course, there are also additional benefits to filtering our results sets, in that the total document count is reduced and we can provide analytics (facets) on the set of logically-matching documents in order to help the user further explore and refine their results set. Finally, there are plenty of scenarios where "having logical matches" should actually be considered among the most important features in the ranking function, and thus simply filtering on logical matches up-front can greatly simplify the relevance calculation. We'll discuss these tradeoffs in the next section.

3.2.5 Logical matching: weighting the relationships between terms in a query

We just mentioned that filtering results before scoring them is primarily a performance optimization and that the first few pages of search results would likely look the same regardless of whether you filter the results or just do relevance ranking.

This only holds true, however, if your relevance function successfully contains features which already appropriately boost better logical matches. For example, consider the difference between expectations for the following queries:

1. `"statue of liberty"`
2. `statue AND of AND liberty`
3. `statue OR of OR liberty`
4. `statue of liberty`

From a logical matching standpoint, the first query will be very precise, only matching documents contain the *exact* phrase `statue of liberty`. The second query will only match documents containing all of the terms `statue`, `of`, and `liberty`, but not necessarily as a phrase. The third query will match any document containing any of the three terms, which means documents *only* containing `of` will match, but that documents containing `statue` and `liberty` should rank much higher due to TF-IDF and the BM25 scoring calculation.

In theory, if phrase boosting is turned on as a feature then documents containing the full phrase should rank highest, followed by documents containing all terms, followed by documents containing any of the words. Assuming that happens, then you should see the same ordering of results regardless of whether you filter them to logical, Boolean matches, or whether you only sort based on a relevance function.

In practice, though, users often consider the logical structure of their queries to be highly relevant to the documents they expect to see, so respecting this logical structure and filtering *before* ranking allows you to remove results which users' queries indicate are safe to remove.

Sometimes the logical structure of user queries is ambiguous, however, such as with our fourth example: the query `statue of liberty`. Does this logically mean `statue AND of AND liberty`, `statue OR of OR liberty`, or something more nuanced like `(statue AND OF) OR (statue AND liberty) OR (of AND liberty)`, which essentially means "match at least two of three terms". Using the "minimum match" (`mm`) parameter in Solr enables you to control these kinds of matching thresholds easily, even on a per-query basis:

- 100% of query terms must match (equivalent to `statue AND of AND liberty`):

```
q=statue of liberty&
 mm=100%
```

- At least one query term + 0% of additional query terms must match (equivalent to `statue OR of OR liberty`):

```
q=statue of liberty&
 mm=0%
```

- At least two query terms must match (equivalent to `(statue AND of) OR (statue AND liberty) OR (of AND liberty)`):

```
q=statue of liberty&
 mm=2
```

The `mm` parameter in Solr allows you to specify a minimum match threshold as either a percentage (0% to 100%) of terms, a number of terms (1 to N terms), or as a step function like `mm=2<-30% 5<3`, which means "All terms are required if there are less than 2 terms, up to 30% of terms can be missing if there are less than 5 terms, and at least 3 terms must exist if there are 5 or more terms. The `mm` parameter works with the `edismax` query parser, which is the primary query parser we will use for text-matching queries in this book. You can consult the edismax section of the Solr Reference Guide for more details on how to fine-tune your logical matching rules with these minimum match capabilities.

When thinking about constructing relevance functions, the idea of filtering and scoring can often get mixed up, particularly since Solr itself mixes concerns in the query parameter. We'll attempt to separate these concerns in the next section.

3.2.6 Separating concerns: filtering vs. scoring

In section 3.2.4 we differentiated between the ideas of "matching" and "ranking". Matching of results is logical and is implemented by filtering search results down to a subset of documents, whereas ranking of results is qualitative, and is implemented by scoring all documents relative to the query and then sorting them by that calculated score. In this section, we'll cover some techniques to provide maximum flexibility in controlling matching and ranking by cleanly separating out the concerns of filtering and scoring.

Solr has two primary ways to control filtering and scoring, the "query" (`q` parameter) and the "filters" (zero or more `fq` parameters). Consider the following request:

```
q=the cat in the hat&
 fq=category:books&
 fq=audience:kid&
 defType=edismax&
 mm=100%&
 qf=description
```

In this query, Solr is being instructed to filter the possible result set down to only documents with a value of `books` in the `category` field and also a value of `kid` in the `audience` field. In addition to those filters, however, the query itself also acts as a filter, so the result set gets further filtered down to only documents also containing (100%) of the values `the`, `cat`, `in`, and `hat` in the `description` field.

The logical difference between the `q` and `fq` parameters is that the `fq` only acts as a filter, whereas the `q` acts as *both* a filter and feature vector for relevance ranking. This dual use of the `q` parameter is helpful default behavior for queries, but mixing the concerns of filtering and scoring in the same parameter can be suboptimal for more advanced queries, especially if we're simply trying to manipulate the relevance calculation and not arbitrarily removing results from our document set.

There are a few ways to address this:

1. Model the `q` parameter as a function (functions only count toward relevance and do not filter):

```
q={!func}query("{!type=edismax qf=description mm=100% v=$query}")&
  fq={!cache=false v=$query}&
  query=the cat in the hat
```

2. Make your query match all documents (no filtering or scoring) and apply a Boost Query (`bq`) parameter to influence relevance without scoring:

```
q=*:*
  &bq={!type=edismax qf=description mm=100% v=$query}&
  fq={!cache=false v=$query}&
  query=the cat in the hat
```

Between these three parameters, `q` both filters and then boosts based upon relevance, `fq` only filters, and `bq` only boosts. As such, both of these approaches are logically equivalent, but we'll go with option 2 throughout this book since it is a bit cleaner to use the dedicated `bq` parameter which was designed to contribute toward the relevance calculation without filtering.

You may have noticed that both versions of the query also contain a filter query which filters on the query:

```
q=*:*
  &bq={!type=edismax qf=description mm=100% v=$query}&
  fq={!cache=false v=$query}&
  query=the cat in the hat
```

Since the `q` parameter intentionally no longer filters our search results, this `fq` parameter is now required if we still want to filter to the user-entered query. By constructing our queries this way, we allow our relevance function to be entirely separated from the filtering logic, which often makes it much easier to construct complex ranking functions. The special `cache=false` parameter there is used to turn off caching of the filter. Caching of filters is turned on by default in Solr since filters tend to be reused often across requests. Since the `$query` parameter is user-entered and wildly variable in this case (not frequently reused across requests), it doesn't make sense to pollute the search engine's caches with these values. If you try to filter on user-entered queries without turning the cache off, it will waste system resources and likely slow down your search engine.

The overarching theme here is that it is possible to cleanly separate logical filtering from ranking features in order to maintain full control and flexibility over your search results. While going through this effort may be overkill for simple text-based ranking, separating out these concerns becomes critical when attempting to build out more sophisticated ranking functions.

Now that you understand the mechanics of how to construct these kinds of purpose-built ranking functions, let's wrap up this chapter with a brief discussion of how to apply these techniques to implement user and domain-specific relevance ranking.

3.3 Implementing user and domain-specific relevance ranking

In section 3.2, we walked through how to easily and dynamically modify the parameters of our query-to-document similarity algorithm, including passing in our own functions as features which contribute to the score, in addition to just text-based relevance ranking.

While text-based relevance ranking using BM25, TF-IDF vector cosine similarity, or some other kind of statistics-based approach on word occurrences can provide decent "general" search relevance out of the box, it can't hold its own against good domain-specific relevance factors. For example, if you travel to Boston, Massachusetts in the United States and you open up a restaurant finding app on your phone and search for `hamburger`, you probably won't be very happy with the search engine if the top answers are for `Five Guys Burgers and Fries in Austin, Texas`, `McDonald's in Anchorage, Alaska`, and `Hungry Jack in Sydney, Australia`. Even if these were the best keyword matches and even if they were the most popular results for the query, the expectation from users is that you will consider "distance from my location" as one of the most (if not *the* most) important of factors in the relevance determination.

In fact, most people would probably intuitively be able to tell you that the following attributes matter the most to them within these various domains:

- Restaurant Search: `geographical proximity`, `user-specific dietary restrictions`, `user-specific taste preferences`, and `price range`
- News Search: `freshness` (date), `popularity`, and `geographical area`
- Ecommerce: `likelihood of conversion` (click-through, add-to-cart, and/or purchase)
- Movie Search: `name match` (title, actor, etc.), `popularity` of document, `release date`, `critic review score`
- Job Search: `job title`, `job level`, `compensation range`, `geographical proximity`, `job industry`
- Web Search: `keyword match on page`, `popularity of page`, `popularity of website`, `location of match on page` (in title, header, body, etc.), `quality of page` (duplicate content, spammy content, etc.), topic match between page and query.

Obviously these are just examples, and you can probably think of many more factors that would matter and impact relevance for each use case. The point, however, is that every search engine and domain has unique features which need to be considered to deliver an optimal search experience.

Instead of walking through how to manually construct queries for each of the scenarios above (hopefully the last section gave you the tools you need to figure that out), this book will instead focus on showing you how to use machine learning to build an AI-powered search engine that can automatically learn how to both generate and weight these kinds of features to determine the optimal relevance algorithm.

This chapter has barely scratched the surface on the myriad ways that you can control and manipulate the matching and ranking functions in order to return the best content and domain-specific relevance-ranked search results. An entire profession exists - called "relevance engineering" - that is dedicated to tuning search relevancy using techniques like this within many organizations. If you'd like to dive deeper, I highly recommend the book *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016), which provides an expert guide on this kind of relevance engineering.

The purpose of this chapter was to give you the base knowledge and tools you'll need in the coming chapters to impact relevance ranking as we begin integrating more automated machine learning techniques into our search applications. We'll begin applying all of this in our next chapter on crowdsourced relevance.

3.4 Summary

- We can map queries and documents to dense or sparse numerical vectors and then assign a relevance rank to documents based upon either a cosine similarity between the vectors or some other similarity scoring calculation
- Leveraging TF-IDF or the BM25 similarity calculations (based upon TF-IDF) for our text similarity scores provides us with a more meaningful measure of feature (keyword) importance in our queries and documents, enabling improved text ranking over just looking at term matches alone.
- Text similarity scoring is just one of many kinds of functions we can invoke as a feature within our queries for relevance ranking. We can inject functions within our queries along with keyword matching and scoring, as each keyword phrase is effectively just a ranking function anyway.
- It is useful to separate "filtering" and "scoring" as separate concerns in order to have better control when specifying our own ranking functions.
- In order to optimize relevance, we need to create domain-specific relevance functions and also leverage user-specific features instead of relying just on keyword matching and ranking. We'll focus on doing this through automated learning approaches throughout the rest of this book.

4 Crowdsourced relevance

This chapter covers

- Harnessing your users' collective insights to improve the relevance of your search platform
- Collecting and working with user behavioral signals
- Leveraging Reflected Intelligence to create self-tuning models like signals boosting, collaborative recommendations and personalization, and machine-learned ranking
- Building an end-to-end signals boosting model
- Crowdsourced learning from content-based signals

In chapter one, we introduced the dimensions of user intent as "content understanding", "user understanding", and "domain understanding". In order to create an optimal AI-powered search platform, we need to be able to combine each of these contexts to understand our users' query intent. The question, though, is how do we derive these understandings?

Many different sources of information may exist from which we can learn documents, databases, internal knowledge graphs, user behavior, domain experts, and so on. Some organizations have teams that manually tag documents with topics or categories, and some even outsource these tasks using tools like Amazon Mechanical Turk, which allows them to crowdsource answers from people all around the world. For identifying malicious behavior or errors on websites, companies often allow their users to report problems and even suggest corrections. All of these are examples of crowdsourcing - relying upon the inputs from many people to learn new information.

When it comes to search relevance, crowdsourcing can play a vital role, though it is usually important not to annoy your valued customers by constantly asking them for help. Fortunately, it is often possible to learn from your users implicitly based upon their behaviors without having to bother them by explicitly asking for input. For example, if you are trying to find the best documents to return for a given search, why not examine your logs to determine which documents other users most often clicked on in response to the same query?

4.1 Working with User Signals

Every time a customer takes an action - whether it be issuing a query, clicking on a result, purchasing a product, or otherwise taking some action on the search results - this provides a signal of that user’s intent. We can log each of these signals and then process them in order to learn insights about each user, about groups of users, and about an entire user base. This section introduces you to the power of leveraging user signals, introduces a sample ecommerce dataset we will use throughout the book, and walks you through the mechanics of collecting, storing, and processing the user signals.

4.1.1 Signals vs. Content

When building search engines, we have two high-level types of data - "content" and "signals". Most of the content we deal with is in the form of documents. Documents can represent web pages, product listings, computer files, images, facts, or any other type of information that we may want to search through. Content documents usually contain fields with some kind of text through which we can search and find relationships, along with additional fields representing other attributes related to the content (author, size, color, dates, and so on). The defining characteristic of these content documents is that they contain the information through which people are searching (and hopefully also the answers for which they are searching). In addition to traditional documents, however, many other kinds of content can be incorporated into a search experience. An externally-built knowledge graph, a list of entities (people, places, things), customer or employee-created comments, tags, or attributes that are added to the documents, and so on, are all forms of content.

"Signals", on the other hand, reflect how users engage with content. When someone issues a query, they receive a set of documents with content. Perhaps they click on a result, add it to their shopping cart, bookmark the document, or take other similar actions. We refer to these interactions as signals, and their defining characteristic is that they provide external insights that can be used to understand how users want to interact with content. Of course, those signals can also later be added to documents along with the pre-existing content, and if you are building an application that allows for searching through the signals, those signals actually then become new content for that application. Notwithstanding that in some cases signals can also be treated as content depending on the use case, the point here is that there are two key sources of information we can use to improve search: the attributes of the items being searched upon (content), and the observed user interactions with the items (signals).

For many of the important tasks we undertake when building AI-powered search, we can derive similar outcomes by using either the content or the signals, but they give us two different perspectives of relevance. In ideal cases, we can actually bring in both of those perspectives to build an even smarter system, but it is useful to understand the strengths and weaknesses of each approach to understand how to best leverage them.

As an example, if we are trying to find a synonym for the word "driver", we can look through all the text content for words that commonly appear in the same documents. We may, in this case, find words in priority order (by percentage of documents they appear within) like "taxi" (40%), "car" (35%), "golf" (15%), "club" (12%), "printer"(3%), "linux" (3%), and "windows" (1%). Similarly, we could look at our collected signals for all users who searched for `driver` and aggregate the most common other keywords for which they searched, and find similar words in priority order like "screwdriver" (50%), "printer" (30%), "windows" (25%), "mac" (15%), "golf" (2%), "club" (2%). The lists derived from signals versus content might be similar, or they could look very different. The content-based approach tells us the most represented meanings within our documents, whereas the signals-based approach tells us the most represented meanings being looked for by our *users*.

Since our end goal is to present users what they are looking for, we would tend to favor the signals-derived meanings over the content-derived meanings in most cases. However, what if we don’t actually have good content that maps to the signals-derived meaning? Do we use the content-derived meaning, or do we try to suggest other related searches based upon the signals data? What if we don’t have enough signals or if the signals data is not very clean? Can we somehow clean up the signals-derived data using the content-derived data?

We run into similar questions with recommendations: content-based recommendations leverage attributes in documents but don’t understand users, whereas signals-based recommendations don’t understand content attributes and don’t work at all on items which don’t have sufficient interactions. Content-based recommendations may recommend on features that are unimportant to users, whereas signals-based recommendations may create self-reinforcing loops where people only interact with items they are recommended, and then only those items get recommended because they were the only ones with which users interacted.

Ideally, we want to create a balanced system that can leverage the best of both content-derived and signals-derived intelligence. While this chapter focuses primarily on signals-derived, crowdsourced intelligence, a major goal of this book is to show how to balance and combine both content-based and signals-based approaches to yield an optimal AI-powered search experience.

4.1.2 Setting up our product and signals datasets (RetroTech)

We leverage various datasets throughout this book as we explore different use cases, but it is also valuable to have a consistent example that we can build on as we progress. We will set that central dataset up in this section and will continue to build upon it for many chapters to come.

Web search and Ecommerce are two of the most universally-recognizable use cases for AI-powered search today. We feel that of the two, Ecommerce presents the best opportunity for exploring the widest variety of AI-powered search techniques. Ecommerce examples will also map more easily to the real-world use cases being delivered by the largest portion of the readers of this book. It's worth noting that most techniques in this book apply across use cases - web search, enterprise search, site search, desktop search, mail search, ediscovery, job search, product search, support portal search - you name it. The deciding factor for when to use any given technique typically relates more to the volume and variety of content and signals than it does to the particular use case.

With that said, let's introduce our example use case and dataset: RetroTech!

THE RETROTECH USE CASE

Throughout much of the book, we will benefit by having a robust search use case with lots of data and user interactions. Ecommerce search provides one of the most concrete use cases for the value of AI-powered search techniques, and it is also one of the most well understood problems among readers, so we've created an ecommerce dataset to help us explore this domain: the RetroTech dataset.

With aggressive competition among retailers selling the latest and greatest electronics, multimedia, and tech products, it is hard for a small online business to compete. However, a niche but emerging segment of the population chooses to avoid the latest and greatest products and instead falls back to the familiar technology of decades past. The RetroTech company was launched to meet the needs of this unique group of consumers, offering vintage hardware, software, and multimedia products that may be hard to find on today's shelves.

Let's load the dataset so we can get started learning about the relationships between documents and user signals, and how crowdsourced intelligence can improve our search relevance.

LOADING THE PRODUCT CATALOG

The RetroTech website has around 50,000 products available for sale, so we'll need to load those into our search engine to get started so that we can search through them. If you built the *AI-Powered Search* code base to run the chapter 3 examples, then your search engine is already up and running. If you haven't yet done that, the instructions for building and running all of the book's examples are found in Appendix A, which you can run through now to get setup.

With your search engine up and running, the next thing we need to do is download the Retrotech dataset that accompanies this book. The dataset includes two CSV files, one containing all of Retrotech's products, and another containing one year of signals data from Retrotech's users. Listing 4.1 shows a few rows of the product catalog dataset to get you familiar with the format.

Listing 4.1 Exploring the RetroTech product catalog

```
"upc","name","manufacturer","shortDescription","longDescription"
"096009010836","Fists of Bruce Lee - Dolby - DVD",\N,\N,\N
"043396061965","The Professional - Widescreen Uncut - DVD",\N,\N,\N
"085391862024","Pokemon the Movie: 2000 - DVD",\N,\N,\N
"067003016025","Summerbreeze - CD","Nettwerk",\N,\N
"731454813822","Back for the First Time [PA] - CD","Def Jam South",\N,\N
"024543008200","Big Momma's House - Widescreen - DVD",\N,\N,\N
"031398751823","Kids - DVD",\N,\N,\N
"037628413929","20 Grandes Exitos - CD","Sony Discos Inc.",\N,\N
"060768972223","Power Of Trinity (Box) - CD","Sanctuary Records",\N,\N
```

You can see that products are identified by a UPC (Universal Product Code) number and then have a name, a manufacturer, and both a short description (used as a teaser in search results) and a long description (the full description used on product details pages).

Since our goal is to search for products, our next step will be to send them to the search engine to be indexed. To enable search on our RetroTech product catalog, let’s run the document indexing code in Listing 4.2 to send the product documents to the search engine.

Listing 4.2 Send product documents to the search engine

```
#Create Products Collection
products_collection="products"
create_collection(products_collection)

#Modify Schema to make some fields explicitly searchable by keyword
upsert_text_field(products_collection, "upc")
upsert_text_field(products_collection, "name")
upsert_text_field(products_collection, "longDescription")
upsert_text_field(products_collection, "manufacturer")

print("Loading Products...")
csvFile = "../data/retrotech/products.csv"
product_update_opts={"zkhost": "aips-zk", "collection": products_collection,
➥"gen_uniq_key": "true", "commit_within": "5000"}
csvDF = spark.read.format("com.databricks.spark.csv").option(
➥"header", "true").option("inferSchema", "true").load(csvFile)
csvDF.write.format("solr").options(**product_update_opts).mode(
➥"overwrite").save()
print("Products Schema: ")
csvDF.printSchema()
print("Status: Success")
```

Results:

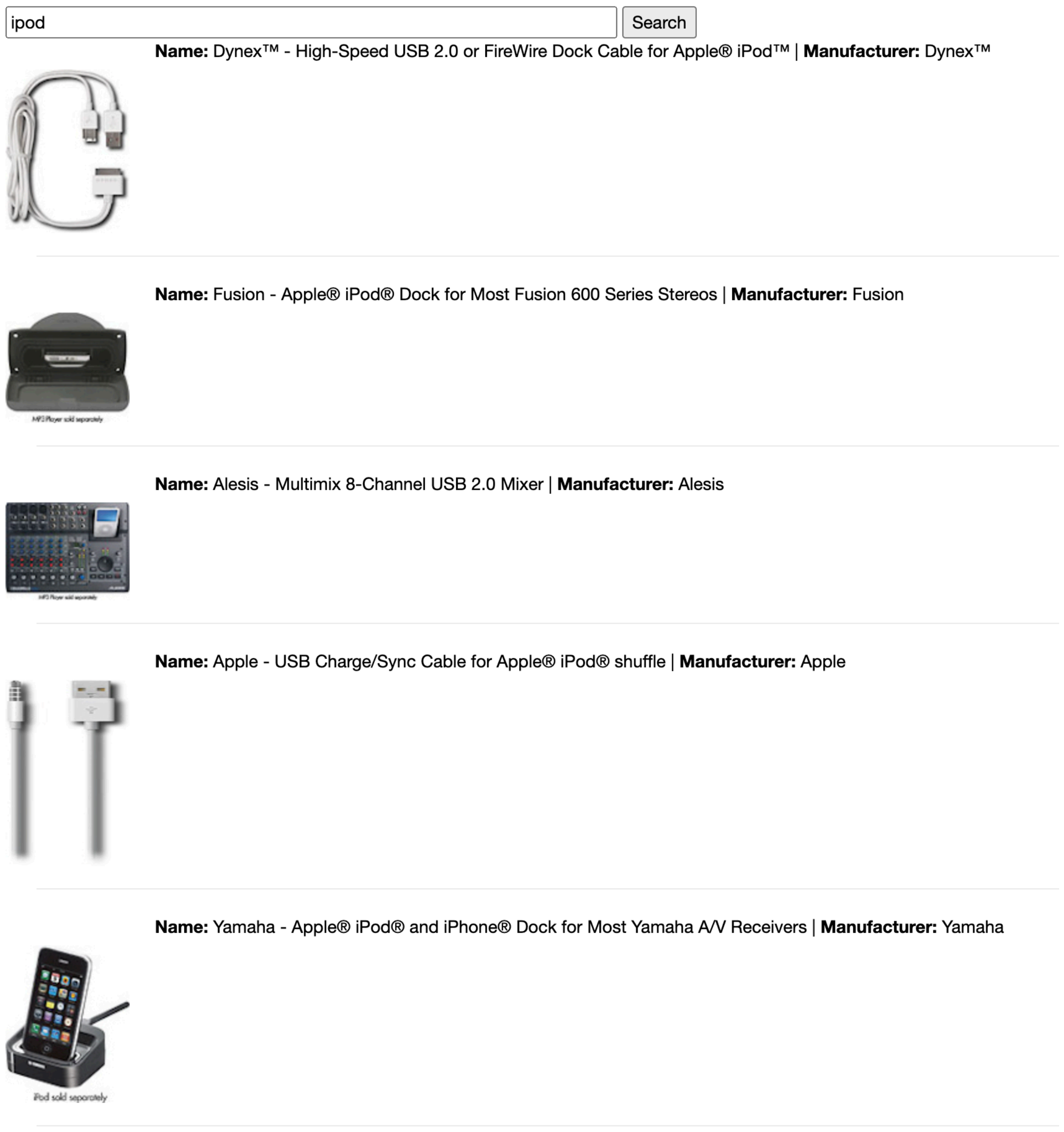
```
Wiping 'products' collection
Creating 'products' collection
Status: Success
Adding 'upc' field to collection
Status: Success
Adding 'name' field to collection
Status: Success
Adding 'longDescription' field to collection
Status: Success
Adding 'manufacturer' field to collection
Status: Success
Loading Products...
Products Schema:
root
 |-- upc: long (nullable = true)
 |-- name: string (nullable = true)
 |-- manufacturer: string (nullable = true)
 |-- shortDescription: string (nullable = true)
 |-- longDescription: string (nullable = true)

Status: Success
```

Finally, to verify that the documents are now indexed and searchable, let's run an example keyword search. Listing 4.3 shows an example search for `ipod`, a true classic device!

Listing 4.3 Running a search on the product catalog

```
query = "ipod"

collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax",
      "sort": "score desc, upc asc"
    }
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

The results of the `ipod` search from Listing 4.3 are shown in Figure 4.1, demonstrating that our products are now indexed and searchable. Unfortunately, the relevance of the results is quite poor, however.



Name: Dynex™ - High-Speed USB 2.0 or FireWire Dock Cable for Apple® iPod™ | **Manufacturer:** Dynex™

Name: Fusion - Apple® iPod® Dock for Most Fusion 600 Series Stereos | **Manufacturer:** Fusion

Name: Alesis - Multimix 8-Channel USB 2.0 Mixer | **Manufacturer:** Alesis

Name: Apple - USB Charge/Sync Cable for Apple® iPod® shuffle | **Manufacturer:** Apple

Name: Yamaha - Apple® iPod® and iPhone® Dock for Most Yamaha A/V Receivers | **Manufacturer:** Yamaha

Figure 4.1 Product Search Results. We can see that the product catalog has been indexed and a query for `ipod` now returns search results.

While the quality of the search results ranking is not very good, we at least now have an out of the box "keyword matching" search engine that we can begin improving. We’ll use this as our base and start introducing more intelligent AI-powered search features throughout the rest of the book. Our next step will be to introduce our signals data.

LOADING THE SIGNALS DATA

Since RetroTech is running on your computer, you're not going to have real users visiting, running searches, clicking and purchasing, and otherwise generating signals. Because of that, we've generated a dataset for you to use that approximates the kind of signal activity you'd expect in similar real-world datasets.

For simplicity, we will store our signals in the search engine to enable them to be leveraged both in real-time search scenarios and for external processing. Running Listing 4.4 will simulate and index some sample signals that we can leverage throughout the rest of the chapter.

Listing 4.4 Indexing the User Signals Dataset

```
#Create Signals Collection
signals_collection="signals"
create_collection(signals_collection)

print("Loading Signals...")
csvFile = "../data/retrotech/signals.csv"
signals_update_opts={"zkhost": "aips-zk", "collection": signals_collection,
➥"gen_uniq_key": "true", "commit_within": "5000"}
csvDF = spark.read.format("com.databricks.spark.csv").option(
➥"header", "true").option("inferSchema", "true").load(csvFile)
csvDF.write.format("solr").options(**signals_update_opts).mode(
➥"overwrite").save()
print("Signals Schema: ")
csvDF.printSchema()
print("Status: Success")
```

Results:

```
Wiping 'signals' collection
Creating 'signals' collection
Status: Success
Loading Signals...
Signals Schema:
root
 |-- query_id: string (nullable = true)
 |-- user: string (nullable = true)
 |-- type: string (nullable = true)
 |-- target: string (nullable = true)
 |-- signal_time: timestamp (nullable = true)

Status: Success
```

With our Retrotech product and signals data all loaded, we'll soon begin exploring ways to use the signals data to enhance our search relevance. Before we dive into these crowdsourced relevance techniques, though, let's first explore the signals data a bit so we can understand how signals are structured, used, and collected in real-world systems.

4.1.3 Exploring the signals data

Different types of signals have different attributes which need to be recorded. For example, for a "query" signal, we want to record the user's keywords. For a "click" signal, we want to record which document was clicked upon, as well as which query resulted in the click. For later analysis, we'd also want to record which documents were returned to and possibly viewed by a user after a query.

In order to make things more extensible and avoid custom code for every new signal type, we've adopted a generic format for representing signals in this book. This will likely differ from how you log your signals, but as long as you can subsequently map your signals into this format, then all of the code in this book should work without requiring use-case specific modifications.

The signals format we use in this book is as follows:

- **query_id**: a unique id for the query signal that originated this signal.
- **user**: an identifier representing a specific user of the search engine
- **type**: what kind of signal (query, click, purchase, and so on)
- **target**: the content to which the signal at this `signal_time` applies.
- **signal_time**: the date and time the signal occurred

As an example, assume a user performed the following sequence of actions:

1. issued a query for `ipad` and had three documents (`doc1`, `doc2`, and `doc3`) returned.
2. clicked on `doc1`.
3. went back and clicked on `doc3`.
4. added `doc3` to the shopping cart.
5. went back and searched for `ipad cover` and had two documents returned (`doc4`, `doc5`).
6. clicked on `doc4`.
7. added `doc4` to the shopping cart.
8. purchased the items in the shopping cart (`doc3`, `doc4`).

These interactions would result in the following signals:

Table 4.1 Example signals format

query_id	user	type	target	signal_time
1	u123	query	ipad	2020-05-01-09:00:00
1	u123	results	doc1,doc2,doc3	2020-05-01-09:00:00
1	u123	click	doc1	2020-05-01-09:00:10
1	u123	click	doc3	2020-05-01-09:00:29
1	u123	add-to-cart	doc1	2020-05-01-09:03:40
2	u123	query	ipad cover	2020-05-01-09:04:00
2	u123	results	doc4,doc5	2020-05-01-09:04:00
2	u123	click	doc4	2020-05-01-09:04:40
2	u123	add-to-cart	doc4	2020-05-01-09:05:50
1	u123	purchase	doc3	2020-05-01-09:07:15
2	u123	purchase	doc4	2020-05-01-09:07:15

A few things to note about the signals format:

1. **The `query` type and the `results` type signal are broken up into separate signals.** This isn't necessary, as they occur at the same time, but this allows us to keep the table structure consistent and not have to add an extra `results` column that only applies to the `query` signal. Also, if the user hits the "next page" link or scrolls down the page and sees additional results, this structure will allow us to append a new signal at the new time for those results being returned without having to go back and modify the original signal.
2. **Every signal ties back to the `query_id` of the original `query` signal that started the series of content interactions.** The `query_id` is not just a reference to the keywords entered by the user, but is instead a reference to the specific `query` signal identifying a time-stamped instance of the user's query. Because results for the same query keywords can change over time, this enables us to do more sophisticated processing of how users reacted to the specific set of results they were shown for a query.
3. **Most signal types contain one item in the `target`, but the `results` type contains an ordered list of documents.** The reason is that it's important to preserve the exact ordering of the search results, so the "target" is really an ordered list of documents in this case instead of independent documents. The order of results will matter for some algorithms we introduce later in the book to measure relevance.
4. **The checkout resulted in a separate `purchase` signal for each item instead of just one `checkout` signal.** This is because we need to track that the two items that were purchased originated from separate queries. You could, of course, also add a `checkout` signal type if you wanted to track the transaction, and possibly list the two purchases as the `target`, but this is superfluous for our needs in this book, so we'll avoid this added complexity.

With these raw signals available as our building blocks, we can now start thinking about how to link the signals together to begin learning about our users and their interests. In the next section, we'll discuss ways to model users, sessions, and requests within our search platform.

4.1.4 Modeling users, sessions, and requests

In the last section, we looked at the structure of user signals as a list of independent interactions tied back to an original query. We assumed that a "user" was present with a unique ID, but how does one identify and track a unique user? Furthermore, once you have identified how to track unique users, what is the best way to break their interactions up into sessions to understand when their context may have changed?

The concept of a user in web search can be very fluid. If your search engine has authenticated (logged-in) users, then you already have some kind of an internal user ID to track them. If your search engine supports unauthenticated access or is publicly available, however, then you will have many running searches with no formal user ID. That doesn't mean you can't track them, however; it just requires a more fluid interpretation of what a "user" means. The reason we want to track a user is so that we can relate different signals together to learn interaction patterns, and without some shared identifier representing that the same user is issuing multiple requests, it is not easy to tie those interactions together.

If we think of the known information as a hierarchy from "best" representation of a user to "worst," it will look something like the following:

- *User ID*: a unique user ID that persists across devices (authenticated)
- *Device ID*: a unique id that persists across sessions on the same device (such as a device ID or an IP address + device fingerprint)
- *Browser ID*: a unique ID that persists across sessions in the same application/browser only (a persistent cookie id)
- *Session ID*: a unique ID that persists across a single session (such as cookie in a browser's incognito mode)
- *Request ID*: a unique ID that only persists for a single request (a browser with cookies turned off)

In most modern search applications, and certainly in most ecommerce applications, we typically have to deal with all of these. As a rule of thumb, you want to tie a user to the most durable identifier - the one as high up the list as possible. Both the links between request IDs and session IDs, as well as the links between session IDs and browser IDs, are through the user's cookie, so ultimately the browser ID (persistent unique ID stored in the cookie) is the common denominator for each of these.

Specifically,

- If a user has persistent cookies enabled, one browser ID can have many session IDs, which can have many request IDs.
- If a user clears cookies after each session (such as by using incognito mode), each browser ID has only one session ID, which can have many request IDs.
- If a user turns off cookies, then each request ID has a new session ID and a new browser ID.

When building search platforms, most organizations do not properly plan for and design their signals tracking mechanisms. By not being able to correlate visitors' queries with specific resulting actions (order of result viewed, which results were clicked, and any subsequent interactions with the clicked items), many organizations make it difficult to maximize the relevance of their search platform through enhanced analytics and through automated search relevance approaches. In some cases, it is possible to derive missing signals tracking information after the fact (such as modeling signals into likely sessions using timestamps), but it is usually best to design the system to better handle user tracking upfront to prevent potential information loss. Richer signals will allow you to optimally relate user interactions such that they can add maximum value to your AI-powered search efforts. In section 4.2, we'll introduce how these rich signals actually get used to improve relevance through a process known as "Reflected Intelligence"

4.2 Introduction to Reflected Intelligence

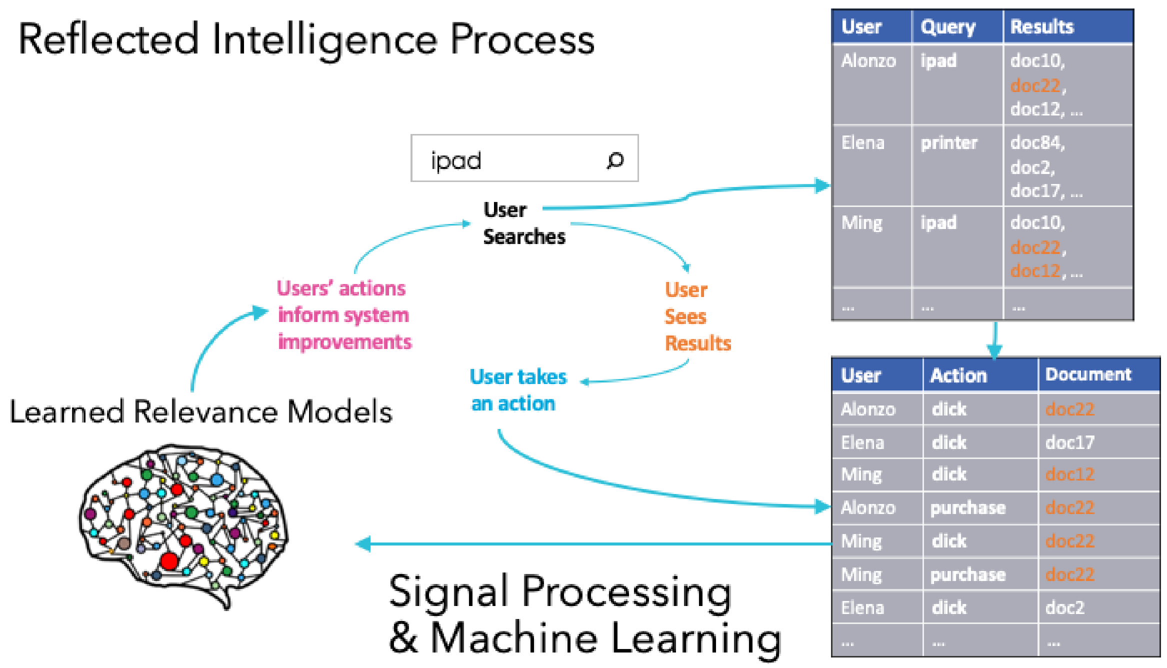
In the last section, we covered how to capture signals from users as they interaction with your search engine. While the signals themselves are useful to help us understand how our search engine is being used, they also serve as the primary inputs for building models that can continually learn from user interactions and enable your search engine to self-tune its relevance model. In this section we'll introduce how these self-tuning model work through the concept of Reflected Intelligence.

4.2.1 What is Reflected Intelligence?

Imagine you are an employee at a hardware store and someone asks you where they can find a hammer, and you tell them "aisle two". A few minutes later you see the same person walk from aisle two to aisle five without a hammer, and then walk out of aisle five holding a hammer. The next day someone else asks for a hammer, you also tell them "aisle two", and you observe a nearly identical pattern of behavior. You would be a pretty lousy employee if you didn't spot this pattern and adjust your advice going forward to provide a better experience for your customers. Now imagine if you continued to give the same, poor answer hundreds or even thousands of times to every new customer who came through the door looking for a hammer.

Unfortunately, this is exactly how most search engines tend to work - they have a fairly static set of documents that are returned for each query, regardless of who each user is or how prior users have reacted to the list of documents shown. Thankfully, we can improve upon this substantially by applying machine learning techniques to the collected signals from user interactions. This enables us to learn about users' intent from their signals and then reflect that knowledge back to improve future search results. This process is called *Reflected Intelligence*.

Reflected Intelligence is all about creating feedback loops that constantly learn and improve based on evolving user interactions. Figure 4.2 demonstrates a high-level overview of implementing a Reflected Intelligence process.


Reflected Intelligence Process
ipad
User Searches
User Sees Results
User takes an action
Users' actions inform system improvements
Learned Relevance Models
Signal Processing & Machine Learning

User	Query	Results
Alonzo	ipad	doc10, doc22, doc12, ...
Elena	printer	doc84, doc2, doc17, ...
Ming	ipad	doc10, doc22, doc12, ...
...	...	...

User	Action	Document
Alonzo	click	doc22
Elena	click	doc17
Ming	click	doc12
Alonzo	purchase	doc22
Ming	click	doc22
Ming	purchase	doc22
Elena	click	doc2
...	...	...

Figure 4.2 Reflected Intelligence Process. A user issues a query, sees results, and takes a set of actions. Those actions (signals) are then processed to create learned relevance models that improve future searches.

Let's walk through the Reflected Intelligence process shown in Figure 4.2. A user named Alonzo runs a search, entering the term `ipad` in the search box. A query signal is logged, containing the list of all search results displayed to Alonzo. Alonzo then sees the list of search results, and takes some actions. In the figure, Alonzo clicks on a document (`doc22`) and then purchases the product that document represents, resulting in two additional corresponding signals. All of Alonzo's signals, along with the signals from every other user, can then be aggregated and otherwise processed by various machine learning algorithms to create learned relevance models.

These learned relevance models may boost the most popular results for specific queries, personalize results for each specific user and their interests, or even learn which general attributes of the documents being searched tend to matter the most and tune the ranking algorithms to factor those attributes. The models could also learn how to better interpret user queries, such as identifying common misspellings, phrases, synonyms, or other linguistic patterns and domain-specific terminology.

Once these learned relevance models are generated, they can then be deployed back into the production search engine and immediately be applied to enhance the outcomes of future queries. The process then begins again, with the next user running a search, seeing (now hopefully improved) search results, and interacting with those results. This process creates a self-learning system which improves with every additional user interaction, getting continually smarter and more relevant over time, and also automatically adjusting as user interests and content evolve.

In the following sections, we explore a few categories of reflected intelligence models, including signals boosting (popularized relevance), collaborative filtering (personalized relevance), and learning to rank (generalized relevance). We'll start with one of the simplest and also most most impactful: signals boosting models.

4.2.2 Popularized Relevance through Signals Boosting

The most popular queries sent to your search engine tend to also be the most important ones to optimize from a relevance standpoint. Thankfully, since more popular queries generate more signals, this means that they are generally much easier to crowdsource reflected intelligence models for in order to improve relevance.

Signals boosting is one of the simplest forms of Reflected Intelligence, but also one of the most impactful for improving relevance of your most popular, highest-traffic queries.

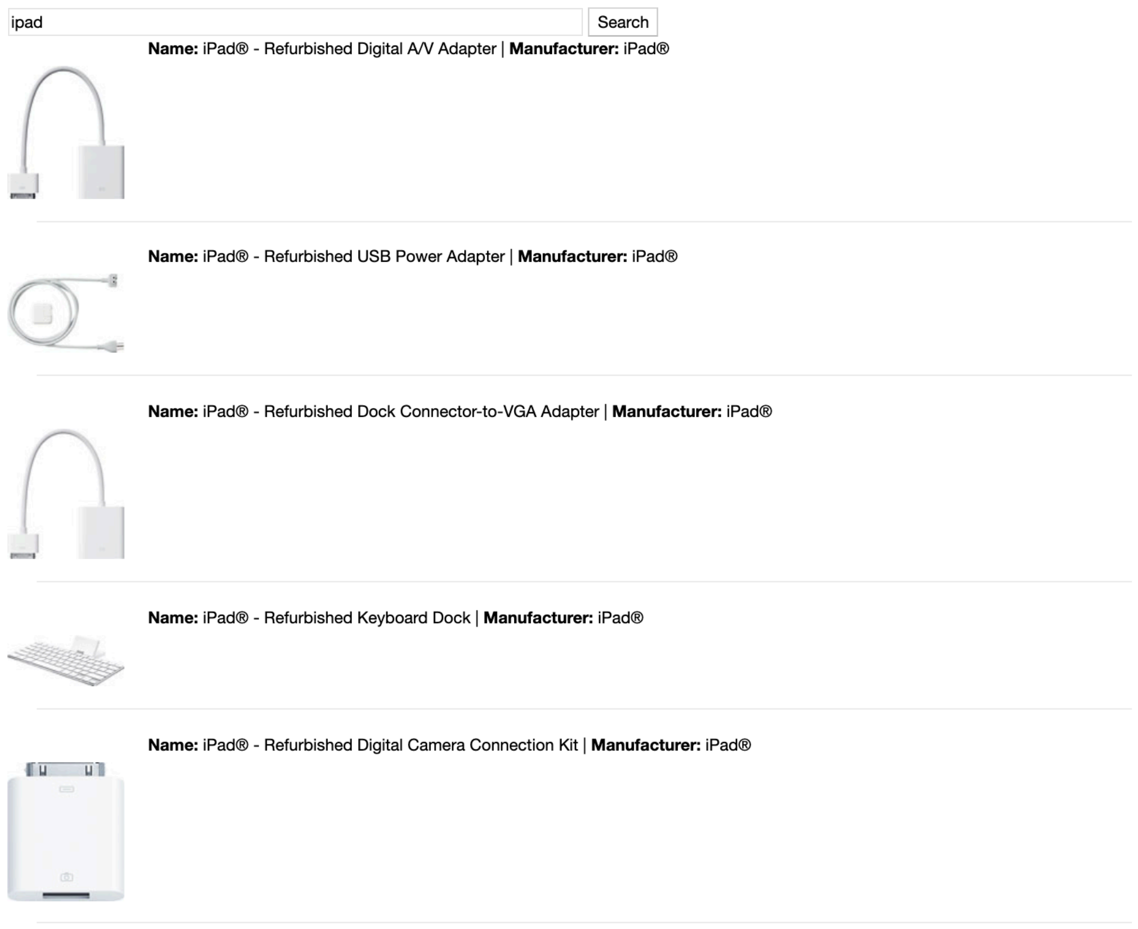
Signals boosting models operate well on the most common and highest volume queries, making them an ideal way to learn a "popularized relevance" model.

Listing 4.5 demonstrates an out of the box search for the query `ipad` in our RetroTech search engine.

Listing 4.5 Sending a keyword search for the query `ipad` to the search engine.

```
query = "ipad"

collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax"
    }
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

As expected, the results of this query will return many documents containing the keyword `ipad` in them, and, based upon what we learned in chapter 3 about how keyword relevance is scored leveraging TF-IDF and the BM25 ranking algorithm, the documents containing the term `ipad` the most times will typically rank the highest. Figure 4.3 shows the results of the query from Listing 4.5.


ipad

Search

Name: iPad® - Refurbished Digital A/V Adapter | **Manufacturer:** iPad®

Name: iPad® - Refurbished USB Power Adapter | **Manufacturer:** iPad®

Name: iPad® - Refurbished Dock Connector-to-VGA Adapter | **Manufacturer:** iPad®

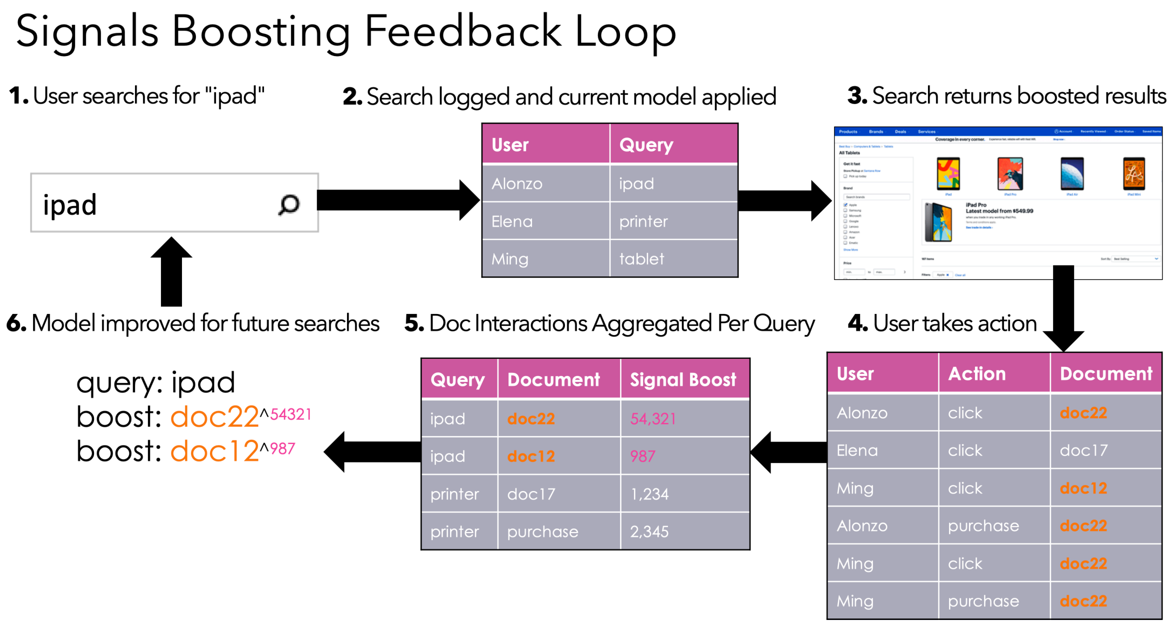
Name: iPad® - Refurbished Keyboard Dock | **Manufacturer:** iPad®

Name: iPad® - Refurbished Digital Camera Connection Kit | **Manufacturer:** iPad®

Figure 4.3 Results of a keyword search for the query `ipad`. Results are returned primarily based upon number of occurrences of the keyword, so accessories mentioning the keyword multiple times rank higher than the actual product the user intended to see.

While these results all contain the word "ipad" in their content multiple times, most users would be disappointed with these results since they are secondary accessories as opposed to the main product type that was the focus of the search. As you can probably guess, it is very hard to figure out the main product versus the secondary accessories just from the text in the product documents. For very popular queries, however, it is likely that many customers will run the same queries over and over again and fight through the frustrating search results to ultimately find the real products they are seeking. Signals Boosting is a technique for leveraging this aggregate user behavior from popular queries in order to automatically learn and return the best products.

Figure 4.4 demonstrates how how signals boosting works as a continuous feedback loop.


Signals Boosting Feedback Loop
1. User searches for "ipad"
ipad
2. Search logged and current model applied
3. Search returns boosted results
4. User takes action
5. Doc Interactions Aggregated Per Query
6. Model improved for future searches
query: ipad
boost: doc22^54321
boost: doc12^987

User	Query
Alonzo	ipad
Elena	printer
Ming	tablet

User	Action	Document
Alonzo	click	doc22
Elena	click	doc17
Ming	click	doc12
Alonzo	purchase	doc22
Ming	click	doc22
Ming	purchase	doc22

Query	Document	Signal Boost
ipad	doc22	54,321
ipad	doc12	987
printer	doc17	1,234
printer	purchase	2,345

Figure 4.4 Signals Boosting Feedback Loop. A user’s search is logged, and the current signals boosting model is applied to return boosted results. After users take action on those results, the signals from all user interactions with documents are aggregated by originating query to generate an updated signals boosting model to further improve future searches.

Once your products are indexed and you’ve started collecting signals for your users' queries and document interactions, the only additional steps necessary for implementing signals boosting are to aggregate your signals, and then to add your aggregated signals as boosts to either your queries or your documents. Listing 4.6 demonstrates a simple model for aggregating signals into a side-car collection.

SIDEBAR **Sidecar Collections**

Sidecar collections are additional collections that sit in your search engine alongside a primary collection and which contain other useful data to improve your search application. In our ecommerce example, our primary collection is the `products` collection, and we’ve already introduced our `signals` collection, which can be considered a sidecar collection. In this section we will also introduce a `signals_boosting` sidecar collection, which we will leverage at query time to enhance our queries. Throughout the book, we’ll introduce several other sidecar collections to store the inputs for and outputs of our self-learning models.

Listing 4.6 Generating a signals boosting model through aggregating signals.

```
products_collection="products"
signals_collection="signals"
signals_boosting_collection="signals_boosting"

create_collection(signals_boosting_collection)

signals_opts={"zkhost": "aips-zk", "collection": signals_collection}
signals_boosting_opts={"zkhost": "aips-zk", "collection":
➥signals_boosting_collection, "gen_uniq_key": "true",
➥"commit_within": "5000"}

df = spark.read.format("solr").options(**signals_opts).load()
df.registerTempTable("signals")

print("Aggregating Signals to Create Signals Boosts...")

signals_aggregation_query = """
select q.target as query, c.target as doc, count(c.target) as boost
  from signals c left join signals q on c.query_id = q.query_id
  where c.type = 'click' AND q.type = 'query'
  group by query, doc
  order by boost desc
"""

spark.sql(signals_aggregation_query).write.format("solr").options(
➥**signals_boosting_opts).mode("overwrite").save()
print("Signals Aggregation Completed!")
```

Results:

```
Wiping 'signals_aggregation' collection
Creating 'signals_aggregation' collection
Status: Success
Aggregating Signals to Create Signals Boosts...
Signals Aggregation Completed!
```

The most important part of Listing 4.6 is the `signals_aggregation_query`, which we've actually just defined as a SQL query to keep the example more readable. For every query, we are getting the list of documents that users have clicked on in the search results for that query, along with a count of how many times the document has been clicked on. By ordering the documents by the count of times clicked for each query, we now have an ordered list of the documents that users tended to choose to interact with for each query.

The intuition here is that users tend to choose the products they believe are the most relevant, so if we were to boost these documents, then we would expect our top search results to become more relevant. We'll test this theory out in Listing 4.7 by using these aggregated counts as Signals Boosts on our next query. Let's revisit our previous query for `ipad`.

Listing 4.7 Generating a Signals Boosting Query to improve search relevance ranking for top queries and documents.

```
query = "ipad"

signals_boosts_query = {
    "query": query,
    "fields": ["doc", "boost"],
    "limit": 10,
    "params": {
      "defType": "edismax",
      "qf": "query",
      "sort": "boost desc"
    }
}

signals_boosts = requests.post(solr_url + signals_boosting_collection +
➥"/select", json=signals_boosts_query).json()["response"]["docs"]
print("Boost Documents: \n")
print(signals_boosts)

product_boosts = ""
for entry in signals_boosts:
    if len(product_boosts) > 0:  product_boosts += " "
    product_boosts += '"' + entry['doc'] + '"^' + str(entry['boost'])

print("\nBoost Query: \n" + product_boosts)


collection = "products"
request = {
    "query": query,
    "fields": ["upc", "name", "manufacturer", "score"],
    "limit": 5,
    "params": {
      "qf": "name manufacturer longDescription",
      "defType": "edismax",
      "indent": "true",
      "sort": "score desc, upc asc",
      "qf": "name manufacturer longDescription",
      "boost": "sum(1,query({! df=upc v=$signals_boosting}))",
      "signals_boosting": product_boosts
    }
}

search_results = requests.post(solr_url + collection + "/select", json=request)
➥.json()["response"]["docs"]
display(HTML(render_search_results(query, search_results)))
```

Boost Documents:

```
[{'doc': '885909457588', 'boost': 966}, {'doc': '885909457595',
➥'boost': 205}, {'doc': '885909471812', 'boost': 202},
➥{'doc': '886111287055', 'boost': 109}, {'doc': '843404073153',
➥'boost': 73}, {'doc': '635753493559', 'boost': 62}, {'doc':
➥'885909457601', 'boost': 62}, {'doc': '885909472376', 'boost': 61},
➥{'doc': '610839379408', 'boost': 29}, {'doc': '884962753071',
➥'boost': 28}]
```

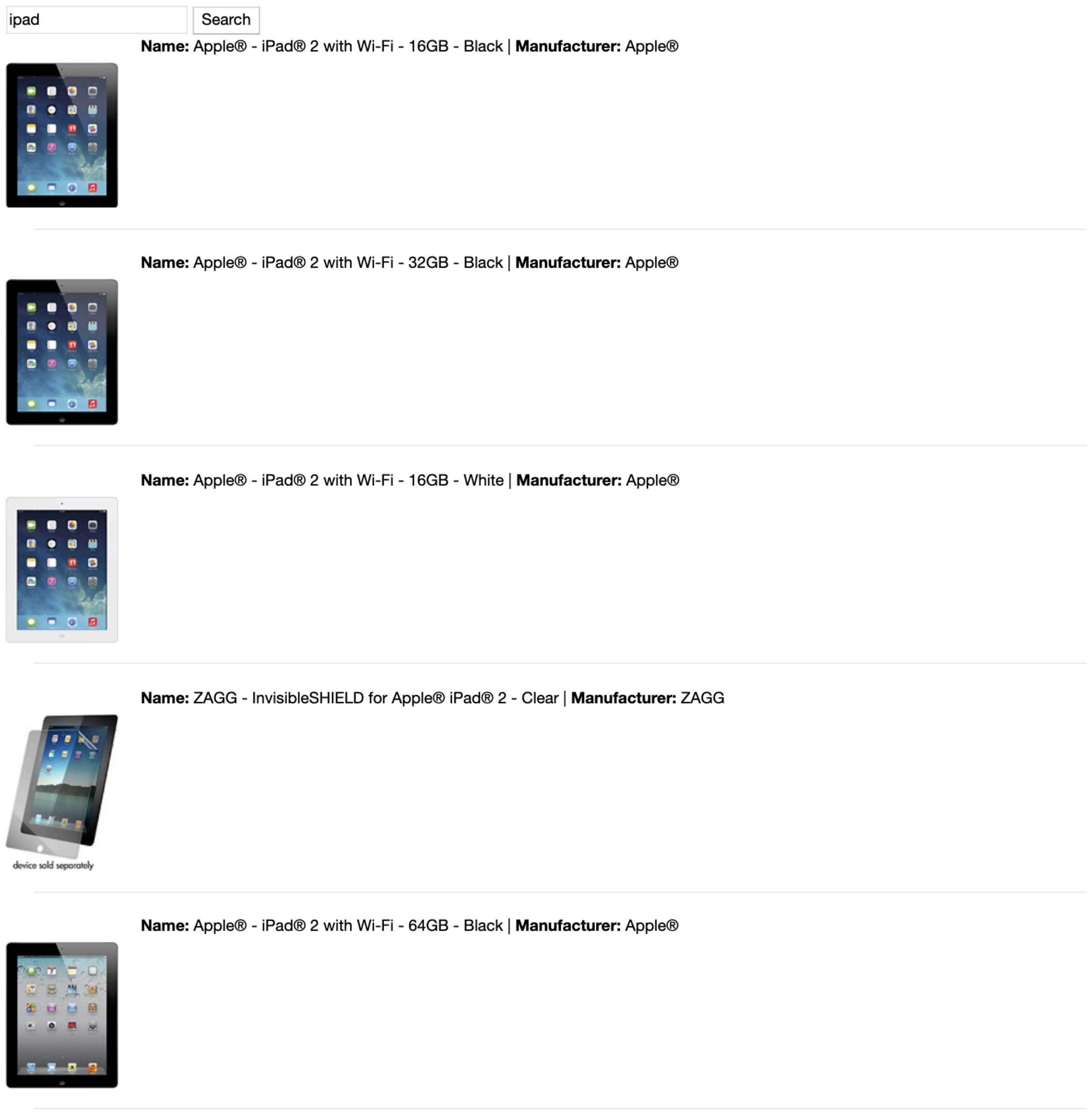
Boost Query:

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
➥"843404073153"^73 "635753493559"^62 "885909457601"^62 "885909472376"^61
➥"610839379408"^29 "884962753071"^28
```

The query in Listing 4.7 does two noteworthy things:

1. It queries the `signals_boosting` sidecar collection to look up the ordered (by boost) list of documents that received the highest signals boosts for the query, and it transforms those signals boosts into a query to the search engine
2. It then passes that boosting query to the search engine with the following relevance boosting parameters:
   - `"boost": "sum(1,query({! df=upc v=$signals_boosting}))"`
   - `"signals_boosting": product_boosts` where `product_boosts` is the "Boost Query" output generated from Listing 4.7.

Once the query is executed, we can look at how it improves our search results. If you remember from Figure 4.3, our original keyword search for `ipad` returned mostly iPad accessories, as opposed to actual iPad devices. Figure 4.5 demonstrates the new results based upon signals boosting being applied on top of the keyword query.



Name: Apple® - iPad® 2 with Wi-Fi - 16GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 32GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 16GB - White | **Manufacturer:** Apple®

Name: ZAGG - InvisibleSHIELD for Apple® iPad® 2 - Clear | **Manufacturer:** ZAGG

Name: Apple® - iPad® 2 with Wi-Fi - 64GB - Black | **Manufacturer:** Apple®

Figure 4.5 Search results with Signals Boosting Enabled. Instead of iPad accessories showing up as before, we now see actual iPads, because we have crowdsourced the answers based upon the documents with which users actually choose to interact.

The new results after applying signals boosting are significantly better than the keyword-only results. We now see actual iPads, the product the user typed in and almost certainly intended to find. You can expect to see similarly good results from most of the other popular queries in your search engine, since the more people that interact with them, the greater the reliability of this crowdsourcing approach for figuring out relevance. Of course, as we move further down the list of popular products, the relevance improvements from signals boosting will start to decline, and with insufficient signals we may even reduce relevance in many cases. Thankfully, we'll introduce many other techniques that can better handle the queries without adequate signals volume.

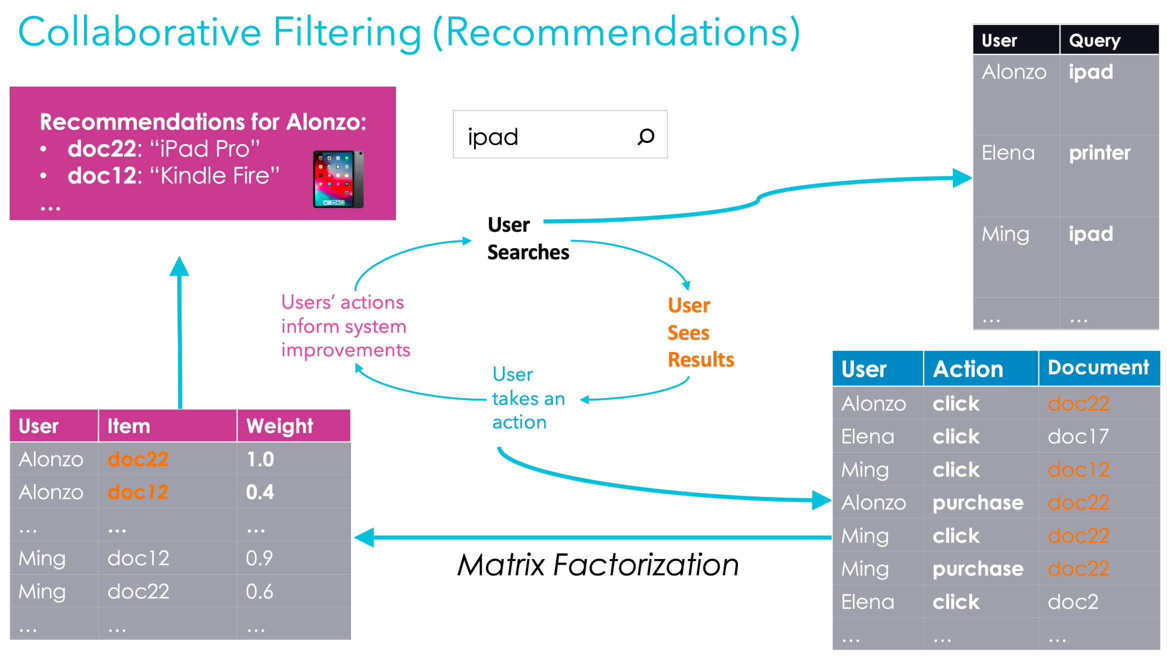
The goal of this section was to walk you through an initial, concrete example of implementing an end-to-end reflected intelligence model. The signals aggregation used in this impelementation was very simple, though the results speak for themselves. There are many considerations and nuances to consider when implementing a signals boosting model - whether to boost at query time or on documents, how to increase the weight of newer signals versus older signals, how to avoid malicious users trying to boost particular products in the search results by generating bogus signals for the system to interpret, how to introduce and blend signals from different sources, and so on. We'll cover each of these topic in detail in chapter 8, Signals Boosting Models.

Let's move on from Signals Boosting for now, however, and discuss a few other types of Reflected Intelligence models.

4.2.3 Personalized Relevance through Collaborative Filtering

In the section 4.2.2, we covered signals boosting, which we referred to as "popularized relevance", since it determines the most popular answers to common queries across all users. In this section, we'll introduce a Reflected Intelligence approach called collaborative filtering, which would be better described as "personalized relevance". Whereas popularized relevance determines which results are usually the most popular across many users, personalized relevance focuses on determining which items are most likely to be relevant for a specific user.

Collaborative filtering is the process of using observations about the preferences of some users to predict the preferences of other users. You've no doubt seen collaborative filtering in action many times before. It is the most popular type of algorithm used by recommendation engines, and it is the source of the common "users who liked this item also liked these items" recommendations lists that appear on many websites. Figure 4.6 demonstrates how collaborative filtering follows this same Reflected Intelligence feedback loop that we saw for signals boosting models.



User	Query
Alonzo	ipad
Elena	printer
Ming	ipad
...	...

User	Action	Document
Alonzo	click	doc22
Elena	click	doc17
Ming	click	doc12
Alonzo	purchase	doc22
Ming	click	doc22
Ming	purchase	doc22
Elena	click	doc2
...	...	...

User	Item	Weight
Alonzo	doc22	1.0
Alonzo	doc12	0.4
...	...	...
Ming	doc12	0.9
Ming	doc22	0.6
...	...	...

Figure 4.6 Collaborative filtering for user-to-item recommendations. Based upon his past behavior, our user (Alonzo) receives recommendations based upon items other users liked, where those users also interacted with the same items as Alonzo.

Like signals boosting, collaborative filtering is built leveraging a continuous feedback loop, where signals are collected, models are built based upon those signals, those models are then used to generate relevant matches, and the results of those interactions are then logged again as additional signals. Just as in other Reflected Intelligence models, the interactions between users and items are logged as signals, as also shown in Figure 4.6. Collaborative filtering approaches will typically generate a user-item interaction matrix mapping each user to each item (document), with the strength of relationship between each user and item being based upon the strength of the positive interactions (clicks, purchases, ratings, and so on).

If the interaction matrix is sufficiently populated, it is possible to infer recommendations from it for any particular user or item by directly looking up the other users who interacted with the same item and then boosting other items (similar to signals boosting) with which those users also interacted. If the user-item interaction matrix is too sparsely populated, however, then a *matrix factorization* approach will typically be applied.

Matrix factorization is the process of breaking the user-item interaction matrix into two matrices: one mapping users to latent "factors", and another mapping those latent factors to items. This is similar to the dimensionality reduction approach we mentioned in chapter 3, where we switched from mapping phrases associated with food items from exact keywords (a vector including an element for every word in the inverted index), to a much smaller number of dimensions that described the food items and allowed us to match the meaning without having to map to every specific item (keyword). This dimensionality reduction makes it possible to derive preferences of users for items, as well as similarity between items, based upon very sparse data.

In the context of matrix factorization for collaborative filtering, the latent factors represent attributes of our documents which are learned to be important indicators of shared interests across users. By matching other documents based upon these factors, we are using crowdsourcing to find other similar documents matching the same shared interests.

You can generate recommendations in other ways, which we'll explore later in the book, such as through content-based recommendations. Collaborative filtering is unique, however, in that it can learn preferences and tastes of users for other documents without having to know anything about the content of the documents - all decisions are made entirely by observing the interactions of users with content and determining the strength of the similarity based upon those observations. We will dive much more deeply into collaborative filtering, including code examples, when we discuss implementing Personalized Search in chapter 9.

As powerful as collaborative filtering can be for learning user interests and tastes based entirely on crowdsourced relevance, it unfortunately suffers from a major flaw known as the *cold start problem*.

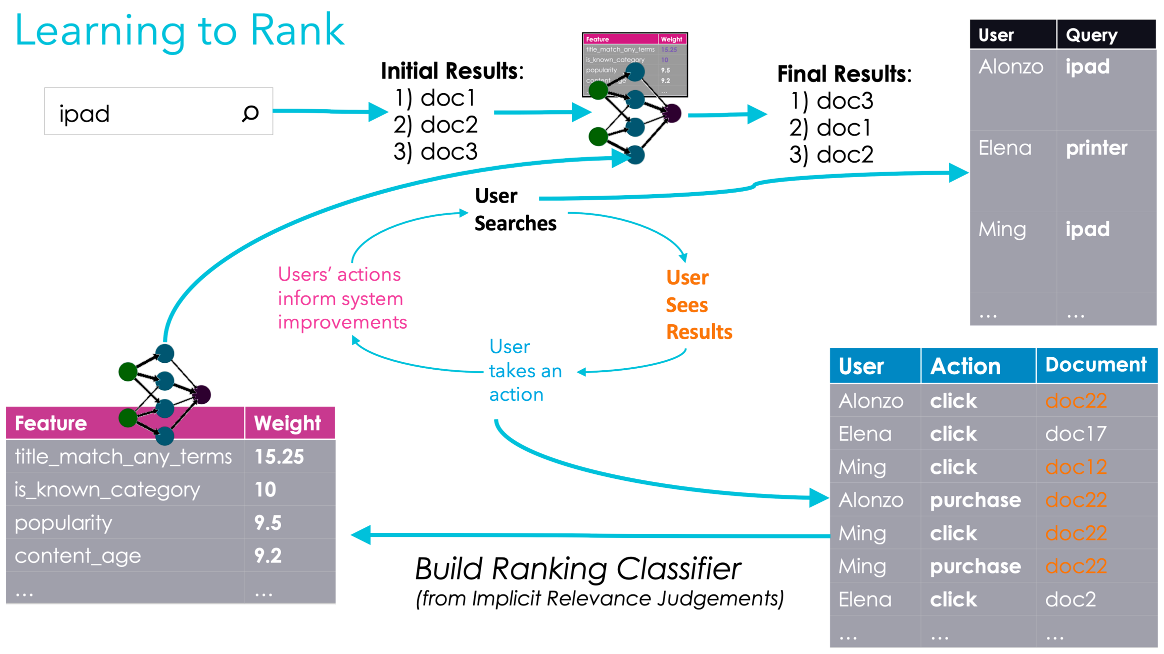
The cold start problem describes a scenario where returning results is dependent upon the existance of signals, but where new documents that have never generated signals are not getting displayed. This creates a catch-22 situation where new content is unlikely to be shown to users (a prerequisite for generating signals) because it has not yet generated any signals (which are required for the content to be shown). To some degree, signals boosting models demonstrate a similar problem, where documents that are already popular tend to receive higher boost, resulting in them getting even more signals, while documents that have never been seen get no signals boosting. This process creates a self-reinforcing cycle that can lead to a lack of diversity in search results.

Instead of only having popularized relevance or personalized relevance, which are dependent upon user interactions with specific items, it is generally also necessary to leverage a more generalized relevance model that can apply to all searches and documents and not just the most popular ones. This goes a long way toward solving the cold-start problem. In section 4.2.4, we explore how crowdsourced relevance can be generalized through a technique called Learning to Rank.

4.2.4 Generalized Relevance through Learning to Rank

Since signals boosting (popularized relevance, section 4.2.2) and collaborative filtering (personalized relevance, section 4.2.3) only work on documents which already have signals. This means that a substantial portion of queries and documents will not benefit from these approaches until they start receiving traffic and generating signals. This is where Learning to Rank proves valuable as a form of generalized relevance.

Learning to Rank, also known as Machine-learned Ranking, is the process of building and using a ranking classifier that can score how well any documents (even those never seen before) match any arbitrary query. You can think of the ranking classifier as a trained relevance model. Instead of manually tuning search boosts and other parameters, the Learning to Rank process trains a machine learning model that can learn the important features of your documents and then score search results appropriately. Figure 4.7 shows the general flow for rolling out Learning to Rank



User	Query
Alonzo	ipad
Elena	printer
Ming	ipad
…	…

User	Action	Document
Alonzo	click	doc22
Elena	click	doc17
Ming	click	doc12
Alonzo	purchase	doc22
Ming	click	doc22
Ming	purchase	doc22
Elena	click	doc2
…	…	…

Feature	Weight
title_match_any_terms	15.25
is_known_category	10
popularity	9.5
content_age	9.2
…	…

Figure 4.7 Learning to Rank (Generalized Relevance). A ranking classifier is built from user judgments about the known relevance of documents for each query (training set). That more advanced classifier model is then used to re-rank search results so that the top ranked documents are more relevant.

In a learning to rank system, the same high-level Reflected Intelligence process applies (refer to Figure 4.2) that we saw in signals boosting and collaborative filtering. The difference is that Learning to Rank can use relevance judgments lists (maps of queries to their ideal ranked set of documents) to automatically train a relevance model that can be applied generally to all queries. You'll see that the output of the "Build Ranking Classifier" step is a model of relevance features (`title_match_any_terms`, `is_known_caegory`, `popularity`, `content_age`), and that model gets deployed into the production search engine periodically to enhance search results. The features in a very simple machine-learned ranking model might be readable like this, but there is no requirement that a ranking classifier actually be interpretable and explainable like this, and many advanced, deep-learning-based ranking classifiers are not.

Additionally, notice in Figure 4.7 that the live user flow goes sequentially from searching on the word `ipad` to an initial set of search results, to running through the deployed learning to rank classifier, to returning a final set of re-sorted search results. This final set of results is re-ranked based upon the learned ranking function in the ranking classifier. Since the ranking classifier is typically much more intelligent and leverages more complicated ranking parameters than a traditional keyword-ranking relevance model, it is usually way too slow to use the ranking classifier to score all of the matching documents in the search engine. Instead, Learning to Rank will often still use an initial, faster ranking function (such as BM25) to find a the top-N documents (usually hundreds or thousands of documents) and then only run that subset of documents through the ranking classifier to get a better relevance ordering for the top results. It is possible to leverage the ranking classifier as the main relevance function instead of applying this re-ranking technique, but it is more common to see a re-ranking approach, since it is typically much faster while still yielding approximately the same resuls.

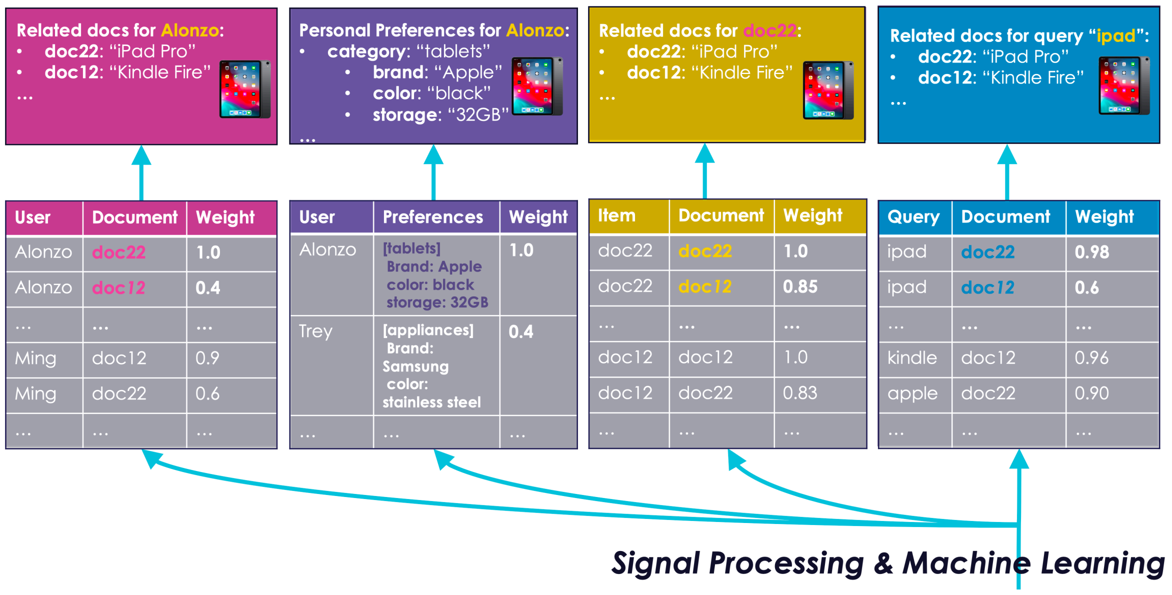
Learning to Rank can leverage either explicit relevance judgements (created manually by experts) or implicit judgements (derived from user signals), or some combination of the two. We’ll dive deep into examples of implementing Learning to Rank from both explicit and implicit judgements lists in chapter 10 and 11.

In section 4.2.5, we discuss a few additional examples of useful signals-based Reflected Intelligence models.

4.2.5 Other reflected intelligence models

In addition to diving deeper into Signals Boosting (chapter 8), Collaborative Filtering (chapter 9), and Learning to Rank (chapter 10), we explore many other kinds of reflected intelligence throughout this book. In chapter 6, we explore mining user queries to automatically learn domain-specific phrases, common misspellings, synonyms, and related terms, and in chapter 11 we explore automated ways of learning relevance judgments from users from their interactions so we can automatically generate training data for interesting machine learning approaches.

In general, every interaction between a user and some content creates a connection - an edge in a graph - that we can use to understand emerging relationships and derive deeper insights. Figure 4.8 demonstrates some of the various relationships we can learn by exploring this interaction graph.



Related docs for Alonzo:
- doc22: "iPad Pro"
- doc12: "Kindle Fire"

…

Personal Preferences for Alonzo:
- category: "tablets"
  - brand: "Apple"
  - color: "black"
  - storage: "32GB"

…

Related docs for doc22:
- doc22: "iPad Pro"
- doc12: "Kindle Fire"

…

Related docs for query "ipad":
- doc22: "iPad Pro"
- doc12: "Kindle Fire"

…

User	Document	Weight
Alonzo	doc22	1.0
Alonzo	doc12	0.4
…	…	…
Ming	doc12	0.9
Ming	doc22	0.6
…	…	…

User	Preferences	Weight
Alonzo	[tablets] Brand: Apple color: black storage: 32GB	1.0
Trey	[appliances] Brand: Samsung color: stainless steel	0.4
…	…	…

Item	Document	Weight
doc22	doc22	1.0
doc22	doc12	0.85
…	…	…
doc12	doc12	1.0
doc12	doc22	0.83
…	…	…

Query	Document	Weight
ipad	doc22	0.98
ipad	doc12	0.6
…	…	…
kindle	doc12	0.96
apple	doc22	0.90
…	…	…

Figure 4.8 Many Reflected Intelligence models. The first box represents user-to-item similarity for recommendations, the second shows learning of specific attribute-based preferences for a user's profile, the third shows learning item-to-item based similarity for recommendations, and the last shows learning query-to-item recommendations.

Figure 4.8 shows how the same incoming signals data can be processed differently through various signal aggregation and machine learning approaches to learn similarity between users and items, to learn specific attribute-based preferences to generate a profile of a user's interests, to learn similarity between items, and to generate query to item recommendations. We'll continue to explore these techniques in the chapters to come, but it is good to keep in mind that the signals data contains a treasure trove of potential insights that often provide just as much benefit as the documents from which their interactions are derived. Reflected Intelligence is the concept of learning from your users and mirroring that back what was learned, so it is not constrained to only the signals boosting, collaborative filtering, and the Learning to Rank techniques we've described. In section 4.2.6, we even even discuss a few ways to derive Reflected Intelligence from content instead of signals.

4.2.6 Crowdsourcing from content

While we typically think of crowdsourcing as asking people to provide input, we've seen thus far in this chapter that implicit feedback can often provide as much or even more value in aggregate across many user signals. While this chapter has been entirely focused on leveraging user signals to do this crowdsourcing, it's also important to point out that it is often possible to use content itself to crowdsource intelligence for your AI-powered search platform.

For example, if you are trying to figure out the general quality of your documents, you may be able to look at customer reviews to either get product rating or to see if the product has been reported as abusive or spam. If the customer has left comments, you may be able to run a *sentiment analysis* algorithm on the text, which is a type of algorithm that can determine if the comments are positive, neutral, or negative. Based upon the detected sentiment, you can then boost or penalize the source documents accordingly.

We mentioned that in chapter 6 we'll walk through how to mine user signals to automatically learn domain specific terminology (phrases, misspellings, synonyms, and so on). Just as we can take user queries and interactions to learn this terminology, we should also realize that our documents were themselves typically written by people, and that very similar kinds of relationships between terminology are therefore reflected in the written content, as well. We'll explore these content-based relationships further in the next chapter.

One of the most well-known search algorithms in existence is the Page Rank algorithm - the breakthrough algorithm which enabled Google to rise to prominence as the most relevant search engine and stay there for many years. Page rank essentially goes beyond the text in any given document and looks at the behavior of all other web page creators to see how they have linked to other documents within their own documents. By measuring the incoming and outgoing links, it is then possible to create a measure of "quality" of a website based upon the assumption that people are more likely to link to higher-quality, more authoritative sources, and that those higher-quality sources are less likely to link to lower quality sources. This idea of going beyond the content that exists within a single document and instead relating it to external content - whether it be direct user interactions (signals), comments and feedback posted on a forum, links between websites, or even just the usage of terminology in different, nuanced ways across other documents is incredibly powerful. The art and science of leveraging all the available information about your content and from your users, is key to building a highly-relevant AI-powered search engine. In chapter 5, we look at the concept of knowledge graphs and how we can leverage some of these relationships embedded in the links between documents to automatically learn domain understanding from our content.

4.3 Summary

- Content and signals are the two sources of "fuel" to power an AI-powered search engine, with signals being the primary source for crowdsourced relevance.
- Reflected Intelligence is the process of creating continuously-learning feedback loops that improve from each user interaction and reflect that learned intelligence back to automatically increase the relevance of future results.
- Signals boosting is a form of "popularized" relevance, which usually has the biggest impact on your highest-volume, most popular queries.
- Collaborative filtering is a form of "personalized" relevance, which is able to use patterns of user interaction with items to learn preferences of users or the strength of relationship between items, and to then recommend similar items based upon those learned relationships.
- Learning to Rank is a form of "generalized" relevance, and is the process of training a ranking classifier based upon relevance judgements lists (queries mapped to correctly-ranked lists of documents) that can be applied to rank all documents and avoid the cold start problem.
- Other kinds of Reflected Intelligence exist, including some techniques for leveraging content (instead of just signals) for crowdsourced relevance.

5 Knowledge graph learning

This chapter covers

- **Building and working with knowledge graphs**
- **Implementing open information extraction to generate knowledge graphs from text**
- **Using semantic knowledge graphs for query expansion and rewriting and arbitrary relationship discovery**
- **Interpreting documents with semantic knowledge graphs to power content-based document recommendations**

In the last chapter we focused on learning relationships between documents based upon crowdsourced interactions linking those documents. While these interactions were primarily user behavioral signals, we closed out the chapter also discussing links between documents that appear within the documents themselves - for example, leveraging hyperlinks between documents. In chapter 2, we also discussed how the textual content in the documents, rather than being "unstructured data", is really more like a giant graph of "hyperstructured data" containing a rich graph of semantic relationships connecting the many character sequences, terms, and phrases that exist across the fields within our collections of documents.

In this chapter, we will demonstrate how to leverage this giant graph of rich relationships within our content in order to better interpret your domain-specific terminology. We’ll accomplish this through the use of both traditional knowledge graphs - which enable explicit modeling of relationships within a domain, and through semantic knowledge graphs, which enable real-time inference of nuanced semantic relationships within a domain.

We’ll also play with several fun datasets in this chapter in order to show some variety in how knowledge graphs can be built and applied to improve query undertanding across different domains.

5.1 Working with knowledge graphs

In section 1.4, we introduced the idea of *knowledge graphs* and discussed how they relate to other types of knowledge models such as ontologies, taxonomies, synonyms, and alternative labels. Knowledge graphs, if you recall, integrate each of the other types of knowledge models together, so we are discussing them all collectively as we build out and refer to "knowledge graphs" in this chapter.

A knowledge graph (or any graph, for that matter), is represented through the concept of nodes and edges. Nodes are often also referred to as vertices, but we'll use the terminology "node" in this book. A *node* is an entity represented in the knowledge graph (such as a term, a person, place, thing, or concept), whereas an *edge* represents a relationship between two nodes. Figure 5.1 shows an example of a graph displaying nodes and edges.

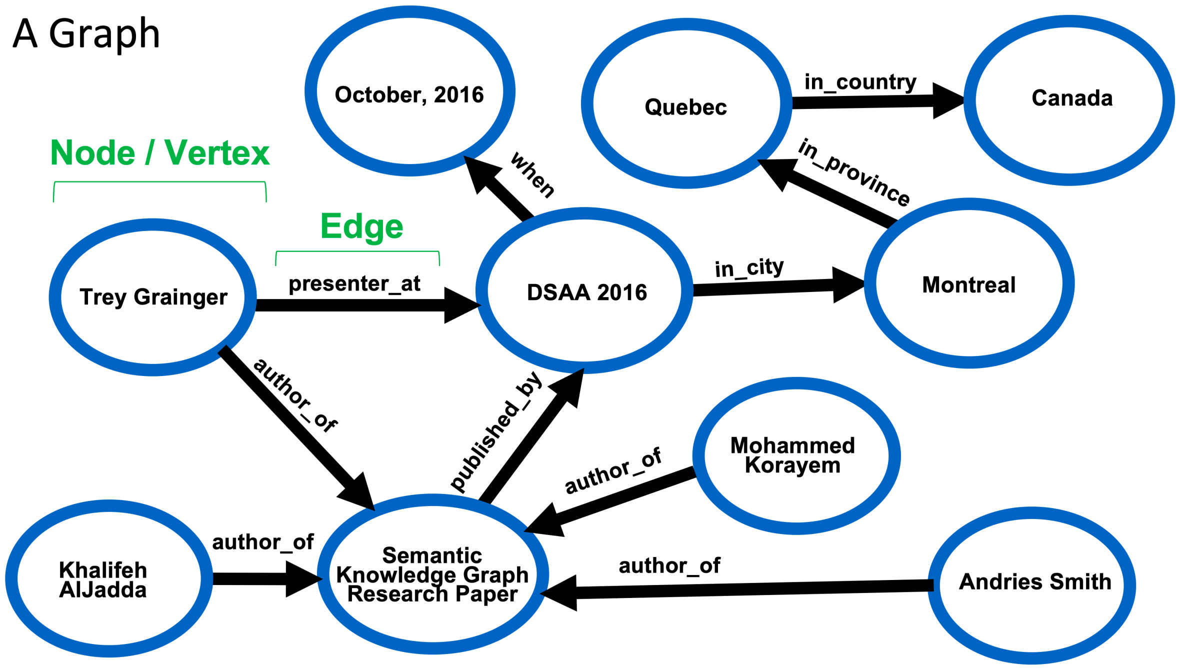


Figure 5.1 A Graph Structure. Graphs are composed of "nodes" (also known as "vertices") that represent entities, and edges which represent a relationship with another node. Graphs provide a way to model knowledge and infer new insights by traversing (or "following") the edges between nodes.

In this figure, you can see four nodes representing authors, one node representing a research paper they wrote together, one node representing the academic conference at which the paper was presented and published, and then nodes representing the city, province, country, and dates during which the conference was held. By traversing (or "following") the independent edges between nodes, you could for example, infer that one of the authors was in Canada in Montreal in October 2016. While any structure with nodes and edges like this is considered a graph, this particular graph represents factual knowledge and is therefore also considered a knowledge graph.

There are numerous ways to build and represent knowledge graphs, both through explicitly modeling data into a graph containing nodes and edges, as well as through dynamically materializing (discovering) nodes and edges from your data on the fly though what's known as a *semantic knowledge graph*. In this chapter, we'll walk through examples of building an explicit knowledge graph by hand, of auto-generating an explicit knowledge graph, and of leveraging a semantic knowledge graph that is already present within your search index.

To get started with knowledge graphs, you have essentially three options:

1. Build your own knowledge graph from scratch using a graph database (Neo4j, ArangoDB, etc.)
2. Plug in a pre-existing knowledge graph (ConceptNet, DBPedia, etc.)
3. Auto-generate a knowledge graph from your data, leveraging your content directly to extract knowledge.

Each approach has its strengths and weaknesses, and they are not necessarily mutually exclusive. If you are building a general knowledge search engine (such as a web search engine), then leveraging the second option of leveraging a pre-existing knowledge graph is a great place to start. If your search engine is more domain-specific, however, there is a good chance that your domain-specific entities and terminology will not be present, and that you will really need to create your own bespoke knowledge graph.

In this chapter we will focus primarily on the third option - auto-generating a knowledge graph from your content. The other techniques (manually generating a knowledge graph and leveraging a pre-existing knowledge graph) are already covered well in external materials - just do a web search for technologies like SPARQL, RDF Triples, and Apache Jena or for pre-existing knowledge graphs like DBPedia and Yago. That being said, you will ultimately need to be able to override your knowledge graph and add your own content, so we will include examples of how you can integrate both explicitly-defined (built with a specific list of pre-defined relationships) and implicitly-defined knowledge graphs (auto-generated relationships discovered dynamically from the data) into your search platform. We'll start off with an overview of how to integrate an explicitly-defined knowledge graph.

5.2 Building a knowledge graph explicitly into your search engine

Many organizations spend considerable resources building out knowledge graphs for their organizations, but then end up having trouble figuring out the best way to integrate those knowledge graphs within their search engines. We have fortunately chosen a search engine (Apache Solr) for our examples that has explicit graph traversal capabilities directly built in, so there is no need to pull in a new, external system to implement or traverse our knowledge graphs in this book.

While there may be some advantages to using a different tool such as Neo4J or ArangoDB, such as more sophisticated graph traversal semantics, using an external graph database like this makes coordinating requests, keeping data in sync, and infrastucture management more complex. Additionally, because some kinds of graph operations can only be done effectively in the search engine, like the semantic knowledge graph traversals we'll encounter shortly, leveraging the search engine as a unified platform search and knowledge graph capabilities enables us to combine the best of both worlds.

We will focus extensively on implementing a semantic search search system in chapter 7, including semantic query parsing, phrase extraction, misspelling detection, synonym expansion, and query rewriting, all of which will be modeled into an explicitly-built knowledge graph. Since the purpose of this chapter is to focus on knowledge graph *learning*, we'll save the discussion of explicit knowledge graph building and traversal until chapter 7 when we can tie everything from this chapter and chapter 6 (Learning Domain-specific Language) together into the appropriate knowledge graph structure.

5.3 Automatic extraction of knowledge graphs from content

While it is necessary to be able to manually add nodes and edges to your knowledge graphs, as we mentioned in the previous section, maintaining a large scale knowledge graph manually is very challenging. It requires substantial subject matter expertise, must be actively kept up to date with changing information, and is subject to the biases and errors of those maintaining it.

Open Information Extraction is an evolving area of Natural Language Processing (NLP) research. Open Information Extraction aims to extract facts directly out of your text content. This is often done using NLP libraries and language models to parse sentences and assess the dependency graph between them. A *dependency graph* is a break down of the parts of speech for each word and phrase in a sentence, along with an indication of which words refer to which other words. In this section, we'll use a language model and dependency graphs to extract two different types of relationships: arbitrary relationships and hyponym relationships.

5.3.1 Extracting arbitrary relationships from text

Given the "hyperstructured" nature of text and the rich relationships expressed within typical sentences and paragraphs, it stands to reason that we should be able to identify both the subjects of sentences and the ways in which they are related. In this section, we'll focus on extracting arbitrary relationships between the entities descibed within the sentences of our text content.

By analyzing the nouns and verbs within a sentence, it is often possible to infer a fact that is present in the sentence by mapping the subject, verb, and object of the sentence into an RDF triple. An *RDF triple* is a three part-datapoint representing a subject (starting node), relationship (edge), and object (ending node). For example, in the sentence "Colin attends Riverside High School", the verb "attends" can be extracted as relationship type connecting the subject ("Colin") with the object ("Riverside High School"). The RDF triple is therefore ("Colin", "attends", "Riverside High School").

Listing 5.1 walks through an example of using the Python-based SpaCy library to extract facts from text content. Spacy is leading Natural Language Processing library that ships with state of the art statistical neural network models for part of speech tagging, dependency parsing, text categorization, and named entity recognition.

Listing 5.1 Extracting arbitrary relationships using SpaCy NLP to relate nouns and verbs

```
text = """
Data Scientists build machine learning models. They also write code.
➥Companies employ Data Scientists.
Software Engineers also write code. Companies employ Software Engineers.
"""

def generate_graph(text):
    parsed_text = lang_model(text)
    parsed_text = resolve_coreferences(parsed_text)   ❶
    sentences = get_sentences(parsed_text)   ❷
    facts=list()
    for sentence in sentences:
        facts.extend(resolve_facts(sentence))   ❸
    return facts

graph = generate_graph(text)
for i in graph: print(i)
```

❶ "they" becomes "Data Scientists"

❷ "Data Scientists also write code." maps to ['nsubj, 'advmod', ROOT', 'dobj', 'punct']

❸ Fact are generated like: (subj:Companies, rel:employ, obj:"data scientists")

Results:

```
sentence: Data Scientists build machine learning models.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Data Scientists also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Companies employ Data Scientists.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Software Engineers also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
---------------------
sentence: Companies employ Software Engineers.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
---------------------
['Data Scientists', 'build', 'machine learning models']
['Data Scientists', 'write', 'code']
['Companies', 'employ', 'Data Scientists']
['Software Engineers', 'write', 'code']
['Companies', 'employ', 'Software Engineers']
```

As you can see, the example code has taken the text content, parsed it into sentences, and then determined the subjects, relationships, and objects within those sentences. Those subject/relationship/object tuples can then be saved off into a explicitly-built knowledge graph and traversed.

Figure 5.2 provides a visualization of this extracted graph.

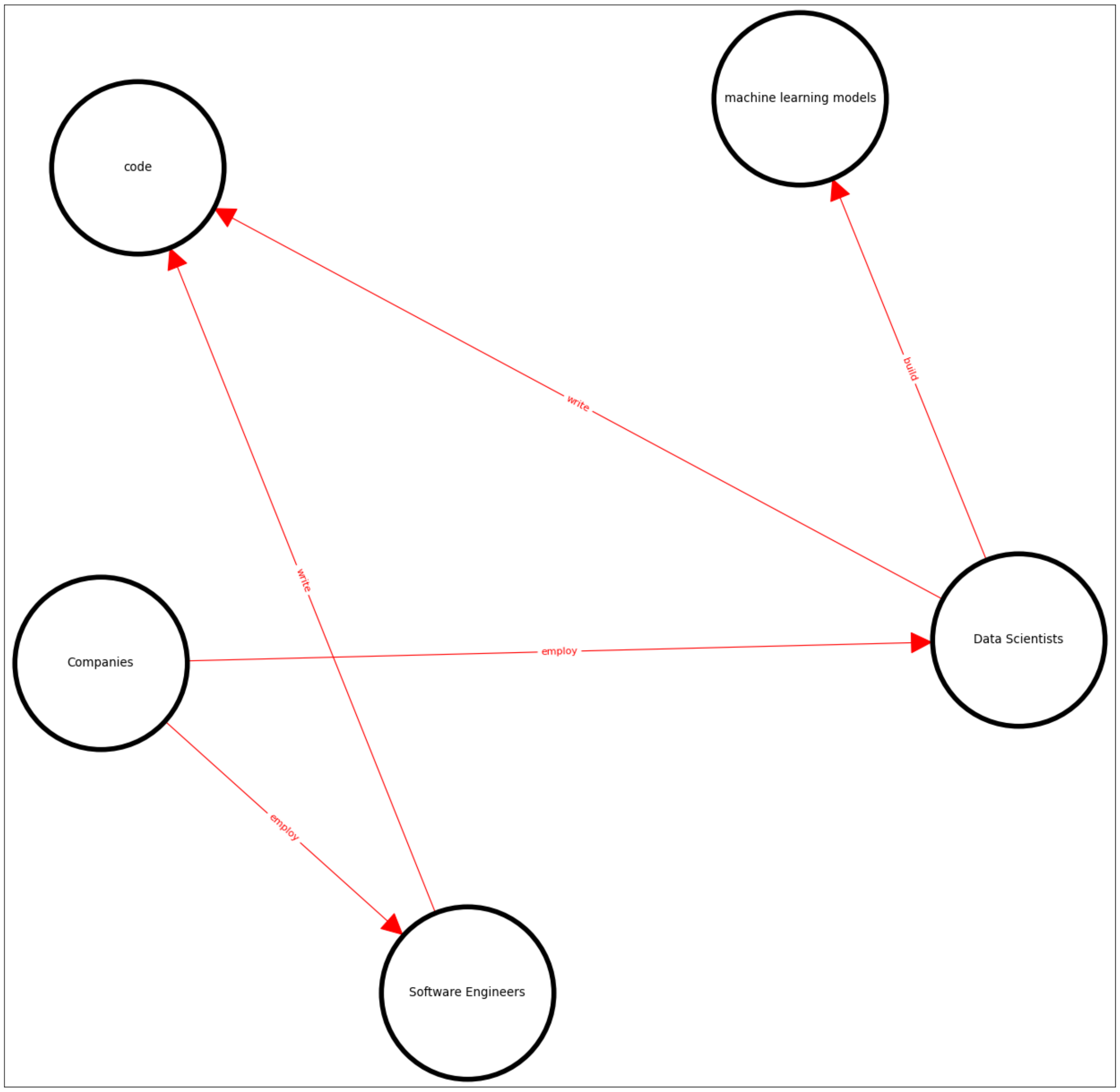


Figure 5.2 Extracted Knowledge Graph. The nodes and edges in this graph were automatically extracted from textual content based upon part of speech patterns.

The example in Figure 5.2 is very simple, and much more involved algorithms exist to extract facts from more sophisticated linguistic patterns. While we are using the SpaCy library in the code example, which leverages a deep-learning-based neural language model to detect parts of speech, phrases, and dependencies and co-references with the input text, the mechanism we leverged for then parsing those linguistic ouputs was more rule-based, following known semantic patterns within the English language.

Unfortunately, when parsing arbitrary verbs into relationships this way, the extracted relationships can become quite noisy. Since verbs conjugate differently, have synonyms, and have overlapping meanings, it is often necessary to prune, merge, and otherwise cleanup any list of arbitrary extracted relationships.

In contrast, some relationship types are much simpler, such as statistical relationships ("is related to") and hyponyms ("is a"). We'll spend the rest of the chapter covering these two special types, starting with hyponyms.

5.3.2 Extracting hyponyms from text

While it can be challenging mapping arbitrary verbs to clean lists of relationships within a knowledge graph, extracting hyponyms and hypernyms can be much easier. *Hyponyms* are entities that maintain an "is a" or "is instance of" relationship with a more general form of the entites, with the more general form being called a *hypernym*. For example, for the relationships between the terms *phillips head*, *screwdriver*, and *tool*, we would say that *phillips head* is a hyponym of *screwdriver*, that *tool* in a hypernym of *screwdriver*, and that *screwdriver* is both a hypernym of *phillips head* and a hyponym of *tool*.

One common and fairly accurate way to extract hyponym / hypernym relationships from text is through the use of Hearst patterns.[2]. These patterns describe common linguistic templates that reliably indicate the presence of hyponyms within sentences. Listing 5.2 demonstrates a few examples of such patterns.

Listing 5.2 Example Hearst Patterns which identify hyponym relationships to their hypernyms

```
simple_hearst_patterns = [
    (  '(NP_\\w+ (, )?such as (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '(such NP_\\w+ (, )?as (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '((NP_\\w+ ?(, )?)+(and |or )?other NP_\\w+)',
        'last'
    ),
    (
        '(NP_\\w+ (, )?include (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
    (
        '(NP_\\w+ (, )?especially (NP_\\w+ ?(, )?(and |or )?)+)',
        'first'
    ),
]
```

Each of these five simple patterns are represented as a Python tuple, with the first entry being a *regular expression*, and the second being a position within the pattern match (i.e. *first* or *last*). If you are unfamiliar with regular expressions, they provide a common and powerful syntax for pattern matching within strings. Anywhere you see the *NP_* characters, this stands for the existence of a *noun phrase* within a sentence, and the position specified in the second element of the tuple (*first* or *last*) indicates which noun phrase in the sentence represents the hypernym, with all other noun phrases matching the pattern considered the hyponyms.

In our example code in Listing 5.3, we run through almost 50 of these Hearst patterns to match many combinations of "is a" relationships within our content.

Listing 5.3 Extracting hyponym relationships using Hearst Patterns

```
text_content = """
Many data scientists have skills such as machine learning, python,
➥deep learning, apache spark, or collaborative filtering, among others
Job candidates most prefer job benefits such as commute time, company
➥culture, and compensation.
Google, Microsoft, or other tech companies might sponsor the conference.
Big cities, such as San Francisco, New York, and Miami appeal to new
➥graduates.
Many job roles, especially software engineering, registered nurse, and
➥DevOps Engineer are in high demand.
There are such job benefits as 401(k) matching, work from home, and
➥flexible spending accounts.
Programming languages, i.e. python, java, ruby and scala.
"""

h = HearstPatterns()
extracted_relationships = h.find_hyponyms(text_content)

facts = list()
for pair in extracted_relationships:
    facts.append([pair[0], "is_a", pair[1]])

print(*facts, sep="\n")
```

Response:

```
['machine learning', 'is_a', 'skill']
['python', 'is_a', 'skill']
['deep learning', 'is_a', 'skill']
['apache spark', 'is_a', 'skill']
['collaborative filtering', 'is_a', 'skill']
['commute time', 'is_a', 'job benefit']
['company culture', 'is_a', 'job benefit']
['compensation', 'is_a', 'job benefit']
['Google', 'is_a', 'tech company']
['Microsoft', 'is_a', 'tech company']
['San Francisco', 'is_a', 'big city']
['New York', 'is_a', 'big city']
['Miami', 'is_a', 'big city']
['software engineering', 'is_a', 'job role']
['registered nurse', 'is_a', 'job role']
['DevOps Engineer', 'is_a', 'job role']
['python', 'is_a', 'programming language']
['java', 'is_a', 'programming language']
['ruby', 'is_a', 'programming language']
['scala', 'is_a', 'programming language']
```

As you can see from Listing 5.3, by focusing on extracting a fixed type (and the most prevalent type of relationship - the "is_a" relationship), we are able to generate a nice, clean list of taxonomical relationships with the more specific term (the hyponym) pointing to the more general term (the hypernym) with an "is_a" edge. Figure 5.3 demonstrates this generated graph visually.

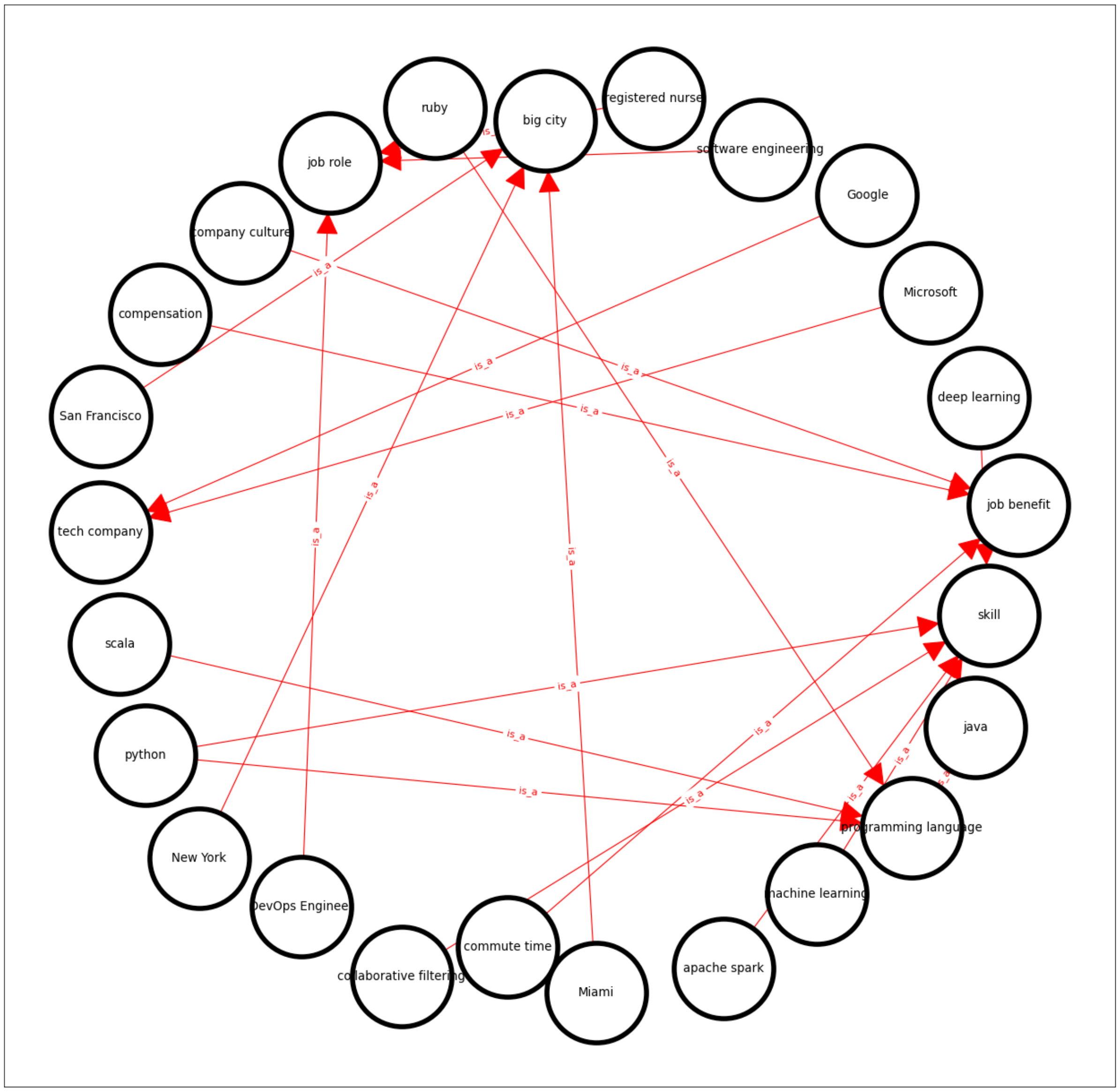


Figure 5.3 Knowledge graph derived from Hearst Patterns. We can see that all nodes are connected to other nodes through an "is_a" edge.

The inconsistency and noise that exists with arbitrary relationship extraction in the last section is now gone. We could still have ambiguity about the relationship between similar terms (for example, misspellings, alternative spellings, known phrases, or synonyms), but those are much easier to resolve. In fact, we'll spend the entire next chapter discussing how to learn this kind of domain-specific language from your signals and content in order to leverage it when interpreting incoming user queries.

While it can be useful to extract information from our text into an explicit knowledge graph for later traversal, the reality is that this kind of extraction is a lossy process, as the representation of the items gets disconnected from the originating context of those items within our content (the surrounding text and documents containing the text). In the next section, we’ll introduce an entirely different kind of knowledge graph - a semantic knowledge graph - that is optimized to enable real-time traversal and ranking of the relationships between terms and phrases within our content without having to be explititly built and without having to even separate terms from their original textual context.

5.4 Learning intent by traversing semantic knowledge graphs

In chapter 2, sections 2.1-2.2, we discussed the myth of text content being "unstructured data", and how in reality text documents represent hyper-structured data. We discussed the distributional hypothesis ("a word shall be known by the company it keeps"), and walked through how character sequences, terms, and phrases (or other arbitrary term sequences) can be thought of as fuzzy foreign keys relating similar concepts between documents. We also discussed how these links between documents can be thought of as edges in a giant graph of relationships, enabling us to learn the contextual meaning of the terms and entities present within our corpus of documents.

In this section, we introduce a semantic knowledge graph, a tool and technique which will enable us to traverse that giant graph of semantic relationships present within our documents.

5.4.1 What is a semantic knowledge graph?

A semantic knowledge graph is a "compact, auto-generated model for real-time traversal and ranking of any relationship within a domain".[3] Whereas a search engine typically finds and ranks documents relative to a query (a query to documents match), we can think of a semantic knowledge graph as a search engine that instead finds and ranks *terms* that best match a query.

For example, if we indexed a collection of documents about health topics and we searched for the pain reliever `advil`, instead of returning documents that contain the term `advil`, a semantic knowledge graph would automatically (with no manual list creation or data modeling required) return values like:

```
advil  0.71
motrin  0.60
aleve  0.47
ibuprofen  0.38
alleve  0.37
```

Results like these could be though of as "dynamic synonyms", but instead of the terms having the same meaning, they are more like conceptually-related terms. You could expand a traditional search engine query for "advil" to include these other terms, for example, in order to improve the recall of your search results or boost documents that conceptually match the meaning of `advil`, instead of just those containing the 5-letter string of `a-d-v-i-l`.

In addition to finding related terms, a semantic knowledge graph is able to traverse between fields in your inverted index ("find most related skills to this job title"), to traverse multiple levels deep ("find most related job titles to this query, and then find the most related skills for this query and each of those job titles"), and to use any arbitrary query you can send to the search engine as a node in the graph traversal to find semantically-related terms in any field.

The use cases for a semantic knowledge graph are diverse. It can be used for query expansion, for generating content-based recommendations, for query classification, for query disambiguation, for anomaly detection, for data cleansing, and for predictive analytics. We'll explore several of these throughout the remainder of this chapter, as soon as we get our datasets for testing our semantic knowledge graphs set up.

5.4.2 Indexing the datasets

A semantic knowledge graph works best on datasets where there is a decent overlap of terms being used together across documents. The more often two words tend to appear within documents the better we are able to determine whether those terms appear statistically more often we would expect.

While Wikipedia is often a good starting dataset for many use cases, since Wikipedia usually has a single page about a major topic that is supposed to be authoritative, there is actually not a ton of overlap across douments, making Wikipedia a less than optimal dataset for this use case.

That being said, any kind of website where users submit the content (questions, forum posts, job postings, reviews) will tend to make for an excellent dataset for a semantic knowledge graph use case.

For this chapter, we have selected two primary datasets, a *jobs* dataset (job board postings) and a series of StackExchange data dumps including posts from the following forums:

- health
- scifi
- devops
- outdoors
- travel
- cooking

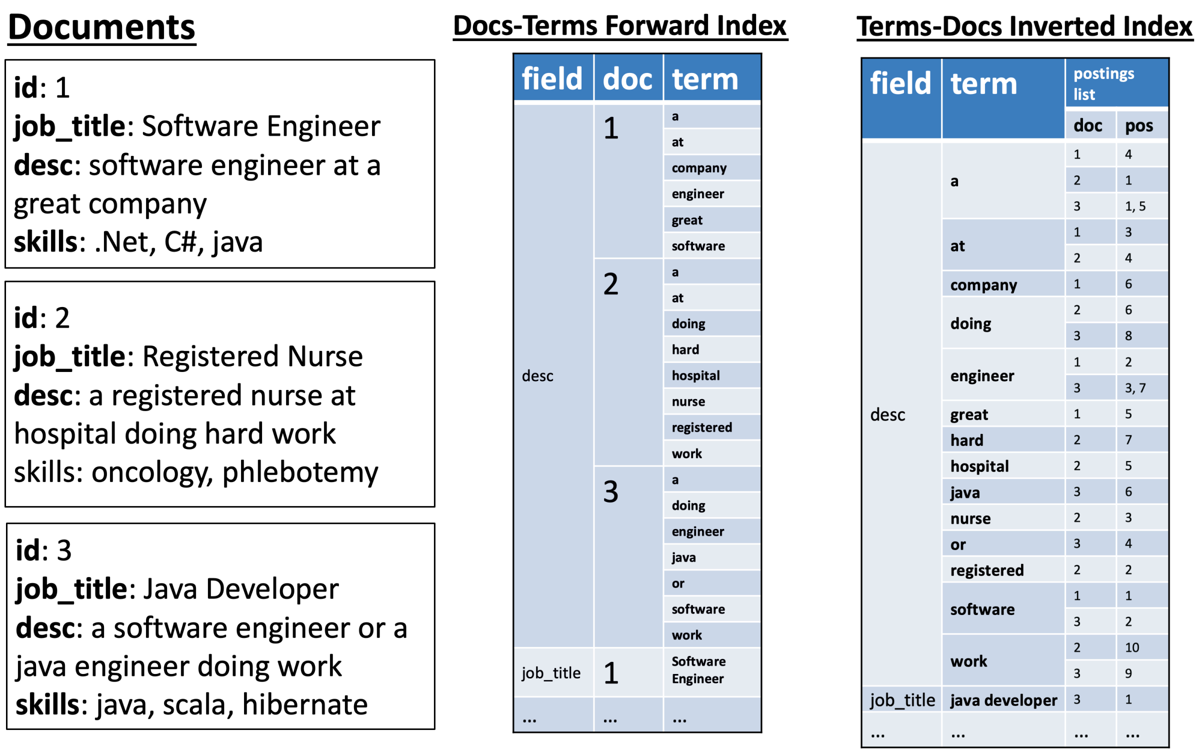
In order to index the datasets, please run through the Index Datasets Jupyter notebook prior to running any of the semantic knowledge graph examples

5.4.3 Structure of a semantic knowledge graph

In order to make best use of a semantic knowledge graph, it is useful to understand how the graph works based upon its underlying structure.

Unlike a traditional knowledge graph, which must be explicitly modeled into nodes and edges, a semantic knowledge graph is *materialized* from the underlying inverted index of your search engine. This means that all you have to do to use a semantic knowledge graph is to index documents into your search engine. No extra data modeling is required.

The inverted index and a corresponding forward index then serve as the underlying data structure which enables real-time traversal and ranking of any arbitrary semantic relationships present within your collection of documents. Figure 5.4 demonstrates how documents get mapped into both the forward index and the inverted index.



Documents

id: 1
job_title: Software Engineer
desc: software engineer at a great company
skills: .Net, C#, java

id: 2
job_title: Registered Nurse
desc: a registered nurse at hospital doing hard work
skills: oncology, phlebotemy

id: 3
job_title: Java Developer
desc: a software engineer or a java engineer doing work
skills: java, scala, hibernate

Docs-Terms Forward Index

field	doc	term
desc	1	a
		at
		company
		engineer
		great
		software
	2	a
		at
		doing
		hard
		hospital
		nurse
		registered
		work
	3	a
		doing
		engineer
		java
		or
		software
		work
job_title	1	Software Engineer
...	...	...

Terms-Docs Inverted Index

field	term	postings list	
		doc	pos
desc	a	1	4
		2	1
		3	1, 5
	at	1	3
		2	4
	company	1	6
	doing	2	6
		3	8
	engineer	1	2
		3	3, 7
	great	1	5
	hard	2	7
	hospital	2	5
	java	3	6
	nurse	2	3
	or	3	4
	registered	2	2
	software	1	1
		3	2
	work	2	10
		3	9
job_title	java developer	3	1
...	...	...	...

Figure 5.4 Inverted Index and Forward Index. Documents get mapped into an inverted index, which maps documents lists of terms, and a forward index, which maps terms back to lists of documents. Having the ability to map both directions will prove important for graph traversal and relationship discovery.

On the left of the figure, you can see three documents, each of which have a `job_title` field, a `desc` field, and a `skills` field. The right side of the figure shows how these documents are mapped into your search engine. We see that the inverted index maps each field to a list of terms, and then maps each term to a postings list containing a list of documents (along with positions in the documents, as well as some other data not included in the figure). This makes it quick and efficient to look up any term in any field and find the set of all documents containing that term.

In addition to the well-known inverted index, however, you will also see the less-well known *forward index* in the center of Figure 5.4. A forward index can be thought of as an *uninverted index*: for each field, it maps each document to a list of terms contained within that document. A forward index is what search engines use to generate facets on search results. In Lucene-based search engines like Solr and Elasticsearch, a forward index is usually generated at index time for a field by enabling a feature called `docValues` on the field. Alternatively, Apache Solr also allows you to generate the same forward index by "uninverting" the inverted index in memory on the fly at query time, enabling faceting even on fields for which `docValues` weren't added to the index.

If you have the ability to search for any arbitrary queries and find sets of documents through an inverted index, and then you also have the ability to take arbitrary sets of documents and look up terms in those documents, this means that by doing two traversals (terms to documents to terms) that you can find all of the related terms that also appear together in documents containing the original query. Figure 5.5 demonstrates how such a traversal would occur, including a data structure view, a set theory view, and a graph view.

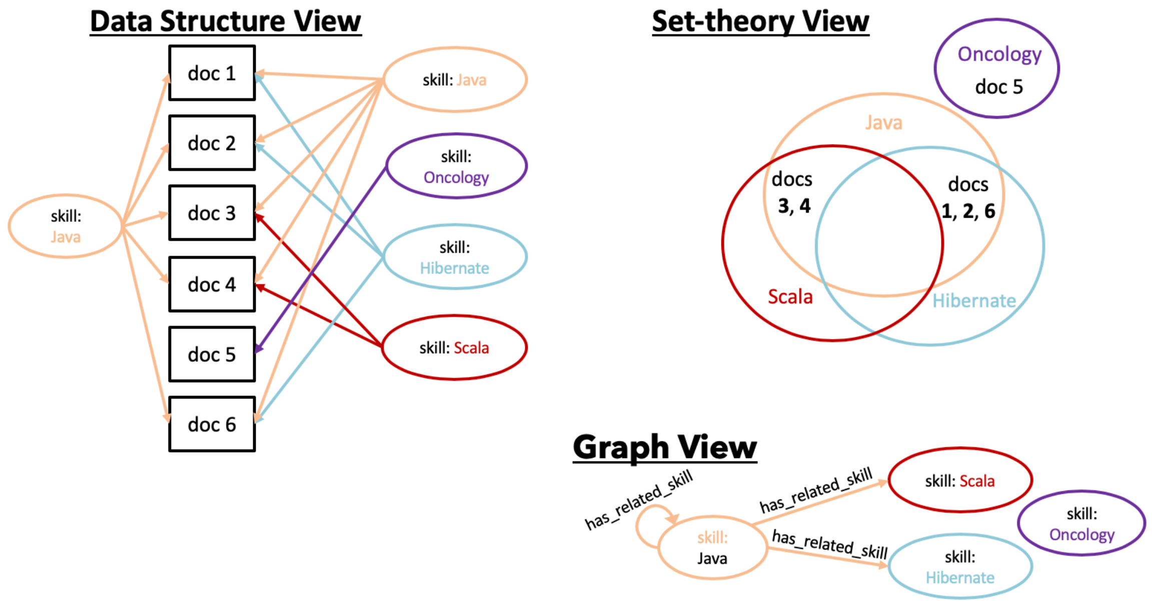


Figure 5.5 Three Representations of a semantic knowledge graph. The Data Structure View shows terms mapped to sets of documents, the Set-theory View shows how the intersection of sets of documents actually forms the relationship between them, and the Graph View, showing the nodes and edges.

In the data structure view, which represents our inverted and forward indices, we see how terms are related to documents based upon whether they appear within them. Those "links" are ultimately only present if there is an intersection between the docs that any two nodes (terms) appear within in the set-theory view. The graph view, finally, demonstrates a third view into the same underlying data structures, but in this case we see nodes (instead of document sets) and edges (instead of intersecting document sets). Essentially, our semantic knowledge graph exists as an abstraction on top of the inverted index that is already built and updated anytime our search engine indexes content.

We typically think of the primary function of search engines being to accept a query, to find matching documents, and to return those documents in a relevance-ranked order. We devoted all of chapter 3 to discussing this process, walking through matching (sections 3.2.4 - 3.2.6), TF*IDF ranking (section 3.1), and the commonly-used BM25 ranking function (section 3.2). However, with a semantic knowledge graph we focus on matching and ranking related *terms*, as opposed to related documents.

Any arbitrary query (anything you can express in Solr syntax) can be a node in your graph, and you can traverse from that node to any other term (or arbitrary query) in any field. Additionally, since each traversal of an edge between two nodes leverages both an inverted index (terms to docs) and a forward index (docs to terms), it is trivial to chain these traversals together into a multi-level graph traversal, as shown in Figue 5.6.

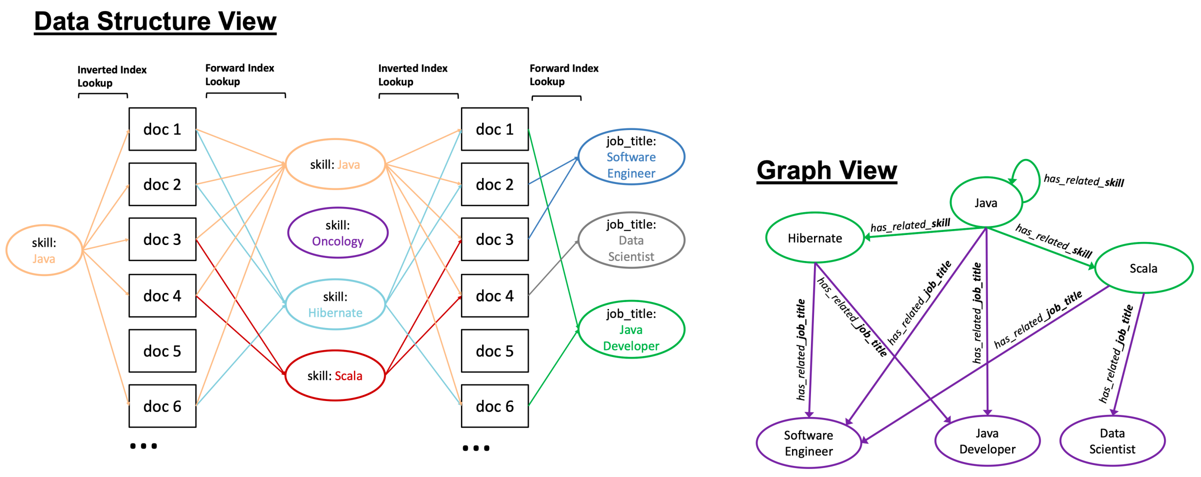


Figure 5.6 Multi-level Graph Traversal. In the data structure view, we see two traversals, through the inverted index and then the forward index each time. In the graph structure view, we see the corresponding two-level traversal from skills, to skills, to job titles.

In the figure, you see a traversal from a skill (*java*) to a layer of other skills (*java*, *oncology*, *hibernate*, and *scala*), to a layer of job titles (*software engineer*, *data scientist*, *java developer*). You an see that not all nodes are connected - the node for *oncology*, for example, does not appear in the graph traversal view because none of the original nodes can connect to it through any edges since there are no overlapping documents.

Given that not all possible nodes are going to be relevant for any given traversal, it is also important for semantic knowledge graphs to be able score and assign a weight to the relationships between nodes so that those edges can be prioritized during any graph traversal. We will cover the scoring and assignment of weights to edged in the next section.

5.4.4 Calculating edge weights to score relatedness of nodes

Given that the primary function of a semantic knowledge graph is to discover and score the strength of the semantic relationship between nodes (where the nodes can represent any word, phrase, or arbitrary query), the ability to generate a semantic similarity score is critical. But what does "semantic similarity" actually mean?

If you recall the Distributional Hypothesis, which was introduced in chapter 2, it says that words that appear together in the same contexts and with similar distributions tend to share similar meanings. Intuitively this makes sense - the terms "pain" or "swelling" will be more likely to occur in documents that also mention "advil", "ibuprofen", or "ice pack" than in some random documents. Interestingly, though, "ice pack" may also occur in documents containing terms like "cooler", "road trip", or "cold", whereas "advil" and "ibuprofen" likely would not.

These examples show words with similar meanings with their contexts, but let’s also consider words like "a", "the", "of", "and", "if", "they", and countless other very common/stop words. Indeed, these words will also appear heavily within the same contexts of "pain", "swelling", "advil", "ibuprofen", or any of the other words we examined. This points to the second part of the distributional hypothesis - that the words must also occur with similar distributions. In essence, this means that given some number of documents containing a first term, any second term tends to be semantically similar to the first term if it co-occurs in the same documents as the first term more often that it co-occurs in documents with other random terms.

Practically, since "the" or "a" tend to co-occur commonly with almost all other terms, they are not considered semantically similar to those terms even though their level of co-occurrence is high. For terms like "pain" and "ibuprofen", however, they occur together statistically way more often than either term appears with random other terms, and therefore they are considered semantically similar.

One easy way to measure this semantic relatedness is through the relatedness calculation (z) that follows:

```
z = (countFg(x) - t\otalDocsFG * probBG(x)) / sqrt(t\otalDocsFG * probBG(x)
➥* (1 - probBG(x))
```

This relatedness calculation (conceptually similar to a "z-score" in a normal distribution) relies on the concept of a "foreground" set of documents and a "background" set of documents, and enables the distribution of the term "*x*" to be statistically compared between the two sets. For example, if the foreground set was all documents matching the query "pain", and the background set was all documents, then the relatedness of the word "advil" would be a measure of how much more often "advil" occurs in documents also containing the word "pain" (foreground set) versus in any random document (background set).

If two terms are highly related, their relatedness will be a positive number approaching `1.0`. If the terms are highly unrelated (they tend to occur in divergent domains only) then the score would be closer to `-1.0`. Finally, term that just aren't semantically related at all - like stopwords, will tend to have a relatedness score close to zero.

Apache Solr has semantic knowledge graph capabilities built directly into its faceting API. Faceting provides the ability to traverse from terms to sets of documents to terms, and a *relatedness* aggregation function implements the semantic similarity calculation we just described. Listing 5.4 demonstrates a search for "advil" within a dataset of Stack Exchange Health forum discussions, and enables us to find the most semantically-related terms.

Listing 5.4 Discovering related nodes. We traverse from our original query of "advil" to the most semantically-related terms (nodes) in the "body" field of our documents.

```
collection="health"

request = {
    "params": {
        "qf": "title body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "query": "advil",
    "facet": {
        "body": {
            "type": "terms",
            "field": "body",
            "sort": { "relatedness": "desc"},
            "mincount": 2,
            "limit": 8,
            "facet": {
                "relatedness": {
                    "type": "func",
                    "func": "relatedness($fore,$back)"
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()

for bucket in search_results["facets"]["body"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["relatedness"][
  ➥"relatedness"]))
```

Results:

```
advil  0.70986
motrin  0.59897
aleve  0.4662
ibuprofen  0.38264
alleve  0.36649
tylenol  0.33048
naproxen  0.31226
acetaminophen  0.17706
```

As you can see, out of all terms within any of the text of the forum posts in the Stackexchange Health dataset, the ranked order of the most semantically related terms to "advil" was a nice, clean list of other pain killers which are most similar to "advil". This is the magic of leveraging the distributional hypothesis to discover and rank terms by semantic similarity - it provides us with the ability to automatically discover relationships on the fly that can be used to further improve our understanding of incoming queries. In the next section, we’ll discuss how we can apply this understanding to improve query relevance.

5.4.5 Using semantic knowledge graphs for query expansion

Matching and ranking on only the keywords entered during a search does not always provide sufficient context to find and rank the best results. In these cases, you can significantly improve the quality of search results by expanding or otherwise augmenting queries to include conceptually-related terms. In this section, we’ll walk through how to generate these related terms, and we’ll demonstrate several strategies for applying those term to enhance they quality of your search results.

Given the ability to start with any keyword or query and to find highly-related other terms in any field, one obvious use case for a semantic knowledge graph is for dynamically expanding queries to include related terms. This enables documents to match which do not necessarily contain the exact keywords entered by the user but which do contain other terms which carry a very similar meaning.

For example, instead of a user’s query for "advil" (from our last example) only matching documents with the string "advil" in them, we could use the semantic knowledge graph to automatically discover related terms and then match additional documents and boost the relevance score based upon the relatedness score of each expanded term. The final query submitted to the search engine, instead of being `advil`, might instead be something like `advil OR motrin^0.59897 OR aleve^0.4662 OR ibuprofen^.3824 OR ….`

Let’s walk through the steps needed to implement this kind of query expansion, leveraging a dataset from a different domain this time (our scifi dataset). Listing 5.5 provides the first step in this process, running a search for the obscure search term "vibranium" on our scifi dataset.

Listing 5.5 Discovering context for unknown terms. In this case, an obscure query for "vibranium" brings back useful context for anyone unfamiliar with the fictional term.

```
query = "vibranium"

collection = "stackexchange"

request = {
    "query": query,
    "params": {
        "qf": "title body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "facet": {
        "body": {
            "type": "terms",
            "field": "body",
            "sort": { "relatedness": "desc"},
            "mincount": 2,
            "limit": 8,
            "facet": {
                "relatedness": {
                    "type": "func",
                    "func": "relatedness($fore,$back)"
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()

for bucket in search_results["facets"]["body"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["relatedness"][
  ➥"relatedness"]))
```

Response:

```
vibranium  0.92227
wakandan  0.75429
wakanda  0.75295
adamantium  0.7447
panther's  0.69835
klaue  0.68083
klaw  0.65195
panther  0.65169
```

For anyone unfamiliar with the term "vibranium", it is a strong, fictional metal that exists in Marvel comic books and movies (best popularized through the 2018 Hollywood hit *Black Panther*). The most related terms that came back were related to "Wakanda", the fictional country from which vibranium originates, "adamantium", another strong (fictional) metal from Marvel comics, and the words "panther" (from the name Black Panther) and the name Klaue and alternative spelling Klaw, a character in the Black Panther comic books and movie that is heavily associated with the metal vibranium.

You may or may not have any familiarity with vibranium or any of these related pieces of information - and the fact that you don't need to is exactly why an auto-generated knowledge graph is so useful. By leveraging a semantic knowledge graph and expanding your query to include additional related context, you can drastically improve the recall of your search requests, and by boosting results that best match your query conceptually (as opposed to just the text), you may also be able to improve the precision of your top-ranked search results.

Listing 5.6 demonstrates an example of translating this original query along with the semantic knowledge graph output into an expanded query.

Listing 5.6 Generation of an expanded query from the related nodes returned from the semantic knowledge graph.

```
query_expansion = ""

terms = search_results["facets"]["body"]["buckets"]
for bucket in search_results["facets"]["body"]["buckets"]:
  term = bucket["val"]
  boost = bucket["relatedness"]["relatedness"]
  if len(query_expansion) > 0:
    query_expansion += " "
  query_expansion += " " + term + "^" + str(boost)

expanded_query = query + "^5" + query_expansion

print("Expanded Query:\n" + expanded_query)
```

Results:

```
Expanded Query:
vibranium^5 vibranium^0.92228  wakandan^0.75429  wakanda^0.75295
➥adamantium^0.7447  panther's^0.69835  klaue^0.68083  klaw^0.65195
➥panther^0.65169
```

In this case, we are doing a simple Boolean OR search for any of the keywords related to the original query (`vibranium`), boosting the original query term's weight by a factor of 5x and weighting each subsequent term's impact on the relevance score based upon it's semantic similarity score. The choice to boost the original term by 5x is arbitrary - you can choose any value here to assign a relative relevance boost versus the other (expanded) terms.

You might also notice that the term vibranium appears twice - first as the original term and then again as an expanded term (since the term is *also* the most semantically similar to itself). This will almost always be the case if you are searching for individual keywords, but since your query might have phrases or other constructs that make the original query different than the terms that actually come back (if any), it is usually a good idea to still include the original query as part of the expanded/rewritten query so users don't get frustrated by it being ignored.

While the prior expanded query should rank results pretty well (prioritizing documents matching multiple related terms), it is also heavily focused on recall (expanding to include anything relevant) as opposed to precision (ensuring everything included is relevant). There are many different ways to construct an augmented query, depending on yout primary goals.

Rewritten queries can perform a simple expansion, require a minimum percentage or number of terms to match, require specific terms like the original query to match, or even just change the ranking of the same initial results set. Listing 5.7 demonstrates several examples, leveraging minimum match thresholds and percentages, which can tilt the scale between precision and recall as needed.

Listing 5.7 Different Query Augmentation Strategies.

```
simple_expansion = 'q={!edismax qf="title body" mm="0%"}' + query + " " +
➥query_expansion
increase_conceptual_precision = 'q={!edismax qf="title body" mm="30%"}' +
➥query + " " + query_expansion
increase_precision_reduce_recall = 'q={!edismax qf="title body" mm="2"}' +
➥query + " AND " + ( query_expansion + )
same_results_better_ranking = 'q={!edismax qf="title body" mm="2"}' + query
➥\ + "&boost=" + "query($expanded_query)&expanded_query=" +
➥query_expansion

print("Simple Query Expansion:\n" + simple_expansion)
print("\nIncreased Precision, Reduced Recall Query:\n" +
➥increase_conceptual_precision)
print("\nIncreased Precision, No Reduction in Recall:\n" +
➥increase_precision_reduce_recall)
print("\nSlightly Increased Recall Query:\n" + slightly_increased_precision)
print("\nSame Results, Better Conceptual Ranking:\n" +
➥same_results_better_ranking)
```

The final queries for each of these different query expansions techniques are as follows:

Simple Query Expansion:

```
q={!edismax qf="title body" mm="0%"}vibranium vibranium^0.92227
➥wakandan^0.75429  wakanda^0.75295  adamantium^0.7447  panther's^0.69835
➥klaue^0.68083  klaw^0.65195  panther^0.65169
```

This simple query expansion is the same as previously described, matching any documents containing either the original query or any of the semantically-related terms.

Increased Precision, Reduced Recall Query:

```
q={!edismax qf="title body" mm="30%"}vibranium vibranium^0.92227
➥wakandan^0.75429  wakanda^0.75295  adamantium^0.7447  panther's^0.69835
➥klaue^0.68083  klaw^0.65195  panther^0.65169
```

This example specifies a "minimum match" threshold of 30% (`mm=30%`), meaning that in order for a document to match it must contain at least 30% (rounded down) of the terms in the query.

Increased Precision, No Reduction in Recall:

```
q={!edismax qf="title body" mm="2"}vibranium AND (vibranium^0.92227
➥wakandan^0.75429  wakanda^0.75295  adamantium^0.7447  panther's^0.69835
➥klaue^0.68083  klaw^0.65195  panther^0.65169)
```

This query requires the original query term (`vibranium`) to match, and *also* requires that at least two terms match (`vibranium` plus one more term). This means that a document must contain an additional terms beyond just the original query, but must also still match the original query.

Slightly Increased Recall Query:

```
q={!edismax qf="title body" mm="2"}vibranium vibranium^0.92227
➥wakandan^0.75429  wakanda^0.75295  adamantium^0.7447  panther's^0.69835
➥klaue^0.68083  klaw^0.65195  panther^0.65169
```

This query requires two terms to match, but does not explicitly require the original query, so it can expand to other documents which are conceptually similar but don't necessarily have to contain the original query term. Since the term `vibranium` is repeated twice, any document containing just the word `vibranium` will also match.

Same Results, Better Conceptual Ranking:

```
q={!edismax qf="title body" mm="2"}vibranium
&boost=query($expanded_query)
&expanded_query=vibranium^5 vibranium^0.92227  wakandan^0.75429
➥wakanda^0.75295  adamantium^0.7447  panther's^0.69835  klaue^0.68083
➥klaw^0.65195  panther^0.65169
```

This final query returns the exact same documents as the original query for `vibranium`, but it ranks them differently according to how well they match the semantically-similar terms from the knowledge graph. This ensures the keyword exists in all matched documents and that all documents containing the user's query are returned, but it then enables the ranking to be greatly improved to better understand the domain-specific context for the term to boost more relevant documents.

Of course, there are an unlimited number of possible query permutations you can explore when rewriting your query to include enhanced semantic context, but the above examples should provide a good sense of the kinds of options available and tradeoffs you'll want to consider.

5.4.6 Using semantic knowledge graphs for content-based recommendations

In the last section, we explored how to augment queries by discovering and leveraging related nodes from the semantic knowledge graph, including multiple ways of structuring rewritten queries to optimize for precision, recall, or even improved conceptual ranking over the same results. In addition to expanding queries with semantically-related terms, it is also possible to use the semantic knowledge graph to generate content-based recommendations by translating document into queries based upon semantic similarity of the terms within the documents.

Since nodes in the semantic knowledge graph can represent any arbitrary query, we can take the content from documents (individual terms, phrases, or other values) and model them as arbitrary nodes to be scored relative to some known context about the document. This means we can take dozens or hundreds of terms from a document, score them all relative to the topic of the document, and then take the top most semantically-similar terms and use them to generate a query best representing the nuanced, contextual meaning of the document.

Listing 5.8 walks through an example of translating a document that is classified as a "star wars" document and ranking all the terms in the document relative to that topic.

Listing 5.8 Content-based Recommendations. We can pass the terms and phrases of a document to the semantic knowledge graph, score their semantic similarity to the topic of the document, and generate a query representing the most semantically-important elements of the document.

```
import collections
from mergedeep import merge

print(solr_url)
collection="stackexchange"
classification="star wars"

document="""this doc contains the words luke, magneto, cyclops, darth vader,
           princess leia, wolverine, apple, banana, galaxy, force, blaster,
           and chloe."""

#run an entity extractor to parse out keywords to score
parsed_document = ["this", "doc", "contains", "the", "words", "luke", \
                   "magneto", "cyclops", "darth vader", "princess leia", \
                   "wolverine", "apple", "banana", "galaxy", "force", \
                   "blaster", "and", "chloe"]

request = {"query": classification, "params": {}, "facet": {}}

i=0
for term in parsed_document:
    i+=1
    key = "t" + str(i)
    key2 = "${" + key + "}"
    request["params"][key] = term
    request["facet"][key2] = {
        "type": "query",
        "q": "{!edismax qf=${qf} v=" + key2 + "}",
        "facet": {"stats": "${relatedness_func}"}
    }


print(json.dumps(request,indent="  "))

full_request = merge(request_template, request)
search_results = requests.post(solr_url + collection + "/select",
➥json=full_request).json()

def parse_scores(search_results):
    results = collections.OrderedDict()
    for key in search_results["facets"]:
        if key != "count" and key != "" and "stats" in search_results[
        ➥"facets"][key]:
            relatedness = search_results["facets"][key]["stats"][
            ➥"relatedness"]
            results[key] = relatedness
    return list(reversed(sorted(results.items(), key=lambda kv: kv[1])))

scored_terms = parse_scores(search_results)

for scored_term in scored_terms:
    print (scored_term)
```

Generated Knowledge Graph Lookup:

```
{
  "query": "star wars",
  "params": {
    "t1": "this",
    "t2": "doc",
    "t3": "contains",
    "t4": "the",
    "t5": "words",
    "t6": "luke",
    "t7": "magneto",
    "t8": "cyclops",
    "t9": "darth vader",
    "t10": "princess leia",
    "t11": "wolverine",
    "t12": "apple",
    "t13": "banana",
    "t14": "galaxy",
    "t15": "force",
    "t16": "blaster",
    "t17": "and",
    "t18": "chloe"
  },
  "facet": {
    "${t1}": {
      "type": "query",
      "q": "{!edismax qf=${qf} v=${t1}}",
      "facet": {
        "stats": "${relatedness_func}"
      }
    },
    ...
    "${t18}": {
      "type": "query",
      "q": "{!edismax qf=${qf} v=${t18}}",
      "facet": {
        "stats": "${relatedness_func}"
      }
    }
  }
}
```

Scored Nodes:

```
('luke', 0.66366)
('darth vader', 0.6311)
('force', 0.59269)
('galaxy', 0.45858)
('blaster', 0.39121)
('princess leia', 0.25119)
('this', 0.13569)
('the', 0.12405)
('words', 0.08457)
('and', 0.07755)
('contains', 0.04734)
('banana', -0.00128)
('doc', -0.00185)
('cyclops', -0.00418)
('wolverine', -0.0103)
('magneto', -0.01112)
('apple', -0.01861)
```

From these results, you can see a list of terms from the document that is nicely ordered based upon semantic similarity to the topic of "star wars". Terms with lower scores will have no relatedness or a negative relatedness with the specified topic. If we filter to terms above a `0.25` positive relatedness, as performed in Listing 5.9, we get a very clean list of relavant terms from the document.

Listing 5.9 Mapping scored phrases from the document to a query, filtering out low relatedness scores.

```
rec_query = ""

for scored_term in scored_terms:
  term = scored_term[0]
  boost = scored_term[1]
  if len(rec_query) > 0:
    rec_query += " "
  if boost > 0.25:
    rec_query += term + "^" + str(boost)

print("Expanded Query:\n" + rec_query)
```

Expanded Query:

```
luke^0.66366 "darth vader"^0.6311 force^0.59269 galaxy^0.45858
➥blaster^0.39121 "princess leia"^0.25119
```

Listing 5.10 demonstrates the last step in this process, actually running the search to return the top ranked documents most semantically-similar to the original document.

Listing 5.10 Content-based Recommendations Request passing in the scored terms as the query.

```
import collections

collection="stackexchange"

request = {
    "params": {
        "qf": "title body",
        "defType": "edismax",
        "rows": 5,
        "echoParams": "none",
        "omitHeader": "true",
        "mm": "0",
        "fl": "title",
        "fq": "title:[* TO *]" #only show docs with titles to make the example readable
    },
    "query": rec_query
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()
print(json.dumps(search_results, indent="  "))
```

Response:

```
{
  "response": {
    "numFound": 2511,
    "start": 0,
    "docs": [
      {
        "title": "Did Luke know the &quot;Chosen One&quot; prophecy?"
      },
      {
        "title": "Why couldn't Snoke or Kylo Ren trace Luke using the Force?"
      },
      {
        "title": "Was Darth Vader at his strongest during Episode III?"
      },
      {
        "title": "Was/is Darth Vader the most powerful force user?"
      },
      {
        "title": "Who/what exactly does Darth Vader believe taught Luke
        ➥between the events of “The Empire Strikes Back” and “Return of
        ➥the Jedi?”"
      }
    ]
  }
}
```

As you can see, we have just created a content-based recommendations algorithm. We discussed the idea of leveraging user behavioral signals (searches, clicks, etc.) to generate recommendations (collaborative filtering) in chapter 4, but we will not always have sufficient signals to rely solely on signals-based recommendations approaches. As a result, having the ability to take a document and find other similar documents based upon content of the starting document provides us with an excellent additional tool with which we can provide context- and domain-aware recommendations without relying on user signals.

While the example in this section generated a content-based recommendations query based upon terms actually in the starting document, it is worth keeping in mind that the semantic knowledge graph is not restricted to using the terms passed in. You could add an extra level to the traversal to find additional terms that are semantically-related to the terms in the original document, but not actually contained within it. This can be particularly useful for niche topics where not enough documents match the recommendations query, as traversing further will open up new possibilities for exploration.

In the next section, we’ll take a quick step beyond the "is_related_to" relationships and see if we can leverage the semantic knowledge graph to also generate and traverse some more interesting edges.

5.4.7 Using semantic knowledge graphs to model arbitrary relationships

Thus far, all of our semantic knowledge graph traversals have leveraged an "is_related_to" relationship. That is to say, we've been finding the strength of the semantic relationship between two words or phrases using the relatedness function, but we have only measured that the nodes are "related", not how they are related. What if we could find other kinds of edges between nodes instead of just "is_related_to" type edges? In this section, we'll explore how to do exactly that.

If you recall, the nodes in a semantic knowledge graph are materialized on the fly by executing a query that matches a set of documents. If the node you start with is `engineer`, that node is internally represented as the set of all documents containing the word `engineer`. If the node is labeled as `software engineer`, that node is internally represented as the set of all documents containing the term `software` intersected with all documents containing the term `engineer`. If the search is for `"software engineer" OR java` then it is internally represented as the set of all documents containing the term `software` one position before the term `engineer` (a phrase) unioned with the set of all documents containing the term `java`. All queries, regardless of their complexity, are internally represented as a set of documents.

You may also recall that an edge is formed by finding the set of documents which two nodes share in common. This means that *both* nodes and edges are internally represented using the same mechanism - a set of documents. Practically speaking, this means that if we can construct a node using a query that approximates an interesting relationship (as opposed to an entity), that we can relate two nodes together through the "relationship node" in a similar way to how an edge would be used to relate the nodes together in a traditional graph structure.

Let's work through an example. Revisiting our scifi dataset let's say we wanted to ask a question about "Jean Grey", one of the popular characters from Marvel Comics X-Men comic books, TV shows, and movies. Specifically, let's say that we wanted to figure out who was in love with Jean Grey.

We can accomplish this by using a starting node of "Jean Grey", traversing to the node "in love with", and then requesting the top related terms associated with "in love with" within the context of "Jean Grey". Listing 5.11 demonstrates this query. By traversing through a node designed to capture an explicit linguistic relationship ("in love with" in this case), we can use the intermediate node to model an edge between the starting and terminating node.

Listing 5.11 Materializing an edge through a "relationship node".

```
collection = "scifi"

starting_node = '"jean grey"'
relationship = '"in love with"'

request = {
    "query": starting_node,
    "params": {
        "qf": "body",
        "fore": "{!type=$defType qf=$qf v=$q}",
        "back": "*:*",
        "defType": "edismax",
        "rows": 0,
        "echoParams": "none",
        "omitHeader": "true"
    },
    "facet": {
        "in_love_with":{
            "type": "query",
            "query": "{!edismax qf=body v=$relationship}",
            "facet": {
                "terminating_nodes": {
                    "type": "terms",
                    "field": "body",
                    "mincount": 25,
                    "limit": 9,
                    "sort": { "body_relatedness": "desc"},
                    "facet": {
                        "body_relatedness": {
                            "type": "func",
                            "func": "relatedness($fore,$back)"
                        }
                    }
                }
            }
        }
    }
}

search_results = requests.post(solr_url + collection + "/select",
➥json=request).json()

for bucket in search_results["facets"]["in_love_with"][
➥"terminating_nodes"]["buckets"]:
  print(str(bucket["val"]) + "  " + str(bucket["body_relatedness"][
  ➥"relatedness"]))
```

Response:

```
jean  0.85044
grey  0.74965
cyclops  0.61313
summers  0.60624
xavier  0.54697
wolverine  0.49361
x  0.46596
mutant  0.46248
magneto  0.43692
```

If the X-Men comics or the names coming back in the response are unfamiliar to you, here's a quick crash course: Jean Grey has recurring relationships with two mutants, one named Cyclops (real name: Scott Summers) and one named Wolverine. Additionally, and unknown to most fans, two of Jean Grey's mentors, Professor Charles Xavier and Magneto, were known to have a love interest in Jean Grey at points throughtout the comic books.

If we examine the results from Listing 5.7, we see all of these expected names listed. The first two terms `jean` and `grey` are obviously the most related, since we are searching for `in love` relative to `jean grey`, so her name is obviously going to be highly semantically related to itself. The next two terms, `cyclops` and `summers` both refer to the same person, Jean's most prominent love interest. Then we see `xavier` and `wolverine`, and the final result in the list is for `magneto`. Figure 5.7 demonstrates the underlying graph relationships for this traversal visually.

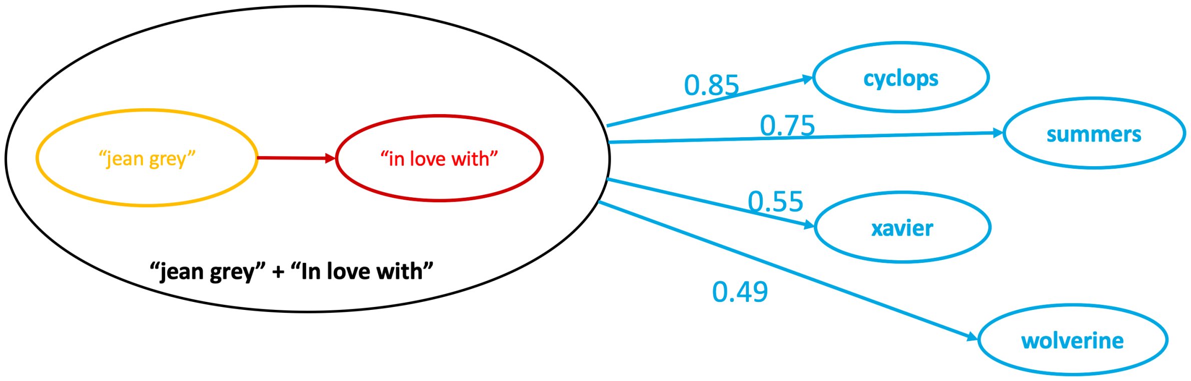


Figure 5.7 Traversing arbitrarily-defined edge types. By materializing a new node with the combined context of both the originating node ("jean grey") and a new node ("in love with"), we can traverse from that combined node ("jean grey" + "in love with") to other nodes. This is equivalent saying we are traversing from "jean grey" through an edge of "in love with" to the other nodes.

By using an intermediate node (i.e. `in love with`) to model a relationship between other nodes, we can form any arbitrarily-typed edge between nodes, as long as we can express that edge as a search query.

While the results of our graph traversal in Listing 5.11 were pretty good overall, we do see, however, that the terms `x` (presumably from "x-men") and `mutant` also show up. Jean Grey and all of the other listed people are mutants in the X-Men comics, which is why these terms are so semantically related. They are clearly not great answers to the question "Who is in love with Jean Grey?", however.

This brings up an important point: the semantic knowledge graph is a statistical knowledge graph. The existence of the "in love with" relationship is purely based upon statistical correlations of terms within our collection, so just like with any ontology learning approach, there is going to be noise. That being said, for an auto-generated graph with no explicit modeling of entities (we just indexed inidividual keywords, with no sense that they represent people), these results are actually quite good.

If we wanted to improve the quality of these results, one of the easiest things to do would be to run pre-processing on the content to extract out entities (people, places, and things) and index those instead of just single-term keywords. This would cause actual people's names (i.e. "Scott Summers", "Charles Xavier", "Jean Grey") to be returned instead of just individual keywords (`summers`, `xavier`, `jean`, `grey`).

It is also worth pointing out that the traversal of relationships depends entirely on whether those relationships were discussed in the underlying corpus of documents. In this case, plenty of forum posts exist discussing each of these peoples' relationships with Jean Grey. Had insufficient documents existed, the results returned may have been poor or non-existent. In order to avoid noise in our results, we set a *mincount* threshold of *25*, indicating that at least 25 documents must exist discussing "jean grey", "in love with", and the other terms found and scored. We recommend setting a *mincount*, as well, to some number greater than *1* to avoid false positives.

While exploring arbitrary linguistic relationships like `in love with` can be useful for an exploratory standpoint, it is usually sufficient from a query understanding standpoint to stick with the default "is related to" relationship and just leverage the relatedness scores between terms for most semantic search use cases. It can still be useful to traverse through multiple levels of relationships to generate better context, however. Specifically, it can be useful to traverse from a term to a classification field to provide some additional context, and then to related meanings of the term within that category. We'll cover this strategy in more detail chapter 6, where we'll focuse on disambiguating terms with multiple meanings.

5.5 Using knowledge graphs for semantic search

By providing the ability to accept arbitrary queries and dynamically discover related terms on the fly in a very context sensitive way, semantic knowledge graphs become a key tool for query interpretation and relevance ranking. We've seen that not only can semantic knowledgege graphs help interpret and expand queries, but that they also provide the ability classify queries and keywords on the fly, as well as to disambiguate multiple senses of the terms in each query.

We also explored early in the chapter how to build expicit knowledge graphs, both manually and through open information extraction techniques, and how to traverse those graphs to pull in useful facts. What may not be obvious yet, however, is how to actually parse arbitrary incoming queries and look up the appropriate pieces in the knowledge graph. We'll spend the majority of chapter 7 covering how to build out an end-to-end semantic search system which can parse queries and integrate each of these knowledge graph capabilities. Before we do that, however, there are some very specific kinds of relationships we need to add to our knowledge graph that are particularly important for search engines, such as misspellings, synonym, and domain-specific phrases. We'll cover how to automatically learn each of these sources of domain-specific terminology from your user signals or your content as we move into the next chapter on learning domain-specific language.

5.6 Summary

- Knowledge graphs model the relationships between entities within your domain and can be built explicitly with known relationships or can be extracted dynamically from your content.
- Open information extraction, the process of extracting facts from your content (subject, relationship, object triples) can be used to learn arbitrary relationships (typically results in noisy data) or to extract hyponym/hypernym relationships (less noisy) from text into an explicit knowledge graph.
- Semantic knowledge graphs enable traversal and ranking of arbitrary semantic relationships between any content within your search index. This allows you to use your content directly as a knowledge graph without any data modeling required beyond just indexing your content.
- Content-based recommendations that don’t rely on user signals can be generated by ranking the most semantically-interesting terms and phrases from documents and using them as a query to find and rank other related documents.
- Semantic knowledge graphs enable better understanding of user intent by powering domain- and context-sensitive query expansion and rewriting, relationship discovery.

6 Using context to learn domain-specific language

This chapter covers

- **Classifying query intent**
- **Query sense disambiguation**
- **Learning related phrases from user signals**
- **Identifying key phrases from user signals**
- **Learning misspellings and alternate term variations from user signals**

In chapter 5 we demonstrated both how to generate and leverage a semantic knowledge graph, and how to extract entities, facts, and relationships explicitly into a knowledge graph. Both of these techniques relied upon navigating either the linguistic connections between terms in a single document or the statistical cooccurrences of terms across multiple documents and contexts. We showed how you can use knowledge graphs to find related terms, and how those related terms can integrate into various query rewriting strategies to increase recall or precision while implementing a conceptual search as opposed to just a text-based keyword match.

In this chapter, we'll dive deeper into the idea of understanding query intent and into the nuances of using different contexts to interpret domain-specific terminology in queries. We'll start off by exploring query classification and then showing how those classifications can be used to disambiguate queries with multiple potential meanings. Both of these approaches will extend our usage of semantic knowledge graphs from the last chapter.

While those semantic-knowledge-graph-based based approaches are great at better contextualizing and interpreting queries, they continue to rely upon having high-quality documents that accurately represent your domain. As such, their efficacy for interpreting user queries depends on how well the queries overlap with the content being searched.

For example, if 75% of your users are searching for clothing, but most of your inventory is films and digital media, then when they search for "shorts" and all the results are videos with short run-time (known as "digital shorts" in the film industry), most of your users will be confused with the results. Given the data in your query logs, it would be better if "shorts" could map to other related terms most commonly found in your query signals like "pants", "clothing", and "shirts".

It can thus be very beneficial to not only rely on the content of your documents to learn relationships between terms and phrases, but to also leverage your user-generated signals. For the second half of this chapter, we'll thus explore how to learn related phases from user signals, how to extract key phrases from user signals, and how to identify common misspellings and alternate spellings from use signals.

By leveraging both content-based approaches and real user interactions to drive your understanding of domain-specific terminology, your search engine will be able to better understand true user intent and react appropriately.

6.1 Classifying query intent

The goal or intent of a query can often matter as much as the keywords themselves. A search for "driver crashed" can mean two *very* different things in the context a search for news or travel content, versus in a technology context. Similarly, someone searching in ecommerce for a specific product name or product id may have an intent to see that very specific item and a high liklihood to want to purchase that item, whereas as more a general search like "kitchen appliances" may signal the intent to just browse results and research what kinds of products may be available.

In both contexts, building a query classifier can prove useful to determine the general kind of query being issued. Depending on the domain, this query context could be automatically applied (limiting the category of documents), or it could be used to modify the relevance algorithm (automatically boost specific products or even skip the results page altogether and just go straight to the product page). In this section, we'll show how to use the semantic knowledge graph from chapter 5 as a classifier for incoming queries to build a query classifier.

K-Nearest Neighbor classification is a type of classification which takes a datapoint (such as a query or term) and tries to find the top K other datapoints that are the most similar in a vector space. A semantic knowledge graph traversal essentially does a k-nearest neighbor search at each level of the graph traversal. This means that if we have a "category" or "classification" field present on our documents, that we can actually ask the semantic knowledge graph to "find the category with the highest relatedness to my starting node". Since the starting node is typically a user's query, this means we can use a semantic knowledge graph to classify the query.

To continue our momentum from the last chapter, let’s continue to leverage the Stackexchange datasets for this section and the next, as we’ve already assembled them into a semantic knowledge graph that can be extended for query classification (this section) and query-sense disambiguation (section 6.2).

Listing 6.1 demonstrates running a search for different sets of keywords and then returning category classifications. For simplicity, since we have indexed multiple different Stack Exchange categories (scifi, health, cooking, devops, etc.), we’ll use those categories as our classifications. Let’s find the most semantically-related categories for a few queries.

Listing 6.1 Query classification leveraging the semantic knowledge graph.

```
def run_query_classification(query,keywords_field="body",
➥classification_field="category",classification_limit=5,
➥min_occurrences=5):

    classification_query = {
        "params": {
            "qf": keywords_field,
            "fore": "{!type=$defType qf=$qf v=$q}",
            "back": "*:*",
            "defType": "edismax",
            "rows": 0,
            "echoParams": "none",
            "omitHeader": "true"
        },
        "query": query,
        "facet": {
            "classification":{
                "type": "terms",
                "field": classification_field,
                "sort": { "classification_relatedness": "desc"},
                "mincount": min_occurrences,
                "limit": classification_limit,
                "facet": {
                    "classification_relatedness": {
                        "type": "func",
                        "func": "relatedness($fore,$back)"
                    }
                }
            }
        }
    }

    search_results = requests.post(solr_url + collection + "/select",
    ➥json=classification_query).json()

    print("Query: " + query)
    print("  Classifications: ")
    for classification_bucket in search_results["facets"][
    ➥"classification"]["buckets"]:
        print("    " + str(classification_bucket["val"]) + "  " +
        ➥str(classification_bucket["classification_relatedness"][
        ➥"relatedness"]))
    print("\n")

run_query_classification( query="docker", classification_field="category",
➥classification_limit=3 )
run_query_classification( query="airplane", classification_field="category",
➥classification_limit=1 )
run_query_classification( query="airplane AND crash",
➥classification_field="category", classification_limit=2 )
run_query_classification( query="camping", classification_field="category",
➥classification_limit=2 )
run_query_classification( query="alien", classification_field="category",
➥classification_limit=1 )
run_query_classification( query="passport", classification_field="category",
➥classification_limit=1 )
run_query_classification( query="driver", classification_field="category",
➥classification_limit=2 )
run_query_classification( query="driver AND taxi",
➥classification_field="category", classification_limit=2 )
run_query_classification( query="driver AND install",
➥classification_field="category", classification_limit=2 )
```

Results:

```
Query: docker
  Classifications:
    devops  0.8376

Query: airplane
  Classifications:
    travel  0.20591

Query: airplane AND crash
  Classifications:
    scifi  0.01938
    travel  -0.01068

Query: camping
  Classifications:
    outdoors  0.40323
    travel  0.10778

Query: alien
  Classifications:
    scifi  0.51953

Query: passport
  Classifications:
    travel  0.73494

Query: driver
  Classifications:
    travel  0.23835
    devops  0.04461

Query: driver AND taxi
  Classifications:
    travel  0.1525
    scifi  -0.1301

Query: driver AND install
  Classifications:
    devops  0.1661
    travel  -0.03103
```

This request leverages the semantic knowledge graph to find the top K nearest neighbors based upon a comparison of the semantic similarity between the query and each available classification (within the category field). As you can see, each query was assigned one or more potential classifications based upon its semantic similarity with each classification.

We see classification scores for each potential category classification for each query, with `airplane` and `passport` classified to `travel`, `camping` classified to `outdoors`, and `alien` classified to `scifi`. When we refine the `airplane` query to a more specific query like `airplane AND crash`, however, we see that the category changes from `travel` to `scifi`, because documents about airplane crashes are more likely to occur within `scifi` documents than `travel` documents.

Similarly, we can see a word like `driver`, which can have multiple polysemous (ambiguous) meanings, returns two potential classifications (`travel` or `devops`), but with the `travel` category being the clear choice when no other context is provided. When additional context *is* provided, however, we can see that the query `driver AND taxi` gets appropriately classified to the `travel` category, while `driver AND install` gets appropriately classified to the `devops` category.

Because of the ability for the semantic knowledge graph to find semantic relationships using the context of any arbitrary query, this makes it an ideal tool for on-the-fly classification of arbitrarily-complex incoming queries. Based upon this classification, you can then auto-apply filters on those categories, route queries to a specific type of algorithm or landing page, or even disambiguate the meaning of the terms within the query. We'll explore applying the query intent further to improve some live queries in chater 7.

Not only can queries benefit from being classified, but we just saw an example of an ambiguous term (`driver`) that further needs to have its multiple meanings differentiated so the search engine uses the correct interpretation. This can be accomplished by adding just one more graph traversal to our query, which we'll walk through next.

6.2 Query sense disambiguation

One of the hardest challenges in interpreting users' intent from their queries is understanding exactly what they mean by each word. The problem of polysemy, or ambiguous terms, can significantly affect your search results.

For example, if someone comes to your search engine and searches for the term "driver", this could have many different possible meanings. Some of these meanings include:

1. A vehicle operator (the taxi driver)
2. Software which makes part of a computer work (install a printer driver or other device driver)
3. A kind of golf club (swing the driver)
4. A kind of tool (i.e. screwdriver)
5. Something that moves an effort forward (key driver of success)

Likewise if someone searches for a "server", this could mean someone who takes orders and waits on tables at a restaurant, or it could mean a computer that runs some software as a service. Figure 6.1 demonstrates these two potential contexts, annd the kinds of related terms one might find within each context.

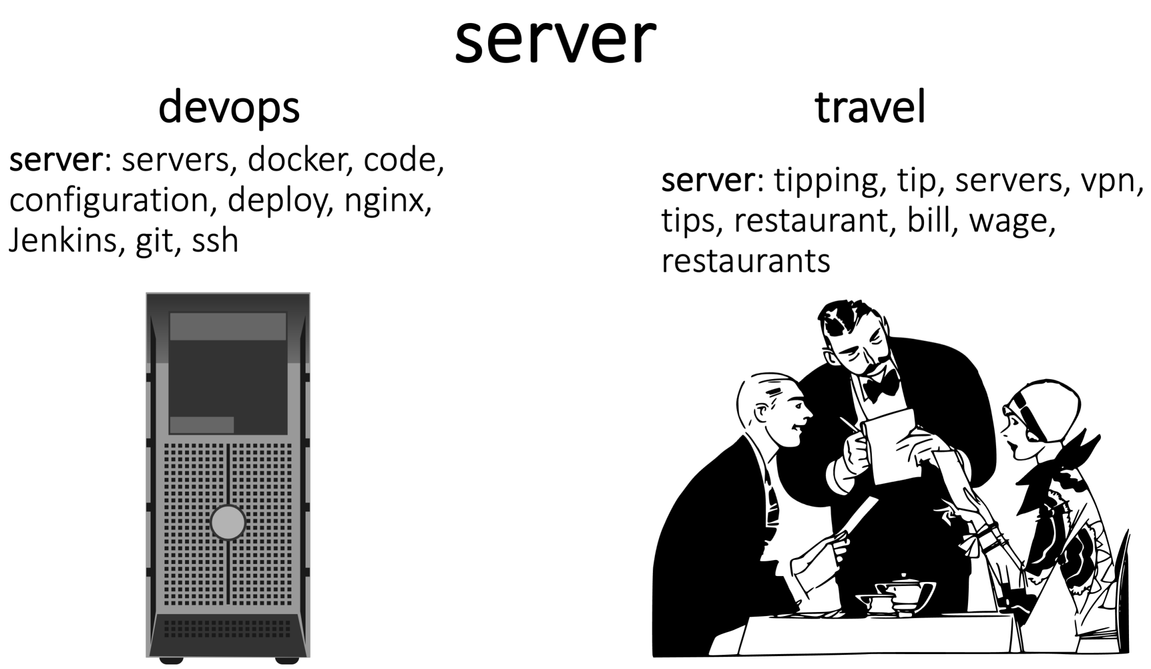

server
devops
server: servers, docker, code, configuration, deploy, nginx, Jenkins, git, ssh
travel
server: tipping, tip, servers, vpn, tips, restaurant, bill, wage, restaurants

Figure 6.1 Differentiating multiple senses of the ambiguous term "server"

Ideally we want our search engine to be able to disambiguate each of these word senses and generate a unique list of related terms within each disambiguated context. It is not sufficient to simply blend multiple potential meanings together within a single list of related terms, as the searcher clearly has a particular intent in mind that we should try to understand and represent.

In section 6.1, we demonstrated how to use the semantic knowledge graph to automatically classify queries into a set of known categories. Given that we already know how to classify our queries, it is trivial to add an additional traversal after the query classification to contextualize the list of related terms to each specific query classification.

In other words, by traversing from query to classification and then to terms, we are able to generate a list of terms that describe a contextualized interpretation of the original query within each of the top classifications.

Listing 6.2 demonstrates a function which will execute this kind of disambiguating query against the semantic knowledge graph.

Listing 6.2 Disambiguating a query's intent across different contexts

```
def run_disambiguation_query(query,keywords_field="body",context_field="category",
    ➥keywords_limit=10,context_limit=5,min_occurrences=5):

    disambiguation_query = {
        "params": {
            "qf": keywords_field,
            "fore": "{!type=$defType qf=$qf v=$q}",
            "back": "*:*",
            "defType": "edismax",
            "rows": 0,
            "echoParams": "none",
            "omitHeader": "true"
        },
        "query": query,
        "facet": {
            "context":{
                "type": "terms",
                "field": context_field,
                "sort": { "context_relatedness": "desc"},
                "mincount": min_occurrences,
                "limit": context_limit,
                "facet": {
                    "context_relatedness": {
                        "type": "func",
                        "func": "relatedness($fore,$back)"
                    },
                    "keywords": {
                        "type": "terms",
                        "field": keywords_field,
                        "mincount": min_occurrences,
                        "limit": keywords_limit,
                        "sort": { "keywords_relatedness": "desc"},
                        "facet": {
                            "keywords_relatedness": {
                                "type": "func",
                                "func": "relatedness($fore,$back)"
                            }
                        }
                    }
                }
            }
        }
    }

    search_results = requests.post(solr_url + collection + "/select",
    ➥json=disambiguation_query).json()

    print("Query: " + query)
    for context_bucket in search_results["facets"]["context"]["buckets"]:
        print("  Context: " + str(context_bucket["val"]) + "  " +
        ➥str(context_bucket["context_relatedness"]["relatedness"]))
        print("    Keywords: ")
        for keywords_bucket in context_bucket["keywords"]["buckets"]:
            print("      " + str(keywords_bucket["val"]) + "  " +
            ➥str(keywords_bucket["keywords_relatedness"]["relatedness"]))
        print ("\n")
```

By traversing first to a specific context (`category` field) and then to keywords (`body` field), we can find the most related contexts and then the most related terms to the original query that are specific to that context. You can see from this listing that a `context` field (the `category` field by default) and a `keywords` field (the `body` field by default) are used as part of a two-level traversal.

For any query that is passed in, we first find the most semantically-related category, and then within that category we find the most semantically related terms to the original query within that category.

Listing 6.3 demonstrates how to call this function, passing in three different queries containing ambiguous terms for which we want to find differentiated meanings, and Table 5.1 demonstrates the results of these three graph traversals.

Listing 6.3 Running Query Disambiguation for Several Queries. Each disambiguation context (`category` field) is scored relative the query, and each discovered keyword (`body` field) is scored relative to both the query and the disambiguation context.

```
run_disambiguation_query( query="server", context_field="category",
➥keywords_field="body" )
run_disambiguation_query( query="driver", context_field="category",
➥keywords_field="body", context_limit=2 )
run_disambiguation_query( query="chef", context_field="category",
➥keywords_field="body", context_limit=2 )
```

The results of the queries in Listing 6.3 can be found in Tables 6.1 - 6.3.

Table 6.1 Contextualized related terms lists by category for the query "server"

Query: server		
Context: devops 0.787	Context: scifi -0.27326	Context: travel -0.28334
Keywords:	Keywords:	Keywords:
server 0.91786	server 0.56847	server 0.74462
servers 0.69526	computer 0.16903	tipping 0.47834
docker 0.66753	computers 0.16403	tip 0.39491
code 0.65852	servers 0.14156	servers 0.30689
configuration 0.60976	virtual 0.12126	vpn 0.27551
deploy 0.60332	communicate 0.09928	tips 0.19982
nginx 0.5847	real 0.098	restaurant 0.19672
jenkins 0.57877	storage 0.09732	bill 0.16507
git 0.56514	system 0.08375	wage 0.1555
ssh 0.55581	inside 0.0771	restaurants 0.15309

Table 6.1 shows the top most semantically-related categories for the query `server`, followed by the most semantically-related keywords from the `body` field within each of those contexts. Based upon the data, we see that the category of `devops` is the most semantically related (positive score of 0.787), whereas the next two categories both contained substantially negative scores (-0.27326 for `scifi` and 0.28334 for `travel`). This indicates that when someone searches for the query `server`, the devops category is overwhelmingly the most likely category in which that term is going to be meaningful.

If we look at the different terms lists that come back for each of the categories, we also see several distinct meanings arise. In the `devops` category a very specific meaning of the term `server` is intended, specifically focused on tools related to managing, building, and deploying code to a computer server. In the `scifi` category, a more general understanding of a server is communicated, but still related to computing. In the travel category, on the other hand, the overwhelming sense of the word server is related to someone working in a restaurant, as we see terms like `tipping`, `restaurant`, and `bill` showing up. Interestingly, one particular kind of computer server, a `vpn` also shows up, because this is the one kind of server that is highly recommended for people to use when traveling to protect their internet communications.

When implementing an intelligent search application using this data, if you know the context of the user is related to travel, it would make sense to use the specific meaning within the travel category. Absent that kind of context, however, the best choice is typically to choose either the most semantically-related category or the most popular category among your users.

Table 6.2 Contextualized related terms lists by category for the query "driver"

Query: driver	
`Context: travel  0.23835`	`Context: devops  0.04461`
`  Keywords:`	`  Keywords:`
`    driver  0.91524`	`    driver  0.71977`
`    drivers  0.68676`	`    ipam  0.70462`
`    taxi  0.6008`	`    aufs  0.63954`
`    car  0.54811`	`    overlayfs  0.63954`
`    license  0.51488`	`    container_name  0.63523`
`    driving  0.50301`	`    overlay2  0.56817`
`    taxis  0.45885`	`    cgroup  0.55933`
`    vehicle  0.45044`	`    docker  0.54676`
`    drive  0.43806`	`    compose.yml  0.52032`
`    traffic  0.43721`	`    compose  0.4626`

Table 6.2 demonstrates a query disambiguation for the query "driver". In this case, there are two related categories, with `travel` being the most semantically-related (score: 0.23835) and `devops` being much less semantically-related. We can see two very distinct meanings of driver appear within each of these contexts, with driver in the travel category being related to `taxi`, `car`, `license`, `driving`, and `vehicle`, whereas within the `devops` category driver is related to `ipam`, `aufs`, `overlayfs`, and so on, which are all different kinds of computer-related drivers.

If someone searches for the word `driver` in your search engine, they clearly do not intend to find documents about both of these meanings of the word driver, and they might be confused if you included both meanings in your search results by only searching for the string `driver` in your inverted index. There are several ways to deal with multiple potential meanings for queried keywords, such as grouping results by meaning to highlight the differences, choosing only the most likely meaning, carefully interspersing different meanings within the search results to provide diversity, or providing alternative query suggestions for different contexts, but usually an intentional choice here is much better than just lazily lumping multiple different meaning together. By leveraging a semantic interpretation of the query like this section demonstrates, you can much better understand your users' intent and deliver more relevant, contextualized search results.

As a final example, Table 6.3 demonstrates the query disambiguation for the query `chef`.

Table 6.3 Contextualized related terms lists by category for the query "chef"

Query: chef	

Context: devops 0.4461
  Keywords:
    chef 0.90443
    cookbooks 0.76403
    puppet 0.75893
    docs.chef.io 0.71064
    cookbook 0.69893
    ansible 0.64411
    www.chef.io 0.614
    learn.chef.io 0.61141
    default.rb 0.58501
    configuration 0.57775
	Context: cooking 0.15151
	  Keywords:
	    chef 0.82034
	    cooking 0.29139
	    recipe 0.2572
	    taste 0.21781
	    restaurant 0.2158
	    cook 0.20727
	    ingredients 0.20257
	    pan 0.18803
	    recipes 0.18285
	    fried 0.17033

The top two contexts for the query `chef` both show reasonably positive relatedness scores, indicating that both meanings are likely interpretations. While the `devops` context has a higher score (0.4461) than the `cooking` context (0.15151), it would still be important to take the user's context into consideration as best as possible when choosing between these two meanings. The meaning of `chef` within the `devops` context is related to the Chef configuration management software used to build and deploy servers (related terms include `puppet`, `ansible`, etc.) whereas within the cooking context it is referring to a person who prepares food (`cooking`, `taste`, `restaurant`, `ingredients`, etc.).

Interestingly, the Chef software makes used of the idea of "recipies" and "cookbooks" in its terminology, which was originally borrowed from the idea of a chef in a kitchen, so we may even see overlap between terms as we go further down the list, even though those other terms are actually also ambiguous across the two broad classifications we are using (`devops` and `cooking`)

Of course, if you have a more fine-grained classification available on your documents, you may be able to derive even more nuanced, contextualized interpretations of your users' queries, making a semantic knowledge graph highly effective at nuanced query interpretation and expansion. By combining query classification, term disambiguation, and query expansion (see section 5.4.5) together, a semantic knowledge graph can power enhanced domain-specific and highly contextualized semantic search capabilities within your AI-powered search engine. We'll dive into using these techniques further in chapter 7 when we apply them in a live semantic search use case.

6.3 Learning related phrases from query signals

In chapter 5 and in this chapter thusfar, you’ve seen how to leverage your content as a knowledge graph to discover related terms, to classify queries, and to disambiguate nuanced meanings of terms. While these techniques are powerful, they are also entirely dependent upon the quality of your documents and how well they represent what your user’s queries. Throughout the rest of this chapter, we’ll explore the other major source of knowledge about your domain - your user queries and signals. In many use cases, your user signals may provide similar, if not more useful insights for interpreting queries than your content.

As a starting point for learning domain-specific terminology from real user behavior, let’s consider what your query logs represent. For every query to your search engine, a query log contains an identifier for the person running the search, the query that was run, and the timestamp of the query. This means that if a single user searches for multiple terms, you can group those searches together and also tell in which order the terms were entered.

While not always true, one reasonable assumption is that if someone entered two different queries within a very close time-span of each other, that the second query is likely to be either a refinement of the first query, or about a related topic. Figure 6.1 demonstrates a realistic sequence of searches you might find for a single user in your query logs.

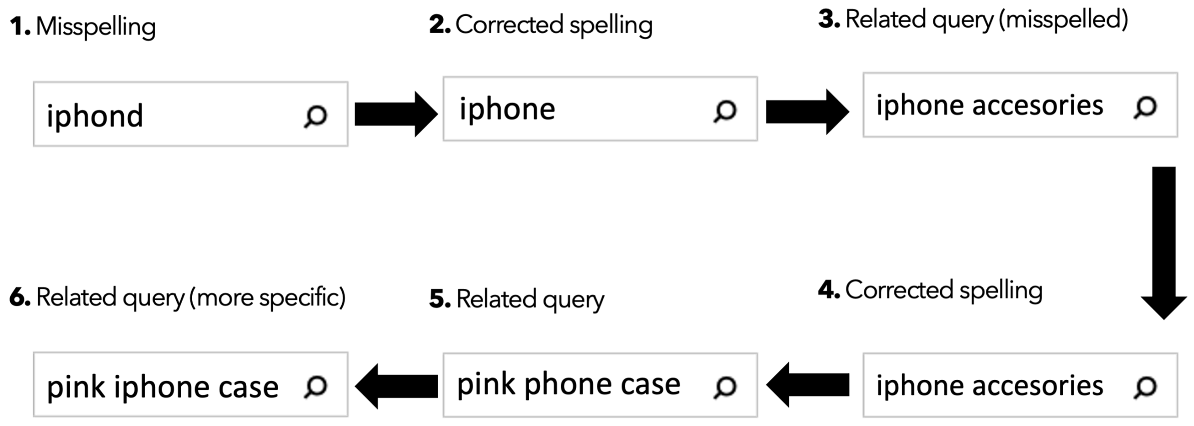


Figure 6.2 A typical sequence of searches from query logs for a particular user

When looking at these queries, we intuitively understand that "iphond" is a misspelling of "iphone", that "iphone accesories" is a misspelling of "iphone accessories", and that "iphone", "pink phone case", and "pink iphone case" are all related terms. We’ll deal with the misspellings in later section, so we can just consider those to also be related terms for now.

While it is not wise to depend on a single user’s signals to deduce that two queries are related, if you see the same combinations of queries entered by many different users then it is probably a safe bet that those queries are related. As we demonstrated in chapter 5, queries can be expanded to include related terms and improve recall. Whereas in section 5.4.5 we expanded queries based upon related terms found within our collection of documents, in this section, we’ll instead find related terms based upon query signals.

6.3.1 Mining query logs for related queries

Before mining user signals for related queries, let’s first convert our signals into a simpler format for processing. Listing 6.4 provides a transformation from our generic signal structure to a simple structure that maps each occurrence of a query term to the user who searched for that term.

Listing 6.4 Mapping signals into keyword, user pairs

```
#Calculation:
signals_collection="signals"
signals_opts={"zkhost": "aips-zk", "collection": signals_collection}
df = spark.read.format("solr").options(**signals_opts).load()
df.createOrReplaceTempView("signals")
spark.sql("""
select lower(searches.target) as keyword, searches.user as user
from signals as searches
where searches.type='query'
""").createOrReplaceTempView('user_searches')

#Show Results:
spark.sql("""select count(*) from user_searches """).show(1)
print("Simplified signals format:")
spark.sql("""select * from user_searches """).show(3)
```

Results:

```
+--------+
|    rows|
+--------+
|  725459|
+--------+

Simplified signals format:
+----------------+-------+
|         keyword|   user|
+----------------+-------+
|             gps| u79559|
|surge protectors|u644168|
|      headphones| u35624|
+----------------+-------+
only showing top 3 rows
```

You can see from Figure 6.4 that there are over 725K queries occurrences represented, along with the user who sent each query. Our goal is ultimately to find related queries, so the next processing step is to find all query pairs where both queries were submitted by a shared user. In other words, assuming individual users tend to search for related keywords, let’s find the number of coocurrences of each pair of keywords among users, since more occurrences likely means the keywords are more related.

Listing 6.5 shows each query pair where both queries were searched by the same user, along with the number of users that searches for both queries (`users_cooc`).

Listing 6.5 Total occurrences and coocurrences of queries, sorted by how many users searched for the queries.

```
#Calculation:
spark.sql('''
select k1.keyword as keyword1, k2.keyword as keyword2,
➥count(distinct k1.user) users_cooc
from user_searches k1 join user_searches k2 on k1.user = k2.user where
➥k1.keyword > k2.keyword
group by k1.keyword, k2.keyword ''').createOrReplaceTempView(
➥'keywords_users_cooc')

spark.sql('''
select keyword ,  count(distinct user) users_occ
from user_searches
group by keyword ''').createOrReplaceTempView('keywords_users_oc')

#Show Results:
spark.sql('''select * from keywords_users_oc order by users_occ desc''').
➥show(10)
spark.sql('''select count(1) as keywords_users_cooc from
➥keywords_users_cooc''').show()
spark.sql('''select * from keywords_users_cooc order by users_cooc desc''').
➥show(10)
```

Results:

```
+-----------+---------+
|    keyword|users_occ|
+-----------+---------+
|     lcd tv|     8449|
|       ipad|     7749|
|hp touchpad|     7144|
|  iphone 4s|     4642|
|   touchpad|     4019|
|     laptop|     3625|
|    laptops|     3435|
|      beats|     3282|
|       ipod|     3164|
| ipod touch|     2992|
+-----------+---------+
only showing top 10 rows

+-------------------+
|keywords_users_cooc|
+-------------------+
|             244876|
+-------------------+

+-------------+---------------+----------+
|     keyword1|       keyword2|users_cooc|
+-------------+---------------+----------+
|green lantern|captain america|        23|
|    iphone 4s|         iphone|        21|
|       laptop|      hp laptop|        20|
|         thor|captain america|        18|
|   skullcandy|          beats|        17|
|    iphone 4s|       iphone 4|        17|
|         bose|          beats|        17|
|      macbook|            mac|        16|
|      laptops|         laptop|        16|
|         thor|  green lantern|        16|
+-------------+---------------+----------+
only showing top 10 rows
```

The first result from Listing 6.5 shows the most search for queries. While these may be the most popular queries, they aren't necessarily the queries that coocur the most often with other queries. The second result from Listing 6.6 shows the total number of query pairs (`244,876`) where both queries were searched by the same user at least once.

The final result shows the top query pairs, sorted by number of users who search for both queries. Of the top results, you'll notice that `iphone 4s`, `iphone`, and `iphone 4` are highly related and that `laptop`, `laptops`, and `hp laptop` are also highly related. Specifically, `iphone` is a more gneneral search for `iphone 4`, which is a more general search for `iphone`, and `hp laptop` is a more general form of `laptop`, for which `laptops` is a spelling variation. You'll also notice that `thor`, `captain america`, and `green lantern` are all related (comic book super heroes), as well as `skullcandy`, `beats`, and `bose` (all related to headphones).

You'll also notice, however, that the top result only has `23` coocurring users, which means the number of data points is fairly sparse and that as we move further down the list, the reliance of coocurrence is going to be challenging and likely include a lot of noise. In the next section, we'll explore a technique to combine signals together along a different axis (product interactions), which can help with this sparsity problem.

While directly aggregating the number of searches into coocurrences by users helps find the most popular query pairs, poplarity of searches isn't the only metric useful for finding relatedness. The keywords `and` and `of` are highly coocurring, as are `phones`, `movies`, `computers`, and `electronics`, because they are all general words that many people search. In order to also focus on the strength of the relationship between terms independent of their individual popularity, we can leverage a technique called Pointwise Mutual Information.

Pointwise Mutual Information (PMI) is a measure of the correlation between any two events. In the context of natural language processing, PMI tells the liklihood of two words occurring together because they are related versus the liklihood of them occurring together by chance. There are multiple functions that can be used to calculate and normalize PMI, but we'll use a variation called PMI^k, where `k = 2`, which does a better job than PMI at keeping scores consistent regardless of word frequencies.

The formula for calculating PMI^2 is as follows:

$$PMI^2(k1, k2) = log \frac{P(k1, k2)^2}{P(k1) \times P(k2))}$$

Figure 6.3 AciiMath

In our implementation, `k1` and `k2` represent two different keywords which we want to compare. `P(k1,k2)` represents how often the same user searches for both keywords, whereas `P(k1)` and `p(k2)` represent how often a users only searches for the the first keyword or second keyword, respectively. Intuitively, if the keywords appear together more often than they would be expected to based upon their likelihood of randomly appearing together, then they will have a higher PMI^2 score. The higher the score, the more likely the terms are to be semantically related.

Listing 6.6 demonstrates the PMI^2 calculation on our coocurring query pairs dataset.

Listing 6.6 PMI2 calculation on user searches

```
#Calculation:
spark.sql('''
select k1.keyword as k1, k2.keyword as k2, k1_k2.users_cooc, k1.users_occ
➥as n_users1,k2.users_occ as n_users2,
log(pow(k1_k2.users_cooc,2) / (k1.users_occ*k2.users_occ)) as pmi2
from keywords_users_cooc as k1_k2
join
keywords_users_oc as k1 on k1_k2.keyword1= k1.keyword
join
keywords_users_oc as k2 on k1_k2.keyword2 = k2.keyword
''').registerTempTable('user_related_keywords_pmi')


#Show Results:
spark.sql( '''
select * from user_related_keywords_pmi where users_cooc >10 order by pmi2
➥desc
''').show(10)
```

Results:

```
+-----------------+--------------------+----------+--------+--------+
➥------------------+
|               k1|                  k2|users_cooc|n_users1|n_users2|
➥              pmi2|
+-----------------+--------------------+----------+--------+--------+
➥------------------+
|  iphone 4s cases|      iphone 4 cases|        10|     158|     740|
➥-7.064075033237091|
|     sony laptops|          hp laptops|         8|     209|     432|
➥-7.251876756849249|
|otterbox iphone 4|            otterbox|         7|     122|     787|
➥-7.580428995040033|
|    green lantern|     captain america|        23|     963|    1091|
➥-7.593914965772897|
|          kenwood|              alpine|        13|     584|     717|
➥-7.815078108504774|
|      sony laptop|         dell laptop|        10|     620|     451|
➥-7.936016631553724|
|   wireless mouse|           godfather|         6|     407|     248|
➥-7.938722993151467|
|       hp laptops|        dell laptops|         6|     432|     269|
➥ -8.07961802938984|
|      mp3 players|        dvd recorder|         6|     334|     365|
➥-8.127519408103081|
|          quicken|portable dvd players|         6|     281|     434|
➥ -8.12788026497804|
+-----------------+--------------------+----------+--------+--------+
➥------------------+
only showing top 10 rows
```

The results from Listing 6.6 are sorted by PMI^2 score (highest to lowest), and we set a minimum occurrences threshold at >5 to help remove noise. You can now see some results like hp `laptops`, `dell laptops`, and `sony laptops` showing up as related, as well as brands like `kenwood` and `alpine`. Notably, however, there is also noise in the pairs like `wireless mouse` with `godfather` and `quicken` with `portable dvd players`. One caveat of using PMI is that a small number of occurrences together across a few users can lead to noise more easily than when using coocurrence, which is based upon the assumption of many coocurrences.

One way to blend the benefits of both the coocurrence model and the PI^2 models is to create a composite score. This will provide a blend of popularity and liklihood of occurrence together, which should move query pairs that match on both scores to the top of the list. Listing 6.7 demonstrates one way to blend these two measures together. Specifically, we take a ranked list of all coocurrence scores (r1) and a ranked list of all PMI^2 scores (r2) and blend them together to generate a composite ranking score as shown in Figure 6.4.

Listing 6.7 Composite Ranking Score combining coocurrence and PMI^2 ranking

```
text{comp_score}(q1,q2) = \frac{((r1(q1,q2) + r2(q1,q2))/(r1(q1,q2) \times
➥r2(q1,q2)) )}2
```

The `comp_score`, or composite rank score,shown in Figure 6.4 assigns a high score query pairs where their rank in the coocurrence list (`r1`) and their rank in the PMI^2 list (`r2`) is high, and a lower rank as the terms move further down in the rank lists. The end results is a blended ranking that considers both the populartiy (coocurrence) and the liklihood of relatedness of queries despite their popularity (PMI^2). Listing 6.7 shows how to calculate the `comp_score` with our query data.

Listing 6.8 Calculate a composite score, blending coocurrence and PMI

```
#Calculation:
spark.sql('''
select  *, (r1 + r2 /( r1 * r2))/2 as comp_score from (
 select *,
   rank() over (partition by 1 order by users_cooc desc )  r1 ,
   rank() over (partition by 1 order by pmi2 desc )  r2
  from user_related_keywords_pmi ) a  '''
).registerTempTable('users_related_keywords_comp_score')

#Show Results:
spark.sql( '''
  select k1, k2, users_cooc, pmi2, r1, r2, comp_score
  from users_related_keywords_comp_score
  where users_cooc >= 10
''').show(20)
```

Results:

```
+-------------+--------------+----------+------------------+---+------+
➥------------------+
|           k1|            k2|users_cooc|              pmi2| r1|    r2|
➥        comp_score|
+-------------+--------------+----------+------------------+---+------+
➥------------------+
|green lantern|captain america|        23| -7.593914965772897|  1|  8626|
➥                1.0|
|    iphone 4s|         iphone|        21|-10.216737746029027|  2| 56156|
➥               1.25|
|       laptop|      hp laptop|        20| -9.132682838345458|  3| 20383|
➥1.6666666666666667|
|         thor|captain america|        18| -8.483026598234463|  4| 13190|
➥              2.125|
|         bose|          beats|        17|-10.074222345094169|  5| 51916|
➥                2.6|
|    iphone 4s|       iphone 4|        17| -10.07559536143275|  5| 51964|
➥                2.6|
|   skullcandy|          beats|        17|  -9.00066454587719|  5| 18792|
➥                2.6|
|         thor|  green lantern|        16| -8.593796095512284|  8| 14074|
➥             4.0625|
|      laptops|         laptop|        16|-10.792204327465662|  8| 80240|
➥             4.0625|
|      macbook|            mac|        16| -9.891277373272931|  8| 45464|
➥             4.0625|
|   headphones|   beats by dre|        15|  -9.98923457501079| 11| 49046|
➥ 5.545454545454546|
|  macbook air|        macbook|        15| -9.442537922965805| 11| 26943|
➥ 5.545454545454546|
|  macbook pro|        macbook|        15|  -9.73733746318645| 11| 39448|
➥ 5.545454545454546|
|  macbook pro|    macbook air|        13| -9.207068753875852| 14| 21301|
➥ 7.035714285714286|
|         nook|         kindle|        13| -9.661503425798296| 14| 36232|
➥ 7.035714285714286|
|       ipad 2|           ipad|        13| -11.76529194320276| 14|196829|
➥ 7.035714285714286|
|      kenwood|         alpine|        13| -7.815078108504774| 14|  9502|
➥ 7.035714285714286|
|   ipod touch|           ipad|        13|-11.829117705935245| 14|200871|
➥ 7.035714285714286|
|   skullcandy|     headphones|        12| -9.318865873777165| 19| 23317|
➥ 9.526315789473685|
|      macbook|          apple|        12|-10.465639011826868| 19| 62087|
➥ 9.526315789473685|
+-------------+--------------+----------+------------------+---+------+
➥------------------+
only showing top 20 rows
```

Overall, the composite rank score does a reasonable job blending our coocurrence and PMI^2 metrics to overcome the limitations of each. The top results shown in Listing 6.7 all look reasonable. One items we already noted as problematic in this section, however, is that the coocurrence numbers are very sparse. Specifically, the highest coocurrence of any query pairs, out of over 700,000 query signals, was `23` overlapping users for for `green lantern` and `captain america`, per Listing 6.5.

In the next section, we'll show a way to overcome this sparse data problem, where there is a lack of overlap between users for specific query pairs. We'll accomplish this by aggregating many users together into a larger group with similar behaviors. Specifically, we'll switch our focus to the products with which the user queries interact, as opposed to each individual user.

6.3.2 Finding related queries through product interactions

The technique used to find related terms in section 1.3.1 depends upon lots of users searching for overlapping queries. As we saw, with over `700K` query signals, the highest overlap of any query pair was `23` users. Because the data can be so sparse, it can often make sense to aggregate on something other than users.

In this section, we'll show how to use the same technique (leveraging coocurrence + PMI[2]), but rolling up based upon product click signals instead.

In section 1.3.1, we looked exclusively at query signals, but in this section we want to actually map every keyword to both the user who searched for it and also any products the user clicked on as a result of the search, which is demonstrated in Listing 6.8.

Listing 6.9 Mapping raw signals into keyword, user, product pairs

```
#Calculation:
spark.sql("""select lower(searches.target) as keyword,
➥searches.user as user, clicks.target as product
from signals as searches right join signals as clicks on searches.query_id
➥= clicks.query_id
where searches.type='query' and clicks.type = 'click'""").
➥createOrReplaceTempView('keyword_click_product') ❶

#Show Results:
print("Original signals format: ")
spark.sql(''' select * from signals where type='query' ''').show(3)
print("Simplified signals format: ")
spark.sql(''' select * from keyword_click_product ''').show(3)
```

❶ Find every keyword + user + click combination

Results:

```
Original signals format:
+--------------------+-----------+-------------------+-----------+-----+
➥-------+
|                  id|   query_id|         signal_time|     target| type|
➥   user|
+--------------------+-----------+-------------------+-----------+-----+
➥-------+
|00001ba7-b74c-421...|u164451_0_1|2020-02-20 14:24:...|MacBook pro|query|
➥u164451|
|0001465f-cfe1-427...|u608945_0_1|2019-06-15 18:08:...|        g74|query|
➥u608945|
|000173d5-f570-485...| u93764_0_1|2019-11-30 19:56:...|Pioneer avh|query|
➥ u93764|
+--------------------+-----------+-------------------+-----------+-----+
➥-------+
only showing top 3 rows

Simplified signals format:
+-----------------+-------+------------+
|          keyword|   user|     product|
+-----------------+-------+------------+
|lord of the rings|u100793|794043140617|
|        subwoofer|u100953|713034050223|
|         game boy|u100981|841872143378|
+-----------------+-------+------------+
only showing top 3 rows
```

The transformation in Listing 6.8 combines separate query and click signals into single rows with three key columns: `keyword`, `user`, and `product`. Using this data, we'll now be able to determine the strength of the relationship between any two keywords based upon their use across independent users to find the same products.

Listing 6.9 generates pairs of keywords to determine their potential relationship for all keyword pairs where both keywords were used in a query for the same document. The intuition behind looking for overlapping queries per user in section 1.3.1 was that each user is likely to search for related items, but it's also the case that each product is likely to be searched for by related queries. This shifts our mental model from "find how many users searched for both queries" to "find how many documents were found by both queries across all users".

The results of this transformation transformation in Listing 6.9 includes the following colums:

- `k1`, `k2` : the two keyword that are potentially related because they both resulted in a click on the same product
- `n_users1` : the number of users who searched for `k1` that clicked on a product the was also clicked on after a search by some user for `k2`
- `n_users2`: the number of users who searched for `k2` that clicked on a product that was also clicked on after a search by some user for `k1`
- `users_cooc` : `n_users1` + `nusers2`. Represents the total number of users who searched for either `k1` or `k2` and visited a product visited by other searchers for `k1` or `k2`.
- `n_products` : the number of products that were clicked on by searchers of both `k1` and `k2`.

Listing 6.10 Creating a view of the signals just for processing queries

```
#Calculation:
spark.sql('''
select k1.keyword as k1, k2.keyword as k2, sum(p1) n_users1,sum(p2) n_users2,
sum(p1+p2) as users_cooc, count(1) n_products
from
(select keyword, product, count(1) as p1 from keyword_click_product group by
➥keyword, product) as k1
join
(select keyword, product, count(1) as p2 from keyword_click_product group by
➥keyword, product) as k2
on k1.product = k2.product
where k1.keyword > k2.keyword
group by k1.keyword, k2.keyword
''').createOrReplaceTempView('keyword_click_product_cooc')

#Show Results:
spark.sql('''select count(1) as keyword_click_product_cooc from
➥keyword_click_product_cooc''').show()
spark.sql('''select * from keyword_click_product_cooc order by
➥n_products desc''').show(20)
```

Results:

```
+--------------------------+
|keyword_click_product_cooc|
+--------------------------+
|                   1579710|
+--------------------------+

+--------------+-------------+--------+--------+----------+----------+
|            k1|           k2|n_users1|n_users2|users_cooc|n_products|
+--------------+-------------+--------+--------+----------+----------+
|       laptops|       laptop|    3251|    3345|      6596|       187|
|       tablets|       tablet|    1510|    1629|      3139|       155|
|        tablet|         ipad|    1468|    7067|      8535|       146|
|       tablets|         ipad|    1359|    7048|      8407|       132|
|       cameras|       camera|     637|     688|      1325|       116|
|          ipad|        apple|    6706|    1129|      7835|       111|
|      iphone 4|       iphone|    1313|    1754|      3067|       108|
|    headphones|  head phones|    1829|     492|      2321|       106|
|        ipad 2|         ipad|    2736|    6738|      9474|        98|
|     computers|     computer|     536|     392|       928|        98|
|iphone 4 cases|iphone 4 case|     648|     810|      1458|        95|
|        laptop|    computers|    2794|     349|      3143|        94|
|       netbook|       laptop|    1017|    2887|      3904|        94|
|       netbook|      laptops|    1018|    2781|      3799|        91|
|    headphones|    headphone|    1617|     367|      1984|        90|
|        laptop|           hp|    2078|     749|      2827|        89|
|        tablet|    computers|    1124|     449|      1573|        89|
|       laptops|    computers|    2734|     331|      3065|        88|
|           mac|        apple|    1668|    1218|      2886|        88|
|     tablet pc|       tablet|     296|    1408|      1704|        87|
+--------------+-------------+--------+--------+----------+----------+
only showing top 20 rows
```

The `users_cooc` and `n_products` calculations are two different ways to look at overall signal quality for how confident we are that two terms `k1` and `k2` are related. The results are sorted by `n_products` currently, and you can see that the list of relationships at the top of the list is quite clean, containing:

- *Spelling variations*: laptops laptop ; headphones head phone ; etc.
- *Brand associations*: tablet ipad ; laptop hp ; mac apple ; etc.
- *Synonyms/Alternate Names*: netbook laptop ; tablet pc tablet
- *General/Specific Refinements*: tablet ipad ; iphone iphone 4 ; computers tablet ; computers laptops

You can write custom, domain-specific algorithms to identify some of these specific types of relationships, as we'll do for spelling variations in section 1.5, for example.

It is also possible to use the `n_users1` and `n_users2` to identify which of the two keywords is the more popular. In the example of a spelling variation, for example, we see that `headphones` is used more commonly than `head phone` (`1829` vs. `492` users), and is also more common than `headphone` (`1617` vs. `367` users). Likewise, we see that `tablet` is much more common in usage than `tablet pc` (`1408` vs. `296` users).

While our current list of keywords pairs looks clean, it currently only represents the keyword pairs that both occurred together in searches that led to the same products. Getting a sense of the actual popularity of each keyword overall will provide a better sense of which specific keywords are the most important for our knowledge graph. Listing 6.10 calculates the most popular keywords from our query signals that resulted in at least one product click.

Listing 6.11 Computing the popularity of each keyword

```
#Calculation:
spark.sql('''
select keyword, count(1) as n_users from keyword_click_product group by
➥keyword
''').registerTempTable('keyword_click_product_oc')

#Show Results:
spark.sql('''select count(1) as keyword_click_product_oc from
➥keyword_click_product_oc''').show()
spark.sql('''select * from keyword_click_product_oc order by
➥n_users desc''').show(20)
```

Results:

```
+------------------------+
|keyword_click_product_oc|
+------------------------+
|                   13744|
+------------------------+

+------------+-------+
|     keyword|n_users|
+------------+-------+
|        ipad|   7554|
| hp touchpad|   4829|
|      lcd tv|   4606|
|   iphone 4s|   4585|
|      laptop|   3554|
|       beats|   3498|
|     laptops|   3369|
|        ipod|   2949|
|  ipod touch|   2931|
|      ipad 2|   2842|
|      kindle|   2833|
|    touchpad|   2785|
|   star wars|   2564|
|      iphone|   2430|
|beats by dre|   2328|
|     macbook|   2313|
|  headphones|   2270|
|        bose|   2071|
|         ps3|   2041|
|         mac|   1851|
+------------+-------+
only showing top 20 rows
```

This list is identical to the list from Listing 6.5, but now instead of showing number of users who searched for a keyword, it is showing number of users who searched for a keyword and also clicked on a product. We'll use this as our master list of queries for the PMI^2 calculation.

With our query pairs list and query popularity now generated based upon queries and product interactions, the rest of our calculations (PMI^2 and Composite Score) are exactly the same as in section 1.3.1, so we'll omit them here (they are included in the notebooks for you to run). After caclulating the PMI^2 and final composite scores, Listing 6.11 shows the final results of our product-interaction-based related terms calculations.

Listing 6.12 Final related terms scoring based upon product interactions

```
# ...calculate PMI2, per Listing 6.6
# ...calculate comp_score, per Listing 6.7

#Show Results
spark.sql( '''
  select count(1) product_related_keywords_comp_scores from
  ➥product_related_keywords_comp_score
''').show()

spark.sql( '''
  select k1, k2, n_users1, n_users2, pmi2, comp_score
  from product_related_keywords_comp_score
  order by comp_score asc
''').show(20)
```

Results:

```
+------------------------------------+
|product_related_keywords_comp_scores|
+------------------------------------+
|                             1579710|
+------------------------------------+

+----------+-----------+--------+--------+------------------+
➥------------------+
|        k1|         k2|n_users1|n_users2|              pmi2|
➥        comp_score|
+----------+-----------+--------+--------+------------------+
➥------------------+
|      ipad|hp touchpad|    7554|    4829|1.2318940540272372|
➥               1.0|
|    ipad 2|       ipad|    2842|    7554| 1.430517155037946|
➥              1.25|
|    tablet|       ipad|    1818|    7554|1.6685364924472557|
➥1.6666666666666667|
|  touchpad|       ipad|    2785|    7554|1.2231908670315748|
➥             2.125|
|   tablets|       ipad|    1627|    7554|1.7493143317791537|
➥               2.6|
|     ipad2|       ipad|    1254|    7554|1.9027023623302282|
➥3.0833333333333335|
|      ipad|      apple|    7554|    1814|1.4995901756327583|
➥3.5714285714285716|
|  touchpad|hp touchpad|    2785|    4829|1.3943192464710108|
➥            4.0625|
|      ipad|  hp tablet|    7554|    1421|1.5940745096856273|
➥ 4.555555555555555|
|ipod touch|       ipad|    2931|    7554|0.8634782989267505|
➥              5.05|
|      ipad|      i pad|    7554|     612| 2.415162433949984|
➥ 5.545454545454546|
|    kindle|       ipad|    2833|    7554| 0.827835342752348|
➥ 6.041666666666667|
|    laptop|       ipad|    3554|    7554|0.5933664189857987|
➥ 6.538461538461538|
|      ipad| apple ipad|    7554|     326|2.9163836526446025|
➥ 7.035714285714286|
|    ipad 2|hp touchpad|    2842|    4829|1.1805849845414993|
➥ 7.533333333333333|
|   laptops|     laptop|    3369|    3554|1.2902371152378296|
➥           8.03125|
|      ipad|         hp|    7554|    1125| 1.534242656892875|
➥ 8.529411764705882|
|     ipads|       ipad|     254|    7554|3.0147863057446345|
➥ 9.027777777777779|
|      ipad|  htc flyer|    7554|    1834|1.0160007504012176|
➥ 9.526315789473685|
|      ipad|    i pad 2|    7554|     204| 3.180197301966425|
➥            10.025|
+----------+-----------+--------+--------+------------------+
➥------------------+
only showing top 20 rows
```

The results of 6.11 show the benefit of aggregating at a less granular level. By looking at all queries that led to a particular product being clicked, the list of query pairs is now much larger than in section 1.3.1, where aggregated query pairs down to the individual users. You can see that there are now `1,579,710` query pairs under consideration versus `244,876` (Listing 6.5) when aggregating by user.

Further, you can see that the related queries include more fine-grained variations for top queries (ipad, ipad 2, ipad2, i pad, ipads, i pad 2). Having more granular variations like this will come in handy if combining this related term discovery with other algorithms, like misspelling detection, which we'll cover in section 1.5.

Between the semantic knowledge graph techniques in the last chapter and the query log mining in this chapter, you've now seen multiple techniques for discovering related terms. Before we can apply related terms to a query, however, it is is important to be able to first identify the terms already present in a query. In the next section, we'll cover how to identify known phrases from our query signals.

6.4 Phrase detection from user signals

In section 5.3 we showed how to extract arbitrary phrases and relationships from your documents leveraging open information extraction. Those extracted phrases can be a gold mine, as they help define a list of important phrases to look for in users queries.

In section 5.3, we implemented open information extraction techniques to build a knowledge graph from text documents. In that process, we using Natural Language Processing (NLP) models to extract entities from noun phrases in text. While this can go a long way toward discovering all of the relevant domain-specific phrases within your content, this approach suffers from two different problems:

1. *It generates a lot of noise*: not every nouns phrase across your potentially massive set of documents is important, and the odds of identifying incorrect phrases (false positives) increases as your number of documents increases.
2. *It ignores what your users care about*: The real measure of user interest is communicated by what they search. They may only be interested in a subset of your content, and they may be looking for things that aren't even represented well within your content.

In this section, we'll focus on how to also identify important domain-specific phrases from your user signals.

6.4.1 Treating queries as entities

The easiest way to extract entities from queries is to just treat the full queries themselves as entities. In use case like our RetroTech ecommerce site, this works very well, as many of the queries are actual product names, categories, brand names, company names, or people's names (actors, musicians, etc.). Given that context, most of the high-popularity queries end up being entities that can be used directly as phrases without needing any special parsing.

If you look back the output of Listing 6.10, you'll find the following the most popular queries (by number of users issuing them):

```
+------------+-------+
|     keyword|n_users|
+------------+-------+
|        ipad|   7554|
| hp touchpad|   4829|
|      lcd tv|   4606|
|   iphone 4s|   4585|
|      laptop|   3554|
|      ...
+------------+-------+
```

Each of these represent entities that belong in a known-entities list, with many of them being multi-word phrases. In this case, the simplest method for extracting entities is also the most powerful - just leverage the queries as your entities list. The higher the number of unique users issue each query, the higher the confidence you can have that it should be added to your entities list.

If you can cross-reference the queries with you documents to find phrases that overlap in both, that is one way to reduce potential false positives from noisy queries. Additionally, if you have different fields in your documents, like product name or company, you can cross-reference your queries with those fields to assign a type to the entities found within your queries.

Ultimately, depending ond the complexity of your queries, simply leveraging the most common searches as your key entities and phrases may be the simplest possible approach to generate a list of known phrases to extract in future queries.

6.4.2 Extracting entities from more complex queries

In some use cases, the queries may contain more noise (boolean structure, advanced query operators, etc.) and therefore may not be directly usable as entities. In those cases, the best approach to extracting entities may be to re-apply the entity extraction strategies from chapter 5, but on your query signals.

Out of the box, our search engine parses queries as individual keywords, and looks those individual keywords up in the inverted index, which also contains individual keywords. For example, a query for `new york city` will be automatically interpreted as the boolean query `new AND york AND city` (or if you set the default operator to `OR` then `new OR york OR city`), and then the relevance ranking algorithms will score each keyword individually instead of understanding that certain words combine to make phrases that then take on a different meaning.

Being able to identify and extract domain-specific phrases from queries can enable more accurate query interpretation and relevance, however. We already demonstrated one way to extract domain-specific phrases from documents in section 5.3, using the Spacy NLP library to do a dependency parse and extra out noun phrases. While queries are often too short to perform a true dependency parse, it still possible to apply some part of speech filtering on any discovered phrases in queries to limit to noun phrases. If you need to split sections of queries apart, you can also tokenize the queries (see 6.14) and remove query syntax (`and`, `or`, etc.) prior to looking for phrases to extract. Handling the specific query patterns for your application may might require some domain-specific query parsing logic, but if your queries are largely single phrases or easily tokenizable into multiple phrases, your queries likely represent the best source of domain-specific phrases to extract and add to your knowledge graph.

6.5 Misspellings and alternative representations

We've covered detecting domain-specific phrases and finding related phrases, but there are two very important sub-categories of related phrases that typically require special handling: misspellings and alternative spellings. When entering queries, users will commonly misspell their keywords, and the general expectation is that an AI-powered search system will be able to understand and properly handle those misspellings.

Whereas some general related phrases for "laptop" might be "computer", "netbook", or "tablet", misspellings would look more like "latop", "laptok", or "lapptop". Alternate spellings are functionally no different than misspellings, but occur when multiple valid variations for a term exist (such as "specialized" vs. "specialised" or "cybersecurity" vs. "cyber security"). In the case of both misspellings and alternative spellings, the end goal is usually to normalize the less common variant into the more common, canonical form and then search for the canonical version.

Spell checking can be implemented in multiple ways. In this section, we'll cover out-of-the box document-based spell checking that is found in most search engines, and we'll also show how user signals can be mined to find more fine-tuned spelling corrections based upon real user interactions with your search engine.

6.5.1 Learning spelling corrections from documents

Most search engines contain some form of spell checking capabilities out of the box based upon the terms found within a collection's documents. Apache Solr, for example, provides a file-based spell checking component, a dictionary-based spell checking component, and an index-based spellchecking component. The file-based spell checker requires assembling a list of terms that can be spell corrected to, the dictionary-based spellchecking component can build a list of terms to be spell corrected to from fields in an index, and the index-based spell checker can use a field on the main index to spell check against directly without having to build a separate spellchecking index. Additionally, if someone has built a list of spelling correction offline, one can leverage a synonym list in Solr to directly replace or expand any misspellings to their canonical form.

Elasticsearch and OpenSearch have some similar spellchecking capabilities, even allowing specific contexts to be passed in to refine the scope of the spelling suggestions to a particular category or geographical location.

While we encourage you to test out these out-of-the-box spell checking algorithms, they all unfortunately suffer from a major problem: lack of user context. Specifically, anytime a keyword is searched that doesn't appear a minimum number of times in the index, the spell checking component begins looking at all terms in the index that are "off by the minimum number of characters", and they then return the most prevalent keywords in the index that match that criteria. Listing 6.12 shows an example of where Solr's out-of-the-box index-based spellchecking configuration falls short.

Listing 6.13 Using out-of-the-box spelling corrections on documents.

```
collection="products"
query="moden"

request = {
    "params": {
        "q.op": "and",
        "rows": 0,
        "indent": "on"
    },
    "query": query,
}

search_results = requests.post(solr_url + collection + "/spell",
➥json=request).json()
print(json.dumps(search_results["spellcheck"]["collations"], indent=4))
```

Results:

```
[
    "collation",
    {
        "collationQuery": "modern",
        "hits": 42,
        "misspellingsAndCorrections": [
            "moden",
            "modern"
        ]
    },
    "collation",
    {
        "collationQuery": "model",
        "hits": 40,
        "misspellingsAndCorrections": [
            "moden",
            "model"
        ]
    },
    "collation",
    {
        "collationQuery": "modem",
        "hits": 29,
        "misspellingsAndCorrections": [
            "moden",
            "modem"
        ]
    },
    "collation",
    {
        "collationQuery": "modena",
        "hits": 1,
        "misspellingsAndCorrections": [
            "moden",
            "modena"
        ]
    },
    "collation",
    {
        "collationQuery": "modes",
        "hits": 1,
        "misspellingsAndCorrections": [
            "moden",
            "modes"
        ]
    }
]
```

In Listing 6.12, you can see a user query for `moden`. The spell checker returns back the suggested spelling corrections of `modern`, `model`, and `modem`, plus a few other suggestions that only appear in a single document and which we'll ignore. Since our collection is tech products, it may be obvious which of these is likely the best spelling correction: it's `modem`. In fact, it would be pretty unlikely that a user would intentionally search for `modern` or `model` as standalone queries, as those are both fairly generic terms that would only make sense within a context containing other words.

The content-based index has no way to distinguish easily that end users would be unlikely to actually search for `modern` or `model`, though, despite those terms occurring many times in the documents being searched. As such, while the content-based spell checkers can work quite well in many cases, it can often be more accurate to learn spelling corrections from users' query behavior.

6.5.2 Learning spelling corrections from user signals

Returning to our core thesis from section 6.3 that users tend to search for related queries until they find the expected results, it is obvious that a user who misspelled a particular query and received bad results would try to then correct their query.

We already know how to find related phrases (section 6.3), but in this section we'll cover how to specifically distinguish a misspelling from user signals. This task largely comes down to two goals:

1. Find terms with similar spellings.
2. Figure out which term is the correct spelling vs. he misspelled variant.

For this task, we'll rely solely on query signals, though perform some up-front normalization to make the query analysis case insensitive and to limit to avoid signal spam. You can read more about this kind of signals normalization in sections 8.2-8.3. Listing 6.13 shows this query to grab our normalized query signals.

Listing 6.14 Get all queries from the user signals, removing duplicates per user and ignoring case

```
query_signals = spark.sql("""
  select lower(searches.target) as keyword,   ❶
  searches.user as user
  from signals as searches where searches.type='query'
  group by keyword, user"""   ❷
  ).collect()
```

❶ Lowercasing the queries makes the query analysis ignore uppercase vs. lowercased variants

❷ Grouping by user prevents spam from a single user entering the same query many times

For purposes of this section, we're going to assume that the queries contain multiple different keywords and that we want to treat each individual keywords as a potential spelling variant. This will allow individual terms to be found an substituted within a future query, as opposed to treating the entire query as a single phrase. It will also allow us to throw out certain terms that are likely to be noise, such as stopwords or standalone numbers.

Listing 6.14 demonstrates the process of tokenizing each query to generate a word list upon which we can do further analysis.

Listing 6.15 Tokenize and filter the terms within each query to generate a cleaned up word list

```
stop_words = set(stopwords.words('english'))   ❶
word_list = defaultdict(int)

for row in query_signals:
    query = row["keyword"]
    tokenizer = RegexpTokenizer(r'\w+')   ❷
    tokens   = tokenizer.tokenize(query)   ❷

    for token in tokens:
        if token not in stop_words and len(token) > 3 and not
        ➥token.isdigit():    ❸
            word_list[token] += 1   ❹
```

❶ Define stopwords which shouldn’t be considered as misspellings or corrections

❷ Split the query on whitespace into individual terms

❸ Remove noisy terms including stopwords, very short terms, and numbers

❹ Count the occurrences of each remaining token

Once the list of tokens has been cleaned up, the next step is to determine high-occurrency tokens versus infrequently-occurring tokens. Since misspellings will occur relatively infrequently and correct spellings will occur more frequently, we will use the relative occurrences to determine which version is the most likely canonical spelling and which variations are the misspellings.

In order to ensure our spell correction list is as clean as possible, we’ll set some thresholds for popular terms and some thresholds for low-occurrence terms that are more likely misspellings. Because some collections may contain hundreds of documents and other collections could contain millions, we can’t just look at an absolute number for these thresholds, so we’ll use quantiles instead. Listing 6.15 shows the calculations for each of the quatiles between 0.1 and 0.9. You can imagine the quantiles like a bell curve, where 0.5 shows the median number of searches for a term overall, the 0.2 shows the number of searches for the term for which 20% of other terms occur less frequently.

Listing 6.16 Get all queries from the user signals, removing duplicates per user

```
quantiles_to_check = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
quantile_values = np.quantile(np.array(list(word_list.values())),
➥quantiles_to_check)
quantiles = dict(zip(quantiles_to_check, quantile_values))
quantiles
```

Results:

```
{0.1: 5.0,
 0.2: 6.0,
 0.3: 8.0,
 0.4: 12.0,
 0.5: 16.0,
 0.6: 25.0,
 0.7: 47.0,
 0.8: 142.20000000000027,
 0.9: 333.2000000000007}
```

Here, we see that 80% of terms are searched for 142.2 times or less (so 20% of terms are searched for this many times or more). Likewise only 20% of terms are searched for 6.0 times or less. Using the Pareto principle, let’s assume that the most of our misspellings fall within the least-searched 20% of our terms and that the majority of our most important terms fall within the top 20% of searched queries. If you want higher precision (only spellcheck high-value terms and only correct if there’s a low probability of false positives), you can push these to the 0.1 quartile for misspellings and the 0.9 quartile for for correctly-spelled terms, or you can go the other direction to attempts to generate a larger misspelling list with a higher change of false positives.

In Listing 6.16, we’ll divide the terms into buckets, assigning low-frequency terms to the "misspell_candidates" bucket and the high-quantity terms to the "correction_candidates" bucket. These buckets will he be starting point for finding high-quality spelling corrections when enough users search for both the misspelling candidate and the correction candidate.

Listing 6.17 Get all queries from the user signals, removing duplicates per user

```
misspell_candidates = []
correction_candidates = []
misspell_counts = []
correction_counts = []
misspell_length = []
correction_length = []
misspell_initial = []
correction_initial = []

for k, v in word_list.items():
    if v <= quantiles[0.2] :   ❶
        misspell_candidates.append(k)   ❶
        misspell_counts.append(v)   ❷
        misspell_length.append(len(k))   ❸
        misspell_initial.append(k[0])   ❹
    if v >= quantiles[0.8]:   ❺
        correction_candidates.append(k)   ❺
        correction_counts.append(v)   ❺
        correction_length.append(len(k))   ❺
        correction_initial.append(k[0])   ❺
```

❶ Terms at or below the 0.2 quantile are added to the `misspell` candidates list

❷ The number of searches is kept to keep track of popularity

❸ The length of the term will be used later to set thresholds for edit distance calculations

❹ the first letter of the term is stored to limit the scope of the misspellings checked

5 The top 20% of terms has the same data stored, but in the `correction_candidates` list

In order to efficiently compare all of the `misspell_candidates` and the `correction_candidates`, we'll first load them into dataframes in Listing 6.17. You can imagine that the `correction_candidates` is a pristine list of the most popular searched terms. The more interesting list is the `misspell_candidates` list, which upon an initial sampling (also in Listing 6.14) should provide a good sense of how many of the less-commonly-searched terms actually represent misspellings.

Listing 6.18 Get all queries from the user signals, removing duplicates per user

```
misspell_candidates_df = pd.DataFrame({
  "misspell":misspell_candidates,
  "misspell_counts":misspell_counts,
  "misspell_length":misspell_length,
  "initial":misspell_initial})

correction_candidates_df = pd.DataFrame({
  "correction":correction_candidates,
  "correction_counts":correction_counts,
  "correction_length":correction_length,
  "initial":correction_initial})

misspell_candidates_df.head(10)
```

Results

```
    misspell    misspell_counts       misspell_length        initial
    --------    ---------------     ---------------     -------
0    misery        5                  6                  m
1    mute        6                  4                  m
2    math        6                  4                  m
3    singin        6                  6                  s
4    thirteen    5                  8                  t
5    nintendogs    6                  10                  n
6    livewire    6                  8                  l
7    viewer        6                  6                  v
8    gorillaz    6                  8                  g
9    tosheba        6                  7                  t
```

You can see in Listin 6.17 that of the 10 samples misspelling candidates, at least three ("singin", "nintendogs", and "tosheba") are clearly misspellings. You can also see that the terms vary in misspell_length, and that the longer the term is the more likely it is to have multiple incorrect characters. For example, `nintendogs` is a misspelling of `nintendo`, so it is off by two characters, whereas if "singin" and "tosheba" are both less than 8 characters, and only off by one character.

When we compare misspell candidates with correction candidates, it is important that we consider the term lenght when deciding how many character differences (or edit distances) are allowed. Listing 6.18 shows a `good_match` function which we'll use as a general heuristic for how many edit distances a term match can be off by while still considering the misspelling a likely permutation of the correction candidate.

Listing 6.19 Get all queries from the user signals, removing duplicates per user

```
def good_match(len1, len2, edit_dist): #allow longer words have more
➥edit distance
    match = 0
    min_length = min(len1, len2)
    if min_length < 8:
        if edit_dist == 1: match = 1
    elif min_length < 11:
        if edit_dist <= 2: match = 1
    else:
        if edit_dist == 3: match = 1
    return match
```

With our misspell_candidates and our correction_candidates loaded into dataframes and our `good_match` function ready to go, it's time to actually generate our spelling correction list. Just like in section 1.6.1 where spelling corrections were generated from edit distances and counts of term occurrences within our collection of documents, in Listing 6.19 we'll be generating spelling corrections based upon edit distances and term occurrences within out query logs.

Listing 6.20 Generate final list of misspellings mapped to corrections

```
matches_candidates = pd.merge(misspell_candidates_df, correction_candidates_df, on="initial")  ❶

matches_candidates["edit_dist"] = matches_candidates.apply(lambda row:
➥nltk.edit_distance(row.misspell,row.correction), axis=1)  ❷
matches_candidates["good_match"] = matches_candidates.apply(lambda row:
➥good_match(row.misspell_length, row.correction_length, row.edit_dist),
➥axis=1)  ❸

matches = matches_candidates[matches_candidates["good_match"] == 1].drop(
➥["initial","good_match"],axis=1)

matches_final.sort_values(by=['correction_counts'], ascending=[False])[
➥["misspell", "correction", "misspell_counts", "correction_counts",
➥"edit_dist"]].head(20)  ❹
```

❶ Join the misspell list with the correction list based on whether they start with the same initial character. This drastically reduces matching time.

❷ Calculated the edit distance between each misspell candidate and correction candidate

❸ Apply the `good_match` function based upon the lengths of the terms and the edit distance

❹ Sample 20 of the results sorted from highest popularity to lowest

Results:

```
    misspell    correction        misspell_counts    correction_counts    edit_dist
81    latop         laptop            5               14258              1
156    touxhpad    touchpad        5               11578              1
21    cape        case            5               7596              1
85    loptops        laptops            5               5628              1
11    bluetooh    bluetooth        5               4499              1
178    wats        wars            5               4196              1
77    kimdle        kindle            5               4159              1
96    moden        modem            5               3598              1
106    phono        phone            5               3073              1
111    poker        power            5               2993              1
159    transfomer    transformers    5               2889              2
95    mircosoft    microsoft        5               2265              2
141    sistem        system            5               2185              1
97    motorla        motorola        5               2022              1
12    blur        blue            5               1916              1
177    walls        wall            5               1818              1
167    turttle        turtle            5               1583              1
135    share        sharp            5               1531              1
84    logictech    logitech        5               1488              1
151    teater        theater            5               1430              1
```

As you can see, we now have a relatively clean list of spelling corrections based upon user signals as opposed to just term occurrences with documents. In fact, our query earlier of `moden` now maps correctly to `modem` as opposed to unlikely search terms like `model` and `modern` like we saw in the document-based spelling correction in Listing 6.12.

There are numerous other ways that one could go about creating a spelling correction model. If you wanted to generate multi-term spelling corrections from documents, you could generate bigram and trigrams and perform chained Bayesian analysis on probabilities of consecutive terms occurring. Likewise to generate multi-term spelling correction from query signals, you could simply choose not to tokenize queries before applying the algorithm in this section, and then you'd be spell correcting an entire query instead of the individual terms. For example, if you replace Listing 6.14 with Listing 6.20, then you'll get spelling corrections for entire queries as opposed to just individual keywords.

Listing 6.21 Remove tokenization of queries, and find misspellings on entire query strings

```
stop_words = set(stopwords.words('english'))
word_list = defaultdict(int)

for row in query_signals:
    query = row["keyword"].strip()

    if query not in stop_words and len(query) > 3 and not query.isdigit():
        word_list[query] += 1

#run Listing 16.12-16.15 again...
...
matches_final.sort_values(by=['correction_counts'], ascending=[False])[
➥["misspell", "correction", "misspell_counts", "correction_counts",
➥"edit_dist"]].head(20)
```

Results:

```
                misspell        correction        misspell_counts
              ➥correction_counts    edit_dist
181                ipad.            ipad            6                    7
➥749                 1
154                hp touchpad 32    hp touchpad        5
➥7144                  3
155                hp toucpad        hp touchpad        6
➥7144                  1
153                hp tochpad        hp touchpad        6
➥7144                  1
190                iphone s4        iphone 4s        5
➥4642                  2
193                iphone4 s        iphone 4s        5
➥4642                  2
194                iphones 4s        iphone 4s        5
➥4642                  1
412                touchpaf        touchpad        5
➥4019                  1
406                tochpad            touchpad        6
➥4019                  1
407                toichpad        touchpad        6
➥4019                  1
229                latop            laptop            5
➥3625                  1
228                laptopa            laptops            6
➥3435                  1
237                loptops            laptops            5
➥3435                  1
205                ipods touch        ipod touch        6
➥2992                  1
204                ipod tuch        ipod touch        6
➥2992                  1
165                i pod tuch        ipod touch        5
➥2992                  2
173                ipad 2            ipad 2            6
➥2807                  1
215                kimdle            kindle            5
➥2716                  1
206                ipone            iphone            6
➥2599                  1
192                iphone3            iphone            6
➥2599                  1
```

Note that the single-term words are largely the same, but now multi-word queries have also been spell corrected. This is a great way to normalize product names such as `iphone4 s`, `iphone4 s` and `iphone s4` all being correctly mapped to the canonical `iphone 4s`. Note that in some cases this can be a lossy process, though, as `hp touchpad 32` maps to `hp touchpad`, and `iphone3` maps to `iphone`. As such, depending on your use case, you may find it beneficial to only spell correct individual terms, or otherwise to include special handling in your `good_match` function for brand variations to ensure the spellcheck code doesn't mistakenly delete relevant query context.

6.6 Pulling it all together

In this chapter, we dove deeper into understand the context and meaning of domain-specific language. We showed how to use semantic knowledge graphs to classify queries and disambiguate terms that have different or nuanced meanings based upon the context. We also explored how to mine relationships from user signals, which usually provides a better context for understanding your users than looking at your documents alone. We also showed how to extract phrases, misspellings, and alternate spellings from query signals, enabling domain-specific terminology to be learned directly from users as opposed to only from documents.

At this point, you should feel confident being able to learn domain specific phrases from documents or use signals, to learn related phrases from documents or user signals, to classify queries to your available content, and to disambiguate the meaning of terminology based upon the query classification. Each of these techniques are now critical tools in your toolbox for interpreting query intent.

Our goal isn’t just to assemble a large toolbox, though. Our goal is to leverage each of these tools where appropriate to build an end-to-end semantic search layer. This means we need to model known phrases into our knowledge graph, to be able to extract those phrases from incoming queries in real-time, to handle misspellings, query classifications, and disambiguation on the fly for incoming terms, and to ultimately generate a rewritten query to the search index that uses each of our AI-powered search techniques. In the next chapter, we’ll show you how to assemble each of these techniques into a working semantic search system designed to best interpret and model query intent.

6.7 Summary

- Classifying queries using a semantic knowledge graph can help interpret query intent and improve query routing and filtering
- Query-sense disambuguation can help determine nuanced understanding of a user’s query, particularly for terms with significantly divergent meanings across different contexts.
- In addition to learning from douments, domin-specific phrases and related phrases can also be learned from user signals.
- Misspellings and alternate spelling variations can be learned from both documents and from user signals, with document-based approaches being more robust, but user-signal-based approaches better representing user intent.

8 Signals boosting models

This chapter covers

- Aggregating user signals to create a popularity-based ranking model
- Normalizing signals to best enhance relevance for noisy query input
- Fighting signal spam and user manipulation of crowdsourced signals
- Applying time decays to prioritize recent signals as more relevant
- Blending multiple signal types together into a unified signals boosting model
- Scaling signals boosting for flexibility and performance using query time vs. index-time signals boosting.

In chapter 4, we covered three different categories of reflected intelligence: Signals Boosting (popularized relevance), Collaborative Filtering (personalized relevance), and Learning to Rank (generalized relevance). In this chapter, we'll dive deeper into the first of these, implementing Signals Boosting to enhance the relevance ranking of your most popular queries and documents.

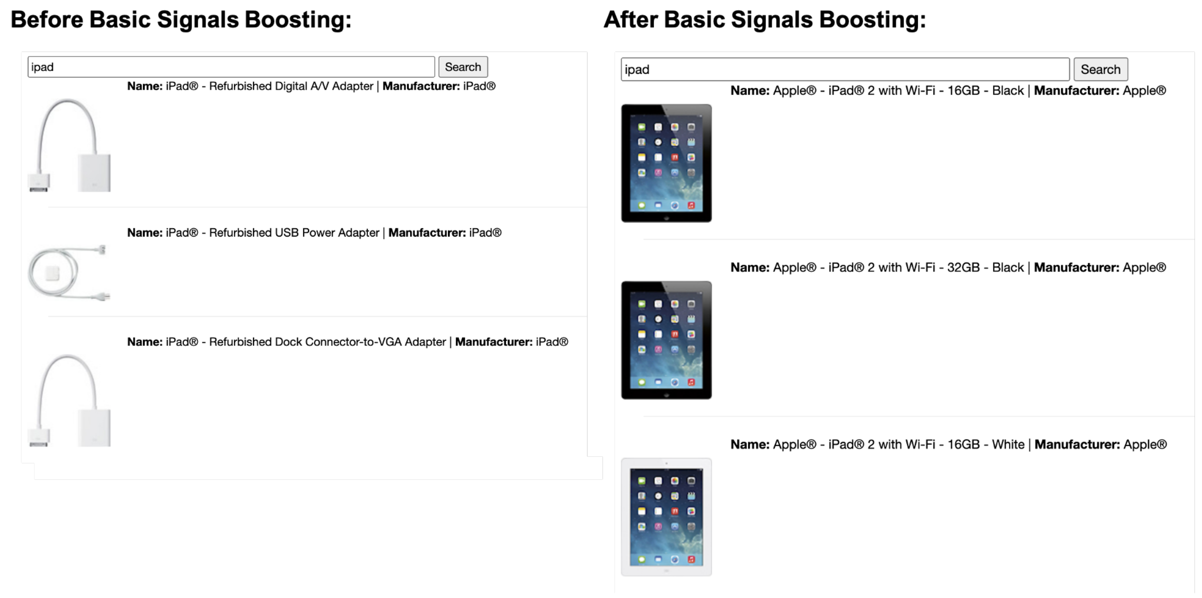
In most search engines, you will find that a relatively small number of queries tend to make up a large portion of your total query volume. These popular queries, called *head queries*, also tend to lead to more signals (such as clicks and purchases in an ecommerce use case), which enable stronger inferences about the popularity of top search results.

Signals boosting models directly harness these stronger inferences and are the key to ensuring your most important and highest-visibility queries are best tuned to return the most relevant documents.

8.1 Basic signals boosting

In section 4.2.2, we built our first signals boosting model on the Retrotech dataset, enabling a significant boost in relevance for the most frequently searched and clicked search results. In this section, we'll quickly recap the process of creating a simple signals boosting model, which we will build upon in the upcoming sections to cater to some more advanced needs.

You'll recall from section 4.2.2 that signals boosting models aggregate all useful activity signals on documents (such as click signals) that occur as the result of a specific query. We used a search for `ipad`, and boosted each document based upon how many total times it was previously clicked in the results for that searc. Figure 8.1 demonstrates the before (no signal boosting) and after (signals boosting on) search results for the query `ipad` previously demonstrated in section 4.3.2.



Before Basic Signals Boosting:

ipad [Search]

Name: iPad® - Refurbished Digital A/V Adapter | **Manufacturer:** iPad®

Name: iPad® - Refurbished USB Power Adapter | **Manufacturer:** iPad®

Name: iPad® - Refurbished Dock Connector-to-VGA Adapter | **Manufacturer:** iPad®

After Basic Signals Boosting:

ipad [Search]

Name: Apple® - iPad® 2 with Wi-Fi - 16GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 32GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 16GB - White | **Manufacturer:** Apple®

Figure 8.1 Before and after applying a signals boosting model. Signals boosting improves relevance by pushing the most popular items to the top of the search results.

The signals boosting model that led to the improved relevance in Figure 8.1 is a basic signals boosting model. It looks at all documents ever clicked for a given query, and then applies a boost equal to the total number of past clicks on that document for that query.

While the basic signal boosting model covered in section 4.3.2 provides greatly improved relevance, it is unfortunately susceptible some data biases and even manipulation. In section 8.2, we'll discuss some techniques for removing noise in the signals to maximize the quality your signals boosting models and reduce the opportunity for undesirable biases.

8.2 Normalizing Signals

It is important to normalize incoming user queries prior to aggregation so that variations are treated as the same query. Given that end users can enter any arbitrary text as a query, this means that the aggregated signals are inherently noisy. The basic signals boosting model from chapter 4 (and recapped in section 8.1) does no normaliation. It generates aggregated boosts for each query and document pair, but since incoming queries haven't been normalized into a common form, this means that variations of a query will be treated as entirely separate queries. Listing 8.1 demonstrates a list of all queries that boosted the most popular iPad model in their search results.

Listing 8.1 Find the most popular queries associated with the most popular iPad model

```
query = "885909457588" #most popular iPad model

def show_raw_boosted_queries(signals_boosting_collection):
    signals_boosts_query = {
        "query": "\"" + query + "\"",
        "fields": ["query", "boost"],
        "limit": 20,
        "params": {
          "defType": "edismax",
          "qf": "doc",
          "sort": "boost desc"
        }
    }

    signals_boosts = requests.post(solr_url + signals_boosting_collection
                     + "/select", json=signals_boosts_query).json()[
                     ➥"response"]["docs"]

    boosted_queries = ""
    for entry in signals_boosts:
        boosted_queries += '"' + entry['query'] + '" : ' +
        ➥str(entry['boost']) + "\n"

    print("Raw Boosted Queries")
    print(boosted_queries)

signals_boosting_collection = "basic_signals_boosts"
show_raw_boosted_queries(signals_boosting_collection)
```

Results:

```
Raw Boosted Queries
"iPad" : 1050
"ipad" : 966
"Ipad" : 829
"iPad 2" : 509
"ipad 2" : 347
"Ipad2" : 261
"ipad2" : 238
"Ipad 2" : 213
"I pad" : 203
"i pad" : 133
"IPad" : 77
"Apple" : 76
"I pad 2" : 60
"apple ipad" : 55
"Apple iPad" : 53
"ipads" : 43
"tablets" : 42
"apple" : 41
"iPads" : 38
"i pad 2" : 38
```

You can see from the output of Listing 8.1 that many variations of the same queries exist in the basic signals boosting model. The biggest culprit of the variations seems to be case-sensitivity, as we see `iPad`, `ipad`, `Ipad`, and `IPad` as common variants. Spacing appears to be another issue, with `ipad 2` vs `i pad 2` vs. `ipad2`. We even see singular vs. plural representations in `ipad` vs. `ipads`.

Given that most keyword search fields are case insensitive, and that many also ignore plural representations of terms and split on case changes and letter to number transitions between words, keeping separate query terms and boosts for variations that are non-distinguishable by the search engine can be counter productive. Not only is it unnecessary, but is actually diffuses the value of your signals, since the signals are divided across different variations of the same keywords with lower boosts as opposed to being coalesced into more meaningful queries with stronger boosts.

It is up to you to figure out how sophisticated your query normalization should be prior to signals aggregation, but even just lowercasing incoming queries to make the signals aggregation case insensitive can go a long way. Listing 8.2 demonstrates the same basic signals aggregation as before, but this time with the queries lowercased first.

Listing 8.2 Basic case-insensitivity normalization of boosted queries. Now "iPad", "Ipad", and "IPad" be treated as the same query and share signals boosts.

```
signals_collection = "signals"
signals_boosting_collection = "normalized_signals_boosts"

normalized_signals_aggregation_query = """
      select lower(q.target) as query,   ❶
          c.target as doc,
          count(c.target) as boost   ❷
        from signals c left join signals q on c.query_id = q.query_id
        where c.type = 'click' AND q.type = 'query'
        group by query, doc   ❷
        order by boost desc
        """

aggregate_signals(signals_collection, signals_boosting_collection,
➥normalized_signals_aggregation_query)

show_raw_boosted_queries(signals_boosting_collection)
```

❶ Normalizing case by lowercasing each query

❷ Grouping by normalized query increases the count of signals for that query, increasing the signal boost

Results:

```
Raw Boosted Queries
"ipad" : 2939
"ipad 2" : 1104
"ipad2" : 540
"i pad" : 341
"apple ipad" : 152
"ipads" : 123
"apple" : 118
"i pad 2" : 99
"tablets" : 67
"tablet" : 61
"ipad 1" : 52
"apple ipad 2" : 27
"hp touchpad" : 26
"ipaq" : 20
"i pad2" : 19
"wi" : 19
"apple computers" : 18
"apple i pad" : 15
"ipad 2 16gb" : 15
"samsung galaxy" : 14
```

That list of raw boosted queries is already looking much cleaner! Not only is there less redundancy, but you'll notice that the strength of the signals boosts has increased, because more signals are being attributed to a canonical form of the query (the lowercased version).

Often just lowercasing the queries, and maybe removing whitespace or extraneous characters, is sufficient normalization of queries prior to signals aggregation. The important takeaway from this section, though, is that the signals boosting model becomes stronger the better you are able to ensure that identical queries are treated identically when they are aggregated together.

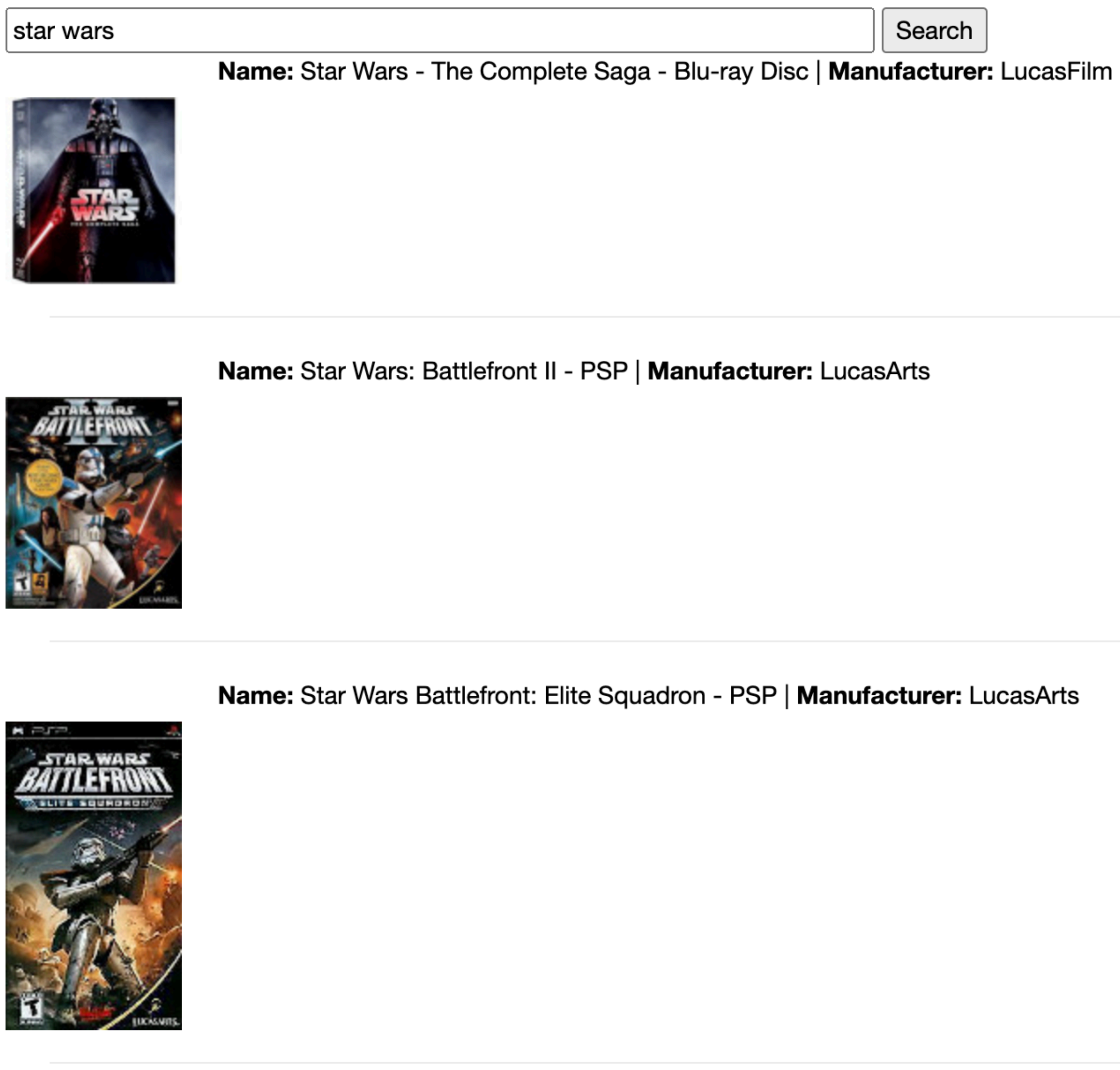
Variations in queries aren’t the only kind of noise we need to worry about in our data, however. In the next section, we’ll talk about how to overcome significant potential problems caused by spam in our user-generated click signals.

8.3 Fighting Signal Spam

Anytime we use crowdsourced data, such as click signals, to influence the behavior of the search engine, we need to ask ourselves "How might our users manipulate the data inputs to create an undesirable result?". In this section, we’ll demonstrate how a user could spam the search engine with click signals to manipulate search results, and we’ll show you how to stop it.

8.3.1 Using signal spam to manipulate search results

Let’s imagine we have a user who, for whatever reason, really hates Star Wars and thinks that the most recent movies are complete garbage. They feel so strongly, in fact, they they want to ensure any searches for `star wars` always return a physical trash can for purchase as the top search result. This user knows a thing or two about search engines and has noticed that your killer relevance algorithms seem to be leveraging user signals and signals boosting. Figure 8.2 shows the default response for the query `star wars`, with signals boosting bringing the most popular products to the top of the search results.



star wars

Search

Name: Star Wars - The Complete Saga - Blu-ray Disc | **Manufacturer:** LucasFilm

Name: Star Wars: Battlefront II - PSP | **Manufacturer:** LucasArts

Name: Star Wars Battlefront: Elite Squadron - PSP | **Manufacturer:** LucasArts

Figure 8.2 The most popular search results for the query "star wars", with signals boosting turned on. These are the expected results when there is no malicious signal spam.

The user decides that since your search engine ranking is based upon popular items, that they will spam the search engine with a bunch of searches for `star wars` and follow up with a bunch of fake clicks on the Star Wars themed trash can they found, in order to try to make the trash can show up at the top of the search results.

In order to simulate this scenario, we’ll run a simple script in Listing 8.3 to generate 5000 queries for `star wars` and 5000 corresponding clicks on the trash can after running that query.

Listing 8.3 Generating spam queries and clicks to manipulate the ranking of a document due to signals boosting.

```
import datetime

spam_user = "u8675309"
spam_query = "star wars"

spam_signal_boost_doc_upc = "45626176"   ❶

num = 0
while (num < 5000):  ❷
    query_id = "u8675309_0_" + str(num)

    next_query_signal = {
        "query_id": query_id,
        "user": spam_user,
        "type":"query",
        "target": spam_query,
        "signal_time": datetime.datetime.now().strftime(
        ➥"%Y-%m-%dT%H:%M:%SZ"),
        "id":"spam_signal_query_" + str(num)
    }

    next_click_signal = {
        "query_id": query_id,
        "user": spam_user,
        "type":"click",
        "target": spam_signal_boost_doc_upc,
        "signal_time": datetime.datetime.now().strftime(
        ➥"%Y-%m-%dT%H:%M:%SZ"),
        "id":"spam_signal_click_" + str(num)
    }

    collection = "signals"
    requests.post(solr_url + collection + "/update/json/docs",
    ➥json=next_query_signal)  ❷
    requests.post(solr_url + collection + "/update/json/docs",
    ➥json=next_click_signal)  ❷
    num+=1

requests.post(solr_url + collection + "/update/json/docs?commit=true")  ❸

signals_collection = "signals"
signals_aggregation_collection = "signals_boosts_with_spam"
aggregate_signals(signals_collection, signals_aggregation_collection,
➥normalized_signals_aggregation_query)  ❹
```

❶ Document for trash can the spammer wants to move to the top of the search results

❷ Send 5,000 query and click signals to the search engine

❸ Commit the signals to the index

❹ Run the signals aggregation to generate the signals boosting model including the spammy signals

Listing 8.3 sends thousands of spammy query and click signals to our search engine, modeling the same outcome we would see if a user searched and clicked on a particular search result thousands of times. The listing then re-runs the basic signals aggregation to see the impact those signals have on our signals boosting model.

To see the impact on our search results, Listing 8.4 runs a search for the query `star wars`, now incorporating the manipulated signals boosting model in order to see the effect of the malicious user’s spammy click behavior.

Listing 8.4 Search results for query "star wars" using the manipulated signals boosting model

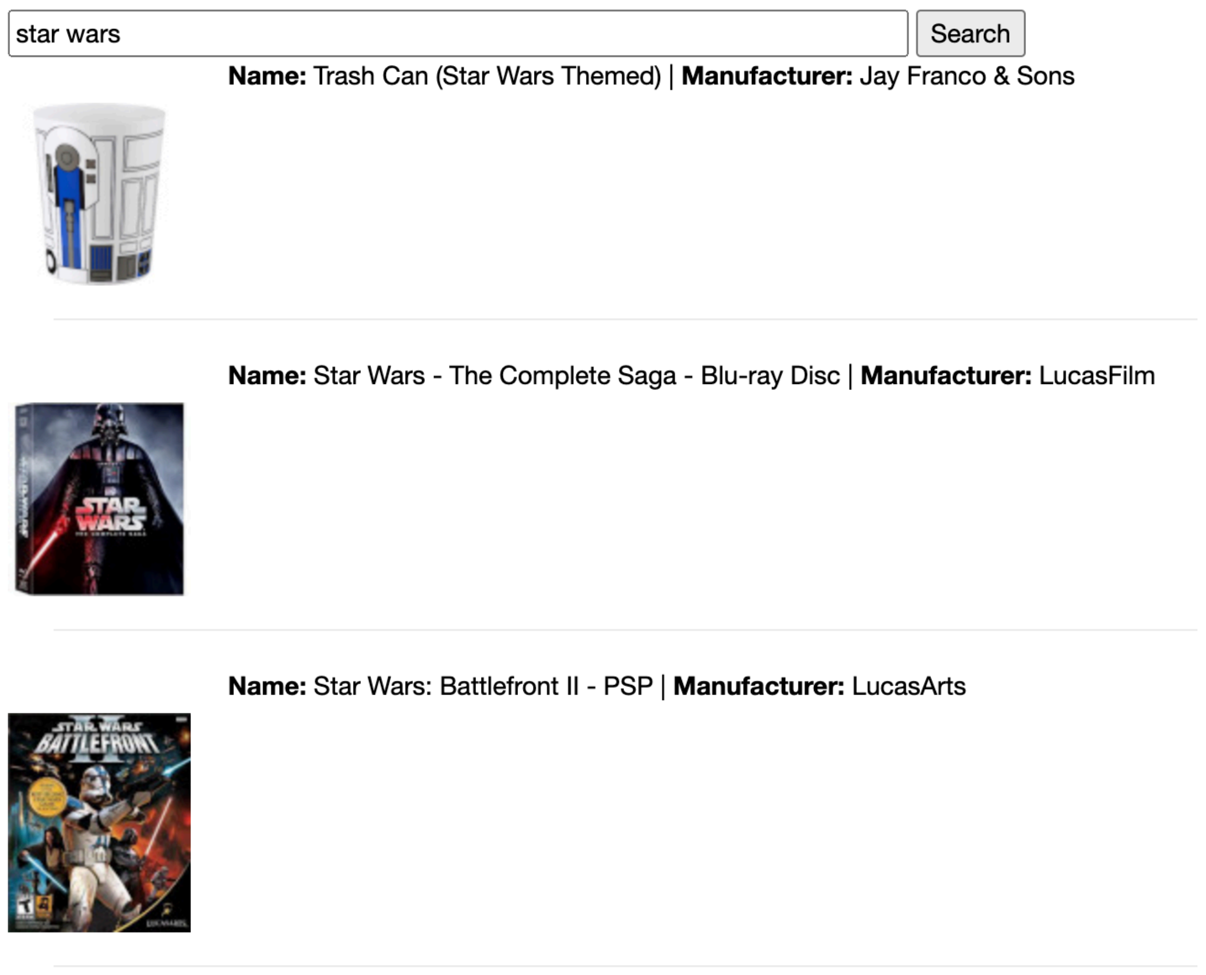
```
query = "star wars"
collection = "products"

signals_boosts = get_query_time_boosts(query, "signals_boosts_with_spam")   ❶
boosted_query = get_main_query(query, signals_boosts)    ❷

search_results = requests.post(solr_url + collection + "/select",
➥json=boosted_query).json()["response"]["docs"]
print(search_results)
display(HTML(render_search_results(query, search_results)))   ❸
```

❶ Load signals boosts from the signals boosting model that included the spammy signals

❷ Boost the "star wars" using the signals boosting model

❸ Display the search results

Figure 8.3 shows the new manipulated search results generated from Listing 8.4, with the Star Wars trash can returned in the top spot.



star wars

Name: Trash Can (Star Wars Themed) | **Manufacturer:** Jay Franco & Sons

Name: Star Wars - The Complete Saga - Blu-ray Disc | **Manufacturer:** LucasFilm

Name: Star Wars: Battlefront II - PSP | **Manufacturer:** LucasArts

Figure 8.3 Search results manipulated by a user spamming the search engine with fake signals to affect the top result. The user was able to modify the top result just by clicking on it many times.

The spammer was successful, and these manipulated search results will now be seen by every subsequent visitor to the Retrotech website who searches for `star wars`! Looks like we're going to need to make our signals boosting model more robust to combat this kind of signal spam from malicious users.

8.3.2 Combatting signal spam through user-based filtering

If you are going to use crowdsourced data like user signals to influence your search engine ranking, then it is important to take steps to minimize the ability for your users to manipulate your signals-based ranking algorithm.

In order to combat the "Star Wars trash can" problem we just demonstrated, the simplest technique to start would be to ensure that duplicate clicks by the same user only get one "vote" in the signals boosting aggregation. That way, whether a malicious user clicks one time or a million times, their clicks only count as one signal and therefore have no material impact on the signals boosting model. Listing 8.5 reworks the signals aggregation query to only count unique click signals from each user.

Listing 8.5 Deduplicating signals per user to prevent undue influence by a single user

```
signals_collection = "signals"
signals_aggregation_collection = "signals_boosts_anti_spam"

anti_spam_aggregation_query = """
  select query, doc, count(doc) as boost from (
    select c.user, lower(q.target) as query, c.target as doc,
    max(c.signal_time) as boost   ❷
    from signals c left join signals q on c.query_id = q.query_id
    where c.type = 'click' AND q.type = 'query'
    group by c.user,   ❶
      q.target, c.target
  ) as x
  group by query, doc
  order by boost desc
"""

aggregate_signals(signals_collection, signals_aggregation_collection, anti_spam_aggregation_query)
```

❶ Group by user the limit each user to only one "vote" per query/doc pair in the signals boosting model

❷ Signal date is the most recent signal from the user only if there are duplicates

If we re-run the `star wars` query from Listing 8.5 with this new `signals_boosts_anti_spam` model, we'll now see that our normal search results have returned and look the same again as Figure 8.2. This is because the extra, spammy signals from our malicious user have now all been reduced to a single bad signal, which we show in Table 8.1.

You can see that the aggregated signal count in the "signals_anti_spam" model has a total much closer to the `normalized_signals_boosts` model that we built before the spam signals were generated. Since each user is limited to one signal per query/document pair in the `signals_boosts_anti_spam` model, the ability for users to manipulate the signal boosting model is now substantially reduced.

Table 8.1 The 5000 spammy signals have been deduplicated to one signal in the antispam signal boosting model model

model	query	doc	boost
before spam signals (`normalized_signals_boosts`)	star wars	400032015667	0 (no signals yet)
after spam signals (`normalized_signals_boosts`)	star wars	400032015667	5000
after spam signals (`signals_boosts_anti_spam`)	star wars	400032015667	1

You could, of course, identify any user accounts that appear to be spamming your search engine and remove their signals entirely from your signals boosting aggregation, but reducing the reach of the signals through deduplication is simpler and often accomplishes the same end goal of restoring a good crowdsourced relevance ranking.

In our example from Listing 8.5, we leveraged user IDs as the key identifier to deduplicate spammy signals, but any identifier will work here: user ID, session ID, browser ID, IP address, or even some kind of browser fingerprint. As long as you find some identifier to uniquely identify users or to otherwise identify low-quality traffic (like bots and web scrapers), then you can use that information to deduplicate signals. If none of those techniques work and you have too much noise in your click signals, you can also choose to only look at click signals from known (authenticated) users who you presumably have much more confidence in being legitimate traffic.

One final way to mitigate signal spam is to find a way to separate the important signal types from the noisy ones that can be easily-manipulated. For example, generating signals from running queries and clicking on search results is easy. Signals from purchasing a product are much harder to manipulate, however, as they require users to log in or enter payment information before a purchase will be recorded. The odds of someone maliciously purchasing 5,000 Star Wars trash cans are quite low, because there are multiple financial and logistical barriers to doing this.

Not only is it valuable to weight purchases as stronger signals than clicks from the standpoint of fighting spam, it is also valuable from a relevance standpoint to give purchases a higher weight, because they are more clear indicators of intent than just clicks. In the next section, we'll walk through exactly how to combine different signal types into a signals boosting model that considers the relative importance of each different signal type.

8.4 Combining multiple signal types

Thusfar we've only worked with two signals types - queries and clicks. For some search engines (such as web search engines), click signals may be the only good source of crowd-sourced data available to build a signals boosting model. Often times, however, many different signal types exist that can provide additional and often much better inputs for building a signals boosting model.

In our Retrotech dataset, we have several signal types that are common to ecommerce use cases:

- query
- click
- add-to-cart
- purchase

While clicks in response to queries are helpful, they don't necessarily imply a strong interest in the product, as someone could just be browsing to see what's available. If someone adds a product to their shopping cart, this typically represents a much stronger signal of interest than a click. A purchase is then an even stronger signal that a user is interested in a product, as the user is willing to pay money to receive the item for which they searched.

While some ecommerce websites may receive enough traffic to ignore click signals entirely and only focus on add-to-cart and purchase signals, it is often more useful to make use of all signal types when calculating signals boosts. Thankfully, combining multiple signal types is as simple as just assigning relative weights as multipliers to each signal type when performing the signals aggregation:

```
signals_boost = (1 * sum(click_signals)) + (10 * sum(add_to_cart_signals)) +
➥(25 * sum(purchase_signals))
```

By counting each click as 1 signal, each add-to-cart as 10 signals, and each purchase as 25 signals, this makes each purchase carry 25 times as much weight in the signals boosting model than just a click. In other words, 25 different people would need to click on a product in response to a query to count as much as one person actually purchasing the product as a result of the same query.

This helps reduce noise from less reliable signals and boost more reliable signals, while still making use of the large volume of less reliable signals in cases (like new or obscure items) where better signals are less prevalent. Listing 8.6 demonstrates a signals aggregation designed to combine different signal types with different weights.

Listing 8.6 Combining multiple signal types with different weights

```
signals_collection="signals"
signals_aggregation_collection="signals_boosts_weighted_types"

mixed_signal_types_aggregation = """
select query, doc,
( (1 * click_boost) + (10 * add_to_cart_boost) + (25 * purchase_boost) )
➥as boost   ❶
from (
  select query, doc,
    sum(click) as click_boost,   ❷
    sum(add_to_cart) as add_to_cart_boost,   ❷
    sum(purchase) as purchase_boost   ❷
  from (
      select lower(q.target) as query, cap.target as doc,
        if(cap.type = 'click', 1, 0) as click,
        if(cap.type = 'add-to-cart', 1, 0) as  add_to_cart,
        if(cap.type = 'purchase', 1, 0) as purchase
      from signals cap left join signals q on cap.query_id = q.query_id
      where (cap.type != 'query' AND q.type = 'query')
    ) raw_signals
  group by query, doc
) as per_type_boosts
"""

aggregate_signals(signals_collection, signals_aggregation_collection,
➥mixed_signal_types_aggregation)
```

❶ Multiple signals combined with different relative weights toward to total boost value

❷ Each signal type gets summed up independently before being combined

You can see from the SQL query that the overall boost for each query / document par is calculated by counting all clicks with a weight of 1, counting all add-to-cart signals and multiplying them by a weight of 10, and then counting all purchase signals and multiplying them by a weight of 25.

These suggested weights of 10x for add-to-cart signals and 25x for purchase signals should work well in practice in many ecommerce scenarios, but these relative weights are also fully configurable for each domain. Your website may be set up such that almost everyone who adds a product to their cart purchases the product (for example, a grocery store delivery app, where the only purpose of using the website is to fill a shopping cart and purchase). In these cases, you could find that adding an item to a shopping cart adds no additional value, but that *removing* an item from a shopping cart should actually carry a negative weight indicating the product is a bad match for the query.

In this case, you may want to introduce the idea of *negative signals boosts*. Just as we've discussed clicks, add-to-carts, and purchases as signals of user intent, your user experience may also have numerous ways to measure user dissatisfaction with your search results. For example, you may have a thumbs-down button, a remove from cart button, or you may be able to track product returns after a purchase. You may even want to count documents in the search results which were skipped over, and record a "skip" signal for those documents to indicate the user saw them but didn't show interest. We'll cover the topic of managing clicked versus skipped documents further in chapter 11 when we discuss click modeling.

Thankfully, handling negative feedback is just as easy as handling positive signals: instead of just assigning increasingly positive weights to signals, you can also assign increasingly negative weights to negative signals. For example:

```
positive_signals = (1 * sum(click_signals) ) + ( 10 * sum(
➥add_to_cart_signals) ) + ( 25 * (purchase_signals) ) + ( 0.025 * sum(
➥seen_doc_signals) )

negative_signals = ( -0.025 * sum(skipped_doc_signals) ) + ( -20 * sum(
➥remove_from_cart_signals) ) + ( -100 * sum(returned_item_signals) ) +
➥( -50 * sum(negative_post_about_item_in_review_signals) )

type_based_signal_weight = positive_signals + negative_signals
```

This simple, linear function provides a highly configurable signals-based ranking model, taking in multiple input parameters and returning a ranking score based upon the relative weights of those parameters. You can combine as many useful signals as you want into this weighted signals aggregation to improve the robustness of the model. Of course, tuning the weights of each of the signal types to achieve an optimal balance may take some effort. You can do this manually, or you can leverage a machine learning technique called Learning to Rank to do this. We'll explore Learning to Rank in-depth in chapters 10 and 11.

Not only is it important to weight different kinds of signals relative to each other, but it can also sometimes be necessary to weight the *same* kind of signals differently against each other. In the next section, we'll discuss one key example of doing this: assigning higher value to more recent interactions.

8.5 Time decays and short-lived signals

Signals don't always maintain their usefulness indefinitely. In the last section, we showed how signals boosting models can be adjusted to weight different kinds of signals as more important than others. In this section, we'll address a different challenge - factoring in the "temporal value" of signals as they age and become less useful.

Imagine three different search engine use cases:

- an ecommere search engine with stable products,
- a job search engine, and
- a news website.

If we have an ecommerce search engine, like Retrotech, the documents (products) often stay around for years, and the best products are often those that have a long track record of interest.

If we have a job search engine, the documents (jobs) may only stick around for a few weeks or months until the job is filled, and then they disappear forever. While the documents are present, however, newer clicks or job applications aren't necessarily any more important as signals than older interactions.

In a news search engine, while the news articles stick around forever, newer articles are generally way more important than older articles, and newer signals definitely are more important than older signals, as people's interests change on a daily, if not hourly basis.

Let's dive into these usecases and demonstrate how to best handle signals boosting for time-sensitive documents vs. time-sensitive signals.

8.5.1 Handling time-sensitive documents

In our Retrotech use case, our documents are intentionally old, having been around for a decade or more, and interest in them likely only increases as the products become older and more "retro". As such, we don’t often have massive spikes in popularity for items, and newer signals don’t necessarily carry significantly more importance than older signals. This type of use case is a bit atypical, but plenty of search use cases do deal with more "static" document sets like this. The best solution in this case is the strategy we’ve already taken thusfar in this chapter: to process all signals within a reasonable time period of months or years and give them fairly equal weight. When all time periods carry the same weight, this also means that the signals boosting model likely doesn’t need to be rebuilt that often, since the model only changes slowly over time and the frequent processing of signals is unnecessary computational overhead.

In a job search use case, however, the scenario is very different. For the sake of argument, let’s say that on average it takes 30 days to fill a job opening. This means the document representing that job will only be present in the search engine for 30 days, and that any signals collected for that document are only useful for signals boosting during that 30 days window. When a job is posted, it will typically be very popular for the first few days since it is new and is likely to attract many existing job seekers, but all interactions with that job at any point during the 30 days are just as useful. In this case, all click signals should get an equal weight, and all job application signals should likewise receive an equal weight (at a weight higher than the click signals, of course). Given the very short lifetime of the documents, however, it is important that all signals are used as quickly as possible in order to make the best use of their value.

Use cases with short-lived documents, like in the job search use case, don’t usually make good candidates for signals boosting, as the documents often get deleted by the time the signals boosting model becomes any good. As a result, it can often make more sense to look at personalized models (like collaborative filtering, covered in chapter 9) and generalizable relevance models (like Learning to Rank, covered in chapters 10 and 11) for these use cases instead.

In both the Retrotech use case and the job search use case, the signals were just as useful for the entire duration of the document’s existence. In the news search use case, which we’ll see next, the time sensitivity is more related to the age of the documents and the signals themselves.

8.5.2 Handling time-sensitive signals

In a news search engine use case, the most recently published news gets the most visibility and usually the most interaction, so most recent signals are considerably more valuable than older signals. Some news items may be very popular and relevant for days or longer, but generally the signals from the last ten minutes are more valuable than the signals from the last hour, which are more valuable than the signals from the last day, and so on. News search is an extreme use case where signals both need to be processed quickly and where more recent signals need to be weighted as substantially more important than older signals.

One easy way to model this is by using a decay function, such as a half-life function, which cuts the weight assigned to a signal by half (50%) over equally-spaced time spans. For example, a decay function with a half-life of 30 days would assign 100% weight to a signal that happens `now` , 75% weight to a signal from 15 days ago, 50% weight to a signal from 30 days ago, 25% weight to a signal from 60 days ago, 12.5% weight to a signal from 90 days ago, and so on. The math for implementing a decay function is:

```
time_based_signal_weight = starting_weight * 0.5^(signal_age/half_life)
```

When applying this calculation, the `starting_weight` will usually be the relative weight of a signal based upon it’s type, for example a weight of `1` for clicks, `10` for add-to-cart signals, and `25` for purchase signals. If you are not combining multiple signal types then the `starting_weight` will just be `1`.

The `signal_age` is how old the signal is, and the `half_life` is how long it takes for the signal to lose half of it’s value. Figure 8.4 demonstrates how this decay function impacts signals weights over time for different half-life values.

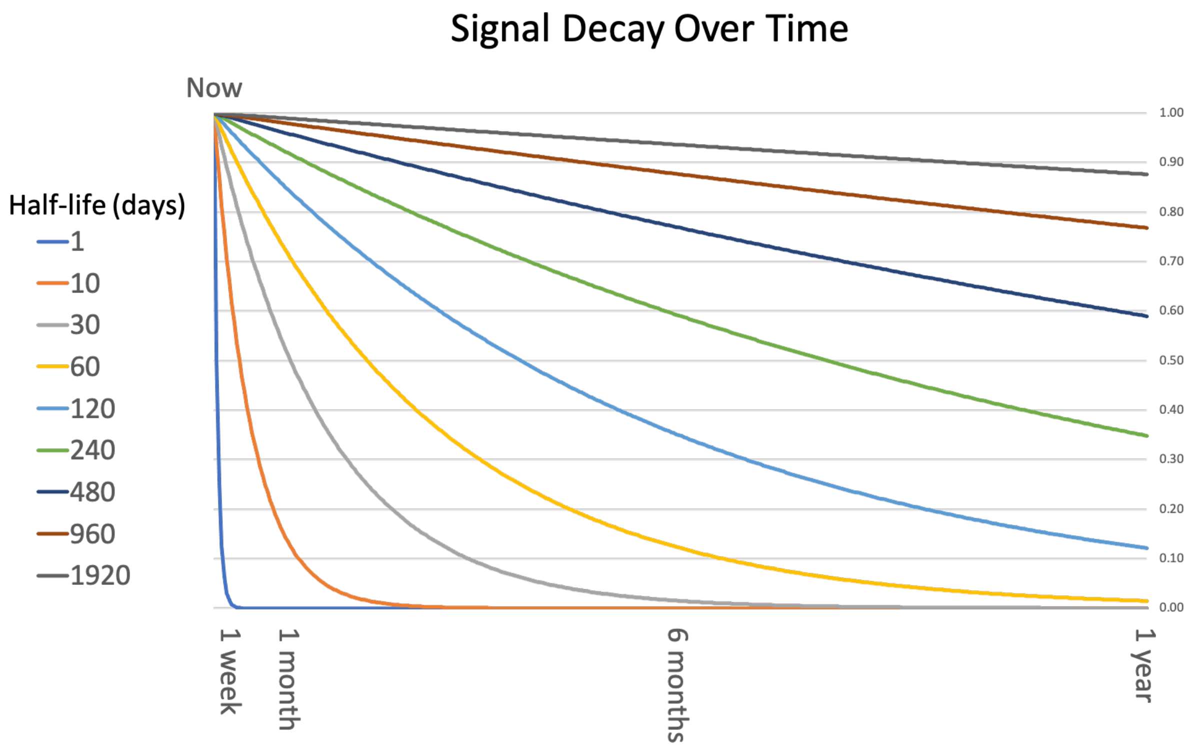


Figure 8.4 Signal decay over time based upon various half life values. As the halflife increases, individual signals maintain their boosting power for longer.

The one day half-life is very aggressive and is pretty impractical in most usecases, as it is unlikely you would be able to collect enough signals in a day to power meaningful signals boosting, and the liklihood of your signals becoming irrelevant that quickly is low.

The 30 day, 60 day, and 120 day half-lifes do a good job of aggressively discounting older signals, but keeping their residual value contributing to the model over a six to twelve month period. If you have really long-lived documents, you could push out even longer, making use of signals over the course of many years. Listing 8.7 demonstrates an updated signal aggregation query that implements a half-life of 30 days for each signal:

Listing 8.7 Appling a time decay function to the signals boosting model

```
signals_collection="signals"
signals_boosting_collection="signals_boosts_time_weighted"

half_life_days = 30
target_date = '2020-06-01 00:00:00.0000' #Will usually be now(), but can be any past time
    ➥you want to emphasize signals from.
signal_weight = 1 #can make this a function to differentiate weights for different signal types

time_decay_aggregation = """
select query, doc, sum(time_weighted_boost) as boost from (
    select user, query, doc, """ + signal_weight + """ * pow(0.5, (
    ➥datediff('""" + target_date + "', signal_time) / "
    ➥+ str(half_life_days) + """)) as time_weighted_boost from (
        select c.user as user, lower(q.target) as query, c.target as doc,
        max(c.signal_time) as signal_time
        from signals c left join signals q on c.query_id = q.query_id
        where c.type = 'click' AND q.type = 'query'
        AND c.signal_time <= '""" + target_date + """'
        group by c.user, q.target, c.target
    ) as raw_signals
) as time_weighted_signals
group by query, doc
order by boost desc
"""

aggregate_signals(signals_collection, signals_boosting_collection,
➥time_decay_aggregation)
```

This decay function has a few unique configurable parameters:

- It contains a `half_life_days` parameter, which calculates a weighted average using a configurable half-life, which we've set as 30 days to start.
- It contains a `signal_weight` parameter, which can be replaced with a function returning a weight by signal type, as shown in the last section (click = 1, add to cart = 10, purchase = 25, etc.).
- It contains a `target_date` parameter, which is the date at which a signal gets the full value of `1`. Any signals before this date will be decayed based upon the half-life, and any signals after this date will be ignored (filtered out).

Your `target_date` will usually be the current date, so that you are making use of your most up-to-date signals and assigning them the highest weight. However, you could also apply it to past periods if your documents have seasonal patterns that repeat monthly or yearly.

While our product documents don't change very often, and the most recent signals aren't necessarily any more valuable than older signals, there are potentially annual patterns we could find in a normal ecommerce data set. For example, certain types of products may tend to be more popular around major holidays like Mother's day, Father's day, and Black Friday. Likewise, searches for something like a "shovel" may take on a different meaning in the summer (shovel for digging dirt) versus the winter (shovel for removing snow from the sidewalk). If you explore your signals, any number of trends may emerge for which time sensitivity should impact how your signals are weighted.

Ultimately, Signals are a lagging indicator. They are a reflection of what your users just did, but they are only useful as predictions of future behavior if the patterns learned are likely to repeat themselves.

Having now explored techniques for improving our signals models through query normalization, mitigating spam and relevance manipulation, combining multiple signal types with different relative weights, and applying time decays to signals, you should be able flexibly implement the signals boosting models most appropriate for your use case. When rolling out signals boosting at scale, however, there are two different approaches you can take to optimize for flexibility versus performance. We'll cover these two approaches in the next section.

8.6 Index-time vs. Query-time boosting: balancing scale vs. flexibility

All the the signals boosting models in the chapter have been demonstrated using *query-time boosting*, which loads signal boosts from a separate `signals_boosts` collection for each user query at query time and modifies the user's query to add the boosts prior to sending it to the search engine. It is also possible to implement boosting models using *index-time boosting*, where boosts are added directly to documents for the queries to which those boosts apply. In this section, we'll discuss the benefits and tradeoffs of each of these approaches.

8.6.1 Tradeoffs when using query-time boosting

Query-time boosting, as we've seen, turns each query into a two step process, where each incoming user query is looked up in the `signals_boosting` collection, and then any found boosted documents are used to modify the user's query. Query-time boosting is the most common way to implement signals boosting, but it comes with both its benefits and drawbacks.

BENEFITS OF QUERY-TIME BOOSTING

Query-time boosting's primary architecural characteristic is that it keeps the main search collection (`products`) and the signals boosting collection (`*_signals_boosts`) separate. This separation provides a number of benefits, including:

1. Allowing the signals for each query to be updated incrementally by only modifying the one document representing that query
2. Allowing boosting to be turned on or off easily by just not doing a lookup or modifying the user's query
3. Allowing different signals boosting algorithms to be swapped in at any time

Ultimately, by boosting specific documents for a given query at query time, the flexibility to change the boosts at any point in time based upon the current context is the major advantage of query-time signals boosting.

DRAWBACKS OF QUERY-TIME BOOSTING

While flexible, query time boosting also introduces some significant downsides with regard to query performance, scale, and relevance which may make it inappropriate for certain use cases. Specifically, query-time boosts:

1. Require an extra search to lookup boosts before the boosted search is executed, adding more processing (executing two searches) and latency (the final query has to wait on the results of the signals lookup query before being processed)
2. Doesn't handle long-lists of documents to boost for a query in a scalable way, requiring tradeoffs between user experience and relevance versus query speed and scale
3. Doesn't support search results pagination very well

The first downside is straight-forward, as each query essentially becomes two queries executed back-to-back, which increases the total search time. The second downside may not be as obvious, however, so it is worth exploring a bit further.

In query-time boosting, we look up a specific number of documents to boost higher in the search results for a query. For example, in our `ipad` search example from Figure 8.1 (see Listing 4.7 for the code), the boost for the query ultimately becomes:

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
"843404073153"^73  "885909457601"^62  "635753493559"^62  "885909472376"^61
"610839379408"^29 "884962753071"^28
```

This boost contains 10 documents, but only because that is the number of boosts we requested. Assuming we only showed ten documents on the first page, then the whole first page will look good… but what if the user navigates to page 2? In this case there won't be any boosted documents shown, because only the first 10 documents with signals for the query were boosted!

In order to boost documents for the second page, we would need to ensure we have at least enough document boosts to cover the full first two pages, which means increasing from 10 boosts to 20 boosts (modifying the "limit" parameter to 20 on the boost lookup query):

```
"885909457588"^966 "885909457595"^205 "885909471812"^202 "886111287055"^109
"843404073153"^73  "635753493559"^62  "885909457601"^62  "885909472376"^61
"610839379408"^29  "884962753071"^28  "635753490879"^27  "885909457632"^26
"885909393404"^26  "716829772249"^23  "821793013776"^21  "027242798236"^15
"600603132827"^14 "886111271283"^14 "722868830062"^13 "092636260712"^13
```

You can thus mostly solve this problem by increasing the number of boosts looked up every time someone navigates to the "next" page, but this will very quickly slow down subsequent queries, as page 3 will require looking up and applying 30 boosts, page 10 will require 100 boosts, and so on. For a use case where only a small number of boosted documents exists for each query this is not a big problem, but for many use cases, there may be hundreds or thousands of documents that would benefit from being boosted. In our query example of `ipad`, for example, there are more than 200 documents which contain aggregated signals, so most of those documents will never be boosted at all unless someone pages very deep into the search results, and at that point the queries are likely to be slow, and at some point could even time out.

Only including a subset of the boosts presents another problem, as well: search results aren't always strictly ordered by the boost value! We've made the assumption that requesting the top 10 boosts will be enough for the first page of 10 results, but in reality the boost is only one of the factors that affects relevance. It could be that documents further down in the boost list have a higher base relevance score and that if their boosts were also loaded that they would jump up to the first page of search results.

As a result, as a user navigates from page one to two and the number of boosts loaded increases, some of the results might jump up to page one and never be seen or jump down to page two and be see again as a duplicate. When someone then moves on to page three, the results from all three pages could further get shuffled around.

Even if these results are much more relevant than search results without signals boosting applied, it doesn't make a very optimal user experience. Index-time signals boosting can help overcome these drawbacks, as we'll show in the next section.

8.6.2 Implementing Index-time signals boosting

Index-time signals boosting turns the signals boosting problem on its head - instead of boosting popular documents for queries at query time, we boost popular queries for documents at indexing time. This is accomplished by adding popular queries to a field in each document, along with their boost value. Then, at query time, we simply search on the new field, and if the field contains the term from our query then it it gets automatically boosted based upon the boost value indexed for the term.

When implementing index-time boosting, we leverage the exact same signals aggregations to generate pairs of documents and boost weights for each query. Once those signals boosts have been generated, we just have to add one additional step to our workflow: updating the products collection to add a field onto each document containing each term for which the document should be boosted, along with the associated numeric boost weight. Listing 8.8 demonstrates this additional step in our workflow.

Listing 8.8 Mapping signals boosts from a separate query-time collection to a field in the main collection

```
signals_boosts_collection="normalized_signals_boosts"
signals_boosts_opts={"zkhost": "aips-zk", "collection":
➥signals_boosts_collection}
df = spark.read.format("solr").options(**signals_boosts_opts).load()
df.registerTempTable(signals_boosts_collection)   ❶


products_collection="products_with_signals_boosts"
products_read_boosts_opts={"zkhost": "aips-zk", "collection":
➥products_collection}
df = spark.read.format("solr").options(**products_read_boosts_opts).load()
df.registerTempTable(products_collection)   ❷

boosts_query = """   ❸
SELECT p.*, b.signals_boosts from (
  SELECT doc, concat_ws(',',collect_list(concat(query, '|', boost))) as
  ➥signals_boosts FROM """ + signals_boosts_collection + """ GROUP BY doc
) b inner join """ + products_collection + """ p on p.upc = b.doc
"""

products_write_boosts_opts={"zkhost": "aips-zk", "collection":
➥products_collection, "gen_uniq_key": "true", "commit_within": "5000"}   ❹
spark.sql(boosts_query).write.format("solr").options(
➥**products_write_boosts_opts).mode("overwrite").save()
```

❶ Load a previously-generated signals boosting model

❷ Register the product table so we can load from it and save back to it with boosts added

❸ Insert all keywords with signals boosts for each document into a new "signals_boosts" field on the document

❹ Save the products back to the products collection, with the updated "signals_boosts" added

The code in Listing 8.9 reads all previously-generated signals boosts for each document and then maps the queries and boosts into a new `signals_boosts` field on each product document as a comma-separated list to terms (user queries) with a corresponding signals boosting weight for each term.

This `signals_boosts` field is a specialized field in Solr containing a DelimitedPayloadBoostFilter, which allows for terms (queries) to be indexed with associated boosts that can be used to influence query-time scoring. For example, for the most popular iPad, the product document will now be modified to look as follows:

```
{...
   "id": "885909457588",
   "name": "Apple® - iPad® 2 with Wi-Fi - 16GB - Black"
   "signals_boosts": "ipad|2939,ipad 2|1104,ipad2|540,i pad|341,apple ipad|
   ➥152,ipads|123,apple|118,i pad 2|99,tablets|67,tablet|61..."
 ...
 }
```

At query time, this `signals_boosts` field will be searched upon, and if the query matches one or more of the values in the field, the score for that document will be boosted relative to boost value.

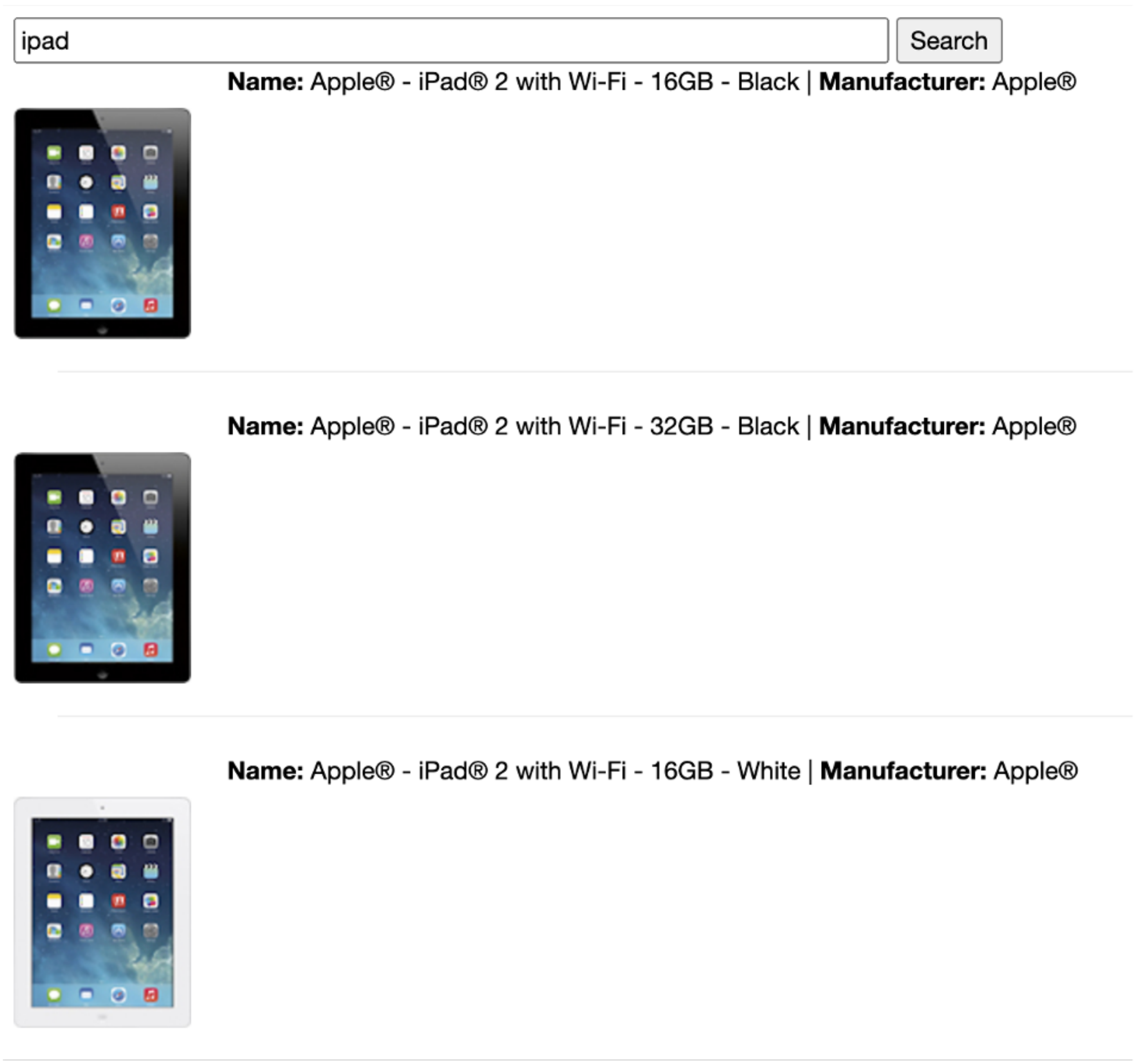
Listing 8.9 demonstrates how to perform a query utilizing index-time signals boosts, harnessing the `payload` function in the search engine to boost based upon the indexed payload (the boost value) associated with the user's query.

Listing 8.9 Performing a query that ranks based upon index-time signals boosts

```
query = "ipad"

def get_query(query, signals_boosts_field):
    request = {
        "query": query,
        "fields": ["upc", "name", "manufacturer", "score"],
        "limit": 3,
        "params": {
          "qf": "name manufacturer longDescription",
          "defType": "edismax",
          "indent": "true",
          "sort": "score desc, upc asc",
          "qf": "name manufacturer longDescription",
          "boost": "payload(" + signals_boosts_field + ", \""    ❶
                            + query + "\", 1, first)"   ❶
        }
    }

    return request

collection = "products_with_signals_boosts"
boosted_query = get_query(query, signals_boosts_field)
print("Main Query:")
print(boosted_query)

search_results = requests.post(solr_url + collection + "/select",
➥json=boosted_query).json()["response"]["docs"]
print("\nSearch Results (Basic Signals Boosting): ")
print(search_results)
display(HTML(render_search_results(query, search_results)))
```

❶ Boosting the relevance score based upon the indexed signals boosts for the query

Figure 8.5 shows the results of the index-time signals boosting. As you can see, the results now look similar to the query-time signals boosting output shown previously in Figure 4.1.



Name: Apple® - iPad® 2 with Wi-Fi - 16GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 32GB - Black | **Manufacturer:** Apple®

Name: Apple® - iPad® 2 with Wi-Fi - 16GB - White | **Manufacturer:** Apple®

Figure 8.5 Index time signals boosting, demonstrating similar results as query-time index boosting.

The relevance scores will likely not be identical when using the `payload` function to boost index-time boosts versus adding document boosts for query-time signals boosting, because the relevance scoring math is a little different. The relative ordering of results should be very similar, though. The index-time signals boosting will apply to all documents with a signals boost as opposed to only the top documents with a signals boost, making index-time boosting more comprehensive, among other benefits.

BENEFITS OF INDEX-TIME BOOSTING

Index-time boosting solves for most of the drawbacks of query-time boosting:

1. The query workflow is simpler and faster because it doesn't require doing two queries - one to look up the signals boosts and another to run a boosted query using those signals boosts.
2. Each query is more efficient and faster per boost as the number of boosts increases, because the boost query is a single keyword search against the boost field as opposed to a boost query for an increasing number of documents which need to be boosted.
3. Results paging is no longer a problem, because ALL documents matching the query are boosted, not just the top-N that can be efficiently loaded and added to the query.

Given these characteristics, index-time boosting can substantially improve relevance and consistency of results ordering by ensuring all queries receive consistent and complete boosting of all their matching documents, and it can substantially improve query speed by making queries more efficient and removing extra lookups prior to execution of the main query to the search engine.

DRAWBACKS OF INDEX-TIME BOOSTING

If index-time boost solves all of the problems of query-time boosting, they why wouldn't we always use index-time signals boosting over query-time signals boosting?

The main drawback of index-time boosting is that since the boost values for a query are indexed onto each document (each document contains the terms for which that document should be boosted), this means that adding or removing a keyword from the signals boosting model requires reindexing all documents associated with that keyword. If signals boosting aggregations are updated incrementally (on a per-keyword basis), then this means potentially reindexing all of the documents within your search engine on a continuous basis. If your signals boosting model is updated in batch for your entire index, then at a minimum this means reindexing potentially all of your documents every time your signals boosting model is regenerated.

This kind of indexing pressure adds operational complexity to your search engine. In order to keep query performance fast and consistent, you will likely want to separate indexing of documents onto separate servers from where the search indexes are hosted for serving queries.

SIDEBAR **Separation of Concerns: Indexing vs. Querying**

In Apache Solr, indexes are broken into one or more "shards", which are partitions containing a subset of the documents in a collection. Each shard can have one or more replicas, which are each exact copies of all the data belonging to their shard. When a search is run, Solr sends the query to one replica of each shard, the query is run in parallel on each of those replicas, and the results are aggregated and returned as a full set of results to the end users. The primary purpose of adding more shards is to allow for more documents to be searched in less time, and the primary purposes of adding more replicas are to add fault tolerance and to enable a larger number of searches to be run against the same number of shards.

Solr has three different types of replicas: NRT (Near-realtime), TLOG (transaction log), and PULL replicas. By default, all replicas are NRT, which means every replica indexes every document update when it comes in. This allows document updates to be available immediately on each replica, but it can also very negatively impact query time on those replicas if lots of documents are being indexed constantly. The other replica types (TLOG and PULL) are able to pull indexes from an NRT replica instead of doing the indexing work in duplicate, which can allow a separation of concerns within the cluster to allow indexing on the NRT replica that is isolated from the querying operations on the TLOG and PULL replicas.

If you plan to do index-time signals boosting and expect to be constantly reindexing signals, you should strongly consider isolating index and query time operations to ensure your query performance isn’t negatively impacted by the significant additional indexing overhead from ongoing indexing of signals boosts.

The other drawback of index-time boosting, which is also related to the requirement that all documents affected by a signal be reindexed upon changes, is that making changes to your signals boosting function can require more planning. For example, if you would like to change your weight for click vs. purchase signals from 1:25 to 1:20, then you may want to create a `signals_boosts_2` field with the new weights, reindex all of your documents adding the new boosts, and then flip over your query to use the new field instead of the original `signals_boosts` field. Otherwise, your boost values and ranking scores will fluctuate inconsistently until all of your documents scores have been updated.

If those drawbacks can be worked around, however, then implementing index-time signals boosting can solve all of the drawback of query-time signals boosting, leading to better query performance, full support for results paging, and use of all signals from all documents as opposed to just a sampling from the most popular documents.

8.7 Summary

- Signals boosting is a type of ranking algorithm which aggregates user signal counts per query and uses those counts as relevance boosts for that query in the future. This ensures the most popular items for each query are pushed to the top of the search results.
- Normalize queries by treating different variations (case, spelling, etc.) as the same query helps clean up noise in user signals and builds a more robust signals boosting model.
- Crowdsourced data is subject to manipulation, so it is important to explicitly prevent spam and malicious signals from impacting the quality of your relevance models.
- You can combine different signal types into a single signals boosting model by assigning relative weights to each signal type and doing a weighted sum of values across signal types. This enables you to give more relevance to stronger signals (positive or negative) and reduce noise from weaker signals.
- Introducing a time-decay function enables recent signals to carry more weight than older signals, allowing older signals to phase out over time.
- Signal boosting models can be productionized using query-time signals boosting (more flexible) or index-time signals boosting (more scalable and more consistent relevance ranking).

10

Learning to rank for generalizable search relevance

This chapter covers

- Using machine learning to build generalizable search systems
- Ranking within the search engine using machine learning models
- How learning to rank is different from other machine learning methods
- Building a robust and generalizable ranking model

It's a random Tuesday. You review your search logs. The searches range from the frustrated runner's - `polar m430 running watch charger` - to the worried hypochondriac's - `weird bump on nose - cancer?` - to the curious cinephile's - `william shatner first film`. Despite the fact that many are one-off queries, you know each user expects nothing less than amazing search results.

You feel hopeless. You know many query strings, by themselves, are distressingly rare. You have very little click data to know what's relevant for these searches. Every day gets more challenging: trends, use cases, products, user interfaces, and even languages evolve. How can anyone hope to build search that amazes when users seem to constantly surprise us with new ways of searching?

Despair not, there is hope! In this chapter, we introduce *generalizable search systems*. Generalizable search learns the underlying patterns that drive relevance. Instead of memorizing that the article entitled "Zits: bumps on nose" is the answer for the query `weird bump on nose - cancer?`, we observe the underlying pattern - that a strong title match corresponds to high probability of relevance. If we can learn these patterns, encoding them into a model, then we can give relevant results *even for search queries we've never seen*.

This chapter explores *Learning to Rank* (LTR): a technique using machine learning to create generalized relevance ranking models. We'll prepare, train, and search with LTR models using the search engine.

10.1 What is Learning to Rank?

Let's explore what LTR does exactly. We'll see how LTR creates generalized relevance by finding patterns that predict relevance. We'll then explore more of the nuts and bolts of building a model.

Recall manual relevance tuning from Chapter 3. We observe factors that correspond with relevant results and we combine those factors mathematically into a *ranking function*. The ranking function returns a relevance score that orders results as closely as possible to our ideal ranking.

For example, consider a movie search engine, with documents like the one in Listing 10.1:

Listing 10.1 A document for the movie The Social Network showing potentially useful fields for use in a ranking function.

```
{'title': ['The Social Network'],
 'overview': ['On a fall night in 2003, Harvard undergrad and computer
   ➥programming genius Mark Zuckerberg sits down at his computer and
   ➥heatedly begins working on a new idea. In a fury of blogging and
   ➥programming, what begins in his dorm room as a small site among
   ➥friends soon becomes a global social network and a revolution in
   ➥communication. A mere six years and 500 million friends later,
   ➥Mark Zuckerberg is the youngest billionaire in history... but for
   ➥this entrepreneur, success leads to both personal and legal
   ➥complications.'],
 'tagline': ["You don't get to 500 million friends without making a few
   ➥enemies."],
 'release_year': 2010}
```

This document comes from TheMovieDB (tmdb) corpus (http://themoviedb.org), which we'll use in this chapter. If you wish to follow along with the code for this chapter please use this chapter's first notebook to listing x.y.

Through endless iterations and tweaks, we might arrive at a generalizable movie ranking function that looks something like Listing 10.2.

Listing 10.2 A manual ranking function combining title, overview, and release year weights as searched keywords

```
q=title:(${keywords})^10 overview:(${keywords})^20 {!func}release_year^0.01
```

Manually optimizing generalized ranking to work over many queries takes effort. Such an optimization is perfect for machine learning.

This is where LTR comes in. *Learning to Rank* (LTR) takes our proposed relevance factors, and learns an optimal ranking function. Learning to Rank takes several forms: from a simple set of linear weights (like the boosts here) to a complex deep learning model.

To learn the ropes, we’ll build a simple LTR model in this chapter. We’ll find the optimal weights for `title`, `overview`, and `release_year` in a scoring function like the one above. With this relatively simple task, however, we’ll see the full lifecycle of developing a Learning to Rank solution.

10.1.1 Implementing learning to rank in the real-world

As we continue to define LTR at a high level, let’s quickly clarify where LTR fits into the overall picture of a search system. Then we can look at the kinds of data we’ll need to build an LTR model.

We focus on building LTR for production search systems, which can be quite different from a research context. We not only need relevant results, but results returned suitably fast, with mainstream, well-understood search techniques.

For this reason, our examples focus on building Learning to Rank with Solr’s Learning to Rank plugin. This plugin reranks using models within the search engine, improving performance and scaling learning to rank to true 'big data' problems. These lessons go beyond Solr, though - Elasticsearch also has a community provided plugin (https://github.com/o19s/elasticsearch-learning-to-rank). The Elasticsearch plugin has nearly identical concepts, and other search systems will tend to follow the same general steps outlined in this chapter.

Figure 10.1 outlines the workflow for developing a practical Learning to Rank solution.

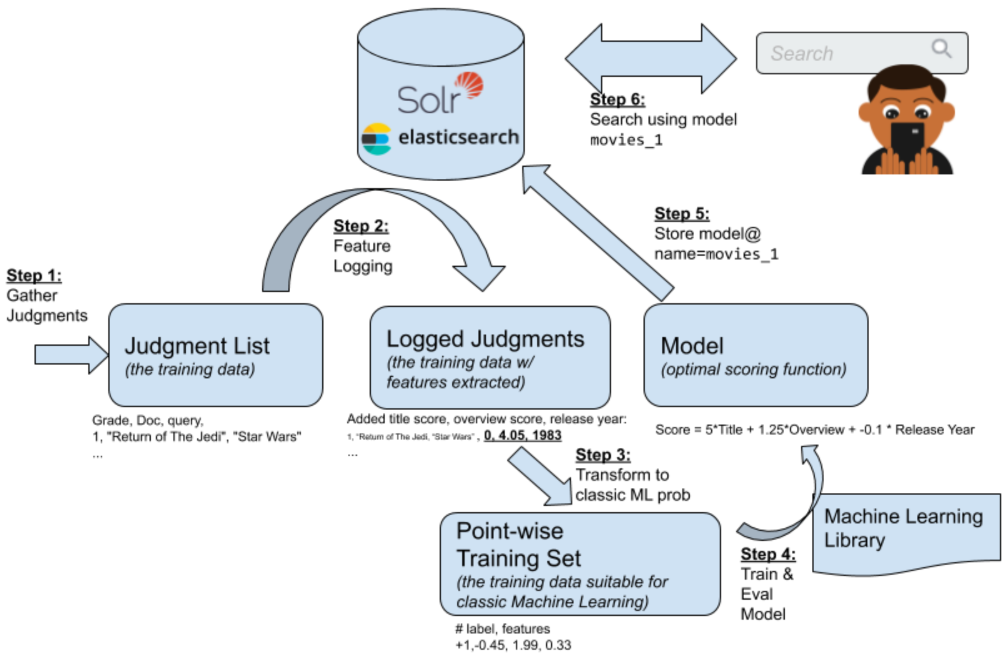


Figure 10.1 LTR systems transform our training data (judgment lists) into models that generalize relevance ranking. This type of system lets us find the underlying patterns in our training data.

In Figure 10.2, you may notice similarities between LTR and traditional machine-learning-based classification or regression systems. But the exceptions are what make it interesting. Table 10.1 maps definitions between traditional machine learning objectives and learning to rank.

Table 10.1 Classic Machine Learning vs Learning to Rank.

Concept	Traditional Machine Learning	Learning to Rank
Training Data	Set of historical or "true" examples the model should try to predict. IE stock prices on past days, like "Apple" was $125 on June 6th, 2021	A judgment list: A judgment simply labels a document as relevant or irrelevant for a query. In Figure 10.2, "Return of the Jedi" is labeled relevant (grade of `1`), for the query `star wars`
Feature	The data we can use to predict the training data. IE Apple had 147,000 employees and revenue of $90 billion	Data used so that relevant results rank higher than irrelevant ones, ideally values the search engine can compute quickly. Our features are search queries like the `title:(${keywords})` from Listing 10.2
Model	The algorithm that takes features as input to make a prediction. Given Apple has 157,000 employees on July 6th, 2021 with $95 billion in revenue, the model might predict a stock price of $135 for that date	Combines the ranking features (search queries) together to assign a relevance *score* to each potential search result. Results are sorted by score descending, hopefully placing more relevant results first

This chapter follows the steps in Figure 10.2 to train an LTR model, as outlined in greater depth below:

1. *Gather judgments*: we derive judgments from clicks or other sources. We’ll cover this step in depth in Chapter 11.
2. *Feature logging*: to train a model, we must combine the judgments with features to see the overall pattern. This step requires us to ask the search engine to store and compute queries representing the features.
3. *Transform to traditional machine learning problem*: you’ll see that most LTR really is about translating the ranking task into something that looks more like the "Traditional Machine Learning" column in Table 10.1
4. *Train and evaluate model*: here we construct our model and confirm it is, indeed, generalizable, and thus will perform well for queries it hasn’t even seen yet.
5. *Store the model*: we upload the model to our search infrastructure, tell the search engine which features to use as input, and enable it for users to use in their searches
6. *Search using the model*: we finally can execute searches using the model!

The rest of the chapter will walk through each of these steps in detail to build our first LTR implementation. Let’s get cracking!

10.2 Step 1: A judgment list, starting with the training data

You saw what LTR is at a high level, let's begin to get into the nitty gritty. Before implementing LTR we must first learn about the data used to train an LTR model: the judgment list.

A *judgment list* is a list of relevance labels or *grades*, which each indicate relevance of a document to a query. Grades can come in a variety of forms. For now we'll stick to simple *binary judgments* - a 0 to indicate an irrelevant document, and a 1 for a relevant one.

Using a `Judgment` class provided with this book's code, we label "The Social Network" as relevant for the query `social network` by creating a `Judgment` below:

```
from ltr.judgments import Judgment
Judgment(grade=1, keywords='social network', doc_id=37799)
```

It's more interesting to look over multiple queries. In Listing 10.3, we have `social network` and `star wars` as two different queries, with movies graded as relevant or irrelevant.

Listing 10.3 Labeling movies relevant (grade=1) or irrelevant (grade=0) for queries `social network` and `star wars`

```
mini_judg_list=[
    # for 'social network' query
    Judgment(grade=1, keywords='social network', doc_id='37799'),
    ➥# The Social Network
    Judgment(grade=0, keywords='social network', doc_id='267752'),
    ➥# #chicagoGirl
    Judgment(grade=0, keywords='social network', doc_id='38408'),
    ➥# Life As We Know It
    Judgment(grade=0, keywords='social network', doc_id='28303'),
    ➥# The Cheyenne Social Club

    # for 'star wars' query
    Judgment(grade=1, keywords='star wars', doc_id='11'),
    ➥# Star Wars, A New Hope
    Judgment(grade=1, keywords='star wars', doc_id='1892'),
    ➥# Return of the Jedi
    Judgment(grade=0, keywords='star wars', doc_id='54138'),
    ➥# Star Trek Into Darkness
    Judgment(grade=0, keywords='star wars', doc_id='85783'),
    ➥# The Star
    Judgment(grade=0, keywords='star wars', doc_id='325553'),
    ➥# Battlestar Galactica
]
```

You can see that in Listing 10.3, we labeled "Star Trek" and "Battlestar Galactica" as irrelevant for `star wars`, but "Return Of The Jedi" as relevant.

You're hopefully asking yourself - where did these grades come from!? Hand labeled by movie experts? Based on user clicks? Good questions! Creating a good training set, based on user interactions with search results, is crucial for getting LTR to work well. To get training data in bulk, we usually derive these labels from click traffic using an algorithm known as a *click model*. As this step is so foundational, we dedicated all of Chapter 11 to diving deeper into the topic.

Each judgment also has a `features` vector, which can be used to train a model. The first feature in the `features` vector could be made to correspond to the `title` BM25 score, the second to the `overview` BM25 score, and so on. We haven’t populated the `features` vectors yet, however, so they’ll currently be empty if you try to inspect them:

```
In: mini_judg_list[0].features
Out: []
```

Let’s use the search engine to gather some features!

10.3 Step 2 - feature logging and engineering

With our training data in place, we now need to learn what predicts relevance. For example, by noting a pattern like "relevant results in our judgments correspond to strong title matches". How, exactly, you define "title match" is what feature engineering is all about. In this section you’ll see what, exactly, a feature is and how to use a modern search engine to engineer and extract these features from a corpus.

For the purposes of LTR, a *feature* is some numerical attribute of the document, the query, or the query-document relationship. Features are the mathematical building blocks we use to build a ranking function. You’ve already seen a manual ranking function with features from Listing 10.2 : the keyword’s score in the `title` field is one such feature. As are the `release_year` and `overview` keyword scores.

```
q=title:(${keywords})^10 overview:(${keywords})^20 {!func}release_year^0.01
```

Of course, the features you end up using could be more complex or domain-specific, such as commute distance in job search, or some knowledge graph relationship between query and document. Anything you can compute relatively quickly when a user searches might be a reasonable feature.

Feature logging takes a judgment list and computes features for each labeled query-document pair. If we computed the values of each component of Listing 10.2 for the query `social network` we would arrive at something like Table 10.2.

Table 10.2 Features logged for the keywords `social network` for relevant (grade=1) / irrelevant (grade=0) documents

Grade	Movie	`title:(${keywords})`	`overview:(${keywords})`	`{!func}release_year`
1	Social Network	8.243603	3.8143613	2010.0
0	#chicagoGirl	0.0	6.0172443	2013.0
0	Life As We Know It	0.0	4.353118	2010.0
0	The Cheyene Social Club	3.4286604	3.1086721	1970.0

A machine learning algorithm might examine feature values from Table 10.2 and converge on a good ranking function. From just the data in Table 10.2, it seems such an algorithm might produce a ranking function with a higher weight for the `title` feature and lower weights for the other features.

Before we get to the algorithms, however, we need to examine the feature logging workflow in a production search system.

10.3.1 Storing features in a modern search engine

Modern search engines that support LTR, like Solr and Elasticsearch, help us store, manage, and extract features. Let's examine how Solr, in particular, performs this task.

Solr LTR tracks the features logged in a *feature store* - a list of named features. It's crucial that we log features for training in a manner consistent with how the search engine will execute the model. Much of the LTR plugin's job is to help store and manage features and keep things consistent.

As shown in Listing 10.4, creating a feature store in Solr is a simple HTTP PUT. Here we create three features: a title field relevance score `title_bm25`, an overview field relevance score, `overview_bm25`, and the value of the `release_year` field. BM25 here corresponds to the BM25 based scoring defined in Chapter 3, Solr's default method of scoring term matches in text fields using term frequency and other index statistics.

Listing 10.4 Creating three features for Learning to Rank.

```
feature_set = [
    {
      "name" : "title_bm25",   ❶
      "store": "movies",   ❷
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "title:(${keywords})"   ❸
      }
    },
    {
      "name" : "overview_bm25",   ❹
      "store": "movies",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "overview:(${keywords})"
      }
    },
    {
      "name" : "release_year",   ❺
      "store": "movies",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!func}release_year"
}}]

requests.put('http://aips-solr:8983/solr/tmdb/schema/feature-store',
             json=feature_set)
```

❶ The name of the feature

❷ The feature store where the feature will be saved

❸ A parametrized feature, taking the keywords (i.e. `star wars`), and searching the title field

❹ A second feature, searching a different field

❺ A document-only feature, the `release_year` of the movie

The first two features are parameterized: they each take the search keywords (i.e. `social network`, `star wars`) and execute a search on the corresponding field. The final feature simply retrieves the release year of the movie (purely a document feature). Note the syntax used in `params` is simply a Solr query, letting you leverage the full power of Solr's extensive Query DSL to craft features.

10.3.2 Logging features from our Solr corpus

With features loaded into Solr, our next focus will be to log features for every row in our judgment list. After we get this last bit of plumbing out of the way, we can then train a model that can observe relationships between relevant and irrelevant document for each query.

For each unique query in our judgment list, we need to extract the features for the query's graded documents. For `social network` in the mini judgment list above, we have one relevant document (`37799`) and three irrelevant documents (`267752`, `38408`, and `28303`).

An example of feature logging for the query `social network` is shown in Listing 10.5.

Listing 10.5 Logging the values from our feature store for the docs present in our judgment list for the query `social network`.

```
logging_solr_query = {
 "fl": "id,title,[features store=movies efi.keywords=\"social network\"]",   ❶
 'q': "id:37799 OR id:267752 OR id:38408 OR id:28303",   ❷
 'rows': 10,
 'wt': 'json'
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select',
                     data=logging_solr_query)
resp.json()
```

❶ The feature store to execute and parameters to use when executing each feature

❷ Relevant and irrelevant documents for the `social network` query

Results

```
'docs': [{'id': '38408',
  'title': 'Life As We Know It',
  '[features]': 'title_bm25=0.0,overview_bm25=4.353118,
  ➥release_year=2010.0'},   ❶
 {'id': '37799',
  'title': 'The Social Network',
  '[features]': 'title_bm25=8.243603,overview_bm25=3.8143613,
  ➥release_year=2010.0'},
 {'id': '28303',
  'title': 'The Cheyenne Social Club',
  '[features]': 'title_bm25=3.4286604,overview_bm25=3.1086721,
  ➥release_year=1970.0'},
 {'id': '267752',
  'title': '#chicagoGirl',
  '[features]': 'title_bm25=0.0,overview_bm25=6.0172443,
  ➥release_year=2013.0'}]}}
```

❶ Each feature value logged for this movie for `social network`

The import syntax in Listing 10.5 is the square brackets passed to `fl` (field list). This syntax tells Solr to add a computed field to each document response, containing the feature data, for the specified feature store (here `movies`). The `efi` parameter stands for *external feature information* and is used to pass query keywords (here `social network`) and any additional query-time information needed to compute each feature. Notice in the response, we retrieve each of the 4 documents we requested, along with every feature computed for that document.

With some mundane Python book-keeping (listing x.y), We can fill-in the features for the query `social network` in our training set from this response. We end up with Listing 10.6, a partially filled out training set. Here, features are filled in for the query `social network`; features are still needed for the query `star wars`

Listing 10.6 The judgment list, with features appended for just the query `social network`

```
[Judgment(grade=1,qid=1,keywords=social network,doc_id=37799,features=[
➥8.243603, 3.8143613, 2010.0],weight=1),   ❶
 Judgment(grade=0,qid=1,keywords=social network,doc_id=267752,features=[
 ➥0.0, 6.0172443, 2013.0],weight=1),
 Judgment(grade=0,qid=1,keywords=social network,doc_id=38408,features=[
 ➥0.0, 4.353118, 2010.0],weight=1),   ❷
 Judgment(grade=0,qid=1,keywords=social network,doc_id=28303,features=[
 ➥3.4286604, 3.1086721, 1970.0], weight=1),
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=11,features=[],
 ➥weight=1),   ❸
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=1892,features=[],
 ➥weight=1),   ❸
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=54138,features=[],
 ➥weight=1),   ❸
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=85783,features=[],
 ➥weight=1),   ❸
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=325553,features=[],
 ➥weight=1)]   ❸
```

❶ The judgment for movie "The Social Network" relative to query `social network`, including the logged feature values

❷ An irrelevant document for the query `social network` (note the low `title_bm25` score)

❸ We have yet to log features for the `star wars` judgments

In Listing 10.6, as we might expect, the first feature value corresponds to the first feature in our feature store (`title_bm25`); the second value to the second feature in our feature store (`overview_bm25`), and so on. We then repeat Listing 10.6 for the query `star wars`, log the corresponding graded documents, resulting in a fully logged training set as shown in Listing 10.7.

Listing 10.7 The judgment list with features appended for both the queries `star wars` and `social network`.

```
[Judgment(grade=1,qid=1,keywords=social network,doc_id=37799,
➥features=[8.243603, 3.8143613, 2010.0],weight=1),
 Judgment(grade=0,qid=1,keywords=social network,doc_id=267752,
 ➥features=[0.0, 6.0172443, 2013.0],weight=1),
 Judgment(grade=0,qid=1,keywords=social network,doc_id=38408,
 ➥features=[0.0, 4.353118, 2010.0],weight=1),
 Judgment(grade=0,qid=1,keywords=social network,doc_id=28303,
 ➥features=[3.4286604, 3.1086721, 1970.0],weight=1),
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=11,
 ➥features=[6.7963624, 0.0, 1977.0],weight=1),   ❶
 Judgment(grade=1,qid=2,keywords=star wars,doc_id=1892,
 ➥features=[0.0, 1.9681965, 1983.0],weight=1),
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=54138,
 ➥features=[2.444128, 0.0, 2013.0],weight=1),
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=85783,
 ➥features=[3.1871135, 0.0, 1952.0],weight=1),
 Judgment(grade=0,qid=2,keywords=star wars,doc_id=325553,
 ➥features=[0.0, 0.0, 2003.0],weight=1)]
```

❶ We have now logged features for query `star wars`

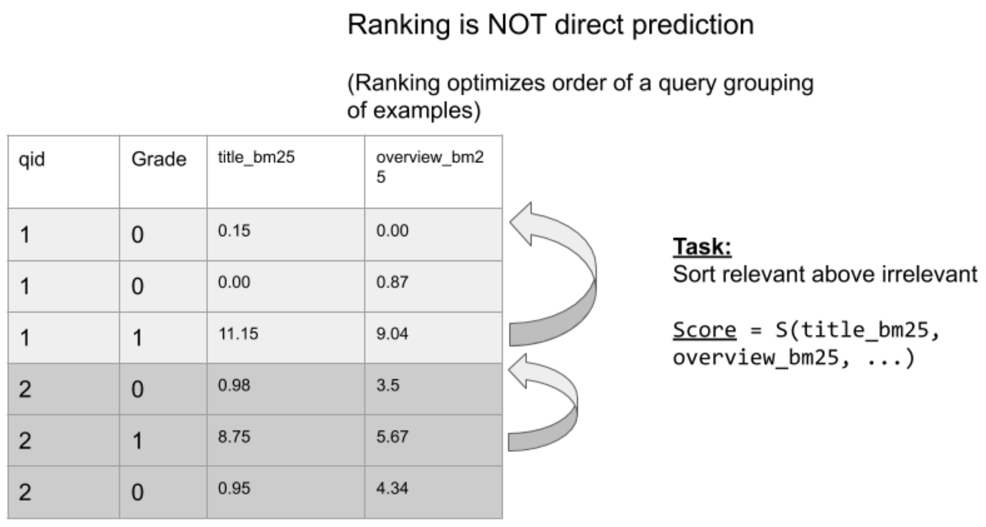
Just two queries with a few judgments per query isn't that interesting, however. In this chapter, we'll use a judgment list with about a hundred movie queries, each with about 40 movies graded as relevant / irrelevant. Code for loading and logging features for this larger training set essentially repeats the Solr request shown in Listing 10.7. We won't repeat the lengthy code here (you can find it in the listing x.y). The end result of this feature logging looks just like Listing 10.7, just created from a much larger judgment list.

We'll next move on to consider how to handle the problem of ranking as a machine learning problem.

10.4 Step 3 - transforming LTR to a traditional machine learning problem

In this section we're going to explore ranking as a machine learning problem. This will help us understand how to apply existing, classic machine learning knowledge to our Learning to Rank task.

The task of Learning to Rank is to look over many relevant / irrelevant training examples for a query, and to then build a model to bring relevant documents to the top, and push irrelevant ones to the bottom. Each training example doesn't have much value by itself, but only in relation to how its ordered alongside its peers in a query. Figure 10.2 shows this task, with two queries. The goal is to find a scoring function that can use the features to correctly order results.



Ranking is NOT direct prediction

(Ranking optimizes order of a query grouping of examples)

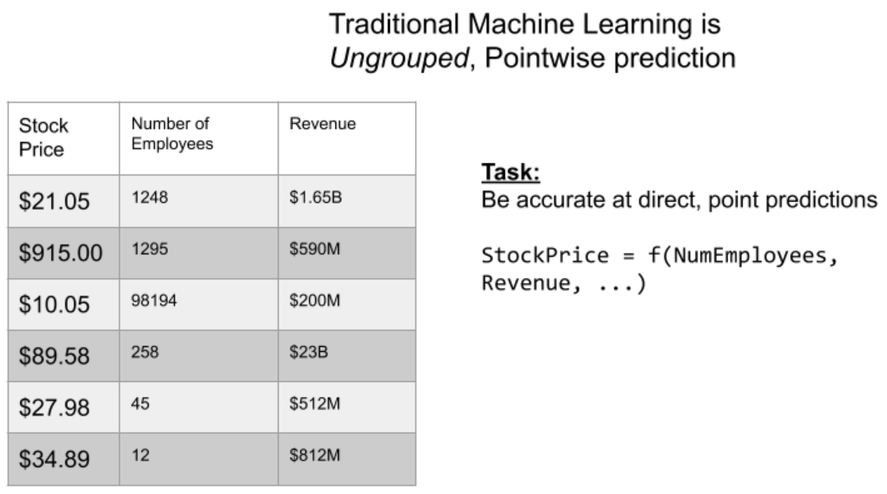
qid	Grade	title_bm25	overview_bm25
1	0	0.15	0.00
1	0	0.00	0.87
1	1	11.15	9.04
2	0	0.98	3.5
2	1	8.75	5.67
2	0	0.95	4.34

Task:
Sort relevant above irrelevant

```
Score = S(title_bm25,
overview_bm25, ...)
```

Figure 10.2 Learning to Rank is about placing each query's result set in the ideal order, not predicting individual relevance grades. That means we need to look at each query as a case unto itself.

Contrast LTR with a more classic *point-wise machine learning* task: a task like predicting a company's stock price as mentioned in Table 10.2 earlier. Here, we can evaluate the model's accuracy on each training example in isolation. We know, just by looking at one company, how well we predicted that company's stock price. Compare Figure 10.3 showing a point-wise task to Figure 10.2. Notice in Figure 10.3 that the learned function attempts to predict the stock price directly, whereas with LTR, the function's output is only meaningful for ordering items relative to their peers for a query.



Traditional Machine Learning is
Ungrouped, Pointwise prediction

Stock Price	Number of Employees	Revenue
$21.05	1248	$1.65B
$915.00	1295	$590M
$10.05	98194	$200M
$89.58	258	$23B
$27.98	45	$512M
$34.89	12	$812M

Task:
Be accurate at direct, point predictions

```
StockPrice = f(NumEmployees,
Revenue, ...)
```

Figure 10.3 Point-wise Machine Learning tries to optimize predictions of individual points (such as a stock price or the temperature). Search relevance is a different problem than point-wise prediction. Instead we need to optimize a ranking of examples grouped by search query.

LTR appears to be a very different animal than point-wise machine learning. But most LTR methods use clever alchemy to transmogrify ranking into a point-wise machine learning task like stock price prediction.

We'll take a look at one model's method for transforming the ranking task in the next section by exploring a popular LTR model named SVMRank.

10.4.1 SVMRank: Transforming ranking to binary classification

At the core of LTR is the model: the actual algorithm that learns the relationship between relevance / irrelevance and the features like `title_bm25`, `overview_bm25`, etc. In this section, we'll explore one such model, SVMRank, first understanding what "SVM" even stands for, then digging in to how it can be used to build a great, generalizable LTR model.

SVMRank transforms relevance into a binary classification problem. *Binary Classification* simply means classifying items as one of two classes (like 'relevant' vs 'irrelevant'; 'adult' vs 'child'; 'cat' vs 'dog') using the available features.

An *SVM* or *Support Vector Machine* is one method of performing binary classification. While we won't go in-depth into SVMs, you need not be a machine learning expert to follow the discussion. Nevertheless, you might want to get a deeper overview of SVM's from a book like *Grokking Machine Learning* by Luis Serrano (https://www.manning.com/books/grokking-machine-learning).

Intuitively an SVM finds the best, most generalizable line (or really hyperplane) to draw between the two classes. If trying to build a model to predict whether an animal is a dog or cat, we might look the heights and weights of known dogs or cats and draw a line separating the two classes as shown in Figure 10.4.

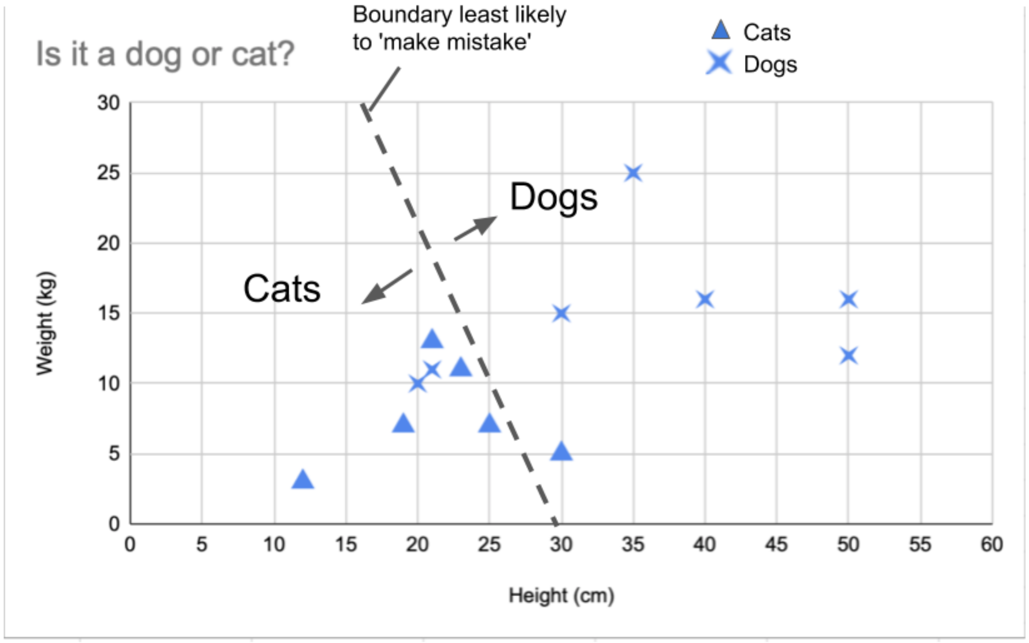


Figure 10.4 SVM Example: Is an animal a dog or a cat? This hyperplane (the line) separates these two cases based on the two features height and weight. Soon you'll see how we might do something similar to separate relevant and irrelevant search results for a query.

A good line drawn between the classes, or *separating hyperplane*, attempts to minimize the mistakes it makes in the training data (fewer dogs on the cat side and vice versa). We also want a hyperplane that is *generalizable*, meaning that it will probably do a good job of classifying animals that weren't seen during training. After all, what good is a model if it can't make predictions about new data? It's not very AI-powered!

Another detail to know about SVMs is they can be sensitive to the range of our features. For example, imagine if our height feature was millimeters instead of centimeters like in Figure 10.5. It forces the data to stretch out on the x-axis, and the separating hyperplane looks quite different!

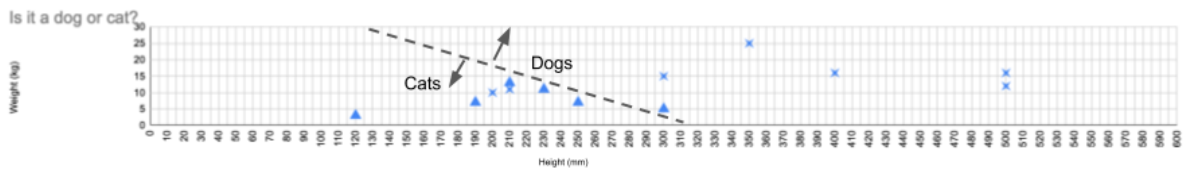


Figure 10.5 Separating hyperplane impacted by range of one of the features. This causes SVMs to be sensitive to the range of features, and thus we need to normalize the features so one feature doesn't create undue influence on the model.

SVMs work best when our data is *normalized*. Normalization just means forcing features to a comparable range. We'll normalize our data by mapping 0 to the mean of the feature values. So if the average `release_year` is 1990, movies released in 1990 will by normalized to 0. We'll also map +1/-1 to one standard deviation above or below the mean. So if the standard deviation of movie release years is `22` years, then movies in 2012 turn into a `1.0`; movies in 1968 turn into a `-1.0`. We can repeat this for `title_bm25` and `overview_bm25` using those feature's means and standard deviations in our training data. This helps makes features a bit more comparable when finding a separating hyperplane.

With that brief background out of the way, could an SVM help us somehow separate relevant from irrelevant documents? Even for queries it hasn't seen before? Yes! That's exactly what SVMRank hopes to do. Let's walk through how SVMRank works.

10.4.2 Transforming our LTR training data to binary classification

With LTR, we must reframe the task from ranking to a traditional machine learning task. In this section, we'll explore how SVMRank transforms ranking into a binary classification task suitable for an SVM.

Before we get started, let's inspect the fully logged training set from the end of step 2 for our two favorite queries, `star wars` and `social network`. In this section, we'll focus on just two features (`title_bm25` and `overview_bm25`) to help us explore feature relationships graphically. Figure 10.6 shows these two features for every graded `star wars` and `social network` document, labeling some prominent movies from the training set.

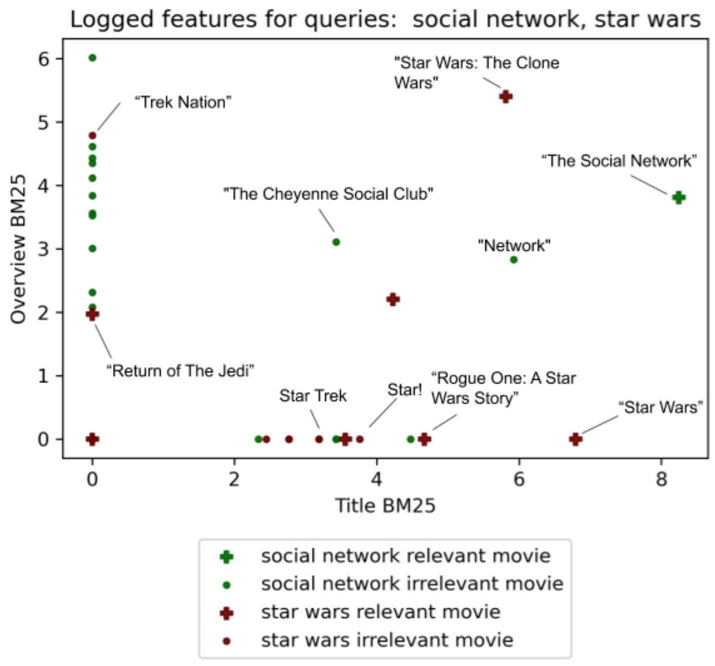


Figure 10.6 Logged feature scores for `social network` and `star wars` queries

FIRST, NORMALIZE THE LTR FEATURES

We're first going normalize each feature. Listing 10.8 takes the logged output from Step 2 and normalizes features into `normed_judgments`.

Listing 10.8 Normalize our logged LTR training set to a normalized one

```
means, std_devs, normed_judgments = normalize_features(logged_judgments)
logged_judgments[360], normed_judgments[360]
```

Results

```
(Judgment(grade=1,qid=11,keywords=social network,doc_id=37799,
  ➥features=[8.243603, 3.8143613, 2010.0],weight=1,   ❶
 Judgment(grade=1,qid=11,keywords=social network,doc_id=37799,
   ➥features=[4.482941696779275, 2.100049660821875, 0.8347155270825962],
   ➥weight=1)   ❷
```

❶ Unnormalized example, with raw title_bm25, overview_bm25, and release_year

❷ Same example, normalized

You can see that the output from Listing 10.8 shows first the logged BM25 scores for title and overview (8.243603, 3.8143613), alongside the release year (2010). These features are then normalized, where 8.243603 `title_bm25` corresponds to 4.4829 standard deviations above the mean `title_bm25`, and so on for each feature.

We plot the normalized features in Figure 10.7. This looks very similar to Figure 10.6, with only the scale on the each axis differing.

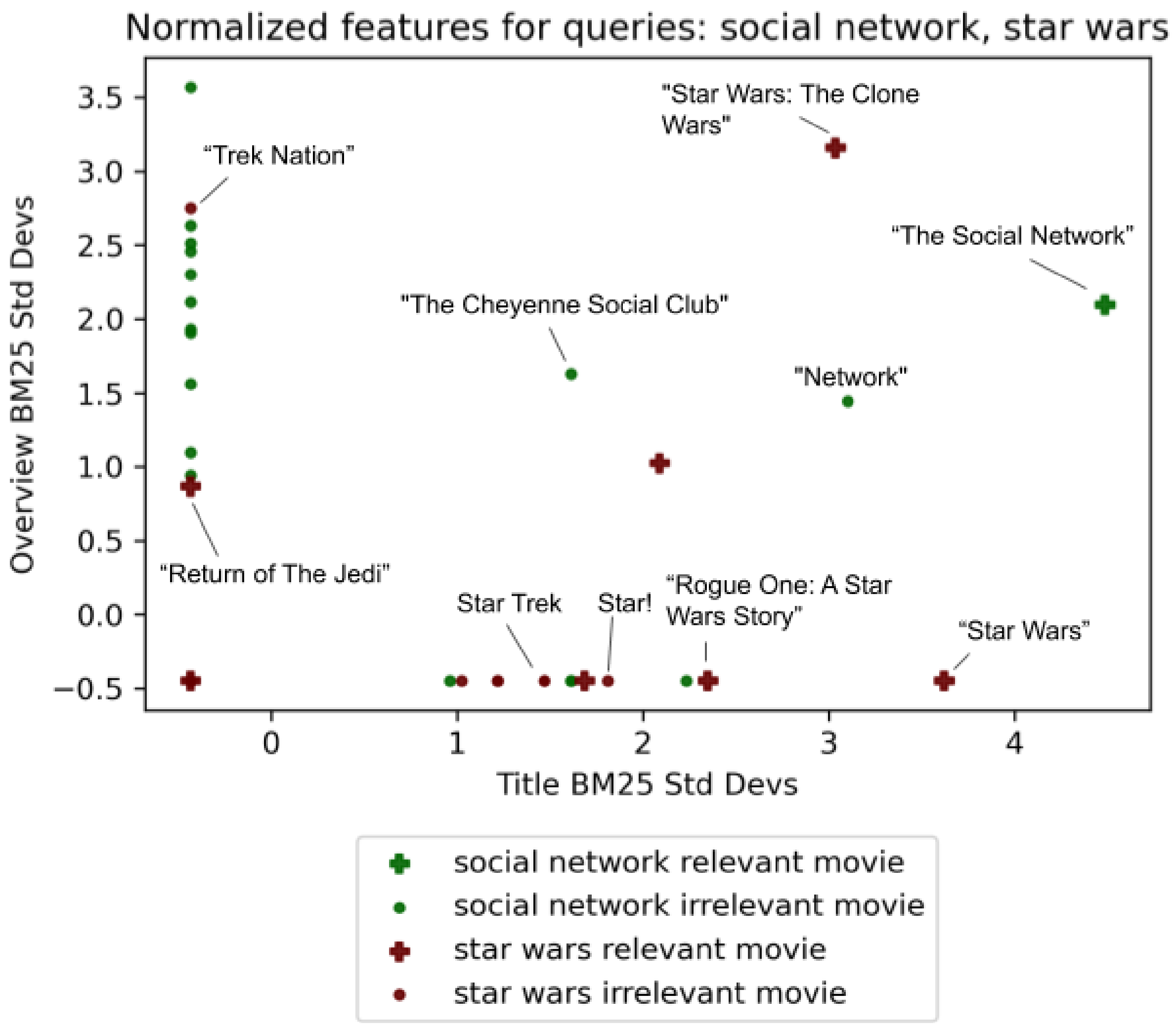


Figure 10.7 Normalized `star wars` and `social network` graded movies. Each increment in the graph is a standard deviation above or below the mean.

Next we'll turn ranking into a binary classification learning problem to separate the relevant from irrelevant results.

SECOND, COMPUTE THE PAIR-WISE DIFFERENCES

With normalized data, we've forced features to a consistent range. Now our SVM should not be biased by features that just so happen to have very large ranges. In this section we're ready to transform the task into a binary classification problem, setting the stage for us to train our model!

SVM Rank uses a *pair-wise* transformation to reformulate LTR to a binary classification problem. *Pair-wise* simply means turning ranking into the task of minimizing out-of-order pairs for a query.

In the rest of this section, we'll carefully walk through SVMRank's pair-wise algorithm, outlined in the psuedocode in Listing 10.9. Before we do that, let's discuss the algorithm at a high level.

This algorithm takes every query's judgment, and compares it to every other judgment for that same query. It computes the feature differences (`feature_deltas`) between every relevant and irrelevant pair for that query. When adding to `feature_deltas`, if the first judgment is more relevant than the second, it's labeled with a `+1` in `predictor_deltas` - and vice versa. In the end, the algorithm builds a brand new training set from the old one: the `feature_deltas` and `predictor_deltas`, which is more suitable to binary classification.

Listing 10.9 Pseudocode for SVMRank training data transformation.

```
foreach query in queries:
   foreach judged_document_1 in query.judgments:
      foreach judged_document_2 in query.judgments:
         if judged_document_1.grade > judged_document_2.grade:
            predictor_deltas.append(+1)  ❶
            feature_deltas.append( judged_document_1.features -
              ➥judged_document_2.features)  ❷
         else if  judged_document_1.grade < judged_document_2.grade:
            predictor_deltas.append(-1)  ❸
            feature_deltas.append( judged_document_1.features -
              ➥judged_document_2.features)  ❷
```

❶ Store a label of +1 for 1 more relevant than 2

❷ Store the feature deltas

❸ Store a label of -1 for 1 less relevant than 2

Figure 10.9 plots the resulting `feature_deltas` for the `social network` and `star wars` data from Figure 10.8, with some prominent pair-wise differences highlighted.

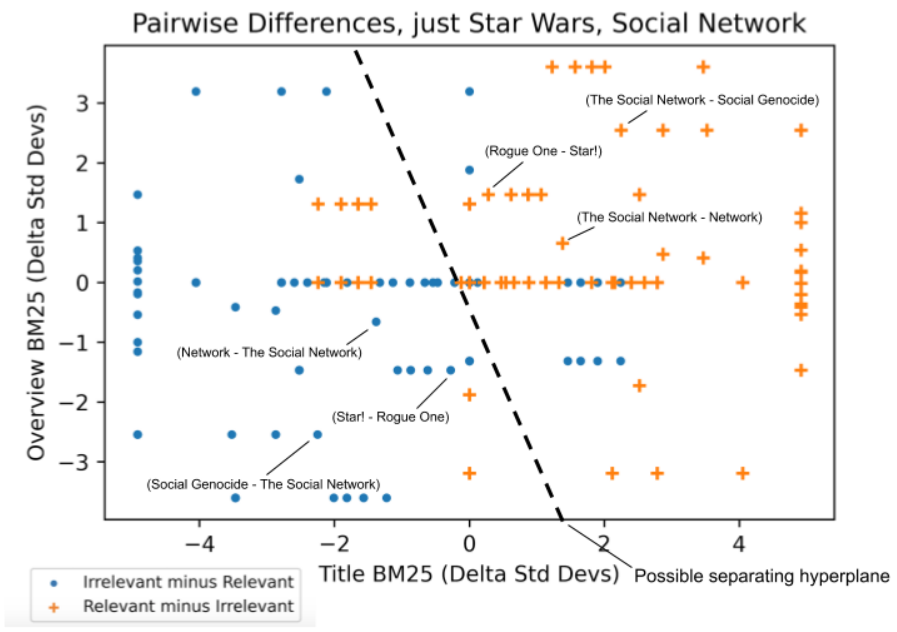


Figure 10.8 Pair-wise differences after SVMRank's transformation for `social network` and `star wars` documents along with a candidate separating hyperplane.

You'll notice the relevant minus irrelevant pair-wise deltas (+) tend to be towards the upper right. Meaning relevant documents have a higher `title_bm25` and `overview_bm25` when compared to irrelevant ones.

Ok, that's a lot to digest! Fret not - baby steps! If we walk through a few examples carefully, step-by-step, you'll see how this algorithm constructs the data points in Figure 10.9. This algorithm compares relevant and irrelevant documents for each query. So, first let's compare two documents ("Network" and "The Social Network") within the query `social network`, as in Figure 10.9.

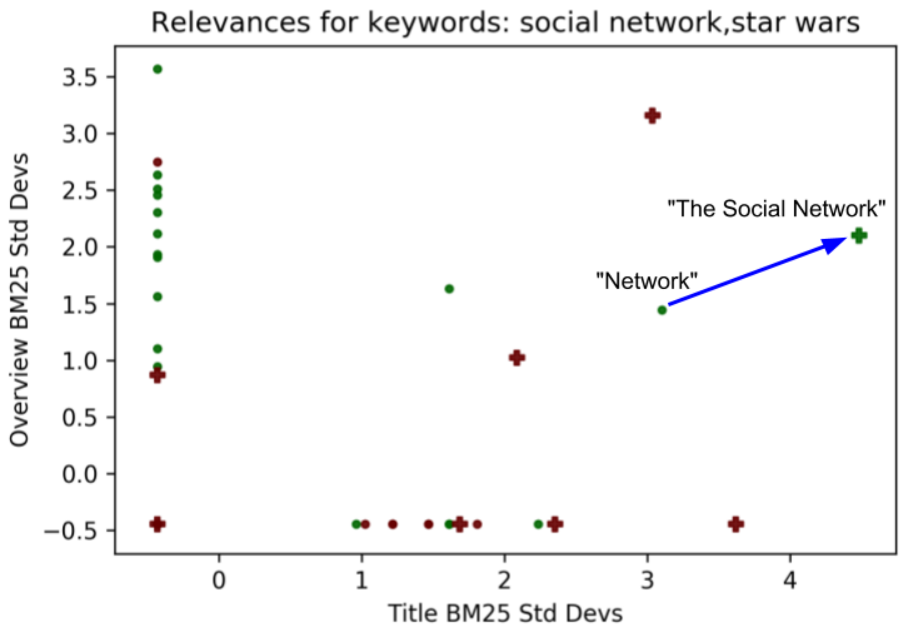


Figure 10.9 Comparing "Network" to "The Social Network" for query `social network`

The features for "The Social Network" are:

```
[4.483, 2.100]  # title_bm25 is 4.483 stddevs above mean, overview_bm25 is
➥2.100 stddevs above mean
```

The features for "Network" are:

```
[3.101, 1.443]  # title_bm25 is 3.101 stddevs above mean, overview_bm25 is
➥1.443 stddevs above mean
```

We then insert the delta between "The Social Network" and "Network" in Listing 10.10.

Listing 10.10 Labeling "The Social Network" vs "Network" delta into feature_deltas

```
predictor_deltas.append(+1)
feature_deltas.append( [4.483, 2.100] - [3.101, 1.443]) # appends [
➥1.382, 0.657]
```

Restating Listing 10.10 in English, we might say, here is one example of a movie, "The Social Network", that’s more relevant than another, "Network" for this query `social network`. Interesting! Let’s look at what makes them different! Of course "difference" in math means subtraction, which we do here. Ah yes, after taking the difference we see "The Social Network"'s `title_bm25` is 1.382 standard deviations higher than "Network"'s; `overview_bm25` is 0.657 standard deviations higher. Indeed, note the + for "The Social Network - Network" in Figure 10.8 showing the point [1.382, 0.657] amongst the deltas.

The algorithm would also note "Network" is less relevant than "The Social Network", as shown in Figure 10.10.

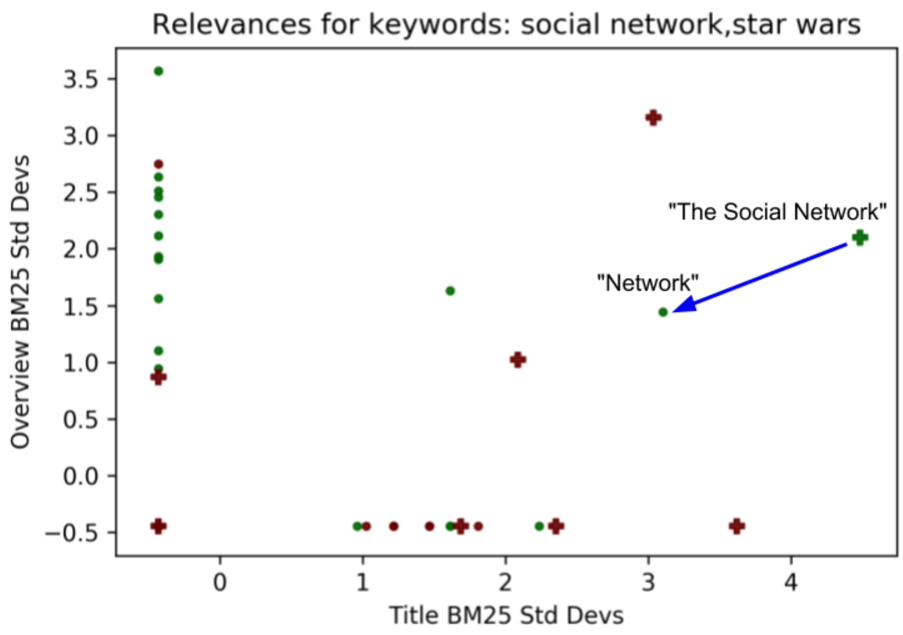


Figure 10.10 Comparing "Network" to "The Social Network" for the query `social network`

Just as in Listing 10.9, we capture in code this difference in relevance between these two documents. This time, however, in the opposite direction (irrelevant minus relevant). So no surprise we see the same values, but in the negative.

```
predictor_deltas.append(-1)
feature_deltas.append([3.101, 1.443] - [4.483, 2.100] ) # [-1.382, -0.657]
```

In Figure 10.11, we move onto another relevant-irrelevant comparison of two documents for the query `social network`, appending another comparison to the new training set.

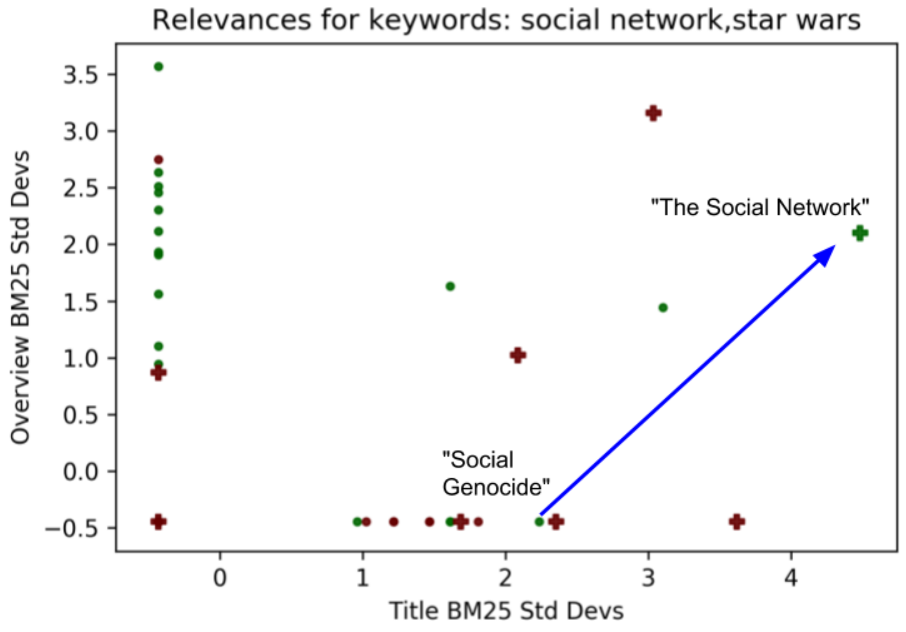


Figure 10.11 Comparing 'Social Genocide' to 'The Social Network' for the query `social network`

We show appending both directions of the comparison from Figure 10.11 in Listing 10.11, adding both a positive value (when the more relevant document is listed first) and a negative value (when the less relevant document is listed first).

Listing 10.11 Adding "Social Genocide" and "The Social Network" to point-wise training data

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append( [4.483, 2.100] - [2.234, -0.444]) # [2.249, 2.544]
# Negative example
predictor_deltas.append(-1)
feature_deltas.append([2.234, -0.444] - [4.483, 2.100]  ) # [-2.249, -2.544]
```

Once we iterate every pair-wise difference between documents matching the query `social network` to create a point-wise training set, we move on to also logging differences for other queries. Figure 10.12 shows differences for a second query, this time comparing the relevance of documents matching the query `star wars`.

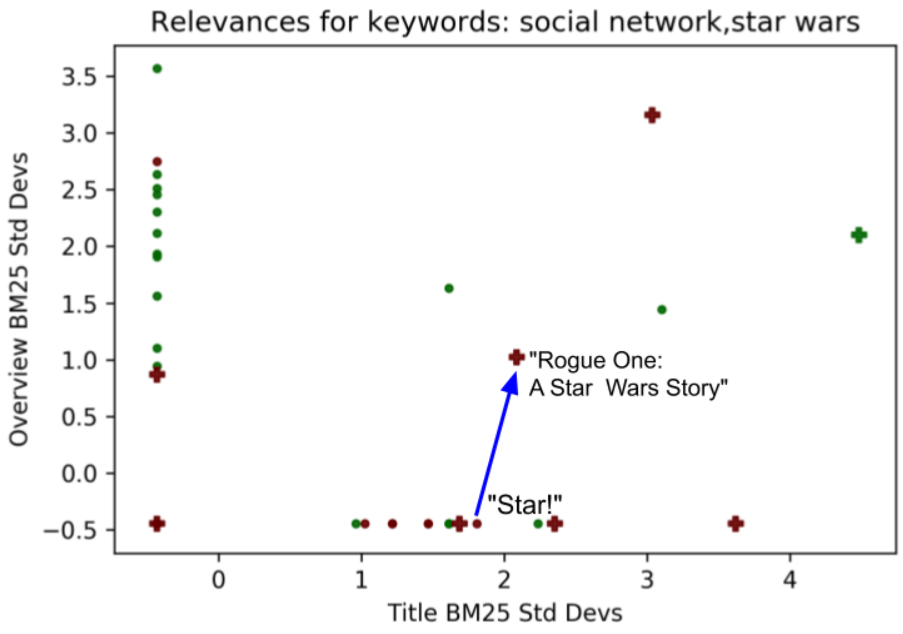


Figure 10.12 Comparing 'Rogue One: A Star Wars Movie' to 'Star!' for the query 'star wars'. Now we've moved on from `social network` and have begun to look at patterns within another query.

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append( [2.088, 1.024] - [1.808, -0.444])   ❶
# Negative example
predictor_deltas.append(-1)
feature_deltas.append([1.808, -0.444] - [2.088, 1.024])   ❷
```

❶ Rogue One minus Star!

❷ Star! minus Rogue One

We continue this process of calculating differences between feature values for relevant vs. irrelevant documents, until we have calculated all the pairwise differences together for our training and test queries.

You could see back in Figure 10.8 that the positive examples show a positive `title_bm25` delta, and possibly a slightly positive `overview_bm25` delta. This becomes even more clear if if we calculate deltas over the full dataset of 100 queries, as shown in Figure 10.13.

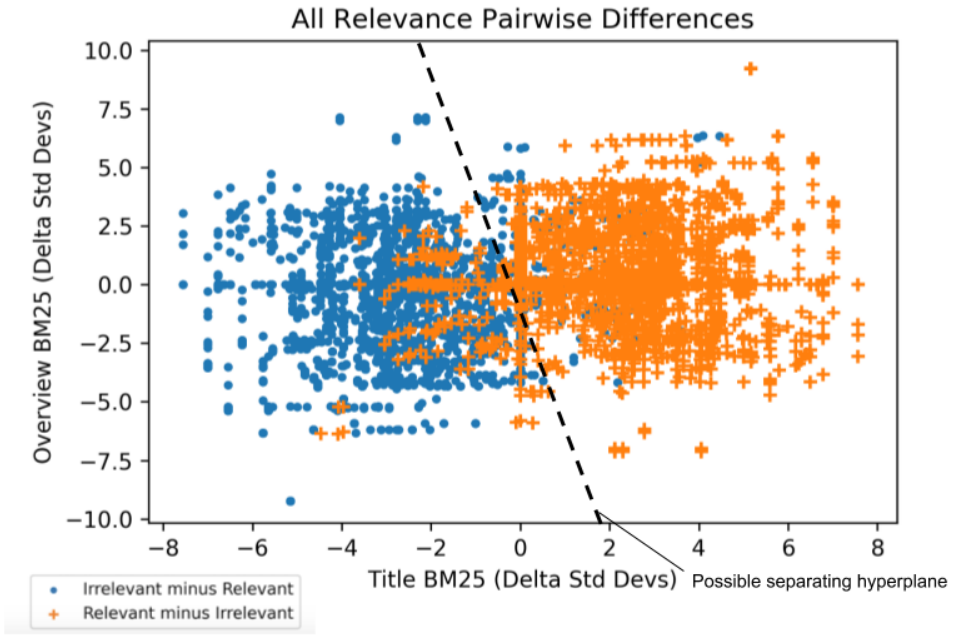


Figure 10.13 Full training set with hyperplane separating relevant from irrelevant documents. We see a pattern! Relevant documents have a higher `title_bm25` and perhaps a modestly higher `overview_bm25`.

Interesting! It is now very easy to visually identify that a larger Title BM25 score match is highly correlated with a document being relevant for a query, and that having a higher Overview BM25 score is at least somewhat positively correlated.

It's worth taking a step back and asking whether this formulation of ranking is appropriate for your domain. Other LTR methods have their own ways of performing this trick of mapping pair-wise comparisons into classification problems, but it's important to get under the hood to see how your chosen solution performs. Other methods, like LambdaMART, perform their own tricks, but they may alternatively directly optimize for classic search relevance ranking metrics like precision or Discounted Cumulative Gain (DGC).

Next up, we'll cover how to train a robust model to capture the patterns in our fully-transformed ranking data set.

10.5 Step 4—Training (and testing!) the model

Good machine Learning clearly requires a lot of data preparation! Luckily, you've arrived at the section where we actually train a model! With the `feature_deltas` and `predictor_deltas` from the last section, we now have a training set suitable for training a classic machine learning model. This model will let us predict when documents might be relevant: even for queries and documents it hasn't seen yet!

10.5.1 Turning a separating hyperplane's vector into a scoring function

We've seen how SVMRank's separating hyperplane can classify and separate irrelevant examples from the relevant ones. You might be thinking, "that's cool, but our task was actually to find 'optimal' weights for our features, not just to classify documents!". Let's look at how to actually *score* search results using this hyperplane!

It turns out the separating hyperplane also gives us just what we need to learn optimal weights. Any hyperplane is defined by the vector orthogonal to the plane. So when an SVM machine learning library does it's work, it actually gives us a sense of the weights that each feature should have, as shown in Figure 10.14.

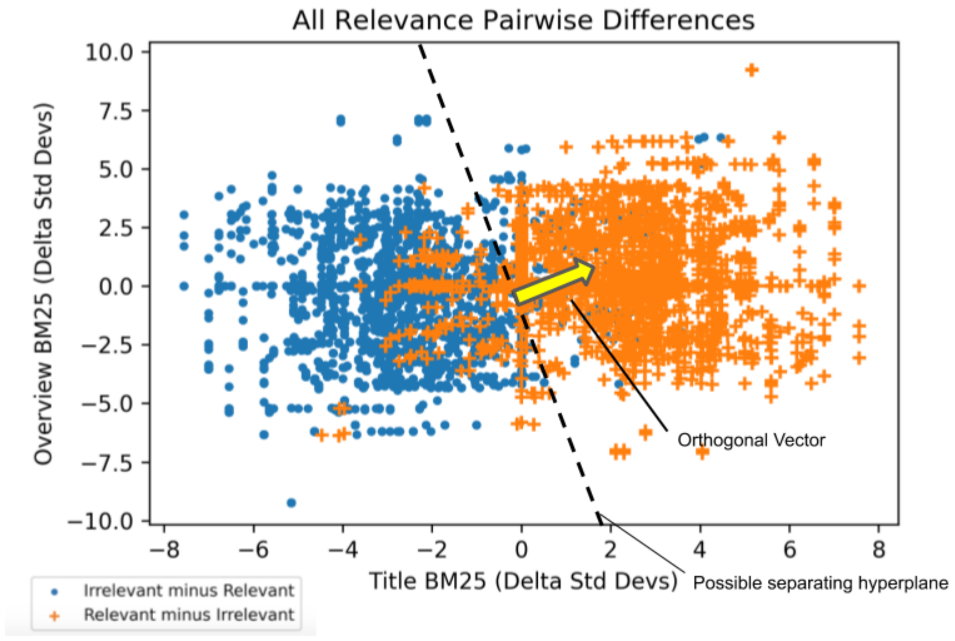


Figure 10.14 Full training set with candidate separating hyperplane, showing the orthogonal vector defining the hyperplane. Note the orthogonal vector points in the direction of relevance.

Think about what this orthogonal vector represents. This vector points in the direction of relevance! It says relevant examples are this way, and irrelevant ones are in the opposite direction. This vector *definitely* points to `title_bm25` having a strong influence on relevance, with some smaller influence from `overview_bm25`. This vector might be something like:

```
[0.65, 0.40]
```

We used the algorithm in Listing 10.9 to compute the deltas needed to perform classification between irrelevant / relevant examples. If we train an SVM on this data, as in Listing 10.12, the model gives us the vector defining the separating hyperplane.

Listing 10.12 Training a linear SVM with scikit learn

```
from sklearn import svm
model = svm.LinearSVC(max_iter=10000, verbose=1)   ❶
model.fit(feature_deltas, predictor_deltas)   ❷
model.coef_   ❸
```

❶ Create a Linear Model with sklearn

❷ Fit to deltas using an SVM

❸ The vector that defines the separating hyperplane

Results:

```
array([[0.40512169, 0.29006365, 0.14451721]])
```

Listing 10.12 trains an SVM to separate the `predictor_deltas` (remember they're `+1` and `-1` s) using the corresponding `feature_deltas` (the deltas in normalized `title_bm25`, `overview_bm25` and `release_year` features). The resulting model is a vector orthogonal to the separating hyperplane. As expected, it shows a strong weight on `title_bm25`, a more modest one on `overview_bm25`, and a weaker weight on `release_year`.

10.5.2 Taking the model for a test drive

How does this model work as a ranking function? Let's suppose the user types in the query `wrath of khan`. How might this model score the document "Star Trek II: The Wrath of Khan" relative to the keywords `wrath of khan` using this model? The unnormalized feature vector indicates a strong title and overview match for `wrath of khan`:

```
[5.9217176, 3.401492, 1982.0]
```

Normalizing it, each feature value is this many standard deviations above/below each feature's mean:

```
[3.099, 1.825, -0.568]
```

We simply multiply each normalized feature with its corresponding `coef_` value. Summing them together, gives us a relevance score:

```
(3.099 * 0.405) + (1.825 * 0.290) + (-0.568 * 0.1445) = 1.702
```

How would this model rank "Star Trek III: The Search for Spock" relative to "Star Trek II: Wrath of Khan" for our query `wrath of khan`? Hopefully not nearly as highly! Indeed it doesn't:

```
[0.0,0.0,1984.0] # Raw Features
[-0.432, -0.444, -0.468] # Normalized features
(-0.432 * 0.405) + (-0.444 * 0.290) + (-0.468 * 0.1445) = -0.371
```

The model seems to be getting what we suspect to be the right answer towards the top.

10.5.3 Validating the model

Kicking the tires on 1-2 queries helps us spot problems, but we'd prefer a more systematic way of checking if the model is generalizable. After all, we want to sleep at night knowing search will work on queries this model hasn't seen yet!

One difference between LTR and classic machine learning is we usually evaluate queries, not individual data points, to prove our model is effective. So we'll perform a test/training split at the query level. It will let us spot types of queries with problems. We'll evaluate using a simple precision metric, counting the proportion of results in the top N (with N=4 in our case) that are relevant. You should choose the relevance metric best suited to your own use case.

First we will randomly put our queries into a test or training set, as shown in Listing 10.13.

Listing 10.13 Simple test/training split at the query level.

```
all_qids = list(set([j.qid for j in normed_judgments]))
random.shuffle(all_qids)
proportion_train=0.1

test_train_split_idx = int(len(all_qids) * proportion_train)    ❶
test_qids=all_qids[:test_train_split_idx]
train_qids=all_qids[test_train_split_idx:]

train_data = []; test_data=[]
for j in normed_judgments:   ❷
    if j.qid in train_qids:
        train_data.append(j)
    elif j.qid in test_qids:
        test_data.append(j)
```

❶ Place some queries in a training on test set

❷ Place the judgments in either the training or test set based on their associated query id

With training data split, we can perform the pairwise transform trick from Step 3. We can then retrain on just the training data in Listing 10.14.

Listing 10.14 Train just on training data

```
train_data_features, train_data_predictors = pairwise_transform(train_data)

from sklearn import svm
model = svm.LinearSVC(max_iter=10000, verbose=1)
model.fit(train_data_features, train_data_predictors)   ❶
model.coef_[0]
```

❶ Fit only to training data

Now we have held back the test data. Just like a good teacher, we don't want to give the "student" all the answers. We want to see if the model has learned anything beyond rote memorization of training examples.

In Listing 10.15 we evaluate our model using the test data. This code loops over every test query and ranks every test judgment using the model (`rank` function omitted). It then computes precision for the top 4 judgements.

Listing 10.15 Can our model generalize beyond the training data?

```
def eval_model(test_data, model):

    tot_prec = 0
    num_queries = 0

    for qid, query_judgments in groupby(test_data, key=lambda j: j.qid):   ❶
        query_judgments = list(query_judgments)

        ranked = rank(query_judgments, model)   ❷

        tot_relevant = 0
        for j in ranked[:4]:   ❸
            if j.grade == 1:
                tot_relevant += 1
        query_prec = tot_relevant/4.0
        tot_prec += query_prec
        num_queries += 1

    return tot_prec / num_queries
```

❶ For each test query

❷ Score each example and rank this query using the model

❸ Compute precision for this query

On multiple runs, you should expect a precision of approximately 0.3 - 0.4 . Not bad for our first iteration, where we just guessed at a few features (`title_bm25`, `overview_bm25`, and `release_year`)!

In LTR, you can always look back at previous steps to see what might be improved. This precision test is the first time we've been able to systematically evaluate our model, so it's a natural time to revisit the features to see how the precision might be improved in subsequent runs. Go all the way back up to Step 2. See what examples are on the wrong side of the separating hyperplane. For example, if you look at Figure 10.8, the 3rd Star Wars movie, "Return of the Jedi", fits a pattern of a relevant document that doesn't have a keyword match in the title. In the absence of a title, what other features might be added, to help capture that a movie belongs in a specific collection like Star Wars? Perhaps there is a TMDB movie property indicating this we could experiment with?

For now, though, let’s take the model we just built and see how to deploy it to production.

10.6 Steps 5 and 6 - upload a model and search

In this section, let’s finally tell Solr about our model. Then we can see the fruit of our efforts: actual search results!

Originally we presented our objective to be finding 'ideal' boosts for a manual ranking function like the one in Listing 10.2:

```
q="title:(${keywords})^10 overview:(${keywords})^20
➥{{!func}}release_year^0.01"
```

This manual function indeed multiplies each feature by a weight (the boost), and sums the results. But it turns out we actually don’t want Solr to multiply the *raw* feature values. Instead we need the feature values to be normalized.

Luckily, Solr LTR lets us store a linear model along with feature normalization statistics. We saved off the `means` and `std_devs` of each feature, which Solr can use to normalize values for any document being evaluated. We just need to provide this information to Solr when storing a model, as we do in Listing 10.16.

Listing 10.16 Uploading model to Solr with normalization and weights for each feature.

```
PUT http://aips-solr:8983/solr/tmdb/schema/model-store
{
  "store": "movies",   ❶
  "class": "org.apache.solr.ltr.model.LinearModel",
  "name": "movie_titles",
  "features": [
    {
      "name": "title_bm25",   ❷
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",   ❸
        "params": {   ❸
          "avg": "1.5939970007512951",   ❸
          "std": "3.689972140122766"   ❸
        }
      }
    },
    {
      "name": "overview_bm25",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1.4658440933160637",
          "std": "3.2978986984657808"
        }
      }
    },
    {
      "name": "release_year",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1993.3349740932642",
          "std": "19.964916628520722"
        }
      }
    }
  ],
  "params": {
    "weights": {
      "title_bm25": 0.40512169,   ❹
      "overview_bm25": 0.29006365,
      "release_year": 0.14451721
    }
  }
}
```

❶ Feature store to locate the features

❷ Which feature to execute before evaluating this model

❸ How to normalize this feature before applying the weight

❹ The weight of this feature in the model

Note that the model is associated with a feature store so it can lookup feature names to compute them when evaluating the model in Solr.

Finally, we can issue a search using Solr’s LTR query parser in Listing 10.17.

Listing 10.17 Rank 60,000 documents with our model to search for `harry potter`

```
request = {
    "fields": ["title", "id", "score"],
    "limit": 5,
    "params": {
      "q": "{!ltr reRankDocs=60000 model=movie_model efi.keywords=\
      ➥"harry potter\"}",   ❶
    }
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select', json=request)

resp.json()["response"]["docs"]
```

❶ Execute our model over 60000 documents with the specified parameters

Results:

```
{'id': '570724', 'title': ['The Story of Harry Potter'], 'score': 2.786719,
➥'_score': 2.786719}
{'id': '116972', 'title': ['Discovering the Real World of Harry Potter'],
➥'score': 2.5646381, '_score': 2.5646381}
{'id': '672', 'title': ['Harry Potter and the Chamber of Secrets'],
➥'score': 2.3106465, '_score': 2.3106465}
{'id': '671', 'title': ["Harry Potter and the Philosopher's Stone"],
➥'score': 2.293983, '_score': 2.293983}
{'id': '393135', 'title': ['Harry Potter and the Ten Years Later'],
➥'score': 2.2162843, '_score': 2.2162843}
```

Notice in Listing 10.17 the use of the term 'rerank'. As this implies, LTR usually happens as a second phase on top of another query. In Listing 10.17 there's not an initial query. So what happens in this case? The model is applied to the documents in the order they were indexed (essentially random). You can see from the results that the model seems effective at our query, since it ranked a full 60,000 documents.

Usually, we don't want to rerank over such a large set of results, though. We'd rather have a quick, baseline match with simple scoring (such as BM25 or a simple edismax query), and THEN choose to do a more expensive rerank leveraging LTR over a smaller number of candidate documents. Listing 10.18 demonstrates how to implement a baseline search, reranking the top 500 using our LTR model.

Listing 10.18 Rerank top 500 with `movie_model` to search for `harry potter` after the top results from the initial, quick ranking calculation.

```
request = {
    "fields": ["title", "id", "score"],
    "limit": 5,
    "params": {
      "rq": "{!ltr reRankDocs=500 model=movie_model efi.keywords=\
      ➥"harry potter\"}",   ❶
      "qf": "title overview",   ❷
      "defType": "edismax",   ❷
      "q": "harry potter"   ❷
    }
}

resp = requests.post('http://aips-solr:8983/solr/tmdb/select', json=request)
resp.json()["response"]["docs"]
```

❶ Rerank top 500 of first pass Solr query

❷ First pass Solr query - a simple edismax query

10.6.1 A note on LTR performance

Phew, you've done it! As you can see, there's many steps required to building a real-world LTR model. Let's close the loop with some thoughts on additional practical performance constraints in LTR systems:

1. *Model complexity* - the more complex the model, the more accurate it *might* be. A simpler model can be faster and easier to understand, though perhaps less accurate. Here we've stuck to a very simple model (a set of linear weights). Imagine a complex deep-learning model, how well would that work? Would the complexity be worth it? Would it be as generalizable?
2. *Rerank depth* - the deeper you rerank, the more you might find additional documents that could be hidden gems. On the other hand, the deeper you rerank, the more compute cycles your model spends scoring results in your live search engine cluster.
3. *Feature complexity* - if you compute very complex features at query time, they might help your model. However they'll slow down evaluation and search response time.
4. *Number of features* - a model with many features might lead to higher relevance. However it will also take more time to compute every feature on each document, so ask yourself which features are crucial. Many academic LTR systems use hundreds. Practical LTR systems usually boil these down to dozens. You will almost always see diminishing returns as you continue adding additional features, so choosing the right cut-off threshold on number of features is important.

10.7 Rinse and repeat

Congrats! You've done one full cycle of Learning to Rank! Like many data problems though, you'll likely need to continue iterating on the problem. There's always something new you can do to improve.

On your second iteration, you might begin thinking through some of the following considerations:

1. *New and better features*. Are there types of queries or examples on which the model performs poorly? Such as `title` searches where there’s no `title` mention (`star wars` is not mentioned in the title of "Return of the Jedi". What features could capture these?). Could we incorporate lessons from chapters 1-9 to construct more advanced features?
2. *Training data coverage of all features*. The flip-side of more features is more training data. As you increase features you’d like to try, you should be wondering whether your training data has enough examples of relevant / irrelevant documents across each different combination of your features. Otherwise your model won’t know how to leverage features to solve the problem.
3. *Different model architectures*. We used a relatively simple model that expects features to linearly correlate with relevance. Yet relevance can often be non-linear. A shopper searching for iPads might expect the most recent Apple iPad release, except when they add the word 'cable', as in 'iPad cable'. For that query, they might just want the cheapest cable they can find.

In the next chapter, we will focus in on the foundation of good LTR: great judgments! This will give you even greater confidence in your iterations, pushing the boundaries on automating this process as your maturity increases.

10.8 Summary

- Learning to Rank (LTR) builds generalized ranking functions that can apply across all searches, using robust machine learning techniques
- In Solr LTR, features generally correspond to Solr queries. Solr LTR lets you store and log features for the purposes of training, and later applying, a ranking model.
- We have tremendous freedom in what features might be used to generalize relevance. Features could be properties of queries (like number of terms), properties of documents (like a popularity), or relationship between queries and documents (like BM25 or other relevance scores).
- To do LTR well, and apply well known machine learning techniques, we typically reformulate the relevance ranking problem into a classic machine learning problem.
- SVMRank creates simple linear weights on normalized feature values, a good first step on your LtR journey
- To be truly useful, we need our model to generalize beyond what it’s learned! We can confirm LTR can generalize by placing some training data into a test set. Then later, we can evaluate the model’s generalizability by using test data.
- Once a LTR model is loaded into your search engine, be sure to consider performance (as in speed) tradeoffs with relevance. Real-life search systems require both!

11 Building learning to rank training data from user clicks

This chapter covers

- **Automating Learning to Rank (LTR) retraining from user behavioral signals (clicks, etc.)**
- **Transforming user signals into implicit LTR training data using click models**
- **Why raw clicks alone don’t work well to build LTR training data**
- **Compensating for the user’s tendancy to click farther up the search results page, regardless of relevance**
- **Handling documents with fewer clicks in the training data**

In Chapter 10, we went step-by-step to train a Learning to Rank (LTR) model. Like walking through the mechanics of building a car, we saw the underlying nuts and bolts of LTR model training. In this chapter we treat the LTR training process as a black box. In other words, we step away from LTR internals, instead treating LTR more like a self-driving car, fine tuning its trip toward a final destination.

Recall that LTR relies on accurate training data in order to be effective. LTR training data describes how users expect search results to be optimally ranked. The training data provides the directions we input into our LTR self-driving car. As you’ll see, knowing what’s relevant based on user interactions comes with many challenges. If we can overcome these challenges and gain high confidence in our training data, though, then we can build *Automated Learning to Rank*: a system that regularly retrains LTR to capture the latest user relevance expectations.

As training data is so central to automated LTR, the challenges become not "what model/features/search engine should we use?" but more fundamentally: "*What do users want from search?*", "*How do we turn that into training data?*", and "*How do we know whether that training data is any good?*". By improving our confidence in the answers to these questions, we can put LTR (re)training on autopilot, as shown in Figure 11.1

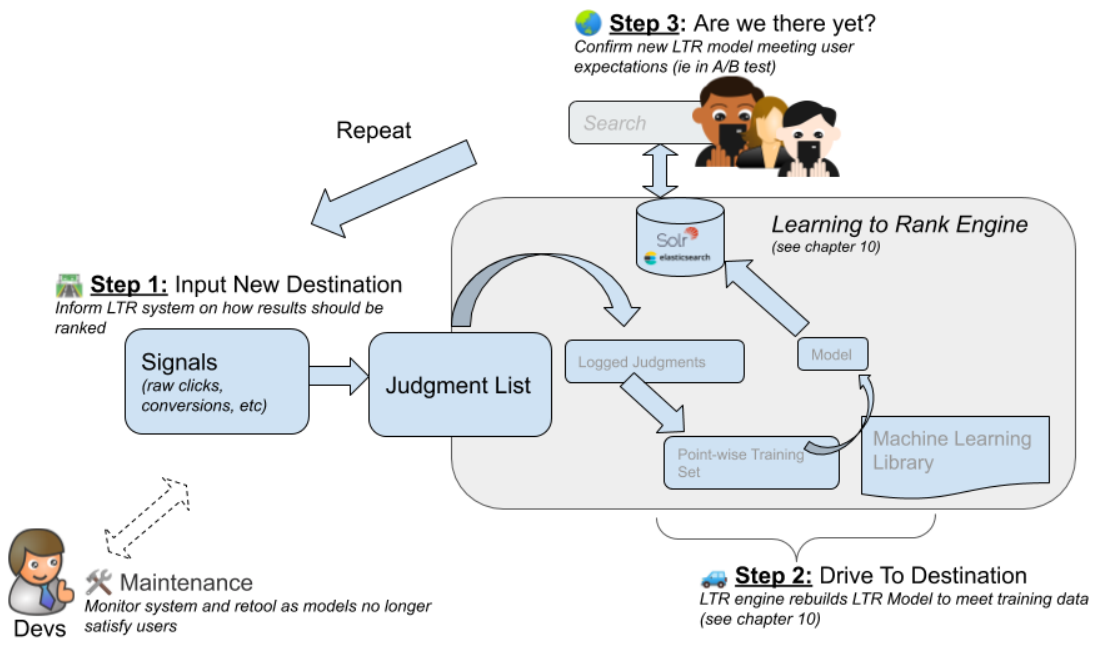

Step 3: Are we there yet?
Confirm new LTR model meeting user expectations (ie in A/B test)
Search
Repeat
Learning to Rank Engine
(see chapter 10)
Solr
elasticsearch
Step 1: Input New Destination
Inform LTR system on how results should be ranked
Signals
(raw clicks, conversions, etc)
Judgment List
Logged Judgments
Model
Point-wise Training Set
Machine Learning Library
Maintenance
Monitor system and retool as models no longer satisfy users
Devs
Step 2: Drive To Destination
LTR engine rebuilds LTR Model to meet training data
(see chapter 10)

Figure 11.1 An Automated Learning to Rank system automatically learns and retrains from the user's signals. This helps build models based on what actual users consider relevant over a many queries.

To briefly walk through each step in the Automated LTR process:

- **Step 1 - Input New Destination**: we input into the LTR system training data that describes ideal relevance based on our understanding of user behavioral signals such as searches, clicks, and conversions (covered in this chapter).
- **Step 2 - Drive To Destination**: our LTR system retrains an LTR model using the provided training data (as covered in Chapter 10).
- **Step 3 - Are We There Yet?**: Is the model truly is helping users? And should future models perhaps explore alternate routes? (covered in Chapter 12).

Automated LTR repeats steps 1-3 continuously to automatically optimize relevance. The search team monitors Automated LTR's performance and intervenes as needed. This is the *maintenance* portion in Figure 11.1. During maintenance, we open the hood to explore new LTR features and other model adjustments. Maintenance could also mean revisiting Step 1 to correct our understanding of user behaviors to build more reliable, robust training data. After all, without good training data, we could follow Chapter 10 to the tee, and still fail to satisfy our users.

This chapter starts our exploration of automated LTR by focusing on Step 1. To learn about Step 1, we'll first define the task of deriving training data from user clicks. We'll then spend this chapter overcoming common challenges with search click data. At the end of this chapter, you'll be able to build models with reliable training data. Chapter 12 finishes our automated LTR exploration by observing the model interact with live users. In particular, Chapter 12 teaches you how to overcome search data's most fundamental flaw: *presentation bias* we'll never know whether certain results are relevant unless we give them a chance to be clicked!

11.1 (Re)creating judgment lists from signals

We mentioned that we need to overcome biases when creating LTR training data from clicks. However, before we dig into those biases, we'll explore the implications of using clicks instead of manual labels for LTR training data. We'll then take a naive, first stab at crafting training data in this section, finally reflecting on what went well or not so well. This will set us up for the rest of the chapter to explore debiasing these results is sections 11.2 and beyond.

11.1.1 Generating implicit, probabilistic judgments from signals

Let's lay a foundation for how to use behavioral signals as LTR training data. Then we'll dive into the nitty gritty of constructing reliable judgment list.

In chapter 10 we discussed LTR training data, referred to as *judgment lists* or *judgments*. These are labels or *grades* for how relevant potential search results are for a given query. In chapter 10 we used movies as our example. We labeled movies with either grade of 1 (relevant) or 0 (irrelevant), such as the example in Listing 11.1.

Listing 11.1 Labeling movies relevant or irrelevant

```
mini_judg_list=[
    # for 'social network' query
    Judgment(grade=1, keywords='social network', doc_id='37799'),
    [CA]# The Social Network
    Judgment(grade=0, keywords='social network', doc_id='267752'),
    [CA]# chicagoGirl
    Judgment(grade=0, keywords='social network', doc_id='38408'),
    [CA]# Life As We Know It
    Judgment(grade=0, keywords='social network', doc_id='28303'),
    [CA]# The Cheyenne Social Club

    # for 'star wars' query
    Judgment(grade=1, keywords='star wars', doc_id='11'),
    [CA]# Star Wars, A New Hope
    Judgment(grade=1, keywords='star wars', doc_id='1892'),
    [CA]# Return of the Jedi
    Judgment(grade=0, keywords='star wars', doc_id='54138'),
    [CA]# Star Trek Into Darkness
    Judgment(grade=0, keywords='star wars', doc_id='85783'),
    [CA]# The Star
    Judgment(grade=0, keywords='star wars', doc_id='325553'),
    [CA]# Battlestar Galactica
]
```

There are many techniques for generating judgment lists. This isn't a comprehensive chapter on judgment lists and their many applications - instead we are specifically focused on LTR training data. For this reason, we only discuss judgments generated from user click signals, called *implicit judgments*. We call these judgments *implicit* because they derive from real user interactions with the search application as they search and click. This is in contrast to *explicit* judgments where raters directly label search results as relevant / irrelevant.

Implicit judgments are ideal for automating LTR for these reasons:

1. **Recency**: we have ready access to user traffic. We can automate training today's LTR model on the latest user search expectations.
2. **More data at less cost**: setting up a task to capture explicit judgments, even with crowdsourcing, is time consuming and expensive to do well at scale. Capturing judgments from live users allows us to leverage the existing user base to do this work for us.
3. **Capturing real use cases**: implicit judgments capture real users doing actual tasks with your search app. Contrast this with the artificial setting of a lab where explicit raters think carefully, perhaps unrealistically so, about artificial tasks.

However, clicks come with much more fuzziness. We don't know why a user clicked on a given search result. Further, users are not homogeneous, some will interpret one result as relevant, while others think otherwise. Search interactions also contain biases that need to be overcome, creating additional uncertainty around a model's calculations, which we'll discuss in great detail in this chapter and the next.

For these reasons, instead of a binary judgment, click models create *probabilistic judgments*. The grade represents the probability (between 0.0 to 1.0) that a random user would consider the result to be relevant or not. As an example, a good click model might restate the judgments from Listing 11.1 with something more like Listing 11.2.

Listing 11.2 Labeling movie query relevance probabilistically

```
mini_judg_list=[
    Judgment(grade=0.99, keywords='social network', doc_id='37799'),
    [CA]# The Social Network
    Judgment(grade=0.01, keywords='social network', doc_id='267752'),
    [CA]# chicagoGirl
    Judgment(grade=0.01, keywords='social network', doc_id='38408'),
    [CA]# Life As We Know It
    Judgment(grade=0.01, keywords='social network', doc_id='28303'),
    [CA]# The Cheyenne Social Club

    # for 'star wars' query
    Judgment(grade=0.99, keywords='star wars', doc_id='11'),
    [CA]# Star Wars, A New Hope
    Judgment(grade=0.80, keywords='star wars', doc_id='1892'),
    [CA]# Return of the Jedi
    Judgment(grade=0.20, keywords='star wars', doc_id='54138'),
    [CA]# Star Trek Into Darkness
    Judgment(grade=0.01, keywords='star wars', doc_id='85783'),
    [CA]# The Star
    Judgment(grade=0.20, keywords='star wars', doc_id='325553'),
    [CA]# Battlestar Galactica
]
```

Notice the Star Wars movies in Listing 11.2 - the `grade` has become quite a bit more interesting. The original Star Wars movie now has a very high probability of relevance (`0.99`). The sequel "Return of The Jedi" has a slightly lower probability. Other science fiction movies ("Star Trek Into Darkness" and "Battlestar Galactica") have ratings a bit higher than `0`, as likely the Star Wars fan might also enjoy these movies. "The Star" is completely unrelated - it's a children's animated movie about the first Christmas - so it receives a low `0.01` relevance probability.

11.1.2 Training an LTR model using probabilistic judgments

We just introduced the idea a relevance grade could be probabilistic, that is between 0-1. Now, let’s consider how to apply the lessons from Chapter 10 to train a model using these fuzzier, probabilistic judgments.

Generally, you might consider these options when training a model:

- **Quantize the grades** - quite simply you can set arbitrary cut-offs before training to convert the grades to an acceptable format. You might assign a grade > 0.75 as relevant (or `1`). Anything less than 0.75 is considered irrelevant (or `0`). Other algorithms, like LambdaMART, accept a range of grades like 1 - 4, and these could have cutoffs as well, such as assigning anything < 0.25 to a grade of 1, anything >= 0.25 but < 0.5 a grade of 2, and so on. With these algorithms, you could create 100 such labels, assigning 0.00 a grade of 0, 0.01 a grade of 1, and so on until 1 is assigned to a grade of 100 prior to training.
- **Just use the floating point judgments** - the SVMRank algorithm from chapter 10 subtracted a more relevant item’s features from a less relevant item’s features (and vice versa) and built a classifier to tell relevant from irrelevant. We did this with binary judgments, but there would be nothing preventing us from also doing this with probabilistic judgments. Here if "Return of The Jedi" (grade = 0.8) is considered more relevant than "Star Trek Into Darkness" (grade = 0.2), then we simply note "Return of The Jedi" as more relevant than "Star Trek Into Darkness" (labeling the difference as +1). Then we perform the same pairwise subtraction we would perform from chapter 10, subtracting features of "Star Trek Into Darkness" from those of "Return of The Jedi" to create a full training example.

Retraining the model with judgments in this chapter would extensively repeat the code from Chapter 10. We have included a full end-to-end LTR training example using the click model we arrive at by the end of this chapter. To view this example, visit the Notebook entitled listing x.y (as a reminder - see Appendix A for how to set up your environment).

Time to get back to the code and see our first click model!

11.1.3 Click-through Rate: Your First Click Model

Now that you see the judgments a click model generates, and how to use them to train a model, next we’ll examine a first, naive, pass at a click model. After that, we’ll take a step back to focus primarily on a more sophisticated, general purpose click model, finally exploring the core biases inherent in processing search click signals. As we focus on a single click model to automate LTR, we encourage you to read the book *Click Models for Web Search* by Chuklin, Markov, and de Rijke to see a broader selection of click models you could use.

We’ll return to the RetroTech dataset. This data conveniently comes bundled with user click signals. We’ve also reverse-engineered from these signals the kind of raw session data you need to build high-quality judgments. We’ll make use of the `pandas` library to perform tabular computations on session data.

In Listing 11.3 we examine a sample search session for the movie "Transformers Dark of The Moon". This raw session information is your starting point, the bare minimum information needed to develop a judgment list from user signals.

Listing 11.3 Examining a search session

```
QUERY='transformers dark of the moon'
query_sessions = sessions[sessions['query'] == QUERY]   ❶
query_sessions[query_sessions['sess_id'] == 2]   ❷
```

❶ Select sessions for query `transformers dark of the moon`

❷ Examine a single search session shown to the user

Output:

```
sess_id query                          rank    doc_id          clicked
2        transformers dark of the moon    0.0        47875842328        False
2        transformers dark of the moon    1.0        24543701538        True
2        transformers dark of the moon    2.0        25192107191        False
2        transformers dark of the moon    3.0        47875841420        False
2        transformers dark of the moon    4.0        786936817218    False
2        transformers dark of the moon    5.0        47875842335        False
2        transformers dark of the moon    6.0        47875841406        False
2        transformers dark of the moon    7.0        97360810042        False
2        transformers dark of the moon    8.0        24543750949        False
2        transformers dark of the moon    9.0        36725235564        False
2        transformers dark of the moon    10.0    47875841369        False
2        transformers dark of the moon    11.0    97363560449     False
2        transformers dark of the moon    12.0    400192926087    False
2        transformers dark of the moon    13.0    97363532149     False
2        transformers dark of the moon    14.0    93624956037     False
```

Listing 11.3 corresponds to a single search session, `sess_id` 2, for query `transformer dark of the moon`. This session includes the query `transformers dark of the moon`, the ranked results seen by the user, and whether each result was clicked. These three elements (query, top N results given to user, what was clicked) are the core ingredients a click model needs to do its work.

Search sessions will frequently differ. Another session, even seconds later, could have a slightly different ranking presented to the user. The search index might have changed or a new relevance algorithm been deployed to production. We encourage you to retry Listing 11.2 with another `sess_id` to compare sessions.

Let's consume this data into judgments using our first click model: simple Click-Through Rate.

BUILDING JUDGMENTS FROM CLICK-THROUGH RATE

Now that we understand the input, let's build a simple click model. We'll start simple to get comfortable with the data, and then we can step back to see the flaws from our first pass. This will allow us to think carefully about the quality of the generated judgments for Automated LTR in the rest of this chapter.

Our first click model is Click-Through Rate. *Click-Through Rate* or *CTR* is the number of clicks received on a search result divided by the number of times it appeared in search results. If a result is clicked every single time the search engine returns the result, the CTR will be 1. If it's never clicked, CTR will be 0. Sounds simple enough - what could go wrong?

Let's see CTR in action. We can look over every result for our query `transformers dark of the moon` and consider clicks with respect to the number of sessions in which the `doc_id` was returned. Listing 11.4 shows a CTR computation and the resulting CTR value per document.

Listing 11.4 Computing Click-Through Rate

```
QUERY='transformers dark of the moon'
query_sessions = sessions[sessions['query'] == QUERY]   ❶

click_counts = query_sessions.groupby('doc_id')['clicked'].sum()   ❷
sess_counts = query_sessions.groupby('doc_id')['sess_id'].nunique()   ❸
ctrs = click_counts / sess_counts   ❹

ctrs.sort_values(ascending=False)   ❺
```

❶ Select sessions for query `transformers dark of the moon`

❷ Sum all clicks for each product identifier (`doc_id`)

❸ Count the sessions that product occurs in

❹ Determine the proportion of sessions that received clicks (CTR)

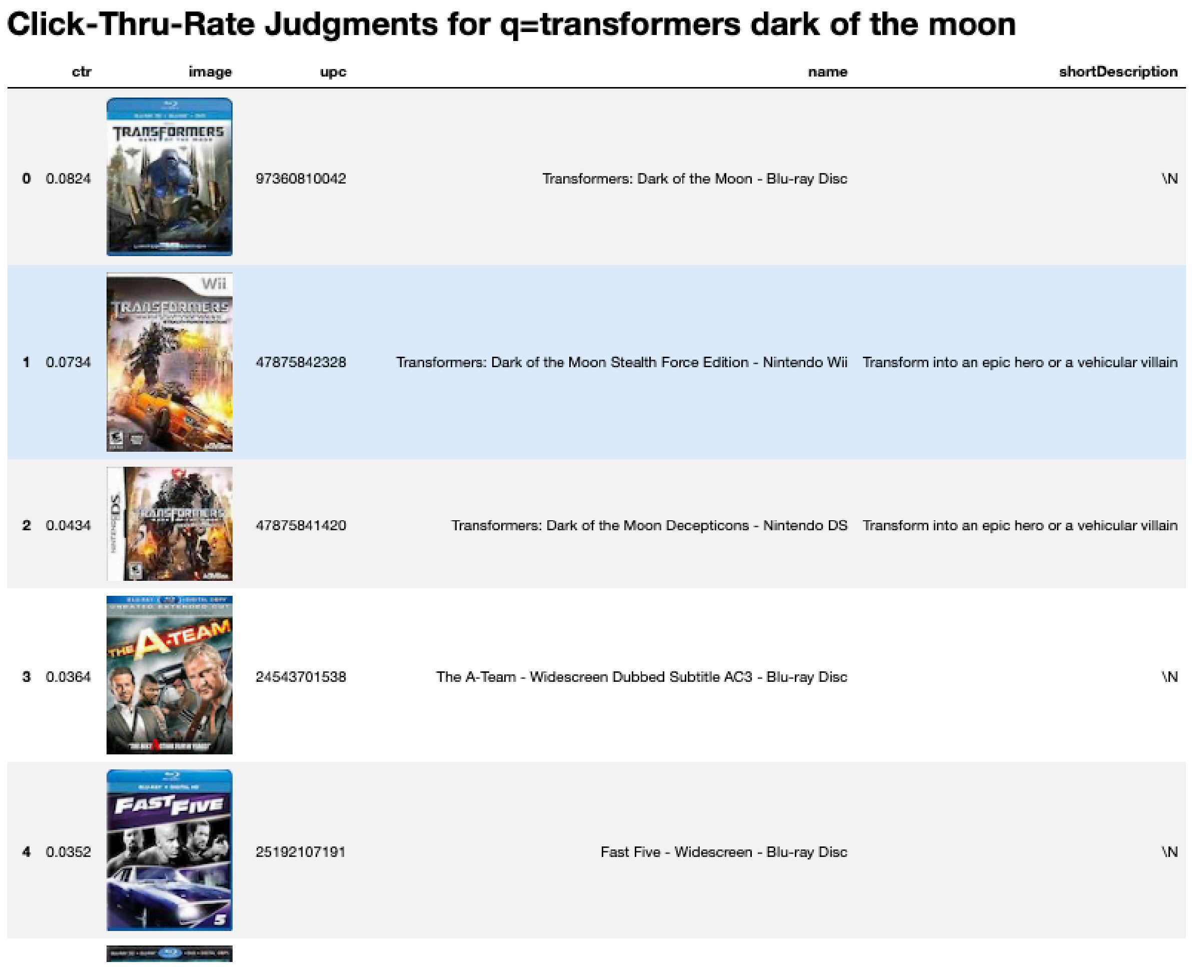
❺ Display top clicked results

Output:

```
doc_id          CTR
97360810042     0.0824
47875842328     0.0734
47875841420     0.0434
24543701538     0.0364
25192107191     0.0352
786936817218    0.0236
97363560449     0.0192
47875841406     0.0160
400192926087    0.0124
47875842335     0.0106
97363532149     0.0084
93624956037     0.0082
36725235564     0.0082
47875841369     0.0074
24543750949     0.0062
```

In Listing 11.4 we look over every session to sum the clicks per `doc_id` computing `clicks_per_count`. We also count the number of unique sessions for that document in `sess_counts`. Finally, we compute `ctrs` by dividing `click_count / sess_counts`. We see that document `97360810042` has the highest CTR and `24543750949` the lowest.

The snippet outputs the *ideal search results* according to the click model. That is, if our LTR model did everything right, it would produce this exact ranking. Throughout this chapter and the next, we'll frequently visually display this ideal ranking to understand whether the click model builds reasonable training data. We can see CTR's rendered ideal for `transformers dark of the moon` in Figure 11.2.


Click-Thru-Rate Judgments for q=transformers dark of the moon

	ctr	image	upc	name	shortDescription
0	0.0824		97360810042	Transformers: Dark of the Moon - Blu-ray Disc	\N
1	0.0734		47875842328	Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii	Transform into an epic hero or a vehicular villain
2	0.0434		47875841420	Transformers: Dark of the Moon Decepticons - Nintendo DS	Transform into an epic hero or a vehicular villain
3	0.0364		24543701538	The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc	\N
4	0.0352		25192107191	Fast Five - Widescreen - Blu-ray Disc	\N

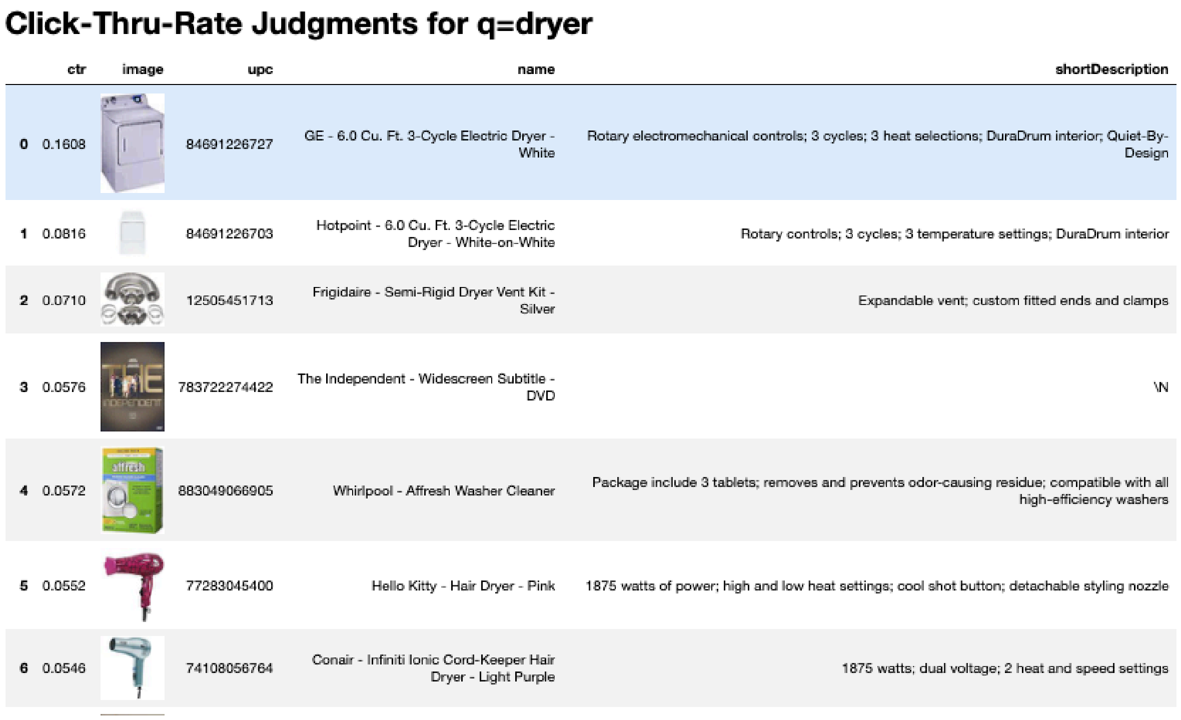
Figure 11.2 Ideal search results according to click-through rate for "Transformers Dark Of The Moon". Viewing a click model's ideal search results lets us see where our LTR model will be steered.

Examining the results of Figure 11.2, a couple things jump out:

1. The CTR for our top result (The blu ray of the movie "Transformers: Dark of the Moon") seems rather low (0.0824, only a little better than the next 0.0734 judgment). We might expect the blu ray's relevance grade to be much higher than other results.
2. The DVD for "Transformers: Dark of The Moon" doesn't even show up. It sits far below seemingly unrelated movies and secondary video games about the movie Dark of The

Moon. We would expect the DVD to rank higher, maybe as high or higher than the Blu Ray.

But perhaps `transformers dark of the moon` is just a weird query. Let's repeat for something completely unrelated. This time `dryer` in Figure 11.3.



Click-Thru-Rate Judgments for q=dryer

	ctr	image	upc	name	shortDescription
0	0.1608		84691226727	GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White	Rotary electromechanical controls; 3 cycles; 3 heat selections; DuraDrum interior; Quiet-By-Design
1	0.0816		84691226703	Hotpoint - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White-on-White	Rotary controls; 3 cycles; 3 temperature settings; DuraDrum interior
2	0.0710		12505451713	Frigidaire - Semi-Rigid Dryer Vent Kit - Silver	Expandable vent; custom fitted ends and clamps
3	0.0576		783722274422	The Independent - Widescreen Subtitle - DVD	\N
4	0.0572		883049066905	Whirlpool - Affresh Washer Cleaner	Package include 3 tablets; removes and prevents odor-causing residue; compatible with all high-efficiency washers
5	0.0552		77283045400	Hello Kitty - Hair Dryer - Pink	1875 watts of power; high and low heat settings; cool shot button; detachable styling nozzle
6	0.0546		74108056764	Conair - Infiniti Ionic Cord-Keeper Hair Dryer - Light Purple	1875 watts; dual voltage; 2 heat and speed settings

Figure 11.3 Ideal search results according to click-through rate for "Dryer". Here we note the strange movie "The Independent" that perhaps isn't actually relevant for this query?

In Figure 11.3 we see other odd looking results:

1. First two results are clothes dryers, this seems good
2. Following the clothes dryers are clothes dryer parts. Hmm OK?
3. A movie called "The Independent" shows up? This seems completely random. Why would this be rated so highly?
4. Next there's a washer accessory. Kind of related.
5. Finally, hair dryers, which, perhaps is another meaning of the word `dryer`

What do you think of the judgments produced by CTR? Think back to what you learned in Chapter 10. Remember this is the foundation, the very target, of your LTR model. Do you think they would lead to a good LTR model that would ultimately succeed if put into production?

We also encourage you to ask yourself a more fundamental question: how could we even tell if a judgment list is good? Our subjective interpretation could be as flawed as the data in a click model! We'll consider this in more depth in Chapter 12. For this chapter, we'll let our instincts guide us to possible issues.

11.1.4 Common biases in judgments

We've seen so far that we can create probabilistic judgments - those with grades between 0-1 - simply by dividing the number of clicks on a product by the number of times that product is returned by search. The output, however, seemed to be a bit wanting as it included movies actually unrelated to the "Transformers" franchise. We also saw a movie placed in the search results for `dryer`!

It turns out search click data is full of biases. Here we'll briefly define what we mean by 'bias', before going step-by-step to explore each of these biases in the RetroTech click data.

With click models, a *bias* is a reason raw user click data can have nothing to do with the relevance of search results. Instead biases define how clicks (or lack of clicks) reflect user psychology, search user interface design, or noisy data. We separate biases into two broad groups. *Non algorithmic* and *algorithmic* biases. Algorithmic biases are biases inherent in the ranking, display, and interaction with search results. Non algorithmic biases occur for reasons only indirectly related to search ranking.

Algorithmic biases can include:

1. *Position Bias*: Users click on higher ranked results more than lower ranked results
2. *Confidence Bias*: Documents with little signal data influence the judgments the same as documents with much more data
3. *Presentation Bias*: If search never surfaces results, users never click them. So the click model won't know whether they're relevant!

Non-algorithmic biases, on the other hand, are biases like:

1. *Attractiveness Bias*: Some results appear attractive and generate clicks (perhaps due to better images or wording selection), but turn out to be spammy or just irrelevant
2. *Performance Bias*: Users give up on slow search, get distracted, and end up not clicking anything

This book is *AI-Powered* Search after all, so we will focus our discussion on *algorithmic* biases in search clickstream data. But, of course, non-algorithmic biases matter too! Search is a complex ecosystem that goes beyond relevance ranking. If results are frequently clicked, but follow on actions like sales or other conversions don't occur, it might not quite be a ranking problem, but perhaps you have a problem with spammy products. Or you might have an issue with the product pages or checkout process. You may find yourself asked to improve 'relevance' when the limiting factor is actually the user experience, the content, or the speed of search. For more, we recommend the article "An Introduction to Search Quality" by Max Irwin (https://opensourceconnections.com/blog/2018/11/19/an-introduction-to-search-quality/)

As we explore additional click models, we'll cover position bias and confidence bias in the rest of this chapter. Presentation bias will be covered in Chapter 12. For an additional, comprehensive perspective on overcoming biases, we recommend the paper *Click Probability, Certainty and Context: A practical Bayesian approach to deriving search relevance judgments from web tracking* by René Kriegler.

Now that we've reflected on our first click model, let's work to overcome the first bias.

11.2 Overcoming Position Bias: The Search Engine Returned it higher, it must be better!

In the previous section, we saw our first click model in action: simple Click-Through Rate. This divided the number of times a product was clicked in search by the number of times it was returned in the top results. We saw that this was quite a flawed approach, noting numerous reasons this could be biased. It's time to begin tackling those issues!

Let's work hands-on with a click model designed to overcome position bias, often the first bias click models work to overcome.

11.2.1 Defining Position Bias

Position bias is present in of most search systems. If users are shown search results, they tend to prefer highly ranked search results over lower ones - even when those lower results are in fact more relevant. Joachims, et. al in their paper *Evaluating the Accuracy of Implicit Feedback from Clicks and Query Reformulations in Web Search* (https://www.cs.cornell.edu/people/tj/publications/joachims_etal_07a.pdf) discuss several reasons for position biases to exist:

1. *Trust bias* - users trust the search engine must know what it's doing, so they interact with higher results more
2. *Scanning behaviors* - users examine search results in specific patterns, such as top-to-bottom, and often don't explore everying in front of them
3. *Visibility* - higher ranked results are likely to be rendered in the users screen, so users need to scroll to see the remaining results

With these factors in mind, let's see if we can detect position bias in the RetroTech sessions.

11.2.2 Position bias in RetroTech data

How much position bias exists in the sessions in the RetroTech dataset? If we can quantify this, then we can consider how exactly to remedy this issue. Let's assess the bias quickly before we consider a new click model for overcoming these biases.

By looking across all sessions, across all queries, we can compute an average CTR per rank. This will tell us how much position bias exists in the RetroTech click data. We do this in Listing 11.5

Listing 11.5 Click-through rate per rank in search sessions across all queries

```
num_sessions = len(sessions['sess_id'].unique())    ❶
global_ctrs = sessions.groupby('rank')['clicked'].sum() / num_sessions    ❷
global_ctrs
```

❶ Total number of sessions

❷ Total clicks per position per session

Output:

```
rank
0.0     0.249727
1.0     0.142673
2.0     0.084218
3.0     0.063073
4.0     0.056255
5.0     0.042255
6.0     0.033236
7.0     0.038000
8.0     0.020964
9.0     0.017364
10.0    0.013982
```

You can see in Listing 11.5 that users click higher positions more. The CTR of results at `rank=0` is 0.25, followed by `rank=1` at 0.143 and so on.

Further, we can see position bias when we compare the CTR judgments from earlier to the typical ranking for each product in a query. If position bias is present, then our judgment's ideal ranking will end up resembling the typical ranking shown to users. We can analyze this by averaging the rank of each document over every session to see where they appear.

In Listing 11.6, we show the typical search results page for `transformers dark of the moon` sessions.

Listing 11.6 Examining rank of documents for `transformers dark of the moon`

```
QUERY='transformers dark of the moon'
query_sessions = sessions[sessions['query'] == QUERY]    ❶

avg_rank = query_sessions.groupby('doc_id')['rank'].mean()    ❷

avg_rank.sort_values(ascending=True)
```

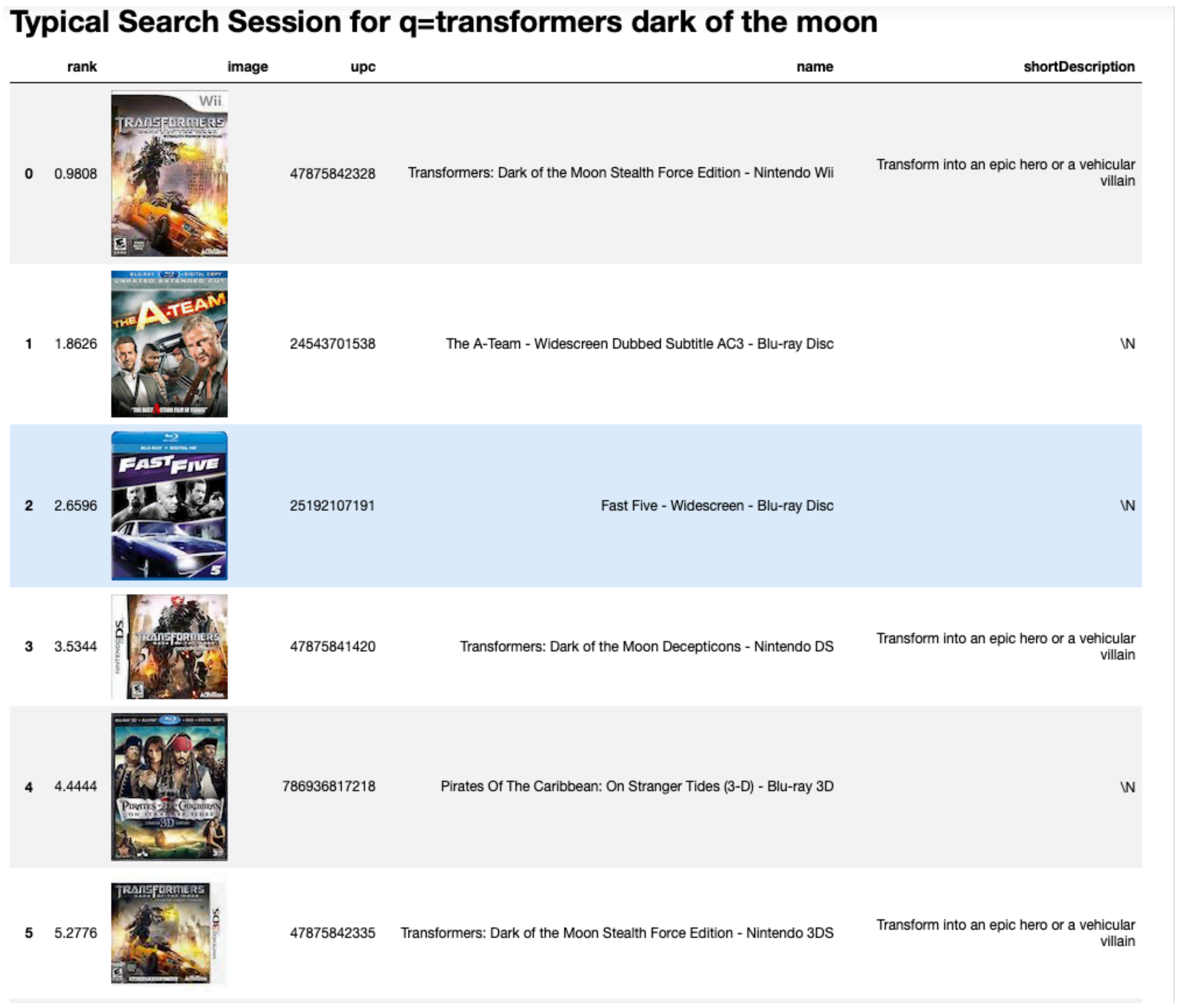
❶ Select sessions for `transformers dark of the moon`

❷ Take the average displayed position of each product in these sessions

Output:

```
doc_id           mean historical rank
47875842328      0.9808
24543701538      1.8626
25192107191      2.6596
47875841420      3.5344
786936817218     4.4444
47875842335      5.2776
47875841406      6.1378
97360810042      7.0130
24543750949      7.8626
36725235564      8.6854
47875841369      9.5796
97363560449     10.4304
93624956037     11.3298
97363532149     12.1494
400192926087    13.0526
```

In Listing 11.6, some documents, like `24543701538` and `47875842328` historically occur towards the top of the search results for this query. They will simply get clicked more due to position bias. The typical results page, shown in Figure 11.5, overlaps quite a lot with the CTR ideal results in Figure 11.2.



Typical Search Session for q=transformers dark of the moon

	rank	image	upc	name	shortDescription
0	0.9808		47875842328	Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii	Transform into an epic hero or a vehicular villain
1	1.8626		24543701538	The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc	\N
2	2.6596		25192107191	Fast Five - Widescreen - Blu-ray Disc	\N
3	3.5344		47875841420	Transformers: Dark of the Moon Decepticons - Nintendo DS	Transform into an epic hero or a vehicular villain
4	4.4444		786936817218	Pirates Of The Caribbean: On Stranger Tides (3-D) - Blu-ray 3D	\N
5	5.2776		47875842335	Transformers: Dark of the Moon Stealth Force Edition - Nintendo 3DS	Transform into an epic hero or a vehicular villain

Figure 11.4 Typical search result page for `transformers dark of the moon` query. Notice the overlap of irrelevant movies like "The A team" and "Fast Five". Also note the high ranking of the Wii game. These explain why the CTR click model seems to erroneously think these are relevant.

Unfortunately, CTR is primarily influenced by position bias. Users click on the odd movies in Figure 11.4 because the search engine returns them highly for this query, not because they are relevant. If we train an LTR model just on CTR, we would be asking the LTR model to optimize for what users already see and interact with.

We must account for position bias when automating LTR. Next let’s see how we could overcome position bias in a more robust click model that compensates for position bias.

11.2.3 A Click Model that Overcomes Position Bias: Simplified Dynamic Bayesian Network

You’ve seen the harm position bias can do in action! If we just use clicks directly, we will just train our LTR model to reinforce the ranking already shown to users. It’s time to introduce a click model that can overcome position bias. We’ll start by discussing a key concept to modeling position bias, that of an "examine". We’ll then introduce one particular click model that uses this "examine" concept to adjust raw clicks to overcome position bias.

HOW CLICK MODELS OVERCOME POSITION BIAS WITH AN "EXAMINE" EVENT

In our first attempt with click-through rate, we didn’t *really* think about how users scan search results. The user likely considered only a few results - biased by position - finally deciding to click one or two. If we can capture which results users consciously consider before clicking, we might be able to overcome position bias. Click models do exactly this by defining the concept of an *examine*. We’ll explore this concept before building a click model that overcomes position bias.

What is an *examine*? You may be familiar with *impressions* - a UI element rendered on the visible part of a user’s screen. In click models, we consider instead an *examine*, the probability a search result was actually consciously considered by the user. As we know, users often fail to notice something right in front of their eyes. You may have even been that user! Figure 11.6 captures this concept, contrasting impressions with examines.

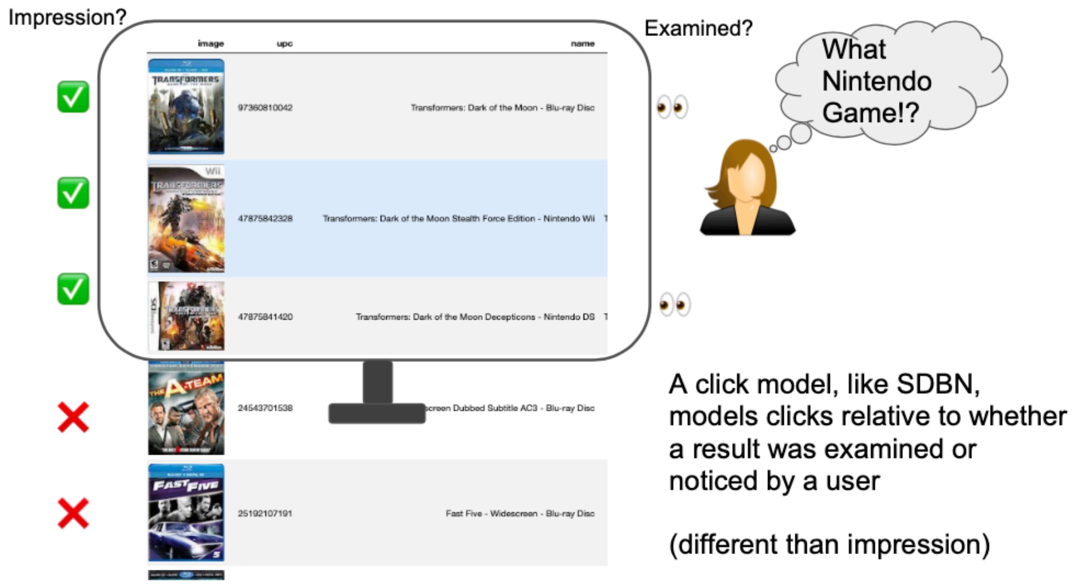

Impression?
image
upc
name
97360810042
Transformers: Dark of the Moon - Blu-ray Disc
47875842328
Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii
47875841420
Transformers: Dark of the Moon Decepticons - Nintendo DS
24543701538
creen Dubbed Subtitle AC3 - Blu-ray Disc
25192107191
Fast Five - Widescreen - Blu-ray Disc
Examined?
What Nintendo Game!?
A click model, like SDBN, models clicks relative to whether a result was examined or noticed by a user
(different than impression)

Figure 11.5 Impressions are whatever gets rendered in the viewport (the monitor shaped square) while examines are what the the user actually consider (the results with eyeballs adjacent). Modeling what users actually examine helps correctly account for how users interact with search results. Every click model does this differently.

You can see in Figure 11.5, the user fails to notice the Nintendo game in the 2nd position - even though it's being rendered on their monitor. If the user didn't examine it, a click model shouldn't penalize the Nintendo Game's relevance.

Why does tracking examines help overcome position bias? Examines are how a click model understands position bias. Another way of saying "position bias" is "we think whether users examine search results depends on the position." So modeling examines right is a core activity of most click models. Some click models, like the *Position-Based Model* (or PBM) attempt to determine an examine probability per position across all searches. Others, like the *Cascading Model*, or as we'll see soon, the *Dynamic Bayesian Network* models, assume that if a result was above the last click on the search page, it likely was examined.

For most click models, the top position usually has a higher examine probability than lower ones. This allows click models to adjust for clicks correctly. Items examined frequently and clicked are rewarded, and seen as more relevant. Those examined but not clicked are seen as less relevant.

To make this more concrete, let's dive deeper into one of those Dynamic Bayesian Network click models to see how it overcomes position bias.

DEFINING SIMPLIFIED DYNAMIC BAYESIAN NETWORK

It's time to get back to the algorithms! We discussed how a click model can overcome position bias by modeling the concept of an examine. Each click model defines this concept differently, and you'll next to see how one popular click model solves this problem. That will complete our position bias discussion and set us up to close the chapter understanding the final problem we'll discuss: confidence bias.

A *Simplified Dynamic Bayesian Network* (or *SDBN*) is a slightly less accurate version of the more complex *Dynamic Bayesian Network* click model (or *DBN*). These click models make the assumption that, within a search session, the probability a user examined a document depends heavily on whether it was positioned at or above the lowest clicked document.

SDBN's algorithm first marks the last click of each session, and then considers every document at or above this last click as examined. Finally, it computes a relevance grade by simply dividing the total clicks on a document by its total examines. We effectively get a kind-of dynamic CTR, tracking per session when users likely examined a result, and carefully using this to account for how users evaluated it's relevance.

Let's follow this algorithm step-by-step. We will first mark the last click of each session in Listing 11.7.

Listing 11.7 Mark which results were examined in each session

```
QUERY='dryer'
sdbn_sessions = sessions[sessions['query'] == QUERY].copy().set_index(
[CA]'sess_id')   ❶
last_click_per_session = sdbn_sessions.groupby(['clicked', 'sess_id'])[
[CA]'rank'].max()[True]   ❷
sdbn_sessions['last_click_rank'] = last_click_per_session   ❸
sdbn_sessions['examined'] = sdbn_sessions['rank'] <= sdbn_sessions[
[CA]'last_click_rank']   ❹

sdbn_sessions.loc[3]   ❺
```

❶ Select all sessions for query 'dryer'

❷ Compute last_click_per_session, the max rank where clicked is `True` per session

❸ Mark the last click rank in each session

❹ Set every position at or above last click to `True` (otherwise its `False`)

❺ Examine session 3 to observe which positions are considered examined

Output (truncated):

```
sess_id    query   rank    doc_id          clicked  last_click_rank    examined
3       dryer   0.0     12505451713     False   9.0             True
3       dryer   1.0     84691226727     False   9.0             True
3       dryer   2.0     883049066905    False   9.0             True
3       dryer   3.0     48231011396     False   9.0             True
3       dryer   4.0     74108056764     False   9.0             True
3       dryer   5.0     77283045400     False   9.0             True
3       dryer   6.0     783722274422    False   9.0             True
3       dryer   7.0     665331101927    False   9.0             True
3       dryer   8.0     14381196320     True    9.0             True
3       dryer   9.0     74108096487     True    9.0             True
3       dryer   10.0    74108007469     False   9.0             False
3       dryer   11.0    12505525766     False   9.0             False
3       dryer   12.0    48231011402     False   9.0             False
```

In Listing 11.7 we find the max rank where `clicked` is true by storing it in `last_click_per_session`. We then mark positions at or above `last_click_rank` as examined in our sessions for `dryer`, as you can see in the output for `sess_id=3`.

With every session updated with examines set to True/False, we now sum the total clicks & examines counts per document across all sessions in Listing 11.8.

Listing 11.8 Sum clicks and examines per doc_id for this query

```
sdbn = sdbn_sessions[sdbn_sessions['examined']].groupby('doc_id')[
[CA]['clicked', 'examined']].sum()   ❶
sdbn
```

❶ Total number of clicks and examines per product for this query

Output (truncated):

```
doc_id          clicked  examined
12505451713     355.0    2707.0
12505525766     268.0    974.0
12505527456     110.0    428.0
14381196320     217.0    1202.0
36172950027     97.0     971.0
36725561977     119.0    572.0
36725578241     130.0    477.0
48231011396     166.0    423.0
...
```

In Listing 11.8, `sdbn_sessions[sdbn_sessions['examined']]` filters only to examined rows. Then, per `doc_id`, we compute the total `clicked` and `examined` counts. You can see some results like `doc_id=36172950027` clearly were examined a lot with relatively few clicks from users.

Finally, we finish the SDBN algorithm in Listing 11.9 by computing clicks over examines.

Listing 11.9 Compute final SDBN grades

```
sdbn['grade'] = sdbn['clicked'] / sdbn['examined']   ❶

sdbn = sdbn.sort_values('grade', ascending=False)
```

❶ Compute `grade` as the proportion of clicks over examines

Output (truncated):

```
doc_id           clicked  examined     grade
856751002097     133.0    323.0        0.411765
48231011396      166.0    423.0        0.392435
84691226727      804.0    2541.0       0.316411
74108007469      208.0    708.0        0.293785
12505525766      268.0    974.0        0.275154
36725578241      130.0    477.0        0.272537
48231011402      213.0    818.0        0.260391
12505527456      110.0    428.0        0.257009
...
```

In the output of Listing 11.9, we see document `856751002097` is seen as most relevant, with a grade of `0.4118` or 133 clicks out of 323 examines.

Let's revisit our two queries to see how the ideal results now look for `dryer` and `transformers dark of the moon`. Figure 11.7 shows results for `dryer`, with Figure 11.8 showing `transformers dark of the moon`

SDBN judgments for dryer

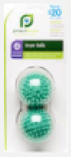
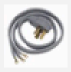
	grade	image	upc	name	shortDescription
0	0.411765		856751002097	Practecol - Dryer Balls (2-Pack)	Suitable for use on most dry cycles; reduces lint, static and wrinkles; improves heat circulation; 2-pack
1	0.392435		48231011396	LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White	ENERGY STAR QualifiedDigital controls; 7 cycles; SpeedWash cycle; 9 wash options; delay-wash; SenseClean system; 6Motion technology; TrueBalance antivibration system
2	0.316411		84691226727	GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White	Rotary electromechanical controls; 3 cycles; 3 heat selections; DuraDrum interior; Quiet-By-Design
3	0.293785		74108007469	Conair - 1875-Watt Folding Handle Hair Dryer - Blue	2 heat/speed settings; cool shot button; dual voltage; professional-length line cord
4	0.275154		12505525766	Smart Choice - 6' 30 Amp 3-Prong Dryer Cord	Heavy-duty PVC insulation; strain relief safety clamp
5	0.272537		36725578241	Samsung - 7.3 Cu. Ft. 7-Cycle Electric Dryer - White	Soft-touch dial controls; 7 preset drying cycles; 4 temperature settings; powdercoat drum; noise reduction package
6	0.260391		48231011402	LG - 7.1 Cu. Ft. 7-Cycle Electric Dryer - White	Electronic controls with LED display; 7 cycles; Dial-A-Cycle option; sensor dry system; 5 temperature levels; 5 drying levels; NeveRust drum; LoDecibel quiet operation

Figure 11.6 Ideal search results for query `dryer` according to SDBN. Notice how SDBN seems to zero in more on results that have more to do with washing clothes.

SDBN judgments for transformers dark of the moon

	grade	image	upc	name	shortDescription
0	0.641745		97360810042	Transformers: Dark of the Moon - Blu-ray Disc	\N
1	0.480620		400192926087	Transformers: Dark of the Moon - Original Soundtrack - CD	\N
2	0.395062		97363560449	Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD	\N
3	0.323077		97363532149	Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD	\N
4	0.266234		93624956037	Transformers: Dark of the Moon - Original Soundtrack - CD	\N
5	0.239713		47875842328	Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii	Transform into an epic hero or a vehicular villain

Figure 11.7 Ideal search results for query `transformers dark of the moon` according to SDBN. We've now surfaced the DVD, blu ray movie, and CD soundtrack.

Subjectively examining Figure 11.7 and 11.8, both sets of judgments appear more intuitive than the CTR judgments. In our `dryer` example, the emphasis appears to be on washing clothes. There's some accessories (such as the dryer balls) that score roughly the same as the dryers themselves.

For `transformers dark of the moon`, we note the very high grade for the blu-ray movie. We also see the DVD and CD soundtrack ranking higher than other secondary "Dark Of The Moon" items such as video games. Somewhat oddly, the soundtrack CD is ranked higher than the movie DVD, perhaps we should look into this more.

Of course, as we've said earlier, we're using our subjective sense for now. In Chapter 12, we'll think more objectively about how we might evaluate judgment quality.

Nevertheless, with position bias more under control, we'll move on to fine tune our judgments to handle another crucial bias you'll need to overcome when using a click model to automate LTR: confidence bias.

11.3 Handling Confidence Bias: not upending your model from a few lucky clicks

In the game of baseball, a player’s batting average tells us the proportion of hits they get for every at bat. A great professional player has a batting average > 0.3. Consider, however, a lucky little league baseball player stepping up to the plate for their first at-bat, getting a hit. Their batting average is technically 1.0! We can conclude, then, that this young child is a baseball prodigy, and will certainly have a great baseball career. Right?

Not quite! In this section, we’re going to explore the relevance side of this lucky little leaguer. What do we do with results that, perhaps simply out of luck, have been examined only a few times, each resuting in a click? These should get a grade of 1.0, right? Indeed, we’ll see this problem in our data! With this problem definition in place, we can then work on solving the issue.

11.3.1 The Low Confidence Problem in RetroTech Click Data

Let’s look at the data to see where low confidence data points are biasing the training data. Then we’ll see how we can compensate for low confidence issues in the SDBN dataset. To define the problem, let’s take time to look at SDBN results for `transformers dark of the moon` and another, rarer query to see common low-confidence situations.

If you recall, it was a bit suspicious that the soundtrack CD for the "Transformers Dark of The Moon" ranked so highly according to SDBN. When we examine the raw data underlying the rankings, we can see a possible problem. In Listing 11.10, we reconstruct the SDBN data for `transformers dark of the moon` to debug this issue, combining Listings 11.7-11.9 into a single code snippet.

Listing 11.10 Recomputing SDBN statistics for `transformers dark of the moon`

```
QUERY='transformers dark of the moon'
sdbn_sessions = sessions[sessions['query'] == QUERY].copy().set_index(
[CA]'sess_id')   ❶

last_click_per_session = sdbn_sessions.groupby(['clicked', 'sess_id'])
[CA]['rank'].max()[True]   ❷
sdbn_sessions['last_click_rank'] = last_click_per_session   ❷
sdbn_sessions['examined'] = sdbn_sessions['rank'] <= sdbn_sessions[
[CA]'last_click_rank']   ❷
sdbn = sdbn_sessions[sdbn_sessions['examined']].groupby('doc_id')[
[CA]['clicked', 'examined']].sum()   ❷
sdbn['grade'] = sdbn['clicked'] / sdbn['examined']   ❷

sdbn = sdbn.sort_values('grade', ascending=False)
sdbn
```

❶ Select `transformers dark of the moon` search sessions

❷ Recompute SDBN clicks, examines, and grades as per Listings 11.7-11.9

Output (truncated):

```
doc_id          clicked examined    grade
97360810042     412.0   642.0       0.641745
400192926087    62.0    129.0       0.480620
97363560449     96.0    243.0       0.395062
97363532149     42.0    130.0       0.323077
93624956037     41.0    154.0       0.266234
47875842328     367.0   1531.0      0.239713
...
```

From the output of Listing 11.10, note the top result, the Blu-ray movie (`doc_id=97360810042`), has far more examines (642) than the soundtrack CD (`doc_id=400192926087` with 129 examines). The Blu-ray's grade is more reliable, given it has had more opportunities to receive clicks. It's unlikely to be dominated by noisy, spurious clicks. The CD, on the other hand, has far fewer examines. Shouldn't the Blu-ray's relevance grade be weighed higher, given it's a more reliable data point compared to the CD with more limited data?

Typically this situation is even starker. Consider the query `blue ray`. You'll note this is a common misspelling of `blu-ray`. As a common mistake, it likely mixes documents with a modest number of examines with documents receiving very few.

In Listing 11.11 we compute the SDBN statistics for `blue ray`.

Listing 11.11 SDBN judgments for a typical torso or long-tail query with little click data to build judgments from.

```
QUERY='blue ray'

sdbn_sessions = sessions[sessions['query'] == QUERY]
sdbn_sessions = sdbn_sessions[sdbn_sessions['sess_id'] < 50050]   ❶
sdbn_sessions = sdbn_sessions.set_index('sess_id')

last_click_per_session = sdbn_sessions.groupby(['clicked', 'sess_id'])[
[CA]'rank'].max()[True]   ❷
sdbn_sessions['last_click_rank'] = last_click_per_session   ❷
sdbn_sessions['examined'] = sdbn_sessions['rank'] <=
[CA]sdbn_sessions['last_click_rank']   ❷
sdbn = sdbn_sessions[sdbn_sessions['examined']].groupby('doc_id')[
[CA]['clicked', 'examined']].sum()   ❷
sdbn['grade'] = sdbn['clicked'] / sdbn['examined']   ❷

sdbn = sdbn.sort_values('grade', ascending=False)
sdbn
```

❶ Randomly sample a few sessions to simulate a typical long-tail case

❷ Recompute SDBN clicks, examines, and grades as per Listings 11.7-11.9

Output (truncated):

```
doc_id          clicked examined    grade
600603132872    1.0        1.0         1.000000
827396513927    14.0    34.0        0.411765
25192073007     8.0     20.0        0.400000
885170033412    6.0     19.0        0.315789
600603141003    8.0     26.0        0.307692
24543672067     8.0     27.0        0.296296
813774010904    2.0     7.0         0.285714
...
```

Looking at the output of Listing 11.11 we see something unsettling. Like the most extreme case of our lucky little league baseball player, the most relevant result, doc `600603132872`, receives a grade of 1 (perfectly relevant) after being examined by only one user! This grade of 1 trumps the next result, which has a grade of 0.411 based on 34 examines. When you consider doc `600603132872` is a set of Blu-ay cases and `827396513927` is a Blu-ray player, this feels more troubling. Our subjective interpretation might rank the player above the cases. Shouldn't the fact that the second result was examined much more count for something?

What we've seen in these examples is *confidence bias* - when a judgment list has many grades based on statistically insignificant, spurious events. We say these spurious events with few examines have low *confidence*, whereas those with more examines provide a higher level of confidence. No matter your click model, you're most likely not working at Google where user search click traffic abounds. You likely have many situations where queries have only a modest amount of traffic. To automate LTR, you'll need to adjust your training data generation to account for the confidence you have in the data.

Now that you've seen the impact of low confidence data, we can move onto some solutions to apply when building your click model.

11.3.2 Using a Beta Prior to Model Confidence Probabilistically

We've just seen a few issues created by valuing low confidence data too highly. Without adjusting your models based on your confidence in the data, you won't be able to build a reliable Automated LTR system. We could just filter these low-confidence examples out, but can we perhaps do something smarter? We'll discuss an approach to preserving all the click-stream data in this section as we introduce the concept of beta distributions. But first, let's discuss why using all data is preferred over simply filtering out the low-confidence examples.

SHOULD WE FILTER OUT LOW CONFIDENCE JUDGMENTS?

In our click model, should we just remove the low-confidence examples? Let's consider why we don't advocate for that, before taking a step back to a more mature solution. We can then finish off strong with a confidence-adjusted SDBN that helps us maximally leverage all of our training data.

Filtering training data, such as below some minimum threshold of examines, reduces the amount of training data you have. Even with a reasonable threshold, documents for a query are typically examined on a power law distribution. Users examine some documents very frequently, while examining the vast majority very infrequently. A threshold can thus remove too many good LTR examples, and cause an LTR model to miss important patterns (remember you're probably not Google!). Even with a threshold, you would be left with the challenge of how to weigh medium-confidence examples against high confidence ones, such as the `transformers dark of the moon` query from earlier.

Instead of a hard cutoff, we advocate for weighing examples based on our confidence in the data. We will do this using a beta distribution prior on the computed relevance grades, we'll then zoom out to apply this solution to fix our SDBN click model judgments.

USING THE BETA DISTRIBUTION TO ADJUST FOR CONFIDENCE

Beta distributions help us draw conclusions from our clicks and examines based on probabilities instead of just biased occurences. However, before we dive straight into using the beta distribution for judgments, let's first examine the usefulness of a beta distribution using a fun and more intuitive example: baseball statistics.

In baseball, a batting average of 0.295 for a player means that when this player goes to bat, there's roughly a 29.5% chance they will get a hit. But if we wanted to know "What's the batting average for that player batting in Fenway Park in September on rainy days", we'd probably have very little information to go on. The player may have only batted in those conditions a handful of times. Maybe they made 2 hits out of 3 tries in those conditions. So we would conclude their batting average in these cases is `2/3` or 0.67. We know by now this conclusion would be a mistake: Do we really think, based on only 3 chances to bat, we can conclude the player has an improbably high 66.7% chance of making a hit? A better way would be to use the 0.295 general batting average as an initial belief, moving slowly away from that assumption as we gradually gain more data on "Fenway Park in September on rainy days" at bats.

The *beta distribution* is a tool used to manage beliefs. It turns a probability, like batting average, or judgment grade, into two values, `a` and `b` that represent that probability as a distribution. The `a` and `b` values can be interpreted as:

- `a`, the successes: the number of at bats with hits we've observed, the number of examines with clicks
- `b`, the failures: the number of at bats without hits we've observed, the number of examines without clicks

With the beta distribution, the property `mean = a / (a+b)` holds, where `mean` is the initial point value like a batting average. Given a `mean`, notice we can find many `a` and `b` values that satisfy `mean = a / (a+b)`. After all `0.295 = 295 / (295 + 705)` as does `0.295 = 1475 / (1475 + 3525)` and so on. Yet each represents a different beta distribution. Keep this property in mind as we move along.

Let's put these pieces together to see how beta distribution prevents us from jumping to conclusions on spurious click (or batting) data.

We could declare our initial belief about any document's relevance grade as `0.125`. This is similar to declaring the baseball player's batting average to be `0.295` as our initial belief of their performance. We can use the beta distribution to update the initial belief for specific cases like "Fenway Park in September on rainy days" or a specific document's relevance for a search query.

The first step is to pick an `a` and `b` that captures our initial belief. For our relevance case, we could choose many a's and b's that satisfy `0.125=a/(a+b).`. Perhaps we choose `a=2.5,b=17.5` as our relevance belief on documents with no clicks, Graphing this, we would see the distribution in Figure 11.9.

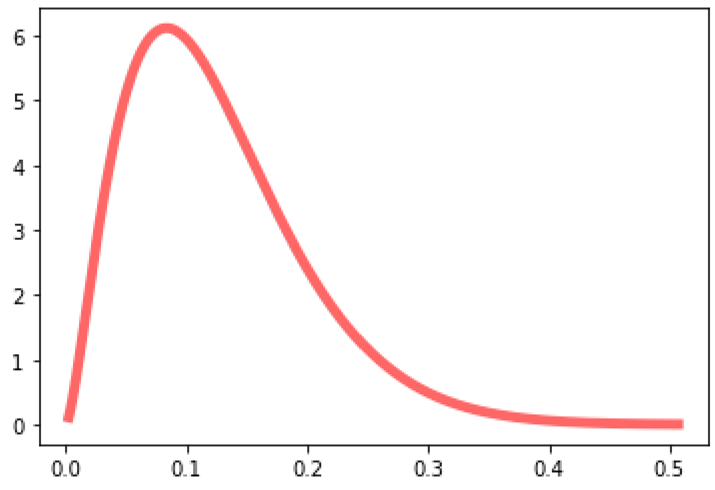


Figure 11.8 Beta distribution for a relevance grade of 0.125. The mean corresponds to our default relevance grade. We see the distribution of most likely relevance grades.

We can observe now what happens when we observe a document's first click, incrementing that document's a to 3.5. In Figure 11.10 we have `a=3.5,b=17.5`.

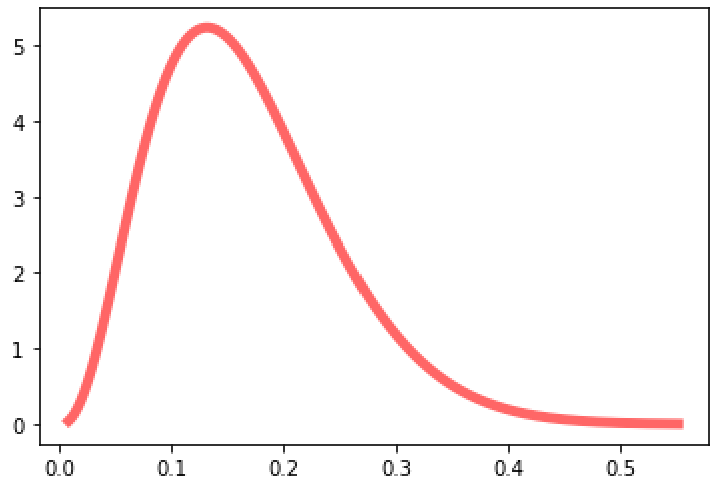


Figure 11.9 Beta distribution for a relevance grade after adding one click, now grade is 0.1666. Updating with a click 'pulls' the probability distribution a little one direction, updating the initial belief.

The new mean relevance grade for the updated distribution is now `3.5/(17.5+3.5)` or 0.16666, effectively "pulling" the initial belief a little higher given its first click. Without the beta distribution, this document would have 1 click and 1 examine, resulting in a grade of 1.

We refer to the 'starting point' probability distribution (the chosen `a` and `b`) as the *prior distribution* or just a *prior*. This is our initial belief in what we think will happen. The distribution after updating `a` and `b` for a specific case, like a document, is a *posterior distribution* or just *posterior*. This is our updated belief.

Recall that we said earlier that many initial `a` and `b` values could be chosen. This has significance as the magnitude of the initial `a` and `b` make our prior weaker or stronger. We could choose any value for `a` and `b` where `a / (a+b) = 0.125`. But note what happens if we choose a very small value `a=0.25,b=1.75`. Then we go to update `a` by incrementing it by 1. The new expected value of the posterior distribution is `1.25 / (1.25+1.75)` or ~`0.416`. That's a major impact for just one click. Conversely, using very high `a` and `b` would make a prior so strong it would barely budge. So when we use the beta distribution, you'll want to tune the magnitude of the prior so updates have the desired effect.

Now that you've seen this handy tool in practice for capturing SDBN grades, lets see the beta distribution help with our SDBN confidence problems.

USING A BETA PRIOR IN SDBN CLICK MODELS

Let's finish the chapter by updating the SDBN click model using the beta distribution. Of course, if you use another click model, like the ones alluded to earlier in this chapter, you'll need to reflect on how confidence can be solved in those cases. The beta distribution might be a useful tool there, as well.

If you recall, the output of SDBN was a count of `clicks` and `examines` for each document. In Listing 11.12 we pick up from Listing 11.11 which computed SDBN for the `blue ray` query. We will choose a prior grade of `0.3` for use with our SDBN model. This is our default grade when we don't have information about the document - possibly derived from the typical grade we see in our judgments. We then compute a prior beta distribution (`prior_a` and `prior_b`) using this prior grade.

Listing 11.12 Compute prior beta distribution using an initial, default grade of 0.3

```
PRIOR_GRADE=0.3   ❶
PRIOR_WEIGHT=100   ❷
sdbn['prior_a'] = PRIOR_GRADE*PRIOR_WEIGHT   ❸
sdbn['prior_b'] = (1-PRIOR_GRADE)*PRIOR_WEIGHT   ❸
```

❶ Default, prior relevance grade

❷ How much confidence to place in the prior (weight = a + b)

❸ Resulting a and b satisfying prior grade = prior_a / (prior_a + prior_b)

Output (truncated):

```
doc_id          clicked     examined   grade       prior_a prior_b
600603132872    1.0      1.0        1.000000    30.0    70.0
827396513927    14.0     34.0       0.411765    30.0    70.0
25192073007     8.0      20.0       0.400000    30.0    70.0
885170033412    6.0      19.0       0.315789    30.0    70.0
...
```

In Listing 11.12, with a weight of 100, you can confirm that `PRIOR_GRADE = prior_a / (prior_a + prior_b)` as `30 / (30+70)` is 0.3. So this has captured an initial probability distribution for our prior.

Next up, in Listing 11.13, we need to compute a posterior distribution and corresponding relevance grade. We do this by incrementing `prior_a` for clicks (our 'successes'), and `prior_b` for examines with no clicks (our 'failures'). Finally we compute an updated grade in `beta_grade`
.

Listing 11.13 Compute a posterior beta distribution for relevance grade

```
sdbn['posterior_a'] = sdbn['prior_a'] +  sdbn['clicked']   ❶
sdbn['posterior_b'] = sdbn['prior_b'] + (sdbn['examined'] -
[CA]sdbn['clicked'])   ❷

sdbn['beta_grade'] = sdbn['posterior_a'] / (sdbn['posterior_a'] +
[CA]sdbn['posterior_b'])   ❸

sdbn.sort_values('beta_grade', ascending=False)
```

❶ Update our belief about the documents relevance by incrementing a by number of clicks

❷ Update our belief about the documents lack-of-relevance by incrementing b examines without clicks

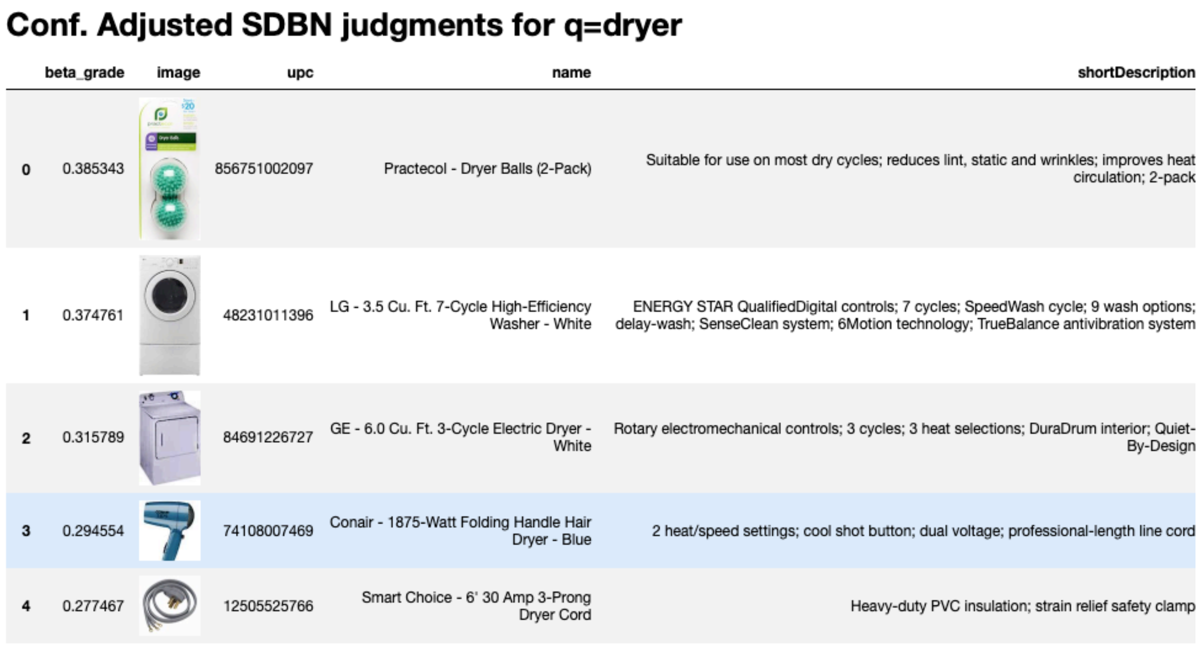
❸ Compute a new grade from posterior a and b

Output (truncated, `prior_a` and `prior_b` omitted, see above):

```
doc_id          clicked examined    grade      ... posterior_a posterior_b
[CA]beta_grade
827396513927    14.0    34.0        0.411765   ... 44.0        90.0
[CA]0.328358
25192073007     8.0     20.0        0.400000   ... 38.0        82.0
[CA]0.316667
600603132872    1.0     1.0         1.000000   ... 31.0        70.0
[CA]0.306931
885170033412    6.0     19.0        0.315789   ... 36.0        83.0
[CA]0.302521
600603141003    8.0     26.0        0.307692   ... 38.0        88.0
[CA]0.301587
24543672067     8.0     27.0        0.296296   ... 38.0        89.0
[CA]0.299213
813774010904    2.0     7.0         0.285714   ... 32.0        75.0
[CA]0.299065
...
```

In Listing 11.13's output notice our new ideal results for "blue ray" by sorting on `beta_grade`. `beta_grade` clusters closer the prior grade of 0.3. Notably our Blu Ray cases have slid to the 3rd most relevant slot - the single click not pushing the grade much past 0.3.

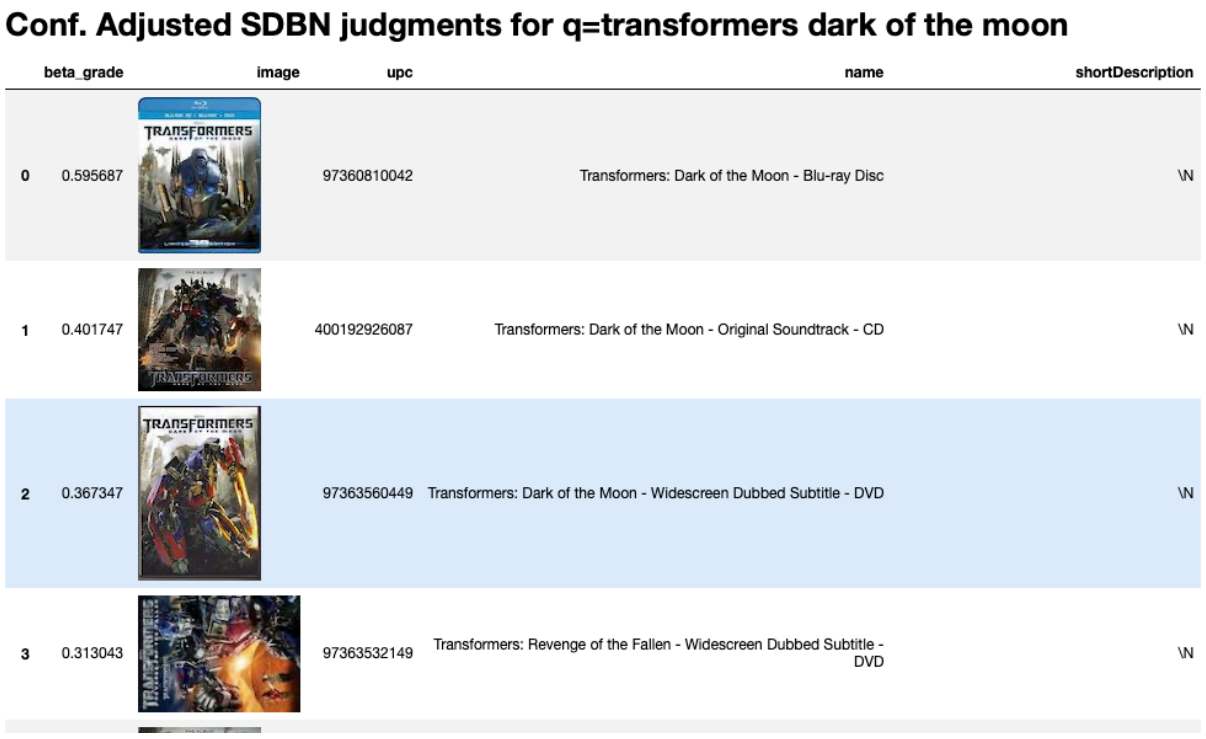
When we repeat this calculation judgments for `dryer` and `transformers dark of the moon` in Figures 11.11 and 11.12, we note the order is the same, however the grades themselves stay closer to the prior of 0.3 depending on our confidence in the data.



Conf. Adjusted SDBN judgments for q=dryer

	beta_grade	image	upc	name	shortDescription
0	0.385343		856751002097	Practecol - Dryer Balls (2-Pack)	Suitable for use on most dry cycles; reduces lint, static and wrinkles; improves heat circulation; 2-pack
1	0.374761		48231011396	LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White	ENERGY STAR QualifiedDigital controls; 7 cycles; SpeedWash cycle; 9 wash options; delay-wash; SenseClean system; 6Motion technology; TrueBalance antivibration system
2	0.315789		84691226727	GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White	Rotary electromechanical controls; 3 cycles; 3 heat selections; DuraDrum interior; Quiet-By-Design
3	0.294554		74108007469	Conair - 1875-Watt Folding Handle Hair Dryer - Blue	2 heat/speed settings; cool shot button; dual voltage; professional-length line cord
4	0.277467		12505525766	Smart Choice - 6' 30 Amp 3-Prong Dryer Cord	Heavy-duty PVC insulation; strain relief safety clamp

Figure 11.10 Beta-adjusted SDBN ideal results for `dryer`. Notice the grades now are more tighly focused around the prior grade 0.3, with some above or below this prior.



Conf. Adjusted SDBN judgments for q=transformers dark of the moon

	beta_grade	image	upc	name	shortDescription
0	0.595687		97360810042	Transformers: Dark of the Moon - Blu-ray Disc	\N
1	0.401747		400192926087	Transformers: Dark of the Moon - Original Soundtrack - CD	\N
2	0.367347		97363560449	Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD	\N
3	0.313043		97363532149	Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD	\N

Figure 11.11 Beta-adjusted SDBN ideal results for `transformers dark of the moon`. Earlier we noted the soundtrack seemed oddly high in its relevance grade despite fewer clicks than the blu ray movie. Here we see the soundtrack's relevance closer to the prior of 0.3.

Examining Figure 11.12 notably shows less confidence in the soundtrack when compared to the SDBN judgments without modeling confidence (Figure 11.10). The grade has dropped from 0.48 to 0.4. Notably, the DVD grade following the CD has not changed much, only changing from

0.39 to 0.36 given our higher confidence in that observation.

Most of your queries won’t be like `dryer` or `transformers dark of the moon`. They’ll be more like `blue ray`. To meaningfully work with these queries in LTR, you’ll need to be able to handle these "small data" problems like lower confidence.

We are beginning to have a more reasonable training set for automating LTR, but there’s still work to do. In the next chapter, we will move to look at the complete search feedback loop. This includes working on presentation bias. Recall this is the bias where users never examine what search never returns to them! How can we add surveilance to the automated LTR feedback loop to both overcome presentation bias and ensure our model - and by extension the judgments - are working as expected? But before we examine those topics in the next chapter, let’s revisit training an LTR model, and invite you to experiment with what you’ve learned so far.

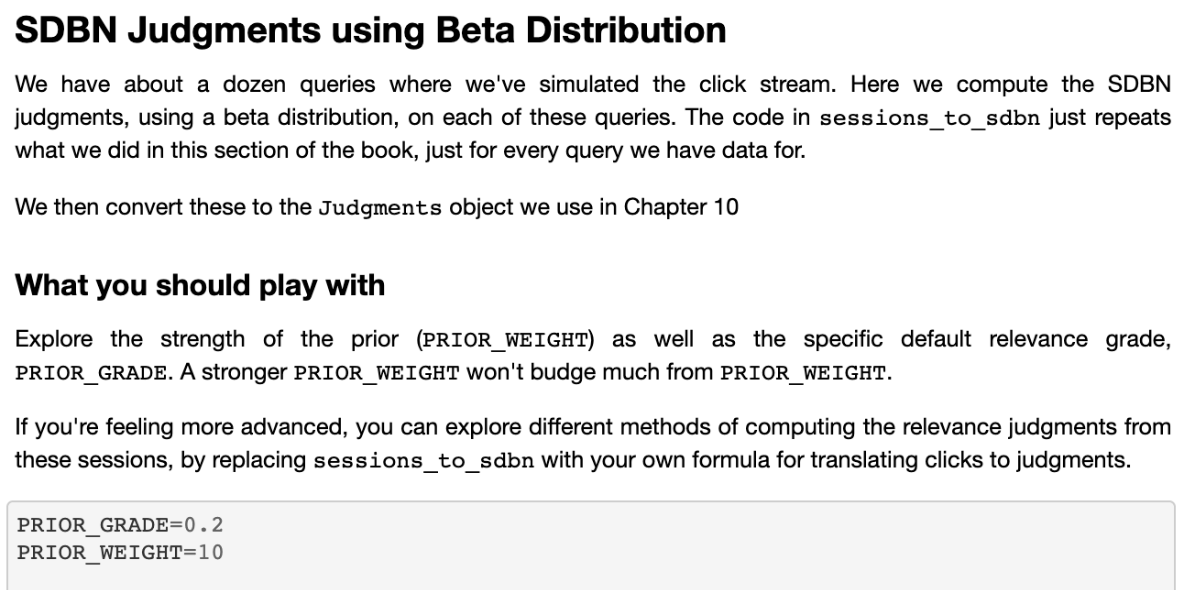
11.4 Exploring your training data in an LTR System

Great work! You’ve made it through both Chapter 10 and 11. You now have what you need to develop reasonable LTR training data and train an LTR model. You’re likely to eager to train a model from your work. Instead of repeating the extensive code from chapter 10 here, we’ve created an listing x.y notebook to allow you to experiment with LTR on the RetroTech data.

In this notebook you can fine-tune the inner LTR engine - the feature engineering and model creation that attempts to satisfy the training data. You can also explore the implications of altering the automated inputs to this engine: the training data itself. All together, this notebook has every step you’ve learned about so far:

1. Processing raw click session data into judgments, using the SDBN click model and a beta prior.
2. Transforming the dataframe into the `Judgments` we used in Chapter 10.
3. Loading a selection of LTR features to use with the Solr LTR plugin.
4. Logging these features from Solr, then performing pair-wise transformation of the data into a suitable training set
5. Training and uploading a model.
6. Searching!

In the Markdown you’ll see "What you should play with" as an invitation to tune, like in Figure 11.13:



SDBN Judgments using Beta Distribution

We have about a dozen queries where we've simulated the click stream. Here we compute the SDBN judgments, using a beta distribution, on each of these queries. The code in `sessions_to_sdbn` just repeats what we did in this section of the book, just for every query we have data for.

We then convert these to the `Judgments` object we use in Chapter 10

What you should play with

Explore the strength of the prior (`PRIOR_WEIGHT`) as well as the specific default relevance grade, `PRIOR_GRADE`. A stronger `PRIOR_WEIGHT` won't budge much from `PRIOR_WEIGHT`.

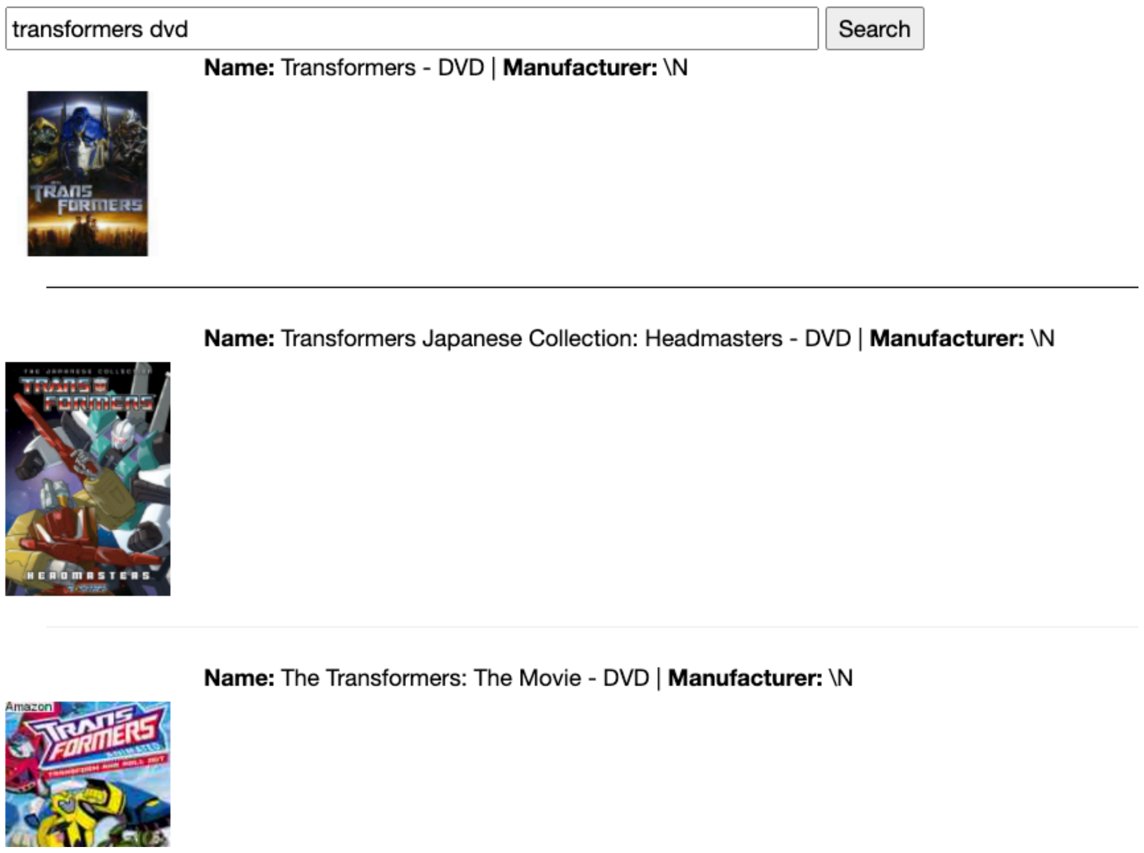
If you're feeling more advanced, you can explore different methods of computing the relevance judgments from these sessions, by replacing `sessions_to_sdbn` with your own formula for translating clicks to judgments.

```
PRIOR_GRADE=0.2
PRIOR_WEIGHT=10
```

Figure 11.12 Notebook exploring the full LTR system. You can take the model for a test drive.

We invite you to tune click-model parameters, think of new features, and different ways of arriving at the final LTR model, discovering which ones seem to yield the best results. While you do this tuning, please be sure to question your own subjective assumptions compared to what the data is showing you!

With out-of-the box tuning, we leave you with Figure 11.14, showing the current search results for the query `transformers dvd`. Try different queries here. How can you help the model better discriminate between relevant and irrelevant documents? Are the issues you encounter due to the training data used? Or is it the features used to construct the model?


transformers dvd

Search

Name: Transformers - DVD | **Manufacturer:** \N

Name: Transformers Japanese Collection: Headmasters - DVD | **Manufacturer:** \N

Name: The Transformers: The Movie - DVD | **Manufacturer:** \N

Figure 11.13 Trained model ranks `transformers dvd`. Do you think you could improve this?

In the next chapter, we’ll finish fleshing out the automated LTR system by performing surveilance on the model. Most crucially, we’ll consider how to overcome *presentation bias*. Even with the adjustments in this chapter, users will still only ever have a chance to act on what the search shows them. So we still have a feedback loop biased heavily by the current relevance ranking. How can we look out for this problem and overcome it? In the next chapter we’ll consider these issues as our LTR model interacts with real life users.

11.5 Summary

- We can automate Learning to Rank (LTR) if we can reliably transform user click data into relevance judgments using a *click model*. However, the click model itself is something that must be evaluated using A/B testing or other forms of testing to ensure the reliability of the Automated LTR system.
- Learned (implicit) relevance judgements lists can be plugged into existing learning to rank training processes to either replace or augment manually-created judgements.
- Raw clicks are usually problematic in automated LTR models due to common biases in how algorithms rank and present search results to users
- Among the visible search results, *position bias* says users prefer results ranked near the top. We overcome position bias by using a click model that tracks the probability a user has examined a document or a position in the search results.
- Most search applications have a lot of spurious click data. When training data is biased towards these spurious results, we have *confidence bias*. We can overcome confidence bias by using the beta distribution to create a prior that we update gradually with new observations as they come in.

12

Overcoming bias in learned relevance models

This chapter covers

- **Using live users to get feedback on our LTR model**
- **A/B testing search relevance solutions with live users**
- **Exploring possible relevant results beyond the top results we always show users**
- **Balancing exploiting what we've learned from historical data and exploring what might be relevant**

So far our Learning to Rank work has happened in the lab. In previous chapters, we built models using automatically constructed training data from user clicks. In this chapter, we'll take our model into the real world for a test drive with (simulated) live users!

Recall that we compared the full Automated Learning to Rank system to a self-driving car. Internally, the car has an engine: the end-to-end model retraining on historical judgements as discussed in Chapter 10. In Chapter 11 we compared our model's training data to self driving car directions: what *should* we optimize to automatically learn judgements based on previous interactions with search results? We built training data, and overcame key biases inherent in click data.

In this chapter, we leave the lab! We monitor our model in the real world. We see where the model does well and understand whether the work in the previous two chapters failed or succeeded. This means exploring a new kind of testing to validate our model: *A/B testing*. In *A/B testing* we randomly assign live users to different models and examine business outcomes (like sales, etc.), to see which performs best. You might be familiar with A/B testing in other contexts, but here we'll zero-in on the implications for an Automated LTR system.

Live users help us not just validate, they also aid in escaping dangerous negative feedback loops our models can find themselves in, as shown in Figure 12.1.

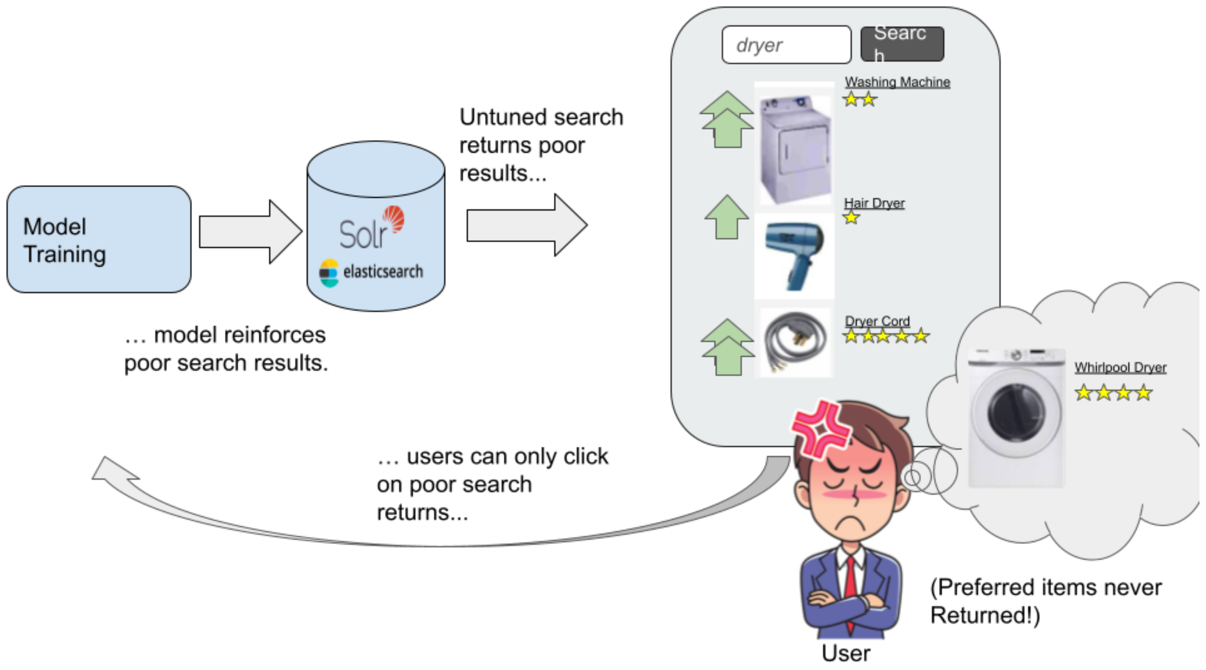

Model Training
Solr
elasticsearch
Untuned search returns poor results...
dryer
Search
Washing Machine
Hair Dryer
Dryer Cord
Whirlpool Dryer
… model reinforces poor search results.
… users can only click on poor search returns...
(Preferred items never Returned!)
User

Figure 12.1 Presentation bias's negative feedback loop. Users never click on what search never returns, thus relevance models can never grow beyond the current model's knowledge.

In Figure 12.1, our model can only learn what's relevant *within results shown to the user*! In other words our model will often parrot back what users see. We have an unfortunate chicken-and-egg problem. Good LTR attempts to optimize for the most clicked results, but users will only click on what's right in front of them. How could LTR possibly get better when training data seems hopelessly biased towards search's current ranking? This bias in training data, to parrot back the current, displayed results, is called *presentation bias*.

After we explore A/B testing, we'll fight presentation bias for the rest of the chapter. Much like a self-driving car that has only learned one suboptimal path, we'll have to strategically explore alternate, promising paths - in our case new types of search results - to learn new patterns for what's relevant to users. If Figure 12.2 we see the Automated LTR loop augmented with exploration.

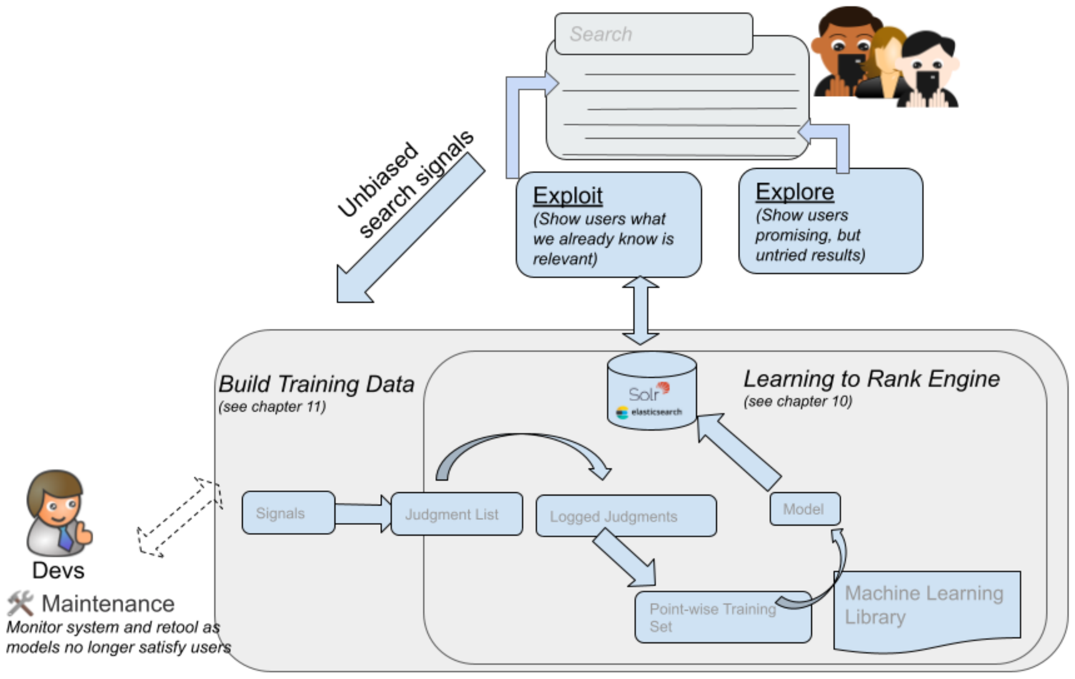


Figure 12.2 Automated LTR meets live users. To be useful, our Automated LTR system must overcome presentation bias by exploring yet to be seen results with users to expand training data coverage.

Before we get to this all important subject, we must first wrap everything we learned in chapters 10 and 11 into a few lines of code. Then we'll be able to iterate quickly, exploring A/B testing and overcoming presentation bias.

12.1 Our Automated LTR engine in a few lines of code

Before we begin to A/B test, we'll gather all our knowledge from chapters 10 and 11 into a small handful of Python helper functions. First we'll define a function to let us rebuild training data from raw session clicks using an SDBN click model (all of chapter 11). Next we'll create an equally simple snippet of code to train a model with that training data (all of chapter 10). We'll very quickly sum up these functions before diving into A/B testing and overcoming presentation bias in the rest of the chapter.

12.1.1 Turning clicks into training data (Chapter 11 in one line of code)

In Chapter 11 we turned clicks into training data. In this section, we briefly revisit that code to create a convenient helper function.

Recall in Chapter 11 We overcame biases in how users click on search results. You learned about the SDBN (Simplified Dynamic Bayesian Network) click model. Now, we simply wrap that code up into a convenience function in this section to regenerate training data as needed.

As a reminder, our click model turns raw clicks into training labels or *grades* mapping how relevant a document is for a keyword. The raw input we need to build the training data includes a query string, the rank of the result as displayed, the document in that position, and whether it was clicked. We can see that data stored in this dataframe:

```
    sess_id    query        rank    doc_id            clicked
0    50002    blue ray    0.0       600603141003    True
1    50002    blue ray    1.0       827396513927    False
2    50002    blue ray    2.0       24543672067        False
3    50002    blue ray    3.0       719192580374    False
4    50002    blue ray    4.0       885170033412    True
```

Given this input, we can wrap all of Chapter 11 into a function that computes our training data. Recall we use the term *judgment list* or *judgments* to refer to our training data. We see our judgments computation in Listing 12.1.

Listing 12.1 Turn sessions into training data (chapter 11 in one line!)

```
sdbn = sessions_to_sdbn(sessions,
                        prior_weight=10,    ❶
                        prior_grade=0.2)    ❷
sdbn
```

❶ How strong the prior should be (see confidence bias chapter 11)

❷ The default probability of a results relevance when we have no evidence

Output (truncated):

```
                                        clicked    examined    grade     beta_grade
query                 doc_id
blue ray            27242815414            42.0    42.0    1.000000    0.846154
                    600603132872        46.0    88.0    0.522727    0.489796
                    827396513927        1304.0    3381.0    0.385685    0.385137
                    600603141003        978.0    2620.0    0.373282    0.372624
                    885170033412        568.0    2184.0    0.260073    0.259799

...    ...    ...    ...    ...    ...
transformers dvd    47875819733            24.0    1679.0    0.014294    0.015394
                    708056579739        23.0    1659.0    0.013864    0.014979
                    879862003524        23.0    1685.0    0.013650    0.014749
                    93624974918            19.0    1653.0    0.011494    0.012628
                    47875839090            16.0    1669.0    0.009587    0.010721
```

Let’s briefly revisit what we learned in Chapter 11 by looking at the output of Listing 12.1. As you can see in the output, we compute a dataframe where each query-document pair has corresponding `clicked` and `examined` counts. Clicks is what it sounds like: the sum of raw clicks this product received for this query. Recall that `examined` corresponds to number of times the click model thinks the user noticed the result.

The statistics `grade` and `beta_grade` are the training labels. These correspond to the probability a document is relevant for the query. Recall that `grade` simply divides `clicked` by `examined`: the naive, first implementation of the SDBN click model. However, we learned in Chapter 11 that it would be better to account for how much information we have (see section 11.3). We don’t want one click with one examine (1/1 = 1.0) to be counted as strongly as a hundred clicks with a hundred examines (100/100 = 1.0). For this reason `beta_grade` places a higher weight on results with more information (preferring the hundred clicks example). We'll therefore use `beta_grade` as opposed to `grade` when retraining LTR models.

This data serves as training data to the models we trained in Chapter 10. Let's next see how we can easily take this training data, train a model, and deploy it to users.

12.1.2 Model training & evaluation in a few function calls

In addition to regenerating training data, we also need to retrain our model before deploying to live users. In this section, we'll explore the convenience functions for our core LTR model training engine. This sets us up to quickly experiment with models through the rest of this chapter.

We wrap model training and offline evaluation in a few simple lines in Listing 12.2.

Listing 12.2 Train and evaluate the model on a few features

```
random.seed(1234)   ❶

feature_set = [   ❷
    {
      "name" : "long_description_bm25",
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "longDescription:(${keywords})"
      }
    },
    {
      "name" : "short_description_constant",
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "shortDescription:(${keywords})^=1"
      }
    }
]

train, test = test_train_split(sdbn, train=0.8)   ❸
ranksvm_ltr(train, model_name='test1', feature_set=feature_set)   ❹
eval_model(test, model_name='test1', sdbn=sdbn)   ❺
```

❶ Random seed so results are reproducible

❷ Define two features searching RetroTech long/short descriptions

❸ Split the sdbn data with 80% of queries in training set

❹ Train and upload RankSVM model to Solr using the train set

❺ Search Solr to evaluate the model on the test queries Output

```
{'blue ray': 0.0,
 'dryer': 0.0,
 'headphones': 0.0,
 'dark of moon': 0.0,
 'transformers dvd': 0.003258006235976338}
```

By looking at Listing 12.2, let’s briefly revisit what we learned in Chapter 10. We define a feature set: the set of Solr queries we want to use to learn a ranking function. We must choose carefully: hoping to find features that can learn the training data from Listing 12.1.

How would we know if our model learned the training data well? Also recall from Chapter 10 that we performed a test-train split at the query level. This reserves some of the training data for evaluation in a test set. We’re like the professor giving the student (here the model) a final exam. You might give students many sample problems to study for the test (the training data). But to see if students truly learned the material, you give them a final exam (the test queries). This helps you evaluate whether the student knows what you’ve taught them before sending them off into the real world.

(As we all know, of course, success in the classroom does not always equate to success in the real world. Graduating our model into the real world, with live users in an A/B test, might show it does not perform as well as we hoped! More on this in a bit.)

Finally, what is the statistic next to each test query? How do we evaluate the students success on the test queries? Recall from chapter 10, we simply used precision (the proportion of relevant queries). This statistic sums the top N grades and divides by N (for us N=10), effectively the average relevance grade. We recommend exploring additional statistics for model training and evaluation that bias towards getting the top positions correct, such as Discounted Cumulative Gain (DCG), Normalized DGC (NDCG), or Expected Reciprocal Rank (ERR). For our purposes, though, your brain is already quite full, so we’ll stay with the simpler precision statistic.

Just looking at the relevance metrics for our test queries from Listing 12.2, our model does quite poorly in offline testing. Certainly by improving our offline metrics, we would see quite an improvement with live users in an A/B test.

12.2 A/B testing a new model

In this section, we’ll simulate running an A/B test and compare Listing 12.2's model to a model that seems to perform better in the lab. We’ll reflect on the results of the A/B test, setting us up to complete the Automated LTR feedback loop we introduced in Chapter 11. We’ll finish by reflecting on what didn’t go so well, spending the remainder of the chapter adding a crucial, missing piece to our Automated LTR feedback loop.

12.2.1 Taking a better model out for a test drive

Our original LTR model hasn’t performed very well, as seen in Listing 12.2's output. In this section we’ll train a new model, and once it looks promising, we’ll deploy the model in an A/B test against the model we trained in Listing 12.2.

Let’s introduce an improved model in Listing 12.3.

Listing 12.3 A new and improved model by changing the features

```
random.seed(1234)   ❶

feature_set_better = [   ❷
    {
      "name" : "name_fuzzy",          ❸
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "name_ngram:(${keywords})"
      }
    },
    {
      "name" : "name_pf2",    ❹
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!edismax qf=name name pf2=name}(${keywords})"
      }
    },
    {
      "name" : "shortDescription_pf2",     ❺
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!edismax qf=shortDescription pf2=shortDescription}(
        ➥${keywords})"
      }
    },
]

sdbn = sessions_to_sdbn(sessions)

train, test = test_train_split(sdbn, train=0.8)
ranksvm_ltr(train, 'test2', feature_set_better)  ❻
eval2 = eval_model(test, 'test2', sdbn=sdbn)      ❻

eval2
```

❶ Random seed so results are reproducible

❷ Three new features for searching RetroTech

❸ Feature performing fuzzy search on name field

❹ Feature searching for terms and phrase bigrams in name field

❺ Feature searching for terms and phrase bigrams in shortDescription field

❻ Train, deploy, and evaluate new model, `test2` Output

```
{'blue ray': 0.0,
 'dryer': 0.07068309073137659,
 'headphones': 0.06426395939086295,
 'dark of moon': 0.25681268708548066,
 'transformers dvd': 0.10077083021678328}
```

The first thing we notice about Listing 12.3 is its output. On the same set of test queries, our model seems to perform much better. This seems promising! Indeed, we’ve chosen a set of features that seems to capture the text-matching aspects of relevance better!

The astute reader might notice we've kept the test queries the same as Listing 12.2. We've intentionally done this for clarity purposes. It's "good enough" to teach you fundamental AI-powered search skills. In real-life, however, we would want a truly random test-train split to better evaluate the model's performance. We might even take things further, performing *cross-validation* - the resampling and training of many models on different test/train dataset splits to ensure the models generalize well without overfitting to the training data. If you'd like to dive deeper into offline model evaluation, we recommend a more general machine learning book, such as *Machine Learning Bootcamp* by Alexey Grigorov (https://www.manning.com/books/machine-learning-bookcamp).

Perhaps your search team feels the model trained in Listing 12.3 has promise, and is good enough to deploy to production! The team's hopes are up, so let's see what happens when we deploy to production for deeper evaluation with live users.

12.2.2 Defining an A/B test in the context of automated LTR

With an Automated LTR retraining loop setup, we can easily deploy promising new models. But a few questions remain unanswered: how do we know whether what we built in the lab actually performs in the real world? It's quite a different thing to handle live, real-world scenarios.

In this section, we'll explore the results of A/B testing with (simulated) live users. We'll see how the A/B test serves as the ultimate arbiter of our Automated LTR system's success: a chance to correct issues in our offline automated LTR model training so our feedback loop can grow increasingly reliable.

You may know about A/B tests. Here we'll learn how they factor into an Automated LTR system. As illustrated in Figure 12.3, an *A/B test* randomly assigns users to two *variants*. Each *variant* contains a distinct set of application features. This might include anything from different button colors to new relevance ranking algorithms. Because users are randomly assigned to the variants, we can more reliably infer which variant performs best on chosen business outcomes such as sales, time spent on the app, user retention, or whatever else the business might choose to prioritize.

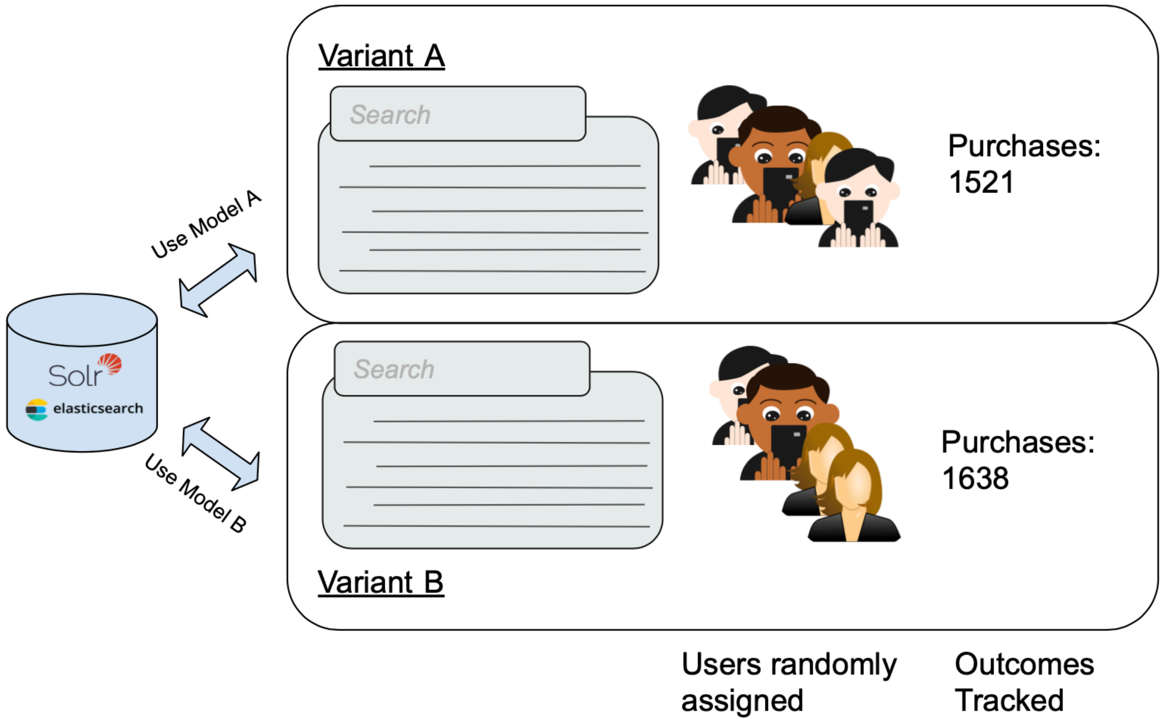


Figure 12.3 A search A/B test. Search users are randomly assigned to two relevance solutions (here two LTR models) with outcomes tracked.

12.2.3 Graduating the better model into an A/B test

Next up, we'll deploy our promising new model `test2` from Listing 12.3 into an A/B test. We'll then explore the implications of the test results. Hopes are high, and your team thinks this model might knock the socks off the competition: the poorly performing model from Listing 12.2, which we'll call `test1`.

In this section we'll simulate an A/B test, assigning 1000 users randomly to each model. In our case, these simulated users have specific items they want to buy. If they see those items, they'll make a purchase, and leave our store happy. If they don't, they might browse around, and most likely leave without making a purchase. Our search team, of course, doesn't know what users hope to buy - this information is hidden from us. We only see a stream of clicks and purchases, which, as we'll see, is heavily influenced by presentation bias.

In Listing 12.4, we have a population of users seeking the newest Transformer's movies by searching for `transformers dvd`. We'll stay focused on this single query during our discussion. Of course, with a real A/B test, we'd look over the full query set. The user population wouldn't be this static. But by zeroing into one query, we can more concretely understand the implications of our A/B test for Automated LTR. For a deeper overview of good A/B testing experimentation, we recommend the Manning book "Tuning Up" by David Sweet (https://www.manning.com/books/tuning-up-from-a-b-testing-to-bayesian-optimization).

Returning back to Listing 12.4, for each run of `a_or_b_model`, a model is assigned at random. Then the function `live_user_query` simulates a user searching with the query and selected model, scanning the results, possibly clicking and making a purchase. Unbeknownst to us, our user population has hidden preferences behind their queries: they hope to purchase items in `wants_to_purchase` in Listing 12.4 with a lower probability of purchasing an item in `might_purchase`. We run `a_or_b_model` 1000 times, collecting the purchases made by users that use each model.

Listing 12.4 Simulated A/B test, focusing only on 'transformers dvd' query

```
random.seed(1234)    ❶

wants_to_purchase = ['97360724240', '97363560449', '97363532149',
➥'97360810042']   ❷
might_purchase = ['97361312743', '97363455349', '97361372389']   ❸

def a_or_b_model(query, a_model, b_model):
    draw = random.random()

    user_made_purchase = False   ❸
    model_name = None
    if draw < 0.5:
        model_name=a_model    ❸
    else:
        model_name=b_model    ❸

    purchase_made = live_user_query(query=query,    ❹
                                   model_name=model_name,
                                   desired=wants_to_purchase,
                                   meh=might_purchase)
    return (model_name, purchase_made)


NUM_USERS=1000
purchases = {'test1': 0, 'test2': 0}
for _ in range(0, NUM_USERS):   ❺

    model_name, purchase_made = a_or_b_model(query='transformers dvd',
                                             a_model='test1',
                                             b_model='test2')

    if purchase_made:   ❻
        purchases[model_name]+= 1

purchases
```

❶ Random seed so results are reproducible

❷ Transformer movies our user population actually wants to buy (unknown to our team)

❸ Randomly each user to model a or b

❹ Simulate user searching and buying: user decides whether to buy if wants_to_purchase/might_purchase in results

❺ Simulate NUM_USERS users performing 'transformers dvd' search

❻ Count total number of purchases made by each model Output

```
{'test1': 21, 'test2': 15}
```

As we see in Listing 12.3's output, our golden student, model `test2`, actually performs *worse* in this A/B test! How can this be? What could have gone wrong to have such good offline test metric performance but poor outcomes in the real world? For the rest of this chapter, we'll dive into what's happening and attempt to address the problem. Thus you'll learn how live users can increase the accuracy of your Automated LTR system, allowing you to retrain with confidence!

12.2.4 When 'good' models go bad: what we can learn about a failed A/B test?

As we saw in Listing 12.4, a lot can change when our model enters the real world. In this section, we reflect on the implications of the A/B test we just ran to see what next steps would be appropriate.

What does it mean when a model performs great in the lab, but fails an A/B test? It means, according to our training data, we built a 'correct' LTR model. Unfortunately that training data misled us. We built a good model, but to the wrong specification. We need to correct issues with the training data itself: the judgments generated from our click model.

But how might problems creep in in our click model based judgments? We saw several issues in Chapter 11: *position bias* and *confidence bias*. Depending on your goals, UX, and domain, additional biases can creep in. In e-commerce, users might be enticed to click an item on sale, skewing the data towards those items. In a research setting, one article might provide a richer summary in the search results than another. Some biases blur the line between 'bias' and actual relevance for that domain. A product with a missing image, for example, might get fewer clicks. It might be technically identical to another 'relevant' product without an image. However to users a product missing an image seems less trustworthy and thus won't be clicked. Is that a bias? Or simply an actual indicator of 'relevance' for this domain, where product trustworthiness is a factor?

To make better judgments, should clicks be ignored or discounted, and instead should we use other behavioral signals? Perhaps follow-on actions after clicking such as a 'like' button, adding an item to a cart, or a "read more" button ought to be included? Perhaps we should ignore 'cheap' or accidental clicks when the user immediately hits the back button after clicking?

Using post-click actions can be valuable. However, we must ask how strongly search ranking influences events like a purchase or add-to-cart, or whether we should attribute other factors. For example, a lack of purchases could indicate an issue with a product display page, or with a complex checkout process, not just the search result's relevance for a specific query.

This seems to go against the A/B test we just ran. Counterintuitively, we might use an outcome like total purchases, in aggregate over all queries, to evaluate an A/B test. All other things in the app remain unchanged except the ranking algorithm. Thus we know any significant change, at the macroscopic, system level, must be caused by the one thing we changed. However, when you zero in to the microscopic: the specific query to document relationship, and away from the big picture, the causality gets complicated. Any single product will have very few purchase (many people view a $1000 television, but very few buy). The data may simply not be there in enough quantity to know whether the purchase related to a product's specific relevance for a query.

Accounting for all of the variations in search UX, domains, and behaviors would fill many books, and still fall short. The search space constantly evolves, with new ways of interacting with search results coming in and out of fashion. In the final analysis, for most scenarios, using clicks and standard click models suffices. Clicks in search UIs has been heavily studied. Still, arriving at good judgments is both an art and science: you may find a slight modification to a click model that accounts for extra signals is important to your domain and may provide tremendous gains in how your model performs in an A/B tests. You can spend as much time perfecting your click model as you spend doing search itself.

However, there is one, universally pernicious training data issue that challenges all click models: *presentation bias*. Regardless of whether we use clicks, or more complex signals, users never interact with what they can't see! We saw in our A/B test, our user population wants to buy a certain set of movies. We might ask: are these movies present in the training data? Would any Automated LTR system even be able to learn they're relevant if users never see them? After all, if they're never shown to users, they'll never be clicked, we'll never know whether they're relevant or not!

Next up we will dive into this difficult problem and learn how to automate optimization of our training data AND models in tandem.

12.3 Overcoming Presentation Bias: Knowing When to Explore vs Exploit

Search experts will commonly remark that "users won't click, what they can't find!". In other words, underneath Automated LTR is a chicken and egg problem. If the relevant result is never returned by the original, poorly tuned system - how could any click-based machine learning system learn that result is relevant? This is *presentation bias*.

In this section, you'll learn about one machine learning tool that selects documents to explore *despite those results having no click data*. This final missing piece of your Automated LTR system helps not just build models optimized for the training data, but actively participates in its own learning to grow the breadth of the available training data. We call a system that participates in its own learning an *Active Learning* system.

Figure 11.15 captures presentation bias. The items on the right could feasibly be relevant for our query, but with no traffic, we have no data to know either way. It would be nice to give some traffic to these results to learn whether or not they are relevant.

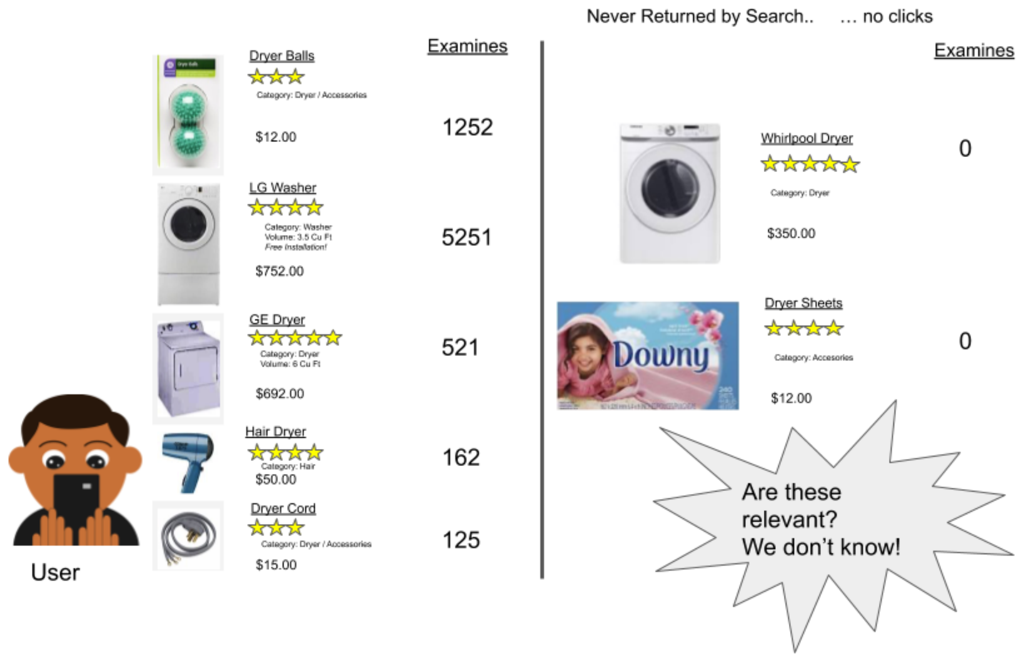


Figure 12.4 Presentation bias

To overcome presentation bias, we must carefully balance *exploiting* our model's current, hard-earned, knowledge and *exploring* beyond that knowledge. This is the *explore vs exploit* tradeoff. Exploring lets us gain knowledge, growing the coverage of our click model to new and different types of documents. However, if we always explore, we will never take advantage of our knowledge. When we *exploit*, we optimize for what we currently know performs well. Exploiting corresponds to our current LTR model that aligns to our training data. Knowing how to systematically balance exploring and exploiting is key - something we'll discuss in the next few sections, using a machine learning process built for this tool.

12.3.1 Presentation bias in RetroTech training data

Let's first analyze the current training data to get a lay of the land. What kinds of search results does the training data lack? Where is our knowledge incomplete? Another way of saying "presentation bias" is that there are potentially relevant search results excluded from the training data: blind spots we must detect and fight against. Once we've defined those blind spots in this section, we can then actively step back to work against them. This will set us up to retrain with a more robust model.

In Listing 12.5 we create a new `feature_set` named `explore`, in which we've created three simple features: `long_desc_match`, `short_desc_match`, and `name_match`, telling us whether a given field match occurs or not. These correspond to features our model has already learned. In addition, we've added a feature `has_promotion`. This feature becomes a `1.0` if the product is on sale and being promoted through marketing channels. We haven't explored this feature before; perhaps it's a blind spot?

Listing 12.5 Analyzing missing types of documents from `transformers dvd` judgments

```
sdbn = sessions_to_sdbn(sessions,     ❶
                        prior_weight=10,
                        prior_grade=0.2)

feature_set = [
    {
      "name" : "long_desc_match",     ❷
      "store": "explore",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "longDescription:(${keywords})^=1"
      }
    },
    {
      "name" : "short_desc_match",     ❷
      "store": "explore",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "shortDescription:(${keywords})^=1"
      }
    },
    {
      "name" : "name_match",       ❷
      "store": "explore",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "name:(${keywords})^=1"
      }
    },
    {
      "name" : "has_promotion",     ❸
      "store": "explore",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "promotion_b:true"
      }
    },
]

sdbn_ftrs = sdbn_with_features(sdbn, feature_set)    ❹
transformers_dvds = sdbn_ftrs[sdbn_ftrs['query'] == 'transformers dvd']   ❺
transformers_dvds
```

❶ Build SDBN judgments from current raw sessions

❷ Features correspond to fields already used to train LTR model

❸ New feature we're exploring for blind spot: promotions

❹ Log the feature values and return the sdbn judgments joined with the feature values

❺ Examine the properties of current `transformers dvd` training data

Output:

```
    query grade long_desc_match    short_desc_match name_match    has_promotion
618    transformers dvd    0.0 1.0    0.0    1.0    0.0
623    transformers dvd    0.0 1.0    1.0    1.0    0.0
622    transformers dvd    0.0    1.0    1.0    1.0    0.0
621    transformers dvd    0.0    1.0    0.0    1.0    0.0
620    transformers dvd    0.0    1.0    0.0    1.0    0.0
619    transformers dvd    0.0    1.0    0.0    1.0    0.0
617    transformers dvd    0.0    0.0    0.0    1.0    0.0
610    transformers dvd    0.3    0.0    0.0    1.0    0.0
615    transformers dvd    0.3    0.0    0.0    1.0    0.0
614    transformers dvd    0.3    0.0    0.0    1.0    0.0
613    transformers dvd    0.3    0.0    0.0    1.0    0.0
612    transformers dvd    0.3    0.0    0.0    1.0    0.0
611    transformers dvd    0.3    0.0    0.0    1.0    0.0
624    transformers dvd    0.0    0.0    0.0    1.0    0.0
616    transformers dvd    0.0    0.0    1.0    1.0    0.0
625    transformers dvd    0.0    1.0    0.0    1.0    0.0
```

We see some gaps in our training data's knowledge in the output of Listing 12.5:

- Every item includes a name match
- No "promotions" are present
- There's a range of long_desc_match and short_desc_match values

Intuitively, if we want to expand our knowledge, we would show users searching for `transformers dvd` something completely outside the box from what's in Listing 12.5's output. That would mean showing users a promoted item, possibly one with no name match. In other words we need to get search out of its own echo chamber by explicitly diversifying what we show to the user away from what's in the training data. The only question is: how much of a risk are we willing to take to improve our knowledge? We don't want to blanket the search results with random products just to broaden our training data.

What we've done so far has not been systematic: we've only eyeballed a single query to see what was missing. How might we automate this? Next up we'll discuss one method for automating exploration using a tool called a Gaussian Process.

12.3.2 Beyond the ad-hoc: thoughtfully exploring with a Gaussian Process

A *Gaussian Process* is a statistical model that makes predictions along with a probability distribution capturing the certainty of that prediction. In this section we'll use a Gaussian Process to select areas for exploration. After this section, we'll create a more robust way of finding gaps in our data than just eyeballing.

GAUSSIAN PROCESS BY EXAMPLE: EXPLORING A NEW RIVER BY EXPLOITING EXISTING KNOWLEDGE

To get at intuition for Gaussian Processes, let’s use a concrete example of real-life exploration. This sets you up for understanding what the underlying math is doing, giving you a sense for how to make explore vs exploit tradeoffs. Bare with us, this analogy will pay off, we promise! You’ll be setup to think more deeply about how we might, mathematically, make explore vs exploit tradeoffs.

Imagine you’re a scientist planning to survey a rarely explored river deep in the wilderness. As you plan your trip, you have only spotty river depth observations from past expeditions to know when its safe to travel. For example, one observation shows the river is two meters deep in April, another time in August it’s one meter deep. You’d like to pick an ideal date for your expedition optimizing for ideal river conditions (i.e. not monsoon season, but also not during a dry spell). However, you’re also scientist - you’d also like to make observations during yet unobserved times of the year to increase the knowledge of the river. So you want to be *somewhat* adventurous. Figure 12.5 shows us river depth measurements that have been made throughout the year you might use to plan your trip

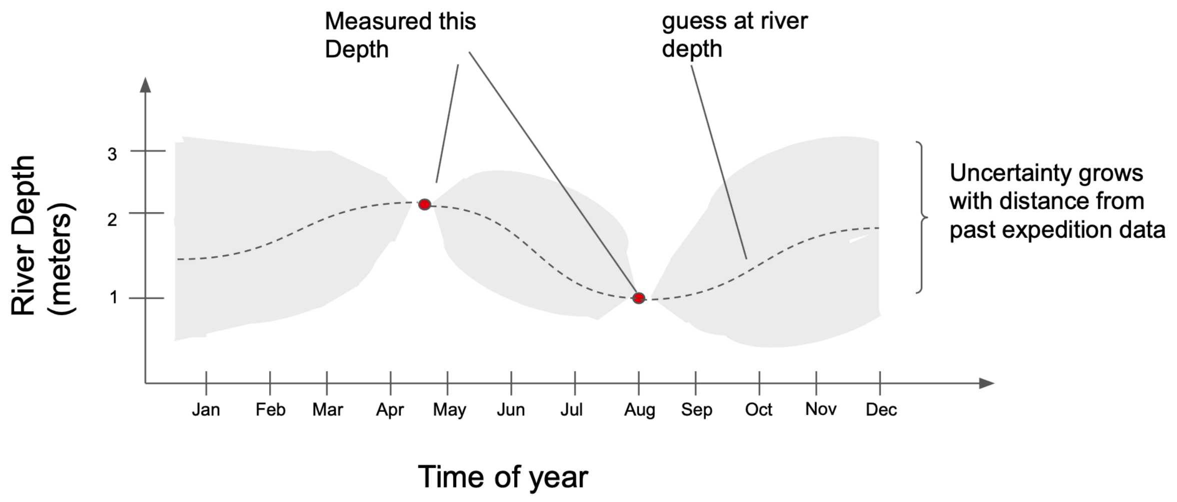


Figure 12.5 Exploring a river, uncertain of the river’s depth grows from past observations. How would we pick the best time of year that was both safe and also maximally increased our knowledge of the river’s depth?

How might we choose a date for the expedition? If you observed a river depth of two meters on April 14th, you would guess that the April 15th depth would be very close to two meters. Traveling during that time might be pleasant: you know the river wouldn’t be excessively flooded. However, you wouldn’t gain much knowledge about the river. What about trying to go several months away from this observation, like January? January would be too far from April to understand the river’s likely depth. We might travel during a treacherous time of the year.

However we'd almost certainly gain new knowledge—perhaps far more than we bargained for! With so little to go on, there's too much risk exploring this time of the year.

In Figure 12.5 we see an educated guess at the river level, based on an expected correlation between adjacent dates (April 15th and 14th should be very close). Our level of certainty decreases as we move away from direct observations: the widening gray zone in Figure 12.5.

Figure 12.5 is a Gaussian Process. It mathematically captures a prediction, along with our uncertainty in each prediction. How does this relate to relevance ranking? Just as nearby dates have similar river levels, similar search results would be similar in their relevance . Consider our `explore` features from Listing 12.5. Those with strong name matches for `transformers dvd`, not promoted, and no short/long description matches would likely have similar relevance grades - all moderately relevant. As we move away from these well trod examples - perhaps adding in promoted items - we grow less certain in our educated guesses. If we go very far, something way outside the box, like a search result with no name match, that's promoted, but with strong short/long description field matches, our uncertainty grows very high. Just like the scientist considering a trip in January, we have almost no ability to make a good guess whether those results could be relevant. It might involve too much risk to show those results to users.

We use Gaussian Processes to balance exploiting existing relevance knowledge with riskier exploration to gain knowledge. Gaussian Processes use incomplete information to make careful tradeoffs in likely quality and the knowledge gained. For example, trading off ideal river conditions or a likely relevant search result with learning more about river conditions or learning about the relevance of a new kind of search result. We can carefully choose how far away from known, safe search results we want to venture to gain new knowledge.

In our `transformers dvd` case, what kind of search result would have high upside, likely also be relevant/safe to explore, but also maximally increase our knowledge? Let's train a Gaussian Process and find out!

12.3.3 Training and Analyzing a Gaussian Process

Next we'll get hands-on to see how a Gaussian Process works. We'll train a Gaussian Process on the `transformers dvd` query. We'll then use it to generate the best exploration candidates. You'll see how we score those exploration candidates to maximally reduce our risk and increase the likelihood we'll gain knowledge.

In Listing 12.6 we train a Gaussian Process, using the `GaussianProcessRegressor` (aka `gpr`) from `sklearn`. This code creates a GaussianProcess that attempts to predict the relevance `grade` as a function of the `explore` features we logged.

Listing 12.6 Train a `GaussianProcessRegressor` on our training data

```
from sklearn.gaussian_process import GaussianProcessRegressor

y_train = transformers_dvds['grade']   ❶
x_train = transformers_dvds[['long_desc_match', 'short_desc_match',
                             'name_match', 'has_promotion']]   ❷

gpr=GaussianProcessRegressor()   ❸
gpr.fit(x_train, y_train)   ❸
```

❶ Predict relevance grades

❷ We use the features logged from the `explore` set above

❸ Create and train the gpr model

Once we've trained a `GaussianProcessRegressor`, we can use it to make predictions. Remember for `GaussianProcessRegressor` that means not only predicting an actual value, it also means the probability distribution of that prediction. This helps us gauge the model's certainty.

In Listing 12.7 we generate candidate feature values we'd like to possibly explore. With our river exploration example, these values correspond to a possible exploration dates for our scientist's expedition. In our case, as each feature can either be 0 or 1, we look at each possible feature value as a candidate.

Listing 12.7 Predicting a set of candidates to explore

```
zero_or_one = [0,1]

index = pd.MultiIndex.from_product([zero_or_one] * 4,
                  names = ['long_desc_match', 'short_desc_match',
                      'name_match', 'has_promotion'])   ❶
explore_options = pd.DataFrame(index=index).reset_index()

predictions_with_std = gpr.predict(explore_options[
➥['long_desc_match', 'short_desc_match', 'name_match',
➥'has_promotion']], return_std=True)   ❷
explore_options['predicted_grade'] = predictions_with_std[0]   ❸
explore_options['prediction_stddev'] = predictions_with_std[1]   ❸

explore_options.sort_values('prediction_stddev')
```

❶ Generate candidates: each feature value we want to explore 0 or 1 for the listed features

❷ Predict grade and standard deviation for those candidates based on gpr's probability distribution

❸ Store the predicted grade and standard deviation from the `gpr`

Output:

```
    long_desc_match short_desc_match  name_match  has_promotion
    ➥predicted_grade    prediction_stddev
2   0               0                 1           0
➥2.250003e-01    0.000000
6   0               1                 1           0
➥-3.569266e-08   0.000000
10  1               0                 1           0
➥-2.853953e-07   0.000000
14  1               1                 1           0
➥-2.458225e-07   0.000000
8   1               0                 0           0
➥-2.042939e-07   0.795059
11  1               0                 1           1
➥-2.042939e-07   0.795059
```

In the output of Listing 12.7 we see a `predicted_grade` - the `gpr’s educated guess on the relevance of that example. We also have `prediction_stddev`, which captures the gray band in Figure 12.5 - how much uncertainty is in the prediction.

We note in the output of Listing 12.7 that the standard deviation is 0 for name matches. In other words: the `gpr` has a lot of information about when `name_match`=1, as we should expect from our earlier ad-hoc analysis. We see after these observations that the standard deviation drops off a cliff. We lack a great deal of knowledge beyond these strong name match examples.

The output begins to show the presentation biases we intuitively detected in Listing 12.5. A tremendous amount of knowledge about name matches, but little knowledge about other cases. Which case would be worth exploring with live users that also minimizes the risk we’ll show users something completely strange in the search results?

In Listing 12.8 we generate and score exploration candidates.

Listing 12.8 Scoring the exploration candidates with Expected Improvement algorithm

```
from scipy.stats import norm

theta = 0.6
explore_options['opportunity'] = explore_options['predicted_grade'] -
➥sdbn['grade'].mean() - theta   ❶

explore_options['prob_of_improvement'] = norm.cdf( (explore_options[
➥'opportunity']) / explore_options['prediction_stddev'])   ❷

explore_options['expected_improvement'] = explore_options['opportunity'] *
➥explore_options['prob_of_improvement'] \
 + explore_options['prediction_stddev'] * norm.pdf( explore_options[
 ➥'opportunity'] / explore_options['prediction_stddev'])   ❸

explore_options.sort_values('expected_improvement', ascending=False).head()   ❹
```

❶ Quantify the 'opportunity' whether the prediction grade is likely to be above or below the typical grade

❷ Probability we’ll improve over mean, considering the amount of uncertainty in the prediction

❸ How much there is to gain given the probability and improvement and the magnitude of the improvement

❹ Sort to show best exploration candidates

Output:

```
long_desc_match short_desc_match  name_match  has_promotion opportunity
➥...   prob_of_improvement expected_improvement
0              0                 0           1             -0.697673
➥...   0.226541            0.121908
1              0                 0           1             -3.660364e-08
➥...   0.200650            0.104241
1              1                 0           1             -4.993262e-08
➥...   0.200650            0.104241
0              1                 0           1             -6.792379e-08
➥...   0.200650            0.104241
0              0                 0           0              1.364694e-01
➥...   0.208978            0.093904
```

Listing 12.8 uses an algorithm called "Expected Improvement" to select the best candidate. We won't dive into the nuts and bolts of this algorithm beyond the basic intuition, so we recommend a distil.pub article entitle "Exploring Bayesian Optimization" (https://distill.pub/2020/bayesian-optimization/) if you'd like to learn more.

The basic intuition behind the Expected Improvement algorithm is to select candidates that have the highest upside. How do we define this? It's either the case that we know to a high degree of certainty there's an upside (standard deviation is low and the predicted grade is high) OR we know that there's a high degree of uncertainty but the predicted grade is still high enough to take a gamble. We can see this on the following line of code from Listing 12.8:

```
explore_options['expected_improvement'] = explore_options['opportunity'] *
➥explore_options['prob_of_improvement'] \
 + explore_options['prediction_stddev'] * norm.pdf( explore_options[
   ➥'opportunity'] / explore_options['prediction_stddev'])
```

More of a 'sure thing' with exploration is covered by:

```
explore_options['opportunity'] * explore_options['prob_of_improvement']
```

Meanwhile the unknown opportunity with wide variance is covered by the second clause (after the +):

```
explore_options['prediction_stddev'] * norm.pdf( explore_options[
➥'opportunity'] / explore_options['prediction_stddev'])
```

In the first clause, you'll notice is opportunity (how much we expect to gain) times the probability that improvement happens corresponds to feeling confident in an outcome. On the other hand, the second clause depends much more on the standard deviation. The higher the standard deviation *and* opportunity, the more likely it will be selected.

We can calibrate our risk tolerance with a parameter called `theta`: the higher this value, the more we prefer candidates with a higher standard deviation. A high `theta` causes `opportunity` to diminish towards 0. This biases scoring to the second clause - the unknown, higher standard deviation cases.

If we set `theta` too high our `gpr` operates purely in a mode of selecting candidates entirely to learn about them with no thought for whether they might be useful to the user. If `theta` is too low, we won't burst out of the bubble very much, and instead we'd bias towards existing knowledge. A high `theta` is analogous to a scientist willing to take a high degree of risk (explore in January) in Figure 12.5, while a very low `theta` would be like traveling during a risk averse time (April 14th). Because we're using this algorithm to augment an existing LTR system, we chose `theta` of 0.6, a bit high, to give us more knowledge.

In the output of Listing 12.8, we see that `gpr` confirms our earlier ad-hoc analysis: we should show users items with promotions. These products will more likely yield greater knowledge, with possibly a high upside from the gamble.

Knowing what we should explore, let's gather products from Solr to show users! Listing 12.9 shows how we might go about selecting products for exploration from Solr. We'll then intersperse one or more "explore" products into the existing model's search results to observe whether these new results get clicks.

Listing 12.9 Selecting a RetroTech product to explore from Solr

```
random.seed(1234)

explore_vect = explore_options.sort_values('expected_improvement',
➥ascending=False).head().iloc[0][['long_desc_match','short_desc_match',
➥'name_match', 'has_promotion']]   ❶

def explore(query, explore_vect, log=False):
    """ Explore according to the provided explore vector, select
        a random doc from that group."""
    query = explore_query(explore_vect, query)   ❷

    draw = random.random()

    request = {
            "fields": ["upc", "name", "manufacturer", "score"],
            "limit": 1,
            "params": {                         ❸
              "q": query,                       ❸
              "sort": f"random_{draw} DESC"    ❸
            }
        }

    if log:
        print(request)

    resp = requests.post('http://aips-solr:8983/solr/products/select',
                                    json=request).json()

    print(resp)

    return resp['response']['docs'][0]['upc']   ❹

explore_upc = explore('transformers dvd', explore_vect)
explore_upc
```

❶ Extract the best exploration candidate based on `expected_improvement`

❷ Translate the explore candidate into a Solr query, such as promoted_b:true to get promoted items

❸ Issue query to solr, and rank results in random order

❹ Select the first UPC for exploration

Output

```
97360810042
```

In Listing 12.9, we take the best exploration candidate - promoted items - then issue a Solr query to fetch documents with those characteristics. We omit some nitty gritty translation of the candidate into a Solr query, but you can imagine if `has_promotion` is set to 1.0 in the candidate, then we would issue a query searching for any item with a promotion: `+promotion_b:true` and so on.

We see in Listing 12.9's output, that for the query `transformers dvd`, the randomly selected promoted product for exploration is UPC 97360810042. This corresponds to "Transformers Dark of The Moon: Two Disc-Blu Ray DVD Combo". Interesting!

What should we do with this document? This comes down to a design decision. A common choice is to slot it into the 3rd position of the results. This is what we'll do, creating a new session like the one below. Note our UPC `97360810042` in the 3rd slot (`rank` == 2.0).

```
    sess_id query             rank  doc_id        clicked
400 100049  transformers dvd  0.0   93624974918   False
401 100049  transformers dvd  1.0   879862003524  False
402 100049  transformers dvd  2.0   97360810042   False   <-- INSERTED
403 100049  transformers dvd  3.0   708056579739  False
```

In the notebook for this chapter, we've simulated many exploration sessions. Each adds a random exploration candidate based on Listing 12.9, simulates whether the added exploration result was clicked, and appends it to a new set of sessions `with_explore_sessions`. Recall these sessions serve as the input we need to compute LTR training data (the SDBN based judgments from Chapter 11 / Listing 12.1).

Finally, we're coming back up for air: we have the data we need to rerun our Automated LTR training loop. We'll see what happens with these added training examples to our training data, and finally how we can fit this exploration into the overall Automated LTR algorithm.

12.3.4 Examining the outcome of our explorations

We've explored by showing some outside-the-box search results to live users. We now have new sessions appended to the original session data stored in the `with_explore_sessions` dataframe. In this section, we'll run the session data through our Automated LTR functions to regenerate training data and train a model. We'll conclude by running this new model in an A/B test to see the results.

As you'll recall, our Automated LTR helpers can regenerate our training data using the `sessions_to_sdbn` function. We do this in Listing 12.10, but this time with our augmented sessions that include exploration data.

Listing 12.10 Regenerate SDBN judgments from new sessions

```
new_sdbn = sessions_to_sdbn(with_explore_sessions,
                            prior_weight=10,
                            prior_grade=0.2)   ❶
new_sdbn.loc['transformers dvd']    ❷
```

❶ Build new SDBN judgments with given beta distribution prior

❷ Output `transformers dvd` judgments Output

```
doc_id      clicked examined  grade    beta_grade
97360724240    19.0    21.0      0.904762  0.677419
97360810042 78.0    87.0      0.896552  0.824742
97368920347 27.0    32.0      0.843750  0.690476
97363455349 731.0   2116.0    0.345463  0.344779
97361312804 726.0   2112.0    0.343750  0.343073
97363560449 733.0   2133.0    0.343647  0.342977
```

We see in Listing 12.10's output a new product has been included. Note in particular the addition of upc `97368920347`. Interestingly, this movie has `promoted_b` set to `true`, meaning it was one of the newly selected "explore" candidates from the previous section:

```
{
    "upc":"97368920347",
    "name":"The Transformers: The Movie - DVD",
    "name_ngram":"The Transformers: The Movie - DVD",
    "name_omit_norms":"The Transformers: The Movie - DVD",
    "name_txt_en_split":"The Transformers: The Movie - DVD",
    "manufacturer":"\\N",
    "shortDescription":"\\N",
    "longDescription":"\\N",
    "promotion_b":true,
    "id":"71a8850d-9295-4a31-a4ef-ccae356be014",
    "_version_":1710117634561802242
}
```

It seems users do seem drawn to promoted products. So let’s move our `has_promotion` feature from the explore feature set to our main model and retrain to see the impact. In Listing 12.11 we train a model with this new feature added to the mix to see the effect.

Listing 12.11 Rebuild model using updated judgments

```
random.seed(1234)

feature_set_better = [
    {
      "name" : "name_fuzzy",
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "name_ngram:(${keywords})"
      }
    },
    {
      "name" : "name_pf2",
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!edismax qf=name name pf2=name}(${keywords})"
      }
    },
    {
      "name" : "shortDescription_pf2",
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "{!edismax qf=shortDescription pf2=shortDescription}(
        ➥${keywords})"
      }
    },
    {
      "name" : "has_promotion",    ❶
      "store": "test",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "promotion_b:true^=1.0"
      }
    }
]

train, test = test_train_split(new_sdbn, train=0.8)    ❷
ranksvm_ltr(train, model_name='test3', feature_set=feature_set_better)    ❸
eval_model(test, model_name='test3', sdbn=new_sdbn)    ❹
```

❶ Adding `has_promotion` to the LTR model we're training

❷ Perform a test-train split at the query level

❸ Train a Rank SVM model on the training data

❹ Evaluate the model on the test data

Output

```
{'blue ray': 0.16923076923076924,
 'dryer': 0.05754359864451608,
 'headphones': 0.2385279187817259,
 'dark of moon': 0.25398773006134967,
 'transformers dvd': 0.563385595947119}
```

Wow! We see comparing the output of the earlier Listing 12.3 to Listing 12.11's that adding a promoted product to the training data creates a significant improvement in our offline test evaluation. The precision of `transformers dvd` jumped significantly! If we issue a search for `transformers dvd`, we see this reflected in our data:

```
search('transformers dvd', 'test3', at=10)
```

Output

```
[{'name': 'The Transformers: The Movie - DVD'},
 {'name': 'Transformers Animated: Transform and Roll Out - DVD'},
 {'name': 'Transformers: Revenge of the Fallen - Widescreen - DVD'},
 {'name': 'Transformers/Transformers: Revenge of the Fallen: Two-Movie Mega
 ➥Collection [2 Discs] - Widescreen - DVD'},
 {'name': 'Transformers: Dark of the Moon - Blu-ray Disc'}]
```

The proof, however, is in the pudding! We've been here before. We know great looking offline results don't always translate to the real world. What happens when we rerun the A/B test from Listing 12.4? If you recall we created a function `a_or_b_model` that randomly selects a model for a user's search. If the results contained an item the user secretly wanted to purchase, a purchase would likely occur. If we use this function to resimulate an A/B test in Listing 12.12, we see our new model appears to have hit the jackpot! Woohoo!

Listing 12.12 Rerun A/B test on new `test3` model

```
NUM_USERS=1000
purchases = {'test1': 0, 'test3': 0}
for _ in range(0, NUM_USERS):          ❶

    model_name, purchase_made = a_or_b_model(query='transformers dvd',    ❷
                                          a_model='test1',     ❷
                                          b_model='test3')    ❷

    if purchase_made:
        purchases[model_name]+= 1    ❸

purchases
```

❶ Simulate 1000 users

❷ Compare searches of transformers dvd between `test1` and `test3` models

❸ Save purchases made per model

Output:

```
{'test1': 14, 'test3': 227}
```

We've added automated exploration to our training data, making an Automated LTR system that not only relearns from the latest user behaviors, but leverages user interactions to explore what else might be relevant. The full system here acts like an attentive young child: often using the knowledge it has, but also eager to play and experiment. The child asks: what will my parents do when I drop milk on the floor? What might it feel like to pick up an insect? How might this rock taste? As adults, we know not to do these things! But when training new models, we need to think like the child: always the student, eager to learn and explore!

12.4 Explore, exploit, gather, rinse, repeat: the full Automated LTR loop

With the final pieces in place, we see how exploring new features helps us to overcome presentation bias. Feature exploration and training data exploration go hand-in-hand, as we learn our presentation biases by understanding the features we lack and may need to engineer into search. In this chapter, we used a simple example with 'promotions', but what other, more complex, features might show blind spots in your training data? In this section, let’s conclude by augmenting the full Automated LTR algorithm to include not just training a model with training data, but also exploration beyond the training data’s current bubble.

To summarize our new, auto-exploring Automated LTR algorithm, we follow these three main steps:

```
1. Exploit: gather known features and train the LTR model for ranking using
➥existing training data
2. Explore: use hypothesized, 'explore' features to eliminate training data
➥blind spots
3. Gather: with a deployed model, and a trained `gpr` model, show
➥explore/exploit search results, and gather clicks to build judgments
```

We can summarize the past three chapters by gathering them up in one final code listing. Listing 12.13 puts all the pieces together (with some internals omitted). Our main decision points in this algorithm are the features used to explore and exploit. We can also go under the hood to change the chosen click model, the LTR model architecture, and our tolerance for risk (the `theta` parameter).

Listing 12.13 Fully Automated LTR algorithm summarized

```
# =========
# EXPLOIT

exploit_feature_set = [         ❶
    {
      "name" : "name_fuzzy",
      "store": "exploit",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "name_ngram:(${keywords})"
      }
    }
    ... # more features for exploitation
]

train, test = test_train_split(sdbn, train=0.8)  ❶
ranksvm_ltr(train, model_name='exploit', feature_set=exploit_feature_set)  ❶
eval_model(test, model_name='exploit', sdbn=new_sdbn)  ❶

# ===============
# EXPLORE

explore_feature_set = [        ❷
    {
      "name" : "manufacturer_match",
      "store": "explore",
      "class" : "org.apache.solr.ltr.feature.SolrFeature",
      "params" : {
        "q" : "manufacturer:(${keywords})^=1"
      }
    }
    ... # more features for exploitation w/ existing features from exploit
]

explore_vect = best_explore_candidate(sdbn, explore_feature_set, theta=0.6)  ❸
explore_upc = explore('transformers dvd', explore_vect)  ❸


# =========
# GATHER                                         ❹
sdbn = sessions_to_sdbn(sessions,                ❹
                        prior_weight=10,         ❹
                        prior_grade=0.2)         ❹

# GOTO "exploit" and repeat full loop    ❺
```

❶ Train LTR model on known-good features and current training data

❷ Hypothesized new features to explore for blind spots

❸ Choose a new candidate for exploration (using "Expected Improvement" algorithm)

❹ Gather more session data and regenerate training data using SDBN click model

❺ Goto "exploit" to rebuild, deciding whether to promote any explore features

In this loop, we capture a better Automated LTR process. We actively learn our training data blinds spots by theorizing the features that might be behind them. Letting the loop run we observe its performance, deciding when to promote explore features to the full, production "exploit" feature set. As we retire old click data, we also can note when old features no longer matter, and when new features become important due to trends and seasonality.

Taken together, this algorithm gives you a robust mechanism for arriving at an ideal ranking that considers a full spectrum of options that could be shown to users. It lets you choose new features to investigate blind spots, arriving at a relevance algorithm that maximizes what users expect to see for their searches.

12.5 Summary

- Performing well in an offline test shows our features can approximate the training data. However, that’s no guarantee of success. An A/B test can show us situations where the training data itself was misleading.
- Training data must monitored for biases and carefully corrected.
- Presentation bias is one of search’s most pernicious issues. Presentation bias happens when our models can’t learn what’s relevant from user clicks. This happens when our search never shows the truly relevant result, so users never click on them!
- We can overcome presentation bias by making the Automated LTR process an active participant in finding blind spots in the training data. Models that do this particpate in *active learning*.
- A *Gaussian Process* is one way to select promising opportunities for exploration. Using a set of features, we can find what’s missing in the training data, selecting those to show users. We can experiment with different ways of describing the data via features to find new and interesting blind-spots and areas of investigation.
- When we put together exploiting existing knowledge with exploring blind spots, we have a better, Automated LTR - reflected intelligence that can automatically explore and exploit features with little internal maintenance.

13 Semantic search with dense vectors

This chapter covers

- Representing the meaning of text with dense vectors
- An introduction to Transformers, and their impact on text representation and retrieval
- Building a fast and accurate autocomplete using transformer models
- Using approximate nearest neighbor (ANN) search to speed up dense vector retrieval
- Semantic-search using dense vectors

In this chapter, we’ll start our journey into the emerging future of search, where we see the swell wave of hypercontextual vectors soak into the beaches of information retrieval.

Our story begins with what you have already learned in section 2.5, that we can represent context as numerical vectors, and we can compare these vectors to see which are closer using a similarity metric. In chapter 2 we demonstrated the concept of searching on dense vectors, a technique known as "dense vector search", but our examples were simple and contrived (searching on made-up food attributes). In this chapter we pose the question - how can we convert real world unstructured text into a high dimensional dense vector space that attempts to model the actual meaning of the text representation. And how can we leverage this representation of knowledge for advanced search applications?

13.1 Language Translation as an Analogy for Text Representation

We’re going to use language translation as an example to understand what we mean by "dense vectors", to get us in the mood for an advanced computational representation of knowledge. Take the following two sentences of "Hello to you!" (English), and "Barev Dzes" (Armenian). These two expressions hold approximately the same meaning: each is a greeting, with some implied sense of formality.

Computationally, to successfully respond to the greeting of "Hello to you!", the machine must both comprehend the meaning of the prompt and also comprehend all the possible ideal responses, in the same vector space. When the answer is decided, the machine must then express it back to a person by generating the label from the answer's vector representation.

We'll call this representation of meaning the *embedding*. Embeddings are used interchangably between natural language processing tasks, and can be further molded to meet specific use cases. In this chapter we will introduce techniques and tools to get embeddings from text, and then use them to significantly enhance query and document interpretation within our search engine.

SIDEBAR Natural Language Processing (NLP)

Natural Language Processing (NLP) is the set of techniques and tools that converts unstructured text into machine actionable data. The field of NLP is quite large, and includes many areas of application. A comprehensive list of the problem areas are maintained at https://nlpprogress.com/

We will be focussing specifically on applying NLP to information retrieval, an important requirement of AI Powered Search!

One important point is worth noting upfront: Behind the two short English and Armenian greetings we mentioned, there are deep cultural nuances. Each of them carries a rich history, and learning them thus carries the context of those histories. This was the case with the semantic knowledge graphs we explored in chapter 5, but those only leveraged the context of documents within the search engine as their model. Transformers are usually trained on much larger corpuses of text, bringing in significantly more of this nuanced context from external sources.

13.1.1 Representation of Meaning through Text Embeddings

How did you, as a baby, child, teen, and beyond, actually learn the meaning of words? You were told, and you consumed knowledge and its representation! People who taught you already had this knowledge and the power to express it. Aside from someone pointing out a cat and saying "kitty" to you, you also watched movies and videos, and then moved on to literature and instructional material. You read books, blogs, periodicals, and letters. Through all this and more, you incorporated the knowledge into yourself, creating in your brain a dense representation of concepts and how they relate to one another and allow you to reason.

Can we impart to machines the very same content from which we obtained this power of language, and then expect them to understand and respond sensibly when queried? Hold on to your hats!

13.2 Search using Dense Vectors

The general theory behind using dense vectors for search instead of sparse vectors requires a general shift in understanding how to process and relate text. This section briefly reviews how sparse vector search works in comparison with dense vector search. We will also introduce **nearest-neighbor search** as one type of similarity used for dense vector search as compared to BM25 (the most common similarity function used for sparse vector search).

SIDEBAR **Nearest-Neighbor Search**

Also known as kNN (k-nearest-neighbor), nearest-neighbor search is the problem space of indexing numerical vectors of a uniform dimensionality into a data structure, and searching that data structure with a query vector for the closest 'k' related vectors. We will use cosine similarity (covered in section 1.2.2) to calculate the similarity between 2 vectors.

13.2.1 A brief refresher on sparse vectors

Sparse vectors require the use of an inverted index. An inverted index is like what you find in the back of any text book - a list of terms that reference their location in the source content. To efficiently find text, we structure information in an index by processing and normalizing tokens into a dictionary with references to the postings (the document identifiers and positions in which they occur). The resulting data structure (the inverted index) allows for fast lookup of those tokens. At search time we tokenize and normalize the query terms and, using the inverted index, match the document hits for retrieval and then apply the BM25 formula to score the documents and rank by similarity, as covered in section 3.2. Applying scores for each query term and document feature gives us a fast and relevant search, but this model suffers from the 'query term dependence' model of relevance, in which the terms (and normalized forms) are retrieved and ranked. The problem is that it uses the presence (and count) of query term strings to search and rank and not the *meaning* behind those strings. Therefore, the relevance scores are only useful in a relative sense to tell you which documents best matched the query, but not to objectively measure if any of the documents were actually good matches. Dense vector approaches, as we'll see, can supply a more global sense of relevance that is also comparable across queries.

13.2.2 A conceptual dense vector search engine

We want to capture the meaning of content when we are processing documents, and when searching we want to retrieve and rank based on the meaning and intent of the query. With this goal in mind, we will process documents and get the **embeddings**, and store them in some kind of embedding index. Then, at search time, we will process the query to get embeddings, and use those query embeddings to search the indexed document embeddings. Figure 13.1 shows a simplified diagram of this process, which will be expanded upon in Section 1.4.

SIDEBAR **Embeddings**

An Embedding is a dense vector of numeric features that persist a representation of knowlege learned from text. Embeddings can span different lengths of content. Different embeddings can be made to represent a single word, sentence, paragraph, or even a whole document.

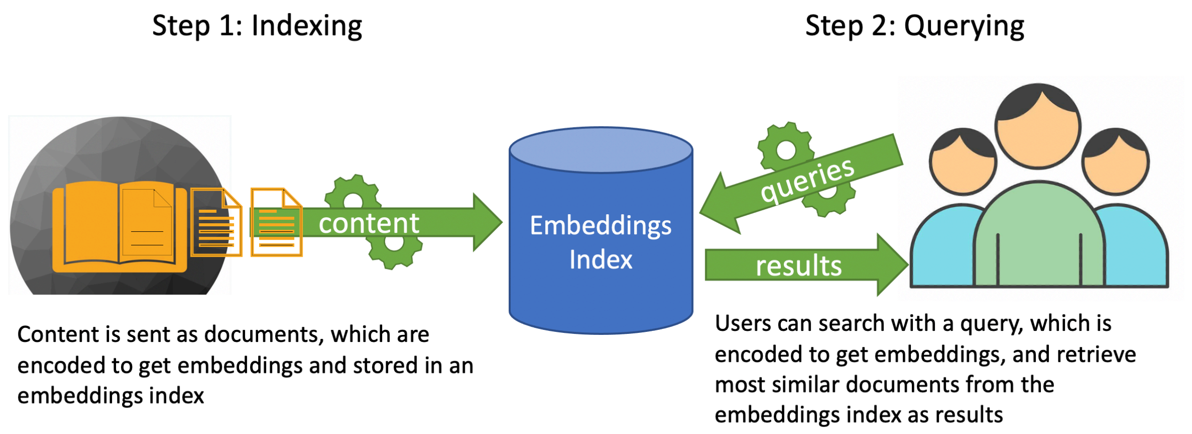

Step 1: Indexing
Step 2: Querying
content
Embeddings Index
queries
results
Content is sent as documents, which are encoded to get embeddings and stored in an embeddings index
Users can search with a query, which is encoded to get embeddings, and retrieve most similar documents from the embeddings index as results

Figure 13.1 Building and searching the embeddings index. The content is processed and added to the index from the left, and from right a user queries the index to retrieve results.

The embeddings for both documents and queries exist in the same vector space. This is very important. If you map documents to one vector space, and queries to another vector space, you'll be matching apples to oranges! So the embeddings must belong to the same space for this to work effectively.

But what is an embedding exactly, and how do we search one? Well, an embedding is a vector of some set number of dimensions. Based on this, we use cosine similarity (which was covered in-depth in Chapters 2 and 3), or some other similar distance measurement, to compare two vectors to each other and get a similarity score. This allows us to compare the vector of a query, to the vectors of all the documents in the content we want to search. The document vectors that are the most similar to the query vector are referred to as *nearest-neighbors*. Figure 13.2 illustrates this with three 2-dimensional vectors.

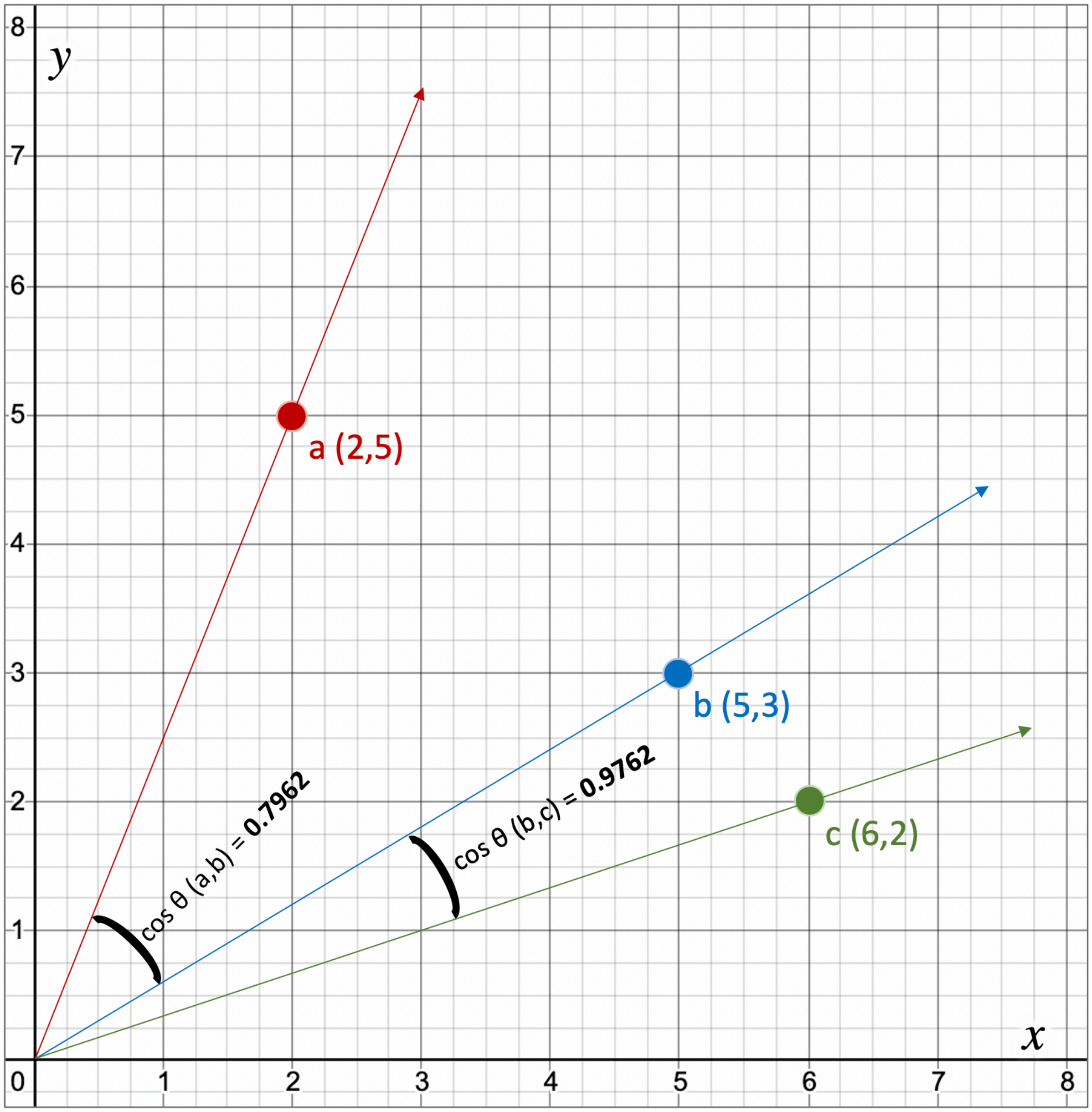


Figure 13.2 Three vectors (a, b, and c) plotted on a cartesian plane. The similarites as cos between a and b, and b and c are illustrated.

From figure 13.2, these are the cosine similarities between all the vectors ordered by highest similarity first.

- *cosine similarity (b,c)* = 0.9762
- *cosine similarity (a,b)* = 0.7962
- *cosine similarity (a,c)* = 0.6459

It is clear that *b* and *c* are closest to each other, so we say that *b* and *c* are the most similar of the three vectors.

Given this basic understanding of similarity and nearest-neighbors, we can easily apply cosine similarity to vectors of any length. So in 3 dimensional space, we'd be comparing vectors with three features (x,y,z). In our dense vector embedding space, we use vectors with hundreds of dimensions! But no matter the number of dimensions, the formula is the same:

Vector Similarity (a, b):

$$\cos(\theta) = \frac{a \cdot b}{|a| \times |b|}$$

Figure 13.3 Formula for the cosine similarity of two vectors

See section 3.1 for a recap of using this cosine similarity calculation to score the similarity between vectors. With this fundamental understanding of dense vector search and nearest neighbor similarity now in place, the next critical step is to figure out a way to get these mysterious embeddings!

13.3 Getting Text Embeddings by using using a Transformer Encoder

This section will introduce what a Transformer is, what its encoder is, and why to use it.

13.3.1 What is a Transformer?

Transformers are a class of deep neural network architectures, that are optimized towards encoding meaning as vectors, and decoding the meaning into a transformed representation of that meaning. They do this by first representing term labels as dense vectors by using their contexts in a sentence (the encoding part), and then leveraging an output model to translate the vectors into different text representations (the decoding part). One beautiful aspect of this approach is the separation of concerns between encoding and decoding. We will take advantage of this feature and use the encoding mechanism just to obtain the embeddings, which we can then leverage as a semantic representation of meaning independent of any decoding steps.

For you buzzwordians out there, you've probably heard of at least one type of Transformer Encoder: BERT.

SIDEBAR **Representing Meaning**

Recall the example English and Armenian greetings from the introduction in Section 1.1.

Using a specialized transformer and dataset for English to Armenian language translation, it would be possible to train a model to encode the two phrases "Hello to you!" and "Barev Dzes" into similar dense vectors.

However, we’re going to illustrate representation similarities in English only. Since you are reading this book in English, we can assume you already understand it!

Natural Language Processing lies at an intersection between computing, linguistics, and psychology. These areas are all important because of the way language has evolved in the world, and how we’ve engineered machines. Transformers aim to bridge the language-machine gaps through sophisticated techniques using a vast amounts of computing power.

Let’s start our journey with Transformers by understanding how transformer encoder models are trained and what they ultimately learn. To understand the motivations behind Transformers and the mechanisms behind them, it is important to know some history of the underlying concepts.

The year is 1953. Everything is dark. You open your eyes and find yourself sitting in a classroom with 20 other students at their own desks. On your desk is a pencil and a sheet of paper. On the sheet of paper, you see the sentence `Q: I went to the ________ and bought some vegetables.` You already know what to do, and you write down "store" in the blank. You peek over at a classmate sitting at the desk next to you, and they have written "market". A chime rings, and the answers are tallied. The most common answer is "store", and there are several with "market" and several with "grocer". This is the Cloze test. It is meant to test reading comprehension.

You are transported now to the year 1995. You are sitting in another classroom with students taking another test. This time, your sheet of paper has a very long paragraph. It looks to be about 60 words long, and is somewhat complicated. It reads:

> *Ours was the marsh country, down by the river, within, as the river wound, twenty miles of the sea. My first most vivid and broad impression of the identity of things seems to me to have been gained on a memorable raw afternoon towards evening. At such a time I found out for certain that this bleak place overgrown with nettles was the churchyard."'*

After the paragraph there is a question listed with a prompt for an answer: `Q: How far away from the sea is the churchyard? A:________` You write in the blank: "twenty miles". You have just completed one of a dozen questions in the Regents English reading comprehension test. Specifically, this tested your attention.

These two tests that you just witnessed are foundational to how we measure written language comprehension. To be able to pass these tests, you have to read, read, read, and read some more. In fact, by the time most people take these tests in school, they have already practiced reading for about 14 years and have amassed a huge amount of contextual knowledge.

These theories form the basis for what are known as Large Language Models - NLP models trained on lots and lots of text (for example, the entirety of Wikipedia, or the entirety of the Common Crawl dataset of the world wide web).

Major breakthroughs in the NLP field culminated in a 2018 paper by Google Research titled "Bidirectional Encoder Representations from Transformers" (aka BERT), which made use of the Cloze test and attention mechanisms to reach state of the art performance on many language comprehension benchmarks. See https://arxiv.org/abs/2007.01127v1 for the groundbreaking BERT paper.

BERT, specifically, performs self-learning by presenting Cloze tests to itself. The training style is 'self-supervised'. This means it is supervised learning, framed as an unsupervised task. This is ideal because it does not require data to be manually labelled beforehand for the initial model pretraining. You can just give it any text, and it will make the tests itself. In the training context, the Cloze test is known as *masked language modeling*. The model starts with a more basic embedding (for example, using the well-known word2vec or GloVe libraries) for each word in the vocabulary, and will randomly remove 15% of the tokens in a sentence for the test. The model then optimizes a loss function that will result in a higher Cloze test success rate. Also during the training process, it uses the surrounding tokens and contexts (the attention). Given a vector in a single training example, the resulting trained output vector is an embedding that contains a deeply learned representation of the word and the surrounding contexts.

We encourage you to learn more about Transformers and BERT if you are interested, by reading the paper. The good news is that aside from that basic understanding - you won't really need to know how BERT works under the covers to be able to get started using it.

What you do need to understand however, is how to get embeddings from a BERT encoder. The basic concept is shown in figure 13.4.

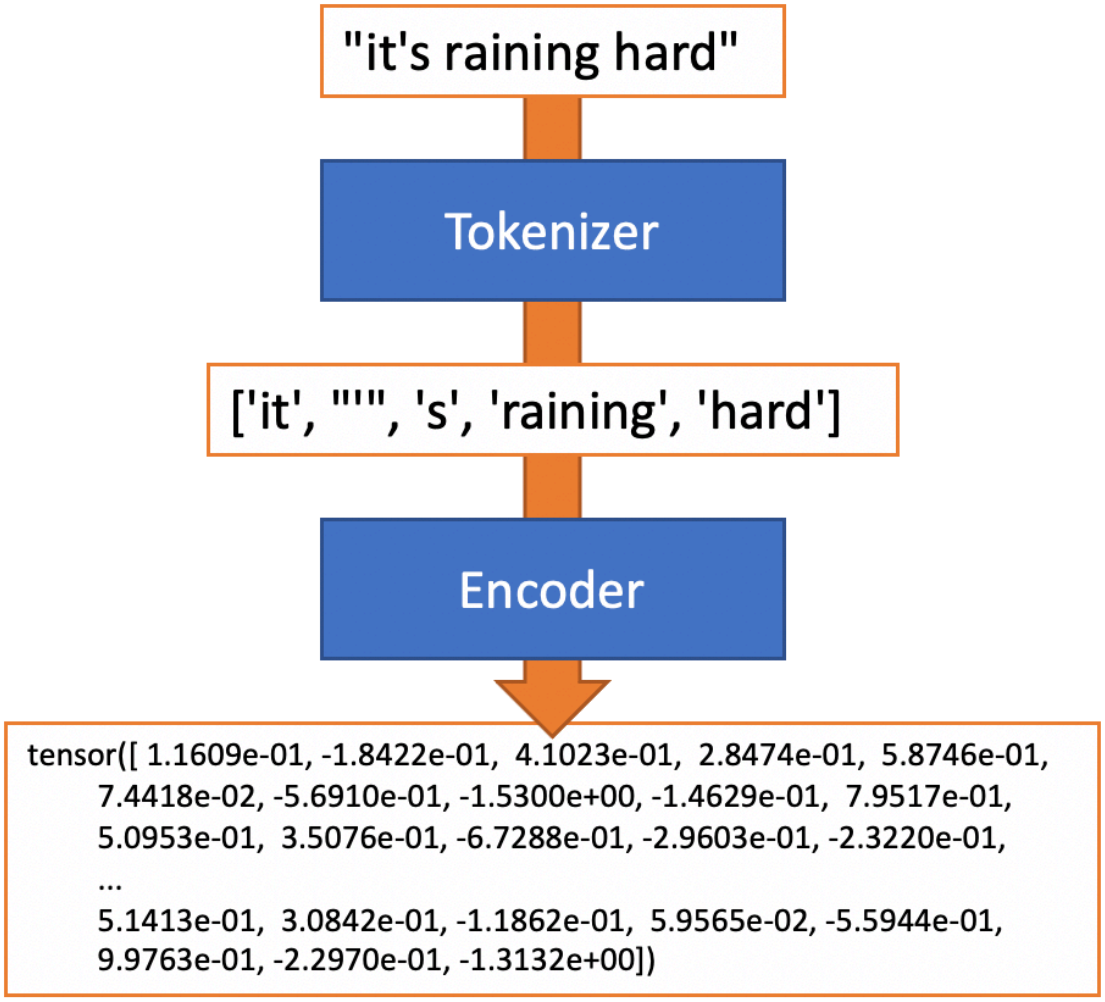

"it's raining hard"
Tokenizer
['it', "'", 's', 'raining', 'hard']
Encoder
tensor([1.1609e-01, -1.8422e-01, 4.1023e-01, 2.8474e-01, 5.8746e-01,
7.4418e-02, -5.6910e-01, -1.5300e+00, -1.4629e-01, 7.9517e-01,
5.0953e-01, 3.5076e-01, -6.7288e-01, -2.9603e-01, -2.3220e-01,
...
5.1413e-01, 3.0842e-01, -1.1862e-01, 5.9565e-02, -5.5944e-01,
9.9763e-01, -2.2970e-01, -1.3132e+00])

Figure 13.4 The Transformer encoder.

In Figure 13.4, we process text by first passing it through a tokenizer. The tokenizer will split text into *word pieces*, which are predefined parts of words that are represented in a vocabulary. This vocabulary is established for the model before it is trained. For example, the term `It's` will be split into three word pieces during tokenization: `it`, `'`, and `s`. The vocabulary used in the BERT paper used 30,000 word pieces. BERT also uses special word pieces to denote the beginning and end of sentences: `[CLS]` and `[SEP]`, respectively. Once tokenized, the token stream is passed into the BERT model for encoding. The encoding process then outputs a *tensor* which is an array of vectors (one vector for each token).

13.3.2 Openly available pre-trained transformer models

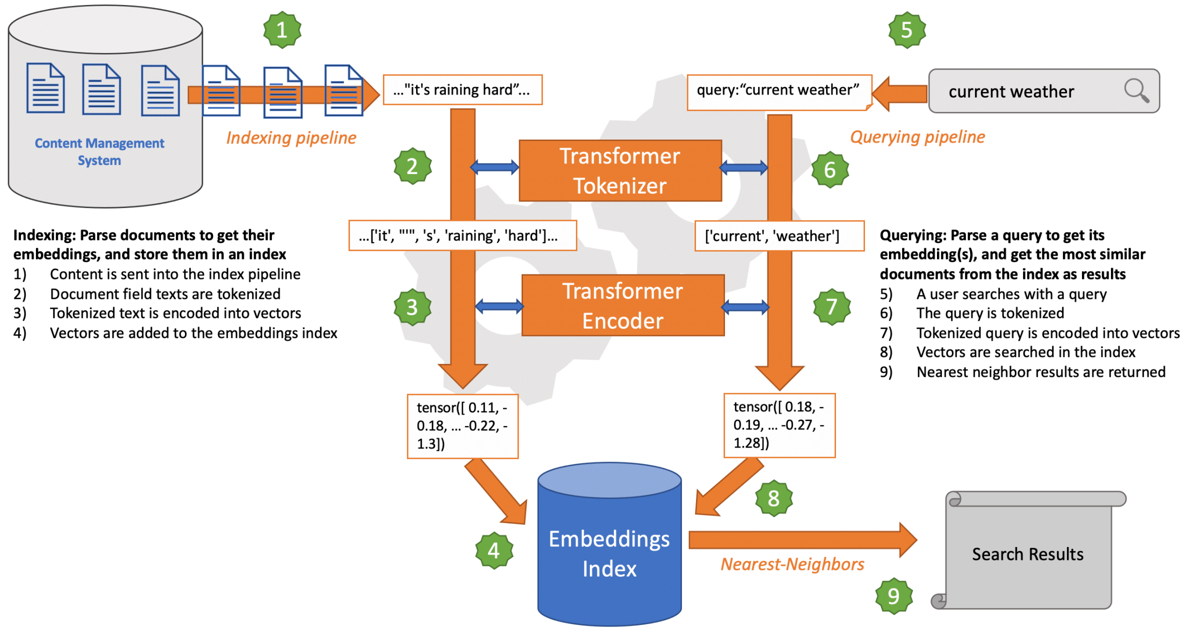
While Transformers enable state of the art language models to be built, having the knowledge and resources to build them from scratch can present a large hurdle for many. One very important aspect of working with Transformers is the large community and open toolsets that make it possible for any engineer to quickly get up and running with the technology. All it takes is some knowledge of Python and an internet connection.

The models that are trained by this process from scratch are very large, and are greater than 500Mb on average. The training itself also takes a large amount of expensive computing power and time, so being able to leverage pre-existing models as a starting point provides a significant advantage. We'll leverage this advantage in the next section as we begin applying one of these models to search.

13.4 Applying Transformers to Search

In this section, we will build a highly accurate natural language autocomplete for search, which will recommend more precise and otherwise related keywords based on a prefix of terms. We will do this by first passing our corpus text through a transformer to get an index of embeddings. Then we will use this index at query time to get the query embedding and return the nearest-neighbor concepts.

To visualize how this all works and what we're building, figure 13.5 shows an architecture diagram that explains all the pieces.


1
5
...''it's raining hard''...
query:''current weather''
current weather
Content Management System
Indexing pipeline
Querying pipeline
2
Transformer Tokenizer
6
...['it', '''', 's', 'raining', 'hard']...
['current', 'weather']
3
Transformer Encoder
7
tensor([0.11, -0.18, ... -0.22, -1.3])
tensor([0.18, -0.19, ... -0.27, -1.28])
8
Embeddings Index
4
Nearest-Neighbors
9
Search Results

Indexing: Parse documents to get their embeddings, and store them in an index
1) Content is sent into the index pipeline
2) Document field texts are tokenized
3) Tokenized text is encoded into vectors
4) Vectors are added to the embeddings index

Querying: Parse a query to get its embedding(s), and get the most similar documents from the index as results
5) A user searches with a query
6) The query is tokenized
7) Tokenized query is encoded into vectors
8) Vectors are searched in the index
9) Nearest neighbor results are returned

Figure 13.5 A conceptual architecture for end-to-end search using transformer encoded vectors.

We have a content source, a nearest neighbor index, a way to retrieve vectors from a transformer, and a similarity formula. We can then build pipelines for all these pieces to process and index content, and then retrieve and rank documents with a query.

13.4.1 Using the Outdoors StackExchange dataset

In chapter 5, we introduced the Outdoors dataset from Stack Exchange. We're choosing to use this dataset again in this chapter for a very important reason: the vocabulary and contexts in the outdoor question and answer domain already have good coverage in the transformer models we'll be using. Specifically, Wikipedia is used when training many transformer models, and Wikipedia has a section specifically on outdoors content (https://en.wikipedia.org/wiki/Outdoor).

Listing 13.1 walks through creating an outdoors dataset collection and indexing data into it.

Listing 13.1 Indexing the outdoors dataset

```
outdoors_collection="outdoors"
create_collection(outdoors_collection)
add_outdoors_fields_to_schema(outdoors_collection)
index_dataset_to_search_engine(outdoors_collection,
➥process_outdoors_dataset())
```

This is the schema for the outdoors collection that is created in the accompanying source code inside the `add_outdoors_fields_to_schema` method referenced Listing 13.1:

```
* url _(string)_
* post_type_id _(integer)_
* accepted_answer_id _(integer)_
* parent_id _(integer)_
* score _(integer)_
* view_count _(integer)_
* body _(text)_
* title _(text)_
* tags _(keyword)_
* answer_count _(integer)_
* owner_user_id _(integer)_
```

The indexed dataset contains documents representing both questions and answers, with answers linked to the original questions through a hierarchical structure.

All documents contain a `post_type_id` field, which differentiates question documents (`post_type_id=1`) from answer documents (`post_type_id=2`), as shown in Listing 13.2.

Listing 13.2 Querying the outdoors dataset with a noun phrase.

```
[
    {
        "id": "7",
        "post_type_id": 1,   ❶
        "accepted_answer_id": 21,   ❷
        "body": "There are a number of ways to purify water, off the top of
        ➥my head we have filters, iodine, and boiling... Which of these
        ➥is the safest? ...",
        "title": "What is the safest way to purify water?"   ❸
    },
    {
        "id": "15",
        "post_type_id": 2,   ❹
        "parent_id": 7,   ❺
        "body": "I have no science to back me up, but the SAFEST way would ,
        ➥be boiling..."   ❻
    },
        {
        "id": "21",   ❼
        "post_type_id": 2,   ❹
        "parent_id": 7,   ❺
        "body": "When you're asking for the safest way to purify water,
        ➥you're asking for the method that removes the most harmful
        ➥stuff from the water, like bacteria, viruses and larger
        ➥impurities like mud or sand. No one method is really perfect
        ➥at removing everything, so I usually use a two-stage approach:
        ➥Filter ... Boil ..."
    },
```

❶ This is a question (`post_type_id=1`)

❷ Document `21` is marked as the accepted answer to the question

❸ The `title` field (only on question documents) summarizes the question

❹ These are answer documents (`post_type_id=2`)

❺ These documents answer the question in the first document (`id=7`)

❻ Answers are provided in the `body` field on answer documents

❼ This is the document marked as the accepted answer on the question docuument (`accepted_answer_id=21`)

In the listing, the first document is the question document (`post_type_id=1`) with the question being represented by the `title` field (`What is the safest way to purify water?`), and the accepted answer to the question marked as document `21`.

All documents answering that question are represented as answer documents (`post_type_id=2`) with a `parent_id=7` corresponding to the id of the original question document (`id=7`).

Answers are always associated with a question and answers don't contain a title. The `body` field of question documents contains elaborations on the question, and the `body` field of answer documents contain the answer text. Several other fields (such as `view_count`, `answer_count`, and `owner_user_id`) are omitted here, but are available in the dataset as metadata fields which can help with search relevance using BM25 mixed with other signals.

With our outdoors dataset indexed, we can now query it. Take a moment to try out some queries and see what types of questions come back. Listing 13.3 searches through the question document titles and provides a good baseline for our search.

Listing 13.3 Querying the outdoors dataset with a noun phrase.

```
query_collection("climbing knots")
```

Response:

```
Query: climbing knots

Ranked Docs:
Question Title:What are the four climbing knots used by Jim Bridwell?
Question Title:What's a good resource for learning to tie knots for climbing?
Question Title:How to tie a figure eight on a bight?
Question Title:Can rock climbers easily transition to canyoning?
...
```

We've now verified we can query the corpus, and see that these are somewhat relevant titles for this noun phrase query. But this is just a basic search. Other queries do not perform nearly as well, for example the query `What is DEET` in Listing 13.4 shows very irrelevant results.

Listing 13.4 Querying the outdoors dataset with a question.

```
query_collection("What is DEET?")
```

Response:

```
Query: What is DEET?

Ranked Docs:
Question Title:What is Geocaching?
Question Title:What is bushcrafting?
Question Title:What is "catskiing"?
Question Title:What is a tent skirt and what is its purpose?
...
```

This shows that traditional keyword-based text search fails for some common natual language use cases. Specifically, the inverted index suffers from the query-term dependency problem. This means that the terms in the query are being matched as strings to the dictionary of the content. This is why you see strong matches for "what is" in the results in Listing 13.4! The *meaning* of the query is not comprehended, so the retrieval can only be based on string matching.

The rest of this chapter will provide the fundamentals needed for using transformers for natural language search, and in chapter 14 we will solve the question answering problem evident in Listing 13.4.

13.4.2 Fine-tuning and the Semantic Text Similarity Benchmark (STS-B)

Using a pre-trained transformer model out of the box is not recommended, because the initial training was done in a general language context without any specific use case in mind. In essence, it is "untuned", and using models this way is akin to indexing content in any search engine and not tuning for relevance.

Therefore, to realize the full potential of Transformers, they need to be refined to accomplish a specific task. This is known as *fine-tuning*. Fine-tuning is the process of taking a pre-trained model, and training it on more fit-for-purpose data to achieve a specific use case goal. For both autocomplete and semantic search, we are interested in fine-tuning to accomplish text similarity discovery tasks.

That brings us to the 'Semantic Text Similarity Benchmark' (STS-B) training and testing set (https://ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark). This benchmark includes passages that are semantically similar and dissimilar, and labeled accordingly. Using this dataset, a model can be fine-tuned to improve the accuracy of nearest-neighbor search between a set of terms and many passages in a corpus, which will be our use case in this chapter. In Chapter 14, however, we will fine-tune our own question-answering model so you can see how it is done.

For our purposes throughout the rest of this chapter, however, we'll choose to use a project that already includes a previously fine-tuned model for this task: SBERT.

13.4.3 Introducing SBERT, a transformer library built around similarity between sentences

SBERT, or *Sentence-BERT* is a technique and Python library based on Transformers that is built on the idea that a BERT model can be fine-tuned in such a way that two semantically similar sentences, and not just tokens, should be represented closer in vector space. Specifically, SBERT uses *pooling* of all the BERT embeddings in one sentence to a single vector. Pooling is a fancy way of saying it combines the values. Once SBERT pools the values, it then trains for similarity between sentences by using a special purpose neural network that learns to optimize for the STS-B task. If you'd like to dive into the specifics, please check out the SBERT paper further. Here is the paper summary written by the authors:

"In this publication, we present Sentence-BERT (SBERT), a modification of the pretrained BERT network that uses siamese and triplet network structures to derive semantically meaningful sentence embeddings that can be compared using cosine-similarity. This reduces the effort for finding the most similar pair from 65 hours with BERT / RoBERTa to about 5 seconds with SBERT, while maintaining the accuracy from BERT. We evaluate SBERT and SRoBERTa on common STS tasks and transfer learning tasks, where it outperforms other state-of-the-art sentence embeddings methods." https://arxiv.org/abs/1908.10084

The upcoming listings will give an overview of how to use SBERT via the `sentence_transformers` Python library. We start by importing the sentence transformers library using a pre-trained model named `roberta-base-nli-stsb-mean-tokens`. The model is based on an architecture called RoBERTa. It's helpful to think of RoBERTa as an evolved and improved version of BERT, with optimized hyperparameters and slight modifications to the original techniques. A *hyperparameter* is like a knob that you turn to make the implementation more or less accurate.

SIDEBAR **Hyperparameters**

In machine learning, hyperparameters are any parameter values that can be changed before training, that will alter the learning process and impact the final resulting model.

Unfortunately you never know what the hyperparameter values should be when you start, so they obtain optimized values over time using iteration and measurement.

Looking at the model name, we also see some terms that you may not recognize including "nli" and "mean-tokens". NLI stands for natural language inference (a subdomain of NLP used for language prediction), and mean-tokens refers to the whole sentence tokenization being pooled together as a mean of the floating point values of the token embeddings. Using mean-tokens returns a single 768 dimension embedding for the entire sentence. Some context might be lost in this process, but it is made up for when the model is trained for the task.

Listing 13.5 imports the `sentence_transformers` library, loads the model, and displays the full network architecture.

Listing 13.5 Loading the RoBERTa sentence_transformers model.

```
from sentence_transformers import SentenceTransformer
stsb = SentenceTransformer('roberta-base-nli-stsb-mean-tokens')
```

Now, the PyTorch object `stsb` holds the neural network architecture for the transformer as well as all the model weights.

With our model loaded, we can now retrieve embeddings from text. This is where the fun really begins. We can take sentences and pass them through the neural network architecture using the pre-trained model and get the embeddings as a result. We have four sentences that we will encode and assess in the upcoming listings, starting with Listing 13.5.

Listing 13.6 demonstrates how to encode multiple phrases into dense vector embeddings.

Listing 13.6 Encoding phrases as dense vector embeddings.

```
phrases = ["it's raining hard","it is wet outside","cars drive fast",
➥"motorcycles are loud"]  ❶
embeddings = stsb.encode(phrases, convert_to_tensor=True)  ❷
print('Number of embeddings:',len(embeddings))
print('Dimensions per embedding:',len(embeddings[0]))
print('The embedding feature values of "it\'s raining hard":')
print(embeddings[0])
```

❶ Our 4 sentences we want to encode. We will pass all of these in to be encoded as a single batch.

❷ Just call stsb.encode, and the abstraction of sentence_transformers does all the heavy lifting for you!

Response:

```
Number of embeddings: 4
Dimensions per embedding: 768
The embedding feature values of "it's raining hard":
tensor([ 1.1609e-01, -1.8422e-01,  4.1023e-01,  2.8474e-01,  5.8746e-01,
         7.4418e-02, -5.6910e-01, -1.5300e+00, -1.4629e-01,  7.9517e-01,
         5.0953e-01,  3.5076e-01, -6.7288e-01, -2.9603e-01, -2.3220e-01,
         ...  ❶
         5.1413e-01,  3.0842e-01, -1.1862e-01,  5.9565e-02, -5.5944e-01,
         9.9763e-01, -2.2970e-01, -1.3132e+00])
```

❶ Collapsed for brevity - there are 768 floating points inside this tensor.

In the listing, we take each sentence and pass it to the encoder. This results in a tensor for each sentence. A 'tensor' is Python object that contains a multidimensional matrix (an array of vectors) that is defined by its dimensionality. Tensors are produced by Transformer encoders, such as SBERT, when encoding text. For our use case, the tensor in Listing 13.6 is an embedding containing 768 dimensions represented as floating point numbers.

With our embeddings, we can now perform cosine similarities to see which phrases are closest neighbors of each other. We'll compare each phrase to every other phrase, and sort them by similarity to see which are most similar. This process is covered step by step in Listing 13.7 and Listing 13.8.

We use a Pytorch built-in library for cosine similarity to do these comparisons, which allows us to just a pass in the embeddings with a single function call. We can then sort the resulting similarities and see which two phrases are most similar to one another and which two are most dissimilar. Listing 13.7 calculates the similarities between each of the phrase embeddings.

Listing 13.7 Comparing all the phrases to each other.

```
from sentence_transformers import util as STutil
similarities = STutil.pytorch_cos_sim(embeddings, embeddings)
print('The shape of the resulting similarities:',similarities.shape)
```

Response:

```
The shape of the resulting similarities: torch.Size([4, 4])
```

We print the shape of the similarities object in Listing 13.6 to see how many comparisons we have. The shape of the similarities object in Listing 13.6 is 4x4. This is because we have 4 phrases, and each phrase has a similarity score to every other phrase and itself. All the similarity scores are between 0.0 (least similar) and 1.0 (most similar). The shape is included here to help show the complexity of comparing many phrases. If there were 100 phrases, the similarities shape would be 100x100. If there were 10000 phrases, the similarities shape would be 10000x10000. So as you add phrases to compare, note that the computational and space costs will increase with complexity n^2, where n is the number of phrases.

With the similarities for our 4 phrases computed, we sort and print them in in listing Listing 13.8.

Listing 13.8 Sorting by similarities and printing the results.

```
a_phrases = []
b_phrases = []
scores = []

for a in range(len(similarities)-1):   ❶
    for b in range(a+1, len(similarities)):   ❷
        a_phrases.append(phrases[a])
        b_phrases.append(phrases[b])
        scores.append(float(similarities[a][b]))   ❸

df = pandas.DataFrame({"phrase a":a_phrases,"phrase b":b_phrases,
➥"score":scores})
df.sort_values(by=["score"],ascending=False,ignore_index=True)   ❹
```

❶ Append all the phrase pairs to a dataframe

❷ We don't bother appending a phrases similarity to itself, as it will always be 1.0

❸ Get the score for each pair

❹ Sort the scores (remember to use ascending=False for highest scores first)

Response:

```
    phrase a           phrase b               score
0   it's raining hard  it is wet outside      0.669060
1   cars drive fast    motorcycles are loud   0.590783
2   it's raining hard  cars drive fast        0.281166
3   it's raining hard  motorcycles are loud   0.280801
4   it is wet outside  motorcycles are loud   0.204867
5   it is wet outside  cars drive fast        0.138172
```

We can now see that the two phrases that are most similar to one another are "it's raining hard" and "it is wet outside". We also see strong similarity for cars and motorcycles.

The two phrases that are most dissimilar are "it is wet outside" and "cars drive fast". It's very clear from these examples that this semantic encoding process is working - we can associate rain with it being moist and wet outside. The dense vector representations captured the context, and even though the words are different the meaning is still there. Note the scores: the top two similar comparisons have a score of greater than 0.59, and the next closest comparison has a score of less than 0.29. This is because only the top two comparisons seem to be similar to one another, as we would perceive them in a natural language understanding task. As intelligent people, we can group rain and wet ('weather'), and we also group cars and motorcycles ('vehicles'). Also interestingly, cars likely drive slower when it is wet on the ground, so that likely explains the low similarity of the last pair.

13.5 Natural Language Autocomplete

Now that we know our vector encoding and similarity process is working well, its time to put this embedding technique to work in a real search use case - natural language autocomplete!

In this section, we will show a practical use for sentence transformers at search time, with a basic and fast autocomplete implementation. Using what we've learned thus far, we will apply it to the outdoors dataset. Using SpaCy (the same Python NLP library used in Chapter 5) we will chunk nouns and verbs to get outdoors' concepts. We will then put these concepts in a dictionary and process them to get their embeddings. Then we will use the dictionary in an approximate nearest neighbor index to query in real time. This will give us the ability to enter a prefix or a term and get the most similar concepts that exist in the dictionary. Finally, we will take those concepts and present them in order of similarities to the user, demonstrating a smart, natural language autocomplete.

Experience and testing show that this works much better than even a well tuned suggester in Solr. We will see that it's much less noisy, and also that similar terms that are not spelled the same will be automatically included in the suggestions. This is because, as follows from listings 13.5 through 13.7, we are not using term strings to compare to one another. Rather, we are comparing the embeddings, and the embeddings contain meaning and context. This is the embodiment of searching for "things not strings".

13.5.1 Getting noun phrases and verb phrases for our nearest-neighbor vocabulary

Using what we learned in the knowledge graph chapter, we will write a simple method to extract *concepts* from the corpus. We won't include any hierarchy, and we won't be building a knowledge graph here. Rather, we just want a reliable list of frequent nouns and verbs.

The concepts in our example are the important "things" and "actions" that people usually search for. We also need to understand the data set, which is best accomplished by spending time reviewing the the concepts and how they relate to one another. Understanding the corpus is critical when building any search application, and there’s no exception when using advanced natural language processing technology.

Listing 13.9 provides a strategy that will provide a decent quality baseline of candidate concepts for our vocabulary while removing significant noise from the autocomplete results.

Listing 13.9 uses the SpaCy matcher to look for patterns as part of speech tags. We also explicitly remove the verb 'to be' from the verb concepts. The verb 'to be' is used frequently in many unuseful situations, and we don’t want that cluttering up our concept suggestions. We could also further improve quality by removing other noisy verbs like 'have' and 'can' but this is just an example for now. Also introduced in this listing is the SpaCy pipe. The SpaCy pipe method accepts a batch size and a number of threads as parameters, then processes text batches in parallel. This increases the throughput and processes text more quickly.

Listing 13.9 The spacy matcher makes quick work of getting the parts of text that we want.

```
...

phrases = []   ❶
sources = []   ❷

matcher = Matcher(nlp.vocab)   ❸
nountags = ['NN','NNP','NNS','NOUN']   ❹
verbtags = ['VB','VBD','VBG','VBN','VBP','VBZ','VERB']   ❺
matcher.add("noun_phrases", [[{"TAG":{"IN": nountags}, "IS_ALPHA":
➥True,"OP":"+"}]])   ❻
matcher.add("verb_phrases", [[{"TAG":{"IN": verbtags}, "IS_ALPHA":
➥True,"OP":"+", "LEMMA":{"NOT_IN":["be"]}}]])   ❼
for doc,idx in nlp.pipe(yieldTuple(df,"body",total=total),
➥batch_size=40, n_threads=4, as_tuples=True):   ❽
    text = doc.text
    matches = matcher(doc)
    for matchid,start,end in matches:   ❾
        span = doc[start:end]
        phrases.append(normalize(span))
        sources.append(span.text)


concepts = {}
labels = {}
for i in range(len(phrases)):
    phrase = phrases[i]
    if phrase not in concepts:
        concepts[phrase] = 0
        labels[phrase] = sources[i]   ❿
    concepts[phrase] += 1
```

❶ All the normalized noun/verb phrases ("concepts") in the corpus

❷ The original text labels that were normalized to the concept

❸ Use the spacy matcher to chunk patterns into concept labels

❹ Part of speech tags that match Nouns

❺ Part of speech tags that match Verbs

❻ Add a noun phrase matching pattern to the SpaCy analysis pipeline

❼ Add the verb phrase matching pattern. You can add more NOT_IN patterns to exclude other "stop word" verbs

❽ Process the *body* field for each Outdoors question, in batches of 40 documents using 4 threads.

❾ Gets all the noun and verb phrase matches and keeps them in the sources and phrases lists

❿ Aggregate the normalized concepts by term frequency

With the method in Listing 13.9, we can now get the list of concepts. When running this on your machine it may take some time, so be patient. Listing 13.10 returns the most prominent concepts and labels from the Outdoors corpus.

Listing 13.10 Examining the most frequent concepts in our corpus.

```
# Set load_from_cache=False to re-extract all the concepts from the corpus.
concepts,labels = getConcepts(outdoors_dataframe,load_from_cache=True)
topcons = {k:v for (k,v) in concepts.items() if v>5 }
print('Total number of labels:',len(labels.keys()))
print('Total number of concepts:',len(concepts.keys()))
print('Concepts with greater than 5 term frequency:',len(topcons.keys()))
print(json.dumps(topcons,indent=2))
```

Response:

```
Total number of labels: 124260
Total number of concepts: 124366
Concepts with greater than 5 term frequency: 12375
{
  "have": 32782,
  "do": 26869,
  "use": 16793,
  "get": 13412,
  "go": 9899,
  "water": 9537,
  "make": 9476,
  ...
  "second": 340,
  "job": 339,
  "chemical": 338,
  "adult": 338,
  "cat": 337,
  "jump": 336,
  "coat": 336,
  ...
  "dyeing": 6,
  "amp hour": 6,
  "molle": 6,
  "rigor mortis": 6,
  "mortis": 6,
  "hydroxide": 6,
  ...
}
```

Aside from getting the concepts for the Outdoors dataset, Listing 13.10 filtered the total dataset to `topcons` which only includes concepts with a frequency greater than five. Filtering will limit noise from terms that don’t appear as often in the corpus. Such noise may include misspellings and rare terms that we don’t want to suggest in an autocomplete scenario, for example.

13.5.2 Getting embeddings

We’re going to perform a complex normalization that will normalize similarly related concepts. But instead of algorithmic normalization (like stemming), we are normalizing to a dense vector space of 768 feature dimensions. Similar to stemming, the purpose of this is to increase **recall** (the percentage of relevant documents successfully returned). But instead of using a stemmer, we’re finding and mapping together closely related concepts. As a reminder, we’re only normalizing noun phrases and verb phrases. Ignoring the other words is similar to stopword removal, but that’s OK, because we want to suggest similar concepts as concisely as possible. We’ll also have a much better representation of the meaning of the kept phrases and their contexts, so in many ways, the surrounding non-noun and non-verb terms are implied.

Now that we have a list of concepts from the last section, we’re going to process them through our SBERT/RoBERTa transformer architecture and model to retrieve the embeddings. This takes quite a while if you don’t have a GPU (about 25 minutes on an average 2019 Macbook Pro), so after we calculate the embeddings the first time, we’ll persist them to a pickle file. A pickle file is a serialized Python object that can be easily stored and loaded to and from disk. If you ever have to re-run the notebook, you can just load the previously created pickle and not waste another half hour re-processing the raw text.

Hyperparameter alert! the term frequency minimum is a hyperparameter, and is set to be greater than five (>= 6) in Listing 13.11 in order to minimize noise from rare terms. We’ll encounter more hyperparameters in other listings in this chapter and the next, especially when we get into fine-tuning. After finishing the rest of the listings in this chapter, we encourage you to come back and change the value of `minimum_frequency`, and see how it alters the results that are retrieved. Indeed you may find a value that’s more suitable and more accurate than what we’ve arrived at here.

Listing 13.11 Retrieving the embeddings of our concept vocabulary.

```
def get_embeddings(concepts,minimum_frequency,load_from_cache=True):
  phrases = [key for (key,tf) in concepts.items() if tf>=minimum_frequency]
  if not load_from_cache:
      embeddings = stsb.encode(phrases, convert_to_tensor=True)
      with open('data/outdoors_embeddings.pickle','wb') as fd:
          pickle.dump(embeddings,fd)
  else:
      with open('data/outdoors_embeddings.pickle','rb') as fd:
          embeddings = pickle.load(fd)
  return phrases,embeddings

minimum_frequency = 6   ❶

#set load_from_cache=False to regenerate the embeddings rather than
➥loading from pickle
phrases,embeddings = get_embeddings(concepts, minimum_frequency,
➥load_from_cache=True)

print('Number of embeddings:',len(embeddings))
print('Dimensions per embedding:',len(embeddings[0]))
```

❶ We are ignoring terms that occur less than this number of times in the entire corpus. Lowering this threshold may lower precision, and raising it may lower recall.

Response:

```
Number of embeddings: 12375
Dimensions per embedding: 768
```

From Listing 13.11 you can see that one embedding was generated from each of our 12375 concepts. All embeddings have the same dimensionality from the same dense vector space and can therefore be directly compared with one another.

Figure 13.6 demonstrates what these embeddings actually look like and how they related to one another when plotted in 3d.

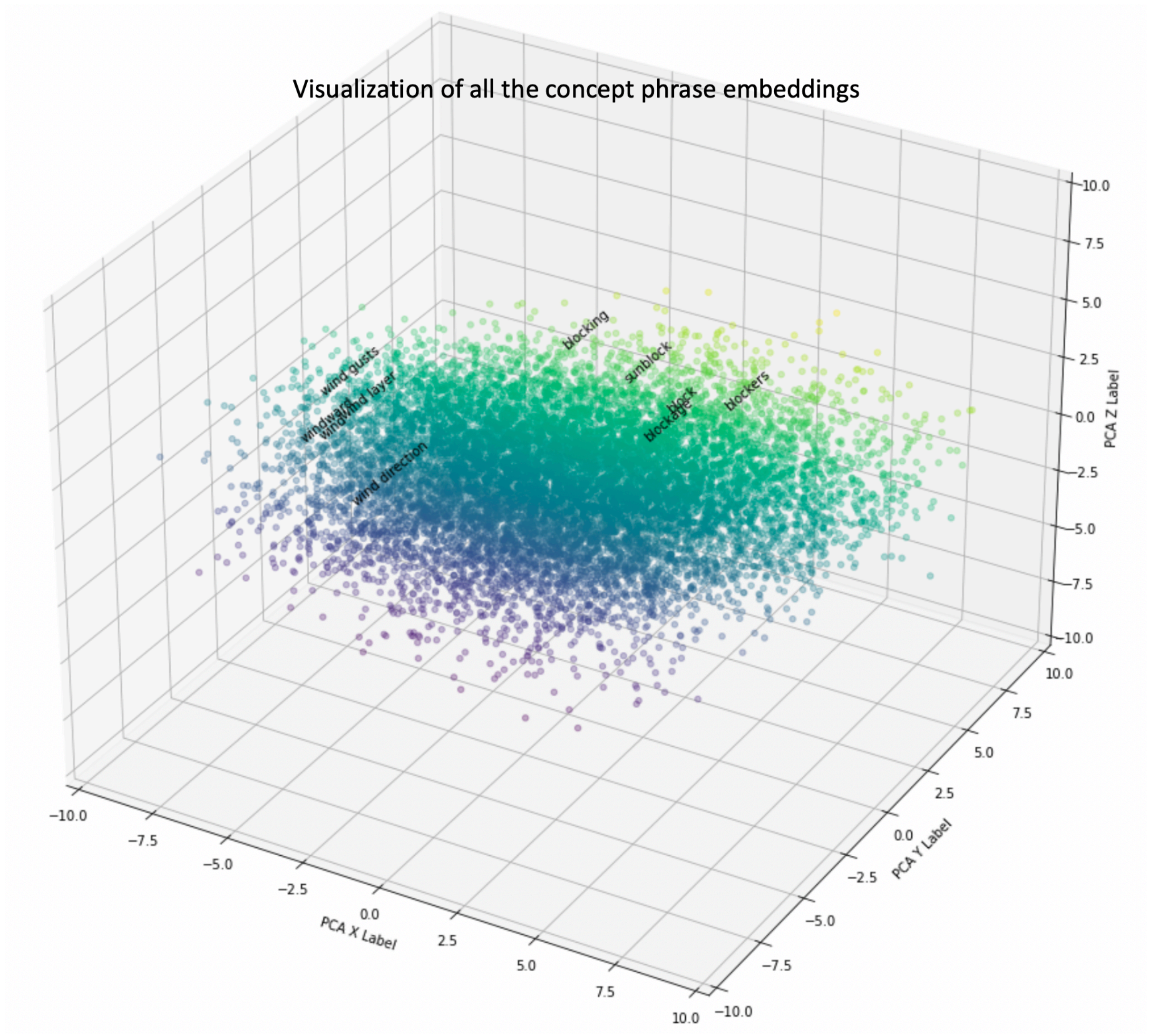


Figure 13.6 The vector space for the concept embeddings mapped to a 3d visualization.

The similarity of some concepts in the figure have been labelled to show neighborhoods of meaning. Concepts related to *wind* and *block* illustrate where they are located in relation to each other in the vector space. We used *dimensionality reduction* to reduce the 768 dimensions for each embedding into 3 dimensions (x,y,z) so they could be easily plotted. Dimensionality reduction is a technique to condense one vector with many features into another vector with fewer features. During this reduction the relationships in the vector space are maintained as much as possible.

SIDEBAR **Dimensionality Reduction loses context.**

A lot of context is lost when performing dimensionality reduction, so the visualization in Figure 13.6 is only presented to give you an intuition of the vector space and concept similarity, not to suggest that reducing to three dimensions is an ideal way to represent the concepts.

With the embeddings calculated from Listing 13.11, we can now perform a massive comparison to see which terms are more closely related to one another. We will do this by calculating cosine similarity for each embedding related to every other embedding. Note that we're limiting the number of embeddings that we're comparing in this example. This is because the high number of calculations that are required to be performed might melt your laptop if you're not careful. If you're not sure what I mean, let's do some quick math. Each embedding has a dimensionality of 768 floating point values. Comparing the top 500 embeddings all against each another results in 500×500×768==192,000,000 floating point calculations. Were we to compare the full list of 12375 embeddings, that would be 12375x12375x768==117,612,000,000 floating point calculations. Not only would this be slow to process, it would take a very large amount of memory.

Listing 13.12 Performs a brute force comparison of the top 505 concepts, to assess how similarities scores are distributed.

Listing 13.12 Explore similarity scores from the head of the vocabulary.

```
similarities = STutil.pytorch_cos_sim(embeddings[0:505],
➥embeddings[0:505])  ❶

a_phrases = []
b_phrases = []
scores = []
for a in range(len(similarities)-1):
    for b in range(a+1, len(similarities)):
        a_phrases.append(phrases[a])
        b_phrases.append(phrases[b])
        scores.append(float(similarities[a][b]))  ❷

comparisons = pandas.DataFrame({"phrase a":a_phrases,"phrase b":b_phrases,
➥"score":scores,"name":"similarity"})
comparisons = comparisons.sort_values(by=["score"], ascending=False,
➥ignore_index=True)  ❸
```

❶ Find the pairs with the highest cosine similarity scores

❷ Appends the similarities for phrase a and phrase b to the list

❸ Sorts the phrase pairs from most to least similar

Response:

```
    phrase a    phrase b    score
0   protect     protection  0.928150
1   climbing    climber     0.923570
2   camp        camping     0.878894
3   climb       climbing    0.833662
4   something   someone     0.821081
5   hike        hiking      0.815187
6   people      person      0.784663
7   climb       climber     0.782961
8   go          leave       0.770643
9   keep        stay        0.768612
10  life        live        0.739865
11  trip        travel      0.730623
12  snow        winter      0.719569
13  fire        burn        0.713174
```

From Listing 13.12, the `comparisons` dataframe now holds a sorted list of all phrases compared to one another, with the most similar being `protect` and `protection` with a cosine similarity of 0.928.

Now that we've calculated the similarities, let's take a look at the distribution of the resulting similarity scores. Seeing the distribution in Figure 13.7 is helpful in understanding the percentage of concepts that are most similar to one another. You see that very few comparisons have a similarity score greater than 0.6, and the vast majority have similarity scores less than that.

Note that the index of *505* is arbitrary and can be changed to larger values for more data to visualize. Remember from what we learned in Listing 13.7: Using `n` concepts yields a tensor of [*n*,*n*] in shape. This yields a total of `505*505 = 255025` similarities for the example in Listing 13.12. Listing 13.13 plots the distribution of the top 505 concept comparison similarity scores.

Listing 13.13 The distribution of similarities.

```
candidate_synonyms = comparisons[comparisons["score"]>0.0]
{
    ggplot(candidate_synonyms, aes('name','score')) +
    geom_violin(color='blue')
}
```

The output of Listing 13.13 is shown in Figure 13.7

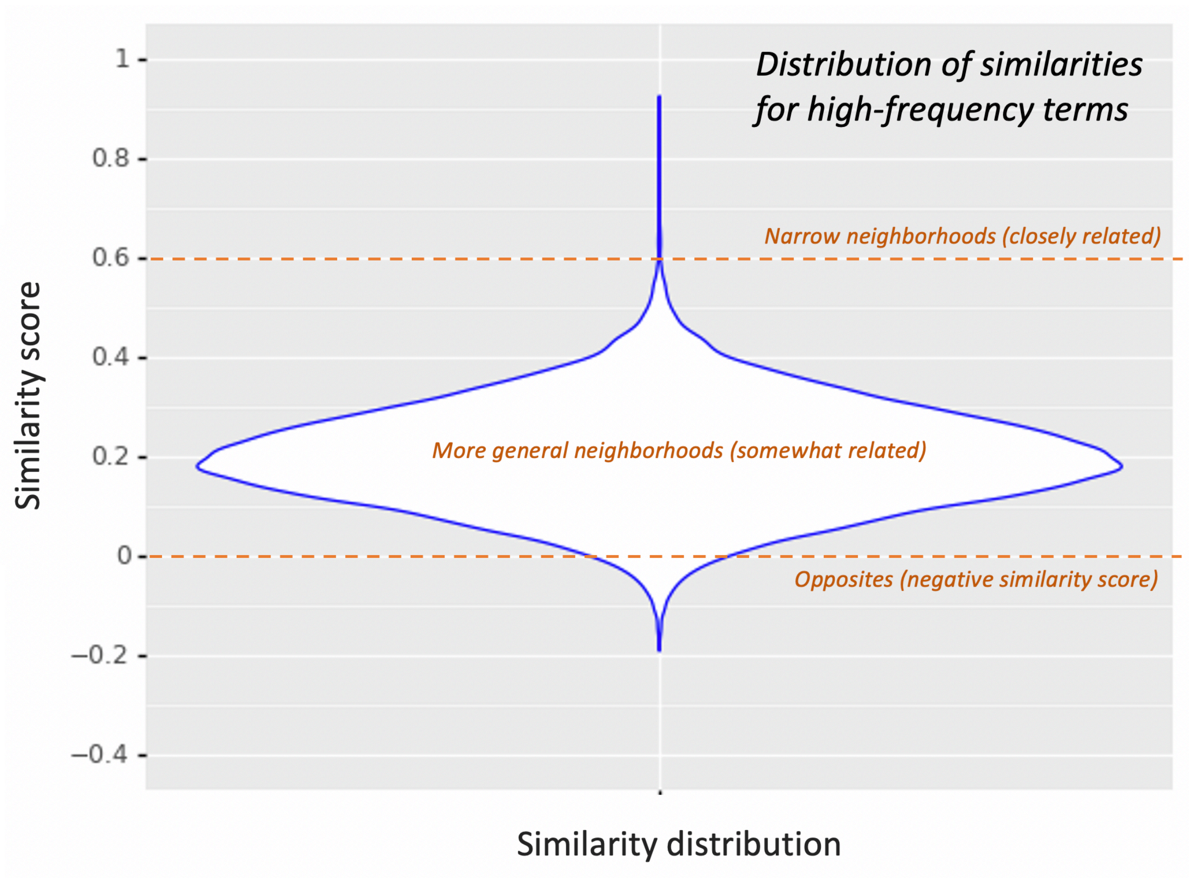


Figure 13.7 Distribution of how the top 505 concepts score with a cosine similarity when compared with each other. Note that very few comparisons result in a score higher than 0.6, and most scores are less than 0.4 (a very low confidence).

We plot the distribution of scores so we can assess them and use our intuition for choosing a baseline similarity threshold at query time (used later in Listing 13.15).

The visualization in Figure 13.7 is very promising. Since most concepts are noted as dissimilar, we can reliably choose a high enough number as a threshold of quality suggestions (such as 0.6 in this example). Remember, when we're performing an autocomplete during search, we're only interested in seeing the top five to ten suggested terms. So this distribution shows us that we can do that reliably.

13.5.3 Approximate Nearest-Neighbor search

Before implementing the working autocomplete, we have one more imporant problem to solve first. The problem is that at query time, we ideally don't want to compare our search terms to all 12,375 other terms. That would be inefficient and slow due to the dimensionality and computational overhead as we witnessed when using the `STutil.pytorch_cos_sim` method. Even if we were willing to caclulate cosine similarities for all of our documents, this will just get slower and slower as we scale to millions of documents, so we ideally would only score documents which have a high chance of being similar.

We can accomplish this goal by performing what is known as an *approximate-nearest-neighbor* (ANN) search. Using ANN search will efficiently give us the most closely related concepts when given a term, without the overhead of calculating embedding similarites across the entire dictionary.

ANN search is meant to trade some accuracy in exchange for a logarithmic computational complexity, and also memory and space efficiencies. To implement our ANN search, we will be using an index-time strategy to store searchable content vectors ahead of time in an specialized data structure. Just like our good old friend the inverted index, think of approximate nearest-neighbor search as the "inverted index" for dense vector search.

For our purposes, we will use *Hierarchical Navigable Small World (HNSW) graphs* to index and query our dense vectors. HNSW is described in the abstract of the paper (https://arxiv.org/abs/1603.09320):

_We present a new approach for the approximate K-nearest neighbor search based on navigable small world graphs with controllable hierarchy (Hierarchical NSW, HNSW)…

Hierarchical NSW incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements._

What this means, is that HNSW will cluster similar vectors together as it builds the index. Navigable Small World graphs work by organizing data into neighborhoods, and connecting the neighborhoods with one another with probable relationships. When a dense vector is being indexed, the most appropriate neighborhood and its potential connections are identified, and stored in the graph data structure.

SIDEBAR **Different ANN Approaches**

In this chapter, we implement the HNSW algorithm for ANN search. HNSW provides a great balance between recall and query throughput, and is currently (as of writing) among the most popular ANN approaches. However, many other ANN approaches exist, including much simpler techniques like Locality Sensitive Hashing (LSH). LSH breaks the vector space into hash buckets (representing neighborhoods in the vector space) and encodes (hashes) each dense vector into one of those buckets. While recall is typically much higher for HNSW versus LSH, HNSW is dependent upon your data to generate the neighborhoods and the neighborhoods can shift over time to better fit your data, whereas LSH neighborhoods (hashes) are generated in a data-independent way, which can better meet some use cases requiring a priori sharding in distributed systems. It may be worthwhile for you to look into different ANN algorithms to find the one that best suit your application.

When an HNSW search is initiated using a dense vector query, it finds the best cluster entry point for the query, and searches for the closest neighbors. There are many other optimization techniques that HNSW implements, and we encourage you to read the paper if you want to learn more.

13.5.4 Approximate Nearest-Neighbor index implementation

For our approximate nearest neighbor search implementation we're going to use a library called Non-Metric Space Library (NMSLIB). This library includes a canonical implementation of the *HNSW* algorithm.

We've chosen this library because not only is it fast, it's easy to use and requires very little code. We will also point out that Lucene recently introduced a new feature in version 9 that includes a dense vector field type and native HNSW calculations. This is not readily available in Solr or Elasticsearch, but an implementation of HNSW is available in engines such as Vespa.ai, Weaviate, Milvus, and others.

NMSLIB is robust and well tested, and is used by many teams for ANN applications. NMSLIB is also more appropriate to show the simplicity of approximate nearest neighbor search without getting into the details of the implementation. There are many other ANN libraries available, and I encourage you to investigate some of them listed on this excellent benchmarking site: http://ann-benchmarks.com/

As shown in Listing 13.14, to begin using NMSLIB we simply import the library, initialize an index, add all of our embeddings to the index as a batch, and then commit. Autocomplete is an ideal use case when building an index in this way, because the vocabulary is rarely updated.

Even though this and other libraries may suffer from write-time performance in certain situations, this won't impact our read-heavy autocomplete application. From a practical standpoint, we can update our index as an evening or weekend offline job, and roll out to production when appropriate. Listing 13.14 Creates our HNSW index from all 12375 embeddings and then performs an example search for concepts similar to the term 'bag'.

Listing 13.14 Approximate nearest neighbor search using NMSLIB.

```
import nmslib
index = nmslib.init(method='hnsw', space='cosinesimil')   ❶
index.addDataPointBatch(embeddings)  ❷
index.createIndex(print_progress=True)  ❸

# Example query for the new index.  The embedding in index 25 is
➥the term 'bag'
ids, distances = index.knnQuery(embeddings[25], k=10)  ❹
matches = [labels[phrases[idx]].lower() for idx in ids]  ❺
print(matches)
```

❶ Initialize a new index, using a HNSW in the cosine similarity space

❷ Adding all the embeddings is easy! The ID for each embedding is its index in the dictionary terms list

❸ This commits the index to memory. This must be done before you can query for nearest neighbors.

❹ Get the top `k` nearest neighbors (10 in this case). The 25th embedding is the term 'bag' in our dictionary

❺ Lookup the label for each term. Labels have not been lemmatized, which looks nicer during display. For example, the term 'summer months' is the label for the phrase 'summer month'.

Response:

```
['bag', 'bag ratings', 'bag cover', 'bag liner', 'garbage bags', 'wag bags',
➥'bag cooking', 'airbag', 'paper bag', 'tea bags']
```

Once the index is created and committed, we run a small example comparing the term `bag` and seeing what comes back. Interestingly, all of these terms are hyponyms, which reveals another ideal result. We are interested in suggesting more precise terms to the user during autocomplete time. This has a higher likelihood of giving the user the chance to select the term most closely associated with a particular information need.

With our index in place and our example confirmed, we can now construct a straightforward query method that accepts any term, and returns the top suggestions. The technique behind SBERT has been trained to provide similar terms from a given set of tokens. Importantly, this method accepts any query whether or not it's already in our dictionary. We first take the query, and retrieve the embeddings by passing the query through the same SBERT encoder. With these embeddings we then get the nearest neighbors from the index. If the similarity score is greater than 0.75 we count it as a match and include that as a suggestion.

With this method we can then try to get suggestions for complete terms such as `mountain hike`, as well as prefixes such as `dehyd`. Listing 13.15 shows our autocomplete `semantic_suggest` method implementation, which performs an approximate nearest neighbor search for concepts.

We will use the threshold dist>0.75 to only return similar terms for which we see a high confidence in similarity.

SIDEBAR **Chose a good similarity threshold**

We arrived at the 0.75 threshold by looking at the distribution from Figure 13.7, and going with our gut. This should be further tuned by looking at actual examples for actual user queries.

Our query might not be in the dictionary, but that's OK! We can get the embeddings on demand.

Note that this may be a CPU bottleneck in production. It is advised to measure the throughput at scale, and add hardware accordingly.

Listing 13.15 Our autocomplete method encodes a query and returns the K-nearest-neighbor concepts.

```
def semantic_suggest(query,k=20):   ❶
    matches = []
    embeddings = stsb.encode([query], convert_to_tensor=True)   ❷
    ids, distances = index.knnQuery(embeddings[0], k=k)
    for i in range(len(ids)):
        text = phrases[ids[i]]
        dist = 1.0-distances[i]
        if dist>0.75:   ❸
            matches.append((text,dist))
    if not len(matches):
        matches.append((phrases[ids[1]],1.0-distances[1]))   ❹
    return matches
```

❶ We set k=20 for illustration. In a production application this would likely be 5 to 10.

❷ Get the embeddings for the query

❸ We're only returning the terms with 0.75 or higher cosine similarity.

❹ No neighbors found! Return just the original term.

Response:

```
Results for: mountain hike

mountain hike   1.0
mountain hiking 0.9756487607955933
mountain trail  0.8470598459243774
mountain guides 0.7870422601699829
mountain terrain    0.7799180746078491
mountain climbing   0.7756213545799255
mountain ridge  0.7680723071098328
winter hikes    0.7541308999061584


Results for: dehyd

dehydrated  0.9417487382888794
dehydration 0.9317409992218018
rehydration 0.852516770362854
dehydrator  0.8514857292175293
hydration   0.8362184166908264
hydrating   0.8358256816864014
rehydrating 0.8222473859786987
hydrated    0.8123505115509033
hydration pack  0.7883822917938232
hydration system    0.7768828868865967
```

We've done it! We can now efficiently serve up an autocomplete based on transformer embeddings and approximate-nearest-neighbor search.

Overall, the accuracy of results for many queries with this model is quite impressive. But beware, it's extremely important to use a labeled data set to calculate accuracy before deploying a solution like this to real customers. When implementing question-answering in chapter 14, we will demonstrate using labeled data in this way for a real accuracy measurement.

Try out some more autocomplete values on your own if you'd like. Which ones work well? Which ones don't?

13.6 Semantic Search with large language model embeddings

Using what we've learned so far, we will now take dense vector search to the next level: we are going to query the document embeddings with the query embeddings as a recall step at search time.

We specifically started with autocomplete as our first implementation, because it was helpful to understand the basics of language similarity. It is essential that you have strong intuitions about why things are similar or dissimilar in a vector space, because otherwise you will endlessly chase recall issues when using language embeddings. To build that intuition we started with matching and scoring basic concepts only a few words in length each.

With that intuition, we will now move on to comparing entire sentences. We're going to do this to perform semantic search for titles, starting with a query and then return a list of documents. Remember that we're searching on the Outdoors Stack Exchange dataset, so the document titles are really the summary of the questions being asked by the contributor. As a bonus, we can actually use the same implementation from the last section to search for question titles that are similar to one another.

This section will be quick, because it is for the most part, a repeat of the encoding and similarity methods from the previous section. In fact, the code in this section is even shorter, since we don't need to extract concepts!

Here are the steps we'll follow:

1. Get the embeddings for all the titles in the outdoors dataset.
2. Create an NMSLIB index with the embeddings.
3. Get the embeddings for a query.
4. Search the NMSLIB index.
5. Show the nearest neighbor titles.

13.6.1 Getting titles and their embeddings

Our NMSLIB index will be made up of title embeddings. We're using the exact same method as we did in the autocomplete example, but instead of transforming concepts now we're transforming the titles of all the questions that the outdoors community has asked. Listing 13.16 shows the process of encoding the titles into embeddings.

Listing 13.16 Encode the titles into embeddings

```
titles = list(filter(None, list(outdoors_dataframe['title'])))   ❶
embeddings = get_embeddings(titles)   ❷

print('Number of embeddings:',len(embeddings))
print('Dimensions per embedding:',len(embeddings[0]))
```

❶ We get the titles for every question in the outdoors corpus.

❷ Get the embeddings for the titles (this takes a little while)

Response:

```
Number of embeddings: 5331
Dimensions per embedding: 768
```

We have encoded 5331 titles into embeddings, and we will plot the title embedding similarity distribution in figure 13.8.

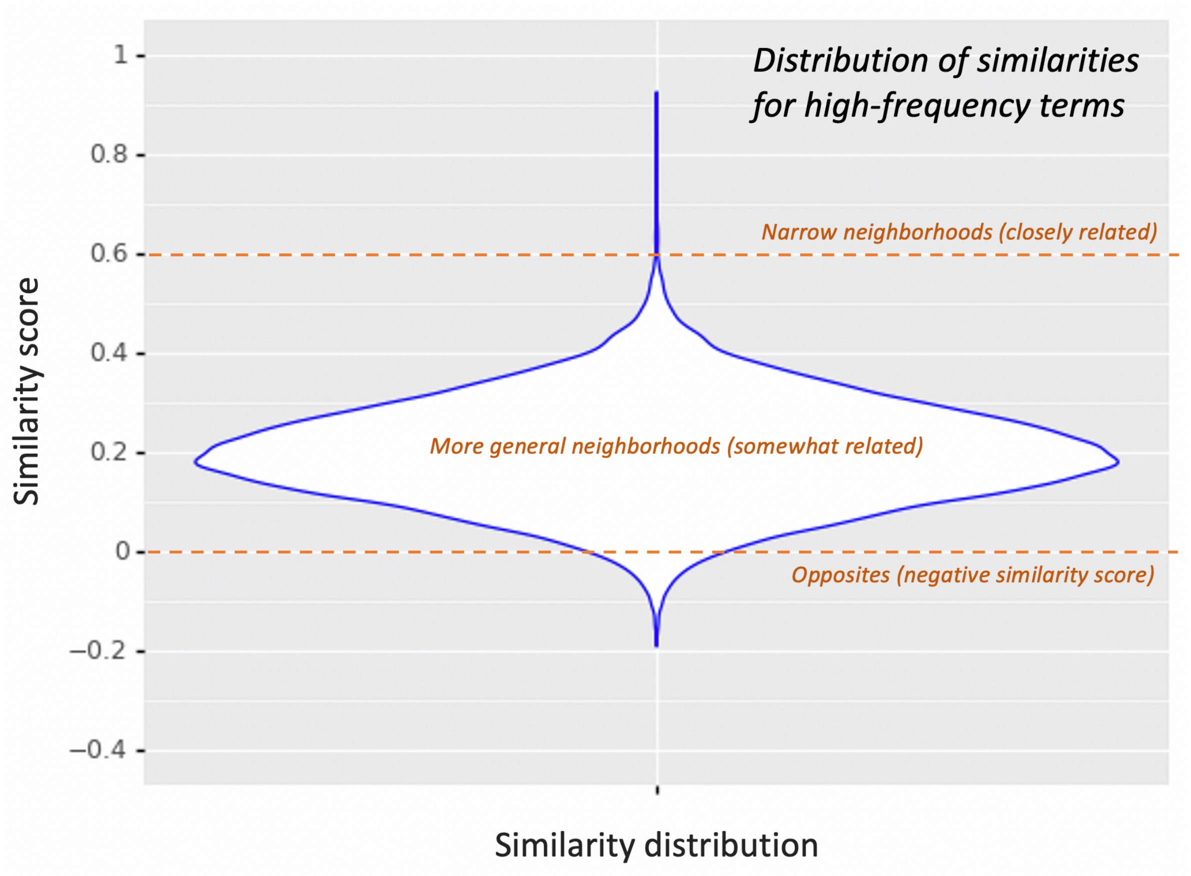


Figure 13.8 The similarities of all the title embeddings to each another

Compare Figure 13.8 to the concept similarity distributions from from Figure 13.7. Note the slightly different shape and score distributions, due to the difference between titles and concepts. Figure 13.7 has a longer 'needle' on top. This is because titles are more specific, and therefore will relate differently than broader noun and verb phrases.

13.6.2 Creating and searching the nearest-neighbor index

Now that we have generated the the embeddings for all the question titles in the corpus, we can easily create the nearest-neighbor index, as shown in Listing 13.17

Listing 13.17 Create the ANN title embeddings index

```
import nmslib
index = nmslib.init(method='hnsw', space='cosinesimil')
index.addDataPointBatch(embeddings)
index.createIndex(print_progress=True)
```

With our newly created index, searching is easy! Shown in Listing 13.18 is the new method `semantic_search`, which implements ANN search for question titles given a query. Note that the this is very similar to that of the `semantic_suggest` from Listing 13.15 that we implemented for autocomplete.

Listing 13.18 Perform a semantic search for titles

```
def semantic_search(query,k=10,minimum_similarity=0.6):   ❶
    matches = []
    embeddings = stsb.encode([query], convert_to_tensor=True)
    ids, distances = index.knnQuery(embeddings[0], k=k)
    for i in range(len(ids)):
        text = titles[ids[i]]
        dist = 1.0-distances[i]
        if dist>minimum_similarity:   ❷
            matches.append((text,dist))
    if not len(matches):
        matches.append((titles[ids[1]],1.0-distances[1]))

    print_labels(query,matches)

semantic_search('mountain hike')   ❸
```

❶ Accepts a query and defaults k to the 10 closest neighbors

❷ Hyperparameter alert! Changing this value to something other than 0.6 alters the recall for the search.

❸ Perform an ANN Search against the titles index!

Response

```
Results for: mountain hike
0.725 | How is elevation gain and change measured for hiking trails?
0.706 | Hints for hiking the west highland way
0.698 | Fitness for hiking to Everest base camp
0.697 | Which altitude profile and height is optimal for Everesting by
➥hiking?
0.678 | How to prepare on hiking routes?
0.678 | Long distance hiking trail markings in North America or
➥parts thereof
0.675 | How far is a reasonable distance for someone to hike on
➥their first trip?
0.668 | How to plan a day hike
0.666 | How do I Plan a Hiking Trip to Rocky Mountain National Park, CO
0.665 | Is there special etiquette for hiking the Appalachian Trail
➥(AT) during AT Season
```

Yay! That was easy. Now let’s take a good moment and reflect on these results. Are they all relevant? Yes - they are all absolutely questions related to the query `mountain hike`. BUT, and this is very important, are they the MOST relevant documents? We don’t know! The reason we don’t know, is that `mountain hike` does not provide much context at all. So while the titles are all semantically similar to the query, we don’t have enough information to know if they are the documents we should surface for the user.

That said, it is clear that this embedding-based approach to search brings interesting new capabilities to our matching and ranking tool box, providing the ability to conceptually relate results. Whether those results are better or not depends on the context, though.

SIDEBAR **Reranking results found with dense vector similarity**

Note that in Listing 13.17 we chose the default `minimum_similarity` distance score threshold to be greater than 0.6. Examine the title similarity distributions in Figure 13.8, would you change this number to be something different than 0.6?

We can set `minimum_similarity` to a value lower than 0.6 (for example 0.5) to potentially increase recall, and change `k` to be a value higher than 10 as a rerank window size (for example 250). Then, using this larger resultset you can perform a rerank using cosine similarity with one feature among many in a Learning to Rank reranking step.

While Solr does not provide this as a capability yet, it is possible in some other engines such as Vespa.ai. Using what you learned in Chapter 10, think how would you go about incorporating dense vector similarity into a learning-to-rank model.

And thus we return back to the game of automating relevance tuning. Cosine similarity of embeddings is one of many signals or features in a mature AI-powered search stack. This similarity is a feature that will be used alongside Personalization, Learning-to-Rank, and

Knowledge Graphs, for a robust search experience. The trend is that nearest neighbor dense vector search is rapidly growing in popularity, and will likely eventually supplant BM25 as the foundational retrieval and ranking techniques used when searching unstructured text.

With what you’ve learned in this chapter, you should now be able to do the following with your own content: * Assess and choose an existing fine-tuned transformer encoder model that matches your use case * Encode important text from your documents and add them to an embeddings index * Build an autocomplete pipeline to accept plain text queries and quickly return the most closely related concepts * Add a powerful high-recall Semantic-Search to your product

The technology underlying dense vector search still needs improvement, as embeddings are heavy and slow, and sparse term vectors are much smaller and faster. But, tremendous forward progress continues to be made towards productionizing these dense vector search techniques, and for good reason. Not only does searching on vectors enable better semantic search on text, it also enables cutting edge approaches to image search, question answering, and other more advanced search use cases. In the next chapter, we’ll discuss several of these emerging use cases on the frontier of AI-Powered Search.

Appendix A: Running the code examples

This Appendix covers:

- How this book's source code examples are packaged
- Pulling the AI-Powered Search source code
- Building and running the examples
- Working with Jupyter
- Working with Docker

During your journey through *AI-Powered Search*, we'll walk through a lot of code and running software examples demonstrating the techniques within this book. This appendix will show you how to easily setup and run the accompanying source code so that you can interact and experiment with live, running examples as you work through the material.

A.1 Overall Structure of Code Examples

In order to build an AI-powered search system, it is necessary to integrate many components and libraries. For our core search engine, we will leverage Apache Solr, which internally leverages Apache Zookeeper. For significant data processing and machine learning tasks, we'll leverage systems like Apache Spark and Tensorflow. We'll leverage Python as our primary programming language for code examples, and will thus need to install and manage many Python library dependencies, in addition to other system dependencies (like Java) which several of our systems require. Of course, we also need the ablity to actually execute our code examples and see the results in a user-friendly way, which we'll accomplish through the use of Jupyter notebooks.

Instead of having you install dozens of software libraries and hundreds of dependencies to make this all work, however, we are making this process as easy as possible for readers by packaging all of the examples in this book into Docker containers which are already fully-configured and and ready to use. This means that there is a single prerequisite you must install before running the code examples in this book: Docker.

Docker enables the creation and running of tiny containers - fully-functioning virtual machines that run only a light-weight operating system with all of the needed software and dependencies already installed and configured.

Once all of the services are running, all of the code listings in the book will be available through Jupyter notebooks, which will serve as the interface for walking through and experimenting with the code examples and seeing the resulting outputs.

A.2 Pulling the source code

The source code accompanying this book is available at: github.com/treygrainger/ai-powered-search

To pull the code, either use an installed Git client or open up a terminal into your preferred development folder and run one of the following commands:

```
git clone https://github.com/treygrainger/ai-powered-search.git
```

or

```
git clone git@github.com:treygrainger/ai-powered-search.git
```

You should now have a new folder in your current directory called `ai-powered-search`, which contains all of the source code for the book.

If you do not have Git installed or can't pull the code through one of the above commands, there is also an option to download the source code as a zip through your web browser, which you would then just need to unzip into your development folder.

Feel free to rename or move this `ai-powered-search` folder if you wish; throughout the rest of this book, we'll simply refer to this directory using the variable `$AIPS_HOME`.

A.3 Building and running the code

As previously mentioned, Docker is the one key dependency you must install on your system in order to build and run the AI-Powered Search examples. We will not cover this installation process here as it is system-dependent and changes from time to time, so please visit www.docker.com for download and installation instructions.

Once you have Docker installed, all you need to do is run the following command:

```
cd $AIPS_HOME/docker
docker-compose up
```

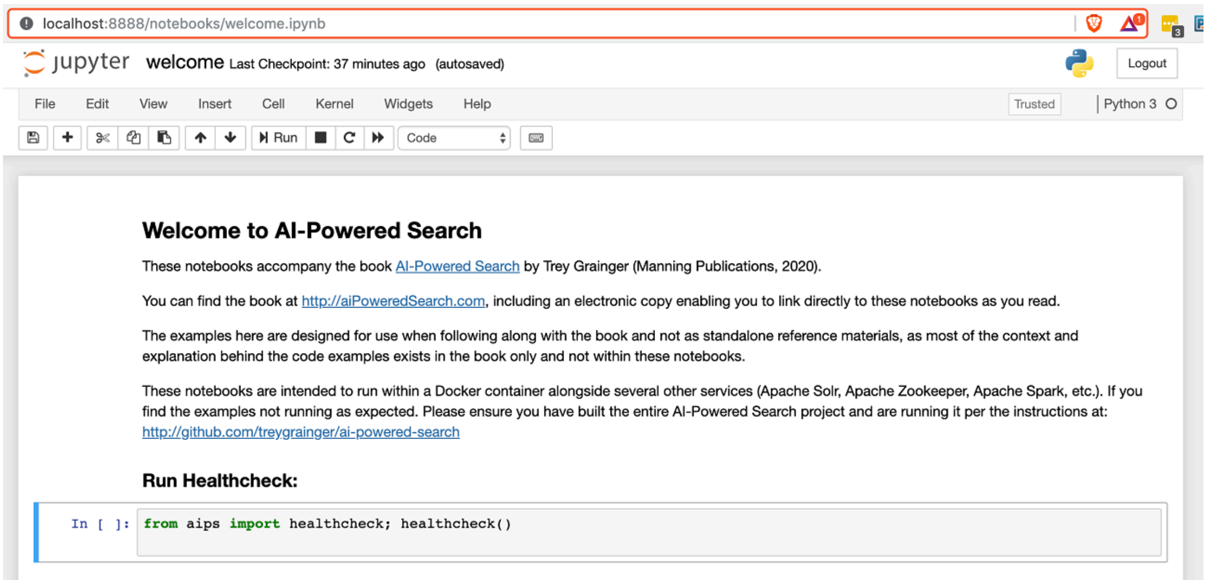
TIP The `docker-compose up` command in the foreground of your console, which allows you to see all logs streaming by in real-time, but which also means your containers will all die if you close the console. If you would like to instead run the containers in the background and continue using your console, you can pass in the `-d` or `--detach` parameter (`docker-compose -d`). If you launch like this, be sure to explicitly run `docker-compose down` when you are finished to stop the containers from running indefinitely in the background consuming resources.

This is helpful for seeing live logs, but also means the containers will be stopped as soon as you close the console. If you'd like to start them up to continue running in the back

This command will take a while to run the first time, as it is pulling in all of the software, operating systems, and dependencies needed to build and run the software accompanying this book.

Once the command finishes, however, you will have all of the necessary services (Jupyter, Solr, Zookeeper, Spark, etc.) running in separate Docker containers. Now, to get started, simply open up your web browser and go to:

```
http://localhost:8888/notebooks/welcome.ipynb
```


localhost:8888/notebooks/welcome.ipynb
jupyter
welcome Last Checkpoint: 37 minutes ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code

Welcome to AI-Powered Search

These notebooks accompany the book AI-Powered Search by Trey Grainger (Manning Publications, 2020).

You can find the book at http://aiPoweredSearch.com, including an electronic copy enabling you to link directly to these notebooks as you read.

The examples here are designed for use when following along with the book and not as standalone reference materials, as most of the context and explanation behind the code examples exists in the book only and not within these notebooks.

These notebooks are intended to run within a Docker container alongside several other services (Apache Solr, Apache Zookeeper, Apache Spark, etc.). If you find the examples not running as expected. Please ensure you have built the entire AI-Powered Search project and are running it per the instructions at: http://github.com/treygrainger/ai-powered-search

Run Healthcheck:

```
In [ ]: from aips import healthcheck; healthcheck()
```

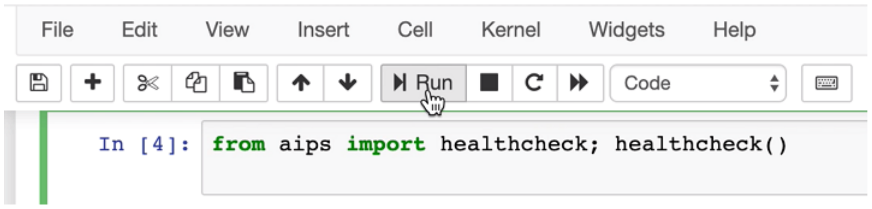
Figure A.1 Welcome screen. Once you see this, the AI-Powered Search containers are built and the Jupyter notebooks are running.

A.4 Working with Jupyter

Once you load the `welcome.ipynb` notebook, you’ll see a few data cells on the screen, including an introduction message, a "health check" script, and a table of contents to various notebooks containing executable examples from the book.

If you’ve never used Jupyter before, it is a tool that lets you mix markup (usually instructions and explanation) and code in your browser, and to edit, run, and interact with the output from the code examples. This makes learning much easier, as you don’t have to use command line tools and can instead interact entirely with ready-to-execute examples with the push of a button.

You’ll notice a toolbar near the top of the screen (below the menu bar), which allows you to interact with the sections of content (called "cells") within the notebook. You can use these to navigate up and down, to stop and restart the notebook, or to execute each cell sequentially using the "Run" button.

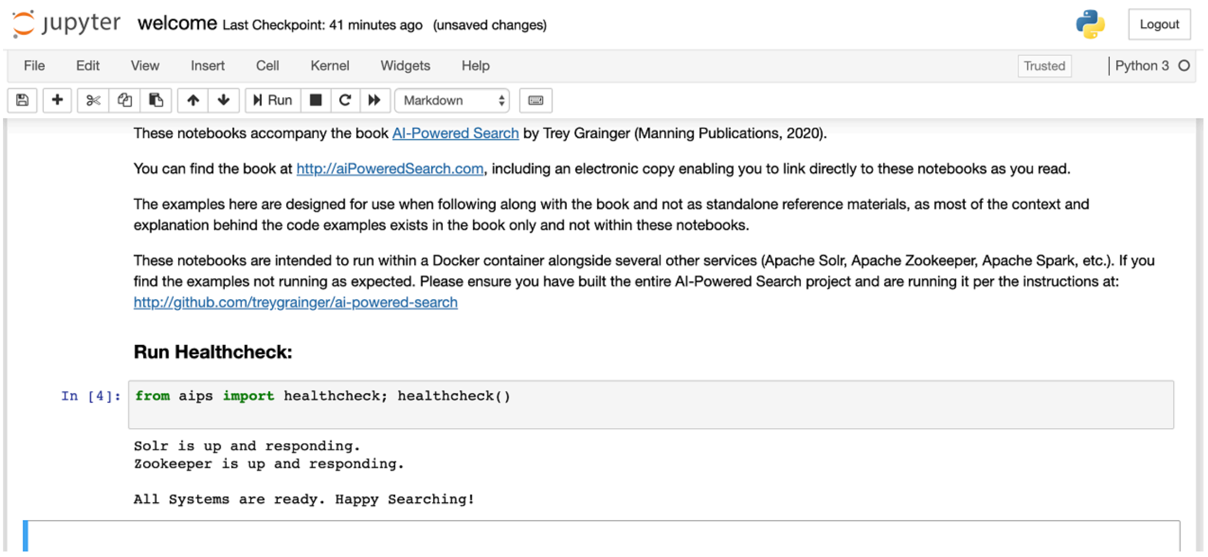


```
In [4]: from aips import healthcheck; healthcheck()
```

Figure A.2 Running code examples. Clicking "Run" in the toolbar will execute any examples in the current cell (if any) and proceed to the next cell.

In Figure A1.2, clicking "Run" while the healthcheck code cell is highlighted will result in the healthcheck executing to confirm that all Docker containers are running and that the services running within them are healthy and responding.

Figure A1.3 shows the response you will see when everything is running as expected.


jupyter
welcome Last Checkpoint: 41 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Markdown

These notebooks accompany the book AI-Powered Search by Trey Grainger (Manning Publications, 2020).

You can find the book at http://aiPoweredSearch.com, including an electronic copy enabling you to link directly to these notebooks as you read.

The examples here are designed for use when following along with the book and not as standalone reference materials, as most of the context and explanation behind the code examples exists in the book only and not within these notebooks.

These notebooks are intended to run within a Docker container alongside several other services (Apache Solr, Apache Zookeeper, Apache Spark, etc.). If you find the examples not running as expected. Please ensure you have built the entire AI-Powered Search project and are running it per the instructions at: http://github.com/treygrainger/ai-powered-search

Run Healthcheck:

```
In [4]: from aips import healthcheck; healthcheck()
```

```
Solr is up and responding.
Zookeeper is up and responding.

All Systems are ready. Happy Searching!
```

Figure A.3 Healthcheck Success. You should see this message if everything is running correctly.

At this point you can scroll down to the table of contents and proceed through the notebooks for each chapter. Of course, since the explanation behind the examples is contained within the book, you’ll probably prefer to work through the examples as you’re reading through the book so that you have the appropriate background understanding when running them. The Jupyter notebooks are not intended to be stand-alone examples, so you’ll probably want to keep the book close by to provide context.

All Jupyter notebooks are designed to be independently idempotent. This means that, while all steps in a notebook need to be executed in order to guarantee a successful result, that you can always start back over at the beginning of any notebook and it will "reset" to the expected results necessary to make the following steps succeed. If ever you experience errors in a notebook, just go back to the first cell on the page and run through the whole notebook again!

A.5 Working with Docker

While everything in the previous sections should work as expected, it’s of course possible you could run into problems along the way. The most likely challenge you’ll face is for one of your Docker containers, or the service running inside of it, to fail. It’s also possible, if you’re making changes to underlying data or config in one of the services, for example, that you could put it in a bad state.

When this happens, you can always tear down your containers and start over. To do this, just run:

```
cd $AIPS_HOME/docker
docker-compose down && docker-compose up
```

Keep in mind that if you’re doing anything on your cluster *other* than running through the examples, that any work you’ve done will be lost. In general, the examples are designed to be transient. If you want to preserve your work across container stops and starts, you can modify the `docker-compose.yaml` file to make your data volumes `external` and thus persistent. Please refer the the Docker documentation if you plan to make changes there, as the mechanisms and APIs can change from time to time.

If you ever modify the code examples or your configuration, it is also possible you may need to rebuild your Docker images. When you run `docker-compose up` the first time, it will build your images and start them, but it doesn't rebuild with changes made since the first build. To rebuild everything prior to starting, you can instead run:

```
cd $AIPS_HOME/docker
docker-compose build && docker-compose up
```

This should give you everything you need to run through all of the notebooks and code in *AI-Powered Search*. Happy searching!

Notes

1. John Rupert Firth (1957). "A synopsis of linguistic theory 1930-1955." In Special Volume of the Philological Society. Oxford: Oxford University Press.
2. Marti Hearst. “Automatic Acquisition of Hyponyms from Large Text Corpora”. 1992. The 15th International Conference on Computational Linguistics
3. Trey Grainger et al. “The Semantic Knowledge Graph: A Compact, Auto-Generated Model for Real-Time Traversal and Ranking of any Relationship within a Domain.” 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA) (2016): 420-429.