Graph Algorithms for Data Science

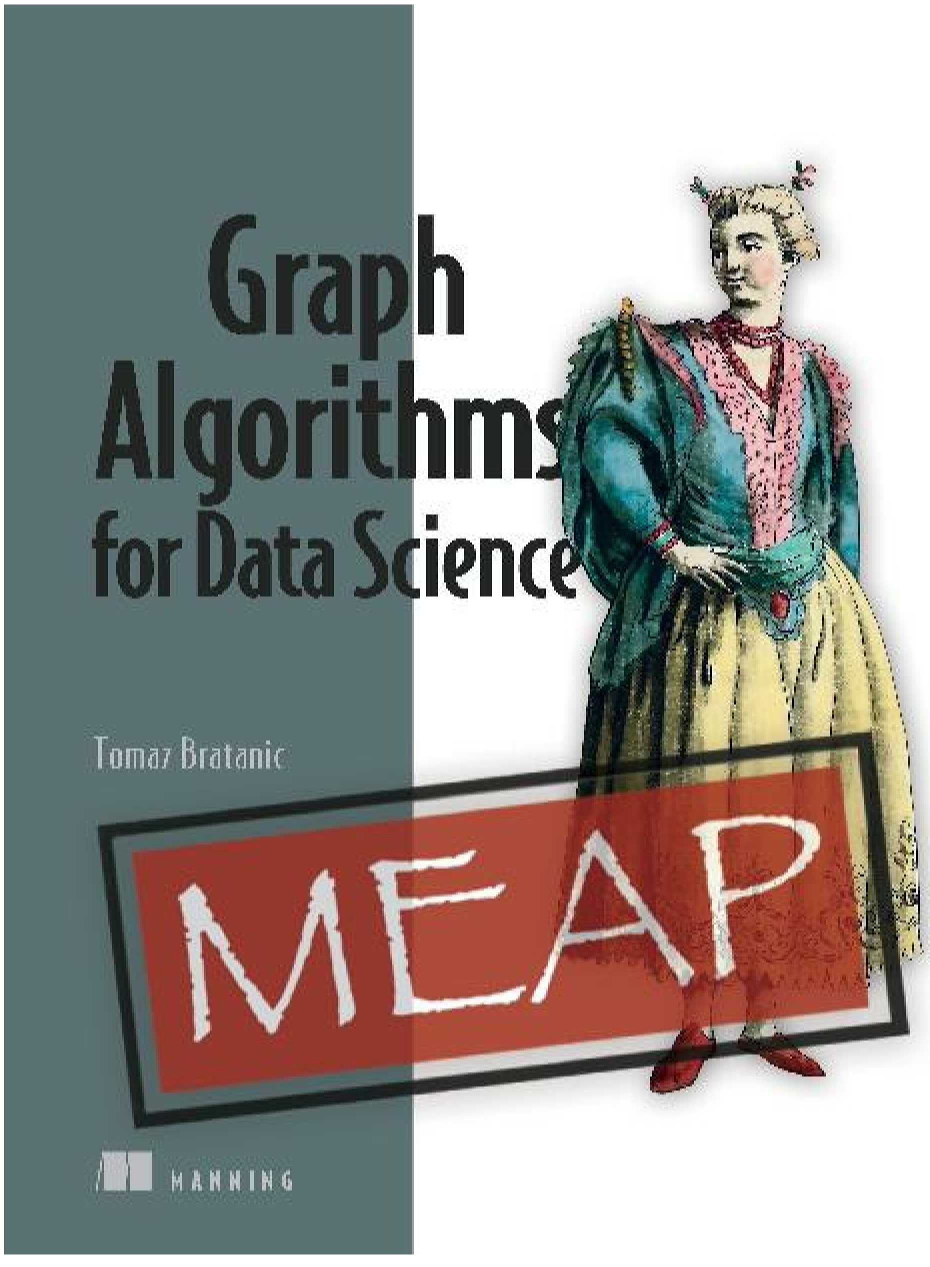
Tomaz Bratanic

MEAP

MANNING

Graph Algorithms for Data Science

Tomaz Bratanic



MANNING

Graph Algorithms for Data Science

Graph Algorithms for Data Science

Tomaz Bratanic

MANNING

MEAP VERSION 7

MANNING PUBLICATIONS

Welcome

Thanks for purchasing the MEAP for *Graph Algorithms for Data Science*.

This book has been written for anyone with no experience with graphs to more experienced graph users to augment their understanding of graph algorithms and their role in the various analysis. With traditional analytics, you are trying to make sense of data points, whereas with graph analytics, you are more focused on analyzing connections between data points. Graph algorithms are designed to investigate those connections between data points and help you explore who is well connected, who has the most influence, how communities form, and more.

When I first started experimenting with graph analytics five years ago, there weren't many tools available, and you would often need to combine a few tools helping you construct a graph and then analyze it. Not only that, but there weren't many tutorials or courses available that would explain the graph analysis workflow in simple terms. For this reason, I've decided to write this book that would minimize the number of tools needed to get started and explain graph algorithms and analysis workflow in basic terms.

I will present the advantages of using a graph database and teach you how to construct a graph from structured and unstructured data using NLP techniques. Along the way, you will learn how to use Cypher query language to manipulate graph structure and extract valuable insights. Next, I will walk you through the typical graph algorithms like PageRank and community detection/clustering algorithms and demonstrate how to use them in practice. The last part of the book will focus on graph machine learning, specifically how to translate graph topology and structure into machine learning model input by using node embedding models and graph neural networks.

Please let me know your thoughts and ideas in the liveBook Discussion forum on what's been written so far. Your feedback is invaluable in improving and increasing the understanding of this book.

Thanks again for your interest and for purchasing the MEAP!

—Tomaž Bratanič

In this book

1 Graphs and network science: An introduction

This chapter covers:

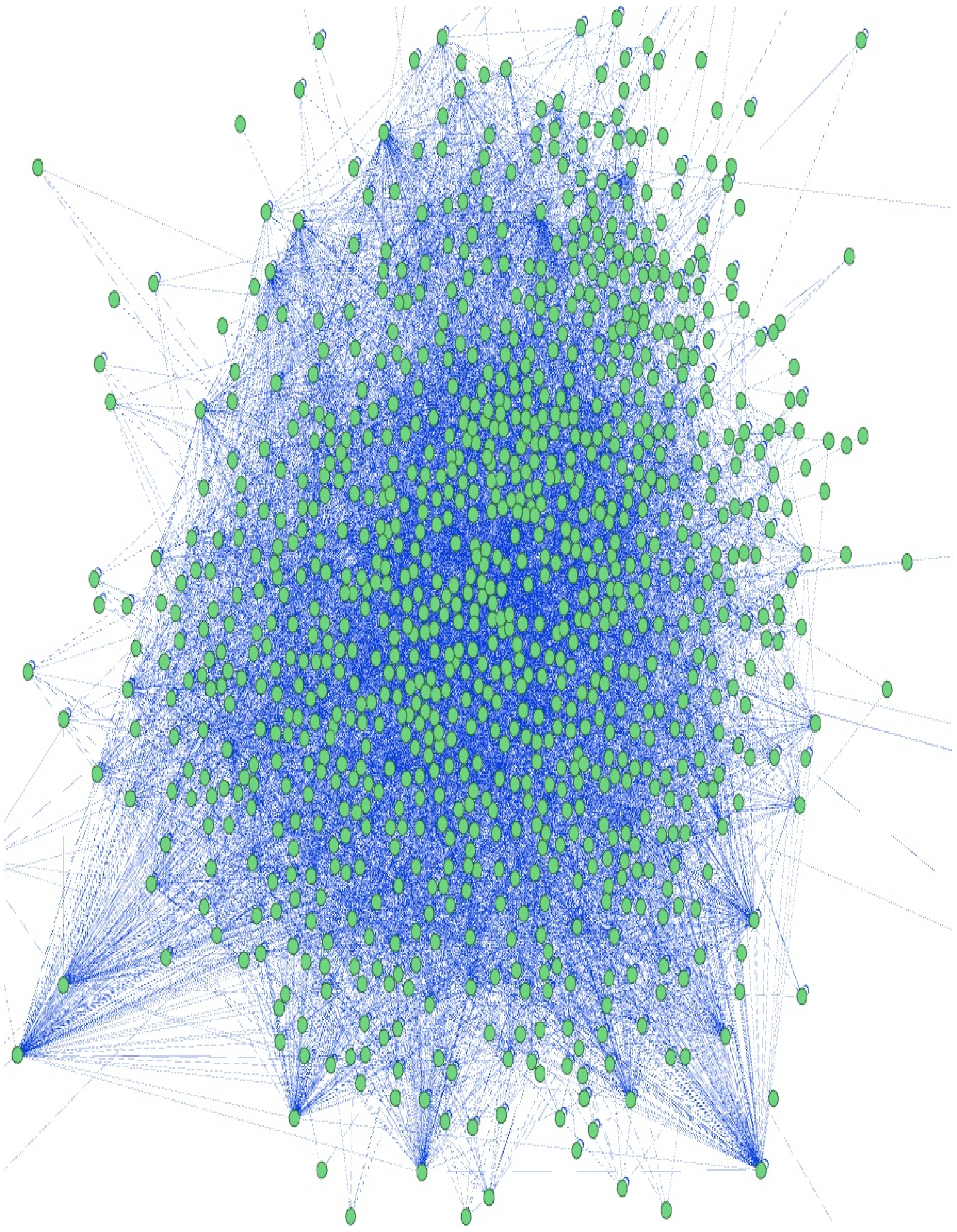
- Introducing networks and graphs
- Introducing node degree characterization of a network
- Spotting graph-shaped tasks
- Introducing machine learning on graphs

Networks are everywhere, and they do matter. First of all, where are these networks? Communication networks are one example. For example, the internet consists of routers. Routers analyze the incoming data, determine the optimal path to the destination, and forward the data to the next device along the route. Another example are the social media platforms. You use those platforms to connect with other users. Most of your connections are local, ranging from your family and friends to coworkers. And then you have some connections from distant friends that can span oceans and continents. When you map all those connections, what you end up with is referred to as a social network.

Figure 1.1. World-wide social network.

Also very interestingly, your biological existence depends on networks. Proteins are called the building blocks of the body. They form the machinery that helps sustain life. Proteins rarely act alone as their functions tend to be regulated. The identification of protein interactions can lead to a better understanding of diseases and the development of drugs and treatments. The process of mapping those interactions results in protein-protein interaction networks, also known as PPI.

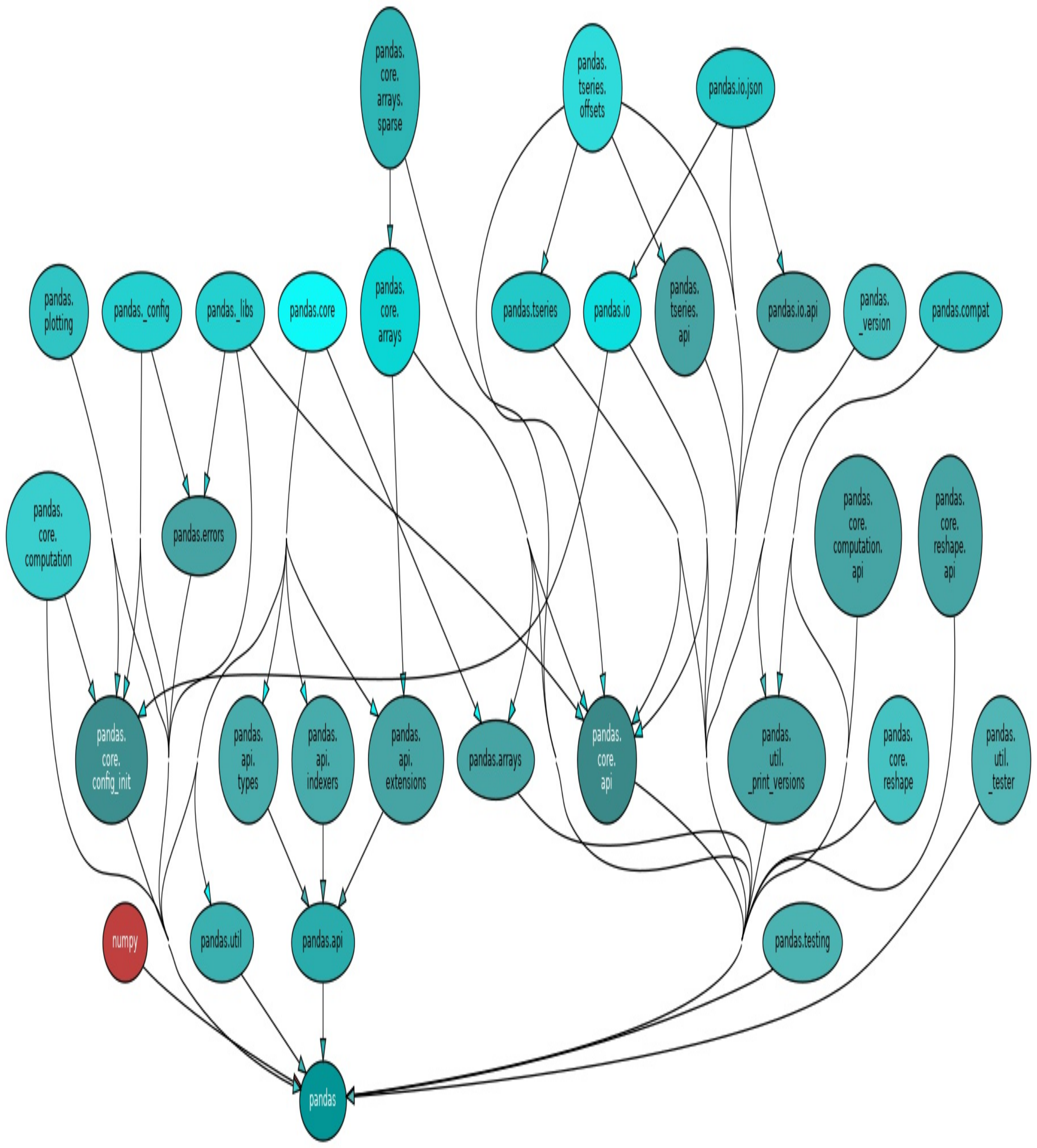
Figure 1.2. Protein-protein interaction network.



One can also map the connections between neurons in a brain as a network, also known as the connectome. Not only is it possible, but scientists from Google AI and a research team from Janelia Research Campus in Virginia have released a detailed network of the neuronal connectivity of an animal as well as the human brain.[Shan Xu et al, 2020]

I hope by now, you have realized that networks are everywhere. You just have to open your eyes to see them. When looking at the world, you can spot networks occurring in politics, markets, art, and even the dependencies between code modules form as a network. If you’ve ever developed any code beyond Hello World, you probably imported other libraries or code from your other modules.

Figure 1.3. Python code module dependency network.



In Figure 1.3, you can observe that the Pandas library has an external dependency on the NumPy library. All the other dependencies are between internal code modules in the Pandas library and are several levels deep.

Now we can move on to the second question. Why do networks matter? The usefulness of networks and the underlying graph theory lies in its ability to model many real-world situations and applications. In Figure 1.1, we visualized a network where the nodes represent people, and the relationships represent friendship links. Instead of treating entities as independent from one another in your analysis, you must realize that we live in a connected world. Using the graph approach to analysis, you consider that the world is connected and, therefore, the entities are often not independent. Your job as a network scientist is to map and understand those connections in real-world scenarios. In the social network example, given the friendship network topology, you can find influencers or leaders. Information about a network's leaders might help you spread your information throughout the whole graph more efficiently. You could also try to find the nodes that would break the network if they were removed. Finding the most critical nodes can help you disrupt or upgrade the resilience of a network. Another example would be to find segments or communities of like-minded people. The information about the segments could be used as input to your recommendation engine or to improve your target advertising persona. Understanding the segments of customers you are dealing with is also very helpful in a business domain. Instead of looking at people's friendship links, you might be interested in their purchasing and product usage behavior patterns. There are so many real-world network applications that it is impossible to squeeze them into a single chapter or even a book.

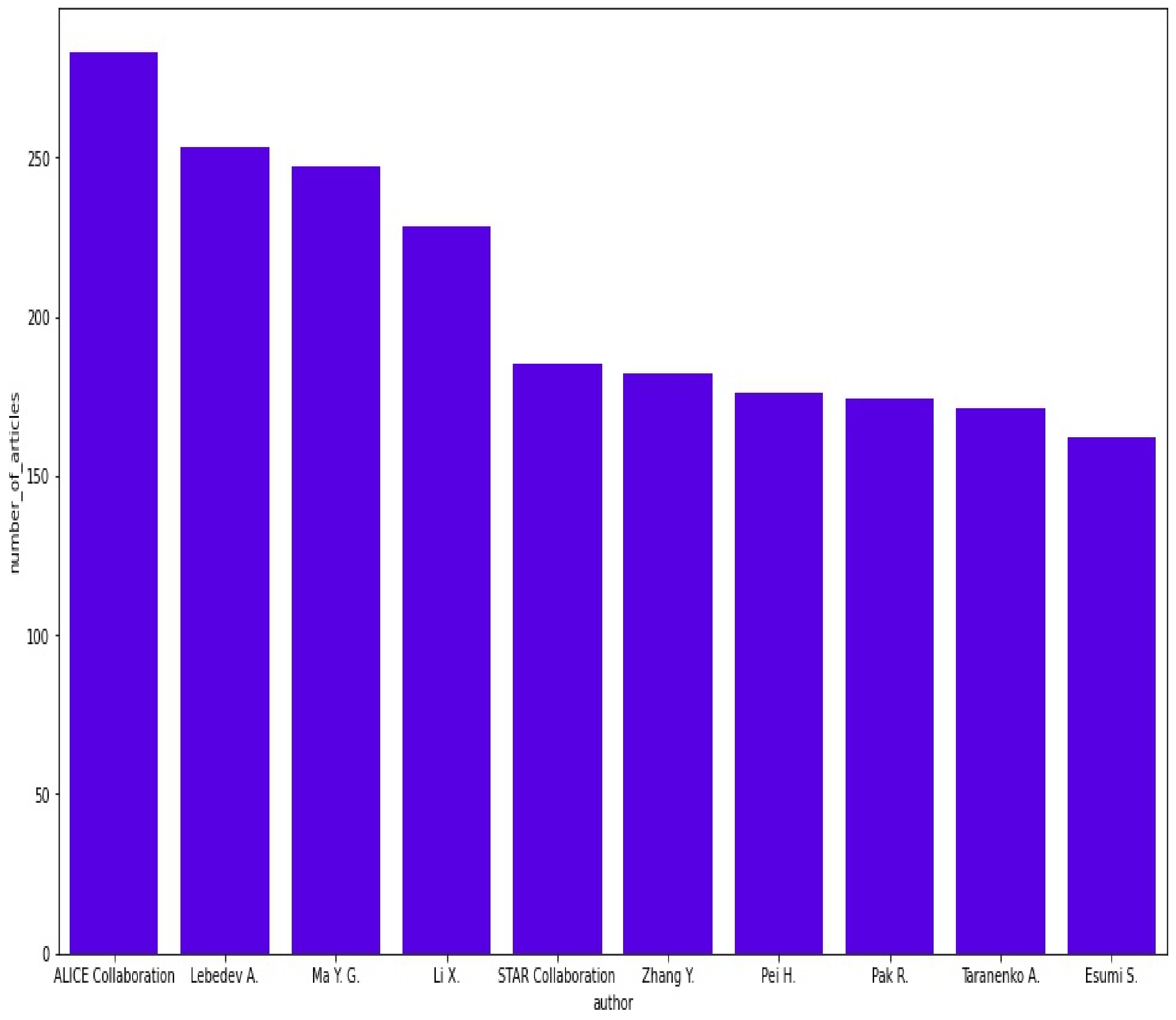
1.1 Introduction to graph theory

Before we delve more into network science and its applications, you will first have to learn a bit of graph theory. First of all, what is the difference between graphs and networks? The term *graph* is used more in mathematical terminology and refers to a general language for describing and analyzing entities and their relations or interactions. On the other hand, the term *network* is used to describe a real-world dataset containing entities and their relationships, such as a social network, a communication network, or a road network. In practice, the distinction between the terms *graph* and *network* is blurred, and the terms are often used interchangeably. In this book, I will try to stick to using the term graph when introducing mathematical theory and concepts and use the term network when describing real-world entities and their relationships.

1.1.1 What is a graph?

Interestingly, if you google images of a graph, you will find similar images like the following:

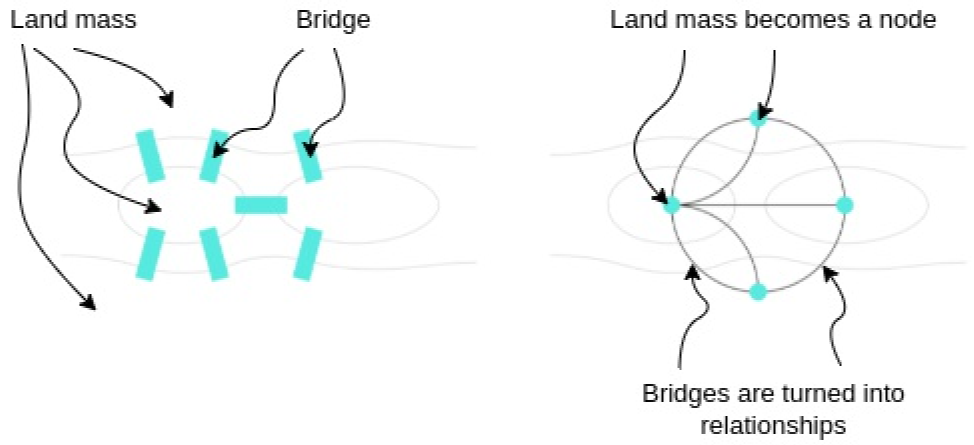
Figure 1.4. A sample bar chart visualization.



Even the first definition of a graph in the Merriam Webster dictionary describes a diagram representing a variable compared to that of one or more other variables. In this book, we will refer to any diagram or a graph that visualizes a series of one or more points, lines, or areas as a chart. Furthermore, in this book, the term graph is reserved to describe a set of nodes (also known as vertices) and relationships (also known as connections, edges, or links).

The history of graph theory can be traced to the 18th century when the Swiss mathematician Leonhard Euler solved the Königsberg bridge problem. [Leonhard Euler, 1736] In that time, seven bridges spanned over the forked river in Königsberg. According to folklore, the puzzle arose of whether a person could take a walk through the town in a way in which they would cross each bridge exactly once. Euler argued that no such path exists. Euler’s solution is simple, once you look at the problem from a graph perspective.

Figure 1.5. A graph visualization of the Königsberg bridge problem.



The problem of finding a way that crosses every bridge exactly once turns into finding a path through the graph that traverses every relationship exactly once. For a walk that crosses every relationship in a graph exactly once to be possible, either none or precisely two nodes can have an odd number of edges attached to them. However, in the Königsberg bridge problem, all nodes have an odd number of relationships, making a walk that crosses every bridge once impossible. In solving this puzzle, Euler started a field that is today known as graph theory.

1.2 How to spot a graph-shaped problem

Before beginning with network analysis, you must ask yourself if a graph-based approach is suitable for your problem. While almost any use-case can be modeled as a graph, a specific set of scenarios is uniquely suited for graph-based analysis.

The first scenario deals with *self-referencing* relationships between entities of the same type. In relational databases, a self-referencing relationship occurs between data points within the same table. One such example would be a Facebook friendship network, where you have a single entity type and multiple relationships between them.

Figure 1.6. Facebook friendship network.

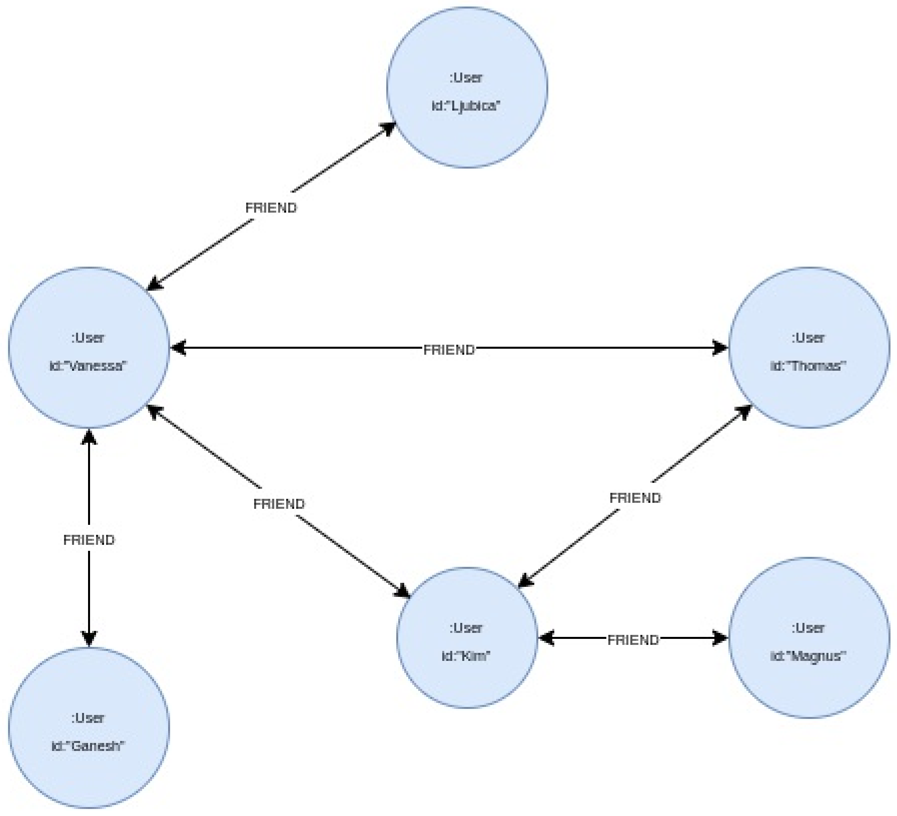


Figure 1.6 represents a small friendship network, where there is a single type of nodes called `User` that can have one or more relationship to other users. Graphs are commonly encouraged to use in analysis when you are dealing with many data joins. As far as I have seen, the most frequent and straightforward demonstration of graphs is the *friend of a friend* query. The

idea behind the friend of a friend query is that you are interested in persons who are two or more hops or joins distant from the original person. Having many joins when dealing with larger datasets might be computationally expensive in traditional relational databases but is quick and easy when storing the data in a graph database due to treating relationships as a first-class data structure. The trick is that instead of computing joins between data entities by scanning foreign key indexes at query runtime, native graph databases store a physical representation of relationships at import. This trick delivers a better query performance when traversing relationships between your data points. There are many scenarios for a graph with self-referencing relationships:

- Detecting social networks influencers
- Analyzing consequences of vulnerabities in a dependency network
- Inspecting organizational hierarchies

Another fairly common graph scenario is discovering paths or routes between entities or locations. Most of you have probably used navigational systems to find the most optimal route for your travels.

Figure 1.7. Transportation network.

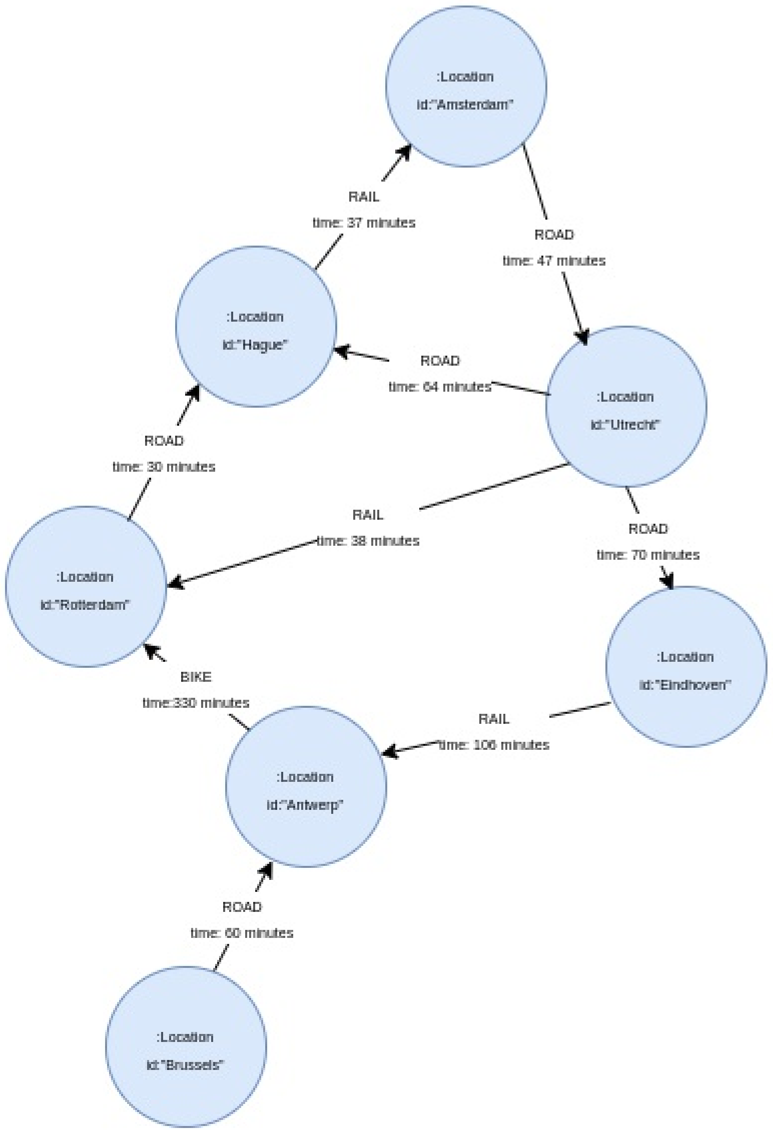


Figure 1.7 visualizes an example transportation network between cities in Belgium and Netherlands. The cities are represented as nodes, while the transportation modes are represented as relationships between cities. For example, you could bike from Antwerp to Rotterdam in about 330 minutes or take the train from Hague to Amsterdam for 37 minutes.

Like mentioned, you could use a transportation network to calculate the most optimal route based on your specifications. The route could be optimized by time, distance, or cost. You could also analyze the network as a whole and try to predict traffic congestions based on the network structure or find critical connections that would disrupt the whole network if, for example, an accident occurred. In a relational database, you would have to hypothesize the order of relationships you must join to find an available path between two entities. In the example in Figure 1.7, there are three relationship options you could choose to traverse. You could hop from one city to another using the road, railroad, or bike network. Another problem you might face with traditional databases is that you don’t know beforehand how many relationships you must traverse to get from node A to node B. Not knowing beforehand precisely which and how many relationships you must traverse could lead to potentially complex and computationally expensive queries. Treating your data as a graph helps you mitigate those two problems.

Finding optimal routes can be applied on following scenarios:

- Logistics and routing
- Infrastructure management
- Finding optimal paths to make new contacts

Another very powerful use-case for graphs is examining indirect or hidden relationships. Consider the following graph:

Figure 1.8. User-item network.

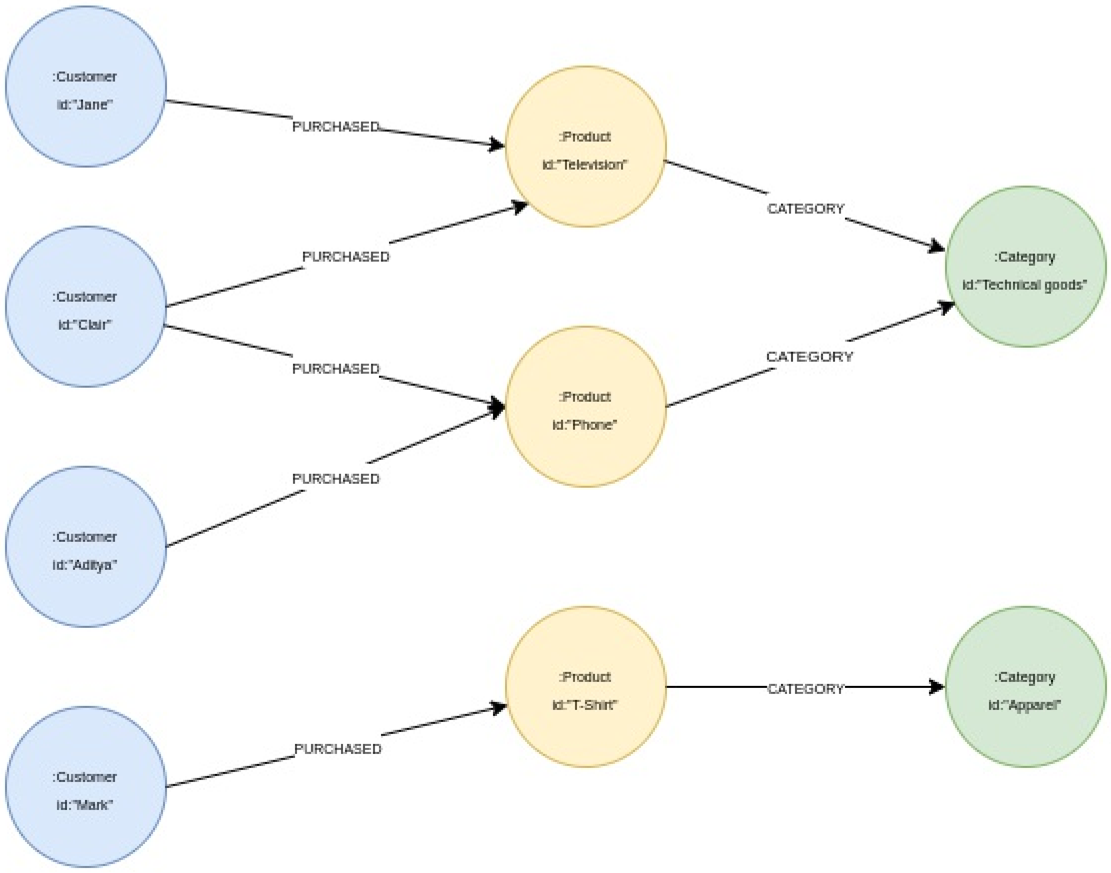


Figure 1.8 represents a network of customers and their purchases. The purchased products can also be categorized into groups like apparel or technical goods. You can observe that both Clair and Aditya bought a phone that falls into the technical goods category.

While there are no direct relationships between customers, you can compare their purchasing patterns and find similar customers. Essentially, you could define segments of customers using this approach. These types of graphs are also frequently used in collaborative filtering recommender systems, where you search for similar customers or products commonly purchased together. You could also examine how many purchases have items that span across many product categories. There are many more scenarios where *user-item* networks come in handy, like movie recommendations on Netflix or song recommendations on Spotify. Essentially, any scenario where a person is rating, purchasing, or voting for an item can be modeled as a user-item network. As another example, think of app store reviews or parliament members voting on laws and resolutions. You could then investigate how similarly members of parliament vote and compare that to their political party association.

1.3 Machine learning on graphs

In the last couple of years, the field of *machine learning on graphs* has taken of. The main idea behind graph machine learning is to manually define or automatically learn the node representations and encode them in the embedding space.

Figure 1.9. Encoding node position into the embedding space. Copyright (c) 2017 Manan Shah, SNAP Group

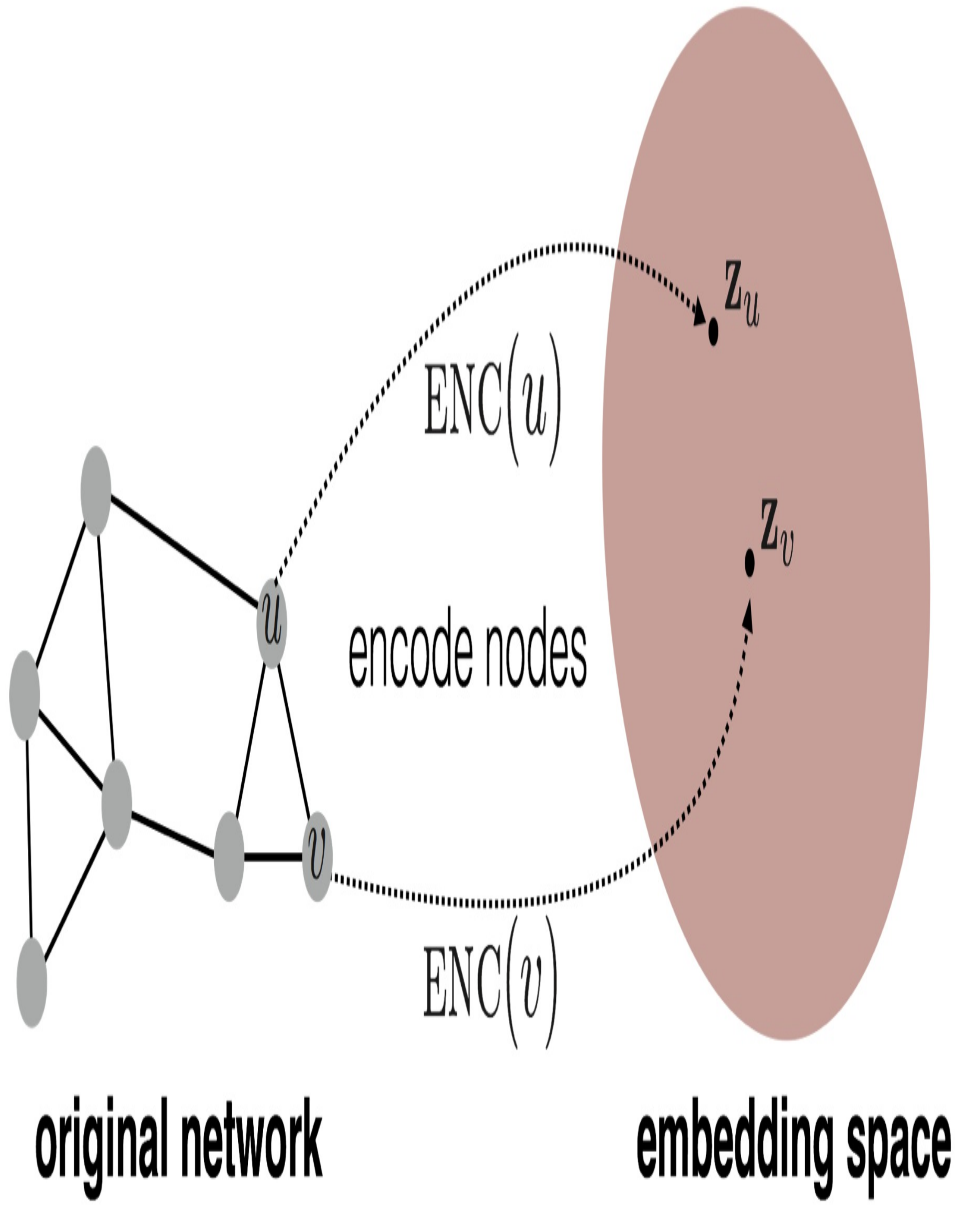
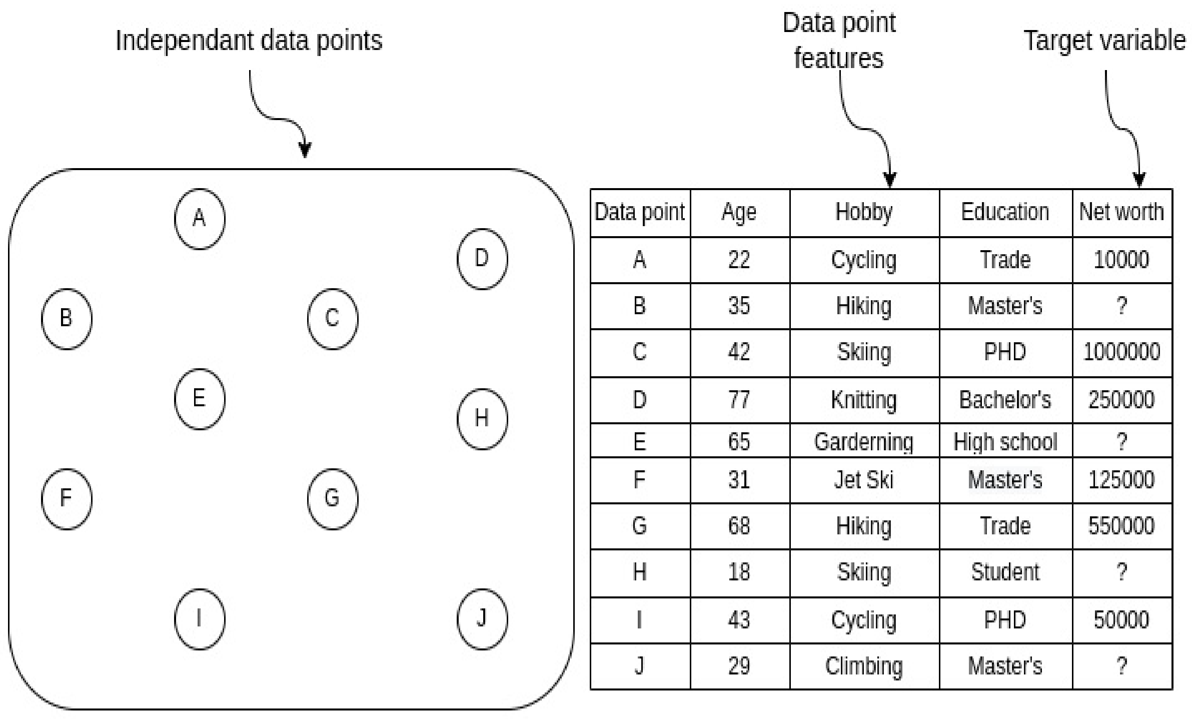


Figure 1.9 demonstrates the idea of encoding nodes in a network into embedding or euclidian space. In a traditional machine learning workflow,

each data point is represented as a vector of integers or floating points. The vectors are then fed into a machine learning model during training and inference. The primary challenge of machine learning on graphs is finding a way to represent or encode network structure as a vector to be easily fed into a machine learning model.

For example, let’s say you have been given the task of predicting a person’s net worth based on their characteristics and attributes. The dataset contains features that describe each data point and the target variable that needs to be predicted. With supervised classification, the training data contains both the features as well as the target variable value, which you can use to train your machine learning or deep neural network model. Once you have trained the model, you can use it to predict the net worth of previously unseen data points and examine how well it works.

Figure 1.10. Traditional machine learning approach, where you treat each data point as independant.



Data point	Age	Hobby	Education	Net worth
A	22	Cycling	Trade	10000
B	35	Hiking	Master's	?
C	42	Skiing	PHD	1000000
D	77	Knitting	Bachelor's	250000
E	65	Garderning	High school	?
F	31	Jet Ski	Master's	125000
G	68	Hiking	Trade	550000
H	18	Skiing	Student	?
I	43	Cycling	PHD	50000
J	29	Climbing	Master's	?

Figure 1.10 shows an example dataset, where each data point is described by features such as age, hobby, and education. The data points are considered

independent, which means they are not related or connected. You would then train a machine learning model based on the available features to predict a person’s net worth.

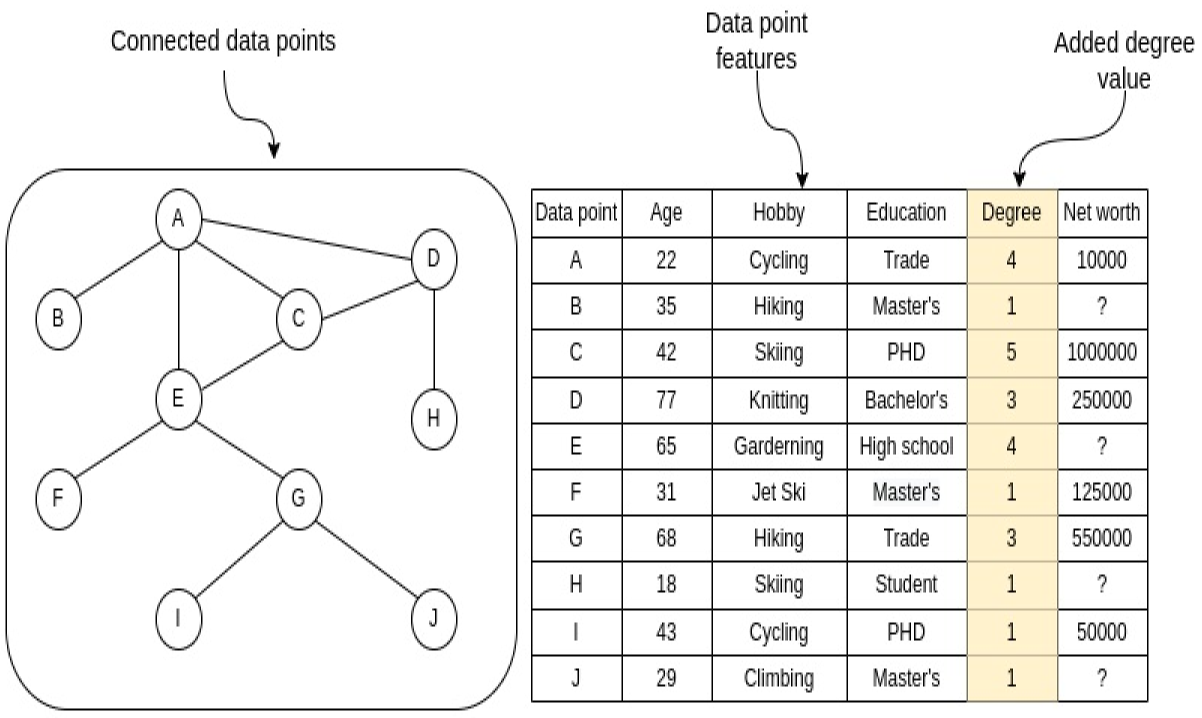
You might know that people are very interconnected, and many people will tell you that networking is a vital part of getting more and better job or other opportunities in life. Since you haven’t encoded any of the networking attributes as data point features, you will skip all that information about connections that might help you more accurately predict a person’s net worth better. Essentially, you treat each data point as independent and ignore its context. Here is where graph machine learning and node representation learning come into play.

First, you need to map those connections or relationships between people. The relationships could be of various types like friends, coworkers, family, or others. Similarly, you could also define the strength of a connection. For example, if you hang out with a friend almost every day, the relationship is stronger than if you only meet once every couple of months. In graph terminology, the strength of the relationship is characterized by its weight. For simplicity’s sake, the following example will not differentiate between relationship types or weights.

The most basic network characteristic of a node is its degree. A node degree is a *local characteristic* of a node and is simply the count of its relationships. A local characteristic of a node does not take into account the whole network. Specifically, node degree considers only direct neighbors, but other local characteristics of a node could, for example, consider friends of a friend. If you consider only direct neighbors, you can say that you are examining a node’s neighborhood one hop away. Likewise, if you would also consider friends of friends, you are considering all nodes that are at most two hops away.

In the example of predicting a person’s net worth, one could argue that a person with more connections will have more opportunities in life, which could correlate with their net worth. Another benefit of having many connections is having access to more experts in their respective fields. Talking to experts and understanding their points of view could help you make better decisions, which could also correlate to your net worth.

Figure 1.11. Add the number of connections each person has as a feature.



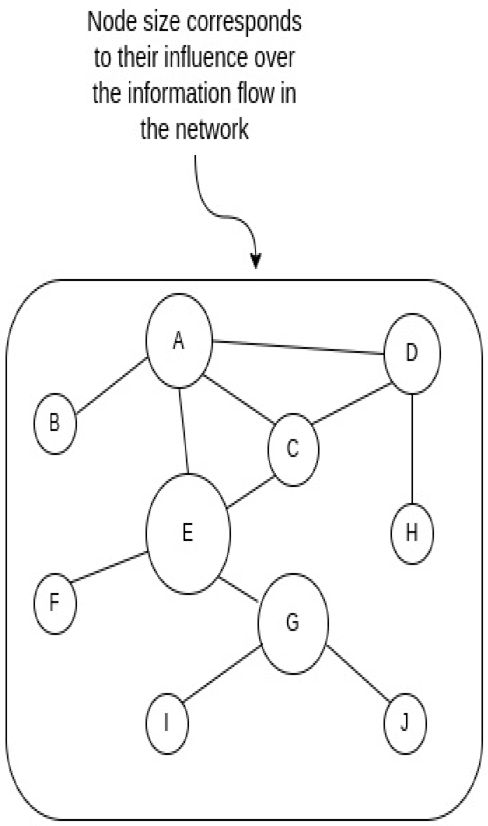
Data point	Age	Hobby	Education	Degree	Net worth
A	22	Cycling	Trade	4	10000
B	35	Hiking	Master's	1	?
C	42	Skiing	PHD	5	1000000
D	77	Knitting	Bachelor's	3	250000
E	65	Gardening	High school	4	?
F	31	Jet Ski	Master's	1	125000
G	68	Hiking	Trade	3	550000
H	18	Skiing	Student	1	?
I	43	Cycling	PHD	1	50000
J	29	Climbing	Master's	1	?

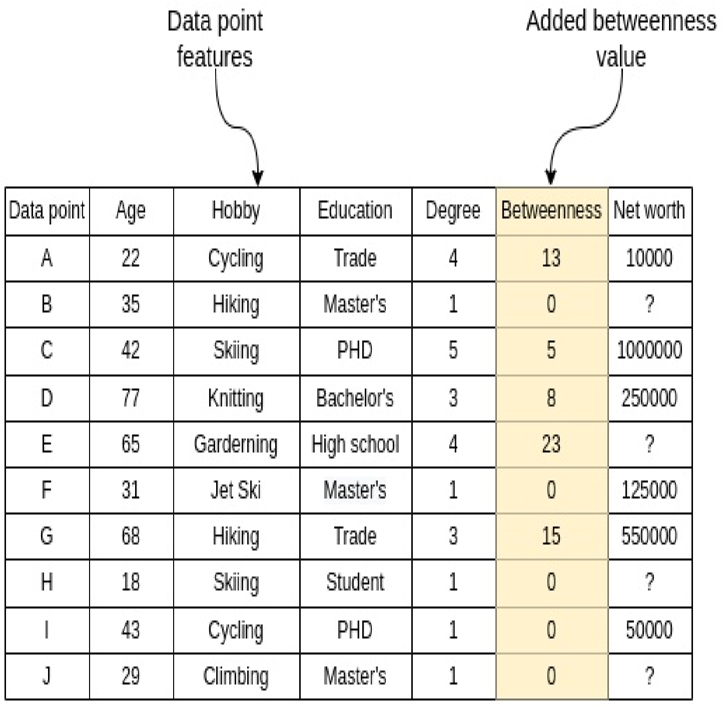
Left hand-side of Figure 1.11 visualizes mapped connections between data points. In this example, you can say that connections represent friendships between persons. We hypothesized that the number of friendship links might correlate with one’s net worth. To incorporate the count of relationships each node has (node degree) as a model feature, you simply count the connections and add them as an element in the data point representation table.

With the node degree example, you have just added a single "graphy" feature to describe a data point, while the rest of the machine learning workflow remains the same. While simply counting the number of connections does not require specialized storage or tools, it is a nice example to get you thinking about relationships between the data points and how you could use them in your machine learning workflow.

Graph algorithms usually refer to more global operations that consider the whole network as an input. Imagine that the information in the network can only flow through existing connections. For example, in Figure 1.11, if node A wants to communicate with node I, the information must flow through nodes E and G. A graph algorithm called *Betweenness centrality* can be used to identify the node’s influence over the information flow in the network. The assumption behind the Betweenness centrality is that the information always flows along the shortest paths between pairs of nodes. Then, the node’s importance over the information flow is simply the count of those shortest paths that pass through the node. The Betweenness centrality can be used to identify bridges between different communities or, in our example, find nodes with a strong influence over the information flow of the network.

Figure 1.12. Add the betweenness centrality as a feature of a person.


Node size corresponds to their influence over the information flow in the network
A
B
C
D
E
F
G
H
I
J


Data point features
Added betweenness value

Data point	Age	Hobby	Education	Degree	Betweenness	Net worth
A	22	Cycling	Trade	4	13	10000
B	35	Hiking	Master's	1	0	?
C	42	Skiing	PHD	5	5	1000000
D	77	Knitting	Bachelor's	3	8	250000
E	65	Gardening	High school	4	23	?
F	31	Jet Ski	Master's	1	0	125000
G	68	Hiking	Trade	3	15	550000
H	18	Skiing	Student	1	0	?
I	43	Cycling	PHD	1	0	50000
J	29	Climbing	Master's	1	0	?

In the example in Figure 1.12, node E is the bridge between the upper and bottom communities. First of all, it makes node E be the vital connection or a bridge between the two communities. If node E was removed from the network, the two communities couldn’t communicate anymore. Secondly, as a lot of information flows through it, it is probably well informed and has access to the type of information other nodes do not. Lastly, it can withhold passing the information forward and effectively cutting the communication lines between half of the network.

In the context of social networks, a person with better access to information can make better decisions. Not only that, but they can exert their influence over the information flow. As a practical example, say that you are a manager who reports directly to the CEO of the company. All the information from the CEO to your managees flows through you. You get to hear both sides of the story but have the option to decide which information you will pass along to the CEO or your subordinates. Since data or information is sometimes regarded as being a more valuable resource than oil, one’s influence over the information flow could correlate with their net worth. Similarly, as with the node degree, you could use the Betweenness centrality score as one of the model’s features that predicts a person’s net worth.

So far, both node degree and Betweenness centrality values are examples of manual feature engineering. Manual feature engineering is a process of identifying relevant data points representations that could correlate with the target variable by hand. Recent advancements in machine learning on graphs focus heavily on automatically encoding a node’s network position. The field of research that studies automatically encoding nodes' position in a network as a vector is called *node representation learning*. The key here is the learning part, where you train an embedding model or a neural network to describe a node in the network with a feature vector.

You might be familiar with image or text embedding models, which perform similarly to node embedding models. Both images and text can be represented as a graph.

Figure 1.13. Both image and text can be represented as a graph.

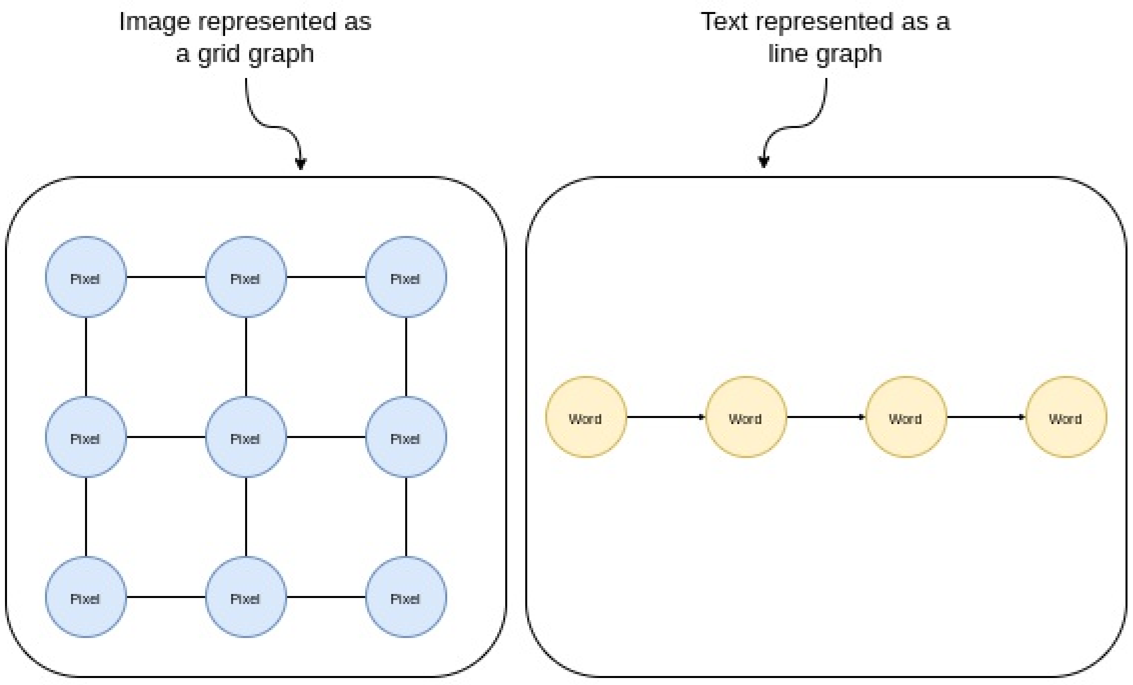
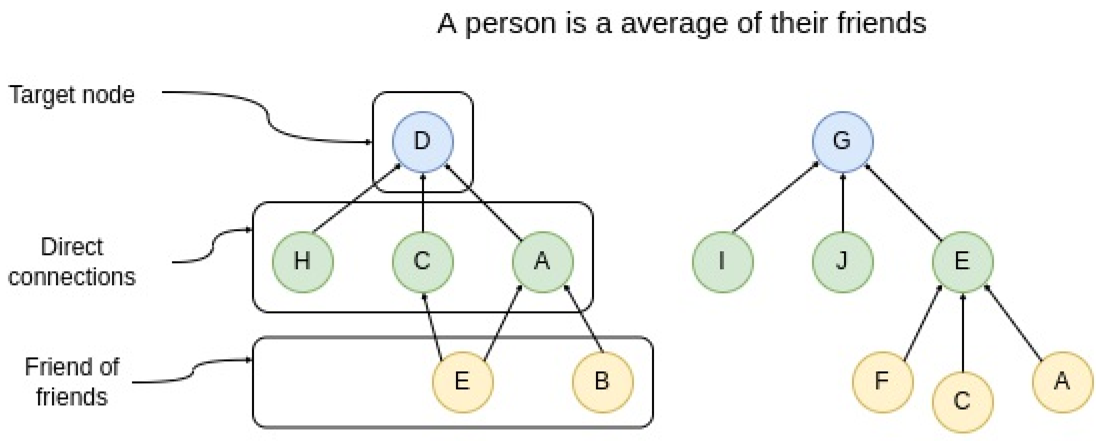


Image represented as a grid graph
Text represented as a line graph
Pixel
Word

Text and images have a pre-determined graph shape used to derive word or image representations by the embedding models. Without going into specifics of text or image embedding models, they essentially derive a pixel or word vector representation by observing its neighborhood. Most, if not all, real-world networks don’t have a pre-determined shape and can vary significantly from domain to domain. While the node embedding models take great inspiration from both word and image embedding models, they are designed to work on graphs with any shape and form.

Probably you have come across an anecdote that a person is an average of their best friends. I’ve also read some articles that not only your direct friends, but also friends of friends, influence your life options and choices. Instead of manually describing relevant person features as we did before, we can utilize a node embedding model that will automatically encode a node’s neighborhood as a vector that can be used in a downstream machine learning flow.

Figure 1.14. Learn node representation automatically by aggregating its neighborhood.



Some node embedding models encode only the nodes' position in the network, while others also consider nodes' properties. For example, *graph neural networks* consider both node properties and their position in a network to derive the final vector representation. In the example scenario of predicting a person’s net worth, you could utilize a graph neural network to derive a node’s vector representation that could be used to predict a person’s net worth more accurately.

Through the practical examples in this book, you will learn how to spot graph-shaped problems and construct a graph. Next, you will learn how to calculate and interpret both the local and global node characteristics like the node degree and the Betweenness centrality. The last part of the book is dedicated to machine learning on graphs, where you will learn how to improve your model’s accuracy by encoding the network’s structure as your data point feature sets.

1.4 Summary

- Networks are everywhere and they do matter
- A bar or line chart is not regarded as a graph in this book
- Problems that require a graph-based approach have interconnected data points such as self-referencing relationships in a social network or paths in a transportation network
- Sometimes the relationships between data points are not explicitly defined but can be inferred based on indirect patterns, as in the example

of users purchasing products
- Node degree attribute represents the count of relationships a node has
- Node degree is a local characteristic that examines the node’s direct neighborhood
- Graph algorithms like the Betweeness centrality move beyond the direct neighborhood of a node and inspect the whole network
- Node embedding models are used to automatically encode a node’s network position as a vector
- Encoding network information for a downstream machine learning task can greatly improve your accuracy

1.5 References

[Shan Xu et al, 2020] Xu CS, Januszewski M, Lu Z, Takemura S-Y, Hayworth KJ, Huang G, Shinomiya K, Maitin-Shepard J, Ackerman D, Berg S, et al. A connectome of the adult Drosophila central brain. bioRxiv. 2020 [accessed 2021 Jan 21]:2020.01.21.911859. www.biorxiv.org/content/10.1101/2020.01.21.911859v1. doi:10.1101/2020.01.21.911859

[Leonhard Euler, 1736], "Solutio problematis ad geometriam situs pertinentis". Comment. Acad. Sci. U. Petrop 8, 128–40, 1736

[Erdős Rényi, 1959] P. ERDŐS-A. RÉNYI, On random graphs. I,Publicationes Mathematicae (Debrecen),6 (1959), pp. 290–297.

[S.Brin and L. Page, 1998] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 30(1-7):107–117, 1998.

[Albert, R., Jeong, H. & Barabási, 1999] Albert, R., Jeong, H. & Barabási, AL. Diameter of the World-Wide Web. Nature 401, 130–131 (1999). doi.org/10.1038/43601

[Barabási and Albert, 1999] Barabási, A.L., & Albert, R. (1999). Emergence of Scaling in Random Networks. Science, 286(5439), 509–512.

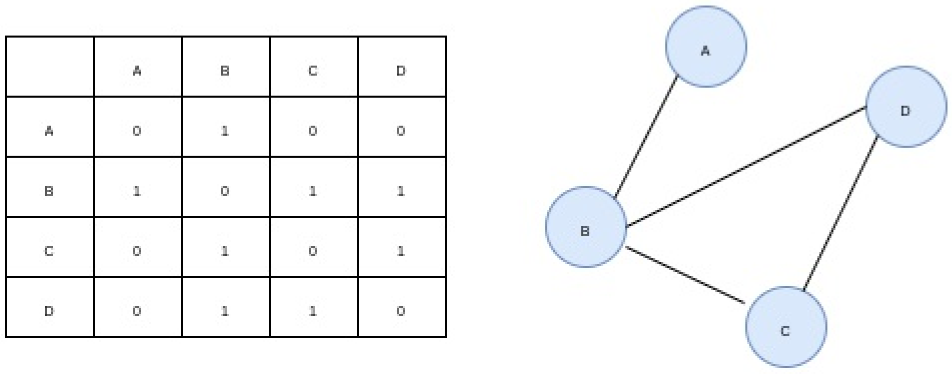
2 Representing network structure - design your first graph model

This chapter covers:

- Math and text representation of graphs
- Introducing graph databases
- Labeled-property graph model schema design
- Extracting information from text

In the previous chapter, you learned the basics of graph theory. Before you dig more into practical network analysis, you first have to learn how to practically represent network structures. The most basic graph representation is the mathematical data structure *adjacency matrix*.

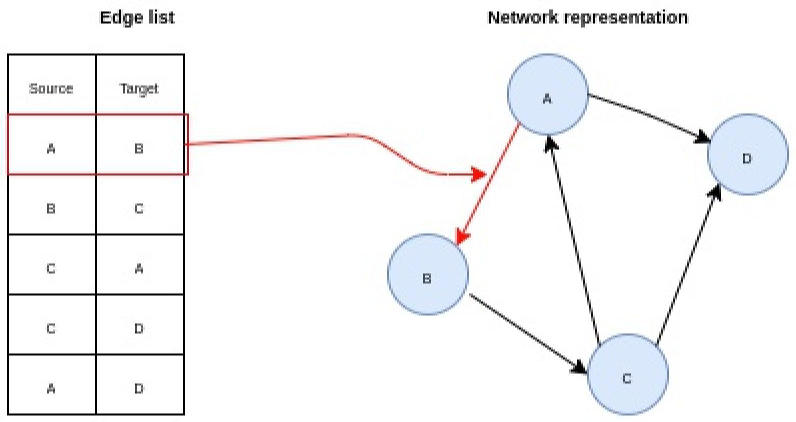
Figure 2.1. Adjacency matrix representing a network structure



	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

An adjacency matrix is a square matrix, where the matrix elements indicate whether pairs of nodes are connected or not in the graph. The adjancency matrix dimensions are equal to the number of nodes in the graph. It can also be expanded to represent weighted graphs. Instead of having zeroes indicating the presence of the relationships, you store the relationship weight as the matrix element. You will not be using it in the examples of this book, but you can check out the adjacency matrix appendix if you want to learn more. Another mathematical structure to represent networks is called the *edge list* data structure.

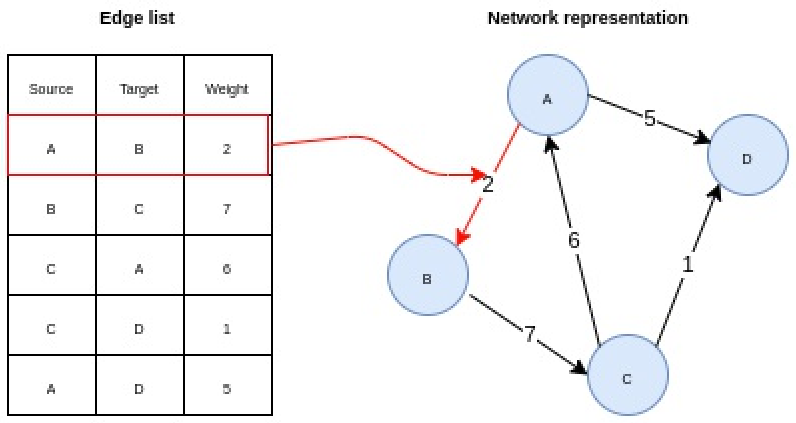
Figure 2.2. A directed unweighted graph represented with a edge list data structure



Source	Target
A	B
B	C
C	A
C	D
A	D

An edge or relationship list is a simple data structure where each row represents a relationship of a given network. The first row in the edge list of Figure 2.2 represents a directed relationship from node A to node B. There are five rows in the edge list, which represent five relationships of the network. It also supports representing multi-graphs, where you can have multiple relationships between a given pair of nodes. An edge list can be expanded to hold the information about relationship weights.

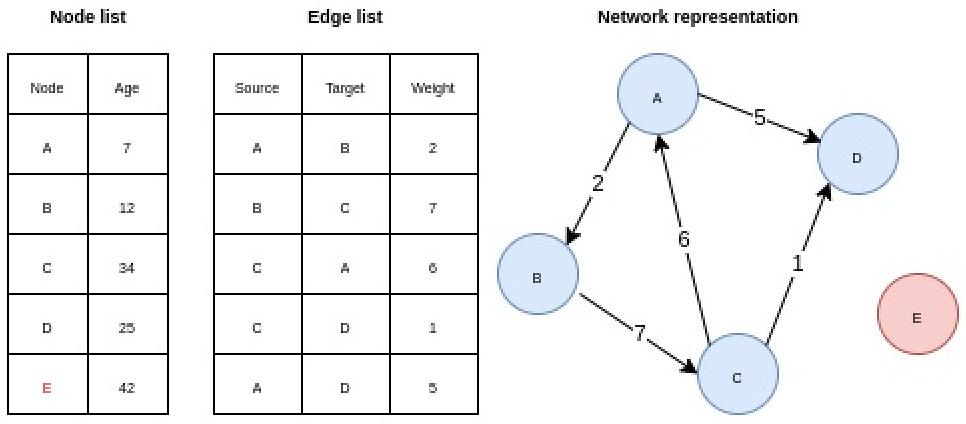
Figure 2.3. A directed weighted graph represented with a edge list data structure



Source	Target	Weight
A	B	2
B	C	7
C	A	6
C	D	1
A	D	5

The value of the relationship weight is stored in a separate column. You could also store additional information about the relationship in the edge list, such as the time component. One limitation of the edge list is that it does not allow isolated nodes to be present. Isolated nodes are nodes without any relationships. This limitation can be solved by introducing a *node list* next to the edge list.

Figure 2.4. With addition of the node list, you can represent isolated nodes in a network



Node list

Node	Age
A	7
B	12
C	34
D	25
E	42

Edge list

Source	Target	Weight
A	B	2
B	C	7
C	A	6
C	D	1
A	D	5

By introducing the node list next to the relationship list, you can describe networks with isolated nodes present. The network in Figure 2.4 has an isolated node E. Node E is isolated because it is described in the node list but has no entries in the relationship list. The node list can also be expanded to store various properties of the nodes. For example, the node list in Figure 2.4 contains information about nodes' age. Node and relationship lists are very frequently used as input to network visualization tools. When trying to visualize a network, you could store the size and the color of the visualized node as additional properties in the node list. Node and edge lists are useful when you have a defined graph structure that doesn’t require additional data manipulations or transformations. However, in practice, I have noticed that you often need to transform and manipulate the network data to fit your problem best. A typical example would be translating indirect relationships into direct ones. While you could use a scripting language for data transformations directly with node and edge lists, I recommend using a graph database and dedicated graph-pattern query languages.

Before you learn more about graph databases, we will also look at the text representation of simple networks. The text representation of networks comes in handy when you want to communicate the network structure via text quickly. We will borrow the syntax from the Cypher query language. Cypher’s syntax provides a visual way to match patterns of nodes and relationships in the graph using ASCII-Art syntax. Its syntax describing nodes and relationships is also the basis for the future Graph Query Language (GQL), which aims to unify the graph-pattern query language the same as SQL did for relational databases. An example node representation in Cypher looks like the following:

(:Person {name:"Thomas"})

To depict nodes in Cypher, you surround a node with parentheses, for example (node). The colon is used to describe the type of a node. In the above example, the node type is defined as `Person`. Nodes can also have properties, which are depicted as key-value pairs inside the brackets of a node. There is a single key value pair inside the curly brackets, `{name:"Thomas"}`, that represents the name property of the node. Relationships in Cypher are surrounded with a square bracket.

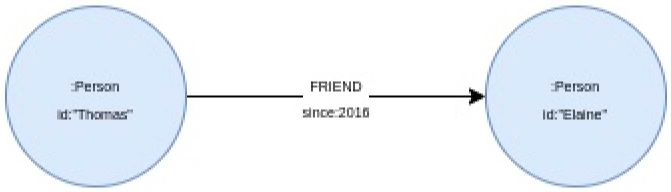
-[:FRIEND{since:2016}]->

Similarly to nodes, you describe the type of relationship with a colon. Relationships can also have properties defined as key-value pairs inside the curly brackets. A relationship can never exist in solitude without existing source and target nodes. Cypher syntax is frequently used to describe patterns of a network. For example, you can specify a simple friendship network with the following syntax:

(:Person {name:"Thomas"})-[:FRIEND {since:2016}]->(:Person {name:"Elaine"})

This Cypher syntax describes a friendship relationship between Thomas and Elaine and can be visualized as the following network.

Figure 2.5. Example friendship network between Thomas and Elaine



Both Thomas and Elaine are persons, and they are friends since 2016. If you look carefully, you can observe a direction indicator of the relationship at the end of the text representation. With it, you can differentiate between directed and undirected relationships. If you want to describe the friendship relationship as undirected, all you have to do is omit the relationship direction indicator.

-[:FRIEND{since:"2016"}]-

I need to add a small disclaimer here. Many of the graph databases don’t directly support storing undirected relationships. I will show you how to deal with undirected relationships in a graph database in Chapter 3.

Try to represent the relationship you have with the organization you are employed at with the Cypher syntax. There are two types of nodes present in this graph pattern, a person and an organization or a business. You can also add additional node or relationship properties as you see fit.

I could describe my relationship with the Manning publication using the following Cypher syntax:

(:Person {name:"Tomaz"})-[:WRITES_FOR {since:2020}]->(:Organization {name:"Manning Publication"})

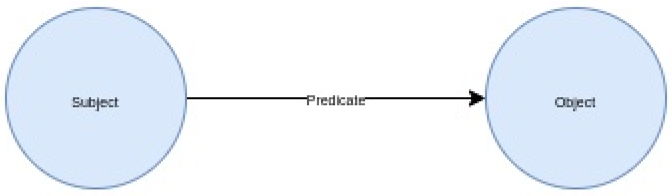
2.1 Graph databases

A *native graph database* is typically a type of NoSQL database designed to store graph representations. There are also some implementations of graph layers on top of SQL databases, but we won’t cover them here. The key difference between a native graph database and other databases is that native graph databases support *index-free adjacency*. Index-free adjacency ensures that you can traverse relationships without using an index. This allows the performance of graph traversal to be independent of the overall graph size. The query cost is only associated with the part of the graph touched or walked by the query. For example, if you were to traverse all the relationships of a node, the query performance is only dependant on the number of connections a node has. On the other hand, traditional relational databases traverse relationships by performing join operations. Join operation is typically an intersection operation between two sets, where the relational database uses an index to detect where those two sets overlap. As a consequence, the size of the data or the overall graph affects the query performance. Also, the cost of queries in a relational database grows exponentially with the number of joins. From a performance perspective, the index-free adjacency versus a traditional join operation is the most important thing to consider when thinking about using a native graph database. In general, the graph databases can be split into two categories based on the underlying graph model they use.

2.1.1 RDF Graph Database

First, you will learn about the *Resource Description Framework* (RDF) graph model. While most RDF graph databases do not support index-free adjacency, it is still relevant to learn about their underlying graph model. RDF databases, also known as triplestores, are designed to store and allow the retrieval of *triples* using graph-pattern queries. A *triple* is a data entity that consists of subject, predicate, and object. The triple data structure can be visualized as following:

Figure 2.6. A graph visualization of a triplet with two nodes(Subject and Object) and a relationship connecting them (Predicate).



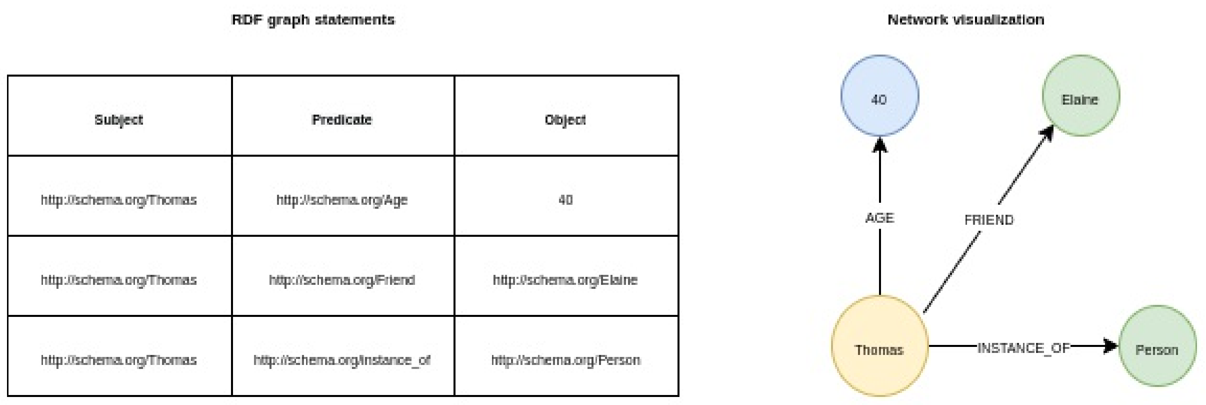
Although not explicitly specified in a triple, by RDF standard, there can be three kinds of nodes in an RDF graph:

- IRI: An IRI or an internationalized resource identifier denotes any real-world entity or concept. An IRI is used to identify nodes in the graph in an unambiguous way. You can describe a single node in multiple triples by using the same resource identifier (IRI).
- Literal: Literals are values used to represent datatypes like strings, numbers, or dates.
- Blank node: Blank nodes are nodes without an identifier used in special RDF modeling scenarios.

Predicates can be interpreted as relationships between two nodes or as defining an attribute value. As the underlying data structure is a triple, an attribute value is stored as a literal node, and the predicate forms a relationship between the subject node and the literal object node. Each predicate has a resource identifier that defines the type of relationship. The relationships are always directed as they point from the subject to the object node. For example, let’s say that you want to represent the following information as an RDF graph:

- Thomas is 40 years old
- Thomas is friend with Elaine
- Thomas is a person

Figure 2.7. Table and network visualization of a RDF graph, where Thomas is 40 years old and is friends with Elaine



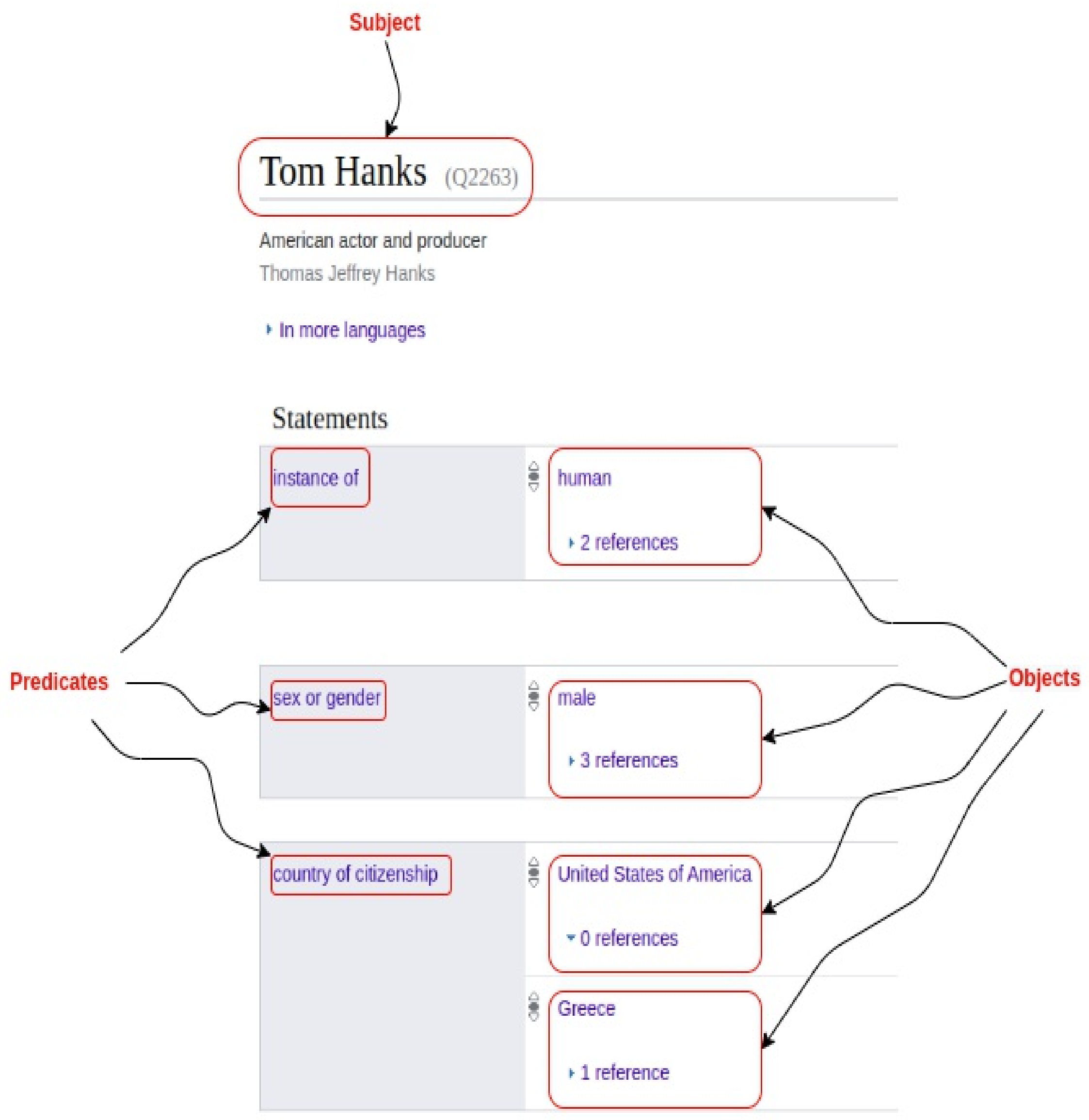
RDF graph statements

Subject	Predicate	Object
http://schema.org/Thomas	http://schema.org/Age	40
http://schema.org/Thomas	http://schema.org/Friend	http://schema.org/Elaine
http://schema.org/Thomas	http://schema.org/instance_of	http://schema.org/Person

You can observe that you are doing a complete data normalization by representing a network with an RDF graph model. RDF’s abstraction level is a triple, where the subject and object of the triple are represented as nodes, and the predicate is represented as a relationship. Consequently, there is no internal structure available on the nodes and relationships, meaning that you are not dealing with internal node or relationship properties. but store the properties as a separate node with a literal value. Although, in practice, you might consider literal node values as node attributes, the underlying triple data structure treats them as separate nodes. In the example in Figure 2.7, the node Thomas is disambiguated by using a resource identifier "http://schema.org/Thomas". Resource identifiers follow the structure of a URI. For this example, I have made up the resource identifiers values. I have also omitted the schema.org prefix in the network visualization for readability purposes. The node Thomas has three relationships. Two of them point to another resource node, while the age relationship is pointing to a literal value and can be interpreted as a node attribute.

In my opinion, WikiData is the most famous graph database based on the RDF graph model. WikiData acts as central storage for the structured data of its sister projects Wikipedia and others. It features a SPARQL endpoint that you can use to extract relevant data. This book will teach you the most basic SPARQL queries that will allow you to fetch relevant data from WikiData and enrich your graph with additional information. You can also explore the WikiData graph with your favorite internet browser. For example, you can look up and find information about Tom Hanks on the WikiData webpage.

Figure 2.8. WikiData web page with information about Tom Hanks available at www.wikidata.org/wiki/Q2263



Tom Hanks is a resource node in the WikiData graph. Its id is Q2263 and in the Figure 2.8 is always regarded as the subject of the triple data structure. That being said, this does not limit Tom Hanks to appear as a object of the triple data structure in any other examples. You can observe that Tom Hanks is a citizen of both the United States of America and Greece.

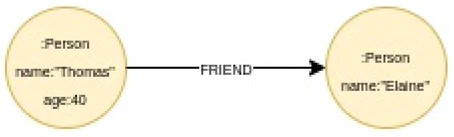
As mentioned, most RDF graph databases do not support index-free adjacency. For this reason, I will not use them as a source of truth for network analysis in this book. However, a lot of world knowledge is stored in RDF databases such as WikiData and DBpedia. Both WikiData and DBpedia provide publicly available APIs that you can use to retrieve information and enrich your existing graph with it. In later chapters, you will use the WikiData SPARQL API to enrich your graph. Also, the RDF has a couple of standard serialization formats that are helpful when exchanging network data, so if you ever come across them, you will know the underlying structure they represent.

2.1.2 Labeled-property graph database

The other category of graph databases is based on the *labeled property graph model* (LPG). A key difference between the labeled property graph model and the RDF model is that with LPG, both nodes and relationships can have internal structures. The internal structures are node or relationship properties stored as key-value pairs. Nodes also have a special type of property called a label, which is used to represent node roles in your domain. For example, you could use the label to categorize whether a node represents a person or an organization. All relationships are directed and have exactly a single type assigned to them. As mentioned, relationships can also store properties. To demonstrate a simple LPG graph model, I will use the same information as in the RDF example:

- Thomas is 40 years old
- Thomas is friend with Elaine
- Thomas is a person

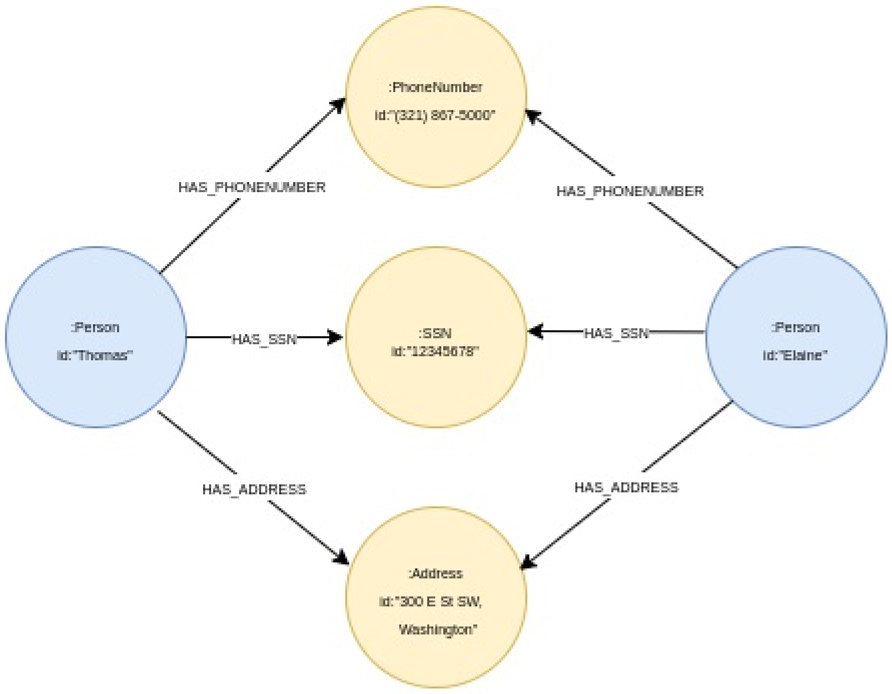
Figure 2.9. Labeled-property graph model representing the example data



As you can observe, the key difference from the RDF approach to graph modeling is that labeled-property graph(LPG) supports both node and relationship properties stored as key-value pairs. In the example in Figure 2.9, the age information is now stored as an internal node property. Another key difference is that you can group or categorize nodes into distinct sets using a node label. In Figure 2.9, both Thomas and Elaine nodes have a label that indicates they are categorized as persons. You might have observe that the Cypher query syntax you learned before is used to describe LPG model domain.

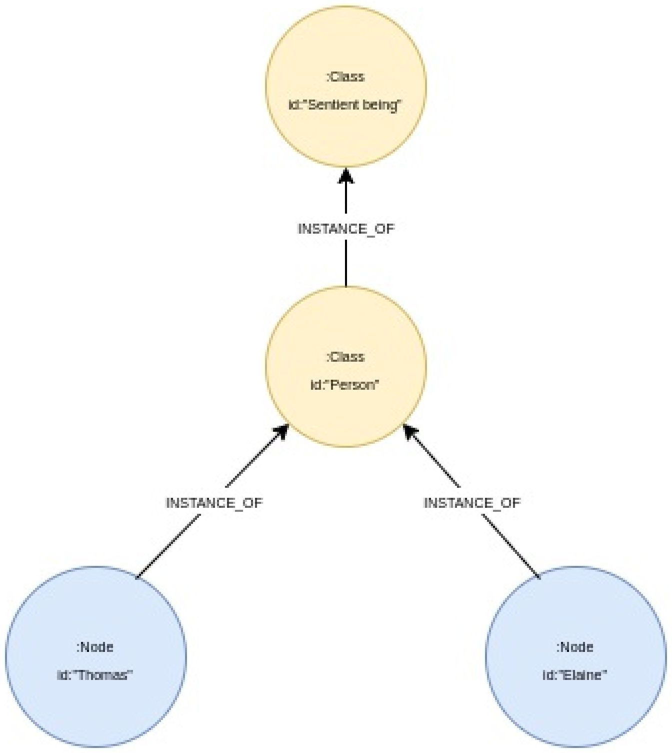
In some domain use cases, you might still want to represent literal values as separate nodes. A typical example is the fraud detection scenario, where you are interested in examining customers who share the same address, social security number, or phone number.

Figure 2.10. Labeled-property graph model representing a fraud investigation domain



Nothing is stopping you from modeling your graph where some literal values are represented as separate nodes. The graph model depends on your task, and with the LPG model, you can represent a literal value both as an internal node property (key-value pair) as well as a separate node. Similarly, you have the option to represent the label as a separate node as well. A classic example would be describing class hierarchy with the LPG model.

Figure 2.11. Labeled-property graph model representing a class hierarchy domain



As mentioned, the graph model depends on the task you are trying to solve. With labeled property graphs, the abstraction level is nodes, relationships, labels, and properties.

In this book, you will be using an LPG graph database as a source of truth for graph analysis. You will learn how to use node and edge lists to construct an LPG graph and also fetch data from an RDF graph database (Wikipedia) to enrich the graph model. It is completely fine if you want to use an RDF graph database as the source of truth in your further graph analysis. I will, however, not go through the details of constructing an RDF graph model to best fit your graph analysis task in this book.

2.2 Designing your first labeled-property graph model

Now imagine a scenario in which a client asks you to perform a network analysis of the Twitter social network. The client will provide all the relevant data. Your task is to represent the data as a graph and gain various insights by performing network analysis. You will be using the LPG model to represent the Twitter social network. A general approach to graph modeling is working it backward and starting with the questions you want to answer. Here, unfortunately, no specific questions were posed in the assignment. It is your job as a network scientist to do your best and find as many insights as possible. You can start by trying to describe the domain you have at hand. In the Twitter example, the most basic specifications would be:

- A user can follow other users
- A user can publish tweets
- A user can retweet posts from other users

In this section, you will learn how to develop a LPG graph model.

2.2.1 Follower network

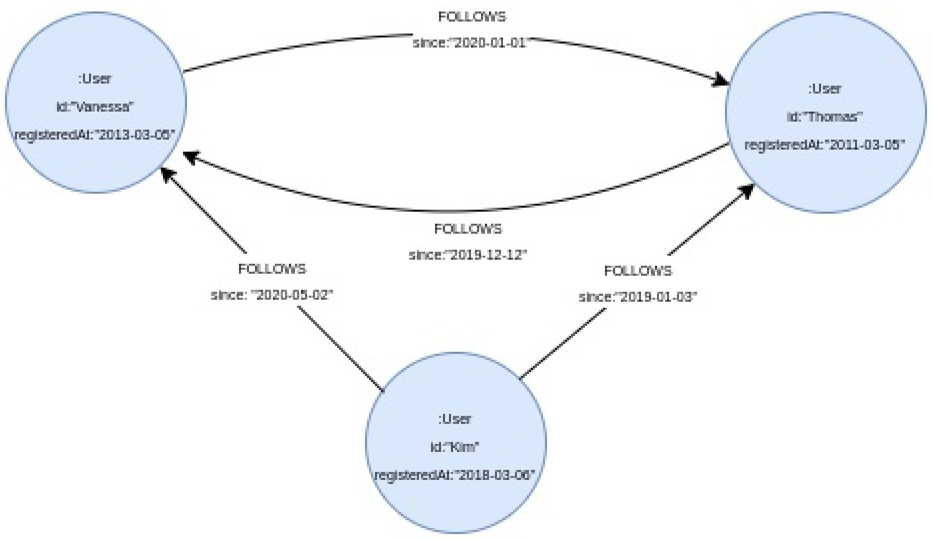
On Twitter, you have the option to follow other users. By following users, you are subscribing to their activity and indicate that you would like to see followed users' tweets on your feed. The follower network specification is:

- A user can follow other users

As an exercise, try to design the follower graph model. As a rule of thumb, you would like the entities to be represented as nodes. You can also borrow some logic from English grammar. The subject and object of a sentence are often represented as nodes, and the verb describes the relationship. Adjectives can be translated into properties. When designing graph model, take into consideration whether the direction of a relationship holds semantic value.

There is no "correct" way to design a graph model. However, there are some general guidelines that you should follow. In the follower network specification, both the subject and the object of the sentence are users and can be represented as nodes. Relationships can be used to represent the verb of the specification sentence. Here, it makes sense to represent a follow interaction as a relationship between two users.

Figure 2.12. Twitter follower network model



A single relationship pattern from the Figure 2.12 can be represented with the Cypher's pattern syntax as:

(:User{id:"Vanessa", registeredAt:"2019-03-05"})-[:FOLLOWS{since:"2020-01-01"}]->(:User{id:"Thomas", registeredAt:"2011-03-05"})

Each node in Figure 2.12 has a label `User` attached to it. The node label is used to categorize nodes that represent users. As there is only a single label of nodes present, you are dealing with a monopartite network. A good practice is to have an *unique identifier* for node disambiguation in place for all the nodes of the network. In the Twitter social network, you could use the Twitter handle as the user's unique identifier. The data also contains the signup date for users, which you can store as the node's `registeredAt` property. The specification did not explicitly state that the `FOLLOWS` relationship is not symmetrical. In Figure 2.12, you can observe that Kim follows Vanessa, but Vanessa does not follow Kim. In other words, Kim will see Vanessa's tweets on her feed, while Vanessa will not see anything from Kim. You can conclude that the direction of the relationships holds semantic value, and as a result, you are dealing with a directed network. The FOLLOWS relationship has no notion of strength assigned to it, which implies that the Twitter follower network is unweighted. You do, however, know when the relationship was created. The information about the relationship creation date can be stored as relationship property.

Before continuing, you can examine the types of questions or insights you could answer using this graph model. Social networks are a prime example where you could try to identify influencers. A simple metric to evaluate influencers is by looking at a users' direct followers count. The more followers a user has, the more widely his tweets will be distributed. However, there is a difference if your followers are also influential or not. Having a Fortune 500 CEO following you might be more influential than having your neighbor following you. Probably, the most famous graph algorithm to evaluate the *transitive* influence of a node is PageRank. A transitive relationship is an indirect relationship between two nodes, where for example a node A is connected to node B, and node B is connected to node C. In this example, node A is not directly related to node C but has an indirect relationship through node B, which implies they are transitively connected. Suppose that a Fortune 500 CEO has more connections than your neighbor. In that case, you will gain more transitive relations and consequently network influence by having a Fortune 500 CEO follow you than having your neighbor following you. Another type of analysis often used for social networks is to try and deduce community structure. In the last couple of years, it has become increasingly popular to use graphs for predictive analytics. For example, the popular saying is that a person is the average of his closest (five) friends. You could use this assumption and try to predict a property of a person based on the users he follows. Since the followers' graph contains the time component for both the nodes and relationships, you could also examine how the network evolved over time and use that information to predict how it will grow in the future.

2.2.2 User - Tweet network

Tweets are the primary way you can share content on Twitter. The simplest description of User - Tweet network is:

- A user can publish a tweet

To get into the flow of designing graph models, try to develop a graph model that describes the above specification. In the long run, it will be beneficial for you if you take a whiteboard or a sketchbook and draw some basic graph models. When I am designing a graph model, I am trying to answer the following questions:

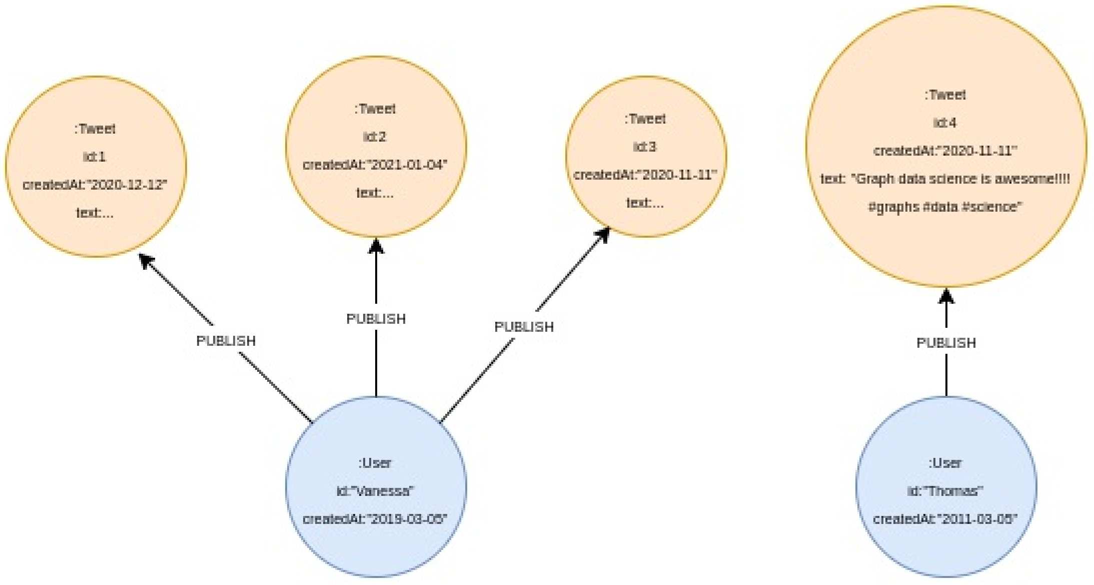
- How many different types of nodes are present?
- What kind of properties do these nodes have?
- Which property would you use to store the unique identifier of nodes?
- What type of relationship are present?
- Is a single relationship type enough to accurately describe your domain?
- Does the relationship direction hold any semantic value?
- How do properties qualify or quantify relationships?

If you again try to develop a graph model from the specification, you might come up with the following graph pattern:

(:User)-[:PUBLISH]->(:Tweet)

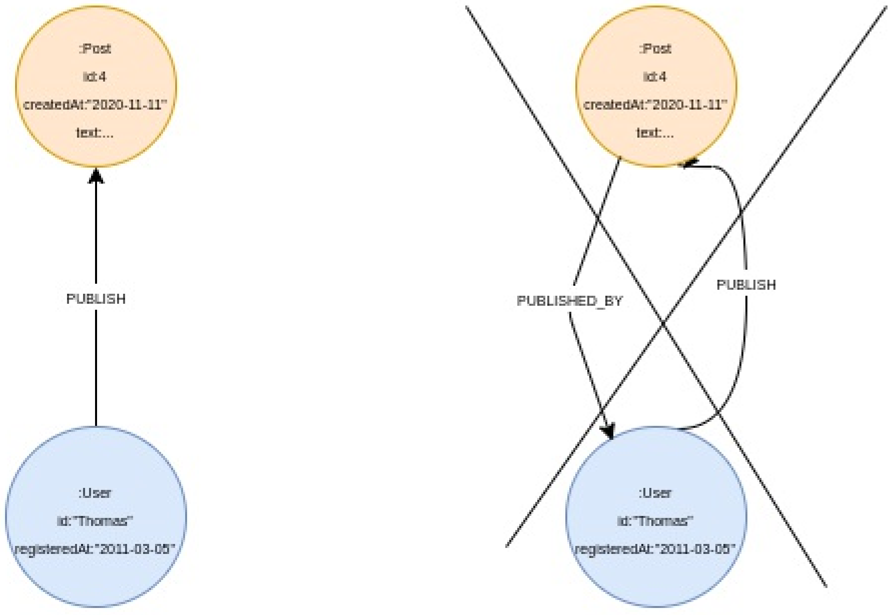
For more compact text representation of the model, you can usually leave the node and relationship properties out of the Cypher pattern syntax. There are two types of nodes present, users and tweets. All you need to add is a relationship between them to indicate the author of the tweet. A good practice is to describe your graph model as domain-specific as possible. You could have also used a more generic label for tweets such as a `Post`. With more generic labels, you might run into issues along the way if you were asked to add additional information about users from Facebook, LinkedIn, or Stackflow. In this scenario, you wouldn't want to merge content from all social media platforms under a single generic node label `Post`.

Figure 2.13. User - Tweet network



You have already established an unique identifier for users in the previous exercise. Now you can do something similar and assume there is an unique id created for each tweet when it is created. The unique identifier, creation time, and the tweet's text are stored as the properties of a `Tweet` node. As there are two distinct labels of nodes present, you are dealing with a bipartite network. In this example, the direction of the relationship is not that important. You could turn the relationship direction around and change the relationship type to something more appropriate like `PUBLISHED_BY`. Both approaches are correct. You have to pick one and stick to it. My preference is to define relationship types in active voice, which is PUBLISH in this example. What you don't want to have is both modeling options present at the same time. You don't have to worry about query performance as there is no penalty for traversing a relationship in the opposite direction in a native graph database.

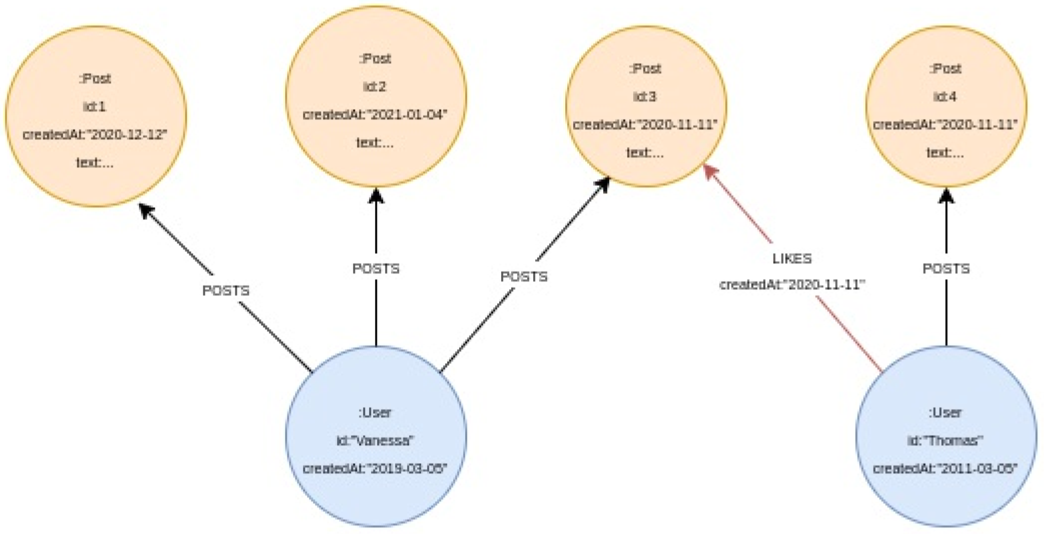
Figure 2.14. A bipartite network graph model, where a single relationship type is enough to contain all the relevant information



The right-side example in Figure 2.14 demonstrates what you should avoid when you are developing a graph model. Adding the relationship in the opposite direction adds no semantic value and is entirely redundant. You also don't have to worry about getting from a Tweet to a User. With Cypher query language syntax, you can traverse any relationship in the opposite direction, or you can completely ignore the direction.

Another question that might come up is why you store the tweet creation date as a property of the node and not the relationship. It is a valid question, and as always, it mostly depends on the domain you are dealing with. In the Twitter universe, a tweet can only have a single author. Having a single author implies that a tweet will have precisely one `PUBLISH` relationship pointing to it. And so, a tweet is created only once. When you are making such decisions, you should always include the types of queries you want to execute on this graph model. If you think of the most basic use case, where you want to count the number of tweets per day, it is simpler to have the creation date stored as a property of the tweet node instead of the `PUBLISH` relationship. You will learn more about how graph-pattern query languages work in the following chapter. Users liking a tweet would be an example where adding the creation date on the relationship makes more sense. While there can be only a single tweet author, many users can like a given tweet. To capture the creation time of a like from a particular user, it makes sense to store the creation time information as relationship property. If you wanted to store the time information about the likes on the `Tweet` node in the form of an array of dates, you would lose the information about which users gave a like at that particular time.

Figure 2.15. An example of a twitter post LIKE relationship, where it makes sense to store the creation date as a property of the relationship



From the network science perspective, having only `PUBLISH` relationships between users and tweets is not so interesting. As each tweet has only a single relationship pointing to it, there are no overlaps or similarities between tweets that you could try to analyze. However, if you added LIKES relationships to the model, you could analyze which users like the same or similar content and create segments of users based on the content they like.

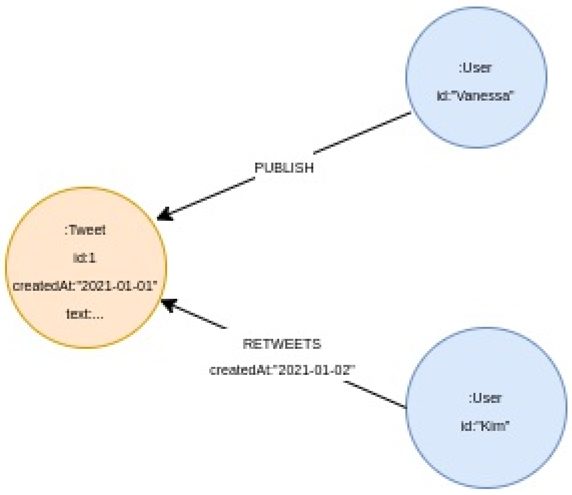
2.2.3 Retweet network

The only remaining task specification is the definition of a graph model for retweets. When users strongly react to a tweet, they might want to share it with their followers to amplify its reach. In this case, they have the option of retweeting the original tweet. Optionally, users can like the retweet, and those likes do not count towards the original tweet. The task specification was defined as:

- A user can retweet posts from other users

This specification is a bit more complex than the previous two. You can’t just extract the subject and the object of the sentence and use that to describe the nodes. The User - Tweet network is already defined, so you can expand on that and add the retweets somehow in the graph model. There are several graph model variations that you can choose from. A simple option is to add the RETWEET relationship between the user and the original tweet.

Figure 2.16. Graph model of the retweet network, where the RETWEET relationship is defined between the original tweet and the user who retweeted

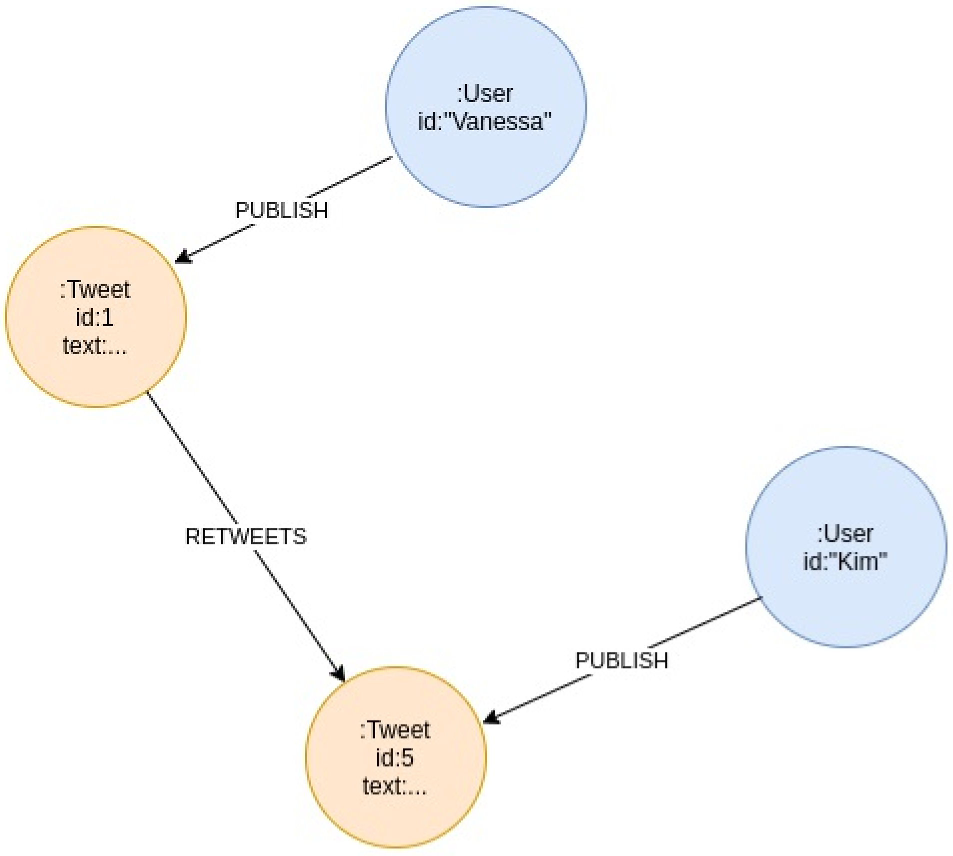


Cypher syntax to describe this pattern looks like:

(:User)-[:PUBLISH]->(:Tweet)<-[:RETWEETS]-(:User)

Here, you are using the RETWEET relationship between a user and the original tweet. This graph schema does not treat a retweet as an actual tweet. This is neither good nor bad. It all depends on your use case and what do you want to achieve with the analysis. There is, however, a slight problem with this approach. On Twitter, a retweet can also have likes that are tied to the retweet and not the original post. With the LPG model, you can’t create a relationship that is pointing to another relationship. Using the graph model in Figure 2.16, you lose the ability to attach likes to the retweet. Later in this chapter, you will extract information from retweets text, such as hashtags and mentions. There, you will again face the issue of not having the ability to attach hashtags and mention information to the retweet relationship. If the Twitter domain did not allow separate likes that are not counted separately, then having a retweet as a relationship would make sense. Unfortunately, that is not the case when dealing with the Twitter domain. To solve this issue, you can treat the retweet as a separate tweet that references the original tweet to avoid this limitation. This way, you keep the graph consistency while also allowing you to add additional information to retweets later on.

Figure 2.17. Graph model of the retweet network, where the retweet is treated as an tweet that references the original tweet



You can still easily differentiate between tweets and retweets. For now, don’t worry about the underlying data and how you can retrieve it. You will learn more about the Twitter data source in the next chapter. When retweeting a tweet on Twitter, a user can optionally add their comment to the retweet. In this example, we have skipped this scenario because it was not a part of the specifications. However, we will take a look at how to model quote tweet interactions in the next chapter. A retweet has an outgoing RETWEETS relationship, and the original tweet can only have an incoming RETWEETS relationship. The graph model where you store the retweet as a separate node also allows you to add other relationships to it later in the analysis if needed. Again, you want to evaluate how you can use this graph to extract insights. Remember, most often, users retweet a tweet when they strongly react to its content and want to share it with their followers. You could count the retweets and try to identify the most popular tweet topics or their authors. You could also infer a new direct relationship between users based on the retweet pattern. Translating indirect graph patterns to direct relationships is a frequent intermediate step in the network analysis. In the retweet network case, you could infer a direct relationship between users based on how often they retweet other users.

Figure 2.18. Inferring an new network from an indirect pattern of retweet relationships

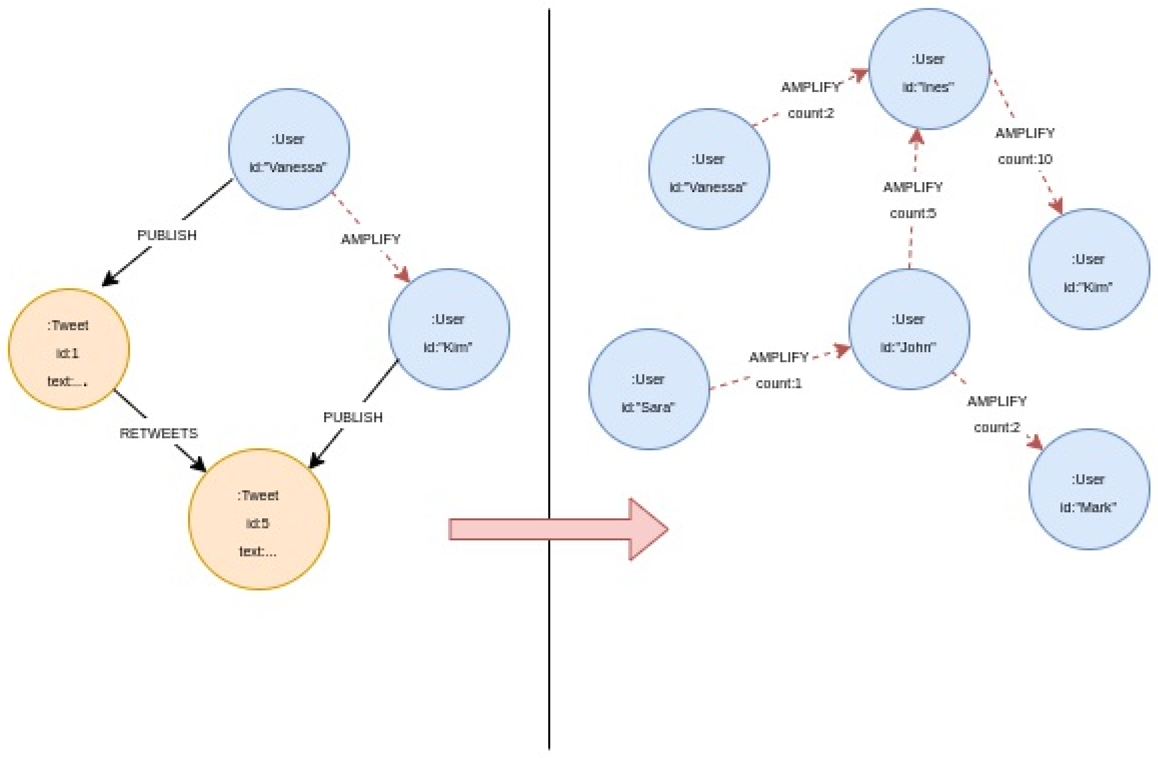


Figure 2.18 demonstrates a scenario where Vanessa just retweeted a tweet from Kim. If you assume that a retweet is always a positive interaction, you

could presume that Vanessa actively promotes Kim’s tweet and amplifies its reach. This indirect amplification pattern can be translated to a direct relationship between Kim and Vanessa, as shown in Figure 2.18. You can use the text representation to show that you take the following graph pattern:

(:User)-[:PUBLISH]->(:Tweet)<-[:RETWEETS]-(:User)

And translate this indirect relationship pattern into a direct relationship between users:

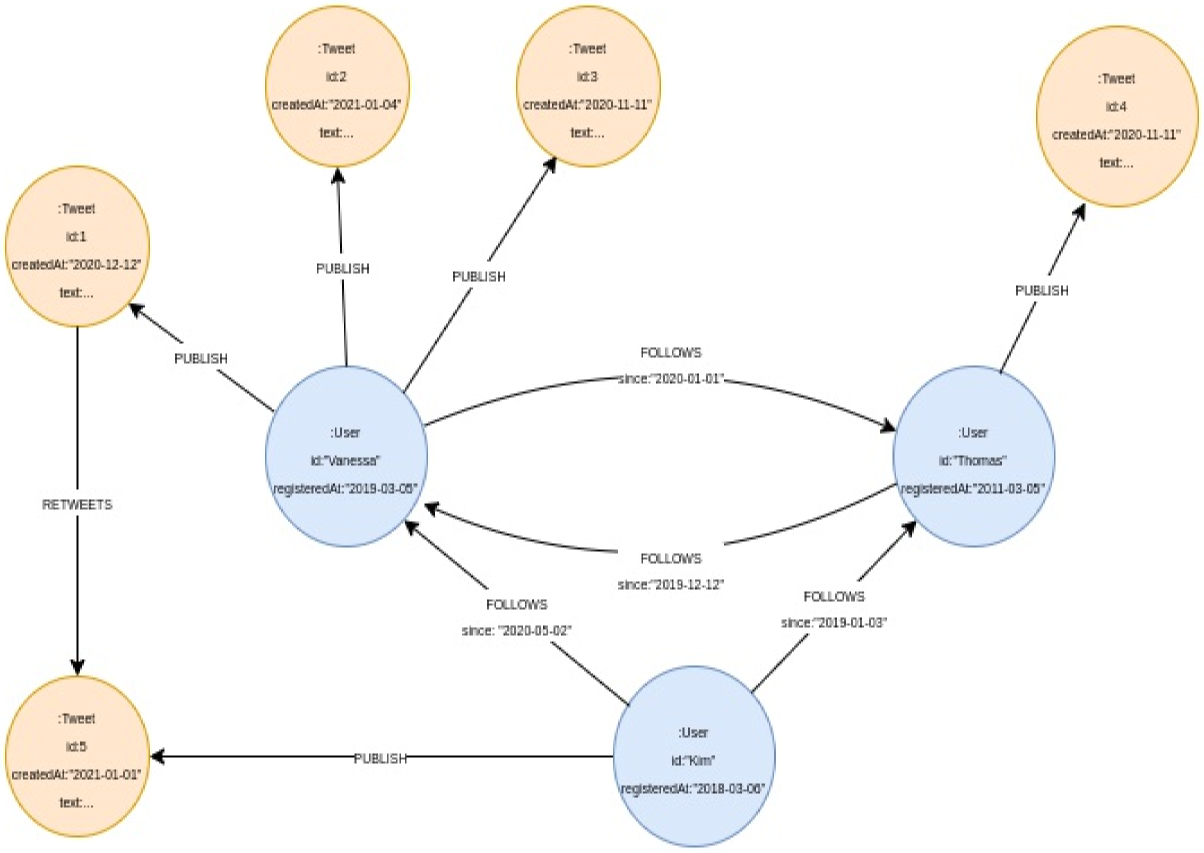
(:User)-[:AMPLIFY]->(:User)

By translating an indirect pattern to a direct relationship, you are creating a new inferred network. The type of the inferred relationship depends on your domain use case. Here, I have chosen the `AMPLIFY` type, as the retweet amplifies the reach of the original tweet. The new inferred `AMPLIFY` relationship is directed as the direction of the relationship holds semantic value. It is also weighted as you can quantify the strength of the `AMPLIFY` relationship by counting the number of retweets. If a user retweets a post from another user a single time, then the weight of the `AMPLIFY` relationships is one. However, if a user regularly retweets posts from another user, the weight would be equal to the number of retweets. You could, again, search for influencers within this inferred network or try to find communities of users who actively support each other by promoting their content.

2.2.4 Representing graph schema

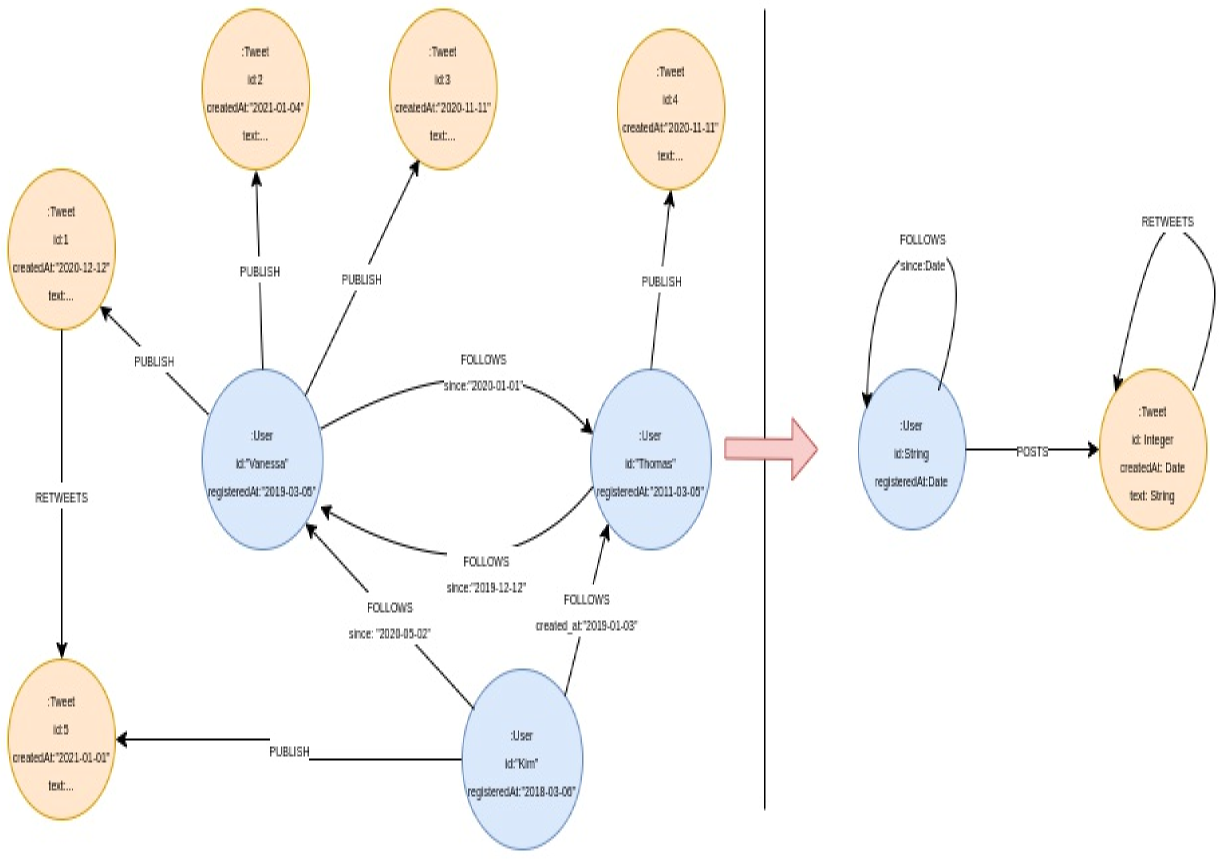
The beauty of the graph approach to data modeling is that you can always connect new information to an existing graph. You can now combine all the graph model decisions so far into a single graph model.

Figure 2.19. Labeled-property graph representing a twitter network, where users can follow one another and they can also publish and retweet posts.



Over the years, people have come up with ways to represent an LPG graph schema. There is no official standard to present an LPG graph schema, but I will show you how most people I have seen approach it.

Figure 2.20. Twitter social network graph model representation

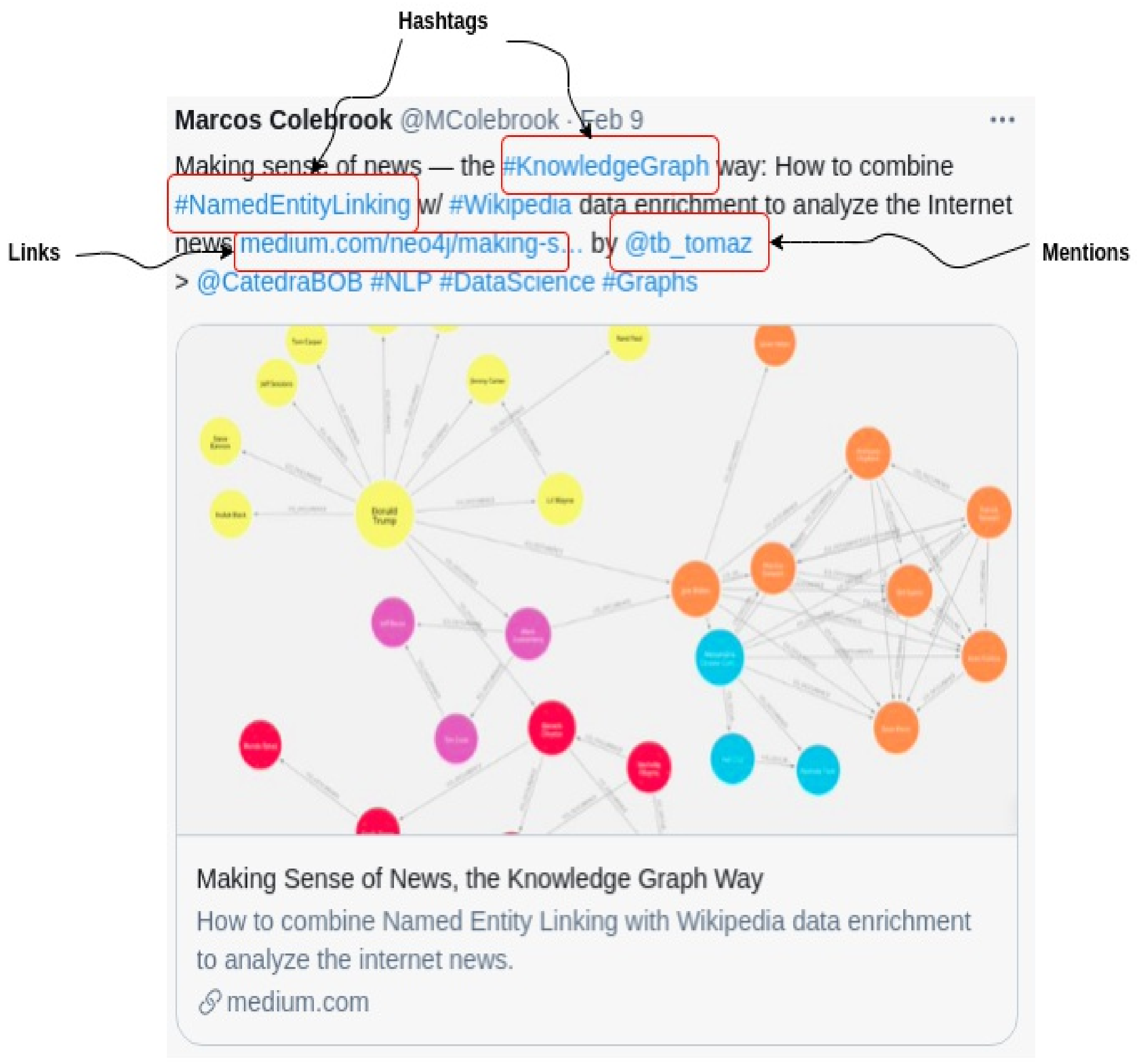


All node types or labels are represented as a single node. There are two different labels of nodes present in the current Twitter network representation. You can describe them as two nodes in the graph schema representation. The node properties are added to the node representing each label. Some people like to add example values to the node properties, but I prefer to add their data type as the value. For now, there is no agreed way to visualize if a node property is optional or not or if it is used as a unique identifier. Relationships between the same label of nodes are represented as a self-loop. A self-loop is a relationship that has the same start and end node. On the right side of Figure 2.20, there are two self-loops present. The `FOLLOWS` relationship starts and points to nodes with a type `User`. Unfortunately, again, there is no agreed way of presenting when a relationship direction has semantic value (should be treated as directed or undirected) or not. For now, you have to read the fine print that comes along with the graph schema visualization.

2.3 Extracting knowledge from text

You have learned how to construct a graph model based on the graphy features of the data. Next, you will learn how you can extract relevant information from the text and incorporate them into your knowledge graph model. To get a sense of what information you can extract from the tweet content, let’s look at an example tweet.

Figure 2.21. An example tweet where mentions, links, and hashtags are present



Marcos Colebrook @MColebrook · Feb 9 •••

Making sense of news — the #KnowledgeGraph way: How to combine #NamedEntityLinking w/ #Wikipedia data enrichment to analyze the Internet news medium.com/neo4j/making-s... by @tb_tomaz > @CatedraBOB #NLP #DataScience #Graphs

Making Sense of News, the Knowledge Graph Way

How to combine Named Entity Linking with Wikipedia data enrichment to analyze the internet news.

medium.com

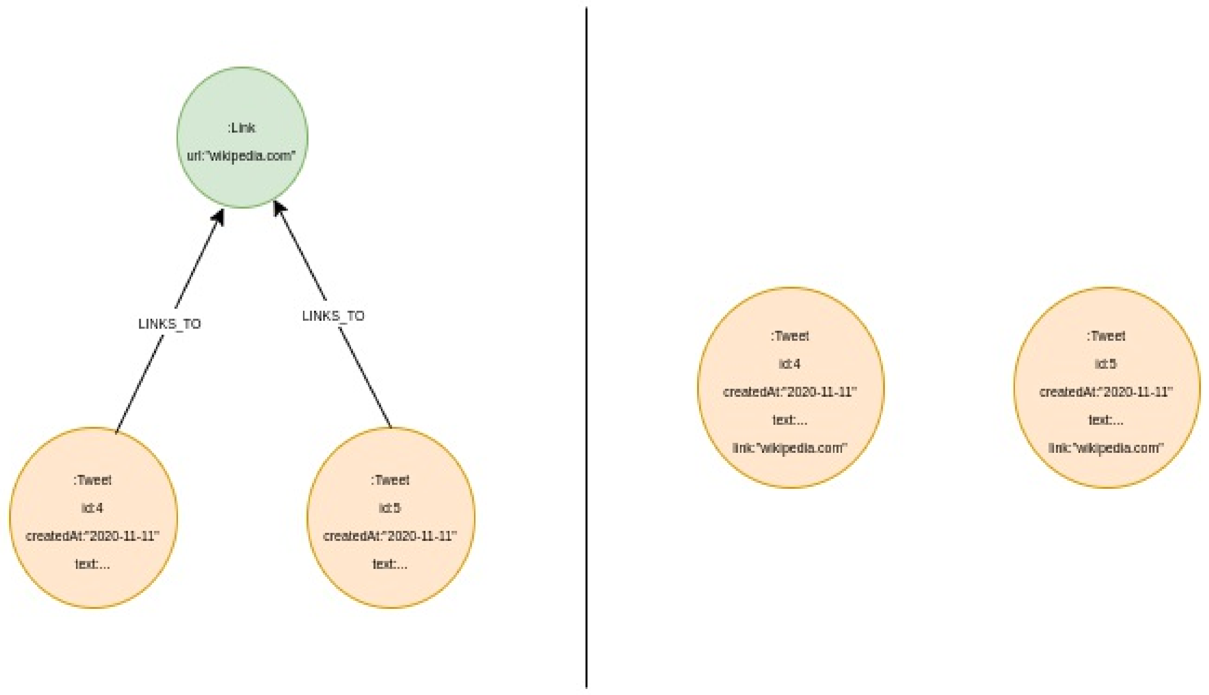
2.3.1 Links

The first information you could extract from tweet content is any link or URLs mentioned in the tweet. It should be a pretty straightforward process to process any links from the text.

Now that you have some experience developing a labeled-property graph (LPG) model, what do you think is the best approach to add the link information in the Twitter graph model? You have already defined the tweet in the model, now you only need to add the extracted links information to them.

I can think of two options. You could either store the url as the tweet node property or you could store it as a separate node and add a relationship between the link and the tweet.

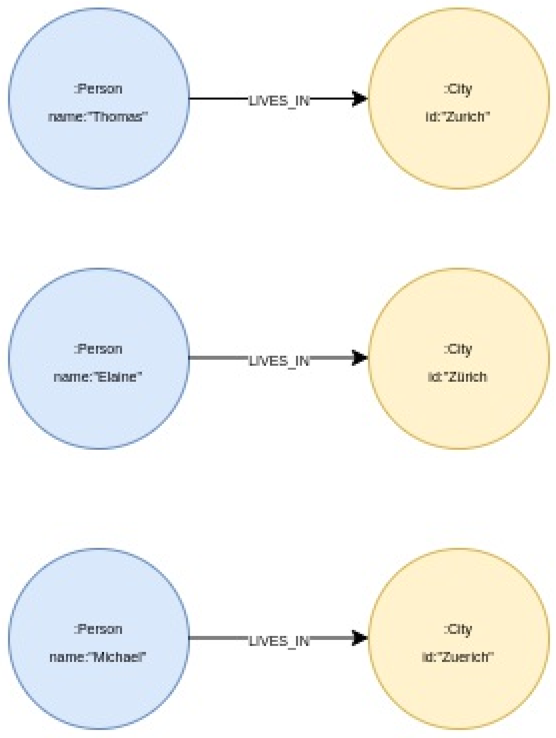
Figure 2.22. Two options for storing the extracted link from the tweet



Which approach do you think is better? With both approaches, you have the option to store one or many links as you can use a list of strings as a node property. The real question depends on your queries. Do you want to group tweets with the same links in your analysis or not? If you store links as node properties, it will be computationally more expensive to find tweets with the same link as you need to compare all elements in a list between all pairs of tweets. It would be similar to a relational database join, where you scan foreign keys at query runtime to find intersections of joins between pairs of nodes. You can avoid that by storing links as separate nodes. By using separate nodes to represent links, you only need to traverse two relationships to get from one tweet to another or even all tweets with the same links. This approach will scale much better if you are interested in grouping nodes with the same links.

When considering whether you want to store information as separate nodes or node properties, one thing to note is to examine if the values are standardized. A typical example I've seen is with city names.

Figure 2.23. An example graph, where city names that are not standardized

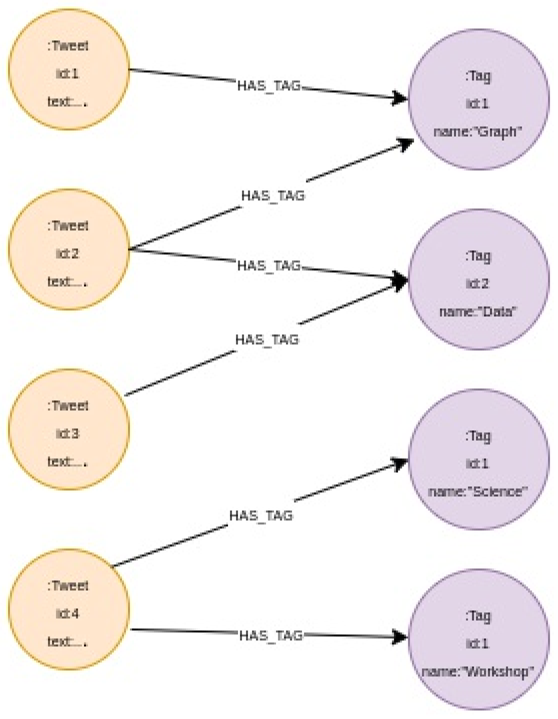


When discrete values are not standardized, storing that information as separate nodes don’t make sense. The whole point of using separate nodes to store information is to allow faster traversals at query runtime. When values are not standardized, you lose the ability to traverse fast between persons that live in the same city. A rule of thumb is that a single real-world entity or concept should be represented as a single node in the graph. In Figure 2.22, Zürich is represented as three different nodes, which would return invalid information if you were trying to find persons who live in the same city. While storing the city information as a node property does not solve the issue of finding people who live in the same city, at least you don’t represent a single real-world entity as multiple nodes in your graph. Another consideration to take into account is how specific that information is. For example, suppose the information is very unspecific and doesn’t add much value from the information point of view, like the gender of a user. In that case, it is better to store that information as a node property. One reason for that is that you avoid having nodes that can be connected to large parts of the graphs. With the gender example, you could have nodes connected to almost half of the users in the graph. Nodes connected to large parts of the graphs are called *super nodes*, and you generally want to avoid them in your graph as they can hinder query performance.

2.3.2 Hashtags

The other information you can extract from tweet content is the hashtags. People use the hashtag symbol (#) before a relevant keyword or phrase in their tweet to categorize them. A tweet can contain many hashtags. It makes sense to store hashtags as separate nodes and connect tweets to them.

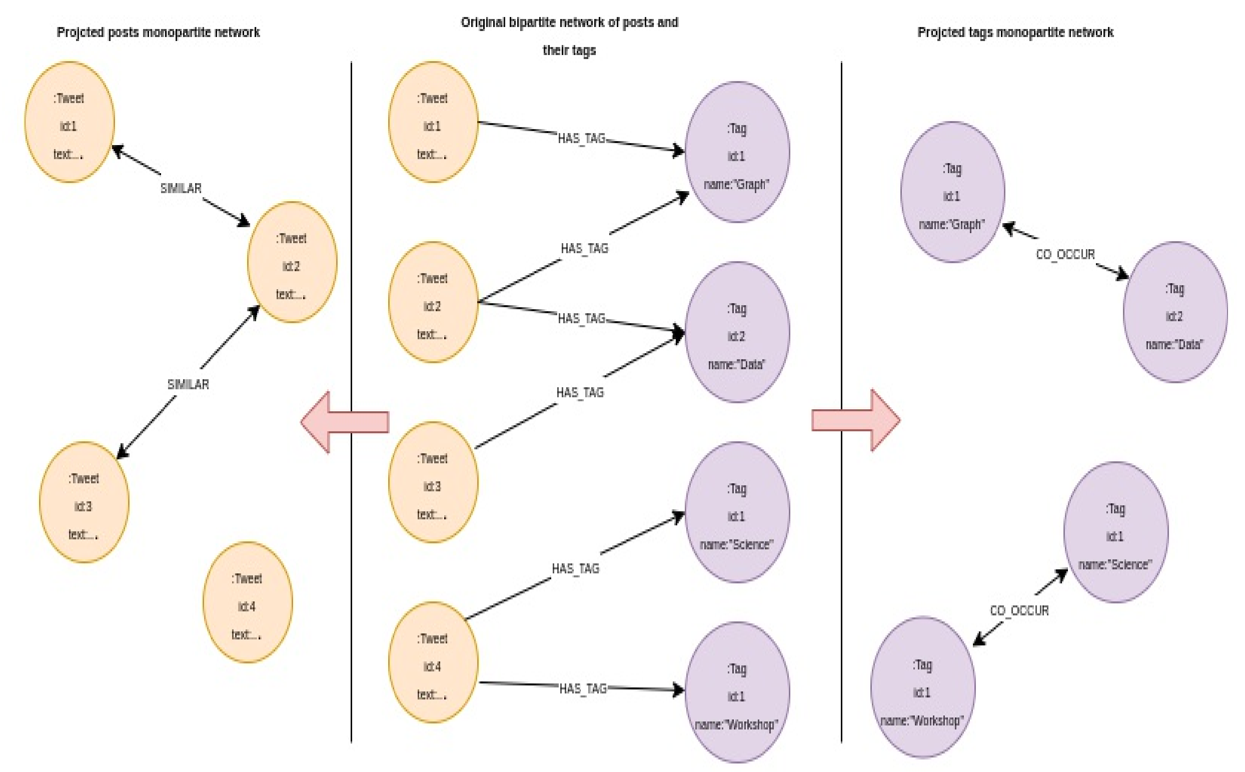
Figure 2.24. Graph model of tweet’s hashtags



An important consideration is that you want to avoid generic relationship types like `HAS`, where you could use it in many scenarios. You want your relationship types to have meaning, for example, if you traverse a `LINKS_TO` relationship from a tweet, you will always land at `Link` node. Similarly, if you traverse a `HAS_TAG` relationship from a tweet, you will always arrive to a `Tag` node. What you don't want to end up is with a single relationship type that can lead to many different node types. Using a generic relationship type can hinder the expressiveness of a graph model and also negatively affect query processing times.

The graph in the Figure 2.23 is a bipartite network consisting of tweet and hashtag nodes. When dealing with a bipartite network, it is common analysis workflow step to project it to a monopartite network. This process is also called network folding. While network folding is not a frequently used term, I have seen Jure Leskovec from Stanford use it, so I added it for completeness. For example, if a pair of tweets share the same hashtag, you could assume that they are somehow connected. It is a similar process as you have seen in the retweet network, where you translate indirect graph patterns to direct relationships.

Figure 2.25. Twitter hashtag network folding or monopartite network projection



You can observe that you always have the option to project a bipartite network to both types of nodes. In a bipartite network of tweets and their hashtags, you can project it to a monopartite network of tweets or hashtags. How do you name the new inferred relationship depends on the domain. I have added a similar relationship between tweets as I assume that tweets with more overlapping hashtags can be deemed more similar. On the other hand, hashtags can also be similar if they frequently co-occur. To put it in text representation of networks, you can transform the following indirect graph pattern:

(:Tweet)-[:HAS_TAG]->(:Tag)<-[:HAS_TAG]-(:Tweet)

to a more direct graph pattern:

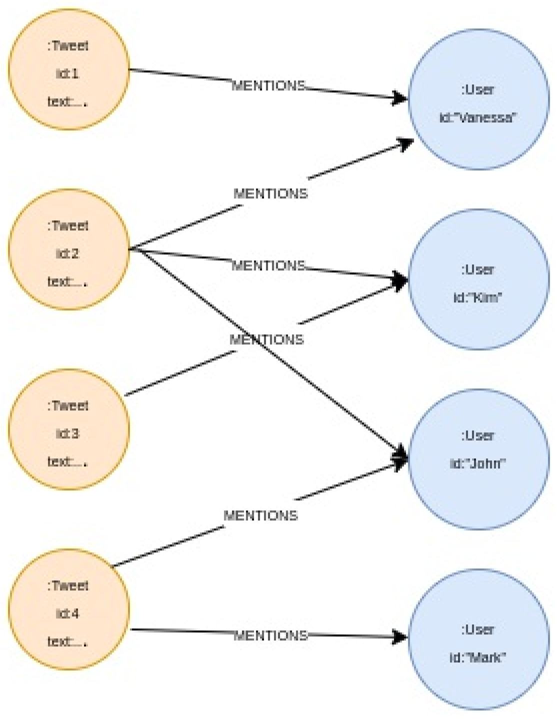
(:Tweet)-[:SIMILAR]-(:Tweet)

You might have noticed that a common approach to network analysis is to reduce a complex graph pattern to a network with a single type of nodes and relationships. This is because most classical graph algorithms like centrality or community detection algorithms are designed to have a monopartite network as an input. With monopartite projections, the direction of the relationship often does not hold any semantic value. If tweet A is similar to tweet B, then also tweet B is similar to tweet A. You can also quantify the strength of similarity between two tweets by counting the number of common hashtags they have. As a result, most inferred similarity networks are undirected and weighted, like in this example. On the other hand, previously inferred amplification network based on retweets is directed and also weighted. You could analyze the inferred similarity network of tweets and try to find users who publish similar content. Note that you could also combine information from other parts of the graph to infer a new monopartite network.

2.3.3 Mentions

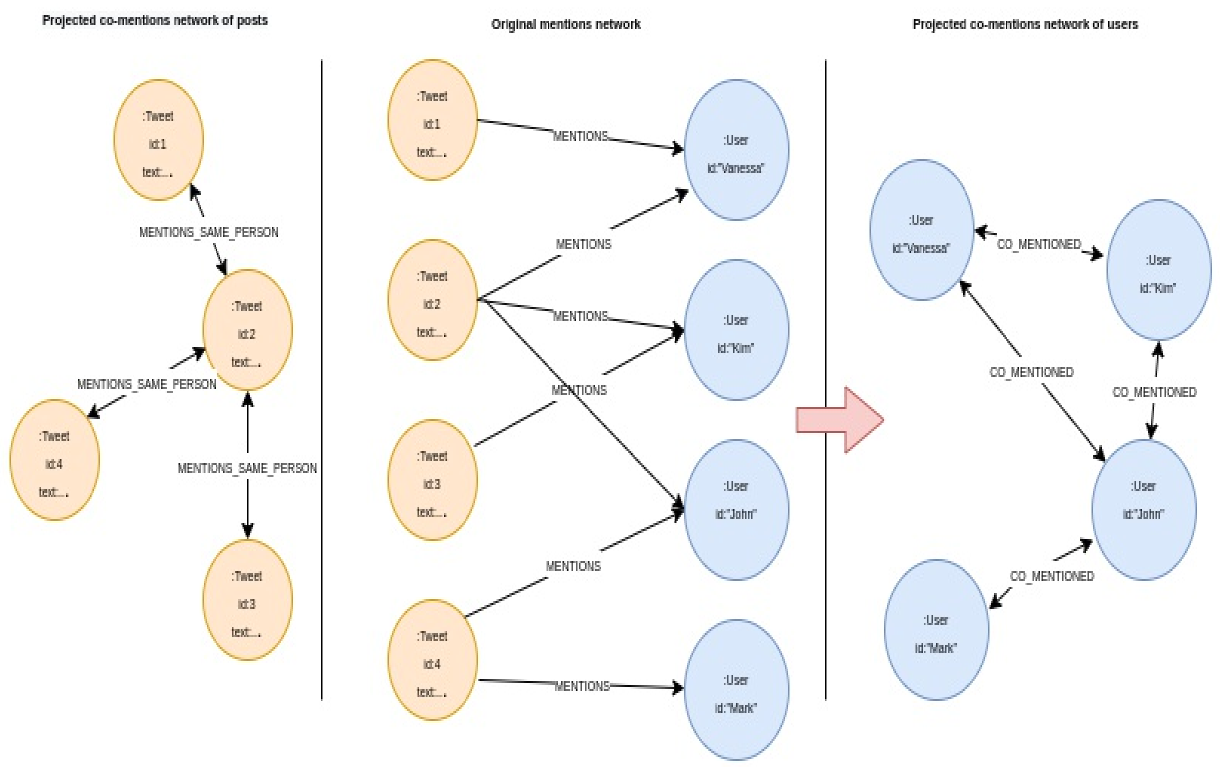
A user can mention other users in his tweet by using the mention symbol (@). A mention can be understood as an invitation to comment or a callout, and other times you only want to notify users to look at specific content. You already have users defined in the graph schema, so it only makes sense to connect tweets to mentioned users.

Figure 2.26. Graph model of tweets’s mentions



Similar to hashtag network, the mention network is also a classic bipartite network. As such, you can project it to a monopartite network of users or tweets.

Figure 2.27. Twitter mention network folding or monopartite projection



For example, you could analyze which users are frequently co-mentioned and try to examine the community structure of the inferred co-mention network between users. Another thing you could examine is if the mentioned person interacted with the tweet or not. You could also combine mention information with other information in the graph and inspect the most common hashtags of tweets where a user is being mentioned. One thing you could also do is to take the mention information and turn them into follower recommendations. Suppose a user is notifying you about specific concent, but you don’t already follower the content author. In this case, one could recommend you follow them directly.

2.3.4 Final Twitter social network schema

Designing graph model schema is an iterative process. You have slowly added additional information to the graph model, and it gradually became richer in knowledge. While adding new data to the graph model, designing a self-describing graph schema is recommended. With a self-describing graph model, you can avoid additional work to create a schema manual for others to learn about the information the graph stores and how to query it. Lastly, the graph schema might change depending on the queries you will be executing, as you might want to optimize the performance of specific queries. You can observe the following graph structure if you put all the graph model design considerations you made so far into a single visualization.

Figure 2.28. Twiter social network with extracted knowledge from text

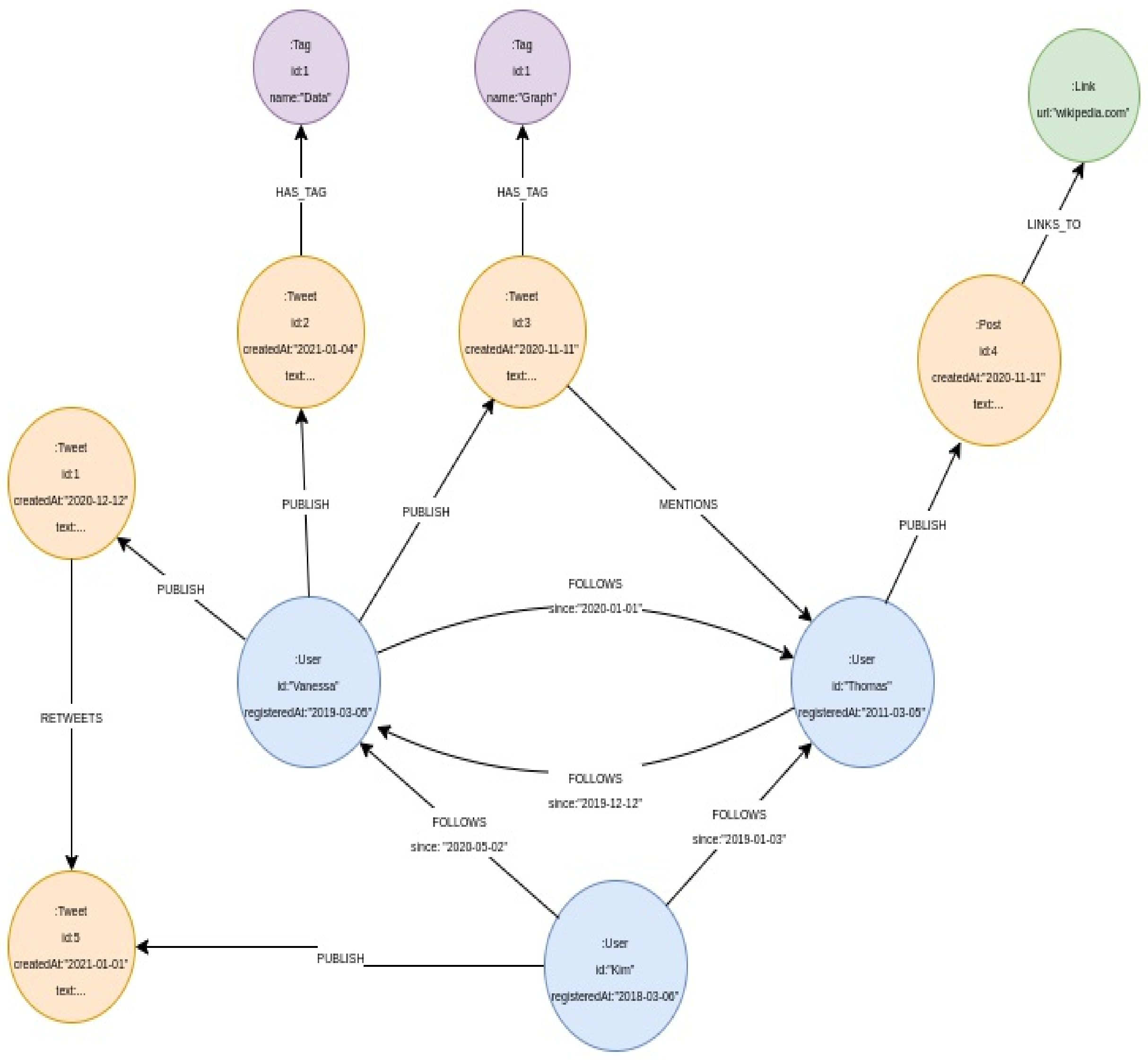
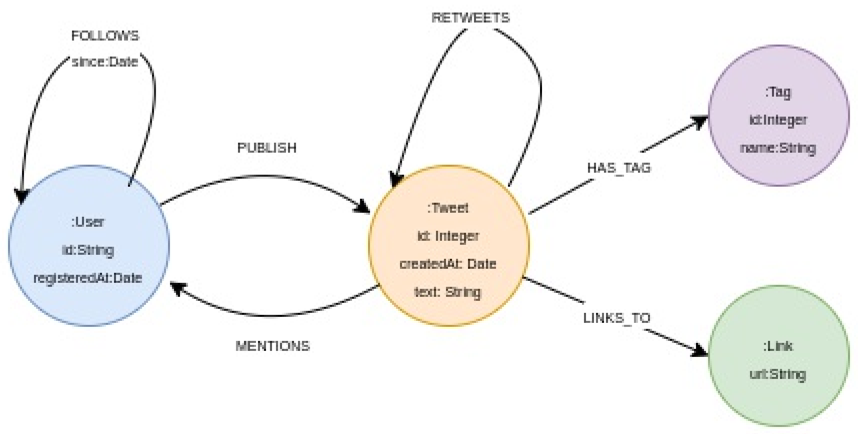


Figure 2.27 visualizes an example Twitter network, where there are four distinct types or labels of nodes present. There are users, tweets, hashtags, and link nodes in the final Twitter graph model. Along the way, you have

also introduced six different types of relationships. If you sum it up, you can represent this example network with the following graph schema:

Figure 2.29. Final Twitter social network graph schema representation



You only want to add the inferred relationships that will be actually instantiated to the graph schema. Inferred relationships and similarity networks are created based on assumptions you might make during the network analysis. You don’t know which inferred relationships will be instantiated yet, so it makes sense to leave them out of the graph schema for now.

You might have noticed that a common theme to network analysis is to translate indirect graph patterns and relationships to direct ones. Using a graph database with a dedicated graph-pattern query language makes it easier for you to instantiate those network transformations. For example, you can translate the retweet links between tweets to direct amplification relationships between users. Another frequent scenario is translating a bipartite network to a monopartite network. The reason behind monopartite projections is that most graph algorithms are designed to work on networks with a single type of node and relationships. In the next chapter, you will learn the basics of Cypher and how to import a network based on the graph model you derived in this chapter.

2.4 Summary

- Mathematical data structures to represent graphs are adjacency matrix and edge list
- Most graph databases fall into either the RDF or the LPG categories
- Graph modeling is an iterative process
- Graph model can be represented with the graph model schema visualization
- You can extract knowledge from text with simple text processing
- Reducing indirect graph patterns to direct relationships is a common approach in network analysis

3 Your first steps with the Cypher query language

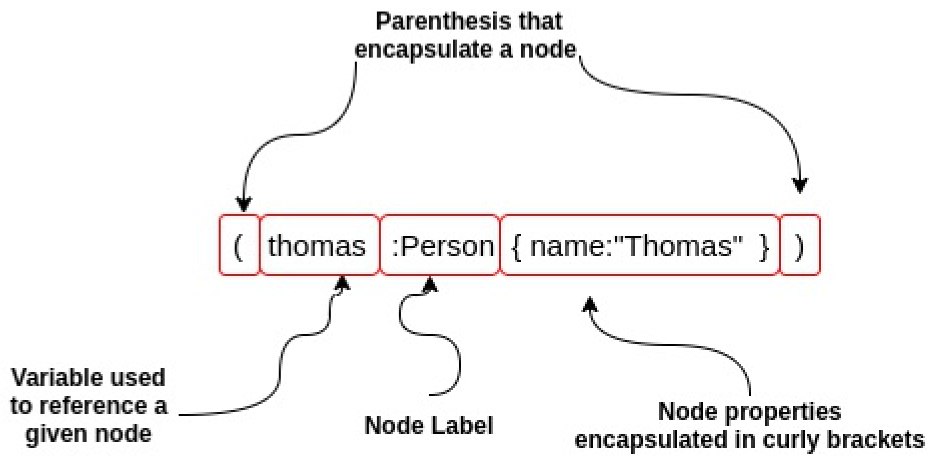
This chapter covers:

- Introducing the Cypher query language
- Cypher query clauses
- Best practices for importing a CSV into a graph database

So far, you have learned a bit of graph theory and how to approach the property graph modeling process. Now, you will begin to learn how to perform network analysis through practical use cases. To follow the examples in this book, you need to set up a Neo4j development environment. If you need some help with the setup, I have added the Neo4j Development Environment appendix to help you get started.

This chapter will introduce Cypher query language clauses and best practices for importing data into a graph database. First, I will do a quick recap of using Cypher query language syntax to represent networks in a textual format. If you are already familiar with the Cypher query syntax, you can skip most of the chapter and just import the data as shown in the last section. Remember from the previous chapter, the Cypher syntax uses parenthesis to encapsulate a representation of a node.

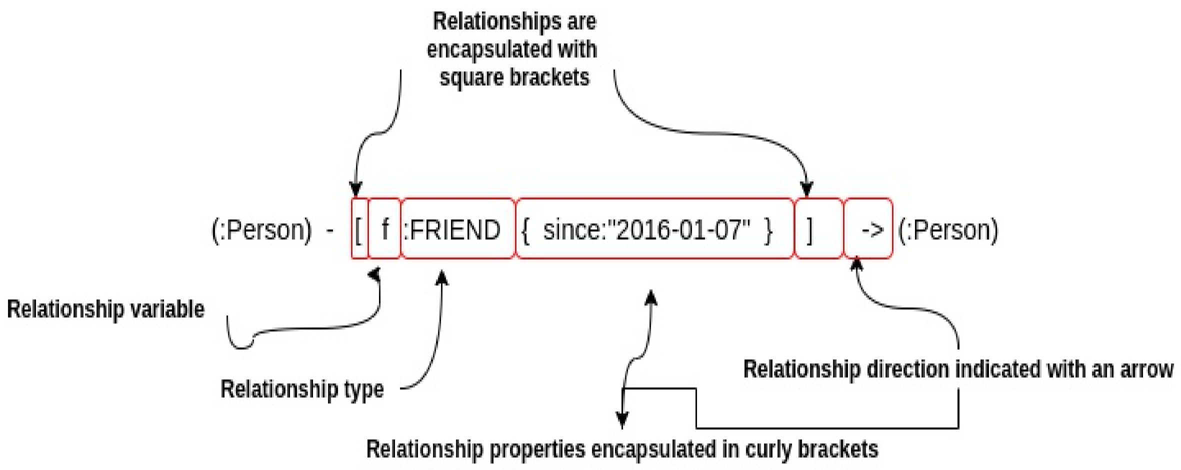
Figure 3.1. Cypher query language syntax to represent a node with its label and properties


Parenthesis that encapsulate a node
(
thomas
:Person
{ name:"Thomas" }
)
Variable used to reference a given node
Node Label
Node properties encapsulated in curly brackets

In this example, I have described a node with a label `Person`. The label of the node is always preceded by a colon. The node properties are key-value pairs wrapped inside the curly brackets. The example node has only a single property with a key `name` and its value "Thomas". In Cypher, you can also add a variable at the start of the node. A variable is used as a reference to the specific node. You can choose any name for the reference variable. In this example, I have chosen reference variable to be `thomas`. The node variable allows you to refer to the particular node later in the Cypher statement, and it is only valid within the context of a single Cypher statement. You can use the node variable to access its properties, labels, use it in expressions, or create new patterns related to a given node.

In Cypher, the relationships are represented with square brackets.

Figure 3.2. Cypher query language syntax to represent a relationship with its type and properties


Relationships are encapsulated with square brackets
(:Person) - [f :FRIEND { since:"2016-01-07" }] -> (:Person)
Relationship variable
Relationship type
Relationship direction indicated with an arrow
Relationship properties encapsulated in curly brackets

A relationship can only exist when it is adjacent to a source and target node. When you are describing a relationship with Cypher, you always need to include the adjacent nodes. Each relationship has a single type. Similar to node labels, the relationship type is also preceded by a colon. The example in Figure 3.2 describes a relationship with a type `FRIEND`. Relationship properties are described like node properties, and each relationship can be assigned a variable, in this example `f`, that can be used later in the Cypher statement to refer to the given connection.

In Neo4j, each relationship is stored as directed. You can, however, ignore the relationship direction when executing a Cypher query or a graph algorithm on top of the stored graph. A common practice in Neo4j is to store an undirected relationship as a single directed relationship. When executing Cypher queries, you can then ignore the direction of the relationship and treat it as undirected. While this is a relevant aspect in Neo4j, it is hard to understand at first, so I will show you practical examples of how to differentiate between storing the graph and how you query it throughout the book.

3.1 Cypher query language clauses

Armed with the knowledge of how to describe a node and a relationship in Cypher, you will now begin to learn the Cypher clauses. To follow along with the examples, you need to have a working Neo4j environment ready. I recommend you use the Neo4j Browser to execute Cypher queries. Again, if you need some help with setting up Neo4j environment and accessing the Neo4j Browser, I suggest you look at the Neo4j Development Environment appendix.

3.1.1 RETURN clause

The first Cypher clause you will learn is the *`RETURN`* clause. While Cypher query language clauses are not case-sensitive, it is preferred to write Cypher clauses in upper case for easier readibility. Find more information about Cypher style guide by following this link. The `RETURN` clause is used to retrieve information from the database via a graph query. There can be only a single `RETURN` clause in a Cypher query and only as the last clause of query. Exceptions, where you can have multiple RETURN clauses in a query, are unions and subqueries.

Execute the following Cypher query in Neo4j Browser.

Listing 3.1. An example of a RETURN clause, where a key "name" is retrieved

```
RETURN 'Alicia' AS name
```

Executing the Cypher query in the Listing 3.1 returns a table with a single row and a single column, where `name` is title of the column and `Alicia` is its only value. The mentioned query also uses the *`AS`* operator. With the `AS` operator, you can name or alias a variable reference.

3.1.2 WITH clause

Using the *`WITH`* clause, you can manipulate the data as an intermediate step before passing the results to the next part of the Cypher query. The intermediate data manipulations within a query before passing them on to the next part can be one or more of the following:

- filter results
- select results
- aggregate results

- paginate results
- limit results

Listing 3.2. An example of a WITH clause, where the WITH clause is used to define new data references

```
WITH 3 AS x, 6 AS y
RETURN x * y AS result
```

In the Listing 3.2, the `WITH` clause is used to define `x` and `y` variables and the `RETURN` clause returns a multiplication value of `x` and `y`.

As an exercise, try to define your first and last name in the `WITH` statement and return your full name in the `RETURN` statement. The structure of the query should be very similar to the query in Listing 3.2. You can concatenate strings in Cypher using the `+` operator.

There can also be multiple `WITH` clauses in sequence in a single query. Using multiple `WITH` statements is useful when you have to do a lot of output manipulation as an intermediate step of the Cypher query.

Listing 3.3. An example of using two WITH clauses in a sequence

```
WITH 'Tomaz' AS first_name
WITH 'Bratanic' AS last_name, first_name
RETURN first_name + ' ' + last_name AS result
```

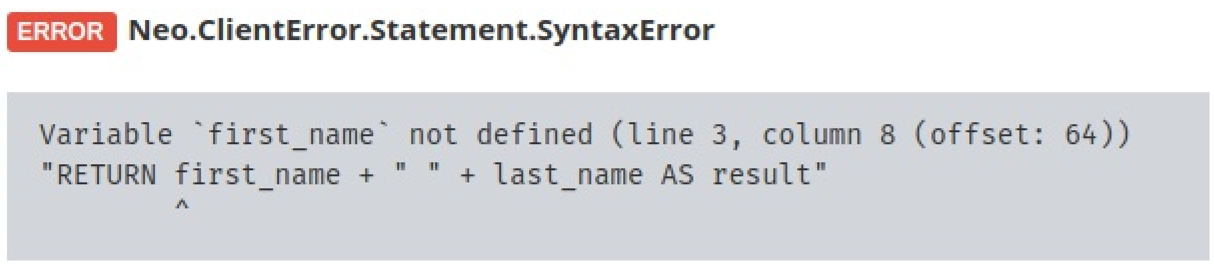
In Listing 3.3, the `first_name` variable is defined in the first `WITH` clause and the `last_name` variable in the second `WITH` clause. You could define them both in a single `WITH` clause, but I have decided to split them into two clauses for this demonstration. You might have noticed why I have included the `first_name` variable in the second `WITH` clause as well. The `WITH` clause affects the variables in scope. Any variables not included in the `WITH` clause are not carried over to the next part of the query. I think that as a part of becoming familiar with any tool is to get familiar with the errors.

Execute the following query to generate your first error.

Listing 3.4. An example cypher query that raises a missing variable definition error

```
WITH 'Michael' AS first_name
WITH 'Jackson' AS last_name
RETURN first_name + ' ' + last_name AS result
```

Figure 3.3. Cypher syntax error where a variable is not defined



```
ERROR Neo.ClientError.Statement.SyntaxError

Variable `first_name` not defined (line 3, column 8 (offset: 64))
"RETURN first_name + " " + last_name AS result"
        ^
```

Anytime you see a variable not defined error, you have either misspelled a variable or name or forgot to include it in the `WITH` clause.

The `WITH` clause is also frequently used to filter intermediate results. Adding the `WHERE` clause, you have the ability to filter the results of the `WITH` statement.

Listing 3.5. An example cypher query that uses the WHERE clause to filter results of a WITH statement

```
WITH 'Elon' AS first_name, 'Musk' as last_name
WHERE first_name = 'Elon'
RETURN *
```

Cypher query in Listing 3.5 uses the `WHERE` clause to filter only results where the `first_name` variable is equal to "Elon". You might have noticed that the mentioned query uses a wildcard operator `*` in the `RETURN` clause. The wildcard operator `*` will return all variables that are in scope and it can also be used in the `WITH` clause. An important thing to note is that a `WHERE` clause only looks at and filters the output of the `WITH` statement and cannot stand on its own.

3.1.3 CREATE clause

The *`CREATE`* clause is used to store nodes and relationships in the graph database. Using Cypher syntax to describe node and relationship patterns, you can store any graph pattern you can imagine.

You will begin by creating a single node in the graph.

Listing 3.6. Cypher query that stores a node with a label Person and a property "name" with value "Satish"

```
CREATE (p:Person{name:'Satish'})
RETURN p
```

The statement in Listing 3.6 creates a new node with a label `Person` and a single node property. You can observe that adding a reference variable to the new node allows you to return it as an output of the query. With every execution of the query in Listing 3.6, a new node will be created in the database. The `CREATE` clause does not check for existing data in the graph database. It blindly follows the command to create a new pattern in the database.

To get some exercise with graph creation, create a new node in the graph with a label `Person` and two node properties. The first node property holds the information about your name and the second node property should hold the information about your age.

You can use multiple `CREATE` clauses in a single Cypher statement. By using the variable of nodes and relationships, you can modify and connect them in the subsequent `CREATE` queries.

Listing 3.7. Cypher query that stores two nodes and a relationship between them in the database

```
CREATE (elaine:Person{name:'Elaine'}), (michael:Person {name: 'Mi
CREATE (elaine)-[f:FRIEND]->(michael)
RETURN *
```

Cypher statement in Listing 3.7 demonstrates how to create two nodes and a relationship between them. In the first `CREATE` clause, two nodes are created, and in the second `CREATE` clause, a relationship between them is added. While you could combine these two `CREATE` clauses into a single clause, it is recommended as best practice to create nodes and relationships separately.

Try to create two nodes, one that represents you and one that represents the organization you work at. Probably, you will want to use different node labels to describe a person and an organization. In the same Cypher statement, then also create a relationship between yourself and your employer. You can try to add a relationship property indicating when you started to work for your current role.

Remember, you can only store directed relationships in the Neo4j graph database. Let’s see what happens when you try to create an undirected relationship.

Listing 3.8. Cypher query that tries to stores an undirected relationship in the database and fails

```
CREATE (elaine:Person{name:'Elaine'}), (michael:Person {name: 'Mi
CREATE (elaine)-[:FRIEND]-(michael)
RETURN *
```

As mentioned, this query fails due to only being able to store directed relationships in the database. While the relationship direction arrow seems such a small part of the query, it is very influential to how the query will behave.

Another common misconception among beginners is that they forget that the reference variables are only visible within the same query and have to be defined in the intermediate `WITH` scopes if there are any. As stated previously, the `CREATE` statement performs no database lookup before inserting new data in the graph. The following Cypher query looks ok at first glance, but it is actually very terrible.

Listing 3.9. Cypher query that stores two empty nodes with no labels and a relationship between them in the database

```
CREATE (ankit)-[f:FRIEND]->(elaine)
RETURN *
```

Can you deduce why? As the `CREATE` statement performs no database lookups and does not have variable reference visibility between Cypher queries, it will just create a new pattern we have described. Interestingly, the query in Listing 3.9 creates a `FRIEND` relationship between two nodes with no labels and no properties. You have to be very careful to avoid these types of situations. There are no situations in the LPG model where you would want to have nodes without any label stored in the database. At a minimum, you can add a generic Node label to each node.

The labeled-property graph model is so flexible that it allows you to create nodes without labels or properties. You should always strive to add at least a label to every node you store in the database. Having labeled nodes will help with model readability and also query execution performances. I can safely say that if you have nodes without a label in your graph, something is wrong with either your model or your import process.

The goal of this exercise is to create three nodes and two relationships between them. The nodes should represent the city, country, and continent you currently live in. Add a relationship between the city and the country, and the second relationship between the country and the continent. Take a couple of minutes to decide what relationship types you want to use and the direction of relationships.

3.1.4 MATCH clause

Using the *`MATCH`* clause, you can search for existing graph patterns stored in the database. Cypher is a declarative query language, which means you only need to specify the pattern you are interested in and let the query engine take care of how to retrieve those patterns from the database. In the previous section, you have created at least three nodes with the label `Person` and different `name` property values. If you want to find a `Person` node with a specific `name` property value, you can use the following query.

Listing 3.10. Cypher query that searches and retrieves any nodes with a label Person, that have a "name" property with a value of "Satish"

```
MATCH (p:Person {name:'Satish'})
RETURN p
```

You can observe why it was critical first to learn how to describe node and relationship patterns with Cypher before you wrote your first Cypher clause. When you know how to describe a graph pattern, you can use the `MATCH` clause to retrieve it from the database. The query in Listing 3.10 uses the so-called *inline* graph pattern matching. Inline pattern matching uses Cypher pattern syntax to describe a node or relationship pattern with its labels and properties. The opposite of inline pattern matching is using a `WHERE` clause to describe a graph pattern.

Listing 3.11. Cypher query that searches and retrieves any nodes with a label Person, that have a "name" property with a value of "Satish" using a WHERE clause

```
MATCH (p)
WHERE p:Person AND p.name = 'Satish'
RETURN p
```

The query in Listing 3.11 will produce the exact same query plan and results as the query in Listing 3.10. The inline syntax is just syntactic sugar that reads better for humans. Using the `WHERE` clause, you have described you want to retrieve a node with a label `Person` and a `name` property with a value of "Satish". While inline graph pattern matching is limited to the equality operator, meaning you can only describe a node with an exact label and node properties, the `WHERE` clause allows for more flexibility. My personal preference is to describe the node label with inline graph pattern and provide additional matching filters in the `WHERE` clause.

Listing 3.12. Cypher query that combines inline graph pattern matching with a WHERE clause to describe a graph pattern

```
MATCH (p:Person)
WHERE p.name = 'Satish' OR p.name = 'Elaine'
RETURN p.name AS person
```

As an exercise, try to retrieve all the nodes with a label `Person` from the database. You can use the inline graph pattern description, or you can use a `WHERE` clause. In the `RETURN` statement, only return the `name` properties of the nodes.

You can always have multiple `MATCH` clauses in a sequence. Similar to the `WITH` clause, the `WHERE` clause only applies to the previous `MATCH` clause. If you use many `MATCH` clauses in a sequence, make sure to append a `WHERE` clause to each `MATCH` clause where needed.

Listing 3.13. Cypher query that combines inline graph pattern matching with a WHERE clause to describe a graph pattern

```
MATCH (satish:Person)
WHERE satish.name = 'Satish'
MATCH (elaine:Person)
WHERE elaine.name = 'Elaine'
RETURN *
```

A `WHERE` clause can only exist when it follows a `WITH`, `MATCH`, or an `OPTIONAL MATCH` clause. When you have many `MATCH` or `WITH` clauses in sequence, make sure to append the `WHERE` clause after each of them where needed. You might sometimes get the same results even if you only use a single `WHERE` clause after multiple `MATCH` statements, but the query performance will most likely be worse. You can learn more about query performance and optimization in the Cypher profiling appendix.

The `MATCH` clause is often used to find existing nodes or relationships in the database and then insert additional data with a `CREATE` or `MERGE` clause. For example, you could use the `MATCH` clause to find nodes labeled `Person` with names "Elaine" and "Satish" and create a new relationship between them.

Listing 3.14. Cypher query that find two nodes in the database and creates a new FRIEND relationship between them

```
MATCH (from:Person), (to:Person)
WHERE from.name = 'Satish' AND to.name = 'Elaine'
CREATE (from)-[:FRIEND]->(to)
RETURN *
```

The statement in Listing 3.14 combines the `MATCH` and the `CREATE` clause to create a new relationship between existing nodes in the database.

If you haven’t yet created a `Person` node with your name as the value of the `name` node property, please do that first. As a next step, in a separate query, use the `MATCH` clause to find the `Person` nodes with your name and "Elaine", that also needs to exist in your database, and create a new `FRIEND` relationship between them. You can add any additional relationship properties you think are appropriate.

A crucial concept when using the `MATCH` clause is to recognize that if a single `MATCH` clause within the query does not find any data matching the provided pattern in the database, the query will return no results. If you use a single `MATCH` clause to retrieve a non-existing graph pattern from the database, you will get no results.

Listing 3.15. Cypher query that matches a non-existent graph pattern in the database

```
MATCH (org:Organization)
WHERE org.name = 'Acme Inc'
RETURN *
```

It is intuitive that when you try to retrieve a non-existent graph pattern from the database, you will get no results. What is not so intuitive is that when you have multiple `MATCH` clauses in sequence, if only a single `MATCH` clause tries to retrieve a non-existent pattern from the database, the whole query will return no results.

Listing 3.16. Cypher query that matches both an existing and a non-existing graph pattern in the database

```
MATCH (p:Person)
WHERE p.name = 'Satish'
MATCH (org:Organization)
WHERE org.name = 'Acme Inc'
RETURN *
```

The query in Listing 3.16 first tries to find a `Person` node with a `name` property "Satish". You have already executed this part of the query before, so you know that this pattern exist in the database. The second `MATCH` clause tries to retrieve a non-existent pattern from the database. If only a single `MATCH` clause in the query retrieves no pattern from the database, the result of the query will be empty.

OPTIONAL MATCH clause

If you do not want your query to stop when a single `MATCH` clause finds no existing graph patterns in the database, you can use the *`OPTIONAL MATCH`* clause. The `OPTIONAL MATCH` clause would return a `null` value if no matching patterns were found in the database instead of returning no results, behaving similarly as an OUTER JOIN in SQL. You can rewrite the query in Listing 3.16 to expect and handle non-existing `Organization` pattern by using the `OPTIONAL MATCH` clause.

Listing 3.17. Cypher query that matches both an existing and a non-existing graph pattern in the database

```
MATCH (p:Person)
WHERE p.name = 'Satish'
OPTIONAL MATCH (org:Organization)
WHERE org.name = 'Acme Inc'
RETURN *
```

By using the `OPTIONAL MATCH` clause, the query does not return empty results when no graph patterns are found.

3.1.5 Set clause

A *`SET`* clause is used to update labels of nodes and properties of both nodes and relationships. `SET` clause is very often used in combination with the `MATCH` clause to update existing node or relationship properties.

Listing 3.18. Cypher query that uses a SET clause to update existing node properties

```
MATCH (t:Person)
WHERE t.name = 'Satish'
SET t.interest = 'Gardening',
    t.hungry = True
```

There is also a special syntax for `SET` clause to change or mutate many properties using a *map* data structure. The *map* data structure comes from Java and is identical to a dictionary in Python or a JSON object in Javascript.

Listing 3.19. Cypher statement that uses a map data structure in combination with the SET clause to update many node properties

```
MATCH (e:Person)
WHERE e.name = 'Elaine'
SET e += {hungry: false, pet: 'dog'}
```

Note that if the `+=` operator of the `SET` clause is replaced with only `=`, then it overrides all existing properties with only those provided in the map.

By now, there is hopefully a `Person` node with your name in the database. Use the `SET` clause to add additional node properties, such as the information about your favorite food or your pet name.

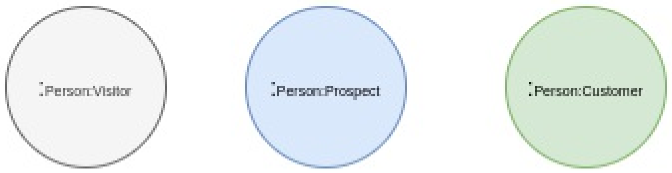
With the `SET` clause, you can also add additional labels to nodes.

Listing 3.20. Cypher query that adds a secondary label to an existing node

```
MATCH (t:Person)
WHERE t.name = 'Satish'
SET t:Author
```

Multiple node labels are helpful when you want to tag your nodes for faster and easier retrieval. In the example in Listing 3.20, you have added the `Author` label to the Satish node, and in the following exercise, you will add the `Reader` label to the node representing you. This way, you can easily differentiate between readers and authors in your database. A good guideline to follow when using multiple node labels is that node labels should be *semantically orthogonal*. Semantically orthogonal means that node labels shouldn't hold the same or similar meaning and should have nothing to do with one another. Seconodary node labels are used to group nodes into different buckets, so that each subset is easily accessible. In my work, I have also noticed that using multiple labels is helpful in scenarios where you pre-calculate some values and assign additional node labels based on those values. For example, if you work with customer in a marketing funnel, you can add the secondary label to a node according to its funnel stage.

Figure 3.4. Using multiple node labels to assign customer funnel stage



Match the `Person` node representing you, i.e., it has your name as the name property value, and add a secondary `Reader` label to it.

3.1.6 REMOVE clause

The *`REMOVE`* clause is the opposite of the `SET` clause. It is used to remove node labels and node and relationship properties. Removing a node property can also be understood as setting its value to `null`. If you want to remove the `hungry` property from the `Person` node with the name "Satish", you can execute the following Cypher query.

Listing 3.21. Cypher query that removes a node property from an existing node in the database

```
MATCH (t:Person)
WHERE t.name = 'Satish'
REMOVE t.hungry
```

With the `REMOVE` clause, you can also remove labels from existing nodes.

Listing 3.22. Cypher query that removes a node label from an existing node in the database

```
MATCH (t:Person)
WHERE t.name = 'Satish'
REMOVE t:Author
```

3.1.7 DELETE clause

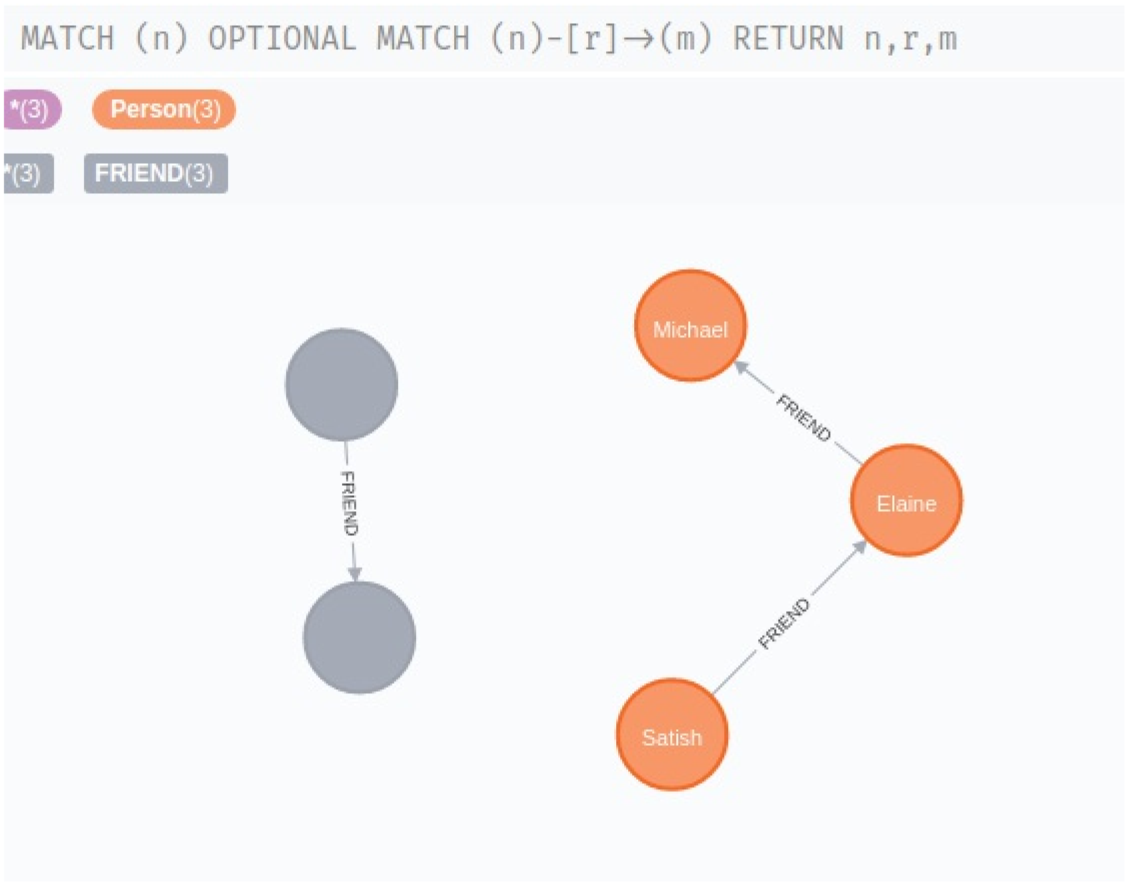
The *`DELETE`* clause is used to delete nodes and relationships in the database. You can first inspect the content of your graph database (if the graph is very tiny), with the following Cypher query

Listing 3.23. Cypher query that retrieves all nodes and relationships in the database

```
MATCH (n)
OPTIONAL MATCH (n)-[r]->(m)
RETURN n,r,m
```

If you run the query in the Listing 3.23 in Neo4j Browser, you should get a similar graph visualization to mine.

Figure 3.5. Visual representation of the current stored graph in database


```
MATCH (n) OPTIONAL MATCH (n)-[r]->(m) RETURN n,r,m
```

There are currently five nodes and three relationships in the database. You could have a few more nodes and relationships if you completed the exercises. That is no problem. First, you will delete the relationship between `Person` nodes with the `name` properties "Satish" and "Elaine". To perform a graph pattern deletion, you must first use the `MATCH` clause to find the graph pattern and then use the `DELETE` clause to delete it from the database.

Listing 3.24. Cypher query that deletes a relationship between Person nodes with the name properties "Satish" and "Elaine".

```
MATCH (n:Person)-[r]->(m:Person)
WHERE n.name = 'Satish' AND m.name = 'Elaine'
DELETE r
```

In Listing 3.24, the `MATCH` clause first matches any relationships directed from the `Person` node representing Satish to the node representing Elaine. Notice that I didn’t define any relationship type in the `MATCH` clause. When you omit the relationship type in the graph pattern description, the `MATCH` clause will search for relationships of any type between described nodes. Very similarly, you can also delete a node from a database.

Listing 3.25. Cypher query that deletes a single node from the database.

```
MATCH (n:Person)
WHERE n.name = 'Satish'
DELETE n
```

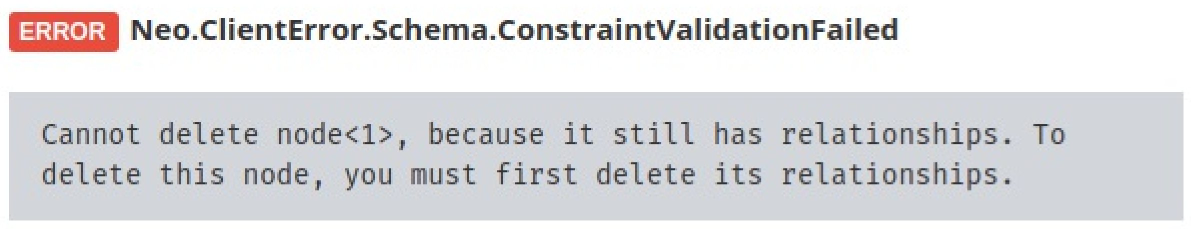
To get to know a new Cypher error, you can execute the following query.

Listing 3.26. Cypher query that deletes a single node from the database.

```
MATCH (n:Person)
WHERE n.name = 'Elaine'
DELETE n
```

You might wonder why you could delete the node representing Satish, but you cannot delete a node representing Elaine. Luckily, the error is very descriptive.

Figure 3.6. An error when you want to delete a node that has existing relationships to other nodes



ERROR Neo.ClientError.Schema.ConstraintValidationFailed

```
Cannot delete node<1>, because it still has relationships. To
delete this node, you must first delete its relationships.
```

You cannot delete a node that still has relationships attached to it.

DETACH DELETE clause

As deleting nodes with existing relationships is a frequent procedure, the Cypher query language provides a *`DETACH DELETE`* clause that first deletes all the relationships attached to a node and then deletes the node itself. You can try to delete the node representing Elaine with the `DETACH DELETE` clause.

Listing 3.27. Cypher statement that deletes a single node and all of its relationships from the database by using the DETACH DELETE clause

```
MATCH (n:Person)
WHERE n.name = 'Elaine'
DETACH DELETE n
```

This statement deleted both the relationships attached to the node as well as the node itself.

Try to delete the node representing yourself or the node representing a Person with the name "Michael". If the given node still has existing relationships, you must use the `DETACH DELETE` clause to first delete the relationships and then delete the node.

A Cypher statement that might come in handy when you are toying around with a graph database, hopefully not in production, is to delete all the nodes and relationships in the database.

Listing 3.28. Cypher statement that deletes all the nodes and relationships in the database

```
MATCH (n)
DETACH DELETE n
```

Query in Listing 3.28 will first use the `MATCH` clause to find all the nodes in the database. As you don’t include any node label in the node description, the query engine will return all nodes in the database. With the `DETACH DELETE` clause, you instruct the query engine first to delete all attached relationships to a node and then the node itself. Once the statement is finished, you should be left with an empty database.

3.1.8 MERGE clause

In this section, I will assume that you are starting with an empty database. If you still have data stored inside the database, please run the query in Listing 3.28.

The `MERGE` clause can be understood as a combination of using both `MATCH` and `CREATE` clauses. Using the `MERGE` clause, you instruct the query engine first to try to match a given graph pattern, and if it does not exist, it should then create this pattern.

Listing 3.29. Cypher query that uses MERGE clause to ensure that a Person node with a name Alicia exists in the database

```
MERGE (a:Person {name:'Alicia'})
```

The `MERGE` clause only supports inline graph pattern description and cannot be used in combination with a `WHERE` clause. The statement in Listing 3.29 ensures that a `Person` node with the `name` property "Alicia" exists in the database. You can rerun this query multiple times, and there will always be precisely a single `Person` node with the name "Alicia" in the database. A statement that can be rerun multiple times and always output the same results is also known as an *idempotent* statement. When you import data into the graph database, it is advisable to use `MERGE` instead of the `CREATE` clause. Using the `MERGE` clause, you don’t have to worry about later deduplication of nodes, and you can rerun a query multiple times without corrupting your database structure. What do you think will happen if we try to use `MERGE` clause to describe a `Person` node with the name "Alicia" and an additional node property location?

Listing 3.30. Cypher query merges a single node with two node properties

```
MERGE (t:Person {name:'Alicia', location:'Chicago'})
```

Try to match and retrieve all nodes in the database that have a `Person` label and a `name` property with value "Alicia".

The query to retrieve all nodes with a label `Person` and a `name` property "Alicia" is as follows:

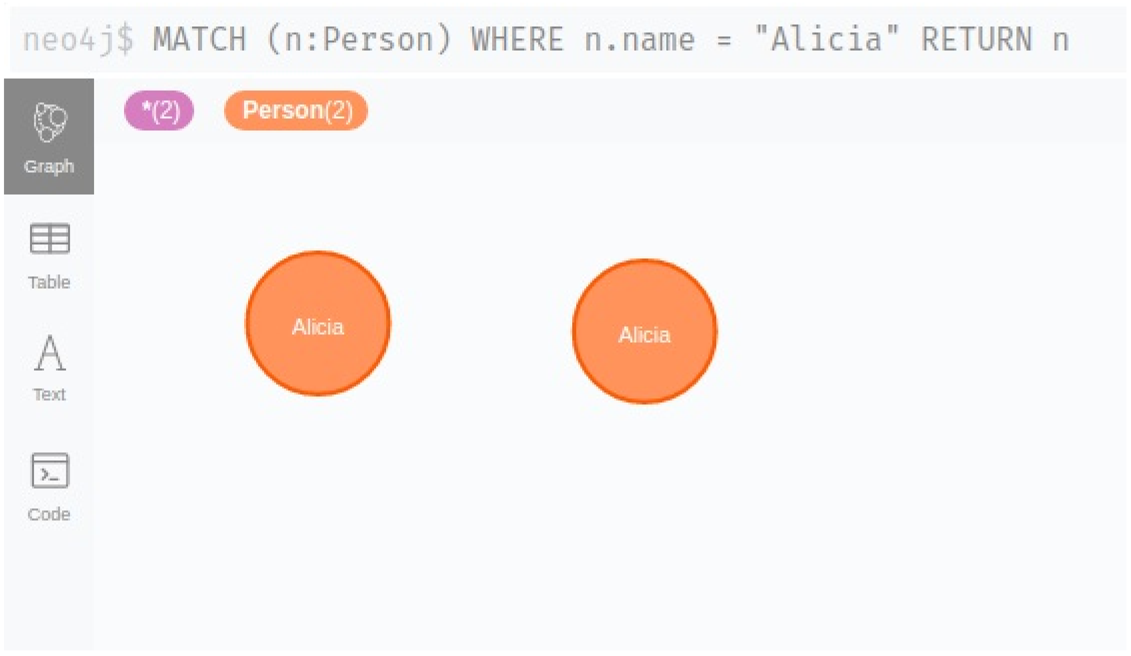
Listing 3.31. Cypher query that retrieves all nodes with a label Person and name property "Alicia"

```
MATCH (n:Person)
WHERE n.name = 'Alicia'
```

```
RETURN n
```

If you are using the Neo4j Browser interface, you can quickly observe in the graph visualization that there exist two nodes with a label `Person` and the `name` property "Alicia".

Figure 3.7. Results of the query in Listing 3.31



```
neo4j$ MATCH (n:Person) WHERE n.name = "Alicia" RETURN n
```

There are two `Person` nodes with the same `name` property in the database. Wasn't it mentioned just before that the `MERGE` clause is idempotent? Remember, the `MERGE` clause first tries to match an existing graph pattern, and only if it does not exist, it then creates the full given graph pattern. When you executed the query in Listing 3.30 to merge a `Person` node with two node properties, the graph engine first searched for the given pattern. A `Person` node with the `name` property "Alicia" and `location` property "Chicago" did not exist in the database at that moment. Following the `MERGE` logic, it then created a new `Person` node with those two properties.

When designing a graph model, a best practice is to define a unique identifier for each node label. A unique identifier consists of defining a unique property value for each node in the graph. For example, if you assumed that the `name` property of the `Person` nodes is unique, you could use the following `MERGE` clause to import `Person` nodes.

Listing 3.32. Cypher query that merges a node on its unique identifier property and then adds additional properties to the node

```
MERGE (t:Person{name:"Amy"})
ON CREATE SET t.location = "Taj Mahal", t.createdAt = datetime()
ON MATCH SET t.updatedAt = datetime()
```

A `MERGE` clause can be followed by optional *`ON CREATE SET`* and *`ON MATCH SET`*. In the `MERGE` clause, you have used the unique identifier of nodes to merge the nodes. If the node is created during this query, you can define additional node properties that should be set with the *`ON CREATE SET`* clause. Conversely, if the node with the label `Person` and the `name` property "Amy" already existed in the database, then the *`ON MATCH SET`* clause will be invoked.

The Cypher statement in Listing 3.32 will first merge a `Person` node with the `name` property "Amy". If a node with a label `Person` and the `name` property "Amy" does not exist in the database before, then the `MERGE` clause will create one and invoke the `ON CREATE SET` clause to set the `location` and `createdAt` properties on the node. Suppose the mentioned node already existed in the database. In that case, the query engine will not create any new nodes, and it will only update the `updatedAt` node property with the current time as described in the `ON MATCH SET` clause. The *datetime()* function in Cypher returns the current time.

Very frequently, your import query will look like the following:

Listing 3.33. Cypher query that merges two nodes and then merges a relationship between them

```
MERGE (j:Person{name:"Jane"})
MERGE (s:Person{name:"Samay"})
MERGE (j)-[:FRIEND]->(s)
```

The most frequent Cypher structure when importing data into Neo4j is first to merge the nodes separately and then merge any relationships between them. Using the `MERGE` clause, you don’t have to worry about data duplication or multiple query executions. The statement in Listing 3.34 will ensure that the three graph patterns with two nodes describing Jane and Samay and a `FRIEND` relationship exists between them. You can rerun this query multiple times and the output will always be the same.

The `MERGE` clause also supports merging an undirected relationship. This fact is a bit confusing. At the beginning of the chapter, I mentioned that you can only store a directed relationship in the Neo4j database. Let’s see what happens if you run the following query.

Listing 3.34. Cypher query that merges two nodes and then merges an undirected relationship between them

```
MERGE (j:Person{name:"Alex"})
MERGE (s:Person{name:"Andrea"})
MERGE (j)-[f:FRIEND]-(s)
RETURN *
```

You can observe that two new `Person` nodes were created. When you describe an undirected relationship in the `MERGE` clause, the query engine first tries to match the relationship while ignoring the direction. Practically, it searches for a relationship in both directions. If there are no relationships between the nodes in any direction, it then creates a new relationship pointing from the left to the right node. Having the ability to describe an undirected relationship in the `MERGE` clause allows us to import undirected networks more conveniently. If you assume that the `FRIEND` relationship is undirected, meaning that if Alex is friends with Andrea, then also Andrea is friends with Alex, then you could only store a single directed relationship between them and treat it as undirected when you are executing graph algorithms or queries. You will learn more about this approach in the following chapters. For now, it is enough that you are aware that it is possible to describe an undirected relationship in the `MERGE` clause.

When creating or importing data to Neo4j, you want to split a Cypher statement into multiple `MERGE` clauses and merge nodes and relationships separately. When merging nodes, the best approach is to only include the node’s unique identifier property in the `MERGE` clause and add additional node properties with the `ON MATCH SET` or `ON CREATE SET` clauses.

Handling relationships is a bit different. If there can be at most a single relationship of one type between two nodes, like the `FRIEND` example, then do not include any relationship properties in the `MERGE` clause. Instead, use the `ON`

CREATE SET ` or `ON MATCH SET clauses to set any relationship properties. However, if your graph model contains multiple relationships of the same type between a pair of nodes, then only use the unique identifier property of the relationship in the MERGE statement and set any additional properties the same as above.

3.2 Importing CSV files with Cypher

You have learned the basic Cypher clauses that will help you get started. Now, you will learn how to import data from external sources. A frequent input data structure for a graph database is either a CSV or a JSON format. The first data structure you will learn how to import is the CSV-like data structure. Interestingly, dealing with CSV files and importing data from a relational database is almost identical. In both scenarios, you are dealing with a table that has, hopefully, named columns. In this section, you will define unique constraints and import a Twitter dataset into Neo4j graph database.

3.2.1 Cleanup the database

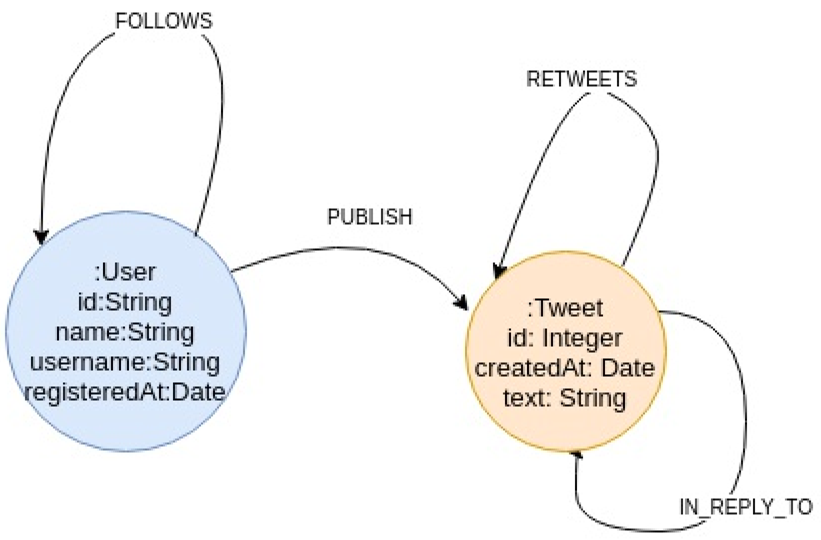
You need to empty the database before continuing as you don’t want random nodes from the previous examples to persist.

Listing 3.35. Cypher query that deletes all the nodes and relationships in the database

```
MATCH (n)
DETACH DELETE n
```

3.2.2 Twitter graph model

Figure 3.8. Initial Twitter graph model you will import



In the previous chapter, you went through a graph model design process. There were no data limitations. You just assumed that you could get any relevant data. Like anything in life, you can't always get what you asked for. But there might be some additional data that wasn't considered before.

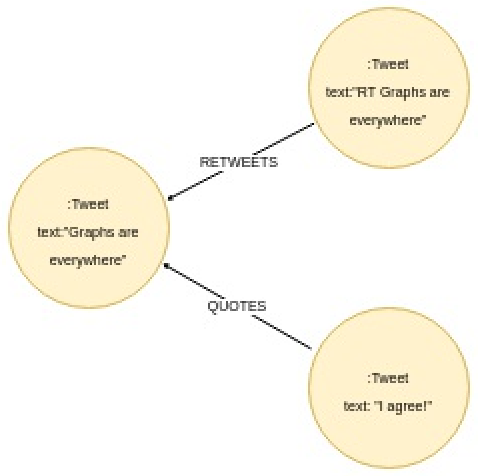
You will start by importing the follower network between users. There was a `since` property of the `FOLLOWS` relationship in the initial graph model. Unfortunately, the Twitter API doesn't provide the date of creating the `FOLLOWS` relationship, so you will have to remove it from the model.

This is not a problem with the initial graph modeling design. You have made some assumptions that later didn't hold up. That is why the graph modeling process is iterative. You start with some assumptions and change the graph model accordingly as you learn more. On the other hand, the assumption that there can only be a single author of a given tweet turns out valid. And you did not take into account that a user can also reply to a given tweet and not just retweet it. The graph model has been updated to support storing the information when a tweet was made in response to another tweet by adding an `IN_REPLY_TO` relationship. I also wanted to introduce the difference between just retweeting a post or adding a comment to the retweet. The Twitter interface allows you to add a comment to the retweet by using the Quote Tweet option.

Figure 3.9. Add a comment to the retweet by using the Quote Tweet option

Because adding a comment has different semantics than just retweeting a post, I wanted to differentiate between the two scenarios by using different relationship types.

Figure 3.10. Differentiate between retweets and quotes by using different relationship types



By clearly differentiating between retweets and quotes, it would be easier for you to find quotes in the graph and analyze their responses. For example, you could use NLP techniques to detect the sentiment of the comments and examine which tweets or users are more likely to receive positive or negative sentiment comments. Unfortunately, while I was thinking of including them in our graph import, I did not fetch any quote tweets during my scraping process, so we will skip importing and analyzing them.

In the initial import, you will also ignore the hashtags, mentions, and links of a tweet.

3.2.3 Unique constraints

The Neo4j graph database model is considered to be *schema-less*, meaning that you can add any types of nodes and relationships without defining the graph schema model. There are, however, some constraints you can add to your graph model to ensure data integrity. In my graph journey, I have only used the *unique node property constraint* so far. There are two benefits of using the unique node property constraint. The first benefit is to ensure that the value of a given node property is unique for all the nodes with a specific label. As a beginner, this feature is handy as it lets you know and stops an import query that would corrupt data integrity. An additional benefit of defining a unique node constraint is that it automatically creates an index on the specified node property. By creating an index on the specified node property, you will optimize the performance of import and analytical Cypher queries. For the initial import, you will define two node unique constraints. One unique constraint will ensure that there can only be a single `User` node with a specific `id` property in the database. The second unique constraint guarantees that the `id` property of nodes with label `Tweet` will be unique for each node.

Listing 3.36. Cypher query that defines two unique node constraints

```
CREATE CONSTRAINT IF NOT EXISTS ON (u:User) ASSERT u.id IS UNIQUE
CREATE CONSTRAINT IF NOT EXISTS ON (p:Tweet) ASSERT p.id IS UNIQU
```

3.2.4 LOAD CSV clause

Cypher query language has a *`LOAD CSV`* clause that enables you to open and retrieve information from CSV files. The `LOAD CSV` clause can fetch local CSV files as well as CSV files from the internet. Having the ability to fetch CSV files from the internet comes in very handy as you don’t have to download the CSV files to your local computer first. I have stored all the relevant CSV files on GitHub (github.com/tomasonjo/graphs-network-science) for easier access. The `LOAD CSV` clause can load CSV files that contain a header or not. If the header is present, each row of the CSV file will be available as a map data structure that can be used later in the query. Conversely, when there is no header present, the rows will be available as lists. The `LOAD CSV` can also be used in combination with a `FIELDTERMINATOR` clause to set a custom delimiter, where for example you are dealing with a tab separated value format.

To retrieve information from a specific CSV file, you can use the following query:

Listing 3.37. Cypher query that fetches and displays information from a CSV file

```
LOAD CSV WITH HEADERS FROM
"https://raw.githubusercontent.com/tomasonjo/graphs-network-scien
WITH row
LIMIT 5
RETURN row
```

The CSV file must be publicly accessible as the `LOAD CSV` clause does not feature any authorization support. The statement in Listing 3.38 also introduces the *`LIMIT`* clause. The `LIMIT` clause is used to limit the number of results you want to retrieve. It can also be used in combination with a `WITH` clause.

One thing to note is that the `LOAD CSV` clause returns all values as strings and makes no attempt to identify data types. You have to convert the values to the correct data type in your import Cypher statements.

3.2.5 Importing the Twitter social network

I have prepared five CSV files that contain the following information:

- User information
- Follower network
- Information about tweets and their authors
- Information about the RETWEETS relationships between posts
- Information about the IN_REPLY_TO relationships between posts

It is a good practice to split the graph import into multiple statements. I could have probably prepared a single CSV file with all the relevant information. Still, it makes more sense to split the import into multiple statements for more readability and faster import performance. If you are dealing with graphs with millions of nodes, then it is advisable to split the import of nodes and relationships. In this case, you are dealing with only thousands of nodes, so you don’t have to worry about query optimization that much. My general rule of thumb is to split the import queries by node labels and relationship types as much as possible.

To begin with, you will import user information into Neo4j. As mentioned, all the data is publicly available on GitHub, so there is no need to download any files. The CSV structure for user information has the following structure:

Table 3.1. User CSV structure

id	name	username	createdAt
333011425	ADEYEMO ADEKUNLE King	ADEYEMOADEKUNL2	2011-07-10T20:36:5
1355257214529892352	Wajdi Alkayal	WajdiAlkayal	2021-01-29T20:51:2
171172327	NLP Excellence	excellenceNLP	2010-07-26T18:48:4

You can use the following Cypher statement to import user information into the Neo4j database.

Listing 3.38. Cypher query that imports user information from a CSV file

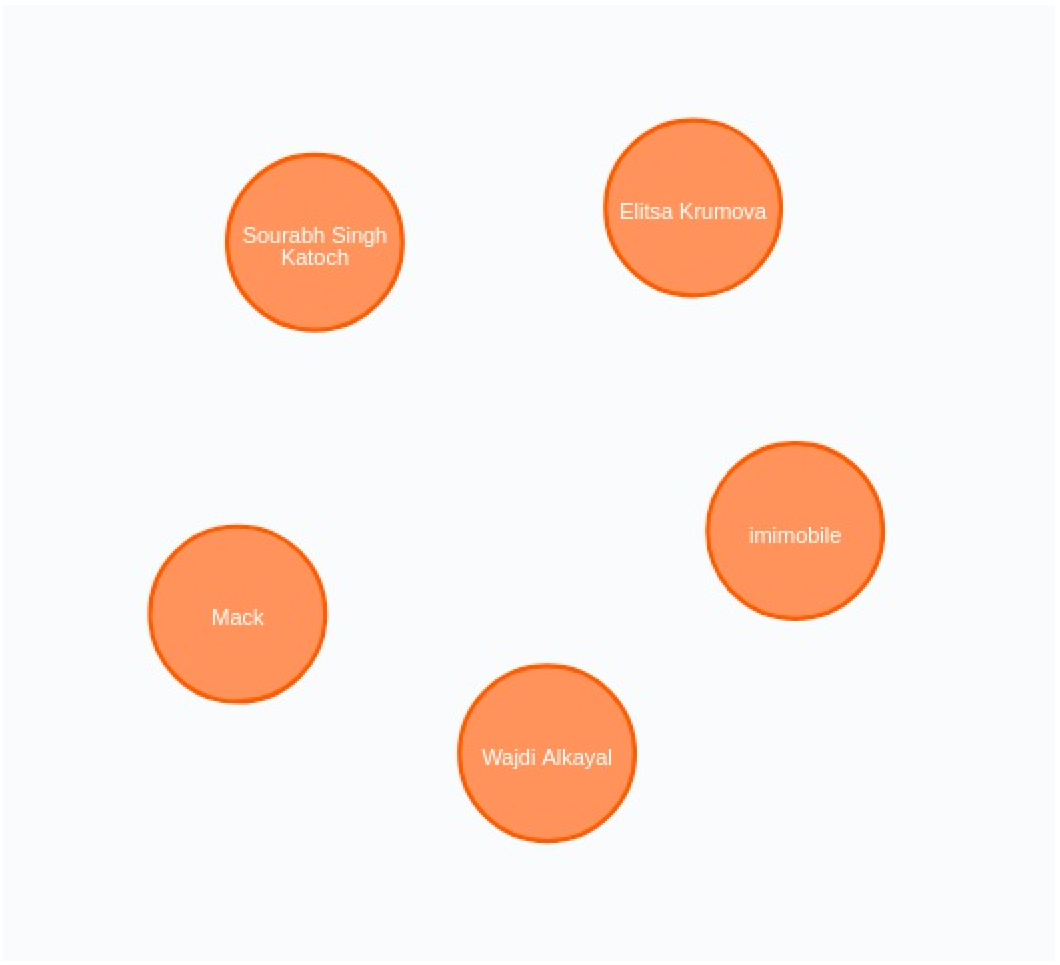
```
LOAD CSV WITH HEADERS FROM
"https://raw.githubusercontent.com/tomasonjo/graphs-network-scien
MERGE (u:User{id:row.id})
ON CREATE SET u.name = row.name,
              u.username = row.username,
              u.registeredAt = datetime(row.createdAt)
```

You could have used the `CREATE` clause to import user information. If you started with an empty database, and if you trust that I have prepared a CSV file without duplicates, the result would be identical.

Dealing with real-world datasets, you often can't afford the luxury of assuming you have clean data present. Hence, it makes sense to write cypher statements that can handle duplicates or other anomalies and are also idempotent. In the Cypher statement in Listing 3.38, the `LOAD CSV` clause first fetches the CSV information from the GitHub repository. The `LOAD CSV` then iterates over every row in the file and executes the Cypher statement that follows. In this example, it executes the `MERGE` clause in combination with `ON CREATE SET` for every row in the CSV file.

Retrieve five random users from the database to inspect the results and validate that the import process of users worked correctly.

Figure 3.11. Visualization of random five users in the database



Currently, you only have nodes without any relationships in the database. You will continue by importing `FOLLOWS` relationships between users. The

CSV file that contains the information about the follower has the following structure:

Table 3.2. Followers CSV structure

source	target
14847675	1355257214529892352
1342812984234680320	1355257214529892352
1398820162732793859	1355257214529892352

The followers CSV file has only two columns. The source column describes the start node id, and the target column describes the end node id of the follower relationships. When dealing with larger CSV files, you can use the *`USING PERIODIC COMMIT`* clause to split the import into several transactions. Splitting the import into several transactions can spare you the headache of running out of memory when doing large imports. By default, `USING PERIODIC COMMIT` clause will split the transaction for every 1000 rows. The followers CSV has almost 25000 rows. Instead of importing the whole CSV file in a single transaction, you should use the `PERIODIC COMMIT` clause to split it into 25 transactions effectively. For some reason, you need to prepend `:auto` when using `PERIODIC COMMIT` in Neo4j Browser. In other cases, for example, when you are using a Neo4j Python driver to import the data, you don't need to prepend the `:auto` operator.

Listing 3.39. Cypher query that imports follower network from a CSV file

```
#A
:auto USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
"https://raw.githubusercontent.com/tomasonjo/graphs-network-scien
#B
MATCH (s:User{id:row.source})
```

```
#C
MATCH (t:User{id:row.target})
#D
MERGE (s)-[:FOLLOWS]->(t)
```

The statement in Listing 3.40 first retrieves the information from a CSV file located on GitHub. The following query steps will be executed for every row in the CSV file. For each row, it matches the source and the target `User` node. Here, you assume all the `User` nodes are already present in the database. Remember, if the `MATCH` clause does not find a pattern, it skips the execution of the rest of the query for a specific row.

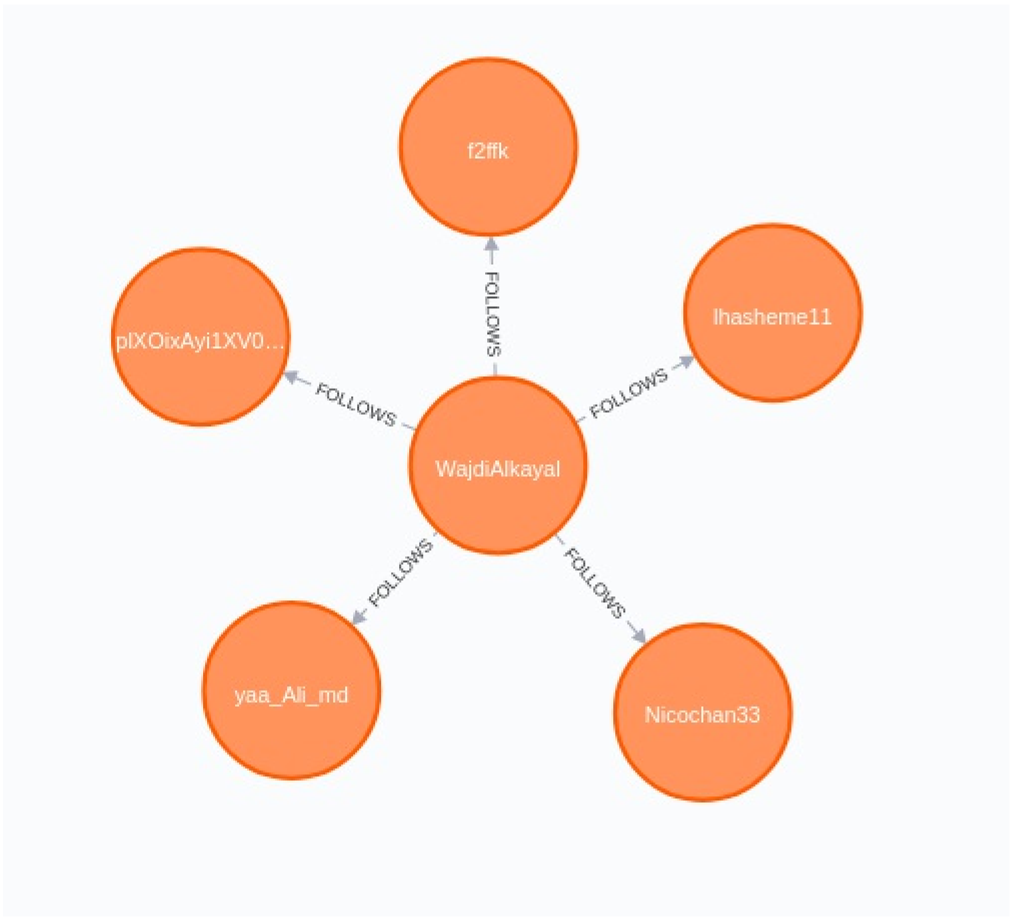
In statement in Listing 3.40, if, for example, a source node was not found by the `MATCH` clause, the Cypher query will skip the creation of the `FOLLOWS` relationship. You could avoid this limitation by using the `MERGE` clause instead of the `MATCH` clause to identify the source and target `User` nodes. There is, however, a drawback that any node created by the `MERGE` clause in this statement would only have the `id` property and no other information as they were missing from the CSV file containing user information.

Once both the source and the target nodes are identified, the query then merges a `FOLLOWS` relationship between them. Using the `MERGE` clause, you ensure that there will be exactly one `FOLLOWS` relationship from the source to the target `User` node in the database. This way, you don’t have to worry about having duplicates in the input CSV file or rerunning the query multiple times. Another critical consideration is that the `FOLLOWS` relationship in the Twitter domain has semantic value. For this reason, you need to add the relationship direction indicator in the `MERGE` clause to ensure that the relationship is imported correctly.

When you are using the `MATCH` clause to identify nodes in Cypher import queries, be aware that no additional nodes will be created during the import, and so, all the relationships between nodes that do not exist in the database will also be skipped during the import process.

Retrieve five `FOLLOWS` relationships from the database to validate the import process.

Figure 3.12. Visualization of random five FOLLOWS relationships in the database



Next, you will import the tweets and their authors. The CSV structure of the tweets is as follows:

Table 3.3. Twitter posts CSV structure

id	text	createdAt	author
12345	Example text	2021-06-01T08:53:22	134281298
1399649667567	Graph data science is cool!	2021-06-01T08:53:18	54353345
	Exploring social	2021-06-	

13996423457567	networks	01T08:45:23	4324323

The id column of the Tweets CSV file describes the Twitter post id that will be used as the unique identifier. The file also includes the text and the date of creation, as well as the id of the author. There are 12000 rows in the CSV file, so you will again use the `PERIODIC COMMIT` clause for batching purposes.

Listing 3.40. Cypher query that imports tweets from a CSV file

```
:auto USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/tom
#A
MATCH (a:User{id:row.author})
#B
MERGE (p:Tweet{id:row.id})
ON CREATE SET p.text = row.text,
              p.createdAt = datetime(row.createdAt)
#C
MERGE (a)-[:PUBLISH]->(p)
```

If you look closely, you can observe that the query structure in Listing 3.41 is similar to the query in Listing 3.40. When you are importing any relationship into the database, you will most likely match or merge both source and target nodes and then connect them. In the first step, the query matches on the `User` node. Next, you use the `MERGE` clause to create the `Tweet` nodes. Although all the `Tweet` nodes need to be created as there are none in the database beforehand, I still like to use the `MERGE` clause to have idempotent queries. It is the best experience for you as a user, and it is an excellent practice to follow. Last but not least, a relationship between the user and the tweet is created.

To validate the import process, retrieve the text property of three random tweet nodes.

In the last two import queries, you will import additional RETWEETS and IN_REPLY_OF relationships. Both the retweets and the in reply to CSV files have the same structure:

Table 3.4. Retweets and in reply to CSV files structure

source	target
14847675	1355257214529892352
1342812984234680320	1355257214529892352
1398820162732793859	1355257214529892352

You will begin by importing the `RETWEETS` relationship.

Listing 3.41. Cypher query that imports retweets relationships from a CSV file

```
LOAD CSV WITH HEADERS FROM
"https://raw.githubusercontent.com/tomasonjo/graphs-network-scien
#A
MATCH (source:Tweet{id:row.source})
#B
MATCH (target:Tweet{id:row.target})
#C
MERGE (source)-[:RETWEETS]->(target)
```

The query structure to add relationships to a database mostly identifies the source and target nodes and then adds a connection between the two. You have to take special care whether you use `MATCH` or `MERGE` to identify nodes. If you use the `MATCH` clause, then no new nodes will be created, and so any relationships that don't have both the source and the target node already in the database will be ignored during the import. On the other hand, if you use the `MERGE` clause, you might end up with new `Tweet` nodes with only the `id` but no `text` property or even the author connected to it. Using the `MERGE` clause to add relationships, you ensure that there will be precisely one relationship of that type between source and target nodes no matter how many times the connection occurs in the underlying data or how often you run the import query. There are options to change the query to import more than a single relationship type between a pair of nodes and still use the `MERGE` clause.

Match five RETWEETS relationships between a pair of tweets from the database. Inspect the text of the original and the retweeted post.

The final import statement imports the `IN_REPLY_TO` relationships. It has almost identical structure as importing the `RETWEETS` relationship, only the type of relationship is changed.

Listing 3.42. Cypher query that imports IN_REPLY_TO relationships from a CSV file

```
LOAD CSV WITH HEADERS FROM
"https://raw.githubusercontent.com/tomasonjo/graphs-network-scien
#A
MATCH (source:Tweet{id:row.source})
#B
MATCH (target:Tweet{id:row.target})
#C
MERGE (source)-[:IN_REPLY_TO]->(target)
```

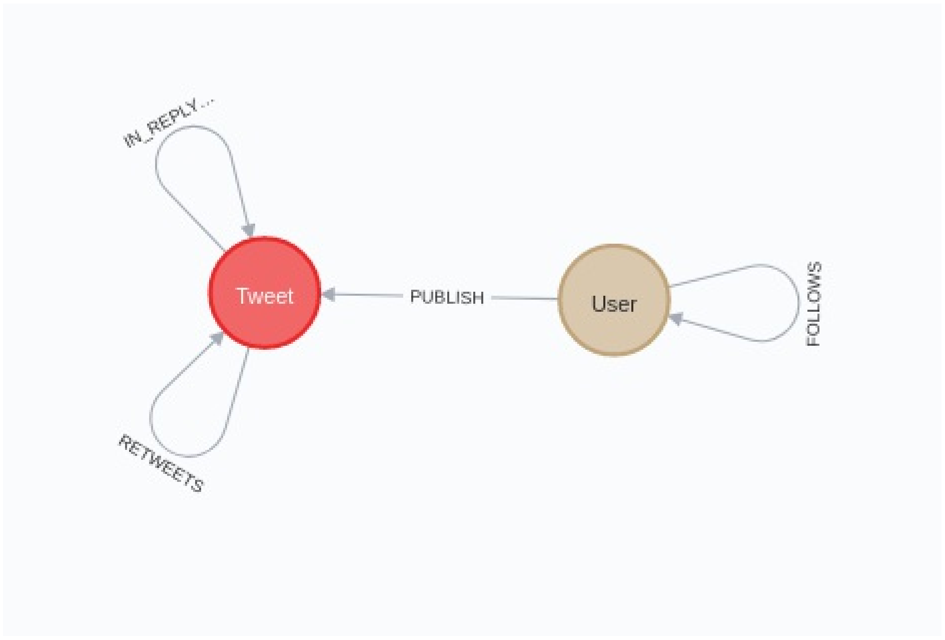
Congratulations, you have imported the initial Twitter graph into Neo4j. Neo4j has a special procedure that enables you to inspect and visualize the graph schema of the stored graph in the database.

Listing 3.43. Schema introspection procedure

```
CALL db.schema.visualization()
```

If you run the schema introspection procedure in Neo4j Browser, you should get the following schema visualization.

Figure 3.13. Visualization of generated graph schema based on the stored graph.



If you get the exact same graph schema visualization, you are ready to learn more about the analytical Cypher queries and graph algorithms in the next chapter. If the generated schema is not identical, rerun all the import queries.

3.3 Summary

- Cypher syntax to describe graph patterns
- Basic Cypher clauses and their caveats
- Unique node property constraints
- How to import a graph from a CSV file
- How to split CSV import into multiple transactions

4 Exploratory graph analysis

This chapter covers

- Exploring a graph with Cypher query language
- Aggregating data with the Cypher query language
- Using existential subqueries to filter by graph patterns
- Accessing relationship store to efficiently count relationships
- Handling query cardinality when using multiple clauses in a single Cypher statement

This chapter will teach you how to perform an exploratory data analysis of the imported Twitter social network using Cypher query language. Imagine you are working as a social media consultant and want to find as many insights as possible. As is typical with any analysis, you begin with an exploratory data analysis to get an overview of the data you are working with.

I will present how I collected the data to give you a feeling of the data you will be working on in this chapter. The imported Twitter social network was scraped using the official Twitter API. I have fetched tweets that are part of the `NLP` or `Knowledge graph` topics. At this time, I had information about tweets and users who were either mentioned or published tweets. Next, I fetched additional meta-data about users in the graph, such as their registration date and follower relationships. All the users in the imported graph have either published a tweet or were mentioned by one. I did not include all the user followers because that would explode the graph, and the network would end up consisting of a couple of million users.

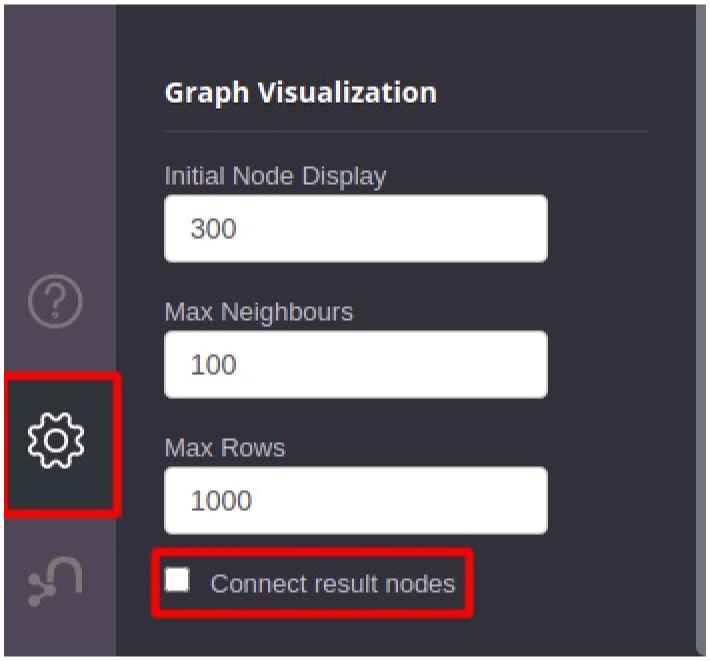
One part of the exploratory graph analysis will consist of counting the number of nodes and relationships in the network. With the Twitter network, it is also essential to know the timeline of the created tweets. You will learn how to aggregate time-based information with Cypher query language. As the last part of the exploratory analysis, you will examine some outliers in the dataset, like users who posted the most tweets or were mentioned the most.

That was the high-level theory overview of what you will be learning in this chapter, so now let’s get to practical examples. To follow along with the examples in this chapter, you must have a Neo4j database instance with Twitter network data imported as described in Chapter 3.

4.1 Exploring the Twitter network with Cypher query language

You will start with exploratory graph analysis using the Cypher query language. The goal of the exploratory analysis is to get to know the dataset and teach you Cypher query syntax that will allow you to aggregate and filter data. I would recommend you use the Neo4j Browser environment, which can be used to develop Cypher queries and return the query results in the form of a table as well as a network visualization. Neo4j Browser has a beginner-friendly feature that visualizes all the relationships between resulting nodes, even when the relationships are not part of the query results.

Figure 4.1. Untick the "Connect result nodes" in Neo4j Browser.


Graph Visualization

Initial Node Display: 300

Max Neighbours: 100

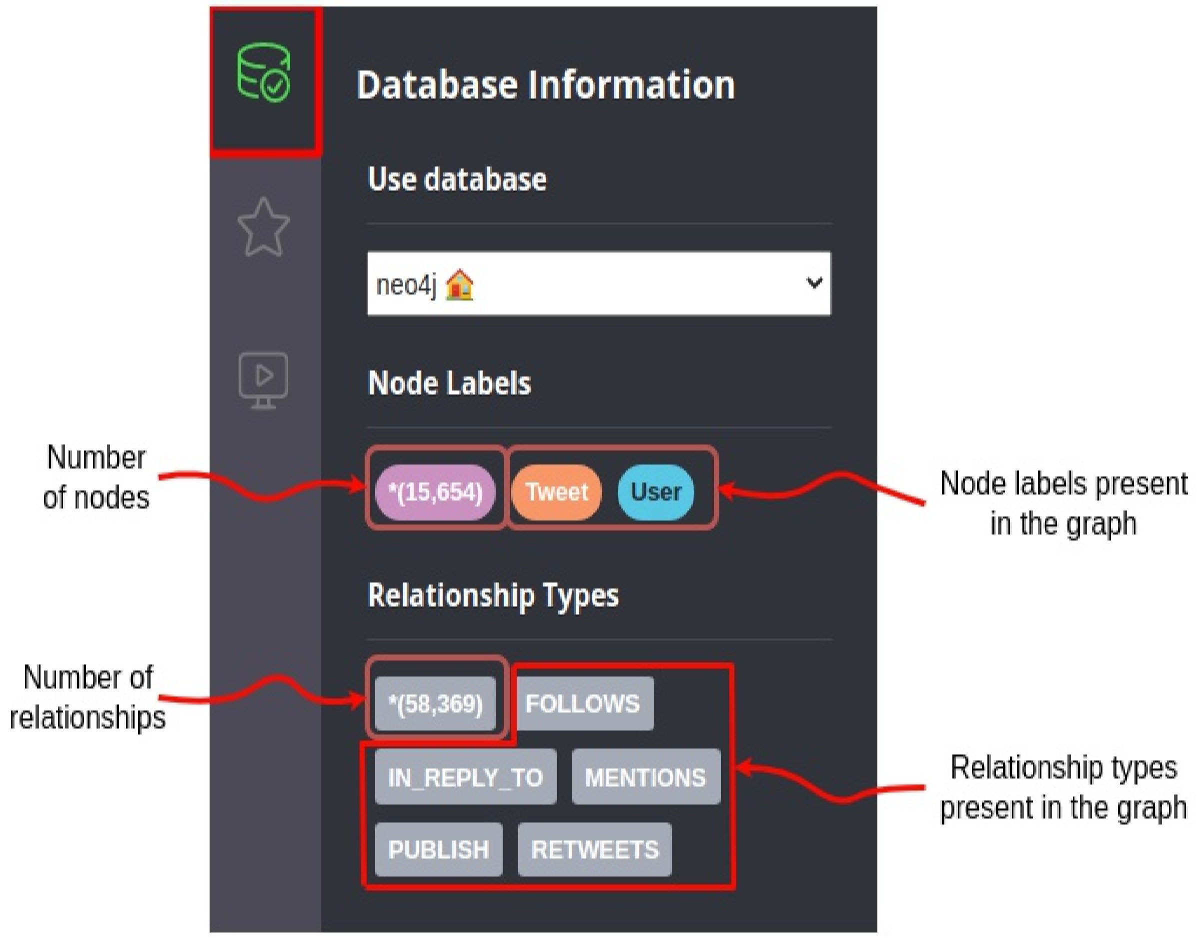
Max Rows: 1000

[ ] Connect result nodes

To avoid confusion, untick the "Connect result nodes" feature as shown in the Figure 4.1.

If you open the database tab in the top-right corner of the Neo4j Browser, it will show a simple report indicating the number and the type of nodes and relationships in the database.

Figure 4.2. Neo4j Browser database report



There is a total of 15654 nodes in the database. As you already know, nodes are labeled either as `Tweet` or `User`. In addition, there are also 58369 connections across four relationships types. Both the node labels and the relationship types in the left-side toolbar are clickable. For example, if you click on the `FOLLOWS` relationship type, the tool will generate a Cypher statement that returns a sample of 25 `FOLLOWS` relationships.

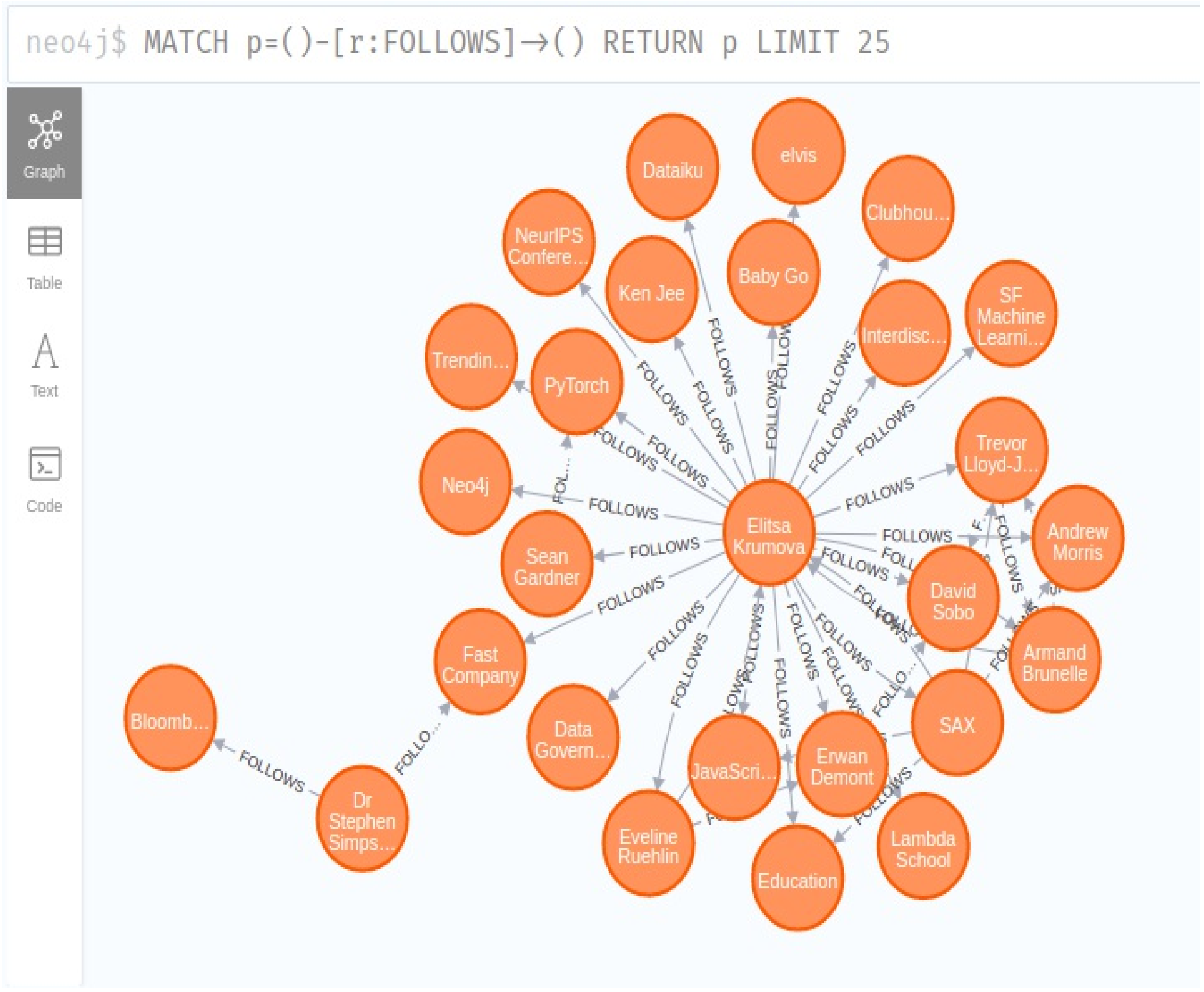
Listing 4.1. Generated cypher query that visualizes a sample of 25 `FOLLOWS` relationships.

```
MATCH p=()-[r:FOLLOWS]->()
RETURN p LIMIT 25;
```

The generated statement in Listing 4.1 returns the results as the *path* data object. A sequence of connected nodes and relationships can be represented as a path data type. Cypher query syntax allows paths to be referenced by a variable name, similar to node and relationship variables.

You should get a similar visualization as shown in the following image by executing the generated Cypher statement.

Figure 4.3. A subgraph of the followers network



```
neo4j$ MATCH p=()-[r:FOLLOWS]→() RETURN p LIMIT 25
```

Exercise 4.1

As an exercise, generate Cypher statements to visualize sample `RETWEETS` and `PUBLISH` relationships. You can click on the relationship types in the left-side toolbar, or you can use the statement in Listing 4.1 as the base and change the relationship type accordingly.

4.1.1 Aggregating data with Cypher query language

Aggregating and counting data points is the basis of all data analysis. In the context of graph analysis, you will first learn how to count various graph patterns like the number of nodes and relationships. If you already have some experience with relational databases and SQL query language, you will see that Cypher follows a similar syntax for aggregating data. The first aggregation function you will learn is the `count()` function. It is used to count the number of rows or values produced by the `MATCH` clause.

To count the number of the nodes in the database, you can execute the following Cypher statement:

Listing 4.2. Count the number of nodes.

```
#1
MATCH (n)
#2
RETURN count(n) AS numberOfNodes;
```

There is a total of 15654 nodes in the graph. The `count()` function appears in two variants:

- `count(*)` - Returns the number of rows produced by Cypher statement
- `count(variable or expression)` - Return the number of non-null values produces by an expression

To test both variants of the `count()` functions, you will count the number of `User` nodes and the non-null values of their `registeredAt` properties.

Listing 4.3. Count the number of `User` nodes and their non-null values of `registeredAt` properties.

```
MATCH (u:User)
```

```
RETURN count(*) AS numberOfRows,
       count(u.registeredAt) AS numberOfUsersWithRegisteredAtDate
```

Table 4.1. Results of the Cypher statement in Listing 4.3

numberOfRows	numberOfUsersWithRegisteredAtDate
3594	3518

There are 3594 `User` nodes in the graph, but only 3518 of them have the non-null `registeredAt` property. The graph is missing the registration date information for 76 users. When preparing a dataset summary, you usually present the number of missing values as a ratio and not as an absolute number. In Neo4j, you have to be careful because when you divide an integer by an integer, the result will also be an integer. To avoid this issue, you can cast either of the variables to float data type.

Execute the following Cypher statement to evaluate the ratio of non-null values for the `registeredAt` node property of `User` nodes.

Listing 4.4. Calculate the ratio of non-null values of `registeredAt` node property.

```
#1
MATCH (u:User)
#2
WITH count(*) AS numberOfRows,
     count(u.registeredAt) AS usersWithRegisteredAtDate
#3
RETURN toFloat(usersWithRegisteredAtDate) / numberOfRows * 100 AS
```

If you forgot to cast either the `usersWithRegisteredAtDate` or `numberOfRows` in Listing 4.4 to float, the result would be 0.

Note

In Neo4j, when you divide an integer value by another integer value, the result will also be of integer data type. If you want the result to be of float type, such as with the ratio example in Listing 4.4, you need to cast either of the variables to float using the `toFloat()` function.

Exercise 4.2

Calculate the ratio of missing values for the `createdAt` node property of the `Tweet` nodes. The result of the statement should be a percentage of non-null values divided by the count of tweets.

The correct answer to Exercise 4.2 is that there are no missing values for the `createdAt` property of `Tweet` nodes.

As you might be used to from your other data projects, you want to aggregate or count values grouped by specific values more often than not. For those familiar with SQL aggregations, you can define the *grouping keys* in the `GROUP BY` statement. Grouping keys are non-aggregate expressions that are used to group the values going into the aggregate functions. Aggregation in Cypher is different from aggregation in SQL. In Cypher, you don't need to specify a grouping key explicitly. As soon as any aggregation function is used in the Cypher statement, all non-aggregated columns in the `WITH` or `RETURN` clause become grouping keys. With Cypher query language, the grouping keys are implicitly defined as you don't have to explicitly add a `GROUP BY` statement after the aggregation functions.

Suppose you want to count the number of nodes grouped by their node label. The function to extract the node labels is `labels()`. You only need to provide the `labels()` function as the grouping key alongside an aggregation function that will count the number of nodes.

Listing 4.5. Count the number of nodes by labels.

```
MATCH (n)
RETURN labels(n) AS labels,
       count(n) AS count;
```

Table 4.2. Count of nodes grouped by their label.

labels	count
["User"]	3594
["Tweet"]	12060

At the moment, there are only two types of nodes in the graph. You can observe that the node labels are returned as a list. The list data type indicates that you can have multiple labels on a single node. Assigning a secondary node label is helpful in a network analysis when we want to speed up subsequent queries by tagging relevant subsections of nodes. As mentioned, you might notice the lack of a `GROUP BY` statement. In Cypher, you don't need to explicitly specify a grouping key. *As soon as an aggregation function is used, all non-aggregated result columns become grouping keys.*

Note

With Cypher query language, the grouping keys are defined implicitly, meaning that all non-aggregated columns in a `WITH` or `RETURN` clause automatically become grouping keys.

Exercise 4.3

Count the number of relationships by their type. You can use the `type()` function to extract relationship type.

To count the number of relationships grouped by their type, you start by describing a relationship pattern with Cypher syntax. Note that you cannot describe a relationship without its adjacent nodes. Because you are interested in counting the relationships types, you must not specify any node labels or relationship types in the Cypher pattern. In the last part of the statement, you use the `type()` function to extract the relationship type and use it as a grouping key in combination with the `count()` aggregation function.

The solution of Exercise 4.3 produces the following output.

Table 4.3. Count of relationships grouped by their type.

labels	count
"PUBLISH"	12060
"FOLLOWS"	24888
"RETWEETS"	8619
"MENTIONS"	12379
"IN_REPLY_TO"	423

The number of `PUBLISH` relationships is identical to the number of `Tweet` nodes. With the Twitter social network, a single `TWEET` has precisely a single author, indicated by the `PUBLISH` relationship.

Interestingly, 8619 out of 12060 tweets are retweets, and 423 tweets are replies. It looks like only around 30% of the tweets are original content. This is not so unusual as, for example, the research by Joshua Hawthorne et al. [Hawthorne, Joshua et al., 2013] shows that tweets by more prominent users can have even more than three retweets per tweet. Not surprisingly, when researchers examined the tweets by former US presidents [Minot JR et al., 2021], the number of retweets is at least an order of magnitude higher than usual.

What’s a bit surprising is that there are more mentions than tweets. I did not manually parse the mentioned information as that was automatically provided by the official Twitter API. What I’ve noticed is that when a user retweets another user, he is automatically mentioned in the retweet.

Exercise 4.4

Inspect the text of a retweet and compare it to the original tweet’s text. Use the `LIMIT` clause to limit the number of results to 1.

The solution of Exercise 4.4 produces the following output.

Table 4.4. A single comparison of tweet and retweet’s text.

retweetText	originalText
"RT @Eli_Krumova: 5 Best Practices: Writing Clean & Professional #SQL #Code t.co/Y4DepLfOOn v/ @SourabhSKatoch #DataScience #AI #ML…"	"5 Best Practices: Writing Clean & Professional #SQL #Code t.co/Y4DepLfOOn v/ @SourabhSKatoch #DataScience #AI #ML #MachineLearning #IoT #IIoT #IoTPL #Python #RStats #Cloud #CyberSecurity #Serverless #RPA #NLP #programming #coding #100DaysOfCode #DEVCommunity #CodeNewbie t.co/ma03V8btZB t.co/TOnwwHgaHQ

One immediately obvious thing is that the retweet’s text is trimmed to a fixed length and does not always contain the original tweet’s complete text. Another more subtle difference is that the retweet’s text is prepended with RT followed by the original author’s handle. It seems that Twitter automatically prepends the original user’s handle in the retweet and treats it as a mention. I had no idea this was the case. *It is a good practice to always begin with exploratory graph analysis before diving into graph algorithms to spot such* *abnormalities.*

Exercise 4.5

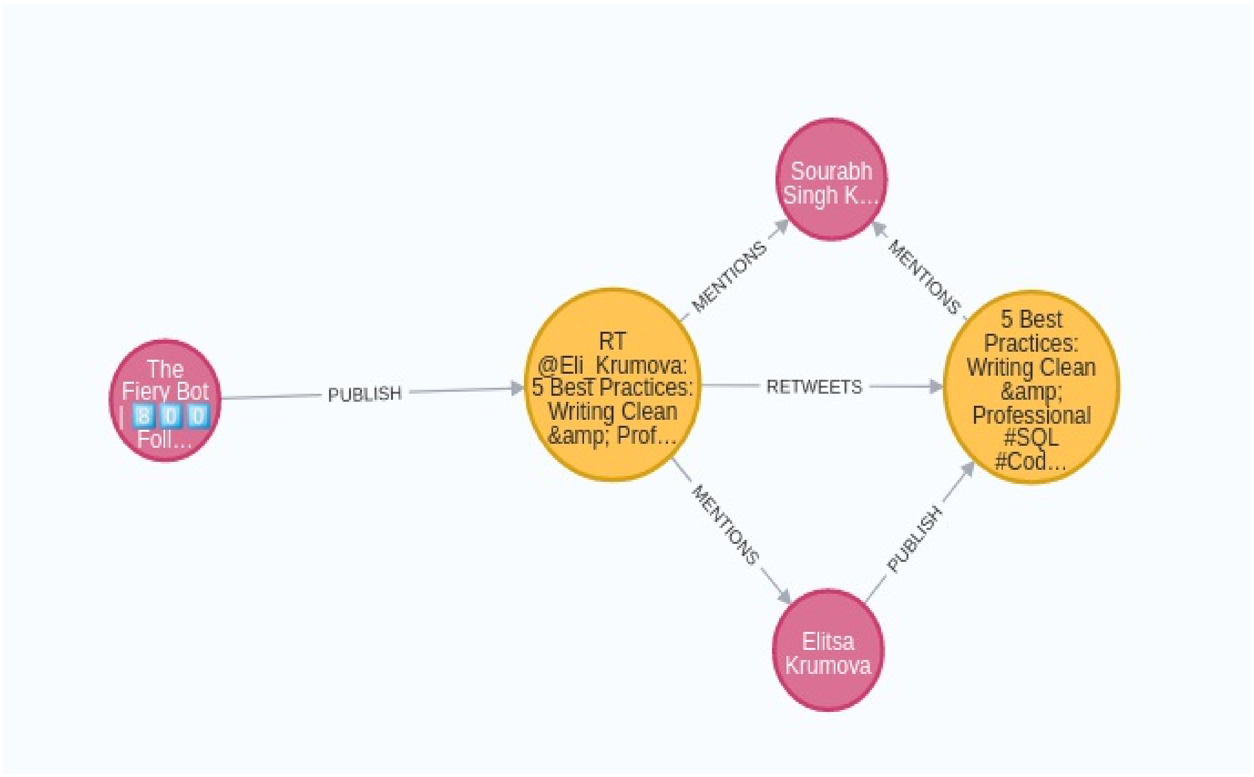
For those of you who are more visually oriented, try to visualize a single graph pattern where a user retweeted a post from another user. Include the `MENTION` relationships of both the original and the retweeted post. Follow these hints to help you construct the desired Cypher statement:

- Match a graph pattern that describes a retweet, original tweet, and their authors
- Use the `WITH` clause in combination with the `LIMIT` clause to limit results to a single described pattern
- Separately match the `MENTION` relationships of the original and the retweeted post
- Visualizing networks in Neo4j Browser is easiest by returning one or multiple path objects

This is a bit more advanced exercise, so take it step by step to construct the final Cypher statement. You can examine the results after each step to make sure you have correctly described the desired graph pattern.

The solution of Exercise 4.5 produces the following network visualization in Neo4j Browser.

Figure 4.4. By default, a retweet also mentions the original tweet author.



The Fiery Bot | 8 0 0 Foll...
PUBLISH
RT @Eli_Krumova: 5 Best Practices: Writing Clean &amp; Prof...
MENTIONS
Sourabh Singh K...
MENTIONS
RETWEETS
5 Best Practices: Writing Clean &amp; Professional #SQL #Cod...
MENTIONS
PUBLISH
Elitsa Krumova

4.1.2 Time aggregations

As with any dataset, it is essential to learn the timeline of the data points. Both the `User` and the `Tweet` nodes contain the datetime properties. First, you will evaluate the time windows of the tweets. You can use the `min()` and `max()` functions on the datetime property to get the earliest and the last date values.

Listing 4.6. Retrieve the earliest and last created date values of tweets

```
MATCH (n:Tweet)
RETURN min(n.createdAt) AS earliestDate, max(n.createdAt) as last
```

The first tweet in the dataset was on the 12th of August 2016, and the last one was on the 1st of June 2021. There is a five-year span between the first and the last tweet. While this information is nice, it is not very descriptive. To get a better feeling for the time window, you will calculate the distribution of tweets by year.

In Cypher, a datetime property behaves like an object. You can access datetime attributes such as the year and the month by using the following Cypher syntax:

Listing 4.7. Extract datetime attributes.

```
MATCH (t:Tweet)
WITH t LIMIT 1
RETURN t.createdAt.year AS year,
       t.createdAt.month AS month,
       t.createdAt.day AS day,
       t.createdAt.epochSeconds AS epochSeconds;
```

Table 4.5. Datetime attributes of a sample tweet

year	month	day	epochSeconds
2021	6	1	1622537602

You can then use the datetime attributes in combination with aggregation functions.

Exercise 4.6

Calculate the distribution of tweets grouped by created year. Remember, Cypher uses implicit grouping key aggregations, so all you need to add to the `RETURN` statement is the `year` column and the `count` operator.

The solution of Exercise 4.6 produces the following output.

Table 4.6. Distribution of tweets by their creation date

year	count
2021	12029

2020	19
2019	6
2018	3
2016	3

Even though the time window between the first and the last tweet is five years, almost all the tweets are were created in 2021. Let’s drill it down even more.

Exercise 4.7

Use the `MATCH` clause in combination with the `WHERE` clause to select all the tweets that were created in the year 2021. You can filter datetime attributes like you would other node properties. In the next step, calculate the distribution of tweets by their creation month. Use both the creation year and month as grouping keys.

By now, you probably already got into the habit of adding grouping keys as non-aggregate values in the `RETURN` or `WITH` clause. Other than that, you only need to be careful to use the `WHERE` clause to match only tweets that were created in 2021.

The solution of Exercise 4.7 produces the following output.

Table 4.7. Distribution of tweets by their creation date

year	month	count

2021	6	2695
2021	5	8507
2021	4	376
2021	3	432
2021	2	8
2021	1	11

Around 93% (11202 / 12060) of the tweets were created in May and June of 2021. This might give you a hint about when I was scraping the data.

Exercise 4.8

Before you move to the rest of the chapter, I want to present you with a challenge. Can you prepare a Cypher statement that will return the top four days with the highest count of created tweets? Although you haven’t seen this exact example, you already have some experience with all the clauses required to construct this Cypher statement. Here are some hints that should help you:

- Start by matching all tweets
- Use the creation year, month, and day as the grouping keys along with the `count()` aggregation
- Use the `ORDER BY` clause to order the results based on the count descending
- Use the `LIMIT` clause to return only the top three days

Exercise 4.8 is designed to test you on the implicit grouping aggregations in Cypher and to use some of the previously learned clauses together. Please take a couple of minutes and try to solve it on your own. I would recommend you return the results after each step of the query to evaluate if you are on the right track.

The solution of Exercise 4.8 produces the following output.

Table 4.8. Distribution of tweets by their creation date

year	month	day	count
2021	5	31	6185
2021	6	1	2695
2021	5	30	1847
2021	5	28	62

Interestingly, how you started with a five-year time window and were able to narrow it down to only three days by gradually digging deeper. The vast majority of the tweets in the dataset were created between the 30th of May and the 1st of June. This information will help you evaluate the number of tweets and mentions given this timeframe.

4.1.3 Filtering graph patterns

Now you will investigate the network of mentions more thoroughly. You already know that there are 12379 `MENTIONS` relationships, but now you want to determine how many distinct users have been mentioned. How would you construct the Cypher statement to retrieve the number of distinct users that have been mentioned? As a beginner, my first thought would be to use the following statement.

Listing 4.8. Count the number of occurrences where a user is being mentioned.

```
MATCH (u:User)<-[:MENTIONS]-(:Tweet)
RETURN count(u) AS countOfMentionedUsers;
```

At first glance, the statement in Listing 4.14 looks valid. You matched the users who were mentioned and then returned the count of users. But the `count()` function doesn’t count the number of distinct users, it counts the number of occurrences where the user variable is not null. You have actually counted the number of graph patterns where a `User` node has an incoming `MENTIONS` relationship originating from a `Tweet` node. One way to count the number of distinct users who were mentioned is by using the *distinct* prefix. The `distinct` prefix is used to count the number of unique values of a reference variable or expression.

Listing 4.9. Count the number of distinct users who were mentioned.

```
MATCH (u:User)<-[:MENTIONS]-(:Tweet)
RETURN count(u) AS numberOfOccurences,
       count(distinct u) AS numberOfDistinctUsers;
```

Table 4.9. Number of occurences and the count of distinct users who were mentioned in a tweet.

numberOfOccurences	numberOfDistinctUsers
12379	1632

There are a total of 1632 distinct users who were mentioned at least once. When doing any query aggregations, you also have to keep in mind the query *cardinality* and what you are actually counting. Cardinality is the number of rows or records of the input stream to the operation. Cypher operations execute per record or row of the input stream to the operation. In Listing 4.15, the `MATCH` clause produces 12379 rows. These rows are then used as an input to the `count` operator. Using the `count(u)` operator, you are counting the number of non-null values of `u` reference variable. Since the `MATCH` clause will produce no null values for the `u` variable, the result of the `count(u)` operation is 12379.

Note

The cardinality of the query will also affect its performance. You want to keep the cardinality as low as possible to achieve the best execution speed. Prefix your statement with the `PROFILE` clause to compare the performance of queries. If you want to learn more about the query planner and how to optimize query performance, read the Cypher planner appendix.

You don't have to expand all the `MENTIONS` relationships to get a list of users who have were mentioned in at least a single tweet. Using the *existential subqueries* in `WHERE` clause, you can filter on graph patterns. An existential subquery can be used to determine if a specified pattern exists at least once in the graph. You can think of it as an expansion or an upgrade of the `WHERE` clause in combination with the graph patterns, where you can introduce new reference variables or even use other clauses like `MATCH` in the subquery. The subquery begins and ends with curly brackets `{}`. You can use any of the variables from the outer query and use them to describe a graph pattern. However, any new variables you introduce in the subquery are not carried over to the main query.

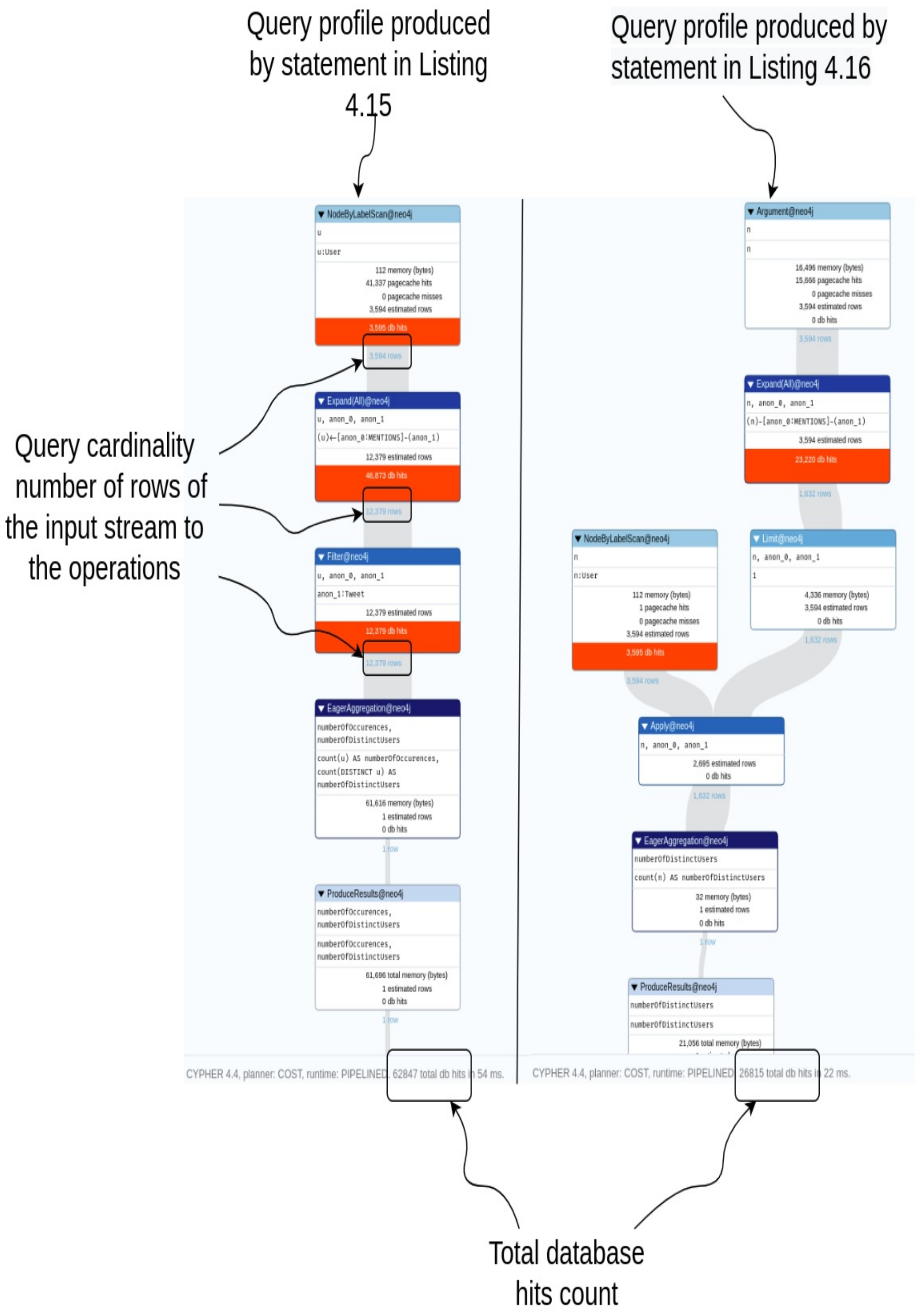
Listing 4.10. Count the number of distinct users who were mentioned in tweets

```
MATCH (n:User)
WHERE EXISTS { (n)<-[:MENTIONS]->() }
RETURN count(n) AS numberOfDistinctUsers;
```

The Cypher statement in Listing 4.16 produces the identical count of distinct users to the query result in Listing 4.15 and is also more performant. The syntax used in Listing 4.16 is useful for finding nodes in the network that are part of at least a single described graph pattern. In this example, you don't care if a user was mentioned once or hundred times. You just want to match the distinct users who were mentioned at least once.

To demonstrate which of the two Cypher statements is more optimized, I will show the query execution plan that is produced by prefixing the statements with the `PROFILE` clause.

Figure 4.5. Comparison of the query execution plans for Cypher statements in Listing 4.15 and 4.16.


Query profile produced by statement in Listing 4.15
Query profile produced by statement in Listing 4.16
Query cardinality number of rows of the input stream to the operations
Total database hits count

```
▼ NodeByLabelScan@neo4j
u
u:User
112 memory (bytes)
41,337 pagecache hits
0 pagecache misses
3,594 estimated rows
3,595 db hits
3,594 rows

▼ Expand(All)@neo4j
u, anon_0, anon_1
(u)←[anon_0:MENTIONS]-(anon_1)
12,379 estimated rows
46,873 db hits
12,379 rows

▼ Filter@neo4j
u, anon_0, anon_1
anon_1:Tweet
12,379 estimated rows
12,379 db hits
12,379 rows

▼ EagerAggregation@neo4j
numberOfOccurences,
numberOfDistinctUsers
count(u) AS numberOfOccurences,
count(DISTINCT u) AS
numberOfDistinctUsers
61,616 memory (bytes)
1 estimated rows
0 db hits
1 row

▼ ProduceResults@neo4j
numberOfOccurences,
numberOfDistinctUsers
numberOfOccurences,
numberOfDistinctUsers
61,696 total memory (bytes)
1 estimated rows
0 db hits
1 row

CYPHER 4.4, planner: COST, runtime: PIPELINED. 62847 total db hits in 54 ms.
```

```
▼ Argument@neo4j
n
n
16,496 memory (bytes)
15,666 pagecache hits
0 pagecache misses
3,594 estimated rows
0 db hits
3,594 rows

▼ Expand(All)@neo4j
n, anon_0, anon_1
(n)-[anon_0:MENTIONS]-(anon_1)
3,594 estimated rows
23,220 db hits
1,632 rows

▼ NodeByLabelScan@neo4j
n
n:User
112 memory (bytes)
1 pagecache hits
0 pagecache misses
3,594 estimated rows
3,595 db hits
3,594 rows

▼ Limit@neo4j
n, anon_0, anon_1
1
4,336 memory (bytes)
3,594 estimated rows
0 db hits
1,632 rows

▼ Apply@neo4j
n, anon_0, anon_1
2,695 estimated rows
0 db hits
1,632 rows

▼ EagerAggregation@neo4j
numberOfDistinctUsers
count(n) AS numberOfDistinctUsers
32 memory (bytes)
1 estimated rows
0 db hits
1 row

▼ ProduceResults@neo4j
numberOfDistinctUsers
numberOfDistinctUsers
21,056 total memory (bytes)

CYPHER 4.4, planner: COST, runtime: PIPELINED. 26815 total db hits in 22 ms.
```

Each Cypher statement gets translated into a set of database operations. While the understanding of database operations is beyond the scope of this book, you can evaluate the performance of the query execution by examining the *total database hits* count. The Cypher statement in Listing 4.15 produces 62847 database hits, and the statement in Listing 4.16 has 26815 database hits. With this information, you can deduce that the Cypher statement in Listing 4.16 is more optimized.

Exercise 4.9

Count the number of distinct users who have published at least a single tweet.

Using the existential subqueries in combination with graph patterns is also very helpful when negating a graph pattern. You could match the whole pattern in the previous example and use the `distinct` to get the correct count. However, when you want to negate a graph pattern, you cannot use it in the `MATCH` clause. Hence, those are prime examples of using the existential subqueries in the `WHERE` clause to negate a graph pattern.

In this example, you will count the number of distinct users who were mentioned, but haven't themselves published a single tweet. You must negate the outgoing `PUBLISH` relationships to filter out users without any tweets.

Listing 4.11. Count the number of users who were mentioned but haven't published a single tweet.

```
MATCH (u:User)
WHERE EXISTS { (u)<-[:MENTIONS]->() } AND
  NOT EXISTS { (u)-[:PUBLISH]->() }
RETURN count(*) AS countOfDistinctUsers
```

Around half of the distinct users (809 / 1632) mentioned in a tweet haven't published any tweets themselves in our dataset. As shown in Listing 4.17, you can easily combine multiple graph pattern predicates to filter out nodes that fit the described graph patterns.

As mentioned, you can also introduce new reference variables in the existential subquery. For example, if you wanted to count the number of users mentioned in a tweet and discount the mentions that are part of the retweet pattern, you would need to introduce a new reference variable.

You will use the existential subquery to count the number of users mentioned in a tweet and ignore the retweet mention pattern.

Listing 4.12. Count the number of users who were mentioned in a tweet and discount the retweet mention pattern with an existential subquery.

```
# A
MATCH (u:User)<-[:MENTIONS]-(tweet:Tweet)
# B
WHERE NOT EXISTS {
  (original)<-[:PUBLISH]-(u)<-[:MENTIONS]-(tweet)-[:RETWEETS]->(o
}
# C
RETURN count(distinct u) AS countOfUsers

# A Use the MATCH clause to identify the pattern in which a User
# B Use the existential query to negate graph patterns where the
# C Use the distinct operator to return the distinct number of us
```

You needed to use the existential subquery in the B part of the Listing 4.18 to be able to introduce the reference variable `original`. The reference to the `original` is needed as you only want to discount the specific `MENTION` relationships that are part of the retweet pattern.

The results of the statement in Listing 4.18 is 1206. Therefore, around 26% (426 / 1632) of the users who were mentioned have an incoming `MENTION` relationship only because their posts were retweeted. Interestingly enough, also around 33% (1206 / 3594) of all users were mentioned in a tweet if you discard retweets. And if you completed the Exercise 4.9, you know that around 75% (2764 / 3594) of all users have published at least a single tweet.

Exercise 4.10

Find the top five users who had the most distinct tweets retweeted. To make it easier for you, I have prepared a template Cypher statement that you need to fill in.

Listing 4.13. Template query for Exercise 4.10.

```
MATCH (n:User)-[:PUBLISH]->(t:Tweet)
# A
_Fill in the WHERE_
WITH n, count(*) AS numberOfRetweets
# B
_Fill in the ORDER BY_
RETURN n.username AS user, numberOfRetweets
# C
_Fill in the LIMIT_

# A Use a combination of WHERE clause with a graph pattern to fil
# B Use the ORDER BY clause to order by numberOfRetweets descendi
# C Use the LIMIT to return only top five users
```

By solving the Exercise 4.10, you should get the following results.

Table 4.10. Top five users with the highest count of tweets that were retweeted.

user	numberOfRetweets
"IainLJBrown"	754
"SuzanneC0leman"	314
"Eli_Krumova"	31
"Paula_Piccard"	31
"Analytics_699"	26

It seems that "IainLJBrown" has by far the most tweets that were retweeted. In the second place, with 354 tweets that were retweeted, is "SuzanneC0leman". You could probably think of them as influencers as they publish a lot but also have their followers retweet their posts a lot. After that, there is an order of magnitude drop to only 31 retweeted posts from "Eli_Krumova" and "Paula_Piccard".

4.1.4 Counting relationships in Neo4j

The last thing I will show you in this chapter is how to count relationships with Cypher in Neo4j efficiently. Although you already know how to count graph patterns and filter them, there is a simple yet very performant trick to keep in mind when counting relationships.

For example, you could use the following Cypher statement if you wanted to get the top five most mentioned users.

Listing 4.14. Retrieve the top five most mentioned users.

```
MATCH (u:User)<-[:MENTIONS]-(:Tweet)
WITH u, count(*) AS mentions
ORDER BY mentions DESC LIMIT 5
RETURN u.username AS user, mentions
```

There is nothing wrong with the statement in Listing 4.20. However, Neo4j is a native graph database and it stores the count of relationships by type and direction for each node. One reason is that counting relationships if a frequent operation and the second reason is that it help with query planner optimization, which we won’t go into in this book. The *relationships store count* is only relevant when you are counting the number of relationships and applying no other filters. As soon as you apply any additional filters or predicates, the query will expand relationships and apply described filters. However, when you only want to count the number of relationships a node has you can use the `size()` operator and describe the desired graph pattern.

Listing 4.15. Optimized way of retrieving the top five most mentioned users.

```
MATCH (u:User)
WITH u, size((u)<-[:MENTIONS]-()) AS mentions
ORDER BY mentions DESC LIMIT 5
RETURN u.username AS user, mentions
```

Table 4.11. Top five users who were mentioned the most

user	mentions
"IainLJBrown"	3646
"SuzanneC0leman"	673
"Analytics_699"	476
"Paula_Piccard"	460
"Eli_Krumova"	283

By far, the most mentioned user is "IainLJBrown". If you are like me, you are probably wondering what’s the distribution of those mentioned. Is he frequently retweeted, are posts that are mentioning him frequently retweeted, or do people just like to mention him? From the results of Exercise 4.10, you already know that he has 754 posts that were retweeted.

When performing multiple aggregation in sequence, you have to be mindful of the intermediate cardinality. For example, say that you have two `MATCH` clauses in a row:

Listing 4.16. Example how multiple `MATCH` clauses affect the query cardinality.

```
MATCH (u:User)
MATCH (t:Tweet)
RETURN count(*) AS numberOfRows,
       count(u) AS countOfUsers,
       count(t) AS countOfTweets
```

Table 4.12. Multiple aggregation in sequence without reducing cardinality

numberOfRows	countOfUsers	countOfUsers
43343640	43343640	43343640

You already know that this result doesn’t make sense at all. First of all, the number of users and tweets is identical, and you definitely don’t have 43 million nodes in the graph. So why do you get these results? Each `MATCH` or `OPTIONAL MATCH` produces a certain number of rows. Any subsequent `MATCH` or `OPTIONAL MATCH` clauses will be executed as many times as the rows produced by the previous `MATCH` clause. The first `MATCH` in Listing 4.22 produces 3594 rows. The second `MATCH` is then executed for each produced row separately. Effectively, the second `MATCH` will be executed 3594 times. There are 12060 tweets in our graph, so if you multiply 12060 * 3594, you will get the 43 million rows.

How do you avoid this problem? In this example, you can reduce the cardinality before the second `MATCH` clause to one, so that the second `MATCH` clause will be executed only once. You can use any of the aggregating functions to reduce the cardinality. Let’s say you want to count the number of users and tweets in the graph. In this case, you can use the `count()` function after the first `MATCH` clause to reduce the cardinality.

Listing 4.17. Reducing cardinality between multiple `MATCH` clauses in a sequence.

```
MATCH (u:User)
# A
WITH count(u) AS countOfUsers
MATCH (t:Tweet)
RETURN count(*) AS numberOfRows, countOfUsers, count(t) AS countO

# A Reduce cardinality to 1 before executing subsequent `MATCH` c
```

Table 4.13. Multiple aggregation in sequence with reducing intermediate cardinality

numberOfRows	countOfUsers	countOfUsers
12060	3594	12060

By reducing the intermediate cardinality after the first `MATCH` to one, you are making sure that any subsequent `MATCH` clauses will be executed only once. This will help you with query performance as well as getting accurate results.

Exercise 4.11

Calculate the mention distribution for the user "IainLJBrown". Mentions can come in three forms:

- Someone retweeted posts from "IainLJBrown"
- Someone posts an original tweet and mentions "IainLJBrown"
- Someone retweets a posts that mentions "IainLJBrown"

Make sure to reduce the cardinality after each `MATCH` or `OPTIONAL MATCH` clause. Because you don’t know beforehand if mentions to "IainLJBrown" fall into all three categories, I advise you to use the `OPTIONAL MATCH` when counting the mentions distribution.

The solution to exercise 4.11 is the following:

Listing 4.18. Calculate the distribution of mentions for user "IainLJBrown"

```
# A
MATCH (u:User)
WHERE u.username = "IainLJBrown"
# B
OPTIONAL MATCH (u)-[:PUBLISH]->(rt)<-[:RETWEETS]-()
WITH u, count(rt) AS numberOfRetweets
# C
OPTIONAL MATCH (u)<-[:MENTIONS]-(t)
WHERE NOT (t)-[:RETWEETS]->()
WITH u, numberOfRetweets, count(t) AS mentionsInOriginalTweets
# D
OPTIONAL MATCH (u)<-[:MENTIONS]-(ort)
```

```
WHERE (ort)-[:RETWEETS]->() AND NOT (ort)-[:RETWEETS]->()<-[:PUBL
WITH u, numberOfRetweets, mentionsInOriginalTweets, count(ort) AS
RETURN u.username AS user, numberOfRetweets,
       mentionsInOriginalTweets, mentionsInRetweets

# A Identify the user
# B Count the number of retweets their posts have received
# C Count the number of mentions in original posts
# D Count the number of mentions in retweets and exclude retweets
```

Table 4.14. Distribution of mentions for "IainLJBrown"

user	numberOfRetweets	mentionsInOriginalTweets	mentionsInRe
"IainLJBrown"	3643	2	1

A very important detail is that you have to use the `count()` operator directly after each `OPTIONAL MATCH` clause and not only at the end. This way, you handle reduce the in-between cardinality to 1 after each `OPTIONAL MATCH` clause and your count won't explode. There are a couple of other ways you could get this result, so if your query is a little different but produces the same results, then it's all ok. Almost all of the mentions for the user "IainLJBrown" come from their posts being retweeted. They were only mentioned in two original tweets, and probably one of them was retweeted once. If you combine the information from Exercise 4.10, you know that 754 of his posts were retweeted 3643. In this Twitter subgraph, he can definitely be regarded as an influencer.

Exercise 4.12

Fetch the top five users who have published the most tweets or retweets. Use the `size()` operator to produce a more performant Cypher statement.

Congratulations, by completing all of the exercises, you have learned about Cypher aggregations and filtering.

4.2 Summary

- Cypher aggregations use implicit grouping keys
- As soon as an aggregation function is used, all non-aggregated columns become grouping keys
- Existential subqueries can help you efficiently filter using graph patterns
- Existential subqueries are especially useful when you want to negate a graph pattern
- Retrieving relationship count can be optimized by using the `size()` operator
- Cardinality is the number of rows or records of the input stream to the operation
- When execution multiple clauses or aggregation functions in sequence, you have to be mindful of the intermediate query cardinality
- You can prefix any Cypher statement with the `PROFILE` clause to evaluate its performance by examining total database hits

4.3 References

[Hawthorne, Joshua et al., 2013] Hawthorne, Joshua & Houston, J. Brian & Mckinney, Mitchell. (2013). Live-Tweeting a Presidential Primary Debate Exploring New Political Conversations. Social Science Computer Review. 31. 552-562. 10.1177/0894439313490643.

[Minot JR et al., 2021] Minot JR, Arnold MV, Alshaabi T, Danforth CM, Dodds PS (2021) Ratioing the President: An exploration of public engagement with Obama and Trump on Twitter. PLOS ONE 16(4): e0248880. doi.org/10.1371/journal.pone.0248880

4.4 Solutions to exercises

The solution to Exercise 4.1 is the following:

Listing 4.19. Generated cypher query that visualizes a sample of 25 `RETWEETS` relationships.

```
MATCH p=()-[r:RETWEETS]->()
```

```
RETURN p LIMIT 25;
```

The solution to Exercise 4.2 is the following:

Listing 4.20. Calculate the ratio of non-null values of `createdAt` node property of tweets.

```
MATCH (u:Tweet)
WITH count(*) AS numberOfRows,
     count(u.createdAt) AS tweetsWithCreatedAtDate
RETURN toFloat(tweetsWithCreatedAtDate) / numberOfRows * 100 AS r
```

The solution to Exercise 4.3 is the following:

Listing 4.21. Count the number of relationships grouped by their type.

```
MATCH ()-[r]->()
RETURN type(r) AS relationshipType, count(r) AS countOfRels
```

The solution to Exercise 4.4 is the following:

Listing 4.22. Cypher statement to compare the retweet and original tweet's text property.

```
MATCH (rt:Tweet)-[:RETWEETS]->(t:Tweet)
RETURN rt.text AS retweetText, t.text AS originalText
LIMIT 1
```

The solution to Exercise 4.5 is the following:

Listing 4.23. Visualize a single graph pattern where a user retweeted a post from another user.

```
MATCH p=(:User)-[:PUBLISH]->(rt:Tweet)-[:RETWEETS]->(t:Tweet)<-[:
WITH p, rt, t LIMIT 1
MATCH prt=(rt)-[:MENTIONS]->()
MATCH pt=(t)-[:MENTIONS]->()
RETURN p,pt,prt
```

The solution to Exercise 4.6 is the following:

Listing 4.24. Calculate the distribution of tweets by created year.

```
MATCH (t:Tweet)
RETURN t.createdAt.year AS year, count(*) AS count
```

```
ORDER BY year DESC
```

The solution to Exercise 4.7 is the following:

Listing 4.25. Calculate the distribution of tweets created in 2021 by month.

```
MATCH (t:Tweet)
WHERE t.createdAt.year = 2021
RETURN t.createdAt.year AS year,
       t.createdAt.month AS month,
       count(*) as count
ORDER BY year DESC, month DESC
```

The solution to Exercise 4.8 is the following:

Listing 4.26. Determine the top four days by the number of tweets created.

```
MATCH (t:Tweet)
WITH t.createdAt.year AS year,
     t.createdAt.month AS month,
     t.createdAt.day AS day,
     count(*) AS count
ORDER BY count DESC
RETURN year, month, day, count LIMIT 4
```

The solution to Exercise 4.9 is the following:

Listing 4.27. Count the number of distinct users who have published at least a single tweet.

```
MATCH (u:User)
WHERE EXISTS { (u)-[:PUBLISH]->() }
RETURN count(*) AS countOfUsers
```

The solution to Exercise 4.10 is the following:

Listing 4.28. Find the top five users who had the most distinct tweets retweeted.

```
MATCH (n:User)-[:PUBLISH]->(t:Tweet)
WHERE EXISTS { (t)<-[:RETWEETS]-() }
WITH n, count(*) AS numberOfRetweets
ORDER BY numberOfRetweets DESC
RETURN n.username AS user, numberOfRetweets
LIMIT 5
```

The solution to Exercise 4.11 is the following:

Listing 4.29. Calculate the distribution of mentions for user "IainLJBrown"

```
# A
MATCH (u:User)
WHERE u.username = "IainLJBrown"
# B
OPTIONAL MATCH (u)-[:PUBLISH]->(rt)<-[:RETWEETS]-()
WITH u, count(rt) AS numberOfRetweets
# C
OPTIONAL MATCH (u)<-[:MENTIONS]-(t)
WHERE NOT (t)-[:RETWEETS]->()
WITH u, numberOfRetweets, count(t) AS mentionsInOriginalTweets
# D
OPTIONAL MATCH (u)<-[:MENTIONS]-(ort)
WHERE (ort)-[:RETWEETS]->() AND NOT (ort)-[:RETWEETS]->()<-[:PUBL
WITH u, numberOfRetweets, mentionsInOriginalTweets, count(ort) AS
RETURN u.username AS user, numberOfRetweets,
       mentionsInOriginalTweets, mentionsInRetweets

# A Identify the user
# B Count the number of retweets their posts have received
# C Count the number of mentions in original posts
# D Count the number of mentions in retweets and exclude retweets
```

The solution to Exercise 4.12 is the following:

Listing 4.30. Fetch the top five users who have published the most tweets or retweets.

```
MATCH (u:User)
RETURN u.username AS username, size((u)-[:PUBLISH]->()) AS countO
ORDER BY countOfTweets DESC
LIMIT 5
```

5 Introduction to social network analysis

This chapter covers

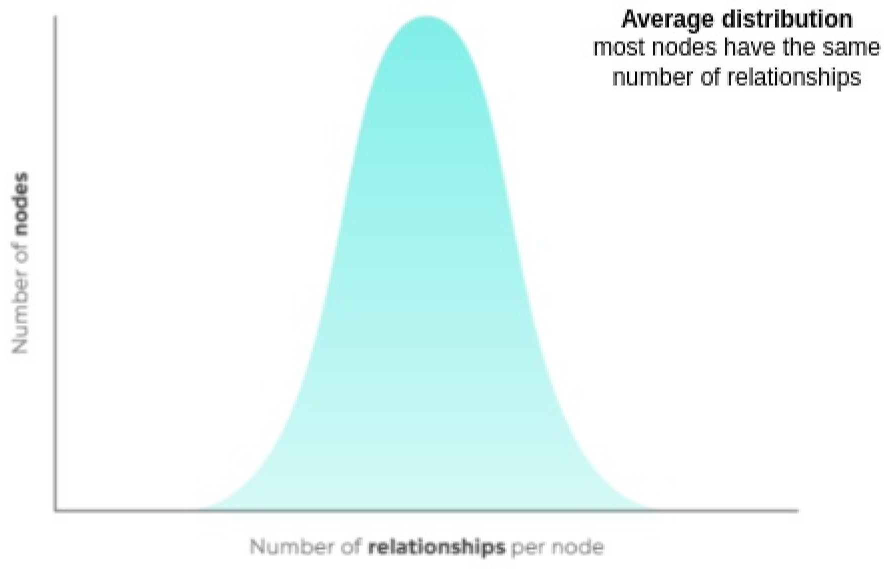
- Random and scale-free degree distribution model
- Using metrics to characterize a network
- Introducting Neo4j Graph Data Science library
- Using Native Projection to project an in-memory graph
- Inspecting the community structure of a graph
- Finding influencers in the network

Social network analysis is a process of investigating network structures and node roles using graph theory and algorithms. One of the earliest people to write about network science was Hungarian author Frigyes Karinthy. He wrote a short story Láncszemek, where he described that even though we think the world is vast, it is, in fact, very tiny. The original story is in Hungarian, but Adam Makkai prepared an English translation. The short story describes a concept that is known today as the small-world concept. To demonstrate his claim, he presented how he could connect himself to someone far from his perspective in 1929. In his example, he showed how he, being in Budapest, could connect to a worker in an American Ford factory. The worker in the Ford factory knows his manager, and that manager probably knows Henry Ford. Henry Ford probably knows an industrialist in Hungary, and that industrialist is perhaps a friend of a friend from Karinthy's perspective. This way, he demonstrated that a Ford company worker is probably four or five handshakes away from an author in Budapest. Over the years, the small-world concept has been rebranded as the six degrees of Kevin Bacon or six degrees of separation.

In the 1950s and 60s, Paul Erdős and Alfréd Rényi started to work on the language to describe a network. In their 1959 paper [Erdős Rényi, 1959], they started to examine how large networks behave. Large networks look so complicated that one might assume they are random. Even in a social network, it is hard to predict who is connected to whom. They assumed those networks must be random as people might randomly meet other people or molecules randomly interact with each other.

An essential aspect of characterizing any network is to look at the node degree distribution. In simple terms, node degree is the count of links each node has.

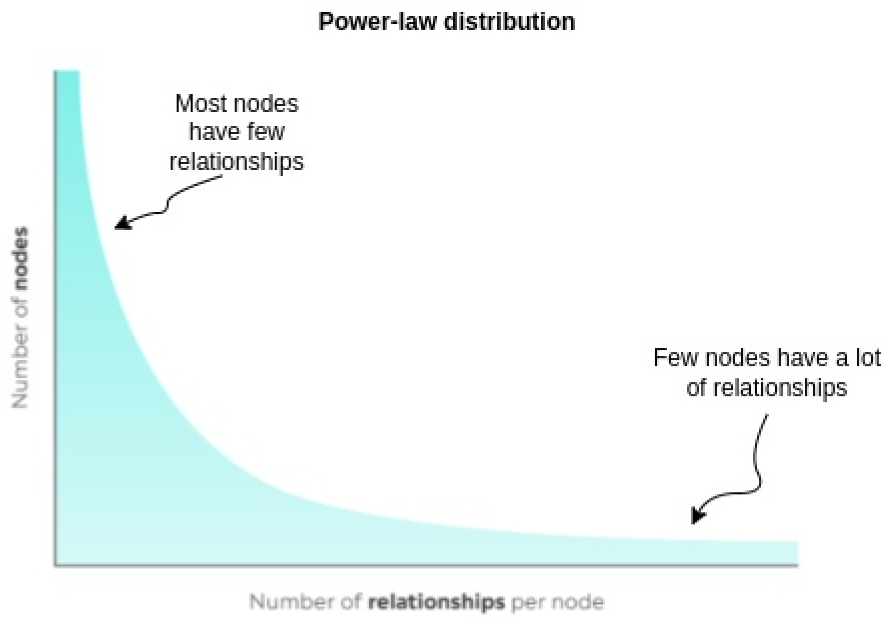
Figure 5.1. Random network degree distribution.



In a random network, the degree distribution will follow the Gaussian distribution. The vast majority of nodes have roughly the same number of links. There won’t be many hugely popular nodes, but there won’t be many isolated nodes either. It turns out that almost no real-world network follows the random network degree distribution. The reason behind this claim is that networks have profound organizing principles. At about the same time Google developed its famous graph algorithm PageRank [S.Brin and L. Page, 1998], Albert Barabási and his colleagues examined the structure of the web. [Albert, R., Jeong, H. & Barabási, 1999] The web consists of web pages and

URL links pointing to other sites. This is essentially a network, where nodes represent web pages and relationships represent their URL links. The assumption was that the web would turn out to be a random network as anyone can publish a web page and choose which sites they want to link to. They discovered that the web degree distribution follows a different pattern.

Figure 5.2. Scale-free degree distribution.



This is a very different degree distribution than expected. On the web, the vast majority of the pages no one cares about. They have one or two links pointing to them. Then there are some pages that have hundreds of millions of links pointing to them. Those pages are Google, Amazon, Yahoo, and others. Such a network is incredibly uneven and is today known as a scale-free network. It was later shown that most of the real-world networks are scale-free networks where a few big hubs hold together many tiny nodes. Why is that so? The truth is that networks have profound organizing principles. For example, think of a group of people. Who is more likely to form new connections, a person with only a few friends or a person that already has many friends? It turns out that a person who already has many friends is more likely to form new relationships. A simple explanation is that they will get invited to more birthday parties and events due to their greater number of existing connections where they can mingle with new folks. Also, they are more likely to get introduced to new people by their existing contacts. This network organizing principle is also known as the *preferential attachment model*, which was made popular by Barabási and Albert[Barabási and Albert, 1999].

5.1 Followers network analysis

Most graph algorithms were designed to be used on monopartite networks. If you recall, a monopartite network contains a single type of nodes and relationships. A typical example is a friendship network where you only have people and their friendship relationships. Another frequently mentioned example is the web network, where you deal with web pages and hyperlinks connecting them. Even when dealing with a multipartite network, it is pretty common to infer or project a monopartite network using various techniques. The next chapter will focus more on inferring monopartite networks.

Here, you will execute your first graph algorithms on the Twitter followers network. Even though the Twitter social graph contains multiple node types and relationships, you can focus your graph analysis on a specific subgraph. The followers network is monopartite as it contains only `User` nodes and `FOLLOWS` relationships. I've chosen it, so you don't have to deal with monopartite projections just yet.

A user can follow another user, but they don't necessarily follow them back. This means you are dealing with a directed network. Also, the relationships don't have any attribute or property that would quantify their strength, which implies you are dealing with an unweighted network.

First, you will learn how to characterize the followers network in terms of connectedness and the density of links. For example, Stanford's SNAP repository contains a variety of graph datasets. If you open the Pokec social network dataset webpage, you can observe that the following characteristics of a network are given along with the data itself.

- Number of nodes
- Number of relationships
- Number of nodes in the largest weakly connected component
- Number of nodes in the largest strongly connected component
- Average local clustering coefficient

To characterize a network, you will be using Cypher query language and graph algorithms. Specifically, in this chapter, you will learn to use some *community detection* and *centrality* graph algorithms. The community detection algorithms will be used to characterize the network and also find tightly-connected groups of users. In the context of networks, a community refers to a densely connected group of nodes, whose members have comparatively fewer connections to other nodes in the network. For example, think of a friendship network.

Figure 5.3. Friendship network with nodes with outlined communities

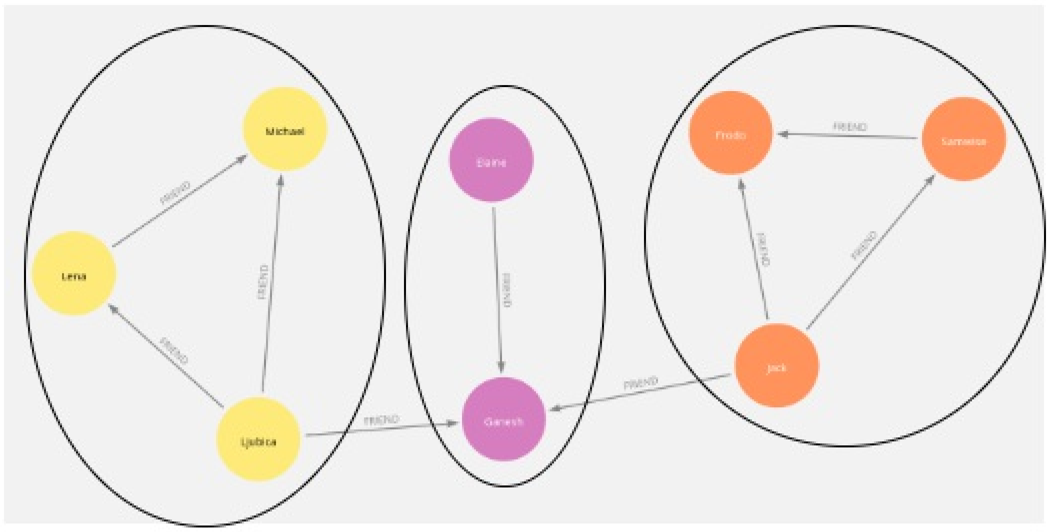


Figure 5.3 visualizes a network of eight people and their friendship connections. The communities are outlines with a circle. You can observe that communities form between nodes that are densely interconnected.

For example, there are three communities in Figure 5.3. On the right side, Frodo, Samwise, and Jack form a community. They all have connections with each other, similar to what you would expect of a group of friends. Although

Jack has a connection with Ganesh, they don’t belong to the same community, as Ganesh doesn’t share any ties to the other friends in Jack’s group. This makes sense if you think about it.

Imagine you have a group of friends with whom you like to go hiking or play board games. Let’s say they are your community of friends. Now, even though you might make a friend at your workplace, that doesn’t automatically make the work friend part of your community. And likely, the work friend also has a separate group of friends they like to play board games with. Only if both your and the work friend community joined and played games together could you consider the two communities merging and becoming a single community with densely connected ties. In that case, you would belong to the same community as your work friend, with whom you now also hang out in your free time.

Community detection techniques can be used to detect various segments of users, discover people with common interests, or recommend new connections within a community. Suppose you take a step away from social networks. In that case, you could use community detection algorithms to group proteins with similar roles or identify physicians' specialties based on the prescription data[Shirazi et al., 2020]. Another application for community detection algorithms is to examine the network structure of scientific collaboration networks[Newman ME, 2001].

Now think about what makes a node have influence over the network. There are a couple of definitions of what makes a node influential. For example, the most basic metric to determine a node’s importance is degree centrality, which simply counts the number of relationships a node has. The higher the count of the relationships, the more influential the node in the network. Another example of node importance is to examine the amount of influence a node has over the flow of information in a network.

Figure 5.4. Friendship network with node size corresponding to their influence over the information flow

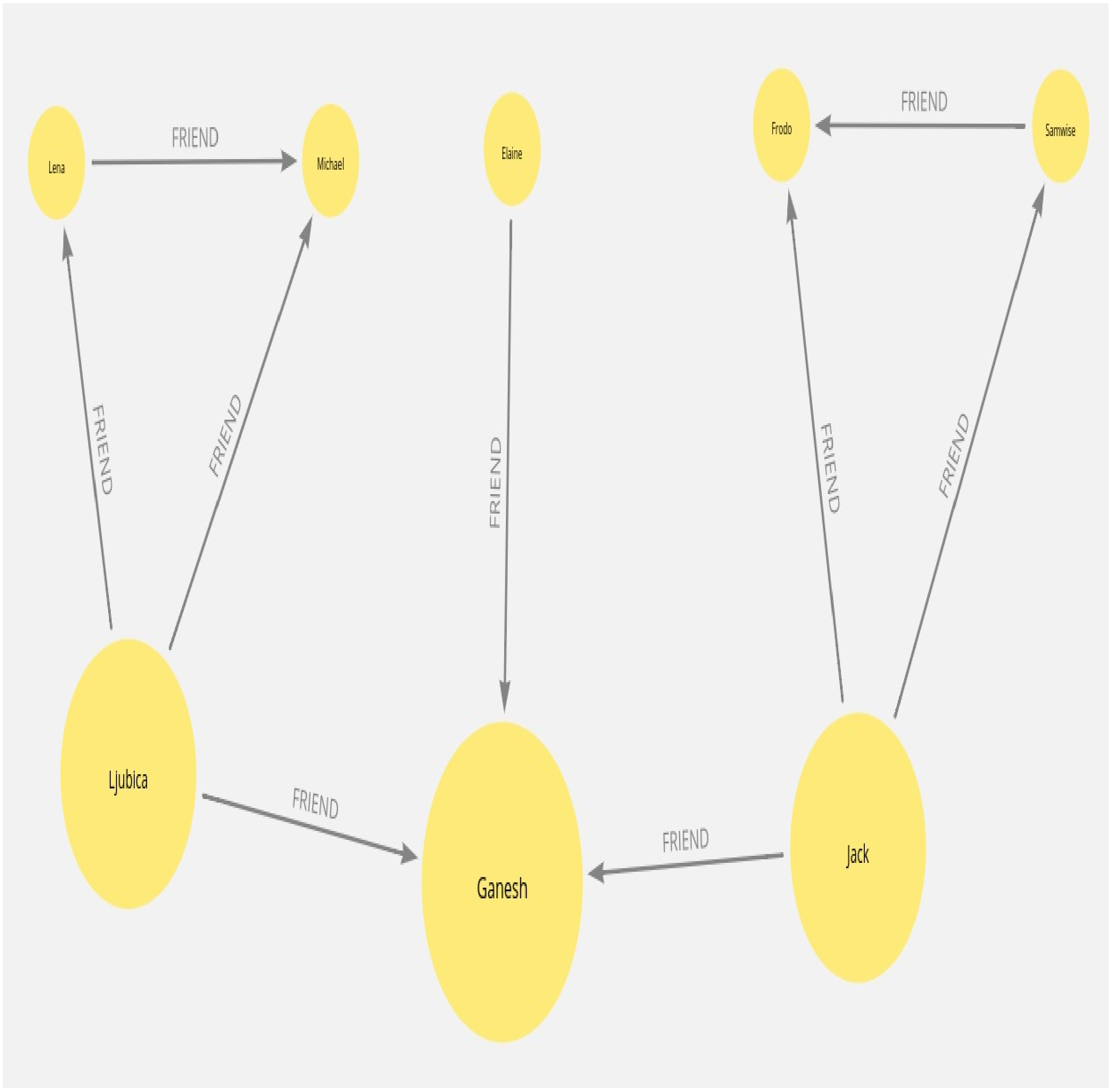


Figure 5.4 visualizes the same network as Figure 5.3. The only difference is that now, the node size in the visualization corresponds to its influence over the information flow. Suppose the information can circulate only through friendship relationships. In that case, Ganesh is the most important node in the network as he is the bridge between all three communities. Ganesh can be thought of as the gatekeeper of information between communities, allowing him to choose which data and when he wants to pass along. Another thing to note is that if Ganesh were removed from the network, it would be broken into three parts. The other two vital nodes are Ljubica and Jack, which connect their community with the rest of the network.

In the last part of this chapter, you will take advantage of centrality algorithms to find the most important or influential users. There are multiple variations of node influence measures. To calculate the influence over the information flow in the network you can use *Betweenness centrality*. Like mentioned, it has various application in social network analysis, but it can also be used to predict congestions in a road network[Kirkley A. et al., 2018]. The most famous node centrality algorithm is probably *PageRank*, which was developed to rank the websites and use the ranking information to produce better search results [S.Brin and L. Page, 1998]. The beauty of the PageRank algorithm is that it can be applied to other domains. For example, it has been used to rank research paper authors based on the citations[Ying Ding et al., 2010]. I’ve also found one example where PageRank is applied to evaluate user reputation on Youtube[Hanm Yo-Sub et al., 2009]. Lastly, it can also be utilized to analyze protein interactions networks[Gábor Iván and Vince Grolmusz, 2011].

Now you can follow practical examples to learn how to utilize Cypher query language and graph algorithms to characterize and evaluate the community structure of the Twitter followers network, followed by identifying the most influential nodes. To follow along with the examples in this chapter, you must have a Neo4j database instance with Twitter network data imported as described in Chapter 3.

5.1.1 Node degree distribution

One of the essential characteristics of a network is the node degree distribution. With a directed network, you can split the degree distribution into *in-degree* and *out-degree* distribution. The node in-degree counts the number of incoming relationships, and the out-degree counts the number of outgoing connections per node.

First, you will examine the out-degree distribution of the followers network. If you want to evaluate any distribution quickly in Neo4j Browser, you can use the *apoc.agg.statistics* function from the APOC library.

Note

The APOC library contains around 450 procedures and functions to help you with various tasks ranging from data integration, batching, and more. While it is not automatically incorporated with Neo4j, I recommend you include it in all your Neo4j projects. You can check out the official documentation to get a sense of all the procedures it features at neo4j.com/labs/apoc/4.4/.

The `apoc.agg.statistics` function returns statistical values such as mean, max, and percentile values of given values. Since, you are only counting the number of relationships per node, you can take advantage of accessing relationship count store with the `size()` function.

Listing 5.1. Evaluate the node out-degree distribution with apoc.agg.statistics function.

```
MATCH (u:User)
WITH u, size((u)-[:FOLLOWS]->()) as outDegree
RETURN apoc.agg.statistics(outDegree)
```

Table 5.1. Out-degree distribution of the followers network

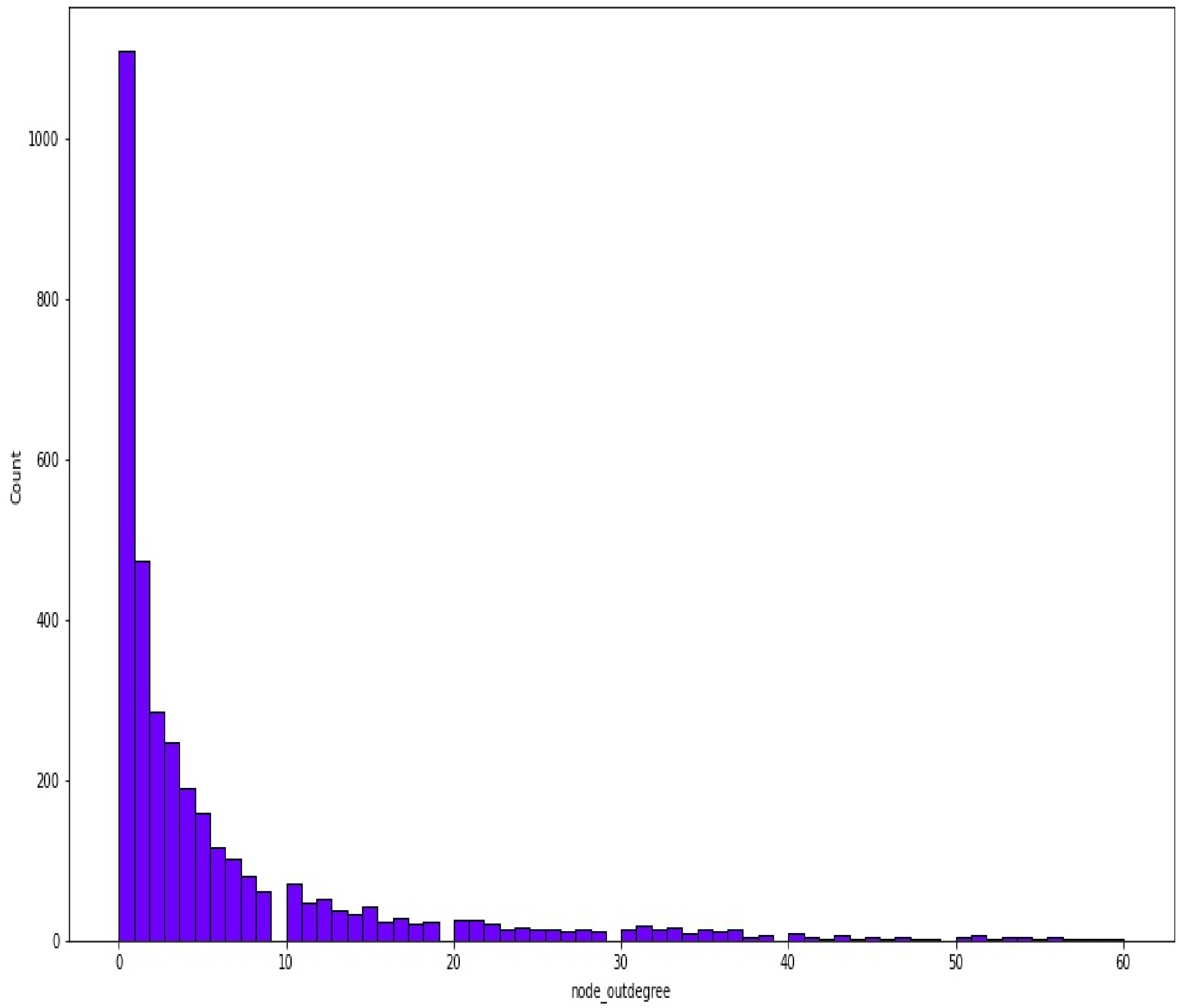
total	3594
min	0
minNonZero	1.0
max	143
mean	6.924874791318865
0.5	2

0.99	57
0.75	8
0.9	21
0.95	32
stdev	11.94885358058576

There are 3594 samples or nodes in the distribution. User nodes have, on average, around seven outgoing relationships. The 0.5 key represents the 50th percentile value, 0.9 key represents the 90th percentile value, and so on. While the average value of outgoing relationships is almost seven, the median value is only 2, which indicates that 50% of nodes have two or fewer outgoing connections. Around 10% of users have more than 21 outgoing relationships.

You can always draw a histogram of out-degree distribution in your favorite visualization library if you are more visually oriented like me.

Figure 5.5. Out-degree distribution chart visualized with a Seaborn histogram.



Visualizing charts is beyond the scope of this book, so I won't go into details of how I produced Figure 5.5. However, I will include the code in the accompanying Jupyter notebook for those who would like to learn how to draw histograms with the Seaborn library in Python.

Interestingly, even a small subgraph of the Twitter network follows the power-law distribution, which is typical for real-world networks. I have limited the bin range to visualize only nodes with an out-degree of 60 or less for chart readability. More than 1000 nodes have zero outgoing connections, and most nodes have less than ten links. You have previously observed that the highest out-degree is 143, and only 5% of nodes have the out-degree higher than 32.

Exercise 5.1

Fetch the top five users with the highest out-degree. Use the `size()` operator to produce a more performant Cypher statement.

The solution to the Exercise 5.1 is:

Listing 5.2. Fetch the top five users with the highest out-degree.

```
MATCH (u:User)
RETURN u.username as user,
       size((u)-[:FOLLOWS]->()) as outDegree
ORDER BY outDegree DESC
LIMIT 5
```

Now, you will repeat the same process to evaluate the in-degree distribution. First, you will use the `apoc.agg.statistics` function to evaluate the in-degree distribution in Neo4j Browser.

Listing 5.3. Evaluate the node out-degree distribution with apoc.agg.statistics function.

```
MATCH (u:User)
WITH u, size((u)<-[:FOLLOWS]-()) as inDegree
RETURN apoc.agg.statistics(inDegree)
```

Table 5.2. Out-degree distribution of the followers network

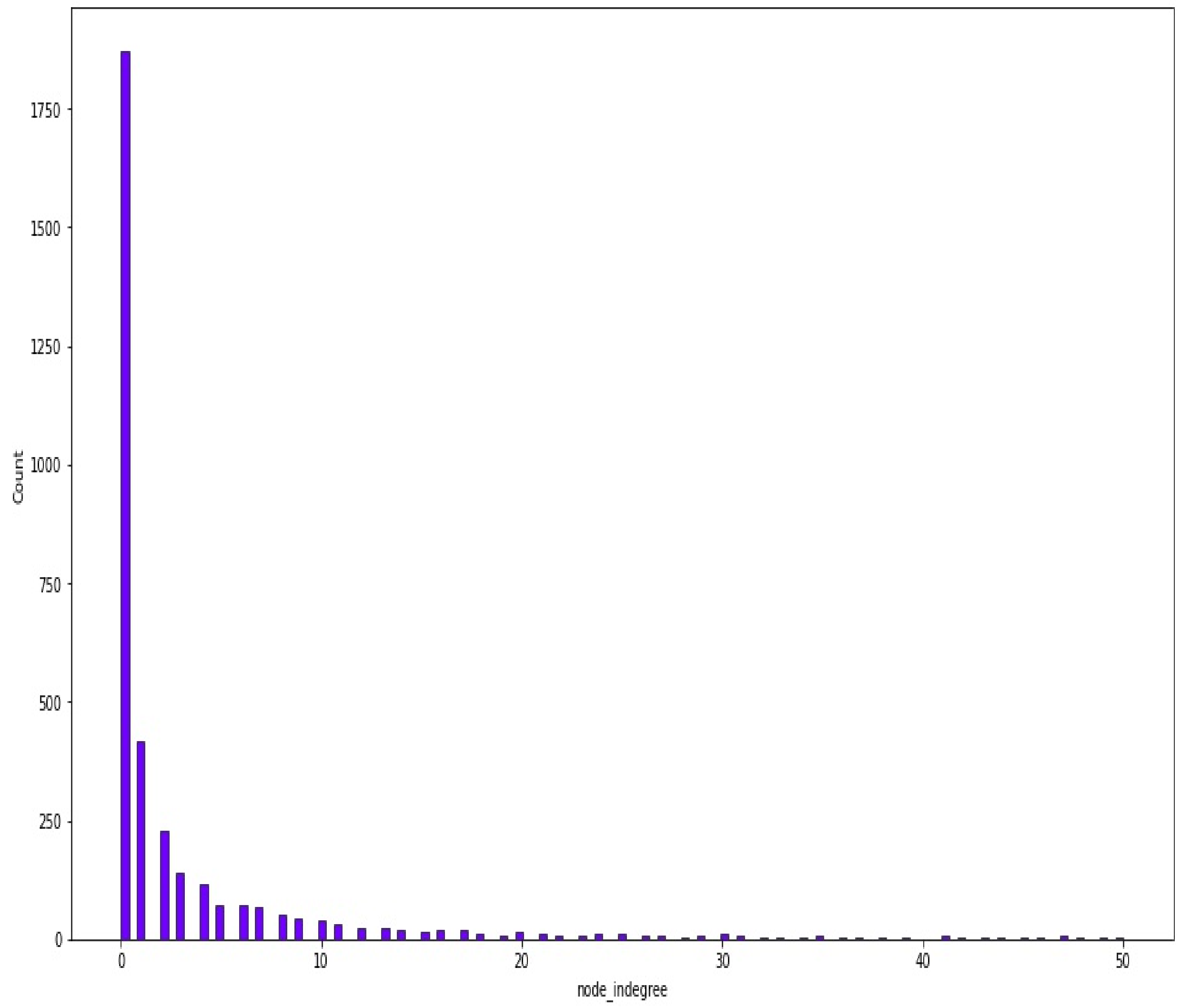
total	3594
min	0
minNonZero	1.0
max	540

mean	6.924874791318865
0.5	0
0.99	112
0.75	4
0.9	16
0.95	35
stdev	22.7640611678852

What immediately caught my eye is that the mean value is identical for out and in-degree. I guess it makes sense, as the total count of nodes and relationships is identical, so the mean values should be the same. More than half of the users have zero incoming connections. While I have scraped the Twitter API for the follower relationships of all users, I have only included relationships between users who have either posted or were mentioned in the 12000 scraped tweets. It looks that around half of the users don’t have any followers included in this subgraph. One outlier has 540 incoming relationships (followers count), meaning that one in seven users follow them.

Again, I’ll visualize the in-degree distribution with the Seaborn library.

Figure 5.6. In-degree distribution chart visualized with a Seaborn histogram.



Though I did not aim to get the power-law distribution of node in and out-degrees, a real-world network tends to exhibit such a distribution.

Exercise 5.2

Fetch the top five users with the highest in-degree (follower count). Use the `size()` operator to produce a more performant Cypher statement.

The solution to the Exercise 5.2 is:

Listing 5.4. Fetch the top five users with the highest in-degree.

```
MATCH (u:User)
```

```
RETURN u.username as user,
       size((u)<-[:FOLLOWS]-()) as inDegree
ORDER BY inDegree DESC
LIMIT 5
```

Table 5.3. Top five users with the highest in-degree

user	inDegree
"elonmusk"	540
"AndrewYNg"	301
"NASA"	267
"OpenAI"	265
"GoogleAI"	264

The highest in-degree users are pretty interesting. Elon Musk takes the crown. It seems that he is popular within the tech community or at least in our Twitter subgraph. The second place takes none other than Andrew Ng. If you have dabbled with any machine learning, you have probably heard of him as he is one of the most famous machine learning instructors.

Exercise 5.3

Remember, I’ve only included users who either published a tweet or were mentioned in one. Pick one of the top five users with the highest in-degree and examine the tweets they published or were mentioned in.

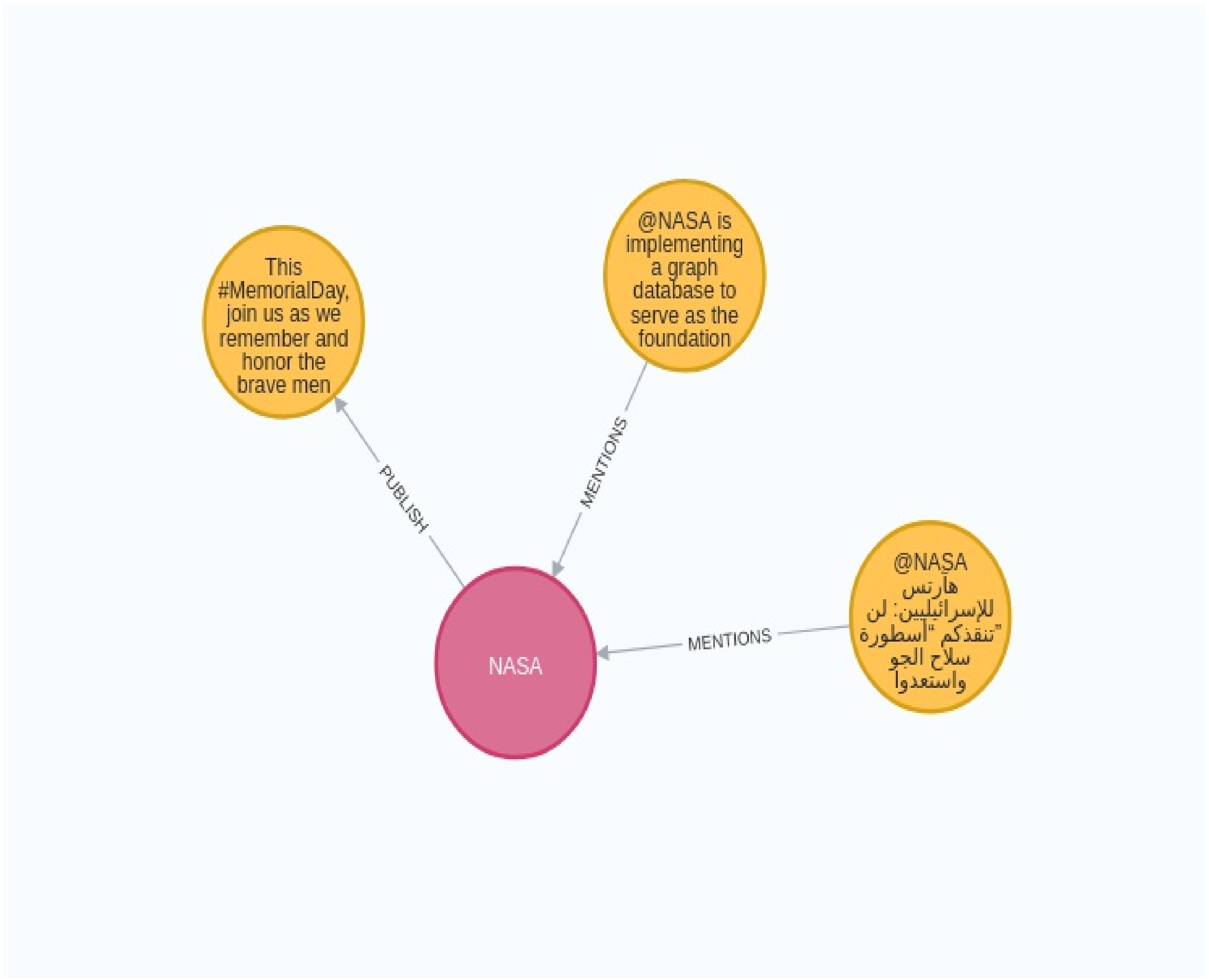
The solution to the Exercise 5.3 is:

Listing 5.5. Examine mentions and published posts for NASA.

```
MATCH (u:User)
WHERE u.username = "NASA"
OPTIONAL MATCH m=(u)<-[:MENTIONS]-()
OPTIONAL MATCH p=(u)-[:PUBLISH]->()
RETURN m,p
```

I’ve chosen to explore NASA’s Twitter. Note that I have used the `OPTIONAL MATCH` as I don’t know beforehand if NASA both published a tweet and was mentioned in one. Again, you could use a couple of variations of the Cypher statement to produce the same results, so don’t worry if you got correct results but used a slightly different Cypher statement. The Cypher statement in Listing 5.5 will produce the following visualization in Neo4j Browser.

Figure 5.7. Network visualization of tweets published by or mentioning NASA.


This #MemorialDay, join us as we remember and honor the brave men
@NASA is implementing a graph database to serve as the foundation
@NASA هآرتس للإسرائيليين: لن تنقذكم "أسطورة" سلاح الجو واستعدوا
PUBLISH
MENTIONS
MENTIONS
NASA

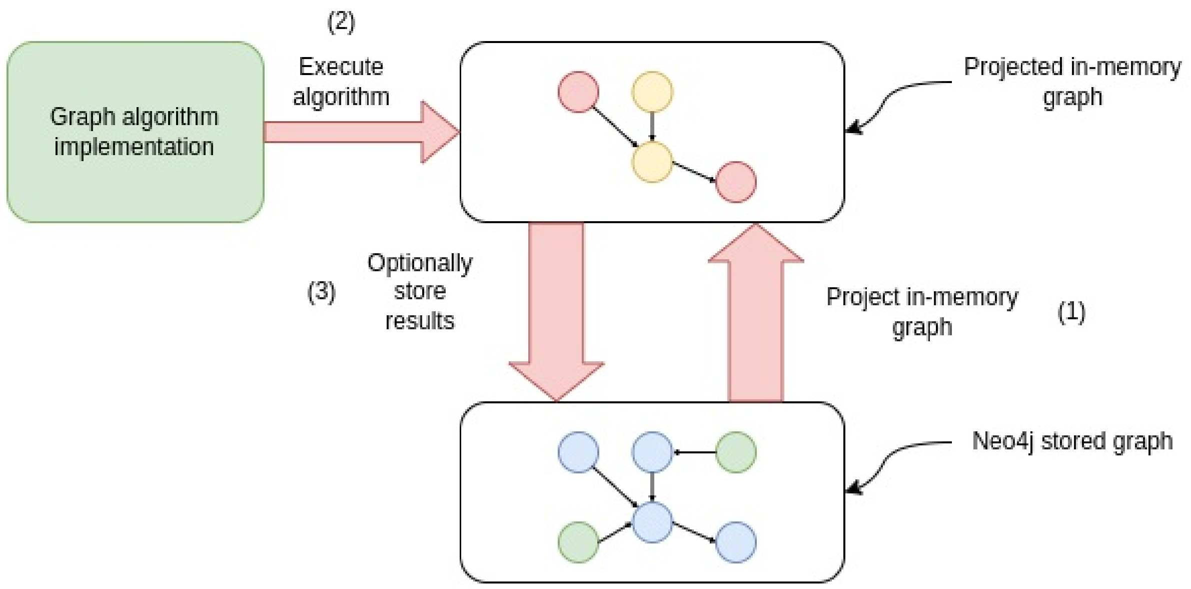
NASA has published a single tweet and was mentioned in two other tweets in our dataset.

5.1.2 Introduction to Neo4j Graph Data Science library

Before continuing with network characterization, you should get familiar with the Neo4j Graph Data Science library (GDS). The Graph Data Science library (GDS) is a plugin for Neo4j that features more than 50 graph algorithms ranging from community detection and centrality to node embedding algorithms and link prediction pipelines, and more. You can get an overview of all available graph algorithms in the official documentation.

Graph algorithms in the GDS library are executed on a *projected in-memory graph* structure separate from the graph stored in the database.

Figure 5.8. Graph Data Science library workflow.



To execute graph algorithms with the GDS library, you first have to project an in-memory graph. The projected graph is stored entirely in-memory using an optimized data structure for scalable and parallel graph algorithm execution. You can create a projected in-memory graph using either *Native projection* or *Cypher projection*.

Native projection is a bit more limited in selecting or filtering a specific subgraph you want to project, as you can only filter based on node labels and relationship types. However, it is the recommended way of projecting a graph as it is highly performant due to reading data directly from Neo4j storage.

The second available option for creating an in-memory graph is the Cypher projection. With it, you get all the flexibility of the Cypher query language to select or filter any specific subgraph you might want to project. Of course, Cypher projection has a drawback as it is slower than Native projection and generally recommended only for the experimental or explorational phase of a project.

As the in-memory graph projection can be costly when dealing with large graphs, the GDS library also features a *Graph Catalog*. Graph Catalog comes in handy when you want to execute multiple graph algorithms on the same projected graph. Instead of having to create an in-memory graph for each algorithm execution separately, you can create an in-memory graph once and then execute multiple graph algorithms on it. The projected graph can then be accessed via its name when executing graph algorithms, so the naming *named graph* stuck with projected graphs stored in a Graph Catalog.

Once the in-memory graph is created, you can execute graph algorithms on top of it. Algorithms come in three tiers of maturity:

- Production-ready: Indicates that the algorithm has been tested with regards to stability and scalability.
- Beta: Indicates that the algorithm is a candidate for the production-quality tier.
- Alpha: Indicates that the algorithm is experimental and might be changed or removed at any time.

Throughout this book, I will mainly try to demonstrate production-ready graph algorithms. I found that there is room for performance improvement with Alpha and Beta tier graph algorithms, but the algorithm results are accurate. So, if you see an algorithm in lower maturity tiers, you can still use it.

Each algorithm has four modes of execution, depending on the use-case.

- stream: returns results as a stream of records and does not store results
- stats: returns a summmary statistics of the result and does not store results
- mutate: writes the results back to the projected in-memory graph. This mode can only be used in combination with a named graph stored in a Graph Catalog. It is very useful when you want to use an output of one graph algorithm as an input to another.
- write: writes the results back to the Neo4j database graph.

5.1.3 Graph Catalog and Native projection

I am a firm believer that one learns best through practical examples. You will begin with network characterization and learn the GDS syntax and algorithm use-cases through examples.

First off, you need to project an in-memory graph. You will use Native Projection to create an in-memory graph that consists of `User` nodes and `FOLLOWS` relationships. The Native Projection syntax is as follows:

Listing 5.6. Native projection syntax to create a named graph in Graph Catalog.

```
CALL gds.graph.project(
    graphName,
    nodeProjection,
    relationshipProjection,
    optional configuration
)
```

GDS procedures are executed using the `CALL` clause in combination with the procedure name. The procedure to store a named graph in Graph Catalog with Native projection is called `gds.graph.project()`. It contains three mandatory and one optional parameters. The first parameter is used to name the graph under which it will be accessed when executing graph algorithms. The second parameter called `nodeProjection` defines the subset of nodes you want to project. Similarly, the `relationshipProjection` parameter specifies which relationships should be considered when creating an in-memory graph. One important thing to note is that a relationship will be skipped during projection if both adjacent nodes are not described in the `nodeProjection` parameter. In GDS terms, the starting node of the relationship is called the *source* node, and the end node is called the *target* node.

To project the followers network, you need to only include `User` nodes and `FOLLOWS` relationships with no additional configuration.

Listing 5.7. Project an in-memory graph consisting of `User` nodes and `FOLLOWS` relationships.

```
CALL gds.graph.project('follower-network', 'User', 'FOLLOWS')
```

The Cypher statement in Listing 5.7 uses Native Projection to store an in-memory graph. The first parameter specifies its name that will be used to access it when executing graph algorithms. The second parameter defines which nodes you want to include in the projection. When you only want to project a single type of nodes, you can define the desired node label as a string. Similarly, when you only want to project a single type of relationships in the third parameter, you can specify the type as a string.

In later chapters, you will learn more about Native Projection and how to create an in-memory graph consisting of multiple node labels and relationship types. The GDS library also supports projecting node and relationship properties, which is useful when dealing with weighted networks.

5.1.4 Weakly Connected Component algorithm

The first graph algorithm you will execute is the *Weakly Connected Component* algorithm, or WCC in short. It is used to find disconnected parts or islands within a network. The WCC algorithm is probably a graph algorithm that should be executed as the first step of any graph analysis to evaluate graph connectivity.

Figure 5.9. Network visualization of two weakly connected components.

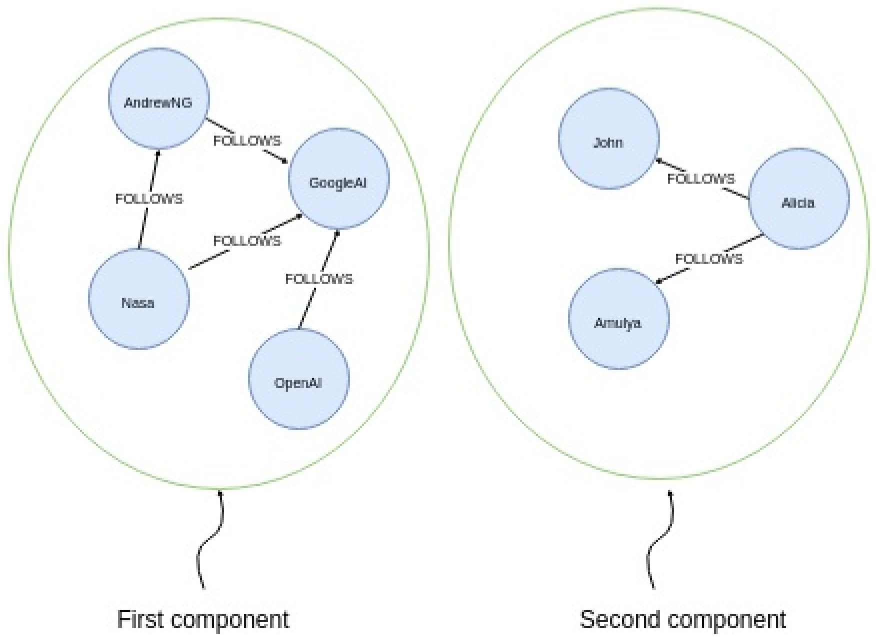


Figure 5.9 visualizes two weakly connected components. One component contains John, Alicia, and Amulya. The other component contains OpenAI, GoogleAI, NASA, and AndrewNG.

Nodes within a single weakly connected component can reach all the other nodes if you ignore the relationship direction. For example, John can reach Alicia even though the relationship is pointed in the other direction. Effectively, you could say that the relationships are treated as undirected. The algorithm considers that all nodes within the same community can reach each other if a path exists between them irrespective of the relationships' direction.

You will execute the WCC algorithm using the `write` mode. As mentioned, the write mode stores the results back to Neo4j database, but also provides summary statistics of the algorithm result. The syntax for graph algorithm procedures in GDS is:

Listing 5.8. Graph algorithm procedure syntax.

```
CALL gds.<algorithm>.<mode>(namedGraph, {optional configuration})
```

When using the `write` mode of an algorithm, you need to provide the mandatory `writeProperty` parameter, which specifies the name of the node property that the algorithm results will be stored to. The procedure to execute the WCC algorithm is `gds.wcc`.

You can execute the WCC algorithm on the followers network and store the results to Neo4j by using the following Cypher statement:

Listing 5.9. Execute the WCC algorithm on the follower network and store the results as a `followerWcc` node property.

```
CALL gds.wcc.write('follower-network', {writeProperty:'followerWc
YIELD componentCount, componentDistribution
```

Table 5.4. Summary statistics for the WCC algorithm executed on the followers network.

componentCount	componentDistribution
	{ "p99": 3, "min": 1,

547	"max": 2997, "mean": 6.570383912248629, "p90": 1, "p50": 1, "p999": 5, "p95": 2, "p75": 1 }

The `write` mode of the algorithm stores the results to the Neo4j database and provides the following summary statistics. The node property that contains the algorithm results identifies the component id to which the node belongs to.

There are 547 disconnected components in the followers network, and the largest contains 2997 members. Most real-world networks have a single connected component containing most of the nodes in the network and a couple of disconnected peripheral components. As the dataset you are analyzing is rather small, it is not unusual to have a higher count of components. I’ve found one analysis on the Twitter network[Myers et al., 2014], where the authors analyzed a snapshot of the Twitter network from 2012 with 175 million users and 20 billion follow relationships. The analysis revealed that around 93% of the users belong in the largest weakly connected component. Another analysis examined the Facebook graph[Ugander et al., 2011], where they learned that the largest weakly connected component contains more than 99% of all nodes.

The `p90` result or the 90th percentile of the component size has a value of 1. This indicates that 90% of the components have only a single member. When a component contains only a single member, this means that the particular node has no relationships.

Exercise 5.4

Count the number of members for the largest five weakly connected components. The component ids are stored under the `followerWcc` property of the `User` nodes. Use the `followerWcc` property as a grouping key in combination with the `count()` function to count the number of members by component.

The result to Exercise 5.4 is:

Listing 5.10. Count the number of users for the largest five weakly connected components.

```
MATCH (u:User)
WITH u.followerWcc AS componentId, count(*) AS countOfMembers
ORDER BY countOfMembers DESC
RETURN componentId, countOfMembers
LIMIT 5
```

Table 5.5. Member count for the largest five weakly connected components.

componentCount	countOfMembers
0	2997
1293	5
1049	3
269	3
335	3

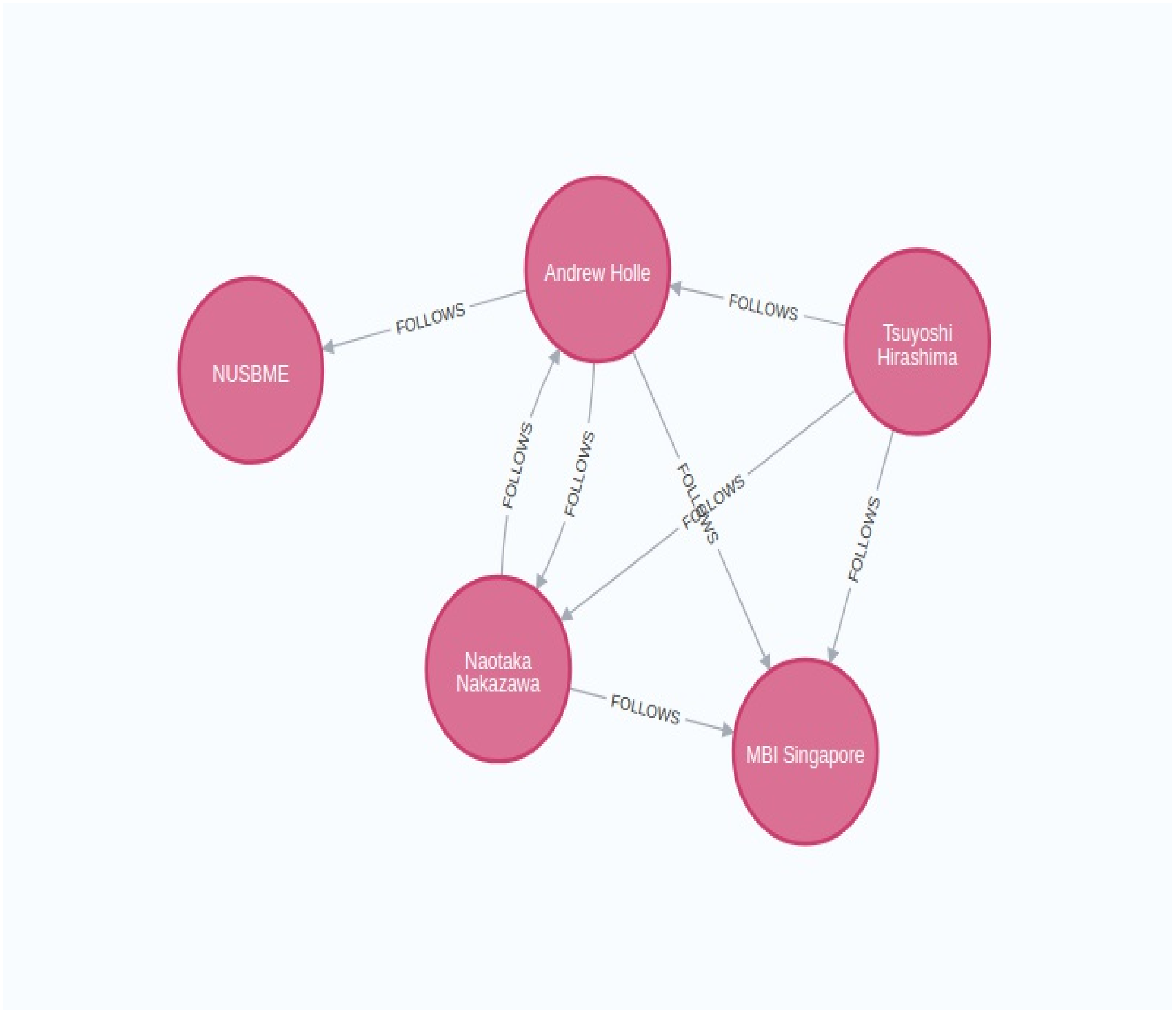
You can observe that there is a single component, which contains 85% of the nodes in the network. The component ids are not deterministic, meaning that you can get different values for the component ids. However the component member distribution should be identical.

The second largest component contains only five members. You can visualize the second largest component in Neo4j Browser with the following Cypher statement. If you have different component ids, make sure to change the component id in the `WHERE` clause.

Listing 5.11. Retrieve User nodes that have no outgoing FOLLOWS relationships.

```
MATCH p=(u:User)-[:FOLLOWS]->()
WHERE u.followerWcc = 1293
RETURN p
```

Figure 5.10. Network visualization of the second largest weakly connected component in the follower network.



I’ve inspected the usernames shown in Figure 5.10 on Twitter and it seems that they are some professors from Singapore and Kyoto universities. They follow each other, but don’t have any connections to the rest of the network in our small Twitter snapshot.

Exercise 5.5

Identify the number of weakly connected components that contain only a single member. Remember, if a weakly connected component contains only a single member, this effectively means that the particular node has no incoming or outgoing relationships. Instead of using the `followerWcc` property to count those components, you can simply filter the `User` nodes that have no `FOLLOWS` relationships and count them. The count will be identical to the number of weakly connected components with a single member.

5.1.5 Strongly Connected Components algorithm

The only difference between the Weakly and *Strongly Connected Components* algorithm (SCC) is that the SCC algorithms considers relationship directions.

Figure 5.11. Network visualization of four strongly connected components.

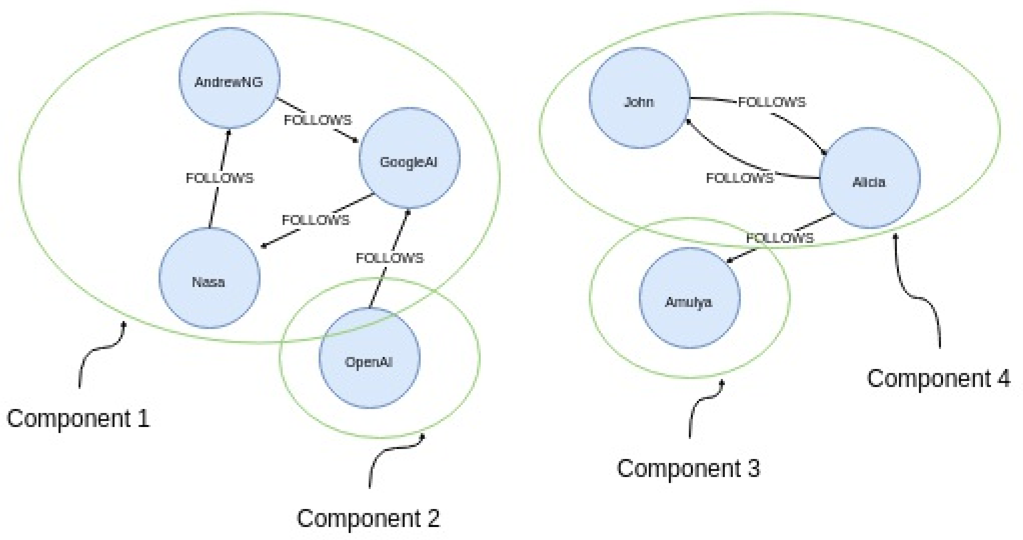


Figure 5.11 visualizes four strongly connected components. The first component contains NASA, AndrewNG, and GoogleAI. You can notice that although OpenAI can reach all of the nodes in the first component, the path from GoogleAI to OpenAI is not possible, as the SCC algorithm does not ignore the relationship direction. Similarly, Amulya in the third component can be reached by Alicia and John, but a directed path from Amulya to either John or Alicia does not exist.

The Strongly Connected Component algorithm is useful when directed paths and reachability plays an important role. For example, imagine a road network, where the nodes represent intersections and relationships represent road connections. For example, many large city centers have a lot of one-way road connections. Using the SCC algorithm, you could evaluate the consequences of closing a single or multiple road connections and how it would affect the reachability of places within the city.

In the context of Twitter, the Strongly Connected Component can be applied to identify smaller well-connected groups of nodes. One research paper[Swati et al., 2016] claims that one could use the SCC algorithm to identify groups of users for more precise marketing targeting. Another article[Efstathiades, Hariton et al., 2016] used the SCC algorithm to suggest a movement of the user only following popular users while not making many connections with other un-popular users. The result is an increased number of strongly connected components over time.

The SCC algorithm in the GDS library is in the Alpha tier but that doesn't mean that you can't use it. The only thing I would be wary of is using it on graphs with several millions of nodes as there is no guarantee that it scales well.

Again, you will use the `write` mode of the algorithm to store the results back to the Neo4j database.

Listing 5.12. Execute the WCC algorithm on the follower network and store the results as a `followerScc` node property.

```
CALL gds.alpha.scc.write('follower-network', {writeProperty:'foll
YIELD nodes, communityCount, p95, p99, p100, maxSetSize
```

Table 5.6. Summary statistics for the SCC algorithm executed on the followers network.

nodes	communityCount	p95	p99	p100	maxSetSize
3594	2704	1	2	796	796

As expected, the count of strongly connected components is higher than the count of weakly connected components. There are 2704 strongly connected components and the largest one contains 796 members.

Exercise 5.6

Count the number of members for the largest five strongly connected components. The component ids are stored under the `followerScc` property of the `User` nodes.

The result to Exercise 5.6 is:

Listing 5.13. Count the number of users for the largest five weakly connected components.

```
MATCH (u:User)
WITH u.followerScc AS componentId, count(*) AS countOfMembers
ORDER BY countOfMembers DESC
RETURN componentId, countOfMembers
LIMIT 5
```

Table 5.7. Member count for the largest five strongly connected components.

componentCount	countOfMembers
0	796
380	20
407	7
36	6
212	4

Similarly as with the WCC algorithm, the community ids are not deterministic. You could get different community ids, but should get the same counts.

Exercise 5.7

Visualize the second-largest strongly connected component in Neo4j Browser. A node can have relationships to nodes in other strongly connected components, so you have to apply a filter to ensure all the nodes are in the second-largest strongly connected component.

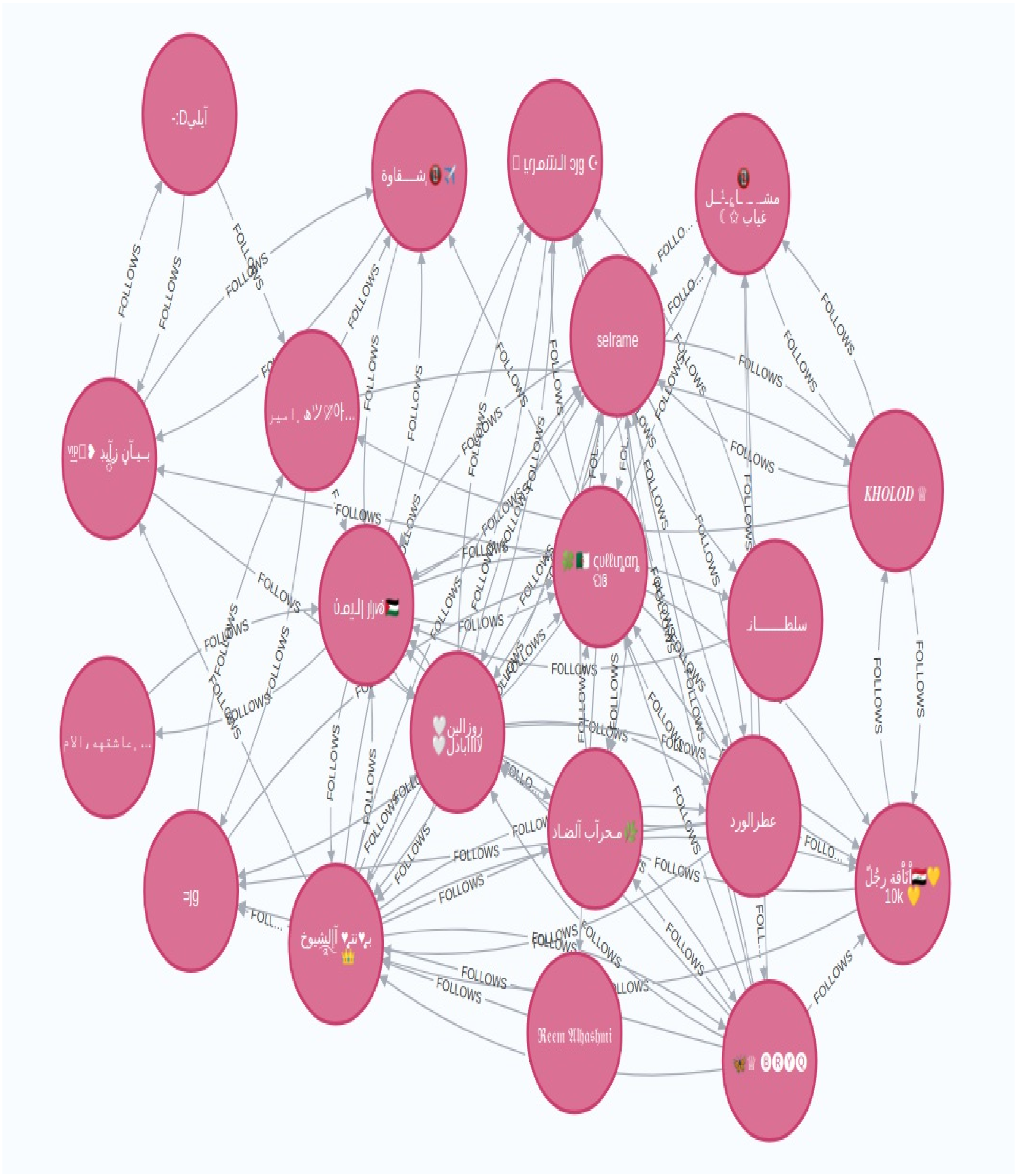
The solution to Exercise 5.7 is:

Listing 5.14. Retrieve User nodes that have no outgoing FOLLOWS relationships.

```
MATCH p=(u1:User)-[:FOLLOWS]->(u2:User)
WHERE u1.followerScc = 380 AND u2.followerScc = 380
RETURN p
```

Again, make sure that you correct the `followerScc` value if needed in the `WHERE` clause. The Cypher statement in Listing 5.14 will produce the following network visualization in Neo4j Browser.

Figure 5.12. Network visualization of the second largest strongly connected component in the follower network.

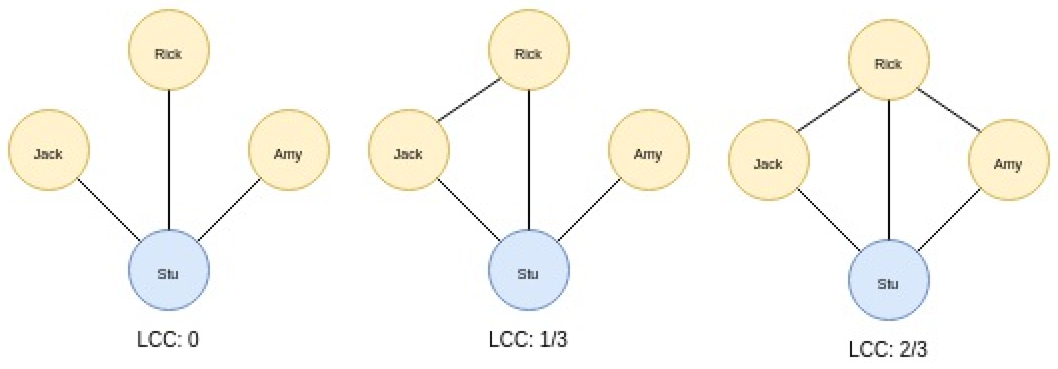


By looking at the visualization in Figure 5.12, you can observe that this community is tightly-knit as there are many connections between the nodes in the group. Judging by the usernames, it seems they all come from the same part of the world. Unfortunately, I am not a language expert, so I have no idea which part of the world it is.

5.1.6 Local clustering coefficient

The *local clustering coefficient*, or LCC for short, is a metric that quantifies how connected or close the neighbors of a particular node are. The LCC value ranges from zero to one. The LCC value of 0 indicates that the neighboring nodes have no connections between each other. On the other hand, the LCC value of 1 indicates that the network of neighbors forms a complete graph, where all the neighbors are connected.

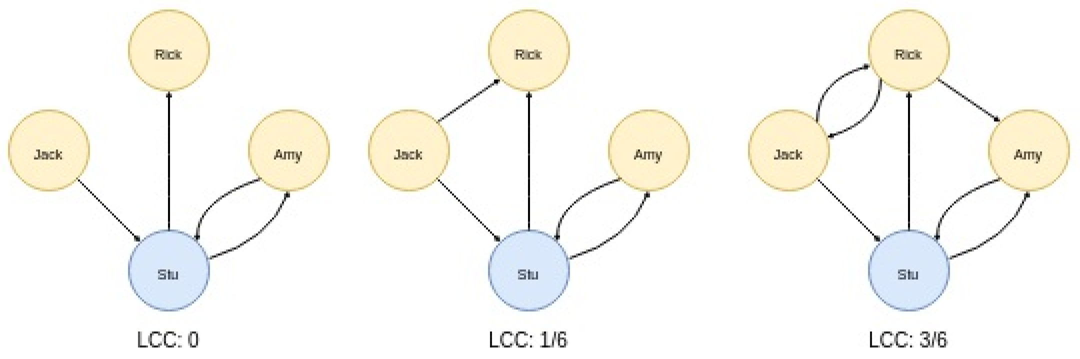
Figure 5.13. Local clustering coefficient values for an undirected graph.



The local clustering coefficient is more easily understood on an undirected graph. For example, in Figure 5.13, Stu has three neighbors. When none of their neighbors has any connections to other neighbors, the LCC value is 0. Thus, Stu has the LCC value of 0 in the left example of Figure 5.13. In the middle example, Stu has the LCC value of 1/3 or 0.33. Stu has three neighbors, so combinatorically, there are three possible relationships between them. As there is only one connection in the middle example of Figure 5.13 between Stu's neighbors, the LCC value for Stu is 1/3. The right-hand side example has two connections between Stu's neighbors, so consequently, the LCC value for Stu is 2/3. If another relationship was created between Jack and Amy, then all the neighbors of the Stu would form a complete graph, which would change Stu's LCC value to 1.

The local clustering coefficient (LCC) algorithm provides a metric to evaluate how strongly the neighbors of a node are connected. You can calculate the LCC value of a single node by dividing the number of existing links between neighbor nodes with the number of possible links between neighbor nodes. You can use the following formula to calculate the LCC on a directed graph as well.

Figure 5.14. Local clustering coefficient values for an directed graph.



With a directed graph, the first difference is that a node has a neighboring node if it has at least a single connection to it. Even though Stu has four connections in Figure 5.14, they only have three distinct neighbors. A neighbor of a node can have one incoming or outgoing connection to the original node or both. Only the count of distinct neighbors is important with the LCC algorithm. With a directed graph, each pair of neighbors can have up to two relationships between them, so the total possible number of connections between three neighbors is six. Again, you only need to count the number of existing connections between neighbors and divide it by the number of possible connections.

Unfortunately, the GDS library only supports the LCC algorithm for an undirected graph. However, as the directed LCC only counts the number of neighboring nodes and their links, you can easily implement the algorithm using only Cypher query language.

Use the following Cypher statement to calculate the directed LCC value of each node and store the results under the `lcc` node property.

Listing 5.15. Calculate the local clustering coefficient on the directed followers network.

```
#  A
```

```
MATCH (u:User)
# B
OPTIONAL MATCH (u)-[:FOLLOWS]-(n)
WITH u,count(distinct n) as neighbors_count
# C
OPTIONAL MATCH (u)-[:FOLLOWS]-()-[r:FOLLOWS]-()-[:FOLLOWS]-(u)
WITH u, neighbors_count, count(distinct r) as existing_links
# D
WITH u,
     CASE WHEN neighbors_count < 2 THEN 0 ELSE
       toFloat(existing_links) / (neighbors_count * (neighbors_co
# E
SET u.lcc = lcc

# A Match all User nodes
# B Count the number of their distinct neighbors
# C Cound the number of distinct link between neighbors
# D Calculate the LCC value
# E Store the LCC value under the lcc node property
```

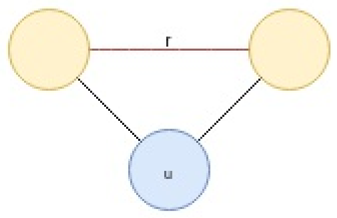
You should already be familiar with most of the Cypher syntax in Listing 5.15. You start by matching all the users in the database. Next, you count the number of distinct neighbors. As some `User` nodes don't have any `FOLLOWS` relationships, you must use the `OPTIONAL MATCH` clause. Using the `MATCH` clause would reduce the cardinality and effectively filter out all the `User` nodes that don't have any `FOLLOWS` relationships. If you remember from the Weakly Connected Components example, there are around 500 `User` nodes that don't have any `FOLLOWS` relationships. Another tiny detail is that the Cypher pattern in the `OPTIONAL MATCH` does not provide a relationship direction. You want to count the number of distinct users irrespective of if they have incoming, outgoing, or both relationships to the original node. As some neighbors can have both incoming and outgoing connections with the original `User` node, you need to use the `distinct` prefix within the `count()` function to get the correct result. The only variable missing before calculating the local clustering coefficient is the count of existing links between neighbors. Again, you should use the `OPTIONAL MATCH` clause as some neighbors might have zero connections, and you don't want to filter those out. I really like the Cypher syntax expressivity of defining the graph pattern that will count the number of links between neighboring nodes.

OPTIONAL MATCH (u)-[:FOLLOWS]-()-[r:FOLLOWS]-()-[:FOLLOWS]-

(u)

You can observe that I used the reference variable `u` twice in this pattern. Effectively, this graph pattern describes all triangles that the `u` node participates in.

Figure 5.15. Visualized Cypher pattern to identify triangles.



As you can see, both the nodes have to be adjacent to the node `u` as described in the Cypher syntax. You are only interested in counting the `r` relationships between neighbors, so you assign a reference variable to it and combine the `count()` function and `distinct` prefix to fetch the count of existing links between neighbors. Similarly, as before, the `FOLLOWS` relationships in the specified graph pattern have no specified direction as you want to consider all possible variations of relationship directions.

Lastly, you can use the LCC algorithm formula to calculate the LCC values for each node.

Equation 5.1

$$\frac{existing\ links}{neighbor\ count * (neighbor\ count - 1)}$$

Equation 5.1 can be used to calculate the directed LCC values. You take the count of existing links and divide it by the possible count of connections between neighbors which is neighbors count times neighbors count - 1. The formula does not work for nodes with the neighbor count of zero or one, as you would end up dividing by zero. By definition, the LCC value for nodes with less than two neighbors is undefined. However, I’ve come across some implementations where they use 0 instead of undefined value for nodes with less than two neighbors, which is what I have also decided to use in this example. I’ve introduced the `CASE` statement to automatically assign the LCC value of 0 for nodes with less than two neighbors. If you have some experience with SQL query language, you will notice that the `CASE` statement is identical in Cypher. In any case, the Cypher syntax for `CASE` clause is:

CASE WHEN predicate THEN x ELSE y END

The predicate value should be a boolean. You can then select the `x` value if the boolean is true or the `y` value if the predicate is false.

Lastly, you store the calculated LCC value under the `lcc` property of `User` nodes. Now that the LCC values are stored in the database, you can go ahead and calculate the average local clustering coefficient.

Listing 5.16. Calculate the average local clustering coefficient.

```
MATCH (u:User)
RETURN avg(u.lcc) as average_lcc
```

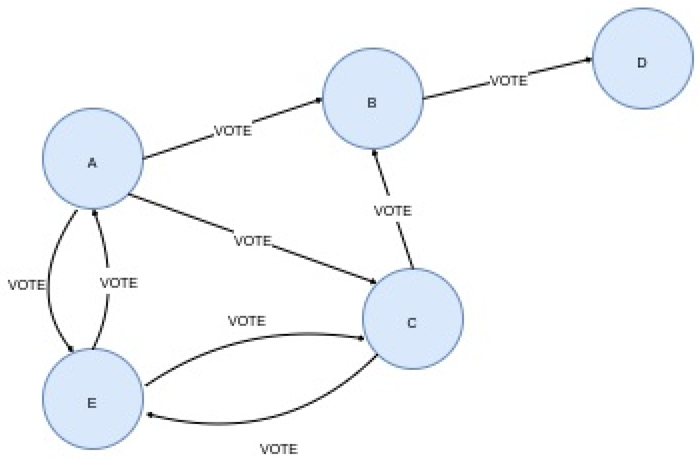
The average local clustering coefficient is 0.06. That’s quite close to 0. One reason for such a small LCC value is that we only have a tiny snapshot of the Twitter network, so the information about followers is limited. Research on a more extensive Twitter network[Myers et al., 2014] demonstrates that the average LCC values are closer to between 0.15 and 0.20. It also seems that users on Twitter are less tightly-knit than on Facebook[Ugander et al., 2011]. That makes sense as one typically connects with their friends and family on Facebook, which is a more strongly connected group of users. On the other hand, one study[Efstathiades, Hariton et al., 2016] suggests that Twitter users prefer to follow elite users or influencers and not connect with their family and real-life friends or neighbors as much.

5.1.7 Finding influencers with the PageRank algorithm

PageRank is probably one of the most famous graph algorithms. It was designed by Larry Page and Sergey Brin [Brin and Page, 1999] and helped make Google Search what it is today. It measures the transitive or directional influence of nodes. For example, the node degree quantifies the influence or importance of a node by considering only its direct neighbors. In contrast, PageRank also considers the indirect relationships with other nodes in the graph spanning over multiple hops. To put it into our Twitter subgraph context, if, for example, Elon Musk or Andrew Ng follows you, you gain more influence than if I followed you. PageRank evaluates the number of followers a particular node has as well as how influential those followers are.

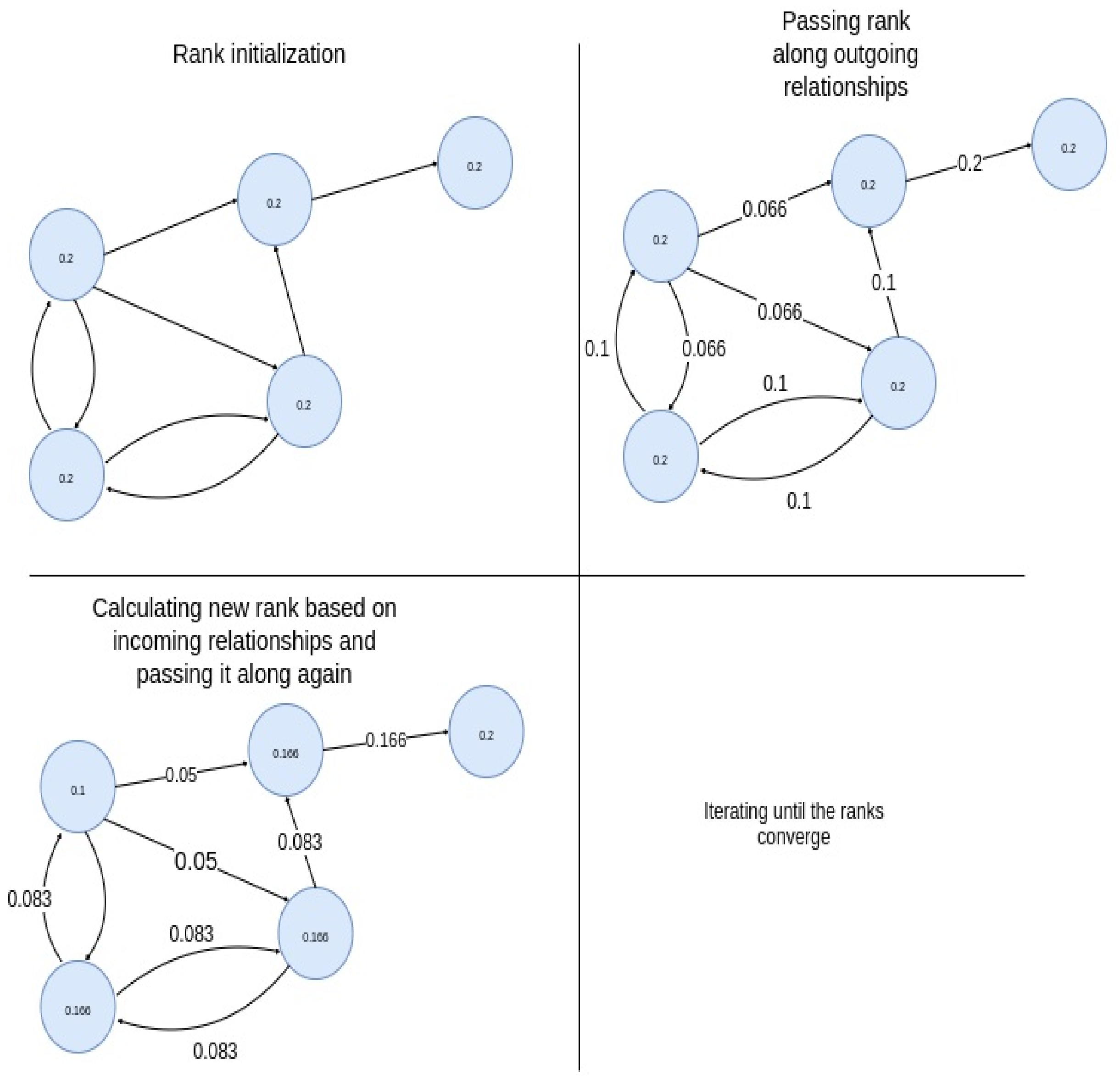
PageRank was initially developed for ranking web pages importance. The algorithm considers every relationship as a vote of influence. I like to think that if a node is pointing to another node, it essentially states that the other node is important or influential.

Figure 5.16. PageRank treats every relationship as a vote of influence.



You can then imagine how the votes flow throughout the network via directed relationships. Each node is initialized with its score being equal to 1 divided by the number of nodes. Then, it passes its rank through its outgoing connections. The amount of influence passed through every relationship equals to the node’s influence divided by the number of outgoing links. After the first iteration, the node’s rank is equal to the sum of incoming scores from other nodes. The algorithm then iterates this process until it converges or until it hits a predefined number of iterations.

Figure 5.17. Simplified PageRank calculation based on network flow.


Rank initialization
0.2
0.2
0.2
0.2
0.2
Passing rank along outgoing relationships
0.066
0.2
0.1
0.066
0.1
0.066
0.1
0.1
Calculating new rank based on incoming relationships and passing it along again
0.1
0.166
0.2
0.166
0.166
0.05
0.166
0.083
0.05
0.083
0.083
0.083
Iterating until the ranks converge

However, the simplified PageRank calculation based on the network flow has a critical flaw. Node D in Figure 5.17 has no outgoing links. A node without any outgoing connections is also known as a *dead end*. The presence of dead ends will cause the PageRank score of some or all nodes in the network to go down to zero as it effectively leaks the rank score out of the network.

The PageRank algorithm introduces the *teleportation* ability to avoid rank leaking. The teleportation introduces a small probability of jumping to a random node instead of following the outgoing links. In the context of exploring web pages, imagine a web surfer traversing the internet. They might follow outgoing links from a web page to a web page or get bored and jump to a random page. The constant that defines the probability a surfer will follow an outgoing link is called the *damping factor*. Consequently, the probability that they will jump to a random page is 1 - damping factor. The typical value of the damping factor is 0.85, indicating that a surfer will jump to a random page about 15% of the time. With the standard PageRank algorithm, the jump to a random node is uniformly distributed between all nodes in the network, meaning that a bored surfer has equal chances to jump to any node in the graph, including the one he is currently visiting. In this case, the intermediate node's rank after each iteration sums both the outgoing link's rank as shown in Figure 5.17 and adds the probability that a surfer will randomly jump to that node. The teleportation ability fixes the scenario where a dead end node leaks the whole network PageRank score, which would effectively be leaving all nodes at rank value zero.

You can execute PageRank on the followers networks with the following Cypher statement:

Listing 5.17. Execute PageRank on the followers network.

```
CALL gds.pageRank.write('follower-network', {writeProperty:'follo
```

Exercise 5.8

Retrieve the top five users with the highest PageRank score. The PageRank score is stored under the `followerPageRank` node property.

The results of the Exercise 5.8 are the following:

Table 5.8. Most important nodes judging by the PageRank score.

componentCount	countOfMembers

"elonmusk"	20.381862706745217
"NASA"	8.653231888111382
"wmktech"	6.937989377788902
"Twitter"	6.937989377788902
"Wajdialkayal1"	6.551413750286345

Elon Musk is by far the most influential user in our Twitter subgraph. Interestingly, Andrew Ng, GoogleAI, and OpenAI were all in the first five positions given the incoming degree but have lost their places when using the PageRank score. Remember, the PageRank evaluates the number of incoming connections as well as how influential the nodes behind the links are. Sometimes, a node with a high PageRank score only has a small number of influential connections.

You can examine the top followers of each user in Table 5.8 with the following Cypher statement:

Listing 5.18. Examine the top five followers for the highest ranking users.

```
# A
MATCH (u:User)<-[:FOLLOWS]-(f)
WHERE u.username IN ["elonmusk", "NASA", "wmktech", "Twitter", "W
# B
WITH u,f
ORDER BY f.followerPageRank DESC
# C
RETURN u.username as user, round(u.followerPageRank, 2) as pagera
ORDER BY pagerankScore DESC

# A Match the the particular group of users using the IN clause
# B Order the intermediate results by the followers pageRank scor
# C Collect the top five followers grouped by the original users
```

The Cypher statement in Listing 5.18 begins by matching the top five users by their `username` property. Instead of using multiple `OR` predicates, you can use the `IN` operator to specify a list of possible values. Then you use the `WITH` statement to order the results by the followers' pageRank score. Lastly, you use the `collect()` function to produce an ordered list of followers by their pageRank score. The `collect()` function keeps the order of the input data. Because you first ordered the results in the `WITH` statement by the followers' PageRank score, the list result of the `collect()` function will contain an ordered list of followers by their PageRank score. Returning only the top five followers per user was achieved with *list slicing*. You might have come across a list or array slicing if you have done any programming or SQL analysis before. The list slicing syntax in Cypher is as follows:

```
array[from..to]
```

The square brackets syntax will extract the array elements from the start index "from", and up to (but excluding) the end index "to". Cypher has a `round()` function that allows you to specify to round any number to the specified precision or decimal point.

The results of the Cypher statement in Listing 5.18 are:

Table 5.9. Most important nodes judging by the PageRank score.

user	pagerankScore	topFiveFollowers
"elonmusk"	20.38	["fchollet", "TheCuriousLuke", "DrLiMengYAN1", "douwekiela", "threadreaderapp"]
		["BIBBI02374449",

"NASA"	8.65	"Lucian2drei", "NYTScience", "CmccClimate", "abhibisht89"]
"wmktech"	6.94	["Wajdialkayal1", "alkayal_wajdi", "Websystemer", "AlkayalWajdi", "SwissCognitive"]
"Twitter"	6.66	["Lucian2drei", "Chuck_Moeller", "Omkar_Raii", "SportsCenter", "philipvollet"]
"Wajdialkayal1"	6.55	["wmktech", "Websystemer", "taylorwfarley", "RiM2ww", "saye2018"]

I was hoping that Elon or NASA would appear under top followers, but unfortunately, they don't follow anyone in our subgraph. If, for example, either Elon or NASA followed a user, their PageRank score would be automatically high because they would have one of the most influential nodes following them. A real-life analogy might be the following. Imagine you just moved to Sweden and don't know anyone except the president of the country. Even though you only have one connection, that connection is very influential, which automatically gives you a lot of influence over the network.

The only exciting follower pattern that can be found in Table 5.9 is that users wmktech and Wajdialkayal1 follow each other. They are both influential but also contribute to each other's importance by following one another.

Neo4j GDS library also supports the *Personalized PageRank* variation. In the PageRank definition, a surfer can get bored and randomly jump to other nodes. With the Personalized PageRank algorithm, you can define which nodes should the surfer jump to when he gets bored. It can be said that by defining the `sourceNodes` to which the surfer is biased to jump to, you are effectively inspecting the influence of nodes by looking through a particular node or multiple nodes point of view.

In this example, you will use the `stream` mode of the Personalized PageRank algorithm. The stream mode returns the results of an algorithm as a stream of records. The syntax of the Personalized PageRank algorithm is almost identical to the PageRank algorithm, except that you are also providing the `sourceNodes` parameter.

Listing 5.19. Run the Personalized PageRank algorithm from the point of view of users who registered in 2016.

```
# A
MATCH (u:User)
WHERE u.registeredAt.year = 2016
WITH collect(u) as sourceNodes
# B
CALL gds.pageRank.stream('follower-network', {sourceNodes: source
YIELD nodeId, score
# C
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 5;

# A Match the source nodes to be used in Personalized PageRank al
# B Execute the Personalized PageRank algorithm
# C Use the `gds.util.asNode` function to match the specific node
```

First, you have to use the `MATCH` clause followed by the `collect()` function to produce a list of all users who registered in 2016. You can then input the collected users as the `sourceNode` parameter. By defining the `sourceNode` parameter, you are instructing the procedure to execute the Personalized PageRank algorithm and use the provided nodes as the restart nodes when teleporting. The `stream` mode of the PageRank algorithms outputs two columns, `nodeId` and `score`. The `nodeId` represents the Neo4j *internal node id*, which the database automatically generates for every node in the database. You can use the `gds.util.asNode()` function to map the `nodeId` value to the actual node instance. The `score` column represents the PageRank score for a particular node.

What would happen if you run the Personalized PageRank and used a node with no outgoing connections as the `sourceNodes` parameter.

Listing 5.20. Run the Personalized PageRank algorithm from the point of view of NASA.

```
MATCH (u:User)
WHERE u.username = "NASA"
WITH collect(u) as sourceNodes
CALL gds.pageRank.stream('follower-network', {sourceNodes: source
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 3;
```

Table 5.10. Most important nodes judging by the PageRank score.

user	score
"NASA"	0.15000000000000002
"ServerlessFan"	0
"dr_sr_simpson"	0

As there are no outgoing connections from the nodes specified in the `sourceNodes` parameter, the PageRank algorithm will keep on restarting at the selected source node, which will, in turn, leave all the other nodes with the PageRank score of zero. When running PageRank with default settings, all nodes with a PageRank score of 0.15 don’t have any incoming relationships. They only get their importance through the surfer randomly jumping to them, but they have no votes of significance from other nodes. With the Personalized PageRank variation, you can also specify which nodes to jump to when bored, which in turn means that some nodes will not even get the PageRank score of 0.15 as the surfer doesn’t randomly jump to them.

Exercise 5.9

Execute the Personalized PageRank algorithm and use the `User` nodes who registered in the year 2019 as the `sourceNodes` parameter.

5.1.8 Drop named graph

Congratulations, you have completed your first network analysis.

After you have completed the planned graph algorithms execution sequence, it is recommended to drop the projected graph from memory. You can release the in-memory graph by using the `gds.graph.drop()` procedure.

Listing 5.21. Release the follower-network graph from memory.

```
CALL gds.graph.drop('follower-network')
```

In the next chapter, you will learn how to infer monopartite networks based on indirect relationships. You will run many of the graph algorithms you learned in this chapter to solidify your skills of executing and understanding their results.

5.2 Summary

- Real-world networks follow power-law distribution of node degree
- For a directed network, the node degree can be split into in-degree, the count of incoming connections, and out-degree, which counts outgoing links
- Graph Data Science library uses a projected in-memory graph to execute

graph algorithm on

- Native projection is the more performant variation of projecting in-memory graphs
- Weakly connected component algorithms is used to identify disconnected parts or islands in the network
- Local clustering coefficient examines how tightly-knit the neighbors of a node are
- PageRank treats each relationship as a vote of influence
- PageRank has a damping factor parameter that specifies how often should the random surfer follow an outgoing link as opposed to jumping to a random node
- With the Personalized PageRank variation, you can specify which nodes should the random surfer teleport to, which gives you a view of the network from a specific point of view

5.3 References

[Shirazi et al., 2020] Saeed Shirazi, Amir Albadvi, Elham Akhondzadeh, Farshad Farzadfar, & Babak Teimourpour (2020). A new application of community detection for identifying the real specialty of physicians. International Journal of Medical Informatics, 140, 104161.

[Newman ME, 2001] Newman ME. The structure of scientific collaboration networks. Proc Natl Acad Sci U S A. 2001 Jan 16;98(2):404-9. doi: 10.1073/pnas.021544898. Epub 2001 Jan 9. PMID: 11149952; PMCID: PMC14598.

[S.Brin and L. Page, 1998] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 30(1-7):107–117, 1998.

[Ying Ding et al., 2010] Ying Ding, Erjia Yan, Arthur Frazho, & James Caverlee. (2010). PageRank for ranking authors in co-citation networks.

[Hanm Yo-Sub et al., 2009] Han, Yo-Sub & Kim, Laehyun & Cha, Jeong-Won. (2009). Evaluation of User Reputation on YouTube. 346-353. 10.1007/978-3-642-02774-1_38.

[Gábor Iván and Vince Grolmusz, 2011] Gábor Iván, Vince Grolmusz, When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks, Bioinformatics, Volume 27, Issue 3, 1 February 2011, Pages 405–407, doi.org/10.1093/bioinformatics/btq680

[Kirkley A. et al., 2018] Kirkley, A., Barbosa, H., Barthelemy, M. et al. From the betweenness centrality in street networks to structural invariants in random planar graphs. Nat Commun 9, 2501 (2018). doi.org/10.1038/s41467-018-04978-z

[Erdős Rényi, 1959] P. ERDŐS-A. RÉNYI, On random graphs. I,Publicationes Mathematicae (Debrecen),6 (1959), pp. 290–297.

[Albert, R., Jeong, H. & Barabási, 1999] Albert, R., Jeong, H. & Barabási, AL. Diameter of the World-Wide Web. Nature 401, 130–131 (1999). doi.org/10.1038/43601

[Barabási and Albert, 1999] Barabási, A.L., & Albert, R. (1999). Emergence of Scaling in Random Networks. Science, 286(5439), 509–512.

[Myers et al., 2014] Myers, S., Sharma, A., Gupta, P., & Lin, J. (2014). Information Network or Social Network? The Structure of the Twitter Follow Graph. In Proceedings of the 23rd International Conference on World Wide Web (pp. 493–498). Association for Computing Machinery.

[Ugander et al., 2011] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. arXiv, 2011.

[Swati et al., 2016] Dhingra, Swati et al. “Finding Strongly Connected Components in a Social Network Graph.” International Journal of Computer Applications 136 (2016): 1-5.

[Efstathiades, Hariton et al., 2016] Efstathiades, Hariton et al. “Online social network evolution: Revisiting the Twitter graph.” 2016 IEEE International Conference on Big Data (Big Data) (2016): 626-635.

6 Projecting monopartite networks with Cypher Projection

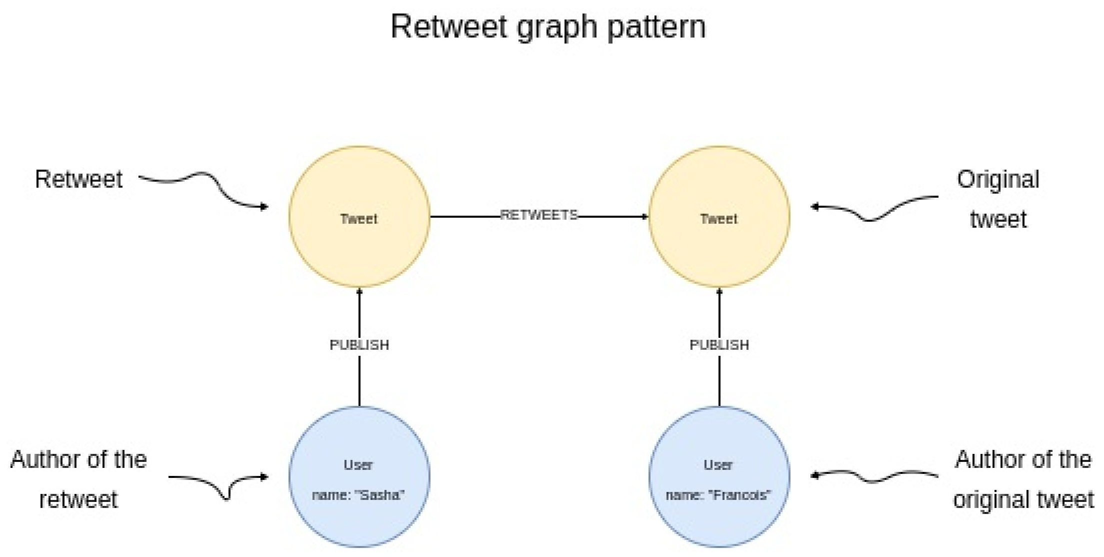
This chapter covers

- Translating an indirect graph pattern into a direct relationship
- Using Cypher projection to project an in-memory graph
- Ignoring self-loops with Cypher projection
- Introducing weighted variant of Degree centrality and PageRank algorithms

In the previous chapter, you performed a network analysis of the Twitter follower network. The decision to start with the follower network was straightforward. Most graph algorithms are designed to be executed on a monopartite network, meaning only a single node and relationship type are present. However, the Twitter social network schema contains multiple node types and relationships. Instead of adjusting graph algorithms to support multipartite networks (multiple node and relationship types), the general approach is to first project a monopartite network (single node and relationship type). I have briefly alluded to this concept in Chapter 2, where I presented some options on how to infer monopartite projections on the Twitter social network.

Suppose you want to analyze the retweet network and find the most influential users based on how often their tweets get retweeted by other users. More often than not, you want to use the PageRank algorithm to identify the most important nodes in the network. You chose the following graph model to represent the retweet pattern in the Twitter social network.

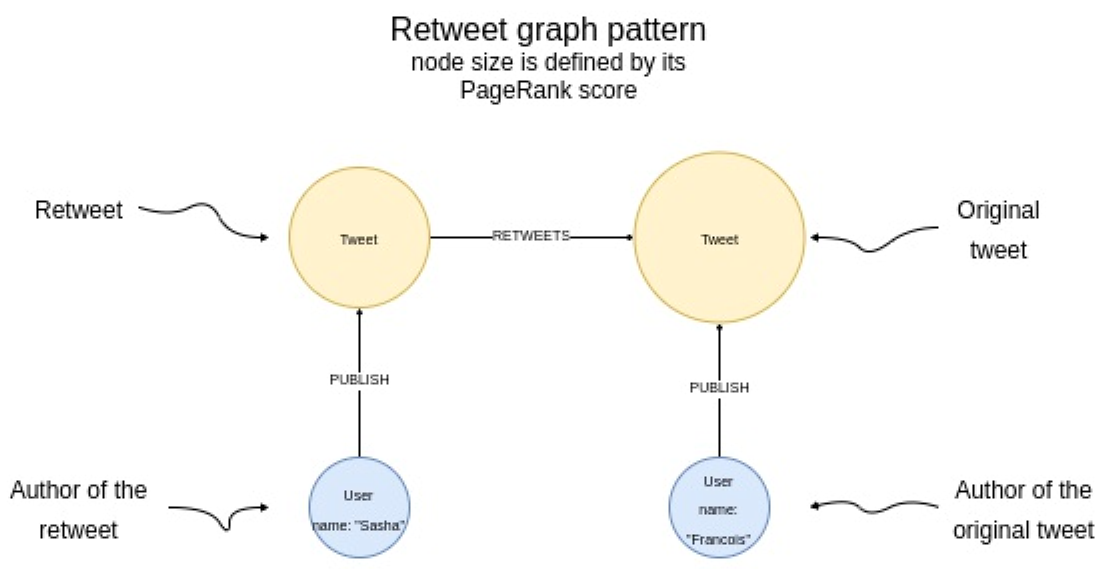
Figure 6.1. Graph model that represents a retweet pattern.



If you were to execute the PageRank algorithm on the network in Figure 6.1, which node do you think would be the most important? Remember that PageRank treats each relationship as a vote of confidence or influence. The influence then flows throughout the network. For me, it is easier to start with the least influential nodes and work my way up to the most important ones. Please take a minute or two to think about it.

Both the original tweet and the retweet have incoming relationships, so they are already more important than users. The retweet has a single incoming relationship, and the original tweet has two incoming connections. With the PageRank algorithm, both the count of incoming links as well as the importance of nodes linking to a particular node is considered when calculating the PageRank score. So, it is not always given that a node with more incoming links will have a higher score. However, in the retweet network in Figure 6.1, the original tweet draws influence from its author as well as the retweet and the retweet’s author. On the other hand, the retweet node draws influence only from the retweet’s author, which means that the retweet will be less important than the original tweet.

Figure 6.2. Retweet pattern where the node size represents its PageRank score.



The objective of the analysis was to determine the most influential users based on their retweet patterns. However, as you can observe, running PageRank on the current retweet network does not help you solve that objective. You might be wondering which graph algorithm to use then. There are so many different graph models in the real world that it doesn't make sense to adjust graph algorithms to all the variations of graph models. So, it is your job to transform the graph to fit the algorithm and not the other way around.

As mentioned, the standard input for the majority of the centrality and community detection algorithms is a monopartite network. Since the objective of the assignment is to analyze the influence of users given the retweet information, the input to graph algorithms should contain `User` nodes. As a monopartite network contains only a single type of nodes, you need to exclude `Tweet` nodes somehow while preserving the information about the retweets. In the retweet example, you could assume that if a user retweets a post from another user, they increase or amplify the reach of the original tweet and, consequently, the author of the original tweet. You can represent how the users amplify other users' reach through a direct relationship between them. How you want to name the new *inferred relationship* depends on your domain and use case. In this example, I will give the new relationship type `AMPLIFY` since it is used to represent how users amplify the reach of one another through retweets. The term inferred relationship means that the relationship is not explicitly defined in data, but is inferred or created based on some assumptions.

Figure 6.3. Translate an indirect retweet pattern into a direct amplify relationship.

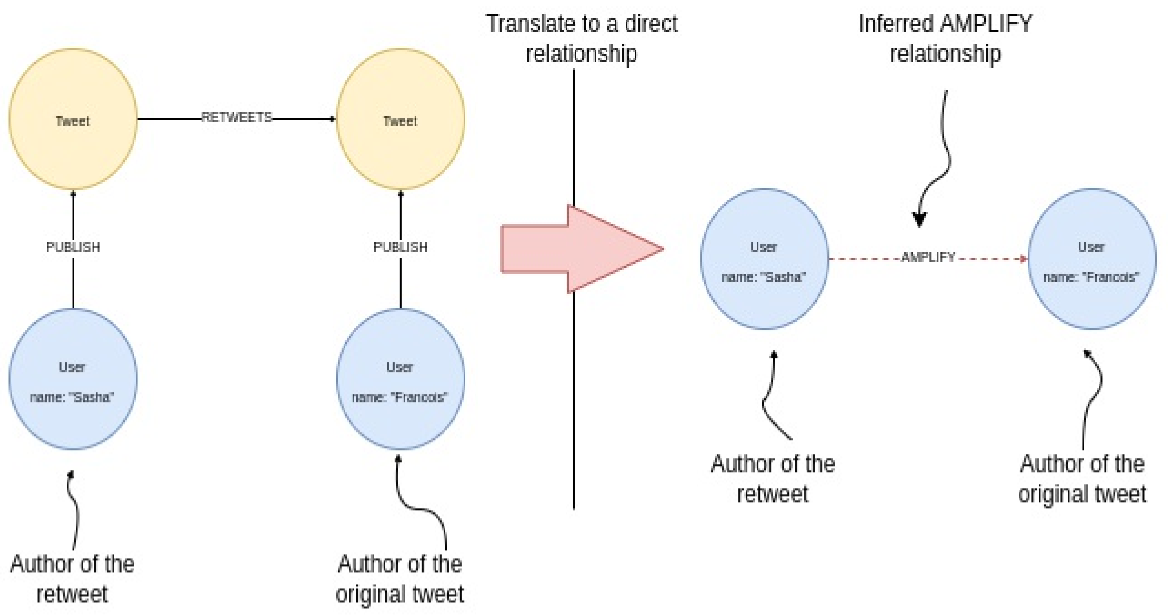


Figure 6.3 visualizes the concept of translating a graph pattern between two `User` nodes that spans over three relationships into a direct link between the two. On the left side of the visualization is the retweet pattern as it is stored in the database. However, since you want to evaluate how influential users are based on the retweet patterns with the PageRank algorithm, you need to transform the indirect path between the two users into a direct relationship, as indicated on the right side of Figure 6.3.

You could easily describe this translation with a Cypher statement.

Listing 6.1. Describe the translation of the indirect retweet pattern to a direct amplify relationship.

```
MATCH (s:User)-[:PUBLISH]->()-[:RETWEETS]->()<-[:PUBLISH]-(t:User
CREATE (s)-[:AMPLIFY]->(t);
```

After you have transformed all the retweet patterns into a direct relationship, you end up with a monopartite network that contains only `User` nodes and `AMPLIFY` relationships.

Figure 6.4. Projected monopartite network of users and amplify relationships.

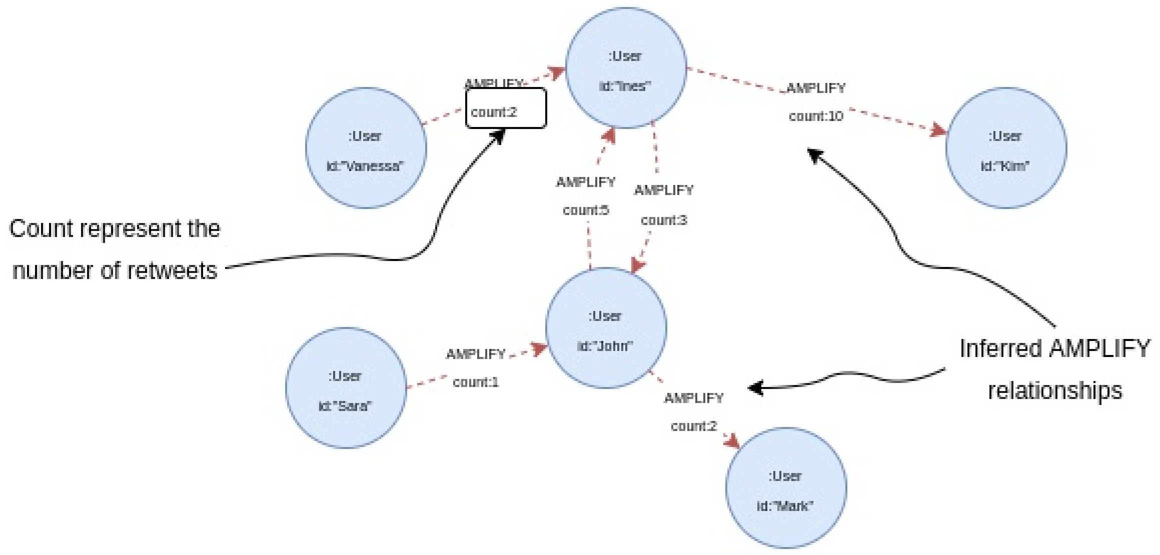


Figure 6.4 visualizes a subgraph of the projected or inferred monopartite network that represents `User` nodes and `AMPLIFY` relationships, constructed based on the retweet pattern. Since a user can retweet posts from other users multiple times, you can store the count as the relationship property. Consequently, you can describe the network in Figure 6.4 as directed and weighted.

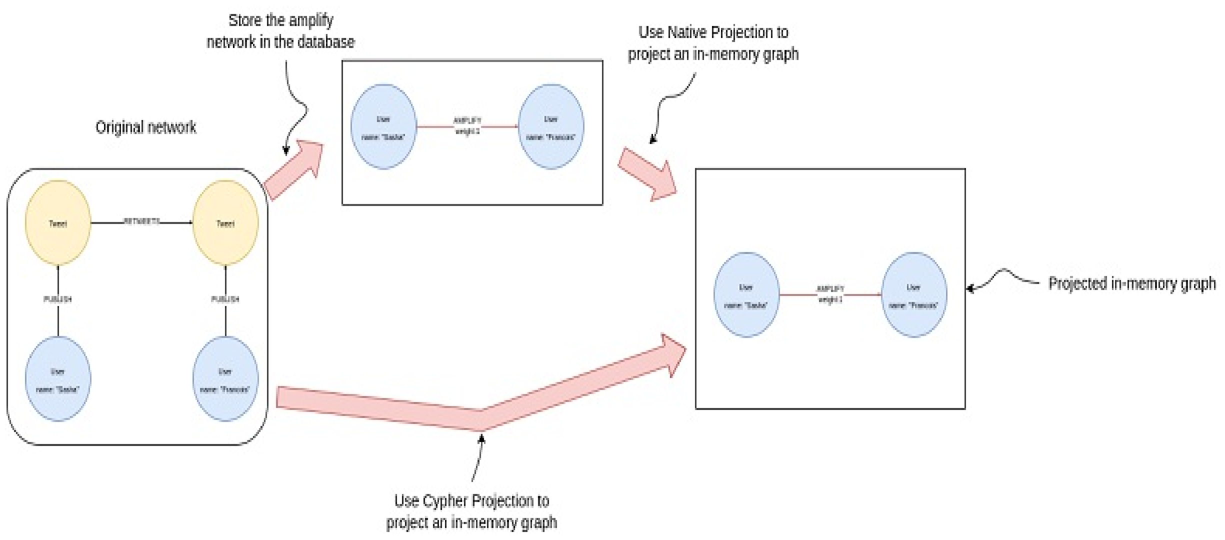
The inferred amplify network could then be used to examine the users who produce the best (most sharable) content under the assumption that the retweet means that a user liked the content of the original tweet. I would imagine that if you want to express your disagreement with the tweet’s content, you would quote the tweet and describe your dispute with the tweet. You defined that the quote and retweet should be stored under a different relationship type in the original graph schema. However, since there are no quotes in our dataset, you could assume that all the retweets are positive, meaning that users agree with the original tweet’s content. Many studies have been published using the retweet network ranging from a network analysis of the European Parliament [Cherepnalkoski & Mozetic, 2015] to science and health-related clusters retweet clusters on Twitter during the COVID-19 pandemic [Durazzi et al., 2021].

You will learn more details and caveats of various approaches to inferring monopartite networks through practical examples. To follow the exercises in this chapter, you need to have the Twitter network imported into the Neo4j database as described in Chapter 3.

6.1 Translate an indirect multi-hop path into a direct relationship

You will begin by translating the multi-hop retweet relationship into a direct `AMPLIFY` relationship. With Neo4j GDS, you could take two different approaches to accomplish this task.

Figure 6.5. Two options to translate an indirect multi-hop path to a direct relationship in a projected graph.


Store the amplify network in the database
Use Native Projection to project an in-memory graph
Original network
Tweet
RETWEETS
Tweet
PUBLISH
PUBLISH
User name: "Sasha"
User name: "Francois"
AMPLIFY weight:1
Projected in-memory graph
Use Cypher Projection to project an in-memory graph

While Native projection works nicely with co-occurrence networks, which you will learn more about in the next chapter, it is limited because it doesn’t support custom graph transformations. If you want to project a custom transformation of the original network with Native projection, you first have to materialize it in your Neo4j database. On the other hand, you can use Cypher projection to load a *virtual graph* into memory. In this context, a virtual graph is a graph that is not stored in the database and is constructed only at projection time. While Cypher Projection has worse performance than Native Projection and is not recommended for the production phase, it is a great tool to transform smaller graphs without having to materialize them in the database first.

6.1.1 Cypher Projection

Cypher Projection is a more flexible and expressive approach to projecting an in-memory graph. As you might deduce from the feature's name, you can use Cypher statements to define the nodes and relationships you want to load in the in-memory graph. Cypher Projection procedure is called `gds.graph.project.cypher` and has three mandatory parameters.

Listing 6.2. Cypher projection syntax.

```
CALL gds.graph.project.cypher(
    #1
    'graphName'
    #2
    'MATCH (n)
     RETURN id(n) AS id, labels(n) AS labels, n.property AS prope
    #3
    'MATCH (s)-[r]->(t)
     RETURN id(s) AS source, id(t) AS target,
            type(r) AS type, r.weight AS weight'
)
```

You can think of the Cypher Projection as projecting the node and edge list described using Cypher statements. The first parameter is used to define the name of the projected in-memory graph and the second parameter is the *Node Cypher statement*. With the Node Cypher statement, you describe the nodes you want to project and optionally their labels or properties. The Node Cypher statement has two reserved columns. The first and the only mandatory column is the `id` column. The `id` column expects *internal ids* of the nodes you want to project. Every time you create a node in Neo4j, an internal id is automatically assigned to it. You can access the internal node id by using the `id()` function. The second optional reserved column is the `labels` column. With the `labels` column, you can describe the node label, which can be later used to filter nodes at algorithm run-time. Lastly, you can provide any node property in the Node Cypher statement you want to include in the projection.

The third mandatory parameter of Cypher Projection is the so-called *Relationship Cypher statement*. You use the Relationship Cypher statement to describe the relationships you want to project. As mentioned, you can project existing relationships in the database, or you can load virtual connections that are not materialized in the database. The Relationship Cypher statement has two mandatory columns and one reserved optional column. To describe a relationship, you use the internal id of the start and the end node. The reserved column for describing the start node is the `source` column, while the reserved column for the end node is the `target` column. The reserved optional column is the `type` column that defines the relationship type. Describing the relationship type allows you additional relationship filtering at algorithm execution time. Similarly, as with the Node Cypher statement, you can describe any relationship properties you want to include in the projection.

Note

Cypher Projection is a more flexible and expressive approach to describing the graph you project. Essentially, it is a way of defining the projected graph using the node and relationship lists. As the node and relationship lists are defined using Cypher statements, you can take full advantage of the expressiveness of the Cypher Query Language to filter or transform the desired graph projection without having to materialize it first in the database. However, there is a downside to this approach as the performance of the Cypher Projection is worse than the Native Projection. Due to worse performance, Cypher Projection is not recommended for larger graphs or the production phase.

Now you will use Cypher Projection to load the transformed retweet amplify network as an in-memory graph. In the Node Cypher statement, you will include users who retweeted other users or published a tweet that was retweeted.

Exercise 6.1

Construct a Cypher statement that will return only users who retweeted other users or were themselves retweeted. Use the `MATCH` clause combined with the `WHERE` clause to apply the above filter. Return only the first five internal node ids in the `RETURN` statement. The internal node id can be accessed with the `id()` function.

The solution of Exercise 6.1 produces the following output.

Table 6.1. Internal ids of users who retweeted or were themselves retweeted.

id
0
1
2
4
6

The Cypher statement in Listing 6.3 produces the node list, which consists of internal node ids that will be projected in the in-memory graph. Since there is only a single node type present, defining the node labels does not bring any additional benefits. You also don’t need any node properties to examine the retweet amplify network, so you haven’t included any of them. Don’t worry if your output is slightly different as the internal node ids are not explicitly defined and could vary from import to import.

You have to prepare the Relationship Cypher statement before using Cypher Projection to load the in-memory graph and execute graph algorithms.

Exercise 6.2

Describe the retweet pattern in the `MATCH` statement. Return the internal node id of the user that retweeted another user under the `source` column and the user who was retweeted under the `target` column. Since a user can retweet another user multiple times, count the number of retweets and return the number of retweets under the `weight` column. Return only the first five rows of the result.

The solution of Exercise 6.2 produces the following output.

Table 6.2. Internal ids of users who retweeted or were themselves retweeted.

source	target	weight
1	4	7
185	4	19
142	4	7
211	4	1
952	4	1

The Cypher statement in Listing 6.4 produces an edge list, where the relationships are represented by its `source` and `target` node id. You have calculated an additional `weight` property that depicts the number of retweets between the pair of users. The retweet amplification network can be described as a directed, weighted network.

You can now combine the Node and Relationship Cypher statements and use them as an input to the Cypher Projection procedure.

Listing 6.3. Load the amplify retweet network as an in-memory graph using the Cypher Projection.

```
CALL gds.graph.project.cypher(
 #1
 'amplify',
 #2
 'MATCH (u:User) WHERE EXISTS { (u)-[:PUBLISH]->()-[:RETWEETS]-()
  RETURN id(u) AS id',
 #3
 'MATCH (s:User)-[:PUBLISH]->()-[:RETWEETS]->()<-[:PUBLISH]-(t:Us
  RETURN id(s) AS source, id(t) AS target, count(*) AS weight')
 YIELD graphName, nodeCount, relationshipCount
```

The Cypher Projection procedure will return the following output.

Table 6.3. Cypher Projection procedure output.

graphName	nodeCount	relationshipCount
"amplify"	1828	2719

There are 1828 nodes and 2719 relationships in the projected in-memory `amplify` graph. Now, you will perform a short network analysis of the retweet amplification network to reinforce your experience with executing graph algorithms.

6.1.2 Degree centrality

First, you will evaluate the node degree distribution of the inferred network.

In the previous chapter, you used a plain Cypher to calculate and visualize the node degree distribution. Here, the retweet amplification network is not materialized in the database, so you don’t have the option of using a plain Cypher statement to calculate the node degree distribution. Instead, you can use the GDS degree centrality algorithm `gds.degree` to evaluate the node degree distribution

You can use the `stats` mode of the algorithm to examine the node degree distribution. By default, the `gds.degree` centrality calculates the out-degree. Remember, the out-degree is the count of outgoing relationships a node has, while the in-degree counts the incoming links.

Listing 6.4. Evaluate the out-degree distribution of the inferred retweet amplification network.

```
CALL gds.degree.stats('amplify')
YIELD centralityDistribution
```

Table 6.4. Out-degree distribution of the retweet amplification network

p99	18.00011444091797
min	0.0
max	146.00096893310547
mean	1.4874205599728507
p90	2.0000076293945312
p50	1.0

p999	48.00023651123047
p95	4.000022888183594
p75	1.0

On average, a node in the network has around 1.5 outgoing relationships. The `pX` values represent the percentile values. For example, the `p75` represents the 75th percentile value of 1.0, which means that 75% of nodes have one or zero outgoing relationships. You can deduce that the inferred retweet network is a sparse.

Exercise 6.3

Use the degree centrality algorithm to calculate and return the top five nodes with the highest out-degree in the retweet amplification network. Use the `stream` mode of the algorithm to stream the results without storing them in the database or the projected graph. The `stream` mode of the `gds.degree` algorithm outputs `nodeId` and `score` columns. Use the `gds.util.asNode` to map the node ids to node instances and retrieve the `username` property for the top five nodes.

The solution of Exercise 6.3 produces the following output.

Table 6.5. Top five users by out-degree in the retweet amplification network.

user	score
"textsla"	146.0
"godfrey_G_"	61.0

"iPythonistaBot"	48.0
"Beka "Bexx" Modebade"	36.0
"chidambara09"	33.0

The user `textsla` has retweeted posts from 146 different users. It wouldn’t surprise me if most of the users on this list had automatic retweets in place for specific hashtags. Since you are dealing with a weighted network, you can also evaluate the weighted out-degree distribution. Most of the GDS library graph algorithms support the algorithms' weighted variations by using the `relationshipWeightProperty` configuration parameter.

Evaluate the weighted out-degree distribution of the retweet amplification network with the following Cypher statement.

Listing 6.5. Evaluate the weighted out-degree distribution of the inferred retweet amplification network.

```
CALL gds.degree.stats('amplify', {relationshipWeightProperty:'wei
YIELD centralityDistribution
```

Table 6.6. Weighted out-degree distribution of the retweet amplification network

p99	65.00048065185547
min	0.0
max	2006.0078048706055

mean	4.715000173456038
p90	3.0000076293945312
p50	1.0
p999	310.00194549560547
p95	7.000022888183594
p75	1.0

Although the average weighted out-degree is 4.7, the 75th percentile is only 1 and the 90th percentile only raises to 3. It seems that there are a few outliers that raise the average of the whole population. For example, a single user has 2006 retweets. I would venture a guess that most of the highest retweeting users are actually bots.

Exercise 6.4

Use the degree centrality algorithm to calculate and return the top five nodes with the highest weighted out-degree in the retweet amplification network. The solution is almost identical to the Exercise 5.3, except that you include the `relationshipWeightProperty` parameter to calculate the weighted out-degree.

The outgoing node degree can help you evaluate and identify users spreading or distributing content the most through the network. On the other hand, you can use the in-degree distribution to identify users who produce the most sharable (best?) content. The node degree centrality algorithm has a `orientation` parameter that allows you to evaluate in-degree, out-degree, or combination of both. The `orientation` parameter has three possible inputs:

- NATURAL - Evaluate out-degree (count of outgoing relationships)
- REVERSE - Evaluate in-degree (count of incoming relationships)
- UNDIRECTED - Evaluate the sum of both in- and out-degrees

Therefore, you can use the `orientation` parameter to evaluate the in-degree distribution by setting it to `REVERSE`.

Listing 6.6. Return the top five users with the highest in-degree.

```
CALL gds.degree.stats('amplify', {orientation:'REVERSE'})
YIELD centralityDistribution
```

Table 6.7. In-degree distribution of the retweet amplification network

p99	29.00011444091797
min	0.0
max	117.00048065185547
mean	1.4874205599728507
p90	3.0000076293945312
p50	0.0
p999	90.00048065185547

p95	7.000022888183594
p75	1.0

More than 50% of the users haven’t been retweeted even once. In some way, it makes sense that there are fewer users who write content that is retweeted than users who do the retweeting. If you remember from the previous chapter as well, the in and out-degree mean will always be identical, as the number of connections and users stays the same, only the relationship direction is reversed. Interestingly, in this example, also the 75th and the 90th percentile values are identical. It seems that the in-degree distribution is a bit more top-heavy than the out-degree. This would imply that several content creators consistently produce content that is being retweeted. Maybe that implies producing quality content, but we would have to investigate further. Perhaps only their hashtag game is strong.

Exercise 6.5

Use the degree centrality algorithm to calculate and return the top five nodes with the highest in-degree in the retweet amplification network. Use the `stream` mode of the algorithm to stream the results without storing them in the database or the projected graph.

The solution of Exercise 6.5 produces the following output.

Table 6.8. Top five users by in-degree in the retweet amplification network.

user	score
"Paula_Piccard"	117.0
"IainLJBrown"	96.0

"Eli_Krumova"	90.0
"Analytics_699"	69.0
"annargrs"	65.0

A total of 117 different users have retweeted Paula_Piccard. The difference between the first and fifth places is not as big as the out-degree distribution. One might assume that they produce relevant and quality content as they are often retweeted. You would have to scrape more tweets with the relevant hashtags for more accurate results.

Exercise 6.6

Evaluate the weighted in-degree distribution with the `stats` mode of the degree centrality algorithm. If you need some help, you can look at the examples for the weighted out-degree and include the `orientation` parameter. After that, use the `stream` mode of the algorithm to identify the top five users with the highest weighted in-degree.

You will now execute some of the graph algorithms you learned in Chapter 4 to consolidate your knowledge of using them.

6.1.3 Weakly Connected Components

The Weakly Connected Components algorithm should be part of almost every network analysis. With it, you can evaluate how connected the network is and identify disconnected components.

Execute the `stats` mode of the WCC algorithm to get a rough feeling of how disconnected the network is.

Listing 6.7. Evaluate the number and the size of disconnected components.

```
CALL gds.wcc.stats('amplify')
YIELD componentCount, componentDistribution
```

Table 6.9. Summary statistics for the WCC algorithm executed on the retweet amplification network.

componentCount	componentDistribution
207	{ "p99": 28, "min": 1, "max": 1082, "mean": 8.830917874396135, "p90": 6, "p50": 2, "p999": 1082, "p95": 13, "p75": 3 }

The largest component consists of 1082 members, which is around 60% of the total users in the retweet amplification network. As mentioned before, most real-world networks will contain a single super component containing most of the network’s nodes and then a couple of smaller components on the side. What is weird to me is that minimum size components contain only a single member. With the Cypher Projection, you have filtered users who retweeted or were retweeted. My first thought was that there should be no components with only a single member. As this is an unexpected result, it is worth exploring.

Execute the following Cypher statement to examine sample components with a single member.

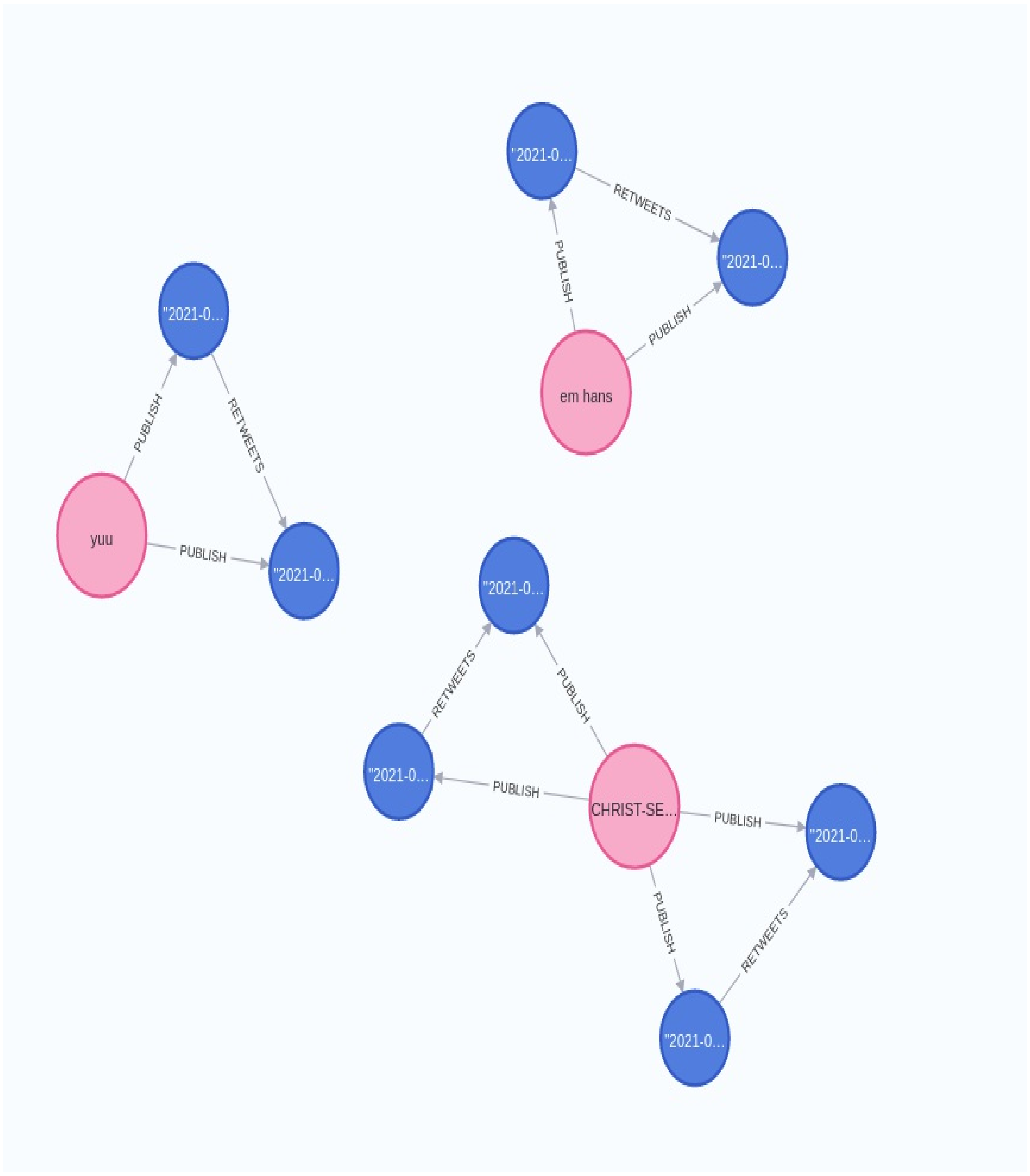
Listing 6.8. Examine sample components with a single member.

```
#1
CALL gds.wcc.stream('amplify')
YIELD nodeId, componentId
#2
WITH componentId, collect(nodeId) AS componentMembers,
     count(*) AS componentSize
#3
WHERE componentSize = 1
#4
WITH componentMembers[0] AS id
LIMIT 3
#5
MATCH p=(n)-[:PUBLISH]->()-[:RETWEETS]-()
WHERE id(n)=id
RETURN p
```

The Cypher statement in Listing 6.13 starts by executing the `stream` mode of the WCC algorithm on the retweet amplification network. The `stream` mode of the WCC algorithm outputs the `nodeId` column, representing the internal node id of the node, and the `componentId` column, which describes to which component the node belongs. In the B step of the Cypher statement, you aggregate by the `componentId` to calculate the component size and collect its members' node ids. Afterward, you use the `WHERE` clause to filter components with only a single member. As there should be only a single element in the `componentMembers` list of the single-member components, you can easily extract the only node id using the square bracket syntax in combination with its index position. To not overwhelm the results visualization, you will only examine three components with a single member. The last thing you need to do is match the retweets pattern for the three specific node ids.

The Cypher statement in Listing 6.13 will produce the following visualization in Neo4j Browser.

Figure 6.6. Components that contain a single member, where users retweeted themselves.



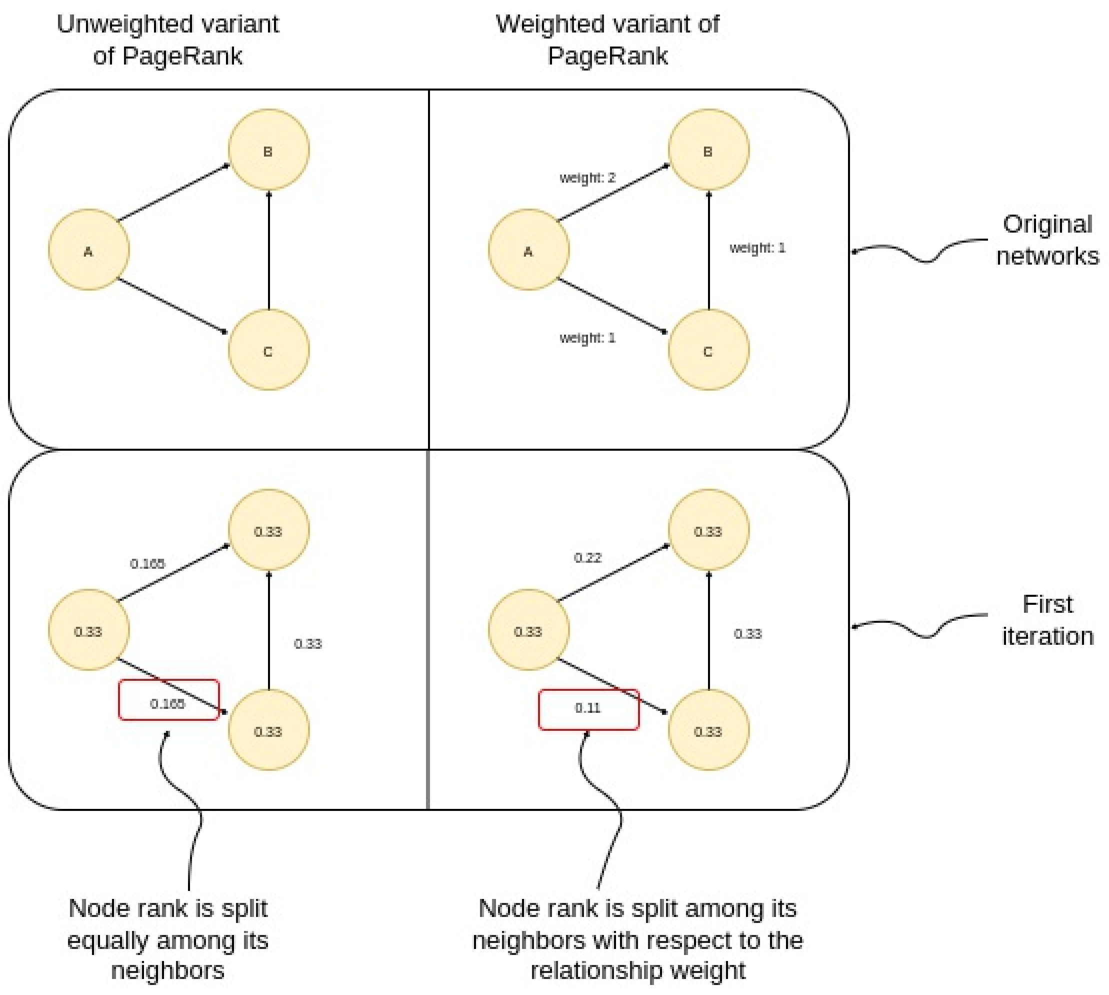
On the Twitter social network, a user can also retweet their posts. In graph theory, a *self-loop* is a relationship with the same start and end node. As mentioned, the WCC algorithm is helpful to help you identify how connected the network is as well as identify various unexpected patterns.

6.1.4 Weighted PageRank

In the last part of the retweet amplification network analysis, you will execute the weighted PageRank algorithm to identify potential influencers. Remember, the PageRank algorithm considers both the number of incoming connections and the nodes' importance that links to it. Instead of simply analyzing which user has the most retweets, you are also evaluating which other influential nodes in the network retweeted them.

The weighted variant of the PageRank algorithm also considers the relationship weight when calculating node importance. With the unweighted variant of PageRank, the node’s importance is equally spread amongst its neighbors. On the other hand, with the weighted variant, each neighbor gets the share of importance related to the relationship weight.

Figure 6.7. Difference between weighted and unweighted PageRank calculation in a single iteration.


Unweighted variant of PageRank
Weighted variant of PageRank
B
A
C
weight: 2
weight: 1
weight: 1
Original networks
0.33
0.165
0.33
0.165
0.22
0.11
First iteration
Node rank is split equally among its neighbors
Node rank is split among its neighbors with respect to the relationship weight

As mentioned, the weighted variant of the PageRank algorithm considers the relationship weight when it calculates how the influence spreads across the network. Figure 6.7 visualizes a simple network consisting of three nodes. The difference between the weighted and unweighted variants is demonstrated as to how node A spreads its influence. With the unweighted variant, nodes B and C get an equal share of importance from node A. In the weighted network, the relationship from node A to C has a weight of 1, and the connection from node A to B has a value of 2. In every iteration of the weighted PageRank algorithm, node B will receive two-thirds of influence from node A and node C will receive only one-third. The equation to calculate the share of influence with the weighted PageRank algorithm is simply dividing the relationship weight by the sum of all outgoing relationship weights.

I think it makes sense to exclude all the self-loops from the network before you execute the weighted PageRank algorithm. The self-loop can be translated as the node stating that it is influential. I think that retweeting your own tweets shouldn’t increase your influence in the network. Unfortunately, there is no magic button you could press to exclude self-loops, so you have to project another in-memory graph using Cypher Projection.

Exercise 6.7

Use the Cypher Projection to load the retweet amplification network into memory and exclude all self-loops. Essentially, you only need to change the Relationship Cypher statement to filter out relationships that start and end at the same node. Name the new projected graph as `amplify-noselfloops`.

Now you can go ahead and execute the weighted PageRank algorithm on the retweet amplification network without self-loops. Similarly, as with the degree centrality, you only need to include the `relationshipWeightProperty` parameter to execute the weighted variant of the algorithm.

Listing 6.9. Execute the PageRank algorithm on the retweet amplification network with no self-loops.

```
CALL gds.pageRank.stream('amplify-noselfloops',
  {relationshipWeightProperty:'weight'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 5
```

Table 6.10. Top five users by weighted PageRank score in the retweet amplification network with no self-loops.

user	score
"Paula_Piccard"	8.270755243786214

"annargrs"	7.836125000000006
"psb_dc"	7.478576023152348
"IainLJBrown"	7.457764370226901
"Eli_Krumova"	6.95963977383344

The top five users by weighted PageRank list is similar to the list of top five users by in-degree. Of course, the Twitter subgraph you are analyzing is relatively tiny. While you haven't analyzed the tweet topics or hashtags, I've mentioned that I scraped the dataset by focusing on the NLP and Knowledge graph topics. So, the following users could be good candidates to follow on Twitter if you are interested in NLP or Knowledge graphs topic updates.

6.1.5 Drop projected in-memory graph

It is important to remember to release the projected in-memory graph once you are done with the analysis to free up memory for other analysis. At the moment, you should have two graph loaded in-memory. The following Cypher statement will drop all currently projected graphs.

Listing 6.10. Release all the projected graphs from memory.

```
#1
CALL gds.graph.list() YIELD graphName
#2
CALL gds.graph.drop(graphName) YIELD nodeCount
RETURN 'dropped ' + graphName
```

6.2 Summary

- Inferring monopartite networks is a frequent step in graph analysis
- Cypher Projection can be used to project a virtual graph (non-existing

relationships in the database)
- GDS library uses two directed relationship that point in the opposite direction to represent an undirected relationship
- You can change the relationship direction or treat it as undirected during in-memory graph projection
- Parameters `nodeLabels` and `relationshipTypes` can be used to consider only a subset of the projected graph as an input to a graph algorithm
- You can use weighted variants of the Degree centrality and Pagerank algorithms by definining the `relationshipWeightProperty` parameter

6.3 References

[Cherepnalkoski & Mozetic, 2015] D. Cherepnalkoski and I. Mozetic, "A Retweet Network Analysis of the European Parliament," 2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), 2015, pp. 350-357, doi: 10.1109/SITIS.2015.8.

[Durazzi et al., 2021] Durazzi, F., Müller, M., Salathé, M. et al. Clusters of science and health related Twitter users become more isolated during the COVID-19 pandemic. Sci Rep 11, 19655 (2021). https://doi.org/10.1038/s41598-021-99301-0

6.4 Solutions to exercises

The solution to Exercise 6.1 is the following:

Listing 6.11. Return the first five internal ids of nodes that retweeted other users or were themselves retweeted.

```
MATCH (u:User)
WHERE EXISTS { (u)-[:PUBLISH]->()-[:RETWEETS]-() }
RETURN id(u) AS id
LIMIT 5
```

The solution to Exercise 6.2 is the following:

Listing 6.12. Prepare the Relationship Cypher statement.

```
MATCH (s:User)-[:PUBLISH]->()-[:RETWEETS]->()<-[:PUBLISH]-(t:User
RETURN id(s) AS source, id(t) AS target, count(*) AS weight
LIMIT 5
```

The solution to Exercise 6.3 is the following:

Listing 6.13. Return the top five users with the highest out-degree.

```
CALL gds.degree.stream('amplify')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 5
```

The solution to Exercise 6.4 is the following:

Listing 6.14. Return the top five users with the highest weighted out-degree.

```
CALL gds.degree.stream('amplify',
  {relationshipWeightProperty:'weight'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 5
```

The solution to Exercise 6.5 is the following:

Listing 6.15. Return the top five users with the highest weighted in-degree.

```
CALL gds.degree.stream('amplify', {orientation:'REVERSE'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username as user, score
ORDER BY score DESC
LIMIT 5
```

The solution to Exercise 6.6 is the following:

Listing 6.16. Return the top five users with the highest weighted in-degree.

```
CALL gds.degree.stream('amplify', {orientation:'REVERSE',
  relationshipWeightProperty:'weight'})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username as user, score
```

```
ORDER BY score DESC
LIMIT 5
```

The solution to Exercise 6.7 is the following:

Listing 6.17. Load the amplify retweet network as an in-memory graph using the Cypher Projection and exclude self-loops.

```
CALL gds.graph.project.cypher(
 'amplify-noselfloops',
 'MATCH (u:User) WHERE (u)-[:PUBLISH]->()-[:RETWEETS]-()
  RETURN id(u) AS id',
 'MATCH (s:User)-[:PUBLISH]->()-[:RETWEETS]->()<-[:PUBLISH]-(t:Us
  WHERE NOT s = t
  RETURN id(s) AS source, id(t) AS target, count(*) AS weight')
 YIELD graphName, nodeCount, relationshipCount
```

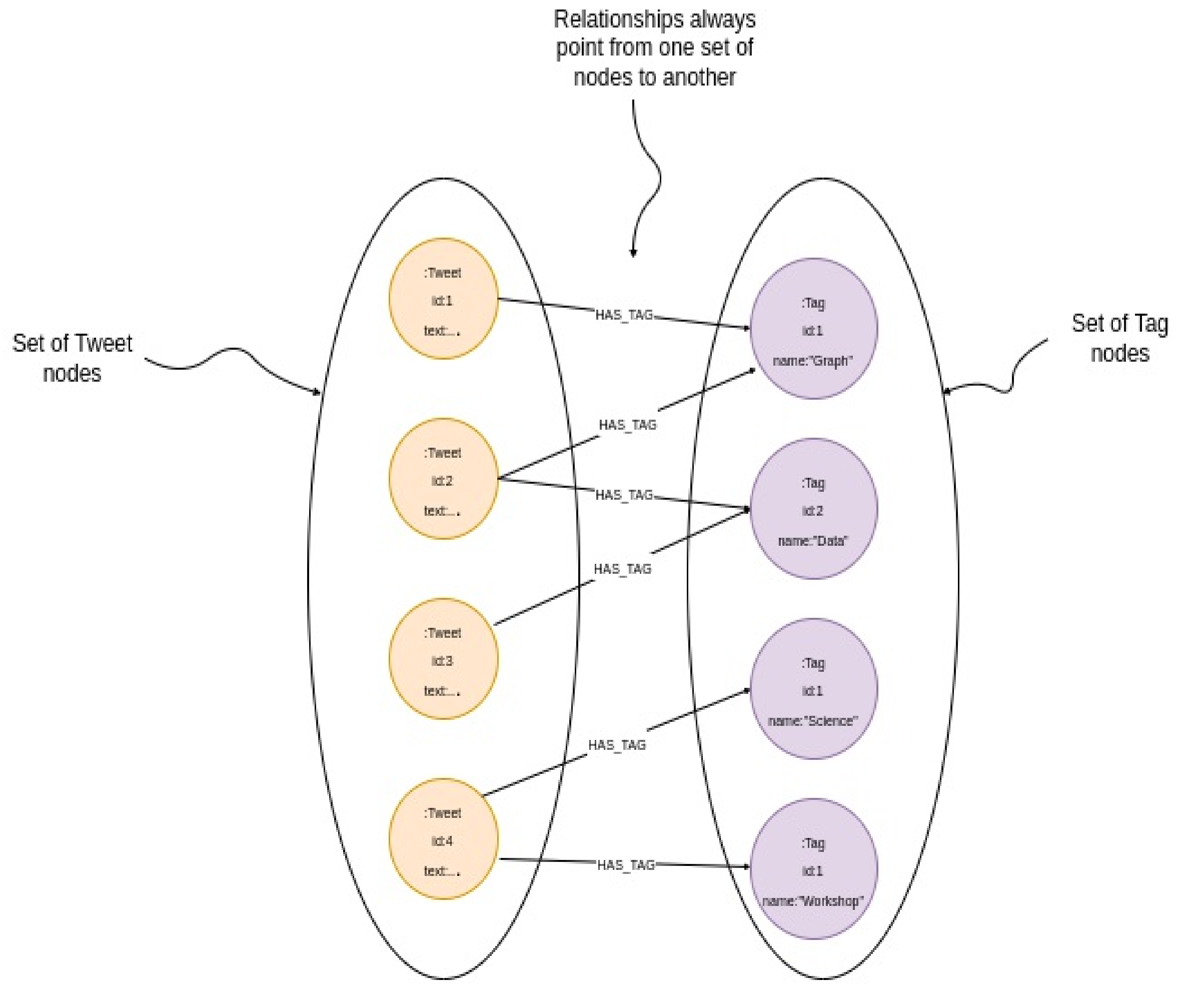
7 Inferring co-occurrence networks based off bipartite networks

This chapter covers

- Extracting hashtags from tweets with Cypher query language
- Calculating Jaccard similarity coefficient
- Constructing and analyzing monopartite networks using Jaccard similarity coefficient
- Using Label Propagation algorithm to evaluate community structure of a network
- Using PageRank to find the most important node within a community

In the previous chapter, you learned how to transform a custom graph pattern into direct relationships to use them as an input to graph algorithms like PageRank. In this chapter, you will focus on bipartite networks and how to project them into monopartite networks. First, a quick refresher of what a bipartite network is.

Figure 7.1. Bipartite network of tweets and hashtags.



A bipartite network contains two sets or types of nodes. For example, Figure 7.1 visualizes the bipartite network of tweets on the left and their hashtags on the right. As you can observe, the relationships always points from one type of nodes to another. There are no direct connections between tweets or hashtags.

Imagine you work in a marketing analytics role for a company that deals with natural language processing and knowledge graphs. Your boss decided it might be worthwhile to start advertising on Twitter. You have been assigned the task of identifying relevant hashtags to target the company’s ideal customer as best as possible. Since the Twitter dataset you have used so far contains tweets about NLP and knowledge graphs, you can analyze the hashtags present in the dataset to identify which of them the company should target.

Your first thought might be to use the PageRank algorithm to identify the most important hashtags. Remember, graph algorithms like PageRank expect monopartite networks as input or at least networks where the influence can flow throughout the connections. With a bipartite network, one type of nodes has only incoming relationships, and the other type has only outgoing connections. If you were to execute PageRank on the example bipartite network of tweets and hashtags, which nodes do you think would come out on top?

Since tweets don’t have any incoming relationships, their PageRank score will equal the chance of the surfer randomly teleporting to them. With the default value of the damping factor of 0.85, the PageRank score for nodes with no incoming connections is 0.15. On the other hand, hashtags have only incoming relationships. The influence flows from tweets to hashtags but does not flow further as there are no outgoing connections from hashtags. In practice, the PageRank rank of hashtags would be equal to their count of incoming relationships (in-degree). However, the actual values of PageRank and in-degree would be different due to distinct score calculation techniques.

Remember, the goal is to determine the most important and relevant hashtags in the dataset. If the definition of the hashtag importance is as simple as their frequency, then using the in-degree metric of the `Tag` nodes would suffice.

However, some of the hashtags in the dataset might not be relevant to the marketing targeting objective. You could completely miss relevant hashtags by looking at only the most frequently mentioned hashtags, as you don’t know if the dataset is skewed towards a particular topic or not. In that case, you would first have to identify hashtags that form various topics. A frequent technique used to identify communities of hashtags that form various topics is to start by examining how often pairs of hashtags co-occur in tweets. By examining how often pairs of hashtags co-occur in the same tweet, you build a *co-occurrence network*. The term co-occurrence network refers to a network construction method that analyzes relationships between various entities in a text. In the case of tweets and their hashtags, you can use the co-occurrence network method to analyze relationships between hashtags that appear in the text of a tweet.

Co-occurrence networks are constructed by connecting pairs of entities in the text using a set of criteria defining co-occurrence. The co-occurrence definition can vary from scenario to scenario. Sometimes the definition of co-occurrence is defined as two entities appearing in the same sentence. However, you could also define co-occurrence as two entities appearing in the same article or even when the pair of entities appear within a specific distance in a text.

Figure 7.2. Co-occurence network of medical keywords.

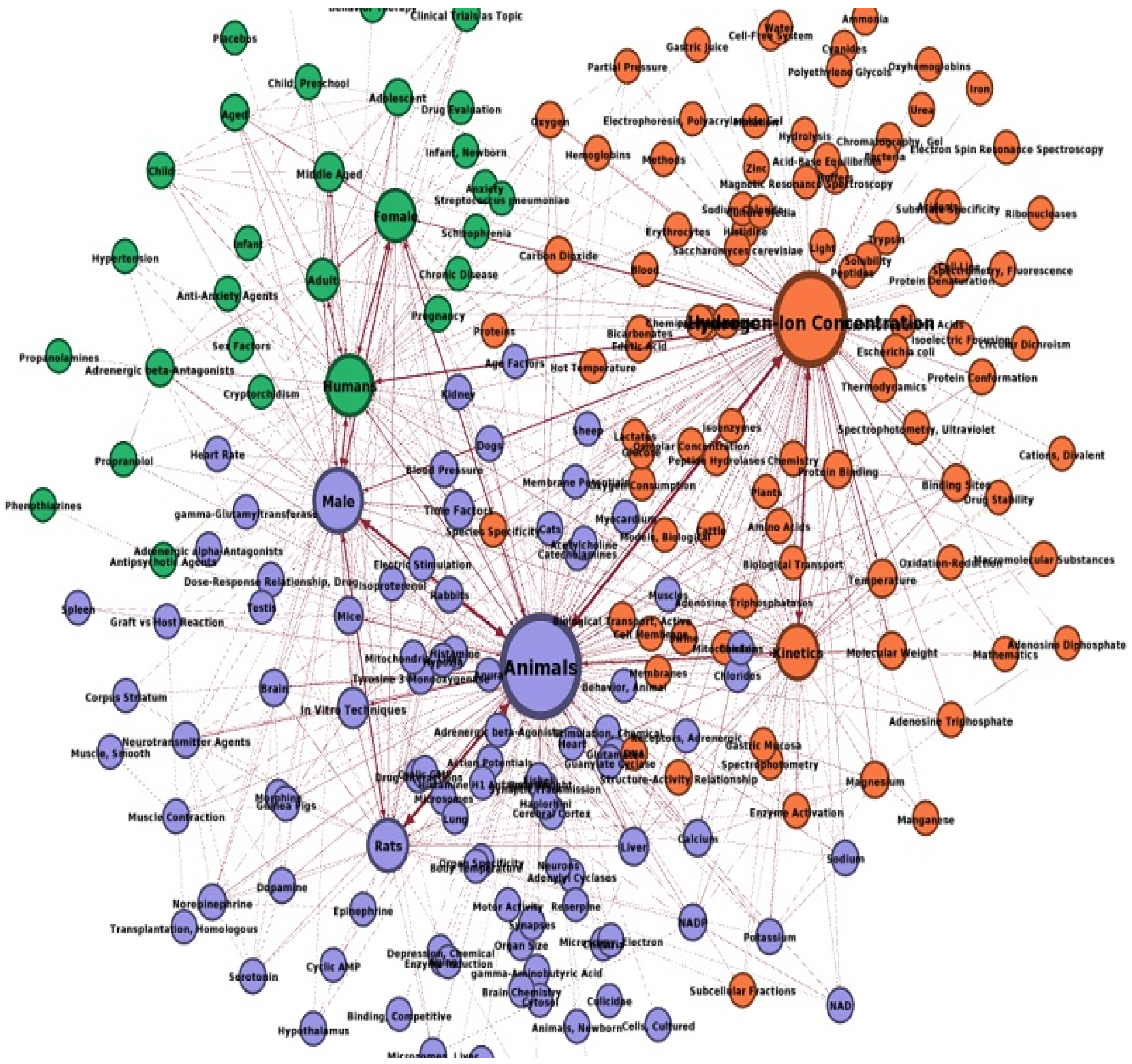
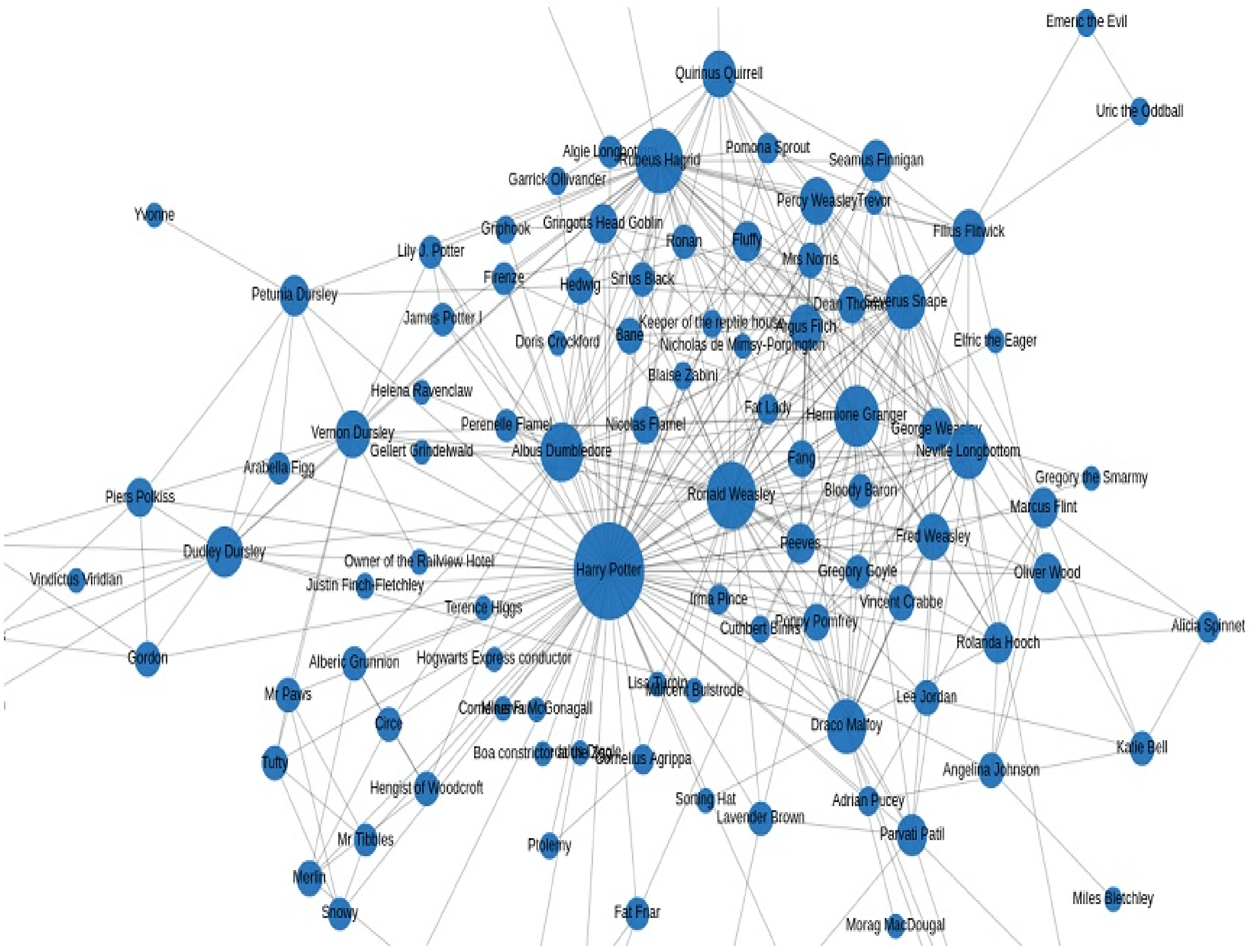


Figure 7.2 visualizes the co-occurrence network of keywords in medical articles. In this example, co-occurrence is defined as two keywords occurring

in the same article. The more times a pair of keywords are present in the same article, the stronger the connection between the two. A similar technique was used to analyze the scientific literature surrounding the Covid-19 research. [Al-Zaman, 2021] [Andersen et al., 2020] Not only that, but researchers have also used the biomedical literature co-occurrence network to predict new links. [Kastrin et al., 2014]

However, you are not limited to analyzing only the co-occurrence of keywords in a given text. In the Game of Thrones analysis [Beveridge et al., 2018], Andrew Beveridge popularized analyzing books through the lens of co-occurrence entity networks. The Game of Thrones book analysis consists of two steps. First, he identified all of the characters in the book. In the next step, he defined a co-occurrence event between a pair of characters if they appear within 15 words of one another. In this context, a co-occurrence event can be understood as an interaction between a pair of characters. In my previous works, I have used this technique and constructed a co-occurrence or interaction network between characters in the Harry Potter and the Philosopher’s Stone book.

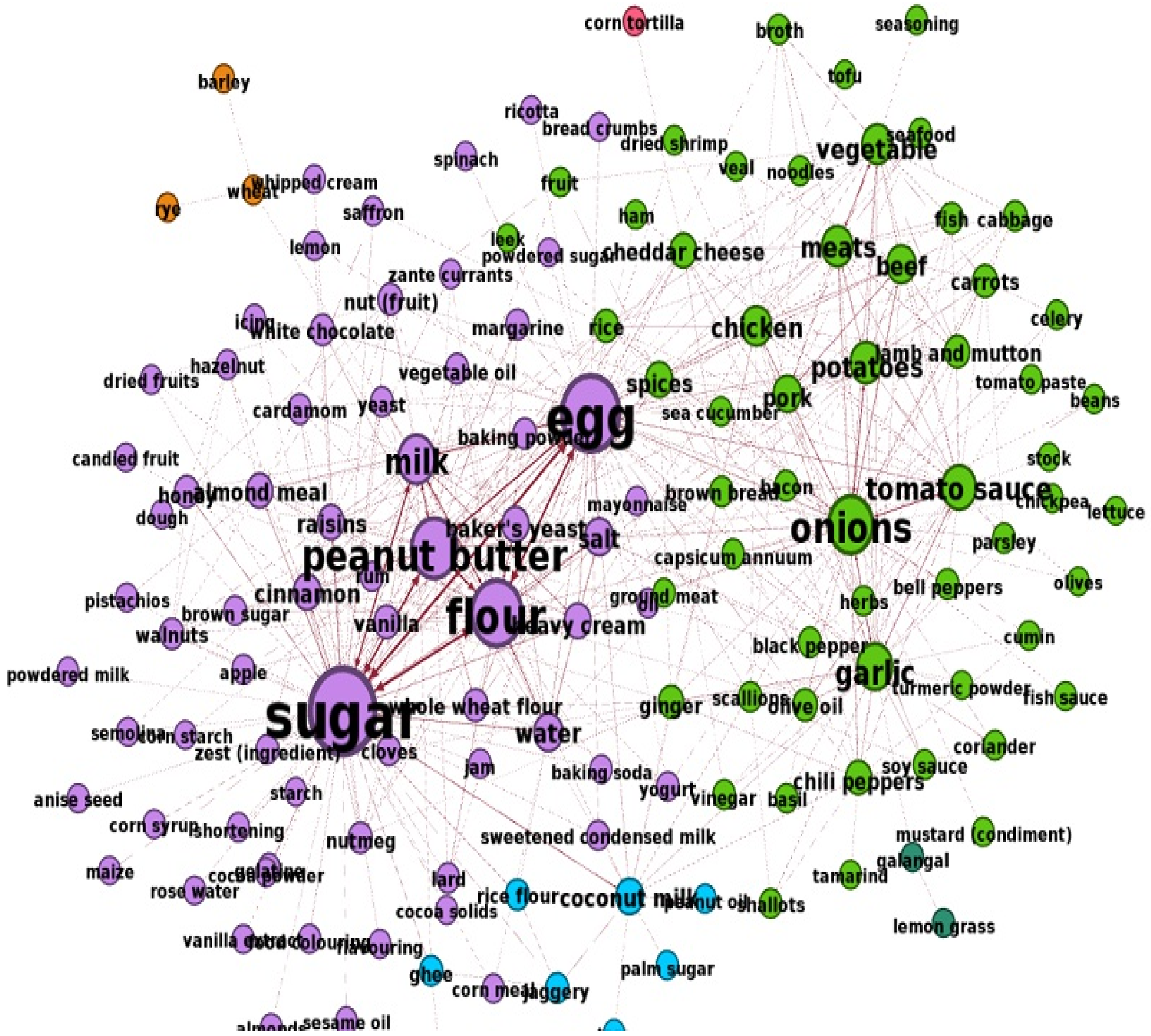
Figure 7.3. Co-occurence network of characters in the first Harry Potter book.



Just by looking at the Figure 7.3, you can evaluate which are the main characters in the first Harry Potter book and evaluate how they interact with each other. The node size is based on the node degree, so the more connections a node has, the bigger the node size. It only makes sense that Harry Potter has the most interactions with other characters, as the book narrative is mostly written from his perspective.

The first two examples of the co-occurrence analysis demonstrate scenarios where keywords or entities are extracted from a text and then connected using an arbitrary co-occurrence event. The third example I've prepared is the visualization of how the ingredients co-occur in various dishes or recipes.

Figure 7.4. Co-occurence network of ingredients in dishes.



Again, the node size in Figure 7.4 is based on the number of connections it has. In the center of the left-hand side community, you can observe eggs, flours, sugar, and milk. I would imagine that the dishes with those ingredients mostly fall under bread, pancakes, or maybe sweets. For some reason, peanut butter is also frequently present in this community. You can observe more of a main-dish type of ingredients on the right-hand side like onions, tomato sauce, potatoes, and meat. Interestingly, I’ve found a research paper[Cooper, 2020] that uses the ingredient co-occurrence network to analyze packaged food in the United States. Another research paper[Kular et al., 2011] uses the ingredient co-occurrence network to examine the relationships between cuisines and various cultures.

Let’s now circle back to the task of identifying the most relevant hashtags to the marketing objective of targeting hashtags relevant to knowledge graphs and natural language processing. You can start with a hypothesis that if two hashtags co-occur within the same tweet, they are somehow related or fall under a similar topic. Based on that hypothesis, the co-occurrence network method would connect hashtags that might have the same or similar overall topic.

Figure 7.5. Constructing a co-occurrence network of tags based on if they co-occur in the same tweet.

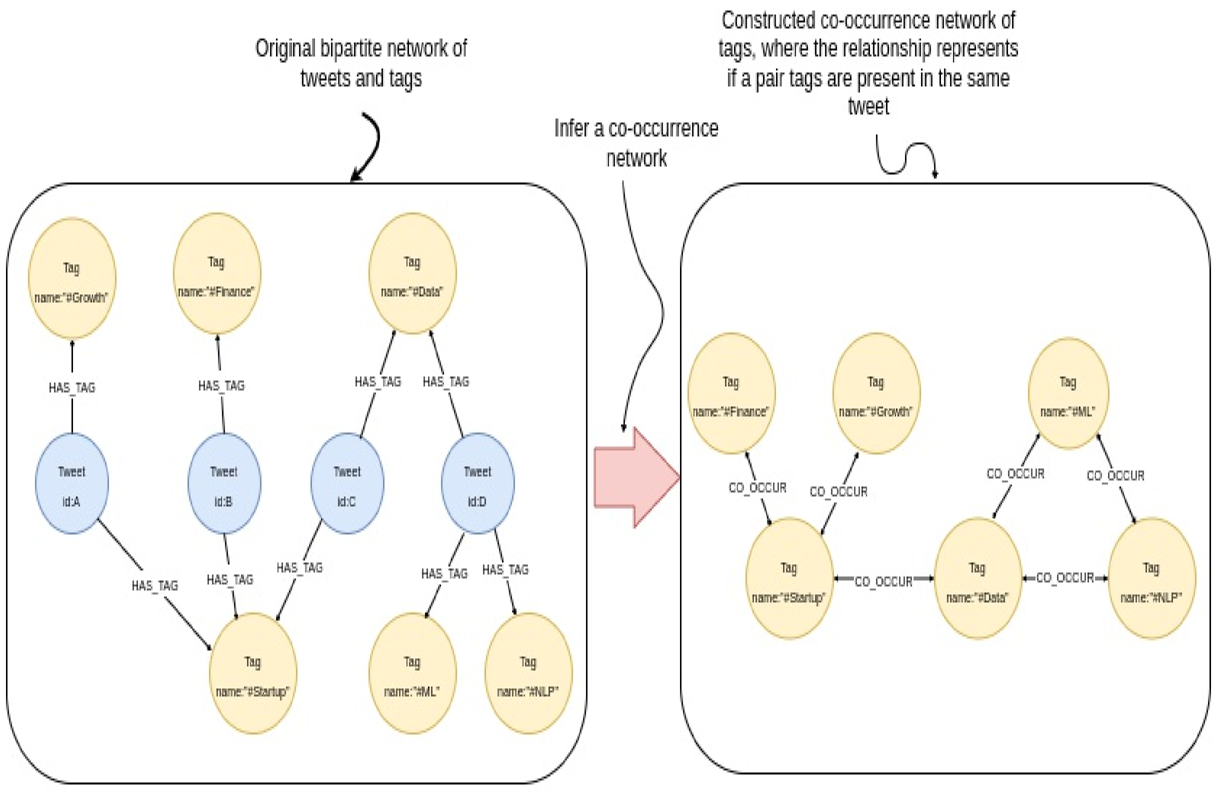

Original bipartite network of tweets and tags
Infer a co-occurrence network
Constructed co-occurrence network of tags, where the relationship represents if a pair tags are present in the same tweet
Tag name:"#Growth"
Tag name:"#Finance"
Tag name:"#Data"
Tweet id:A
Tweet id:B
Tweet id:C
Tweet id:D
Tag name:"#Startup"
Tag name:"#ML"
Tag name:"#NLP"
HAS_TAG
CO_OCCUR

Figure 7.5 visualizes the process of constructing a co-occurrence network of hashtags based on the original bipartite network of tweets and hashtags. On the left-hand side of the visualization is the original bipartite network. A tweet can contain multiple hashtags. For example, tweet A has #Growth and #Startup hashtags. In this example, the co-occurrence is defined as a pair of hashtags co-occurring in the same tweet. Hence, on the right-hand side of the visualization, where the co-occurrence network is visualized, there is a relationship between the #Finance and the #Startup hashtags as they appear in the same tweet. If you look at tweet D, you can notice that it has three hashtags #Data, #ML, and #NLP. Since the hashtags co-occur in the same tweet, there is a relationship between all three hashtags, as indicated on the right-hand side of Figure 7.5.

This data transformation can be described with the following Cypher statement.

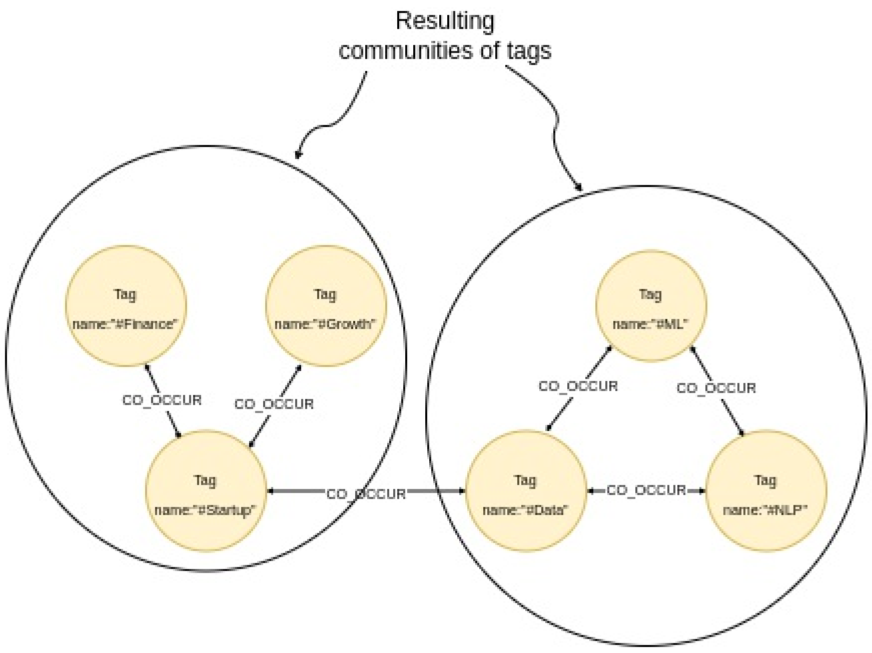
Listing 7.1. Describe the construction of a hashtag co-occurrence network with a Cypher statement.

```
MATCH (s:Tag)<-[:HAS_TAG]-(:Tweet)-[:HAS_TAG]->(t:Tag)
CREATE (s)-[:CO_OCCURRENCE]->(t);
```

Hashtags are used to index or define the topic of a tweet. By analyzing the co-occurrence network of hashtags, you learn which hashtags on Twitter often overlap, and potentially fall under the same overall topic. Constructing a Twitter hashtag co-occurrence network was used in multiple studies. For example, you could analyze how do hashtags help drive virality [Pervin, 2015] [Wang et al., 2016]. Another paper studies how the hashtag co-occurrence connections differ from their semantic similarity[Türker & Sulak, 2018]. Lastly, I’ve found an exciting article[Vitale, 2018] that uses the hashtag co-occurrence network to provide novel information and trends regarding smoking habits.

With the requirement to find the most relevant hashtags within knowledge graph and natural language topics, you first have to identify which hashtags form a higher-level topic. Here, you circle back to the hypothesis made before. If two hashtags frequently co-occur in the same tweet, you can assume they are related and fall under the same topic. To find communities or clusters of hashtags that form a topic, you can utilize community detection algorithms like the *Label Propagation algorithm* (LPA). Label Propagation (LPA) is an algorithm to evaluate the community structure of a network. Most of the literature on the internet introduces the LPA as a semi-supervised algorithm, where you can input initial communities for some nodes in the network. However, here you will use the unsupervised variant of the LPA as you won’t present any initial communities. The unsupervised variant of the LPA works as follows. First, it assigns a unique community label to each node. Then it iterates over the network and updates each node label to the one most of its neighbors have. The idea behind this iteration is that a single community label can quickly become dominant in a densely connected group of nodes. Once the LPA reaches convergence, the algorithm stops, and the resulting node labels represent their communities.

Figure 7.6. Identifying communities within the hashtag co-occurrence network.



Communities represent densely connected groups of nodes with sparser links between groups. For example, Figure 7.6 visualizes two groups or communities of hashtags. The left community contains #Finance, #Growth, and #Startup hashtags. You could assign the left community as a more business-oriented topic. On the other side, the right community in Figure 7.6 consists of #ML, #Data, and #NLP hashtags. Again, you could try to deduct the overall topic of the right community. In this example, something like a computer science or data science oriented topic would fit.

After you have identified communities of hashtags that form topics, you can use the PageRank algorithm to find the most central nodes within communities. Remember, we assume that the co-occurrence between a pair of hashtags implies that they are somewhat related or similar. If you were to execute the PageRank algorithm on each community separately, you would identify the most central nodes of communities. You can assume that the most central nodes in the community are its representatives, as the PageRank algorithm treats every relationship as a vote. In the example of hashtag co-occurrence network, it is a vote of similarity or relatedness. So, the most similar hashtags to all the other hashtags in the community will rank the highest, which you can interpret as community representatives.

To follow the exercises in this chapter, you need to have the Twitter network imported into the Neo4j database as described in Chapter 3.

7.1 Extracting hashtags from tweets

Before you can infer the hashtag co-occurrence network, you first have to extract hashtags from the tweet's content. You will only extract hashtags from tweets that are not retweets, as the retweets have the same hashtags as the original tweets and would only skew the results. The process of extracting hashtags is elementary text processing. You split the tweet text by the whitespace or newline characters to create a list of words. Then you need to filter out words that start with a hashtag # sign. Once you have completed these two steps, you have successfully extracted hashtags from a tweet and can store them in the database.

Listing 7.2. Extract hashtags from tweets that are not retweets.

```
#1
MATCH (t:Tweet)
WHERE NOT EXISTS { (t)-[:RETWEETS]->() }
#2
WITH t, replace(t.text, '\n', ' ') AS cleanText
#3
WITH t, split(cleanText, ' ') AS tokens
#4
WITH t, [el IN tokens WHERE el STARTS WITH "#" |
            toLower(replace(el,",",""))] AS hashtags
#5
WHERE size(hashtags) > 0
RETURN hashtags LIMIT 5
```

Table 7.1. Extracted hashtags.

hashtags
["#mindset", "#nlp", "#meditation", "#heartmath", "#bioresonance", "#mindcoaching", "#anxiety", "#hypnosis", "#mentalhealth"]
["#sql", "#code", "#datascience", "#ai", "#ml", "#machinelearning", "#iot", "#iiot", "#iotpl", "#python", "#rstats", "#cloud", "#cybersecurity"]
["#coaching", "#nlp", "#progressnotperfection", "#growthmindset", "#trainyourbrain", "#parentyourselffirst"]
["#acl2021nlp"]
["#medium", "#machinelearning", "#nlp", "#deeplearning", "#autocorrect", "#spellcheck", "#ai", "#tds", "#python"]

Interestingly, the NLP hashtag means natural language processing in the tech community, while the personal development community uses NLP as an acronym for neuro-linguistic programming. While I was aiming to scrape only tweets around natural language processing and knowledge graph topics, it seems there are also some self-development tweets in the dataset. Having more diverse topics in our dataset is not a problem. It makes your analysis more interesting, as you will learn the driving hashtags behind more computer science-oriented topics as well as self-help topics.

Most marketing platforms allow you to specify particular keywords or hashtags and exclude the undesired ones. Since your objective as a marketing analytics person is to design a targeting strategy on Twitter, you can also prepare a list of hashtags to be excluded. The company deals with natural language processing, so you would want to target the #nlp hashtag. You have learned that the #nlp hashtag is also popular in self-help topics, and since self-help topics are not relevant to the company, it makes sense to exclude the hashtags that fall under the self-help topics.

The Cypher statement in Listing 7.2 starts by matching all the tweets that don’t have an outgoing `RETWEETS` relationship. To create a list of words for each tweet, you use the combination of `replace` and `split` functions. First, you use the `replace` function to replace all new line characters with the whitespace characters. The syntax for the `replace` function is:

Listing 7.3. Replace function syntax.

```
replace(string, search, replace)
```

Replace is a basic function in most scripting languages and query language, so I hope it doesn’t need any additional explanation. Similarly, the `split` is also a very basic function and has the following syntax:

Listing 7.4. Split function syntax.

```
split(string, delimiter)
```

The input to the `split` function is a string and a delimiter character, while the output is a list of elements. Again, a basic function that is available in most if not all programming languages. The last thing to do is to filter words that start with a hashtag `#` character. You can filter out hashtags from the list of words by using the *list comprehension* syntax. At first sight, it seems that the list comprehension function took some inspiration from the Python syntax.

Listing 7.5. List comprehension syntax.

```
[element in list WHERE predicate | element]
```

The list comprehension syntax is wrapped by square brackets. The `element IN list` syntax is used to define a variable to reference an element in the list. Unlike Python, element manipulation and transformation can be defined right after the pipe `|` character instead of directly in the variable assignment. You have removed commas and lowered the text in the element transformation part of the list comprehension syntax to not differentiate between #NLP and #nlp hashtags. You can also filter items in the list by using the `WHERE` clause.

Lastly, you use the `size()` function to filter tweets with at least a single hashtag. The `size()` function returns the number of items in a list. In the previous chapter, you learned to use the `size()` function to access the node degree in an optimized way, but it can also be used to count the length of a list.

Before continuing with the co-occurrence analysis, you will extract hashtags and store them in the database. Every time you add a new node label in the database, it is advisable to identify the unique property of the nodes and define a unique constraint. With the hashtags, each node should represent a single hashtag, so you can simply define a unique constraint on the `id` property of `Tag` nodes.

Listing 7.6. Define unique constraint for `Tag` nodes on the `id` property.

```
CREATE CONSTRAINT IF NOT EXISTS ON (t:Tag) ASSERT t.id IS UNIQUE;
```

Finally, you can execute the following Cypher statement to extract and store hashtags in the database.

Listing 7.7. Extract hashtags and store them to the database.

```
MATCH (t:Tweet)
WHERE NOT EXISTS { (t)-[:RETWEETS]->() }
WITH t, replace(t.text, '\n', ' ') AS cleanText
WITH t, split(cleanText, ' ') AS tokens
WITH t, [el IN tokens WHERE el STARTS WITH "#" |
            toLower(replace(el, ",", " "))] AS hashtags
WHERE size(hashtags) > 0
#1
UNWIND hashtags AS tag_id
MERGE (tag:Tag {id: tag_id})
MERGE (t)-[:HAS_TAG]->(tag)
```

The Cypher statement in Listing 7.7 introduces the `UNWIND` clause. The `UNWIND` clause is used to transform a list of elements into rows, similar to a `FOR` loop in various scripting languages. Essentially, you iterate over each element in the list and, in this case, merge a `Tag` node and connect it to the `Tweet` node. The `UNWIND` clause is always followed by the `AS` operator to assign a reference variable to the element value in the produced rows.

The following Cypher statement demonstrates a simple usage of the `UNWIND` clause.

Listing 7.8. UNWIND clause syntax.

```
UNWIND [1, 2, 3] AS i
RETURN i
```

Table 7.2. UNWIND clause transforms a list of elements into rows.

i
1
2
3

Exercise 7.1

Hashtags are now stored and connected to the `Tweet` nodes in the database. Before jumping to the co-occurrence analysis, investigate which hashtags appear in most tweets and retweets. Remember, you only stored hashtags for original tweets (not retweets) in the database. Therefore, first, match the original tweets in which the hashtags appeared. Next, count how many times those tweets were retweeted and return the top five hashtags by the sum of combined counts of original tweets and retweets. Since not all tweets are retweeted, use the `OPTIONAL MATCH` to count the number of retweets.

Table 7.3. Most popular hashtags in tweets and retweets.

hashtag	originalTweetsCount	retweetCount
#nlp	1848	7532
#ai	1554	7169
#machinelearning	1474	7007
#datascience	1455	6736
#bigdata	1358	6577

The most popular hashtags are #nlp, #ai, #machinelearning, and #datascience. Judging by the retweet count, they must frequently co-occur in the same tweets as there are only 12 thousand tweets and retweets in total.

Now, you will proceed with the co-occurrence part of the hashtag analysis.

7.2 Analyzing the co-occurrence network

You can use Cypher query language to evaluate which hashtags most frequently co-occur.

Exercise 7.2

Evaluate which hashtags most frequently co-occur. Use the `MATCH` clause to define a graph pattern where two hashtags are present in the same tweet and then use the `count()` function to count the number of tweets in which a pair of hashtags co-occur. Return only the top five most co-occurring pairs of hashtags.

The solution to Exercise 7.2 is the following:

Listing 7.9. Examine the top five most co-occuring pairs of hashtags.

```
MATCH (h1:Tag)<-[:HAS_TAG]-()-[:HAS_TAG]->(h2:Tag)
#1
WHERE id(h1) < id(h2)
WITH h1,h2,count(*) AS cooccurrences
ORDER BY cooccurrences DESC LIMIT 5
RETURN h1.id AS tag1, h2.id AS tag2, cooccurrences
```

Table 7.4. Top five most co-occuring pairs of hashtags.

tag1	tag2	cooccurrences
#ai	#nlp	1507
#machinelearning	#nlp	1428
#datascience	#nlp	1410
#ai	#machinelearning	1410
#datascience	#ai	1405

Exercise 7.2 did not mention that you should ideally remove duplicates from the output as you haven't learned how to do it yet. Since every hashtag will appear as the `h1` variable as well as the `h2` variable, the results will contain duplicates. Using the strategy of deduplicating results with the `id(h1) <`

`id(h2)` is the most frequent I’ve seen in practice.

You could use a similar Cypher statement to project the co-occurrence network with the Cypher Projection. The resulting co-occurrence network based on the Cypher statement in Listing 7.9 would look the following:

Figure 7.7. Sample weighted and undirected co-occurrence network.

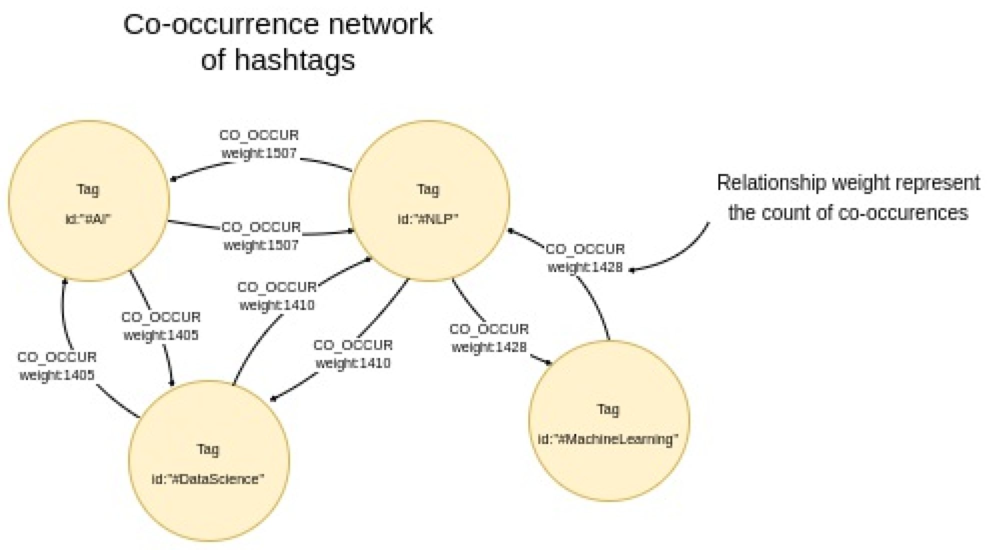
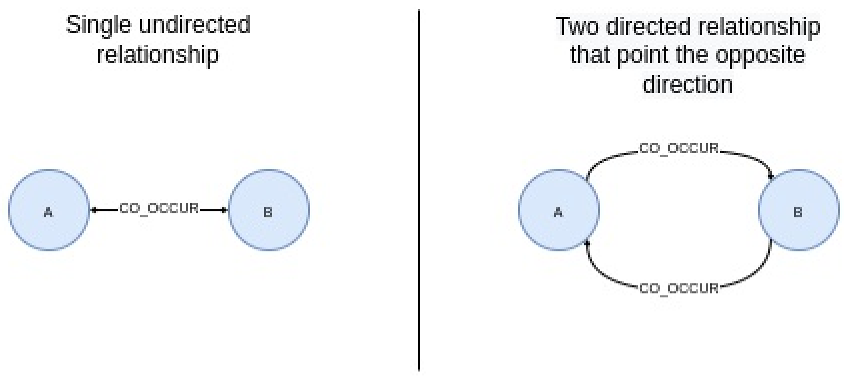


Figure 7.7 visualizes a sample hashtag co-occurrence network, where the relationship weight represents the count of co-occurrences. You might wonder why there are two relationships between each pair of nodes in the opposite direction. If hashtag #NLP co-occurs with hashtag #AI, that directly implies that hashtag #AI also co-occurs with hashtag #NLP. In the context of graphs, you could say that the `CO_OCCUR` relationship is undirected as the direction of the connection is not essential. However, the GDS library has no concept of undirected relationships. A key concept behind an undirected relationship is that it allows traversals in both directions. You can replicate this functionality in a directed network by transforming a single undirected connection into two directed links that point in the opposite direction.

Figure 7.8. A single undirected relationship can be represented as two directed relationships that point in the opposite direction.


Single undirected relationship
A
CO_OCCUR
B
Two directed relationship that point the opposite direction
CO_OCCUR
A
B
CO_OCCUR

Note

Graph Data Science library has no notion of undirected relationships. When dealing with an undirected network in GDS, you represent each relationship in the network as two directed relationships that point in the opposite direction. In the Node Similarity algorithm example, the algorithm’s output is an undirected network, where each undirected relationship is represented as two directed relationships, as shown in Figure 7.8. The GDS library also allows transforming a single relationship into two relationships that point in the opposite direction during projection time.

While there is nothing wrong with using the count of co-occurrences as the relationship weight, a more frequent approach is to use the *Jaccard similarity coefficient* to evaluate the similarity between nodes. The Jaccard similarity coefficient is simple to understand as it only involves dividing the intersection by the union of two sets.

Figure 7.9. An example of two baskets with overlapping products.

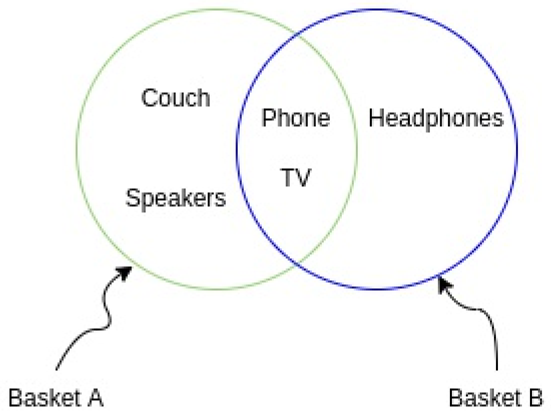
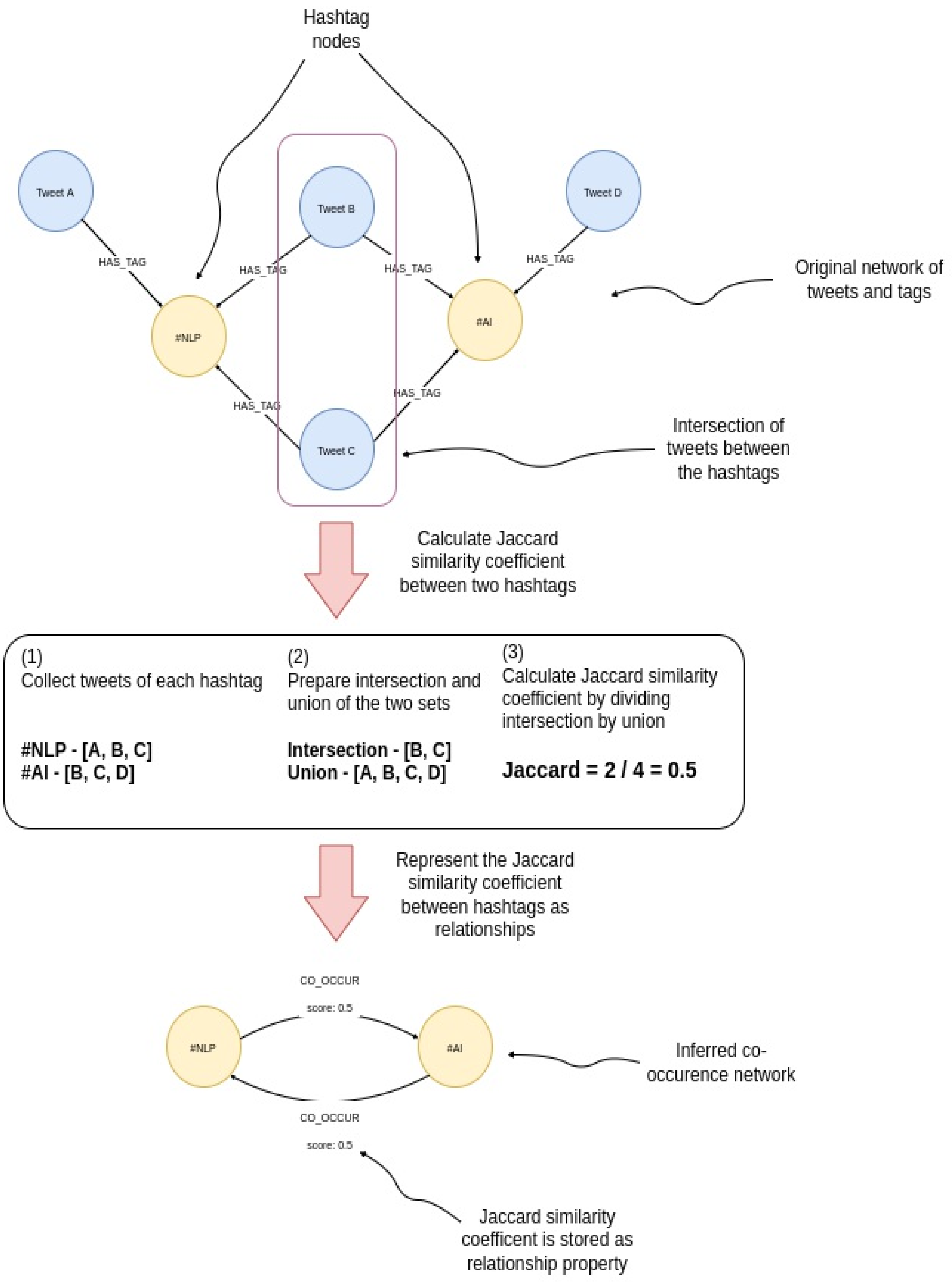


Figure 7.9 visualizes two baskets, where each basket contains a set of products. For example, basket A includes a couch, speakers, phone, and TV, while basket B contains a phone, tv, and headphones. If you want to calculate the Jaccard similarity coefficient between the two baskets, you first calculate the intersection and union of the two sets of products. Both baskets contain a phone and TV, the intersection of the two sets. There are five different products spread across both baskets, which is the union of the two sets. To calculate the Jaccard similarity coefficient, you simply divide the intersection (2) by the union (5) of the two sets, which results in 0.4. The added benefit of the Jaccard similarity coefficient is that it provides a metric that can be used to evaluate, in this example, how similar two baskets are based on their products.

Jaccard similarity coefficient ranges from values 0 to 1. When there is no intersection of members between two sets, the Jaccard similarity coefficient equals 0. For example, let’s say that basket A contains a sandwich and juice and basket B contains a TV. There is no intersection of items between baskets A and B, which consequently indicates that the Jaccard similarity coefficient between the two baskets is 0. On the other hand, the Jaccard similarity coefficient between two sets with identical members is 1. When two sets have the Jaccard similarity coefficient of 1, it implies that the two sets have the same number of members, with identical members in both sets. In this example, both baskets A and B contain a sandwich and juice. However, if we were to add or remove any item from either basket, the Jaccard similarity would no longer be 1.

The process of evaluating the hashtag overlap with the Jaccard similarity coefficient is the following:

Figure 7.10. Using Jaccard similarity coefficient to examine hashtag overlap.


Hashtag nodes
Tweet A
Tweet B
Tweet C
Tweet D
HAS_TAG
HAS_TAG
HAS_TAG
HAS_TAG
HAS_TAG
HAS_TAG
#NLP
#AI
Original network of tweets and tags
Intersection of tweets between the hashtags
Calculate Jaccard similarity coefficient between two hashtags
(1) Collect tweets of each hashtag
#NLP - [A, B, C]
#AI - [B, C, D]
(2) Prepare intersection and union of the two sets
Intersection - [B, C]
Union - [A, B, C, D]
(3) Calculate Jaccard similarity coefficient by dividing intersection by union
Jaccard = 2 / 4 = 0.5
Represent the Jaccard similarity coefficient between hashtags as relationships
CO_OCCUR
score: 0.5
#NLP
#AI
CO_OCCUR
score: 0.5
Inferred co-occurence network
Jaccard similarity coefficent is stored as relationship property

In graph context, a typical input to the Jaccard similarity algorithm is a bipartite network consisting of two types or sets of nodes. The idea behind using the Jaccard similarity algorithm is to project a monopartite graph based on the bipartite input graph. Figure 7.10 visualizes the process to transform a network consisting of tweets and hashtags to a monopartite network of hashtags based on how many tweets they have in common. The process if the following:

- For each hashtag, you first collect the set of tweets in which it appeared.
- In the next step, you iterate over each pair of hashtags and calculate the Jaccard similarity coefficient by dividing the intersection of the two sets by their union.
- Lastly, you have the option to store the similarity coefficient between a pair of nodes in the form of a relationship.

The semantics of the inferred relationships depends on the domain. You could chose the `CO_OCCUR` type of relationships in the hashtag example. In the basket example, the inferred relationship could have a `SIMILAR` type.

The Jaccard similarity coefficient is a symmetric similarity metric. If node A is similar to node B, that directly implies that node B is similar to node A. Similarly to the Cypher Projection example above, the resulting co-occurrence or similarity network will be undirected. Also, we can store the resulting Jaccard similarity coefficient between node as relationship properties.

7.2.1 Node Similarity algorithm

Since the Jaccard similarity coefficient can be used to evaluate how similar a pair of nodes are, the GDS developers deemed it make sense to name the algorithm *Node similarity algorithm*. Node similarity algorithm compares sets of nodes based on their neighbors using the Jaccard similarity coefficient or the Overlap coefficient. A typical input is a bipartite network consisting of two types of nodes. Nodes with outgoing relationships are being compared, while their outgoing neighbors are used to construct the comparison set.

Figure 7.11. How Node Similarity algorithm constructs comparison sets and evaluates similarity.

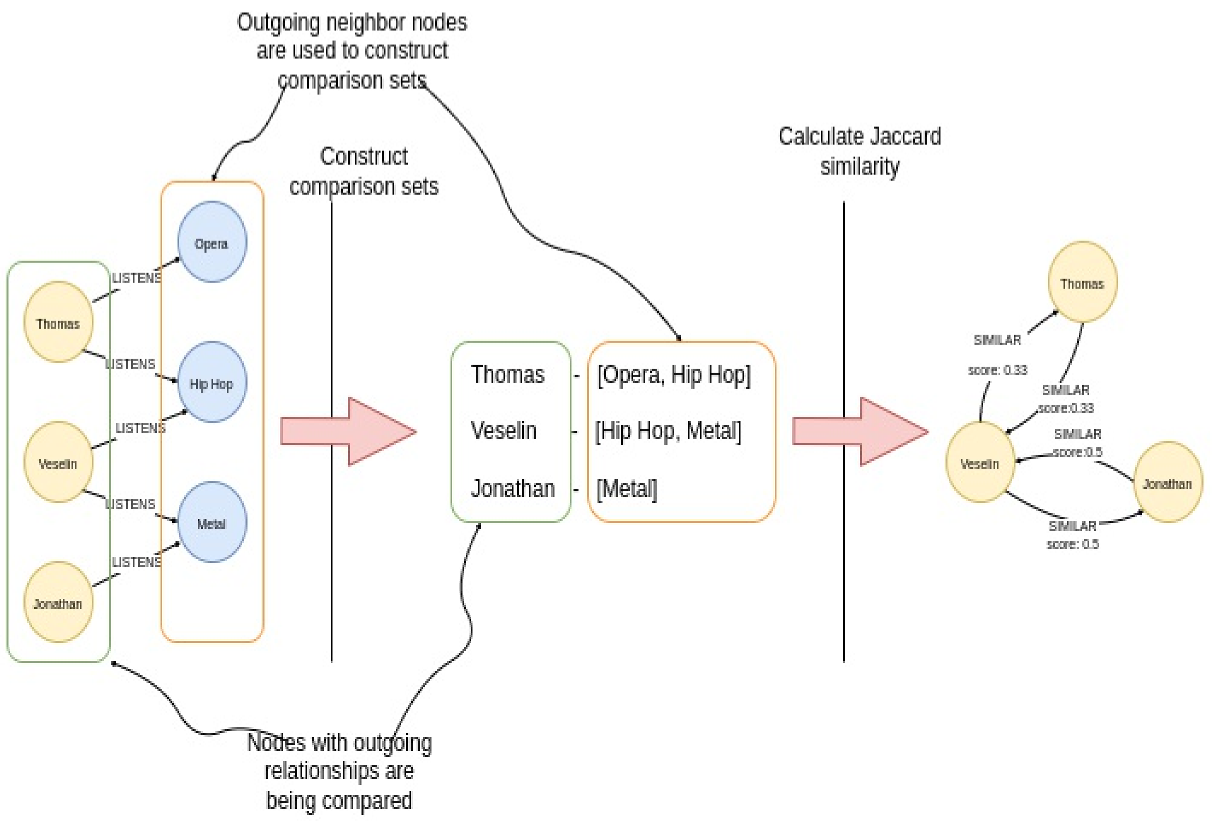

Outgoing neighbor nodes are used to construct comparison sets
Construct comparison sets
Calculate Jaccard similarity
Thomas
Veselin
Jonathan
LISTENS
Opera
Hip Hop
Metal
Thomas - [Opera, Hip Hop]
Veselin - [Hip Hop, Metal]
Jonathan - [Metal]
SIMILAR score: 0.33
SIMILAR score:0.33
SIMILAR score:0.5
SIMILAR score: 0.5
Nodes with outgoing relationships are being compared

Figure 7.11 visualizes a simple network of users and musical genres. The `LISTENS` relationships are directed from users to genres. In this scenario, the Node Similarity algorithm will compare users based on their outgoing neighborhood, which consists of the musical genres they are listening to. Realizing which nodes are being compared by the Node Similarity algorithm is crucial to executing the algorithm correctly. In our Twitter social network, the `HAS_TAG` relationships point from `Tweet` to `Tag` nodes. If you wouldn't reverse the direction of the relationship, you would effectively be comparing tweets based on how many tags they have in common. The GDS library allows reversing the relationship direction during projection, so you don't have to transform the underlying stored graph.

When you want to transform the relationship direction during projection, you need to use the configuration map syntax to describe the projected relationships.

Listing 7.10. Configuration map to describe the relationship type and its orientation.

```
{ALIAS_OF_TYPE: {type:'RELATIONSHIP_TYPE',
                 orientation: 'NATURAL',
                 properties:['property1','property2']}
```

Instead of simply specifying the relationship as a string, you need to construct a relationship configuration map. The `ALIAS_OF_TYPE` key specifies under which name the projected relationship will be available in the in-memory graph. The alias doesn’t have to be identical to relationship types that are stored in the database. Each alias key has a value that consists of a map describing which relationship types should be projected, their orientation, and optional properties. You can manipulate and transform the relationship direction with the `orientation` key. It has three possible values.

- NATURAL: each relationship is projected the same way as it is stored in the database
- REVERSE: each relationship is reversed during graph projection
- UNDIRECTED: each relationship is projected in both natural and reverse orientation

With the `orientation` configuration, you have the option to project the relationship as is, reverse its direction, or treat it as undirected. As mentioned, to treat a relationship as undirected, the engine simply duplicates the relationship in the opposite direction.

Moving on to the hashtag co-occurrence task, you need to project both `Tweet` and `Tag` nodes and include the `HAS_TAG` relationship with a reversed direction.

Listing 7.11. Project `Tweet` and `Tag` nodes and include reversed `HAS_TAG` relationships.

```
CALL gds.graph.project(
    #1
    'tags',
    #2
    ['Tweet', 'Tag'],
    #3
    {REVERSED_HAS_TAG: {orientation:'REVERSE', type:'HAS_TAG'}});
```

Listing 7.11 introduces two new Native Projection syntax options. First, you can specify multiple node labels to be projected using a list. In Listing 7.11, you described both `Tweet` and `Tag` nodes to be projected. Secondly, you used the configuration map syntax to describe the projected relationships. The projected relationships will be available under the `REVERSED_HAS_TAG` alias and contain `HAS_TAG` connections with a reversed relationship direction.

Now that you have the network of tweets and hashtags projected, you can use the Node Similarity algorithm to infer a monopartite network. The two most crucial hyper-parameters of the Node Similarity algorithm are the `topK` and `similarityCutoff` parameters. With them, you can affect how dense or sparse will the inferred network be. The `similarityCutoff` parameter defines the threshold value of the Jaccard similarity coefficient between a pair of nodes that are still regarded as similar. For example, if the `similarityCutoff` is 0.5, then the relationships will be considered only between the pairs of nodes with the Jaccard similarity score of 0.5 or higher. On the other hand, the `topK` parameter specifies the limit on the number of similar relationships per node. As you can directly affect how many relationships should be stored with the `topK` and the `similarityCutoff` parameters, you consequently describe how sparse or dense the inferred co-occurrence network will be.

Defining how sparse or dense the inferred co-occurrence network between hashtags should be will directly correlate with how broad the communities of hashtags that form a topic will be. For example, if you use the `topK` value of 1, each node will have only a single outgoing relationship to its most similar neighbor. However, if you were to increase the `topK` value to 2, each node would have two outgoing relationships that specify which two nodes are the most similar to it.

Figure 7.12. Comparison of how different topK values affect the density of the resulting co-occurrence network.

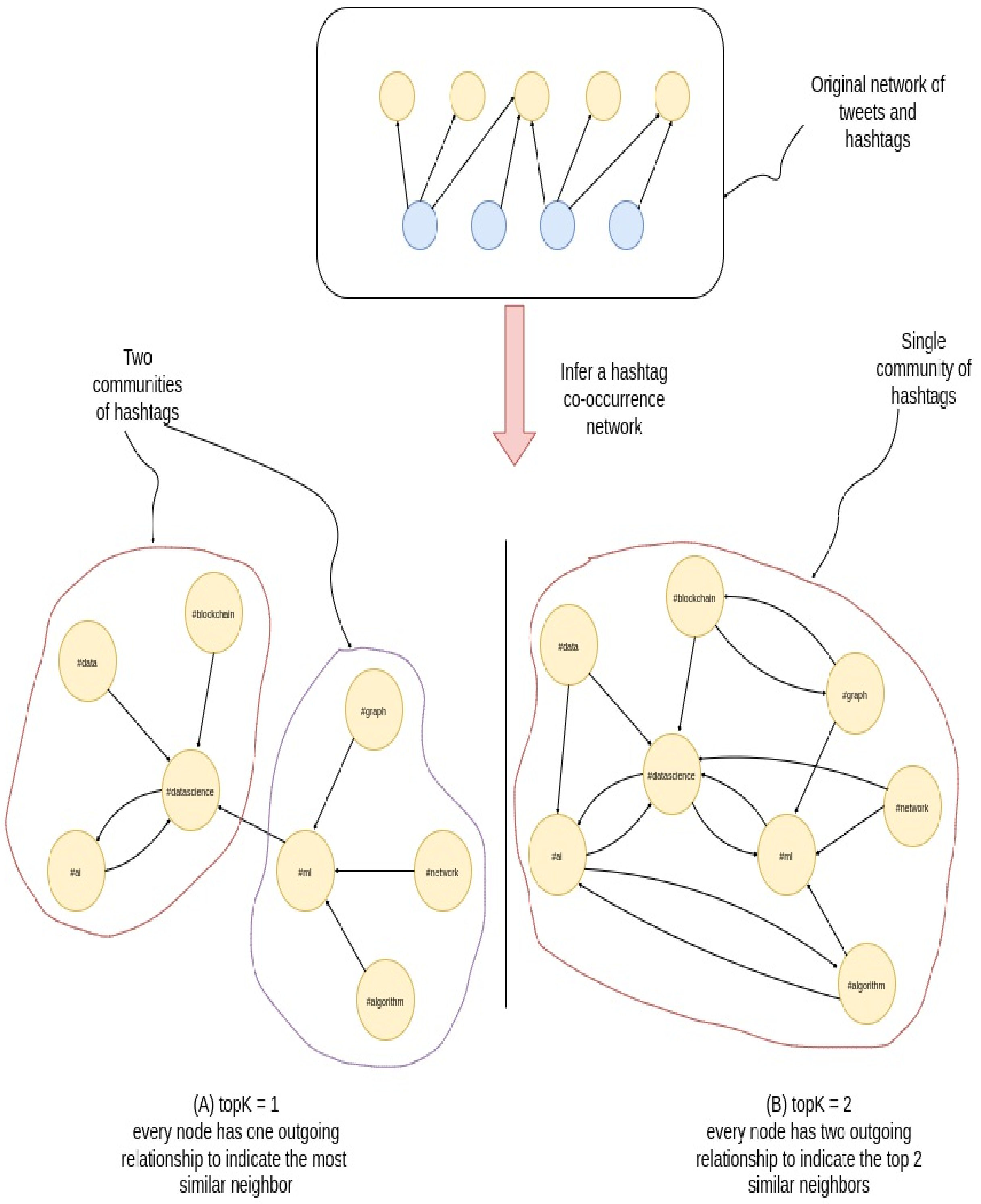

Original network of tweets and hashtags
Infer a hashtag co-occurrence network
Two communities of hashtags
Single community of hashtags
#blockchain
#data
#datascience
#ai
#graph
#ml
#network
#algorithm
(A) topK = 1
every node has one outgoing relationship to indicate the most similar neighbor
(B) topK = 2
every node has two outgoing relationship to indicate the top 2 similar neighbors

Figure 7.12 visualizes a comparison of inferred co-occurrence networks if different `topK` values are used. As mentioned, the input to the Node Similarity algorithm is usually a bipartite network. In this example, you have a bipartite network of tweets and hashtags. The Node Similarity algorithm will then evaluate how similar the hashtags are based on the number of tweets they appear in together. Once the Jaccard similarity coefficient between pairs of hashtags is calculated, the algorithm will output the results as relationships between hashtags. You can observe a co-occurrence network with the `topK` value of 1 on the left-hand side of Figure 7.12. Using the `topK` value of 1, each hashtag will have a single outgoing relationship indicating its most similar hashtag. The resulting co-occurrence network on the left-hand side of Figure 7.12 has eight nodes and eight relationships. For example, the #data hashtag is the most similar to the #datascience hashtag. Even though I previously stated that the Jaccard similarity coefficient is a symmetric similarity metric, the #datascience hashtag does not have a reverse relationship to the #data hashtag. Why is that so? The reason is that once you apply a `topK` filter to the Node Similarity algorithm, you lose the guarantee that all relationships will be symmetric. If you were to set the `topK` value to the number of the nodes in the inferred co-occurrence network, all relationships would be symmetric.

Going back to your scenario, once the co-occurrence network is created, the idea is to use an algorithm like the Label Propagation to find communities of tightly-connected hashtags. A community is defined as a group of densely interconnected nodes that might have sparser connections with other groups. When using a lower `topK` value, there will be fewer connections in the inferred co-occurrence network. Consequently, the size of communities will be smaller as there will be less densely interconnected nodes. Since the communities will be smaller, there will be more of them throughout the network. You defined that each community of hashtags will be regarded as a single topic. Therefore, by adjusting the `topK` value of the Node Similarity algorithm, you are essentially influencing how big or small the resulting communities of hashtags will be. The larger the community of hashtags is, the more broad the resulting topic will be. When the communities of hashtags are larger, you might produce a less granular targeting strategy. On the other hand, using a smaller `topK` value would help you find smaller communities of hashtags and consequently devise a more narrow marketing strategy.

In Figure 7.12, you can observe that when using a `topK` value of 1, the community detection algorithm identified two communities in the resulting co-occurrence network. One community consists of #ai, #datascience, #data, and #blockchain hashtags, while the other community contains #ml, #graph, #network, #algorithm. When you use a higher `topK` value, the resulting co-occurrence network will be more interconnected, and consequently, a community detection algorithm will identify larger communities. A larger community of hashtags will result in more broad topics in your use case. On the right-hand side of Figure 7.12, you can observe that using a `topK` value of 2 produces a more densely-connected co-occurrence network. Since the nodes are more densely-connected, a community detection algorithm identified larger and fewer communities. In Figure 7.12, the algorithm identified only a single community of hashtags when using a higher `topK` value of 2. Defining the `topK` parameter and the `similarityCutoff` parameter is a mix of science and art and depends on your use case. By default, the `topK` parameter value is 10, and the `similarityCutoff` is 1E-42, just slightly above zero. You can evaluate how dense the inferred network would be with default parameter values by using the `stats` mode of the Node Similarity algorithm.

Listing 7.12. Evaluate the Jaccard coefficient distribution of with default parameters.

```
CALL gds.nodeSimilarity.stats('tags', {similarityMetric: 'JACCARD
YIELD nodesCompared, similarityPairs, similarityDistribution
```

Table 7.5. Similarity distribution of the Node Similarity algorithm with default parameters.

nodesCompared	similarityPairs	similarityDistribution
		{ "p1": 0.0005411244928836823, "max": 1.000007625669241, "p50": 0.4400005303323269,

2093	13402	"p10": 0.007575776427984238, "p75": 1.000007625669241, "p25": 0.11111116036772728, "mean": 0.4971548691088963, "stdDev": 0.39913025157864984 }

Executing the Node Similarity algorithm with default parameters would create 13402 relationships between 2093 hashtags. The average similarity score of those relationships is 0.49. Note that this distribution summary does not include the similarity score between all pairs of nodes, but only for the top ten similar neighbors of a node, as that is the default `topK` value. Interestingly, the median value (p50) is close to the average similarity value, and around 25% of relationships have the maximum possible similarity score of 1. When the similarity score is 1, a pair of hashtags are always present together in a tweet.

You can use the `similarityCutoff` parameter to exclude relationships between pairs of hashtags with a similarity score lower than the threshold.

Listing 7.13. Use similarityCutoff parameter to define the similarity score threshold.

```
CALL gds.nodeSimilarity.stats('tags',
  {similarityMetric: 'JACCARD', similarityCutoff:0.33})
YIELD nodesCompared, similarityPairs, similarityDistribution
```

Table 7.6. Similarity distribution of the Node Similarity algorithm by setting the similarityCutoff parameter.

nodesCompared	similarityPairs	similarityDistribution
2093	7733	{ "p1": 0.3333320617675781, "max": 1.000007625669241, "p50": 1.0000057220458984, "p10": 0.3636360168457031, "p75": 1.0000057220458984, "p25": 0.5000019073486328, "mean": 0.7893913483135638, "stdDev": 0.2606311318508915 }

You can observe that by setting the `similarityCutoff` value to 0.33, only 7733 relationships would be created instead of 13402 with default parameter values. The 25th percentile value is 0.5, and interestingly, the median value is already the maximum score of 1. The average node degree of the resulted network would be around 4. As indicated by the similarity distribution, the relationships would be created between pairs of very similar or highly co-occuring hashtags as the median value is already 1.0.

Exercise 7.3

Test out various combinations of the `topK` and `similarityCutoff` parameters using the stats mode of the Node Similarity algorithm and evaluate how changing their values affect the density of the inferred network.

Unfortunately, there is no clear-cut solution to defining the `topK` and `similarityCutoff` parameters. It reminds me of the Goldilocks dilemma. They have to be just right. If you infer too dense a graph, further analysis of the inferred network might not produce valuable insights. The same applies if you infer too sparse of a graph. As a beginner, I advise you to try various parameter configurations and inspect downstream results. Later, you could apply automatic hyper-parameter optimization methods when you grasp the underlying data structure and how configuration values affect results.

With the hashtag co-occurrence example, you will use the `similarityCutoff` value of 0.25 and the `topK` value of 50. As you will execute other graph algorithms on the inferred co-occurrence network, you will use the `mutate` mode of the Node Similarity algorithm. The `mutate` mode stores the inferred network to the in-memory graph, which allows you to use the results of the Node Similarity algorithm as input to other graph algorithms.

Listing 7.14. Mutate the hashtag co-occurrence network to the in-memory graph.

```
CALL gds.nodeSimilarity.mutate('tags',
  {topK:50, similarityCutoff:0.25,
    mutateRelationshipType:'CO_OCCURRENCE',
    mutateProperty:'score',
    similarityMetric: 'JACCARD'})
YIELD nodesCompared, relationshipsWritten
```

The inferred co-occurrence network of hashtags contains 2093 nodes and 9992 relationships.

7.2.2 Co-occurence network characterization

Before moving on to the community detection part, you will consolidate your knowledge of characterizing a network using graph algorithms.

You can use the node degree algorithm to further evaluate the node degree distribution of the inferred co-occurrence network. The thing is that the projected `tags` graph now contains `Tweet` and `Tag` nodes as well as `REVERSE_HAS_TAG` and `CO_OCCURRENCE` relationships. You can filter which nodes or relationships the algorithm should consider at the algorithm execution time with the `nodeLabels` and `relationshipTypes` parameters.

Listing 7.15. Evaluate node degree distribution of the hashtag co-occurrence network.

```
CALL gds.degree.stats('tags',
  {nodeLabels:['Tag'], relationshipTypes:['CO_OCCURRENCE']})
YIELD centralityDistribution
```

Table 7.7. Node degree distribution of the hashtag co-occurrence network

p99	21.00011444091797
min	0.0
max	40.00023651123047
mean	5.351917056393738
p90	13.000053405761719
p50	3.0000076293945312
p999	29.00011444091797
p95	17.00011444091797

p75	8.000053405761719

Both the `nodeLabels` and the `relationshipTypes` parameters expect a list as an input. The ability to filter nodes and relationships at algorithm execution time is a convenient feature that allows you to analyze various parts of the projected graph or analyze a newly inferred network.

The average node degree of the hashtag co-occurrence network is 5.3. Some of the hashtags have no `CO_OCCURRENCE` relationships, while at least one hashtag frequently co-occurs with 40 other hashtags. The `topK` parameter value of 50 did not affect the resulted network, as the highest degree is only 41.

Exercise 7.4

Execute the Weakly-Connected components algorithm on the hashtag co-occurrence network and store the results to the database as a node property `tcWcc`. Provide the `nodeLabels` and `relationshipTypes` parameters so that the algorithm will only consider the desired subset of the projected graph. Use the `write` mode of the algorithm to store the results back to the database.

Table 7.8. Summary statistics for the WCC algorithm executed on the hashtag co-occurrence network.

componentCount	componentDistribution
469	{ "p99": 19, "min": 1, "max": 491, "mean": 3.9808102345415777, "p90": 6,

	"p50": 2, "p999": 491, "p95": 11, "p75": 3 }

The `write` mode of the WCC algorithm also provides the high-level summary of the results, similar to the `stats` mode. There are 469 components in the hashtag co-occurrence network, and the largest contains 491 member, which is around 25% of the whole network. You can imagine that you are dealing with quite a disconnected network as most of the components have 10 or less members.

Exercise 7.5

Identify how many components have ten or fewer members. First, you will need to count how many members are in each component based on their `tcWcc` property. After the first aggregation, you need to apply the filter and ignore components with more than ten members. In the last step, you simply use the `count` function again to count the number of filtered components.

445 components out of 467 have ten or fewer members. One of the reasons why the inferred network is so disconnected is because you are dealing with a tiny subset of the Twitter social network. I think that adding more data would help to connect some of the components. On the other hand, hashtags like meditation or self-help will probably never frequently co-occur with AI or machine learning, and even if they do by some chance, they will never reach the similarity threshold where the co-occurrence relationship will be created between them.

7.2.3 Inspect community structure with Label Propagation algorithm

So far, you have only learned how to use the Weakly and Strongly Connected

Components algorithms to evaluate the community structure. As the last part of this chapter, you will learn how to use the Label Propagation algorithm (LPA) to find non-overlapping communities of hashtags. What is the difference between a community and a component? With the Weakly Connected components algorithm, a component consists of nodes that can reach one another in the graph when ignoring the relationship direction. On the other hand, a community is defined as a group of densely interconnected nodes that might have sparser connections with other groups.

Figure 7.13. Example visualization of a network community structure.

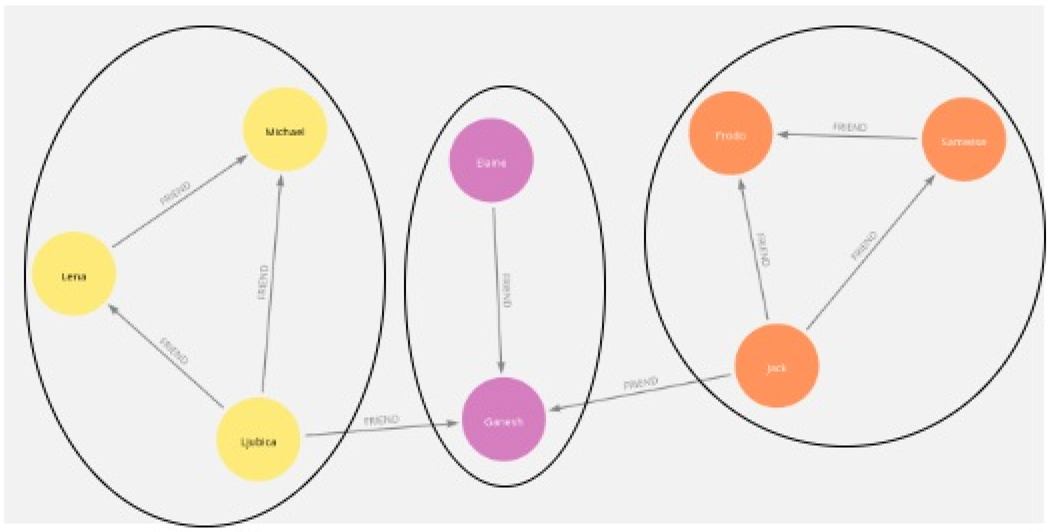


Figure 7.13 visualizes a network that consists of only a single weakly-connected component. However, when you run a community detection algorithm like Label Propagation (LPA) on this network, the algorithm will identify groups of highly-connected nodes. There are three communities in Figure 7.13. For example, there is a community on the left-hand side where members are highly-connected. Similarly, there is another community with densely connected nodes on the right-hand side of the visualization, while one of those nodes also have connections to central community.

You can execute the `mutate` mode of the Label Propagation algorithm with the following Cypher statement.

Listing 7.16. Execute the Label Propagation algorithm on the hashtag co-occurrence network and

store the results to the in-memory graph.

```
CALL gds.labelPropagation.mutate('tags',
  {nodeLabels:['Tag'], relationshipTypes: ['CO_OCCURRENCE'],
   mutateProperty:'community'})
YIELD communityCount, communityDistribution;
```

As you can see, most of the graph algorithms follow the same syntax, which makes it easy to try out various graph algorithms. Again, you had to use the `nodeLabels` and the `relationshipTypes` parameters to select the hashtag co-occurrence network.

If you want to evaluate the results with Cypher, you need to store the mutated `community` property from the in-memory graph to the Neo4j stored graph. You can store the node properties from the in-memory graph to the database with the *gds.graph.writeNodeProperties* procedure.

Run the following Cypher statement to store the mutated `community` property from the in-memory graph to the database.

Listing 7.17. Write the mutated in-memory graph node properties to the database.

```
CALL gds.graph.writeNodeProperties('tags', ['community'])
YIELD propertiesWritten
```

The algorithm results are now available as the `community` node property of the `Tag` nodes.

You will now inspect the five largest communities and examine some of their members.

Listing 7.18. Inspect the five largest communities of hashtags.

```
MATCH (t:Tag)
RETURN t.community AS community,
       count(*) AS communitySize,
       collect(t.id)[..7] AS exampleMembers
ORDER BY communitySize DESC
LIMIT 5
```

Table 7.9. Top five largest communities of hashtags

community	communitySize	exampleMembers
15809	43	["#mentalism", "#respect", "#special-needs", "#mondayvibes", "#goals", "#mindset", "#anxiety"]
15828	42	["#auto_tagging", "#data_entry", "#itrules", "#writingcommunity", "#feg", "#crypto", "#tsa"]
17537	35	["#programming", "#ml", "#iiot", "#iotpl", "#rstats", "#cybersecurity", "#serverless"]
16093	34	["#nlpimpulse", "#iserlohn", "#zoom", "#selbstbild", "#selbstwert", "#spiegelbild", "#werte"]
		["#artproject", "#nft", "#art", "#nfts",

16271	31	"#oculusquest", "#gaming", "#xrhub"]

The largest community of hashtags has 43 members. From the looks of it, the overall topic of the largest community is focused on mental health and personal growth. At first, I wasn't expecting these types of tweets in the dataset, but now I know that NLP can refer to both Natural Language Processing or Neuro-Linguistic Programming. The third and fourth largest communities are centered around computer science and software development. On the other hand, the fifth largest community seems to revolve around NFTs, and interestingly also VR like Oculus Quest are mentioned.

You can remove the limit on the members as well as the limit of rows to further analyze the community structure.

Exercise 7.6

Identify the community of hashtags where:

- `#nlp` is a member
- `#graph` is a member

Table 7.10. Communities where #nlp or #graph are members

15699	["#graphdatabases", "#hcm", "#peopleanalytics", "#hranalytics", "#graphdatascience", "#twitch", "#graph", "#neo4j"]
17533	["#datascience", "#ai", "#machinelearning", "#iot", "#python", "#nlp", "#100daysofcode", "#deeplearning",

	"#artificialintelligence", "#bigdata", "#robots"]

The results of Exercise 7.6 provide recommendations for hashtags that you can use to devise a marketing strategy for your company. You could also explore other communities and search for other hashtags that might be relevant to your company. On the other hand, you would probably want to exclude topics of hashtags that are not relevant, like the self-help domain in this example.

Sometimes you get large communities and want to identify only the most central hashtags. In the marketing strategy example, you might want to identify the most central hashtags of self-help communities to exclude them in your targeting. By identifying the most central hashtags to exclude, you would probably most efficiently exclude particular topics without excluding hundreds of hashtags. You can run the PageRank algorithm to find representatives of communities. To find representatives with the PageRank algorithm, you need to execute it on each community separately. Unfortunately, you can't filter by mutated node properties at algorithm execution time. But, you can use the *subgraph projection* feature, which allows you to project a subset of an existing in-memory graph by specifying node and relationship filters.

Listing 7.19. Subgraph Projection syntax.

```
CALL gds.beta.graph.project.subgraph(
  graphName: String, -> name of the new projected graph
  fromGraphName: String, -> name of the existing projected graph
  nodeFilter: String, -> predicate used to filter nodes
  relationshipFilter: String -> predicate used to filter relation
)
```

You can use the *nodeFilter* parameter to filter nodes based on node properties or labels. Similarly, you can use *relationshipFilter* parameter to filter relationships based on their properties and types. Filter predicates are Cypher predicates for either a node or a relationship. The filter predicate always needs to evaluate to true or false. Variable names within predicates are not arbitrary chosen. A node predicate must refer to variable `n`, while the relationship predicate must refer to variable `r`.

The reason you have used the `mutate` mode of the Label Propagation algorithm is that you can now used the mutated properties for subgraph projections. If you would have used the `write` mode directly, the Label Propagation algorithm results would not be available in the in-memory graph, and so, you couldn’t filter on them.

You can execute the following Cypher statement to project a subgraph that contains only the largest community of hashtags.

Listing 7.20. Project a subgraph that contains only the largest community of hashtags

```
CALL gds.beta.graph.project.subgraph(
  'largest-community',
  'tags',
#1
  'n.community = 15809',
#2
  '*'
)
```

Note that the community ids from the Label Propagation algorithm are not deterministic, meaning that different values of community ids might be assigned. If the id of the largest community is different in your case, you need to change the #A part of the Cypher statement in Listing 7.20. Finally, you can execute the PageRank algorithm on the newly projected `largest-community` in-memory graph to identify its representatives.

Listing 7.21. Identify representatives of the largest community with the PageRank algorithm.

```
CALL gds.pageRank.stream('largest-community')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS tag, score
ORDER BY score DESC
LIMIT 5
```

Table 7.11. Top five representatives of the largest community of hashtags

tag	score
#selfcare	1.854847517775231
#healing	1.854847517775231
#meditation	1.7425376890993822
#mindset	1.7156089968110972
#magic	1.3470665285015495

You would want to exclude the hashtags in Table 7.11 when targeting the `#nlp` hashtag to capture the natural language processing topic but exclude the non-relevant self-help domain.

Exercise 7.7

Find representatives of other communities. You need to use the subgraph projection feature to filter relevant nodes and then use the PageRank algorithm to find its representatives.

Congratulations, you have learned how to infer a co-occurrence network and analyze its community structure. The most crucial step in analyzing the hashtag co-occurrence network was the definition of the `topK` and `similarityCutoff` parameters of the Node Similarity algorithm. As stated, the `topK` and the `similarityCutoff` parameters will directly affect how dense the inferred co-occurrence network will be. Consequently, the density of the co-occurrence network wil correlate with how large the identified communities will be, which in the hashtag co-occurrence example means how broad the resulting topics will be. I recommend you test out various configurations of the two parameters and examine how it affects the resulting hashtag communities.

7.2.4 Drop projected in-memory graphs

It is important to remember to release the projected in-memory graph once you are done with the analysis to free up memory for other analysis. The following Cypher statement will drop all currently projected graphs.

Listing 7.22. Release all projected graphs from memory.

```
#1
CALL gds.graph.list() YIELD graphName
#2
CALL gds.graph.drop(graphName) YIELD nodeCount
RETURN 'dropped ' + graphName
```

7.3 Summary

- Inferring monopartite networks is a frequent step in graph analysis
- Jaccard similarity coefficient can be calculated by dividing the intersection by the union of two sets
- GDS library uses two directed relationship that point in the opposite direction to represent an undirected relationship
- You can change the relationship direction or treat it as undirected during in-memory graph projection
- Parameters `nodeLabels` and `relationshipTypes` can be used to consider only a subset of the projected graph as an input to a graph algorithm
- Label Propagation algorithm is used to evaluate community structure of a network
- Communities represent densely connected groups of nodes with sparser links between groups
- You can use subgraph projection feature to project a subset of an existing in-memory graph
- PageRank can be used to find representatives of communities in a hashtag co-occurrence network

7.4 References

[Cherepnalkoski & Mozetic, 2015] D. Cherepnalkoski and I. Mozetic, "A Retweet Network Analysis of the European Parliament," 2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), 2015, pp. 350-357, doi: 10.1109/SITIS.2015.8.

[Durazzi et al., 2021] Durazzi, F., Müller, M., Salathé, M. et al. Clusters of science and health related Twitter users become more isolated during the COVID-19 pandemic. Sci Rep 11, 19655 (2021). https://doi.org/10.1038/s41598-021-99301-0

[Al-Zaman, 2021] Al-Zaman, Md. Sayeed. (2021). A bibliometric and co-occurrence analysis of COVID-19-related literature published between December 2019 to June 2020. Science Editing. 8. 57-63. 10.6087/kcse.230.

[Andersen et al., 2020] Andersen, Njål et al. “The emerging COVID-19 research: dynamic and regularly updated science maps and analyses.” BMC medical informatics and decision making vol. 20,1 309. 30 Nov. 2020, doi:10.1186/s12911-020-01321-9

[Kastrin et al., 2014] Kastrin, Andrej & Rindflesch, Thomas & Hristovski, Dimitar. (2014). Link Prediction in a MeSH Co-occurrence Network: Preliminary Results.. Studies in health technology and informatics. 205. 579-583.

[Beveridge et al., 2018] Beveridge, Andrew and Michael M. Chemers. “The Game of Game of Thrones : Networked Concordances and Fractal Dramaturgy.” (2018).

[Cooper, 2020] Kathryn M Cooper (2020). The ingredient co-occurrence network of packaged foods distributed in the United States. Journal of Food Composition and Analysis, 86, 103391.

[Kular et al., 2011] D. K. Kular, R. Menezes and E. Ribeiro, "Using network analysis to understand the relation between cuisine and culture," 2011 IEEE Network Science Workshop, 2011, pp. 38-45, doi: 10.1109/NSW.2011.6004656.

[Pervin, 2015] Pervin, F. (2015). Hashtag Popularity on Twitter: Analyzing Co-occurrence of Multiple Hashtags. In Social Computing and Social Media (pp. 169–182). Springer International Publishing.

[Wang et al., 2016] Wang, R., Liu, W. and Gao, S. (2016), "Hashtags and information virality in networked social movement: Examining hashtag co-occurrence patterns", Online Information Review, Vol. 40 No. 7, pp. 850-866. https://doi.org/10.1108/OIR-12-2015-0378

[Türker & Sulak, 2018] Türker, İ., & Sulak, E. (2018). A multilayer network analysis of hashtags in twitter via co-occurrence and semantic links. International Journal of Modern Physics B, 32(04), 1850029.

[Vitale, 2018] Vitale Nicola. (2018). STUDY ON THE TWITTER HASHTAG-HASHTAG CO-OCCURRENCE NETWORK AND KNOWLEDGE DISCOVERY APPLICATON (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.1289254

7.5 Solutions to exercises

The solution to Exercise 7.1 is the following:

Listing 7.23. Retrieve the top five hashtags by the sum of the combined tweet and retweet count.

```
MATCH (h:Tag)<-[:HAS_TAG]-(t:Tweet)
OPTIONAL MATCH (t)<-[r:RETWEETS]-()
RETURN h.id AS hashtag,
       count(distinct t) AS originalTweetsCount,
       count(r) AS retweetCount
ORDER BY retweetCount + originalTweetsCount DESC
LIMIT 5
```

The solution to Exercise 7.2 is the following:

Listing 7.24. Examine the top five most co-occuring pairs of hashtags.

```
MATCH (h1:Tag)<-[:HAS_TAG]-()-[:HAS_TAG]->(h2:Tag)
#1
WHERE id(h1) < id(h2)
WITH h1,h2,count(*) AS cooccurrences
```

```
ORDER BY cooccurrences DESC LIMIT 5
RETURN h1.id AS tag1, h2.id AS tag2, cooccurrences
```

The solution to Exercise 7.4 is the following:

Listing 7.25. Execute the WCC algorithm on the hashtag co-occurrence network and store the results to the database.

```
CALL gds.wcc.write('tags',
  {writeProperty:'tcWcc',
   nodeLabels: ['Tag'], relationshipTypes: ['CO_OCCURRENCE']})
YIELD componentCount, componentDistribution;
```

The solution to Exercise 7.5 is as follows:

Listing 7.26. Execute the WCC algorithm on the hashtag co-occurrence network and store the results to the database.

```
MATCH (t:Tag)
WITH t.tcWcc AS componentId, count(*) AS componentSize
WHERE componentSize <= 10
RETURN count(*) AS count
```

The solution to Exercise 7.6 is as follows:

Listing 7.27. Identify the members that are in the same community as the #NLP hashtag.

```
MATCH (t:Tag)
WHERE t.id IN ['#nlp', '#graph']
WITH distinct t.community AS target_community
MATCH (o:Tag)
WHERE o.community = target_community
RETURN target_community, collect(o.id) as members
```

8 Constructing a nearest neighbor similarity network

This chapter covers

- Manually extracting node features
- Presenting network motifs and graphlets
- Introducing Betweenness and Closeness centralities
- Constructing a monopartite network based on pairwise cosine similarities
- Using community detection algorithm to complete a user segmentation task

This chapter will describe constructing a similarity network based on node properties or features. Similar to typical machine learning preprocessing workflow, each data point or node can be represented as a vector. In the context of graphs, there are generally two approaches you could take to describe a node as a vector. You could manually produce a set of features that describes a node or use node embedding models to produce vectors representing a node in the network automatically. In this chapter, you will manually create representations of nodes to describe their roles in the network and then use those representations to construct an inferred similarity network.

Figure 8.1. Extract node representations and construct a similarity network based on them.

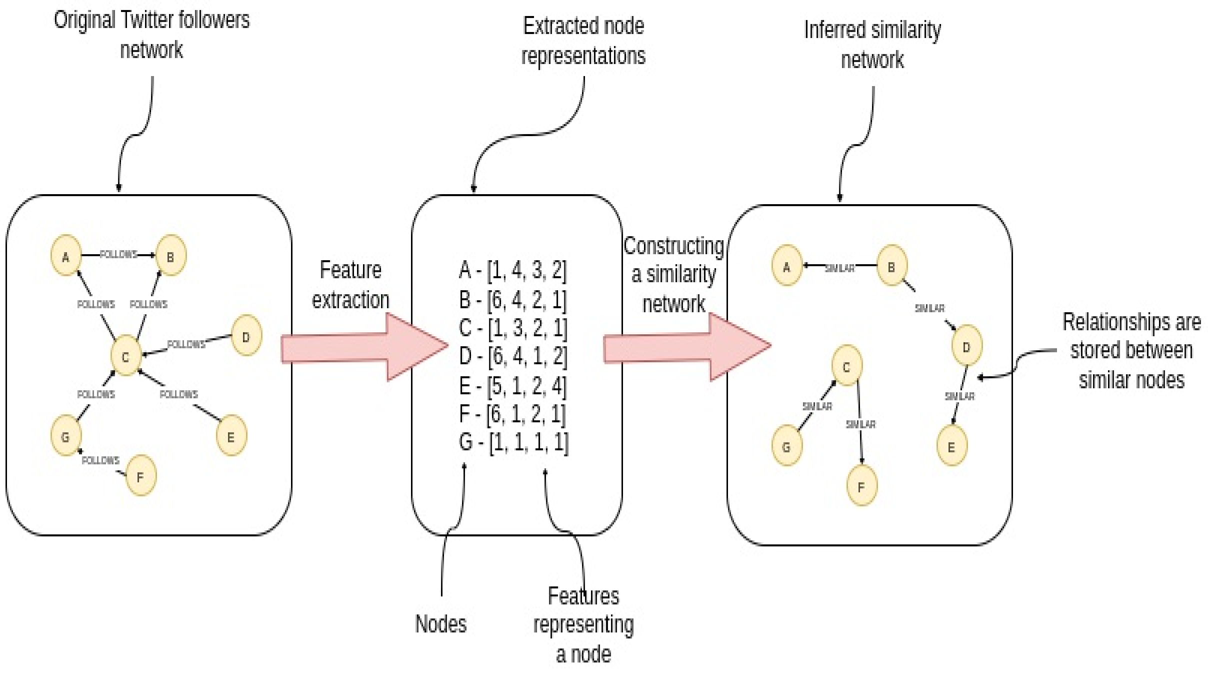
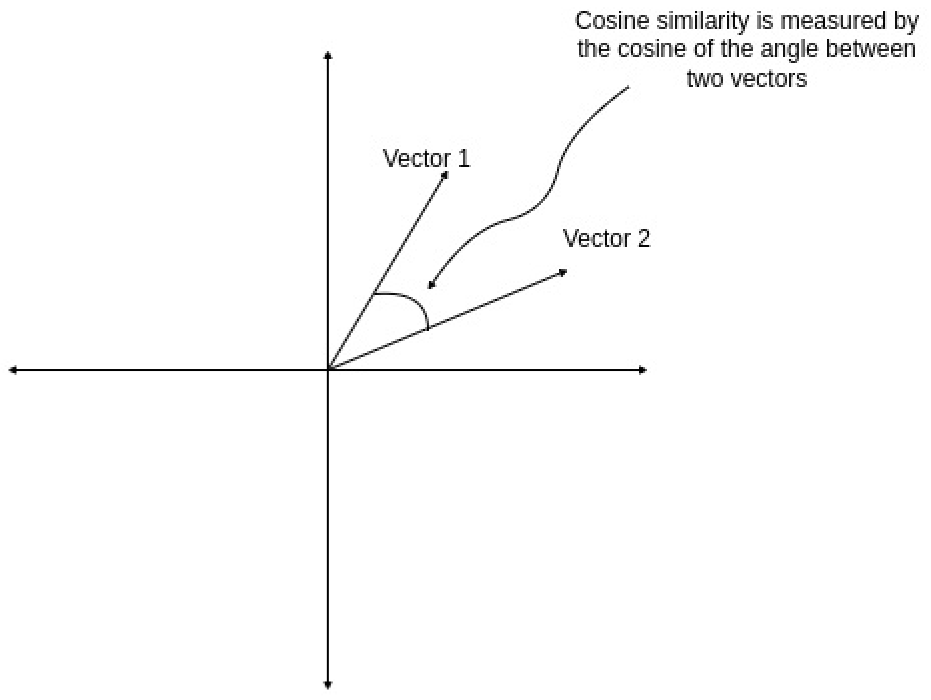

Original Twitter followers network
Extracted node representations
Inferred similarity network
Feature extraction
Constructing a similarity network
Relationships are stored between similar nodes
A - [1, 4, 3, 2]
B - [6, 4, 2, 1]
C - [1, 3, 2, 1]
D - [6, 4, 1, 2]
E - [5, 1, 2, 4]
F - [6, 1, 2, 1]
G - [1, 1, 1, 1]
Nodes
Features representing a node
FOLLOWS
SIMILAR

Figure 8.1 visualizes the process of extracting node features from the follower network. There are multiple approaches to describing a node as a vector. In this chapter, you will manually identify and extract relevant features that will be used to construct a similarity network. After that, you will evaluate how similar the nodes are based on the extracted features. The most common metric to evaluate the similarity between two vectors is *cosine similarity*. Cosine similarity is defined as the cosine of the angle between two vectors. You will calculate cosine similarity between pairs of nodes and store the relationship between nodes deemed similar. Similarly, as in the last chapter, you will define a similarity threshold of when a relationship should be created. Notice that nodes that are connected in the original network are not necessarily connected in the inferred similarity network.

Figure 8.2. Cosine similarity is measured as the cosine of the angle between two vectors.


Cosine similarity is measured by the cosine of the angle between two vectors
Vector 1
Vector 2

Cosine similarity is defined as the cosine of the angle between two vectors as visualized in Figure 8.2. The measure ranges between -1 and 1. When a pair of vectors has an identical direction, meaning that the angle between the vectors is zero, then the cosine similarity is 1. On the other hand, when the two vectors have opposite directions, the cosine similarity is -1. In practice, we deem two vectors similar when their cosine similarity is close to 1.

Imagine you work as an analyst at Twitter. Your supervisor gives you the task of identifying the types of users on the platform. The supervisor doesn't tell you exactly what to look for or how to group users. There are multiple features in the dataset that you could use to describe a user. For example, you know how often and what hashtags they use in their tweets or retweets. You are also aware of who they follow or mention on the platform. Apart from that, you also have some timeline information about when a user or a tweet was created. As with all manual feature engineering, you first have to decide which metrics or features you will use to describe a node. Since you have a small subset of tweets from a small time window, it doesn't make sense to analyze whether users have become inactive by not posting or retweeting anymore. On the other hand, exploring features that would help you split users by content creators and those who primarily only retweet might be interesting. For example, you could take the total count of tweets and the ratio between retweets and all tweets as the first two features. Another interesting metric could be the average time it takes a user to retweet. You could assume that if the average retweet time is minimal, you are most likely dealing with a bot. Another metric that could help you identify bots is inspecting if multiple users post identical content at similar times. Since you have information about followers available, exploring some metrics that encapsulate the position and roles of nodes in the follower network might be worth considering. You will learn how to characterize a node’s immediate neighborhood as well as investigate its *role* in the whole network. A node’s role is a subjective interpretation of the part it plays in the network. For example, you can use the Betweenness centrality algorithm to evaluate which nodes act as bridges between various communities or parts of the network. Similarly, you can use the Closeness centrality to evaluate how close a node is to all the other nodes in the network. Presuming that most if not all information in the Twitter social network spread through follower relationships, you could identify nodes that can disseminate information through the network the fastest due to their position in the network.

After the feature extraction process, you will construct a similarity network between users based on pairwise cosine similarity between their feature vectors. You will then use a community detection algorithm like the Label Propagation algorithm, which was introduced in the previous chapter, to identify various segments of users. Since the relationships connect similar nodes, the community detection algorithm will identify groups of nodes that are densely interconnected in the inferred similarity network. The identified groups of users can be interpreted as user segmentation based on the manually extracted features.

Figure 8.3. Using community detection algorithm to identify segments of users.

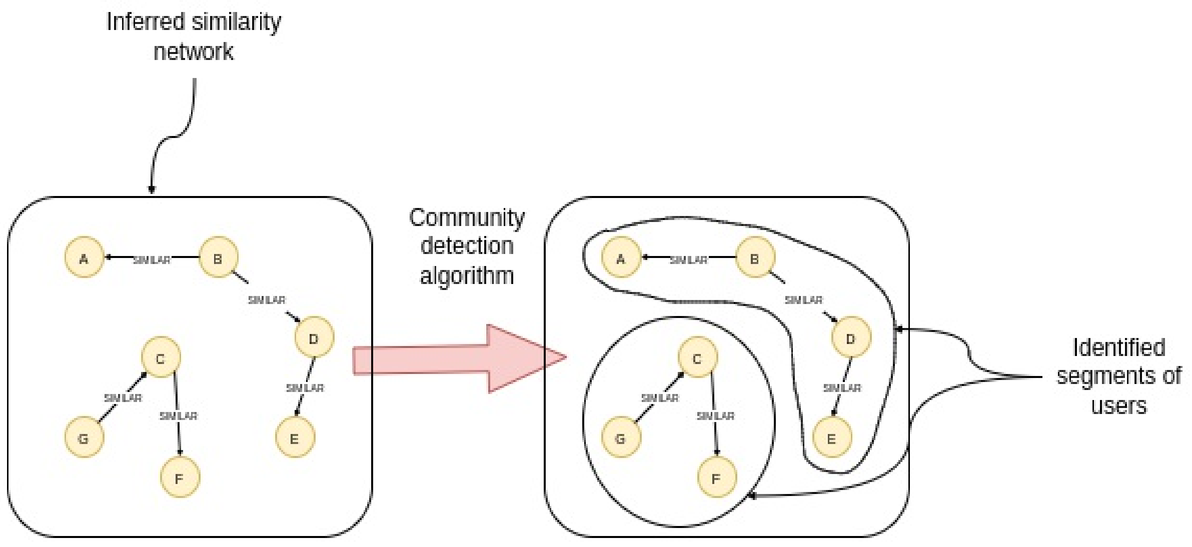

Inferred similarity network
Community detection algorithm
Identified segments of users
A
B
C
D
E
F
G
SIMILAR

Figure 8.3 visualizes the process of using a community detection algorithm on the inferred similarity network to identify groups of users that can be interpreted as segments. The density of the inferred similarity network will directly correlate with the size of communities. You cannot pre-define how many segments you want to identify with this approach. However, you can influence the size of segments by tuning the density of the inferred similarity network.

Multiple research papers [Tinati & Carr, 2012] [Beguerisse-Díaz et al., 2014] focus on defining user roles on the Twitter network. Although extracted features vary from paper to paper, and one can use multiple community detection or clustering techniques to group users together, the underlying idea seems to be always identical. The first part involves identifying and extracting relevant features that describe a user. The feature extraction is done manually, allowing the analyst to explain all the features and their relevance. For example, if you use a model that automatically transforms nodes into vectors, it is hard to explain what those vectors mean. Lastly, researchers then use various community detection or clustering techniques to group users into segments.

You could use the approach of constructing a nearest neighbor graph and evaluating its community structure to identify specific groups or clusters in many other domains. For example, you could use this technique to segment users to create personalized services [Voulodimos et al., 2011] or to cluster customers to improve market forecasting and planning research [Kashwan & Velu, 2013]. You could also use a similar approach to cluster research papers based on their sentence roles [Fukuda & Tomiura, 2018]. While the feature extraction might look very different in different analyses, ranging from employing simple statistics to extracting network features or even document embeddings, the input to the analysis will always be a vector representing each data point. Next, plenty of algorithms are available to group data points based on their vector representations, and I am not here to argue which is best and why. I want to give you an example of using a graph-based approach to unsupervised clustering, where the number of final clusters or communities is not pre-defined.

To follow the exercises in this chapter, you need to have the Twitter network imported into the Neo4j database as described in Chapter 3.

8.1 Feature extraction

As mentioned, the first step in the user segmentation process is the feature extraction. Every node feature will be stored as its property. First, you will use your Cypher knowledge to extract the number of tweets and the ratio between retweets and tweets for each user.

Exercise 8.1

Calculate the number of tweets for each user and store it as the `tweetCount` property. Make sure to include those users that have zero published tweets. Additionally, calculate the ratio between retweets and tweets for each user. Specifically, divide the count of retweets by the sum of retweets and tweets and store it as the `retweetRatio` property. When a user has no retweets or tweets, use a default value of zero. You can use a single or two Cypher statements to calculate both features, whatever is easier for you.

Next, you will evaluate the distribution of how long it takes on average for a user to retweet a tweet after it has been published. One could hypothesize that if a user retweets a lot almost instantly, it is probably a bot.

Listing 8.1. Evaluate the distribution of average duration between a retweet and tweet created dates.

```
MATCH (u:User)-[:PUBLISH]-(retweet)-[:RETWEETS]->(tweet)
#1
WITH u, toInteger(duration.between(
  tweet.createdAt, retweet.createdAt).minutes) AS retweetDelay
#2
WITH u, avg(retweetDelay) AS averageRetweetDelay
RETURN apoc.agg.statistics(averageRetweetDelay,
  [0.05, 0.10, 0.25, 0.5, 0.9]) AS result
```

Listing 8.1 introduces the `duration.between()` function, which is used to calculate the duration between two datetimes. The `duration` temporal type behaves like an object and has multiple methods to extract the duration in years, days, minutes, and more. You can check out all the available methods in the documentation.

Results of Cypher statement in Listing 8.1 are:

Table 8.1. Distribution of average time between retweet and original tweet per user

total	1385
min	0.0
minNonZero	0.05769228935241699
0.1	2.583343267440796
max	1439
0.05	1.0000073909759521

mean	372.21522092560997
0.25	22.00012183189392
0.5	206.00097632408142
0.9	1057.0078122615814
stdev	410.56837279615803

You can observe that you have information for only 1385 users, slightly less than 40% of all users. Five percent of users retweet within a minute and ten percent retweet within 2.5 minutes. We could use a combination of retweets and average time to retweet to identify bots. If a user consistently retweets within a minute or two, you are likely dealing with a bot. You can observe that otherwise, the average time to retweet is around 6 hours, which makes sense for a normal human being who is not constantly looking at their Twitter feed.

Exercise 8.2

Calculate the average duration in minutes between tweet and retweet per user and store it as `timeToRetweet` property. Use the mean value of 372 minutes for users that have never retweeted (have missing values).

Another feature that might indicate bots is looking at if multiple users are posting identical content.

Exercise 8.3

Inspect tweets that are not retweets with identical content, which is available in the `text` property. Additionally, ignore occurrences when a single author posts multiple tweets with the same content.

By solving Exercise 8.3, you can observe that there are only five tweets that all have identical content. Since this feature is present with only five users, only 1 per 1000 users, you will ignore it.

8.1.1 Motifs & Graphlets

Next, you will focus on encoding a user’s role in the follower network. Nodes with similar roles do not have to be next to one another in the network. For example, you could say that users with a large following have a role in producing certain types of content. There could be multiple users with a large following, and they don’t have to follow one another or be close in the network, but they still hold a similar role. In this example, you are effectively examining only the direct neighborhood of a node. You can encode a node’s local neighborhood by counting its positions *graphlets*. A graphlet is a position of a node in a distinctly connected subgraph consisting of k nodes. You might already be familiar with 2-node graphlets, although you probably never heard that name before.

Figure 8.4. Two-node graphlets.

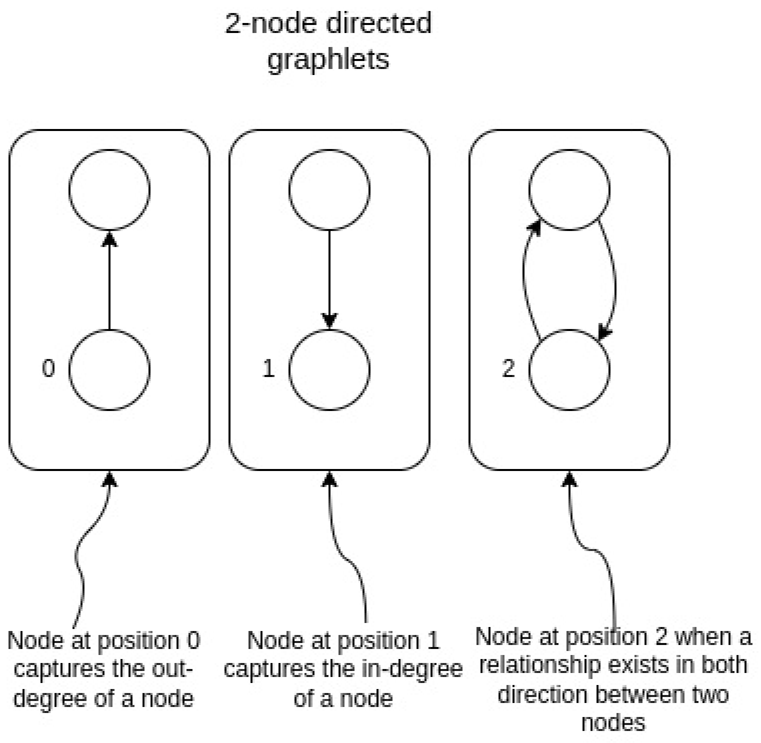

2-node directed graphlets
0
1
2
Node at position 0 captures the out-degree of a node
Node at position 1 captures the in-degree of a node
Node at position 2 when a relationship exists in both direction between two nodes

Figure 8.4 visualized all the two-node directed graphlets. A two-node directed graphlet consists of two nodes and has directed relationships. There are three possible variations of directed relations between two nodes. When you are counting graphlets, you are essentially counting how many times a node is present in that graph pattern. The left-side option shows a node at position 0 that has an outgoing connection. So, if you want to count the graphlet at position zero for a node, you simply count the number of outgoing connections it has. You had already done that before, although you called it an outgoing degree. Similarly, you can count the graphlets at position 1 for each node by evaluating its incoming degree. Lastly, with a directed graph, you can have relationships in both directions between two nodes, as shown on the right-side of Figure 8.4. In some social networks, when two users follow one another, they could be regarded as friends. However, specifically for Twitter, they differentiate between followers and friends in their API documentation, but the definition may lead to ambigous interpretations, so I am unsure precisely what the difference is. Anyhow, you can regard graphlet two as friends in this example.

Exercise 8.4

Calculate the incoming and outgoing degrees for all the users in the follower network and store the results under the `inDegree ` and `outDegree` properties. Additionally, count how many friends (graphlet two) patterns are present for each user and store the output as `friendCount` property.

Next, you will look at 3-node graphlets and calculate some of them to encode a node’s local neighborhood.

Figure 8.5. Three-node graphlets.

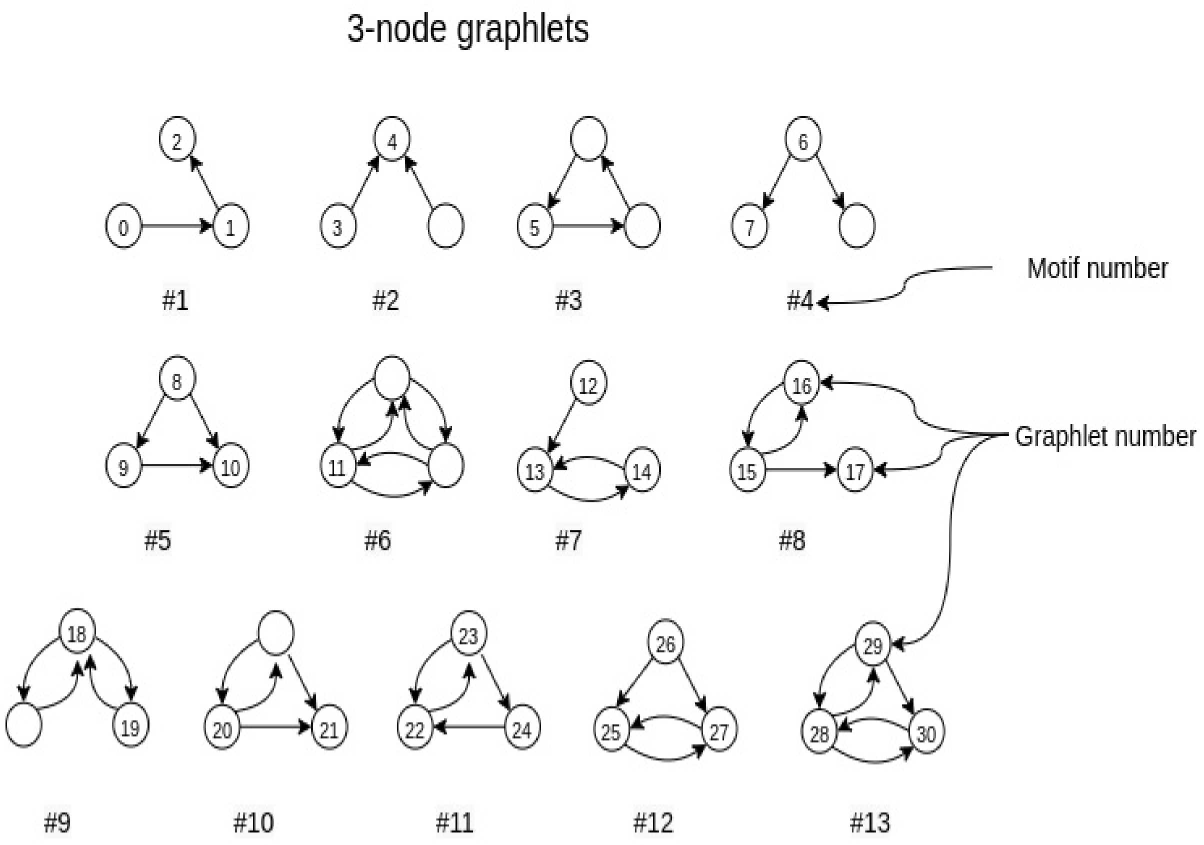


Figure 8.5 visualizes all the 30 variations of directed three-node graphlets. It would be a nice exercise of Cypher to calculate all of them. However, you will only calculate three visualized graphlets in Figure 8.5.

You can also notice that Figure 8.5 shows motif numbers as well as graphlet numbers. What is the difference between the two? A motif is a distinctly connected subgraph, while a graphlet describes a node’s position in the motif. For example, if you look at motif #1, you can observe that it consists of three nodes and two relationships. With motifs, you only count how often this pattern occurs in a network. On the other hand, you can observe that there are three options for a node position in this motif #1, and therefore, there are three graphlets available. Motifs are used to characterize a network structure [Kim et al., 2011], while graphlets come in handy when you want to describe a local neighborhood of a node[Pržulj et al, 2004].

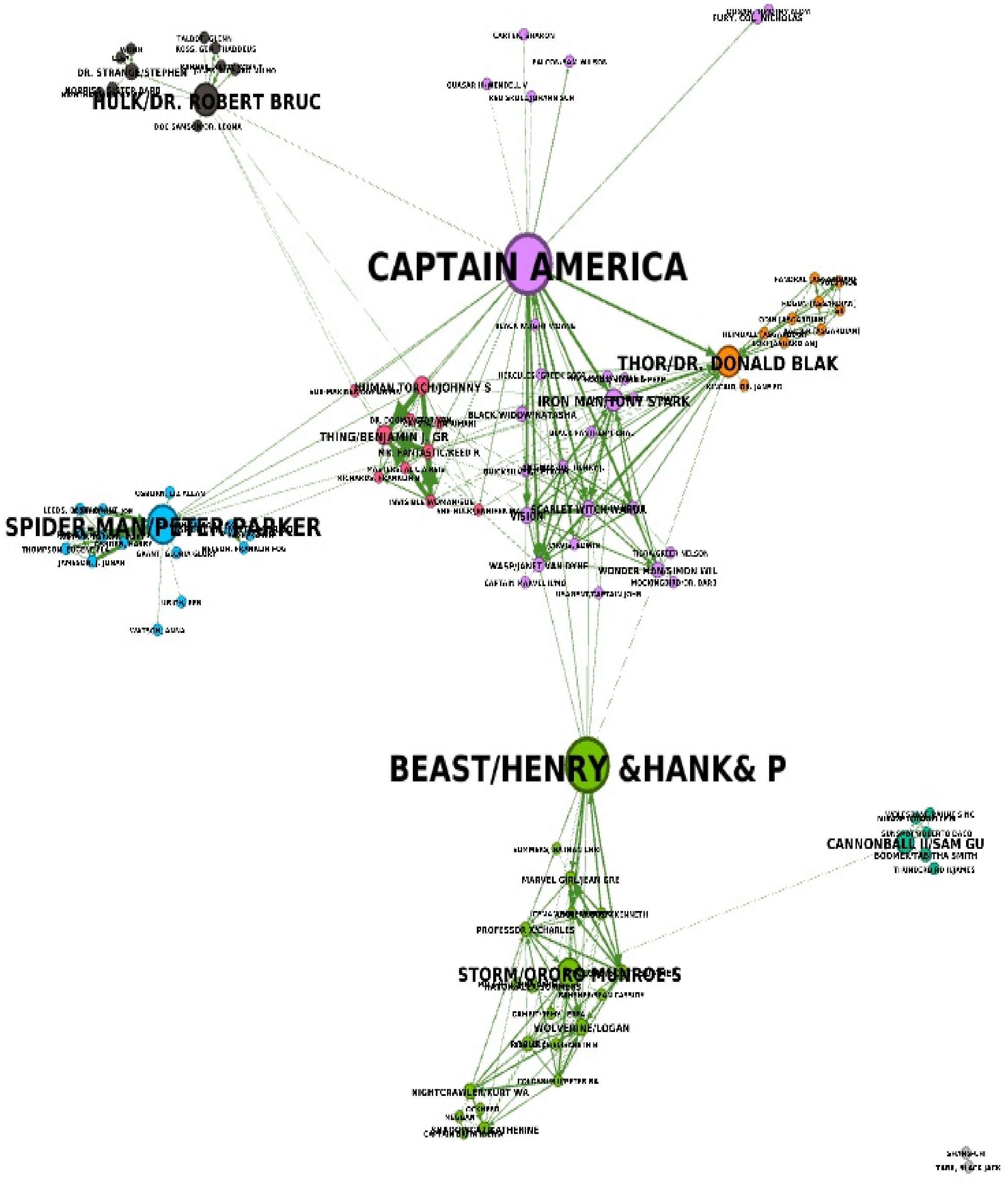
Exercise 8.5

Calculate graphlets 5, 8, and 11 visualized in Figure 8.5 for each user in the follower network and store them as node properties. Store the graphlet 5 under the `graphlet5` node property and so on. I recommend you use a separate Cypher statement for each graphlet calculation.

8.1.2 Betweenness centrality

You have used graphlets to encode the local neighborhood of a node. However, you have not extracted any features that would describe a user’s position in the global network. You will start by executing the Betweenness centrality algorithm to extract a feature that describes how often a user acts as a bridge between various communities. The Betweenness centrality algorithm assumes that all information travels along the shortest paths between nodes. The more often a node lies on those shortest paths, the higher its Betweenness centrality rank.

Figure 8.6. Sample visualization of betweenness centrality rank.



[Figure 8.6](#) visualizes a Marvel network of characters where relationships appear between characters that appeared in the same comic book. The Marvel dataset is available on Kaggle[Sanhueza, 2017] under the CC BY 3.0 license. Both the size of the node and the size of the character name are calculated using the Betweenness centrality. The higher the Betweenness centrality rank, the higher the node and caption size. You can observe that nodes that connect different communities are the largest. For example, Captain America is at the center of the network and acts as a bridge between the central community and all other communities. Another excellent example of the Betweenness centrality is the Beast character, who is the only link between the central and the bottom community in Figure 8.6. If he were to be removed from the network, the network would be split into two separate components. Therefore, the Beast character acts as a bridge between the bottom community and the rest of the network. Acting as a bridge also gives a node influence over the information flow between the two communities.

Before executing the Betweenness centrality algorithm on the follower network, you have to project an in-memory graph. You will use the same projected graph to execute graph algorithms and then construct the nearest neighbor graph. For that reason, you also need to include all the previously calculated node features in the projection.

Completing exercises 8.1 to 8.5 is a requirement to execute the following Cypher statement that projects an in-memory graph.

Listing 8.2. Project the in-memory graph that describes the follower network and includes all the pre-calculated node features.

```
CALL gds.graph.project('knnExample','User', 'FOLLOWS',
 {nodeProperties:['tweetCount', 'retweetRatio', 'timeToRetweet',
  'outDegree', 'friendCount', 'graphlet5', 'graphlet8', 'graphlet
```

Now you can go ahead and execute the Betweeness centrality algorithm. You will use the `mutate` mode to store the results back to the projected graph.

Listing 8.3. Mutate the Betweenness centrality algorithm.

```
CALL gds.betweenness.mutate('knnExample', {mutateProperty:'betwee
```

8.1.3 Closeness centrality

Closeness centrality is a measure that indicates how close a node is to all the other nodes in the network. The algorithm starts by calculating the shortest paths to all the other nodes in the network. Once the shortest paths are calculated, the algorithm sums the distance to all the other nodes. By default, it returns an inverse of the distance sum so that a higher score means that a node has a higher Closeness centrality rank. One can interpret closeness as the potential ability to reach all the other nodes as quickly as possible.

Figure 8.7. Sample visualization of closeness centrality rank.

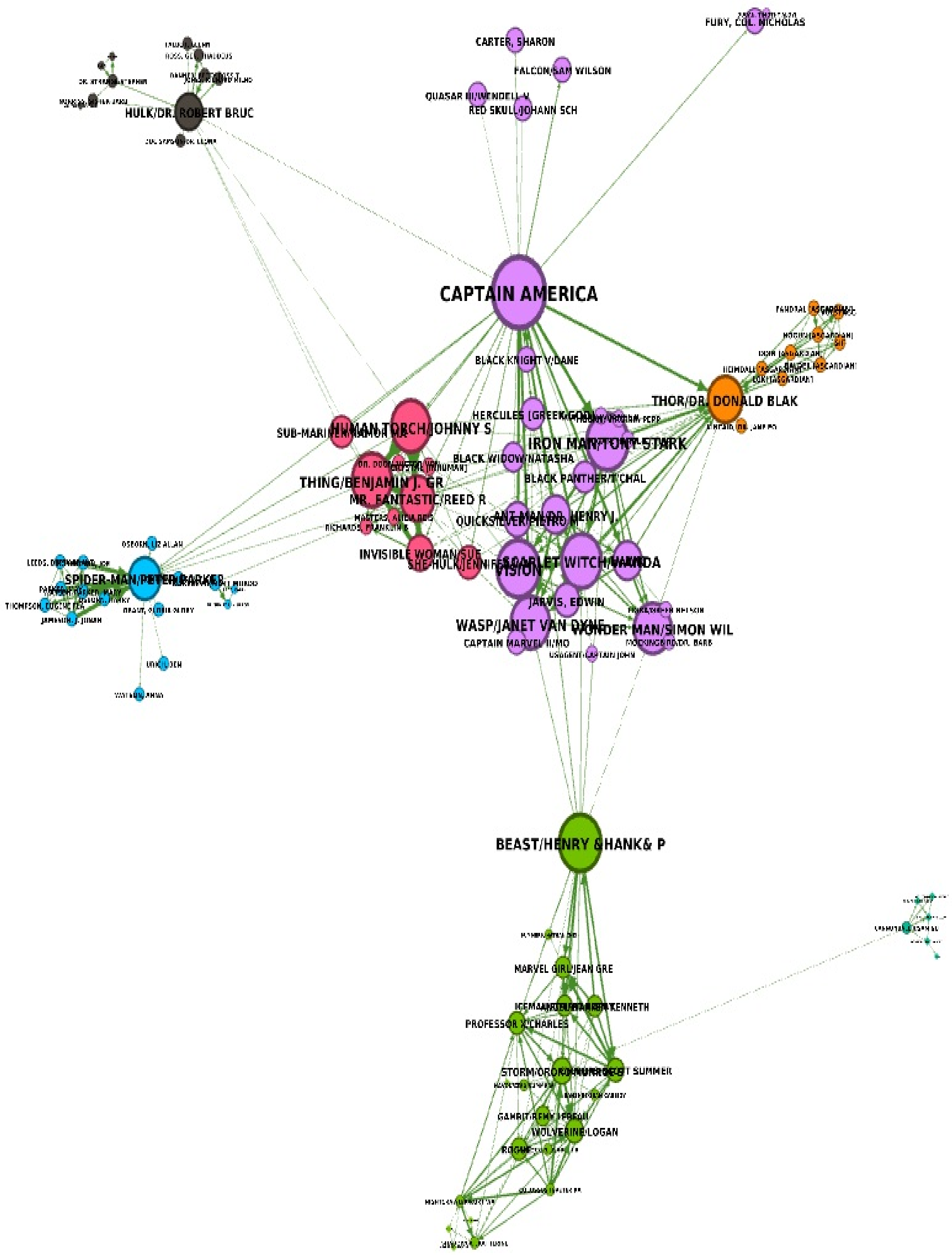


Figure 8.7 visualizes the same Marvel network as the Figure 8.6. The difference is that here the node and the caption size are calculated by the

Closeness centrality algorithm instead of the Betweenness centrality algorithm. You can observe that the largest nodes are in the center of the network, which makes sense as they can reach all the other nodes the fastest. On the other hand, characters on the brinks of the network have a minimal Closeness centrality rank. Captain America is in such a privileged position that he leads in both categories of centralities. On the other hand, for example, Iron Man is trailing far behind Spider-Man and Beast by the Betweenness centrality. However, he is in front of them when looking at the Closeness centrality rank due to his position in the center of the network.

Original Closeness centrality might be unreliable on disconnected graphs. Remember, the algorithm tries to find the shortest path to all the other nodes in the graph. If the original formula is used on disconnected graphs, the shortest path might not exist, and therefore, the sum of all shortest paths from a node might be infinite. In practice, there are several variations of the closeness centrality formula that deal with disconnected graphs. In this example, you will use the Wasserman and Faust variation of the formula[Wasserman & Faust, 1994].

You can execute the `mutate` mode of the Closeness centrality algorithm with the following Cypher statement.

Listing 8.4. Mutate the Closeness centrality algorithm.

```
CALL gds.beta.closeness.mutate('knnExample',
  {mutateProperty:'closeness', useWassermanFaust: true})
```

8.2 Constructing the nearest neighbor graph

You have completed the first step of the user segmentation process by manually extracting the features. The second step is to group or clusters users into segments. As mentioned, several different methods are available to cluster data points based on vector representations. Here, you will construct a nearest neighbor graph based on pairwise cosine similarity between vector representations. Since evaluating cosine similarity between a large number of data points is a relatively frequent process, some algorithm implementations do an intelligent search and avoid comparing all pairs of data points since that doesn’t scale well. Neo4j GDS library implements an efficient similarity search based on the cosine similarity metric.

Before constructing the nearest neighbor graph, it is advisable to explore the distributions and correlations between the features.

8.2.1 Evaluate features

To utilize Cypher’s full expressivity and flexibility to analyze the features, you must store the `closeness` and `betweenness` properties from the projected in-memory graph back to the stored graph.

You can use the `gds.graph.writeNodeProperties` to store the mutated properties back to the database.

Listing 8.5. Store the mutated properties to the database.

```
CALL gds.graph.writeNodeProperties('knnExample',
  ['betweenness', 'closeness'])
```

You will start by examining which node features correlate the most. The GDS library offers a `gds.similarity.pearson(vector1, vector2)` function that calculates the correlation between two vectors. You will compare all pairs of features and identify the most correlating ones. Many clustering techniques are influenced by *feature collinearity* which can skew results. Feature collinearity is a phenomenon when one feature highly correlates with another one.

You can use the following Cypher statement to identify the most correlating features.

Listing 8.6. Identify the five most correlating pair of features.

```
WITH ['tweetCount', 'retweetRatio', 'timeToRetweet', 'friendCount
      'inDegree', 'outDegree', 'graphlet5', 'graphlet8',
      'graphlet11', 'closeness', 'betweenness'] AS features
MATCH (u:User)
# A
UNWIND features as feature1
UNWIND features as feature2
```

```
WITH feature1,
     feature2,
     collect(u[feature1]) as vector1,
     collect(u[feature2]) as vector2
# B
WHERE feature1 < feature2
# C
RETURN feature1,
       feature2,
       gds.similarity.pearson(vector1, vector2) AS correlation
ORDER BY correlation DESC LIMIT 5

# A Use two UNWINDs to compare each feature to all the other
# B Avoid comparing a feature with itself and remove duplicates
# C Calculate correlation
```

Table 8.2. Top five correlating pairs of features

feature1	feature2	correlation
friendCount	graphlet5	0.8173954540589915
graphlet8	outDegree	0.7867637411832583
graphlet11	graphlet5	0.7795975711131173
friendCount	graphlet11	0.6578582639591071
betweenness	friendCount	0.6370096424048863

It appears that some of the features are highly correlated. For example, the `friendCount` highly correlates with `graphlet5`, `graphlet11`, and `betwenness` features. Also, the `graphlet8` variable correlates with the outgoing degree.

To remove some of the highly correlated pairs of features, you will ignore the `friendCount`, `graphlet8`, and `graphlet5` features from from your segmentation process.

Next, you will quickly evaluate the distributions of the remaining features. You can use the following Cypher statement to calculate basic distribution statistics.

Listing 8.7. Mutate the hashtag co-occurrence network to the in-memory graph.

```
WITH ['tweetCount', 'retweetRatio', 'timeToRetweet','inDegree',
      'outDegree', 'graphlet11', 'closeness', 'betweenness'] AS f
MATCH (u:User)
UNWIND features as feature
WITH feature,
     apoc.agg.statistics(u[feature],
                        [0.5,0.75,0.9,0.95,0.99]) as stats
RETURN feature,
       round(stats.min,2) as min,
       round(stats.max,2) as max,
       round(stats.mean,2) as mean,
       round(stats.stdev,2) as stdev,
       round(stats.`0.5`,2) as p50,
       round(stats.`0.75`,2) as p75,
       round(stats.`0.9`,2) as p90,
       round(stats.`0.95`,2) as p95,
       round(stats.`0.99`,2) as p99
```

Table 8.3. Feature distributions

feature	min	max	mean	stdev	p50	p75	p90	p9
"tweetCount"	0.0	754.0	0.96	13.74	0.0	1.0	1.0	2.(
"retweetRatio"	0.0	1.0	0.37	0.48	0.0	1.0	1.0	1.(
"timeToRetweet"	0.0	1439.0	372.08	254.87	372.0	372.0	613.0	94

"inDegree"	0.0	540.0	6.92	22.76	0.0	4.0	16.0	35
"outDegree"	0.0	143.0	6.92	11.95	2.0	8.0	21.0	32
"graphlet11"	0.0	75.0	0.2	1.88	0.0	0.0	0.0	0.
"closeness"	0.0	0.25	0.04	0.06	0.0	0.11	0.13	0.
"betweenness"	0.0	199788.66	2385.97	10885.16	0.0	17.57	4075.66	11

Interestingly, the first thing I noticed is that more than 95% of users have `graphlet11` count of zero. Some might say that you could drop the `graphlet11` feature due to its low variance. However, you will keep it in this example. The Closeness centrality score range from 0.0 to 0.25 with an average of 0.04. On the other hand, the Betweeness centrality is not normalized, so the scores are much higher as it ranges from 0.0 to a value of almost 200 thousand.

While you do not need to normalize features if you are using the cosine similarity metric, you have to be careful if you are using any other metrics. For example, with Euclidean distance, which is simply the distance between two points, a normalization would definitely affect the results.

8.2.2 Inferring the similarity network

You have preprocessed and evaluated the node features. Everything is ready to continue with the user segmentation process. In order to be able to run the community detection algorithm and identify user segments, you first have to infer a similarity network based on a pairwise cosine similarity metric between user vectors. Neo4j GDS library offers an efficient cosine similarity search with the `gds.knn` algorithm. The `gds.knn` algorithm is used to construct a nearest neighbor similarity graph and should not be confused with more mainstream kNN classification or regression models.

Similarly, as in the previous chapter, you can affect how dense or sparse the inferred similarity network will be with the `topK` and `similarityCutoff` parameters. In this example, if you infer a denser network, the resulting communities will be larger, and therefore the user segmentation process will output fewer segments or groups of users. On the other hand, if you infer a sparser similarity network, the segmentation will be more granular. There is no right or wrong way to define the `topk` and `similarityCutoff` parameters. It always depends on your task. In this example, you will use the `topK` value of 65 and leave the `similarityCutoff` at the default value.

Since you need to execute a community detection algorithm on the output of the `gds.knn` algorithm, you will use the `mutate` mode to store the results to the projected graph. The `gds.knn` algorithms creates new relationships between users that pass the similarity threshold defined with the `topK` and `similarityCutoff` parameters.

Run the following Cypher statement to execute the `mutate` mode of the `gds.knn` algorithm.

Listing 8.8. Mutate the user similarity network to the in-memory graph.

```
CALL gds.knn.mutate('knnExample', {
   nodeProperties:['tweetCount', 'retweetRatio', 'timeToRetweet',
      'outDegree', 'graphlet11', 'closeness', 'betweenness'],
   mutateRelationshipType:'SIMILAR',
   mutateProperty:'score',
   topK:65
})
```

8.3 User segmentation with community detection algorithm

The last step in the user segmentation process is to execute a community detection algorithm to identify groups or segments of users. So far, you have used the Label Propagation algorithm to evaluate the community structure of a network. In this chapter, you will instead use the Louvain algorithm. The Louvain algorithm has an identical task as the Label Propagation algorithm to group densely connected nodes into groups or communities. However, it uses slightly different underlying mathematics to achieve this. If you are interested in mathematics, you can read the article in which the Louvain algorithm was proposed[Blondel et al., 2008].

Run the following Cypher statement to `mutate` the results of the Louvain algorithm to the projected in-memory graph.

Listing 8.9. Store the mutated property `userSegmentation` to the database.

```
CALL gds.louvain.mutate('knnExample',
  {relationshipTypes:['SIMILAR'], mutateProperty:'userSegmentatio
```

The `communityCount` output there are 22 identified communities. To further investigate, you have to store the mutated `userSegmentation` property to be able to analyze the segmentation with Cypher.

Listing 8.10. Mutate the hashtag co-occurrence network to the in-memory graph.

```
CALL gds.graph.writeNodeProperties('knnExample', ['userSegmentati
```

Finally, you can go ahead and evaluate the user segmentation results. Use the following Cypher statement to evaluate average feature values for the five largest user segments.

Listing 8.11. Evaluate the user segmentation results.

```
MATCH (u:User)
RETURN u.userSegmentation as community,
       count(*) AS memberCount,
       round(avg(u.tweetCount), 2) AS tweetCount,
       round(avg(u.retweetRatio), 2) AS retweetRatio,
       round(avg(u.timeToRetweet), 2) AS timeToRetweet,
       round(avg(u.inDegree), 2) AS inDegree,
       round(avg(u.outDegree), 2) AS outDegree,
       round(avg(u.graphlet11), 2) AS graphlet11,
       round(avg(u.betweenness), 2) AS betweenness,
       round(avg(u.closeness), 2) AS closeness
ORDER BY memberCount DESC
```

```
LIMIT 5
```

Table 8.4. User segmentation results

community	memberCount	tweetCount	retweetRatio	timeToRetweet	inDegre
270	217	3.5	0.007	385.3	15.28
84	197	1.0	0.001	375.43	0.0
725	179	0.92	0.2	376.72	19.13
737	156	0.0	0.0	372.0	0.0
381	145	0.0	1.0	35.68	2.5

The largest segment contains 217 members, and its members have, on average, 3.5 tweets. They almost don’t do any retweeting since their retweet ratio is 0.07. On the other hand, they have, on average, 15 followers and follow 19 other users. Judging by their high betweenness score, they act as bridges between various communities. On the other hand, the fourth largest community seems to contain inactive and isolated users, at least from our dataset point of view. They don’t have any tweets or retweets and don’t follow anyone or have any followers.

You can remove the `LIMIT` clause from the Cypher statement in Listing 8.11 to evaluate all the 22 segments. Also, you can play around with various `topK` and `similarityCutoff` values of the `gds.knn` algorithm to evaluate how its values affect the user segmentation.

In the previous chapter, you have used the PageRank algorithm to identify representitives of hashtag communities. Here, you can apply the same technique to identify segment representatives. First, use the subgraph projection procedure to filter only users that are in the largest segment. After that, use the PageRank algorithm on the newly filtered projection to identify its representives.

Congratulations, you have learned how to manually extract node features and complete a user segmentation process based on them with the help of `gds.knn` and community detection algorithms.

8.4 Summary

- A node's role in a network can be described with various local neighborhood and global features
- Nodes with a similar role in the network don't have to be close or adjacent
- Motifs are used to characterize a network structure
- Graphlets are used to encode a node's direct neighborhood
- Betweenness centrality is used to identify nodes that act as bridges between various communities
- Closeness centrality identifies nodes that have the potential to share information to all the other nodes the fastest
- A nearest neighbor graph is constructed by evaluating one of the vector similarity measure
- The most common vector similarity measure used is cosine similarity
- It is not vital to normalize input features when using cosine similarity metric to construct a nearest neighbor graph
- Louvain algorithm is very similar to Label Propagation, but uses a different underlying mathematics
- PageRank can be used to find representative nodes in the inferred similarity network

8.5 References

[Tinati & Carr, 2012] Tinati, Ramine & Carr, Leslie & Hall, Wendy & Bentwood, Jonny. (2012). Identifying communicator roles in twitter.

10.1145/2187980.2188256.

[Beguerisse-Díaz et al., 2014] Beguerisse-Díaz, Mariano et al. “Interest communities and flow roles in directed networks: the Twitter network of the UK riots.” Journal of the Royal Society, Interface vol. 11,101 (2014): 20140940. doi:10.1098/rsif.2014.0940

[Voulodimos et al., 2011] Voulodimos, Athanasios & Doulamis, Anastasios & Patrikakis, Charalampos & Sardis, Emmanuel & Karamolegkos, Pantelis. (2011). Employing Clustering Algorithms to Create User Groups for Personalized Context Aware Services Provision. 10.1145/2072627.2072637.

[Kashwan & Velu, 2013] Kashwan, K.R. & Velu, C.. (2013). Customer Segmentation Using Clustering and Data Mining Techniques. International Journal of Computer Theory and Engineering. 5. 856-861. 10.7763/IJCTE.2013.V5.811.

[Fukuda & Tomiura, 2018] Fukuda, S., & Tomiura, Y. (2018). Clustering of research papers based on sentence roles. In ICADL Poster Proceedings. Hamilton, New Zealand: The University of Waikato.

[Kim et al., 2011] Tae-Hwan Kim, Junil Kim, Pat Heslop-Harrison, Kwang-Hyun Cho, Evolutionary design principles and functional characteristics based on kingdom-specific network motifs, Bioinformatics, Volume 27, Issue 2, 15 January 2011, Pages 245–251, https://doi.org/10.1093/bioinformatics/btq633

[Pržulj et al, 2004] N. Pržulj, D. G. Corneil, I. Jurisica, Modeling interactome: scale-free or geometric?, Bioinformatics, Volume 20, Issue 18, 12 December 2004, Pages 3508–3515, https://doi.org/10.1093/bioinformatics/bth436

[Sanhueza, 2017] Claudio Sanhueza (2017, Januar) The Marvel Universe Social Network, Version 1, https://www.kaggle.com/datasets/csanhueza/the-marvel-universe-social-network

[Wasserman & Faust, 1994] Wasserman, S., & Faust, K. (1994). Social network analysis: Methods and applications. Cambridge University Press.

https://doi.org/10.1017/CBO9780511815478

[Blondel et al., 2008] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, & Etienne Lefebvre (2008). Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008(10), P10008.

8.6 Solutions to exercises

The solution to Exercise 8.1 is the following:

Listing 8.12. Calculate the tweet count and retweet ratio for each user.

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:PUBLISH]->(tweet)
WHERE NOT EXISTS { (tweet)-[:RETWEETS]->() }
WITH u, count(tweet) AS tweetCount
OPTIONAL MATCH (u)-[:PUBLISH]->(retweet)
WHERE EXISTS { (retweet)-[:RETWEETS]->() }
WITH u, tweetCount, count(retweet) AS retweetCount
WITH u, tweetCount,
  CASE WHEN tweetCount + retweetCount = 0 THEN 0
    ELSE toFloat(retweetCount) / (tweetCount + retweetCount)
      END AS retweetRatio
SET u.tweetCount = tweetCount,
    u.retweetRatio = retweetRatio
```

The solution to Exercise 8.2 is the following:

Listing 8.13. Calculate average time to retweet per user and store it.

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:PUBLISH]-(retweet)-[:RETWEETS]->(tweet)
WITH u, toInteger(duration.between(
  tweet.createdAt, retweet.createdAt).minutes) AS retweetDelay
WITH u, avg(retweetDelay) AS averageRetweetDelay
SET u.timeToRetweet = coalesce(averageRetweetDelay, 372)
```

The solution to Exercise 8.3 is the following:

Listing 8.14. Calculate average time to retweet per user and store it.

```
MATCH (t1:Tweet), (t2:Tweet)
WHERE NOT EXISTS { (t1)-[:RETWEETS]->() }
  AND NOT EXISTS { (t2)-[:RETWEETS]->() }
  AND id(t1) < id(t2)
  AND NOT EXISTS { (t1)<-[:PUBLISH]-()-[:PUBLISH]->(t2) }
  AND t1.text = t2.text
RETURN t1,t2
```

The solution to Exercise 8.4 is the following:

Listing 8.15. Calculate the 2-node graphlets and store them as node properties.

```
MATCH (u:User)
WITH u,
     size((u)<-[:FOLLOWS]-()) AS inDegree,
     size((u)-[:FOLLOWS]->()) AS outDegree,
     size((u)-[:FOLLOWS]->()-[:FOLLOWS]->(u)) AS friendCount
SET u.inDegree = inDegree,
    u.outDegree = outDegree,
    u.friendCount = friendCount
```

The solution to Exercise 8.5 is the following:

Listing 8.16. Calculate and store the count of graphlet 5.

```
MATCH (u:User)
OPTIONAL MATCH p=(u)-[:FOLLOWS]->()-[:FOLLOWS]->()-[:FOLLOWS]->(u
WITH u, count(p) AS graphlet5
SET u.graphlet5 = graphlet5
```

Listing 8.17. Calculate and store the count of graphlet 8.

```
MATCH (u:User)
OPTIONAL MATCH p=(u)-[:FOLLOWS]->()-[:FOLLOWS]->()<-[:FOLLOWS]-(u
WITH u, count(p) AS graphlet8
SET u.graphlet8 = graphlet8
```

Listing 8.18. Calculate and store the count of graphlet 11.

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:FOLLOWS]->(other1)-[:FOLLOWS]->(other2)-[:FO
               (u)<-[:FOLLOWS]-(other1)<-[:FOLLOWS]-(other2)<-[:F
WHERE id(other1) < id(other2)
WITH u, count(other1) AS graphlet11
```

```
SET u.graphlet11 = graphlet11;
```

The solution to Exercise 8.6 is the following:

Listing 8.19. Project a subgraph that contains only the largest community of hashtags

```
CALL gds.beta.graph.project.subgraph('largestSegment', 'knnExampl
 'n.userSegmentation=270', '*')
```

Listing 8.20. Identify representatives of the particular community with the PageRank algorithm.

```
CALL gds.pageRank.stream('largestSegment',
  {relationshipTypes:['SIMILAR'], relationshipWeightProperty:'sco
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS user, score
ORDER BY score DESC
LIMIT 5
```

9 Node embeddings and classification

This chapter covers

- Introducing node embedding models
- Presenting the difference between transductive and inductive models
- Examining the difference between structural roles and homophily-based embeddings
- Introducing the Node2vec algorithm
- Using Node2vec embeddings in a downstream machine learning task

In the previous chapter, you used a vector to represent each node in the network. The vectors were hand-crafted based on the features you deemed essential. In this chapter, you will learn how to automatically generate node representation vectors using a *node embedding model*. Node embedding models fall under the dimensionality reduction category.

An example of dimensionality reduction is the body mass index (BMI). Body mass index is commonly used to define obesity. To precisely characterize obesity, you could look at a person’s height, weight, measure their fat percentage, muscle content, and waist circumference. In this case, you would be dealing with five input features to predict obesity. Instead of having to measure all five features before an observation can be made, the doctors came up with a body mass index.

Figure 9.1. Body-mass index chart.

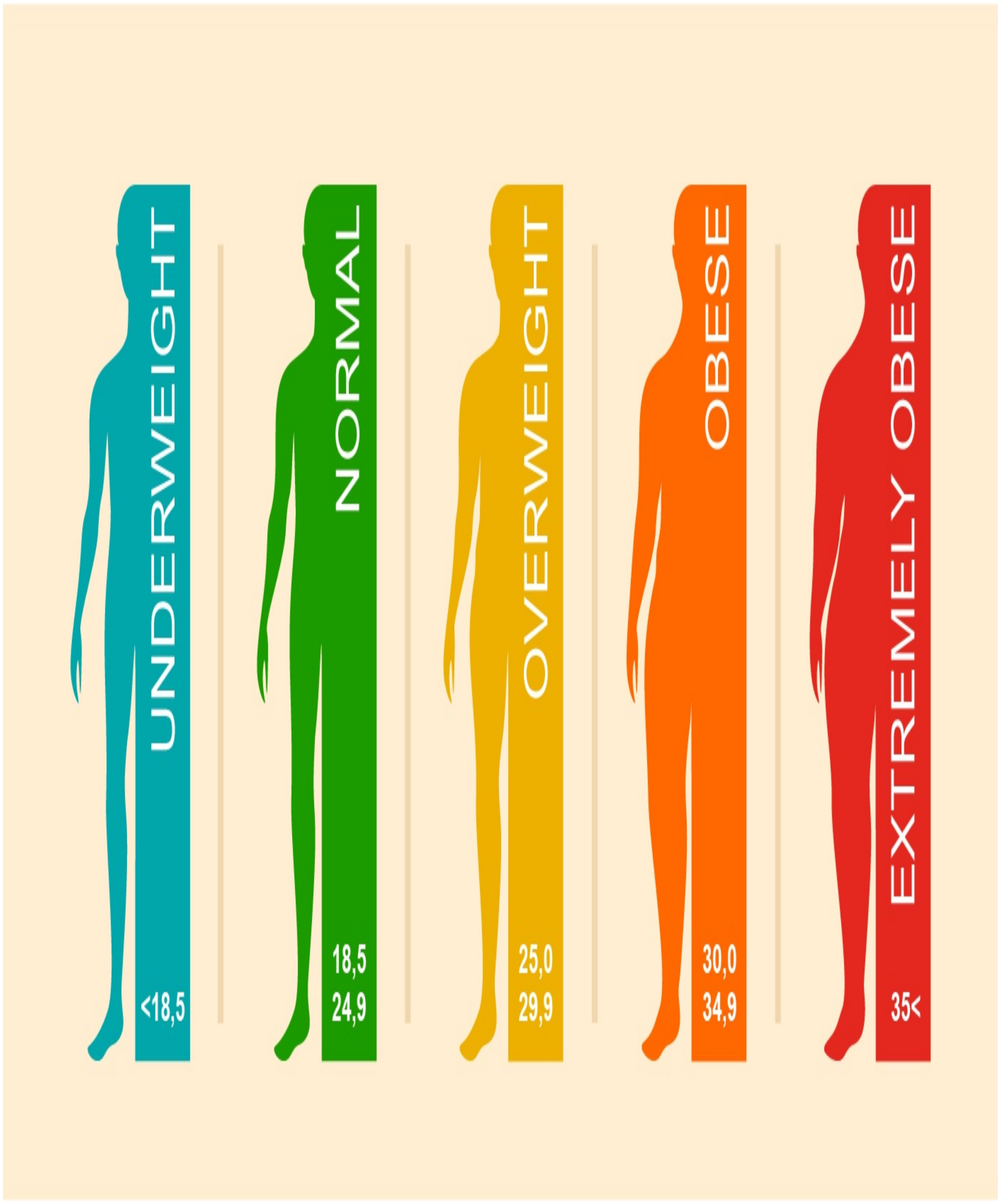
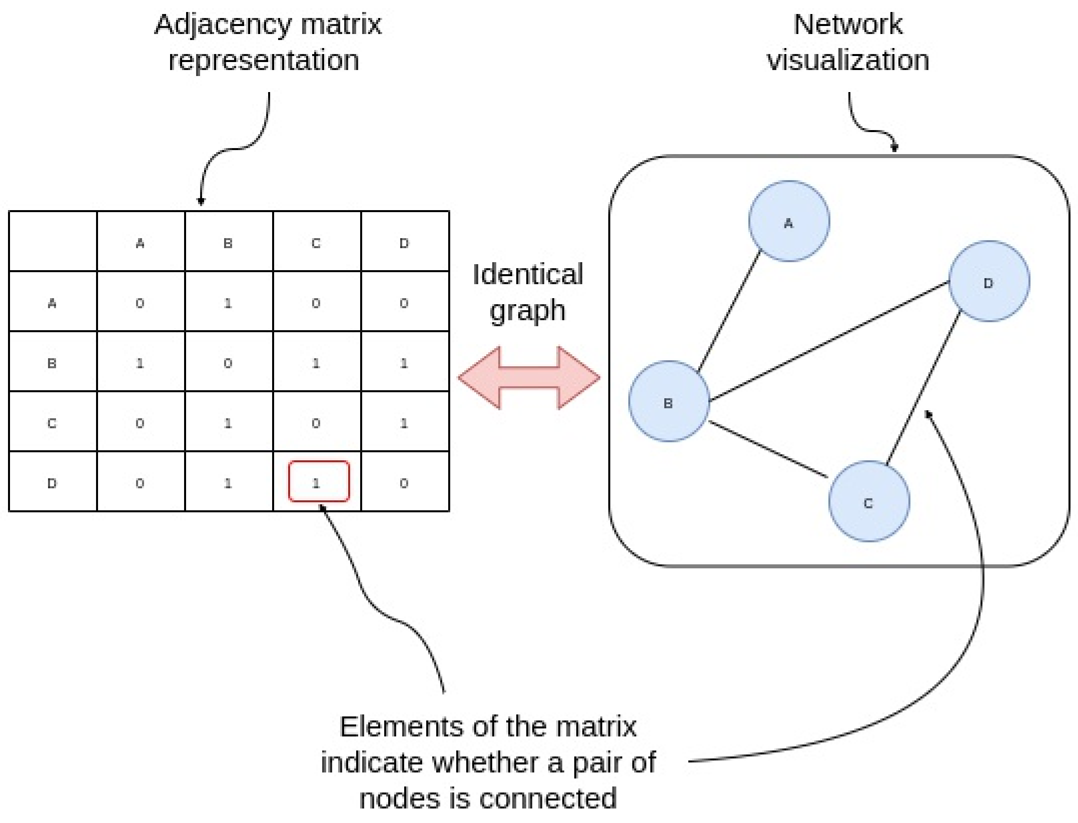


Figure 9.1 visualizes a body mass index scale used to evaluate a person's body type. For example, if the BMI is 35 or above, the BMI scale would regard that person as extremely obese. Body mass index is calculated by dividing a person's weight in kilograms by their height in square meters and

is a rough estimate of body fat. Instead of using five input features, a single embedded feature is a good representation of the expected output. It is a good approximation but by no means a perfect descriptor of obesity. For example, a rugby player would be considered obese given the body mass index, but he probably has more muscles than fat. An embedding model reduces the dimensionality of input features while retaining a strong correlation to a given problem. An added bonus of using an embedding model is that you need to collect less data for training and validating the model. In the case of BMI, you can avoid potentially costly measurements by only comparing the height and weight ratios.

Every graph can be represented as an *adjacency matrix*. An adjacency matrix is a square matrix where the elements indicate whether pairs of nodes are connected. Such a matrix can be regarded as a high-dimensional representation of the network.

Figure 9.2. Adjacency matrix.


Adjacency matrix representation
Network visualization
Identical graph
A
B
C
D
Elements of the matrix indicate whether a pair of nodes is connected

	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

Figure 9.2 visualizes an adjancency matrix representing a graph with four nodes A, B, C, and D. Each element in the adjacency matrix indicates whether the pair of nodes is connected. For example, the element in column C and row D has a value of 1, which indicates that a relationship between nodes C and D is present in the graph. If the value of the element in the matrix is 0, then a relationship between the pair of nodes does not exist.

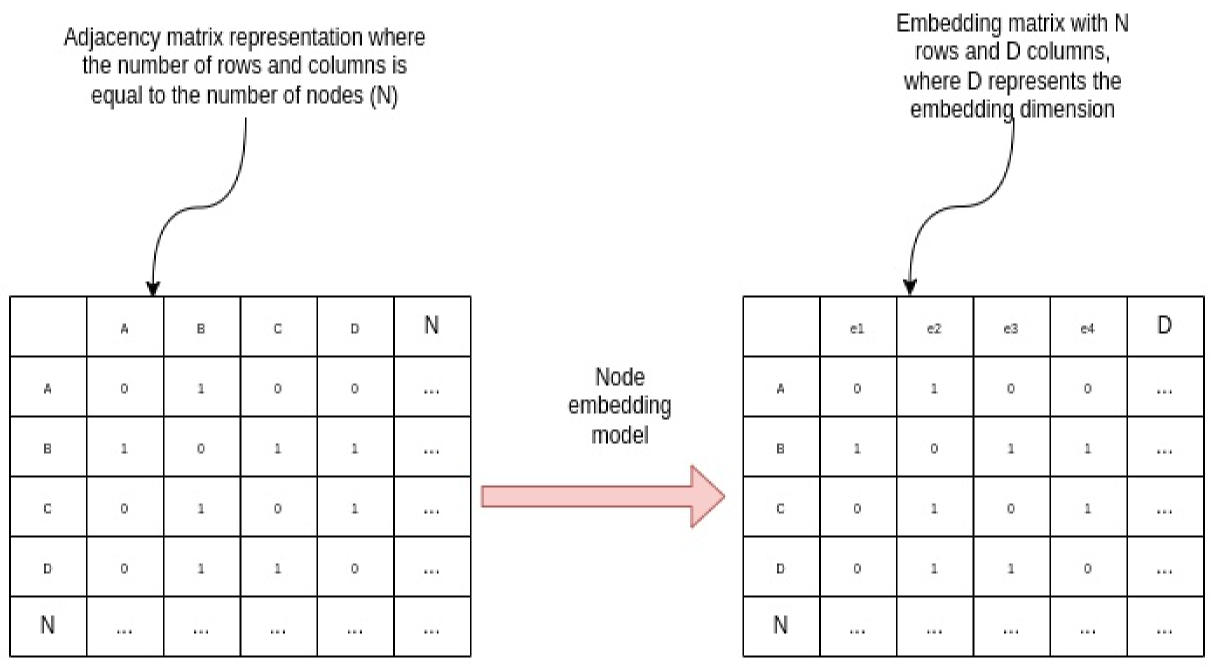
Now imagine you have a graph with a million nodes. In an adjacency matrix, each node would be represented with a row in the matrix that has a million elements. In other words, each node can be described with a vector that has a million elements. Therefore, an adjacency matrix is regarded as a high-dimensional network representation as it grows with the number of nodes in the graph.

Suppose you want to train a machine learning model and somehow use the network structure information as an input feature. Let’s say you use an adjacency matrix as an input. There are a couple of problems with this approach:

- Too many input features
- An ML model is dependant on the size of the graph
- Overfitting

If you add or remove a single node from the graph, the size of the adjacency matrix changes, and your model is no longer functional as there is a different number of input features. Using the adjacency matrix as an input to your model could also cause overfitting. In practice, you often want to embed a node’s local representation to compare nodes with similar neighborhood topology instead of using all relationships between nodes as a feature input. Node embedding techniques try to solve these issues by learning lower-dimensional node representation for any given network. The learned node representations or embeddings should automatically encode the network structure so that the similarity in the embedding space approximates the similarity in the network. A key message is that the node representations are learned instead of manually engineered. The doctors reduced the dimensionality in the BMI example by a manual formula. The node embedding techniques aim to remove painstaking manual feature engineering and provide the best possible node representations by treating the embedding process as a separate machine learning step. The node embedding step is a unsupervised process since it has no training examples to learn from. Node embedding models use techniques based on deep learning and nonlinear dimensionality reduction to achieve this.

Figure 9.3. Adjacency matrix.


Adjacency matrix representation where the number of rows and columns is equal to the number of nodes (N)
Node embedding model
Embedding matrix with N rows and D columns, where D represents the embedding dimension

	A	B	C	D	N
A	0	1	0	0	...
B	1	0	1	1	...
C	0	1	0	1	...
D	0	1	1	0	...
N	...	...	...	...	...

	e1	e2	e3	e4	D
A	0	1	0	0	...
B	1	0	1	1	...
C	0	1	0	1	...
D	0	1	1	0	...
N	...	...	...	...	...

Figure 9.3 visualizes the node embedding process. The node embedding model takes the high-dimensional representation of a graph as an input and outputs a lower-dimensional representation. In the example of a graph with million nodes, each node can be represented with a vector of million elements. Suppose you execute a node embedding model on this graph. With most node embedding models, you can define the embedding dimension. The embedding dimension is the number of elements in the embedding matrix that describe a node. For example, you could set the embedding dimension to be 256. In that case, each node would be described with a vector that contains 256 elements. Reducing the number of elements from a million to 256 is incredibly beneficial as it allows you to efficiently describe the network topology or position of a node in a graph with a lower dimensional vector. These lower dimensional vectors can be used in a downstream machine learning workflow, or they can be used to infer a similarity network using the

nearest neighbour graph algorithm.

9.1 Node embedding models

Node embedding models aim to produce lower dimensional representations of nodes while preserving network structure information.

9.1.1 Homophily versus structural roles approach

However, what does network structure information mean exactly? A common approach is to represent nodes in the embedding space so that neighboring nodes in the graph are close in the embedding space.

Figure 9.4. Homophily approach to node embedding.

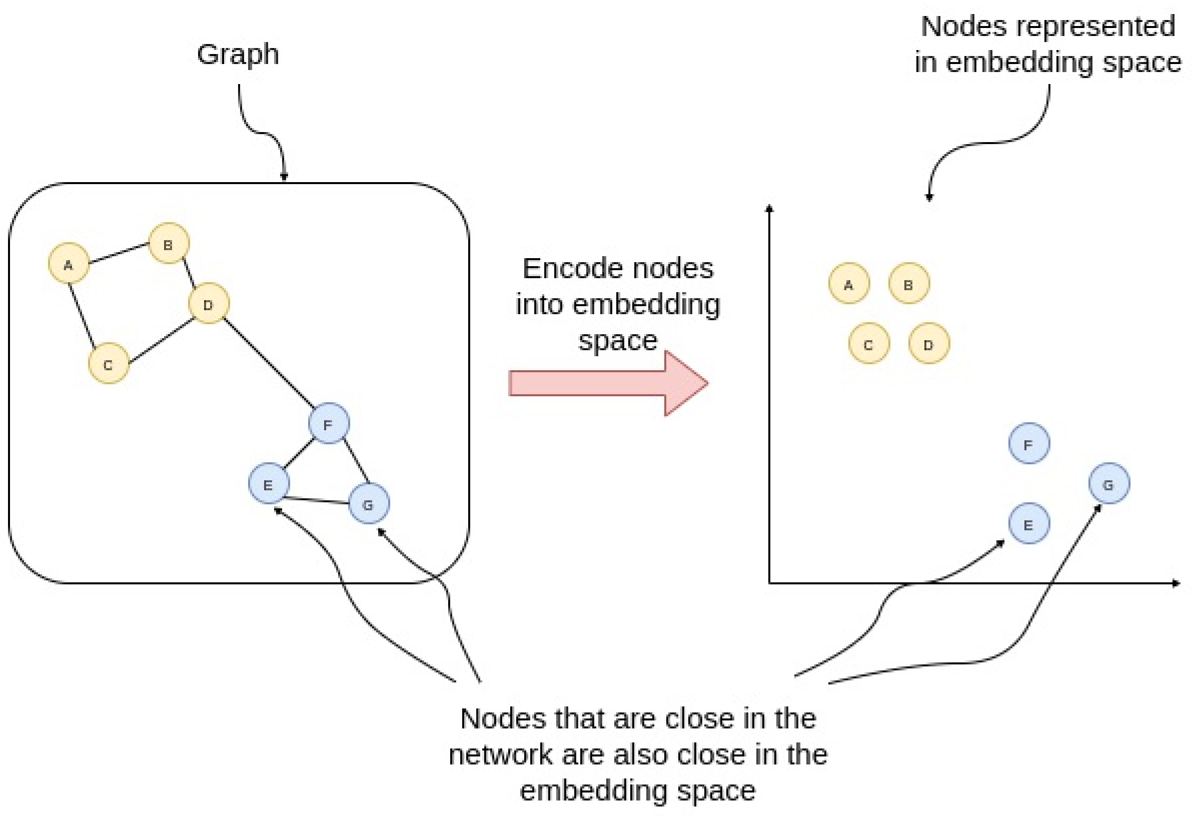
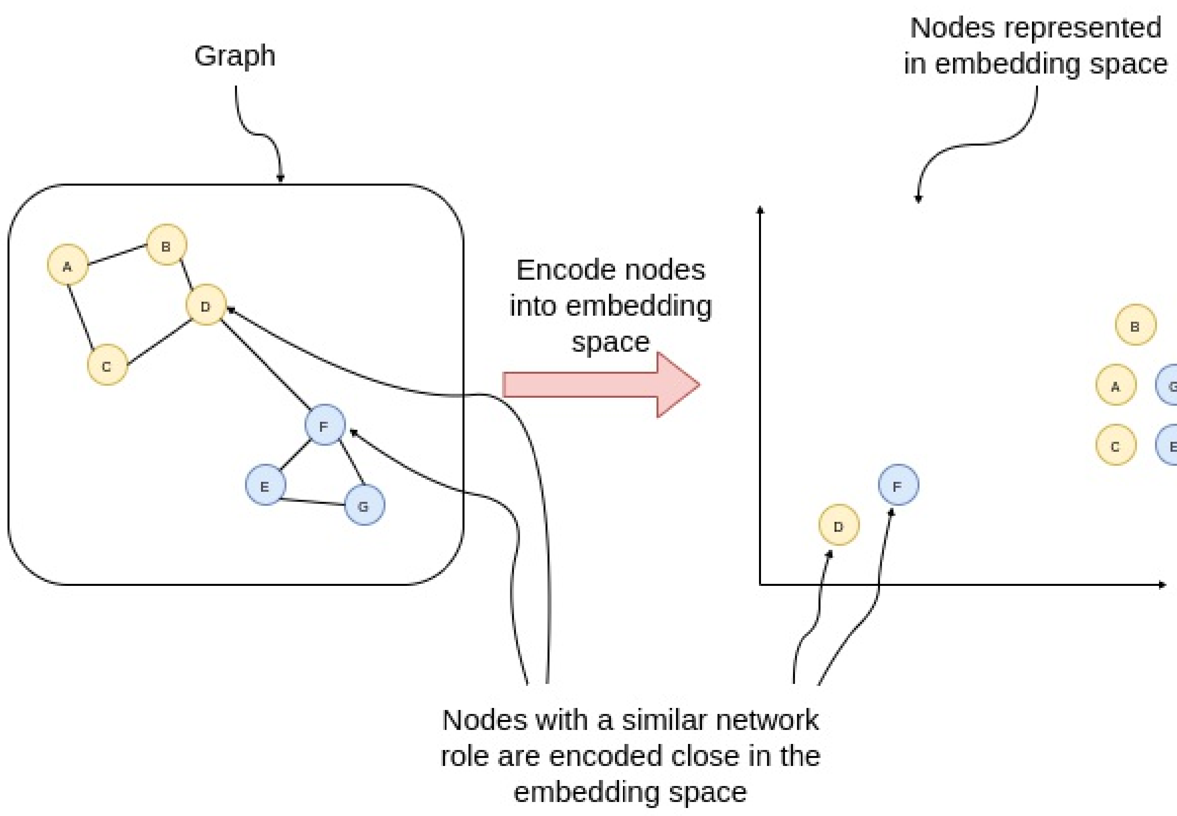

Graph
Nodes represented in embedding space
A
B
C
D
E
F
G
Encode nodes into embedding space
Nodes that are close in the network are also close in the embedding space

Figure 9.4 visualizes the so-called community-based approach to node embeddings. Neighboring nodes in the graph are also close in the embedding space. Therefore, nodes that belong to the same community should be close in the embedding space. This approach is designed under the node *homophily* assumption that connected nodes tend to be similar or have similar labels in a downstream machine learning workflow.

For example, you probably have similar interests as your friends. Suppose you wanted to predict one’s interest. In that case, you could encode their position in the friendship network with a homophily-based node embedding model and train a supervised model based on training examples to predict or recommend one’s interests. If the hypothesis that one has similar interests as their friends is valid, the trained model should perform relatively well. A node embedding algorithm that you could use in this example is the FastRP algorithm [Chen et al., 2019].

Another approach is to encode nodes in the embedding space so that nodes with a similar network role are close in the embedding space.

Figure 9.5. Structural roles approach to node embedding.


Graph
Nodes represented in embedding space
A
B
C
D
E
F
G
Encode nodes into embedding space
B
A
G
C
E
F
D
Nodes with a similar network role are encoded close in the embedding space

You have briefly been introduced with node roles in the previous chapter. Figure 9.5 visualizes the node embedding process, where the nodes are encoded close in the embedding space based on their network *structural roles*. In the Figure 9.5, both nodes D and F act as bridges between the two communities. One can assume that they have similar network roles, and therefore, they are encoded close in the embedding space. You could use the structural role embedding approach to analyze roles of researches in the coauthorship For example, you could use structural role approach to node embedding to analyze roles of researchers in a coauthorship network, or perhaps determine roles of routers on the internet network. For instance, the RolX algorithm [Henderson et al, 2012] is a node embedding algorithm that encodes nodes with a network structural role close in the embedding space.

Which design of the node embedding models you want to use depends on the downstream task you need to complete. Some algorithms like the Node2vec [Grover & Leskovec, 2016] can also produce a combination of the two embedding designs as the output.

9.1.2 Inductive versus transductive embedding models

Some node embedding models have a significant limitation. A typical process of using a node embedding model in a machine learning workflow involves calculating the embeddings and feeding them, for example, into a classification machine learning model. So far, nothing unexpected. But what happens when a new node is introduced into the network? How do you go about calculating the embedding for the new node and inferring its class through the trained classifier?

When dealing with a *transductive* node embedding algorithm, you cannot calculate embeddings for nodes not seen during the initial embedding calculation. You can think that transductive models create a vocabulary during initial computation, where the key of the vocabulary represents a node, and its value represents the embedding. If a node was not seen during the initial computation, it is not present in the vocabulary, and hence, you cannot simply retrieve the embeddings for the new unseen nodes. If you want to calculate the embeddings for the new nodes, you have to calculate the embeddings for the whole graph, meaning all the previously observed nodes as well as the new nodes. Since the embeddings might change for existing nodes, you must also retrain the classification model.

On the other hand, *inductive* node embedding models can calculate embeddings for unseen nodes during the initial computation. For example, you can train a model based on the initial computation of node embeddings. When a new node is introduced, you can calculate the embedding for the new node without re-calculating embeddings for the whole graph. Likewise, you don’t have to re-train the classification model for every new node. Encoding previously unseen nodes is a great advantage when dealing with growing or multiple separate graphs. For instance, you could train a classification model on a single graph and then use it to predict node labels for nodes of different separate graphs. For further reading on inductive models you can read up on the GraphSAGE model [Hamilton et al., 2017].

9.2 Node classification task

Now it is time to start with a practical example. Imagine you are working at

Twitch as a data scientist. Twitch is a streaming platform that makes is to possible for anyone to start streaming their content to the world. In addition, other users can interact with streamers through the chat interface.

Every day, new users join the platform who decide they want to start streaming. Your manager wants you to identify the language of the new streams. Since the platform is worldwide, there are probably around 30 to 50 languages that streamers use. Let’s assume that converting audio to text and running language detection algorithms is not feasible for whatever reason. One of the reasons could be that streamers on Twitch usually play video games, and therefore, audio from video games could distort language detection. What other way could you predict the languages of new streamers? You have information about users who chat in particular streams. One could hypothesize that users mostly chat in a single language. Therefore, if a user chats in two streams, it is likely that both streams are in the same language. For example, if a user is chatting in a Japanese stream and then switches a stream and interacts with the new streamer through chat, the new stream is likely in Japanese. There might be some exceptions with the English language, as, for the most part, many people on the internet have at least a basic understanding of English.

Figure 9.6. Process of predicting the language for new streams.

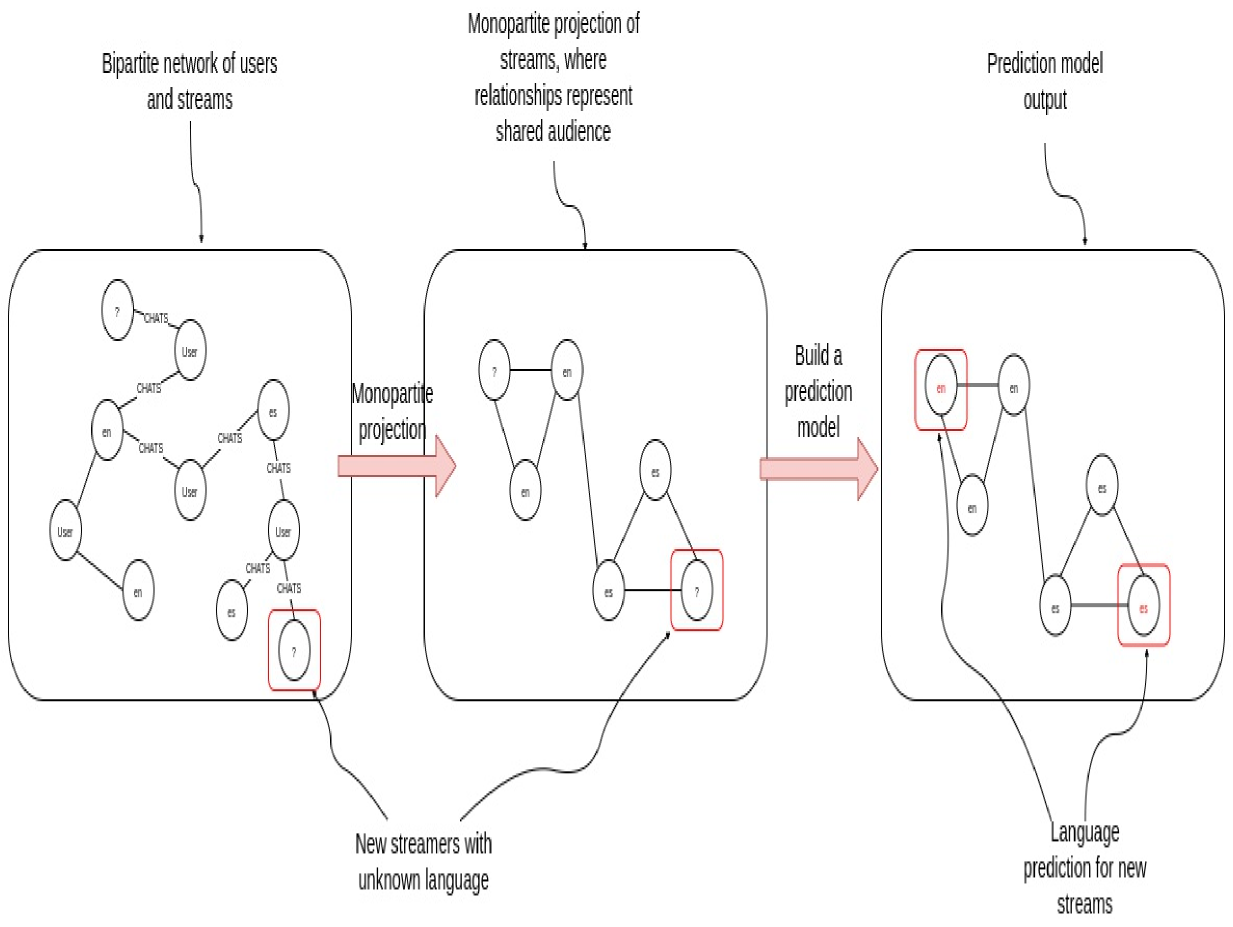

Bipartite network of users and streams
Monopartite projection of streams, where relationships represent shared audience
Prediction model output
Monopartite projection
Build a prediction model
New streamers with unknown language
Language prediction for new streams
CHATS
User
?
en
es

Figure 9.6 visualizes the process of extracting network information to predict the new streamers' languages. Raw data has the structure of a bipartite `(:User)-[:CHATTED]→(:Stream)` graph. The first step in the process is to project a monopartite graph where the nodes represent streams, and the relationships represent the shared audience between them. The schema of the projected monopartite graph can be represented with the following Cypher statement: `(:Stream)-[:SHARED_AUDIENCE]-(:Stream)` The monopartite graph is undirected as if stream A shares the audience with stream B, automatically implying that stream B also shares the audience with stream A. In addition, you can add the count of shared audiences between streamers as a relationship weight. Suppose that extracting raw data and transforming it into a monopartite graph can be done by a data engineer on your team. The data engineer can take a similar approach to what you have learned in Chapter 7 to project a monopartite graph. Your job is now to train a prediction model and evaluate its results.

The idea is to prepare a Jupyter notebook that can be used once a day to predict the languages of new streamers. Remember, if streams are close in the shared audience network, they likely have the same language. Therefore, you will use a node embedding model that uses a homophily-based approach to encoding nodes in the embedding space. One of the most simple and broadly used node embedding models is the node2vec, which you will use in this example. Once the node embeddings are calculated, you will use them for training a random forest classification model based on training examples of streams you already know the language. In the last step of the process, you will evaluate the predictions with a standard classification report and confusion matrix.

To follow the examples, you need to have a Jupyter notebook environment ready and access a Neo4j database. The database should be empty before starting this chapter. You will use the scikit-learn Python library to split the data, train the model, and evaluate the results, so make sure to have it installed. The notebook with all the code in this chapter is also available on GitHub.

9.2.1 Define a connection to Neo4j database

Start by opening a new Jupyter notebook, or download the filled-in notebook from the GitHub link above. You will need to have the following three Python libraries installed to follow the code examples:

- neo4j
- pandas
- scikit-learn

You can install all three libraries with pip or conda package manager.

First, you need to define the connection to the Neo4j database.

Listing 9.1. Define connection to Neo4j.

```
from neo4j import GraphDatabase

url = 'bolt://localhost:7687'
```

```
username = 'neo4j'
password = 'letmein'

driver = GraphDatabase.driver(url, auth=(username, password))
```

Listing 9.1 import the `GraphDatabase` object from the `neo4j` library. To establish the connection with the Neo4j database, you need to fill in and optionally change the credentials. Once the credentials are defined, you pass them to the `driver` method of the `GraphDatabase` object. The driver allows you to spawn sessions in which you can execute arbitrary Cypher statements.

Next, you will define a function that take a Cypher statement as parameter and returns the results as a Pandas Dataframe. Pandas Dataframe is a convenient data structure that can be used to filter, transform, or easily integrate with other Python libraries.

Listing 9.2. Define a function that executes arbitrary Cypher statement and returns a Pandas dataframe.

```
import pandas as pd

def run_query(query):
    with driver.session() as session:
        result = session.run(query)
        return pd.DataFrame([r.values() for r in result], columns
```

9.2.2 Import twitch dataset

Now that the environment is ready, you can circle back to the specified task. Remember, the data engineer on your team was kind enough to extract the information about the streams and chatters and perform the monopartite projection. They prepared two CSV files with relevant information. The first CSV file contains information about nodes in the network.

Table 9.1. Node CSV structure

id	language

129004176	en
50597026	fr
102845970	ko

The node CSV contains information about stream ids and their language. In this example, you have the language information for all the streams so that you will be able to evaluate the classification model accuracy of the test data. It is good practice to define unique constraints on the unique properties of nodes to speed up the import. You will start by defining the unique constraint on the `id` property of the `Stream` nodes.

Listing 9.3. Define constraint on stream.

```
run_query("""
CREATE CONSTRAINT IF NOT EXISTS ON (s:Stream) ASSERT s.id IS UNIQ
""")
```

Since you are working in a Python environment, you need to execute Cypher statements through the `run_query` function as shown in Listing 9.1. The function returns the Pandas Dataframe of the output. Here, however, you are not interested in the result of the Cypher statement, so you don't have to assign the output to a new variable.

Now you can go ahead and import the information about the Twitch streams and their languages. The CSV is available on GitHub, so you can utilize the `LOAD CSV` clause to retrieve and import the CSV information into the database.

Listing 9.4. Import nodes.

```
run_query("""
LOAD CSV WITH HEADERS FROM "https://bit.ly/3JjgKgZ" AS row
MERGE (s:Stream {id: row.id})
SET s.language = row.language
""")
```

The relationship CSV file contains information about shared audiences between streams and their count.

Table 9.2. Relationship CSV structure

source	target	weight
129004176	26490481	524
26490481	213749122	54
129004176	125387632	4591

Relationship CSV contains three columns. The `source` and `target` columns contain the stream ids that have a shared audience, while the `weight` column indicates how many shared users chatted in both streams. You can import the relationship information with the following Cypher statement.

Listing 9.5. Import relationships.

```
run_query("""
USING PERIODIC COMMIT 10000
LOAD CSV WITH HEADERS FROM "https://bit.ly/3S9Uyd8" AS row
MATCH (s:Stream {id:row.source})
MATCH (t:Stream {id:row.target})
MERGE (s)-[r:SHARED_AUDIENCE]->(t)
SET r.weight = toInteger(row.weight)
""")
```

Exercise 9.1

Inspect how many, if any, `Stream` nodes have no incoming or outgoing relationships.

Luckily, there are no *isolated* nodes in the dataset. An isolated node is a node that has no incoming or outgoing relationships. When extracting node features from a dataset, always pay special attention to isolated nodes. For example, if there were some `Stream` nodes without any relationships, that would be a case of missing data. If you waited a few days, hopefully, someone would chat in their stream, and you would create new relationships for that particular stream so that it would not be isolated anymore. On the other hand, isolated `Stream` nodes can have any language. Since most node embedding algorithms encode isolated nodes identically, you would introduce noise to your classification model by including isolated nodes. Therefore, you would want to exclude all isolated nodes from the training and test datasets.

On the other hand, if you are dealing with isolated nodes, and the relationships are not missing, you can include isolated nodes in your workflow. For example, imagine you were to predict a person's net worth based on their network role and position. Suppose a person has no relationships and, therefore, no network influence. In that case, encoding isolated nodes could provide a vital signal to the machine learning model that predicts net worth. Always remember that most node embedding models will encode isolated nodes identically. So if isolated nodes all belong to a single class, then considering them would make sense. However, if isolated nodes belong to various classes, then it would make sense to remove them from the model to remove noise.

Note

Pay special attention to isolated nodes when using node embedding models in your machine learning workflows. Most node embedding models encode isolated nodes identically. There are scenarios where considering isolated nodes makes sense, for example, when all isolated nodes belong to a single class. On the other hand, if isolated nodes belong to multiple classes, you might introduce noise to your classification model, and therefore, it might make sense to ignore isolated nodes altogether.

9.3 Node2vec algorithm

Now that the graph is constructed, it is your job to encode nodes in the embedding space to be able to train the language prediction model based on the network position of the nodes. As mentioned, you will use the node2vec algorithm[Grover & Leskovec, 2016] to achieve this. The node2vec algorithm is transductive and can be fine-tuned to capture either homophily or role-based embeddings.

9.3.1 Word2vec algorithm

The node2vec algorithm is heavily inspired by the word2vec[Mikolov et al., 2013] skip-gram model. Therefore, to properly understand node2vec, you must first understand how the word2vec algorithm works. Word2Vec is a shallow, two-layer neural network that is trained to reconstruct linguistic contexts of words. The objective of the word2vec model is to produce word representation (vectors) given a text corpus. Word representations are positioned in the embedding space such that words that share common contexts in the text corpus are located close to one another in the embedding space. There are two main models used within the context of word2vec.

- Continuous Bag-of-Words (CBOW)
- Skip-gram model

Node2vec is inspired by the skip-gram model, so you will skip the CBOW implementation explanation. The Skip-gram model predicts the context for a given word. The context is defined as the adjacent words to the input term.

Figure 9.7. Process of predicting the language for new streams.

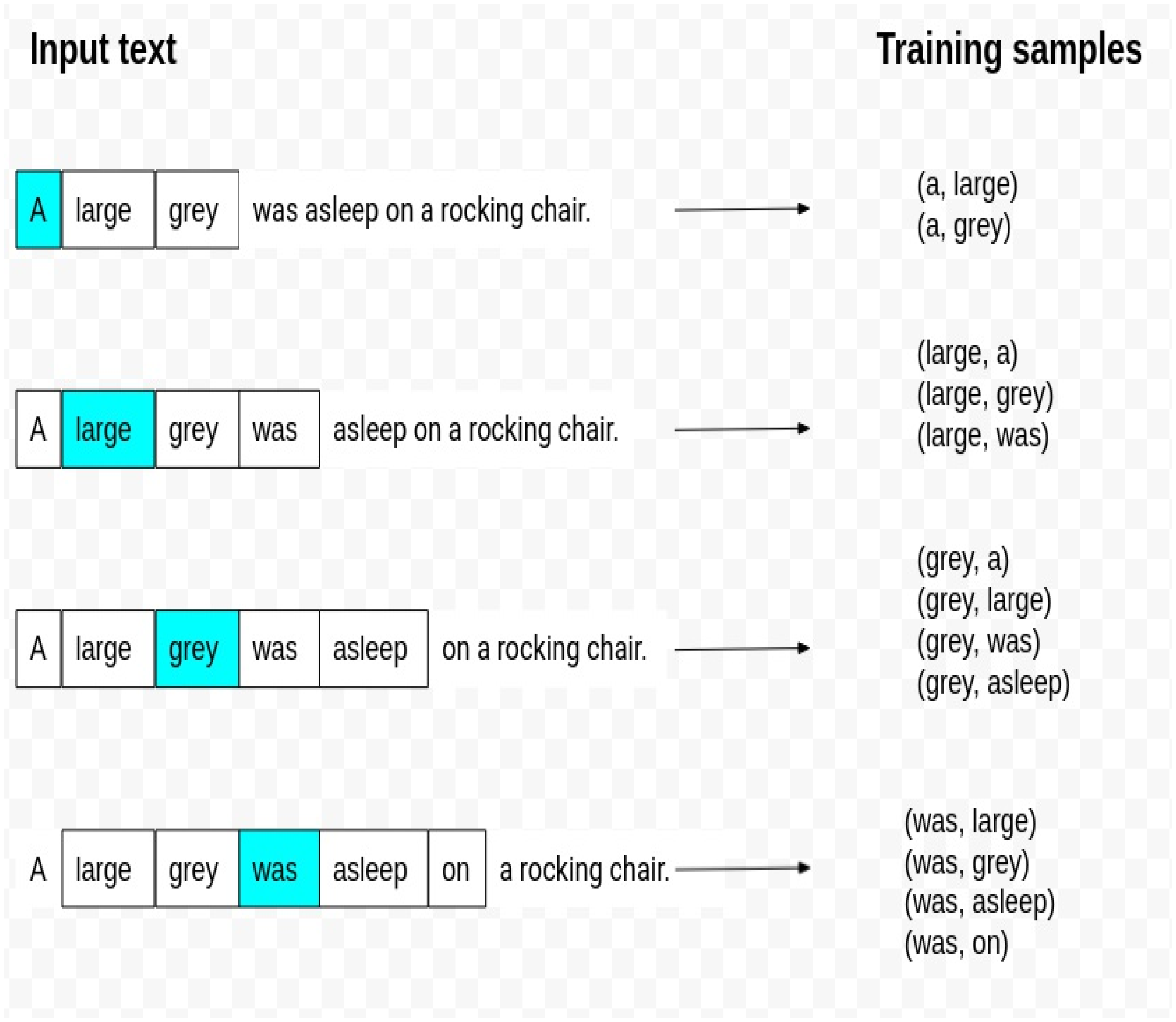

Input text
Training samples
A large grey was asleep on a rocking chair.
(a, large)
(a, grey)
A large grey was asleep on a rocking chair.
(large, a)
(large, grey)
(large, was)
A large grey was asleep on a rocking chair.
(grey, a)
(grey, large)
(grey, was)
(grey, asleep)
A large grey was asleep on a rocking chair.
(was, large)
(was, grey)
(was, asleep)
(was, on)

Figure 9.7 visualized how training pairs of words are collected in a skip-gram model. Remember, the objective of the skip-gram model is to predict context words or words that frequently co-appear with a target word. The algorithm creates training pairs for every word in the text corpus by combining the particular word with its adjacent words. For example, in the third row of Figure 9.7, you can observe that the word "grey" is highlighted and defined as the target word. The algorithm collects training samples by observing its adjacent or neighboring words, representing the context in which the word appears. In this example, two words to the left and the right of the highlighted word are considered when constructing the training pair samples. The maximum distance between words in the context window with the input word in the center is defined as the *window size*.

The training pairs are then feed into shallow two-layer neural network.

Figure 9.8. Word2vec shallow neural network architecture.

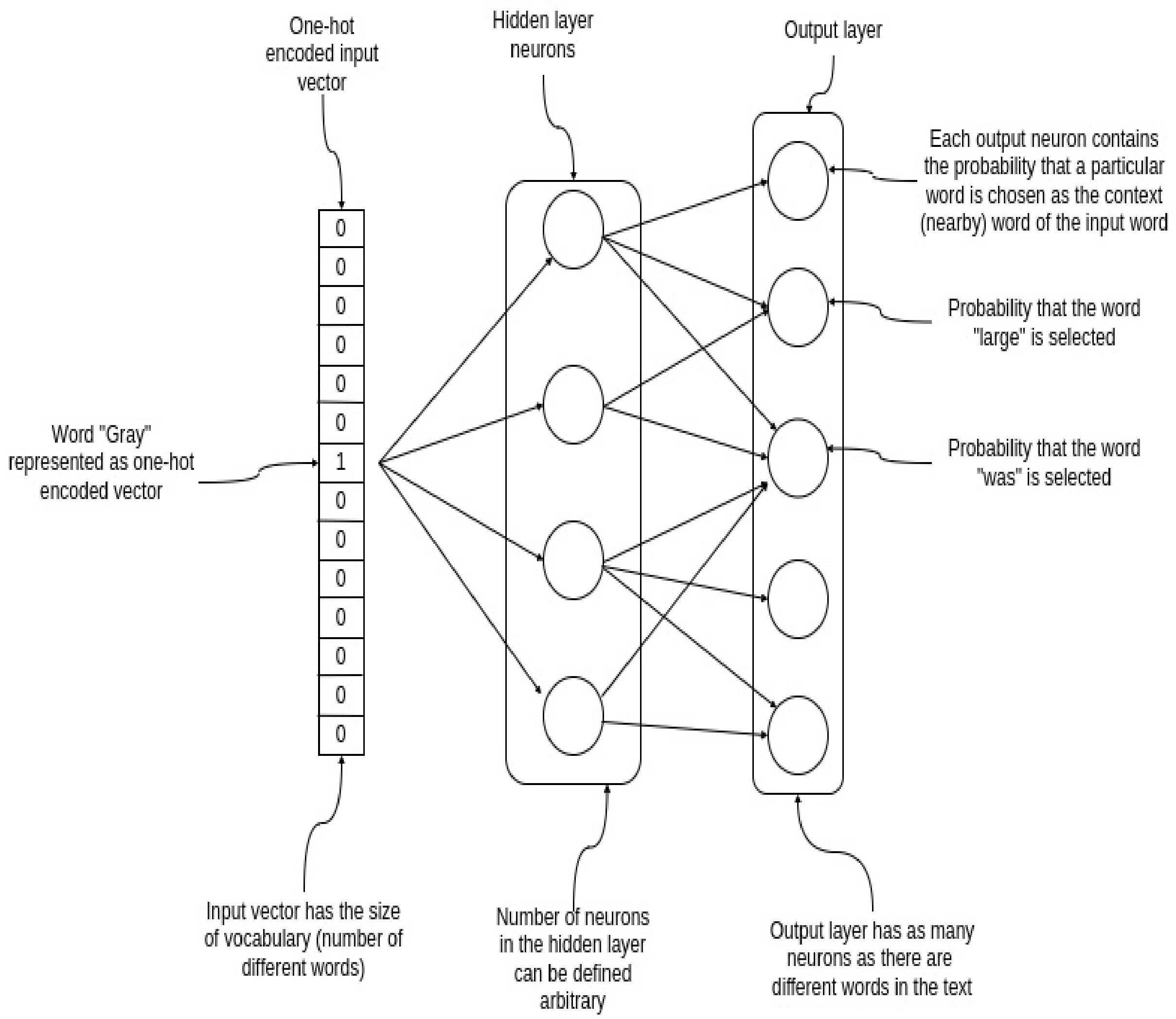
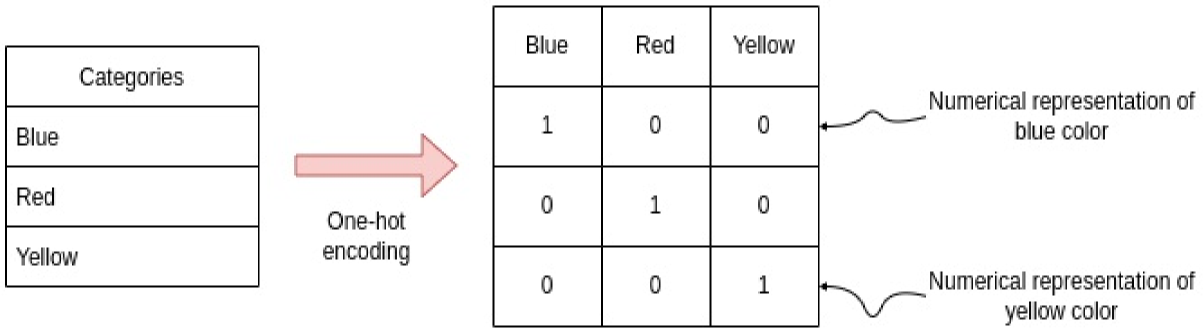

One-hot encoded input vector
Hidden layer neurons
Output layer
Each output neuron contains the probability that a particular word is chosen as the context (nearby) word of the input word
Probability that the word "large" is selected
Probability that the word "was" is selected
Word "Gray" represented as one-hot encoded vector
0
0
0
0
0
0
1
0
0
0
0
0
0
0
Input vector has the size of vocabulary (number of different words)
Number of neurons in the hidden layer can be defined arbitrary
Output layer has as many neurons as there are different words in the text

Figure 9.8 visualizes the word2vec neural network architecture. Don't worry if you have never seen or worked with neural networks. What you need to know if that during the training this neural network, the input is a *one-hot encoded* vector representing the input word, and the output is also a one-hot encoded vector representing the context word.

Figure 9.9. One-hot encoding technique transforms categorical values in to numerical values.


Categories
Blue
Red
Yellow
One-hot encoding
Blue
Red
Yellow
1
0
0
0
1
0
0
0
1
Numerical representation of blue color
Numerical representation of yellow color

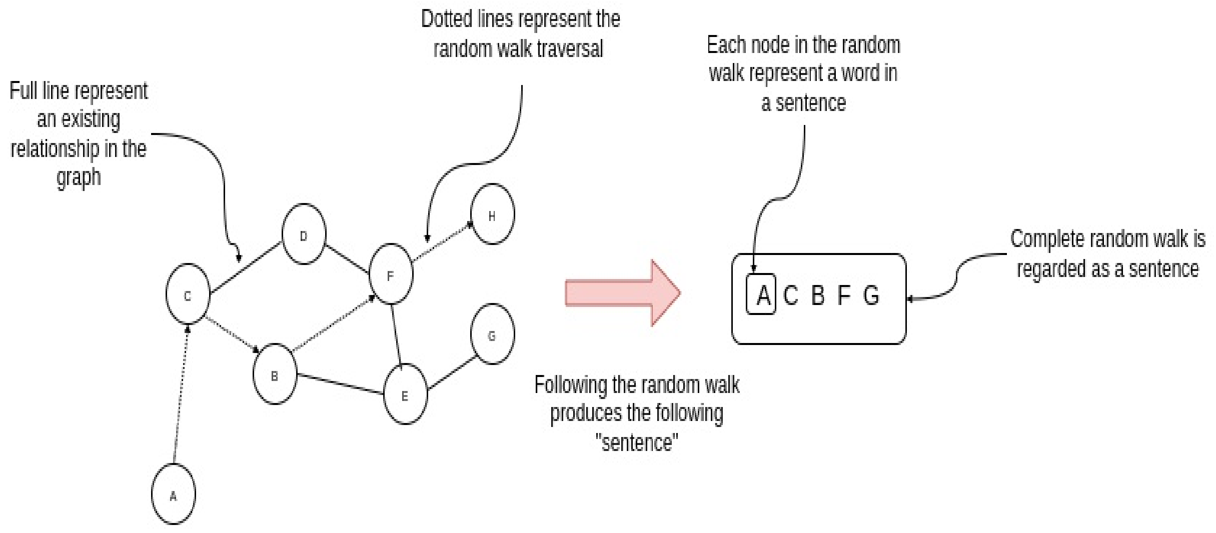
Most machine learning models cannot work directly with categorical values. Therefore, one-hot encoding is commonly applied to convert categorical values into numerical ones. For example, you can see that all the distinct categories in Figure 9.9 transformed into columns through the one-hot encoding process. There are only three distinct categories in Figure 9.9, so there are three columns in the one-hot encoding output. Then, you can see that the category "Blue" is encoded as 1 under the "Blue" column and 0 under all the rest columns. Essentially, the numerical representation of the category "Blue" is [1,0,0]. Likewise, the numerical representation of "Yellow" is [0,0,1]. As you can observe, the one-hot encoded vectors will have a single 1 under the column of the particular category they belong to, while the other elements of the vectors are 0. While this is a pretty straightforward technique, it is trendy as it allows for a simple transformation of categorical values into numerical ones, which can then be fed into machine learning models.

After the training step of the skip-gram model is finished, the neurons in the output layer represent the probability a word will be associated with the input word. Word2vec uses a trick where we aren’t interested in the output vector of the neural network, but rather the goal is to learn the weights of the hidden layer. The weights of the hidden layer are actually the word embedding we are trying to learn. The number of neurons in the hidden layer will determine the *embedding dimension* or the size of the vector representing each word in the vocabulary. Note that the neural network does not consider the offset of the context word, so it does not differentiate between directly adjacent context words to the input and those more distant in the context window or even if the context word precedes or follows the input term. Consequently, the window size parameter has a significant influence on the results of the word embedding. For example, one study [Levy, 2014] finds that larger context window size tends to capture more topic/domain information. In contrast, smaller windows tend to capture more information about the word itself, e.g., what other words are functionally similar.

9.3.2 Random walks

So what does word2vec have to do with node embeddings? Node2vec algorithm uses the skip-gram model under the hood. However, since you are not working with text corpus in a graph, how do you define the training data? The answer is quite clever. Node2vec uses *random walks* to generate a corpus of “sentences” from a given network. A random walk can be interpreted as a drunk person traversing the graph. Of course, you can never be sure of an intoxicated person’s next step, but one thing is certain. A drunk person traversing the graph can only hop onto a neighboring node.

Figure 9.10. Using random walks to produce sentences.

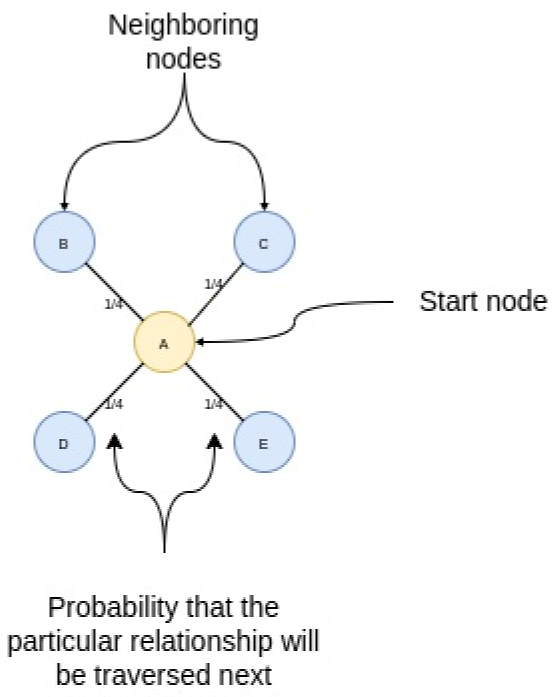

Dotted lines represent the random walk traversal
Full line represent an existing relationship in the graph
Each node in the random walk represent a word in a sentence
Complete random walk is regarded as a sentence
A C B F G
Following the random walk produces the following "sentence"
A
B
C
D
E
F
G
H

The node2vec algorithm uses random walks to produce the sentences, which can be used as input to the word2vec model. In Figure 9.10, the random walk starts at node A and traverses to node H via nodes C, B, and F. The random walk length is decided arbitrarily and can be changed with the *walk length* parameter. Each node in the random walk is treated as a word in the sentence, where the size of the sentence is defined with the walk length parameter.

Random walks start from all the nodes in the graph to make sure to capture all the nodes in the sentences. These sentences are then passed to the word2vec skip-gram model as training examples. That is the whole gist of the node2vec algorithm.

However, the node2vec algorithm implements second-order biased random walks. A step in the first-order random walk only depends on its current state. A step in the first-order random walk only depends on its current state.

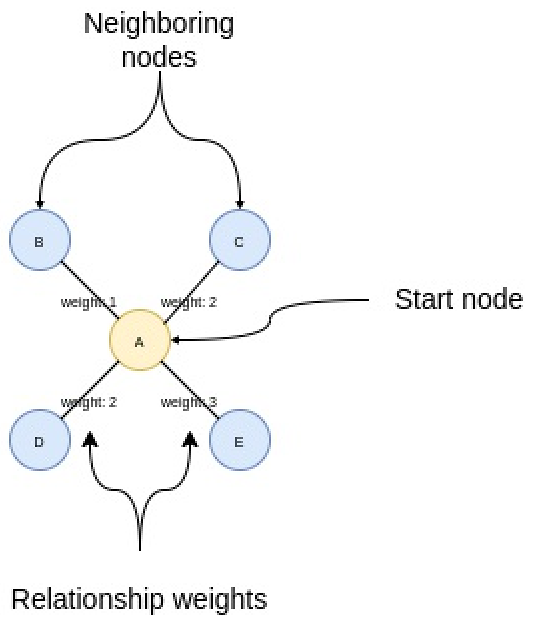
Figure 9.11. First-order random walks.


Neighboring nodes
B
C
1/4
1/4
A
Start node
D
1/4
1/4
E
Probability that the particular relationship will be traversed next

Imagine you have somehow wound up at node A in Figure 9.11. Because the first-order random walk only looks at its current state, the algorithm doesn’t know which node it was at the earlier step. Therefore, the probability of returning to a previous node or any other node is equal. There is no advanced math concept behind the calculation of probability. Node A has four neighbors, so the chance of traversing to any of them is 25% (1/4).

Suppose your graph is weighted, meaning that each relationship has a property that stores its weight. In that case, those weights will be included in the calculation of the traversal probability.

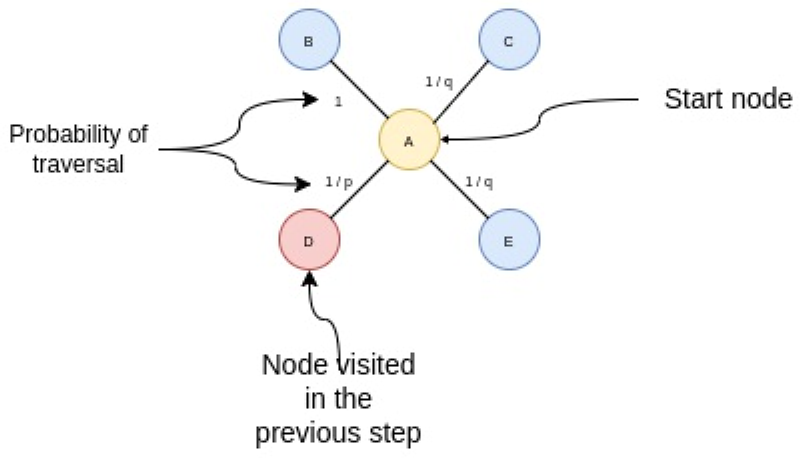
Figure 9.12. First-order random walks.



In a weighted graph, the chance of traversing a particular connection is its weight divided by the sum of all neighboring weights. For example, the probability to traverse from node A to node E in Figure 9.12 is 2 divided by 8 (25%) and the probability to traverse from node A to node D is 37.5%.

On the other hand, second-order walks take into account both the current as well as the previous state. To put it simply, when the algorithm calculates the traversal probabilities, it also considers where it was at the previous step.

Figure 9.13. First-order random walks.



In Figure 9.13, the walk just traversed from node D to node A in the previous step and is now evaluating its next move. The likelihood of backtracking the walk and immediately revisiting a node in the walk is controlled by the *return parameter p*. If the value of return parameter p is low, then the chance of revisiting node D is higher, keeping the random walk closer to the starting node of the walk. Conversely, setting a high value to parameter p ensures lower chances of revisiting node D and avoids 2-hop redundancy in sampling. A higher value of parameter p also encourages moderate graph exploration.

The *inOut parameter q* allows the traversal calculation to differentiate between inward and outward nodes. Setting a high value to parameter q (q > 1) biases the random walk to move towards nodes closer to the node in the previous step. Looking at Figure 9.13, if you set a high value for parameter q, the random walk from node A is biased more towards node B. Such walks obtain a local view of the underlying graph with respect to the starting node in the walk and approximate breadth-first search. In contrast, if the value of q is low (q < 1), the walk is more inclined to visit nodes further away from node D. In Figure 9.13, nodes C and E are further away since they are not neighbors of the node in the previous step. This strategy encourages outward exploration and approximates depth-first search.

Authors of the node2vec algorithm claim that approximating depth-first search will produce more community or homophily-based node embeddings. On the other hand, the breadth-first search strategy for random walks encourages structural role embeddings.

9.3.3 Calculate node2vec embeddings

Now that you have a theoretical understanding of node embeddings and the node2vec algorithm, you will use it in a practical example. As mentioned, your task as a data scientist at Twitch is to predict the languages of new streamers based on shared audiences or chatters between different streams. The graph is already constructed, so you only need to execute the node2vec algorithm and train a classification model. As always, you first have to project an in-memory graph. Relationships represent shared audiences between streams. When stream A shares an audience with stream B, that directly implies that stream B also shares an audience with stream A. Therefore, you can treat the relationships as undirected. Additionally, you know how many users were shared between a pair of streams, which you can represent as a relationship weight.

Execute the following query to project an undirected weighted network of shared audiences between streams.

Listing 9.6. Project the in-memory graph of streams and their shared audience in memory.

```
run_query("""
CALL gds.graph.project("twitch", "Stream",
  {SHARED_AUDIENCE: {orientation: "UNDIRECTED", properties:["weig
""")
```

The Cypher statement in Listing 9.6 projects an in-memory graph named `twitch`. To treat relationships as undirected, you must set the `orientation` parameter value to `UNDIRECTED`. The `properties` parameter of relationships can be used to define the relationship properties to be included in the projection.

Finally, you can go ahead and execute the node2vec algorithm. There are multiple parameters that you could fine-tune to get the best results. However, hyper-parameter optimization is not in the scope of this chapter. You will use the `embeddingDimension` parameter value of eight, which means that each node will be represented with a vector of eight elements. Next, you will define the `inOutFactor` parameter to be 0.5, which encourages more depth-first search walks and produces homophily-based embeddings. In this example, you are not interested in the structural roles of nodes, and you only want to encode how close they are in the graph. All the other parameters will be left at default values.

Execute the following Cypher statement to execute the node2vec algorithm and the results back to the database.

Listing 9.7. Calculate node2vec embeddings and store them to the database.

```
data = run_query("""
CALL gds.beta.node2vec.write('twitch',
  {embeddingDimension:8, relationshipWeightProperty:'weight',
   inOutFactor:0.5, writeProperty:'node2vec'})
""")
```

9.3.4 Evaluate node embeddings

Before you train the language classification model, you will evaluate the embedding results. You will start by examining the cosine and euclidean distance of embeddings of pairs of nodes where a relationship is present. The cosine and euclidean distance distribution can be calculated with Cypher and then visualized with the Seaborn library.

Listing 9.8. Evaluate the cosine and euclidean distance of embeddings of connected nodes

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.figsize"] = [16, 9]

df = run_query("""
MATCH (c1:Stream)-[:SHARED_AUDIENCE]->(c2:Stream)
RETURN gds.similarity.euclideanDistance(
    c1.node2vec, c2.node2vec) AS distance, 'euclidean' as metric
UNION
MATCH (c1:Stream)-[:SHARED_AUDIENCE]->(c2:Stream)
RETURN gds.similarity.cosine(
    c1.node2vec, c2.node2vec) AS distance, 'cosine' as metric
"""
)

sns.displot(
    data=df,
    x="distance",
    col="metric",
    common_bins=False,
    facet_kws=dict(sharex=False),
    height=7,
)
```

The code in Listing 9.8 produces the following visualization.

Figure 9.14. Distribution of cosine and Euclidean distance of embeddings between pairs of nodes where a relationship is present.

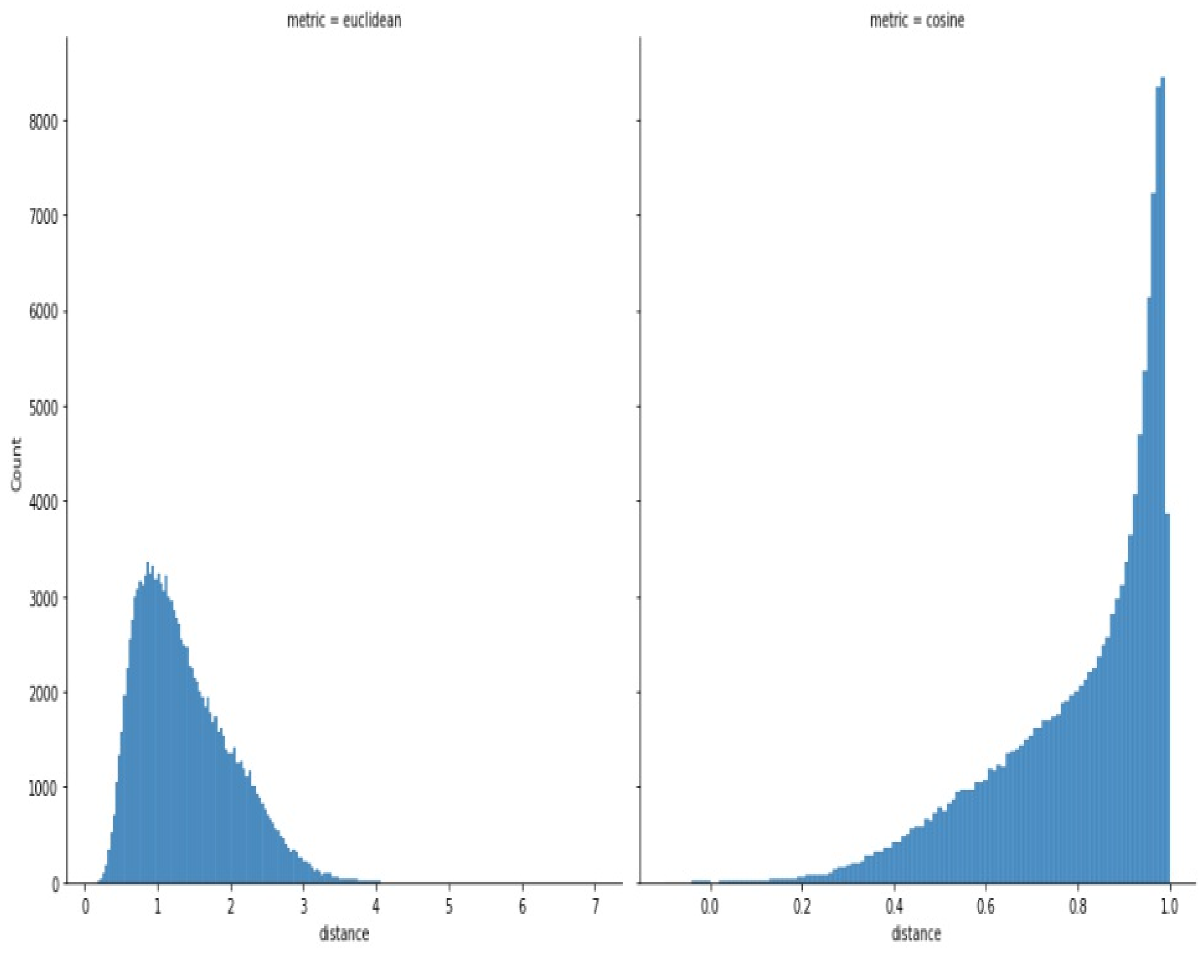


Figure 9.14 visualized the distribution of cosine and Euclidean distance of embeddings between pairs of nodes where a relationship is present. With the Euclidean distance, the lower the value, the more similar or close the nodes in the embedding space. You can observe that the top of the distribution is

slightly below 1. Most of the nodes are very similar based on the Euclidean distance. However, there are some pairs of nodes where the distance is slightly larger. On the other hand, with the cosine similarity, two nodes are very close in the embedding space when the value is close to one. Similarly, most pairs of nodes where the relationship is present have a cosine similarity close to one. So what happens when a pair of nodes have a relationship, but their cosine similarity of embeddings is, for example, less than 0.5?

Using the following code, you can investigate the dependence of cosine similarity between pairs of connected nodes based on their combined degree values.

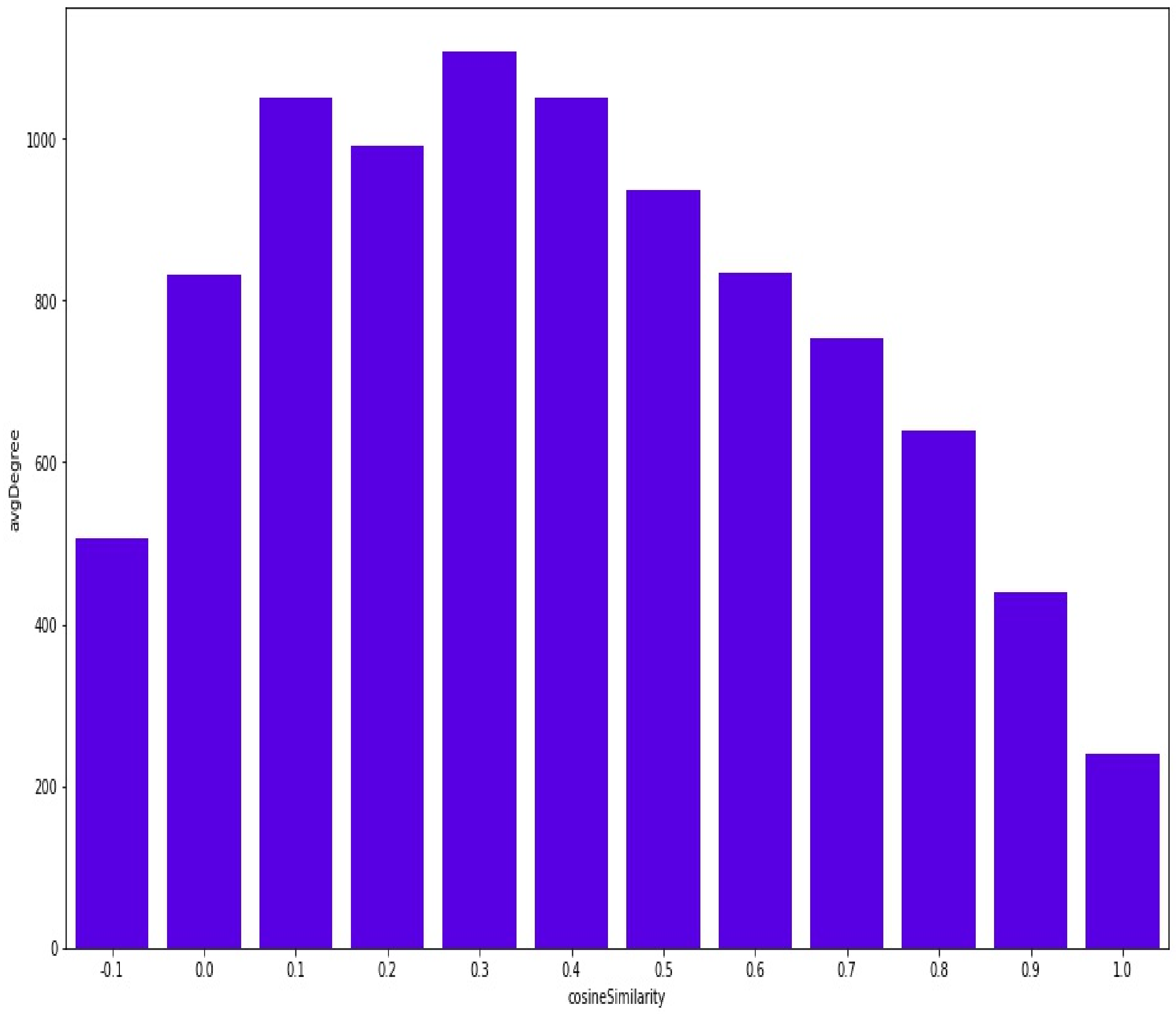
Listing 9.9. Evaluate the dependence of cosine similarity to the combined node degree values.

```
df = run_query("""
MATCH (c1:Stream)-[:SHARED_AUDIENCE]->(c2:Stream)
WITH c1, c2, gds.similarity.cosine(
        c1.node2vec, c2.node2vec) AS cosineSimilarity,
     size((c1)-[:SHARED_AUDIENCE]-()) AS degree1,
     size((c2)-[:SHARED_AUDIENCE]-()) AS degree2
RETURN round(cosineSimilarity,1) AS cosineSimilarity,
       avg(degree1 + degree2) AS avgDegree
ORDER BY cosineSimilarity
"""
)

sns.barplot(data=df, x="cosineSimilarity", y="avgDegree", color="
```

The code in Listing 9.9 produces the following visualization.

Figure 9.15. Distribution of average cosine similarity of connected nodes based on the combined node degree values.



In Figure 9.15, you can clearly see that the more connections a node has, it is, on average, less similar to its neighbors. That makes sense in a way. Imagine if you only have one friend, you can be almost identical to them. However, when you have 100 friends, you can’t be identical to all of them. You can pick some attributes from each friend that you share, but it is practically impossible to be identical to all of them unless they are also all identical.

You have also specified the relationship weight to calculate the node2vec embeddings. The higher the relationship weight, the more biased the random walk is to traverse it. You can examine how the cosine similarity of connected nodes is dependent on the relationship weight with the following code.

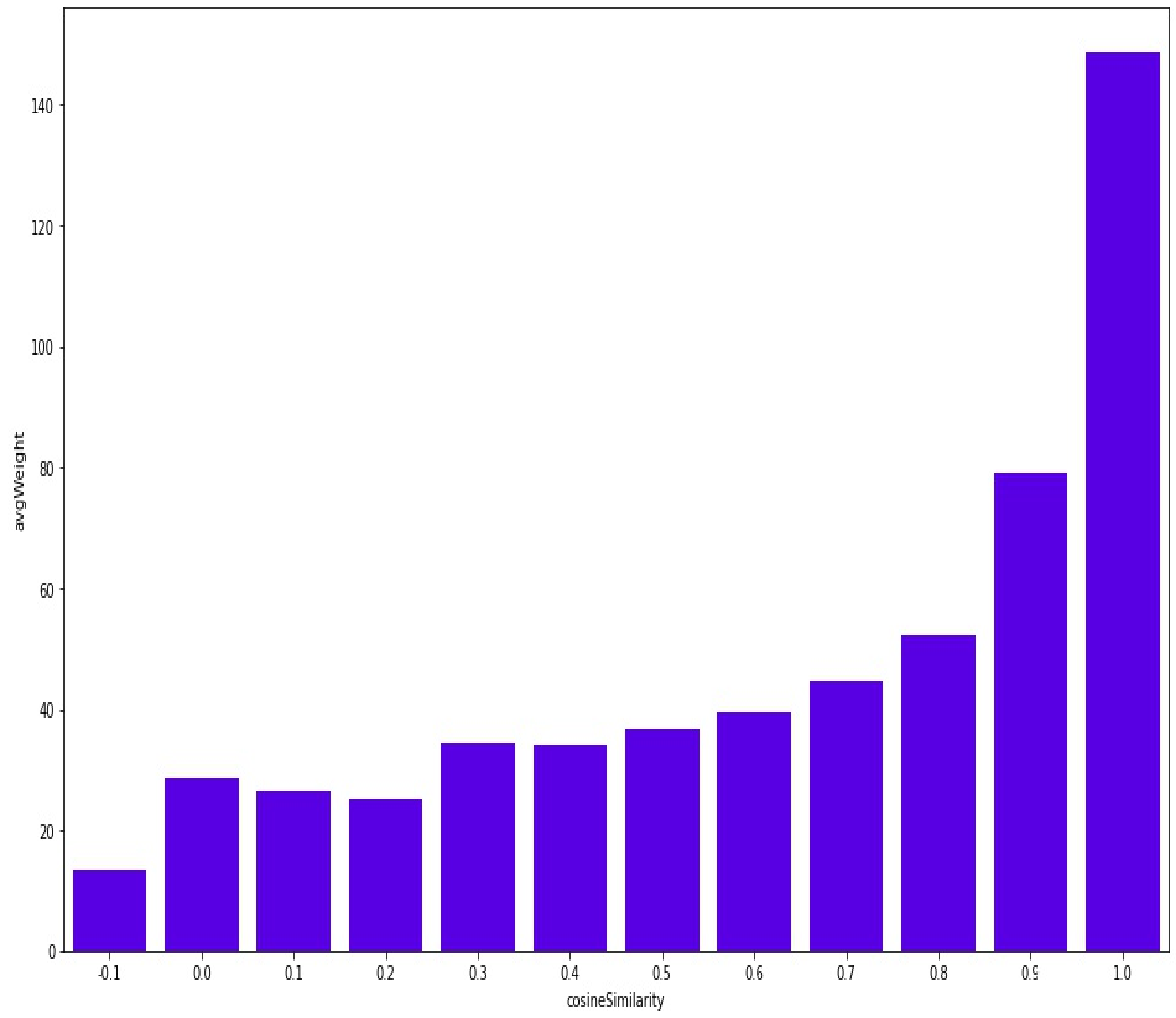
Listing 9.10. Evaluate the dependence of cosine similarity of connected nodes to the relationship weight.

```
df = run_query("""
MATCH (c1:Stream)-[r:SHARED_AUDIENCE]->(c2:Stream)
WITH c1, c2, gds.similarity.cosine(
     c1.node2vec, c2.node2vec) AS cosineSimilarity,
     r.weight AS weight
RETURN round(cosineSimilarity,1) AS cosineSimilarity,
       avg(weight) AS avgWeight
ORDER BY cosineSimilarity
"""
)

sns.barplot(data=df, x="cosineSimilarity", y="avgWeight", color="
```

The code in Listing 9.10 produces the following visualization.

Figure 9.16. Distribution of average cosine similarity of connected nodes based on the combined node degree values.



Again, you can distinctly observe the dependence of the cosine similarity of connected nodes to the relationship weight. The higher the relationship weight, the more likely the random walk is to traverse it. Consequently, the more often a pair of nodes appear closely in the random walk, the more likely their embeddings will be more similar. When a relationship weight is lower, the random walk is biased not to traverse it. Therefore, you can observe some examples where the embeddings are not similar at all, even when there is a relationship between a pair of nodes. One would assume that the pair of nodes are not connected when the cosine distance of their embeddings is close to zero. However, it might simply be the case that the random walk is biased in a way never to traverse the relationship between the two nodes.

9.3.5 Train a classification model

In the final section of this chapter, you will train a classification model to predict the languages of new streamers. First, you must retrieve the relevant data from the database and make the required preprocessing.

Listing 9.11. Retrieve and preprocess relevant data for classification training.

```
data = run_query("""
MATCH (s:Stream)
RETURN s.id AS streamId, s.language AS language, s.node2vec AS em
"""
)
#1
data['output'] = pd.factorize(data['language'])[0]
```

The code in Listing 9.11 begins by retrieving the data from the database. A simple Cypher statement returns stream id, their language, and node embeddings. Since the languages are represented as strings, you need to map or encode them as integers. You can easily encode categorical values such as languages to integers with the `pd.factorize` method.

After this step, the Dataframe should have the following structure.

Table 9.3. Pandas DataFrame structure

streamId	language	embedding	output
129004176	en	[-0.952458918094635, …]	0
50597026	fr	[-0.25458356738090515, …]	1
102845970	ko	[-1.3528306484222412,	2

		…]		

In Table 9.3, you can observe that the `pd.factorize` method encoded the English language under 0. The French language is mapped to 1 and so on.

The `embedding` column contains vectors or lists representing each data point. So, the input to the classification model will be the `embedding` model, and you will train it to predict the integer under the `output` column. In this example, you will use the random forest classifier from the Scikit-learn library. As with all machine learning training, you have to split your data into training and test sets. You will use the `train_test_split` to produce the train and test portions of the dataset.

Execute the following code to train a random forest classification model to predict languages of new streams.

Listing 9.12. Split the dataset and train the random forest model classifier based on the training portion of the dataset.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X = data['embedding'].to_list()
y = data['output'].to_list()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_si
  random_state=0)

rfc = RandomForestClassifier()
rfc.fit(X_train, y_train)
```

9.3.6 Evaluate predictions

The last thing you will do in this chapter is to evaluate the model on the test data. You will begin by examining the *classification report*. A classification report is used to measure the quality of predictions from a machine learning model.

Execute the following code to produce the classification report.

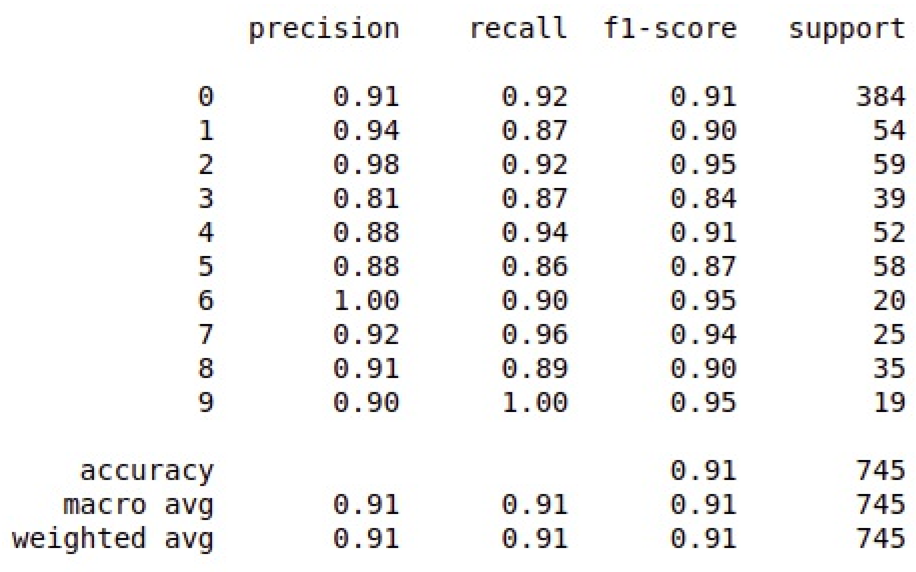
Listing 9.13. Produce the classification report.

```
from sklearn.metrics import classification_report

y_pred = rfc.predict(X_test)
print(classification_report(y_test,y_pred))
```

The code in Listing 9.13 produces the following report.

Figure 9.17. Classification report.



```
              precision    recall  f1-score   support

           0       0.91      0.92      0.91       384
           1       0.94      0.87      0.90        54
           2       0.98      0.92      0.95        59
           3       0.81      0.87      0.84        39
           4       0.88      0.94      0.91        52
           5       0.88      0.86      0.87        58
           6       1.00      0.90      0.95        20
           7       0.92      0.96      0.94        25
           8       0.91      0.89      0.90        35
           9       0.90      1.00      0.95        19

    accuracy                           0.91       745
   macro avg       0.91      0.91      0.91       745
weighted avg       0.91      0.91      0.91       745
```

By looking at Figure 9.17, you can observe that you are dealing with an unbalanced dataset, as there are 384 test data points for the English language and only 54 examples of french streams. Additionally, the language mapped under number nine is Italian and has only 19 test data points. When dealing with unbalanced datasets, it makes sense to examine the F1 score. Both the F1 score and the weighted F1 score are 0.91, which is a great result. The hypothesis that chatters usually chat in streams that share the same language is valid.

Lastly, you will produce the *confusion matrix*. The confusion matrix can help you evaluate actual versus predicted classes of data points. Execute the following code to visualize the confusion matrix.

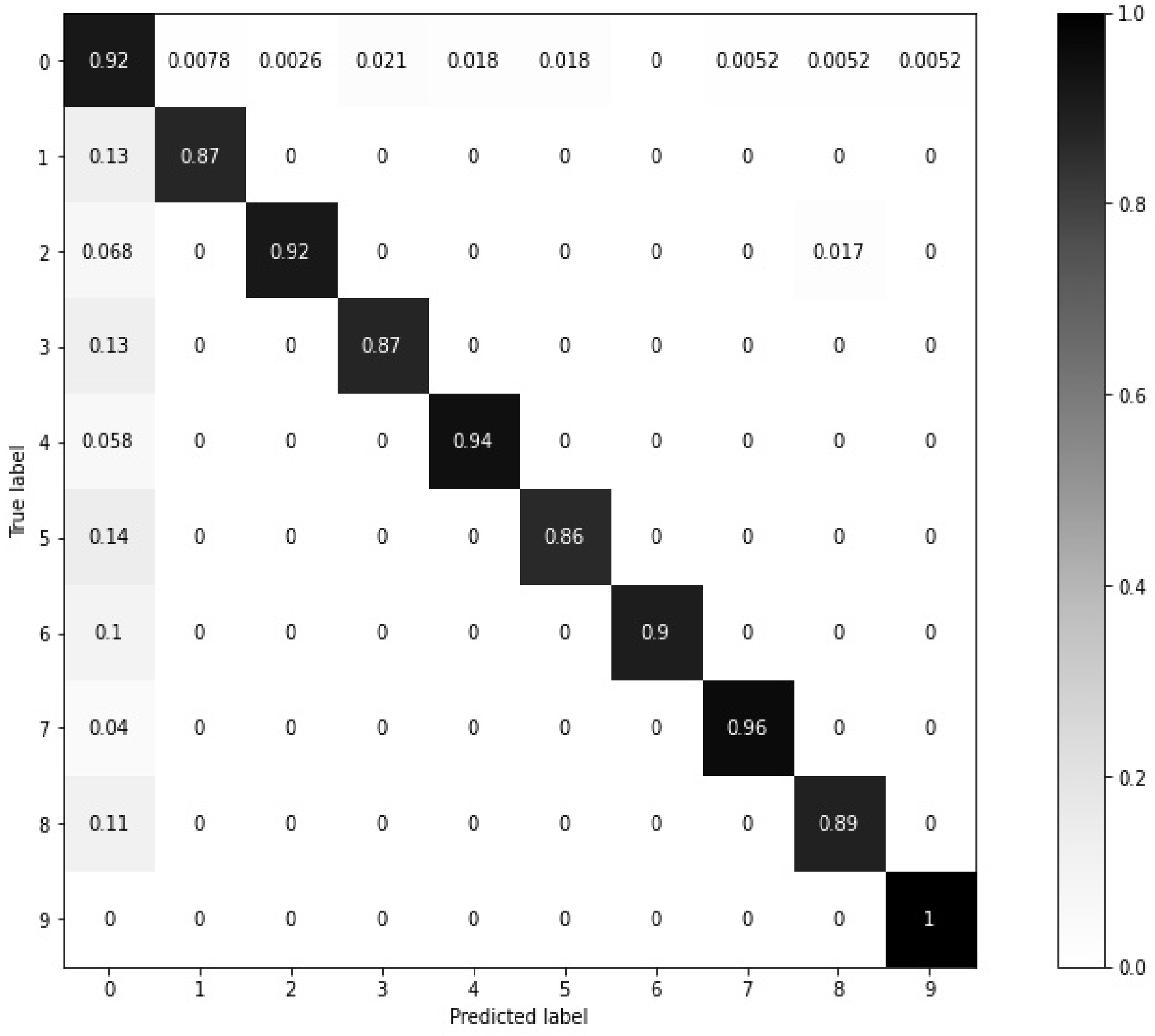
Listing 9.14. Calculate the tweet count and retweet ratio for each user.

```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(y_test, y_pred,
  normalize="true", cmap="Greys")
```

The code in Listing 9.14 produces the following visualization.

Figure 9.18. Confusion matrix.



Remember, the English language is mapped to number 0. You can observe that the model only misclassified between English and other languages by examining the confusion matrix in Figure 9.18. For example, the model never wrongly classified Korean as Portugal language. This makes sense as English is the language of the internet, and so everybody can speak at least their native language and a bit of English.

Exercise 9.2

Try out various configurations of the node2vec algorithm and observe how it affects the cosine distance between embeddings of connected embeddings and the accuracy of the classification model. You can remove the relationship weight parameter to observe how the unweighted variant of the node2vec algorithm behaves or fine-tune `embeddingDimension`, `inOutFactor`, and `returnFactor` parameters. Check out the official documentation for the complete list of node2vec parameters.

Congratulations, you have successfuly trained your first node classification model based on node2vec embeddings.

9.4 Summary

- Node embedding models use a dimensionality reduction technique to produce node representations of arbitrary sizes
- Node embedding models can encode nodes based on their structural roles in the network or can follow a more homophily-based design
- Some of the node embedding models are transductive, which means that they cannot produce embeddings for nodes not seen during the training
- Node2vec algorithm is inspired by the word2vec skip-gram model
- Node2vec algorithm uses random walks to produce sentences, which are then feed into the skip-gram model
- Second-order random walks consider the previous step of the random walk when calculating the next traversal possibilities
- Node2vec can be fine-tuned to produced embeddings based on node structural roles or homophily
- The embedding dimension parameter defines the size of the vector that represents nodes
- Node classification is task of prediction a property or label of a node based on its network features

9.5 References

[Chen et al., 2019] Chen, H., Sultan, S., Tian, Y., Chen, M., & Skiena, S.. (2019). Fast and Accurate Network Embeddings via Very Sparse Random Projection.

[Henderson et al., 2012] Henderson, K., Gallagher, B., Eliassi-Rad, T., Tong, H., Basu, S., Akoglu, L., Koutra, D., Faloutsos, C., & Li, L. (2012). RolX: Structural Role Extraction & Mining in Large Graphs. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1231–1239). Association for Computing Machinery.

[Grover & Leskovec, 2016] Grover, A., & Leskovec, J.. (2016). node2vec: Scalable Feature Learning for Networks.

[Hamilton et al., 2017] William L. Hamilton, Rex Ying, & Jure Leskovec (2017). Inductive Representation Learning on Large Graphs. CoRR, abs/1706.02216.

[Mikolov et al., 2013] Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean: Efficient Estimation of Word Representations in Vector Space, 2013, arXiv:1301.3781

[Levy, 2014] Levy, Y. (2014). Dependency-Based Word Embeddings. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) (pp. 302–308). Association for Computational Linguistics.

9.6 Solutions to exercises

The solution to Exercise 9.1 is the following:

Listing 9.15. Count the nodes with no incoming or outgoing relationships.

```
MATCH (n:Stream)
WHERE NOT EXISTS {(n)--()}
RETURN count(*) AS result
```

10 Link prediction

This chapter covers

- Covering link prediction workflow
- Introducing link prediction dataset split
- Constructing link prediction features based on node pairs
- Training and evaluating a supervised link prediction classification model

Most real-world networks are dynamic and evolve through time. Take, for example, a friendship network of people. People’s friends change over time. They might meet new people or cease to associate with others. You might assume that new connections are forming randomly in a friendship network. However, it turns out that most real-world networks have a profound organizing principle. The studies around link prediction are focused on identifying and understanding various network-evolving mechanisms and applying them to predict future links.

Figure 10.1. Link prediction.

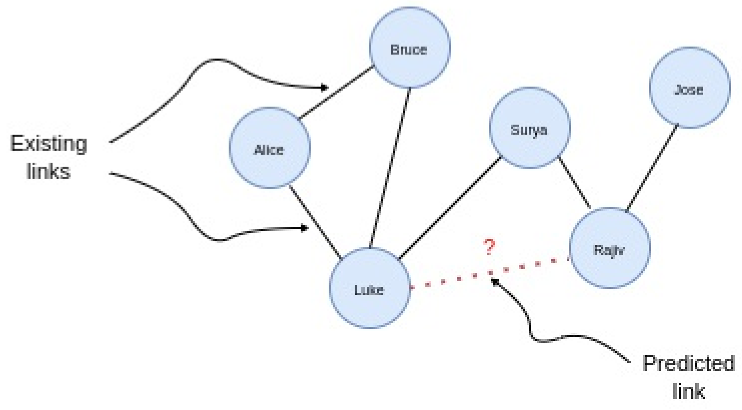
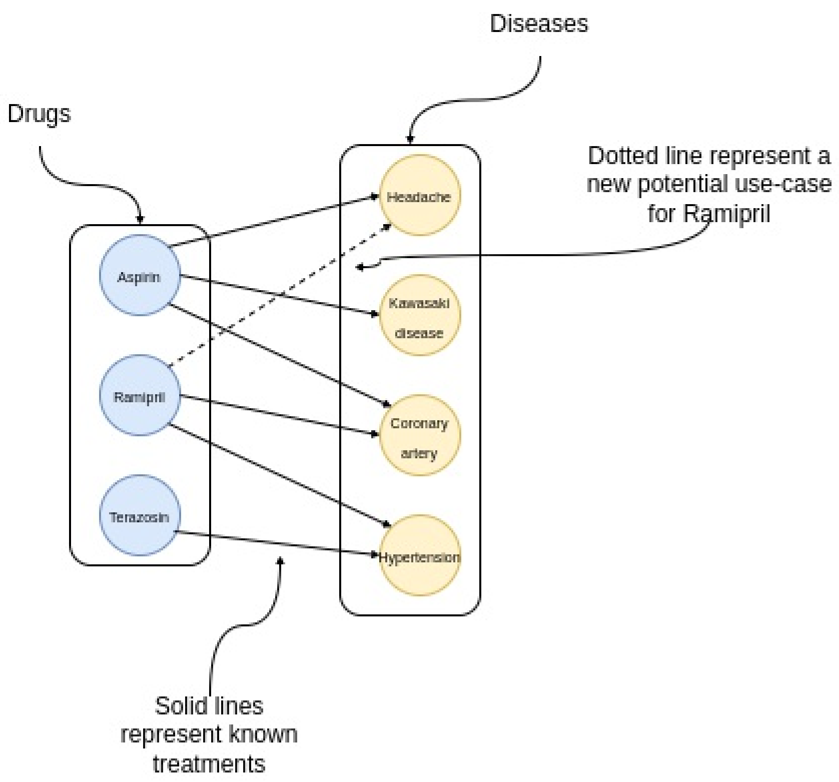


Figure 10.1 visualizes a small network of people, where the relationships represent friendships. Solid lines represent existing connections. As mentioned, friendship networks evolve over time, and people form new connections. Intuitively, you might assume that Luke and Rajiv in Figure

10.1 are more likely to form a future connection, visualized with a dotted line, than Jose and Alicia due to being closer in the network. Unsurprisingly, the number of common friends between a pair of individuals is a good indicator of whether they are likely to meet in the future. Simply put, the closer two individuals are in the network, the higher the probability of forming future links. Predicting future links within a network is one of the main objectives of the link prediction field. Accurately predicting future links can be used in recommender systems, or it can be used to better understand the particular network organizing principles.

On the other hand, there are networks that do not necessarily evolve over time, but we have a limited understanding of their connections. One such example is a biological network of drugs and diseases. A drug often has a narrow variety of diseases it can treat. A clinical trial must be conducted to determine whether a drug can treat any new disease. However, clinical trials are very costly. The other problem with drug repurposing is that there is a vast combination of drugs and diseases on which one could conduct clinical trials. Link prediction techniques can be used to identify missing links in the network. The process of predicting missing links in the network can be thought as *link completion*.

Figure 10.2. Link completion.


Diseases
Drugs
Dotted line represent a new potential use-case for Ramipril
Headache
Aspirin
Kawasaki disease
Ramipril
Coronary artery
Terazosin
Hypertension
Solid lines represent known treatments

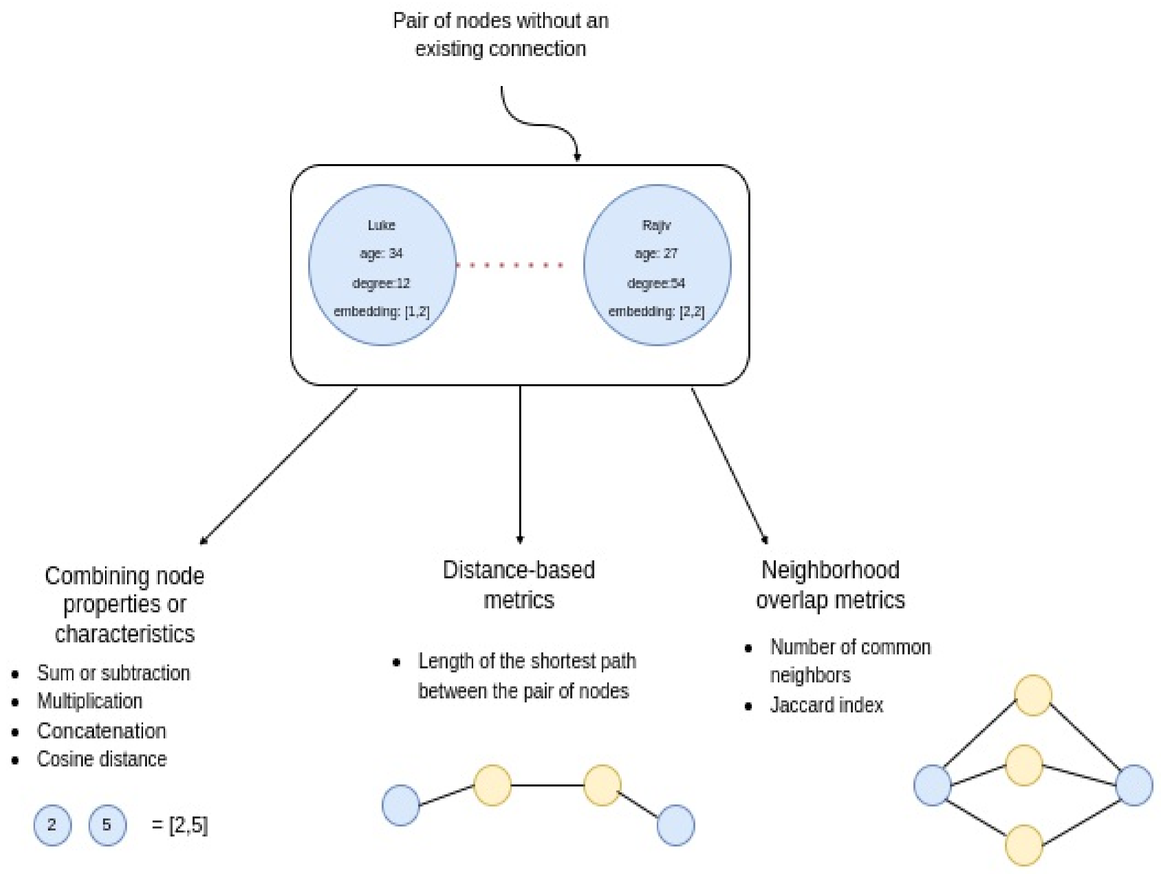
Drug repurposing can be thought of as a link completion process. The network consists of drugs and diseases as visualized in Figure 10.2. The solid lines indicate for which diseases a drug can be used. For example, Aspirin can be used to treat headaches, Kawasaki disease, and coronary artery disease. On the other hand, Ramipril is known to treat coronary artery disease and hypertension. Since both Aspirin and Ramipril can be used to treat coronary artery disease, it might make sense to explore if Ramipril can treat other diseases that Aspirin can. Again, you are simply looking at the number of common neighbors between drugs to base your predictions. In this example, one might conduct a clinical trial to evaluate if Ramipril could be used to treat a headache. Note that this is a simplified version of a drug repurposing scenario. In the real world, much more information about human genes, pathways, and other biological processes is considered.

10.1 Link prediction workflow

As mentioned, most networks follow various organizing principles, intentionally or unintentionally, that you can use in your analysis to predict the probability of a new link between a pair of nodes. For example, in a social network you might assume that people are more likely to become friends if they are of similar age. However, only looking at personal characteristics might be unsatisfactory as this approach might lose a lot of information about a relationship between two persons. Two people might be of similar age, but they don’t mingle in the same social groups, so the probability of them becoming friends is lower. On the other hand, if two people have a lot of common friends, it is more likely that they might meet and become friends. It also turns out that if a person has a lot of friends, they are more likely to form new connections than if they have fewer friends.

When predicting whether a new connection will be established, you are never examining a single node in isolation but rather a pair of nodes in the graph. Therefore, the crucial step in link prediction is to design features that encode a pair of nodes.

Figure 10.3. Encoding pairs of nodes.


Pair of nodes without an existing connection
Luke
age: 34
degree:12
embedding: [1,2]
Rajiv
age: 27
degree:54
embedding: [2,2]
Combining node properties or characteristics
Sum or subtraction
Multiplication
Concatenation
Cosine distance
2
5
= [2,5]
Distance-based metrics
Length of the shortest path between the pair of nodes
Neighborhood overlap metrics
Number of common neighbors
Jaccard index

There are various approaches you could take to encode a pair of nodes. I have grouped them into three categories as presented in Figure 10.3. First of all, you can combine any node properties. For instance, I have used the age difference in the social network example. You could also take the product of node degrees or the cosine similarity of node embeddings. However, you can also simply concatenate embeddings or any other property if that might perform better. Distance-based metrics are another group of features you could use in link prediction. A typical representative of this group is the length of the shortest path between the pair of nodes. Essentially, you calculate the number of hops you need to traverse to get from one node to another and use that as a feature. The underlying idea is that the closer the nodes are in the network, the more likely a link will form between them in the future. The third group of features focuses on evaluating the neighborhood overlap between two nodes. For example, the higher the number of common friends, the greater the likelihood that the pair will meet somewhere in the future. You could also calculate the Jaccard similarity index, which would simply be a normalized version of the common friends count. There are also other metrics that encode the overlap of the local neighborhood that you will learn later in the chapter.

The three categories and their examples presented in Figure 10.3 do not offer an exhaustive list of possible features. There are other ways of combining metrics into link prediction features. However, the groups presented in Figure 10.3 cover most of the methods to generate features you will encounter in your link prediction tasks.

A simple way of predicting links could be that you would calculate the link prediction metrics between pairs of nodes and simply take the arbitrary top number of them as links probable to happen in the future. However, you could also decide to train the classification model using those link prediction metrics. An added value of training a classification model is that it can learn to identify patterns that you might miss when taking an unsupervised approach. In this chapter, you will learn how to calculate various link prediction metrics and train and evaluate a classification model based on those metrics.

Now imagine you are still working at Twitch as a data scientist. You have been tasked with finding ways to improve channel recommendations. So far, you are already using the recommendation system based on shared audiences between different channels. If there is a significant audience overlap between two channels, you can use that information to provide recommendations to users. As a user, you will see those recommendations in the "Users of this channel also watch" section. What could you do to try to improve these recommendations? One idea is that you predict which channels will share their audiences in the future. Then you could provide these predictions as recommendations to users. This approach might improve the overall recommendations as you would recommend channels with existing audience overlap and also channels with a high probability of future audience overlap.

Figure 10.4. Link prediction flow.

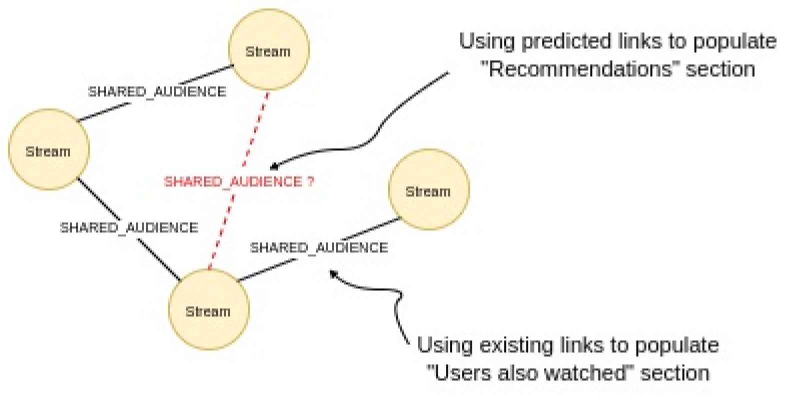
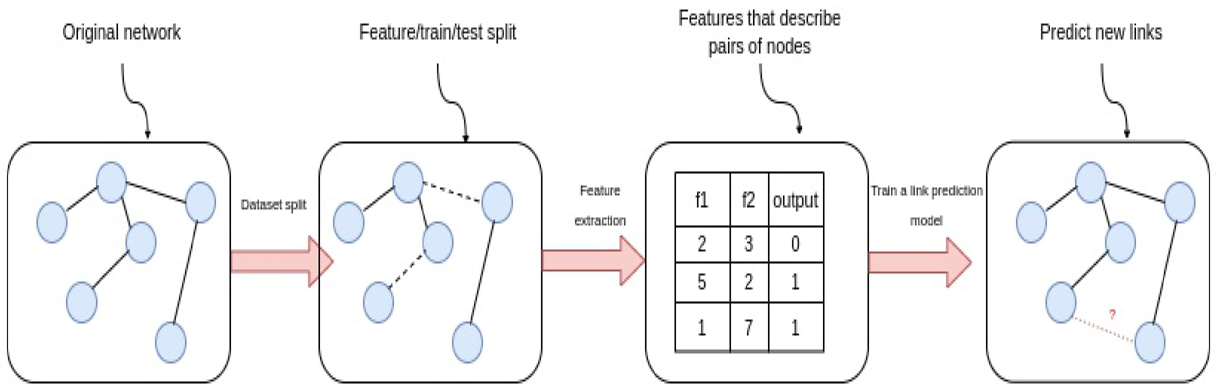

Stream
SHARED_AUDIENCE
Stream
SHARED_AUDIENCE ?
Stream
SHARED_AUDIENCE
SHARED_AUDIENCE
Stream
Stream
Using predicted links to populate "Recommendations" section
Using existing links to populate "Users also watched" section

Figure 10.4 visualizes the network of Twitch channels or streams, where the relationships represent the audience overlap. The solid lines represent existing shared audience overlap and could be used to populate one section of recommendations. As a data scientist, you could use the information about the existing relationships to predict future audience overlap. The future audience overlap predictions can then be used to power your recommendation engine.

Interestingly, using future link predictions as recommendations could, in turn, be thought of as self-fulfilling projections. Similar approaches have been used to recommend movies [Lakshmi & Bhavani, 2021], products [Darke et al., 2017], or even links between medical concepts [Kastrin et al., 2014]. Link prediction can also be used with other techniques to construct a hybrid recommendation engine.

How would you go about training a classification model that could be used to predict future overlap of audiences between channels?

Figure 10.5. Building a link prediction model to recommend Twitch channels.



The high-level overview of the process of training a link prediction classification model is presented in Figure 10.5. The first step is to split the relationships into three distinct sets. One set is used to generate network features, while the other two are used to train and evaluate the classification model. You might run into *data leakage* problems if you used the same relationships to generate the network features and train the classification model. Data leakage occurs when your training data contains information about the output, but similar data will not be available when the model is used for predictions. Leakage frequently leads to high performance during the training and possibly evaluation of the model, but unfortunately doesn’t perform well for new predictions. If you are using any graph features with the link prediction model, you have to take extra care to prevent any *feature leakage*. Leakage in features refers to when a feature contains the same or comparable information as the output. For example, imagine you are using the distance-based shortest path feature between two nodes. By using the same relationships to generate features and train the model, the model would simply learn to classify or predict a connection between all pairs of nodes with a network distance of one. In other words, when the network distance between a pair of nodes is one, there is an existing relationship between the pair of nodes. Therefore, the network distance feature and the classification output would contain the same information, introducing feature leakage. Essentially, you can think of data leakage as cheating on your model evaluation by peeking at the results during training. To avoid data leakage, you need to use one set of relationships to calculate the network features and another relationship set to provide supervised classification examples for training and evaluating the model.

Once the dataset split is done, you need to calculate link prediction metrics that will be used to train a model. As mentioned before, you could calculate network distance, the number of common neighbors, or simply aggregate node properties. You need to calculate the features for both the positive as well as the negative examples of links in the network. Lastly, you use these link prediction features to train a classification model to predict whether a link is probably to happen in the future or not. Given that the dataset was split into training and test sets of relationships, you can evaluate your model on the test set of the dataset. Once you determine that the classification model performs well enough, you can use it in production to generate recommendations for users of your platform.

To follow the exercises in this chapter, you need to have the Twitch network imported into the Neo4j database as described in Chapter 9. The Jupyter notebook with all the code examples in this chapter is available on https://github.com/tomasonjo/graphs-network-science/blob/main/notebooks/Chapter%2010.ipynb.

10.2 Dataset split

You need to split the dataset accordingly in order to be able to evaluate the trained classification model. If your model used no graph-based features like distance-based or neighborhood-based metrics, then you could follow the traditional test-train data split. Imagine that you would try to predict new links between people based on their age and education. Since both of those features are node properties that are not graph-based, you could simply take 80% of existing relationships as the training set and evaluate your model on the remaining 20% of relationships. Obviously, you need to add some negative examples as the model otherwise cannot learn to differentiate between the two outputs and might produce inaccurate predictions. Producing negative examples is not a problem as there are many pairs of nodes that are not connected. In practice, positive link examples scale linearly with the number of nodes in the graph, while negative examples scale quadratically. This could lead to a considerable class imbalance problem. However, it is a frequent step in the link prediction process to subsample the negative classification samples to around the same number as the positive ones. You will learn more about negative example subsampling in the next section.

As soon as you add any graph-based features that capture the similarity or closeness of nodes in the graph, you need to be very mindful of data leakage. Remember, when a feature contains the same or comparable information as the output variable but is unavailable when making predictions, you have introduced data leakage into the workflow. The most obvious example is the network distance between nodes in the graph. If the network features and training examples are calculated on the same set of relationships, then the model would simply learn that relationships exist between nodes that are only one hop away. However, none of the pairs of nodes without a link in the network will be classified as probable to form a connection, as none are one hop away. Even node embeddings based on the homophily principle, introduced in Chapter 9, could be problematic if you didn’t perform a proper dataset split. Overall, most of the graph-based features might introduce some data leakage issues. It is common to split the relationships into three sets to avoid data leakage problems.

- Relationship set used to generate features
- Relationship set used to train the model
- Relationship set used to evaluate the model

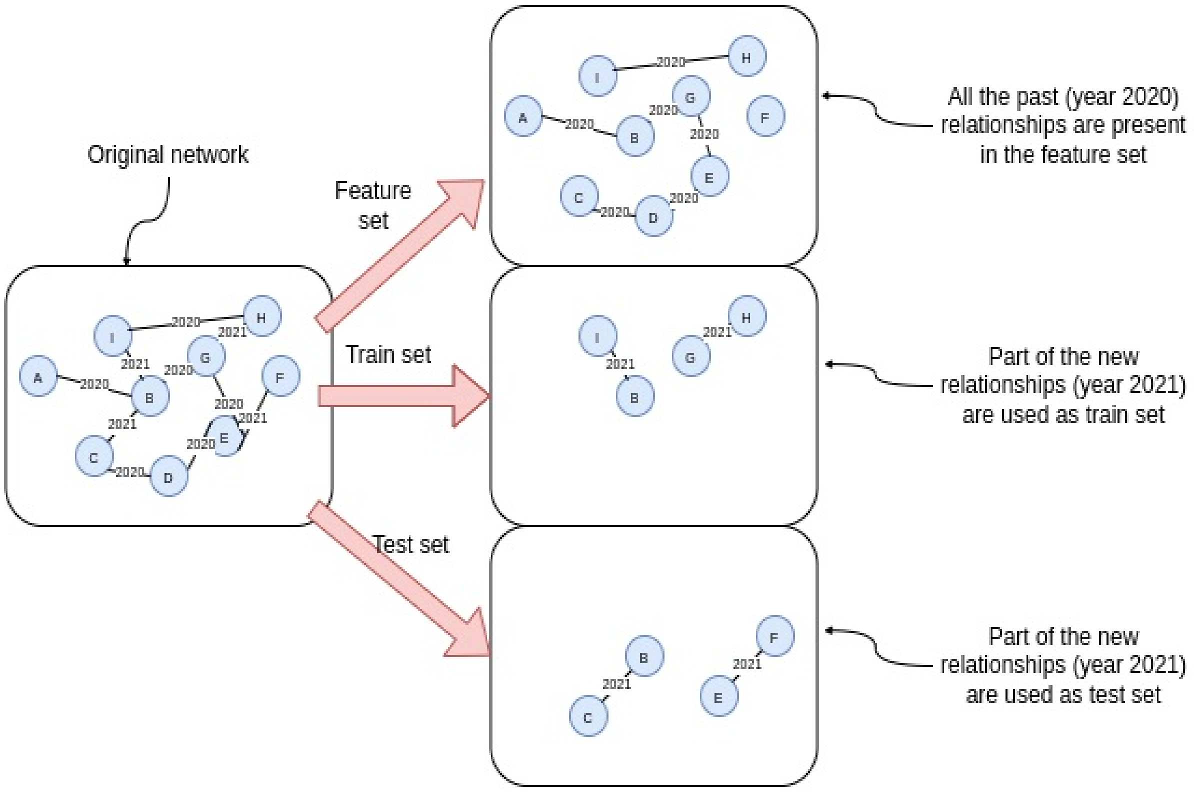
Using separate sets of relationships to generate network features and then train and evaluate the classification, you can avoid feature leakage issues with graph-based features. Therefore, none of the calculated network features will have identical or very comparable information as the output variable. With the network distance example, supervised classification examples will have a minimum network distance of two. Both negative and positive classification examples can have a network distance of two. In turn, the network distance feature is not identical to the output variable and you prevent any feature leakage.

There are many options on how to go about performing the dataset split. In this section you will learn about the *time-based* and *random* dataset split techniques.

10.2.1 Time-based split

In theory, link prediction is a technique to predict future connections based on past ones. You could produce a dataset split based on the time component if you know when the links were created.

Figure 10.6. Time-based approach to relationship split for link prediction task.


Original network
Feature set
Train set
Test set
All the past (year 2020) relationships are present in the feature set
Part of the new relationships (year 2021) are used as train set
Part of the new relationships (year 2021) are used as test set

The original network in Figure 10.6 has relationships created between 2020 and 2021. In this simple graph example, you can use the relationships from 2020 to generate network features. There should be a significant number of relationships in the feature set, as you don’t want the network to be too disconnected or have too many isolated nodes. Having too few relationships in the feature set might produce poor network features, which might, in turn, not be predictive of future links. Relationships created in 2021 in Figure 10.6 are then used to construct the test and train sets. For example, you could use 80% of the newer relationships as the training set and the remaining 20% to evaluate the model. Remember that the relationships used to generate network features should not be used in either the train or the test sets. Optionally, you could also introduce a validation set from the newer relationships created in 2021 if you plan to perform any hyper-parameter optimization of the classification model.

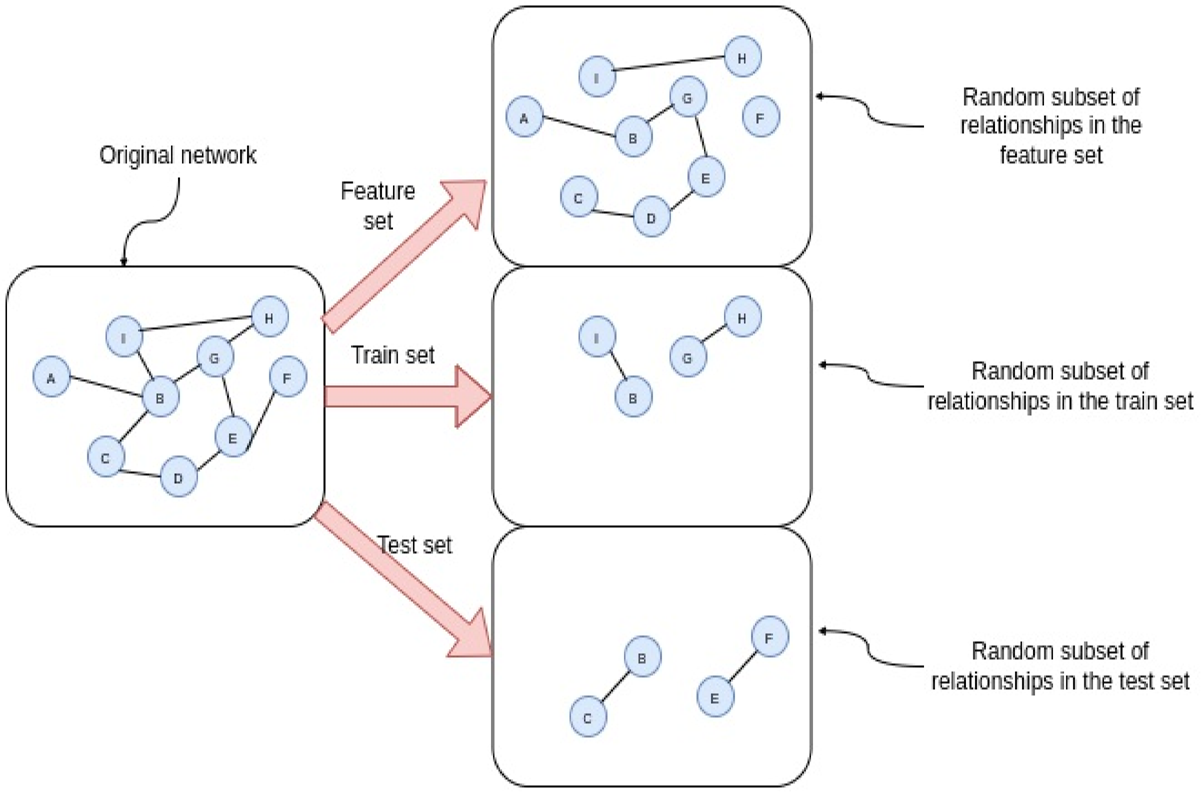
Use the time-based split if possible, as it accurately mimics the scenario of predicting future links. In the example in Figure 10.6, you take existing knowledge about relationships (year 2020) and try to predict the future (year 2021). The additional benefit is that the model should learn to capture the underlying organizing mechanism of the network as the time-based split follows network evolution, which should, in turn, provide better predictions.

Unfortunately, the Twitch dataset of shared audiences doesn’t have the time component of relationships available, so you will have to resort to another method.

10.2.2 Random split

Random split is similar to the time-based one in that you need to produce train and test sets as well as the feature set of relationships. You can’t differentiate between past and future relationships since no time information is available. Instead, you randomly take a subset of relationships in the graph as the train and test sets you will use to train and evaluate the classification model. Random split is also useful for link completion tasks, where you predict missing links in the network and have no time component available.

Figure 10.7. Random approach to relationship split for link prediction task.


Original network
Feature set
Train set
Test set
Random subset of relationships in the feature set
Random subset of relationships in the train set
Random subset of relationships in the test set
A
B
C
D
E
F
G
H
I

You can observe that Figure 10.7 is almost identical to Figure 10.6. The only difference is how to select which relationships belong to which set. With the time-based approach, you can choose the set the relationship belongs to based on the time property. However, since you don't have the time information available, you need to take the random approach. Therefore you select to which set a relationship belongs at random.

Note

The key concept with the link prediction dataset split is that to avoid data leakage, the links used to calculate network features should differ from the supervised link samples used for training and evaluating a classification model. Feature leakage is when a feature contains the same or comparable information as the output variable. For example, if you wouldn't introduce a separate feature set to calculate network features like the network distance, the classification model would learn that pairs of nodes that are one hop or traversal away have a 100% probability of forming or having a link. In turn, the model accuracy on the training and test sets would be 100% as the network distance contains the same information as the output variable. However, the model would be terrible at predicting missing or future links.

Optionally, you can introduce a validation set if you plan to perform any hyper-parameter optimization.

Now you will perform the random split for link prediction on the Twitch shared audience network. You need to have the Neo4j up and running with the Twitch dataset defined in Chapter 9 loaded. Next, you need to open a Jupyter notebook and define the connection to the Neo4j database.

Listing 10.1. Define connection to Neo4j.

```
from neo4j import GraphDatabase
import pandas as pd

url = "bolt://localhost:7687"
username = "neo4j"
password = "letmein"

# Connect to Neo4j
driver = GraphDatabase.driver(url, auth=(username, password))

def run_query(query, params={}):
  with driver.session() as session:
    result = session.run(query, params)
    return pd.DataFrame([r.values() for r in result], columns=res
```

The code in Listing 10.1 defines the connections with the Neo4j database and the `run_query` function that is used to execute any Cypher statement. Make any necessary changes to the `url`, `username`, or `password` variables.

Exercise 10.1

Count the number of relationships in the Twitch shared audience network.

There are 131427 relationships in the dataset. You will begin by constructing the feature set of relationships. Remember, the feature set needs to be the largest as you want to retain as connected a network as possible without too many isolated nodes or disconnected components. In this instance, you can use 90% of all relationships in the feature set, leaving you with around 13000 positive examples for the train and test sets. To construct the feature set, you will create new relationships with a `FEATURE_REL` type. Cypher offers a `rand()` that returns a random floating point number in the range from 0 to 1 and follows an approximate uniform distribution. To select a random subset of relationships for the feature set, you will use the `rand()` function in the Cypher statement.

The following Cypher statement takes approximately 90% of existing `SHARED_AUDIENCE` relationships and creates a new connection with the `FEATURE_REL` between those pairs of nodes.

Listing 10.2. Construct a relationship feature set.

```
run_query("""
MATCH (s1:Stream)-[:SHARED_AUDIENCE]->(s2:Stream)
WITH s1, s2
WHERE rand() <= 0.9
MERGE (s1)-[:FEATURE_REL]->(s2);
""")
```

The code in Listing 10.2 selects and creates a random, non-deterministic set of relationships. Make sure to run it only once, as the relationship split would not be acceptable otherwise. If you have run the query multiple times for any reason, simply delete the `FEATURE_REL` relationships and rerun the Cypher statement in Listing 10.2. Note that you might get a slightly different count of `FEATURE_REL` relationships when rerunning the query due to using the `rand()` function.

Exercise 10.2

Now you will select the relationships for the train and test set. You will start by producing positive samples for the classification model. The positive samples are relationships between the pairs of nodes between which the `SHARED_AUDIENCE` relationships exist but not the `FEATURE_REL` ones.

Match the pairs of nodes between which the `SHARED_AUDIENCE` exists but not the `FEATURE_REL` ones. Next, use the `MERGE` clause to create new relationships between those pairs of nodes with the `TEST_TRAIN` type. Finally, return the count of newly created relationships.

I got the result of 13082 created relationships. You will probably get a different number, but it should be in the same ballpark of around 13 thousand. You have prepared the positive samples for the classification model. Now it is time to select some negative examples where the relationship does not exist.

10.2.3 Negative samples

When training a binary classifier like the link prediction model, you should include both positive and negative examples in the training and test sets. Without negative examples, the model cannot learn to differentiate between the two outputs and might produce inaccurate predictions.

A common characteristic of real-world graphs is that they are sparse. Imagine any big social platform on the internet. You might have hundreds or thousands of friends on the platform. However, there are millions or, in some cases, billions of users on the platform. That means you have only a thousand of a billion relationships possible. In the machine learning context, every user has up to a few thousand positive examples and possibly around a billion negative examples. If you used all the negative examples, you would have to deal with a considerable class imbalance as the positive examples of relationship scale linearly with the number of nodes while the negative examples scale quadratically. Most machine learning models perform best when the number of samples in each class is about the same. However, if the dataset is heavily imbalanced, then you might get a high accuracy by just predicting the majority class every time. With link prediction, if you would predict that no link exists between any pair of nodes, you would probably get around 99% accuracy in most cases. Thus, there would be a high probability of misclassification of the minority class and, consequently, a poor classification model performance. Therefore, it is common to subsample the negative examples and use about the same number of positive and negative samples in most link prediction workflows.

Exercise 10.3

In this exercise, you will select pairs of nodes to construct the negative examples for the classification model. You must select about the same number of negative samples as the positive ones produced in Exercise 10.2. You can use the count of 13094 positive examples or the count you got in Exercise 10.2. The negative examples should be produced in a way that no relationship exists between the pair of nodes in feature, train, or test sets.

Start by matching a pair of nodes where the `SHARED_AUDIENCE` relationship does not exist between them. Next, ensure that you have matched two different nodes and will avoid running into situations where both the source and the target node are the same. Once the pairs of nodes are correctly matched, use the `LIMIT` clause to limit the number of negative examples to around 13 thousand. Finally, create a relationship between the selected pairs of nodes with the `NEGATIVE_TEST_TRAIN` type.

10.3 Network feature engineering

Now you will produce network features that capture the closeness or similarity of pairs of nodes in the network. The idea is that the closer or similar a pair of nodes are given the network metrics, the more likely they are to form a future connection. The future connections will then be used to provide better recommendations to Twitch users.

You might have noticed that the train and test sets are lumped together under the `TEST_TRAIN` and `NEGATIVE_TEST_TRAIN` relationship types. As you need to calculate the link prediction features for both the train and test sets, there is no need to differentiate between the two just yet. Remember, all the graph-based features for the train and test sets will be calculated strictly only on the feature set of relationships to prevent any data leakage.

Again, you have the option to choose between learned or manually defined features. For example, you could use the node2vec algorithm to calculate node embeddings and then use the cosine similarity of embeddings between pairs of nodes as a feature of the classification model. However, since you would be using transductive node embeddings to calculate link prediction features, you would need to retrain the classification model every time a new node is added to the graph. While that might be satisfactory in some scenarios, you probably don’t want to retrain a model every time a new streamer shows up on the platform. Luckily, a lot of research was done about link prediction features from which you can borrow some ideas for feature engineering. It makes sense to start by selecting straightforward and uncomplicated features and evaluating their performance. If there is a need, you can always later use more complex techniques like inductive node embeddings.

You must finish Exercises 10.2 and 10.3 before continuing with the code examples to generate features.

10.3.1 Network distance

The first feature you will calculate is the *network distance*. The network distance is calculated by finding the shortest path between the pair of nodes and then counting the number of relationships in the shortest path.

Figure 10.8. Calculating network distance between a pair of nodes A and E.

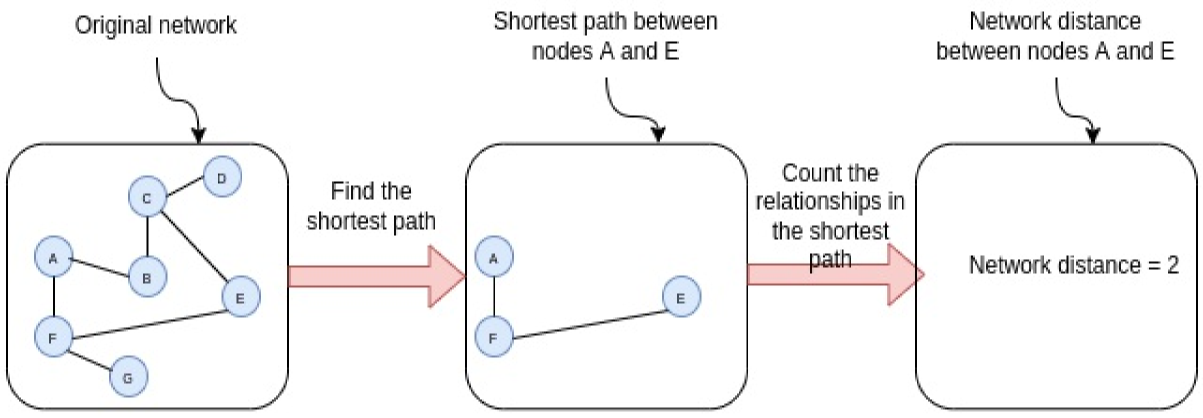

Original network
Shortest path between nodes A and E
Network distance between nodes A and E
Find the shortest path
Count the relationships in the shortest path
Network distance = 2
A
B
C
D
E
F
G

Figure 10.8 visualizes the process of finding the network distance between nodes A and E. In the first step, you need to calculate the shortest path between the particular pair of nodes. When dealing with an unweighted network, the shortest path represents the path that traverses the least relationships to get from one node to another. In the example in Figure 10.8, you must traverse two relationships to get from node A to node E. In other words, the network distance between nodes A and E is two.

The idea behind the network distance is that the closer the two nodes in the network, the more likely they are to form future connections. For example, imagine you are dealing with a link prediction in a social network. The network distance between a pair of persons in the train or test set should never be one, as this would mean that you haven't performed the relationship split correctly. However, if the network distance is two, that would mean that the pair of persons have at least one common friend. If the distance is greater than two, the two persons don't have any common friends and are less likely to form future connections. In theory, the higher the network distance, the less likely a future connection is. In your use-case, the closer the two streams are in the network, the more likely that there will be a significant audience overlap in the future.

Using Cypher query language, you can find the shortest unweighted path with the `shortestPath()` function. In an unweighted path, the traversal of each relationship has an identical cost, so the shortest path between two nodes will always be the count of the total relationships in a path between them. The `shortestPath()` function expects as input a Cypher pattern that defines the source and target nodes as well as the optional allowed relationship types in the path. For more advanced use cases, you can also define the minimum or the maximum number of traversals or relationships in the path.

The following Cypher statement finds the shortest path between Surya and Jim.

Listing 10.3. Finding shortest unweighted paths with Cypher.

```
MATCH (source:Person {name:"Surya"}),
      (target:Person {name:"Jim"}) #1
MATCH p = shortestPath((source)-[:FRIEND|COWORKER*1..10]->(target
RETURN p
```

The first part of Listing 10.3 is a simple `MATCH` clause used to identify the source and target nodes. Next, you need to define the shortest path constraints using the Cypher syntax. The defined graph pattern that defines the shortest path constrains in Listing 10.3 is:

Listing 10.4. Graph pattern used to define the shortest path constraints.

```
(source)-[:FRIEND|COWORKER*]->(target)
```

The Cypher syntax in Listing 10.4 defines the shortest path between the `source` and `target` nodes. One constraint of the shortest path is that it can only traverse `FRIEND` or `COWORKER` relationships. The function ignores all the other relationship types. Note that the relationship direction is also essential. In the example in Listing 10.4, the shortest path algorithm can only traverse outgoing relationships throughout the path. Lastly, you need to add the `*` symbol to allow the algorithm to traverse multiple relationships. If the `*` symbol were missing, one of the shortest path constraints would be that the algorithm can only traverse a single relationship.

Now you will calculate the network distance for all pairs of nodes in the train and test sets of relationships. The test and train sets of pairs of nodes are tagged with the `TEST_TRAIN` and `NEGATIVE_TEST_TRAIN` relationship types. Then you must find the shortest path between all pairs of nodes in the two sets. In the last step, you will calculate the length of the shortest path, which is equivalent to the number of relationships, with the `length()` function.

Listing 10.5. Calculate the network distance between pairs of nodes in the train and test sets.

```
run_query("""
MATCH (s1)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2) #1
MATCH p = shortestPath((s1)-[:FEATURE_REL*]-(s2)) #2
WITH r, length(p) AS networkDistance #3
SET r.networkDistance = networkDistance #4
""")
```

You can notice that there is no direction indicator in the shortest path graph pattern definition in Listing 10.5. Therefore, the shortest path algorithm is able to traverse the relationship in opposite direction as well, effectively treating the relationships as undirected.

10.3.2 Preferential attachment

Another popular metric used in link prediction is the so-called *preferential attachment*. Preferential attachment is an underlying organizing principle occurring in real-world networks where nodes with a higher number of relationships are more likely to make new relationships. In the social network example, people with more friends are more likely to make new connections. They might be invited to more social events or be introduced more due to having many friends. The preferential attachment model was first described by Barabási and Albert [Barabási & Albert, 1999].

Figure 10.9. Nodes with higher degree are more likely to form new connections.

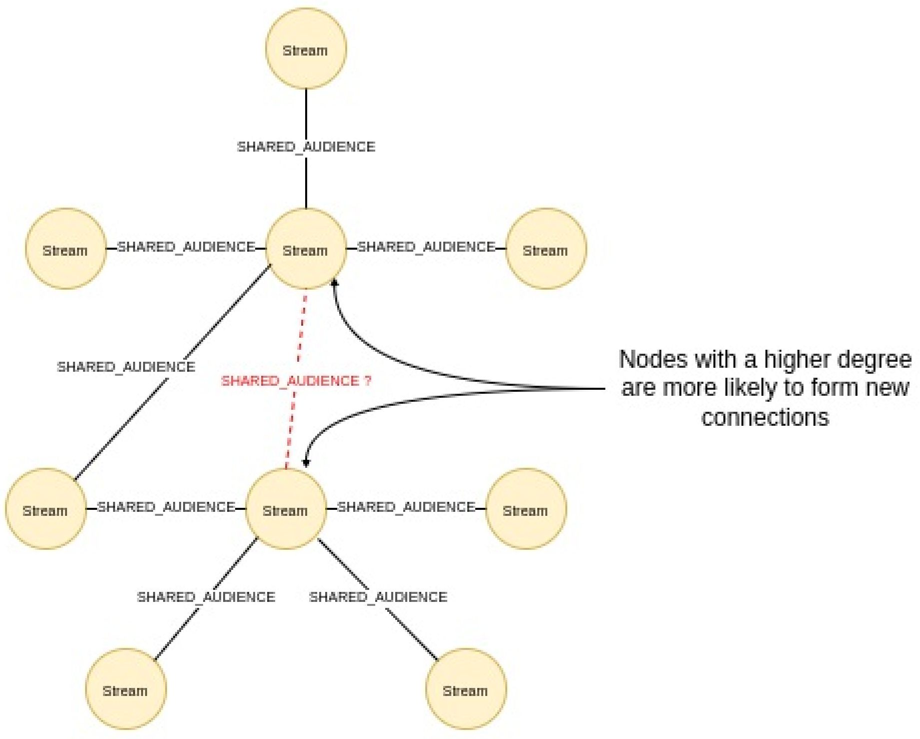


Figure 10.9 visualized two `Stream` nodes in the center with a relatively large node degree. The preferential attachment mechanism assumes that streams that already share a significant audience with many other streams are more likely to form future connections. So, following the preferential attachment principle, you could assume that the two central `Stream` nodes are likely to have a shared audience overlap, as indicated by the dotted line in Figure 10.9.

To calculate the preferential attachment metric between the particular pair of nodes, you need to multiply their node degrees. Essentially, you take the node degree of the first node and multiply it by the node degree of the second node. When a pair of nodes have a high preferential attachment metric, they are more likely to form a connection in the future.

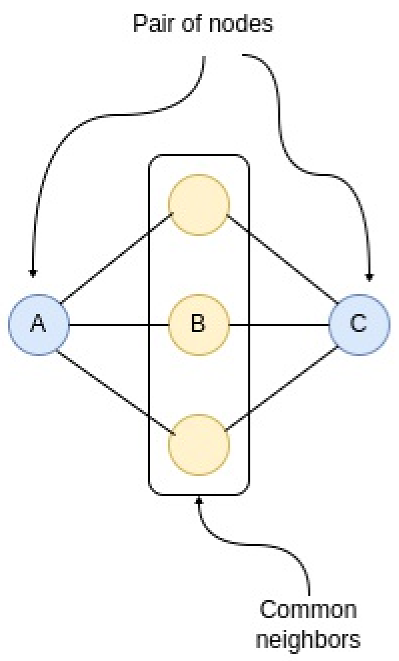
Exercise 10.4

Calculate the preferential attachment metric for pairs of nodes in the train and test sets. Similarly to network distance metric calculation, you start by matching the pairs of nodes that are connected with the `TEST_TRAIN` or `NEGATIVE_TEST_TRAIN` relationships. Next, you calculate the node degrees for both nodes. Make sure to count both incoming and outgoing relationships as the node degree and that you count only the `FEATURE_REL` relationships. Finally, multiply the two node degrees and store the results under the `preferentialAttachment` property of relationships.

10.3.3 Common neighbors

The next metric you will calculate as a link prediction feature is the *common neighbors* metric. The intuition behind the common neighbor metric is simple. The more common neighbors two nodes have, the higher the chance of a link forming in the future. In the context of social networks, the more common friends two people have, the greater the chance that they will meet or be introduced in the future.

Figure 10.10. Common neighbors between a pair of nodes.



Remember, due to the relationship split, none of the pairs of nodes in the train or test set have a direct connection. However, many nodes might have a number of common friends, as visualized in Figure 10.10. Imagine that all the nodes in Figure 10.10 represent Twitch streams. If stream A has an audience overlap with stream B, and stream B overlaps with stream C, then there will likely be an audience overlap between stream A and C in the future. Additionally, the higher the number of common neighbors between two streams, the higher the probability of a future link.

To use the common neighbor metric in the link prediction model, you need to calculate the number of common neighbors between all pairs of nodes in the train and test sets.

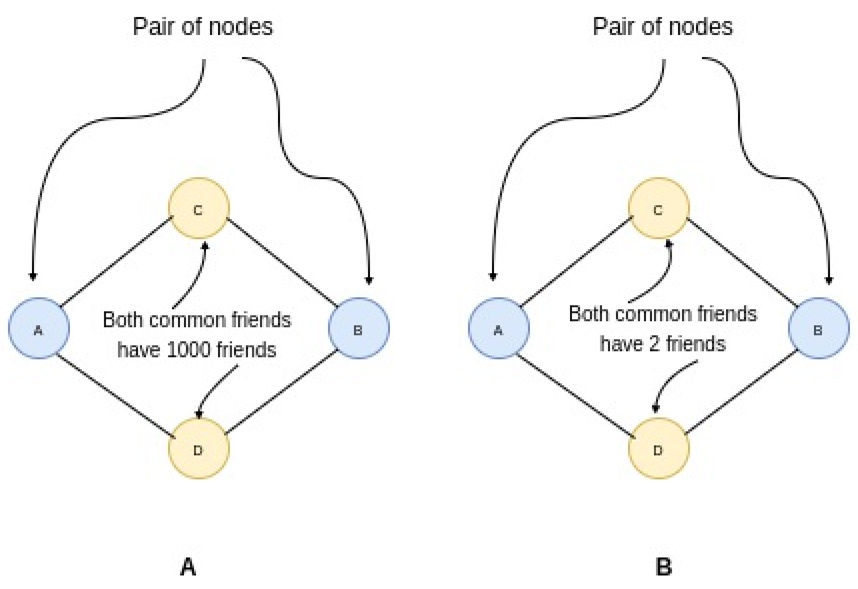
Exercise 10.5

Calculate the common neighbor metric for pairs of nodes in the train and test sets. Similarly to before, you start by matching the pairs of nodes that are connected with the `TEST_TRAIN` or `NEGATIVE_TEST_TRAIN` relationships. Then you need to count the distinct number of common neighbors between the matched pairs of nodes. Make sure to also include the results for pairs of nodes with no common neighbors by using `OPTIONAL MATCH` clause. Finally, store the number of common neighbors between pairs of nodes under the `commonNeighbor` property of relationships.

10.3.4 Adamic-Adar index

Adamic-Adar index is a link prediction metric first described by Adamic and Adar in 2003 [Adamic & Adar, 2003]. The idea behind the Adamic-Adar index is that the smaller degree the common neighbors between a pair of nodes have, the more likely that they will form a connection in the future. Again, imagine you are dealing with a social network. A pair of people have one friend in common. If that common friend has 1000 other friends, it is less likely that they will introduce the particular pair of people than if they only had two friends in total.

Figure 10.11. Intuition behind Adamic-Adar index.


Pair of nodes
C
A
B
D
Both common friends have 1000 friends
A
Pair of nodes
C
A
B
D
Both common friends have 2 friends
B

In example A in Figure 10.11, nodes A and B have two common neighbors or friends. The common friends are nodes C and D. Both nodes C and D have 1000 friends in total. Since common friends of nodes A and B have a broad group of friends themselves, it is less likely that either of the common friends will introduce nodes A and B. On the other hand, in example B in Figure 10.11, common friends of nodes A and B have only two friends in total.

Essentially, nodes C and D are only friends with nodes A and B. Therefore, since the friend circle of common friends is much smaller, it is more likely that, for example, nodes A and B are both invited to a birthday party or other social events that nodes C or D might host. A similar logic could be applied to the Twitch overlap network.

Adamic-Adar index is calculated using the following equation.

Figure 10.12. Adamic-Adar index equation.

$$A(x,y) = \sum_{u \epsilon N(x) \cap N(y)} \frac{1}{log\,|\,N(u)\,|}$$

where N(u) is the set of nodes adjacent to u

Don’t worry if you might not understand all the symbols in Figure 10.12. The Adamic-Adar index is defined as the sum of the inverse logarithmic node degree of common neighbors shared by a pair of nodes. The gist of the Adamic-Adar index calculation is the following.

1. Start by finding all the common neighbors of nodes `x` and `y`.
2. Calculate the node degree of all common neighbors
3. Sum the inverse logarithm of node degree of all common neighbors.

The following Cypher statement calculates the Adamic-Adar index between pairs of nodes in the train and test sets.

Listing 10.6. Calculate the Adamic-Adar index between pairs of nodes in the train and test sets.

```
run_query("""
MATCH (s1:Stream)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2:Stream)
OPTIONAL MATCH (s1)-[:FEATURE_REL]-(neighbor)-[:FEATURE_REL]-(s2)
WITH r, collect(distinct neighbor) AS commonNeighbors #1
UNWIND commonNeighbors AS cn
WITH r, count{ (cn)-[:FEATURE_REL]-() } AS neighborDegree #2
WITH r, sum(1 / log(neighborDegree)) AS adamicAdar #3
SET r.adamicAdar = adamicAdar; #4
""")
```

10.3.5 Clustering coefficient of common neighbors

The last link prediction that you will calculate is the *clustering coefficient of common neighbors*. A clustering coefficient measures the connectedness of the neighbors of a particular node. The value ranges from zero to one. A value of zero indicates that the neighboring nodes have no connections with each other. On the other hand, the value of 1 indicates that the network of neighbors forms a complete graph where all the neighbors are connected.

The clustering coefficient of common neighbors is a link prediction variant where you only calculate how connected the common neighbors of a particular pair of nodes are. Researchers have shown [Wu et al., 2015] that the clustering coefficient of common neighbors can improve the accuracy of link prediction models.

To calculate the local clustering coefficient of common neighbors between a pair of nodes, you need to identify the number of common neighbors as well as the number of links between common neighbors. Once you have that, you only need to divide the number of existing links between neighbors by the potential number of connections. The number of potential connections between neighbors equals the number of links if all neighbors are connected.

The following Cypher statement calculates the clustering coefficient of common neighbors and stores the results as a relationship property.

Listing 10.7. Calculate the clustering coefficient of common neighbors between pairs of nodes in the train and test sets.

```
run_query("""
MATCH (s1:Stream)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2:Stream)
OPTIONAL MATCH (s1)-[:FEATURE_REL]-(neighbor)-[:FEATURE_REL]-(s2)
WITH r, collect(distinct neighbor) AS commonNeighbors,
        count(distinct neighbor) AS commonNeighborCount #1
OPTIONAL MATCH (x)-[cr:FEATURE_REL]->(y) #2
WHERE x IN commonNeighbors AND y IN commonNeighbors
WITH r, commonNeighborCount, count(cr) AS commonNeighborRels
WITH r, CASE WHEN commonNeighborCount < 2 THEN 0 ELSE  #3
   toFloat(commonNeighborRels) / (commonNeighborCount *
                 (commonNeighborCount - 1) / 2) END as clustering
SET r.clusteringCoefficient = clusteringCoefficient #4
""")
```

You might have noticed that you treat relationships in the feature set as undirected at query time throughout the example in this section. The Cypher statement in Listing 10.7 is no different. At first, you ignore the relationship direction when identifying common neighbors. Since the relationships are treated as undirected, the number of potential connections is also 50% less than if you had a directed network. Therefore, the number of potential relationships in Cypher statement is divided by two in the second last line of Listing 10.7.

10.4 Link prediction classification model

The only thing left to do is to train and evaluate a link prediction model. Link prediction is a binary classification problem where you predict whether a link is likely to form in the future or not. You will train a Random Forest classification model to solve the link prediction task based on the features you calculated for the train and test sets of relationships. The Random Forest classification model is used as it is relatively robust to feature scaling and collinearity issues. However, you could have chosen other classification models like the logistic regression or support vector machine.

Use the following Cypher statement to retrieve link prediction features and output from the database.

Listing 10.8. Retrieve link prediction features and class output.

```
data = run_query("""
MATCH (s1)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2)
WITH r.networkDistance AS networkDistance,
     r.preferentialAttachment AS preferentialAttachment,
     r.commonNeighbor AS commonNeighbor,
     r.adamicAdar AS adamicAdar,
     r.clusteringCoefficient AS clusteringCoefficient,
     CASE WHEN r:TEST_TRAIN THEN 1 ELSE 0 END as output
RETURN networkDistance, preferentialAttachment, commonNeighbor,
       adamicAdar, clusteringCoefficient, output
""")
```

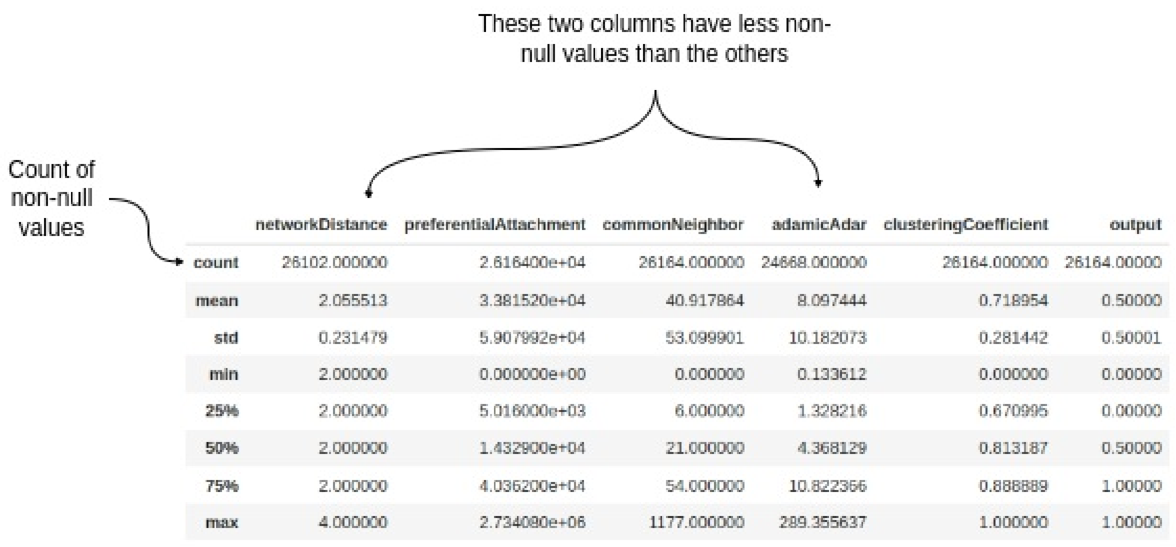
The Cypher statement in Listing 10.8 retrieves the features stored on `TEST_TRAIN` and `NEGATIVE_TEST_TRAIN` relationships. The last column in the results of Listing 10.8 is the `output` column, which differentiates between positive and negative classification examples. Positive examples are tagged with the `TEST_TRAIN` relationship type and are represented with a value of 1, while the negative examples are marked with `NEGATIVE_TEST_TRAIN` and are represented as 0.

Examining the distribution of relevant features is advisable, as with any other machine learning task. Pandas Dataframe has a `describe()` method that calculates the distributions of values in columns.

Listing 10.9. Define connection to Neo4j.

```
data.describe()
```

Figure 10.13. Distribution of link prediction features.


These two columns have less non-null values than the others
Count of non-null values

	networkDistance	preferentialAttachment	commonNeighbor	adamicAdar	clusteringCoefficient	output
count	26102.000000	2.616400e+04	26164.000000	24668.000000	26164.000000	26164.00000
mean	2.055513	3.381520e+04	40.917864	8.097444	0.718954	0.50000
std	0.231479	5.907992e+04	53.099901	10.182073	0.281442	0.50001
min	2.000000	0.000000e+00	0.000000	0.133612	0.000000	0.00000
25%	2.000000	5.016000e+03	6.000000	1.328216	0.670995	0.00000
50%	2.000000	1.432900e+04	21.000000	4.368129	0.813187	0.50000
75%	2.000000	4.036200e+04	54.000000	10.822366	0.888889	1.00000
max	4.000000	2.734080e+06	1177.000000	289.355637	1.000000	1.00000

Figure 10.13 visualizes the distribution of link prediction features. Interestingly, the network distance feature ranges from 2 to 4. However, it is mainly two as the mean network distance is barely 2.055. Moreover, it might not be the most predictable feature in this example due to the low variance. The preferential attachment has a wide range from zero to almost 3 million. Remember, the preferential attachment is calculated by multiplying the degrees of both nodes in the pair. The only way a preferential attachment can be zero is that some nodes have zero connections. While all nodes have

relationships in the original network, that might not be so in the feature set, where some connections are missing due to the data split. Interestingly, the clustering coefficient is relatively high on average.

10.4.1 Missing values

In total, there are 26164 training and test samples. However, Figure 10.13 also indicates that some values are missing in the `networkDistance` and `adamicAdar` columns. For example, there are only 26102 non-null values under the `networkDistance` feature. The network distance is undefined because the two nodes are not in the same component. Therefore, no path exists between the two. As mentioned, isolated nodes in the network might be the leading cause of missing network distance values. You can fill in the missing values with the maximum distance value of four. Remember, the higher the network distance between a pair of nodes, the less likely a link will be formed in the future between them, at least in theory. So, if a pair of nodes is not in the same component, which is a null network distance in this example, you want to choose a value representing a significant network distance to fill in missing values. Therefore, you decided to pick the maximum value of the network distance (4) in the dataset to fill in the missing values.

Another column with missing values is the `adamicAdar`. That might happen when a pair of nodes have no common neighbors. You can fill in the missing values of `adamicAdar` column with the mean Adamic-Adar value of about eight.

Listing 10.10. Fill in the missing values.

```
data['networkDistance'].fillna(4, inplace=True)
data['adamicAdar'].fillna(8.097444, inplace=True)
```

10.4.2 Training the model

With all the preprocessing steps done, you can go ahead and train the link prediction model. The `data` dataframe contains both the train and test sets of relationships. Therefore, you will first use the `train_test_split` from the

Scikit-learn library to split the test and train sets. You will use 80% of the samples as training examples and the remaining 20% to evaluate the model. If you were planning to perform any hyper-parameter optimization of the classification model, you could also produce a validation set. However, optimizing the classification model itself is beyond the scope of this book, so you will skip creating a validation set.

After the dataset split, you will feed the training samples into the Random Forest model, which will learn to predict whether a link is probable in the future or not.

Listing 10.11. Split the train/test sets and train the link prediction model.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X = data.drop("output", axis=1)
y = data["output"].to_list()

X_train, X_test, y_train, y_test = train_test_split( X, y, test_s

rfc = RandomForestClassifier()
rfc.fit(X_train, y_train)
```

The code in Listing 10.11 starts by defining the feature and target columns. The `output` column is used as the target, while all the other columns are used as model features. Next, you perform a test train split with the `train_test_split` function. Lastly, you instantiate a Random Forest model and learn it based on the training samples.

10.4.3 Evaluating the model

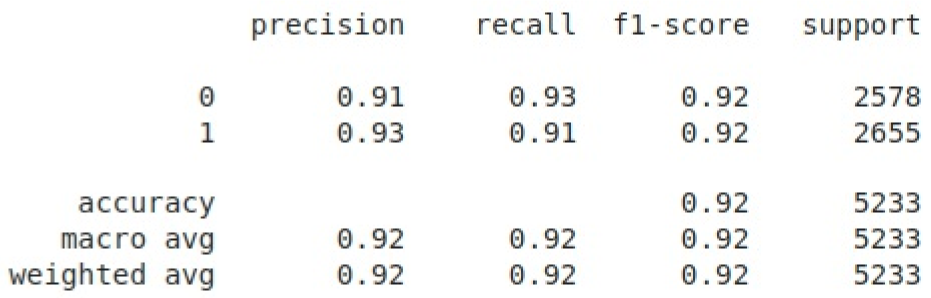
As with all machine learning tasks, you should evaluate the performance of your link prediction model using the test set. You will generate a classification report using a built-in Scikit-learn function.

Listing 10.12. Generate the classification report.

```
from sklearn.metrics import classification_report
```

```
y_pred = rfc.predict(X_test)
print(classification_report(y_test, y_pred))
```

Figure 10.14. Classification report of the link prediction model.



```
              precision    recall  f1-score   support

           0       0.91      0.93      0.92      2578
           1       0.93      0.91      0.92      2655

    accuracy                           0.92      5233
   macro avg       0.92      0.92      0.92      5233
weighted avg       0.92      0.92      0.92      5233
```

Congratulations, you have trained a link prediction model with an accuracy of 92%. The accuracy is a good metric as the ratio between negative and positive samples is even.

Lastly, you can evaluate the feature importance of the trained link prediction model. The following code will produce an ordered dataframe with the features ordered by their importance descending.

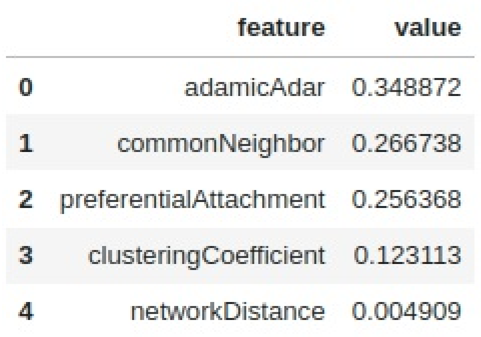
Listing 10.13. Evaluate the feature importance.

```
def feature_importance(columns, classifier):
    features = list(zip(columns, classifier.feature_importances_)
    sorted_features = sorted(features, key = lambda x: x[1]*-1)

    keys = [value[0] for value in sorted_features]
    values = [value[1] for value in sorted_features]
    return pd.DataFrame(data={'feature': keys, 'value': values})

feature_importance(X.columns, rfc)
```

Figure 10.15. Feature importance.



	feature	value
0	adamicAdar	0.348872
1	commonNeighbor	0.266738
2	preferentialAttachment	0.256368
3	clusteringCoefficient	0.123113
4	networkDistance	0.004909

You can observe by the table in Figure 10.15 that the network distance is the least important feature by a wide margin. That was somewhat expected due to the low variance of the network distance feature. Interestingly, the most relevant feature is the Adamic-Adar index, followed by the common neighbor and preferential attachment features. Note that you might get slightly different results due to the random dataset split used at the beginning of this chapter.

10.5 Summary

- Link prediction is a task of predicting future or missing links in the network
- Link prediction models are frequently used in recommender systems
- Link prediction features are designed to encode similarity or distance between pairs of nodes
- Link prediction features can be constructed by aggregating node properties, evaluating network distance, or examining local or global neighborhood overlap
- If you use transductive node embeddings in link prediction workflows, you cannot generate node embeddings for new unseen nodes during training and, therefore, cannot predict future links for nodes that weren't present during the training
- Feature leakage is when a feature contains the same or comparable information as the output variable. You could run into leakage isseus if you used the same relationships to generate network features as well as to train and evaluate a classification model. Therefore, it is necessary to split the dataset into feature, train, and test sets. Optionally, you can introduce a validation set if you plan to perform any hyper-parameter

optimization.

- The feature set is used to calculate network features, while the test and train sets provide classification samples to train and evaluate a model. Optionally, you can also introduce a validation set if you plan to implement any hyper-parameter optimization techniques.
- Using all the negative examples during training would lead to a considerable class imbalance. Therefore, it is common to subsample the negative examples and use about the same number of positive and negative samples.
- Network distance encodes how close a pair of nodes are in the network. The theory states that the closer a pair of nodes is, the more likely they will form a future link.
- Preferential attachment mechanism nicely captures how most real-world networks evolve. The underlying idea is that the rich get richer. Therefore, nodes with a higher number of existing links are more likely to form new links in the future.
- Local neighborhood overlap features can be as simple as the number of common neighbors two nodes have to more advanced, like the Adamic-Adar index, which assumes that the smaller degree the common neighbors between a pair of nodes have, the more likely that the pair will form a connection in the future.

10.6 References

[Lakshmi & Bhavani, 2021] Lakshmi, T., & Bhavani, S.. (2021). Link Prediction Approach to Recommender Systems.

[Darke et al., 2017] Evan Darke, Zhou Zhuang, & Ziyue Wang (2017). Applying Link Prediction to Recommendation Systems for Amazon Products.

[Kastrin et al., 2014] Kastrin, Andrej et al. “Link prediction in a MeSH co-occurrence network: preliminary results.” Studies in health technology and informatics vol. 205 (2014): 579-83.

[Barabási & Albert, 1999] Albert-László Barabási, & Réka Albert (1999). Emergence of Scaling in Random Networks. Science, 286(5439), 509-512.

[Adamic & Adar, 2003] Lada A. Adamic, & Eytan Adar (2003). Friends and neighbors on the Web. Soc. Networks, 25, 211-230.

[Wu et al., 2015] Zhihao Wu, Youfang Lin, Jing Wang, & Steve Gregory (2015). Efficient Link Prediction with Node Clustering Coefficient. CoRR, abs/1510.07819.

10.7 Solutions to exercises

The solution to Exercise 10.1 is the following:

Listing 10.14. Count the number of relationships.

```
run_query("""
MATCH (n)-[:SHARED_AUDIENCE]->()
RETURN count(*) AS result
""")
```

The solution to Exercise 10.2 is the following:

Listing 10.15. Construct the positive example for the test and train sets.

```
# Create test/train rel
# Take the remaining 10%
train_test_size = run_query("""
MATCH (s1)-[:SHARED_AUDIENCE]->(s2)
WHERE NOT EXISTS {(s1)-[:FEATURE_REL]->(s2)}
MERGE (s1)-[r:TEST_TRAIN]->(s2)
RETURN count(r) AS result;
""")
print(train_test_size)
```

The solution to Exercise 10.3 is the following:

Listing 10.16. Construct the negative example for the test and train sets.

```
# Create negative test/train pairs
run_query("""
MATCH (s1:Stream),(s2:Stream)
WHERE NOT EXISTS {(s1)-[:SHARED_AUDIENCE]-(s2)} AND s1 < s2
WITH s1,s2
LIMIT 13082
MERGE (s1)-[:NEGATIVE_TEST_TRAIN]->(s2);
""")
```

The solution to Exercise 10.4 is the following:

Listing 10.17. Calculate the preferential attachment feature for pairs of nodes in the train and test sets.

```
run_query("""
MATCH (s1:Stream)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2)
WITH r, count{ (s1)-[:FEATURE_REL]-() } *
        count{ (s2)-[:FEATURE_REL]-() } AS preferentialAttachment
SET r.preferentialAttachment = preferentialAttachment
""")
```

The solution to Exercise 10.5 is the following:

Listing 10.18. Calculate the common neighbors feature for pairs of nodes in the train and test sets.

```
run_query("""
MATCH (s1:Stream)-[r:TEST_TRAIN|NEGATIVE_TEST_TRAIN]->(s2)
OPTIONAL MATCH (s1)-[:FEATURE_REL]-(neighbor)-[:FEATURE_REL]-(s2)
WITH r, count(distinct neighbor) AS commonNeighbor
SET r.commonNeighbor = commonNeighbor
""")
```

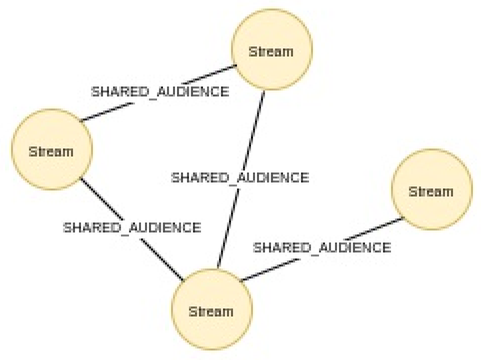
11 Knowledge graph completion

This chapter covers

- Introducing heterogeneous graphs
- Explaining knowledge graph embeddings
- Introducing knowledge graph completion workflow
- Explaining knowledge graph completion results

The previous chapter is an introduction to link prediction and completion techniques. The difference between link prediction and completion is that the first is a workflow to predict future links, while the latter deals with predicting missing links. However, in practice, link prediction and completion workflows are very similar. What wasn't explicitly mentioned is that the link prediction features used in Chapter 10 do not differentiate between various node or relationship types. For example, the number of common neighbors does not differentiate between different relationship or node types. Therefore, the link prediction features used in Chapter 10 work best with monopartite or *homogeneous* graphs. A monopartite or a homogenous graph consists of a single node and relationship type.

Figure 11.1. Homogeneous graph consisting of Stream nodes and SHARED_AUDIENCE relationships.



Visualization in Figure 11.1 depicts a homogeneous graph that consists of a single node type `Stream` and a single relationship type `SHARED_AUDIENCE`.

Suppose you work at a large pharmaceutical company as a data scientist. You have been tasked with predicting additional use cases for existing drugs that the company produces. The strategy of identifying new use cases for existing approved drugs is called *drug repurposing*. The oldest example of drug repurposing is acetylsalicylic acid or better known as Aspirin. It was initially used in 1899 as an analgesic to relieve pain. Later it was repurposed to be used as an antiplatelet aggregation drug. An antiplatelet drug decreases the ability of blood clots to form[Vane, 1971]. Aspirin was later repurposed again as it has been shown that daily administration of aspirin can prevent the development of cancers, particularly colorectal cancer [Rüschoff et al., 1998] [Rothwell et al., 2011].

It is very likely that as a data scientist, you don't have a biomedical background and, therefore, cannot manually pick new potential use cases based on domain expertise. What are your options? You can model known connections between drugs and diseases as a bipartite graph.

Figure 11.2. Bipartite network of existing drugs and known treatments.

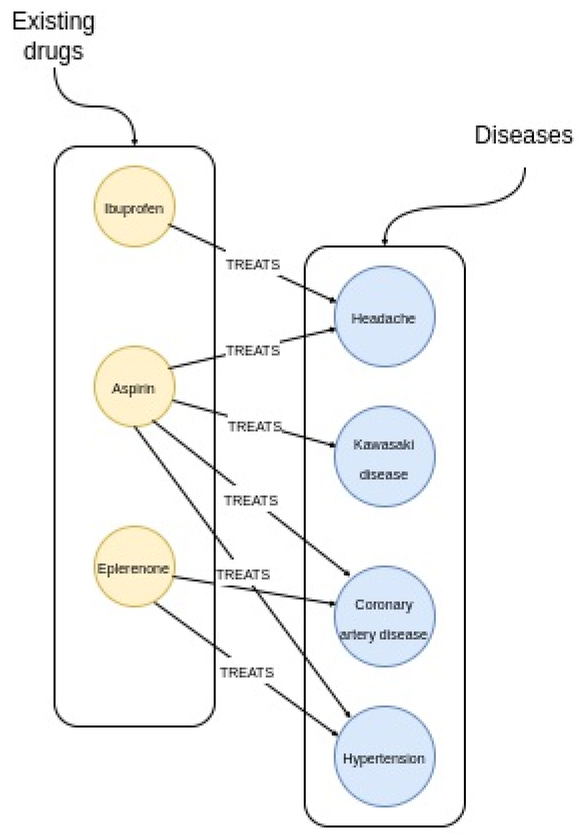
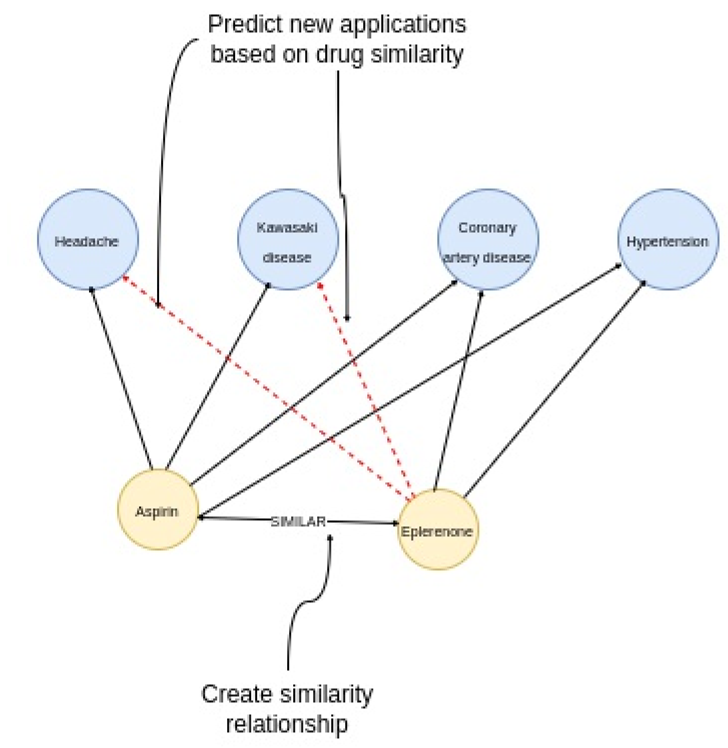


Figure 11.2 visualizes a bipartite network of approved drugs and diseases. The relationships indicate existing applications of drugs for treating conditions. For example, Aspirin can be used to treat headaches, Kawasaki disease, coronary artery disease, and hypertension.

You could create a drug repurposing workflow by first determining similar drugs. Once similar drugs have been identified, you could use that information to predict or recommend new applications for existing drugs.

Figure 11.3. Predict new drug applications based on drug similarity.



The drug repurposing workflow visualized in Figure 11.3 has two steps:

1. Identify similar relationships
2. Recommend new drug applications based of drug similarity

The first step is to identify similar drugs. One idea could be that the more common diseases two drugs treat, the higher the drug similarity between the two. There are several approaches you could take to infer the similarity relationship. You could use the Jaccard similarity coefficient, described in Chapter 7, to calculate the drug similarity. Another idea would be to use a node embedding model like the node2vec, presented in Chapter 9, to calculate node embeddings and compare drugs using the cosine similarity of node embeddings. Lastly, you could also borrow some of the link prediction features described in Chapter 10 to calculate drug similarity. Using any of the mentioned approaches, you would create a similarity relationship with some sort of score between pairs of drugs.

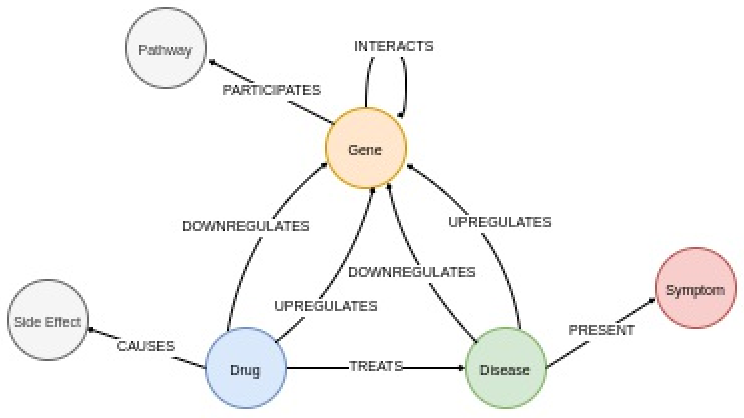
In the second step, you could recommend new drug applications based on the calculated similarity relationships. In the example in Figure 11.3, Aspirin and

Eplerenone are tagged as similar drugs. Therefore, you could predict potential applications for Eplerenone by examining which conditions similar drugs like Aspirin treat. In this specific example, the prediction would be that Eplerenone could be potentially used to treat Kawasaki disease and headaches.

Remember, the link prediction workflow only suggests the priority of evaluating new applications, while the domain experts then decide and potentially conduct clinical trials to determine new drug applications.

The described drug repurposing workflow is valid. However, with this approach, you would overlook a lot of existing biomedical knowledge. There is a lot of data about genes, biological processes, anatomy, and other biomedical information that you could incorporate into your graph and consequently into drug repurposing analysis.

Figure 11.4. Example schema of a complex biomedical graph.



Medical researchers have accrued a lot of knowledge over the years. There are a lot of official medical databases that you can borrow information from to construct a biomedical graph. For example, the graph schema in Figure 11.4 contains several types of nodes spanning from drugs, diseases, genes, side effects, and more. Additionally, there are several types of relationships present. Sometimes, multiple types of relationships are available between particular types of nodes. In Figure 11.4, you can observe that a drug can either upregulate or downregulate a gene.

A complex biomedical graph is an example of a *heterogeneous graph*, where multiple node and relationship types are present. In a drug repurposing workflow, you could use all the available information in a biomedical graph to predict new `TREATS` relationships. However, since the graph schema is more complicated, it requires a different approach to feature engineering than in the previous chapters. If you were inclined to perform a manual feature engineering workflow as described in Chapter 10, you would need to find a way to encode various node and relationship types. For example, the number of common neighbors, as used in Chapter 10, does not differentiate between various node and relationship types. A disease can upregulate or downregulate a gene, and you want to somehow encode them differently. Therefore, manual feature engineering would likely be tedious and labor-intensive while requiring domain expertise. While node embedding algorithms like the node2vec algorithm remove the need for manual feature engineering, they are not designed to differentiate between various node and relationship types. Luckily, you are not the first person to run into this problem. The solution to avoid manual feature engineering while having a model that differentiates between various node and relationship types is to use *knowledge graph embedding* models. Unlike node embedding models, knowledge graph embedding models encode both nodes as well as relationships in the embedding space. The added benefit of encoding relationships in the embedding space is that the embedding model can learn to differentiate between different relationship types.

11.1 Knowledge graph embedding model

As mentioned, the key difference between node embeddings and knowledge graph embedding models is that the latter embeds the relationships as well as nodes. Before delving into theory, you need to familiarize yourself with knowledge graph embedding terminology.

11.1.1 Triple

Knowledge graph embedding models use *triples* to describe graphs. A triple consists of two nodes known as *head* (h) and *tail* (t), and a labeled directed relationship (r).

Figure 11.5. Triple representation.

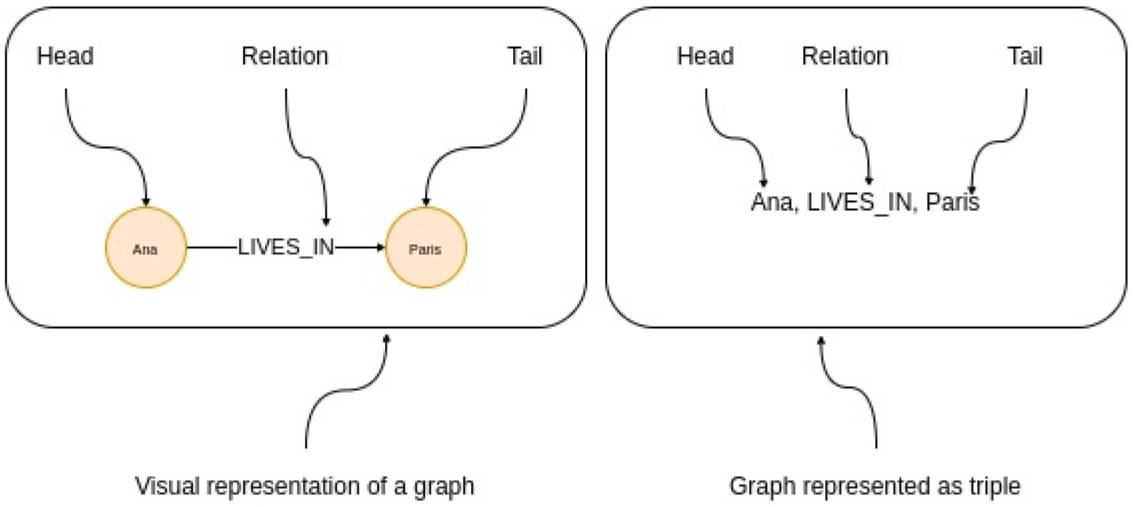


Figure 11.5 pictures a visualization of a sample graph on the left side and a triple representation of the same graph on the right. A triple consists of two nodes, a head (h) and tail (t), and a directed labeled relationship (r). The head node is the source or start node of the relationship, while the target or end node is marked as the tail node. In the example in Figure 11.5, Ana is considered as the head while Paris is the tail node. The whole idea of knowledge graph embeddings is to support heterogeneous graphs and differentiate between various types of relationships. Therefore, the relation label in a triple defines its type. The relation label in Figure 11.5 is `LIVES_IN`.

Exercise 11.1

Construct two triples to define your location. The first triple should contain information about the city you live in, while the second triple connects your city to the country it belongs to. Choose the relation labels you find the most appropriate.

Note

The triple is defined to differentiate between various relationship types or labels. However, there is no explicit definition of node labels. Therefore, the knowledge graph embedding models do not explicitly differentiate between different node types.

11.1.2 TransE

TransE [Bordes et al., 2013] is one of the earliest and most intuitive knowledge graph embedding models. The objective of the TransE method is to calculate low-dimensional vector representations, also known as embeddings, for all the nodes and relationships in the graph. The TransE method is frequently used to demonstrate knowledge graph embeddings as it is simple to illustrate and relatively cheap to calculate.

Figure 11.6. TransE encoding intuition.

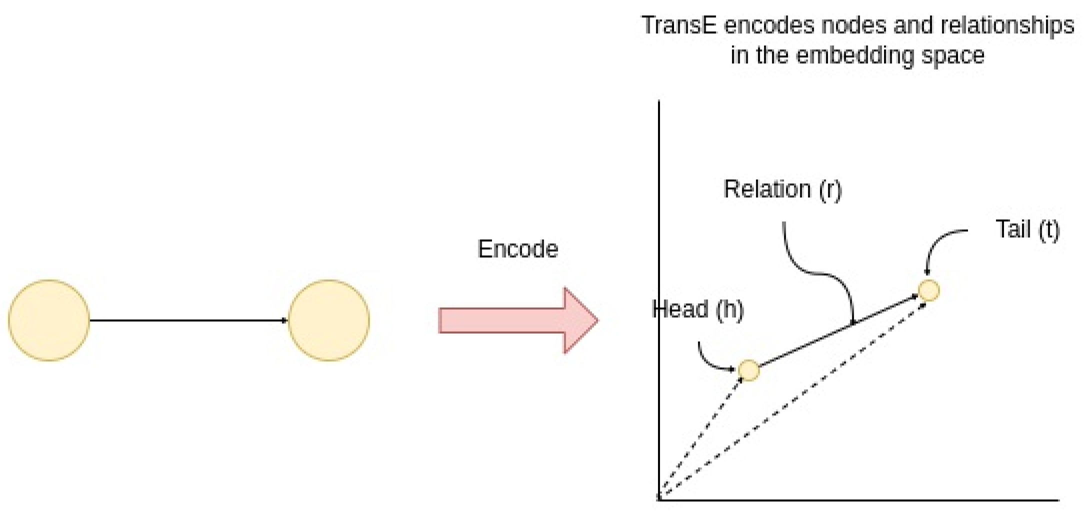

TransE encodes nodes and relationships in the embedding space
Encode
Relation (r)
Tail (t)
Head (h)

Figure 11.6 visualizes the concept of encoding nodes and relationships in the embedding space. The key idea of TransE method is to encode nodes and relationships in the embedding space so that the embedding of the head plus relation should be close to the tail. In Figure 11.6, you can observe that the embedding of the head node plus the embedding of the relationship is precisely equal to the embedding of the tail node.

Figure 11.7. TransE optimization metric.

$$h + r \approx t \text{ if relation is present}$$
$$h + r \neq t \text{ if relation is not present}$$

TransE method tries to produce embeddings so that for every triple in the training set, it minimizes the distance between the sum of the head and the relationship to the tail embedding. This optimization score can be written as `h + r ≈ t` as visualized in Figure 11.7. On the other hand, if a relationship between the head and tail node does not exist, then the sum of head and relation embedding should not be close to the tail (`h + r != t`).

You can read more about the actual mathematical implementation in the original article [Bordes et al., 2013].

11.1.3 TransE limitations

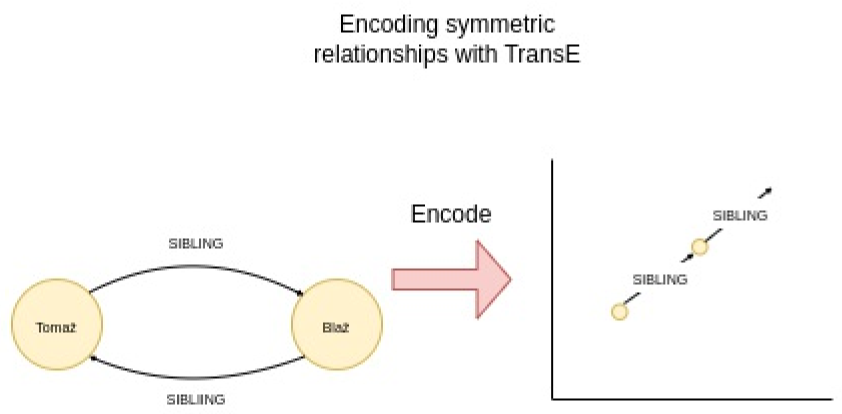
While TransE implementation is simple and intuitive, it has some drawbacks. There are three categories of relationships you will use to evaluate TransE method.

The first category of relationships is the *symetric* relations. The triple data structure does not allow undirected relationships. However, a category of relationships could be treated as undirected. The undirected relationships are referred to as symmetric within the field knowledge graph embedding models. An example triples of symmetric relationships are:

```
Tomaž, SIBLING, Blaž
Blaž, SIBLING, Tomaž
```

If Tomaž is a sibling of Blaž, then Blaž is also a sibling of Tomaž. There is no way around this simple fact. The question is can TransE encode symmetric relationships?

Figure 11.8. Encoding symmetric relationships with TransE.

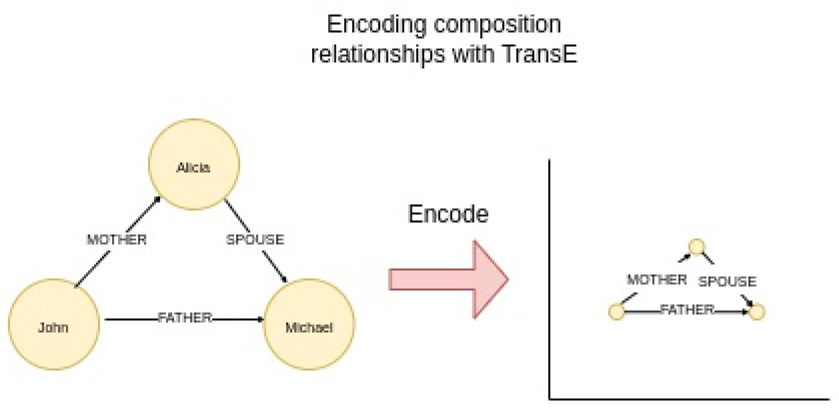


The TransE method produces a particular vector representation for each relationship type. Therefore, the `SIBLING` vector representation in Figure 11.8 has the same direction in both instances. The problem is that a vector representation of the same relationship type cannot point in the opposite direction. One `SIBLING` vector points from the head to the tail node. However, the second `SIBLING` vector starts from the second node and has the same direction as the first `SIBLING` vector. Therefore, the second `SIBLING` vector does not and cannot point back to the first node. Consequently, the TransE does not support symmetric relationships from a theoretical point of view.

The second category of relationships you will evaluate is the *composition* relations. One example of the composition relation is:

```
John, MOTHER, Alicia
Alicia, SPOUSE, Michael
John, FATHER, Michael
```

Figure 11.9. Encoding composition relationships with TransE.

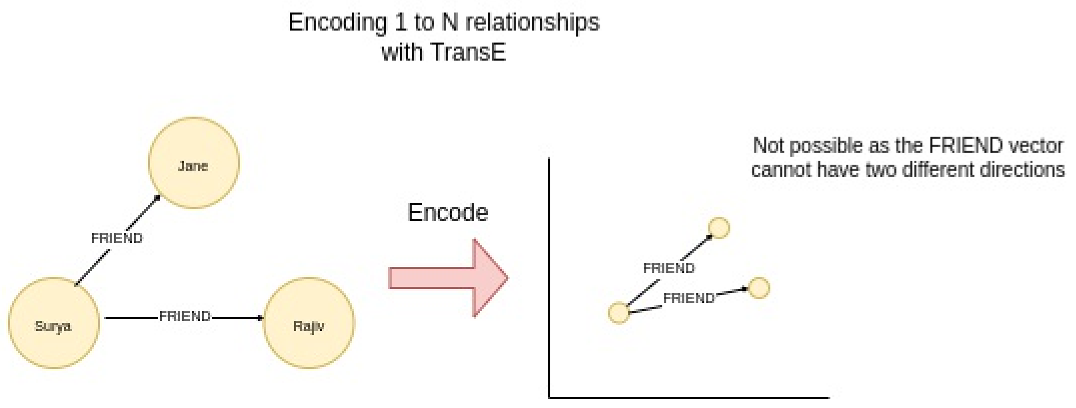


A composition relationship can be constructed by combining two or more different relationships. In the example in Figure 11.9, the `FATHER` relationship can be composed by adding the `MOTHER` and `SPOUSE` relationships. You can observe that one can fit the relationship vectors to fit this graph pattern. Therefore, the TransE method support composite relations.

The last category is the *1-to-N* relationships. Essentially, this scenario happens when a node has the same relationship to multiple other nodes. An example of 1 to N relations is:

```
Surya, FRIEND, Rajiv
Surya, FRIEND, Jane
```

Figure 11.10. Encoding 1-to-N relationships with TransE.


Encoding 1 to N relationships with TransE
Jane
FRIEND
Surya
FRIEND
Rajiv
Encode
Not possible as the FRIEND vector cannot have two different directions
FRIEND
FRIEND

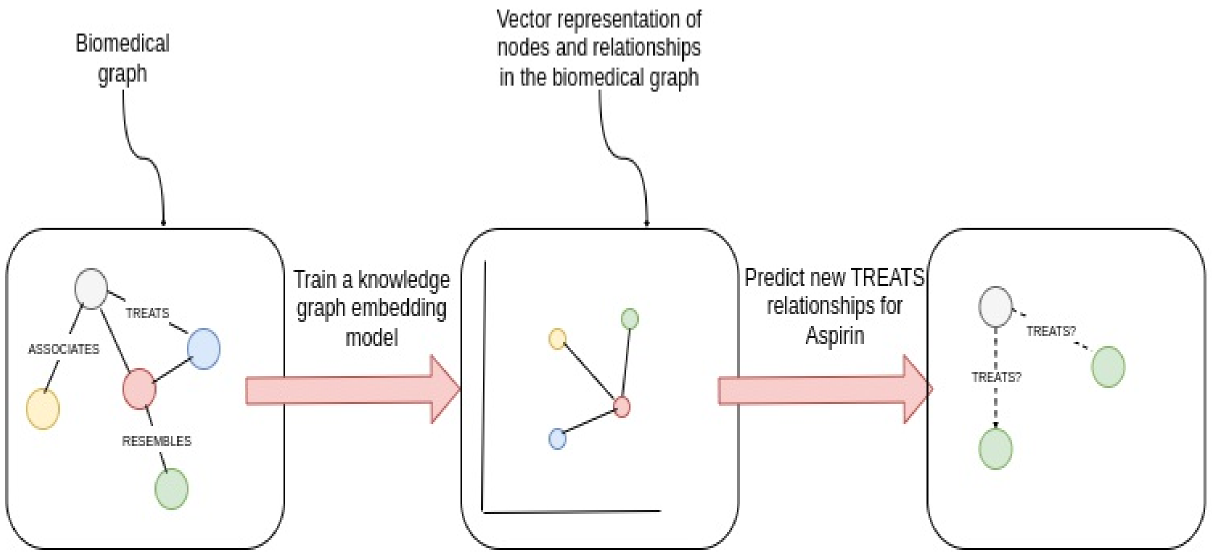
The only way TransE could encode that Surya is friends with both Jane and

Rajiv is that the vector representation of Jane and Rajiv is equal. Having the identical vector representation for Jane and Rajiv does not make sense, as they are different entities in the graph and, therefore, should have different embeddings. The only other solution would be that the `FRIEND` relationship vector would have different directions as visualized in Figure 11.10. However, the TransE method implements only a single vector representation or direction for a given relationship type. Consequently, the TransE method does not support 1-to-N relationships.

11.2 Knowledge graph completion

Now that you have gained a theoretical background in knowledge graph embeddings, you can continue with your task of predicting new applications for existing drugs. Imagine you work at a large pharmaceutical company that produces aspirin. Aspirin is a mass-produced drug, and therefore, a new application could rake in a lot of revenue. The idea is to use existing biomedical knowledge to predict new applications. You have determined that the best course of action would be to apply *knowledge graph completion* techniques to find new potential drug applications, also known as drug repurposing. Knowledge graph completion can be thought of as multi-class link prediction, where you predict new links and their type. You will train a knowledge graph embedding model to encode nodes and relationships in the biomedical graph and then use those embeddings to identify new potential applications for aspirin.

Figure 11.11. Drug repurposing workflow.


Biomedical graph
Vector representation of nodes and relationships in the biomedical graph
TREATS
ASSOCIATES
RESEMBLES
Train a knowledge graph embedding model
Predict new TREATS relationships for Aspirin
TREATS?
TREATS?

The drug repurposing workflow is visualized in Figure 11.11. The basis of the whole flow is a rich and complex biomedical knowledge graph that contains existing drugs, their treatments, and other biomedical entities like genes and pathways. As you are working for a large company, other great people at the company have already mapped and constructed the required biomedical graph. Next, you need to feed the biomedical graph into a knowledge graph embedding model. The Neo4j Graph Data Science library currently does not support knowledge graph embedding models. Since you won’t need to perform any graph transformations or manipulations, you can skip using a graph database altogether. While multiple Python libraries feature knowledge graph embedding models, I prefer *PyKEEN* [Ali et al., 2021] due to its simplicity and easy-to-use interface. Additionaly, PyKEEN implements more than 40 different knowledge graph embedding models along with out-of-the-box support for hyper-parameter optimization. Lastly, you will use a built-in PyKEEN method to predict new applications for Aspirin.

You need to install `pykeen` and `pandas` libraries to follow along with the code examples.

Listing 11.1. Install PyKEEN.

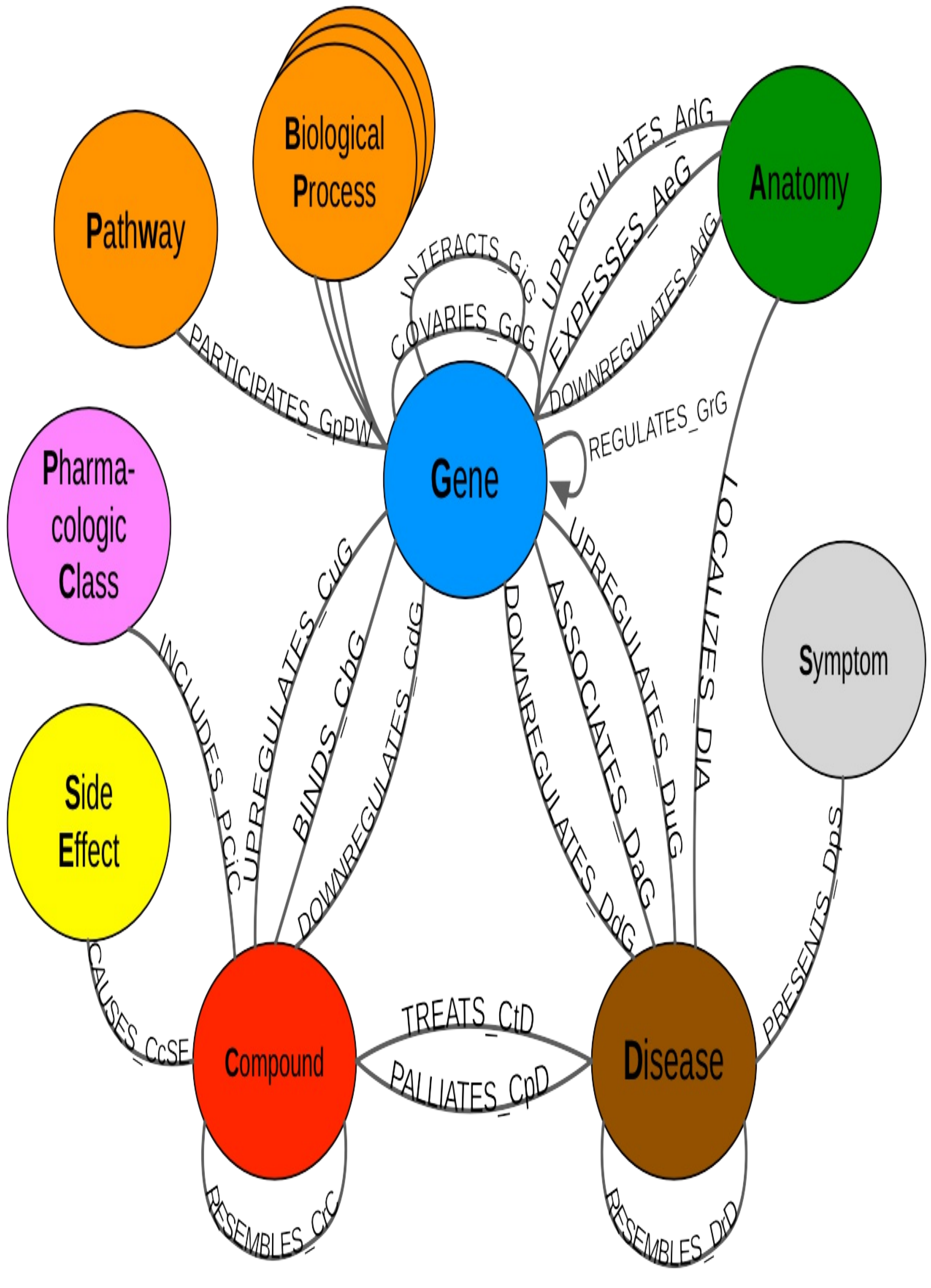
```
pip install pykeen==1.9.0 pandas
```

All the code is available as a Jupyter notebook (https://github.com/tomasonjo/graphs-network-science/blob/main/notebooks/Chapter%2011.ipynb).

11.2.1 Hetionet

Your coworkers have prepared a subset of the Hetionet dataset [Himmelstein et al., 2017] to use. The original Hetionet dataset contains 47,031 nodes (11 types) and 2,250,197 relationships (24 types).

Figure 11.12. Hetionet schema. Image from Himmelstein et al (2017) available under a CC BY 4.0 License at https://elifesciences.org/articles/26726.


Pathway
Biological Process
Anatomy
Gene
Pharma-cologic Class
Side Effect
Symptom
Compound
Disease
PARTICIPATES_GpPW
INTERACTS_GiG
COVARIES_GcG
UPREGULATES_AdG
EXPRESSES_AeG
DOWNREGULATES_AdG
REGULATES_GrG
LOCALIZES_DlA
UPREGULATES_CuG
BINDS_CbG
DOWNREGULATES_CdG
DOWNREGULATES_DdG
ASSOCIATES_DaG
UPREGULATES_DuG
INCLUDES_PCiC
PRESENTS_DpS
CAUSES_CcSE
TREATS_CtD
PALLIATES_CpD
RESEMBLES_CrC
RESEMBLES_DrD

Graph schema of the Hetionet dataset is presented in Figure 11.12. The graph contains various entities like genes, pathways, compounds, and diseases. Additionally, there are 24 different types of relationships present in the graph. Explaining all the medical terminology behind medical entities and their relationships could take a whole book. The most important relationship for drug repurposing workflow is the `TREATS` relationship that starts from a `Compound` node and ends at the `Disease` node. Essentially, the `TREATS` relationship encapsulates existing approved drug treatments. You will use knowledge graph completion techniques to predict new `TREATS` relationships originating from Aspirin or Acetylsalicylic acid node.

You will use a subset of the Hetionet dataset in this example. The subset has the following schema.

Figure 11.13. Graph schema of the Hetionet subset that will be used in the drug repurposing workflow.

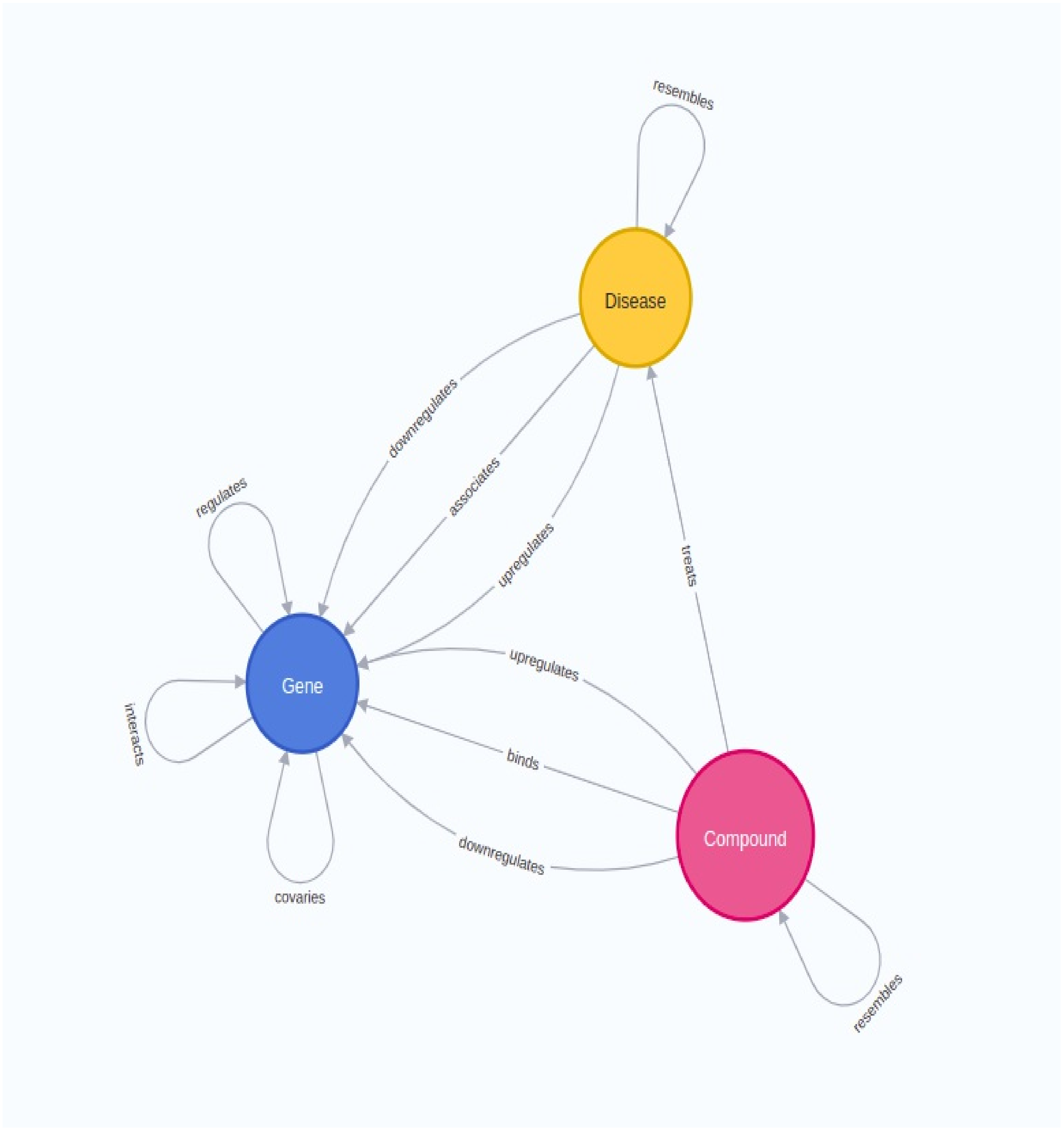


Figure 11.13 presents a subset of the Hetionet dataset that you will use in the drug repurposing workflow. The given subset contains 22634 nodes (3 types) and 561716 relationships (12 types). The graph contains existing approved drug treatments that can be found under the TREATS relationship, along with some additional information about how compounds and diseases interact with genes. The genes can also interact with other genes.

The subset of the Hetionet dataset is available on GitHub (https://github.com/tomasonjo/graphs-network-science/tree/main/dataset/hetionet) and has the following structure.

Table 11.1. Structure of the Hetionet edge CSV file.

source_name	source_label	target_name	target_label	type
SERPINF2	Gene	KLK13	Gene	interacts
SERPINF2	Gene	SSR1	Gene	interacts
SERPINF2	Gene	TGM2	Gene	interacts
SERPINF2	Gene	UBC	Gene	interacts
SERPINF2	Gene	SERPINB12	Gene	interacts

You will use the Pandas library to load the CSV file from GitHub.

Listing 11.2. Load Hetionet subset as a Pandas dataframe.

```
import pandas as pd

data = pd.read_csv(
    "https://bit.ly/3X2qp1r"
)
```

The code in Listing 11.2 first imports the Pandas library. Next, it uses the built-in method `read_csv` to load the Hetionet dataset from GitHub.

11.2.2 Dataset split

As with all machine learning workflows, you need to perform a test-train dataset split. You can feed a graph structure to PyKEEN with a list of triples. Remember, the triple data object consists of head, label, and tail elements.

Listing 11.3. Input triples to PyKEEN.

```
from pykeen.triples import TriplesFactory

tf = TriplesFactory.from_labeled_triples(
    data[["source_name", "type", "target_name"]].values,
)
```

The `TriplesFactory` is a PyKEEN class designed to store triples used for training and evaluating the model. The code in Listing 11.3 uses the `from_labeled_triples` method to input a list of triples from the Pandas dataframe. The `data` dataframe contains additional information about node labels, which we need to filter out. Therefore, the code in Listing 11.3 specifies using columns `source_name`, `type`, `target_name` as triples.

Now that the triples are loaded into PyKEEN, you can perform a dataset split with the following code.

Listing 11.4. Split dataset into train, test, and validation sets.

```
training, testing, validation = tf.split([0.8, 0.1, 0.1])
```

The dataset split is performed with the `split` method as demonstrated in the Listing 11.4. The method takes in an array of three values as a parameter that defines the ratio of the training, testing, and validation sets. The first value defines the ratio of the training set, the second value represents the testing set ratio, and the final number specifies the size of the validation set. The third value can be omitted as it can be calculated from the first two.

11.2.3 Train a PairRE model

While the TransE model is great for an introduction to knowledge graph embedding models, it has its limitations. For example, a single drug can be used to treat multiple diseases. However, as mentioned in the TransE introduction, the TransE method cannot encode 1-to-N relationships, making it a lousy model for biomedical knowledge graphs. Therefore, you will use a later and better model called PairRE [Chao et al., 2020]. PairRE is capable of encoding symmetry, composition, and 1-to-N relationships, which makes it a great model to use for biomedical knowledge graphs. You can check the proposing article if you are interested in details of the mathematical implementation.

The following code trains the PairRE model based on the subset of the Hetionet dataset you were provided.

Listing 11.5. Train a PairRE model.

```
from pykeen.pipeline import pipeline

result = pipeline(
    training=training,
    testing=testing,
    validation=validation,
    #1
    model="PairRE",
    #2
    stopper="early",
    epochs=100,
    #3
    random_seed=42,
)
```

The PairRE model can be trained with a single function, as shown in Listing 11.5. The training, testing, and validation sets are loaded via separate arguments. You can select the model with the `model` argument. There are more than 40 models you can pick from. Check the documentation (https://pykeen.readthedocs.io/en/stable/reference/models.html) for a complete list of available models. The `early` value for the stopper argument evaluates the model every 10 epochs by default. Using the stopper option with `early` value, the training pipeline can stop the training if the model accuracy does not improve with additional epochs. Lastly, the `random_seed` parameter is used to ensure result reproducibility.

The complete list of available pipeline parameters is available in the official documentation
(https://pykeen.readthedocs.io/en/stable/api/pykeen.pipeline.pipeline.html#pyl

Note

The training can be performed on either CPU or GPU devices. However, the training will be faster if you have a GPU device available. If you don’t have a local GPU available, you can always try out free cloud environments like the Google Colab.

11.2.4 Drug application predictions

With the PairRE model trained, you can go ahead and predict new applications for Acetylsalicylic acid or better known as Aspirin. The PyKEEN library offers a `get_prediction_df` function, which allows you to input two elements of a triple and outputs the predictions for the third one. In your example, you input `Acetylsalicylic acid` as the head and `treats` as the relation element. The output of the most probably `tail` nodes is given in a Pandas dataframe structure.

Listing 11.6. Predict new potential use cases for Acetylsalicylic acid.

```
from pykeen.models.predict import get_prediction_df

pred_df = get_prediction_df(
    result.model,
    head_label="Acetylsalicylic acid",
    relation_label="treats",
    remove_known=True,
    triples_factory=result.training,
)
pred_df.head()
```

Table 11.2. Top five predictions for Acetylsalicylic acid.

tail_id	tail_label	score
19821	autistic disorder	-16.378204

19895	polycystic ovary syndrome	-16.702688
19897	prostate cancer	-17.108332
19886	ovarian cancer	-17.225809
19866	malignant glioma	-17.290039

Predictions with `score` value closer to zero are more probable. Your model predicted that Aspirin could be potentially used to treat autistic disorder, polycystic ovary syndrome, and some forms of cancer. These predictions can be used to recommend a clinical trial for a particular drug use case. The clinical trials must be carefully planned as they take a long time and are incredibly costly [Schlander et al., 2021]. Therefore, it is essential to produce as accurate recommendations as possible, as the cost of clinical trials can reach more than a billion dollars.

Exercise 11.2

Predict potential new applications for `Caffeine` with the `get_prediction_df` function.

11.2.5 Explaining predictions

After the predictions have been made, you can search the medical literature for supporting or invalidating research. For example, if you search for a combination of Aspirin and prostate cancer, you can find some articles that might validate your predictions [Joshi et al., 2021]. Given that the Hetionet article was published in 2017, it probably does not contain new medical information from 2021. Hetionet is an aging resource that was restricted to less than 200 diseases. In practice, pharmaceutical and other companies use various text-mining systems deployed at scale to extract knowledge from various medical research articles and trials to keep their biomedical graphs updated with all the latest available information [Bachman et al., 2022].

Having supporting evidence for your predictions shows that the method of using knowledge graph embedding models for knowledge graph completion can yield great results. Suppose you found no supporting literature for your predictions. In that case, you could present existing biomedical connections to domain experts and let them decide if it holds any merit. Even though you didn’t need a graph database for the drug repurposing workflow, it would still be great for explaining predictions. Luckily, your coworkers at the large pharmaceutical company have you covered, or in reality, the authors of the Hetionet have made it available through a read-only Neo4j Browser interface. The Hetionet Browser interface is available at https://neo4j.het.io/browser/.

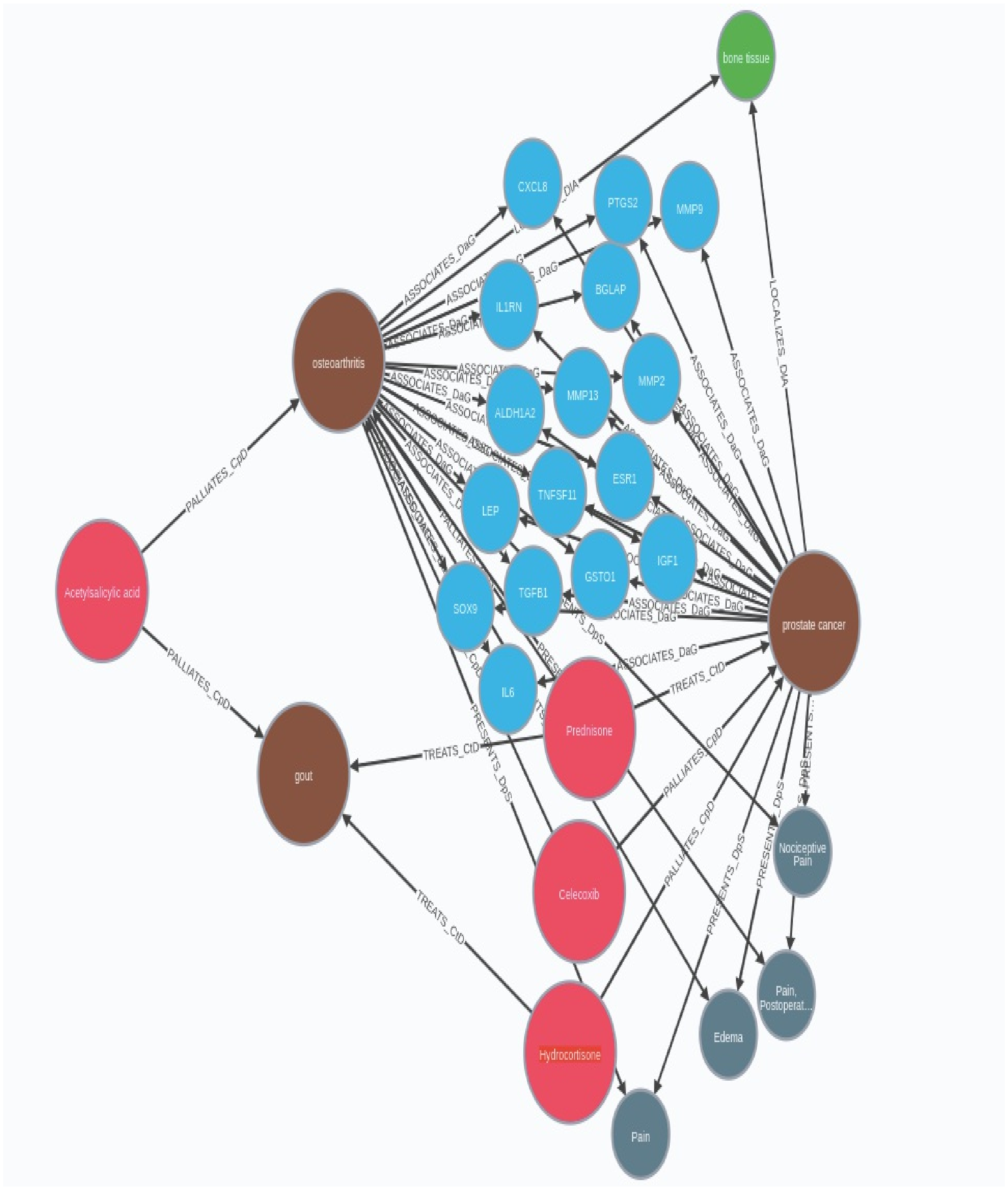
The following Cypher query will visualize the first 25 paths between Acetylsalicylic acid and prostate cancer that are up to 3 hops away.

Listing 11.7. Predict new potential use cases for Acetylsalicylic acid.

```
MATCH (c:Compound {name:"Acetylsalicylic acid"}),
      (d:Disease {name:"prostate cancer"})
MATCH p=(c)-[* ..3]-(d)
RETURN p LIMIT 25
```

The Cypher statement in Listing 11.6 produces the following visualization

Figure 11.14. Existing connections between Acetylsalicylic acid and prostate cancer.



There is the `Acetylsalicylic acid` on the left side of the Figure 11.14, while the `prostate cancer` is on the right. Acetylsalicylic acid can be used to palliate osteoarthritis and gout. Interestingly, osteoarthritis associates with similar genes (blue color nodes) as prostate cancer. Another connection is that existing drugs like Prednisone, Celecoxib, and Hydrocortisone can be used to treat or palliate gout and osteoarthritis. Since Acetylsalicylic acid can also be used to treat gout and osteoarthritis, it could perhaps also be used to treat prostate cancer due to a number of drugs that can treat or palliate both. In any case, a domain expert can evaluate existing connections and make up their own mind. There are 1716 distinct paths with a length of up to three hops between Acetylsalicylic acid and prostate cancer. Therefore, it is hard to visualize them all in a single image, and a domain expert could prioritize connections based on node or relationship types.

Exercise 11.3

Visualize the first 25 paths with a length of up to three hops between Acetylsalicylic acid and autistic disorder. Use the existing Neo4j version of Hetionet graph, which is available through Neo4j Browser at https://neo4j.het.io/browser/.

11.3 Summary

- A heterogeneous or multipartite graph consists of multiple node and relationship types. There could also be numerous relationship types between two entity types.
- A triple data object is used to represent directed graphs, where multiple relationship types are present.
- A triple data object consists of head, relation, and tail elements.
- Knowledge graph embedding models encode nodes and relationships in the embedding space as opposed to node embedding models that only encode nodes.
- Knowledge graph embedding model try to calculate embedding in such a way that for every existing triple the sum of embeddings of head and relation are close to the embedding of the tail node.
- Knowledge graph embedding models are evaluated from a theoretical perspective if they can encode symmetry, inverse, composite, and 1-to-N relationships.
- PairRE model can encode all four categories (symmetry, inverse, composite, and 1-to-N) of different relationships.
- Knowledge graph completion can be thought of as a multi-class link

prediction problem, where you are predicting new links and also their type.
- In a drug repurposing workflow, the predictions have to evaluated by domain experts and then validated through clinical trials in order for them to get approved. The knowledge graph completion is only used to prioritize the most likely candidates.

11.4 References

[Vane, 1971] VANE, J. Inhibition of Prostaglandin Synthesis as a Mechanism of Action for Aspirin-like Drugs. Nature New Biology 231, 232–235 (1971). https://doi.org/10.1038/newbio231232a0

[Rothwell et al., 2011] Rothwell, Peter M et al. “Effect of daily aspirin on long-term risk of death due to cancer: analysis of individual patient data from randomised trials.” Lancet (London, England) vol. 377,9759 (2011): 31-41. doi:10.1016/S0140-6736(10)62110-1

[Rüschoff et al., 1998] Rüschoff, J et al. “Aspirin suppresses the mutator phenotype associated with hereditary nonpolyposis colorectal cancer by genetic selection.” Proceedings of the National Academy of Sciences of the United States of America vol. 95,19 (1998): 11301-6. doi:10.1073/pnas.95.19.11301

[Bordes et al., 2013] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., & Yakhnenko, O. (2013). Translating Embeddings for Modeling Multi-relational Data. In Advances in Neural Information Processing Systems. Curran Associates, Inc..

[Ali et al., 2021] Ali, M., Berrendorf, M., Hoyt, C., Vermue, L., Sharifzadeh, S., Tresp, V., & Lehmann, J. (2021). PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings. Journal of Machine Learning Research, 22(82), 1–6.

[Himmelstein et al., 2017] Himmelstein, D., Lizee, A., Hessler, C., Brueggeman, L., Chen, S., Hadley, D., Green, A., Khankhanian, P., & Baranzini, S. (2017). Systematic integration of biomedical knowledge prioritizes drugs for repurposing. eLife, 6, e26726.

[Chao et al., 2020] Chao, L., He, J., Wang, T., & Chu, W.. (2020). PairRE: Knowledge Graph Embeddings via Paired Relation Vectors.

[Schlander et al., 2021] Schlander, M., Hernandez-Villafuerte, K., Cheng, CY. et al. How Much Does It Cost to Research and Develop a New Drug? A Systematic Review and Assessment. PharmacoEconomics 39, 1243–1269 (2021). https://doi.org/10.1007/s40273-021-01065-y

[Joshi et al., 2021] S.N. Joshi, E.A. Murphy, P. Olaniyi, & R.J. Bryant (2021). The multiple effects of aspirin in prostate cancer patients. Cancer Treatment and Research Communications, 26, 100267.

[Bachman et al., 2022] Bachman, J., Gyori, B., & Sorger, P. (2022). Automated assembly of molecular mechanisms at scale from text mining and curated databases. bioRxiv.

11.5 Solutions to exercises

For me, the solution to Exercise 11.1 is the following:

```
Tomaz, LIVES_IN, Ljubljana
Ljubljana, PART_OF, Slovenia
```

The solution to Exercise 11.2 is the following:

Listing 11.8. Predict new potential use cases for Caffeine.

```
pred_df = get_prediction_df(
    result.model,
    head_label="Caffeine",
    relation_label="treats",
    remove_known=True,
    triples_factory=result.training,
)
pred_df.head()
```

The solution to Exercise 11.3 is the following:

Listing 11.9. Predict new potential use cases for Acetylsalicylic acid.

```
MATCH (c:Compound {name:"Acetylsalicylic acid"}),
      (d:Disease {name:"autistic disorder"})
MATCH p=(c)-[* ..3]-(d)
RETURN p LIMIT 25
```