APPCODA

Mastering Machine Learning with Core ML and Python

A comprehensive guide to understanding machine learning and developing AI-based apps for iOS

Fully Supports Xcode 12 & iOS 14

Vardhan Agrawal

Table of Contents

Introduction

Preface

- Who Should Read This Book
- How to Use This Book
- Conventions and Organization
- Where to Download the Projects

Chapter 1 - Machine Learning at a Glance

- 1-1 What is Machine Learning
  - Machine Learning's Role
  - Methods of Machine Learning
- 1-2 Types of Algorithms
  - Linear Regression
  - Decision Trees
  - Support Vector Machines
  - K-Nearest Neighbors
- 1-3 Machine Learning APIs
  - Implementation Tools
  - Training Tools

Chapter 2 - Building an Image Classification App

- 2-1 Preparing your Project
  - Setting up Xcode
  - Create a New Project
  - Save your Project
  - Preparing to Debug

User Interface Design
Auto Layout
2-2 Pre-Processing Video Feed
Connecting to Code
Interface Builder Outlets
Using the Assistant Editor
Camera Usage Description
Capture Session
Session I/O
Preparing for Processing
2-3 Image Classification and Labeling
Image Classification
Displaying the Results
Chapter 3 - A Primer on Python and Jupiter Notebook
3-1 Jupyter Notebook and Anaconda
What is a Jupyter Notebook?
Installing Anaconda
Creating a Jupyter Notebook
3-2 Basic Python
Variables, Loops, and Control Flow
Data Structures and Functions
Classes, Methods, and Objects
3-3 Uses of Jupyter Notebook
Code and Prose
Data Visualization
Markdown and LaTeX
Chapter 4 - Training Your Own Image Classifier

4-1 Preparing Training Data
Finding the Images
Image Preparation Tips
Getting the Training Data Ready
4-2 Training with Turi Create
Using Turi Create
Training with Turi Create
4-3 Delving into Create ML
Introducing Create ML
What Models Can You Build?
Training with Create ML
Chapter 5 - Natural Language Processing
5-1 What is Natural Language Processing?
Natural Language at a Glance
Challenges of NLP
Types of NLP
5-2 Tokenization, Stemming, and Lemmatization
Understanding and Responding
Internal Steps and Techniques
5-3 Training a Text Classifier
Finding and Sourcing Data
Text Classification Algorithms
Creating your Classifier
Chapter 6 - Sound Classification Models
6-1 Overview and Gathering Sounds
About Sound Classification
Recording Sounds Yourself

Sourcing Sounds from Datasets
6-2 Training a Sound Classifier
Training using Create ML
Testing your Model
6-3 Implementing the Model
Building the Sound Classification App
Capturing and Processing Live Audio
Chapter 7 - Cloud-Based Machine Learning with Firebase
7-1 Firebase at a Glance
Basic Uses of Firebase
Account Creation and Setup
CocoaPods Dependency Manager
7-2 Barcode Scanning and Image Labelling
Starting a Live Feed
Barcode Scanning
Image Labelling
7-3 Translating Between Languages
Creating a Text Experience
Connecting Interface to Code
Language Translation
Chapter 8 - Updatable ML Models for On-Device Training
8-1 How Updatable ML Models Work
User Experience
Transfer Learning
Updatable Models at a Glance
8-2 Open-Source Training Code
Obtaining the Code

Preparation for Training

8-3 Training the Model

Code Explained

Chapter 9 - Action Classification and Style Transfer

9-1 Action Classification

Obtaining Videos

Training the Model

9-2 Image and Video Style Transfer

About Style Transfer

Uses of Style Transfer

9-3 Training your Model

Training, Adjusting, and Testing

Chapter 10 - Supercharge ML Workflow with these Tips and Tricks

10-1 Machine Learning Design

Choose Datasets Carefully

Use Data Augmentation

Embrace Limitations

10-2 Model Conversion and Usage

Using Model Conversion

Using Models Without Core ML

Published by AppCoda Limited

Written by Vardhan Agrawal

Edited by Kelly Chan

ISBN: 978-9-887535-00-3

About the Author

Vardhan Agrawal is a self-taught developer who works primarily with Swift and iOS development. Some of the other languages Vardhan is well-versed with include Java, Javascript, Objective-C, HTML, and CSS. Not only does he tinker with new projects and ideas, but he also makes it possible for others to do the same by teaching others to code. Vardhan is a part of the tutorials team at raywenderlich.com, Envato Tuts, Heartbeat by Fritz, and freeCodeCamp. He has also been recognized for his dedication to social good. He helps disadvantaged students learn to code through TheOpenCode Foundation.

He is also active in the open-source community and has been personally recognized by GitHub for his contributions. Vardhan has also developed several ingenious products to better the lives of people around the world. Whether it's helping the local city solve their chronic street light problem with drones, developing a novel blood pressure monitoring device, or creating a breakthrough in space medicine, Vardhan has a penchant for turning his far-fetched ideas into scientific realities. When he's not coding, you can find him producing and editing videos, working on getting his private pilot license, or playing the violin.

You can reach him via LinkedIn at https://www.linkedin.com/in/vardhanagrawal or Twitter at https://twitter.com/vhanagwal. He can also be found on his website, www.vardhanagrawal.com.

About the Xcode Version

All source code and projects are compatible with Xcode 12 and iOS 14. Xcode 12 is required to follow along with this book. While macOS Big Sur isn't required, it is highly recommended. You can find all appropriate software on the Mac App Store.

- Xcode 12 (https://apps.apple.com/us/app/xcode/id497799835)
- macOS Big Sur (https://apps.apple.com/us/app/macos-big-sur/id1526878132)

Preface

Machine learning, now more than ever, plays a pivotal role in almost everything we do in our digital lives. Whether it's interacting with a virtual assistant like Siri or typing out a message to a friend, machine learning is the technology facilitating those actions. It's clear that machine learning is here to stay, and as such, it's a vital skill to have in the upcoming decades. Since more users will be reliant on such technology, it's important to dive in and hone your machine learning skills to make your apps more engaging, fun, and useful for your users!

Core ML is Apple's end-to-end platform which allows for fast, powerful, and on-device machine learning in your iOS apps. Core ML is a Swift-based framework which allows developers to deploy lightweight models and ship them with their apps. As Core ML continually gets betslter, more developers will begin using it because of it's close-knit compatibility with Apple's hardware.

In this book, I'll not only be covering Core ML in-depth, but I'll also be teaching you about Turi Create, Create ML, Keras, Firebase, and Jupyter Notebooks, just to name a few. These are a few examples of professional tools which are staples for many machine learning experts. Throughout this book, you'll also become proficient with Python, the language that's most frequently used for machine learning. By the end of the book, you would have created a handful of ready-to-use apps such as barcode scanners, image classifiers, and language translators. And, you would have mastered the ins-and-outs of Core ML!

I've been writing Swift since the day it was announced, back in 2014. Ever since, it has become one of my favorite languages to program in. I've also spent the last few years writing high-quality technical tutorials with reputed tutorial sites including Envato Tuts, Heartbeat, freeCodeCamp, and raywenderlich.com. Ranging from machine learning to augmented reality, my tutorials teach readers advanced computer science skills, while neither requiring nor assuming any prior experience.

What I believe is that readers should *never* be asked to paste in code without understanding every line of it. My unique approach lets readers understand the purpose of every character they're typing, and that same approach is used in this book. As a software developer, my responsibility isn't only to code amazing things; that's less than half of the picture. I'm responsible for welcoming newcomers to the world of software development and guiding them every step of the way. Through writing this book, I'm able to reach hundreds of thousands of readers like you, and by the end of the book, you'll know enough to think of a problem and dive straight into engineering the solution.

Whether you're a tech enthusiast or a computer science major, you're sure to get value from reading this book; I'll make sure you'll understand what you're doing and why you're doing it throughout the book. Whether it's building an image classifier or an instrument sound recognizer, my hope is that you'll read all of the explanations and understand the code before typing it out.

Becoming well-versed with machine learning has brought me a lot of opportunities and has given me the ability to solve problems in my community with code, which I likely wouldn't have been able to do otherwise. My goal in writing this book is to reach people from around the world and give them the same opportunities to use computer science in a positive way. Whether you have an app idea, or you just want to get your feet wet in building a machine learning app, this book will take you from where you currently are to a master at building machine learning apps.

After nearly two years of wriggling out of parties, shunning movie theaters, and spending time indoors, I've finished writing this book, which I hope you'll enjoy. Also, I'd like to thank you for purchasing this book! If you're reading this right now, you've helped make all of the time and effort I've spent on this book worth it. And finally, if you have any questions or just want to share something cool you've made, I'd be happy to hear from you! Just Tweet me @vhanagwal or send me a note at vardhanagrawal22@gmail.com. Have fun on your machine learning adventure, and happy coding!

Sincerely,
Vardhan K. Agrawal

Who Should Read This Book

While this book *is* beginner-friendly, it would be helpful to have a working understanding of Swift. Since this book is a machine learning book, you don't have to have any experience with machine learning whatsoever; however, if you haven't coded in Swift before, I highly recommend picking up those skills before reading this book.

If you don't know Swift but know another programming language, you could still follow along; however many of the Swift-specific nuances may not make sense to you. If this is the case, reading **Chapter 2** is a must, since it covers more fundamental topics in iOS development than other chapters.

Since this book is based on Swift and Apple's lineup of products, most chapters require that you're running macOS Catalina (10.15) or later. In addition, the image processing chapters require that you use a physical device to test your apps. If you don't have access to such iOS devices, don't fret. You can still create your apps; however, you won't be able to test them.

How to Use This Book

The chapters in this book are approximately the same length, and the best way to read this book is cover-to-cover. Unlike many other technical books out there, I'll be using a more conversational tone—to help you stay engaged and follow along.

Some chapters, of course, are standalone chapters; however many of them build on each other. If you feel that you're unable to read all of the chapters, or if you're already familiar with a concept, feel free to skip it. If you're getting stuck, you can always revisit the ones you missed. Here's a quick outline of what you'll learn:

- **Chapter 1** teaches you about machine learning from a birds-eye view. You'll explore its applications, different model types, and machine learning APIs.
- **Chapter 2** brings in the hands-on approach that this book is centered around by taking you through building your own image classification app.

- **Chapter 3** delves straight into Python and introduces you to Jupyter Notebooks to prepare you for the upcoming Turi Create chapters.
- **Chapter 4** teaches you to build an image classifier in two ways: through Turi Create and Create ML.
- **Chapter 5** switches gears and teaches you about natural language processing theoretically and then how to build a text classification model.
- **Chapter 6** will introduce yet another application of machine learning: sound classification. You'll first learn about it and then build your own sound classifier.
- **Chapter 7** brings a fresh perspective to mobile machine learning by guiding you through building a couple of apps using a cloud-based API called Firebase.
- **Chapter 8** introduces a more advanced topic: updatable models. This chapter will later guide you line-by-line through an open-source Jupyter Notebook.
- **Chapter 9** introduces action classification and style transfer, two emerging technologies which have the potential to revolutionize machine learning's role in various fields from gaming to art.
- **Chapter 10** reviews and reflects on what you've learned throughout this book and provides tips and tricks to supercharge your machine learning workflow.

If you're looking to *only* learn a particular skill, first check what that skill requires. For example, if you're looking to learn a skill which requires an understanding of Python, it would be helpful to read **Chapter 3** beforehand. Similarly, if that chapter takes you though building an app, it would be best to read **Chapter 2** before reading that chapter.

If you just want to learn the basics, then you can stop reading at **Chapter 6**, but if you want to hone in on more advanced skills, then it's best to stick around until the end. Again, though, the recommended method is to read the entire book cover-to-cover and then go back to the chapters which you need to review.

Conventions and Organization

While writing this book, I've used the following conventions to make the chapters both engaging to read and easier to skim:

Bolded Text is used for:

- important keywords
- new terminology
- file and extension names

`Inline Code` is used for:

- single lines of code
- method and function names
- classes and structs
- filenames, on occasion

Italics is used for:

- providing emphasis
- avoiding pitfalls
- other non-technical uses

Level 4 headers are for:

- providing specific details about a topic
- outlining steps during hands-on lessons

So, if you're ever skimming back through a chapter, keep a lookout for these different types of stylized text. They'll help you get to the information you need much faster.

Where to Download the Projects

This book comes with a lot of hands-on projects to help you understand the machine learning and the integration with iOS apps. All the starter and complete projects are downloadable for all chapters. You can click the link below to download it:

https://link.appcoda.com/mastering-ml-projects

I encourage you download the projects first, so you can follow the rest of the content easily.

Chapter 1
Machine Learning at a Glance

Welcome to the first chapter of this book! I'm certain that you will learn a lot from this book, and by the end, you'll be able to create your own computer vision models, natural language models, and sound processing models to contribute to the ever-growing machine learning community. I'm excited to share the latest and greatest in machine learning technology with you, and I hope you enjoy it!

In the first chapter, you'll learn about how machine learning affects you in your day-to-day business and how you interact with it as you go about your life. You'll also learn about cutting-edge applications of the technology. Next, you'll learn about various types of algorithms, including Linear Regression models, K-Nearest Neighbor models, and Decision Trees.

Last, you'll explore various tools and technologies which will supercharge your machine learning workflows and allow you to train robust, powerful models for your projects. Some of them are beginner-friendly, while some are more advanced. Either way, this chapter is packed with useful information. I suggest returning to it after you've read some of the other chapters to reabsorb what may not have made sense at the beginning. Onwards!

1-1 What is Machine Learning?

Before we dive into learning about different machine learning algorithms and minor implementation details, let's learn about what machine learning actually is and grasp a better understanding of its function in the society. Machine learning, in the simplest terms, is the analysis of statistics to help computers make decisions base on repeatable characteristics found in the data.

A computer identifies these patterns to form an abstract understanding of them, called a *model*, and analyze new data for the same patterns. In other words, it's when a computer performs actions without specifically being told to do so — it's given a set of data and an expected outcome for it to *infer* the path to getting there.

Machine Learning's Role

Machine learning is better now than ever, and is constantly evolving. Many of the things we use on a day-to-day basis rely on machine learning without even knowing it!

Whether it's having your phone predict what you're going to type next, summoning Siri, or seeing relevant ads in your Facebook feed, machine learning is all around us. And now is a great time to learn how to use it to enhance your current apps or make robust new apps.

Medicine

In the medical field, machine learning is being used to recommend lifestyle changes, analyze readings, and even diagnose diseases with high accuracy. Already, many amazing things are possible using the power of machine learning, for example:

- **Irregular Heartbeats** can be recognized while wearing your Apple Watch on a day-to-day basis. Your smartwatch can alert you when it, based on a model, thinks that your heart isn't functioning as it should.
- **Disease Risk-Factor** prediction can be done using prior patient data with characteristics such as age, gender, weight, and more.

Finance

The field of finance is all about numbers and analytics, isn't it? Machine learning can play a massive role in improving the way this field works. Using RNNs, a type of model we'll learn about later in the book, analysts can make a lot of meaningful predictions!

- **Stock Patterns** can be predicted using a model trained with several years of past market data. As unreal as it may sound, this can be done quite accurately using the right machine learning techniques.
- **Loans** can be granted based on the calculated risk for financial institutions, based on characteristics of the customer. This can minimize risk and maximize profit for institutions who grant loans.

Advertising

Nowadays, advertising is a profitable field. This is partially due to the fact that machine learning can match those who supply products and services with those who need them.

- **Social Media Ads** take user data about the pages they follow and the posts they like to make a machine learning model to predict the types of products and services they would be most interested in.
- **Pixels** are bits of code that companies install on their webpages to track data about

customers to retarget them later and make them more likely to buy while minimizing costs.

Clearly, machine learning has a wide variety of applications in our daily life, and there are many opportunities to innovate different fields. Machine learning and artificial intelligence aren't going anywhere soon, so let's continue the journey to learn machine learning. By the end of the book, you'll be a pro!

Methods of Machine Learning

Just as there are many applications of machine learning, there are many ways to train models. There are dozens of ways to represent them, and we'll learn about the specific different machine learning algorithms later in the book. They can be categorized into three main methods: *supervised learning, unsupervised learning, and reinforcement learning*.

Supervised Learning

Supervised learning, as the name suggests, means the model is monitored to ensure accuracy during training. When you learn something in a classroom, you are supervised by an instructor to make sure what you're learning is right and to correct any mistakes you may have made.

Similarly, during supervised learning, all data given to the model has been pre-labeled manually by a human to "give" the model the correct answer. For example, if you wanted to make a model to identify various breeds of dogs, you would give the model a set of images which are already tagged with labels such as "Labrador Retriever" or "Australian Cobberdog" to ensure its accuracy. However, some of the labeled images are not shown to the model using training. These images will be used to test the model later on. Without exposing the labels of these images, the model will try classifying them, and its accuracy can be easily calculated.

Supervised learning is useful for classifying data, as well as predicting a specific output based on one or more inputs. Sometimes though, a large enough dataset of pre-labeled images may not be available, and that's where the other learning types come in.

Unsupervised Learning

From what you've learned so far, you may have noticed that machine learning boils down to one thing: patterns. Machine learning models are amazing at finding patterns, and can sometimes even perform better than a human! In some cases, the correlation in the data that you're presenting to the model may be too complex for you to pre-determine, so there is no way of providing pre-labeled data to the model. In this case, it's up to the model to find relationships between each data point provided.

The model can tell you where the patterns lie and also guide you to different arrangements of images, but it cannot tell you what the patterns actually mean; that analysis is left up to you, the human.

Back to the dog breed example, if you didn't have the expertise to accurately name each dog breed, the neural network could find associations between the dog's tail length, coat color, and muzzle size. This, again, leaves the analysis to you, but it does find meaningful patterns which you can use to make something useful.

There are a few different ways an unsupervised model can use to group, or organize the given data, two of which include:

1. Using **unexpected patterns** to **eliminate** data in a dataset.
2. Using **characteristics** of the data to **group data** together.

There are, of course, several other ways as well, but these are the most commonly used, and they help us make sense of the unsupervised learning model. However, there is a problem with this kind of model, that is you can't be sure whether it's "right" or "wrong" because there is no defined outcome. This can make it difficult to calculate its accuracy as compared to the supervised learning model.

Reinforcement Learning

You can think of a reinforcement learning model like a child; it acts based on a *reward* or *incentive* system, making it constantly correct its own mistakes when being punished and learn good behaviors when being rewarded. Unlike supervised learning and unsupervised learning models, a reinforcement learning model takes a goal, and uses incentive to find the best path to that goal.

Interestingly, this model is able to correlate between current actions and future outcomes, similar to how a human thinks. By using this incentive system, the reinforcement model is able to improve its performance every time.

The more times that it runs, the more accurate the model gets; closer to the optimal path to the end goal! This is used for varios fields such as navigating roads in driverless cars, and for other applications such as competing in video games against even the best players.

As you can see, machine learning has many applications, and comes in many different forms. As you continue through this book, you'll learn about machine learning in depth and how to apply it to your daily life.

1-2 Types of Algorithms

In the previous section, you learned about machine learning's role in society as well as three types of machine learning methods: supervised learning, unsupervised learning, and reinforcement learning.

Since there are so many different use cases for machine learning, there are many different algorithms that suit each of these use cases. In this section, we'll go over some of the most common types of machine learning algorithms and look at some examples where they might be used.

Linear Regression

Linear regression models are the most well-known and frequently used types of machine learning models. You might remember hearing the name in your statistics class during high school or college. It makes sense; machine learning is all about statistics anyway, isn't it? Think of a linear equation:

$y = mx + b$

You realize that there is exactly one y-value for every x-value. These variables are also commonly known as **dependent variables** and **explanatory variables**. Linear regression aims to relate these two variables using a *line of best fit* to make meaning of the data.

Regression Equation

Though the line can also be modeled using the *slope-intercept form*, the following equation is also used in the field, and it's commonly known as the **regression equation**:

$$y = \beta_0 + \beta_1 x$$

It's not much different—it essentially means the same thing, but it has a slightly different style of notation. In this book, we won't be exploring hyper-plane representations of linear regression models, but this form of the equation can be modified to accept multiple inputs — that is — more than one *x-value*.

ε is the **epsilon** or **residual** value, which represents the margin of error in the data. The smaller this value is, the closer the data points are to the *line of best fit*.

If you want to take into consideration this value in the equation, you can simply add it to the end of the equation as follows:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

This is, in actuality, a more complete version of the above equation. Here's a quick summary of what each of these terms represent:

- β_0 is **y-intercept** of the data set.
- β_1 is the **slope** of the line of best fit.
- x is the **explanatory variable** (independent).
- ε is the **error** value.

Manually Calculating Coefficients

If we go back to our simpler equation, the *slope-intercept form*, we can learn how to manually calculate each coefficient in the equation to create our own equation. If you can do it by hand, you will know you can program your computer to do it! As a reminder, the equation is:

$y = mx + b$

In order to find the equation, you'll need a dataset. The table below has a small, manageable dataset for you to follow along with. Of course, you can use your own dataset, and things will work the same.

Subject	**Age (x)**	**Glucose**	xy	x^2	y^2
1	43	99	4257	1849	9801
2	21	65	1365	441	4225
3	25	79	1975	625	6241

Figure 1-1: Linear Regression Data Table

You'll see that **age** is the **explanatory variable**, and **glucose** is the **dependent variable**. On the right, you'll notice that there are some pre-calculated values for xy, x^2, and y^2. In your own data, you may need to calculate these yourself, but they're provided here just to spare you the arithmetic.

The equations below represent the **slope value** and the **y-intercept**, and to some, they may look intimidating.

$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$

$b = \frac{(\sum y)(\sum y^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$

Not to worry, though, we'll dissect these equations so that they make sense, regardless of your mathematical background. The symbol $\sum$ or "sigma" usually means the sum of something. In this case, you'll see $\sum$ precede the terms in the table above insert figure number. This means you'll need to add up all of the values in that given column, and plug the values back into the equation.

In short, if you see $\sum xy$, you need to add together $4257 + 1365 + 1975$ and put the value into the equation. Similarly, if you see $\sum x^2$, you need to add $1849 + 441 + 625$.

These equations should help you gain a better understanding of linear regression models. While they're useful for a variety of cases, they won't work for *all* machine learning applications.

Decision Trees

To understand decision trees, you can think about a tree in nature. It starts with a trunk, and then begins to branch off. Each branch has its own set of branches, which has another set and so on.

Usually, a decision tree is visualized as, well... a tree! Unlike a tree, the root of the decision tree is drawn at the top, and it branches downwards. This visual representation can be very helpful in making sense of an otherwise complicated machine learning algorithm. Each branch split represents a question, condition, or decision to be made.

Decision trees are a *supervised* machine learning algorithm, which you learned about earlier in the previous session. The two main types of problems that can be solved with this approach are **regression problems** and **classification problems**.

Benefits of Decision Trees

While overfitting and underfitting can cause your model to malfunction, these issues shouldn't stop you from using decision trees — they're useful in a variety of settings.

Handling Non-Linear Relationships

Many of the other machine learning models require the presented data has a linear relationship; decision trees, however, do not. This is one of the areas where decision trees outperform other types of models.

You can use a decision tree to predict customer behavior, and other obscurely related datasets — you wouldn't attempt to plug the seemingly random actions of your customers into a formula. With a decision tree, you can see what variables are affecting your customers to make choices.

Datasets with Many Outliers

Since most machine learning models rely on some sort of "averaging" system, they tend to be overly sensitive to outliers. For this reason, rawly collected data needs to be filtered and organized before it can be used to train these sorts of models.

With a decision tree, you don't need to worry about processing your data ahead of time. For example, if you were building a stock prediction model, you would have several outliers when the stock markets changed due to interest rates, political events, or other unrelated variables. These outliers would affect other models, but decision trees would remain ambivalent to these special cases. Clearly, these types of models have a lot of benefits, but they also come with their own drawbacks, too.

Disadvantages of Decision Trees

As you saw above, decision trees can be very versatile and can help you when other types of models begin to find correlations that don’t actually exist. However, outside these cases, decision trees have their own drawbacks. They have two major flaws, which can end up making them a poor choice in your application. These issues are called **overfitting** and **underfitting**. When decision trees work, they perform really well, but when they don't, their problems can adversely affect your results. Thus, it is important to explore their issues.

Overfitting

A more common issue in decision trees is **overfitting**. Overfitting occurs when the tree isn't generalized enough, and it's too specific to the training dataset. It may depend on characteristics not found in other data given to the model. If a tree is overfitted, the "noise" or unintentional variation in the data is considered, expecting each new data input to have the exact same types of variation.

For example, if you had a dataset of fruit pictures, you could have a picture of two apples, one banana, and three watermelons. An **overfitted model** may notice a tree in the background in one of the three watermelon pictures and classify the two watermelons differently from the one with the tree in it. This causes unwanted combinations, making the model less accurate.

Underfitting

The less common issue is **underfitting**. If a model is underfitted, it is too general and may not be specific enough for it to be completely accurate. It may funnel two different cases into the same branch, resulting in a completely distorted result.

Going back to the fruit pictures example, if you had an **underfitted model**, the model could notice that the apples are both round and red, and for future images, it may assume that any fruit which is round and red is an apple, which, as you know, isn't true. If you later gave it a cherry, it may misclassify it as an apple, rendering the model useless.

Support Vector Machines

Support Vector Machines, often called SVMs, are also commonly used alongside or instead of linear regression models and decision trees. A support vector machine, at a basic level, is a machine learning model which separates values into their appropriate labels based on a function it determines.

Trees Versus Bushes

As an example, let's take the problem of classifying whether something is a tree or a bush. Most of the time, you wouldn't have any trouble finding the differences on your own. But, imagine that you're walking through a hiking trail one day, and you see a tree which is shaped like a bush; what tells you it's a tree and not a bush? This is where support vector machines shine: they take the data points from each "class" which look the most similar and then draw a "border wall" between them. Any future images are classified based on which side of the "wall" they fall on.

Support Vectors

So, what is a support vector? As complicated as the name may sound, a *support vector* is simply one or more of the data points which are closest to the other classes. In other words, these are the trees which look like bushes or the apples which look like oranges. A Support Vector Machine uses these support vectors to draw an accurate line, plane, or sphere between the data types that the model is being asked to distinguish between.

1D, 2D, or 3D

As you may have guessed, the "border wall" can be drawn in any number of dimensions; it could be a sphere, a parabola, a curve, or simply a straight line. Furthermore, the *hyperplane* (technical term for "border wall"), can be shaped in any way. For example, a linear SVM would use a straight line, while a polynomial SVM might use a parabolic curve to distinguish between groups of data.

K-Nearest Neighbors

Aside from the fact that this type of model is listed last, K-Nearest Neighbor machine learning models are the most basic ones. In fact, it's so simple that many data scientists don't even consider it a machine learning algorithm.

Delayed Learning

K-Nearest Neighbor algorithms, or KNN for short, don't learn on the data you provide them for training. Instead, they compare the input data (from a user, perhaps) directly to the "training" data. Whichever example is closest to the example image gets chosen as the category or classification label. This method of "waiting" until classification stage to actually learn anything is known as delayed learning.

Selecting Neighbors

Now, you might wonder how a KNN model exactly selects its neighbors. The constant, *k*, stands for the number of neighbors that the K-Nearest Neighbors algorithm would compare the target data with. For example, if *k=3* in an instance of a KNN model, an image of a fruit would be compared with three of the neighbors around it. If two or more of the neighbors are apples, it would determine that the fruit is an apple.

1-3 Machine Learning APIs

Now that you're familiar with the three types of machine learning methods and their variety of algorithms for various use cases, including linear regression models, decision trees, and neural networks.

In the first half of this section, you'll learn about different APIs you can use to apply your new conceptual understanding of machine learning. These range from easy-to-use libraries for implementing machine learning in your apps, to robust, enterprise-level tools of the trade. Later in the section, you'll learn about some tools you can use to train models, such as TensorFlow, Caffe, and Turi Create.

Implementation Tools

Since several pre-trained models already exist, let's first focus on the different implementation tools you can use if you already have a model on hand. Of course, training your own models gives you a lot of leverage, but it can be time consuming, especially if you want to get started with machine learning right away!

Core ML

If you're reading this book, you're likely an Apple Developer, meaning you develop apps for iOS, macOS, watchOS, or tvOS. Core ML is an Apple's machine learning framework, which allows you to integrate machine learning capabilities into your app. It is also used across their own products and services, to enable functionalities such as Siri, QuickType, and more. For developers, Core ML offers powerful tools for vision and natural language processing, but its use is currently limited to Apple's own platforms.

Benefits

Core ML operates on local processing as opposed to cloud processing, which has its own set of benefits and limitations. However, the features offered by this framework are best handled on-device. Since network calls aren't necessary, the user doesn't need an internet connection for your app to work. In addition to this, requests can be handled more quickly and securely than if a cloud service was involved. Lastly, many of Apple's latest devices have GPUs dedicated for machine learning tasks, which Core ML allows you to access directly. This significantly boosts performance and reduces the time it takes for machine learning requests to be processed.

Limitations

When programs are stored in the cloud, they don't require storage on the device. This may pose an issue when you have large models, and make complex requests to Core ML; your app binary may be too big for some users to install. Keep in mind, however, that Core ML is still *the best* for all other cases, especially due to its superior integration with Apple devices.

Firebase ML Kit

For those looking for a more cross-platform tool, Firebase ML Kit is definitely the way to go! In fact, I'll be covering it in one of the chapters towards the end of this book. Firebase ML is also great if you're interested in cloud-based machine learning; when your model sizes might be too large to store locally.

Benefits

The ability to use cloud storage for your models and API calls is great; especially since Core ML cannot do that — yet! Additionally, Firebase has created tons of templates for you to use. They work great with basic functions ranging from barcode scanning to image classification with most of the work already done for you. Finally, Firebase is owned by Google, so you know that they're likely to be reliable and stay up-and-running when you need them.

Limitations

A major limitation of Firebase ML Kit is that it isn't made by Apple. While Apple creates excellent hardware, they often only optimize it for their own softwares. For example, Firebase would not be allowed to use the dedicated machine learning chips on the latest iPhones, which will likely limit its performance. Further, if you exceed the free limits, Firebase will begin to charge you; the price is significantly lower than other cloud-based machine learning services, however, it is not free like Core ML.

IBM Watson

If you're looking for more cutting-edge technology, IBM Watson is a great candidate. Core ML and Watson integrate very well together, and Watson gives you more flexibility to improve your models than Core ML, including models which improve over time (not possible in Core ML).

Benefits

IBM's Watson is definitely ahead of many of the other available tools. The company spends a significant portion of their funds for research and development of Watson. It also includes the ability to customize a fully-functional chatbot for tasks such as patient management and customer service.

Limitations

Watson is more tedious to setup and install, and it may be more expensive if you choose to use it in the long run. Since it's primarily business-oriented, Watson's services usually take more expertise in machine learning to use and/or install; however, it isn't impossible. It simply takes more time, dedication, and patience.

Training Tools

At this point, you probably have no idea on how to train a model; and that's why you have this book! The following section has a couple of the most popular tools for training a model, and I recommend returning to this section after reading the chapters on training your own models. It'll make more sense to you later.

TensorFlow

Like Firebase, TensorFlow is owned and operated by Google. It is a widely used open source platform for data science and machine learning applications, including certain math operations and model training. While it had been previously used as primarily a training tool, it has expanded over the past few years to provide other services.

Compilation Time

TensorFlow is known for its great compilation times and comparatively small file sizes. In fact, it can even run smoothly on low-performance devices such as Raspberry Pi, Arduino boards, and on the cloud. If you have limited computing power on your computer, or if you need to develop energy efficient devices, TensorFlow is a great option.

Visualization

One of the major features of TensorFlow is its advanced visualization tools. It's known for generating beautiful and informative graphs, charts, and diagrams. In fact, it has its own online *Neural Network Playground* (https://playground.tensorflow.org) where you can experiment with the values for different “data” and “layers” to see how your imitation model would respond. So, if you need to present your work to employers, clients, or colleagues, TensorFlow is the tool for the job.

Stability and Scalability

Since it’s operated by Google, TensorFlow is known to be reliable. It uses state-of-the art technology and is constantly being updated. One of the other benefits of TensorFlow is that it allows you to scale your models if needed. For example, if you’re selling software which is gaining traction, you can use your existing TensorFlow models and workflow.

Caffe

Caffe, similar to TensorFlow, is a machine learning framework and was developed at the University of California, Berkeley. If you don’t have much prior experience with machine learning, Caffe is not the best option to get started with. However, it’s an excellent tool for professionals or academicians who specialize in machine learning.

Modular Code

Like some of the other frameworks listed, Caffe is open-source, and its code is known to be some of the best when it comes to mutability and ease of customization. Since Caffe is geared towards professionals, it allows for deeper, source-code-level configurations which may be inconvenient with other platforms. In addition, Caffe integrates well with Core ML, and there are tools readily available to easily export your Caffe models as Core ML models.

Image Processing

In addition to being a pro-tool, Caffe focuses on one aspect of machine learning: image processing with convolutional neural networks. In fact, it isn’t particularly good with natural language processing and other types of machine learning applications. So if you choose to use it, do so for CNNs specifically. Berkeley’s Artificial Intelligence lab has worked extensively in this area, and their expertise reflects in the work they’ve done on Caffe.

Community

Caffe is undoubtedly a popular tool among data scientists. Aside from the power and performance it offers, Caffe also has a large community of individuals who’ve shared their open-source Caffe projects. This is especially helpful for beginners, since it helps them get an idea on where to start. When you’re starting out with this tool, consider taking advantage of the openly available projects out there.

Turi Create

Turi, just recently, was an early-stage artificial intelligence startup which was bought by Apple. Since then, Apple has developed it extensively as an easy-to-use, Python-based machine learning tool. In addition, it has excellent support for Core ML, including the ability to export models directly as Core ML models. Turi Create is excellent for both beginners and more advanced data scientists because it's both easy-to-learn and powerful. You'll learn how to use it in later chapters.

Experimentation

Because of its easy task-based syntax, Turi Create allows you to quickly experiment with a wide range of machine learning models. With a few lines of Python, you can be up and running with a fully-functional machine learning model to see which algorithm, architecture, and approach works best for your project before diving head-first into something you don't know will work.

Optimization

Since its owned by Apple, Turi Create shares some of the technologies which allow Create ML to quickly train relatively large datasets on your computer. By this token, you can start out with a small, experimental dataset and quickly move to a larger model without having to change your underlying code. Also, with macOS's eGPU support, you can easily expand your current laptop's training capabilities to train more quickly, if necessary.

Diverse Models

Unlike Caffe, for example, Turi Create offers a wider range of data types, such as images, sound, numbers, and video, and does them all well. It also doesn't "specialize" in a certain area, so you can quickly switch from field-to-field and change projects without having to switch to a different training platform. This is one of the reasons why this book spends time covering Turi Create in detail.

Conclusion

In this chapter, you learned about how machine learning affects you in your day-to-day business and how you interact with it as you go about your life. You also learned about cutting-edge applications of the technology. Later in the chapter, you learned about various types of algorithms, including Linear Regression models, K-Nearest Neighbor models, and Decision Trees.

Lastly, we explored various tools and technologies which can supercharge your machine learning workflows and allow you to train robust, powerful models for your projects. Some of them are beginner-friendly, while some are more advanced. Either way, this chapter is packed with useful information. I suggest returning to this chapter after you've read some of the other chapters to reabsorb what may not have made sense to you at the beginning.

Chapter 2
Building an Image Classification App

In the previous chapter, you learned about various categories of machine learning and the types of algorithms they come in. Before we learn how to train our own models, let's learn how to create an image classification app first. The app will be able to identify a household object in real time, without requiring an internet connection.

For this chapter, we'll be using Core ML, which you learned about in the previous chapter. Google and several other companies have already given us some models we can use, and they'd be more accurate than anything we'll be able to produce. These models have been trained with hundreds of thousands of images, if not millions! As you have learned in Chapter 1, Core ML models are machine learning models that work with iOS apps, and Xcode automatically compiles these sorts of models into Swift code. You'll learn how to create your own iOS app which uses Core ML and a pre-trained model to classify any objects that it detects through your iPhone's camera.

2-1 Preparing your Project

I assume you have some experience programming in Swift using Xcode, but if you're completely new to this development environment, you will first need to download and install Xcode 12. In this section, you'll learn how to download and install Xcode as well as prepare the necessary tools you'll need to follow along with this chapter. Next, you'll configure wireless debugging to use your physical device to test, debug, and run your applications in a real-world setting as opposed to being subject to the limitations of the Xcode simulator.

However, this section is only targeted towards absolute beginners, so if you know how to create an Xcode project and a basic user interface, skip to the **Capture Session** part in **2-2 Pre-Processing Video Feed** and use the starter project (under `Chapter 2/Starter Project` of the project resources. Refer to preface for the download link).

Towards the end, you'll get an in-depth look at the various ways iOS developers design immersive user interfaces via Xcode's all-mighty Interface Builder. And then, you'll use advanced tools to scale your applications to a variety of devices and screen sizes using Auto-Layout, Stack Views, and Constraints.

Setting up Xcode

As you may have guessed, you'll need Xcode to make your app. In case you're not familiar with Xcode, it's an integrated development environment which allows you to create apps for Apple platforms, including iOS, macOS, watchOS, and tvOS. To follow along with the Xcode-related sections in this book, you'll need Xcode 12 or newer.

If you have a version of Xcode older than Xcode 12, you'll need to update it by visiting the Mac App Store. Or if you don't currently have it installed, you can download it for free on the Mac App Store.

Create a New Project

Once you've had the latest version of Xcode, you need to create a new project to house your image classification app. Open Xcode, and click **Create a new Xcode project**.

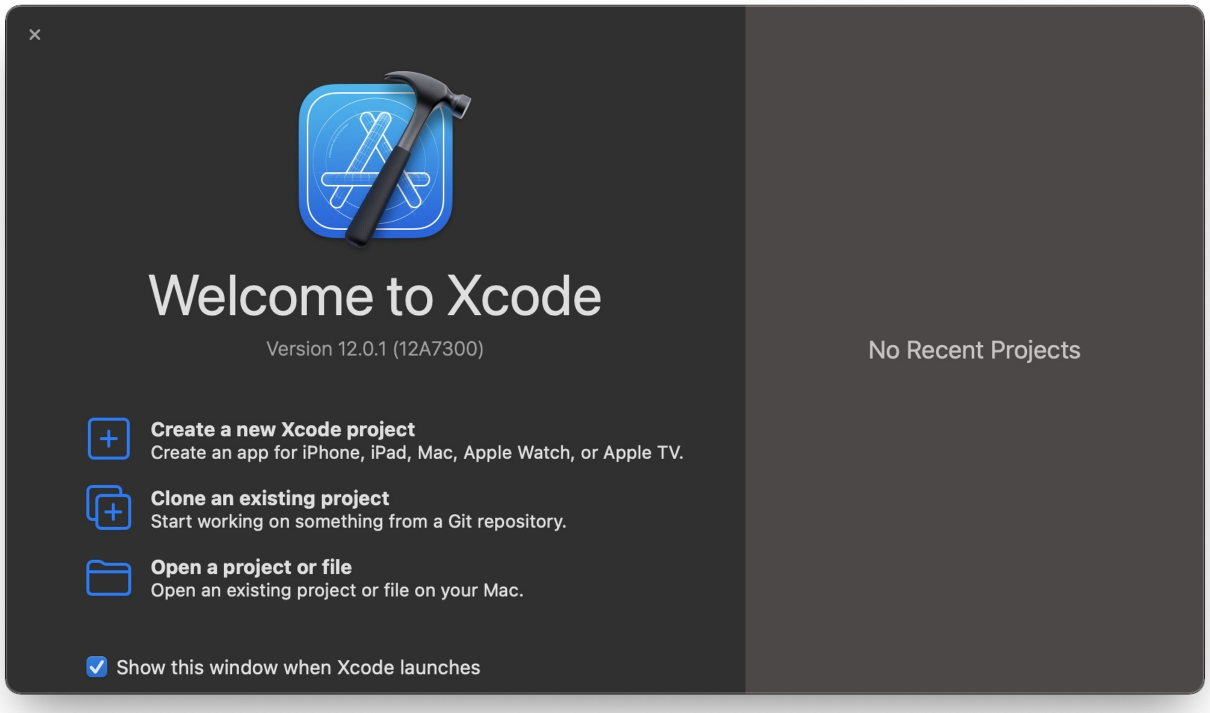

Welcome to Xcode
Version 12.0.1 (12A7300)
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.
Clone an existing project
Start working on something from a Git repository.
Open a project or file
Open an existing project or file on your Mac.
Show this window when Xcode launches
No Recent Projects

Figure 2-1: Launching Xcode

Then, you'll be prompted to choose a template for your app. We'll be starting from scratch, so you can choose **App** under the **iOS** tab, which tells Xcode not to create anything for you, including game graphics, augmented reality templates, or any sample code.

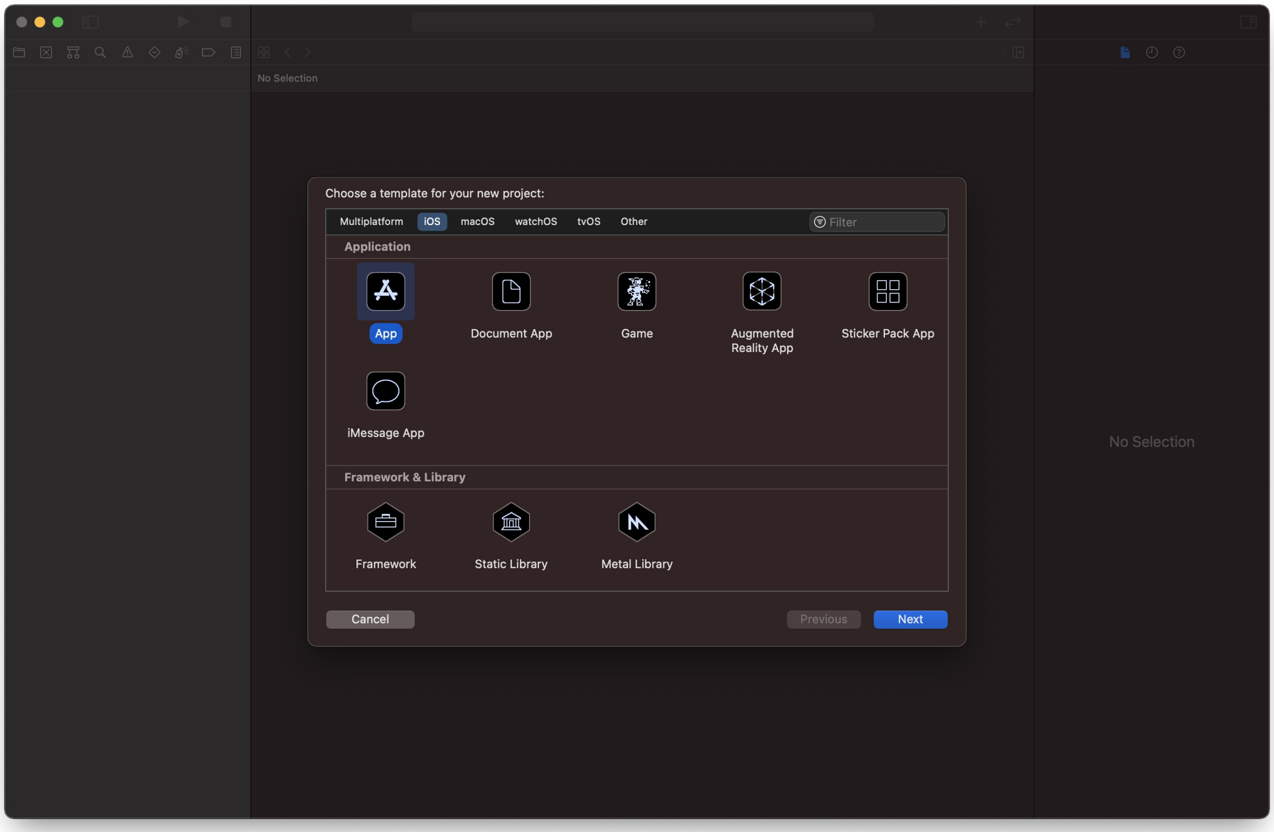
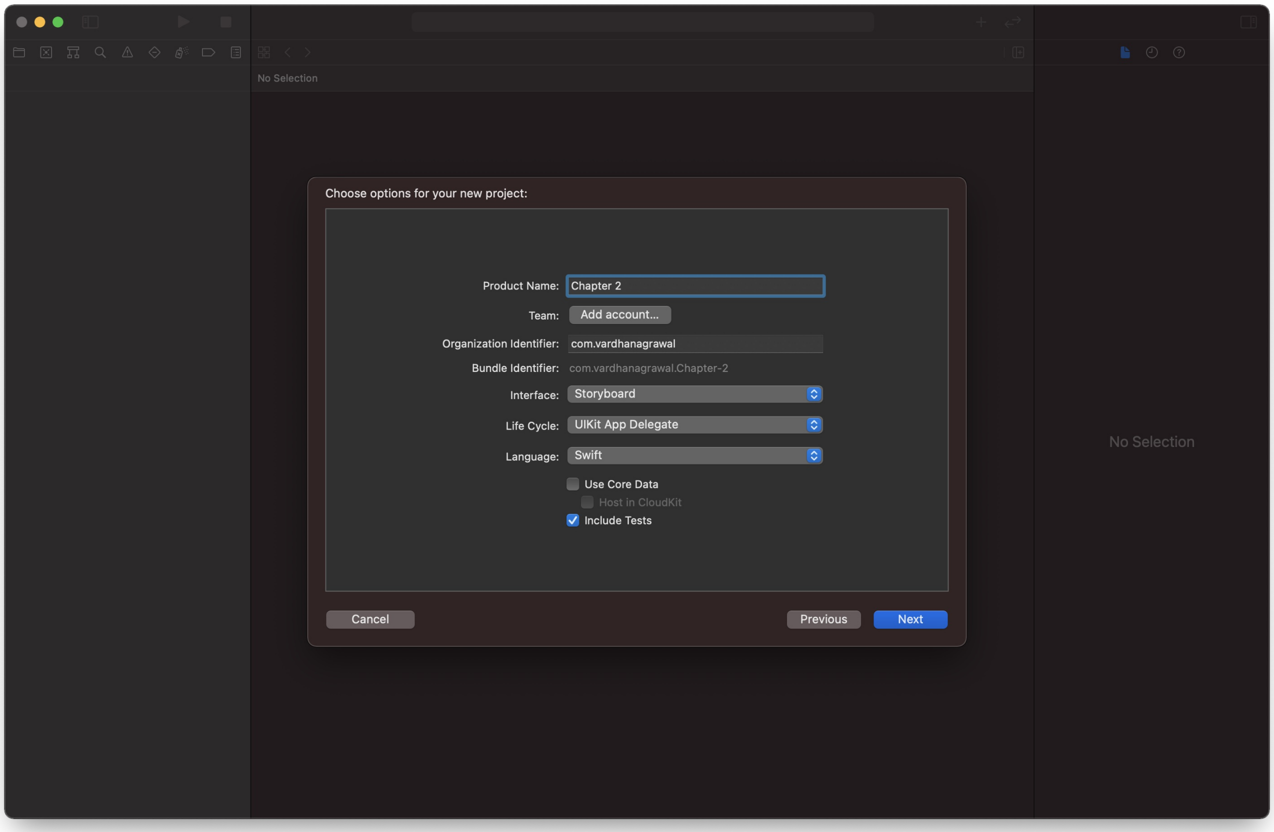


Figure 2-2: Selecting a Template

Save your Project

Next, give your project a useful name. I'll be calling mine **Chapter 2**, since we'll have new projects for each successive chapter in this book. Make sure **Swift** is selected in the dropdown under **Language**, since that's what we'll be using to make our image classification app. For the **Interface** option, please choose **Storyboard**.



No Selection

Choose options for your new project:

Product Name: Chapter 2

Team: Add account...

Organization Identifier: com.vardhanagrawal

Bundle Identifier: com.vardhanagrawal.Chapter-2

Interface: Storyboard

Life Cycle: UIKit App Delegate

Language: Swift

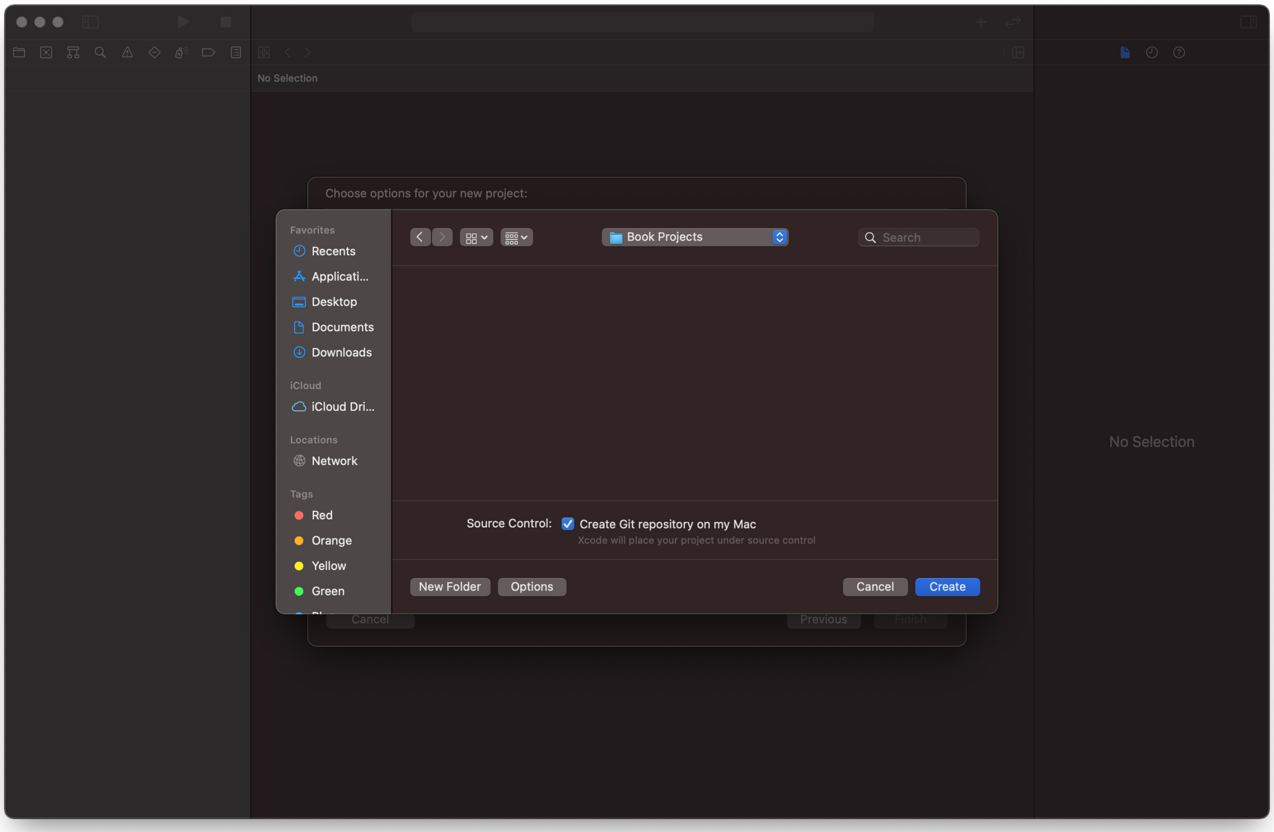
☐ Use Core Data

☐ Host in CloudKit

☑ Include Tests

Figure 2-3: Naming your Project

Last, choose where you'd like to save your project. Choose a safe location, where you'd be least likely to move it. In some cases, moving your Xcode project can cause complications.



Source Control: ✓ Create Git repository on my Mac
Xcode will place your project under source control

Figure 2-4: Saving your Project

Preparing to Debug

Image processing requires the use of your phone's camera (duh, *image* classification). Since the Xcode simulator doesn't have its own camera, you won't be able to use it. Therefore, you'll need an iPhone to follow along. If you don't have one yourself, it would be best to borrow one for this chapter, since you won't be able to catch those annoying bugs without being able to run and test your app.

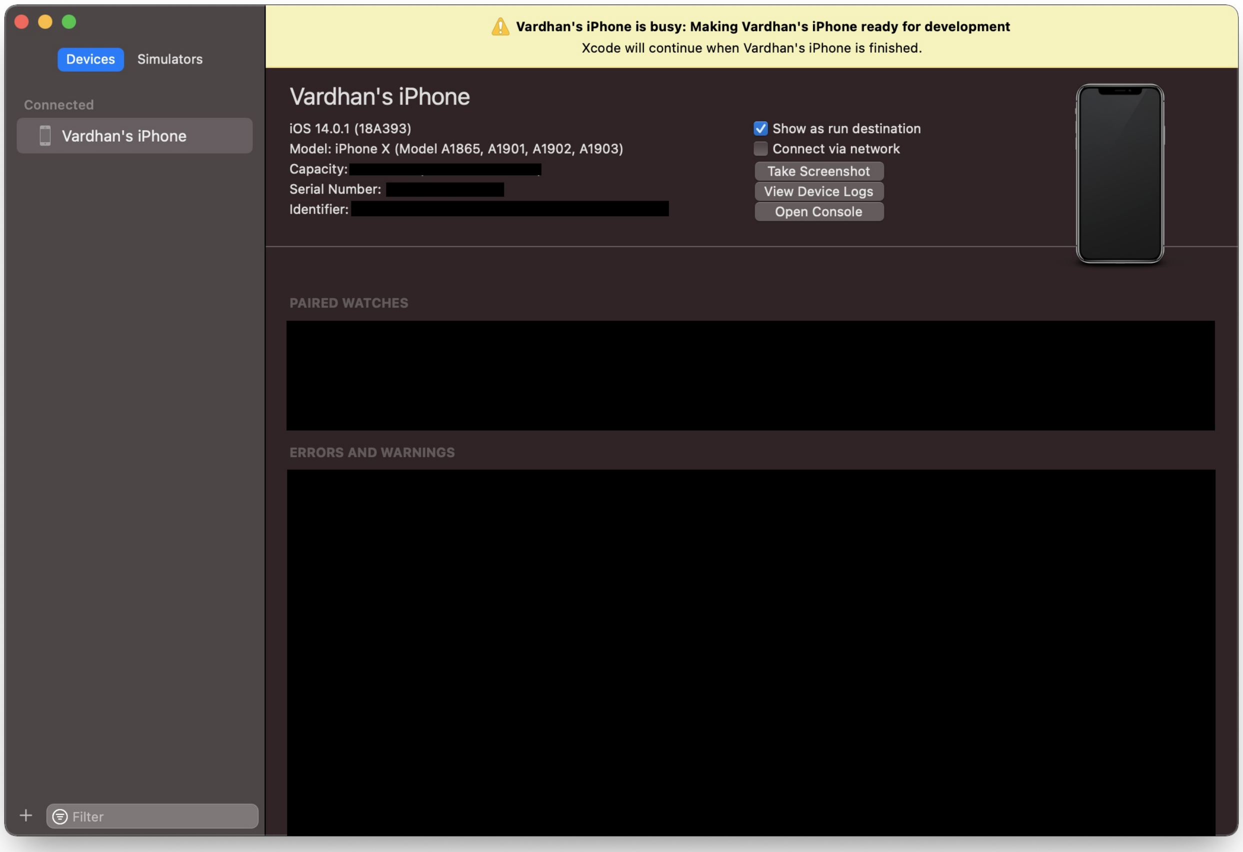
Assuming you already have an iPhone, or have been able to obtain one to follow along with the chapter, let's set it up with Xcode. Previously, you needed to use a cable to connect your target device to your computer to test. But you can now test your apps wirelessly, that is, over Wi-Fi or Bluetooth, as opposed to a physical cable. This makes your workflow as a developer much easier, but it requires some setup.

If you already have your device setup with Xcode for wireless debugging, you can skip ahead, but if not, we'll go through it step-by-step.

Wireless Debugging

To set up Xcode's nifty feature, start by opening Xcode if it has not been opened yet. Then, head over to the menu bar on your Mac and select **Window > Devices and Simulators**. The **Devices and Simulator** window will be opened by then.

Plug your iPhone into your Mac using a lightning cable, and you should see your device pop up in the left side pane of your window. On the right, it'll show you details about your phone, such as the model, the amount of storage, and the serial number. Below, you can see any apps you may have installed and your paired Apple Watch if you have one.


⚠ Vardhan's iPhone is busy: Making Vardhan's iPhone ready for development
Xcode will continue when Vardhan's iPhone is finished.

Vardhan's iPhone

iOS 14.0.1 (18A393)
Model: iPhone X (Model A1865, A1901, A1902, A1903)
Capacity: [redacted]
Serial Number: [redacted]
Identifier: [redacted]

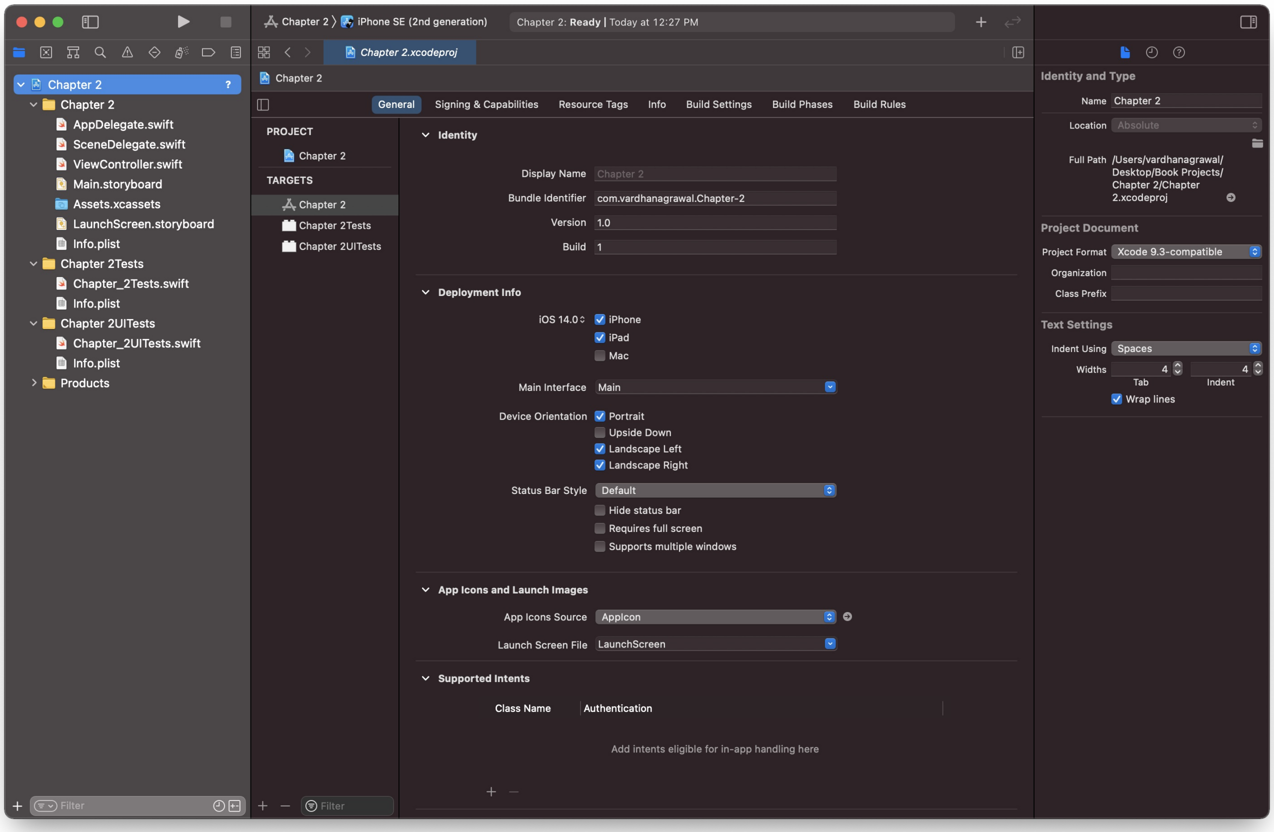
[✓] Show as run destination
[ ] Connect via network

Figure 2-5: Devices and Simulators Window

To complete setting up wireless debugging, check the **Connect via Network** box. While your phone is being setup, you'll see a dialog which says your phone is being setup, but after it's done, you'll be able to disconnect your device and it should work with Xcode, cable-free!

Select Simulator

Now, open up the Xcode project you created, in case it's not already. If you'll recall, it should look something like this:



Chapter 2 〉 iPhone SE (2nd generation) — Chapter 2: **Ready** | Today at 12:27 PM

Chapter 2
- Chapter 2
  - AppDelegate.swift
  - SceneDelegate.swift
  - ViewController.swift
  - Main.storyboard
  - Assets.xcassets
  - LaunchScreen.storyboard
  - Info.plist
- Chapter 2Tests
  - Chapter_2Tests.swift
  - Info.plist
- Chapter 2UITests
  - Chapter_2UITests.swift
  - Info.plist
- Products

PROJECT: Chapter 2
TARGETS: Chapter 2, Chapter 2Tests, Chapter 2UITests

General | Signing & Capabilities | Resource Tags | Info | Build Settings | Build Phases | Build Rules

Identity

Display Name	Chapter 2
Bundle Identifier	com.vardhanagrawal.Chapter-2
Version	1.0
Build	1

Deployment Info

iOS 14.0	✓ iPhone ✓ iPad ☐ Mac
Main Interface	Main
Device Orientation	✓ Portrait ☐ Upside Down ✓ Landscape Left ✓ Landscape Right
Status Bar Style	Default
	☐ Hide status bar ☐ Requires full screen ☐ Supports multiple windows

App Icons and Launch Images

App Icons Source	AppIcon
Launch Screen File	LaunchScreen

Supported Intents

Class Name	Authentication

Add intents eligible for in-app handling here

Identity and Type

Name	Chapter 2
Location	Absolute
Full Path	/Users/vardhanagrawal/Desktop/Book Projects/Chapter 2/Chapter 2.xcodeproj

Project Document

Project Format	Xcode 9.3-compatible
Organization	
Class Prefix	

Text Settings

Indent Using	Spaces
Widths	4 Tab, 4 Indent
	✓ Wrap lines

Figure 2-7: Your Xcode Project

In the upper left corner of the window, you'll see a triangular button called the **Run** button. And next to it, you'll see a device name, such as iPhone 11, iPhone 8 Plus, or something more specific, like Vardhan's iPhone. In order to use your own iPhone for debugging, you'll need to select it from the dropdown menu.

A network icon appears next to your device to indicate that wireless debugging is enabled and that Xcode has a connection to your device. If you don't see this, you may need to restart Xcode, or remove your device and add it back to wireless debugging.

User Interface Design

Finally, we're ready to begin to work on our image classification app! If you've never built a user interface using Xcode's storyboards, you'll gain a working understanding of it here.

For those who'd like to skip ahead, we'll have an image view with two labels, and a view that holds them together by the end of this section. I'll be doing it step-by-step for those who'd like to follow along.

To get started, open the `Main.storyboard` file in the **Project Navigator**, which is on the left pane of your Xcode window.

Image View

The heart of our design will be a `UIImageView` to show the user what they're pointing at in real time. The device will get a video stream from the camera and display it to the user via this view.

Object Library

Head to the **Object Library** to search for an image view. You can access this by clicking on the seventh icon from the right in the uppermost toolbar of Xcode. The icon appears as a box inscribed in a circle.

Now, search for an image view, and you should see something like this:

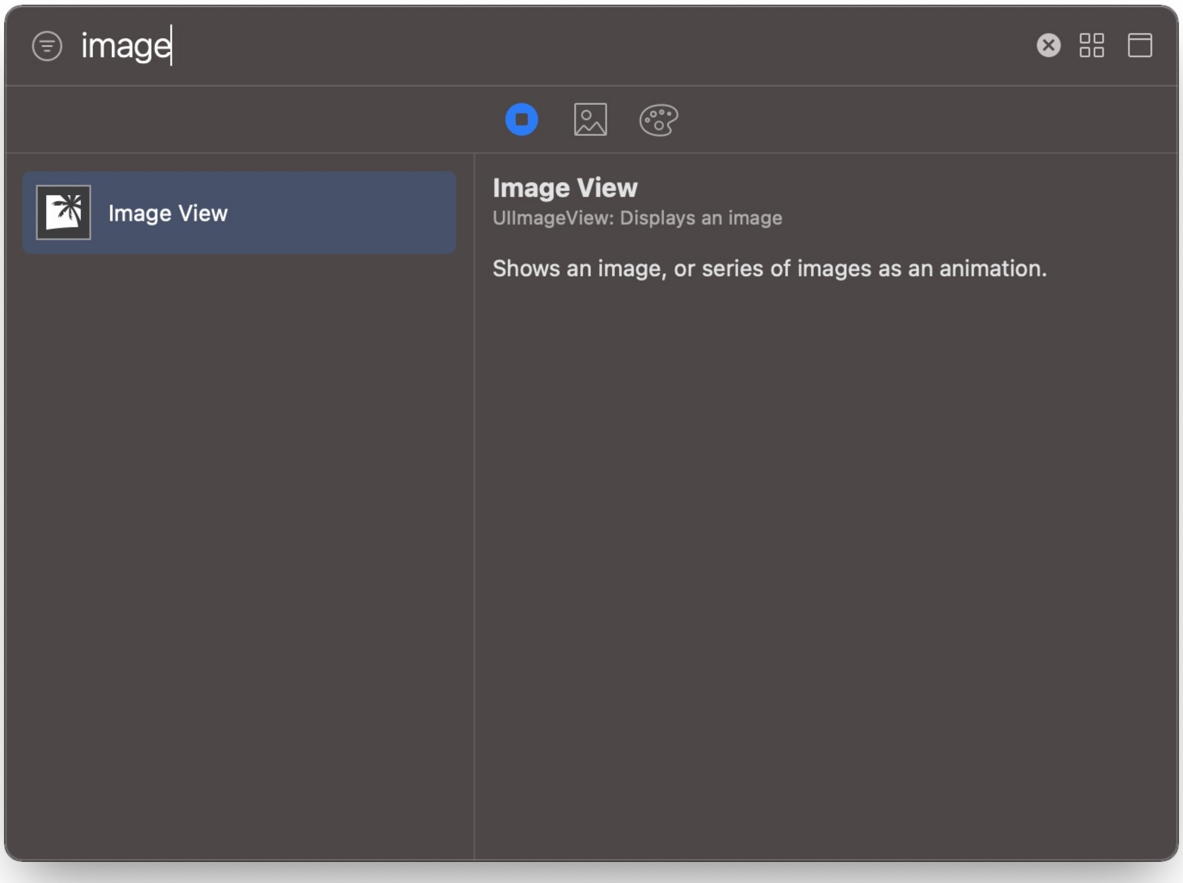


Image View
UIImageView: Displays an image

Shows an image, or series of images as an animation.

Figure 2-8: Object Library

You can drag the image view directly into your storyboard, and it should fill the screen. If it doesn't, use the white squares along the image view to resize it to match the size of your View Controller.

Placeholder

Optionally, you can add a placeholder image to get a sense of how your design will appear in your final app. I got an image of a banana from the internet, and I've used it to fill my image view. To add a placeholder, simply drag an image to your project navigator on the left.

Then, select the image from the dropdown on the right pane (attributes inspector) after selecting the image view on your storyboard. Finally, set the **Content Mode** to **Aspect Fill** to make your image appear appropriately.

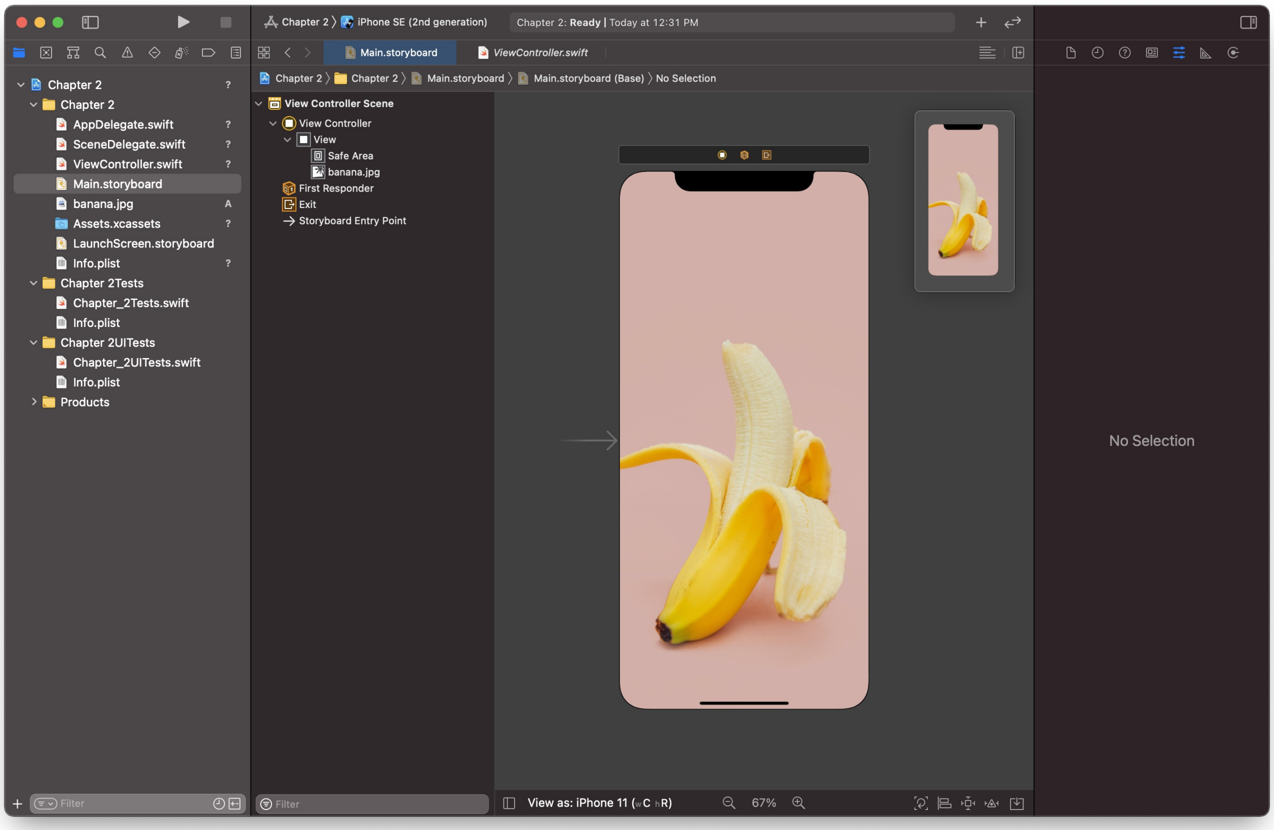


Figure 2-9: Adding a Placeholder

View and Labels

Since this is an image classification app, we need to have a way to tell the user what our app thinks their object is. For this, we'll have a plain white view at the top, which contains two labels: one for the prediction, and one for the confidence. We can build our basic user interface using the storyboard in our Xcode project.

Adding the View

As you did before, head to your Object Library and find a view (it's simply a "view" and not an "image view" or any other type of view). Search “view”, and it should be the second-to-last object in the list. Drag it into your View Controller and position it as you see fit.

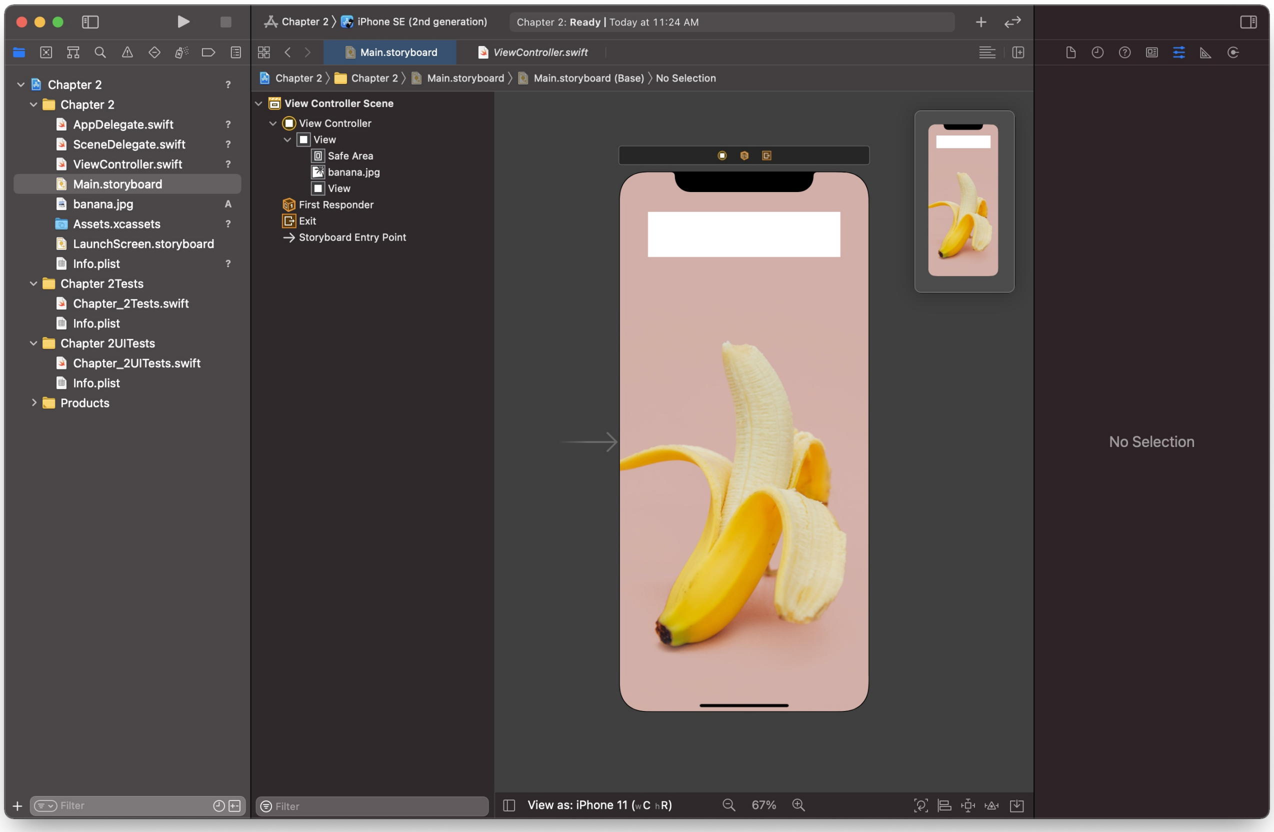


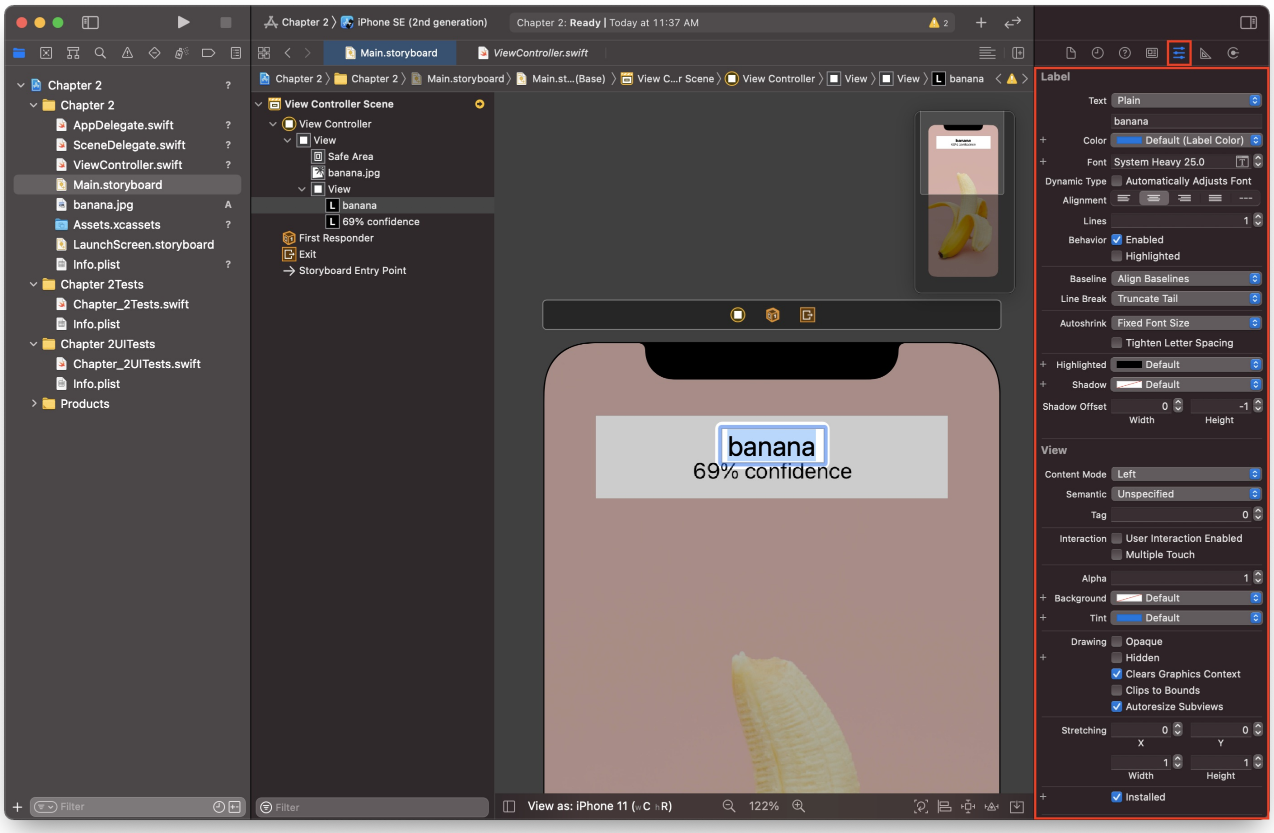
Figure 2-10: Adding a Label

I’ve put mine towards the top of my view, but of course, you can put yours wherever you like. For aesthetics, you can even make yours translucent or use a blur view to add a cool effect for transparency—be creative with it!

Adding the Labels

Now, for the most important part: the labels. We'll be using one label for the prediction and one for the confidence. Similar to the previous two steps, drag in two labels. You can find them by searching for "label" in the object library.

Make sure you drag them *into* the view and not *on top*. You'll see why this is important when we cover auto layout a little later.



Label

Text	Plain
	banana
Color	Default (Label Color)
Font	System Heavy 25.0
Dynamic Type	☐ Automatically Adjusts Font
Lines	1
Behavior	✓ Enabled
	☐ Highlighted
Baseline	Align Baselines
Line Break	Truncate Tail
Autoshrink	Fixed Font Size
	☐ Tighten Letter Spacing
Highlighted	Default
Shadow	Default
Shadow Offset	Width 0, Height -1

View

Content Mode	Left
Semantic	Unspecified
Tag	0
Interaction	☐ User Interaction Enabled
	☐ Multiple Touch
Alpha	1
Background	Default
Tint	Default
Drawing	☐ Opaque
	☐ Hidden
	✓ Clears Graphics Context
	☐ Clips to Bounds
	✓ Autoresize Subviews
Stretching	X 0, Y 0
	Width 1, Height 1
	✓ Installed

Figure 2-11: Attributes Inspector

If you look closely, you might notice that I've styled my labels. If you'd like to do this, make sure that the right pane of Xcode is visible, and then, use the attributes inspector tab to change the size, style, and font of your labels.

Auto Layout

Apple makes a lot of devices, and to support them all, Xcode has a useful tool called “Auto Layout” built into it. Though the name may lead you to believe that it “automatically” sets constraints for you and resizes on multiple devices, you need to tell Xcode where to place your objects in the view.

After you’re done roughly placing your user interface elements via the storyboard, you’re ready to set constraints and anchor your elements.

At the bottom of your storyboard, you’ll notice a menu bar which has the current zoom in the center as well as some icons on the right side. These icons will help you set auto layout constraints for your app.

Constraining the Image View

For starters, let’s constrain the image view to the size of the view. This way, we can make sure that no matter the size of the physical screen, the image view scales to fill the entire screen instead of leaving unwanted gaps or bezels between the actual image and the screen.

To start adding constraints, click on the second icon from the right, which looks like a square with lines surrounding it. A popup should appear, and you’ll see something like this:

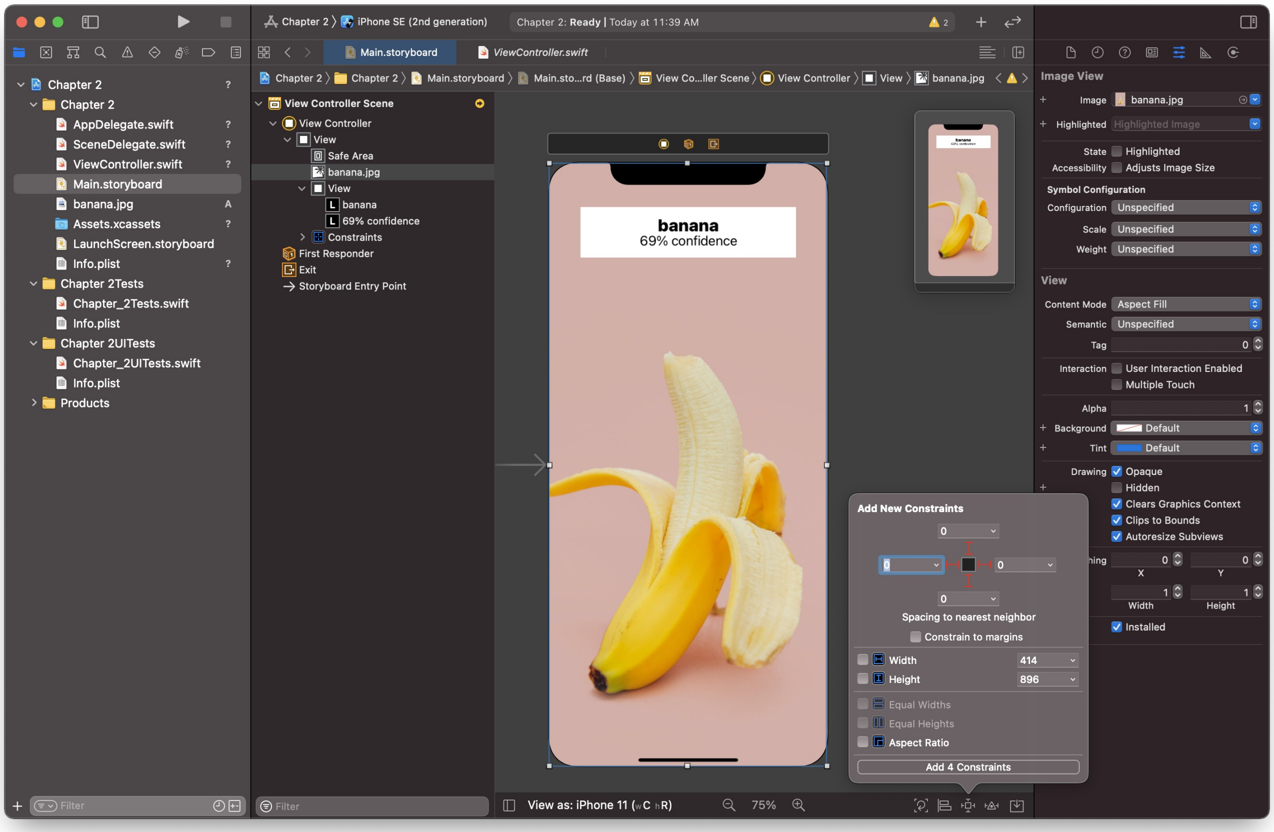


Figure 2-12: Setting Contraints

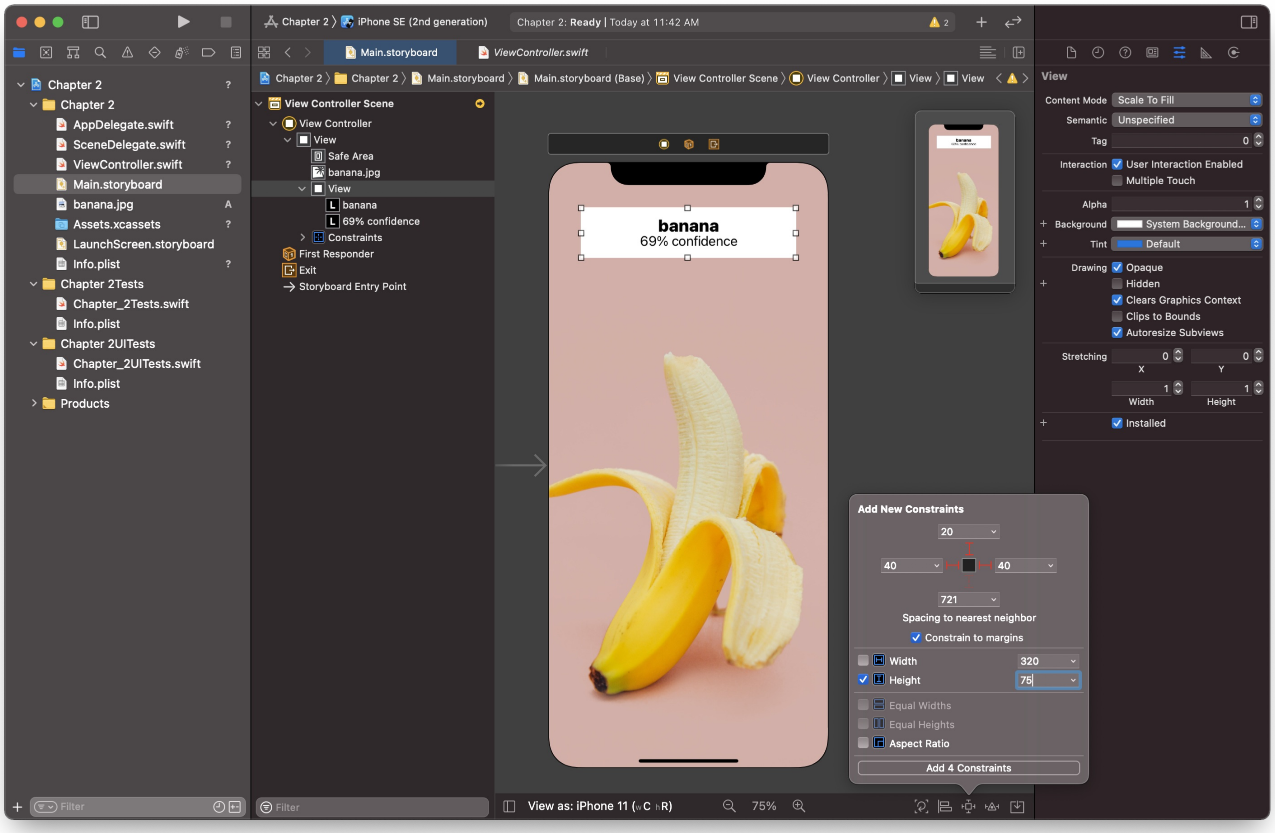
The four numbers surrounding the box tell Xcode where to place the object in terms of its surrounding objects. Since the image view doesn't have surrounding objects other than the screen's edge, these numbers represent the distance of the edges of the image view to the edge of the screen. If we want the image view to fill the screen, all of these numbers must be zero.

Press the **return** key four times to add all of the constraints and hit the **Add Constraints** button at the bottom of the dialogue. If you run your app on different simulators now, you'll notice that other elements, like the view and labels, will be misplaced, but the image view will always fill the screen. This is because Xcode now knows exactly where to place your `UIImageView` based on your layout constraints.

Constraining the Label Container

Remember the white view we created earlier? Our next step is to constrain it to the screen. Similar to how you constrained the image view, open the **Add Constraints** menu — it's the second icon from the right.

This time, however, instead of having zeroes for our constraints, we'll need to specify the distance of the view from the edges of the screen. In my example, I want it to be *20 pixels* from the top and I want a distance of *40 pixels* from either side. Since we aren't specifying a bottom constraint, we need the height to be fixed to *50 pixels*.



Content Mode: Scale To Fill
Semantic: Unspecified
Tag: 0
Interaction: ✓ User Interaction Enabled; Multiple Touch
Alpha: 1
Background: System Background...
Tint: Default
Drawing: ✓ Opaque; Hidden; ✓ Clears Graphics Context; Clips to Bounds; ✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1
✓ Installed

Add New Constraints
Top: 20
Left: 40
Right: 40
Bottom: 721
Spacing to nearest neighbor
✓ Constrain to margins
Width: 320
✓ Height: 75
Equal Widths
Equal Heights
Aspect Ratio
Add 4 Constraints

Figure 2-13: Selecting Constraints

Change the numbers in the dialogue to be *20* on the top, *40* on the left, and *40* on the right. The dotted lines should become red, indicating that a constraint will be set. In addition, check the box next to **height** and set a value of *75* to indicate that the view should always be *75 pixels* tall.

Labels and Stack Views

Hold down the **Shift** key and select both labels, we'll create what's called a *stack view*. A stack view is one of the marvels of auto-layout. In fact, it's one of my top go-to tools for quick user interface layouts.

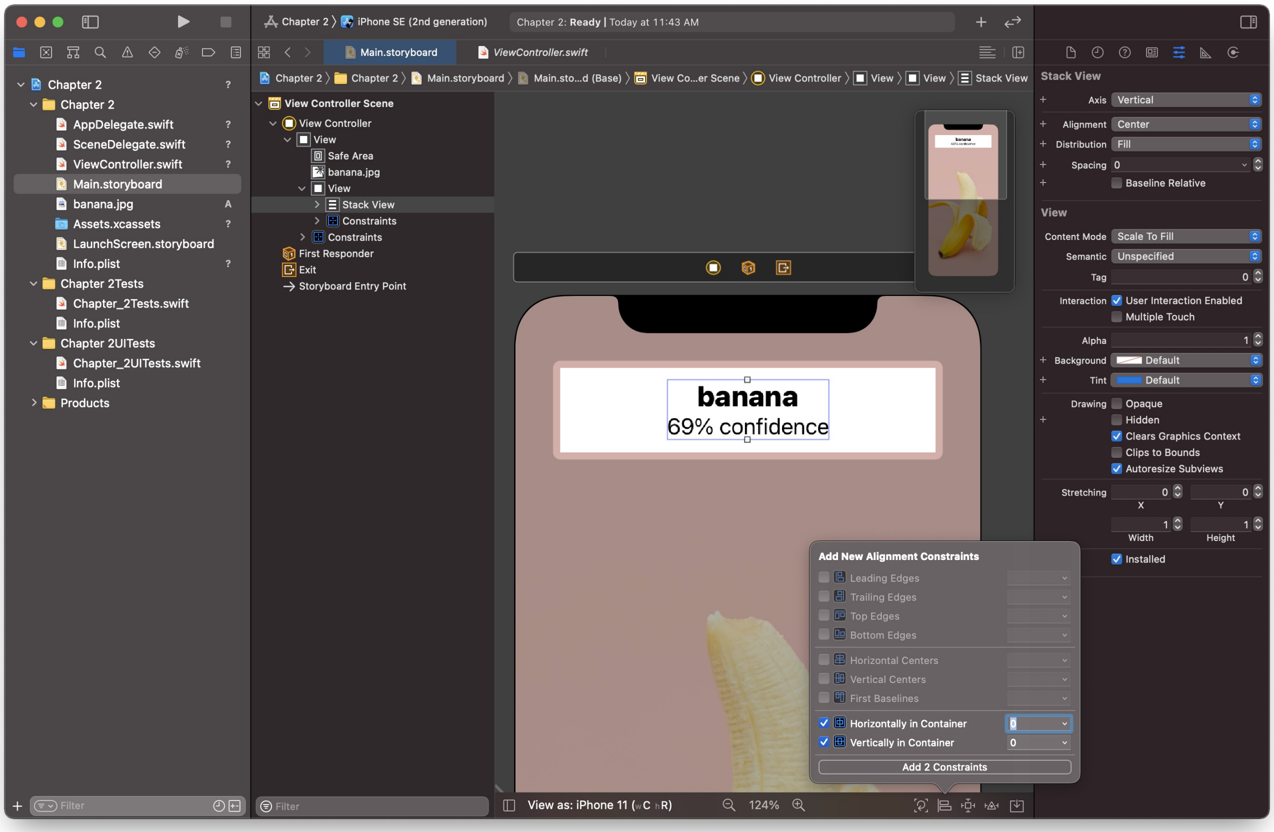
They allow you to create sets of objects which can be anchored as groups, while taking care of spacing, widths, and heights within the stack view automatically — here's where the "auto" part comes in…

Once you've selected the two labels, head to **Editor > Embed In > Stack View**, which should make your two labels appear as if they've been grouped together. This "group" can now be used as an individual unit to set constraints to the two labels simultaneously.

Constraining the Stack View

The last step of ensuring that auto-layout is set correctly, is to constrain the stack view we just created to the inside of the view. If you remember, we ensured that the labels were inside of the view and *not* on top. This allows us to set constraints to the view as opposed to the screen, since the view is already constrained to the screen's bounds.

Select the stack view by clicking on one of the labels, and then click on the edge of the highlighted region. Ensure that you've selected the stack view, and not one of the labels.



Add New Alignment Constraints

- [ ] Leading Edges
- [ ] Trailing Edges
- [ ] Top Edges
- [ ] Bottom Edges
- [ ] Horizontal Centers
- [ ] Vertical Centers
- [ ] First Baselines
- [✓] Horizontally in Container 0
- [✓] Vertically in Container 0

Add 2 Constraints

Stack View

Axis	Vertical
Alignment	Center
Distribution	Fill
Spacing	0

View

Content Mode	Scale To Fill
Semantic	Unspecified
Tag	0
Interaction	✓ User Interaction Enabled
Alpha	1
Background	Default
Tint	Default
Drawing	✓ Clears Graphics Context, ✓ Autoresize Subviews
Stretching	X 0, Y 0, Width 1, Height 1
	✓ Installed

Figure 2-14: Centering the Stack View

We'll use this opportunity to learn one more tool in auto layout — the **Align** tool. This allows you to center your views both vertically and horizontally in a container or align your views. With your stack view selected, click on the third icon from the right, which looks like a sideways bar graph. Then, check the boxes which say **Horizontally in Container** and **Vertically in Container**. These boxes tell Xcode to center your stack view inside the white container view we created earlier.

2-2 Pre-Processing Video Feed

In the previous section, you set up your user interface using the Interface Builder and learned to use some advanced tools, such as Stack Views, Auto-Layout Constraints, and more. These skills will come in handy when you build future apps in this book as well as other iOS apps in the future. If the previous section was a review for you, not to worry. Now, I'll likely cover topics which are new to you, and the rest of the book will only become more advanced from here on out.

In this section, you'll learn how to use the built in camera of your device to input a video feed and pre-process it for machine learning purposes. The frames that you extract from the video feed in this section will be used later to predict the objects present in each scene.

Connecting to Code

You now should have a user interface created for our image recognition app. The last thing left to do before we actually begin coding our app, is to connect the user interface elements from the Interface Builder to our corresponding Swift file, **ViewController.swift**.

Interface Builder Outlets

Open **ViewController.swift** via the left pane of your Xcode window. Inside the `ViewController` class, add the following three lines of code above the `viewDidLoad()` method:

```
@IBOutlet weak var predictionLabel: UILabel!
@IBOutlet weak var confidenceLabel: UILabel!
@IBOutlet weak var imageView: UIImageView!
```

The `@IBOutlet` tag tells Xcode that you're declaring a connection from code to the Interface Builder. If you didn't use the tag, Xcode would assume that you're building your interface purely in code.

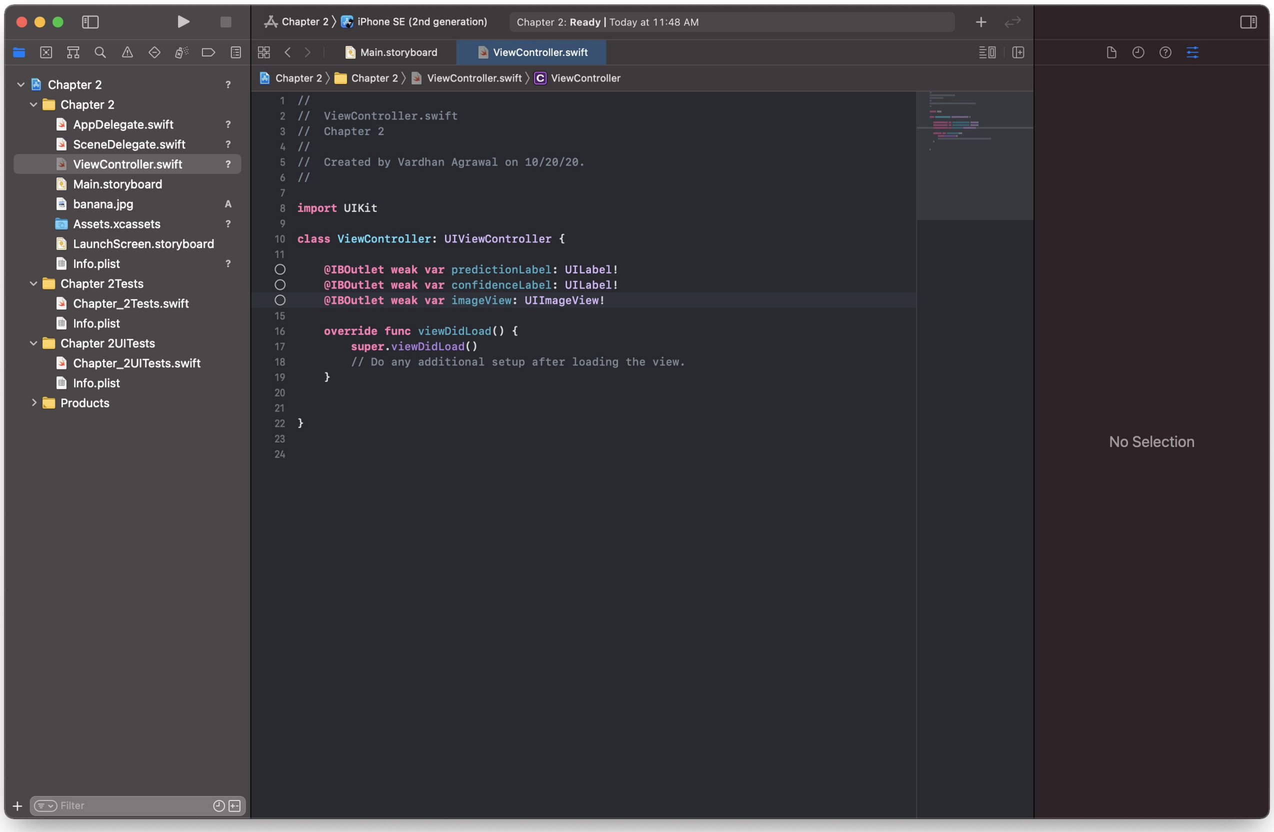
The word `weak` indicates that the Interface Builder has a weak reference to the label or the image view, in this case. This tells Xcode that you want iOS to allocate and deallocate the memory required for the element automatically instead of leaving it in memory at all times. Though you won't be using `weak` most of the time, you'll likely be better off sticking with it. It's okay if you don't fully understand this concept — it's an advanced computer science concept, which Swift handles for you automatically!

Last, we're declaring a regular Swift variable, and giving it the type of the element we're trying to connect to. The exclamation point on the end declares the type as an *Implicitly Unwrapped Optional*, which tells Swift that you know, 100%, that the object will never be `nil`.

Using the Assistant Editor

Now that the code portion is set, we'll use the **Assistant Editor** to help us connect each `@IBOutlet` to their corresponding element in the storyboard.

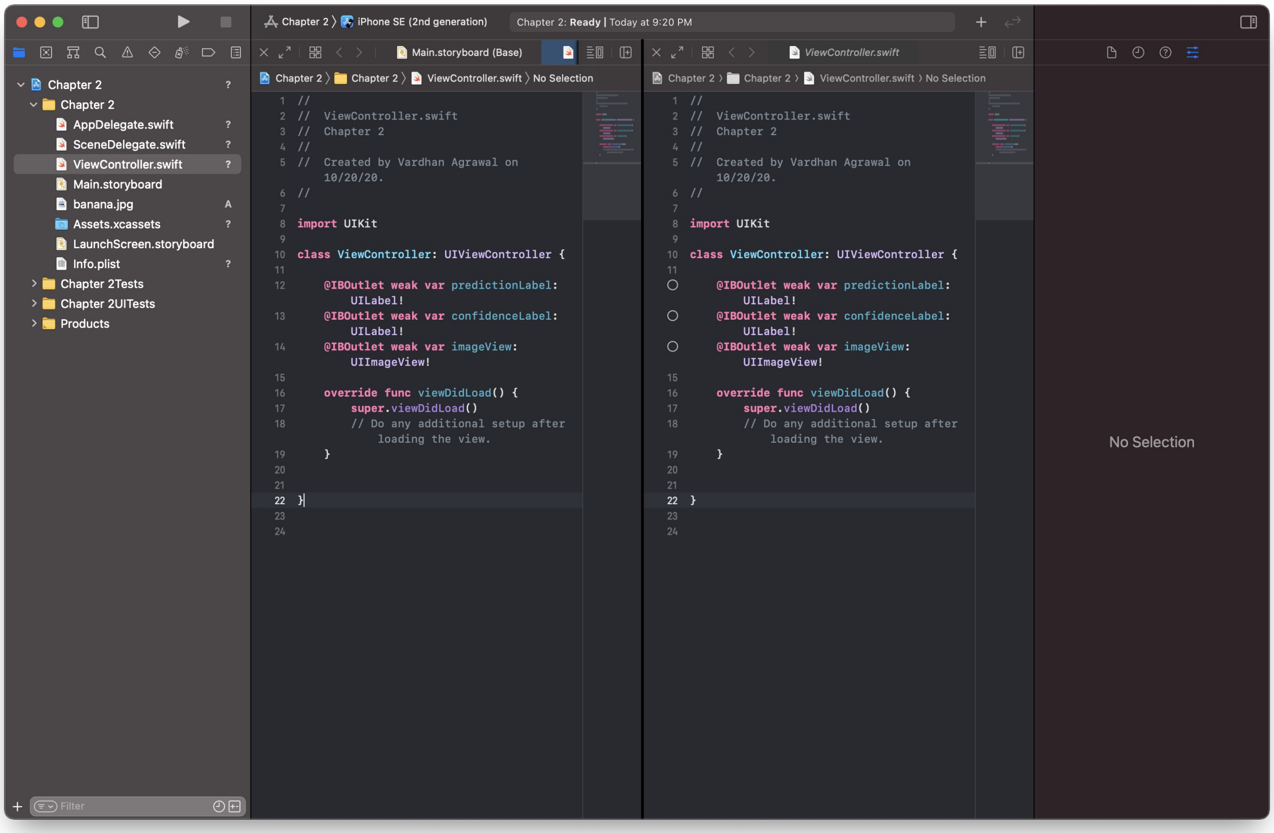
Open your **ViewController.swift** file using the left pane of your Xcode window.



```
//
//  ViewController.swift
//  Chapter 2
//
//  Created by Vardhan Agrawal on 10/20/20.
//

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var predictionLabel: UILabel!
    @IBOutlet weak var confidenceLabel: UILabel!
    @IBOutlet weak var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }


}
```

Figure 2-15: Xcode Editor

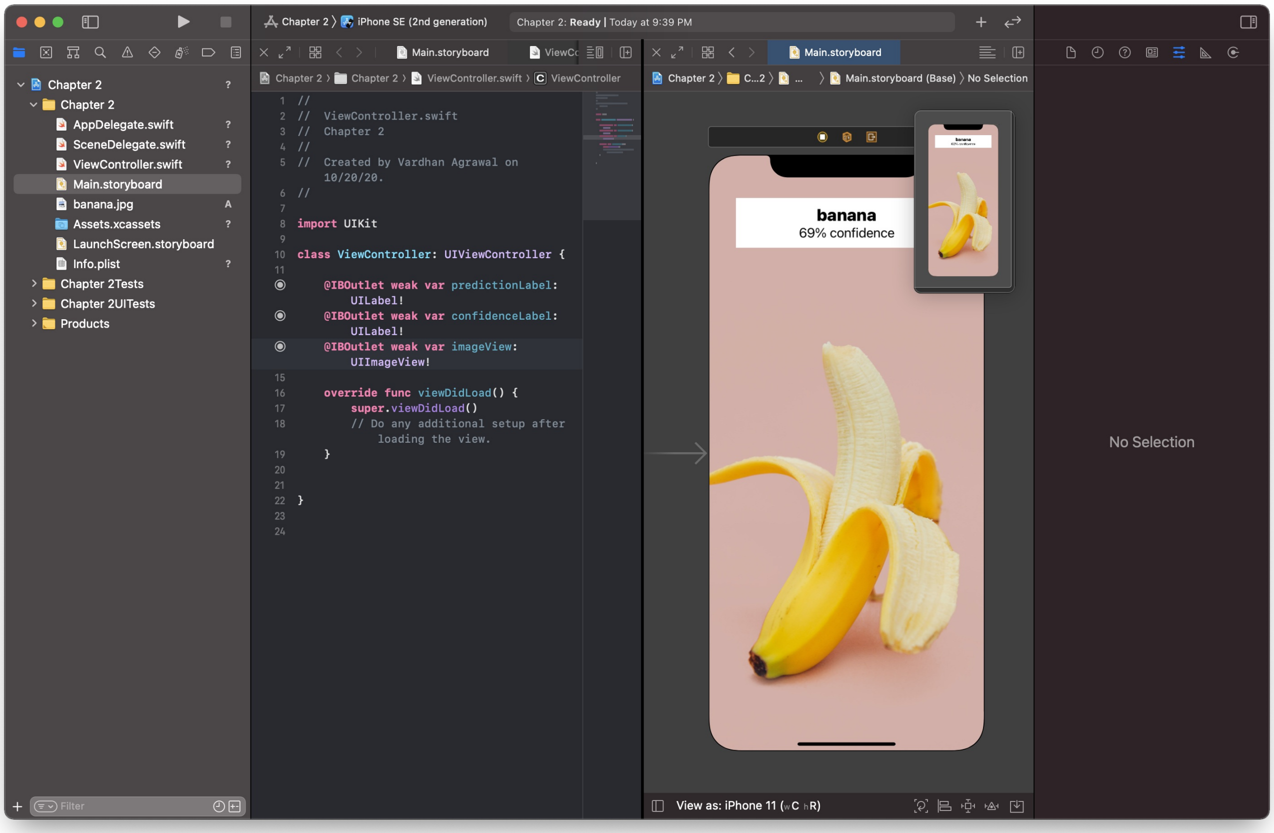
Then, in the top right of your file preview, select the icon which looks like a plus button next to a window. It's *not* This opens the Assistant Editor, which should automatically open your **ViewController.swift** file on the right of the storyboard. If it doesn't open your storyboard, simply select it using the file hierarchy at the top of the new pane.



```
//
//  ViewController.swift
//  Chapter 2
//
//  Created by Vardhan Agrawal on 10/20/20.
//

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var predictionLabel: UILabel!
    @IBOutlet weak var confidenceLabel: UILabel!
    @IBOutlet weak var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }


}
```

Figure 2-16: Assistant Editor

Instead of line numbers, you should see un-filled white circles to the left of each `@IBOutlet`, meaning that they aren't connected to the storyboard. Drag your cursor from the inside of these circles to the corresponding element in your storyboard. The circles will become filled when they're connected successfully.



```swift
//
//  ViewController.swift
//  Chapter 2
//
//  Created by Vardhan Agrawal on
    10/20/20.
//

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var predictionLabel:
        UILabel!
    @IBOutlet weak var confidenceLabel:
        UILabel!
    @IBOutlet weak var imageView:
        UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after
            loading the view.
    }


}
```

Figure 2-17: Assistant Editor with Storyboard

Go ahead and click the Assistant Editor button again to return to your original view. You no longer need your storyboard to be opened alongside your Swift file. By now, you should have three variables in your **ViewController.swift** file. I've named mine as follows:

- `predictionLabel` for the upper label.
- `confidenceLabel` for the lower label.
- `imageView` for the image view.

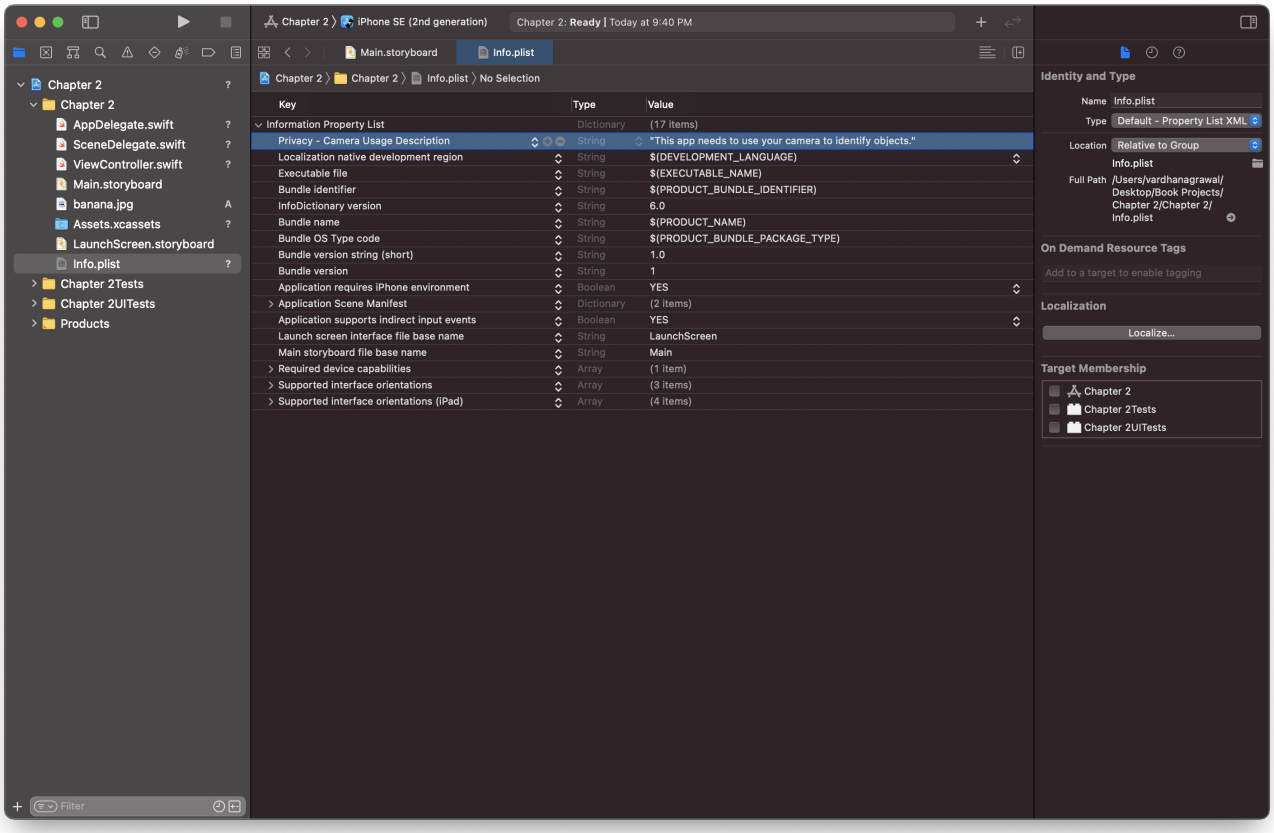
Of course, you don't need to use the same names that I did, but I'll be using them throughout the course of this chapter. If you chose different names, make sure you changed them in all of the places they're referenced.

Camera Usage Description

Since we'll be using the camera for this app, we need to ask the user for permission before doing so — otherwise, our app won't work. Head to your **Info.plist** file first. Then, click the + button next to the **Information Property List** and paste the following:

```
NSCameraUsageDescription
```

This will auto-correct to `Privacy - Camera Usage Description` if done correctly. In the value section for this key, type a string which describes why you need to use the camera.



Key	Type	Value
Information Property List	Dictionary	(17 items)
Privacy - Camera Usage Description	String	"This app needs to use your camera to identify objects."
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(2 items)
Application supports indirect input events	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)

Figure 2-18: Information Property List

Capture Session

Since our app involves recognizing objects through a live video feed instead of a still image, we have some additional work to do in creating a video feed. While this may seem like a long process at first, I'll be explaining each thing step-by-step. It's an important skill to have, even for other camera-related apps.

Conforming to the Delegate

To get updates on what the device's camera is doing, including getting the camera's video feed, we need our `ViewController` class to conform to the `AVCaptureVideoDataOutputSampleBufferDelegate` delegate. Yes, that's a long name, but don't let that scare you!

Import List

To get access to the Audio-Video APIs that Apple provides their developers, we can import the `AVKit` library at the top of our file. Enter the following line of code directly under the import statement for `UIKit`:

```
import AVKit
```

While we're at it, let's import some of the other libraries we'll be needing later on. For the Vision API we'll be using to analyze video frames, let's import the `Vision` framework and for importing machine learning models, let's import `Core ML` like this:

```
import Vision
Import CoreML
```

Great! You're all set with the import statements.

Class Extension

To let our `ViewController` class conform to the `AVCaptureVideoDataOutputSampleBufferDelegate` protocol, we can create an extension of the class and use it to contain all of the methods from our delegate. If you wanted, you could do it without an extension, but industry norms are to use one.

To create an extension that conforms to this protocol, you need to enter the following after the closing brace of your `ViewController` class declaration:

```swift
// MARK: - AVCaptureSession
extension ViewController: AVCaptureVideoDataOutputSampleBufferDelegate {
    func setupSession() {
        // Your code goes here
    }
}
```

An extension in Swift is used to add functionality to a class — in this case, we're adding conformance to the `AVCaptureVideoDataOutputSampleBufferDelegate` protocol to our `ViewController` class.

Inside it, we'll create a method to set up our `AVCaptureSession`. Later on, we'll call this method to set up whatever we need to at the beginning of the app's life cycle.

Session I/O

The purpose of this app is not to get a live video feed, but instead, it is to process individual frames from the camera through a machine learning model. In addition, we'll be displaying the live feed through the `UIImageView` we created earlier.

Capture Session Configuration

Now, it's time to setup an `AVCaptureSession`, which allows you to capture camera input in real-time from your device's camera. Since our app is a *live* image classification app, we'll need to use one.

Declaring your Session

Go ahead and put the following in the `setupSession()` which you declared in your `ViewController` delegate extension.

```
guard let device = AVCaptureDevice.default(for: .video) else { return }
guard let input = try? AVCaptureDeviceInput(device: device) else { return }
```

In the code you just added, we're making sure we have a device capable of streaming video from its camera. Then, the `AVCaptureDeviceInput` (after making sure there aren't any errors thrown) is storing the input from the device to use later.

Next, add the following lines of code:

```
let session = AVCaptureSession()
session.sessionPreset = .hd4K3840x2160
```

You've now created an `AVCaptureSession` object and set its bitrate and resolution to Ultra High-Definition (UHD) at 4K with 3840 pixels x 2160 pixels. Of course, you can change this later if you'd like to adjust aspects of your stream.

Preview Layer

After we're done setting up the device and its capture session, we're ready to create a preview layer, which allows the user to see where their camera is pointing to — after all, you want to see the object you're trying to classify! Later, we'll use `AVCaptureVideoDataOutput` to help us capture the individual frames in the video feed for processing via our machine learning model.

In your `setupSession()` method, add the following:

```
let previewLayer = AVCaptureVideoPreviewLayer(session: session)
previewLayer.frame = view.frame
previewLayer.videoGravity = .resizeAspectFill
imageView.layer.addSublayer(previewLayer)
```

In the first line of code, we've created an instance of the `AVCaptureVideoPreviewLayer`, which takes in our `AVCaptureSession` as a parameter. Then, to display the video feed, we set the size of the preview layer to the size of the entire screen, as denoted by `view.frame`. We also need to set the `videoGravity` so that it fills the screen, while maintaining the aspect ratio of the feed. Last, we used the `layer` attribute of the image view to show the feed inside the image view you created in Chapter 2-1.

Preparing for Processing

Now that we have our video feed taken care of, we're ready to export it frame-by-frame to use it with a machine learning model later on. We can use certain features of the `AVKit` framework to help us reach our end goal.

Session Output

To output the frames of the video feed, we need to create an `AVCaptureVideoDataOutput`. You can do this by pasting the following two lines of code into the same `setupSession()` method.

```
let output = AVCaptureVideoDataOutput()
output.setSampleBufferDelegate(self, queue: DispatchQueue(label: "videoQueue"))
```

Similar to what you did previously, we just created an `AVCaptureVideoDataOutput()`. As you recall, our `ViewController` conforms to the `AVCaptureVideoDataOutputSampleBufferDelegate`, so that we can receive each frame to process it individually from the live video feed. The second line of code from the snippet above sets the `ViewController` to the delegate of the output stream. The `DispatchQueue` is there to ensure that each frame gets sent to the delegate in the correct order.

Starting the Session

Now, we're almost done on the I/O side of our app, but we still need to do one more thing — to add the input and the output, and then start the `AVCaptureSession`.

Add the following at the end of the `setupSession()` method:

```
session.addOutput(output)
session.addInput(input)
session.startRunning()
```

We've used the `addOutput()` and `addInput()` methods of the `AVCaptureSession` we created earlier to gain access to the device's camera, display our live feed to our image view, and extract individual frames in the video for machine learning processing.

Now, we need to finally call the method `setupSession()` inside the `viewDidLoad()` method overridden from the `UIViewController` class. After doing this, your `viewDidLoad()` method should look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupSession()
}
```

2-3 Image Classification and Labeling

In the previous section, you learned how to connect your user interface from the interface builder to actual Swift code. You also learned how to capture a live video feed from your device's camera and then extract individual frames to process. By this point, you should now have an app with a live video feed.

In this section, we'll find an image classification model to use in our app. Next, you'll learn how to use basic image processing techniques to classify frames of our live video feed from the previous section. By the end of the chapter, you'll have a fully-functional image classification app which can, in real time, process and identify objects.

Image Classification

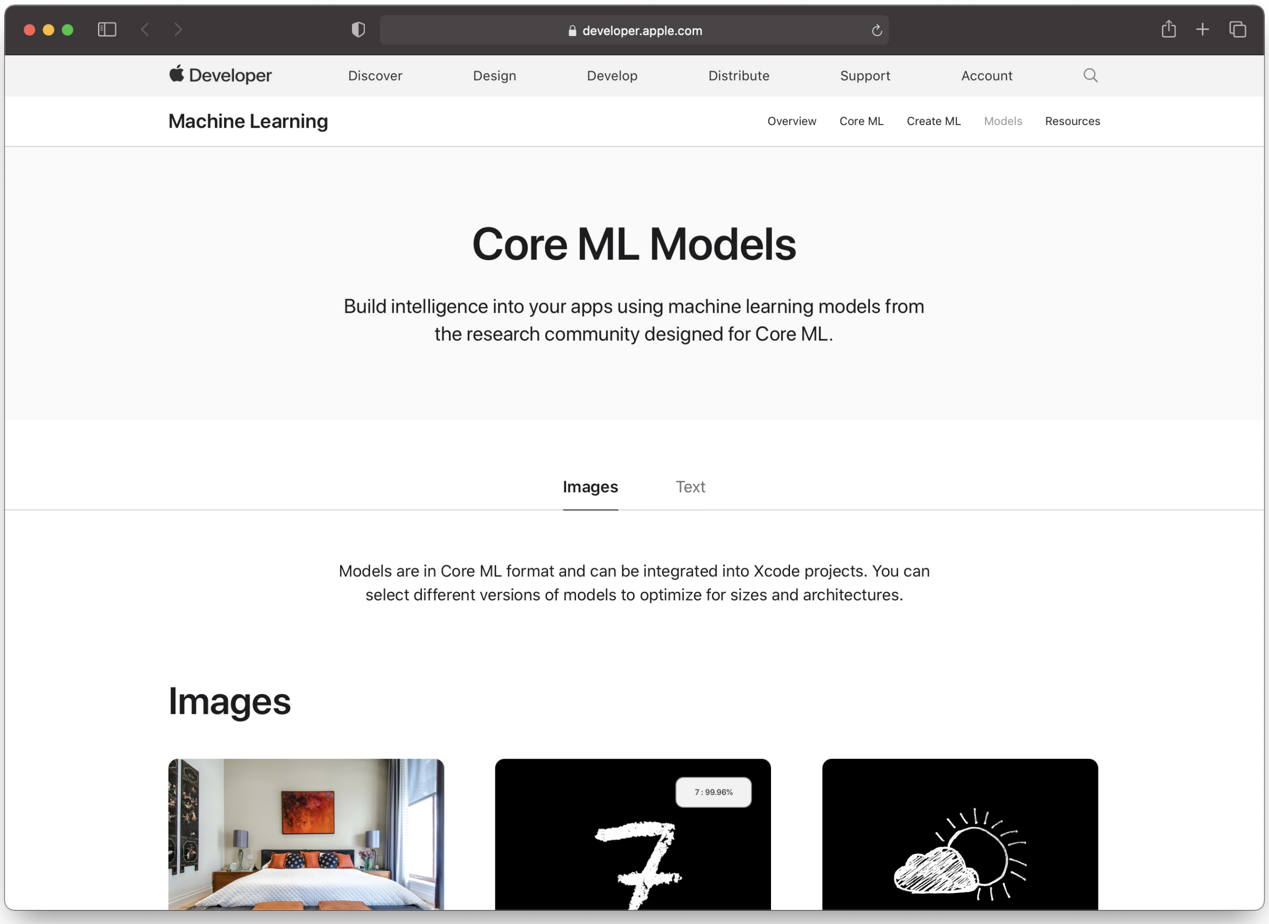
Before we begin processing and classifying our images, let's first find a machine learning model to use. For this chapter, we won't be using a custom model, but in later chapters, you'll learn to build your own models using robust, state-of-the art tools and techniques.

Finding a Model

Since we're using Core ML for this app, Apple's website is a great place to find a machine learning model. We'll find a suitable model on this website for our image classification app to work as desired.

Browsing and Downloading

On the Working with Core ML Models page, you'll find a variety of models to choose from towards the bottom of the page.



Core ML Models

Build intelligence into your apps using machine learning models from the research community designed for Core ML.

Models are in Core ML format and can be integrated into Xcode projects. You can select different versions of models to optimize for sizes and architectures.

Images

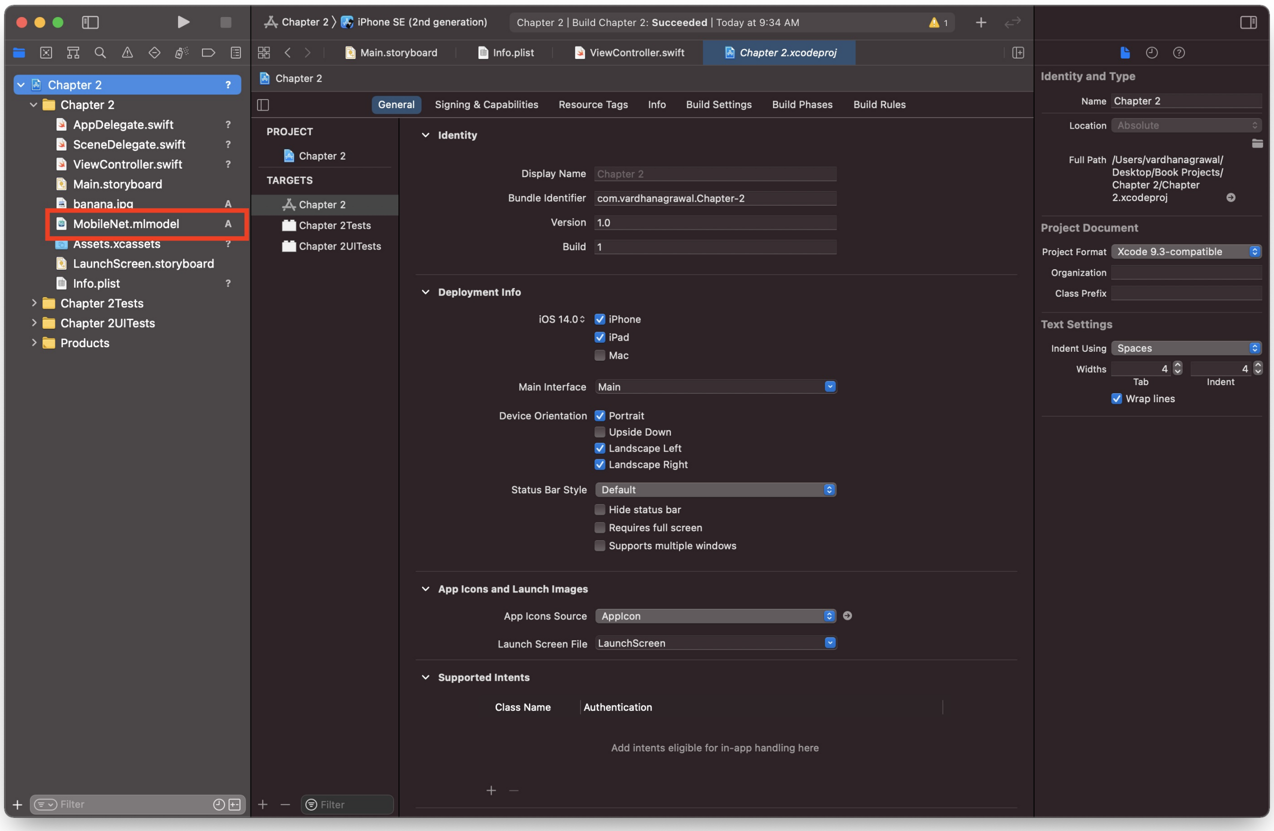
Figure 2-18: Browsing for Core ML Models

For convenience, I've selected the first model listed, which was originally developed by Andrew Howard as a Caffe model, and converted into Core ML later — something which we'll be doing later in the book.

You can download this model by either clicking the **Download Core ML Model** button underneath the model you want, or download it directly.

Importing

Importing the model you've created is easy. All you need to do is to drag the **.mlmodel** file you've just downloaded into your Xcode project. Make sure you do this in the folder which shares a name as your project. This is where all of your other important items located, such as your **ViewController.swift** and **Main.storyboard** files.



Chapter 2 | Build Chapter 2: **Succeeded** | Today at 9:34 AM

Identity
Display Name: Chapter 2
Bundle Identifier: com.vardhanagrawal.Chapter-2
Version: 1.0
Build: 1

Deployment Info
iOS 14.0: ✓ iPhone, ✓ iPad, Mac
Main Interface: Main
Device Orientation: ✓ Portrait, Upside Down, ✓ Landscape Left, ✓ Landscape Right
Status Bar Style: Default
Hide status bar
Requires full screen
Supports multiple windows

App Icons and Launch Images
App Icons Source: AppIcon
Launch Screen File: LaunchScreen

Supported Intents
Class Name | Authentication
Add intents eligible for in-app handling here

Identity and Type
Name: Chapter 2
Location: Absolute
Full Path: /Users/vardhanagrawal/Desktop/Book Projects/Chapter 2/Chapter 2.xcodeproj

Project Document
Project Format: Xcode 9.3-compatible

Text Settings
Indent Using: Spaces
Widths: Tab 4, Indent 4
✓ Wrap lines

Figure 2-19: Imported MobileNet Model

All Core ML models have an accompanying Swift wrapper class, which we can use to access important attributes of the machine learning model as needed throughout the development of the app. While yours may have already been generated, you might need to **Build** your project by pressing **Command + B** or selecting **Product > Build** from the menu bar.

Pixel Buffers

A pixel buffer is a way to store a small amount of image data shortly before it needs to be used. In this app, we'll use a pixel buffer to store individual frames of our video feed to later feed them into the image classification model. Once we're done with the image, we can easily and efficiently dispose of it.

Capturing Output

In the previous section, you created an extension of the `ViewController` class, which conforms to the `AVCaptureVideoDataOutputSampleBufferDelegate`. If you recall, we did this in order to build a live video stream and extract individual frames for image processing. To access these frames, add the following delegate method to your `ViewController` extension:

```
func captureOutput(_ output: AVCaptureOutput, didOutput sampleBuffer: CMSampleBuff
er, from connection: AVCaptureConnection) {
    // your code here
}
```

The `sampleBuffer` parameter of this method is where the delegate passes in an image buffer. It's our job to turn this parameter into a pixel buffer that's compatible with our Core ML model.

Pixel Buffer Conversion

Now, we need to turn the `CMSampleBuffer` from the method call to a `CVPixelBuffer`. To do this, add the following into the delegate method you created earlier:

```
guard let pixelBuffer: CVPixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)
else { return }
```

If you've done Swift development in the past, the `guard-let` statement will look familiar to you, but essentially, it checks whether the pixel buffer can be created using the passed in parameter. If it cannot, the program gracefully returns as opposed to attempting to unwrap a `nil` optional.

Using the Model

Instantiation

Now, it’s time to create a `VNCoreMLModel` version of the file you imported earlier. With a similar concept as the pixel buffer, type the following to create a constant to store the model:

```
guard let model = try? VNCoreMLModel(for: MobileNet(configuration: MLModelConfigur
ation()).model) else { return }
```

This will use the model we imported and its Swift wrapper class to create a `VNCoreMLModel`. We would use the `VLCoreMLModel` for vision and image analysis, since the wrapper class doesn't give us any useful information.

To instantiate a `VNCoreMLModel`, we create an instance of the model's wrapper class by passing a blank configuration (which you could use to adjust the model, if you'd like), and then accessing its `model` property.

Core ML Request

To finally use this model, we’ll need to create a `VNCoreMLModelRequest`, which allows us to pass in the model as a parameter and receive the result — as well as any errors — in a completion handler. Type the following under the `guard-let` statements:

```
let request = VNCoreMLRequest(model: model) { (data, error) in
    // your code here
}
```

The `data` and `error` references can be used inside the declaration of `request` to access the results of the model request. Remember, though, that we haven’t made this request yet; we’re only declaring what to do when we eventually make this request.

Displaying the Results

Excellent job so far! We’re almost finished with our image classification app. Now all we need to do is to display our results to the user interface we created at the beginning of the chapter.

Unwrapping Optionals

As always, there's a catch; we need to first unwrap our results to prevent the program from crashing in the case of a nil-value. Let's use `guard-let` statements like we did before to ensure that we're safely assigning these results to their own constants.

Getting Results

When we receive the result from the model, it's actually not a single result — it's an array which carries a list of results. Since the machine learning model is only a computer, it might be unsure of what the actual object is. For example, it may think your television is a computer monitor, but also think that it's a television, a rectangle, or a picture frame. While the model is most confident that it's a "computer monitor," it also lists these other guesses in case we need to do something else with them.

To extract this list from the parameter, type the following inside the `request` closure:

```
guard let results = data.results as? [VNClassificationObservation] else { return }
```

This assigns the list as an array of `VNClassificationObservation` to a constant called `results`. If this isn't possible, the function returns safely.

First Result

You've learned that the results are a list of possible guesses of the model, so you now know that you need only one of the results; but, the question is which one to choose from an array of several. The results array we just created sorts the observations in descending level of confidence. To access the one we want, `results.first` can be used.

Add the following `guard-let` to your closure:

```
guard let firstObject = results.first else { return }
```

This `firstObject` constant can be used later, with the assumption that we won't have a `nil` value because of our safe optional unwrapping.

User Interface Update

Last, we need to update the user interface created at the beginning of the chapter. However, we'll need to handle it differently since we're doing it from inside an asynchronous type, a closure.

Checking Confidence

We don't want our label to keep changing unless the model is sure of the object it's seeing, to some degree. We can check the confidence and wait it to display the result with a simple `if-statement` :

```
if firstObject.confidence * 100 >= 20 {
    // display observation and confidence
} else {
    // display placeholders
}
```

The `confidence` property of the `VNClassificationObservation` is a value with a floating point decimal, so we'll need to convert it to a percentage by multiplying it by 100. Then, we can check if the confidence is 20% or higher before displaying the observation to the user.

Main Thread

iOS requires you to update the user interface on the main thread since the graphics rendering is done synchronously. In a completion handler, which is what we're currently in, the system sends the process to a background thread. The background thread waits for the model to be done processing the image and gives us the result. While this is all done in a fraction of a second, the other code has already been executed.

To exit the asynchronous program and update the UI, add the following to each condition of your `if-else block` :

```
DispatchQueue.main.async {
    // UI update here
}
```

Now, you can reliably update your user interface after receiving the observations from the model.

Updating the Labels

Finally, let’s update the labels with the information we need. In the first condition of the `if-statement`, add the following lines of code:

```
self.predictionLabel.text = firstObject.identifier.capitalized
self.confidenceLabel.text = String(firstObject.confidence * 100) + "%"
```

Here, we’ve capitalized the first observation’s `identifier` string for a nicer look when displayed. Then, we converted the confidence into a percentage for a more user-friendly way to visualize the confidence. Now, we also need to handle the case where the classifier is unsure of what the object is, which’ll look like this:

```
self.predictionLabel.text = "--"
self.confidenceLabel.text = "--"
```

In this case, we’ll just display two dashes to indicate to the user that our classifier is unable to identify the object the phone is pointing towards, encouraging them to move the phone around — instead of displaying incorrect results on the screen.

Running the Request

Now, after all of this setup, you're ready to actually run the request. This way, all of the code you wrote inside of the closure — including getting the most confident result from the model and displaying it to the user in a good-looking way.

Request Handler

Using the `VNImageRequestHandler`, we'll run the request you created earlier. After the closing brace of the `VNCoreMLRequest` closure (same indent as `let request` line), enter the following:

```
try? VNImageRequestHandler(cvPixelBuffer: pixelBuffer, options: [:]).perform([requ
est])
```

Here, you're using the `VNImageRequestHandler` and passing in the pixel buffer (a single image frame while capturing live video) as well as the requests you created from earlier. All of the code you wrote runs each time the video feed gets a new frame!

Running Code

And, since you're all set, plug your iPhone in and test it out! If you move your phone around your room, you'll see it begin to recognize the objects around you. Awesome, right?

Conclusion

Good work! You’ve now created your own image classification app, which, when pointed at an object, it can identify the object and tell you how confident it is. This chapter was only an introduction to the power of machine learning, however. As you’ll learn throughout the course of the book, you can train your own models as well, which specialize in the tasks you want them to undertake.

As a recap, you learned several important skills in this chapter. These include enabling a live video feed, extracting individual frames into a pixel buffer, setting up a simple user interface with auto-layout constraints, and using the vision framework and a machine learning model to classify visible objects in the live video feed.

In the next chapter, you’ll learn how to improve this app further by integrating augmented reality, an unrelated, but nonetheless interesting feature. Augmented reality allows your labels to be "attached" to the object itself as opposed to being stuck in a view at the top of your screen. Onward!

Chapter 3
A Primer on Python and Jupyter Notebook

In the previous chapter, you built your own image classification app, and learned about using the `AVCaptureSession` as well as other `AVKit` to segment a video feed into frames for processing. You also got a glimpse at building an intuitive user interface. However, you used a pre-built model to classify the objects in your scene.

In this chapter, you'll learn Python, a general-purpose language, which could be considered similar to Swift in some aspects. If you've never used Python before, nor have heard of it, you're in luck; I'll go step-by-step and explain the syntax of Python, no matter your skill level. If you’re a Python pro, you may skip (not recommended) the first section and proceed with the remaining sections of this chapter.

Later in the chapter, you’ll learn about Jupyter Notebook, a boon for everyone working in the data science field. You’ll learn what they are, how to use them, and where they fit into your day-to-day workflow in the world of machine learning.

3-1 Jupyter Notebook and Anaconda

In this section, you’ll learn about Jupyter Notebook, what they’re used for, and how you can use them in your machine learning workflow. You’ll also learn to install the appropriate tools to use Jupyter Notebook, and you’ll be able to create a Jupyter Notebook by the end of this section. You don’t need to know anything about Python just yet; as I’ll be covering that in the next section.

Towards the end of the section, you’ll learn about virtual environments, so that you don’t need to run Jupyter Notebook and Python scripts directly on your Mac. Instead, you’ll use a virtual environment called Anaconda to isolate your scripts, but don’t worry — I’ll cover everything step-by-step, so no prior experience is needed!

What is a Jupyter Notebook?

If you haven’t been in the data science or machine learning industry, it's likely that you’ve never heard of a Jupyter Notebook — and are probably thinking it’s an old-school composition style notebook from Jupyter. Conversely, if you’ve done anything at all with machine learning, you’re likely to have heard about Jupyter Notebook. But, what are they, and what makes them special?

In essence, Jupyter Notebooks are places where you can congregate your code, data, graphs, and text to store for yourself, share with your colleagues, or even publish for the data science community. Formally speaking, they are web applications which usually run locally on your machine and can update live as your notebook changes. As a side note, I’ll be referring to machine learning and data science interchangeably in this chapter — since they're very closely related. While they're not the same field, the distinction isn't important for you to know right now.

Before we begin installing and setting one up, let's take a look at what you can do with a Jupyter Notebook first — since it doesn't seem that interesting to you so far.

Terminal Commands

As developers, we all know how annoying it can be to enter a series of shell commands multiple times over — especially when we're doing the same daily tasks like traversing our file system or installing Cocoapods in our apps.

This is one place where a Jupyter Notebook can be used with ease. You can simply write out your commands (as you'll learn how to do later), and precede by an exclamation point to tell the system that you're dealing with shell commands here.

Presentation Slide Deck

Now, our comfort zone is, obviously, behind a computer. However, it's important to realize that some of our best work happens when we communicate it with those around us, including projects at work, school, or with the open source community. But, sometimes, code can be clumsy to read when projected as plain text on a presentation.

Jupyter Notebook has a feature in which you can generate slides from your blocks of code in your notebooks, which takes care of syntax highlighting and styling for you. This allows you to keep your audience engaged as you present your information, without having to take out time to create a slide deck.

Visualizations and Tables

Sometimes, just seeing a dataset isn't enough, and it's easier to graph it or view it in an organized format, like a table. But, when you do graph your data, it takes too long to update your graphs when you add more points to your dataset, or if you change the dataset altogether.

You guessed it: Jupyter Notebook allows you to create customized charts, graphs, and tables, which update each time you run your code. This makes it nearly seamless to interact with your data in a visual way, while being able to make changes on the fly.

Now, there's a lot more you can actually do with Jupyter Notebooks, many of which you'll learn throughout the course of this chapter. But hopefully, you now have a better idea of what a Jupyter Notebook is and why it's so commonly used in the machine learning industry. As a side note, you're likely to see them referred to as IPython Notebooks, since Jupyter Notebook has evolved from them. This might help if you're checking out older tutorials or resources on the internet.

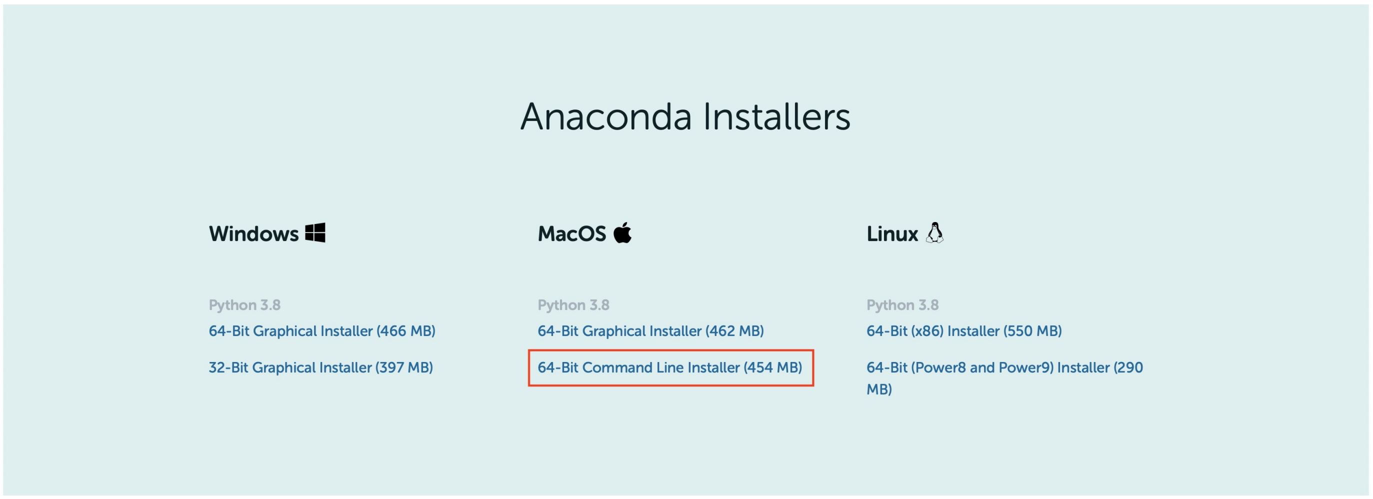
Installing Anaconda

Now, I'm sure you're exited to dive in and begin working with your own Jupyter Notebook. Let's make sure everything is in line first. While you can run these directly on your system, most professionals prefer to use a virtual environment, which allows you to isolate your projects and have unique modules — or dependencies — for each of your projects.

If you aren't aware and create projects directly, Python goes to the same place on your file system to retrieve dependencies. Therefore, if you were using two different versions of the same dependency for two different projects, it may cause problems. So, to turn you into a professional, let's use virtual environments!

As we mentioned before, you can install Jupyter Notebook directly using `pip`, a Python tool for managing dependencies. However, using a pre-packaged module such as Anaconda is the best way to install data science tools for beginners. In addition, you'll have many other tools needed for future projects as well. Let's start installing it.

First, download the **64-Bit Command Line Installer** for Anaconda on the Anaconda Distribution Downloads Page (https://www.anaconda.com/distribution/). You'll need to select your operating system and press the download button. We'll be using the **Python 3.8** version in this book, since that's currently the only version supported by Anaconda.


Anaconda Installers

Windows

Python 3.8
64-Bit Graphical Installer (466 MB)
32-Bit Graphical Installer (397 MB)

MacOS

Python 3.8
64-Bit Graphical Installer (462 MB)
64-Bit Command Line Installer (454 MB)

Linux

Python 3.8
64-Bit (x86) Installer (550 MB)
64-Bit (Power8 and Power9) Installer (290 MB)

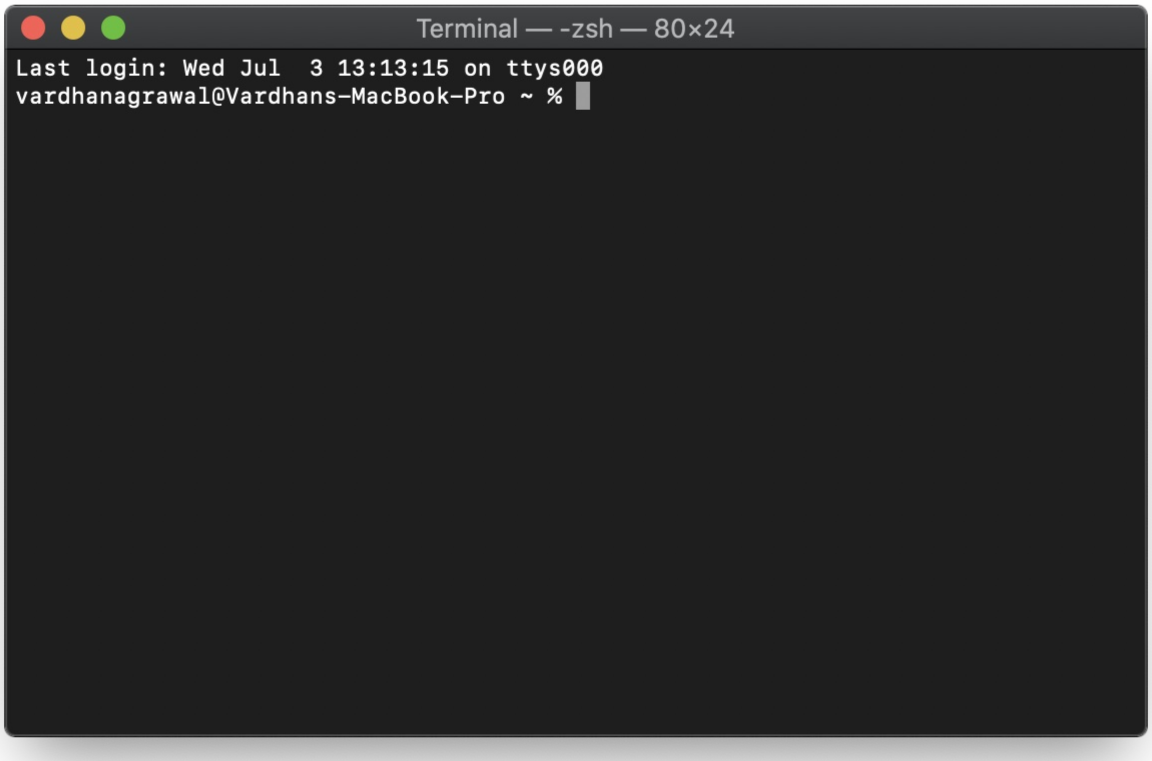
Figure 3-1: Installing Python 3.8

If you're not as comfortable with the command line installer, you can use the **64-Bit Graphical Installer** instead, which is pretty straightforward to use. However, I strongly recommend the command line installer, since it gets you adapted to the command line and is much more reliable (especially when there are conflicting installations on your computer).

Open Terminal

Once you have the file downloaded, open **Terminal**. You'll find this in a folder called **Other** in your **Applications** folder. Alternatively, you can press **Command + Space** to reveal Spotlight Search, where you can search for the Terminal.

Once you find it, you'll see a window with a black background, which looks something like this:



```
Last login: Wed Jul  3 13:13:15 on ttys000
vardhanagrawal@Vardhans-MacBook-Pro ~ %
```

Figure 3-2: Terminal Window

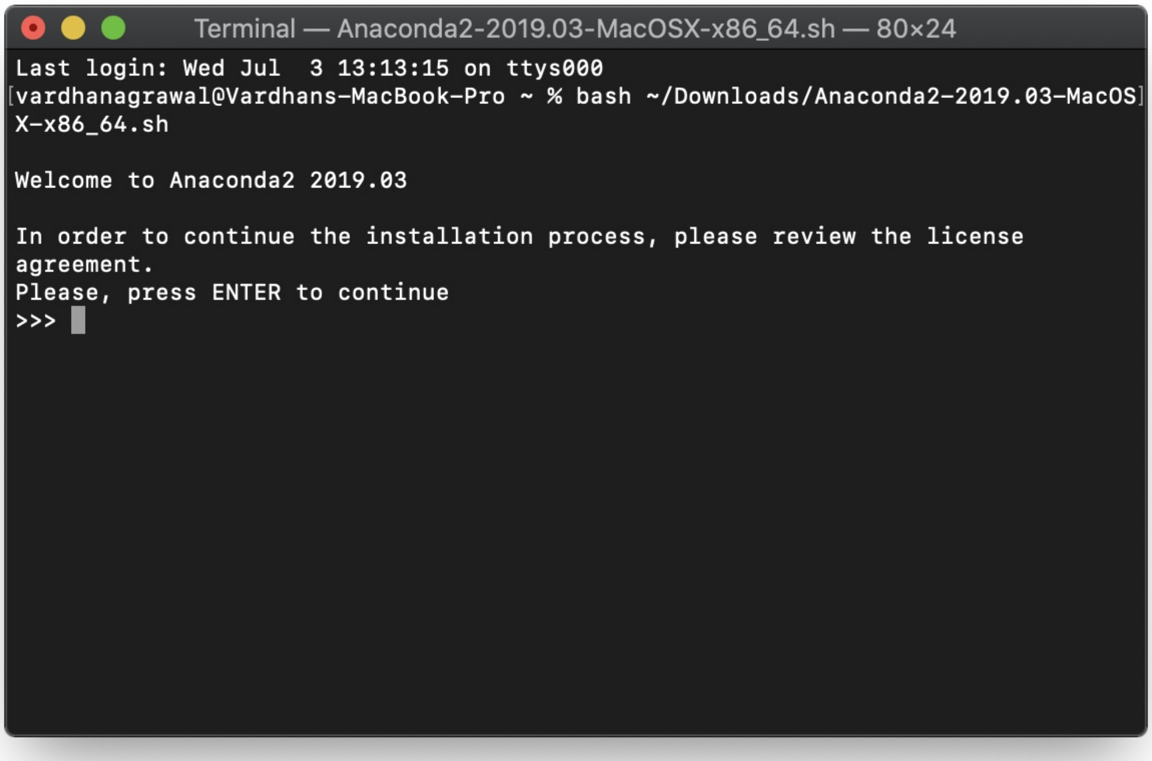
Run the Installer

Now, all you need to do is run the installer. It's worth noting that the `%` or `$` signs before the command are called prompts, and they shouldn't be entered.

To run the installer, type `bash` and then drag the downloaded file into your Terminal window. The path will auto populate and look something like this:

```
% bash ~/Downloads/Anaconda2-2019.03-MacOSX-x86_64.sh
```

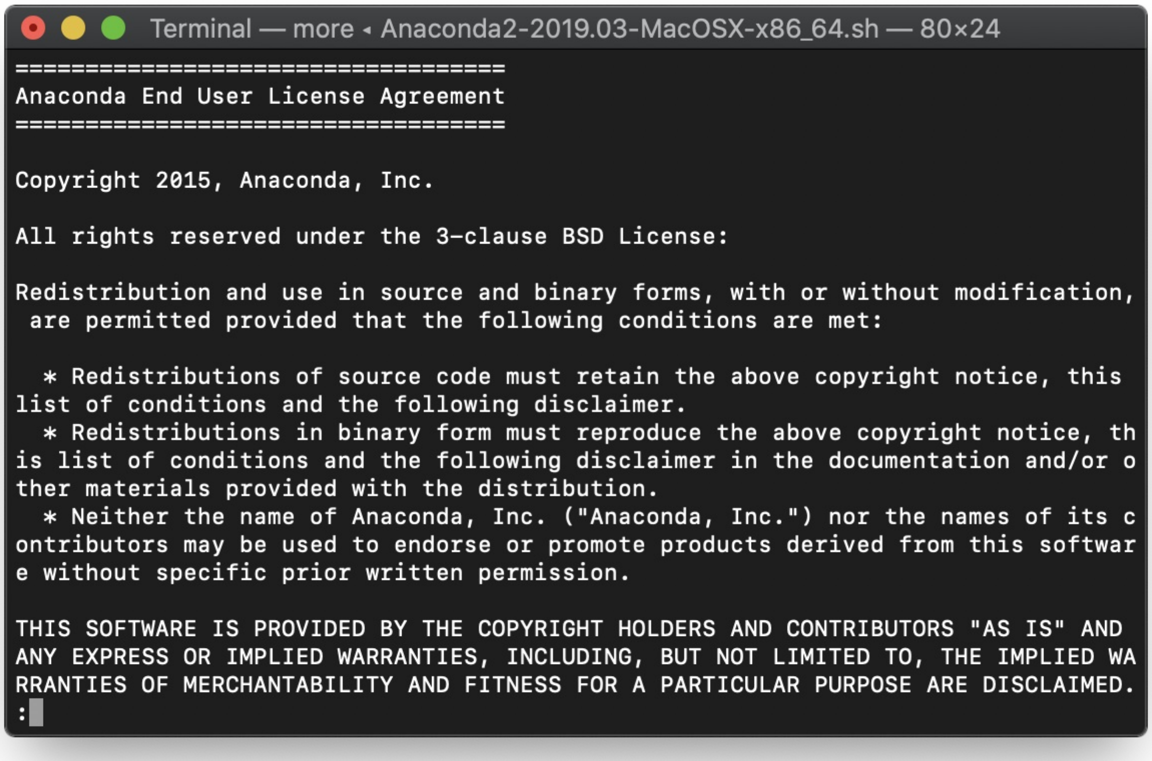
The command above tells your terminal to run the bash shell script, which you downloaded earlier. The file contains a series of commands, or steps, which allow for easy installation of Anaconda. When you've entered this command, you'll see a screen which looks like this:



```
Last login: Wed Jul  3 13:13:15 on ttys000
[vardhanagrawal@Vardhans-MacBook-Pro ~ % bash ~/Downloads/Anaconda2-2019.03-MacOS]
X-x86_64.sh

Welcome to Anaconda2 2019.03

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>> 
```

Figure 3-3: Starting the Anaconda Installer

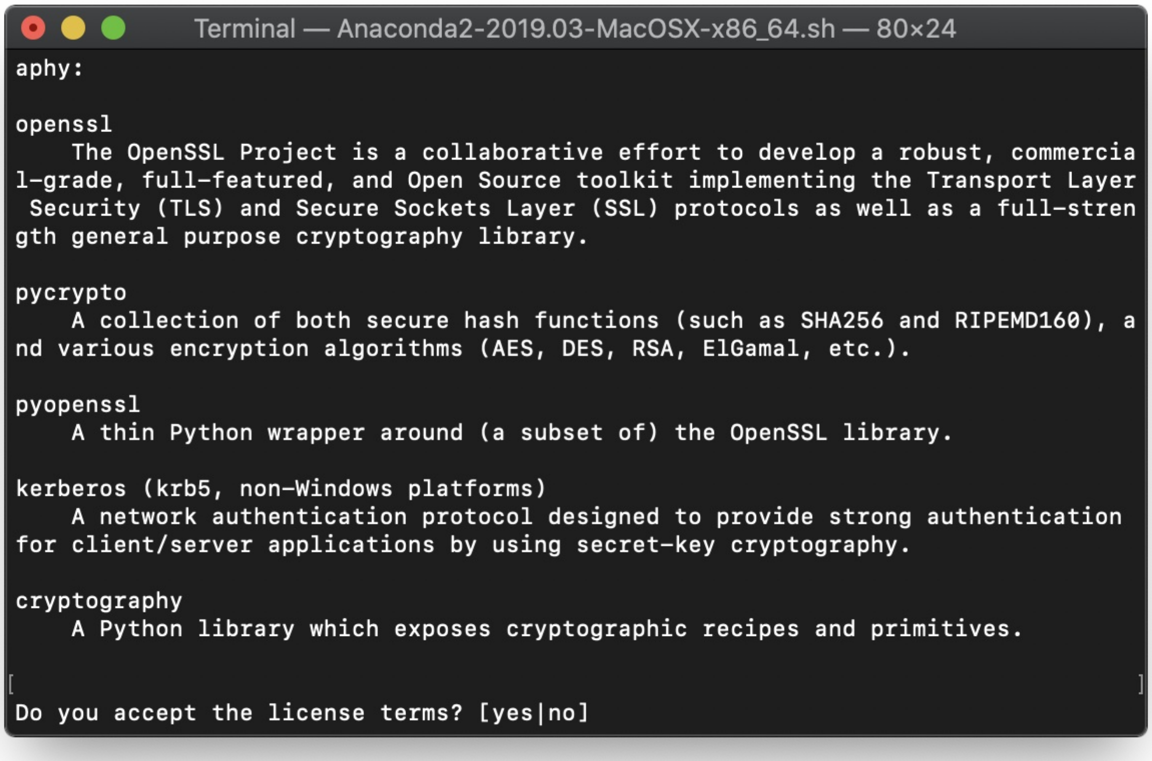
This screen should read "Welcome to Anaconda2 20XX.XX" if you've installed the correct version of Anaconda. Next, press **Enter/Space** to continue to the license agreement.


Terminal — more ◂ Anaconda2-2019.03-MacOSX-x86_64.sh — 80×24

```
===================================
Anaconda End User License Agreement
===================================

Copyright 2015, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

Redistribution and use in source and binary forms, with or without modification,
 are permitted provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.
  * Redistributions in binary form must reproduce the above copyright notice, th
is list of conditions and the following disclaimer in the documentation and/or o
ther materials provided with the distribution.
  * Neither the name of Anaconda, Inc. ("Anaconda, Inc.") nor the names of its c
ontributors may be used to endorse or promote products derived from this softwar
e without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WA
RRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
:
```

Figure 3-4: Anaconda End User Licence Agreement

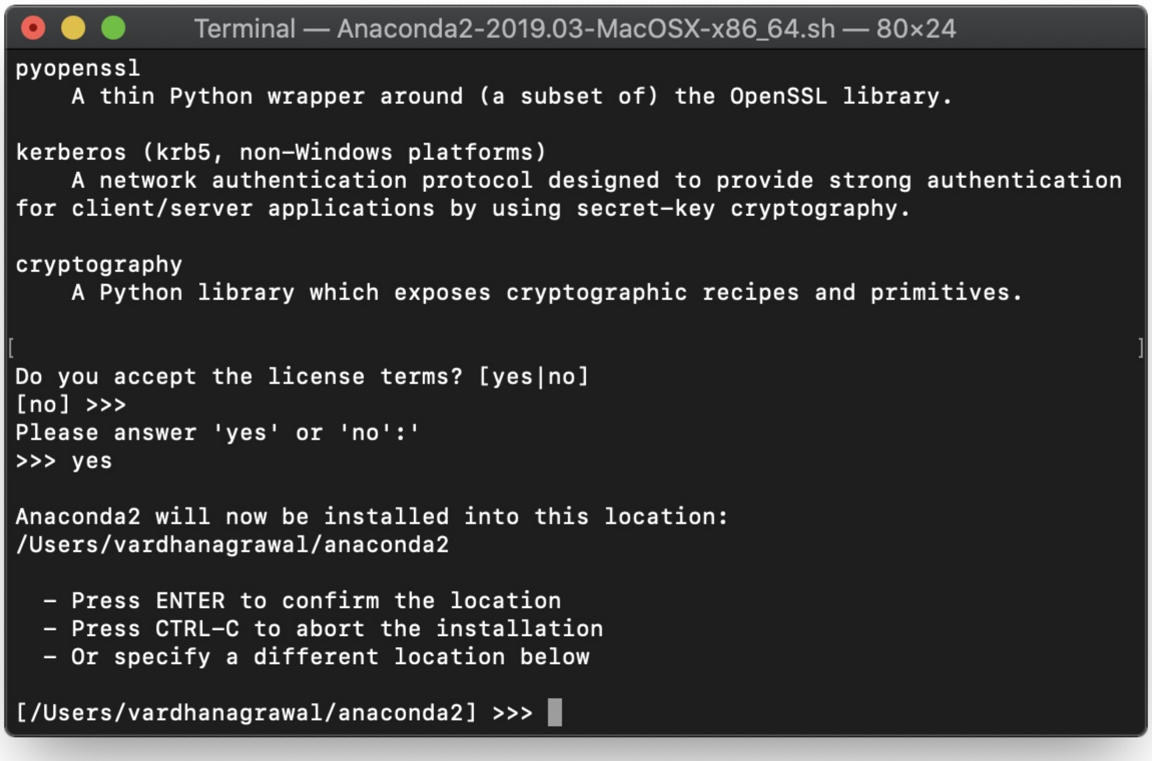
When you see the license agreement, continue pressing **Enter** several times to read through it. When you've reached the end, you'll see text which reads "Do you accept the license terms?"



```
aphy:

openssl
    The OpenSSL Project is a collaborative effort to develop a robust, commercia
l-grade, full-featured, and Open Source toolkit implementing the Transport Layer
 Security (TLS) and Secure Sockets Layer (SSL) protocols as well as a full-stren
gth general purpose cryptography library.

pycrypto
    A collection of both secure hash functions (such as SHA256 and RIPEMD160), a
nd various encryption algorithms (AES, DES, RSA, ElGamal, etc.).

pyopenssl
    A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
    A network authentication protocol designed to provide strong authentication
for client/server applications by using secret-key cryptography.

cryptography
    A Python library which exposes cryptographic recipes and primitives.


Do you accept the license terms? [yes|no]
```

Figure 3-5: Accepting the EULA

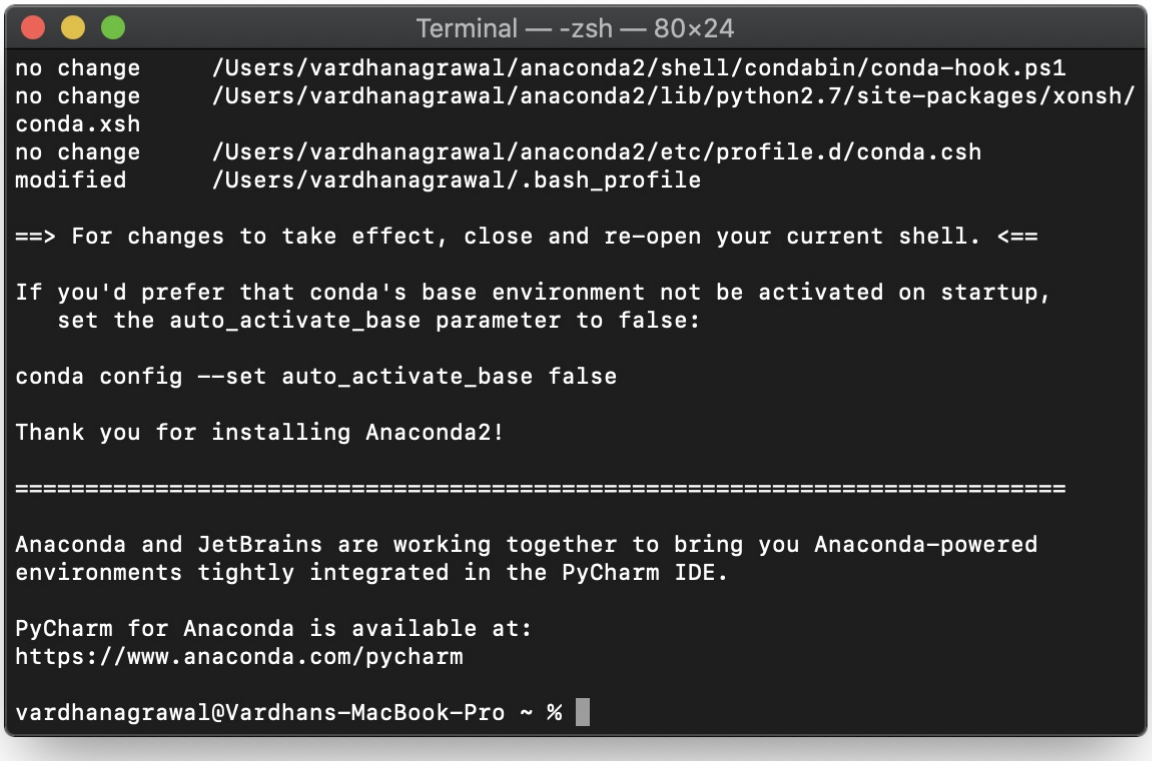
Type "yes" in your terminal window to accept the license terms.



```
pyopenssl
    A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
    A network authentication protocol designed to provide strong authentication
for client/server applications by using secret-key cryptography.

cryptography
    A Python library which exposes cryptographic recipes and primitives.

Do you accept the license terms? [yes|no]
[no] >>>
Please answer 'yes' or 'no':'
>>> yes

Anaconda2 will now be installed into this location:
/Users/vardhanagrawal/anaconda2

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/Users/vardhanagrawal/anaconda2] >>>
```

Figure 3-6: Confirming the Install Location

Then, you'll see a message which asks you to confirm your installation location. I recommend that you press **Enter** and confirm your installation, but if you'd like, you can change where you'd like to install it.



```
no change     /Users/vardhanagrawal/anaconda2/shell/condabin/conda-hook.ps1
no change     /Users/vardhanagrawal/anaconda2/lib/python2.7/site-packages/xonsh/
conda.xsh
no change     /Users/vardhanagrawal/anaconda2/etc/profile.d/conda.csh
modified      /Users/vardhanagrawal/.bash_profile

==> For changes to take effect, close and re-open your current shell. <==

If you'd prefer that conda's base environment not be activated on startup,
   set the auto_activate_base parameter to false:

conda config --set auto_activate_base false

Thank you for installing Anaconda2!

===========================================================================

Anaconda and JetBrains are working together to bring you Anaconda-powered
environments tightly integrated in the PyCharm IDE.

PyCharm for Anaconda is available at:
https://www.anaconda.com/pycharm

vardhanagrawal@Vardhans-MacBook-Pro ~ % 
```

Figure 3-7: Installation Confirmation

Finally, after a few minutes of waiting, you'll see a confirmation screen asking you to close your Terminal window and reopen it to cause the installation to take effect.

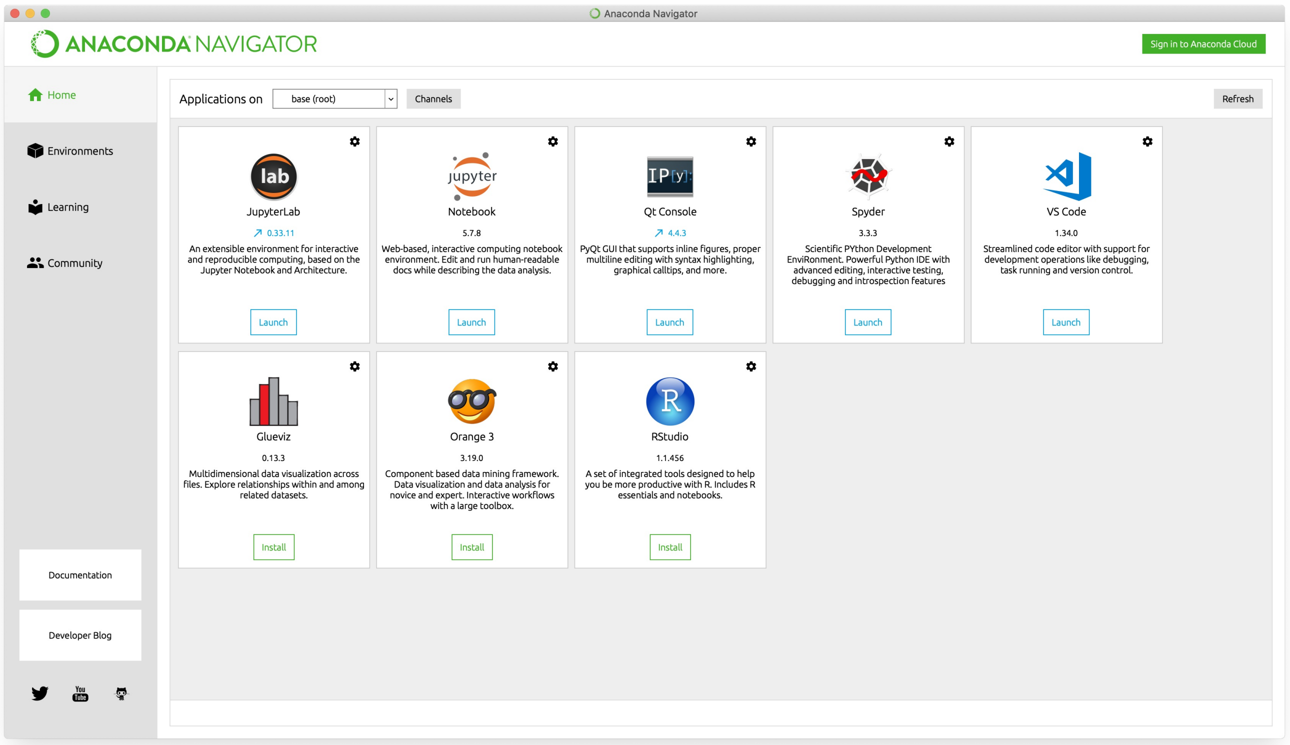
And you're done! You should be proud of yourself for installing Anaconda completely on the command line; if this is your first time using the Terminal, it's a huge accomplishment. Congratulations!

Creating a Jupyter Notebook

Here comes to the moment you've been waiting for — it's time to create your own Jupyter Notebook. In the next section, you'll learn about Python. But before that, you need to create and navigate Jupyter Notebook.

Launch Anaconda

By now, you should have Anaconda installed, which comes bundled with Jupyter Notebook. Start by launching **Anaconda Navigator**, which you can find using Spotlight Search on your Mac. The icon looks like a green circle with a geometric pattern on the left side. When you've launched it, you'll see a screen like this:



An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Streamlined code editor with support for development operations like debugging, task running and version control.

Multidimensional data visualization across files. Explore relationships within and among related datasets.

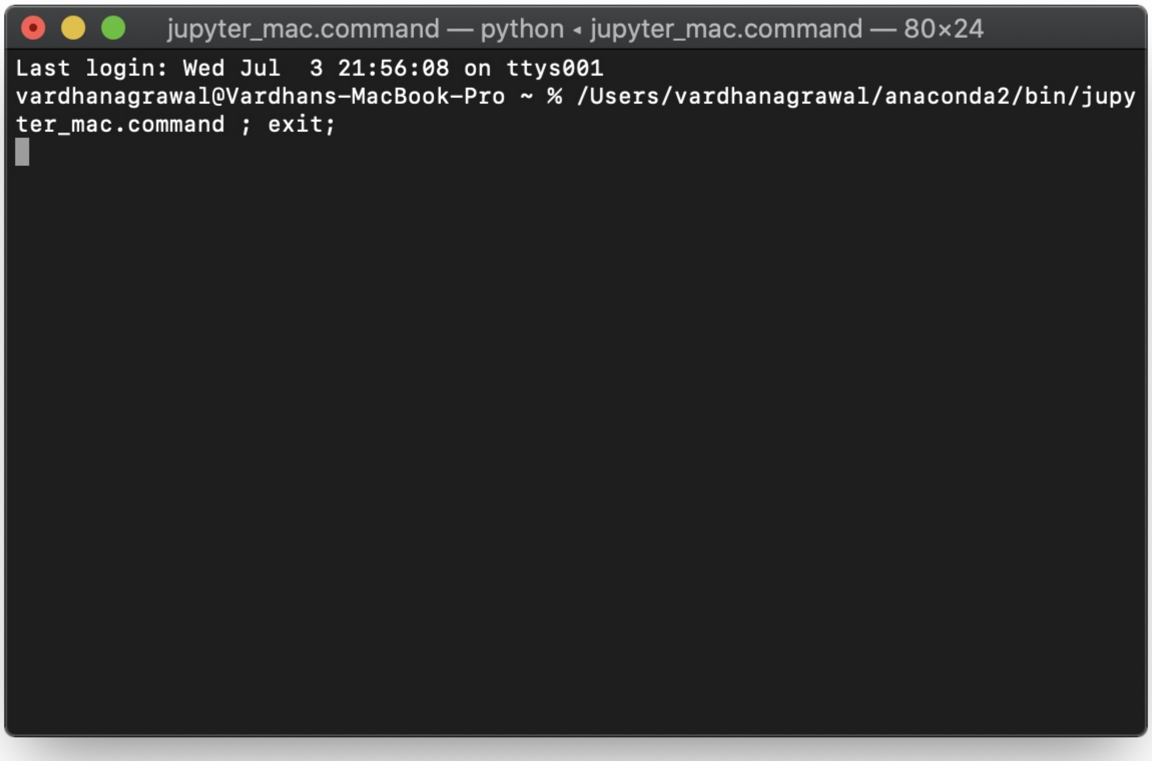
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 3-8: Anaconda Navigator

Launch Jupyter Notebook

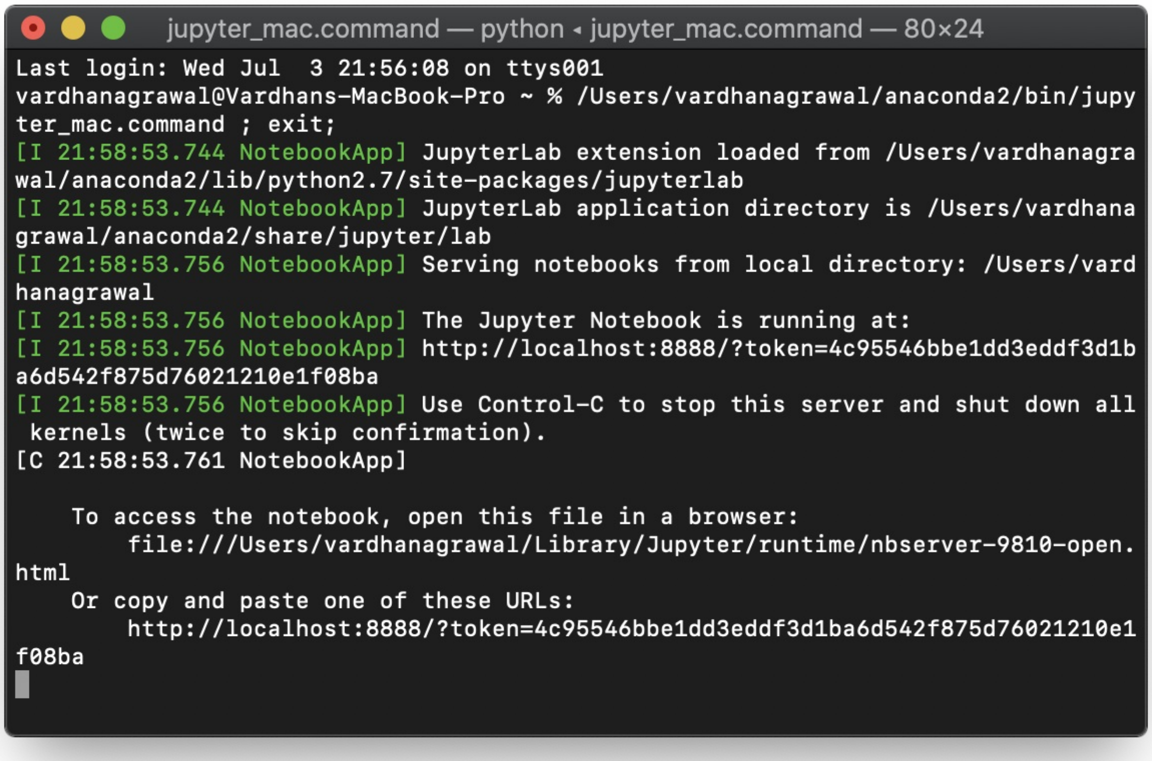
Now, click the **Launch** button, enclosed in a rectangle, underneath the box which says *Jupyter Notebook*. You'll see a Terminal window popup, which looks something like this:



```
Last login: Wed Jul  3 21:56:08 on ttys001
vardhanagrawal@Vardhans-MacBook-Pro ~ % /Users/vardhanagrawal/anaconda2/bin/jupy
ter_mac.command ; exit;
```

Figure 3-9: Command to Launch Jupyter Notebook

After a couple of seconds, the command in the terminal will execute, and you'll see another terminal window which looks like this:



```
Last login: Wed Jul  3 21:56:08 on ttys001
vardhanagrawal@Vardhans-MacBook-Pro ~ % /Users/vardhanagrawal/anaconda2/bin/jupy
ter_mac.command ; exit;
[I 21:58:53.744 NotebookApp] JupyterLab extension loaded from /Users/vardhanagra
wal/anaconda2/lib/python2.7/site-packages/jupyterlab
[I 21:58:53.744 NotebookApp] JupyterLab application directory is /Users/vardhana
grawal/anaconda2/share/jupyter/lab
[I 21:58:53.756 NotebookApp] Serving notebooks from local directory: /Users/vard
hanagrawal
[I 21:58:53.756 NotebookApp] The Jupyter Notebook is running at:
[I 21:58:53.756 NotebookApp] http://localhost:8888/?token=4c95546bbe1dd3eddf3d1b
a6d542f875d76021210e1f08ba
[I 21:58:53.756 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
[C 21:58:53.761 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/vardhanagrawal/Library/Jupyter/runtime/nbserver-9810-open.
html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=4c95546bbe1dd3eddf3d1ba6d542f875d76021210e1
f08ba
```

Figure 3-10: Executing Launch Command

You don’t need to worry about what it says, but the text will help you understand how to use Jupyter Notebook yourself. But, don’t worry if you don’t want to read it. A window will open soon in your web browser, in my case, Safari, to present the Jupyter Notebook interface. You’ll see your file system contents, and you can navigate around similar to your Finder.

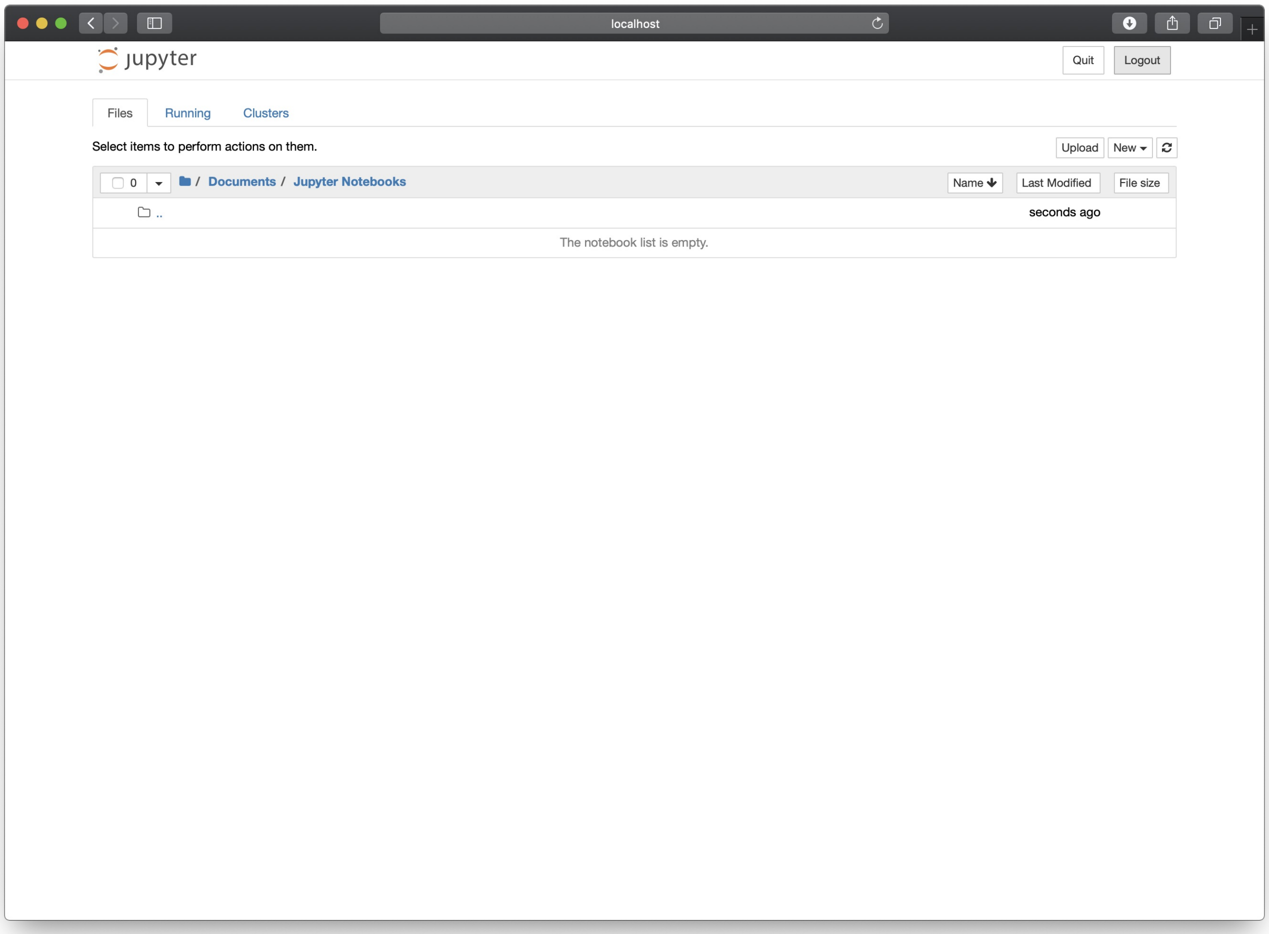

localhost
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/ Documents / Jupyter Notebooks
Name
Last Modified
File size
..
seconds ago
The notebook list is empty.

Figure 3-11: Jupyter Notebook File Directory

Create a New Notebook

With the **New** button listed at the top right, you'll be able to create a new Text File, Folder, and Terminal. If you'd like, you can create a folder to organize your Jupyter notebooks; but if not, go ahead and tap **Python 2** to create a new Jupyter Notebook in the current directory.

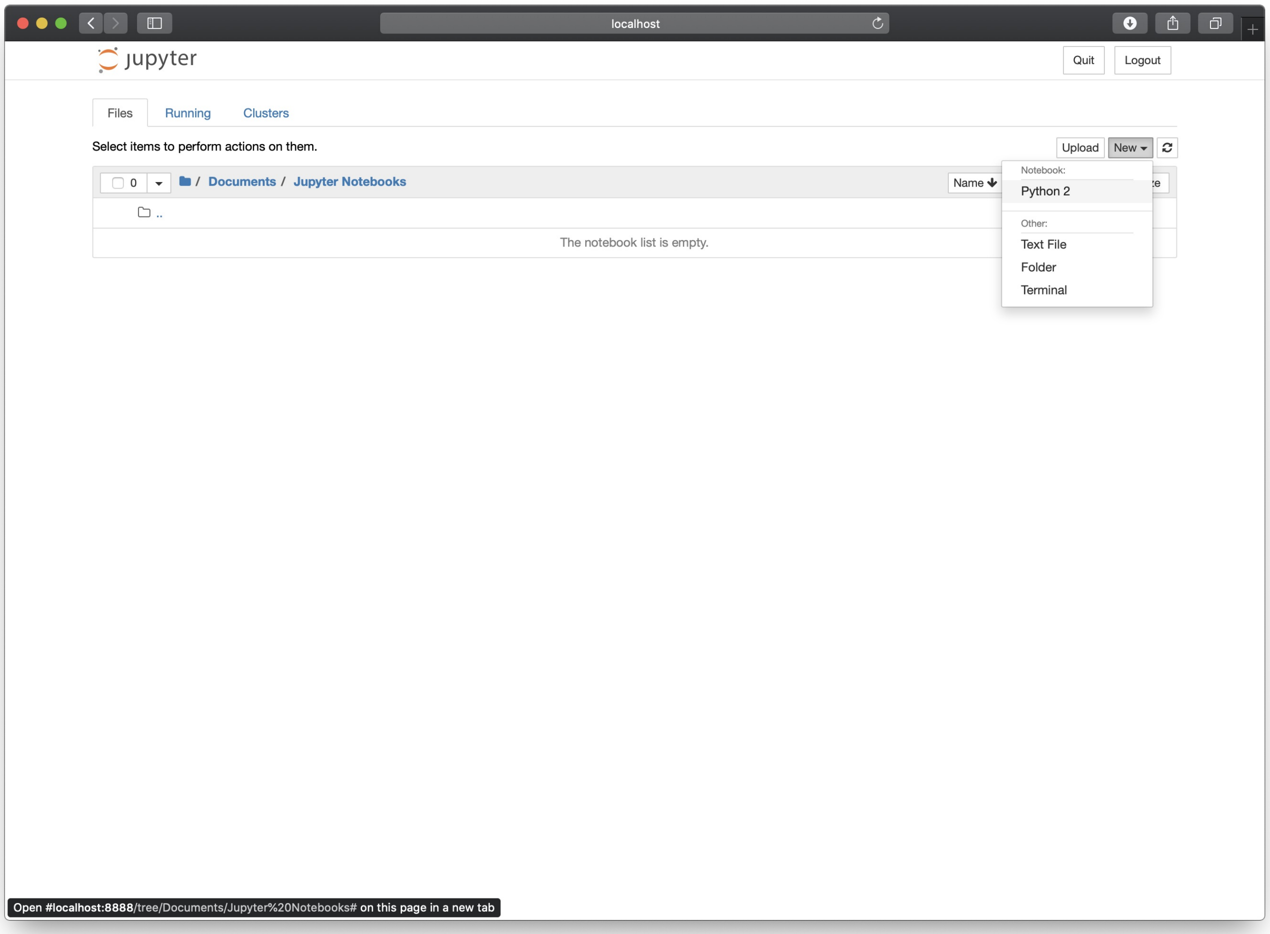

localhost
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
Documents / Jupyter Notebooks
Name
Notebook:
Python 2
Other:
Text File
Folder
Terminal
..
The notebook list is empty.
Open #localhost:8888/tree/Documents/Jupyter%20Notebooks# on this page in a new tab

Figure 3-12: Creating a Jupyter Notebook

Great! A new window should open up with your Jupyter notebook. You'll see an empty box where you can start typing code, and a word-processor-style menu along the top of the webpage.

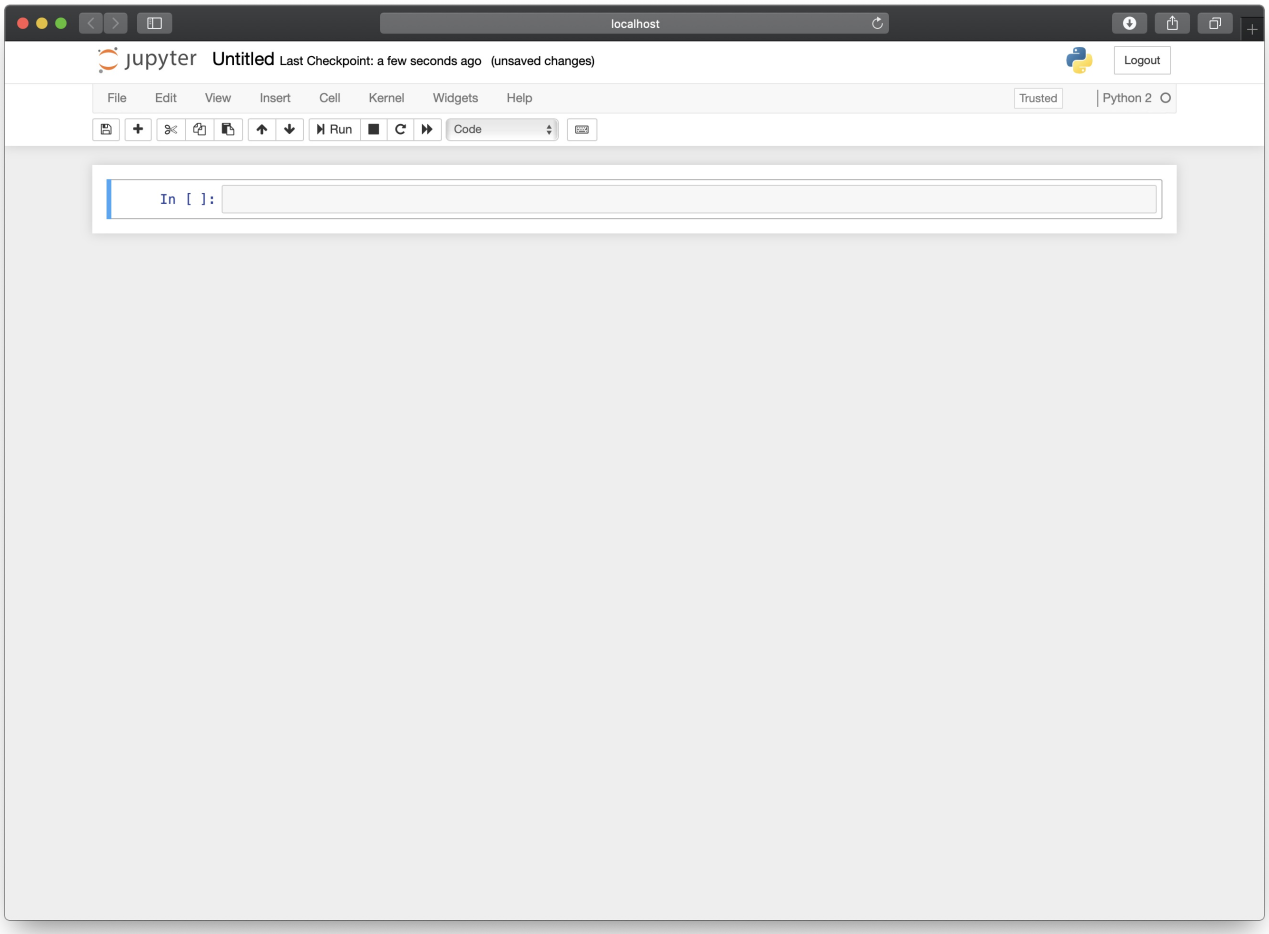
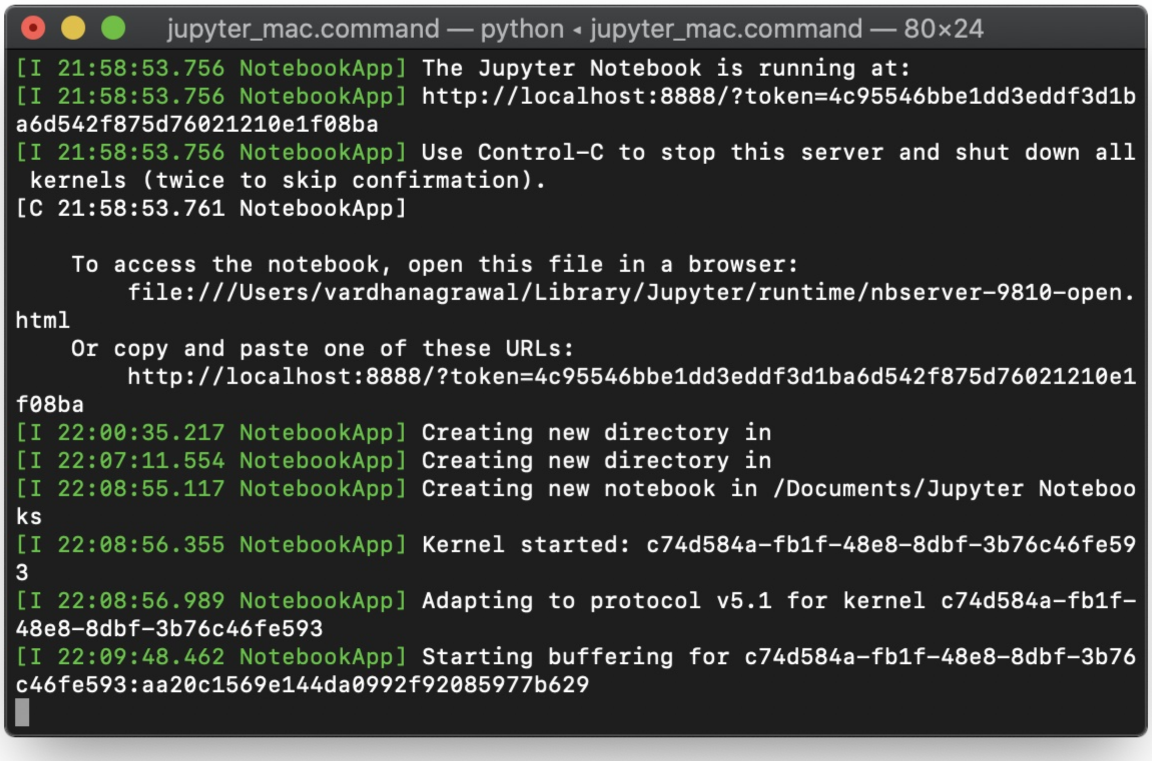


Figure 3-13: New Jupyter Notebook

You'll also see the Terminal window update as you make changes to your notebook, such as creating a new notebook, deleting a notebook, or making changes on one. After creating your notebook, your Terminal will look something like this:



```
[I 21:58:53.756 NotebookApp] The Jupyter Notebook is running at:
[I 21:58:53.756 NotebookApp] http://localhost:8888/?token=4c95546bbe1dd3eddf3d1b
a6d542f875d76021210e1f08ba
[I 21:58:53.756 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
[C 21:58:53.761 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/vardhanagrawal/Library/Jupyter/runtime/nbserver-9810-open.
html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=4c95546bbe1dd3eddf3d1ba6d542f875d76021210e1
f08ba
[I 22:00:35.217 NotebookApp] Creating new directory in
[I 22:07:11.554 NotebookApp] Creating new directory in
[I 22:08:55.117 NotebookApp] Creating new notebook in /Documents/Jupyter Noteboo
ks
[I 22:08:56.355 NotebookApp] Kernel started: c74d584a-fb1f-48e8-8dbf-3b76c46fe59
3
[I 22:08:56.989 NotebookApp] Adapting to protocol v5.1 for kernel c74d584a-fb1f-
48e8-8dbf-3b76c46fe593
[I 22:09:48.462 NotebookApp] Starting buffering for c74d584a-fb1f-48e8-8dbf-3b76
c46fe593:aa20c1569e144da0992f92085977b629
```

Figure 3-14: Notebook Updates in Terminal

When you close your notebook (*File > Close and Halt*), you'll see it appear as a `.ipynb` file in both your Finder and the file navigator within Jupyter Notebook. It may look like this:

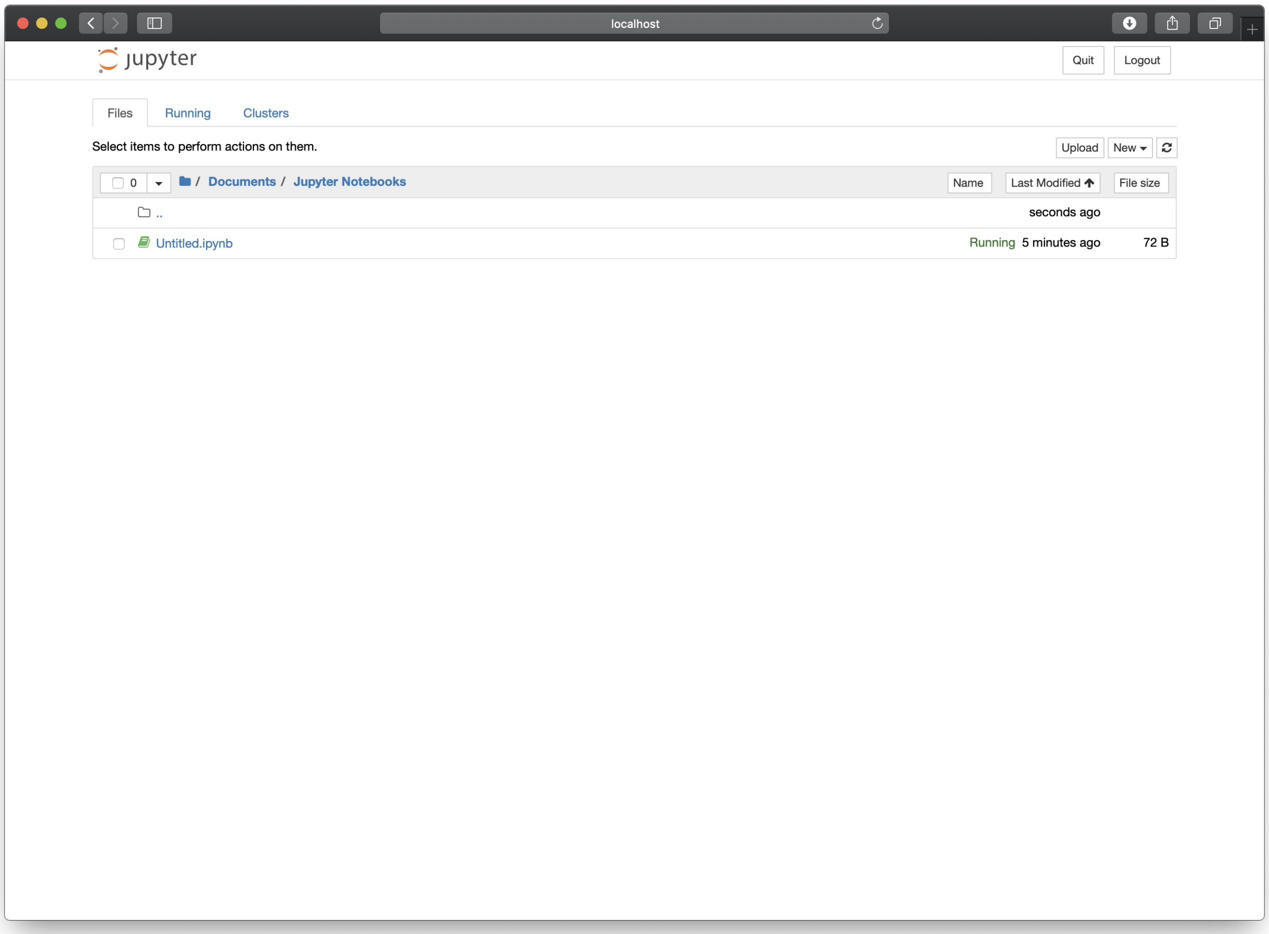
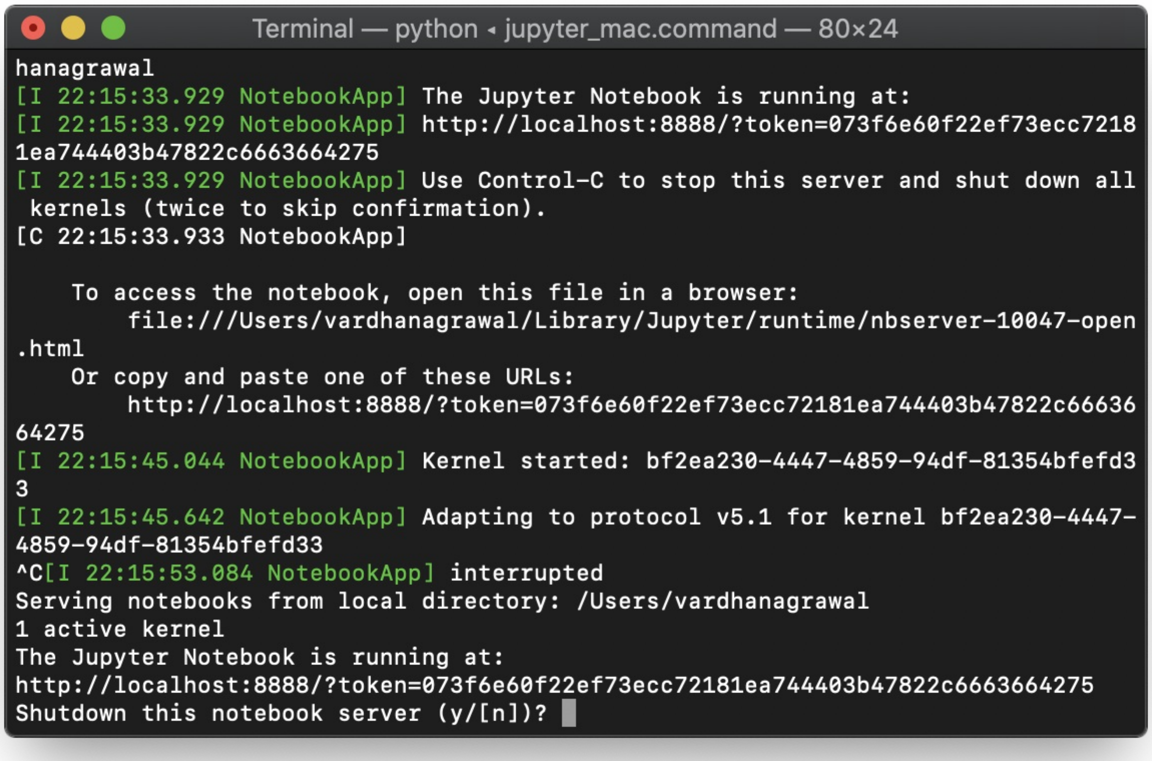


Figure 3-15: Running Jupyter Notebook

Exiting Jupyter Notebook

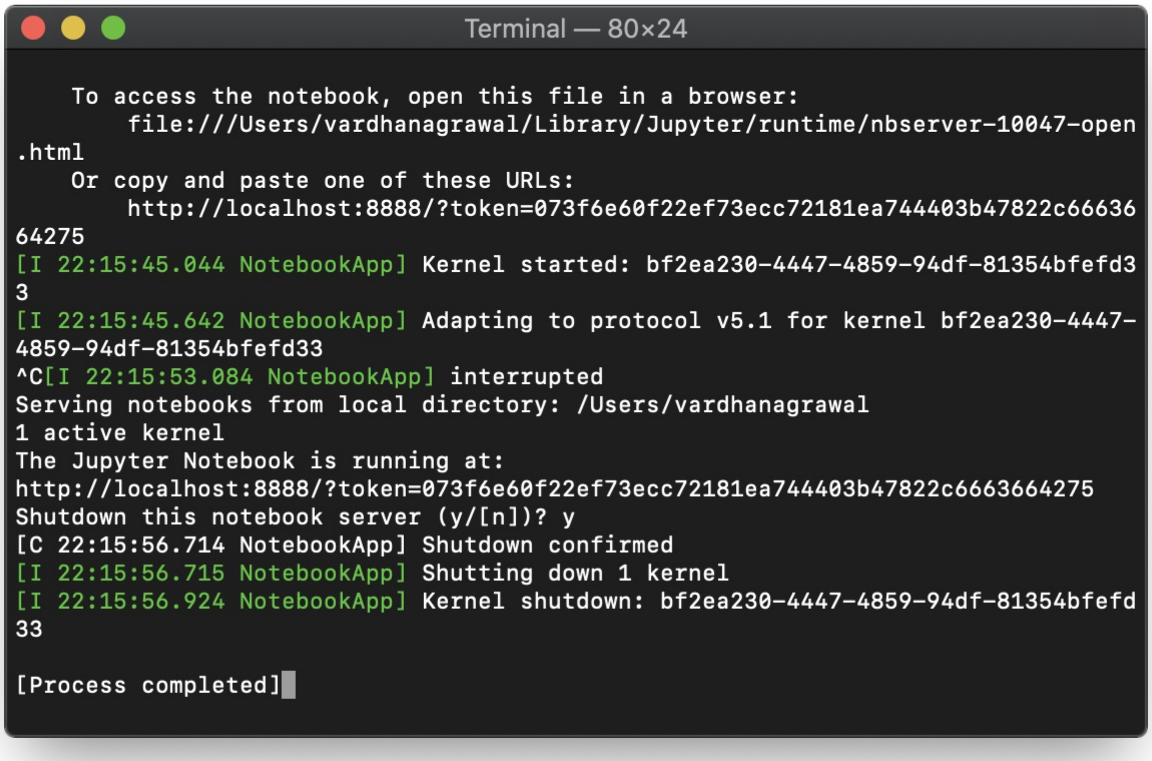
For now, we're done. Now, you need to close your Jupyter Notebook so that it doesn't run eternally on your computer. To do this, head back to your Terminal window and click **Control** + **C** to quit Jupyter Notebook. You'll then see a confirmation message, asking you to confirm whether you'd like to quit your notebook:



```
hanagrawal
[I 22:15:33.929 NotebookApp] The Jupyter Notebook is running at:
[I 22:15:33.929 NotebookApp] http://localhost:8888/?token=073f6e60f22ef73ecc7218
1ea744403b47822c6663664275
[I 22:15:33.929 NotebookApp] Use Control-C to stop this server and shut down all
 kernels (twice to skip confirmation).
[C 22:15:33.933 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/vardhanagrawal/Library/Jupyter/runtime/nbserver-10047-open
.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=073f6e60f22ef73ecc72181ea744403b47822c66636
64275
[I 22:15:45.044 NotebookApp] Kernel started: bf2ea230-4447-4859-94df-81354bfefd3
3
[I 22:15:45.642 NotebookApp] Adapting to protocol v5.1 for kernel bf2ea230-4447-
4859-94df-81354bfefd33
^C[I 22:15:53.084 NotebookApp] interrupted
Serving notebooks from local directory: /Users/vardhanagrawal
1 active kernel
The Jupyter Notebook is running at:
http://localhost:8888/?token=073f6e60f22ef73ecc72181ea744403b47822c6663664275
Shutdown this notebook server (y/[n])?
```

Figure 3-16: Shutting Down Jupyter Notebook

When you see this message, you have 5 seconds to type `y` or `Y` to confirm your notebook shutdown, otherwise, your notebook will continue running. Finally, when you're able to confirm that you want to shutdown your notebook server, you'll get a message which says `[Process completed]` to confirm.



```
    To access the notebook, open this file in a browser:
        file:///Users/vardhanagrawal/Library/Jupyter/runtime/nbserver-10047-open
.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=073f6e60f22ef73ecc72181ea744403b47822c66636
64275
[I 22:15:45.044 NotebookApp] Kernel started: bf2ea230-4447-4859-94df-81354bfefd3
3
[I 22:15:45.642 NotebookApp] Adapting to protocol v5.1 for kernel bf2ea230-4447-
4859-94df-81354bfefd33
^C[I 22:15:53.084 NotebookApp] interrupted
Serving notebooks from local directory: /Users/vardhanagrawal
1 active kernel
The Jupyter Notebook is running at:
http://localhost:8888/?token=073f6e60f22ef73ecc72181ea744403b47822c6663664275
Shutdown this notebook server (y/[n])? y
[C 22:15:56.714 NotebookApp] Shutdown confirmed
[I 22:15:56.715 NotebookApp] Shutting down 1 kernel
[I 22:15:56.924 NotebookApp] Kernel shutdown: bf2ea230-4447-4859-94df-81354bfefd
33

[Process completed]
```

Figure 3-17: Shutdown Confirmation Message

3-2 Basic Python

In the previous section, you learned about what Jupyter Notebook is as well as how to install and set them up on your computer. You also created your first Jupyter Notebook and learned about how Anaconda can help you use various other tools for your data science needs. You now have a better sense of how you can use these tools in your professional machine learning workflow.

In this section, you'll learn about Python, the language in which the world of machine learning is written. Essentially, you'll have a whole new programming language under your belt by the end of the section. If you already have some Python experience, you may skip this section; however, you'll review your fundamentals and may even learn something new about Jupyter Notebook if you stick around.

Variables, Loops, and Control Flow

First, let's get the bare-bone fundamentals out of the way. Variables, loops, and if-statements are likely concepts you've come across in other languages, including Swift. While the syntax may be slightly different in Python, the core fundamentals are very similar.

Variables

In Python, declaring variables is easy. All you need to do is write the variable name, followed by the value of the variable. Like Swift, Python automatically determines the type of the variable, so you don't have to explicitly tell it whether your variable is an integer, boolean, or string.

If I wanted to create a `name` variable and set it to my name, I could do it like this:

```
name = "Vardhan"
```

Easy as that! No semicolons or keywords to use; all you needed to do was say what you wanted. You can now access the `name` variable to mean the string `"Vardhan"`

Loops

As you may know, loops allow programmers to repeat actions without copying and pasting the same code multiple times. If one needed to print a million asterisks, they would use a loop instead of pasting the line to do so a million times.

While Loops

`While` loops allow you to repeat a certain action until a specific condition is met. For instance, if you wanted to print "hello" and increment a variable until that variable is greater than 10, you would do the following:

```
num = 0
while (num < 10):
    print("hello")
    num = num + 1
```

As you can tell in this example, the variable `num` is set to a default value of `0`. Before the code inside the `while` loop is executed, Python checks if `num` is less than `10`. If the test succeeds, the body of the loop is executed, resulting in "hello" being printed and `num` being incremented by one.

For Loops

`For` loops help you get more quantitative control of your looping. If you need to print a list of numbers from 1 to 10, you can easily do this with a for loop, and this task would likely require significantly more effort with a while loop instead. To loop through a set of numbers, you can do the following:

```
for num in range(1, 10):
    print(num)
```

By default, the variable `num` will get incremented by `1`, but you can increment it by another number by adding a third parameter to the range. You can increment it by `3`, for example, as follows:

```
for num in range(1, 10, 3):
    print(num)
```

You can also loop through other types, such as strings, arrays, and tuples. To loop through the characters of the word `apple`, you could do this:

```
word = "apple"
for char in word:
    print(char)
```

In this example, the variable `word` is defined as the string `"apple"`, and we're looping through each character in the string by defining the loop with an existing variable parameter, as opposed to the range function.

Control Flow

Now, you might only want certain code to run under certain conditions, and you can control the flow of your code via `if` and `if-else` statements. Of course, you're likely familiar with this in Swift, and the concept is nearly identical in Python. Examine these lines of code:

```
num = 1
if num < 0:
    print("false")
```

Straightforward, right? If the variable `num` is less than `0`, the program prints "false," otherwise, it prints nothing. But, if you wanted to print something in that otherwise case, you could use an `else` statement such as this:

```
num = 1
if num < 0:
    print("false")
else:
    print("true")
```

Finally, if you want to give the `else` parameter a specific condition, you can do the following to narrow it down further:

```
num = 1
if num < 0:
    print("false")
elif num == 1:
    print("right on")
else:
    print("true")
```

The `elif` statement specifies that if the `num` variable is not less than `0`, then the compiler should check if `num` is equal to `1`. If `num` is not equal to `1`, it should default to printing "true."

Data Structures and Functions

As you may have guessed, Python has many of the same data structures as Swift, Java, and other object-oriented programming languages. Similarly, Python also has primitive data structures and non-primitive data structures, both are likely familiar to you.

Primitive Data Structures

Primitive data structures are those which are the most basic — or pure — types in Python. They're essentially the building blocks of the language and are used quite frequently in day-to-day programming.

Integers and Floats

Integers and floats are both ways of representing numbers. In Swift, these two types work in the same way. An integer can represent whole numbers, such as `1`, `5`, and `9`, while floats can represent numbers with floating points, such as `1.1`, `5.89`, and `3.14`. As you may imagine, these can be useful for a variety of purposes, and you probably already use them without thinking about it.

You can declare an integer like this:

```
num = 1
x = 3
integer = 9
```

Or, you can declare a float like this:

```
num = 1.1
y = 5.89
float = 3.14
```

Booleans

If you only need to represent a true or false value, you can use a boolean, again, something identical to Swift. You can declare a boolean as follows:

```
ketchup = false
mustard = true
```

Simple enough, right? Booleans are perfect to use with control flow, since you can logically execute your code in an easy-to-read way.

Strings

Knitting is a great way to express your creativity — not those type of strings! Strings are a series of characters which spell out words, sentences, or even nothing at all.

In almost all languages, including Python, you can use single or double quotes to define strings. You can define a string like this:

```
name = 'Vardhan'
```

Or, for double quotes, you can do this:

```
name = "Vardhan"
```

And, you can use the `+` operator to concatenate, or add together, two strings and make a larger string, such as the following:

```
first = "Swift "
second = "Rules!"

print(first + second)
```

The program will print the string "Swift Rules!" after combining the two strings `first` and `second` together.

Non-Primitive Data Types

There are many non-primitive data types, unlike the primitives, which there are only four of. Since I'm not teaching a full computer science course here, we'll only talk about the main ones, but you're welcome to Google more if you're interested in learning about the ones not covered in this chapter.

Arrays and Lists

Arrays are one of my favorite data types, since they allow you to store a list of several data types. For example, you can convert a string into an array of characters, or you could store the items in your to-do list in an array. However, in Python, the data structure we call arrays in Swift are known as lists.

While Python does have arrays, they work quite differently with that arrays in Swift. Lists, on the other hand, work exactly the same as arrays in Swift. Other than the confusing naming, they're quite similar in their usage. You can declare a list like this:

```
nums = [1, 5, 9]
```

The list above contains three integers, which are `1`, `5`, and `9`. Unlike Swift, you can put different types in the same array, like this:

```
nums = [true, "like", 3.14, "?"]
```

While it seems unnatural to us Swift developers, the above is actually valid. We can store the boolean `true`, the string "like", the float `3.14`, and the string "?" in the same array.

To access values of an array, you must specify the index at that location and use subscript notation like the following example:

```
nums = [true, "like", 3.14, "?"]
word = nums[1]
```

The variable `word` will be set to "like," since it's located at index 1 of the `nums` array. Remember, arrays always begin at 0, not 1. You can also change values of an array in the same way, using subscript notation.

Tuples

You may or may not be familiar with tuples, but they're quite useful, especially when you don't need the full functionality of an array but still need to store more than one value. One limitation, however, is that tuples are immutable, meaning that they can't be changed after they're defined. You can create a tuple as follows:

```
my_tuple = 1, 4
my_other_tuple = (1, 5)
```

As you can see, there are two ways of writing a tuple: one with parentheses and one without. Similar to an array, you can access tuples using subscript notation:

```
mine = my_tuple[0]
```

Here, the variable `mine` will be set to the value `1`, as common sense might dictate. However, different from arrays, you cannot set tuples in this way.

Dictionaries

As with arrays, you've likely used dictionaries quite extensively in Swift, especially if you've dealt with JSON or networking calls of any kind. A dictionary is just as it sounds: a set of keys and values. Here's how you create a dictionary in Python:

```
contacts = { "Vardhan" : "555-555-5555", "Simon" : "333-3333-3333" }
```

In this dictionary, the two keys are "Vardhan" and "Simon," and their phone numbers — or the associated values — are "555-555-5555" and "333-333-3333," respectively. It's worth noting that the keys and values don't necessarily have to be of the same type in a dictionary.

To access values in a dictionary, you use the key as opposed to the index like in arrays. To get the phone number associated with "Vardhan," you would do the following:

```
phone_num = contacts["Vardhan"]
```

The `phone_num` variable is now set to "555-555-5555" using the subscript notation as we did for arrays and tuples.

Functions

Instead of reusing code, we use functions. You use functions all the time in Swift: not only do you write functions, but you also take advantage of the built-in ones. By using the `def` keyword (similar to `func` in Swift), you can define a function like this:

```
def the_best_function():
    print("I'm actually useless.")
```

The function won't actually do anything until it's called, so to call it, you simply type out the name of the function, again, similar to Swift:

```
the_best_function()
```

To pass parameters into functions, just declare a variable name inside the parentheses to tell Python that it should expect a value passed in when the function is called. To do this, do the following:

```
def my_email(email):
    print("My email is: " + email)
```

When you call the `my_email()` function, you'll need to pass in an email parameter, which, in this case, will be printed out in the program's output.

Classes, Methods, and Objects

To wrap up this section, let's briefly look at classes and objects to see how you can put the "object" in "object-oriented" for the purposes of Python.

Classes

A class, as with Swift, is similar to a template. It outlines how to create an object, but isn't one on its own. To create a class, you must use the `class` keyword like shown below:

```
class Person:
    name = "Tim"
```

In this example, the class `Person` has one attribute: the `name` variable. When creating an instance of this class, known as an object, you can access this field and change it depending on your intent.

Methods

You can also create methods to do various tasks. When functions belong to a class, they're referred to as methods. To change the name of `name`, you could write a function called `rename`, like this:

```
class Person:
    name = "Tim"
    def rename(self, updated_name):
        self.name = updated_name
```

You'll notice that the first argument that `rename(:)` takes in is `self`. This is required for a method in Python. The second parameter is the new name. Inside the method body, you set `self.name` to `updated_name`.

You may be wondering about the `init` method. You can also add an `init` method like this, which allows you to avoid hardcoding your attributes:

```python
def __init__(self, name):
    self.name = name
```

Objects

When you instantiate a class, it's called an object. As mentioned before, a class is a blueprint, and an object is a concrete representation of it. Think of it like a house plan and the house itself. To create an object, do the following:

```python
man = Person()
```

Now, `man` contains an instance of the `Person` class, and you can access the `name` attribute using dot notation. For example, you can set a name variable (outside of the class) as follows:

```python
name = man.name
```

To change the name, we can use the method we created earlier. Again, using dot notation, you can call the method on `man` :

```python
man.rename("Timothy")
```

Now, if you try to access `man.name` , you'll find that it's been changed to "Timothy" from "Tim." However, this change is only valid for the `man` object, not for other instances of the `Person` class.

3-3 Uses of Jupyter Notebook

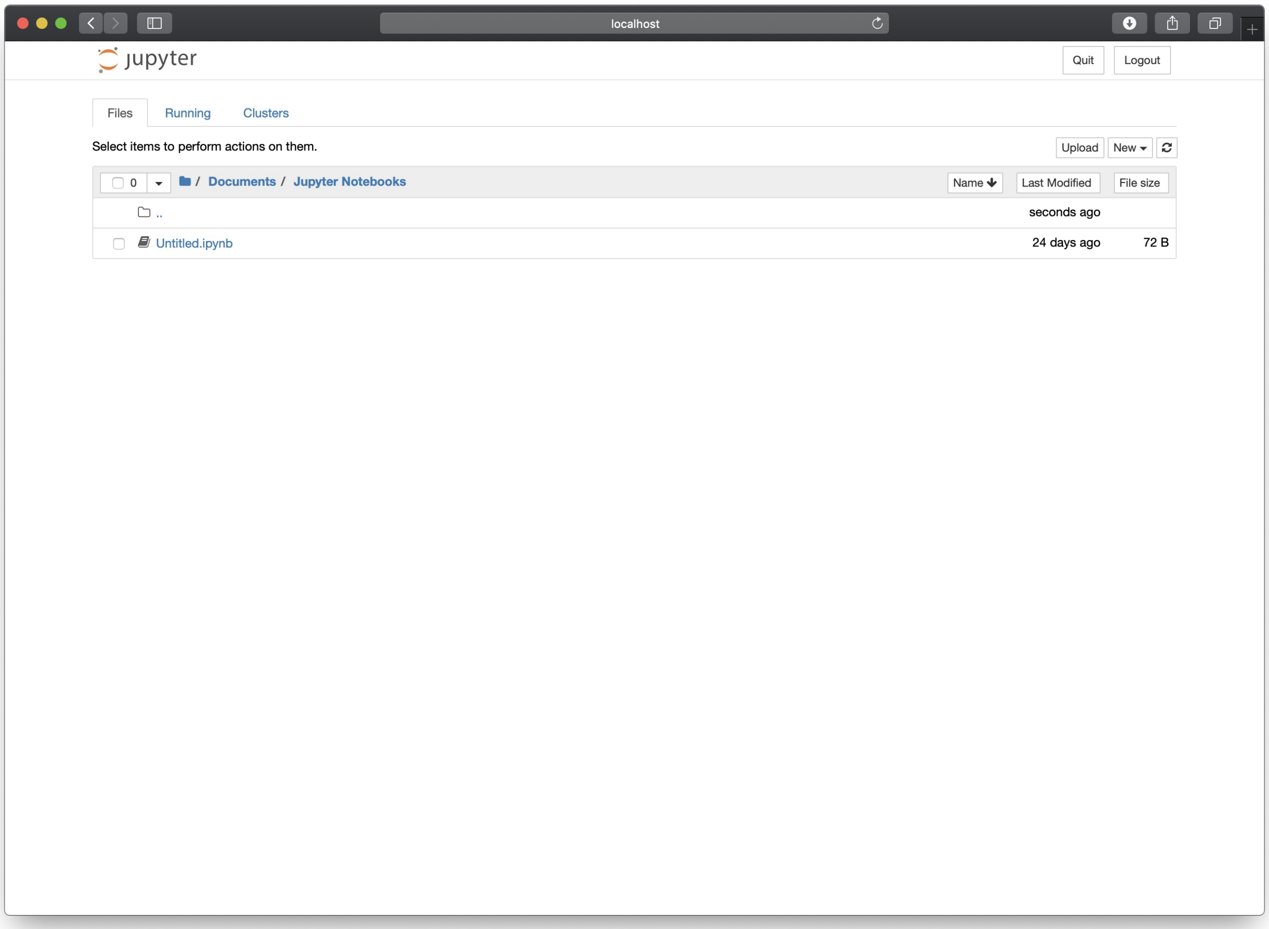
In the previous section, you learned about Python. If you didn’t have any experience with Python, you should now be at a level where you can do basic things in Python. Later in the book, you’ll hone your skills further by building real-world models and programs in this new language. You can now relate your Python skills to Swift and see how similar the two languages are.

Earlier I covered how to install Jupyter Notebook, now let's learn about how you can use Jupyter Notebook, including styling your text, writing clear descriptions, and interlacing your newly learned Python code. You’ll also learn how to represent mathematical expressions using LaTeX and create various heading styles.

Code and Prose

As we discussed earlier in the chapter, one of the major benefits of using Jupyter Notebook is the ability to intersperse code and prose — that is, descriptions — to make your code more like an article or a book, rather than a plain-old Python file. You can also use the output from your code and make your text-based elements of your Jupyter Notebook update dynamically.

Start by creating a new Jupyter notebook like you did earlier in the chapter. No worries if you don’t remember; feel free to refer the pages towards the beginning of the chapter to strengthen your skills. Once you’re ready, create a new Jupyter Notebook:



Select items to perform actions on them.

/ Documents / Jupyter Notebooks

..	seconds ago	
Untitled.ipynb	24 days ago	72 B

Figure 3-18: Jupyter Notebook File Directory

Mine, as you can see, is called **Untitled.ipynb**. After clicking on it, it opens and appears something like this:

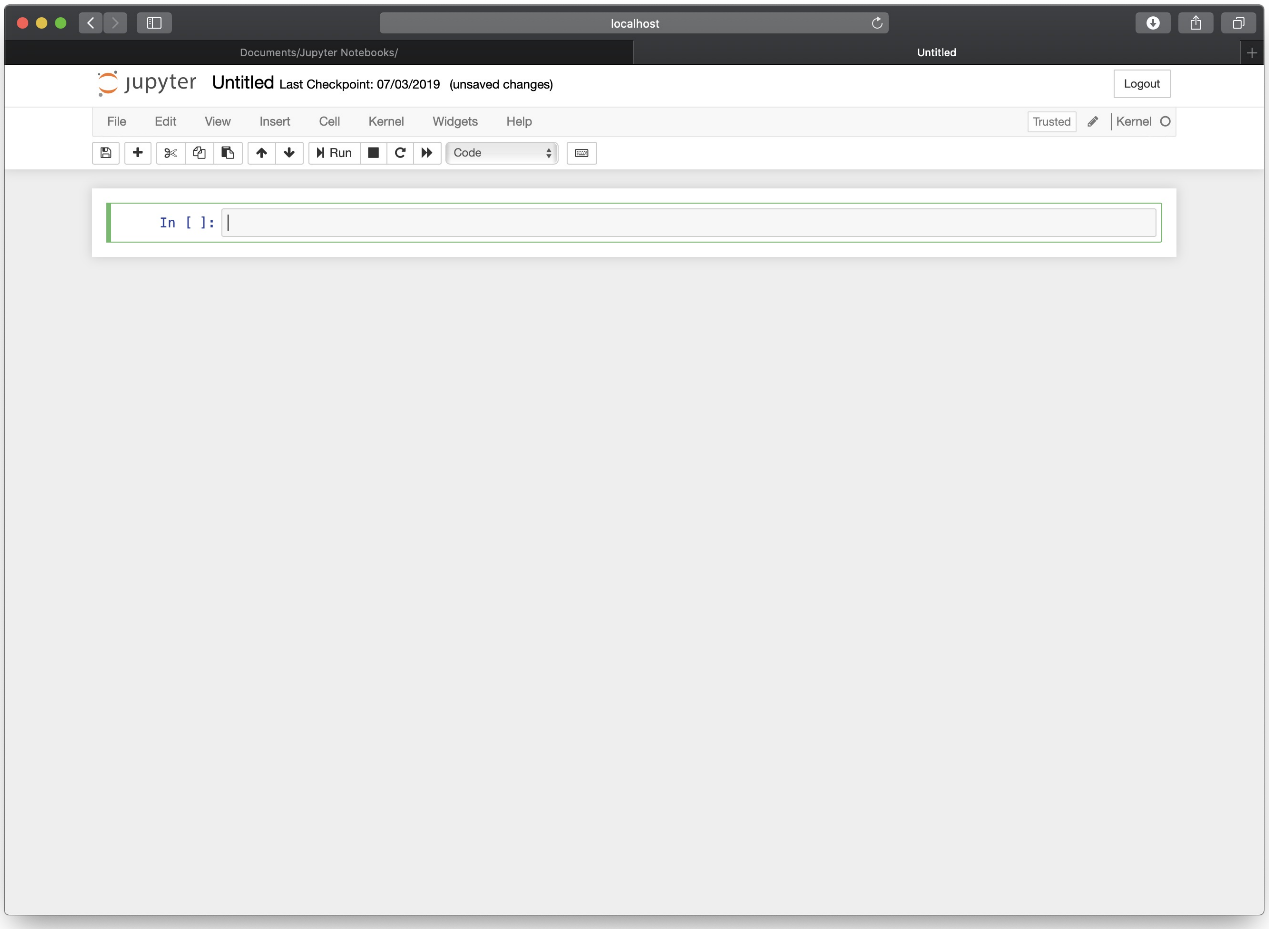


Figure 3-19: New Jupyter Notebook

Great! You now have a new Jupyter Notebook. You can also use the old one you created earlier to proceed.

Python Output

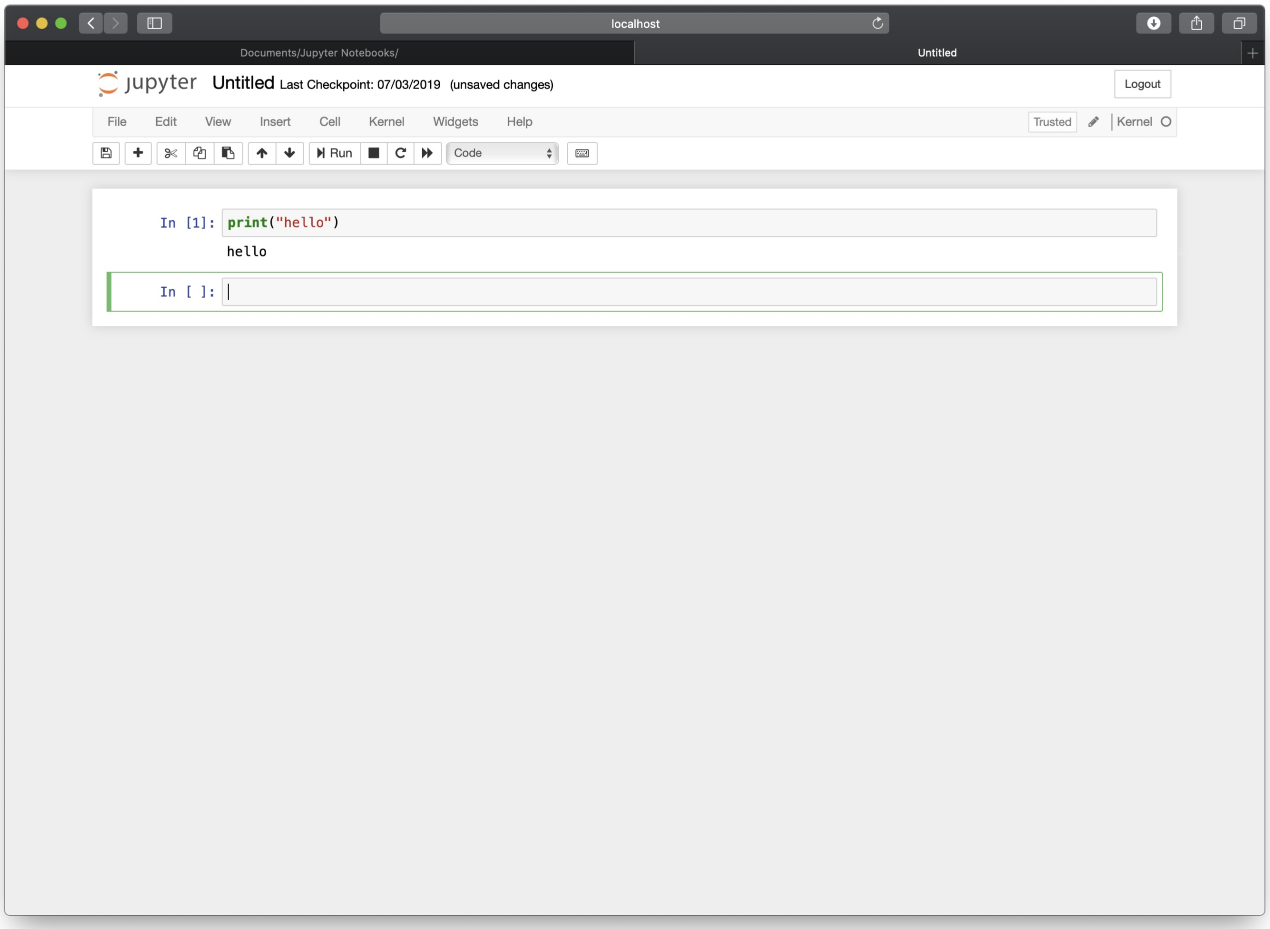
One of the great things about Jupyter Notebook is the ability to get output from individual cells, instead of needing to run the entire notebook for one single output file in the end.

Printing a Message

When you first open your notebook, your cursor will be on the first cell. Here, you can type Python code. For our first example, print the following:

```
print("hello")
```

Pretty self-explanatory, right? And, you must be wondering why we didn't type "Hello World" — it's simple. It's overused, but that's besides the point. When you hit the **Run** button in the toolbar, you'll see the output which looks like this:


localhost
Documents/Jupyter Notebooks/
Untitled
jupyter Untitled Last Checkpoint: 07/03/2019 (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Kernel
Run
Code

```
In [1]: print("hello")
        hello
In [ ]:
```

Figure 3-20: Printing a Message

Printing a List of Numbers

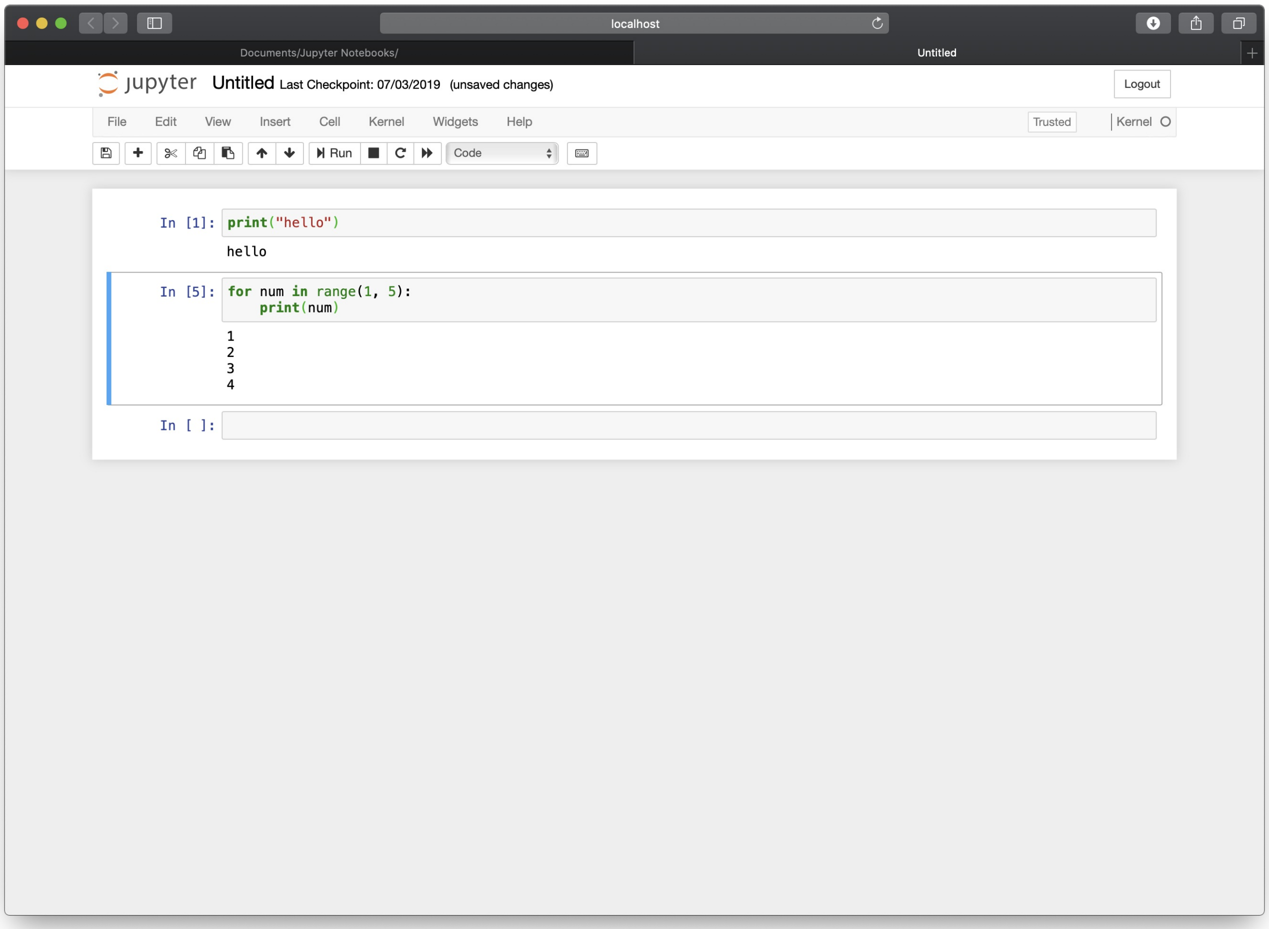
Now, since you haven't yet practiced your newly developed Python skills, let's print the following with `for` loops and `print` statements.

```
1
2
3
4
```

It's simple enough, but it helps you get a sense for what you can do in Jupyter Notebook. Since you already ran the code, a new cell should now be active. In the cell, type the following:

```
for num in range(1, 5):
    print(num)
```

Note that the range goes until `5`, even though we only want numbers only until `4`. This is because, as you may have guessed, the `range(:)` function in Python is exclusive, so by typing `range(1, 5)`, you're telling the `for` loop to go in the interval 1 ≤ `num` < 5, which ultimately gets us the outcome we want:



```
In [1]: print("hello")

hello

In [5]: for num in range(1, 5):
            print(num)

1
2
3
4

In [ ]:
```

Figure 3-21: Printing a List of Numbers

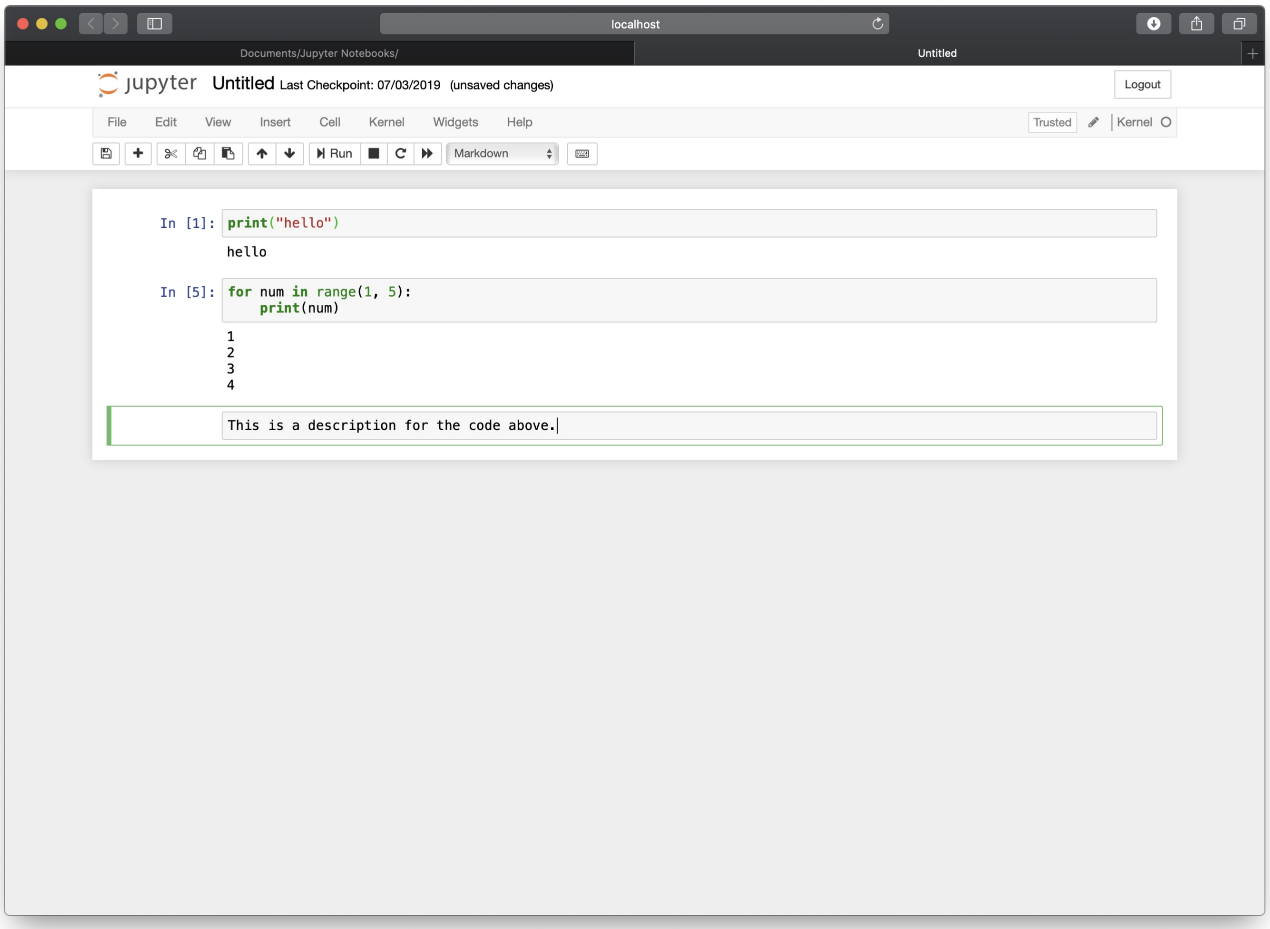
Now, run your code, and you'll see that you've achieved the end-result. Your Jupyter Notebook now has the numbers `1` through `4` printed, each on their own line.

Writing Descriptive Text

Now, our little coding project was fun, but when someone reading the code, they may not understand why the code is there, especially if they're very new to Python. Or, you may have intended to make some changes to the code, and you sent your Jupyter Notebook to a friend. For these scenarios, it's useful to use text in between your code snippets.

With a new cell is selected, change the **Cell Type**, which you can do using the unlabeled dropdown which defaults to read **Code**. You'll need to change this to **Markdown**. You'll see the `In` prompt on the left of the cell disappear, and you can now type a message.

When you do, your screen should look something like this:



```
In [1]: print("hello")
        hello

In [5]: for num in range(1, 5):
            print(num)
        1
        2
        3
        4

        This is a description for the code above.
```

Figure 3-22: Adding a Description Cell

Finally, run your code using the **Run** button, and your text will render into markdown. Don't worry if you don't know what that is yet. We'll be covering it at the very end of this chapter.

Data Visualization

Another reason to use Jupyter Notebook, as you learned, is its ability to visualize your code outputs, and interact with your data (in the data science sense). There are also third-party libraries which can help you produce tables, graphs, and models to see your data at a different level, and, more importantly, help others understand what you're trying to do.

Creating a Graph

The most common tool for data visualization in Python is `matplotlib`. It's an open-source tool, and it can be used for the purposes detailed before. Before starting, make sure your cursor is on a new cell, where it likely already is.

Configuring Inline Graphs

By default, Jupyter Notebook will show you a text representation of your graph, so you need to explicitly tell it to show a graph of your —well — graph. Doing this is very simple; all you need to do is to enter the following lines of code in your new cell:

```
%matplotlib inline
```

Great! You won't run into the issue of having your graphs be invisible anymore. If you're curious, you can find out about the other `%` operators you can use with Jupyter Notebook to display various other things.

Importing Frameworks

First, you'll need to import the `matplotlib` framework, which'll allow you to use the plotting framework in your program. You'll also need to import `numpy`, which handles basic mathematical functions in Python. To do this, continue by adding the following lines of code:

```
import numpy
import matplotlib.pyplot as plot
```

The first one, `numpy`, doesn't have an `as` modifier, which means that it will be referred to as `numpy` throughout the program. However, the `matplotlib` line does, since we don't want to type out `matplotlib.pyplot` each time we want to use it. Instead, by specifying `as plot`, we're telling Python that we'll refer to it as `plot` for short.

Creating the Cosine Function

For our example here, let's create the cosine function. We'll define the `x` and `y` variables separately and then plot them in the next step. This is that step which required us to import the `numpy` framework. Type the following:

```
x = numpy.arange(0, 5 * numpy.pi, 0.001)
y = numpy.cos(x)
```

For the `x` variable, we use the `arange(:)` method to evenly distribute values from 0 to $5\pi$ for our x-coordinates. In addition, the method takes a third parameter which defines the resolution — or frequency — of the distributed values.

For the `y` variable, we simply pass in the `x` values into the graph of cosine, which is conveniently provided to us by the `numpy` framework. This generates the `y` values for each of the `x` values generated earlier.

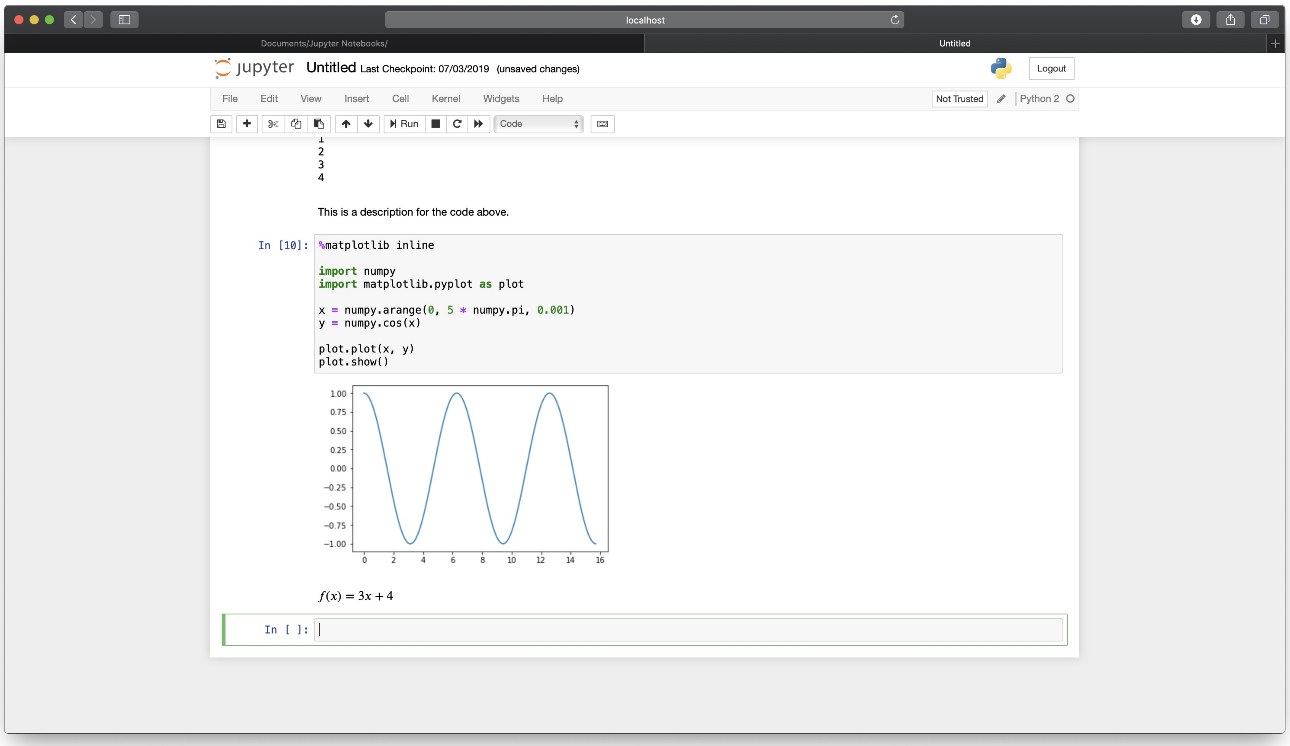
Displaying the Graph

Finally, you're ready to display your graph. And, for this step, we're back using the `matplotlib` framework by passing in the `x` and `y` values we generated in the previous step. Type in the following code to get your graph:

```
plot.plot(x, y)
plot.show()
```

As self explanatory as it is, the `plot(:)` function takes in two parameters: the `x` and the `y` coordinates. It then plots them, as the name suggested. The `show()` method takes no parameters and displays the graph below your cell.

Now, when you run your code, you should see something like this:


localhost
Documents/Jupyter Notebooks/
Untitled
jupyter Untitled Last Checkpoint: 07/03/2019 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 2
Run
Code

```
1
2
3
4
```

This is a description for the code above.

```
In [10]: %matplotlib inline

         import numpy
         import matplotlib.pyplot as plot

         x = numpy.arange(0, 5 * numpy.pi, 0.001)
         y = numpy.cos(x)

         plot.plot(x, y)
         plot.show()
```

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12
14
16

$f(x) = 3x + 4$

```
In [ ]:
```

Figure 3-23: Displaying a Cosine Graph

Later on, you can also add things like chart labels, axis labels, keys, and even use different types of graphs, such as histograms, pie charts, and more. However, the basic concepts are the same for nearly all graphs.

Markdown and LaTeX

We'll conclude this chapter by learning Markdown and LaTeX. Chances are, if you've done any sort of online documentation, especially `README` files on GitHub, you're likely familiar with Markdown. If not, that's okay, we'll review it here anyways. LaTeX, in case you're not familiar with, is used to represent mathematical expressions. A fun fact: this entire book was written in 100% markdown and LaTeX (for the math/numerical bits). While it's crazy to think about, it demonstrates the true power of these tools.

Markdown

Indirectly, you've already used the Markdown feature of Jupyter Notebook. If you recall, you were asked to switch your cell to **Markdown** when you were entering a comment. The same comment, and its surrounding text, can be styled using Markdown.

Markdown allows you to use a series of shorthand symbols to convey italics, bold, headings, and other stylized content in a quick way, without needing to click on buttons in a WYSIWYG (what-you-see-is-what-you-get) editor.

To begin, change your new cell type to **Markdown**. It'll be a dropdown in the top menu which reads **Code** by default. You'll see the `In` prompt on the left of the cell disappear as before.

Headers

Headers are one of the most commonly used features of markdown, since they help you bring some structures to seemingly long-form writing. They're also the simplest to use, since they're just one or more hashtags in a line.

Type the following into your new cell:

```
# I am a title
```

This will render as blue text, and while it'll still retain the hashtag, it will appear larger than it did before. This is called a **Header 1** or a **Title**. This should be used to convey major changes in your Jupyter Notebook, perhaps to separate completely different pieces of code from each other in longer notebooks.

You can also test other header sizes, and see how they appear differently from the others, by doing the following:

```
# I am a title
## I am header 2
### I am header 3
#### I am header 4
```

This way, you'll be able to compare how each of the headers looks in relation to the others. These get progressively smaller, and at some point, it's easier to use bolded text than a header — just a whimsical quirk of Markdown!

Stylized Text

We've all written a Word document or an email in which we're using the common bold, italics, and underline features of our word processors. These common functionalities, while convenient in day-to-day use as keyboard shortcuts, lend themselves easier to quick symbols.

Similar to headings, you can define bolded text like this:

```
**I'm bold.**
```

And, to define italicized text, you can do this:

```
*I'm italicized.*
```

And, to use them both, you can do so by combining them:

```
***I'm both.***
```

Unfortunately, many Markdown editors don't support underlined text, unless it's being used in the context of a link. However, you're unlikely to need underlined text in a Jupyter Notebook — since the features you're being given are already likely an overkill.

Lists

Lists in markdown aren't anything special. You can declare a numbered list by using just numbers, like this:

```
1. Item one.
2. Item two.
3. Item three.
```

And, unordered lists are even easier:

```
* Item.
* Item.
* Item.
```

And, that's it! Now, it's hopefully clearer why developers choose to use Markdown instead of anything else: it's easier than HTML, and it's definitely easier than using the buttons on a menu bar.

LaTeX

Though technically unrelated, LaTeX (pronounced "lay-tech") is like Markdown, but for math. This tool is slightly harder to use, but is very powerful for typing out complicated mathematical expression and special symbols such as $\pi$ and unique structures such as matrices.

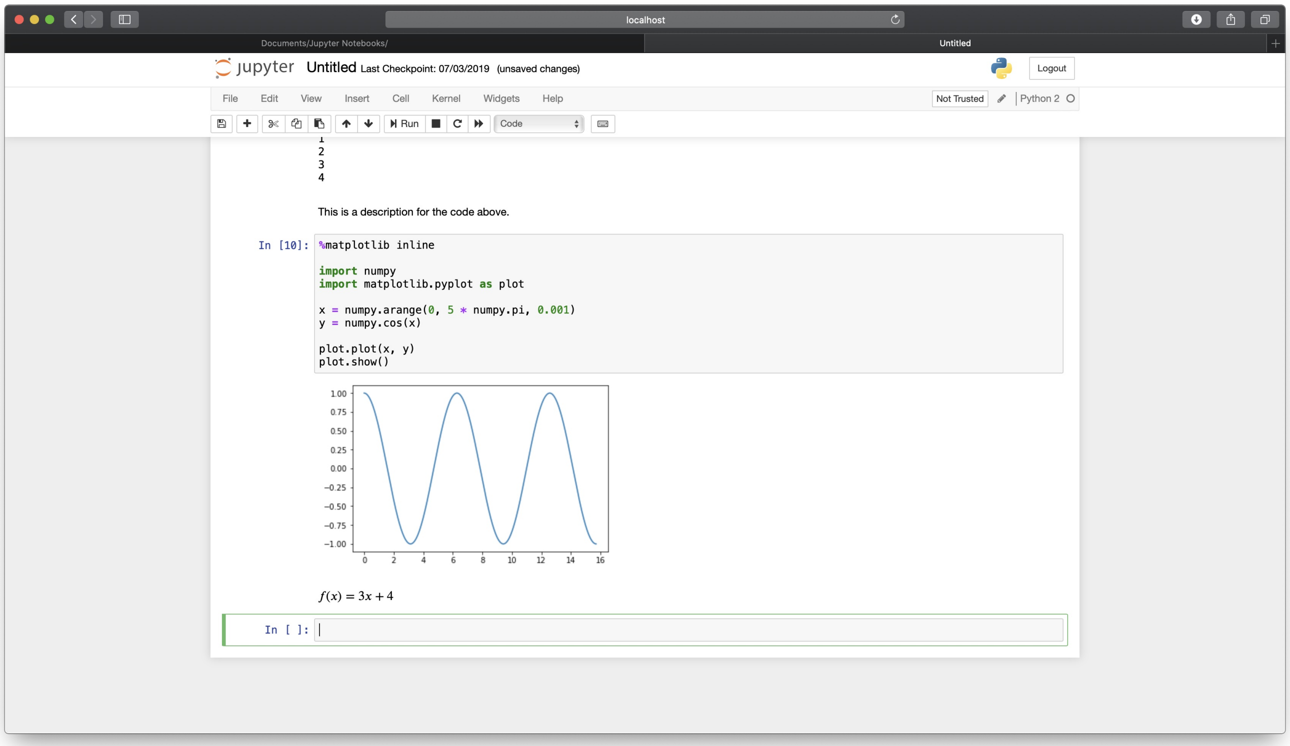
Simple Expressions

The most common use of LaTeX is for inline formulas, meaning those which appear in between text. These don't have their own dedicated lines on your Jupyter notebook, and they appear interspersed between your text (kind of like bolding or italicizing a particular portion).

To enter the equation $f(x) = 3x + 4$ inline, type the following into your cell:

```
$f(x) = 3x + 4$
```

That was simple. All you need to do for inline LaTeX is surround your latex command in dollar (`$`) signs. However, this equation doesn't have any special symbols. It'll render like this:


localhost
Documents/Jupyter Notebooks/
Untitled
jupyter Untitled Last Checkpoint: 07/03/2019 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 2
Run
Code

```
1
2
3
4
```

This is a description for the code above.

```
In [10]: %matplotlib inline

         import numpy
         import matplotlib.pyplot as plot

         x = numpy.arange(0, 5 * numpy.pi, 0.001)
         y = numpy.cos(x)

         plot.plot(x, y)
         plot.show()
```

$f(x) = 3x + 4$

```
In [ ]:
```

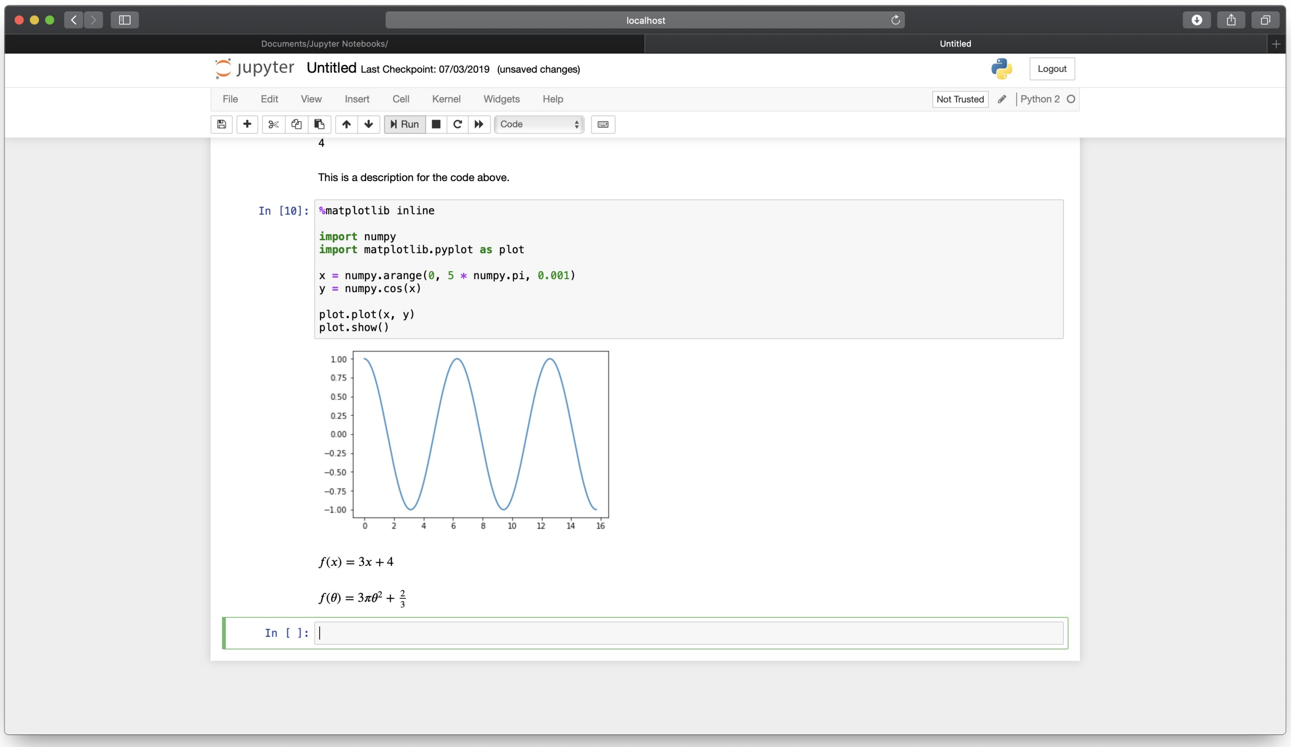
Figure 3-24: Rendering Simple Equations with LaTeX

Complicated Expressions

Let's try $f(\theta) = 3\pi\theta^2 + \frac{2}{3}$. More complicated, right? Imagine typing that out manually. It would take forever, and it would look terrible. Luckily, you can do this with ease in LaTeX, like this:

```
$f(\theta) = 3\pi\theta^2 + \frac{2}{3}$
```

With a backslash, you can enter various commands, many of which are special characters and math symbols. Our variable here is theta (θ), and `\theta` gives you just that. Our coefficient is three pi (3π), and similar to theta, pi can be typed out as `\pi`. To get an exponent, we just use the carrot symbol `^` and then type out the exponent. Lastly, the `\frac{numerator}{denominator}` function gives us a nice looking fraction like $\frac{2}{3}$ instead of an ugly one like 2/3. When you run it, it'll look like this:



localhost
Documents/Jupyter Notebooks/
Untitled
jupyter Untitled Last Checkpoint: 07/03/2019 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 2
Run
Code

```
4

This is a description for the code above.

In [10]: %matplotlib inline

         import numpy
         import matplotlib.pyplot as plot

         x = numpy.arange(0, 5 * numpy.pi, 0.001)
         y = numpy.cos(x)

         plot.plot(x, y)
         plot.show()
```

$f(x) = 3x + 4$

$f(\theta) = 3\pi\theta^2 + \frac{2}{3}$

```
In [ ]:
```

Figure 3-25: Rendering Complex Equations with LaTeX

As you can see, it renders nicely among the other text, as if it were part of the sentence — that's why it's called "inline." You can, of course, do other things with LaTeX, but you're likely not building projects that are too mathematically involved just yet. However, if you're interested, this resource is a great cheat sheet for you if you're interested in learning more about LaTeX.

Conclusion

In this chapter, you learned Python, a general-purpose language, which could be considered similar to Swift in some aspects. I went step-by-step and explained the syntax of Python. If you choose to continue pursuing machine learning or data science, Python will prove to be an invaluable skill in your toolkit as you advance in your career.

You also learned about Jupyter Notebook, a boon for everyone working in the data science field. Towards the beginning, you learned what they are, how to use them, and where they fit into your day-to-day workflow in the world of machine learning. Then, you created one and saw how it can come in handy for machine learning tasks.

Chapter 4
Training Your Own Image Classifier

In the previous chapter, you learned about Jupyter Notebooks, Python, and how you can use those tools in the real world. You also learned about the basics of Python, focusing on how you can relate it to the Swift you have already known. You also learned how to install Jupyter Notebooks using Anaconda and use them with markdown as well as LaTeX. It's now time to put your skills to the test and start training models!

In this chapter, you'll learn about image classification models, what they're useful for, and how you can train your own image classification models for your apps. With Apple's tool, Create ML, we can train these models and change certain parameters to improve the accuracy and research what works best.

By the end, you'll have a working image classification model, which you can drag into your image classification app from Chapter 2 to use right away. You'll also learn how to optimize your models, and ensure that they're fully accurate before you begin shipping them with your apps.

4-1 Preparing Training Data

As with any supervised learning model, you need to specify the "true value" of the inputs to train a machine learning model. An image classification model is no different, and this section focuses on finding, organizing, and labeling these images so that they can be used while training your machine learning model.

By the end of this section, you'll have a folder which contains a set of labeled images to be used in the following section as training data. While we'll be using these images to train Core ML models, you can adjust them to work with other types of models in the future as well.

Finding the Images

To create your own image classifier, the very first step is to prepare your own images. For most commercial apps, you'd capture and label your own images to ensure optimal performance. But this takes *a lot* of time and research to get right. At a large scale, companies like Apple, Google, and Microsoft focus a large portion of their resources towards research of this type and dedicate entire teams to it.

However, you're just learning right now — there's no need to worry about creating your own images. In fact, for most of the apps you make, you'd benefit from using the images that others have already made, provided that you know where to find the images. Remember, you'd better make sure they're labeled for reuse before using anything online, so that you're not limited if you choose to use them commercially in the future.

There are several datasets you can find online to use the work that others have already done for you — there is no point to reinvent the wheel, right?

MNIST (http://yann.lecun.com/exdb/mnist/)

The MNIST dataset, as the name (doesn't) suggest, is an image dataset of 60,000 images and 10,000 additional images for testing (you'll learn about this later). These images contain a wide range of digits drawn by tens of thousands of people. You can use the MNIST database to train your own handwriting classification model and take advantage of the professionally captured images.

SVHN (http://ufldl.stanford.edu/housenumbers/)

SVHN stands for Stanford's Street View House Numbers dataset. Stanford University has gone into over 600,000 house numbers using Google Maps' Street View feature for a real-world application of digit recognition. It's similar to the MNIST database, but designed for a more complex application.

CIFAR-10/100 (http://www.cs.utoronto.ca/%7Ekriz/cifar.html)

If you're interested in a more traditional dataset, you can also go for the CIFAR-10 or CIFAR-100 datasets. They are labeled images from a parent dataset called Tiny Images Dataset, which contains 8,000,000 images. As it's far too many for casual use, these two datasets were created. CIFAR-10 contains 10 classes, and CIFAR-100 contains 100 classes, which means they can support distinguishing between 10 and 100 images, respectively.

COCO (http://cocodataset.org/#home)

Similar to CIFAR-10 and CIFAR-100, the COCO database is a collection of several object categories along with their respective images. COCO stands for Common Objects in Context, which makes sense for an object/image classification dataset. You can use these sorts of models for a wide range of applications, since they cover many day-to-day objects in great detail.

SDD (http://vision.stanford.edu/aditya86/ImageNetDogs/)

The Stanford Dogs Dataset comes from ImageNet, and all-encompassing dataset of various objects. This SDD dataset has 120 different dog breeds, with 20,580 images. That's a lot of images of dogs! In the rest of the chapter, we'll be using a carefully selected subset of this data to simplify our machine learning routine. You can also use the full version when you're ready and/or able.

These models are all robust enough to serve the purpose of a production-level app for the App Store, however, they may have too many complications when you're just starting out — like in this chapter of this book. To ease some of these concerns, I've put together a subset of the Stanford Dogs dataset for you to use in your starter app. You can download it here if you'd like to use this one. It's the one I'll be using for the rest of the chapter.

Image Preparation Tips

For professionally-taken dataset of images, there is nothing to worry about. But if you're using your own data, you'll need to ensure that your images are high-quality and will help the model instead of confusing it more.

Simulated Conditions

When you're taking pictures, make sure that they're in the same scenarios as which your users would be taking pictures. You need to make sure that your model has been thoroughly trained using a certain type of image (the type your user will provide) because this is important to your model's accuracy. However, this might be hard to understand conceptually, so let's look at an example.

Pretend you're building a hot-dog classifier for hot-dogs out in nature — like on a picnic. Avoid using images of hot-dogs indoors, and use more outdoor-taken pictures of them. This will help your classifier hone in on one niche application instead of trying to be too good at everything.

Skip Confusing Images

While it may seem like a good idea to include "edge cases" in your image training dataset, it's likely going to reduce the accuracy of your model, since images like these will introduce characteristics of other classes in the weights of unrelated classes. Our bottom line is: if the image confuses you, don't use it in training. Conversely, though, you should use this image to test your model, just to make sure it's performing at its very best (but don't take the results to heart on these kinds of images).

Balance Classes

One very important thing to keep in mind while training these models is to ensure the same number of images for each label during training. While this may not seem important, it may cause your model to be biased towards the "heavily weighted" pieces of your model. In other words, if you have 2500 images for microphone images, then use 2500 images for headsets. This way, you won't make your model more accurate for some types of images and balance out the weight.

Getting the Training Data Ready

After spending enough time searching — or, if you have the courage to capture your own images — for your dataset, it's time to prepare these images for training. The following are the things you need to keep in mind before plugging these images straight into Create ML.

80-20 Rule

In machine learning, we have a (sometimes unstated) rule, which I like to call the "80-20 rule". From the name, you can infer the definition. It means that 80% of the data should be used for training the actual model, and 20% should be saved for testing.

The keyword here is "should," since this isn't a hard-and-fast rule. Instead, it is more of a guideline to help you stay on track. Nothing *bad* will happen if you don't save 20% of your data for training. However, it will help you get a better sense for how your model is performing if you do so. One example in which you might be better off saving less than 20% for testing is when your dataset is extremely small. In cases like this, your training dataset may take up too large of a chunk of the valuable data which could have been used for training. Therefore, it's always best to collect more data than less.

When you do this, one thing you'll want to keep in mind is to ensure that none of the testing data images are also in the training data, because this will cause you to get inaccurate results when going back to test your Core ML model.

The Folder Name is the Label

Now that you have your images, it's important to structure your files properly for Create ML, Turi Create, and other tools to recognize your images and their labels. Labelling your images is easy, since it only requires you to move your images into folders with the appropriate names. For example, if you had 700 images of apples, you'd put them in a folder called "apple" or a similar name.

The model I provided, i.e. the subset of the Stanford Dogs dataset, is already in the recommended structure to make it easier for you. All you need to do is download it from https://github.com/vhanagwal/dog-breed-dataset.

4-2 Training with Turi Create

In the previous section, you learned about how to find images (or capture your own images) and prepare them for training your image classification model. You also learned what makes good images and which ones to avoid, as well as several great resources for you to use in your future apps. Lastly, I provided you a subset of data for you to utilize, instead of having to jump through the hoops in downloading, setting up, and extracting the images in the linked datasets.

In a moment, you'll use the images — that you either captured or downloaded — to train your own image classification model which can be exported and used in Core ML format. This model can then be dragged directly into your apps, similar to the image classification app we built in Chapter 2.

We will explore two ways to train your own image classifier

1. Using Turi Create
2. Using Create ML

Using Turi Create

The first method is using Turi Create, a Python-based library owned and managed by Apple. By default, your Mac won't come with Turi Create installed, so you'll need to install it before using it to train your image classification model. Not to worry, though, since the installation is pretty straightforward, especially if you've used the command line in the past.

System Requirements

Turi Create has some system requirements, which are pretty standard for almost any software you install — and here they are, if there's any chance your computer doesn't support Turi Create, or you just enjoy reading this kind of stuff.

Turi Create supports:

- macOS 10.12+
- Linux (with glibc 2.12+)
- Windows 10 (via WSL)

Turi Create requires:

- Python 2.7, 3.5, 3.6
- x86_64 architecture
- At least 4 GB of RAM

All this means is that if your Mac is *somewhat* new, you should be able to use Turi Create. If, by any chance, your computer cannot use Turi Create, don't worry, you'll still be able to follow along with other sections in the book, since they involve other technologies which is more likely to work with outdated computers.

Command Line Installation

Once you have system requirements out of the way, you'll need to ensure that Python is installed properly with the correct version, so that you can follow along with the commands in the rest of the section.

Open Terminal

Start by opening **Terminal**, which you'll find either via Launchpad or Spotlight search. You can also find it at the path **Applications > Utilities > Terminal** in your file system.

Checking Python Version

For this section — and throughout the book — we'll be using Python 3.8, since it's the most stable version of Python at the time of writing. Your computer should already have Python installed, so let's check which version you have. Enter the following into your terminal:

```
python --version
```

If your output says Python 3.x, you're good to go! If not, you'll need to install Python 3 from their website. This is a fairly straightforward GUI-based installation, so we won't cover it here.

Installing pip

To install Turi Create, we'll be using *pip* (https://pypi.org/project/pip/). In case you aren't familiar, *pip* is a package manager for Python, which means it helps developers easily install and manage the libraries for your Python-based projects. It's pretty useful and once you've installed it, your life as a developer will become much easier. Luckily, *pip* itself is very easy to install. Assuming you have Python installed, enter the following into Terminal to start the *pip* installation:

```
sudo easy_install pip
```

Now, enter your administrator password (the one you use to log in to your computer). When you're done, hit the **Return** key to begin the installation. After a few minutes, you'll see your prompt again, and then you're done.

Installing Turi Create

After *pip* is installed, Turi Create and other libraries are painless to install. They each require only one line. To install Turi Create, type the following into the terminal:

```
pip install turicreate
```

Your installation should begin, and you'll see the steps printing to the console. In a few minutes, you'll see the prompt again, and you'll know that Turi Create was installed successfully.

Training with Turi Create

Training an image classification model, typically, can be done without writing a single line of code. But this book aims to challenge you — so, we'll start off with the more "involved" method of training an image classifier. This method involves the use of a Python-based library called Turi Create (already acquired by Apple). Don't worry even if you have no experience with it, since I'll go step-by-step through the process.

Setting up the Python File

Now that you have Turi Create installed, you're ready to train your image classification model with it. Take a look at the images you downloaded earlier. The file structure is as follows:

- `image-classification`
    - `training`
        - `cocker-spaniel`
            - `IMG1XX.png`
            - `IMG9XX.png`
        - `golden-retriever`
            - `IMG1XX.png`
            - `IMG9XX.png`

First, type `cd` and drag your `image-classification` folder into your terminal window. You'll get a path and command which looks like this:

```
cd /Users/vardhanagrawal/Desktop/image-classification
```

When you hit the **Return** key, you would have entered the `image-classification` folder via your Terminal. You can now create a file here through the command line. To do this, enter the following command:

```
touch classifier.py
```

When you navigate to the `image-classification` folder from Finder now, you'll find that a new file has appeared named `classifier.py`. Open it with Xcode or your favorite editor.

Writing the Python Script

Now we will write this Python script to load the training data and create the model. At the very beginning of the script, you need to import the necessary frameworks to train your image classification model. Since you've already installed Turi Create onto your system, you can now use it in any projects. All you need to do is import it as follows in the Python file:

```
import turicreate as turi
```

You could omit the `as turi` specifier. However, if you do so, whenever you needed to refer to it, you would have to type out the entire word `turicreate` each time. With the specifier, Python knows that you mean the `turicreate` package when you write `turi` in your program.

To gain access to your file system, you'll need to import the `os` framework, which stands for "operating system." You can use a similar method to import it:

```
import os
```

Since the name is fairly short this time, you don't need to rename it; typing `os` is only two letters and isn't much effort. But if you're feeling extra lazy, you may even rename it to a single character.

Loading Images

Now that your file is all set up, you need to load the images into a format which Python can read, understand, and process. The reason we spent so much time in the previous section doing this was to save time and effort on this step.

First, we need to load the directory of the training data images and store it as a variable. You have already learned how to declare a variable in Python in the previous chapter, and you can apply that knowledge here. Turi's `load_image(:)` method will help us do just that. Type the following after the import statements in the `classifier.py` file:

```
data = turi.image_analysis.load_images('training', with_path=True)
```

Since the `classifier.py` file is in the same folder as the `training` folder, we can directly reference the name of the folder in the first argument of the `load_images` method.

Mapping Folder Names to Classes

If you remember, we named each of the folders a certain dog breed and put all of the images of that breed into its respective folder. This was to be able to use those folder names as labels (known as classes) to identify the objects given to the model. To do this, use the following function:

```
data['label'] = data['path'].apply(lambda path: os.path.basename(os.path.dirname(p
ath)))
```

In case you haven't observed, the `data` variable is a dictionary-like type (which has keys and values), and we're mapping each of the subfolder names to the `label` key. The `basename(:)` method extracts the name of the folder from the path.

Saving Data in an SFrame

An SFrame is a data structure in Turi Create, which can hold large amounts of data. We have a dataset, which we can save for later use. Let's save our data from the previous steps in an SFrame:

```
data.save('image-classifier.sframe')
```

This `save(:)` method, as the name suggests, saves the data with the specified parameter name. You can access the `image-classifier.sframe` later in the program to use the data for training.

Processing Images

By this point, your images are parsed into a format which Turi Create can handle. You now have an `SFrame` saved as a representation of the images we input in the previous step.

Loading SFrame

To use the `SFrame`, you'll need to load it first. While it doesn't make much sense in a small project such as this one, saving and then loading the data later on can be a boon for many larger, more enterprise-level projects. Loading the `SFrame` is just as easy as saving it:

```
data = turi.SFrame('image-classifier.sframe')
```

Here, we've used the name to identify the SFrame and then pull it out to save to the `data` variable. The current value of `data` is now the SFrame we saved earlier. Again, this step is redundant in a small project, but it's worth getting a sense for how this may be used in other projects later.

Splitting Data

As you learned earlier in the book, you should retain 80% of your data for training and put aside 20% for testing the trained model. This way, you can instantly validate your model within Turi Create, without needing to create an app to test it out. In your Python file, split your data like this:

```
testing, training = data.random_split(0.8)
```

In case it isn't clear, the `random_split(:)` method takes in a parameter and randomly splits data into two "buckets." We're saving our testing and training data in two variables, called `testing` and `training`, respectively.

Training, Testing and Exporting

Now, your data is ready to use for training the model. After training, you can use Turi Create's built-in methods to test the model. In the end, you can export it as a Core ML model to use in your iOS, macOS, watchOS, and tvOS apps.

Training the Model

So, after all of this work, we're ready to train and test the model with the data we set aside from the previous step. With the following Python code, you can train your model:

```
classifier = turi.image_classifier.create(training, target='label', model='resnet-
50')
```

In this case, we're creating a variable called `classifier` and storing the return value of `image_classifier(:)`. This model is being trained with the ResNet 50 architecture, which is typically used to differentiate between household objects.

Testing the Model

Once you trained a model, we need to test it to see how it performs. Type the following to test your model against the test data that was randomly selected from the training data:

```
result = classifier.evaluate(testing)
```

The `evaluate(:)` method takes the `testing` data and plugs each image back into the model which you created in the previous line of code. This automatically checks whether the model produced the expected result and how often. Then, type the following:

```
print(result['accuracy'])
```

After storing the results in the `result` variable, we print out the value associated with the `accuracy` key. You'll be able to see how your model performed.

Saving and Exporting the Model

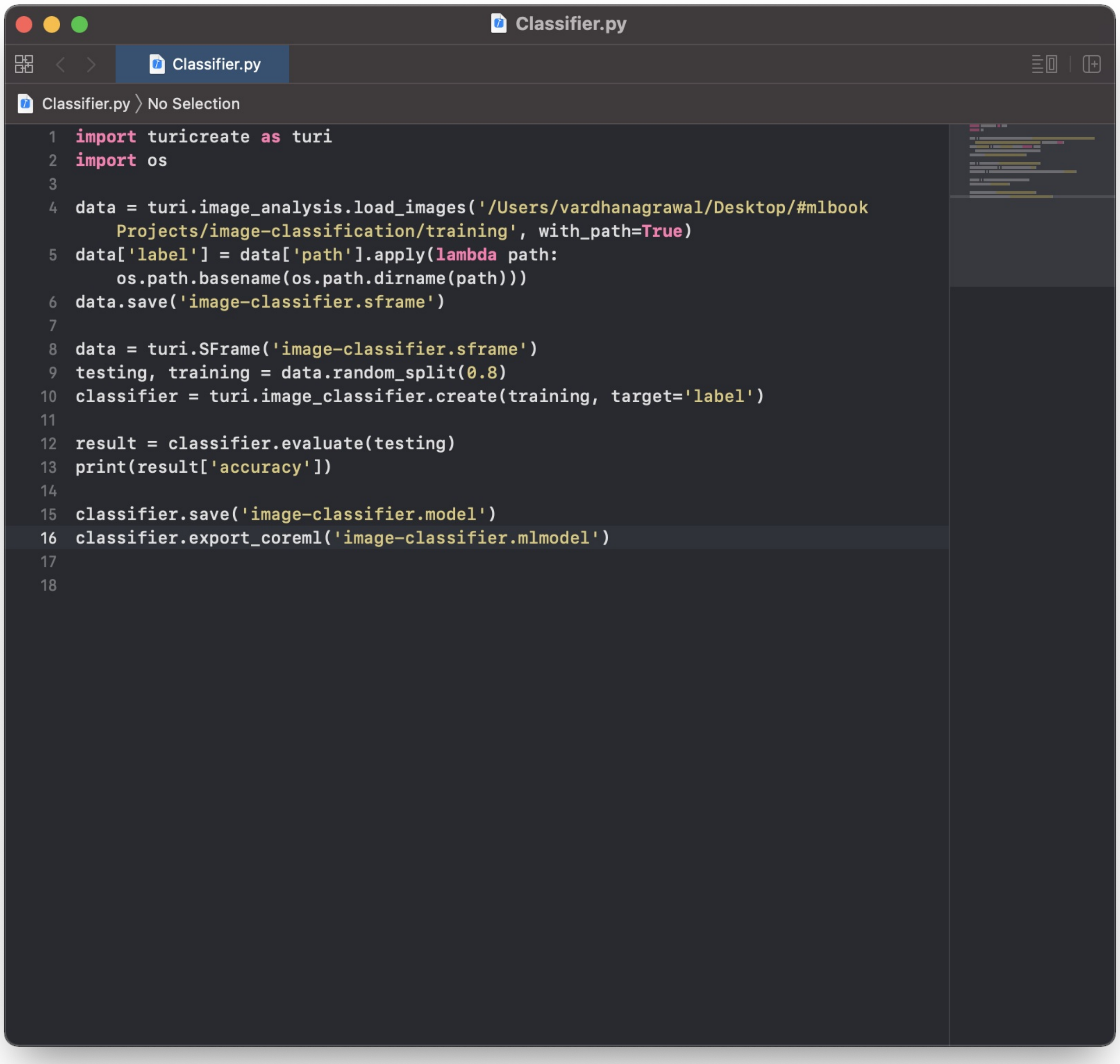
After you're satisfied with your model's performance, you're ready to save it and export it as a Core ML model which can plug straight into your iOS app from Chapter 2. You can save your model like this:

```
classifier.save('image-classifier.model')
```

Finally, you can export the model by typing the following:

```
classifier.export_coreml('image-classifier.mlmodel')
```

This will allow your model to appear in the appropriate format inside of your main `image-classification` folder — the same folder as your `classifier.py` file is located — named `image-classifier.mlmodel` or whatever else you choose to name it. By this point, your entire file should look like this:



```
import turicreate as turi
import os

data = turi.image_analysis.load_images('/Users/vardhanagrawal/Desktop/#mlbook
    Projects/image-classification/training', with_path=True)
data['label'] = data['path'].apply(lambda path:
    os.path.basename(os.path.dirname(path)))
data.save('image-classifier.sframe')

data = turi.SFrame('image-classifier.sframe')
testing, training = data.random_split(0.8)
classifier = turi.image_classifier.create(training, target='label')

result = classifier.evaluate(testing)
print(result['accuracy'])

classifier.save('image-classifier.model')
classifier.export_coreml('image-classifier.mlmodel')
```

Figure 4-1: Python File

Running the Code

Unlike Xcode with Swift, which you're used to, Python files don't compile automatically in the background. Therefore, if you've made any mistakes, they won't be caught by the compiler, and instead, will be found when you run the code.

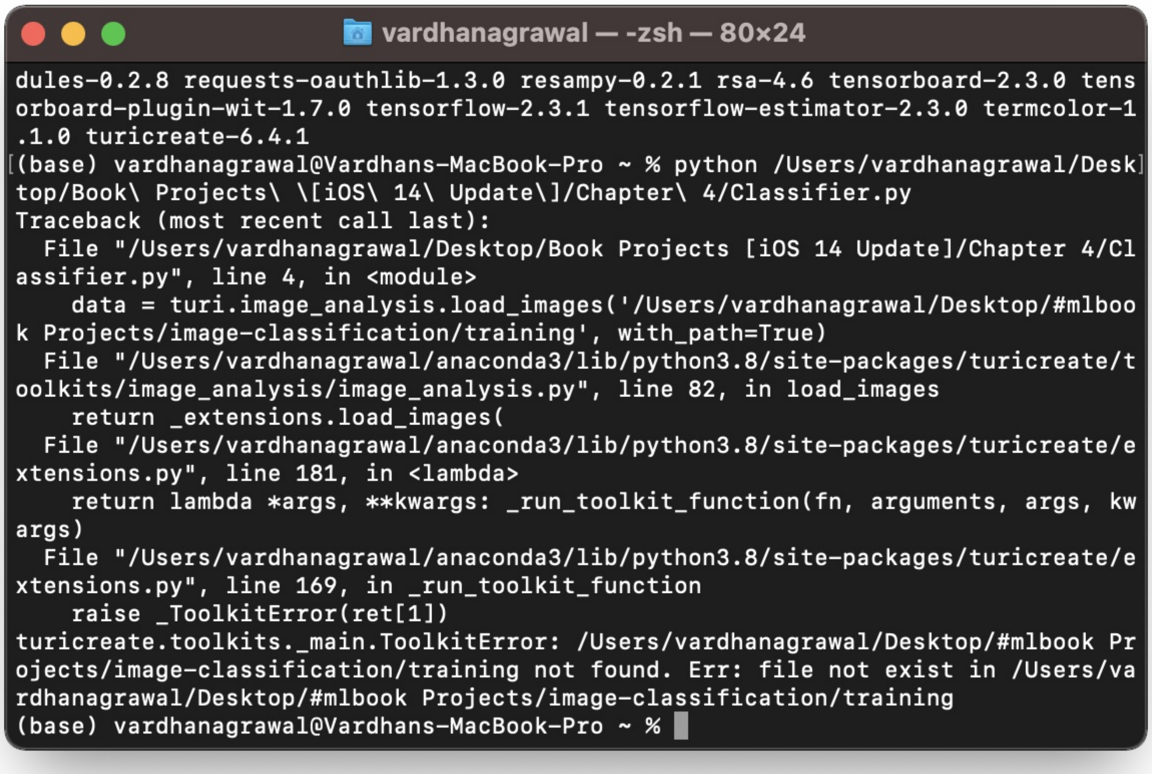
Go back to your command line. Type `cd` and drag your `image-classification` folder into your terminal window. You'll get a path and command which looks like this:

```
cd /Users/vardhanagrawal/Desktop/image-classification
```

When you hit the **Return** key, you would have entered the `image-classification` folder via your Terminal. This is the same step as before, which will render you access to your `classifier.py` file. From here, type:

```
python classifier.py
```

This will run your code and perform all of the expected tasks, all at once. Your Terminal window will appear like this:



```
dules-0.2.8 requests-oauthlib-1.3.0 resampy-0.2.1 rsa-4.6 tensorboard-2.3.0 tens
orboard-plugin-wit-1.7.0 tensorflow-2.3.1 tensorflow-estimator-2.3.0 termcolor-1
.1.0 turicreate-6.4.1
(base) vardhanagrawal@Vardhans-MacBook-Pro ~ % python /Users/vardhanagrawal/Desk
top/Book\ Projects\ \[iOS\ 14\ Update\]/Chapter\ 4/Classifier.py
Traceback (most recent call last):
  File "/Users/vardhanagrawal/Desktop/Book Projects [iOS 14 Update]/Chapter 4/Cl
assifier.py", line 4, in <module>
    data = turi.image_analysis.load_images('/Users/vardhanagrawal/Desktop/#mlboo
k Projects/image-classification/training', with_path=True)
  File "/Users/vardhanagrawal/anaconda3/lib/python3.8/site-packages/turicreate/t
oolkits/image_analysis/image_analysis.py", line 82, in load_images
    return _extensions.load_images(
  File "/Users/vardhanagrawal/anaconda3/lib/python3.8/site-packages/turicreate/e
xtensions.py", line 181, in <lambda>
    return lambda *args, **kwargs: _run_toolkit_function(fn, arguments, args, kw
args)
  File "/Users/vardhanagrawal/anaconda3/lib/python3.8/site-packages/turicreate/e
xtensions.py", line 169, in _run_toolkit_function
    raise _ToolkitError(ret[1])
turicreate.toolkits._main.ToolkitError: /Users/vardhanagrawal/Desktop/#mlbook Pr
ojects/image-classification/training not found. Err: file not exist in /Users/va
rdhanagrawal/Desktop/#mlbook Projects/image-classification/training
(base) vardhanagrawal@Vardhans-MacBook-Pro ~ %
```

Figure 4-2: Running Python Script

The output in the Terminal serves a few purposes. First, it informs you if your code has any errors and references them by line number. Next, it shows you the progress of your classifier. Last, it shows you the results of the tests it's performed using the last 20% of your data, which you saved specifically for testing.

Files and Folders

After a couple minutes of training (depending on your machine, for older machines, it may take over 10 minutes), you'll see multiple files appeared in the `image-classification` folder including:

- `image-classifier.sframe`

- `image-classifier.model`
- `image-classifier.mlmodel`

What we concern is the `.mlmodel` file, since that's the Core ML model file for the iOS app. There's nothing left to do now! Your Core ML model is ready to be used in your app; you can drag it in, in place of any other Core ML model, and replace any code references to it with the name `image-classifier`.

4-3 Delving into Create ML

In the previous section, you created your own image classifier using a Python-based tool called Turi Create. You went through the process of installing *pip* and using it to ultimately install Turi Create using the command line. You also learned some basic Terminal commands to traverse your file system on your Mac. Finally, you ended up with a Core ML model which you can use in your existing apps.

In this section, you'll learn how to create the same image classification model — but this time, with significantly less effort. The reason you did the more complex version first was to challenge yourself and to learn an advanced tool when you develop more advanced machine learning needs throughout your career. However, if you're doing some quick research, Create ML is the way to go!

Introducing Create ML

You just finished creating a Core ML model using Turi Create which allows you to create a variety of robust machine learning models. Admittedly, though, the model we just created wasn't too complex, and there's an easier way to create it — without writing a single line of code!

Create ML comes bundled with Xcode, and when you first open Xcode 12, you'll notice that it installs additional packages. These packages include Create ML, which is a GUI-based machine learning model training software.

To open Create ML, you can search it using Spotlight (press **Command + Space**). The icon is blue and circular with the letters "ML" on it. Alternatively, you can open Xcode. In the menu bar, choose Xcode > Open Developer Tool > Create ML to launch the tool.

Once launched, you'll see a Finder window in which you can select existing projects or create a new document. Click the **New Document** button in the bottom-left corner to create a new project. You'll get a popup like this:

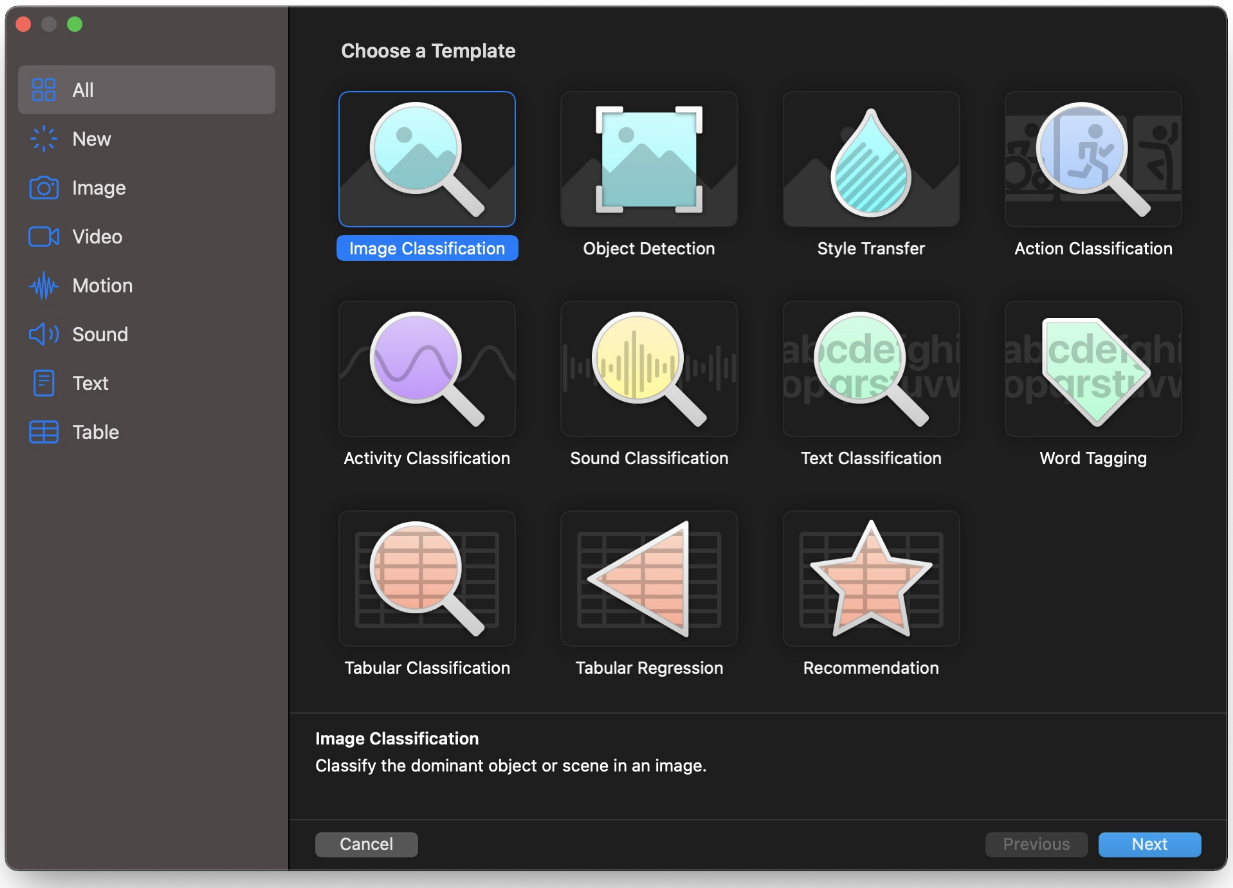


Image Classification
Classify the dominant object or scene in an image.

Figure 4-3: Launching Create ML

This shows you all of the different types of projects you can create with this software. Apple will continue to update it, and in the future, there may be even more applications than there are now. For example, in the beta version of this software, Apple added one feature at a time in each successive release of their beta software.

Now you're ready to start creating your first image classifier in Create ML.

What Models Can You Build?

As you learned, Create ML is an excellent tool which allows you to create moderately complex machine learning models using only a graphical user interface, as opposed to the seemingly discouraging code we wrote in the previous half of this section. Let's take the time to look through Create ML and learn a little bit more about it.

Model Categories

In the left pane of Create ML, you'll see a whole bunch of models, grouped by their categories. At the time of writing, these consist of the following:

- Image
- Sound
- Motion
- Text
- Table

More specifically, these model types have a couple of model templates inside, which allow you to do niche tasks and narrow in on a specific type of function you'd like your model to perform. For a dog breed classifier, for example, you'd use an image classification model.

Specific Models

In this book, we'll be exploring **Image**, **Sound**, and **Text** machine learning models in detail, since these are mainstream and necessary for you to call yourself a machine learning professional!

Image Models come in two basic types: image classification and image segmentation. The two are fairly similar but differ in one major aspect. Image classification models attempt to generalize the entire image as a whole, while image segmentation models — as the name suggests — segment the image into parts and identify the specific parts of the whole.

Sound Models are fairly new. They were launched fairly recently, and Core ML 3 is the first version of the framework to support sound classification on Apple's platforms. This opens up a whole new perspective for developers, since it allows people to create apps which haven't been possible in the past.

Text Models have been around for a while. In fact, they're one of the oldest types of models being at the core of Google when they were first developing their search platform. In Create ML, you can make either a text classifier or a word tagger. These are reminiscent of image models, since a text classifier deals with a chunk of text, while a word tagger deals with specific words.

Training with Create ML

Excellent job! Bootcamp is now over, and you're ready to start training the same image classification model using Create ML. This time, you won't need to write a single line of code, which I've already mentioned dozens of times. Are you ready to train?

Preparing your data

Your data structure for Create ML should be similar to that of Turi Create. However, you need neither of your top-level folder, nor a Python file to hold your code in it. Technically speaking, all you need are the following folders:

- `cocker_spaniel`
- `german_shepherd`
- `golden_retriever`
- `laborador_retriever`
- `standard_poodle`

These folders, as you know, contain your images along with their names, which represents their labels in the machine learning model. Similar to Turi Create, Create ML checks for these labels and uses them to automatically generate text labels when your model is used later in its life.

Starting a Project

You first need to start your project within Create ML. You'll need to choose a type, name your project, and choose a location for it. Then, you can add your training data as input for Create ML to do its magic.

Choose Model Type

The first model listed is the **Image Classification** model, which should be selected by default when you open the Create ML app. However, if it's not, select **Image** in the left window pane and **Image Classification** in the right window pane. Your window should look something like this:

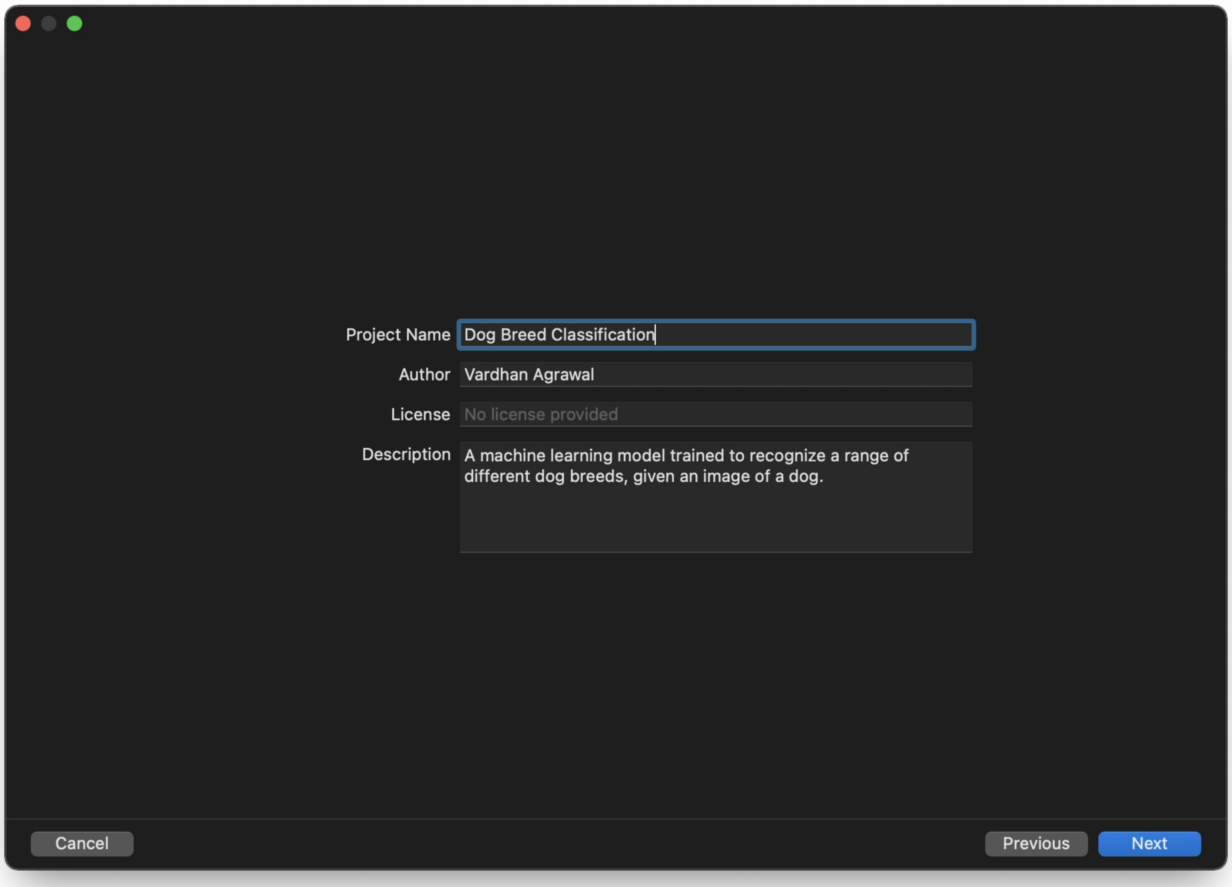

Project Name: Dog Breed Classification
Author: Vardhan Agrawal
License: No license provided
Description: A machine learning model trained to recognize a range of different dog breeds, given an image of a dog.

Figure 4-4: Naming your Create ML Project

Name and Save Project

You'll now be prompted to name your project. You can make this name as entertaining or as boring as you'd like. I'm calling mine plain old "image-classifier." It also gives you a chance to edit the metadata: author and description. These fields will appear when you finally export and use your Core ML model.

Input your Data

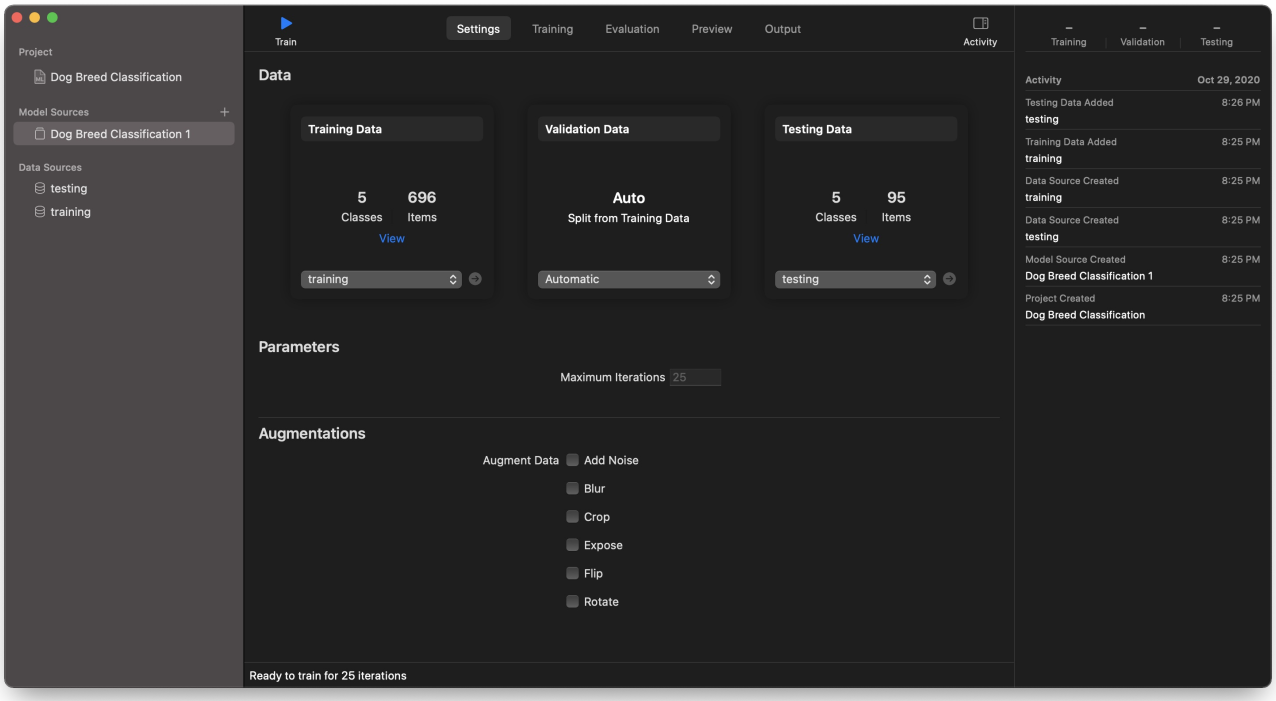
Inputting your data into Create ML is a piece of cake. Simply tap on the **Select Files...** dropdown, and then, select the `training` folder with labeled images inside them. Create ML will automatically infer your labels based on the folder names of the images. This folder can be downloaded from https://github.com/vhanagwal/dog-breed-dataset. You will do something similar with the `testing` folder in a bit.

- `dog-breed-dataset-master`
  - `testing`
    - `standard_poodle`
    - `labrador_retriever`
    - `golden_retriever`
    - `german_shepherd`
    - `cocker_spaniel`
  - `training`
    - `standard_poodle`
    - `labrador_retriever`
    - `golden_retriever`
    - `german_shepherd`
    - `cocker_spaniel`

As discussed in **4-1 Finding and Preparing Data**, I have separated the full dataset with most of the images being used for training and a small amount reserved for testing. Based on your dataset, you can feel free to adjust how many you allocate to each; however, the general principle is around 20% for testing and 80% for training.

You'll need to select the `training` folder, which contains all of your folders and their images inside. Create ML can understand this structure and automatically generate labels for you.

Great! At this point, you should have finished naming your project and importing your image data. When you're done, your window should look something like this:



Data

Training Data: 5 Classes, 696 Items, View — training

Validation Data: Auto, Split from Training Data — Automatic

Testing Data: 5 Classes, 95 Items, View — testing

Parameters

Maximum Iterations 25

Augmentations

Augment Data: Add Noise, Blur, Crop, Expose, Flip, Rotate

Ready to train for 25 iterations

Activity	Oct 29, 2020
Testing Data Added testing	8:26 PM
Training Data Added training	8:25 PM
Data Source Created training	8:25 PM
Data Source Created testing	8:25 PM
Model Source Created Dog Breed Classification 1	8:25 PM
Project Created Dog Breed Classification	8:25 PM

Figure 4-5: Importing the training data

As you can see, it counts the number of **Images** and the number of **Classes**. In case you aren't familiar with the term, classes refers to the object's label. Now you're geared up to train your model.

Exploring Parameters

Though simple, Create ML allows you to configure more complex options if necessary. While not as powerful as Turi Create, it allows for a few additional "tweaks" to your model's performance.

Parameters Tab

First, let's explore **Parameters**.

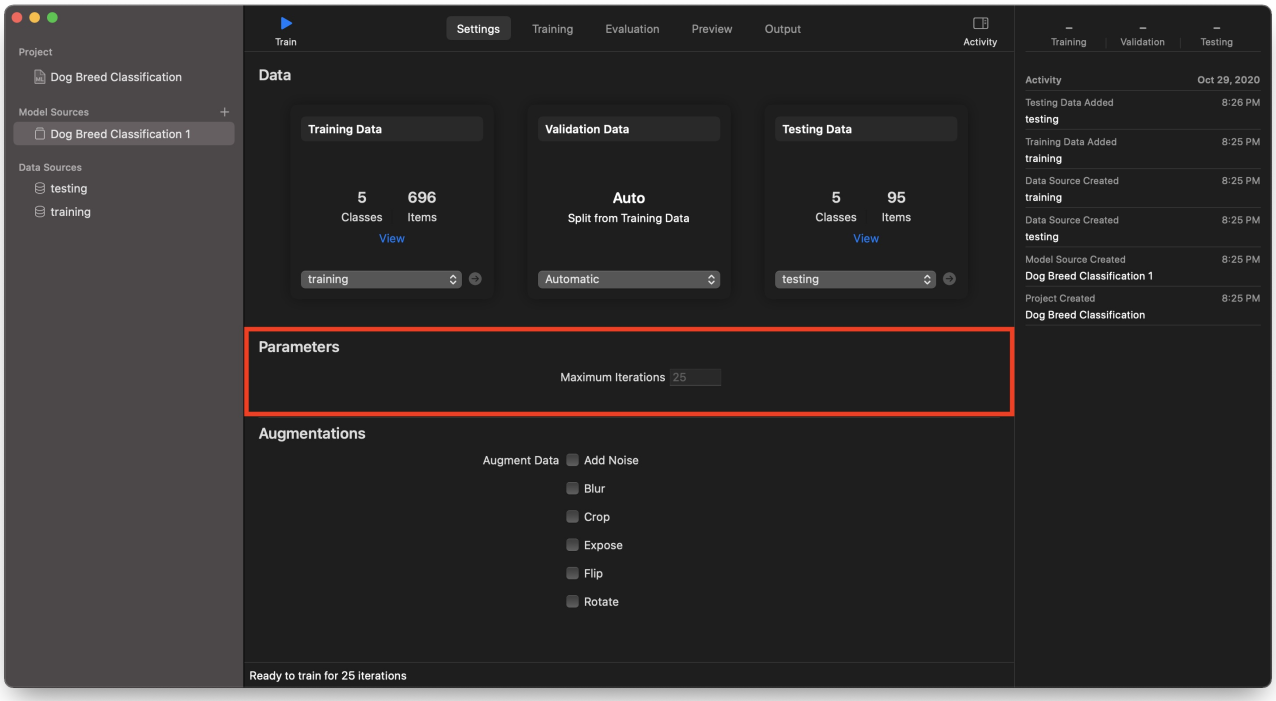
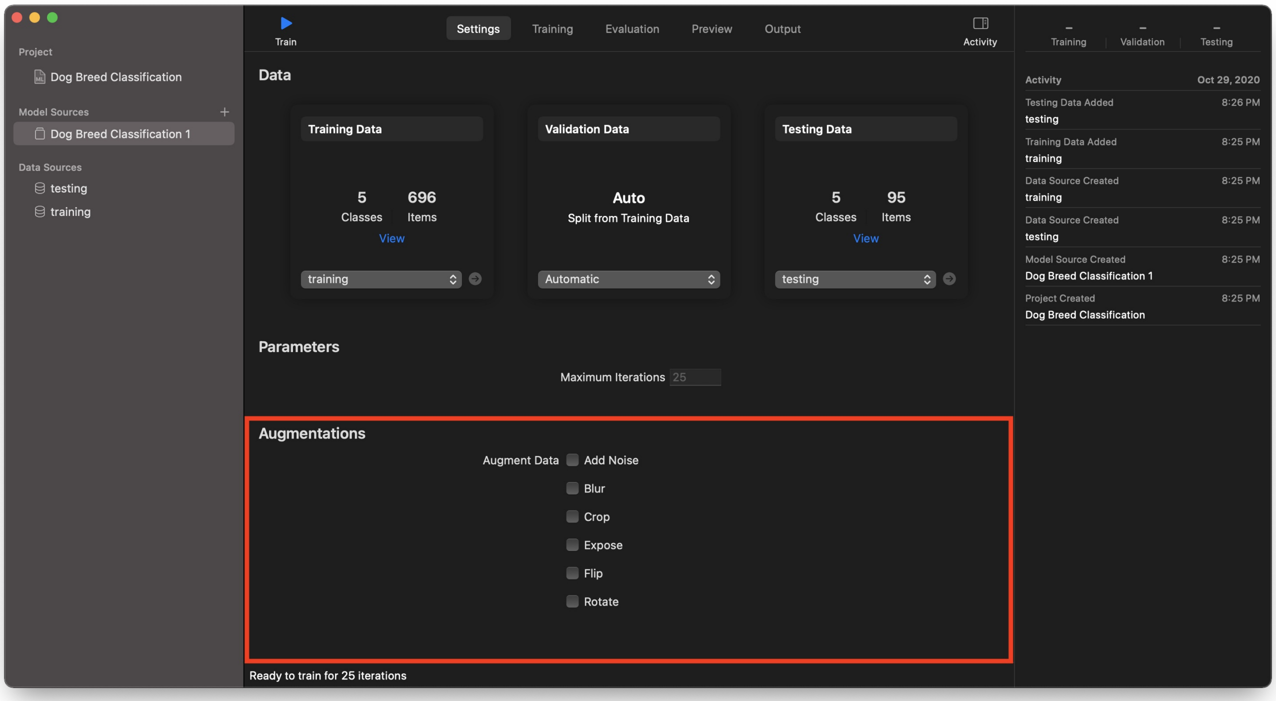


Figure 4-6: Viewing Parameters

Currently, there is only one option, called **Max Iterations**. However, Apple may choose to add more augmentation in future releases of Create ML. The **Max Iterations** feature lets you limit the number of times the model refines itself. For a dataset as small as ours, this number won't matter. But if you're dealing with larger datasets, you'll likely find this option useful to limit the amount of time it takes to train your model.

Augmentations Tab

Next, you can take a look at the **Augmentations** that you can use on your model. Similar to the **Parameters** tab, you can expand it by tapping on the small arrow on the left of the label. It'll look like this:



Data

Training Data: 5 Classes, 696 Items, View — training

Validation Data: Auto, Split from Training Data — Automatic

Testing Data: 5 Classes, 95 Items, View — testing

Parameters

Maximum Iterations 25

Augmentations

Augment Data: Add Noise, Blur, Crop, Expose, Flip, Rotate

Ready to train for 25 iterations

Activity	Oct 29, 2020
Testing Data Added testing	8:26 PM
Training Data Added training	8:25 PM
Data Source Created training	8:25 PM
Data Source Created testing	8:25 PM
Model Source Created Dog Breed Classification 1	8:25 PM
Project Created Dog Breed Classification	8:25 PM

Figure 4-7: Viewing Augmentations

You'll see an array of options here, which are designed to increase your dataset. If you want more data to train your model with, these features will apply these "effects" to the images in your dataset and make a copy of the images. So one will be augmented, and one won't be. This way, the model has doubled the data to learn with. However, you should only use this if you know that the augmentation won't cause your image to be too different from what it'll receive as input.

Training

Now that you've had a chance to explore the various options and configurations for your Create ML model, you're ready to finally train it. We'll use the raw images, however, feel free to apply the augmentations you learned about; they won't make or break your image classifier!

Select Validation Data

Before we start to train our model, we'll have the option of specifying validation data to go with it. Validation data, unlike testing data, is a pre-labeled dataset which isn't used to train the model but is used to check the model at each iteration.

We can allow Create ML to automatically do this for us—by setting aside a subset of the labelled data we've provided. In the second cell, you'll notice that the **Auto** option is already selected for us, which, as the name suggests, means that Create ML will randomly set aside certain images for the validation set.

Change Parameters and Apply Augmentations

In the previous part, you've learned a little bit about the augmentations you can apply before training your model. If you wish to, you may apply them at this step. For the purposes of this chapter, we won't be adding them because our dataset is large enough not to require augmentations and small enough not to require limiting the maximum number of iterations. However, if you'd like to play around with these settings, you may.

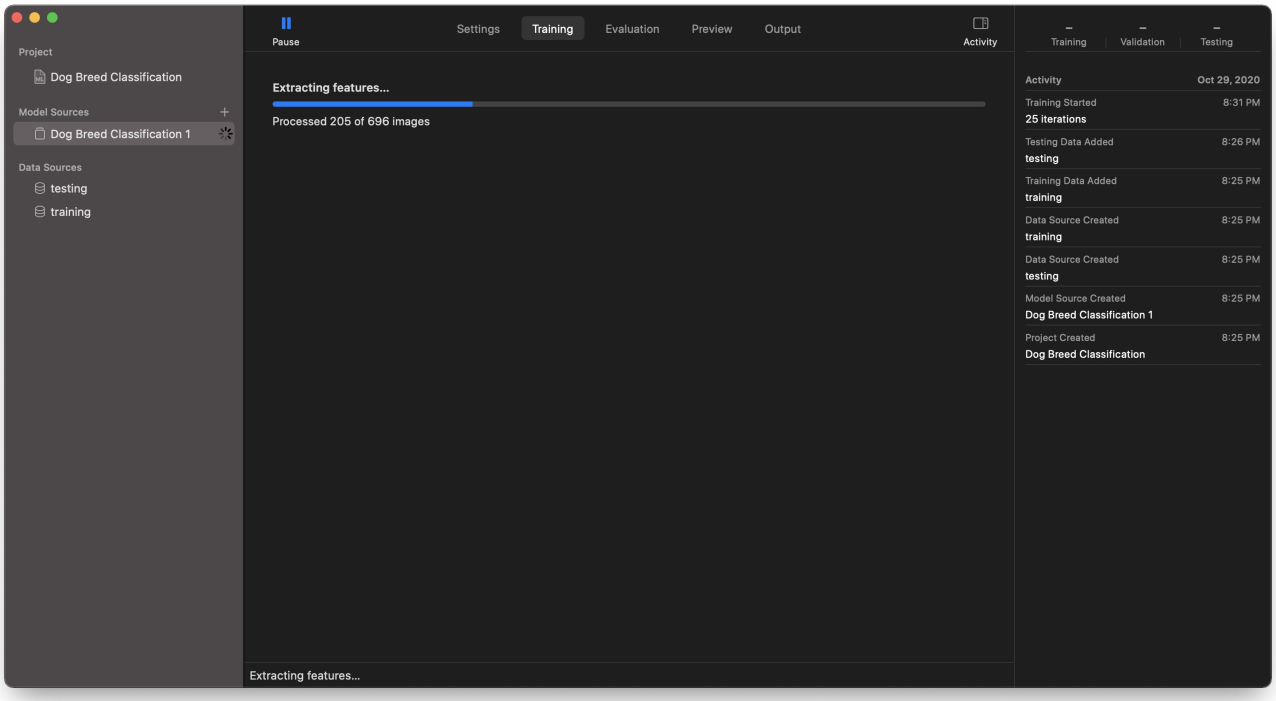
Testing Data

In the repository I provided, you'll find a folder called `testing`. Download these images and upload them to the **Testing Data** section of your Create ML window. At the end of this chapter, we will discuss the importance of testing datasets and how they are important to your machine learning workflow.

Training your Model

Finally, it's time to train your image classification model. Create ML, as you know, makes it much easier to do than if you were doing it using Turi Create.

At the bottom left of your screen, you'll see a message that reads "Ready to Train," which means your dataset has been validated by Create ML and you're ready to train your model. Tap the **Train** button in the top left of the Create ML window. It has a "play" button as its icon, similar to the **Run** button in Xcode. First, Create ML will go through a process of analyzing the features of your images with a progress bar like this:



Extracting features...

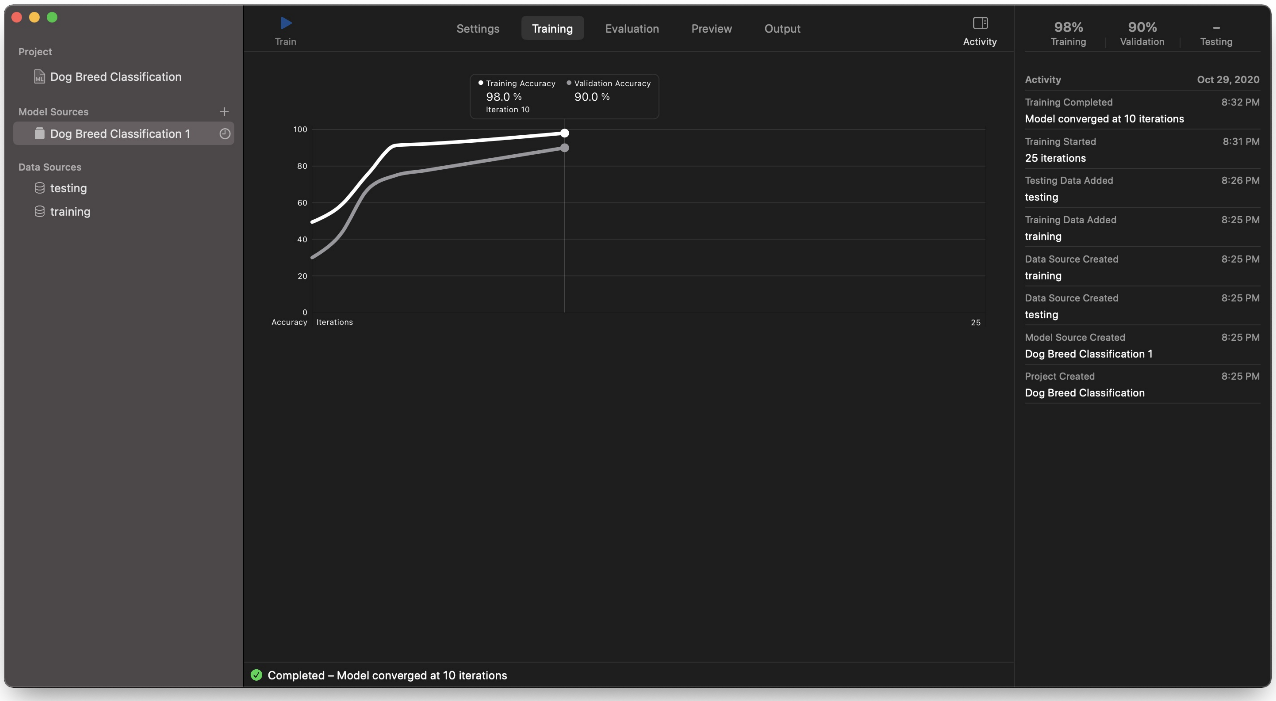
Processed 205 of 696 images

Extracting features...

Activity	Oct 29, 2020
Training Started 25 iterations	8:31 PM
Testing Data Added testing	8:26 PM
Training Data Added training	8:25 PM
Data Source Created training	8:25 PM
Data Source Created testing	8:25 PM
Model Source Created Dog Breed Classification 1	8:25 PM
Project Created Dog Breed Classification	8:25 PM

Figure 4-8: Extracting Features

Then, a graph will appear, and the software will iterate over your model to optimize it:



Training Accuracy 98.0 % · Validation Accuracy 90.0 % · Iteration 10

Activity — Oct 29, 2020

Training Completed — 8:32 PM
Model converged at 10 iterations

Training Started — 8:31 PM
25 iterations

Testing Data Added — 8:26 PM
testing

Training Data Added — 8:25 PM
training

Data Source Created — 8:25 PM
training

Data Source Created — 8:25 PM
testing

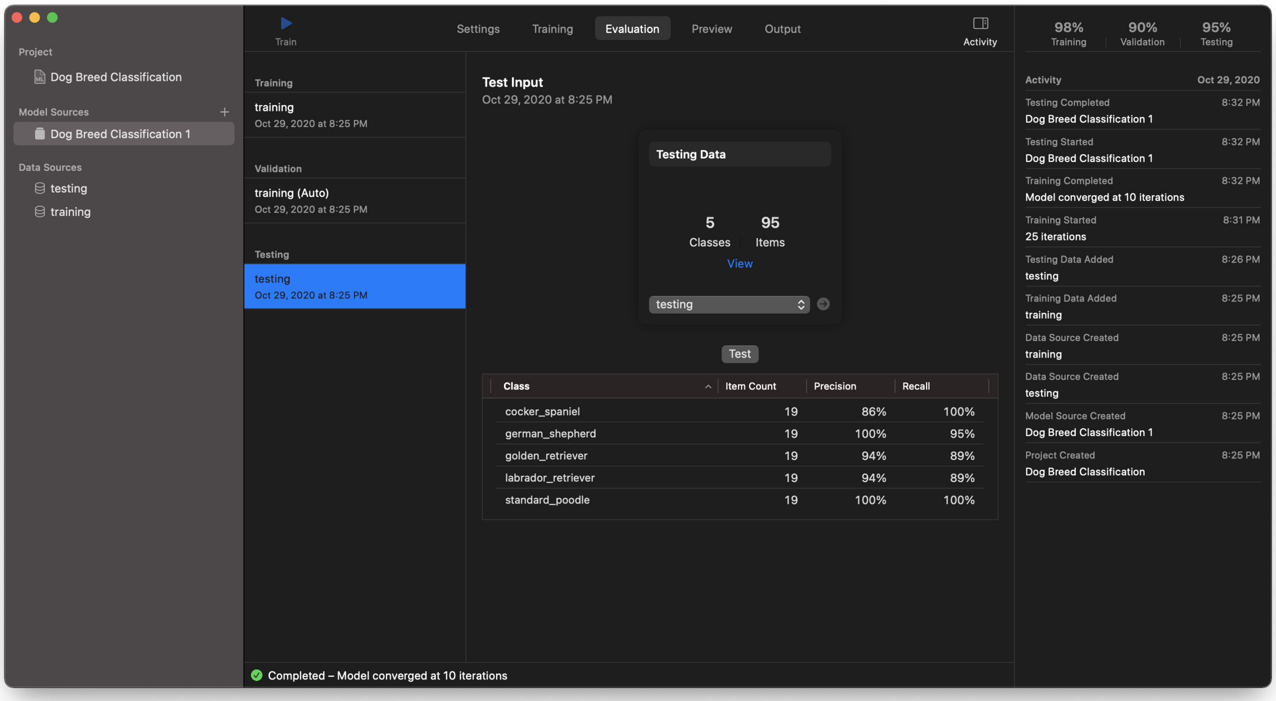
Model Source Created — 8:25 PM
Dog Breed Classification 1

Project Created — 8:25 PM
Dog Breed Classification

Completed – Model converged at 10 iterations

Figure 4-9: Training Model

Last, you'll see percentages appear in the upper right of the main window pane, which tell you how accurate your model is. If you specified any testing data, your results will appear in the center of the screen.


Project
Dog Breed Classification
Model Sources
Dog Breed Classification 1
Data Sources
testing
training

Train | Settings | Training | Evaluation | Preview | Output | Activity

98% Training | 90% Validation | 95% Testing

Training
training
Oct 29, 2020 at 8:25 PM

Validation
training (Auto)
Oct 29, 2020 at 8:25 PM

Testing
testing
Oct 29, 2020 at 8:25 PM

Test Input
Oct 29, 2020 at 8:25 PM

Testing Data
5 Classes
95 Items
View
testing
Test

Class	Item Count	Precision	Recall
cocker_spaniel	19	86%	100%
german_shepherd	19	100%	95%
golden_retriever	19	94%	89%
labrador_retriever	19	94%	89%
standard_poodle	19	100%	100%

Completed – Model converged at 10 iterations

Activity	Oct 29, 2020
Testing Completed Dog Breed Classification 1	8:32 PM
Testing Started Dog Breed Classification 1	8:32 PM
Training Completed Model converged at 10 iterations	8:32 PM
Training Started 25 iterations	8:31 PM
Testing Data Added testing	8:26 PM
Training Data Added training	8:25 PM
Data Source Created training	8:25 PM
Data Source Created testing	8:25 PM
Model Source Created Dog Breed Classification 1	8:25 PM
Project Created Dog Breed Classification	8:25 PM

Figure 4-10: Viewing Precision and Recall

The table at the bottom shows you how accurately each class was trained, as well as there number of images in each class.

Using the Training Model

Your hard work has finally paid off! If you'll recall from the previous section, you ended with a machine learning model in the Core ML format, which you could download and use directly in your apps. You now have a similar model from your Create ML training.

Exporting your Model

The rightmost cell in the top row labeled as **Output** contains the Core ML logo. This represents your trained model, which you may drag onto your desktop or directly into an Xcode project.

Testing Different Models

If you want to create many different models to compare with one-another, you can click the **Add** button in the top-left of your Create ML window. This will let you change your datasets, parameters, or augmentations to see what works best in each scenario.

Evaluating Model Performance

When you train your model, Core ML creates a **validation set** automatically from your training data, while you provide it a **testing set** to use after your model has trained. This can be confusing for beginners, so let's take a look at the nuances.

Validation Set

The validation set, not to be confused with the testing set, evaluates the model *while* training, causing Create ML to adjust the model weights accordingly. This set is created using the data you provided for training, which as 80% of your total dataset (as discussed in **4-1 Finding and Preparing Data**). Another nuance of the validation set is that it isn't used to train the data directly; it is used to evaluate the model after every iteration (or step) in the training process.

Testing Set

The other 20% of the dataset, or the testing set, had never been "seen" by the model. You can use the testing set to determine the final model's performance. The performance of the model does not improve or deteriorate based on the results of the testing set. Providing a testing set only provides you information on how your model is performing to determine whether you need to continue working on it or whether it's ready for use.

Importance of Testing

Testing your models is important because it helps you gain insights on your model's real-world performance before your users see it. Depending on the type of model you're creating, you may have stricter or looser requirements for your model's performance, and using a testing dataset provides the information you need to know to determine whether your model is on-par or needs work.

When looking through your model’s results, look for where your model normally goes wrong. If it’s overfitting the data, try creating a more diverse dataset. If it’s underfitting, try and increase the number of iterations. To learn more about the fundamentals of machine learning, visit **Chapter 1**.

Conclusion

In this chapter, you learned about image classification models, what they're useful for, and how to train your own image classification models to be used in your apps. Through Create ML, you trained these models and changed certain parameters to improve their accuracy.

By the end, you'd created a working image classification model, which you can drag into any iOS app to deploy using Core ML. Lastly, you learned how to optimize your models and ensure that they perform to your standards before you begin shipping them with your apps.

Chapter 5
Natural Language Processing

In the previous chapter, you built an image classification model using two different techniques. To challenge yourself, you first built it in the "hard" way, using your new knowledge about Python from Chapter 3 in Turi Create. You also learned the easier way, using Create ML to make lightweight models for quick on-the-go model training.

In this chapter, you'll learn about linguistics in machine learning, including natural language processing, text classification, and word tagging. In addition to this, you'll learn more about sentiment classifiers, which help identify the mood or intention behind what someone is saying.

By the end of the chapter, you'll have some useful, fully trained models which you can use in your apps. You'll also walk away with a complete understanding of how computer science and linguistics are connected, as well as how you can use the power of natural language processing in your day-to-day workflow.

5-1 What is Natural Language Processing?

Natural Language Processing, or NLP for short, is everywhere. You've likely seen it in action without even knowing it. If you've ever spoken to Siri, typed a message on your phone, or dictated an email, your language was processed by a machine learning algorithm to interpret it, and even to predict what you're about to say (or write) next.

In this section, you'll learn about NLP, Word Tagging, and Text Classification from a high-level perspective. Though interrelated, these concepts have a fine line of distinction between them and are important to know about before training your own linguistics algorithms. By the end of the section, you'll be familiar with the basics of machine learning in linguistics.

Natural Language at a Glance

From a bird's eye view, NLP enables many human-computer interactions you experienced on a daily basis. Virtual assistants and dictation-enabled keyboards are great examples of NLP in action.

Day-to-Day Speech

In order to understand natural language processing, however, you first need to understand what is meant by natural language. Computer code, such as Swift and Python, has defined sets of rules which must be followed closely. For example, if you miss a curly brace in Swift, the compiler won't let you run the code.

However, in day-to-day speech — say with your friend at work or school, you don't need to worry about syntax. Your friend will understand what you're trying to say even if you missed a word or used an incorrect sentence structure. This style of speech is referred to as natural language, in computer science terms.

Written Text

Natural language also comes in the form of written text. Whether it's texting a coworker or jotting down notes, what you're putting down is natural language which doesn't necessarily follow a certain structure — like computer code would. For example, if you made a to-do list, there are infinitely many ways you could say the same task; but in a compiled language, there are fewer and more structured ways to convey your ideas.

Semantic Search

When you ask your digital assistants — such as Siri — to "find files from last night that were shared with Tim," you're using semantic search. In case you aren't familiar with this, the algorithm focuses on the context more than the content of the text. So, when it's looking for files on your computer, it'll check that those were created *last night*, and were *shared* with *Tim*. It won't, however, look for files which contain the word *Tim* in them.

Challenges of NLP

Now that you've seen a couple examples of natural language, you may have a basic sense for what makes natural language processing so difficult. After all, a computer is terrible at "thinking" about what it's doing, but is excellent at repetitive tasks. The problem of natural language processing can be broken down into converting unstructured text into structured, parsable information for a computer.

Vocabulary and Context

While you may think the words you use on a regular basis are just ordinary, it's quite the opposite from a computer's perspective. For example, you may use the word "leaves" in the sentence "I saw the leaves on the tree," you may also use it in a completely different context, "I'll do it when he leaves today." You might understand these differences in a heartbeat, but to an algorithm, understanding context can be very difficult.

Parts of Speech

As you may remember from your English class, words can be categorized into groups called *parts of speech*, some of which include adjectives, nouns, prepositions, and more. A computer doesn't understand language in the same way we do, so it needs to break down sentences into more quantitative measures such as parts of speech.

Phrases

After identifying the parts of speech, the algorithm must recognize phrases. Again, since the algorithm doesn't understand context, it needs a qualitative measure to further group the categorized words. This helps it identify the main subject and verb of the sentence to get a better sense of what the user is saying. At this point, the computer usually has a sentence, which is broken down into phrases, which are broken down into individual words, forming a sort of tree.

Clearly, natural language processing is a challenge. With machine learning, this process becomes slightly more human-like. However, a computer still doesn't process natural language like humans do.

Types of NLP

Now, you should have a better sense of why natural language processing is so difficult; large corporations such as Google, Apple, and Amazon have poured billions into developing their versions of the technology. Since computers cannot process natural language like humans, there are several approaches to help. Keep in mind that there are thousands of different uses of NLP technology, and the approaches discussed below are tailored to those specific use-cases.

Text Summarization

Oftentimes, natural language processing takes extensive expertise to execute well — and for good reason. Because of this, machine learning developers use a more basic approach which works fairly well for most purposes — Text Summarization. At a fundamental level, this approach extracts all the words and counts the most common ones. While easier than many of the other NLP methods, this approach must ensure that the sentences formed make sense. Don't worry if you cannot understand it yet. We'll discuss this technique in more detail in the next section.

Translation

If you've ever used a translation service such as Google Translate, you know firsthand how much of the meaning can be lost while trying to task a computer algorithm with translations — it adds a new level of difficulty in NLP. Not only does a computer need to understand the source language, it needs to produce logical text in the target language. In Google Translate's case, they use their own computer-recognized language.

For example, if you're translating from English to Chinese, the algorithm would take the English string and translate it into a language which only Google's algorithm would understand (i.e. jargon to us humans). If it has doubts about any of the text, it'll go through prior uses of the phrases it doesn't understand and translate it that way. Finally, it'll use machine learning to produce text in Chinese and run it through another algorithm to ensure that it makes sense.

Sentiment Analysis

As the name suggests, sentiment analysis is the use of a natural language processing algorithm to analyze the emotion behind text. This often involves taking out adjectives and qualitative words to check whether a certain text is more positive or more negative. Advertisers use this method to analyze whether there's positivity around their products on social media, or whether their consumers dislike what they've released. In addition, this helps social media platforms censor inappropriate content or hate speech from younger users.

5-2 Tokenization, Stemming, and Lemmatization

In the previous section, you learned about natural language processing from a birds-eye view and some of the challenges that developing this technology poses. You also learned about the various ways you use NLP in your daily life. You are now familiar with how text summarization, language-to-language translations, and sentiment analysis work.

In this section, you'll learn more about tokenization, stemming, lemmatization, and many of the other steps that a NLP algorithm needs to take in order to get to its desired result. In essence, you'll dive deeper into what makes your day-to-day NLP interactions tick, which is especially important given the vast differences in the applications of this technology.

Understanding and Responding

As you've learned from the previous section, there are countless ways that NLP is used in your daily life. Whether it's talking to Siri or having a news article summarized, some forms of NLP is heavily involved in the process. Given that these algorithms are often replacing humans (such as in chatbots), much of the work is split between understanding language and responding to it in a comprehensible manner.

Extracting Meaning from Text

As briefly discussed in the previous section, NLP algorithms have various techniques to understand language. Unlike humans, these algorithms turn natural language into a chunks for quantitative analysis. Regardless of the method they use, however, understanding the gist of what's being spoken or written is integral to an NLP algorithm's success.

Producing Comprehensible Results

In some cases, an NLP algorithm must also be able to create output which makes sense. For example, when you text a chatbot on your company's website, the software needs to understand what you're saying, give you a response that makes sense to you, and conveys what the robot is trying to convey accurately and concisely. This allows for a significantly better user experience and one day may allow natural language processing algorithms to pass the Turing test.

Internal Steps and Techniques

As explored in depth in the previous section, an NLP algorithm's job is no easy task. In fact, both understanding and responding to human-produced language have their unique challenges. It's important to understand the specific steps a typical NLP algorithm takes to reach its end goal. Obviously, some algorithms may use a different approach or skip steps entirely — this is only a general idea of how these algorithms work.

Tokenization

A machine learning model, or an NLP algorithm, must first break up the many words passed in as input. Typically, this involves simply separating words, and in most cases, removing punctuation. Tokenization is usually used on the "understanding" side of natural language processing, as it involves breaking up an existing sentence. Unlike some of the other techniques you'll learn about, tokenization is the most widely used one, ranging from text summarization to language translations.

For example, consider the following phrase:

```
"The quick brown fox jumped over the lazy dog."
```

The tokenized result would look like this:

```
0: "the"
1: "quick"
[***]
7: "lazy"
8: "dog"
```

While seemingly fundamental, this task is vital to the algorithm’s success. Breaking the problem down into a whole bunch of smaller sub-tasks allows the program to have its "first try" at understanding the input text. Tokenization is one of the most straightforward parts of natural language processing, and requires only basic string processing.

Stemming

As you may have guessed from its name, stemming takes the root word ("stem") out of a tokenized word. This often involves removing prefixes such as *pre-, de-, anti-* and suffixes such as *-ing, -illy, -ed*. This greatly simplifies the sentence and makes it easier to interpret the meaning.

Here’s an example of this:

```
"jumped"     ->  "jump"
"running"    ->  "run"
"different"  ->  "differ"
```

As you can see, stemming simply removes prefixes and suffixes which it deems unnecessary to be left with a root word. However, it’s worth noting that stemming does not consider the dictionary definitions of the words it’s altering. So, when an algorithm is in its stemming phase, it may lose some meaning, depending on its complexity. When a word loses its meaning, it’s usually a result of **understemming or overstemming**, which is when too much of the word is left behind or too much of a word gets cut off.

Consider the role that root words play in your own day-to-day language: you have to take a root and add a prefix or suffix to convey what you’re trying to say. This is known as **inflected language**, and it allows you to form grammatically correct sentences.

Lemmatization

Stemming and lemmatization are both very similar: they're both techniques used for **text normalization**. Text normalization is the idea of removing the parts of a word that don't define its meaning. Lemmatization is a process in which a word is "reduced" while staying true to its dictionary definition. The output after lemmatization, unlike stemming, is an actual word.

The following is an example of this:

```
"run", "ran", "running" -> "run"
"hopped", "hopping", "hop" -> "hop"
"eaten", "eating", "eat" -> "eat"
```

In this example, *run, hop,* and *eat* are all actual words, but they may not make grammatical sense in all types of sentences. Lemmatization relies on the word's **lemma**, which is defined as a word's dictionary definition. Therefore, a natural language processing algorithm which uses lemmatization would require a dictionary to operate.

Lemmatization is understandably more complex than stemming as it requires a solid understanding of the target language. Unlike stemming, lemmatization does not rely on the structure of the word, but instead, it focuses on the definition. Before making any changes, it examines the parts of speech, the context, and placement of a word in a sentence to ensure that alterations don't affect the meaning of the word in question.

Chunking

As you learned earlier, tokenization is the separation of sentences or phrases into words or "tokens." Chunking is a technique in which separate words are put together into phrases to form something that makes sense. In a way, chunking is the opposite of tokenization and is used mostly on the "responding" side of natural language processing.

One major part of chunking is called **part-of-speech tagging**. As the name suggests, it classifies words based on whether they're nouns, verbs, prepositions, or adjectives. This allows the algorithm to create a sentence which is structurally sound and sensible to the user.

Another large part of chunking is **named entity recognition**. Named entity recognition is the process of recognizing proper nouns. For example, if the words *Donald, President*, and *Trump* showed up separately, named entity recognition would help a chunking algorithm combine them into a phrase: *President Donald Trump*.

5-3 Training a Text Classifier

In the previous sections, you learned about the basics of natural language processing, ranging from the various use cases of the technology to the steps that NLP algorithms take under the hood. You also saw some examples of how these steps, including tokenization, stemming, lemmatization, and chunking, interact to face the challenge of understanding and generating natural language.

In this section, you'll put your skills to the test and build your own text classifier. Earlier in the book, you created your own image classifier using the Create ML app. Although you can easily create a text classifier with the same method, you'll learn to use Swift playgrounds to make a text classifier in this section. This way, you'll be able to understand what's happening behind the scene, and you'll know multiple ways of doing the same thing — like a true data scientist!

Finding and Sourcing Data

In the image classification chapter, you learned about the importance of finding a good dataset for your image processing needs. I also shared a list of excellent places you can go to find these datasets.

Finding Models

Unlike image classification datasets, which are plentiful on the internet, text classification datasets are harder to come by — partly because of how specific NLP is to a particular genre, style, or type of writing. However, there still are a few good places to look.

GroupLens Movie Dataset (https://grouplens.org/datasets/movielens/latest/)

GroupLens Research has put together a great combination of movie reviews, which they've labelled as positive or negative. If you'd like to use this resource, the link above has a couple variations of the dataset. This dataset is mostly useful for sentiment analysis — especially in the case of customer reviews.

Spambase Dataset (https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/)

The Spambase Dataset has a ton of emails labelled as either spam or not spam. This type of dataset is useful when you would like to classify social media posts, or maybe, you're simply building another spam detector. Either way, this dataset is well-made and is an accurate representation of actual emails.

Cornell Review Dataset (http://www.cs.cornell.edu/people/pabo/movie-review-data/)

Cornell's excellent computer science department came up with another great movie review dataset. This is the one we'll be using in the text classifier a little bit later. It is great to analyze product reviews, business ratings, or movie critic comments. It's also lightweight and doesn't take too long to train on.

Creating Datasets

It doesn't take a lot of data to have a decent starter model. In fact, it's better to test out the data you have before adding more. That's one of the biggest benefits of lightweight, easy, and fast training tools like Create ML and Turi Create. It just takes two steps to create your own model:

Create Folders

First, you need to create folders. Similar to the image processing chapter, you'll need to organize your data inside their corresponding folder. For example, all of the "positive" documents should go in the "positive" folder. This will help Create ML determine your labels.

Write Text Files

Next, you need to create text files. These files can be copied from the internet (always cite your sources!) or generated by yourself. All that matters is that these files you made are representative of the entire sample size. They must also be as close to the target text as possible. For example, if you're making a Twitter sentiment analysis model, don't use formal literature as your training data.

Text Classification Algorithms

Now that you've either found a dataset, created your own, or used the one we're using in this book. You're ready to learn more about the different algorithms you can choose in Create ML. Don't worry, this isn't as long as the previous two sections, and I highly recommend that you read through it carefully. Here are the three algorithm types Create ML supports:

Transfer Learning

Apple's latest addition to Create ML is the support for transfer learning. It allows developers to get the benefits of using large datasets while using small datasets. Transfer Learning, as its name suggests, uses an existing machine learning model to *transfer* the information it learned to a different use case. Since Apple's large machine learning model has learned the semantics of English, your models will have a higher accuracy.

Maximum Entropy

The Maximum Entropy algorithm, or MaxEnt for short, is the fastest option for training. Since this algorithm makes few assumptions about the target data, it's best to use it when you don't know anything about the data you're using. In addition, data scientists often use it for testing because of how much faster it is than the other options that Create ML offers.

Conditional Random Field

Conditional Random Field is a more quantitative approach to natural language processing. It relies on the number of words present rather than the structure of the sentences. This approach usually works best when the phrases in the training data aren't very complex, and don't obscure meaning with double negatives or prefixes such as *anti-*, *de-*, or *un-* that change the meaning of words.

Creating your Classifier

Great work so far! You're now ready to create your own text classifier. In this classifier, you'll use the Cornell Movie Dataset, which I've edited for the purposes of this book and put in a repository for you to download at https://github.com/vhanagwal/sentiment-analysis. You can click the download button to download the dataset, which contains 1000 positive and 1000 negative reviews. I've gone through and split up 20% of the data for testing (in a `testing` folder) and 80% for training (in a `training` folder). Your model will be able to classify a positive comment versus a negative comment by the end. As mentioned before, this is a great dataset to get started with because it uses a realistic, informal style of speaking.

First and foremost, you'll need to create a Swift Playground. In case you haven't used them before, Swift Playground is a place where you can quickly test out your Swift code without creating a fully-fledged Xcode project. It's also excellent for tasks like creating a machine learning model with Create ML.

Swift Playground is a part of Xcode, so go ahead and open a new Xcode window. Then, click on the first option, **get started with a playground**.

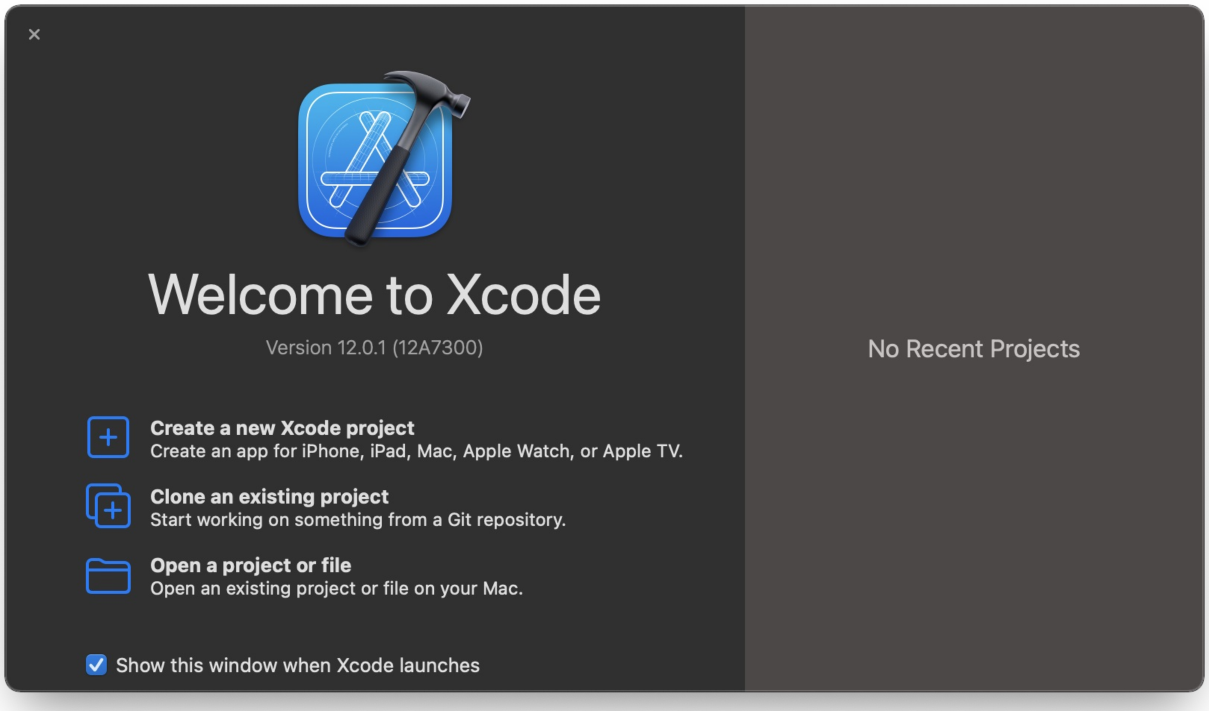

Welcome to Xcode
Version 12.0.1 (12A7300)
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.
Clone an existing project
Start working on something from a Git repository.
Open a project or file
Open an existing project or file on your Mac.
Show this window when Xcode launches
No Recent Projects

Figure 5-1: Opening Xcode

Since you're not creating a game or demonstration in Swift Playgrounds, you don't need to use a template. **Blank** should be selected by default, so click **Next**. Please be sure to select *macOS* at the top instead of iOS. Otherwise, all the rest of the procedures won't work.

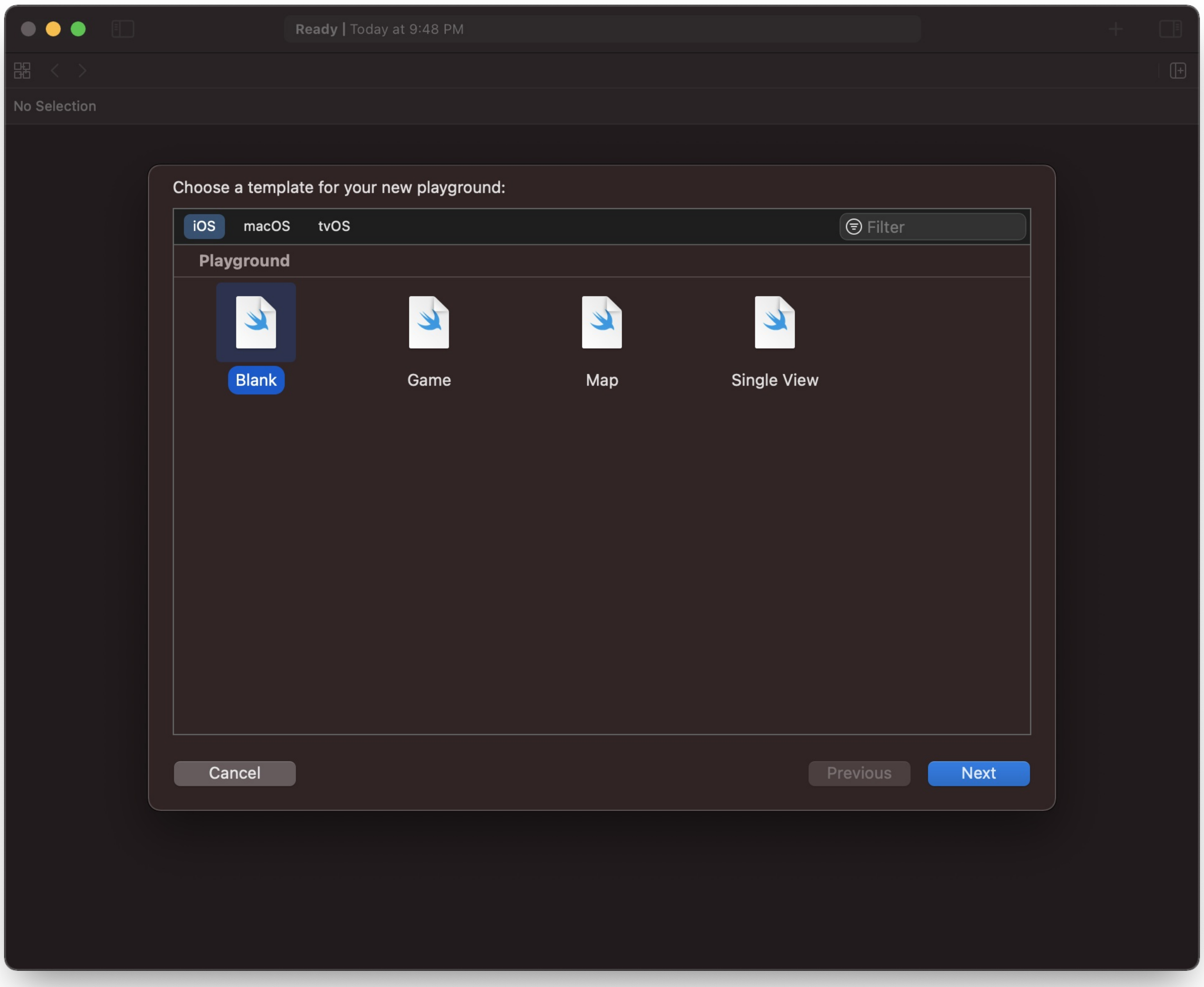
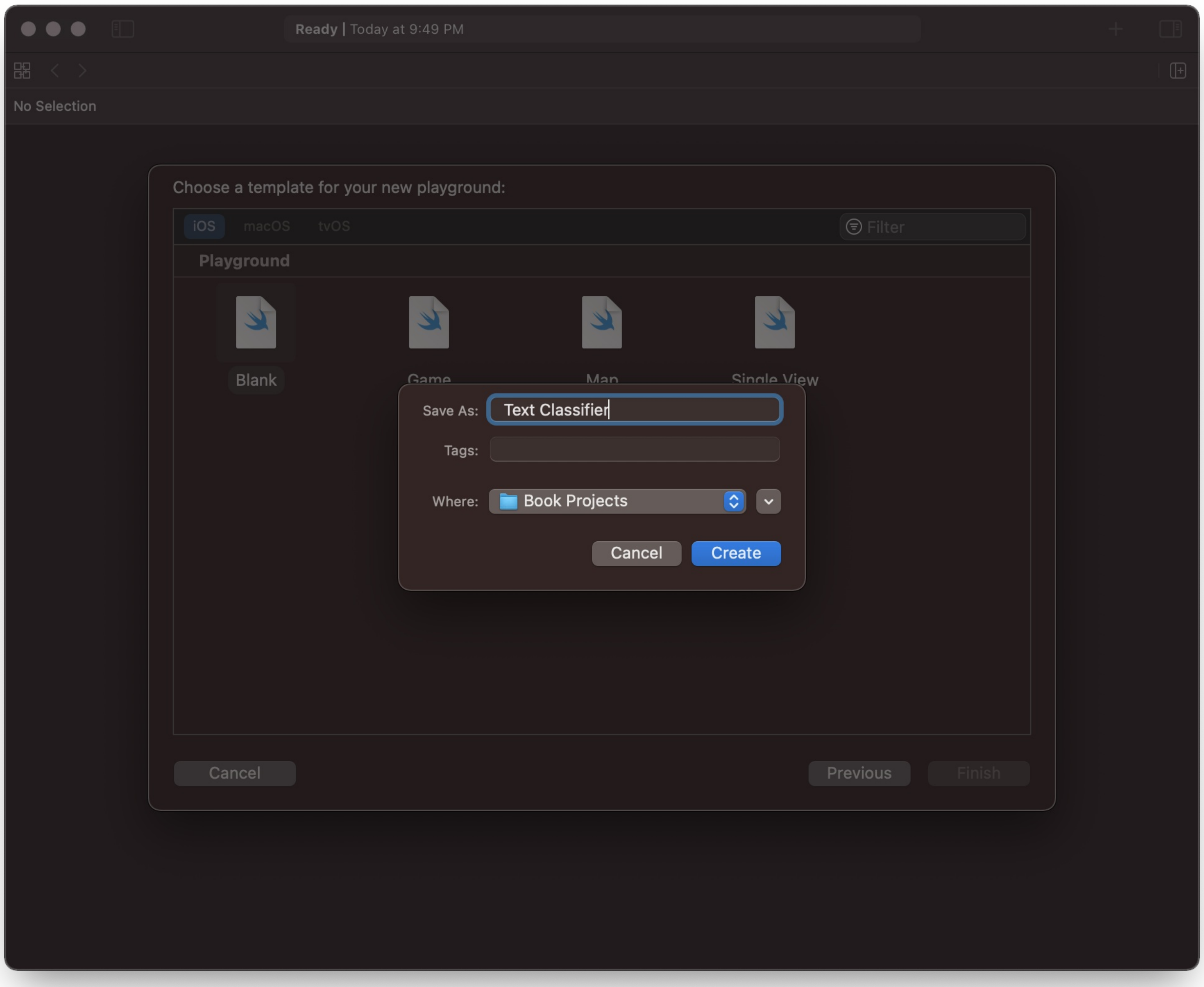


Figure 5-2: Selecting a Template

Next, name your project something fun; something you'll remember. I'm naming mine something boring: "Text Classifier." Then, choose where you want to save your project and click **Create**.

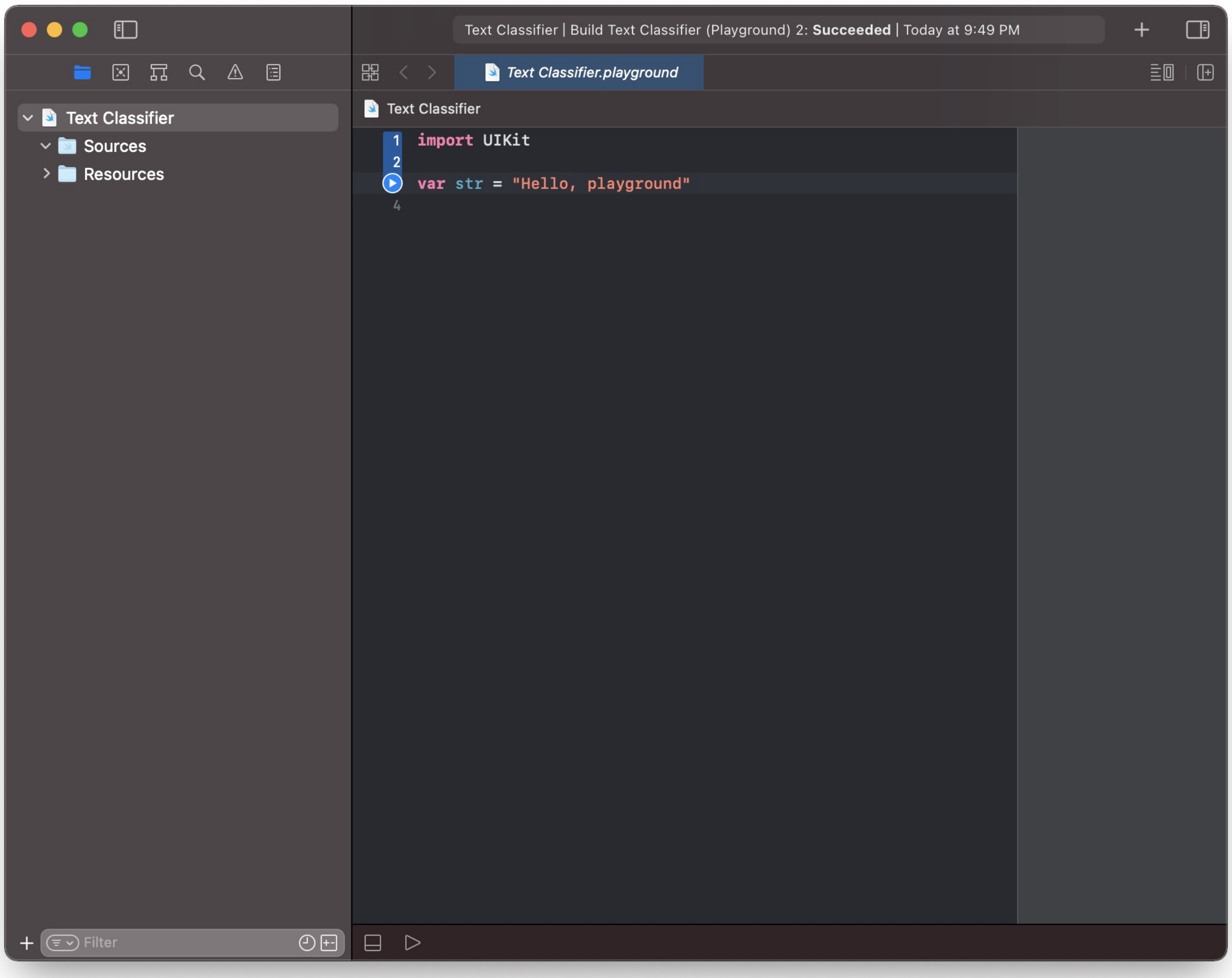


Save As: Text Classifier
Tags:
Where: Book Projects

Figure 5-3: Naming and Saving your Playground

Anatomy of a Swift Playground

Your project is ready, and it should look something like the following. The blue "play" button on the left lets you execute your code partially, and the "play" button at the bottom left of the screen runs all of your code. In the center, you can enter code. That’s all you need to know to get started!


Text Classifier | Build Text Classifier (Playground) 2: Succeeded | Today at 9:49 PM
Text Classifier.playground
Text Classifier
Text Classifier
Sources
Resources
Filter

```
import UIKit

var str = "Hello, playground"
```

Figure 5-4: Playground Editor

Writing the Code

It's now time to start writing the code. Create ML allows you to create a lightweight text classifier in very few lines of code. Don't worry if certain things don't make sense; I'll be explaining each line to make things clear.

Clear Boilerplate Code

Since we’ll be starting from scratch, go ahead and delete everything which is currently in the editor window. After this, you should see a blank canvas and you’re ready to start training your text classifier.

Importing Frameworks

For this part of the book, you’ll be using the CreateML framework, CoreML framework, and basic Foundation framework. With these three framworks combined, you will be able to access to all of the methods you need to build your text classifier. Import all of them like this:

```
import CreateML
import CoreML
import Foundation
```

Loading your Dataset

First, you’ll need to get the dataset you downloaded into a constant that you can access within your code. To do this, declare a new constant and name it `trainingURL`. Then, type the following to declare it:

```
let trainingURL = URL(fileURLWithPath:
    "/Users/vardhanagrawal/Downloads/sentiment-analysis-master/training")
```

This constant essentially stores a URL object which contains the path to your file. Just make sure to change your path to suit your own computer. Assuming you've downloaded the files to your Downloads folder, you can write the code like below by changing the username:

```
let trainingURL = URL(fileURLWithPath:
    "/Users/<YOUR_USERNAME>/Downloads/sentiment-analysis-master/training")
```

By default, macOS saves your downloads at this path, given that your currently logged-in user is entered correctly. Alternatively, you could right-click on your downloaded dataset and click **More Info** to get its path.

Selecting your Algorithm

As you recently learned, there are currently three supported algorithms for text classification in Create ML. They are transfer learning, maximum entropy, and conditional random field. While transfer learning is great, it takes significantly longer time to train. If you have the patience, I recommend going with it. Either way, enter the following line of code into your editor:

```
let parameters = MLTextClassifier.ModelParameters(algorithm:
    .transferLearning(.dynamicEmbedding, revision: nil))
```

For the `algorithm` parameter of the `ModelParameters` method, you need to pass in the algorithm you'd like to use. Your options are the following:

- Conditional Random Field:
  - `.crf(revision: nil)`
- Transfer Learning:
  - `.transferLearning(.staticEmbedding, revision: nil)`
  - `.transferLearning(.dynamicEmbedding, revision: nil)`
- Maximum Entropy:
  - `.maxEnt(revision: nil)`

Whatever you choose to use, you'll need to put it in the `algorithm` parameter of the `ModelParameters` method, and specify `static` or `dynamic` embedding if you choose to us Transfer Learning.

Defining the Classifier

Now that you have your data and your parameters, you're ready to declare your classifier. Since it may throw an error, you'll need to use the `try` keyword to end program execution in case of an error or `nil` value. Type the following:

```
let classifier = try MLTextClassifier(trainingData:
    .labeledDirectories(at: trainingURL))
```

Here, you're trying to create an instance of the `MLTextClassifier` class using the training data from the URL you created earlier. The method attempts to grab data from the provided URL and throws an error if there's nothing there.

Creating Metadata

While this step isn't required, it's always good practice to use metadata correctly. It is what appears when someone tries to open your file. You can define parameters such as the author, description, and permissions, so that people know a little bit about the model if you plan to publish it online or share it with friends. You can define metadata like this:

```
let metadata = MLModelMetadata(
    author: "Vardhan Agrawal",
    shortDescription: "A sentiment classifier.",
    license: "BSD-2",
    version: "1.5.1"
)
```

In this code, I've included my name as the `author` parameter, a quick description in the `shortDescription` parameter, and so on. This is a data type called a `struct` and is named `MLModelMetadata`. We can pass this value into the `write(:)` method later on.

Exporting your Model

Believe it or not, you're almost done with creating your text classifier! All that's left to do is decide where you want your exported model to be and then actually export it. Define your destination path like this:

```
let destinationURL = URL(fileURLWithPath:
    "/Users/vardhanagrawal/Downloads")
```

I've decided to put my model in my Downloads folder, but you can choose to put it wherever you'd like. Again, make sure you have changed the name of the path to something which actually exists on your computer; don't use my name! Finally, export your model as follows:

```
try classifier.write(to: destinationURL,
    metadata: metadata)
```

Again, since there's a URL involved, and there's a degree of uncertainty of whether the `write(:)` method will succeed, you need to use the `try` keyword to catch any errors. In the parameters, you're passing in both the destination URL and metadata constants you created earlier.

Evaluating Model Performance

As you know by now, testing your model's performance using 20% of your dataset is very important. If you haven't yet read Chapter 4, I recommend visiting **4-3 Delving into Create ML** and reading the section labeled **Evaluating Model Performance**. Here, you'll learn about the differences and similarities of the validations sets and the importance of testing your models.

Uploading Testing Data

To test your model, you can upload the 20% of the dataset which I set aside in the provided repository (https://github.com/vhanagwal/sentiment-analysis). This data is also pre-labelled for Create ML to automatically determine the percentage accuracy of the model.

Type the following at the end of your Swift Playground:

```
let testingURL =
    URL(fileURLWithPath: "/Users/vardhanagrawal/Downloads/sentiment-analysis-maste
r/testing")
```

Similar as before, make sure you type in the path which you've saved your dataset, as the above is just an example. In this line, you're storing the URL to the `testing` folder of the dataset.

Testing your Model

Finally, to evaluate your model's performance using the testing set, type the following:

```
print(classifier.evaluation(on: .labeledDirectories(at: testingURL)))
```

This will print out the results of running the testing set through the trained model into your console, which'll look something like this after you run it:

```
Number of examples: 402
Number of classes: 2
Accuracy: 88.06%

******CONFUSION MATRIX******
----------------------------------
True\Pred negative  positive
negative  178       23
positive  25        176

******PRECISION RECALL******
----------------------------------
Class    Precision(%)   Recall(%)
negative 87.68          88.56
positive 88.44          87.56
```

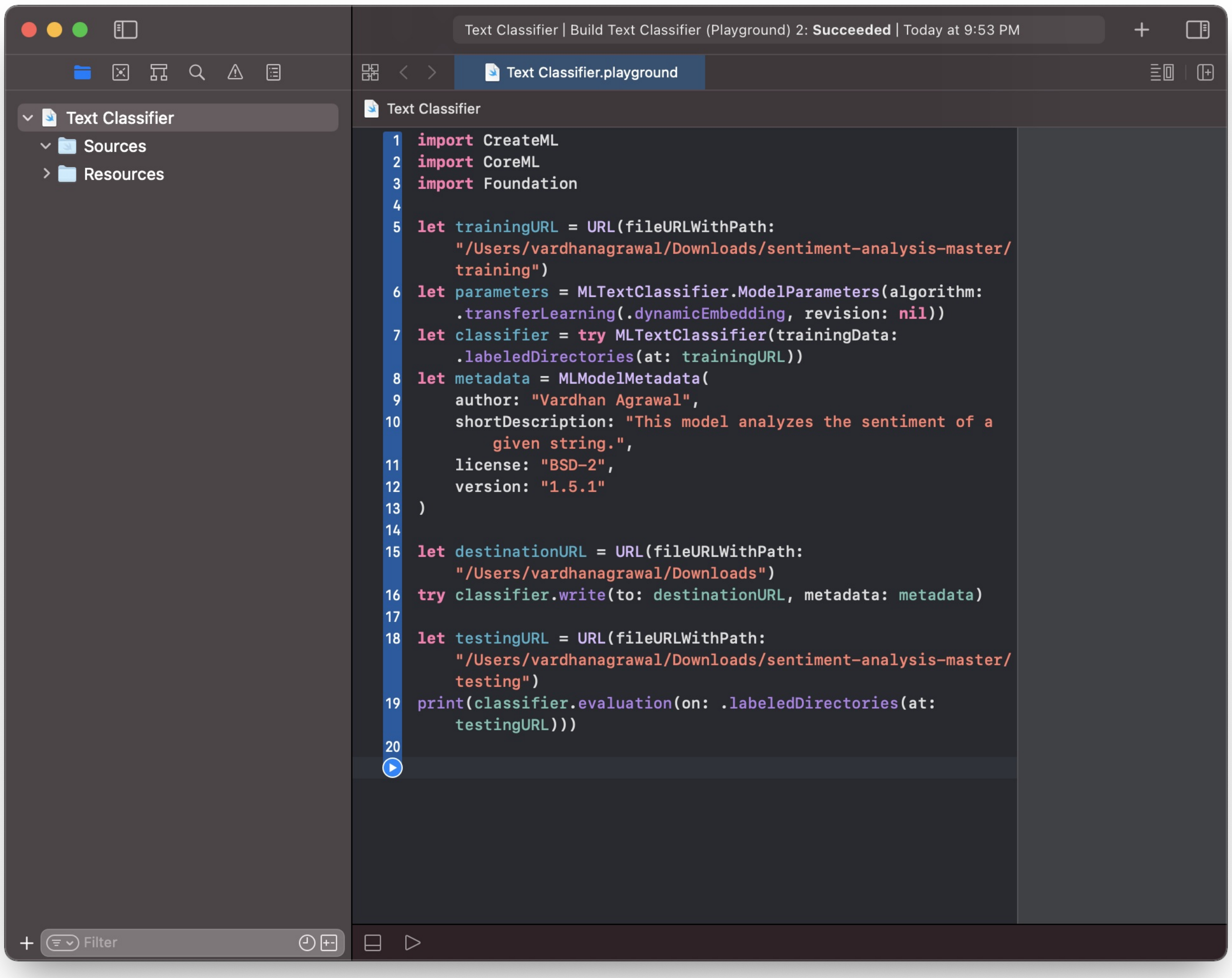
The **accuracy** value (third line) tells you what percentage of the testing set the model got correct. 88% is a great accuracy; however, there's no absolute value which determines how good or bad your model is. Depending on the amount of classes and the dataset size, you'll have different accuracies. If you have a model with hundreds of different classes, for example, 60% may be an acceptable accuracy. But, if you only had two classes, 60% would be considered pretty bad. Make sure you evaluate your model's accuracy based on these parameters. Don't judge a book by it's cover!

The **Confusion Matrix** is a common tool used to evaluate model performance in the data science space, and it shows you how many the model got correct for each class, giving you insights on which classes the model performed best on and which classes the model struggled on.

The **Precision Recall** table is most useful when there are more data available for one class than another. The **Precision** value indicates how many "positives" out of all of the predicted positives in that class were correct, and the **Recall** indicates the number of predicted positives out of the total correct positives. Most of the time, though, you won’t need to use the Precision-Recall table; however, if you want to learn more about this, check out this article.

Run your Code

Now, you’re all done! Just press the "play" button in the lower left of your Xcode window and wait. The console will pop up and show you what’s going on.


Text Classifier | Build Text Classifier (Playground) 2: Succeeded | Today at 9:53 PM
Text Classifier.playground
Text Classifier
Sources
Resources
Filter

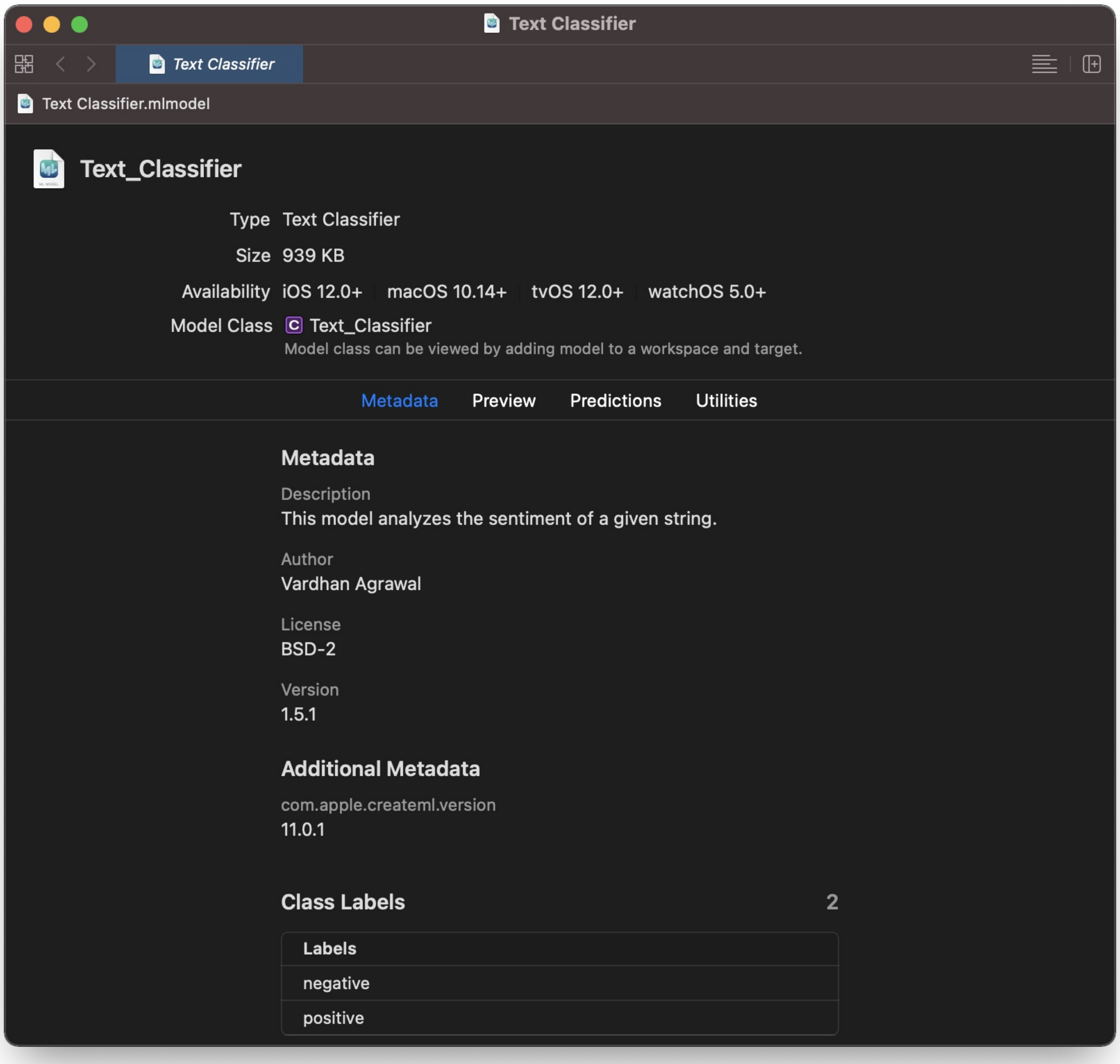
```swift
import CreateML
import CoreML
import Foundation

let trainingURL = URL(fileURLWithPath:
    "/Users/vardhanagrawal/Downloads/sentiment-analysis-master/
    training")
let parameters = MLTextClassifier.ModelParameters(algorithm:
    .transferLearning(.dynamicEmbedding, revision: nil))
let classifier = try MLTextClassifier(trainingData:
    .labeledDirectories(at: trainingURL))
let metadata = MLModelMetadata(
    author: "Vardhan Agrawal",
    shortDescription: "This model analyzes the sentiment of a
        given string.",
    license: "BSD-2",
    version: "1.5.1"
)

let destinationURL = URL(fileURLWithPath:
    "/Users/vardhanagrawal/Downloads")
try classifier.write(to: destinationURL, metadata: metadata)

let testingURL = URL(fileURLWithPath:
    "/Users/vardhanagrawal/Downloads/sentiment-analysis-master/
    testing")
print(classifier.evaluation(on: .labeledDirectories(at:
    testingURL)))

```

Figure 5-5: The final Playground project and the console message

At the end, your console should look something like this:

```
Tokenizing data and extracting features
20% complete
40% complete
60% complete
80% complete
100% complete
Starting MaxEnt training with 1598 samples
Iteration 1 training accuracy 0.500000
Iteration 2 training accuracy 0.950563
Iteration 3 training accuracy 0.989362
Finished MaxEnt training in 0.94 seconds
No file name specified for saving the model, using default name 'TextClassifier.ml
model'.
Trained model successfully saved at /Users/vardhanagrawal/Downloads/TextClassifier
.mlmodel.
----------------------------------
Number of examples: 402
Number of classes: 2
Accuracy: 88.06%

******CONFUSION MATRIX******
----------------------------------
True\Pred negative  positive
negative  178       23
positive  25        176

******PRECISION RECALL******
----------------------------------
Class    Precision(%)   Recall(%)
negative 87.68          88.56
positive 88.44          87.56
```

And—your model should be saved where you specified. You can refer to the console message to figure the file path. For instance, my trained model is saved at `/Users/vardhanagrawal/Downloads/TextClassifier.mlmodel`. Now, you can drag that model into any iOS app and use it similar to the image processing model you created earlier in the book. For more information on different ways to use this model, check out Apple's documentation.



Text_Classifier

Type	Text Classifier
Size	939 KB
Availability	iOS 12.0+ macOS 10.14+ tvOS 12.0+ watchOS 5.0+
Model Class	Text_Classifier Model class can be viewed by adding model to a workspace and target.

Metadata

Description
This model analyzes the sentiment of a given string.

Author
Vardhan Agrawal

License
BSD-2

Version
1.5.1

Additional Metadata

com.apple.createml.version
11.0.1

Class Labels 2

Labels
negative
positive

Figure 5-6: The trained model

Using your Model

Though this app doesn't have a sample project, let's look at how the API calls would work for the finished Text Classifier. When you need to use your model, you must first create an instance of the model (through its wrapper class we discussed earlier):

```swift
let model = TextClassifier()
```

All that's left to do is get the result from the model:

```swift
guard let result = try?
    model.prediction(text: "I'm happy today!") else {
    fatalError("An error occured.")
}
```

Since the `prediction(:)` method throws errors, we'll use the `try?` keyword and use a `guard let` statement to unwrap the optional value. By the end, `result` will be equal to the result from the model with the passed in String.

Conclusion

In this chapter, you learned about linguistics in machine learning, including natural language processing, text classification, and word tagging. In addition to this, you learned more about sentiment classifiers, which help identify the mood or intention behind what someone is saying.

After reading this chapter, you were able to create useful, fully trained models which you can use in your apps. You also walked away with a complete understanding of how computer science and linguistics are connected and how you can use the power of natural language processing in your day-to-day workflow.

Chapter 6
Sound Classification Models

In the previous chapter, you learned about natural language processing and how it applies to your day-to-day life. You also learned about what happens under-the-hood with detailed explanations about each of the algorithms. Finally, you used Swift Playgrounds to create your own text classification model. With all of this knowledge about natural language processing, you should be well-versed with basic NLP-related tasks.

In this chapter, you'll learn about sound classification in detail. With Apple's latest update, Xcode comes bundled with Create ML and a new array of audio processing frameworks. In the beginning of the chapter, you'll learn how to find, record, and source audio clips for sound classification, and later, you'll learn how to organize them correctly.

In the second half of the chapter, you'll use Create ML to train your own sound classifier using the IRMAS audio dataset and learn how to use it in an iOS app. Throughout the way, you'll learn how to use the SoundAnalysis framework and an AVAudioEngine to utilize the built-in hardware microphone, and run the audio clips through your trained machine learning model.

6-1 Overview and Gathering Sounds

Over the preceding chapters, you've learned about different ways to apply machine learning technology: including image and sentiment classification. In the previous chapter, you learned about natural language processing algorithms and how each step works under the hood. At the end, you made your own sentiment classification model using Swift Playgrounds and Create ML.

Sound classification, another excellent application of machine learning technology, allows a model to "hear" the world around it. In this section, you'll learn about sound classification and how to record your own sounds for use in a sound classifier. You'll also learn about effective ways to find pre-recorded sounds, in case you don't want to use your own for model training.

About Sound Classification

Think about the sounds you hear on a daily basis. Whether it's a dog barking or a car passing by, sounds are difficult to quantify. In other words, it's hard to describe sounds, and they appear to be even less structured than natural language. Considering this, sound classification is an impressive feat in machine learning, since it requires a computer to make sense of seemingly random sound inputs.

Classification Methods

There are several ways to classify sounds, including identifying the source, type, or characteristics of the sound. Since sounds are so difficult to define — unlike images, words, or numbers — the best sound classifiers would use the most general approach, rather than narrowing in on specific details of the sound wave.

Source Classification

One way to think about sound classification is source classification. This simply identifies what type of person, animal, or instrument is making the sound. For example, if you wanted to distinguish between a bird's chirp and a dog's bark, you could use source classification. This can be used both in noisy environments, where a model is picking apart a certain sound from the rest; or, it can be used where only one voice can be heard.

Type Classification

Alternatively, if there was no particular sound you were targeting, you could identify where you (or, the microphone) is. For instance, the sounds you would hear at the subway station would be different from those you would hear on the freeway. In both of these cases, there's no particular noise the model is identifying, but instead, it's making a generalization based on all of the sounds put together.

Problems and Techniques

By now, you may be wondering how sound-based machine learning models work if sounds are so seemingly random. Their trick is to hone in on specific features of sounds and then look for those features in all of their input data. In other words, they look for what sounds are unique to certain clips and don't appear in other clips.

Noisy Clips

When we talk about "noise" in a sound classification sense, we mean sounds that are irrelevant in identifying an audio clip. For example, if you're trying to identify a wolf's howl in the wild, the shuffling of leaves and the sound of raindrops would be considered noise, since they're irrelevant in determining whether a wolf is present. Audio clips are especially prone to noise because smartphone microphones aren't particularly good at avoiding background sounds.

Filtering and Normalization

In response to the unwanted noise, one may choose to filter or normalize the audio clip. This may include removing white noise, static, or other disturbances which may hinder a machine learning model's ability to classify sounds. However, it's important to be careful when doing this. Since real world recordings aren't filtered, if your model is only trained on filtered data, it may not perform well when tested with unfiltered data.

Support Vector Machines

As you learned in chapter 1, support vector machines, or SVMs, use training data to draw "borders" that separate or distinguish between different classes. A sound classification model works in a similar way. After it gathers the appropriate features from the waveforms, it arranges them to find similarities and differences that it can use to distinguish sounds.

Recording Sounds Yourself

A great way to get your audio clips for your sound classification model is to record them yourself. Remember, though, that the data you collect quite literally defines the quality of your machine learning model. Here are some tips to record high quality audio clips which you can use to train your sound classifier.

1. Don't Over-Filter

A mistake that many beginners make is trying to stay away from background noises, or worse, trying to simulate a real world environment instead of recording the actual sound. The issue with doing this is that your model is more likely to underperform when tested with real data. Avoiding this mistake can greatly increase the accuracy of your model.

2. Use System Microphone

The microphone on your mobile device might not have the best quality, but it's the only way your users have to record audio. Then, if you're developing a mobile app, your best bet would be to train your classifier using data from the built-in smartphone microphone as opposed to using professional equipment or a different mic.

3. Record Edge Cases

In case you aren't familiar, "edge cases" refer to data points which are close to the boundaries and are likely to "trick" the algorithm. In the sound classification world, an edge case might be a dog's whimper which sounds like a cat. As you record, try your best to seek out some of these confusing sounds to improve your model's performance when dealing with these cases.

Sourcing Sounds from Datasets

Obviously, if the sound data you're looking for is available online, it'll likely be easier not to record your own. In fact, you probably won't need to record sounds yourself unless you're dealing with a super-niche topic such as specific engine noises or extinct animal species.

Datasets

With user-produced videos flooding the internet, there is no shortage of sounds to choose from. People have gone through these and created large datasets for you to use in training your models. Below are a few resources which will help you source audio files for use in a sound classification machine learning model.

AudioSet (https://research.google.com/audioset/)

YouTube has millions of videos published by users, and Google has gone through the paces of labelling 10-second audio clips from these videos for a massive dataset of 2 million audio files. You can use this for almost anything, since there's a wide range of labelled data and in large volumes.

UrbanSound8K (https://urbansounddataset.weebly.com/urbansound8k.html)

As the name suggests, the UrbanSound8K dataset has around 8,000 different audio clips of 10 types of sounds (classes). They include common city noises, such as sirens, street music, and dogs barking. Since the files are meant to represent the sounds an average person would hear while going about their day, models trained with UrbanSound8K are likely to perform well in similar situations.

MIVIA (https://mivia.unisa.it/datasets/audio-analysis/mivia-audio-events/)

The MIVIA dataset brings out the "dark side" in machine learning developers and contains around 6,000 clips of people screaming, guns being fired, and glass breaking. It's great for building security applications or other software which requires detection of criminal activity.

Data Organization

Finally, let's review how to store and organize your training data and get a preview of how to handle your testing data, which is something new covered in this chapter. Depending on where you get your data, it may already be in the required format, or you may need to adjust it.

Training

Similar to the image and text classifiers you've seen, you'll need to create folders to organize your data. Just like before, you should create the same number of folders as labels and store all of the appropriate audio files in their respective folders. This way, you can drag all of them into the Create ML app, which will recognize this structure and correspond the labels to the training data.

Testing

In this chapter, we'll be focusing more on testing than before, so make sure you have extracted some testing data. Many datasets will have a random testing folder, but if not, you'll need to take out some of your training data and put it in an unlabeled "testing" folder. Create ML will then run all of this data through your trained model.

6-2 Training a Sound Classifier

In the previous section, you learned about sound classification and how it works at a glance. You also got some tips on how to record your own sounds effectively with a classification-oriented mindset. Further, you learned about datasets where you can get millions of pre-labeled sound files at no cost. At the end, you learned about organizing your data for easy model training within Create ML.

In this section, you'll train your sound classifier. At the beginning, you'll create a Create ML project to experiment with different sounds. Then, you'll get familiar with your data. After training your machine learning model, we'll focus on testing in various ways to explore the powerful testing features in the Create ML app.

Training using Create ML

In previous chapters, you've used Create ML for both image processing and text classification. The process for creating a sound classification model is similar in many ways, but its applications are vastly different. Let's train a sound classification model using the Create ML app.

Create a Project

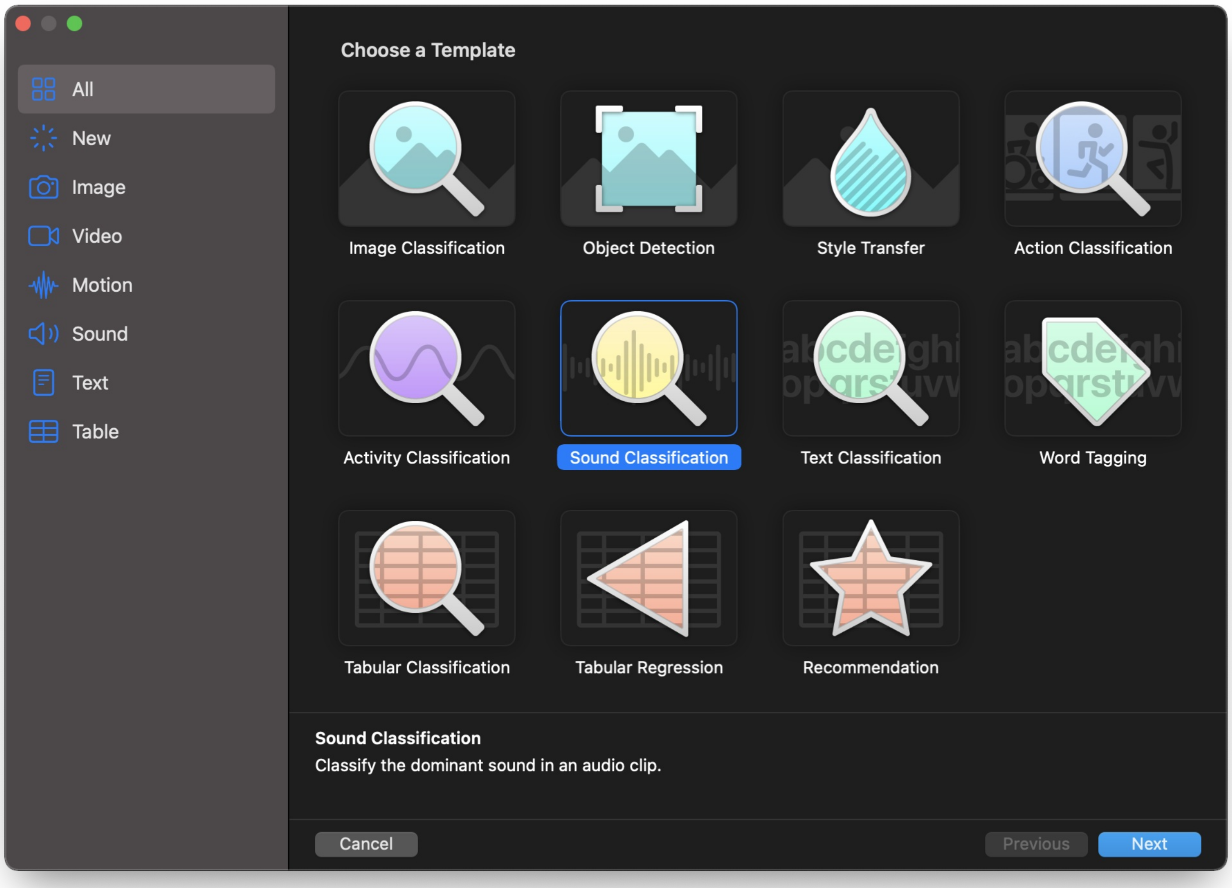
The first step of training a Create ML model is to start a new project. It's worth noting that a project can have multiple models within it, which makes it perfect for testing different datasets, specifications, and training sets.

Open Create ML

First , open your beloved Create ML app. In case you haven't followed the previous chapters, Create ML comes bundled with Xcode and can be found in your Applications folder or using Spotlight Search via the **Command + Space** keyboard shortcut.

Choose Model

In Create ML, choose *File* -> *New Project*. You have many options when it comes to the type of model you choose, including an image classifier, word tagger, and table-based model. For sound, however, there is only sound classification at this point in time. Choose **Sound Classification** and click **Next**.



Sound Classification
Classify the dominant sound in an audio clip.

Figure 6-1: Choosing a Create ML Template

Name and Save Project

Create ML will then prompt you to name your project and choose a location in your file system to save it. Obviously, you may name it whatever you wish, but bear in mind that the description and author will show up within Xcode if you choose to share or publish you model.

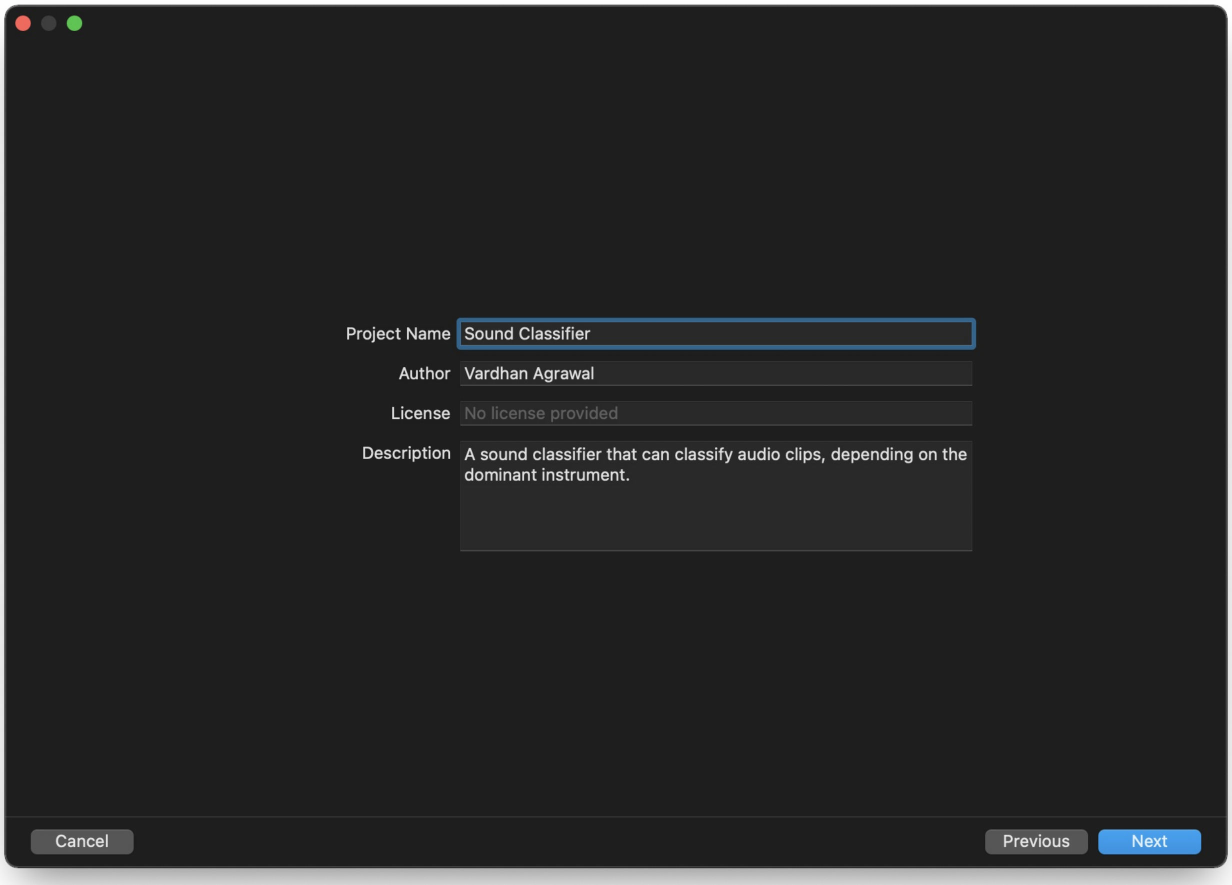

Project Name Sound Classifier
Author Vardhan Agrawal
License No license provided
Description A sound classifier that can classify audio clips, depending on the dominant instrument.

Figure 6-2: Naming Your Project

Gather and Prepare Data

Before you train your sound classification model, you'll need to gather data. For the purposes of this book, I'll be using the IRMAS dataset (https://www.upf.edu/web/mtg/irmas). This dataset labels the dominant instrument playing in each audio clip.

Download Training Data

IRMAS gives you many files to download, all of which 11 GB in total. Fortunately, you don't need all of them to continue. First, you'll need to download the training data, which is named `IRMAS-TrainingData.zip` under the downloads tab on the IRMAS website. Or you can use this direct link (https://zenodo.org/record/1290750#.XuW03GozadZ).

Label Training Data

When you open the unzipped file, you'll find 11 folders, each containing its corresponding audio clips. It should be structured in this way:

- `IRMAS-TrainingData`
    - `README.txt`
    - `cel`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
    - `cla`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
    - `flu`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`

However, when you train your model, you don't want output labels like "cel" and "cla," but instead, something more descriptive, like "cello" and "clarinet." You may use the key below to rename the folders:

Original	New
“cel”	“cello”
“cla”	“clarinet”
“flu”	“flute”
“gac”	“acoustic guitar”
“gel”	“electric guitar”
“org”	“organ”
“pia”	“piano”
“sax”	“saxophone”
“tru”	“trumpet”
“vio”	“violin”
“voi”	“human vocalist”

And, while you don't *need* to rename these files, it might be useful to do so for easier readability in the next section of the chapter, where you'll learn how to create an app which utilizes the model you created.

Remove README File

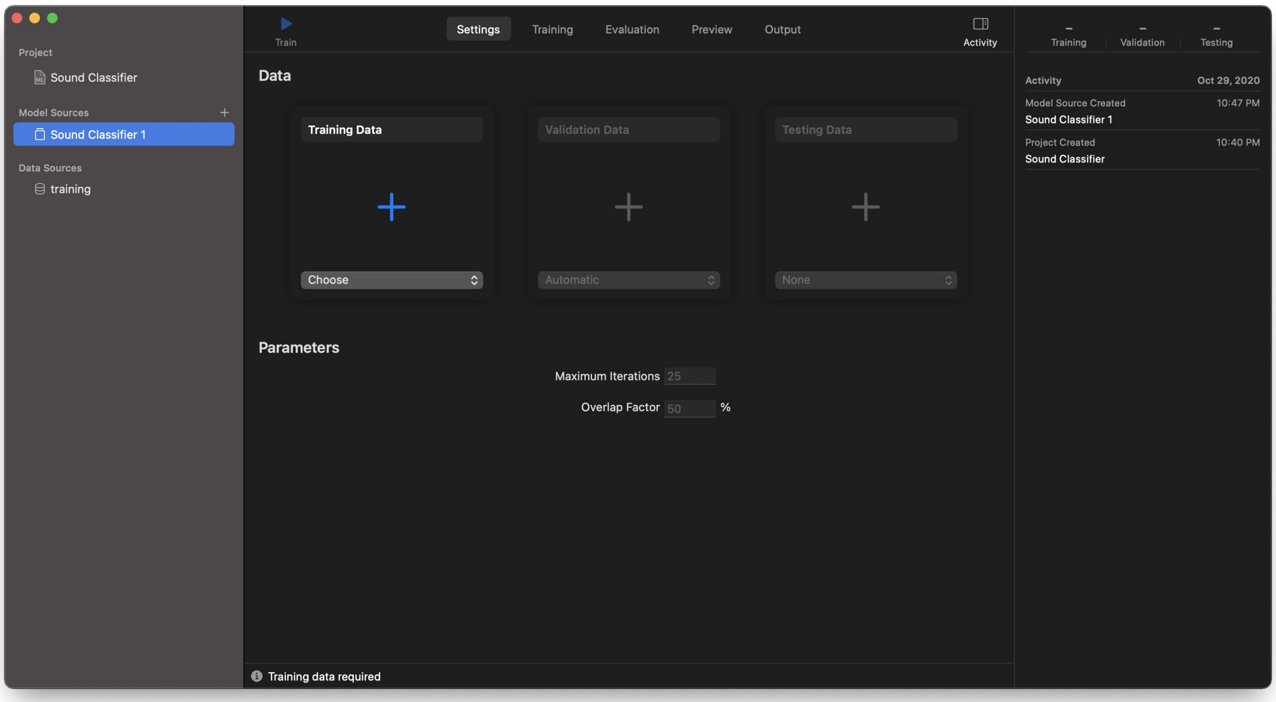
Finally, remove the `README.txt` file in the downloaded folder. Without doing this, Create ML will think it's a part of the training data and will not proceed because it has a different file type than the audio clips.

Download Testing Data

Then, you'll need to download the testing data. Since we're not building a professional-scale model right now, you can just download the first part, named `IRMAS-TestingData-Part1.zip`. Again, you can find the testing data in this direct link (https://zenodo.org/record/1290750#.XuW03GozadZ). Since the testing data isn't pre-ordered for us, we won't use it in the auto-testing section of Core ML; however, we'll use it for manual spot-testing later on.

Train your Model

Now, after your mini-detour from Create ML, it's time to get back on track to training your sound classification model. Open your Create ML project you created earlier. I suggest it is a good time to plug into a power source, since training might take a long time and drain a significant portion of your battery.



Maximum Iterations 25

Overlap Factor 50 %

Training data required

Activity Oct 29, 2020

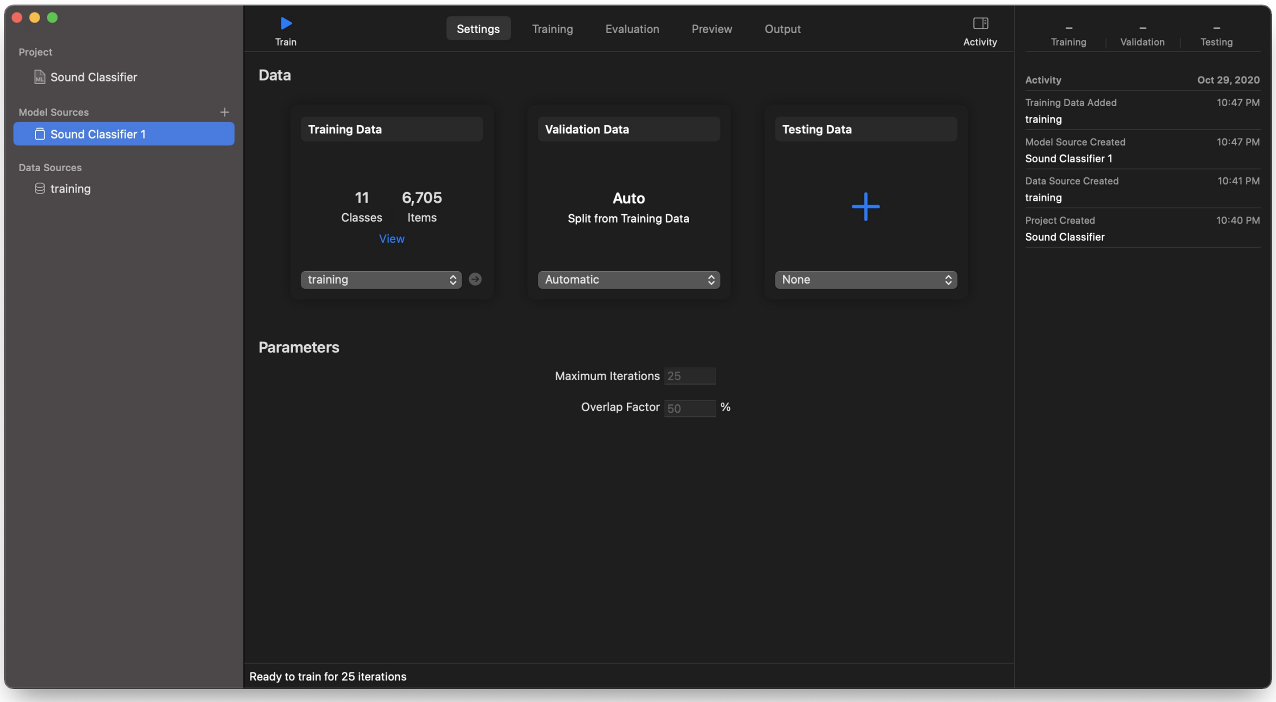
Model Source Created 10:47 PM
Sound Classifier 1

Project Created 10:40 PM
Sound Classifier

Figure 6-3: Blank Create ML Project

Drag in Training Data

Just as with image processing models, you can drag your training data, which is the *IRMAS-TrainingData* folder, directly into the *Training Data* box in Create ML. The tool then automatically loads the training data and you should see a screen similar to figure 6-4. If you have an older machine or you're not willing to wait that long, I suggest removing some of the folders in the training data or thin down the number of files in each folder for faster model training.



Data

Training Data: 11 Classes, 6,705 Items, View — training

Validation Data: Auto, Split from Training Data — Automatic

Testing Data: None

Parameters

Maximum Iterations 25

Overlap Factor 50 %

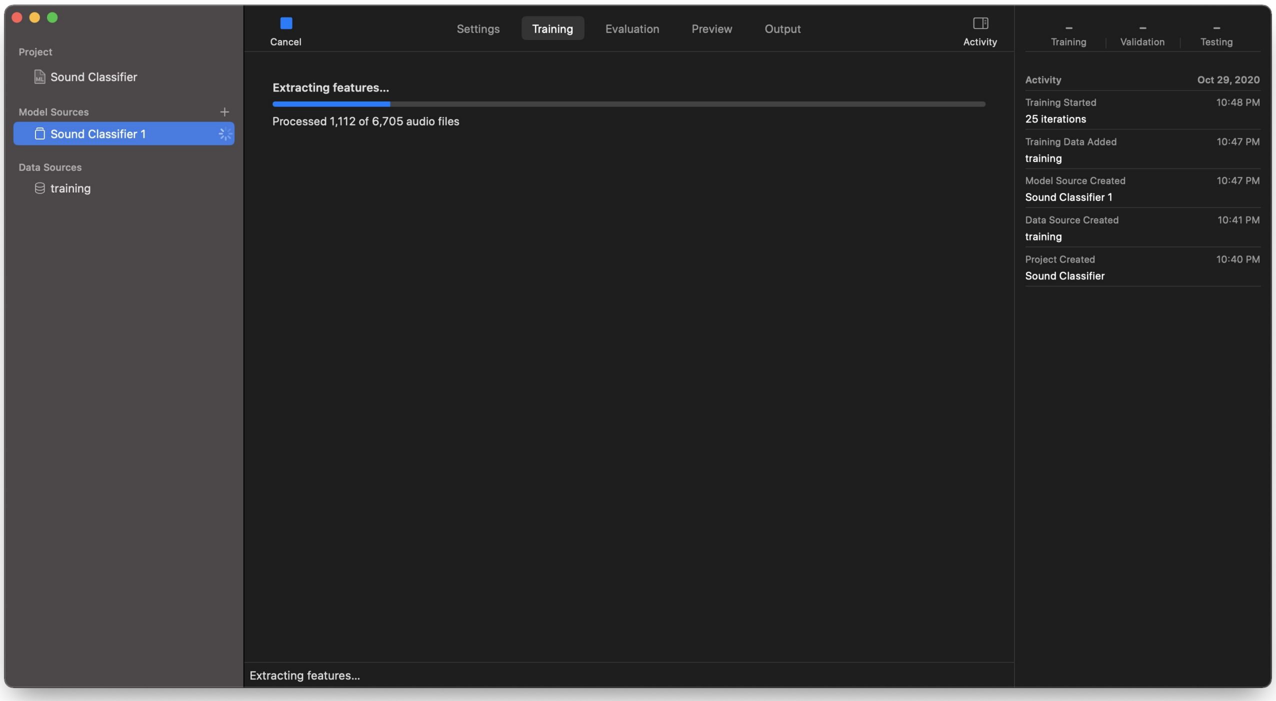
Ready to train for 25 iterations

Activity	Oct 29, 2020
Training Data Added training	10:47 PM
Model Source Created Sound Classifier 1	10:47 PM
Data Source Created training	10:41 PM
Project Created Sound Classifier	10:40 PM

Figure 6-4: Import Audio Files for Training

Next, click the *Play* button, towards the left side of the top toolbar, to start the training. Go and make yourself some popcorn. It may take a while to train your sound classifier, especially if you're using the full dataset.

The model will first extract the features, as you learned before:



Extracting features...

Processed 1,112 of 6,705 audio files

Activity	Oct 29, 2020
Training Started 25 iterations	10:48 PM
Training Data Added training	10:47 PM
Model Source Created Sound Classifier 1	10:47 PM
Data Source Created training	10:41 PM
Project Created Sound Classifier	10:40 PM

Extracting features...

Figure 6-5: Processing Audio Files

Then, it will train the model using a SVM architecture.

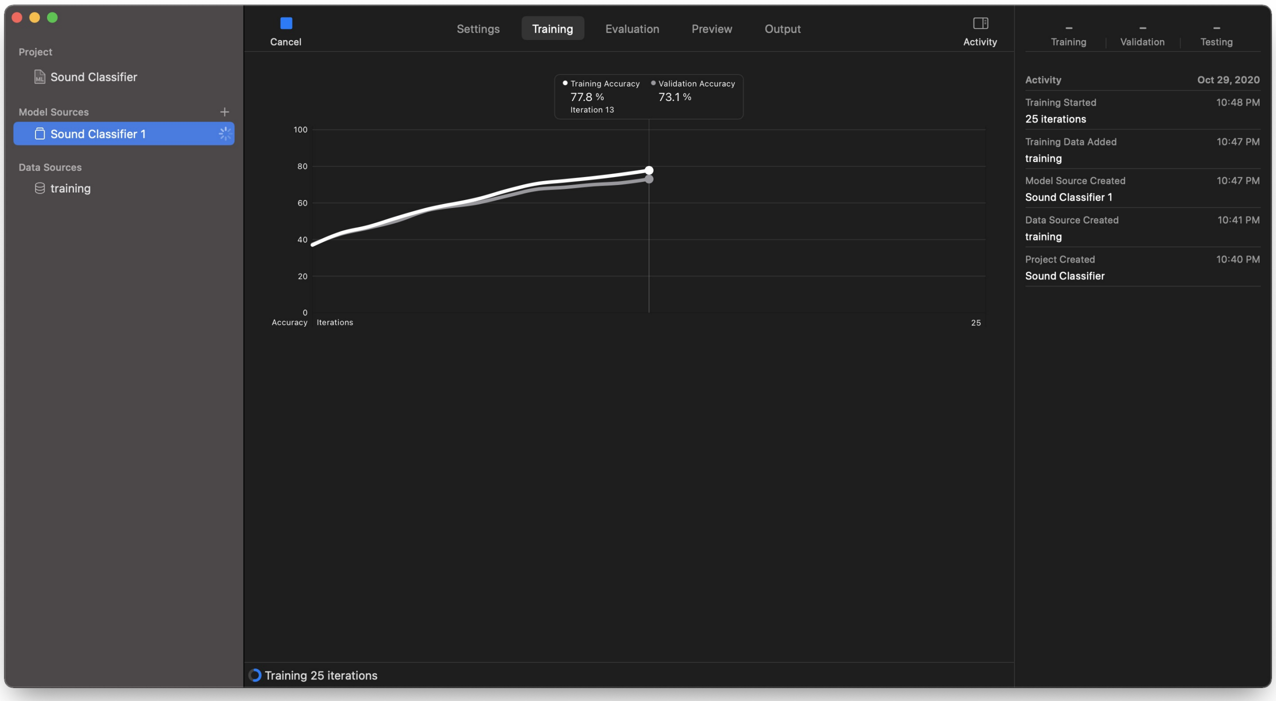

Cancel
Settings
Training
Evaluation
Preview
Output
Activity
Project
Sound Classifier
Model Sources
Sound Classifier 1
Data Sources
training
Training Accuracy 77.8 %
Validation Accuracy 73.1 %
Iteration 13
Accuracy
Iterations
Training 25 iterations
Activity Oct 29, 2020
Training Started 10:48 PM 25 iterations
Training Data Added 10:47 PM training
Model Source Created 10:47 PM Sound Classifier 1
Data Source Created 10:41 PM training
Project Created 10:40 PM Sound Classifier

Figure 6-6: Training the Model

Testing your Model

Testing models after training them is as important as training them. After all, if your model isn't able to correctly classify sounds, it defeats the purpose of its existence. The dataset that we are using in this chapter has, thankfully, provided us with a ton of testing data to work with.

Testing Data

Like the training data, the testing data isn't correctly formatted for Create ML when you download it. Therefore, you'll need to make adjustments to the file hierarchy before you can begin. When you first download the testing data, it'll be structured like this:

- `IRMAS-TestingData-Part1`
    - `README.txt`
    - `Part 1`

        - `(xxx)-xxx.wav`
        - `(xxx)-xxx.txt`

Notice that we have a README file and a folder. Within the folder, there are **both WAV and TXT** files. Obviously, we don't need the text file for Create ML, so we'll need to handle the data in one of two ways below:

1. Automated Testing

With automated testing, Create ML will automatically run your data through the trained model and track the accuracy quantitatively. To take advantage of this feature, you'll need to arrange your testing data into folders in the same way as the training data. If you choose this route, arrange your test data you downloaded earlier like this:

- `IRMAS-TestingData-Part1`
    - `cello`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
    - `clarinet`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`
    - `flute`
        - `[xxx][xxx].wav`
        - `[xxx][xxx].wav`

You will notice that the files have been moved out of the `Part 1` folder and the `README.txt` file has been deleted. Using the text in each song's corresponding text file of the audio files, you'll need to sort your clips into the appropriate folders, which will likely take a lot of time; however, it's up to you whether or not it worths. Then, go to the **Testing** tab in the menu bar and drag your folder with sorted audio files in for your results.

2. Manual Spot Testing

The "easier" way is to remove all of the text files, including the `README.txt` file, and use the **Output** tab to drag your files in. This way, you can playback the clips to see the model's prediction for every few seconds of the soundtrack. While this method won't give you numerical data, it will help you get a general sense of how your model is performing.

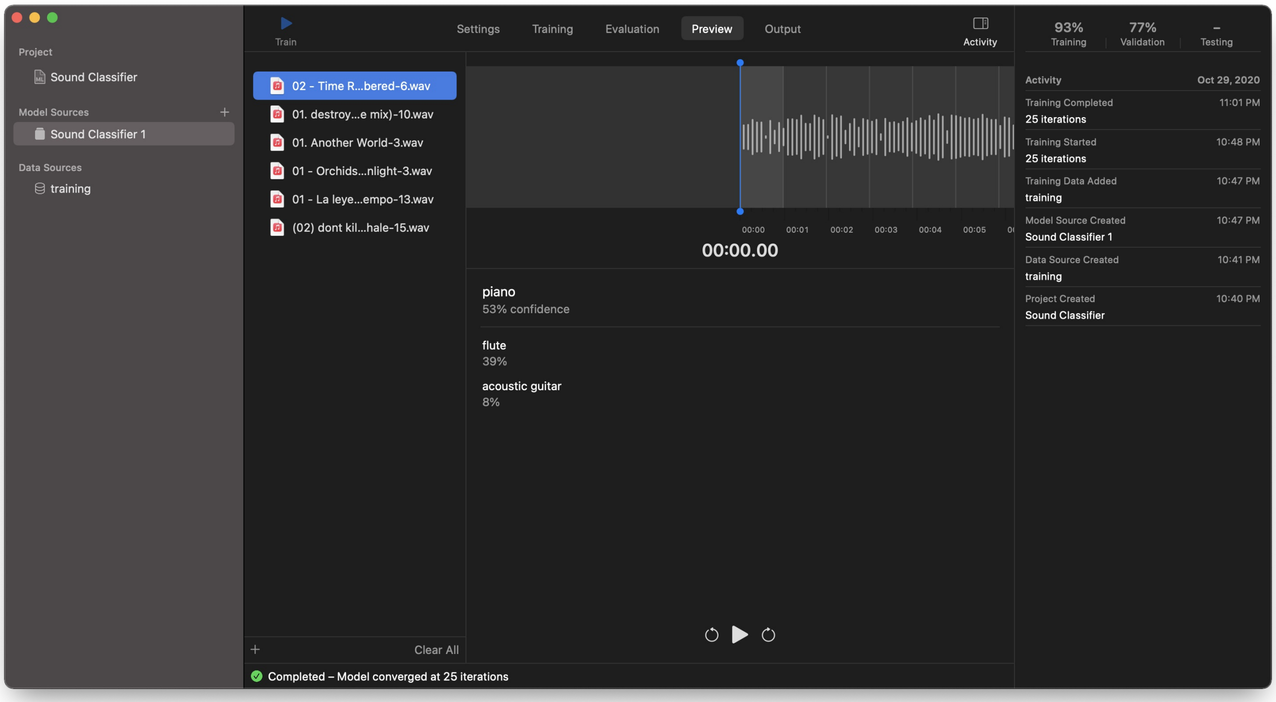


Figure 6-7: Testing Specific Audio Files

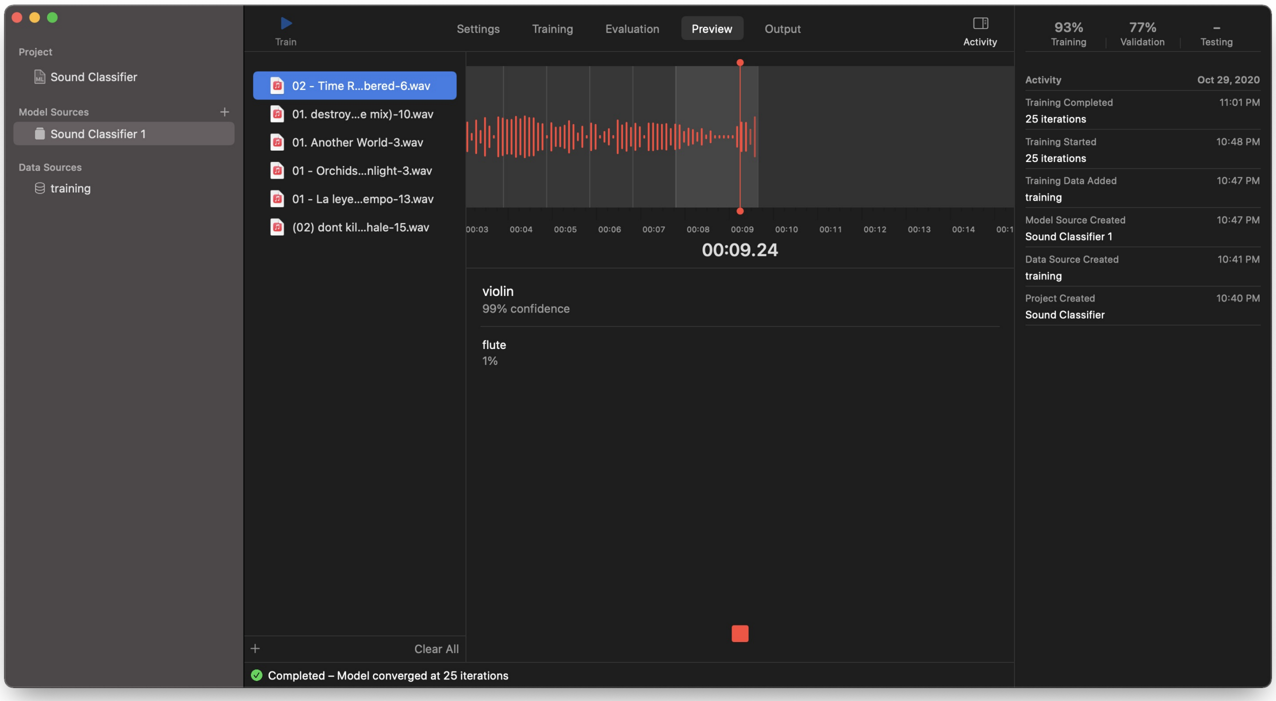
Recording and Exporting

Now, you have a fully trained model, and you're ready to use it in a sound classification app. However, before you do that, let's take a look at one last way to test your data: using your own voice.

Live Recording

One of my favorite features of Create ML is the ability to see how your model performs in real-time without having to create an Xcode project to test it out. To use this excellent feature, tap the + button at the bottom of the **Output** tab. Then, click **Record**

Microphone. This way, you'll be able to talk, sing, or play an instrument and watch your hard work pay off!



02 - Time R...bered-6.wav
01. destroy...e mix)-10.wav
01. Another World-3.wav
01 - Orchids...nlight-3.wav
01 - La leye...empo-13.wav
(02) dont kil...hale-15.wav

00:09.24

violin
99% confidence

flute
1%

Activity	Oct 29, 2020
Training Completed 25 iterations	11:01 PM
Training Started 25 iterations	10:48 PM
Training Data Added training	10:47 PM
Model Source Created Sound Classifier 1	10:47 PM
Data Source Created training	10:41 PM
Project Created Sound Classifier	10:40 PM

Completed – Model converged at 25 iterations

Figure 6-8: Testing a Live Recording

Export your Model

As always, getting your finished model from Create ML is seamless. All you need to do is drag the Core ML file icon under the **Output** tab into your desktop or Xcode. And, that's a wrap! You now have your completed Core ML model, and it's ready for use in the final section of this chapter.

Rename your Model

By default, Create ML names your model `SoundClassifier1.mlmodel`. To make things simpler and more readable, let's rename the file to `SoundClassifier.mlmodel`. You can do this by holding down **Control** and clicking on the file; then, select rename and type the new name. It's a subtle difference, but the simpler name makes it a whole lot easier to use in your app.

6-3 Implementing the Model

At the beginning of the chapter, you learned about sound classification and how to source sounds for model training appropriately. You also learned about some excellent resources which can help you get high-quality audio clips for your models. Later, you used the IRMAS database to train a sound classification model which can detect the dominant instrument in a song. Finally, you tested your model against testing data to measure its performance.

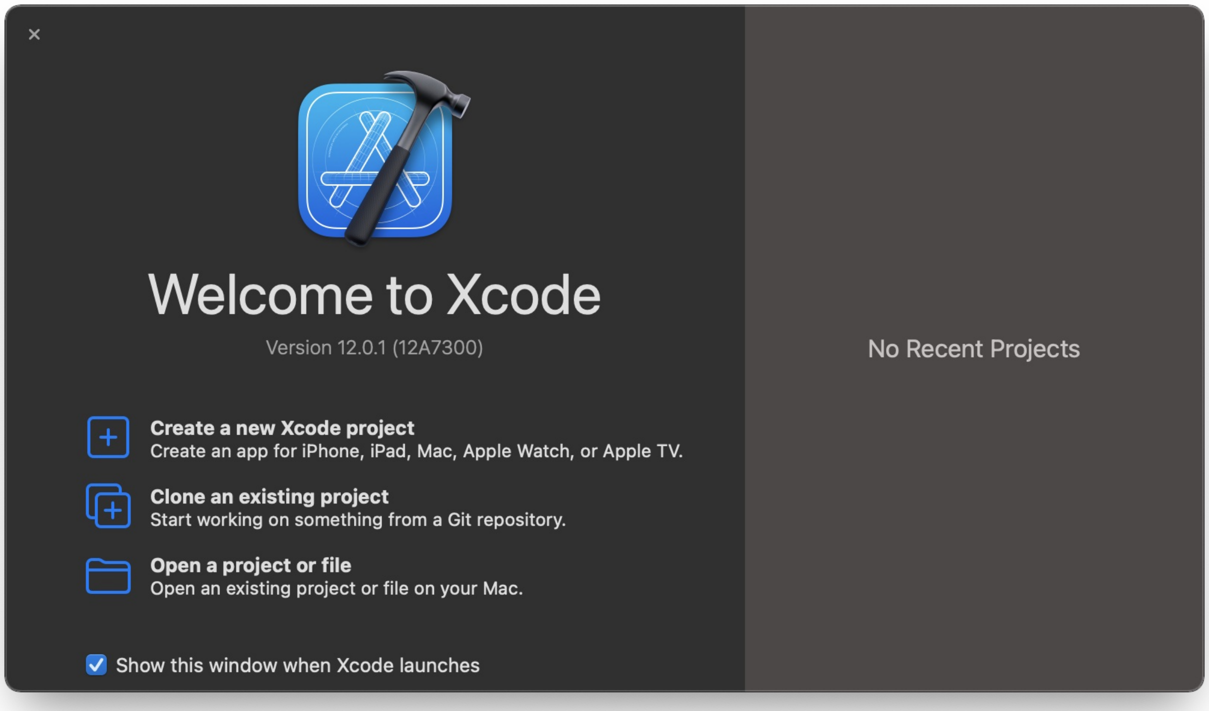
So far, you learned about sound classification and how to source sounds for model training appropriately. You also learned about some excellent resources which can help you get high-quality audio clips for your models. Later, you used the IRMAS database to train a sound classification model which can detect the dominant instrument in a song. Finally, you tested your model against testing data to measure its performance.

In this section, you'll learn how to create an iOS app which uses your sound classifier to record live audio, process it, and display results to your user. Through this process, you'll learn not only how sound classifiers work in real-world apps, but also how to ask for microphone permission, use the SoundAnalysis framework, and connect everything for a coherent, well-built application.

Building the Sound Classification App

The first step, as with any app, is to create an Xcode project. If you'll remember back to Chapter 2, you'll recall that we created an image classification app with a pre-built model. In this section, we're doing something similar, but with the audio classifier you built in the previous section.

You'll first need to name your app, select your language, and set your bundle identifier to begin. You're likely already familiar with this process, so feel free to skip a few steps. To begin, open Xcode.



Welcome to Xcode
Version 12.0.1 (12A7300)
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.
Clone an existing project
Start working on something from a Git repository.
Open a project or file
Open an existing project or file on your Mac.
Show this window when Xcode launches
No Recent Projects

Figure 6-9: Xcode Welcome Screen

As always, we'll be using a **Single View Application** as our template which allows us to write our code from scratch. If you prefer a different template, feel free to use it; however, certain steps may be different if you choose something else.

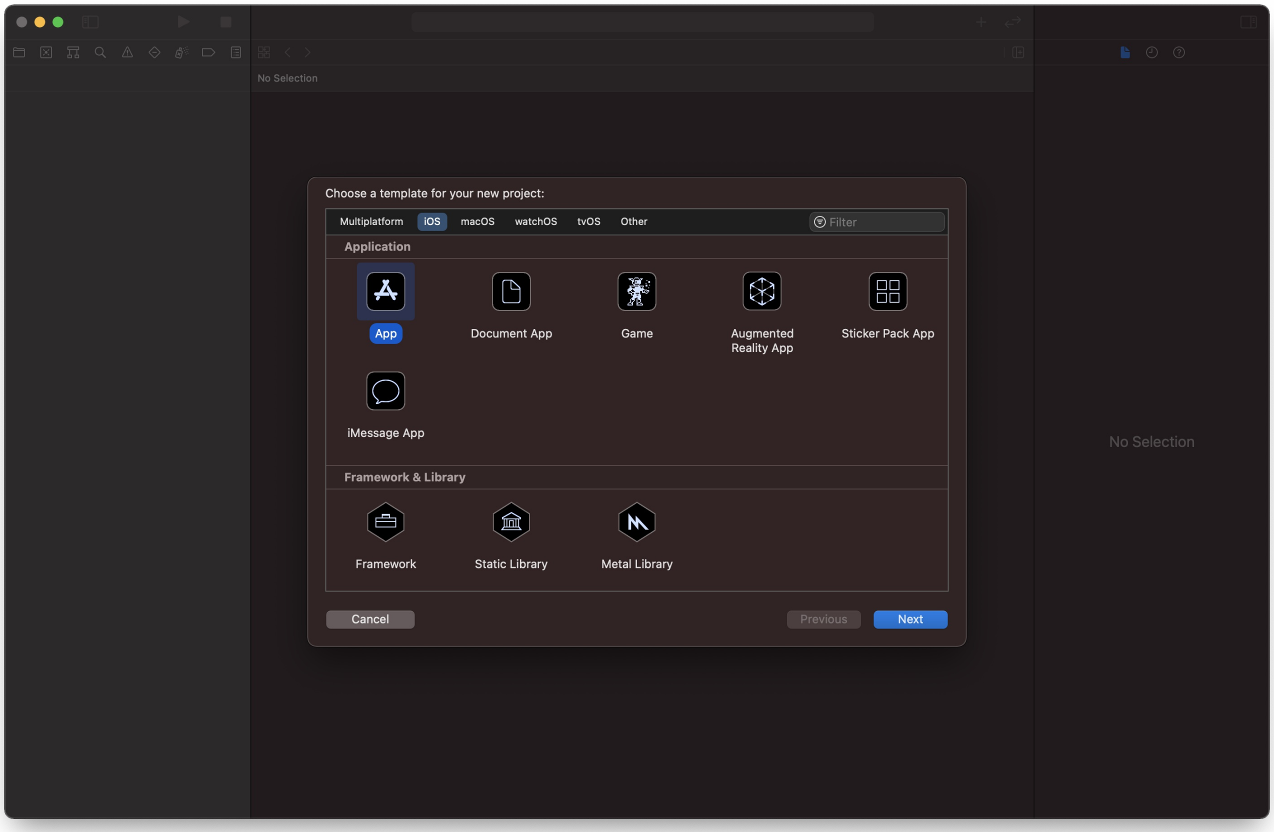
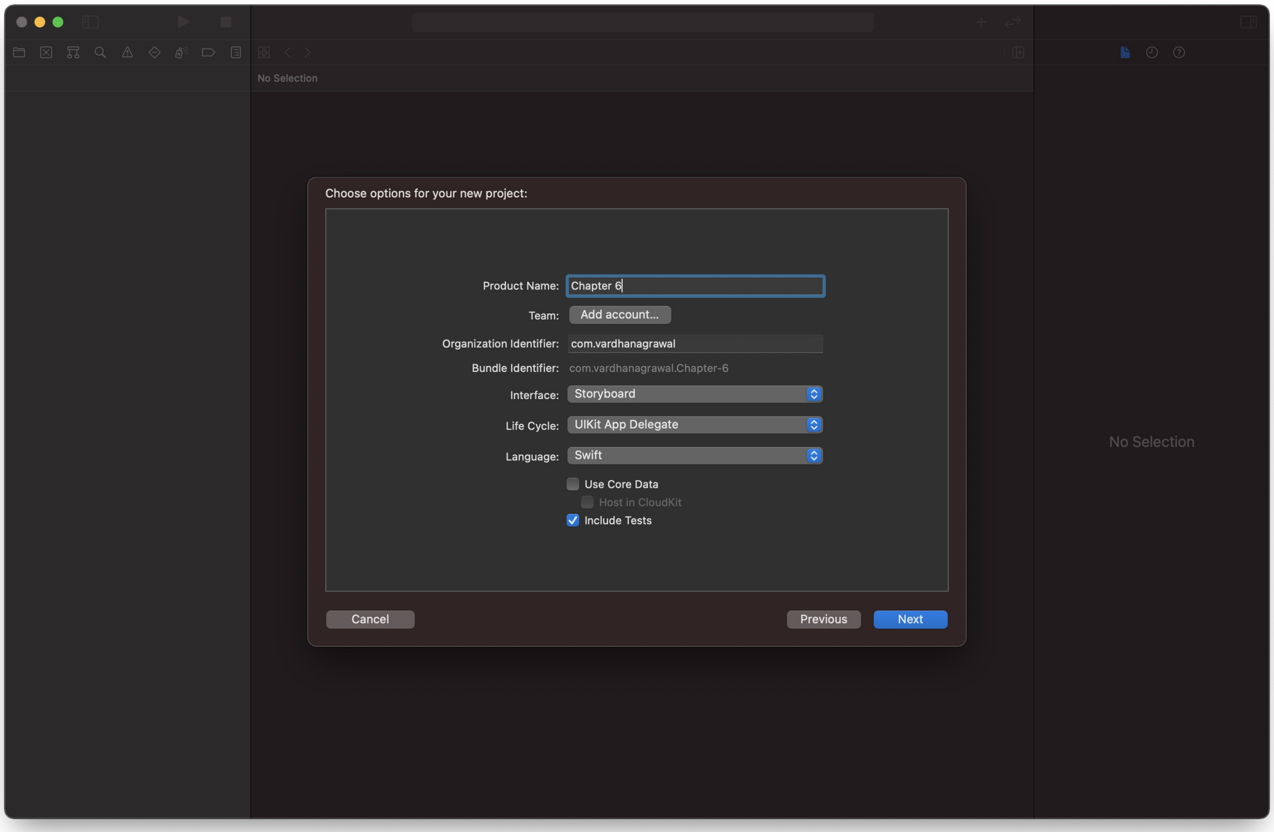


Figure 6-10: Choosing an Xcode Project Template

Xcode will then prompt you to name your project. This name doesn't matter much, so you can choose any convenient name: I'm using **Chapter 6** as my project name. Then select **Swift** for your **Language** and **Storyboard** for your **User Interface**.

It's worth noting that Swift UI is an option as well. At the time of this writing, it isn't widely used yet. So, we'll stick with traditional storyboards. Once you make the appropriate selections, your screen will look something like this:


No Selection

Choose options for your new project:

Product Name: Chapter 6

Team: Add account...

Organization Identifier: com.vardhanagrawal

Bundle Identifier: com.vardhanagrawal.Chapter-6

Interface: Storyboard

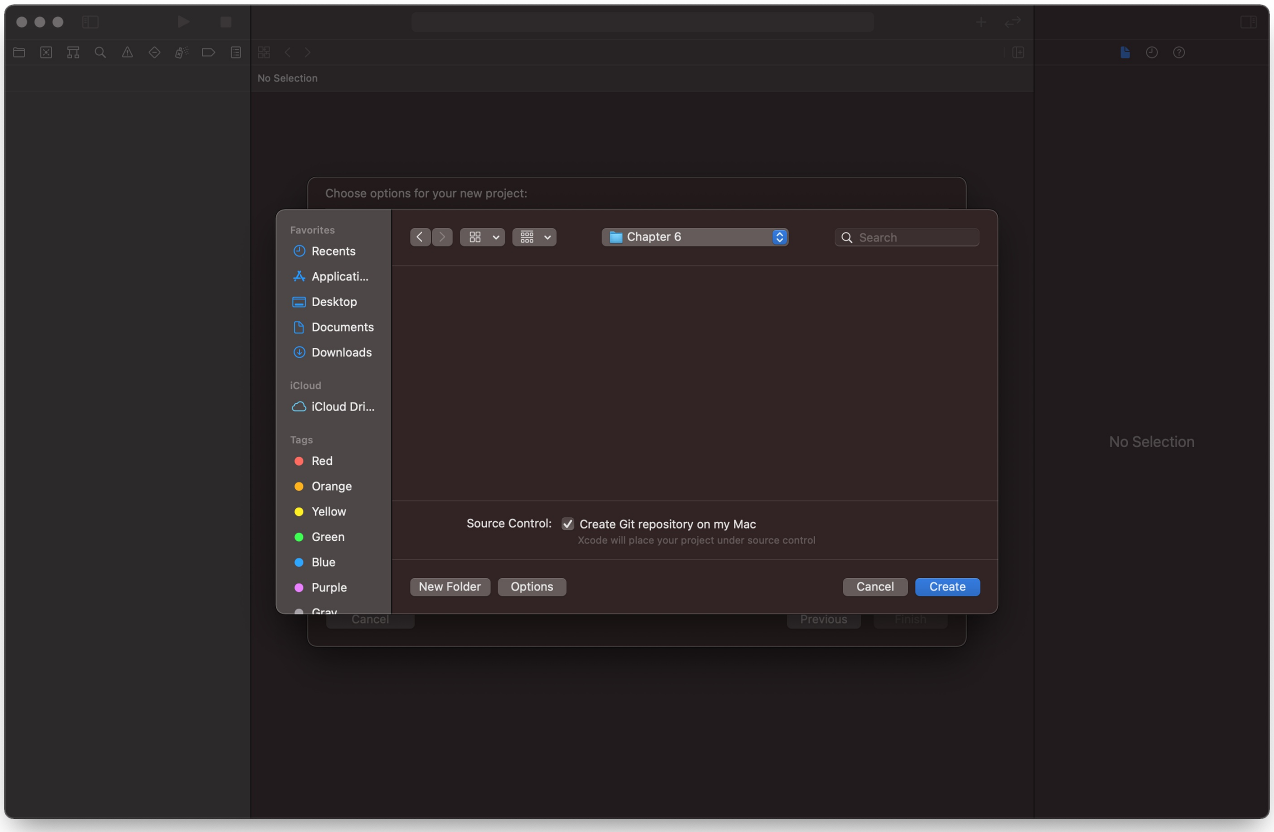
Life Cycle: UIKit App Delegate

Language: Swift

- [ ] Use Core Data
- [ ] Host in CloudKit
- [x] Include Tests

Figure 6-11: Naming Your Project

Finally, Xcode will ask you where you'd like to save your project. Again, you can put it wherever you like, but make sure you store it in a place which you'll remember and be able to access it easily later.



Source Control: ✓ Create Git repository on my Mac
Xcode will place your project under source control

Figure 6-12: Saving Your Project

User Interface and Model

Now, let's take care of the user interface and importing the model we created. Since much of the heavy lifting is done under-the-hood, you don't need to spend too much of your attention on the user interface if you don't want to. However, if you're a more creative person, feel free to make it as fancy as you'd like!

Designing your Interface

To design a user interface, you need to open the **Main.storyboard** file. And, if you haven't read it yet, Chapter 2 covers user interface design in more detail than we will in this chapter. Whatever you do, just make sure you have two things: a prediction label and a confidence label. For reference, my user interface looks like this:

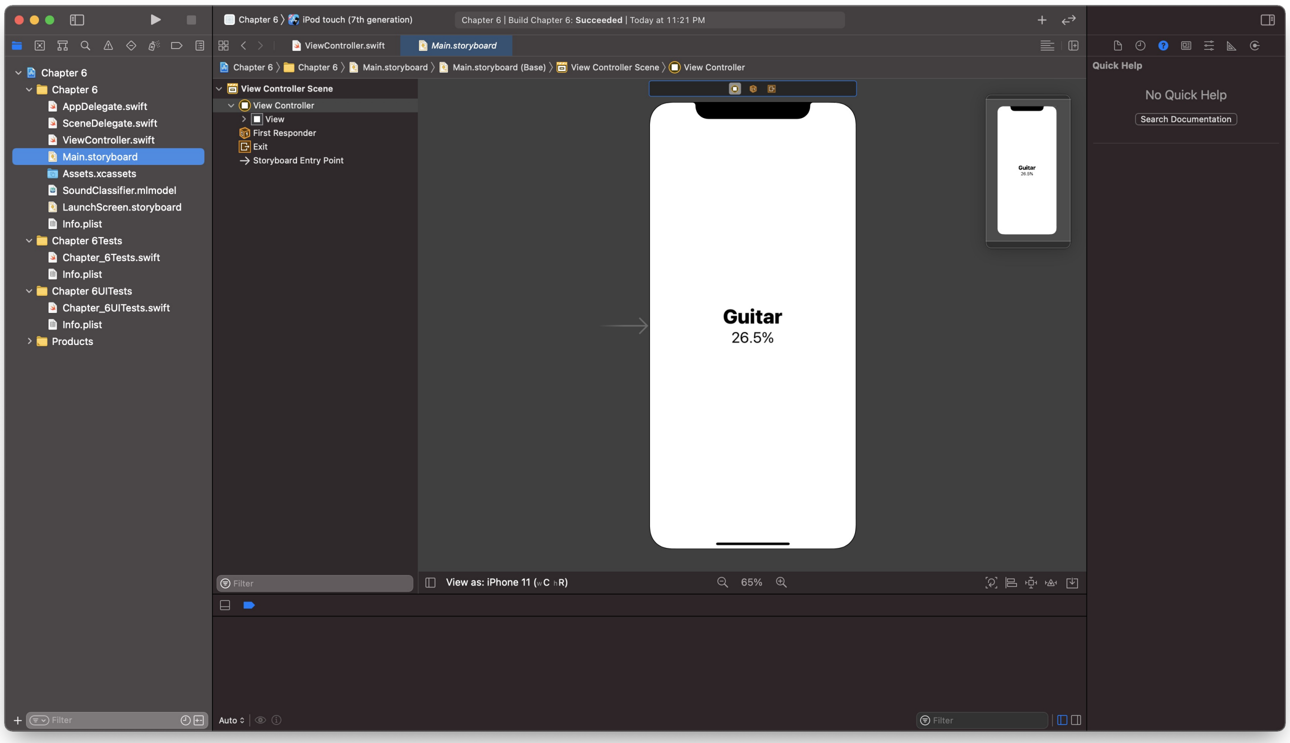
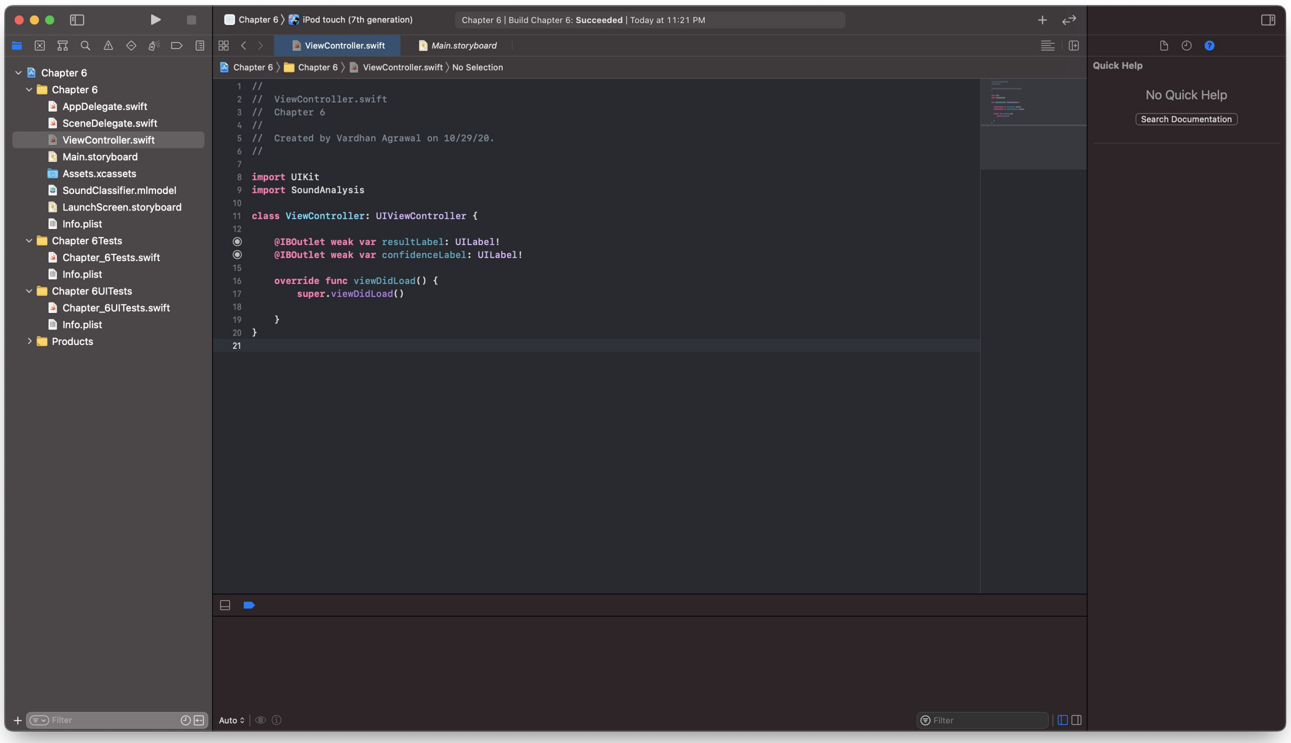


Figure 6-13: Building an Interface

Once you have your user interface, you need to connect it to your code. To do this, open your `ViewController.swift` file from the Project Navigator and type the following under the `ViewController` class declaration:

```
@IBOutlet weak var resultLabel: UILabel!
@IBOutlet weak var confidenceLabel: UILabel!
```

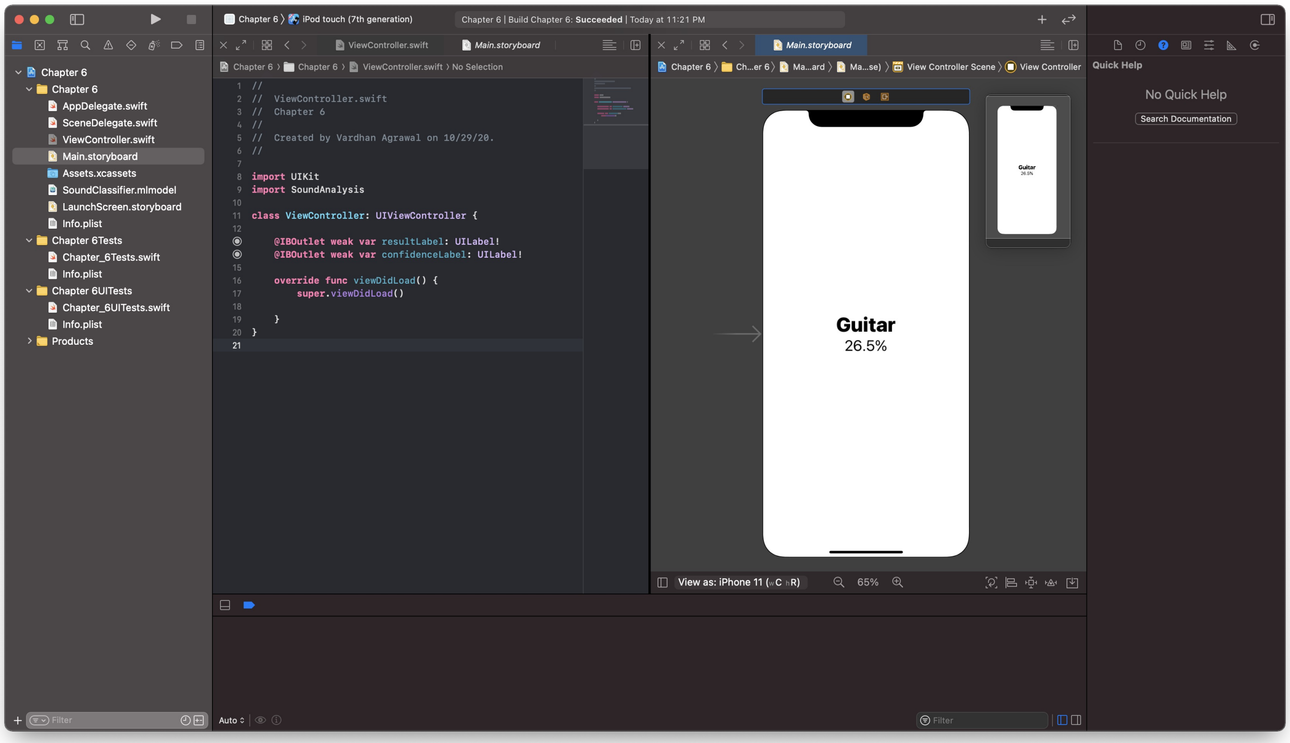
Then, use the **Add Editor** button (top right of current editor) to open the `Main.storyboard` file, where you can link each `@IBOutlet`.



```
//
//  ViewController.swift
//  Chapter 6
//
//  Created by Vardhan Agrawal on 10/29/20.
//

import UIKit
import SoundAnalysis

class ViewController: UIViewController {

    @IBOutlet weak var resultLabel: UILabel!
    @IBOutlet weak var confidenceLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

    }
}
```

Figure 6-14: Adding an Assistant Editor

At the left of the labels, where the line numbers usually sit, you'll find an un-filled circle. You can drag it to the corresponding label in the user interface on the right. After you do this, it'll look something like this with the `resultLabel` variable connecting to *Guitar* and the `confidenceLabel` variable to the percentage label.



```
//
//  ViewController.swift
//  Chapter 6
//
//  Created by Vardhan Agrawal on 10/29/20.
//

import UIKit
import SoundAnalysis

class ViewController: UIViewController {

    @IBOutlet weak var resultLabel: UILabel!
    @IBOutlet weak var confidenceLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

    }
}
```

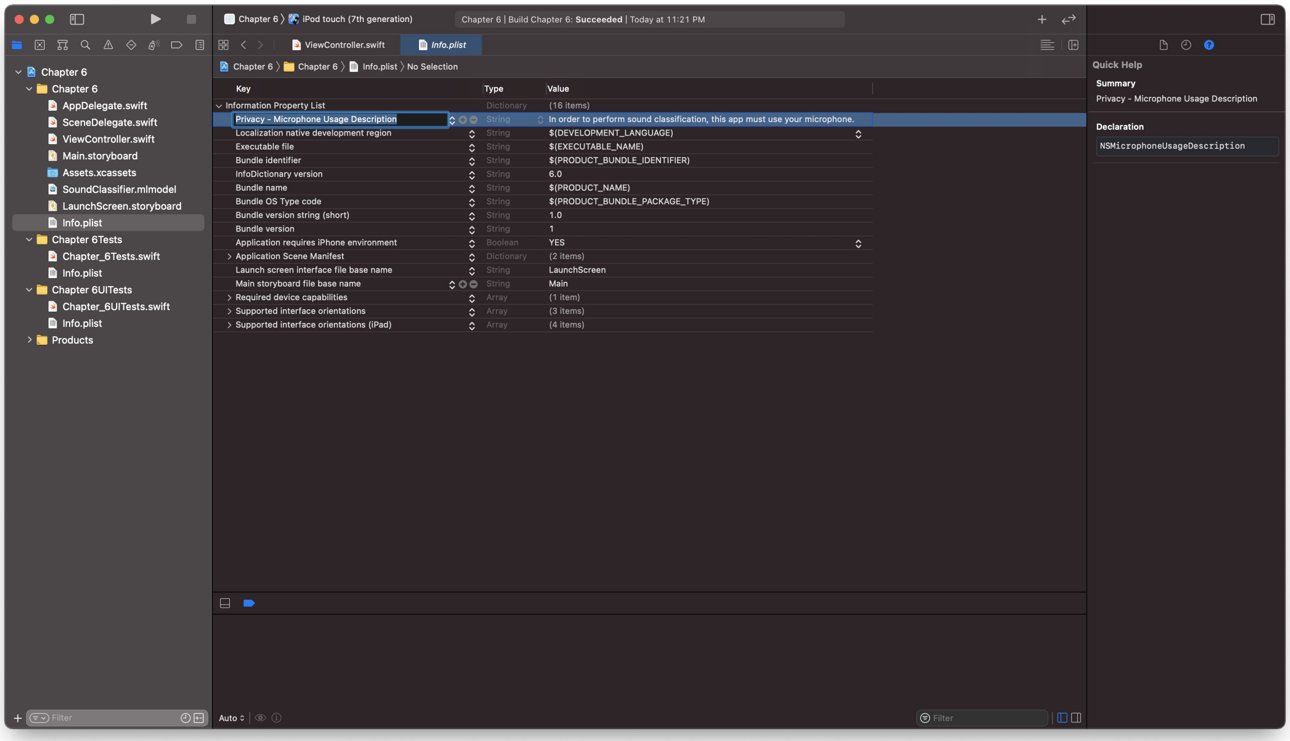
Figure 6-15: Connecting Interface Builder to Code

Setting Microphone Permissions and Importing the ML Model

Apple is highly particular about allowing apps to use sensors such as the microphone, camera, and location services. For this reason, we, as app developers, are required to explain why we need to use the built-in microphone and ask the user for permission.

The `Info.plist` file, or Information Property List, contains important metadata about the app you're creating. This is where you'll put your microphone usage description. You'll find this file in the Project Navigator, which is at the left pane of your Xcode window.

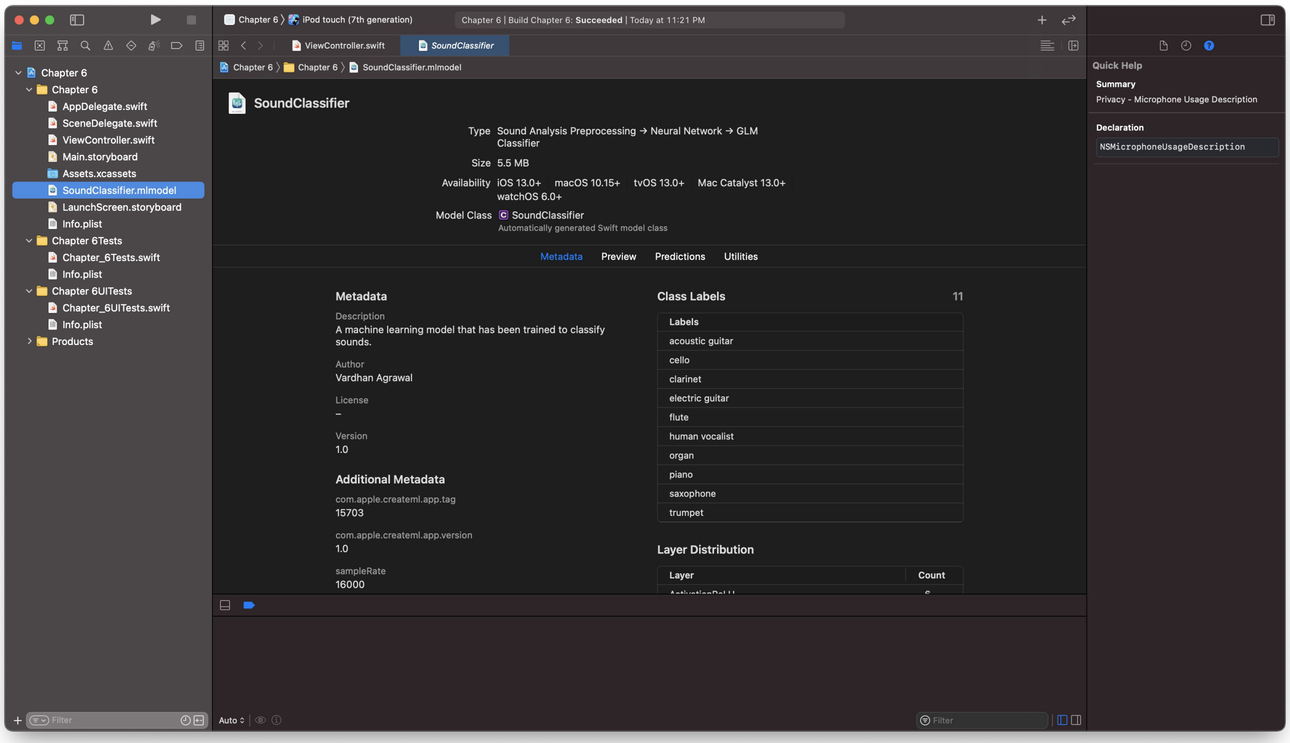
Next to the first cell in the table, which says **Information Property List**, click on the + button. This will allow you to create a new entry of your usage description in the table. When the blank field pops up, enter `NSMicrophoneUsageDescription` for the *key*, and explain why you need the microphone in the *value*. Once done, it'll look something like this:



Key	Type	Value
Information Property List	Dictionary	(16 items)
Privacy - Microphone Usage Description	String	In order to perform sound classification, this app must use your microphone.
Localization native development region	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Application Scene Manifest	Dictionary	(2 items)
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)

Figure 6-16: Edit Info.plist and request microphone permissions

Whether or not you used the IRMAS dataset from the previous section, you should have a trained model somewhere on your computer. If you didn't export your model from Create ML or forgot where you saved it, you can drag your model directly from Create ML into your Xcode project or from your file system into Xcode. Then, Xcode will recognize it and display the metadata you specified:



SoundClassifier

Type	Sound Analysis Preprocessing → Neural Network → GLM Classifier
Size	5.5 MB
Availability	iOS 13.0+ macOS 10.15+ tvOS 13.0+ Mac Catalyst 13.0+ watchOS 6.0+
Model Class	SoundClassifier Automatically generated Swift model class

Metadata

Description
A machine learning model that has been trained to classify sounds.

Author
Vardhan Agrawal

License
–

Version
1.0

Additional Metadata

com.apple.createml.app.tag
15703

com.apple.createml.app.version
1.0

sampleRate
16000

Class Labels 11

Labels
acoustic guitar
cello
clarinet
electric guitar
flute
human vocalist
organ
piano
saxophone
trumpet

Layer Distribution

Layer	Count

Figure 6-17: Importing Sound Classifier

Capturing and Processing Live Audio

Great! You now have your Xcode project ready, your user interface completed, and your trained model imported into your project. The next step is to get a live stream of audio into your app for processing. To do this, you'll be using a built-in API called *AVAudioEngine*, which allows apps to interface with the on-board microphone and speakers of Apple devices.

Capturing Audio

Before we can process the audio, we have to use the phone's microphone to capture sound from the user's environment. We also need to split this data into bite-sized clips, specify its quality, and format it correctly for the Core ML model.

For the tasks we need to do, there's only one framework needed: *SoundAnalysis*. To import it, enter the following under `import UIKit` in your `ViewController.swift` file:

```
import SoundAnalysis
```

This framework is specifically designed for sound classification and contains other tools we'll use later. Now, you'll be able to use it within the `ViewController.swift` file.

To begin capturing sounds, you need to use the framework we have just imported. Declare the following variable, which has an instance of the `AVAudioEngine` class, under the `confidenceLabel` variable:

```
let engine = AVAudioEngine()
```

Similar with all "engines" in real-life, we need to start our `AVAudioEngine` for it to work. However, since the `start()` method of the engine can throw errors, you'll need to make sure it doesn't fail. Inside your `viewDidLoad()` method, enter the following:

```
(try? engine.start()) ?? print("An error occurred.")
```

If you aren't that familiar with Swift, this syntax may look unfamiliar to you; however, it's simple once you dissect it. The `engine.start()` is our way of "starting the engine" we created in the previous step.

The `??` operator, in Swift, is known as the Nil Coalescing Operator. It allows you to specify an alternate value or action if the original return is `nil`. And, since `start()` throws errors, we can catch them using this operator. Alternatively, you could have used the standard `try-catch` block which is more commonly used in other object-oriented programming languages.

Handle the Request

When you use a sound classification model with the *SoundAnalysis* framework, a method called `request` is called by the system, in which you can specify what to do with the data received from the model. Let's define this method now.

To gain access to the `request` model, your `ViewController` class needs to conform to the `SNResultsObserving` protocol. To do this, we will create an extension to adopt it. Insert the following code at the bottom of your file, after the last closing curly brace:

```
extension ViewController: SNResultsObserving {
    // your code here.
}
```

While you could have done this directly in the main `ViewController` class declaration, we use extensions to better organize the code. At this point, Xcode will give you an error, which we'll address in the following step.

The error you probably see is Xcode trying to tell you that the `ViewController` class does not conform to the protocol `SNResultsObserving`. To fix this issue, simply add the `result(:)` method inside the extension as follows:

```
func request(_ request: SNRequest, didProduce result: SNResult) {
    // your code here.
}
```

The `request` parameter of this method is internal, and we don't need to deal with it. However, we do need to deal with the `result` parameter. As the name suggests, the `result` parameter is the classification label produced by the model and is passed in when this method is called by the system.

As with all machine learning tasks in Swift, there are optionals involved. To access the data stored in the `result` parameter, you'll need to cast it as an `SNClassificationResult` (currently an `SNResult` protocol, which isn't a concrete type). To do this, enter the following in the `request(:)` method:

```
guard let result = result as? SNClassificationResult
    else { return }
```

The `guard-let` statement ensures that the `result` parameter can indeed be casted as an `SNClassificationResult` type, and it will simply return if it fails. Next, you need to gather the actual classification text from this object. To do this, enter the following under the previous line:

```
guard let label = result.classifications.first
    else { return }
```

This will collect the `SNClassification` object with the highest confidence from the `result` parameter. More specifically, it will grab the first item from the `classifications` array.

After we have the result in an accessible format, we need to display it to the user. The `label` parameter currently contains a `SNClassification` object, which contains both a label and a confidence value inside it. The confidence value is a long decimal, which look something like this when displayed:

```
92.366666666666666666666666666666666666666667…%
```

However, we'd rather get a value like `92.4%` to keep things looking clean, simple, and elegant. Thankfully, Swift has a quick `round()` method for such decimals, and you can access and round the confidence level like this:

```
let confidence = round(label.confidence * 1000) / 10
```

Great! You now have the rounded value stored in the `confidence` variable. To display the text, do this:

```
DispatchQueue.main.async {
    self.resultLabel.text = "\(label.identifier)"
    self.confidenceLabel.text = "\(confidence)%"
}
```

As you know, the user interface must be updated on the main thread to maintain performance, and therefore, we should wrap it in a block.

Finally, you want to ensure that the app doesn't change the label too fast for the user to read it. To do this, you can check whether the confidence is above 65% (or some other percentage) before updating the user interface. Wrap your user interface code in an if-statement to look like this:

```
if confidence > 65 {
    DispatchQueue.main.async {
        self.resultLabel.text = "\(label.identifier)"
        self.confidenceLabel.text = "\(confidence)%"
    }
}
```

We also want to display that the sound wasn't recognized if the confidence is under our threshold. To do this, add the following else-block:

```
 else {
    DispatchQueue.main.async {
    self.resultLabel.text = "not recognized"
    self.confidenceLabel.text = "--%"
    }
}
```

Here, we're checking that the confidence isn't greater than 65% and then displaying "not recognized" in the results label. We're also displaying dashes in the confidence label to make it clear that we don't have a result.

Request Analysis

You've setup what needs to happen after we ask our model to process our audio. But now, we need to actually make the request so that our `request` method gets called.

Instantiate Model

When you imported your model into Xcode, it automatically created a Swift wrapper class for it. Since my model was called `SoundClassifier.mlmodel`, I can expect a Swift class called `SoundClassifier()` to be created for me to use. Similarly, you'll have a model wrapper class created with the name you chose for it. To instantiate this model, insert the following code under your previous `engine` declaration:

```
let classifier = try! SoundClassifier(configuration: MLModelConfiguration())
```

Now, you can access this single instance of your model in the `ViewController.swift` file, instead of making multiple redundant instances of the same class.

One thing to note is that I've used the `try!` keyword (note the exclamation point). This means that if an error is thrown during the initialization of either `SoundClassifier` or `MLModelConfiguration`, the program will crash.

You should rarely use exclamation points in Swift, but in this case, we don't mind our app crashing because if there's a problem with the classifier, there's likely a bigger problem we have to deal with first. Of course, in production-level code, you would always want to handle errors and show messages to the user instead; but here, we're in our little development sandbox!

Format Audio

Remember that you have created an instance of `AVAudioEngine` for audio capture before setting up the `request` function. Now, you need to format your input audio for analysis. To do this, you'll first need to define the format. Declare the following variable after the `classifier` constant:

```
var format: AVAudioFormat!
```

This is an uninitialized variable. Since we're making it non-optional, the program would crash if we'd forgotten to initialize it. Let's define it in the `viewDidLoad()` method and insert the following line of code after `super.viewDidLoad()` :

```
format = engine.inputNode.inputFormat(forBus: 0)
```

The `inputNode` singleton contains much of the information needed for using the internal microphones to capture sound. It also contains the format, which you accessed using the `inputFormat` method. Please note that

Create Analyzer

Now, you can use the format you created to instantiate another object, an `SNAudioStreamAnalyzer` . This object is used to handle audio input and classification outputs for your app. It will eventually call the `request` method you created earlier. Create another variable under `format` like this:

```
var analyzer: SNAudioStreamAnalyzer!
```

Then, go back to the `viewDidLoad()` method and initialize it:

```
analyzer = SNAudioStreamAnalyzer(format: format)
```

Here, you're simply initializing the `SNAudioStreamAnalyzer` using the `format` from the previous step. This analyzer will allow you to make requests after inputting the audio that came from the audio engine.

Add Request

We're almost done. Now, we have to create an `SNClassifySoundRequest` with our sound classification model as a parameter and then add it to the analyzer we've just created. You need to do this in the `viewWillAppear()` method as such:

```
override func viewWillAppear(_ animated: Bool) {
    if let request = try? SNClassifySoundRequest(mlModel: classifier.model) {
        // your code here.
    }
}
```

Again, we're using an `if-let` statement to catch any possible errors which may occur while trying to instantiate an `SNClassifySoundRequest`. If there is an issue, the system will simply not add the `request` to the analyzer. If there is no error, we need to specify what to do. Inside the block, type the following code:

```
(try? analyzer.add(request, withObserver: self)) ?? print("An error occurred.")
```

If you haven't seen enough optionals today, the `add()` method of `SNAudioStreamAnalyzer` also throws errors. So, we'll need to use the `try?` keyword to add the request we created and print an error message if problems arise. Specifying the observer as `self` will allow our `request` method to get called at the correct time.

Create Audio Tap

Last but not least, you need to install an audio tap to collect a sample of the audio from the live audio stream. Still, inside the `viewWillAppear()` method, add this:

```
engine.inputNode.installTap(onBus: 0, bufferSize: 1024, format: format) { (buffer:
AVAudioPCMBuffer, when: AVAudioTime) in
    // your code here.
}
```

This audio tap will be installed on the `inputNode` singleton from the audio engine. The `installTap` method specifies the bus (same as the format bus), buffer size (which means quality), and format (same as the one we declared earlier). Inside the closure, Swift gives us the buffer itself and the timestamps we need. Finally, make your analysis request like this inside the curly braces:

```
DispatchQueue.main.async {
    self.analyzer.analyze(buffer, atAudioFramePosition: when.sampleTime)
}
```

Here, you simply passed in the closure parameters into the `analyze(:)` method and did so on the main thread. Great! The request will begin to process, and you can now run your app to see the results!

Make sure you do this on a physical device, because the audio framework isn't compatible on the simulator. When you begin talking, playing an instrument, or singing, your app will tell you what the dominant sound is and its confidence level.

Conclusion

In this chapter, you learned about sound classification in detail. With Apple's latest update, Xcode comes bundled with Create ML and a new array of audio processing frameworks. In the beginning of the chapter, you learned how to find, record, and source audio clips for sound classification, and later, you learned how to organize them correctly.

In the second half of the chapter, you used Create ML to train your own sound classifier using the IRMAS audio dataset and learned how to use it in an iOS app. You also learned how to use the SoundAnalysis framework and an AVAudioEngine to utilize the built-in hardware microphone, and run the audio clips through your trained machine learning model.

Chapter 7
Cloud-Based Machine Learning with Firebase

In the previous chapter, you learned all about sound classification. You first learned how to gather and source sounds to train a machine learning model. Then, you actually trained your own sound classification model using Create ML. Finally, you built a fully functional app which could capture audio from your device's microphone and process it using your custom-made model. You learned a lot about the unique caveats of machine learning in the previous chapter.

In this chapter, you'll learn even more about machine learning, through another aspect which we haven't explored yet: *cloud-based machine learning*. You'll first learn how to setup Firebase through a dependency manager called CocoaPods. Next, you'll connect your Xcode project to Firebase. Then, you'll learn different image processing techniques, such as barcode scanning and image classification.

Later in the chapter, we'll shift gears to text-based machine learning, and create a language translation tool to end the chapter. By the end, you'll have an excellent grasp on using Firebase's machine learning tools and have three fully functional apps built!

7-1 Firebase at a Glance

In this chapter, you'll learn about Firebase and how to install it on your computer. You'll also create an account and learn about a dependency manager called CocoaPods, which allows you to download many different dependencies — almost all of them support CocoaPods. This way, you'll greatly increase your app development flexibility, aside from the machine learning aspect.

For those of you who haven't used Firebase in the past, you'll also get a bird's eye view at what you can do with the API. By the end of this section, you would be able to create your own app, configure it with CocoaPods on the command line, and then connect it to the Firebase account you'll create in this section of the book.

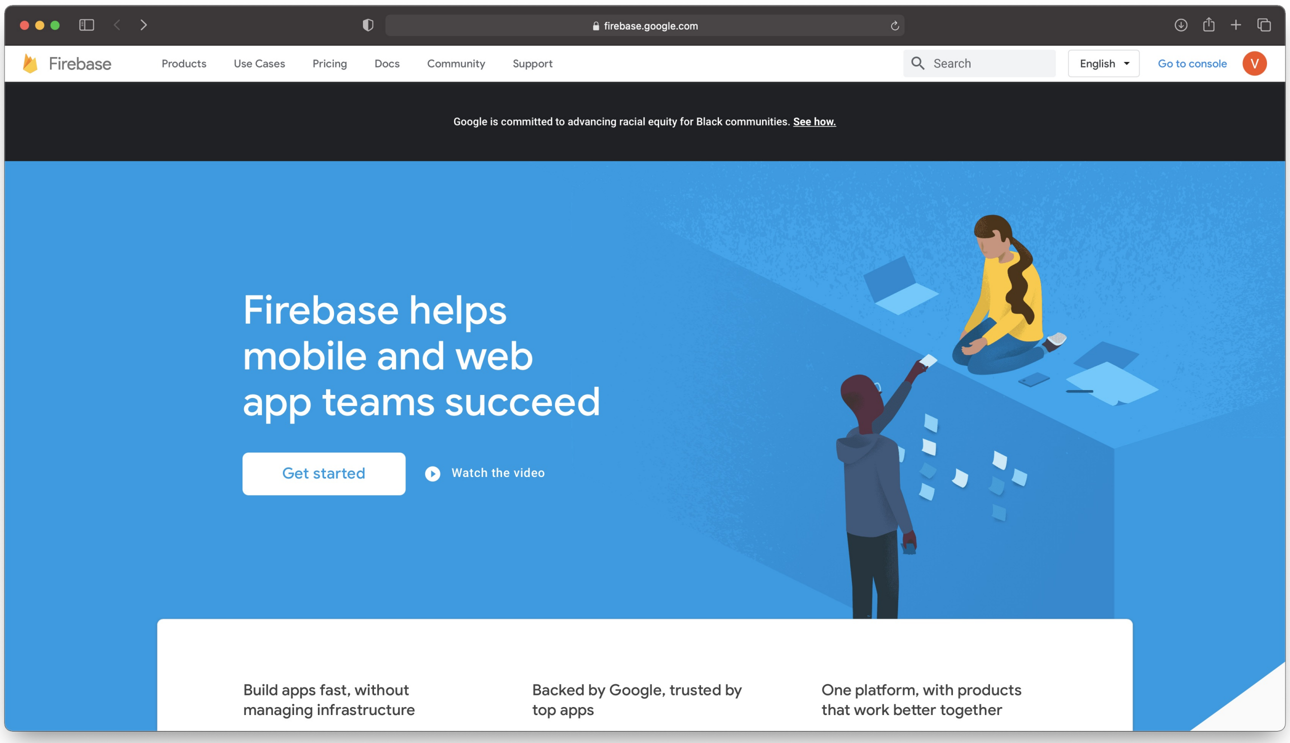

firebase.google.com
Firebase
Products
Use Cases
Pricing
Docs
Community
Support
Search
English
Go to console
Google is committed to advancing racial equity for Black communities. See how.
Firebase helps mobile and web app teams succeed
Get started
Watch the video
Build apps fast, without managing infrastructure
Backed by Google, trusted by top apps
One platform, with products that work better together

Figure 7-1: Working with Firebase

Basic Uses of Firebase

In case you're not familiar with it, Firebase was started as a backend-service which promised realtime data, different from anything else at the time. Instead of using typical HTML `POST` and `GET` requests from the database, Firebase allowed its users to get realtime data through WebSockets. Later, Firebase got bought by Google and has then been integrated into their cloud services division.

Now, with the new integration, the platform offers several new features, including machine learning applications, storage, and even some user-interface options. Nowadays, many of the apps you are using every day are powered by Firebase, because of its cutting-edge security and competitively low prices.

Onboarding and Customer Retention

When your user first opens your app, you can use Firebase to tell them how to use it, how to set up an account, and how to engage with your content. Firebase makes it easy to do that. It also enables you to track your users as they navigate through your app, allowing you to find which features are the most useful to your users.

Bug Testing and Feature Release

If you're unsure whether a feature will fly with your users, you can test it out with a small subset of them first. Coupled with the analytics tools of Firebase, this allows you to do A/B test on your features and launch only what's most relevant. You can also gain insights on how your app is performing, since you can track crashes and other bugs.

Login, Storage, and Chat

The more popular features of Firebase are the abilities to create quick sign in pages, store images and other media files, and implement a chatroom into your apps. Firebase makes it easy to implement these otherwise complex features into your app with only a few lines of code.

Database and Machine Learning

As mentioned before, Firebase started as a — well — database, hence the name. Their database is among the strongest available, allowing your app to get real-time updates without having to “refresh” or “update” the app. In addition, the machine learning division of Firebase allows you to perform some of the tasks we've explored in this book in the cloud, so you can improve your models live. This is the feature we'll be looking at most and exploring in-depth throughout this chapter.

Account Creation and Setup

Before you can start using Firebase, you'll need an account. Since Firebase was acquired by Google, a Google account will be enough. You likely already have a Google account, so you can log in; but if you don't, now is a good time to create one.

Creating a Project

To begin, you'll need to create your first Firebase project. And, once you have your Google account ready, you'll need to visit https://firebase.google.com. Then, log in with your Google account and select "Go to console" to enter the Firebase console.

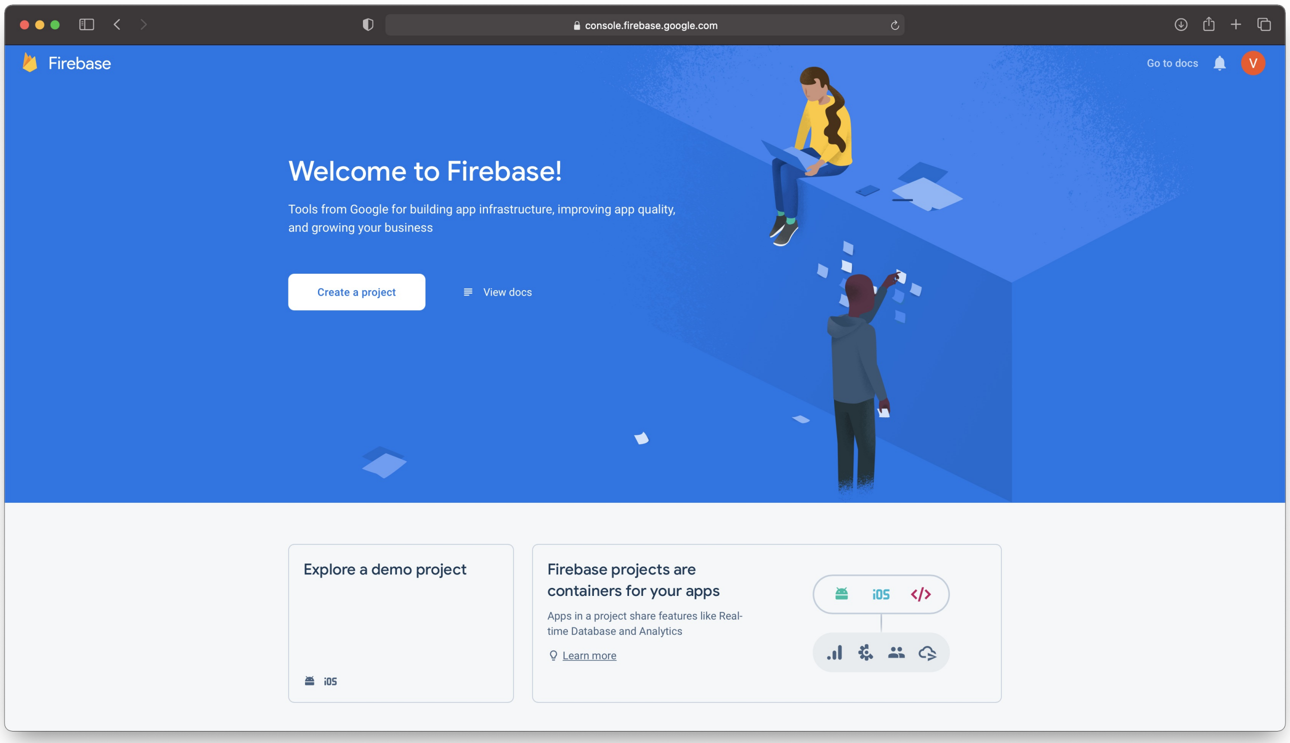
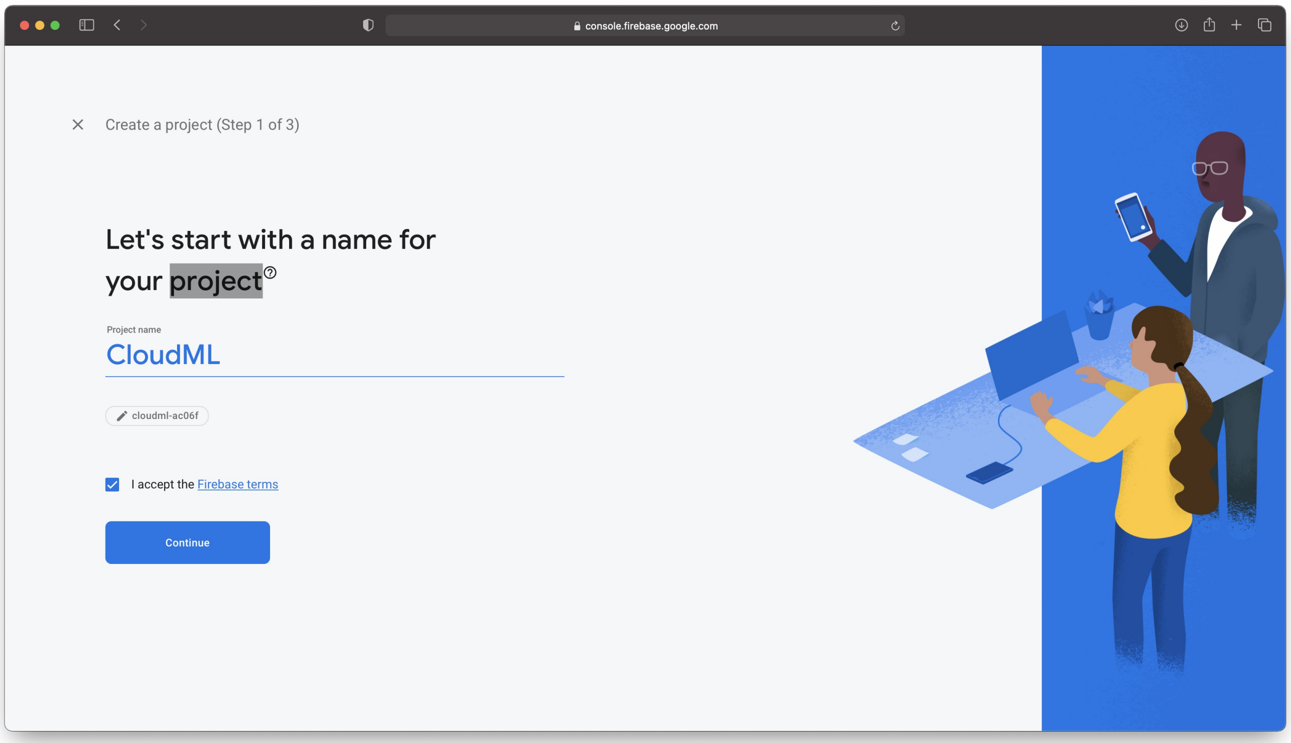

console.firebase.google.com
Firebase
Go to docs
V
Welcome to Firebase!
Tools from Google for building app infrastructure, improving app quality, and growing your business
Create a project
View docs
Explore a demo project
iOS
Firebase projects are containers for your apps
Apps in a project share features like Real-time Database and Analytics
Learn more
iOS
</>

Figure 7-2: Firebase Welcome Screen

In the console, tap the white *Create a Project* (or *Add project*) button in the middle of the screen. Alternatively, you could start with a demo project, but in this book, we'll start from scratch.

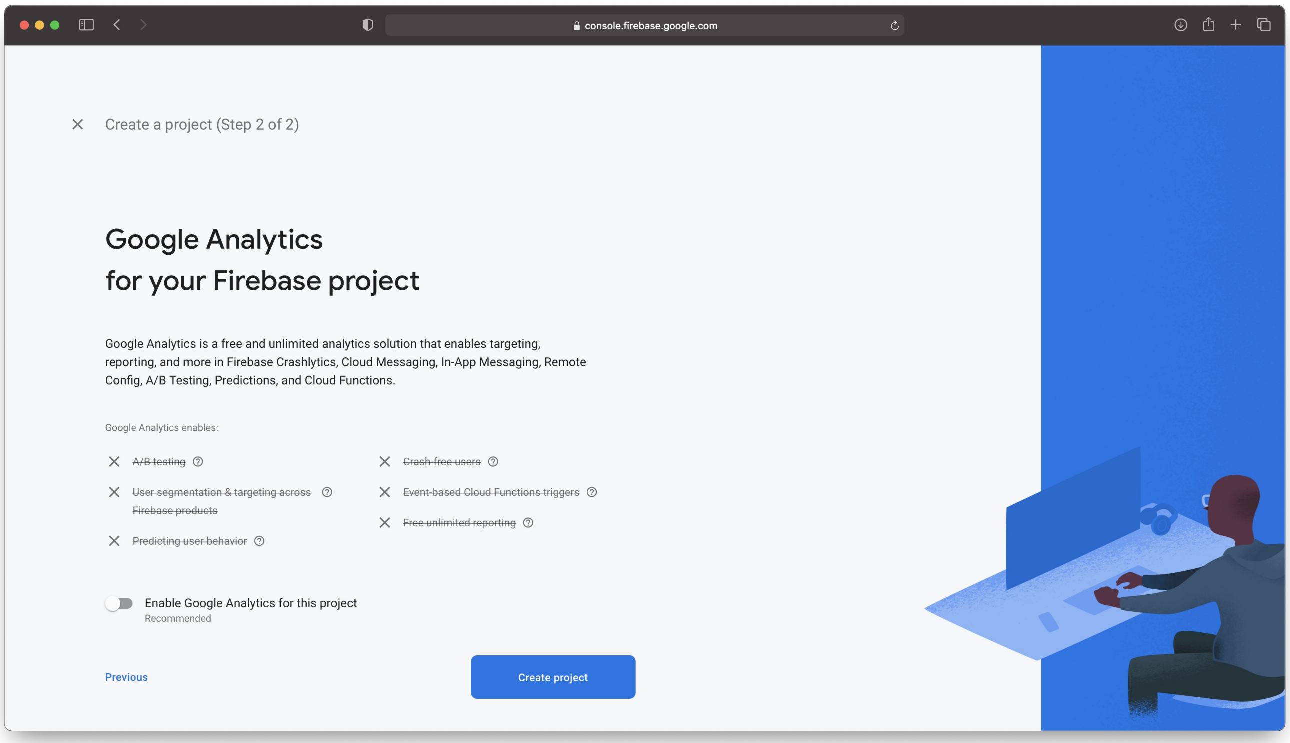


console.firebase.google.com
Create a project (Step 1 of 3)
Let's start with a name for your project
Project name
CloudML
cloudml-ac06f
I accept the Firebase terms
Continue

Figure 7-3: Naming your Project

I'm naming mine *CloudML*, but as usual, you can name your project whatever you'd like. Next, tap the blue **Continue** button underneath.

Now, it'll ask you whether you want to enable analytics. Flip the switch, since this chapter only focuses on machine learning, not analytics. If you enable the option, you'll need to configure a Google Analytics account, so I advise that you proceed without analytics for now. Finally, tap the blue *Create Project* button, and Firebase will start to create your project.



console.firebase.google.com

Create a project (Step 2 of 2)

Google Analytics
for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.

Google Analytics enables:

- ~~A/B testing~~
- ~~User segmentation & targeting across Firebase products~~
- ~~Predicting user behavior~~
- ~~Crash-free users~~
- ~~Event-based Cloud Functions triggers~~
- ~~Free unlimited reporting~~

Enable Google Analytics for this project
Recommended

Figure 7-4: Adding Google Analytics

Once your project has been created, you'll get a project screen in a minute or two, depending on the speed of your internet connection.

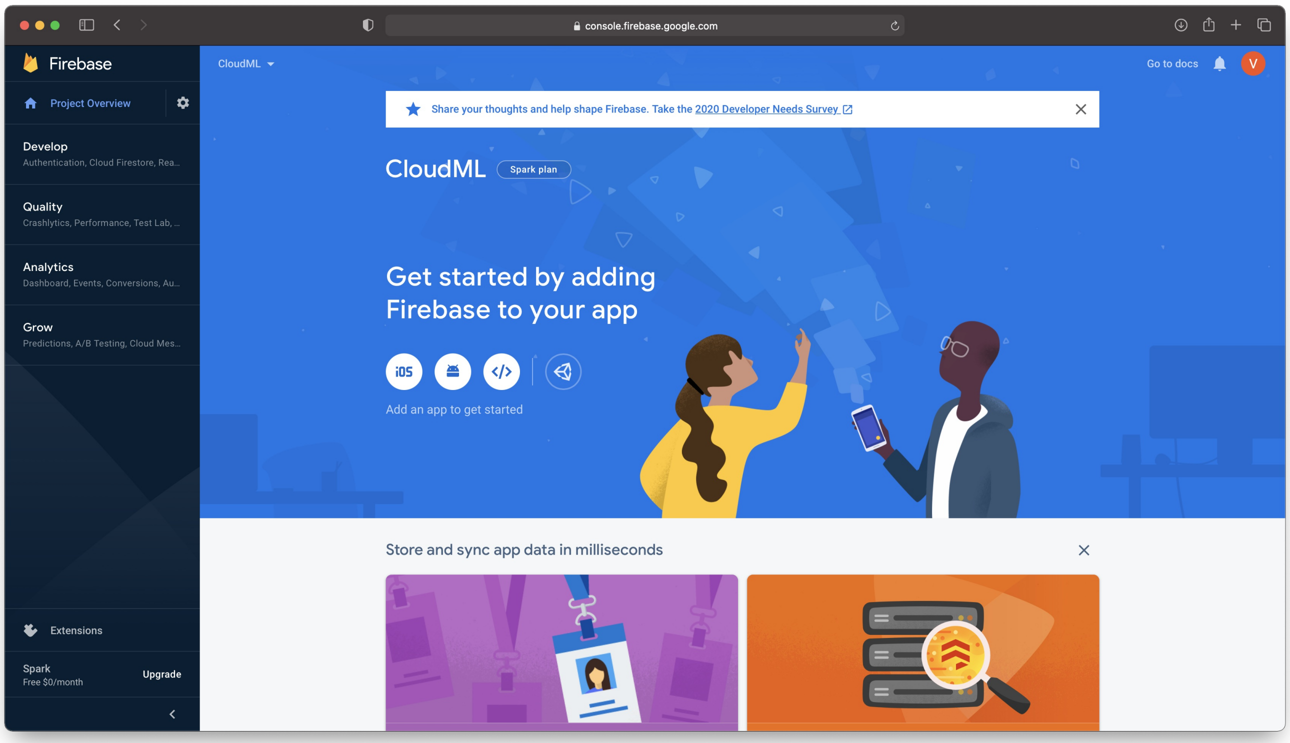

console.firebase.google.com
Firebase
Project Overview
Develop
Authentication, Cloud Firestore, Rea...
Quality
Crashlytics, Performance, Test Lab, ...
Analytics
Dashboard, Events, Conversions, Au...
Grow
Predictions, A/B Testing, Cloud Mes...
Extensions
Spark
Free $0/month
Upgrade
CloudML
Go to docs
Share your thoughts and help shape Firebase. Take the 2020 Developer Needs Survey
CloudML
Spark plan
Get started by adding Firebase to your app
iOS
Add an app to get started
Store and sync app data in milliseconds

Figure 7-5: Firebase Project Console

Great! You've created your first Firebase project, which you can use for a variety of different tasks. Follow along to learn about cloud computing and machine learning tools offered by Firebase. Next up, you will proceed to connecting Firebase with an Xcode project.

Create an Xcode Project

After you've created your Firebase project, you need to create an Xcode project to link it with. By now, you're probably familiar with the process, but if not, let's go through it step-by-step. Feel free to skip this section if you feel confident with this step.

First, you'll need to open Xcode. Again, select **Single View Application** as your template, so you can write your project from scratch. If you prefer, you may use other templates, but a Single View Application works best for connecting with Firebase.

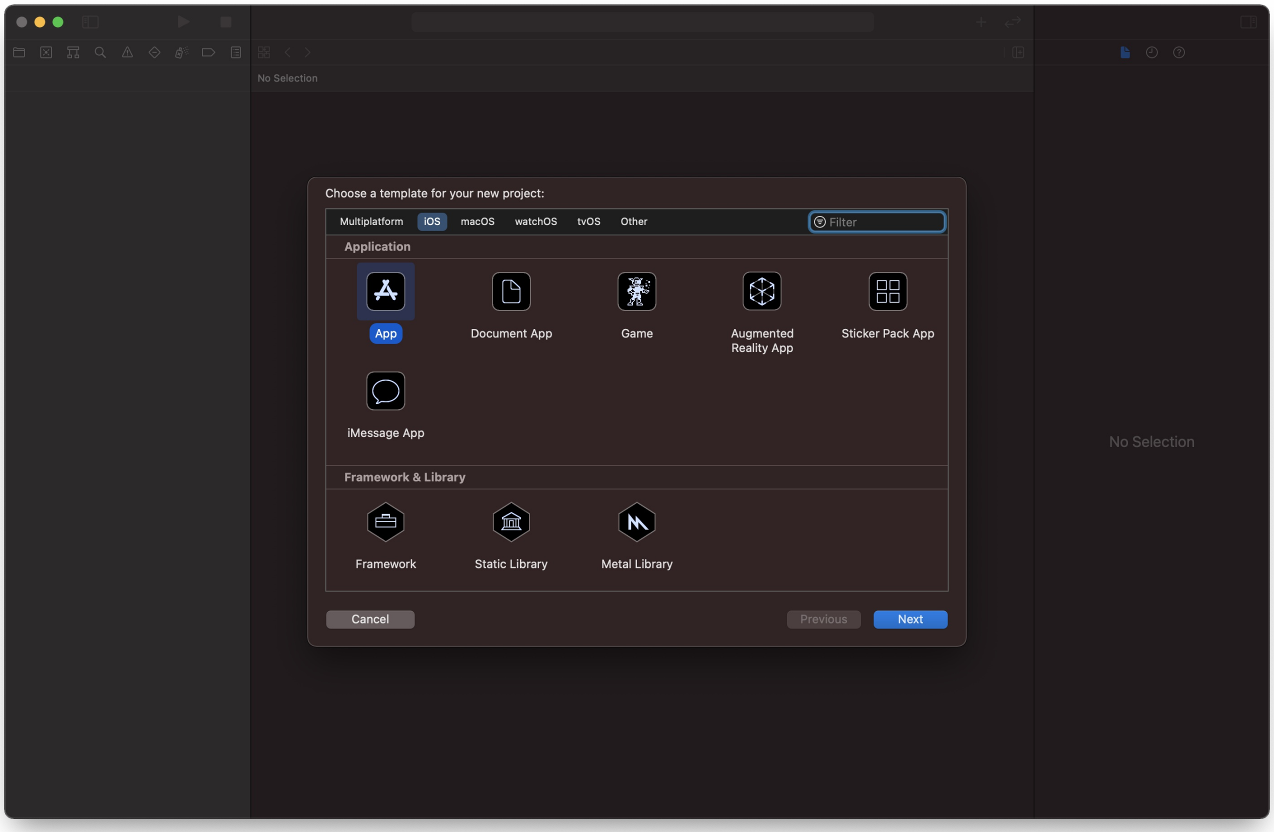


Figure 7-6: Selecting an Xcode Project Template

Xcode will then prompt you to name your project. This name doesn't matter much, so you can choose any convenient name: I'm using *CloudML* as my project name. Then select *Swift* for your Language and *Storyboard* for your User Interface.

Finally, Xcode will ask you where you'd like to save your project. Again, you can put it wherever you like, but make sure you store it in a place which you'll remember and be able to access it easily later.

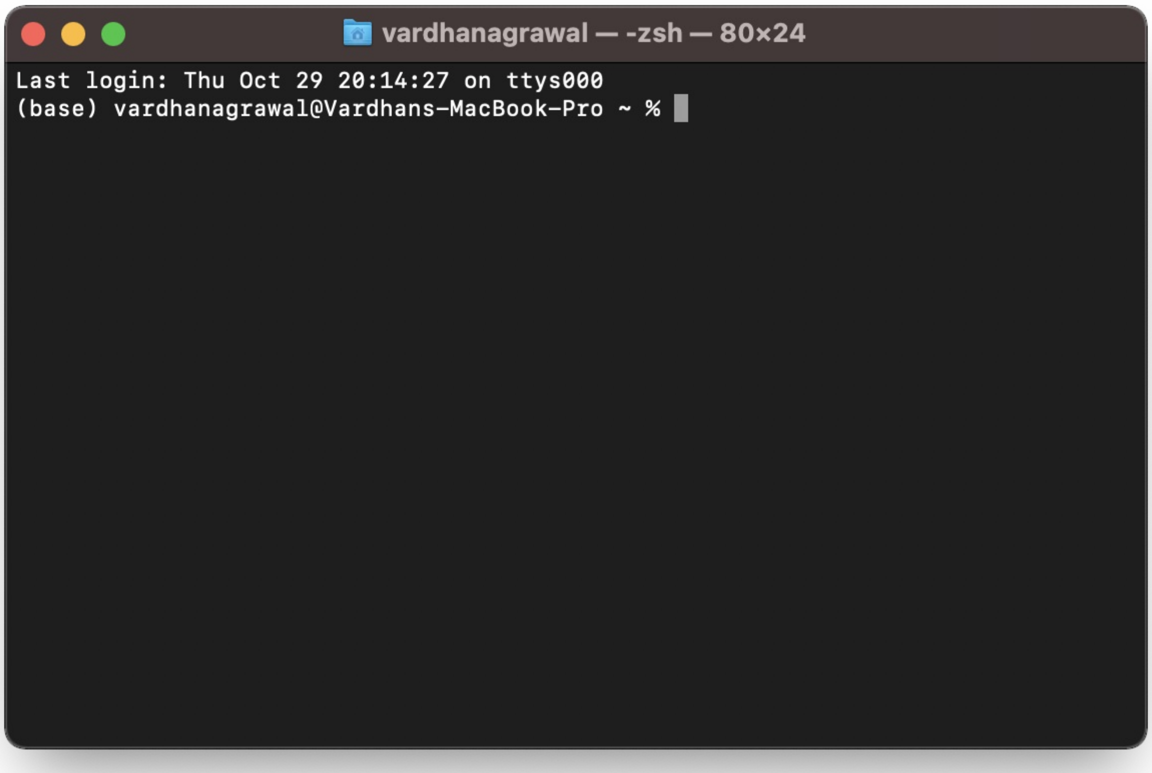
CocoaPods Dependency Manager

Now that you have an Xcode project connected, you're ready to install CocoaPods to add Firebase to your Xcode project. In case you haven't used CocoaPods, it's a dependency manager which helps you easily keep track of the different libraries you're using.

Installation

If you've never used CocoaPods before, you'll need to install it on the command line. You don't need to be familiar with the command line for this installation, since it's significantly easier than installations we've done earlier in this book. If you have used it before, you can skip the installation steps, as they would be redundant for you.

First, open the Terminal app, which gives you access to your computer's command line. You can find this by clicking **Command + Space** and typing “Terminal” or finding it in your file directory in **Applications > Terminal**. When you open it, you'll see a window that looks like this:



```
Last login: Thu Oct 29 20:14:27 on ttys000
(base) vardhanagrawal@Vardhans-MacBook-Pro ~ %
```

Figure 7-7: New Terminal Window

Afterwards, you can install CocoaPods if you don't have it. If you're unsure, you can run the command anyway, as it won't hurt your system. Since CocoaPods is a Ruby-based dependency manager, you can easily install it using the `gem` command. Enter the following (excluding the `%` prompt) into your command line:

```
% sudo gem install cocoapods
```

After you press **Enter**, it will prompt you for your computer password to give `gem` the appropriate permissions it needs to install CocoaPods. After a couple minutes, your installation should finish, and when you see the `%` prompt, you're ready to move on.

Configuration

To use CocoaPods, you'll need to configure it for each project you'd like to use it with. If you'd already had CocoaPods installed, you can resume following along from this point forward.

Enter the Project Directory

We will start by entering the project directory through the command line. To do this, you can type `cd` and drag the folder where your project is located to automatically fill in the path. Alternatively, you could type your directory name manually in this form:

```
% cd /<your directory>
```

Say, if your project is saved under the *Desktop* folder, you can type the command like this:

```
% cd /Desktop/CloudML/
```

Initialize CocoaPods

For every project you want to use with CocoaPods, you have to initialize it separately. While you only have to do the previous step once, you have to do the following step each time you want to use CocoaPods in your project. Enter the following once you're in your project directory:

```
% pod init
```

Press **Enter**, and after a few seconds, you'll find there to be a new **Podfile** file. This is where you'll specify which libraries, or dependencies, you'd like to include as part of your project.

Add Firebase

Finally, all that's left to do is install the relevant Firebase libraries. For now, we'll install the base one, but later in this chapter, you'll install the specific ones you need for cloud-based machine learning tasks!

Edit Podfile

First, open the Podfile with a text editor (e.g. Atom) and locate the following. If you don't want to install a third-party text editor, you can simply open it with TextEdit by double-clicking on the Podfile.

```
## Pods for CloudML
```

This comment indicates where you need to put your Firebase installation line. To add the Firebase pod, type the following under the comment:

```
pod 'Firebase'
```

Install Pods

Now that you've specified the Pods you'd like to use, save the **Podfile** and type the following into your Terminal:

```
% pod install
```

This should install all of the dependencies required for Firebase — at least up until this point. After you're done, you've successfully used CocoaPods to install Firebase to your Xcode project.

As you may have noticed, an additional file with the extension `.xcworkspace` appeared in your project after running the `pod install` command. In essence, this is a wrapper which contains both your actual project and your pods. From now on, you'll need to open this file instead of your `.xcodeproj` file.

Connect Project to Firebase

After you've created your Xcode project and installed Firebase using CocoaPods, you're ready to take the last step and connect it to Firebase. This will allow Firebase to know that your app is truly your app via a series of authentication methods. Don't worry, this is the fun part!

Add an App

In your Project Dashboard on Firebase, you should see the option to “add an app to get started.” Above this text, there are icons for iOS, Android, and Web. Obviously, we'll be adding an iOS app, so click on the circular **iOS** button.

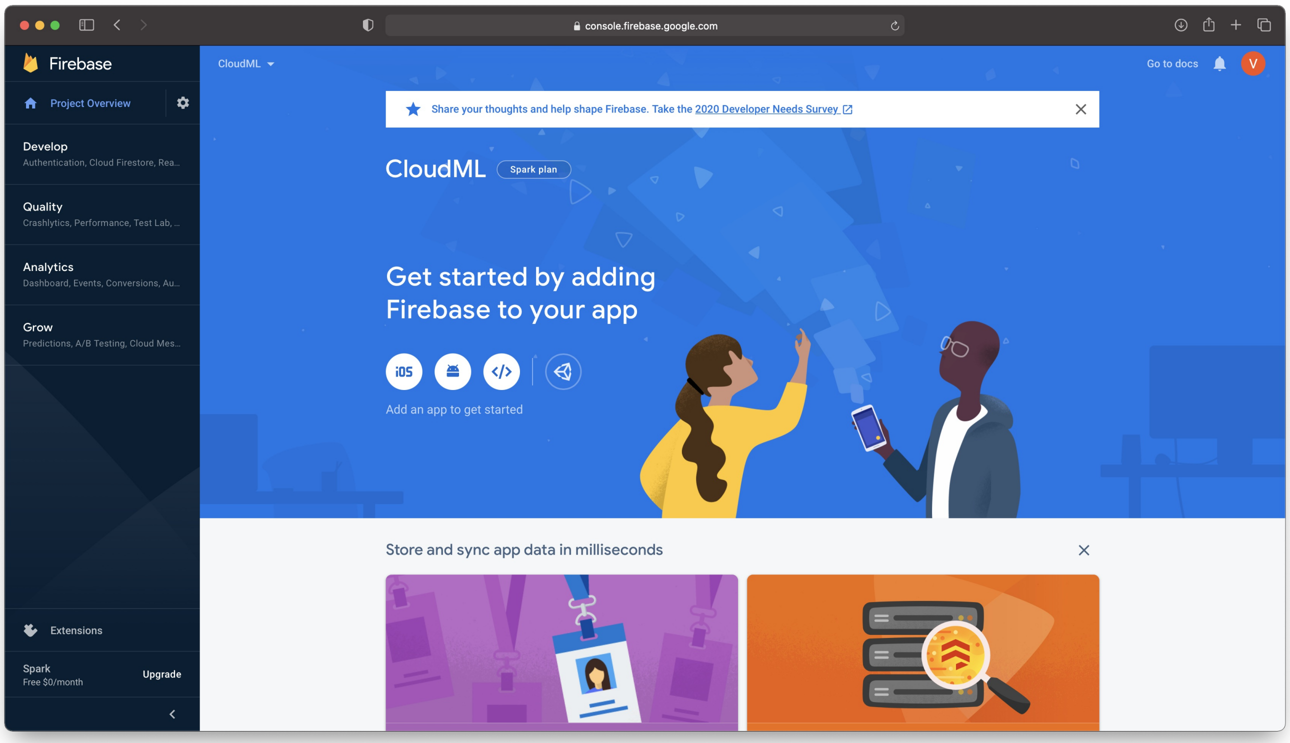

console.firebase.google.com
Firebase
Project Overview
Develop
Authentication, Cloud Firestore, Rea...
Quality
Crashlytics, Performance, Test Lab, ...
Analytics
Dashboard, Events, Conversions, Au...
Grow
Predictions, A/B Testing, Cloud Mes...
Extensions
Spark
Free $0/month
Upgrade
CloudML
Go to docs
Share your thoughts and help shape Firebase. Take the 2020 Developer Needs Survey
CloudML
Spark plan
Get started by adding Firebase to your app
iOS
Add an app to get started
Store and sync app data in milliseconds

Figure 7-8: Firebase Project Console

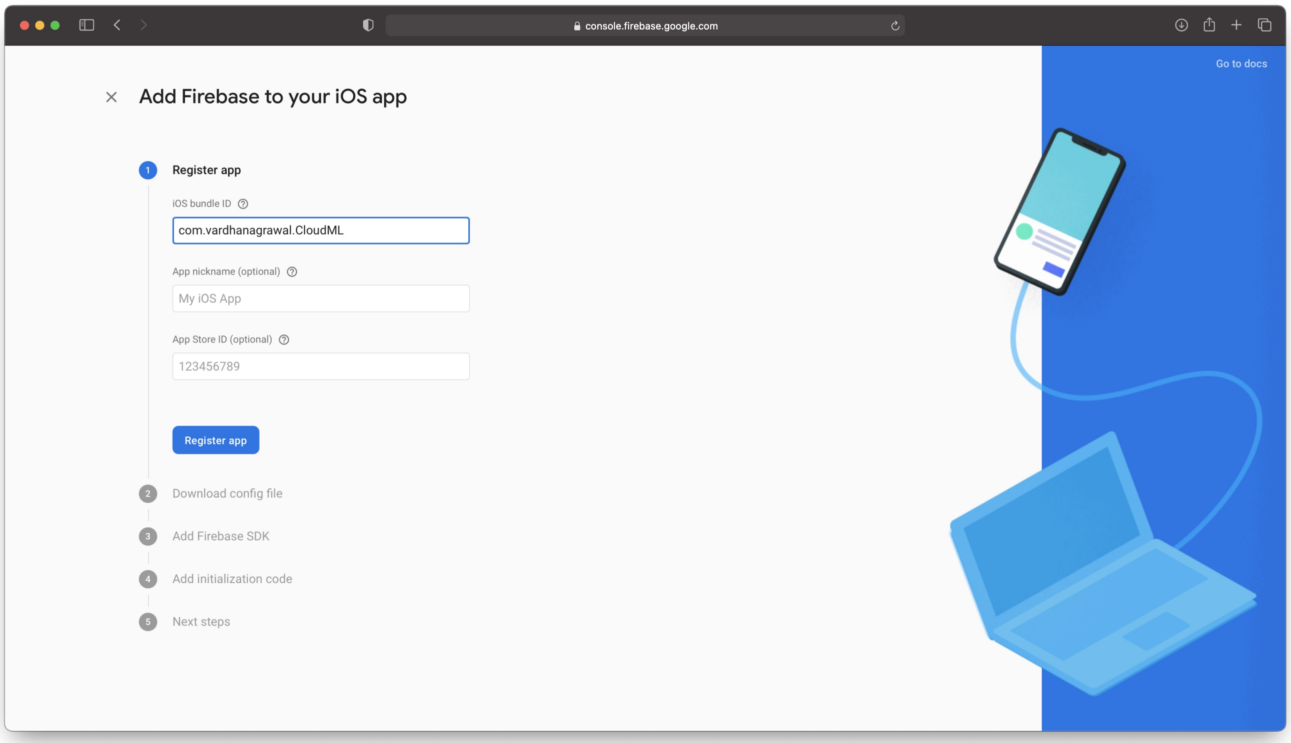
Register Bundle ID

Firebase will then ask for your project's Bundle ID as part of its guided setup process. To find your Bundle ID, open the `.xcworkspace` version of your project and then click the name of it (with the blue icon on the left of it) in the project navigator on the left of the screen.

Then, under **TARGETS**, select your project name and find the **Bundle Identifier** in the center of the screen. Yours should look similar to mine:

```
com.vardhanagrawal.Chapter-8
```

Once you've found this identifier, enter it to where it says **iOS Bundle ID** back in your Firebase console. Then, ignore the other two fields and click **Next**.


Add Firebase to your iOS app

1 Register app

iOS bundle ID: com.vardhanagrawal.CloudML

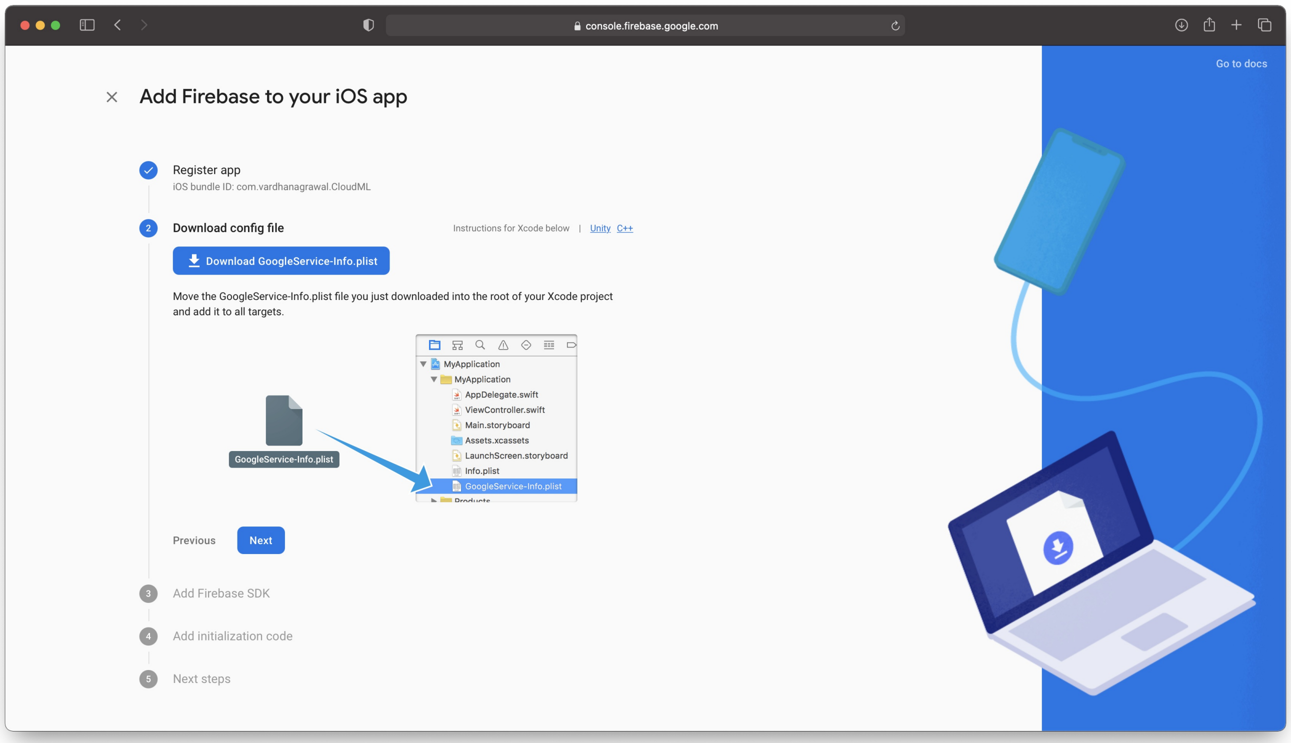
App nickname (optional): My iOS App

App Store ID (optional): 123456789

Figure 7-9: Adding Name and iOS Bundle ID

Download Configuration File

Next, Firebase needs a way to tell your app about your Firebase project. It does this using a `.plist` property list file, where it stores unique information about your project. Firebase should prompt you to download this file, and it should be done in a few seconds.



console.firebase.google.com

Go to docs

Add Firebase to your iOS app

Register app
iOS bundle ID: com.vardhanagrawal.CloudML

Download config file

Instructions for Xcode below | Unity C++

Download GoogleService-Info.plist

Move the GoogleService-Info.plist file you just downloaded into the root of your Xcode project and add it to all targets.

Previous Next

3 Add Firebase SDK

4 Add initialization code

5 Next steps

Figure 7-10: Importing Property List File

Once downloaded, your file will be called `GoogleService-Info.plist` . Drag this file from your Downloads folder into your Xcode project, alongside your other files such as `ViewController.swift` , `AppDelegate.swift` , and `Main.storyboard` .

Configure Firebase

Last, you need to tell your app to configure Firebase as soon as it launches. You can do this in the `AppDelegate.swift` file. The first method you see in the `AppDelegate` class should read like this:

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
            FirebaseApp.configure()
            return true
}
```

Notice that we've added only one line of code:

```
FirebaseApp.configure()
```

And, before Xcode starts showing warnings, let's add an import statement under `import UIKit` to import Firebase:

```
import Firebase
```

Excellent work! You've just finished setting up a Firebase workspace, creating an Xcode project, and connecting everything using CocoaPods. You're becoming a pro! Over the next few sections, you'll learn three different ways to use Firebase's machine learning tools.

7-2 Barcode Scanning and Image Labelling

In the previous section, you learned how to use the CocoaPods dependency manager to create your first Firebase-connected app. To use CocoaPods, you first installed it using Ruby and then initialized an Xcode project with it. After that, you created a Podfile and added Firebase to it.

In this section, you'll learn two ways to use Firebase ML Kit in your application. The first is barcode scanning, and the second is image labelling. By the end, you'll have an app which can scan barcodes and label specific objects in images, all without local models.

Starting a Live Feed

For both barcode scanning and image labelling, we need a live video feed. So, we'll start by making one. Then, you can choose to make either a barcode scanner or an image labelling app. If you'd like to build both, just comment out your work for the one you did first.

Create User Interface

The first step is to create your user interface. Since the app is just scanning barcodes, all it needs is an image view to preview the barcode, and a label to display the barcode's contents.

Open the Interface Builder

Start by opening the `Main.storyboard` file from the file inspector on the left side. As a side-note, make sure you've opened your `.xcworkspace` and *not* the `.xcodeproj` file, since you're using CocoaPods for this project. Once your interface builder is open, you're ready to build the interface.

Add an ImageView

Next, you'll need to add a place for the user to preview the barcode. We can achieve this using a `UIImageView`, similar to the live preview in Chapter 2. If you don't remember how to do this, you can refer back to that chapter for more detailed instructions.

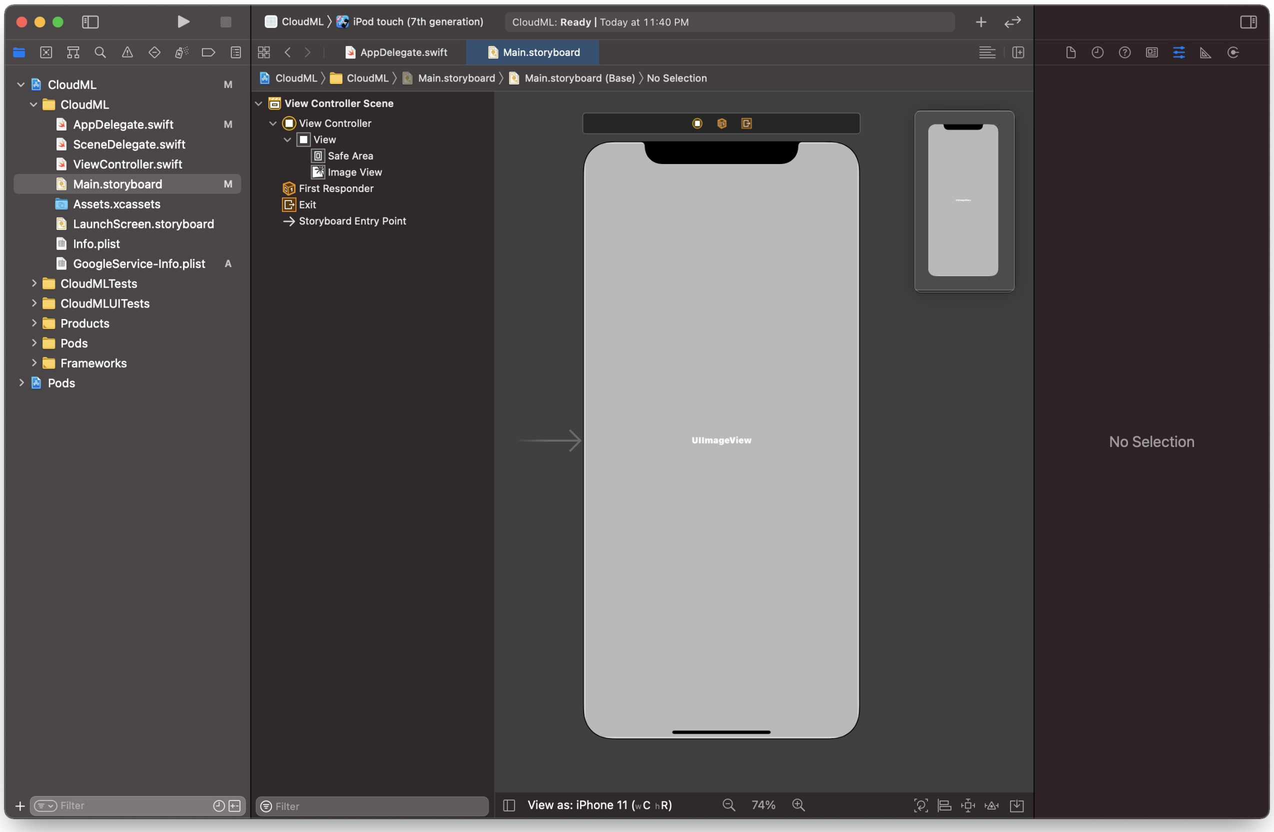


Figure 7-11: Adding an ImageView

Add a Label

Finally, you'll need a label which will tell you the barcode's value. Add this to the storyboard, and place it wherever you'd like. Just make sure it's visible when the app runs. You may also choose to add a background color to create contrast from the image view.

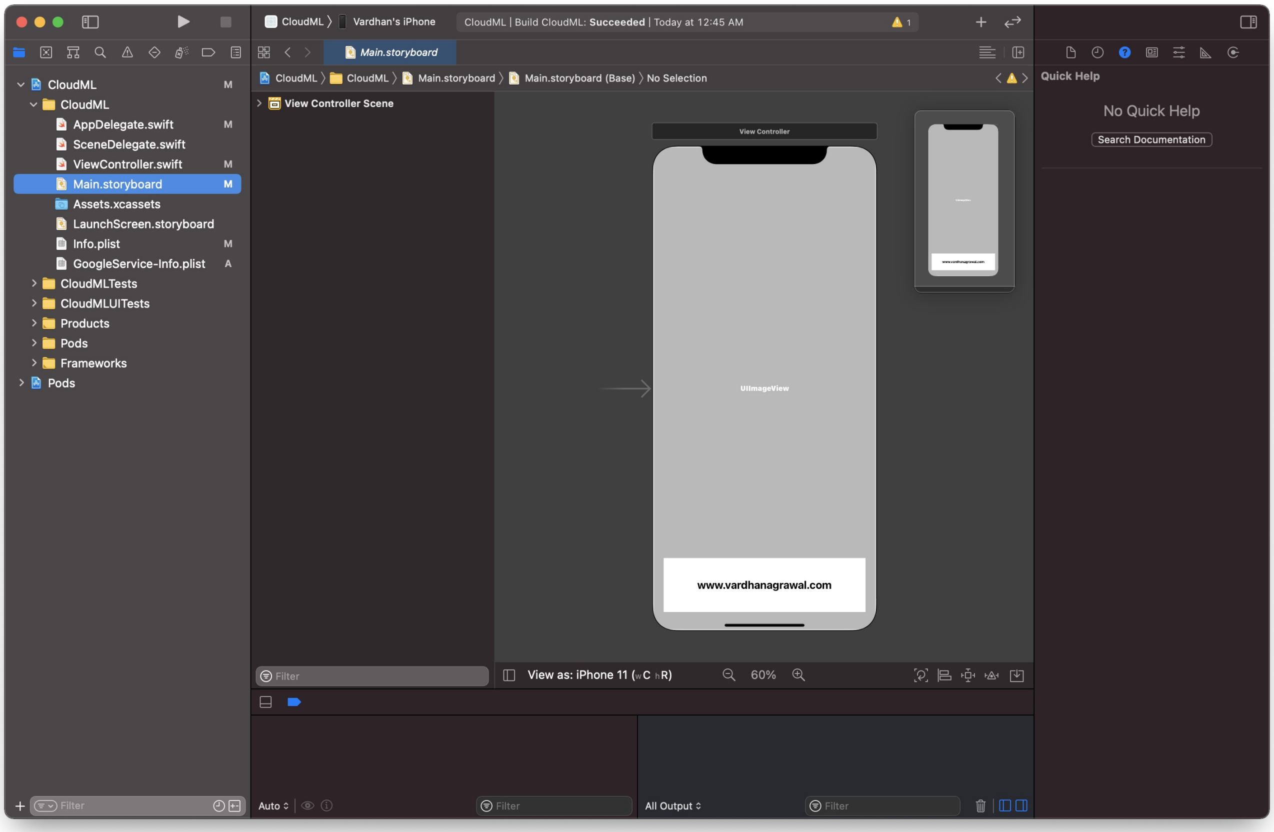


Figure 7-12: Adding a Label for Results

Add Constraints

Again, since we've already covered the concept of constraints previously, I won't be covering them in detail in this chapter. For your app, you'll need to constrain the image view and the label so that they work correctly on multiple devices and orientations.

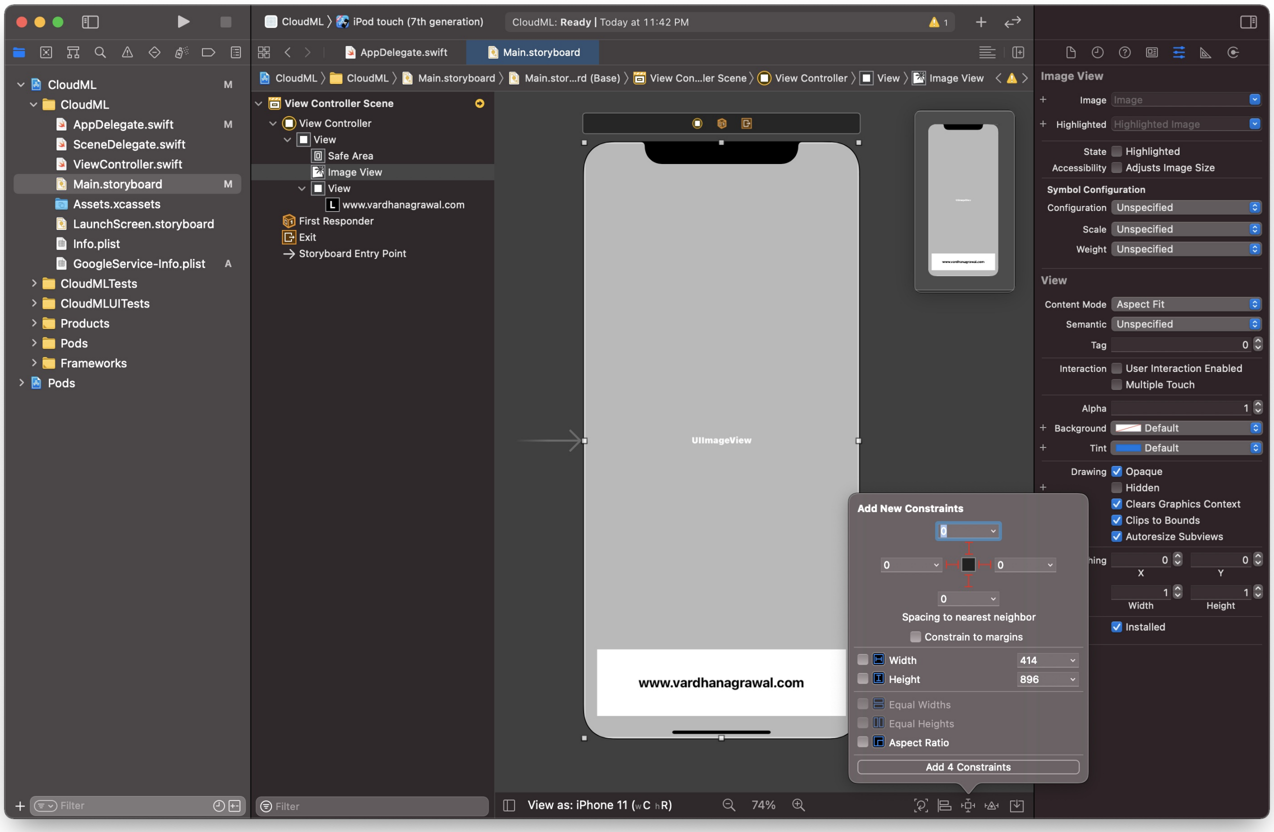


Image View
Image: Image
Highlighted: Highlighted Image
State: Highlighted
Accessibility: Adjusts Image Size
Symbol Configuration
Configuration: Unspecified
Scale: Unspecified
Weight: Unspecified

View
Content Mode: Aspect Fit
Semantic: Unspecified
Tag: 0
Interaction: User Interaction Enabled; Multiple Touch
Alpha: 1
Background: Default
Tint: Default
Drawing: ✓ Opaque; Hidden; ✓ Clears Graphics Context; ✓ Clips to Bounds; ✓ Autoresize Subviews
X: 0 Y: 0
Width: 1 Height: 1
✓ Installed

Add New Constraints
0 / 0 / 0 / 0
Spacing to nearest neighbor
Constrain to margins
Width: 414
Height: 896
Equal Widths
Equal Heights
Aspect Ratio

Figure 7-13: Adding Auto-Layout Constraints

Connect IBOutlets

The last step is to declare your interface elements as `@IBOutlet` variables. Open your `ViewController.swift` file and enter the following inside the `ViewController` class declaration:

```
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var label: UILabel!
```

Next, in your `Main.storyboard` file, connect these outlets by two-finger clicking (right clicking) on the `ViewController`, and then connect the interface elements to their respective variables. When you return to your `ViewController.swift` file, the circles on the left of the `IBOutlets` will be filled, confirming that you've successfully connected the outlets.

Create Live Feed

After your user interface has been created, we need to populate it with a live video feed. This is a very useful skill to have for computer vision applications, since a live feed adds a ton to the user experience.

Import Framework

To start, import the `AVKit` framework, which will allow you to create a live video feed for your app. Simply put the following under the `import UIKit` line of your `ViewController.swift` file:

```
import AVKit
```

Create an Extension

To keep things organized, start by extending `ViewController` after the last closing curly brace. The extension should also conform to `AVCaptureVideoDataOutputSampleBufferDelegate` for our video-related tasks and will look like this:

```
 extension ViewController:
    AVCaptureVideoDataOutputSampleBufferDelegate {
        // your code goes here.
}
```

Create a Method

To contain all of our code, we'll create a method called `setupSession()`. It will take no parameters and do exactly as its name suggests: create and setup the `AVCaptureSession`. Do this inside the extension you've just created in the previous step:

```
func setupSession() {
    // your code goes here.
}
```

Then, call the method in the `viewDidLoad()` method in the main body of the `ViewController` class. This will ensure that everything gets setup after the view loads up. After adding that in, your `viewDidLoad()` method will look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupSession()
}
```

Now, on to the next step!

Declare Input

Inside your extension, type the following lines of code to ensure that your device, in fact, has a camera and that it is available for use. If these lines of code fail, the function will return without crashing your program as indicated by the `guard let` statements:

```
guard let device = AVCaptureDevice.default
     (for: .video) else { return }
guard let input = try? AVCaptureDeviceInput
    (device: device) else { return }
```

Create a Session

Now, you can declare the capture session and specify the quality at which you'll capture video. Here, we'll use the highest quality, but if you're making an app geared for millions of users, you'll need to think about the battery life, efficiency, quality and how these factors affect the usability of your app. Enter the following before the previous lines of code:

```
let session = AVCaptureSession()
session.sessionPreset = .hd4K3840x2160
```

The session, an instance of `AVCaptureSession`, is stored inside of the `session` variable. In the next line, its property `sessionPreset` is set to the highest resolution through an `enum` within the framework.

Allow Video Preview

Obviously, we want our user to be able to see the video captured from the camera, so they can aim it at the correct target. Therefore, we need a preview layer to display the video stream within the image view we added earlier. To do this, add the following into the `setupSession()` method:

```
let previewLayer = AVCaptureVideoPreviewLayer
    (session: session)
previewLayer.frame = view.frame
previewLayer.videoGravity = .resizeAspectFill
imageView.layer.addSublayer(previewLayer)
```

In the code above, first, the `previewLayer` is declared, and the current `session` is passed in. Then, we set the size to the entire screen, regardless of the size of the image view. To prevent distortion, we use `.resizeAspectFill` in the third line, and finally, we add the preview layer to the image view.

Output the Data

After showing the data to the user, we need to output it for Firebase to use later on. Eventually, we'll be extracting individual frames from this live video feed. Add the following after the previous lines of code:

```
let output = AVCaptureVideoDataOutput()
output.setSampleBufferDelegate(self,
    queue: DispatchQueue(label: "videoQueue"))
```

Run the Session

The last thing to do in our `setupSesssion()` method is to add the `input` and `output` specifications to the capture session, and then start running it:

```
session.addOutput(output)
session.addInput(input)
session.startRunning()
```

All set! If you run your app now, you should be able to see the live video feed. Since this is a camera-based application, you'll need to use a physical device to test it out — the Xcode simulator doesn't have any camera support, unfortunately.

Barcode Scanning

Nowadays, barcodes are everywhere. You can find one on the back of your laundry detergent, bag of chips, or even the back of this book, if you're reading the paperback version. Conventionally, barcodes are scanned using a laser-based barcode scanner, but you can also scan them using your smartphone's camera and machine learning.

Finish Housekeeping

Before you can actually start scanning barcodes, you'll need to import the appropriate frameworks and fetch an image from the live video feed. Let's finish get the housekeeping out of the way now.

Add Pods

In the previous section, you configured Firebase. But now, you'll need to install some specific libraries for vision and barcode scanning. For reference, these libraries used to be part of the Firebase SDK, but they have since been moved to Google's ML Kit. So, we'll be using that version here instead.

The first step is to add a couple of pods to your Podfile. On the left side inside the project navigator, you can find the Podfile under the Pods target (with the blue icon).

Open your Podfile and add the following line of code after the `pod ‘Firebase'` you added earlier:

```
pod 'GoogleMLKit/BarcodeScanning'
```

Install Pods

After saving the file, head back to the Terminal and enter your project directory. After getting into the project's folder, type `pod install` to install both of these new pods.

Import Firebase

When that's been completed, go back into the `ViewController.swift` file and add the following import statement after `import UIKit` :

```
import MLKit
```

This tells Xcode that you want to use the MLKit library you imported using Cocoapods in the previous step.

Capture a Pixel Buffer

Earlier, you created a live video stream, and we're ready to use that feed. However, you cannot give Firebase a video feed and expect a result; you'll need to first extract individual frames. To do this, add the following method to your `ViewController` extension, beneath `setupSession()` :

```
func captureOutput(_ output: AVCaptureOutput,
    didOutput sampleBuffer: CMSampleBuffer,
        from connection: AVCaptureConnection) {
    // your code goes here.
}
```

This delegate method gets called for every frame in your video output and gives you the frame in the form of a `CMSampleBuffer` , which you can convert to an image for Firebase.

Scanning Barcodes

Now, you're finally ready to scan barcodes! Scanning barcodes using Firebase happens locally, which means you don't need an internet connection for it to work. For the purposes of this app, we'll simply fetch the value stored in the barcode and display it to the user. So, let's dive in.

Specifying Format

First, let's specify the format of barcodes that you'd like to detect. You can do this by entering the following inside the `captureOutput()` method:

```
let format = BarcodeFormat.all
```

This creates a constant and stores all of the barcode formats inside it. Alternatively, you could select specific formats for your scanner to only detect those. Then, create another constant to store your barcode options:

```
let options = BarcodeScannerOptions(formats: format)
```

Create a Barcode Detector

Next, you'll need to create your actual barcode detector using the options you created earlier. You can do this using the `Vision` API like this:

```
let barcodeDetector = BarcodeScanner.barcodeScanner(options: options)
```

This creates an instance of a `barcodeScanner`, using an initial parameter of the `options` from the previous step. This specifies the barcode types it's able to take.

Extract Image

As mentioned earlier, you can convert the pixel buffer to an image for use in the Firebase API. Luckily, Firebase's `VisionImage()` class takes in a `CMPixelBuffer` as an initial parameter, so you can create an instance of `VisionImage` like this:

```
let image = VisionImage(buffer: sampleBuffer)
```

The `image` constant now stores an image for you to process using the `barcodeDetector` which we created earlier.

Detect Barcodes

After all these configurations, you're ready to scan barcodes using the `barcodeDetector`. You can do this using the `process(:)` method with `image` as a parameter:

```
barcodeDetector.process(image) { (barcodes, error) in
    // your code goes here.
}
```

This will detect the barcodes, but you also need to show the result to the user, with the label we created earlier. To do this, add the following inside the closure of the `process` method:

```
if let value = barcodes?.first?.displayValue {
        self.label.text = value
}
```

Since we're unsure whether a value actually exists in the image, we need to unwrap the optional. The code inside the `if-let` statement only executes if there is a barcode.

To get the value of the barcode, we get the first element of the features array (it's an array for scanning multiple barcodes) and take the `displayValue` from it. Then, inside the `if-let` statement, we set the label's text to the value of the barcode.

Camera Usage Description

But, there's still one last thing you need to do if you want your app to run without crashing: user permissions!

Since we’ll be using the camera for this app, we need to ask the user for permission before doing so — otherwise, our app won’t work. Head to your **Info.plist** file first. Then, click the + button next to the **Information Property List** and paste the following:

```
NSCameraUsageDescription
```

This will auto-correct to `Privacy - Camera Usage Description` if done correctly. In the value section for this key, type a string which describes why you need to use the camera.

Great! You now have a barcode scanning app, which will display the value of any barcodes when you run it. Go ahead and run it; try it out with some barcodes from Google or create your own.

Image Labelling

Creating an image labelling app is much easier with Firebase than it was with Core ML, which we have tried earlier in this book. Also, with Firebase, you have the option of performing the image labelling locally or on the cloud. Both of these methods have their advantages and disadvantages, but we'll be using the on-device version since it's free for an unlimited number of requests.

Finish Housekeeping

If you've already created the barcode scanner, you may have already completed some of these steps. If not, you can follow along with all of them. We'll need to install some pods, import frameworks, and capture a pixel buffer.

Add Pods

There are two new pods you'll need for Firebase's machine learning tasks. To add them, open your Podfile from the `Pods` target on the project navigator on the left side. Then, enter the following where you see the other pods:

```
pod 'GoogleMLKit/ImageLabeling'
```

Install Pods

After updating your Podfile, you'll need to save it. Then, you can start installing the specified pods using the Terminal. Enter your project directory and type the following:

```
pod install
```

Import Firebase

In a few moments, after your pods have been installed, go back into the `ViewController.swift` file and add the following import statement after import UIKit:

```
import MLKit
```

If you've already created the barcode scanner, however, you've already taken care of this step.

Capture a Pixel Buffer

If you haven't created the barcode scanner, you'll need to extract individual frames from the live video feed. To do this, add the following method to your `ViewController` extension, beneath `setupSession()` :

```
func captureOutput(_ output: AVCaptureOutput,
    didOutput sampleBuffer: CMSampleBuffer,
        from connection: AVCaptureConnection) {
    // your code goes here.
}
```

This delegate method gets called for every frame in your video output and gives you the frame in the form of a `CMSampleBuffer` , which you can convert to an image for Firebase.

Label Images

Now, you're ready to label the images you captured as pixel buffers. Remember, if you did the barcode project, comment out your code from the **Scanning Barcodes** section, or it will conflict with what we're about to do now. Alternatively, create a new project and repeat the steps from scratch.

Extract Image

First, you'll need to convert the pixel buffer into an image. Since Firebase's Vision APIs don't directly accept input as a `CMPixelBuffer`, you'll need to convert it to Firebase's own image format. Do the following inside the `captureOutput()` method:

```
let image = VisionImage(buffer: sampleBuffer)
```

This stores the `VisionImage` created using the pixel buffer into a constant called `image`. You can later use `image` in the request.

Define Labeler

Next, you'll need to create an instance of the labeler. Depending on whether you want to do it on device, or on the cloud, your work will be a little bit different. For on-device labelling, type the following:

```
let options = ImageLabelerOptions()
let labeler = ImageLabeler.imageLabeler(options: options)
```

Here, we're declaring a blank set of `ImageLabelerOptions()`, since we don't want to set any custom parameter right now -- just to meet the requirements. And with that, we're creating an image labeler.

A side note: if you want your model to run on the cloud, you'll need to use Firebase's API instead. In the past, it was possible to do it in the same API; however, it appears Google has only chosen to migrate their on-device image labeling to Google ML Kit, leaving the cloud-based image labeler to Firebase.

Process Images

Using the `labeler` you've just created, you can process the video frame. Similar to the barcode scanning app, the `process()` method has a closure, in which it passes the result and error, if applicable. To call this method, type the following:

```
labeler.process(image) { labels, error in
 // your code goes here.
}
```

This runs the `image` from the earlier steps in a machine learning model in the cloud or on your phone, depending on the `labeler` you specified earlier.

Display Results

Finally, you're ready to display the results you've got from the `process()` method in the previous step. Inside the closure of the method, type the following to update your label:

```
if let labels = labels {
    var text = ""
    for label in labels {
        text += "\(label.text)\n"
    }
    self.label.text = text
}
```

First, using an `if-let` statement, you've checked whether the optional `labels` array exists (or if it's `nil`). If it does exist, you're creating a variable string and appending each label to it using a `for-loop` . At the end of the string, you'll notice `\n` , which is a newline character. This means that each detected item will be on a new line in your label.

Allow Multiline Label

Since by default, each `UILabel` only has one line. While we're using multiple lines to display the label, we need to allow our label to have multiple lines. To do this, head to the `viewDidLoad()` method in your `ViewController` class and add the following:

```
label.numberOfLines = 0
```

By setting the number of lines to `0`, we're telling Xcode to allow as many lines as needed to display the entire text. This allows the label to change size dynamically instead of having a fixed, single-lined height.

Now, when you run your app, you should see your images (or video in this case) get classified and displayed on the label. You've successfully built an image labelling app using Firebase, congrats!

7-3 Translating Between Languages

In the previous section, you learned to create a barcode scanner and an image labelling app. You created one (or both) of these apps using a live video feed for a high-quality user experience. You learned to use your device's camera and then extract pixel buffers to process frame-by-frame. By the end, you had two fully functional apps which used Firebase for machine learning purposes.

In this section, you'll learn another application of Firebase's machine learning tools. This time, though, you'll be dealing with natural language processing instead of image processing, as in Chapter 5. By the end of this section, you'll know how to translate text between languages and have an app which can provide on-device, realtime translations between the languages of your choice.

Creating a Text Experience

Before starting with machine learning, we need to create a user interface that can handle text input and display text output. To do this, start by creating a new Xcode project and setting up a new Firebase project. You will need to follow the steps in Chapter 7-1 again to configure CocoaPods and Firebase for a new project. The good news is that practice makes perfect, and doing it again will only make you better at the process!

Design Layout

The first step in creating an app is the interface. Let's take a moment now to design the layout for the app. Even though you can make it as simple or as complex as you'd like, each element is important in the app as a whole.

Add a Text Field

The most important user interface of text-classification apps is the text field, of course. Start by opening your `Main.storyboard` file and adding a text field. Then, customize the placeholder text and adjust the size.

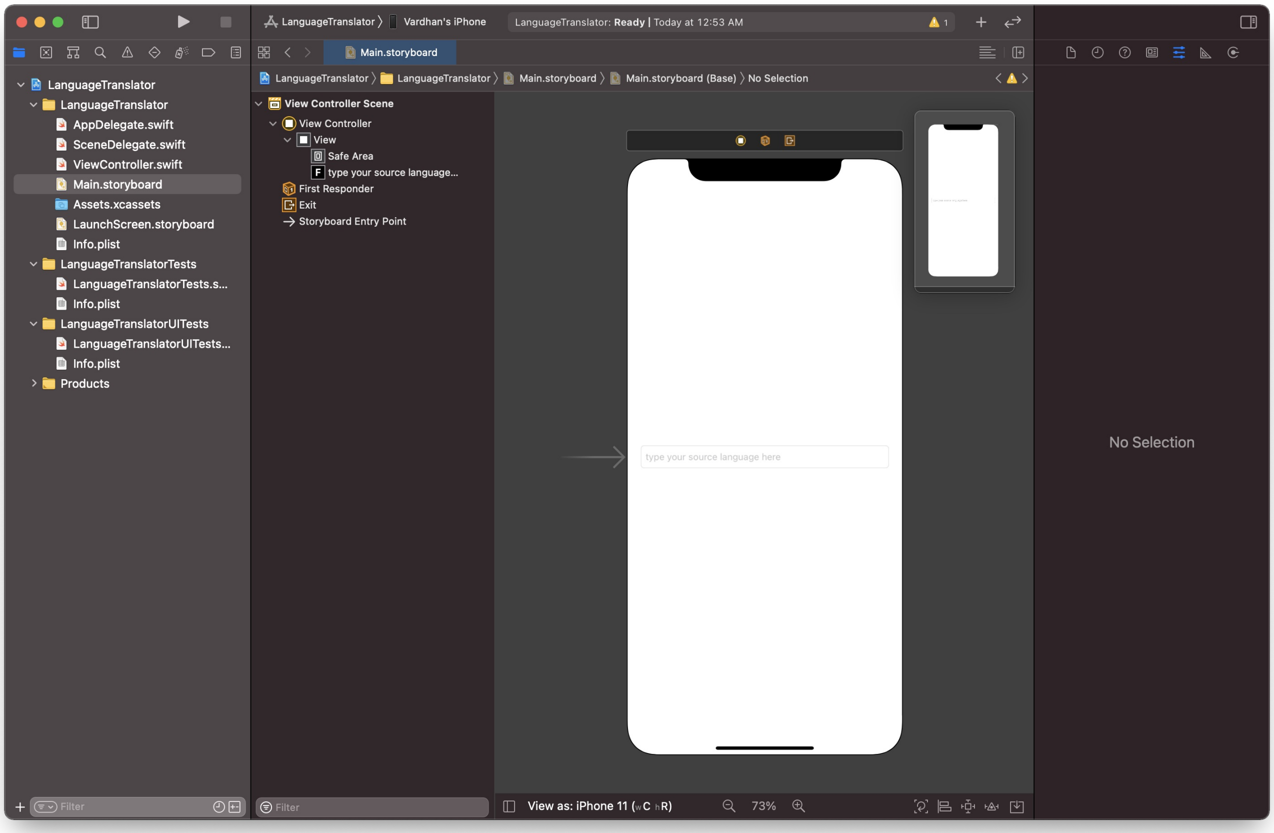

LanguageTranslator 〉 Vardhan's iPhone
LanguageTranslator: Ready | Today at 12:53 AM
Main.storyboard
LanguageTranslator 〉 LanguageTranslator 〉 Main.storyboard 〉 Main.storyboard (Base) 〉 No Selection
LanguageTranslator
LanguageTranslator
AppDelegate.swift
SceneDelegate.swift
ViewController.swift
Main.storyboard
Assets.xcassets
LaunchScreen.storyboard
Info.plist
LanguageTranslatorTests
LanguageTranslatorTests.s...
Info.plist
LanguageTranslatorUITests
LanguageTranslatorUITests...
Info.plist
Products
View Controller Scene
View Controller
View
Safe Area
type your source language...
First Responder
Exit
Storyboard Entry Point
type your source language here
No Selection
Filter
Filter
View as: iPhone 11 (wC hR)
73%

Figure 7-14: Adding a Text Field for Source Language Input

Add a Button

Next, we need a “submit” style button, where the user can tell the app that they’re done typing. For a messaging app, this would be like the “send” button; or, for a searching app, this would be the “search” button. Make sure you put this button where the keyboard won’t block it, since we won’t cover “hiding the keyboard“ in this chapter.

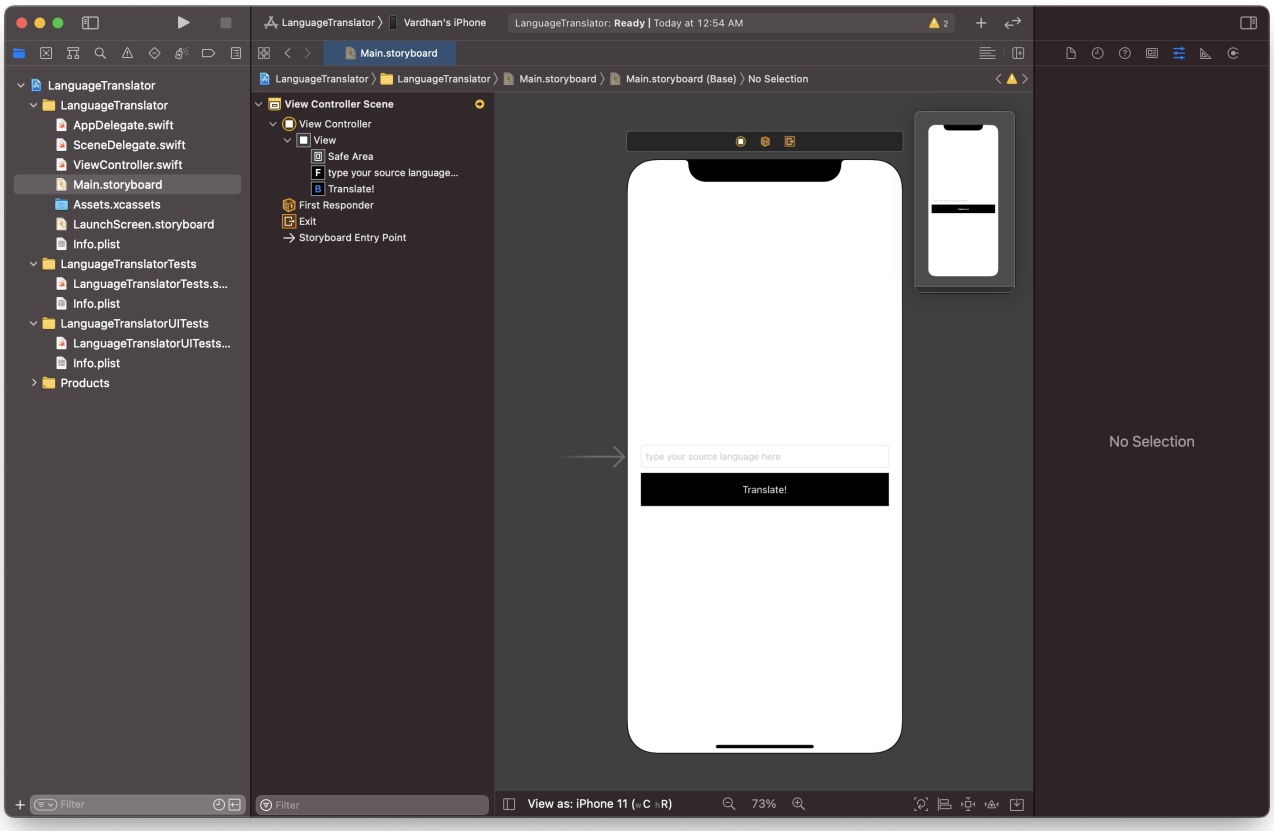


Figure 7-15: Adding a Translate Button

Add a Label

We’ll also need to show output to the user, so add a label to your view. I used a stack view to layout everything correctly, but you can use auto layout just as well. If you’d like, you can change the color or font, but that’s unnecessary for the machine learning part of this chapter.

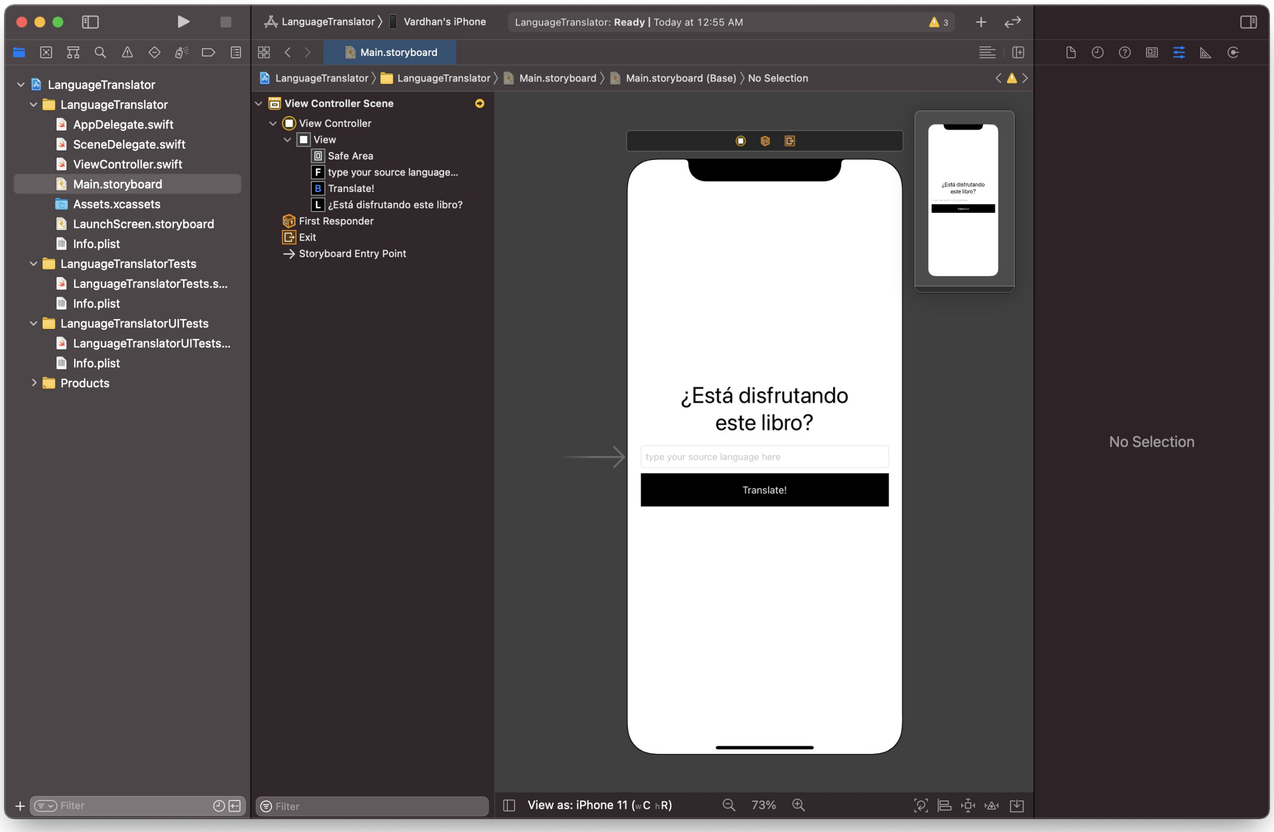


Figure 7-16: Adding a Label for Translated Result String

Connect Interface to Code

As you've done for several times throughout this book, you'll need to connect the user interface elements to your Swift code, so that you can reference them later. As usual, we'll first type out the references, and then, we'll go back and connect them in the `Main.storyboard` file.

Create IBOutlets

First, we'll connect the `@IBOutlets`, which provide us information about the current state of the user interface object. This will apply to both the text field and the label, so declare the following inside the `ViewController` class in the `ViewController.swift` file:

```
@IBOutlet weak var label: UILabel!
@IBOutlet weak var textField: UITextField!
```

Create IBAction

For the “submit” button we created earlier, we want a function to get called when the button is pressed. To do this, we can create a method with the `@IBAction` tag. Conventionally, Swift developers place their `@IBAction` methods at the end of the class. So, type the following before the closing curly brace in the `ViewController.swift` file:

```
@IBAction func submitButtonTapped() {
    // your code here.
}
```

When connected, the `submitButtonTapped()` method will get called whenever the user taps the “submit” button on their screen. In the previous section, we were performing our Firebase actions whenever a new frame was captured from the live video feed. And in this section, we'll do machine learning processing when the “submit” button is tapped.

Connect Interface Elements

Finally, head to your `Main.storyboard` file and two-finger click on each of the user interface elements. Then, drag the circle next to the appropriate variable from the popup to the element. When the circle is filled, you know that they've been connected successfully.

Once that's done, you're finished! You now have the skeleton of an app you can use for text-based machine learning purposes, like the translation app you're about to create.

Language Translation

You've likely used a language translator in the past — whether it was Google Translate in a foreign country or an old-school pocket translator. You may have wondered how these tools function. In Chapter 5, you learned about the inner workings of language processing tools, and in this chapter, you'll learn how to build your own language translator.

Finish Housekeeping

As you may have guessed, there are a few things we need to do before building our language translator. Luckily, since there's no live video feed, all you need to do is to install one additional pod before getting started.

Add Pod

Open your `Podfile` under the `Pods` target of your workspace. You can also open it in the Finder directly or via the Terminal. Then, add an additional pod under the `pod 'Firebase'` line like this:

```
pod 'Firebase/MLNLTranslate'
```

This will add the appropriate framework needed for translation via Firebase.

Install Pod

Then, save your Podfile and install the pod. This is the same procedure as you've been doing, but as you know, practice makes perfect! Navigate to your project directory via the Terminal and enter the following command to install:

```
pod install
```

Import Firebase

When that's been completed, go back into the `ViewController.swift` file and add the following import statement after `import UIKit`. That's all the housekeeping you have to do for this project!

```
import MLKit
```

Translating Strings

Now, you're ready to actually start translating strings from one language to another. In essence, you'll need to download translation models, wait for them to be installed using multithreading, and then create your translation request and display it to the end user.

Language Options

The first logical step in creating a translation tool is specifying the languages you're translating between. Since I know both Spanish and English, I'll be using those as my target and source languages, respectively. Define your languages in the `submitButtonTapped()` from before like this:

```
let options = TranslatorOptions
    (sourceLanguage: .english, targetLanguage: .spanish)
```

This line of code creates an instance of `TranslatorOptions` and specifies both a target and source language. Both parameters are of type `TranslateLanguage`. In the past, the enum for languages was more cryptic, but thankfully, they've made it more obvious. For "English," you simply say `.english`!

Create Translator

Next, you can use your language options to create a translator. I suggest naming this so that you can easily understand the language that it uses. In the future, you may choose to expand the app to support multiple languages, and for this purpose, naming is especially important. Define your translator as follows in the same `submitButtonTapped()` method:

```
let spanishTranslator = Translator.translator(options: options)
```

Here, the `spanishTranslator` constant contains an instance of a `translator`, which converts English into Spanish (through the parameter `options`). You could create an array of these translators if you wanted to use multiple languages, in a similar fashion -- just make sure you change the source and target languages appropriately!

Download Translation Model

To continue with using the translator you created, you need to first download the appropriate translation model from Firebase. You also have the option of specifying the conditions under which a model can be downloaded, including permitting the use of cellular data. Here, we won't specify anything:

```
spanishTranslator.downloadModelIfNeeded() { error in
    // your code goes here.
}
```

When the model has been downloaded successfully, or if it returns with an error, the code inside the body of the closure will execute; perfect for performing our actual task.

Translate your String

Finally, you're ready to translate the user's string into your target language. Inside the body of the `downloadModelIfNeeded()` closure, enter the following code to make the translation request:

```
spanishTranslator.translate(self.textField.text ?? "")
    { translatedText, error in
        // your code goes here.
}
```

The `translate()` method takes in the target string as a parameter, and we are passing in the text entered in the text field. If there's no text (i.e. `self.textField.text` is `nil`), it will pass in an empty string using the nil coalescing operator (`??`). The resulting string, if applicable, will be passed through the `translatedText` variable.

Display Result

Now that the translator has given you a result, you can display it to the user using the label you've made. To do this, first check whether a result exists, and if it does, display it to the user. Do this inside the closure of the `translate()` function.

```
if let translatedText = translatedText {
    self.label.text = translatedText
}
```

That may have been too many closures to follow, so here's how your `submitButtonTapped()` method should look like by the end:

```
@IBAction func submitButtonTapped() {
    let options = TranslatorOptions(sourceLanguage: .en, targetLanguage: .es)
    let spanishTranslator = NaturalLanguage.naturalLanguage().translator(options:
options)

    spanishTranslator.downloadModelIfNeeded() { error in
        spanishTranslator.translate(self.textField.text ?? "") { translatedText, e
rror in
            if let translatedText = translatedText {
                self.label.text = translatedText
            }
        }
    }
}
```

Now, when you run your app, enter some text and press "submit." The first time you do this, it might take a minute, since it's downloading the model from Google ML Kit. However, after the first time, you'll get near-instant translation.

Conclusion

In this chapter, you learned even more about machine learning through a slightly different approach: cloud-based machine learning. First, you learned how to setup Firebase through a dependency manager called CocoaPods. Next, you connected your Xcode project to Firebase. In one part of the chapter, you learned about different image processing techniques, such as barcode scanning and image classification.

Later in the chapter, you learned to create a text-based machine learning and a language translation tool. By now, you should have an excellent grasp on using Firebase's machine learning tools and have three fully functional apps!

Chapter 8
Updatable ML Models for On-Device Training

In the previous chapter, you learned to integrate cloud-based machine learning into your app through Firebase. At the beginning of the chapter, you learned to use CocoaPods to implement Firebase into your Xcode project. Then, you built your fully functioning own barcode scanner, image classifier, and language translation app. Through creating these apps, you learned a lot about cloud-based machine learning and Firebase's APIs in general.

In this chapter, you'll be discovering a more advanced Core ML topic: updatable models. As the name suggests, updatable models are machine learning models which can be augmented over time with user-provided data. In the beginning of the chapter you'll learn all about updatable models and how they work. Within this, you'll learn about the intricacies of how the user interacts with updatable models and how they work from a developer's standpoint.

Later, you'll be learning about tools available on the internet to help you train, source, and interact with updatable models. You'll also obtain open-source software from Apple and refresh your skills on Jupyter Notebooks and Python. Finally, we'll dissect the somewhat complicated code line-by-line to make sure you have a solid understanding of what's going on.

8-1 How Updatable ML Models Work

In this chapter, you'll learn about updatable models. These types of models provide significant benefits over traditional models for certain use cases, since they can adapt to each user's individual needs. Each user is different, and these types of models can improve and personalize each user's experience. By creating such models, we're able to provide the user a more accurate result.

In this section, you'll learn the benefits of an updatable model and how it works. First, you'll discover how they work from a user's standpoint. Then, you'll learn about transfer learning in general. Finally, you'll get a glimpse of the challenges of implementing such technology.

User Experience

From a user's standpoint, updatable models can improve the user experience significantly. Updatable models not only allow for customization, but also make the user feel in control of your app, instead of having presets. Further, it allows for the app experience to improve over time, even if you never pushed an update to the app. Let's look at some examples of how this could work in an app.

Handwritten Symbols

As presented at WWDC, one great example of updatable models being used in an app is a handwritten symbol recognizer. Instead of having the user learn specific ways of drawing Emojis, updatable models allow the user to train the app. By shipping a basic (but updatable) model with your app, you're allowing the user to decide how your app should function, which can be invaluable in some use-cases.

Custom Image Classifier

Another great use of updatable models is for classification — especially objects which aren't standard. For instance, if a user wanted to detect their car keys in an image, it would be nearly impossible to train a model that worked on everyone's car keys. With all the makes and models of cars available, making a generalized model would greatly limit the number of users your app could reach. With updatable models, a user could train the app to recognize their own keys, only requiring a basic model from the start.

Voice Recognition

Most modern voice assistants such as Alexa and Google Assistant are equipped with a voice recognition feature, which allows them to personalize their interactions with each user. Under-the-hood, an updatable model is being used to train on an individual user's voice. So, when it asks you to say *what's the weather like?* or *show me my contacts?* during the set up, you're actually training a machine learning model to recognize your voice.

Transfer Learning

Digging a little bit deeper, updatable models are a lighter-weight version of transfer learning techniques. In other words, when a user inputs data into your app, the algorithms updating the models are strikingly similar to those used when training models using transfer learning, and the underlying technique used by Create ML and other drag-and-drop model training models. Let's learn about transfer learning to get a better understanding of updatable models.

What is Transfer Learning?

As an increasingly popular way to train machine learning models, transfer learning uses an existing model, often trained with a large dataset, and repurposes it to be used for a more specific purpose. For example, if you wanted to create a smartphone classifier, you could use a standard object recognition model and train on a small dataset of smartphones. This would result in an end-product which would be able to detect smartphones.

Benefits of Transfer Learning

While it might not seem obvious at first, transfer learning is an excellent technique for model training and can potentially save you a lot of time and money. Whether it's allowing you to use smaller datasets to train on, or requiring less time to train, transfer learning is a great technique to use. Let's look at some of the benefits of transfer learning:

Smaller Datasets

One of the most significant benefits of using transfer learning to train your models is the ability to use smaller datasets to achieve the same results as using larger ones. So, instead of having to use a million images to get an accurate result, you can use a pre-trained model with a million images and then only provide a hundred of your own. This way, you'll achieve the same result, but with significantly less data.

Shorter Training Times

Since you're using a smaller dataset and the model is already pre-trained, the amount of time it takes to train a model using transfer learning is significantly less. This means that if you're training in the cloud or using a pay-per-hour service, you'll save both time and money through transfer learning. This is another reason that transfer learning is becoming increasingly popular in the machine learning community.

Higher Accuracy

Finally, since you're using a pre-trained model, often with millions of images, you'll inevitably get better results when training with transfer learning. By taking advantage of Google or Apple's models, you're able to attain much higher accuracy than you'd be able to on your own; unless you have millions of custom images to train on.

Updatable Models at a Glance

Finally, before you get to create your own updatable model, let's take a look at how it works under-the-hood. As you learned earlier, updatable models operate similarly to models trained with transfer learning; however, they are not the same. It's worth taking the time to explore how updatable models work in the context of iOS development and Core ML.

Obtaining Models

Any model in the Core ML format cannot be used as an updatable model. As such, you need to download updatable models for use in your apps *or* you need to use a conversion tool to create an updatable model. At the time of writing, it is not possible to train updatable models using the tools provided by Apple; however, you can create these models (or find them online) to easily configure to work with Core ML.

Accepting User Input

When your model is deployed, you can prompt your users to provide data to update the model. You will then use this data to retrain the updatable layers of your Core ML model. Since the principles of updatable models' use are similar to that of transfer learning, your pre-trained model won't require a significant amount of data from the users. After you receive data from the user, you can have your model update on a background thread while the user continues to use their device.

On-Device Training

One of the most praised features of updatable models in Core ML is their ability to train on-device. This might not seem like a big deal at first, but on-device model training is a boon for both the privacy of the user and the operating costs of the app. By allowing for on-device model updating, Core ML enables an easier and faster approach which wasn't possible before.

8-2 Open-Source Training Code

In the previous section, you learned about updatable models at a glance. First, you learned about how updatable models work from a user interface standpoint. Then, you examined the similarities between updatable models and existing machine learning techniques by learning about transfer learning. Finally, you explored some of the specifics of updatable models with Core ML to get ready to implement them.

In this section, you'll be training an updatable nearest neighbor classifier through open-source software provided by Apple. Then, you'll convert this model into an updatable model to use with Core ML. For educational purposes, we'll be looking at an extreme case of updatable models: one in which the model can only provide one output label, until it is eventually updated. We'll also explore a handwritten symbols recognizer. This will clearly illustrate the difference between a boilerplate and an updated model.

Obtaining the Code

Unlike many of the other chapters in this book, you'll be using open-source software instead of writing code from scratch. You'll get a flavor of the software development world, where developers publish open-source libraries for others to build on. This helps the community grow as a whole, instead of individual developers writing code in isolation.

Exploring Open-Source Options

At the time of writing, there aren't many end-to-end options when it comes to open-source software for Core ML updatable models. This is likely because the technology isn't fully utilized yet. That said, Apple provides a handful of Jupyter Notebooks which can be used to create and export updatable models for Core ML. To see Apple's options, check out their GitHub repository. Let's take a look at them here:

MNIST Dataset Model

As explored earlier in the book, the MNIST dataset is designed to train handwritten number classifiers. While the dataset is designed for numbers, it can be used for other characters by making it an updatable model. If this is your intent, you can use this Jupyter Notebook, which trains a Keras model on MNIST and creates two updatable layers in the exported Core ML model. We'll be exploring this one in more detail later on.

Tiny Drawing (Linked) Pipeline Model

This Jupyter Notebook consists of a model trained on user-sourced drawings of simple drawings and sketches. This model will later be exported as an updatable model for use in Core ML. This is relevant for similar uses as the MNIST notebook. However, it is more angled towards small sketches and drawings as opposed to symbols and characters.

Nearest Neighbor Classifier

Being the most bare-bones example on list, the nearest neighbor classifier Jupyter Notebook provides a nearly blank canvas for you to work. In this chapter, this is another example we'll be exploring. By default, this classifier only outputs a default label. After you've updated it through your iOS client, you'll be able to see the difference.

Using Third-Party Tools

Since Apple's own tools don't allow you to directly train updatable models (yet!), you need to use third party tools to train these models. In the above examples, the models were trained manually. However, if you have models already trained in other formats, you can use the `coremltools` framework to convert them into updatable Core ML models.

Popular tools include Keras and Caffe, which allow you to create more fine-tuned models than you could with Core ML. With `coremltools`, you can easily use them as native Core ML models, which is especially convenient for updatable models. There are also a significant number of open-source models trained in these formats. Using Apple's documentation, you can convert them to the Core ML format.

Preparation for Training

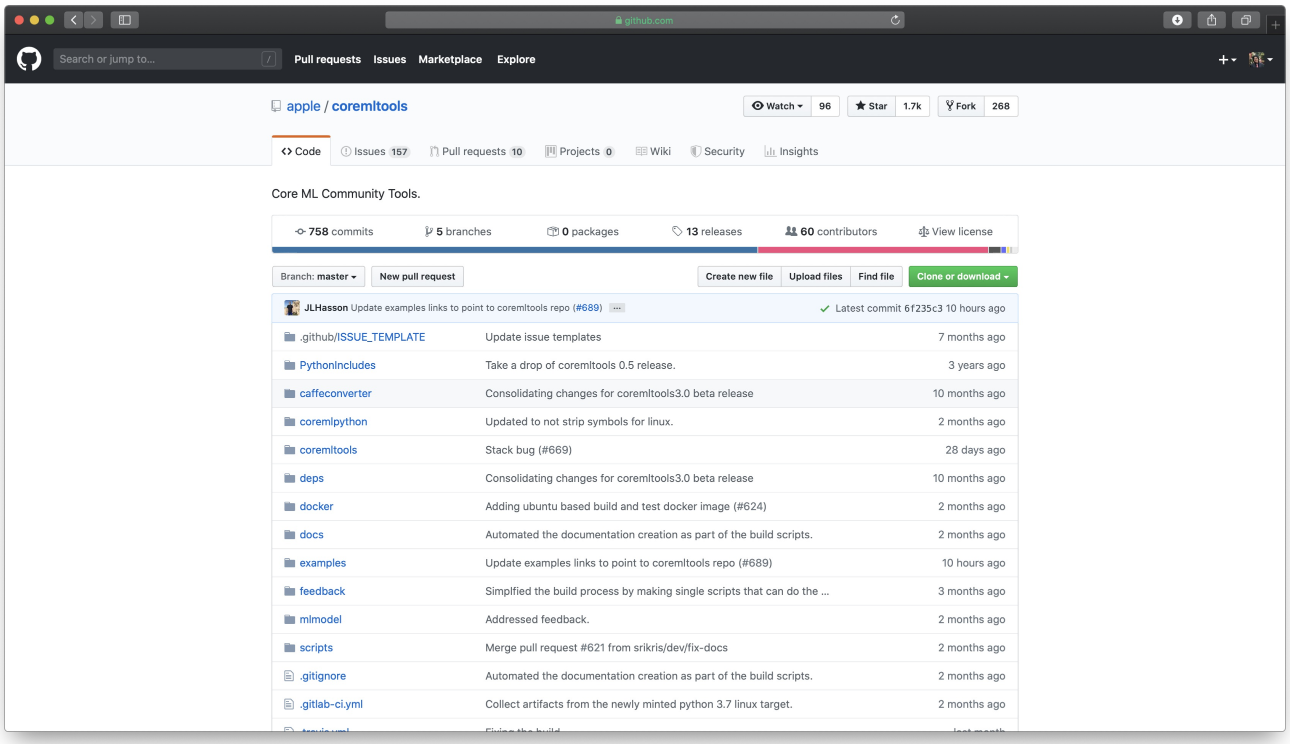
Before you dive into the code and learn how an updatable model is built, it's worth refreshing your Python skills. In Chapter 3, you learned about Python and Jupyter Notebooks. It's time to put your skills to the test! Since we won't be covering the implementation of updatable models in this chapter, the following steps are optional but highly recommended for you to follow. If you don't want to follow along with the installation, you can view the Python code in the GitHub preview for `.ipynb` files.

Installing the Code

As the industry standard for Python scripts, you'll be using Jupyter Notebooks a lot as you continue on your machine learning journey. If you haven't read the chapter yet, I recommend you to do so before trying to install the Jupyter Notebooks here. If you don't want to train models in this chapter, you can follow along anyway as this is more of a "theory" chapter than a hands-on one.

If you don't have a GitHub account, you'll need to create one in order to install the GitHub repository. If you already have one, go ahead and login. You'll be given options to either create a new account or to authenticate with Google, among other choices.

Then, you'll need to install the Core ML Tools repository. You can do this by clicking the green **Clone or Download** button and then **Download ZIP**. This will download the entire `coremltools` repository to your **Downloads** folder.


github.com

Search or jump to... Pull requests Issues Marketplace Explore

apple / coremltools

Watch 96 Star 1.7k Fork 268

Code Issues 157 Pull requests 10 Projects 0 Wiki Security Insights

Core ML Community Tools.

758 commits | 5 branches | 0 packages | 13 releases | 60 contributors | View license

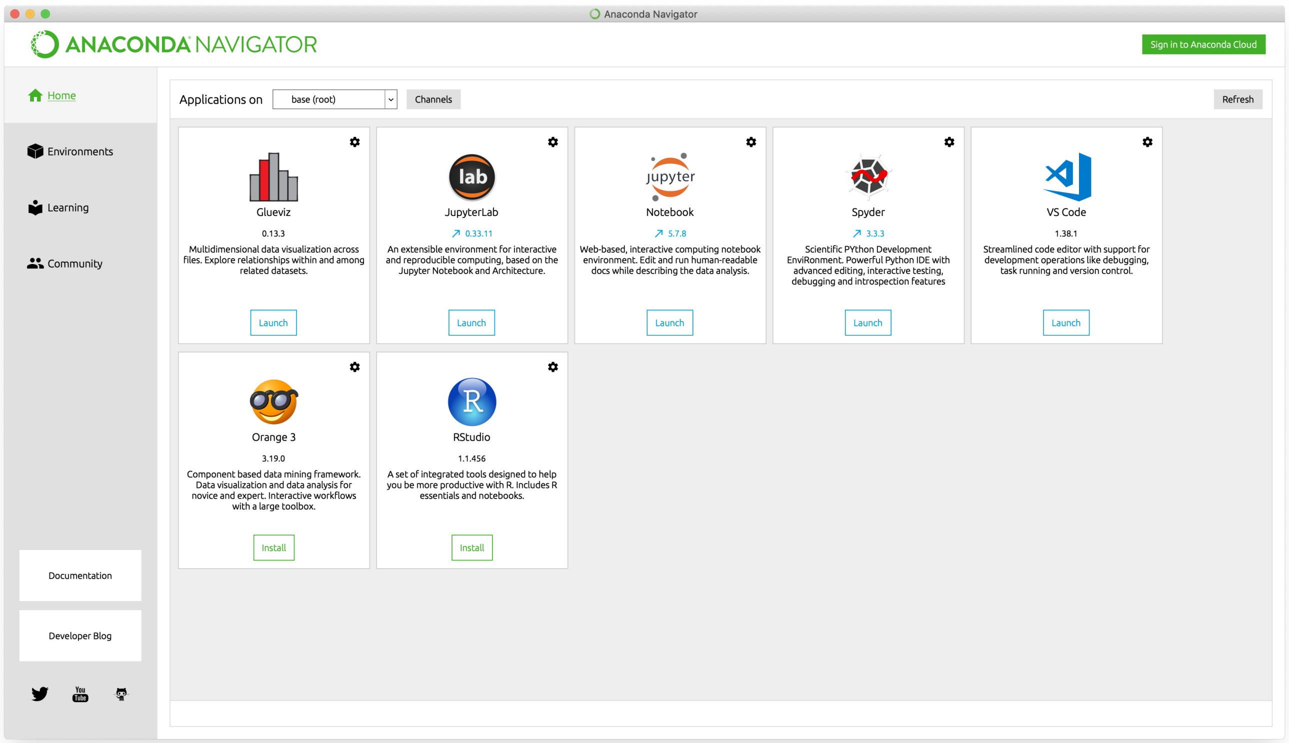
Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

JLHasson Update examples links to point to coremltools repo (#689) ... Latest commit 6f235c3 10 hours ago

.github/ISSUE_TEMPLATE	Update issue templates	7 months ago
PythonIncludes	Take a drop of coremltools 0.5 release.	3 years ago
caffeconverter	Consolidating changes for coremltools3.0 beta release	10 months ago
coremlpython	Updated to not strip symbols for linux.	2 months ago
coremltools	Stack bug (#669)	28 days ago
deps	Consolidating changes for coremltools3.0 beta release	10 months ago
docker	Adding ubuntu based build and test docker image (#624)	2 months ago
docs	Automated the documentation creation as part of the build scripts.	2 months ago
examples	Update examples links to point to coremltools repo (#689)	10 hours ago
feedback	Simplfied the build process by making single scripts that can do the ...	3 months ago
mlmodel	Addressed feedback.	2 months ago
scripts	Merge pull request #621 from srikris/dev/fix-docs	2 months ago
.gitignore	Automated the documentation creation as part of the build scripts.	2 months ago
.gitlab-ci.yml	Collect artifacts from the newly minted python 3.7 linux target.	2 months ago

Figure 8-1: Core ML Tools GitHub Repository

The next steps is to use Anaconda Navigator and launch the Jupyter Notebooks program. This will allow you to view and edit the Python code you just downloaded from the GitHub repository.



Multidimensional data visualization across files. Explore relationships within and among related datasets.

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Streamlined code editor with support for development operations like debugging, task running and version control.

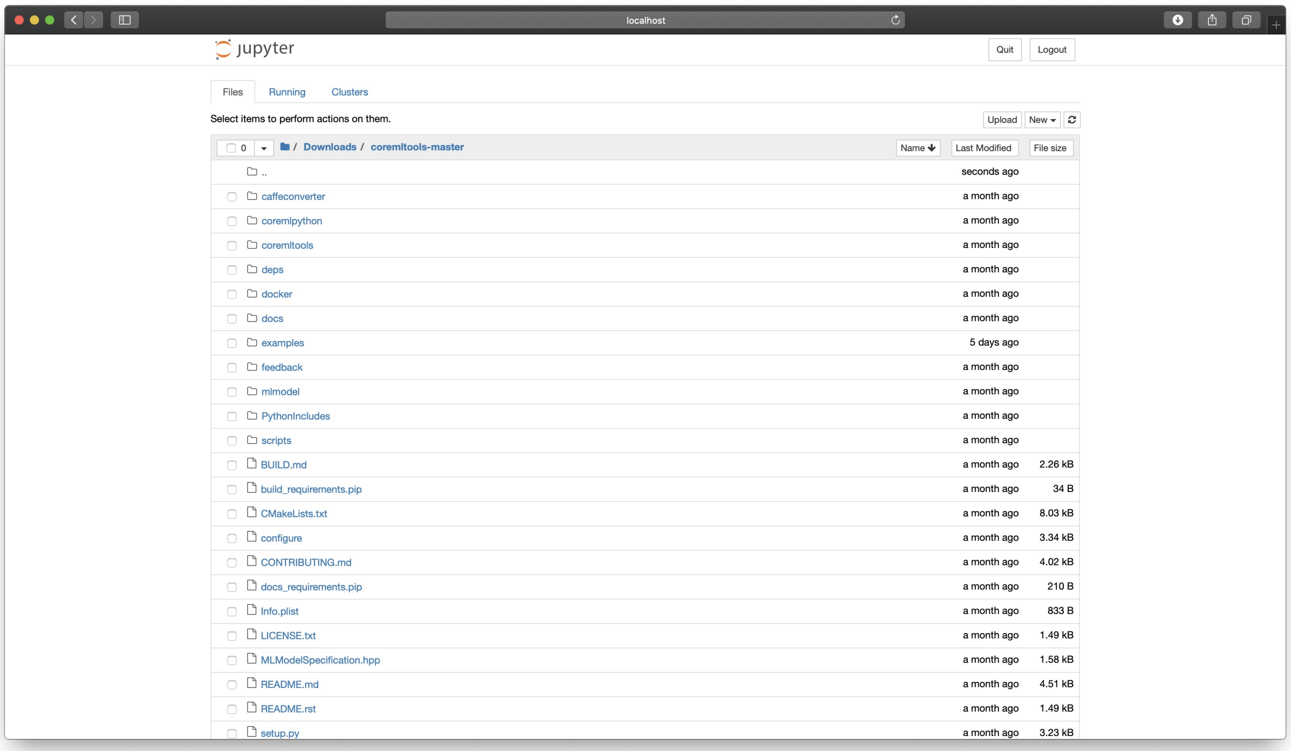
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 8-2: Anaconda Navigator

Launch Jupyter Notebooks

You can use your skills from Chapter 3 to open Jupyter Notebooks. In case you've forgotten, you need to first launch **Anaconda Navigator**, and then click **Launch** under the Jupyter Notebook task. If you do not have **Anaconda** installed on your computer, you'll need to follow along with this chapter using the GitHub preview.



Select items to perform actions on them.

/ Downloads / coremltools-master

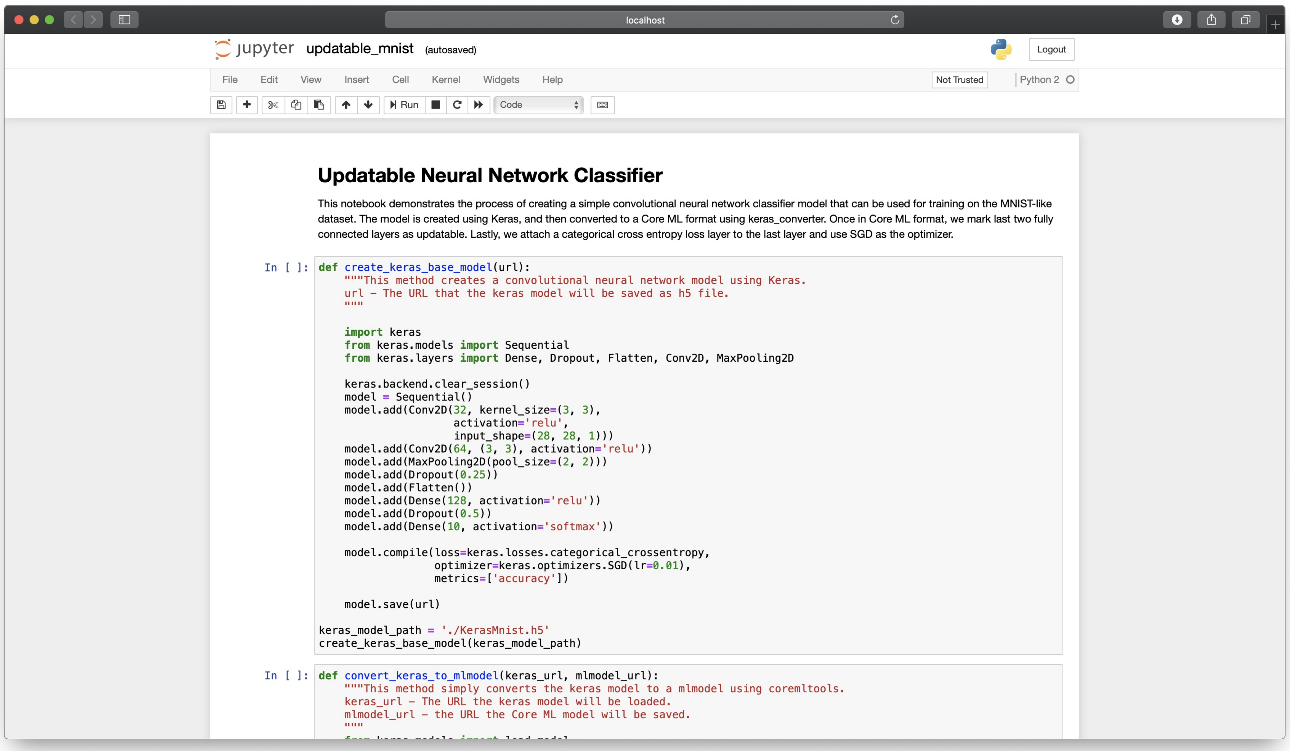
Name	Last Modified	File size
..	seconds ago	
caffeconverter	a month ago	
coremlpython	a month ago	
coremltools	a month ago	
deps	a month ago	
docker	a month ago	
docs	a month ago	
examples	5 days ago	
feedback	a month ago	
mlmodel	a month ago	
PythonIncludes	a month ago	
scripts	a month ago	
BUILD.md	a month ago	2.26 kB
build_requirements.pip	a month ago	34 B
CMakeLists.txt	a month ago	8.03 kB
configure	a month ago	3.34 kB
CONTRIBUTING.md	a month ago	4.02 kB
docs_requirements.pip	a month ago	210 B
Info.plist	a month ago	833 B
LICENSE.txt	a month ago	1.49 kB
MLModelSpecification.hpp	a month ago	1.58 kB
README.md	a month ago	4.51 kB
README.rst	a month ago	1.49 kB
setup.py	a month ago	3.23 kB

Figure 8-3: Jupyter Notebooks File Navigator

Then, you'll see a browser window opened with your files inside. If none of these sounds familiar, I highly recommend referring back to Chapter 3. This will help you hone in on your Jupyter Notebooks skills and learn Python, the language that the code is written in for this chapter.

Open the MNIST Notebook

Using the file explorer in the Jupyter Notebooks window, navigate to `coremltools-master` > `examples` > `updatable_models` and then open the `updatable_mnist.ipynb` file. Next, you should see the Python code appear. From here, you'll be able run individual cells, make changes, and add annotations.


localhost
jupyter updatable_mnist (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 2
Run
Code

Updatable Neural Network Classifier

This notebook demonstrates the process of creating a simple convolutional neural network classifier model that can be used for training on the MNIST-like dataset. The model is created using Keras, and then converted to a Core ML format using keras_converter. Once in Core ML format, we mark last two fully connected layers as updatable. Lastly, we attach a categorical cross entropy loss layer to the last layer and use SGD as the optimizer.

```
In [ ]: def create_keras_base_model(url):
            """This method creates a convolutional neural network model using Keras.
            url - The URL that the keras model will be saved as h5 file.
            """

            import keras
            from keras.models import Sequential
            from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

            keras.backend.clear_session()
            model = Sequential()
            model.add(Conv2D(32, kernel_size=(3, 3),
                             activation='relu',
                             input_shape=(28, 28, 1)))
            model.add(Conv2D(64, (3, 3), activation='relu'))
            model.add(MaxPooling2D(pool_size=(2, 2)))
            model.add(Dropout(0.25))
            model.add(Flatten())
            model.add(Dense(128, activation='relu'))
            model.add(Dropout(0.5))
            model.add(Dense(10, activation='softmax'))

            model.compile(loss=keras.losses.categorical_crossentropy,
                          optimizer=keras.optimizers.SGD(lr=0.01),
                          metrics=['accuracy'])

            model.save(url)

        keras_model_path = './KerasMnist.h5'
        create_keras_base_model(keras_model_path)
```

```
In [ ]: def convert_keras_to_mlmodel(keras_url, mlmodel_url):
            """This method simply converts the keras model to a mlmodel using coremltools.
            keras_url - The URL the keras model will be loaded.
            mlmodel_url - the URL the Core ML model will be saved.
            """
```

Figure 8-4: MNIST Jupyter Notebook

8-3 Training the Model

In the previous section, you learned about open-source options for training and using updatable models with Core ML. You also learned about using third party libraries and converting them to the Core ML format. Finally, you downloaded Apple's GitHub repository and used your existing Jupyter Notebooks knowledge to open it up with Anaconda Navigator.

In this section, you'll learn to train an updatable model using the open-source code you obtained from Apple's GitHub repository. While we won't be going over the implementation of such a model in an iOS app, you will learn a lot about how to train updatable models using Python and `coremltools`.

Code Explained

As mentioned earlier, this chapter is more of a theory chapter than a hands-on one. Even though we'll be building an updatable model, the main part is understanding the code required to do so. In this section, I'll be going over the code line-by-line to dissect how Apple has structured their Jupyter Notebook. To get started, open up the MNIST Jupyter Notebook from Apple's example repository.

Convolutional Neural Network with Keras

The very first step in training an updatable model is to create a base model — one which will be built upon. In this case, the base model will be a convolutional neural network built using Keras and trained using a dataset similar to MNIST.

In case you weren't familiar with, the MNIST database is a publicly available dataset which contains handwritten numbers and their corresponding labels. Also, a convolutional neural network is a machine learning model architecture which is commonly used for image processing. All of the image classification models we've trained in this book are convolutional neural networks.

Defining Method

To keep the code organized, the author of the Jupyter Notebook created a method to handle the creation of the Keras base model and appropriately named it `create_keras_base_model()`. This model was defined in the first cell of the notebook and called later:

```
def create_keras_base_model(url):
```

Importing Frameworks

As usual, it's necessary to import the appropriate frameworks. In this case, we need to import Keras and some specific sub-frameworks. These are necessary only in the creation of the base model, so they exist only within the scope of the `create_keras_base_model()` method declaration.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout,
    Flatten, Conv2D, MaxPooling2D
```

Specifically, we need to import the main `keras` framework and then import the necessary tools to build a convolutional neural network from scratch.

Building a Base CNN

The next step is to build a convolutional neural network, or CNN, from scratch. To do this, we'll need to clear the Keras graph. To ensure the maximum efficiency, this will make sure we start on a blank canvas:

```
keras.backend.clear_session()
```

Then, we create a sequential model and use `model` to refer to it in the code later on. In Keras, a Sequential type model is a linear stack of layers (which is the most types in image classification models).

```
model = Sequential()
```

The CNN machine learning architecture is made up of layers. Each layer performs a specific task to analyze the input image, and in the following code, we're defining these layers from scratch. When you build CNN's with Turi Create and Create ML, however, this step is done using a pre-determined model architecture.

```
model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(28, 28, 1)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Compile and Save the Model

The logical last step in the `create_keras_base_model()` method is to compile the newly built model and then save it. Compiling the model requires you to specify a loss function, which tells you how well (or how poorly) the model is performing. In this case, Apple uses a standard loss function called Cross Entropy. It isn't necessary to understand the details, but you can learn more about loss functions if you'd like.

```
model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.SGD(lr=0.01),
                  metrics=['accuracy'])
model.save(url)
```

There's also a specified optimizer, called Stochastic Gradient Descent, often regarded as the most accurate. This tells Keras which algorithm to use while compiling the model. The other parameter being specified in the call to the `compile()` method is the `metrics` parameter. This tells Keras to optimize the model for accuracy. Finally, we save the model to the passed-in location URL.

Call the Method

After creating a long, complex method to create a base model for the MNIST classifier, we must call the method. First, the URL is stored for `./KerasMnist.h5` and then passed into the model.

```
keras_model_path = './KerasMnist.h5'
create_keras_base_model(keras_model_path)
```

Note that this is done *outside* of the model declaration, so it is no longer indented. If you aren't aware, Python uses indentation to mark blocks of code instead of curly braces like in Swift.

Convert Keras to Core ML

Now that we (or Apple) have saved a Keras model, we need to convert it into the Core ML format in order to continue with our mission. Thankfully, this is pretty straightforward with `coremltools`.

Create a Method

To organize the code, Apple has split this task up into another method. This time, the method takes in two URLs: one where the Keras model is and one where the Core ML models should be.

```
def convert_keras_to_mlmodel(keras_url, mlmodel_url):
```

Load Keras Model

Inside the newly created method, you take the Keras model URL which was passed in through the `convert_keras_to_mlmodel()` method and load it into `keras_model`. To do this, you'll need to import the `load_model` tool from Keras.

```
from keras.models import load_model
keras_model = load_model(keras_url)
```

Structure Core ML Model

Once the Keras model has been successfully loaded to a variable, the Core ML model must be setup. First things first, the Keras converter is imported from the `coremltools` library.

```
from coremltools.converters import
    keras as keras_converter
```

Then, Core ML wants to know the possible labels the model can output. Since we're using MNIST, which is a handwritten digits dataset, we'll simply list the digits from zero through nine and store it in `class_labels`.

```
class_labels =
    ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

We then call the `convert()` method from the converter we imported earlier. This asks us to name our input and output and passes in the class labels from earlier.

```
mlmodel = keras_converter.convert(keras_model,
    input_names=['image'],
    output_names=['digitProbabilities'],
    class_labels=class_labels,
    predicted_feature_name='digit')
```

Once that's done, we can save the newly created Core ML model stored inside of `mlmodel` to the URL passed into the `convert_keras_to_mlmodel()` method.

```
mlmodel.save(mlmodel_url)
```

Call the Method

Similar to the creation of the Keras model, we need to finally call the `convert_keras_to_mlmodel()` method. To do this, we'll need to pass in the `keras_model_path` from the previous cell and create a path to store the Core ML model. Then, we'll pass in both of them as parameters.

```
coreml_model_path = './MNISTDigitClassifier.mlmodel'
convert_keras_to_mlmodel(keras_model_path , coreml_model_path)
```

Again, make sure you do this outside the method declaration by un-indenting it. You wouldn't want to call a method inside a method!

Inspecting Layers Before Updating

With the power of a Jupyter Notebook, Apple uses the `inspect_layers` function to check each layer to know whether it's updatable or not. While this step didn't require to create an updatable model, it is definitely important for understanding.

Import Frameworks

Since `coremltools` isn't yet imported, we need to do that before attempting to inspect the layers of the convolutional neural network.

```
import coremltools
```

Load Model

Then, we must load a model to inspect. Since we have the path of the new Core ML model stored in `coreml_model_path` from the previous cell, we can use that to create a specification.

```
spec = coremltools.utils.load_spec(coreml_model_path)
```

Then, we need to initialize a `NeuralNetworkBuilder` with the specification. In `coremltools` , a `NeuralNetworkBuilder` builds a neural network by its layers and from its specification.

```
builder = coremltools.models.neural_network.NeuralNetworkBuilder
    (spec=spec)
```

Inspect Layers

Since we'll be making the last three layers of the neural network updatable, those are the ones that we should inspect. By calling the `inspect_layers()` method on `builder` and passing in `3`, we can achieve this.

```
builder.inspect_layers(last=3)
```

The output of this, when run, shows that all three layers are not updatable.

Setting Input Types and Descriptions

The next cell is more of a "housekeeping" one. Here, we define what kind of input is allowed for the model and then create descriptions to document the inputs and outputs for the end-user. By looking at Apple's Jupyter Notebook, we're learning not only how to create updatable models, but also the good practices in code.

Inspect Input

Before making any changes to the type of input accepted by the model, it's a good idea to check what the input type was before you make the change. Using the same `builder` from the previous cell and the `inspect_input_features()` method, we're checking the current input type.

```
builder.inspect_input_features()
```

Setting the Input Resolution

To set the input resolution, you'll need to create a specification from the `NeuralNetworkBuilder`. This tells us the specification it was originally created with (see previous cell).

```
neuralnetwork_spec = builder.spec
```

Then, using that specification, we can set the width and height of the image to be a 28x28 sized square.

```
neuralnetwork_spec.description.input[0].type.imageType.width = 28
neuralnetwork_spec.description.input[0].type.imageType.height = 28
```

Since we're doing handwritten digit recognition, we only need a small, square image. This is how the data in the MNIST dataset looks. And as you learned earlier in the book, it's important to structure your input data just like your training data.

Take the Grayscale Version

In a handwritten digit, there's not a whole lot in terms of colors. For this reason, we'll only take the grayscale version of the image. To do this, we first import the `FeatureTypes_pb2` library from `coremltools` and then use the `GRAYSCALE` colorspace. Then, we set the colorspace of the input image to the `grayscale` colorspace we just defined.

```
from coremltools.proto import FeatureTypes_pb2 as _FeatureTypes_pb2
grayscale = _FeatureTypes_pb2.ImageFeatureType.ColorSpace.
    Value('GRAYSCALE')
neuralnetwork_spec.description.input[0].type.imageType.
    colorSpace = grayscale
```

If you're not familiar with colorspaces, they define the structure with which the colors stored in your images. For example, in a grayscale image, there's no need to store the red, green, and blue components of an image, so they are stripped out altogether.

Verify Input Changes

At the top of this cell, Apple checked the input features. Now, we should check them again to make sure the changes we've made are now reflected. We do this using a line of code identical to the first in the cell.

```
builder.inspect_input_features()
```

Setting Metadata

To make life easier for the people using this model, let's update the metadata. First, create descriptions for the input and output that the model produces.

```
neuralnetwork_spec.description.input[0].shortDescription = 'Input image of the han
dwriten digit to classify'
neuralnetwork_spec.description.output[0].shortDescription = 'Probabilities / score
 for each possible digit'
neuralnetwork_spec.description.output[1].shortDescription = 'Predicted digit'
```

Then, you can credit the author of the model and specify the type of license the software, or model, is under. You should also describe the model and what it's designed to do.

```
neuralnetwork_spec.description.metadata.author = 'Core ML Tools'
neuralnetwork_spec.description.metadata.license = 'MIT'
neuralnetwork_spec.description.metadata.shortDescription =
    ('An updatable hand-written digit classifier setup
    to train or be fine-tuned on MNIST like data.')
```

Make the Model Updatable

Finally, the elephant in the room! After preparing everything else, it's finally time to convert our ordinary convolutional neural network into an updatable one. Some procedures of this part might be confusing if you don't have prior experience with machine learning model architectures. But if you follow along closely, everything should make sense!

Declare a Method

For all of the larger tasks, this Jupyter Notebook has been using helper methods. The `make_updatable` method is no exception. This method takes in the path of the current Core ML model and the path where the new model should be saved.

```
def make_updatable(builder, mlmodel_url, mlmodel_updatable_path):
```

Import and Create Specification

Next, since we're using `coremltools` again, we should import it within the function body. Then, we'll set the `builder`'s specification to the `model_spec` variable to access it later. This is the same specification created and used in the previous two cells.

```
import coremltools
model_spec = builder.spec
```

Set the Updatable Layers

If you'll recall, we inspected the layers of the CNN in the last few cells. From the result of this, we can get the names of the layers in the model. To make these layers updatable, we can pass them as a list of names to the `make_updatable()` method on `builder`.

```
builder.make_updatable(['dense_1', 'dense_2'])
```

Set Loss Function and Optimizer

Towards the top of the notebook, in the first cell, a Keras model was created with certain specifications for the loss function and optimizer. The same preferences must be re-specified when creating the updatable model. The first was to use the Cross Entropy Loss function, described earlier.

```
builder.set_categorical_cross_entropy_loss
    (name='lossLayer', input='digitProbabilities')
```

Then, we must re-specify that we're using the Stochastic Gradient Descent optimizer with a learning rate of 0.01 and a batch of 32. These numbers, again, aren't particularly important to understand at this point. However, you can definitely learn more on your own.

```
from coremltools.models.neural_network import SgdParams
builder.set_sgd_optimizer(SgdParams(lr=0.01, batch=32))
```

Last but not least, we need to set the number of epochs. The number of epochs, in machine learning, defines the number of times the model is checked against the loss function. In other words, it’s the number of times that a model is “corrected” based on feedback from the loss function. In this case, we’re using `10` to keep it somewhat light on the GPU during training.

```
builder.set_epochs(10)
```

Adding Metadata

Since we’re treating this updatable model as a separate model, we need new descriptions for its inputs and outputs. Let’s quickly add those in.

```
model_spec.description.trainingInput[0].shortDescription
    = 'Example image of handwritten digit'
model_spec.description.trainingInput[1].shortDescription
    = 'Associated true label (digit) of example image'
```

Saving the Model

Now that the heavy lifting is done, we just need to save the model to the URL specified in the method parameter. First, create an instance of `MLModel` using the model specification. Then, we use the `save()` method and pass in the destination URL.

```
from coremltools.models import MLModel
mlmodel_updatable = MLModel(model_spec)
mlmodel_updatable.save(mlmodel_updatable_path)
```

Call the Method

After creating the method, we must call the method with a file path for the new model. Simply create a variable for the path and then call the `make_updatable` method.

```
coreml_updatable_model_path
    = './UpdatableMNISTDigitClassifier.mlmodel'
make_updatable
    (builder, coreml_model_path, coreml_updatable_model_path)
```

Last Minute Inspections

Finally, the model has been created, converted, and exported as a `.mlmodel` file to your file system. Now, let's again utilize the power of Jupyter Notebooks to check that all our ducks are in a row!

Check Loss Layer

First, we'll check whether the loss layer, or the layer which defines what the model is optimizing for, is correctly optimizing for our `digitProbabilities` target. We can do this similar to how we checked the last three layers of the model.

```
import coremltools
spec = coremltools.utils.load_spec(coreml_updatable_model_path)
builder = coremltools.models.neural_network.NeuralNetworkBuilder
    (spec=spec)
builder.inspect_loss_layers()
```

Inspect the Optimizer and Updatable Layers

In the `make_updatable()` method, we'd specified certain parameters, such as the learning rate and batch size. Let's make sure that these are correctly set.

```
builder.inspect_optimizer()
```

Finally, let's check that the layers we wanted to be updatable are, in fact, updatable.

```
builder.inspect_optimizer()
```

Conclusion

In this chapter, you discovered a more advanced Core ML topic: updatable models. In the beginning of the chapter you learned all about updatable models and how they work. Within this, you learned about the intricacies of how the user interacts with updatable models and how they work from a developer's standpoint.

Later, you learned about tools available on the internet to help you train, source, and interact with updatable models. You also obtained open-source software from Apple and refreshed your skills on Jupyter Notebooks and Python. Finally, you dissected the somewhat complicated code line-by-line to learn best practices in creating updatable models.

Chapter 9
Action Classification and Style Transfer

In the previous chapter, you learned about updatable models, a more advanced topic in Core ML. You started by looking at how updatable models work from a birds-eye view, from both a user and a developer's perspective. Then, you learned about resources for training such models and obtained an open-source Jupyter Notebook to train a MNIST-like updatable model for handwritten symbols classification. Finally, you learned about what each line of code in that notebook does and gained a solid understanding of training updatable models.

In this chapter, you'll learn about action classification and style transfer, two emerging technologies which have the potential to revolutionize machine learning's role in various fields from gaming to art. Action classification is the use of machine learning to observe movements of the human body and make sense of them. In the beginning of the chapter, you'll use action classification to create a fitness classifier after learning how to find, record, and source videos for action classification.

Later in the chapter, we'll dive deep into style transfer. Style transfer is the use of machine learning to replicate the style of a certain image on another, unrelated image. We'll first learn about the structure of style transfer model projects — they're quite different from anything we've seen in the past. Next, you'll learn about different use-cases of style transfer models. Finally, you'll create your own style transfer model to replicate the style of Vincent Van Gogh's "Starry Night" painting. And, along the way, you'll learn about fine-tuning model control, including pausing, adding iterations, and model snapshots within Create ML.

9-1 Action Classification

In this section, you'll learn about Action Classification, a feature of Create ML which allows you to analyze motion of the human body through a video. In past chapters, we've used image classifiers to sequentially analyze each frame in a live video feed to classify objects; however, in this chapter, we'll use Create ML's built-in tools to analyze videos in their entirety. Think of it like analyzing an audio clip in our instrument classification model.

Through Create ML, we'll train a model that takes video clips of people doing day-to-day actions such as walking, waving, and climbing stairs. First, you'll obtain a dataset of videos, train a dataset on them with Create ML, and then test your model with sample videos. You'll be able to quickly see how your model performs. Along the way, you'll learn about adjusting model parameters to improve your results.

Obtaining Videos

The first step, as you're already aware, is to get data to train your model with. Unlike other chapters, though, you'll find that it's harder to find pre-made datasets for Action Classification, and we'll discuss ways you can get data for your models.

Provided Dataset

Since training machine learning models on videos is much less common than training on still images, you'll realize that it's much harder to find full-fledged datasets of videos like you can for images. Fortunately for you, I've gone through to adapt and compile a dataset which is ready-to-use for you, which you can find in the downloads for this book.

Feel free to adapt this dataset for your own uses — if you need to customize it to fit your needs, you can handpick the actions you need and then combine them with other publicly available datasets.

Recording your Own

If you want to create a custom model, I highly recommend recording your own videos. You don't need nearly as many videos to get a good result, and depending on the number of classes, you could get away with as few as 3-5 videos per class. Let's look at tips for you to record your own videos with.

Record in Real-Life Situations

As with other types of data, including images and audio clips, it's important to create a dataset which accurately represents the situations your users will be using your model in. So, if you're creating a workout classifier for at-home workout coaching, then it's best to have your recordings be in home settings — not at a gym.

Use Longer Clips

Back to the exercise example: if you're creating a model which recognizes squats, create a video of you squatting multiple times. This way, Create ML will average out your movements, instead of taking your single squat as the gold standard. Similar to providing more images to an image classification model, using longer video clips with more repetitions will improve your model's performance.

Don't Overdo It

Unless you're using an eGPU, training on video clips can take an unreasonably long time to train. So, try and keep the amount of classes—and the number of videos in each class—as manageable as possible. The last thing you need is training for days, only to find that you're not getting your desired results.

Compiling a Dataset

While it's definitely *harder* to find datasets, it isn't impossible. Sometimes, the best option is to simply find different datasets and compile the data to create your own. Keep in mind, though, that these datasets are often not targeted and contain thousands of videos of random actions—often ones that don't fit under the realm of Action Classification. Here are some resources you could use:

Video Dataset Overview

If you're looking for a single place to find your datasets, I recommend using this database of video datasets. Simply search what you want, and it'll give you a link where you might — or might not — be able to find it. This gives you some preview information ahead of time, too, so it's easier to gloss over the ones you aren't interested in.

VIRAT Video Dataset

With action classification, an important use-case is video surveillance — after all, the key in surveillance is detecting suspicious actions taken by people. The VIRAT Dataset provides both aerial and ground footage of people from security cameras and can be useful if you're creating a model for these use-cases.

HMDB Dataset

If you want more general human movements, you can find them in the Human Motion Database. Here, you'll be able to sort through the movements you'd like and then train your model on those. For example, if you wanted an exercise model, you could select only the exercise videos from the dataset.

VGG Pose Dataset

The VGG Pose Dataset focuses on upper body poses, which can be used to categorize the type of video you're watching. Some examples of videos in this dataset are stand-up comedy, sign language videos, and performing arts. These different types of videos could be used for a video curation app, for example.

Of course, this list isn't exhaustive; however, it should give you a good starting place. Since there isn't a huge number of datasets available yet, you may have to mix and match to get the result you need — or, just record your own!

Training the Model

Now that you have your data all ready to go, it's time to train your model. Though you're working with videos in this particular model, training is surprisingly similar to other models you've trained in this book.

Preparing your Data

If you've used the dataset that I've provided, you can feel free to skip this step. However, I recommend you read over it so that you know what to do when you eventually use your own data. As with other models we've dealt with, you'll need to have training, validation, and testing folders.

Each folder should have a name, such as "validation" and should have the name of each class within in. Make sure these names match, in that you don't call it "pushup" in one folder but call it "pushups" in the other. These folder names are what Create ML uses to identify your videos.

Training

Training data should make up the majority of your videos. In my case, I've provided you a fairly small subset of the HMDB database, so I only have about 8 video clips in each testing folder.

Validation

From what you've learned before, you should set aside at least 20% of your data for validation — this is what Create ML uses to measure your model and tweak its performance while training. In my case, I used 3 images for validation per class.

Testing

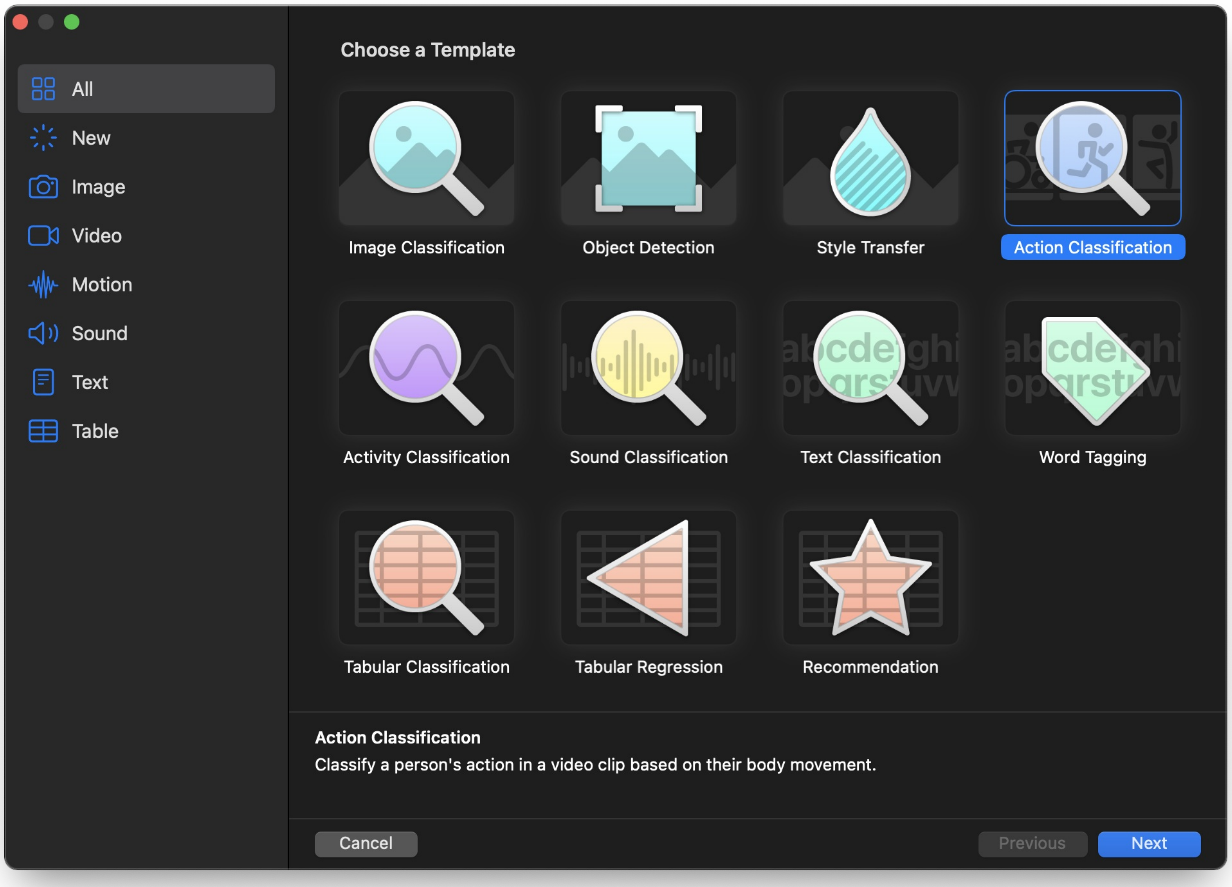
The testing dataset is what Create ML uses to test the model *after* it's been trained and give you the result. You could choose to omit this and test your model manually later on; however, I've included 3 additional videos to test with.

Inputting Data

Once your data is ready, you can create a project in Create ML to put everything in. This is one of the easier steps, and you've done it several times before — so, let's get to it!

Select Model Type

When you open Create ML, you'll be greeted with a screen that asks you to select your project type. The **Action Classification** project — not to be confused with the activity classification type — is the one you need to select.



Action Classification
Classify a person's action in a video clip based on their body movement.

Figure 9-1: Choosing a Create ML Template

Save Model

You'll then be asked to give it a useful name and description — this name will appear on your model when you use it in your apps or share it with others. Then click **Next**.

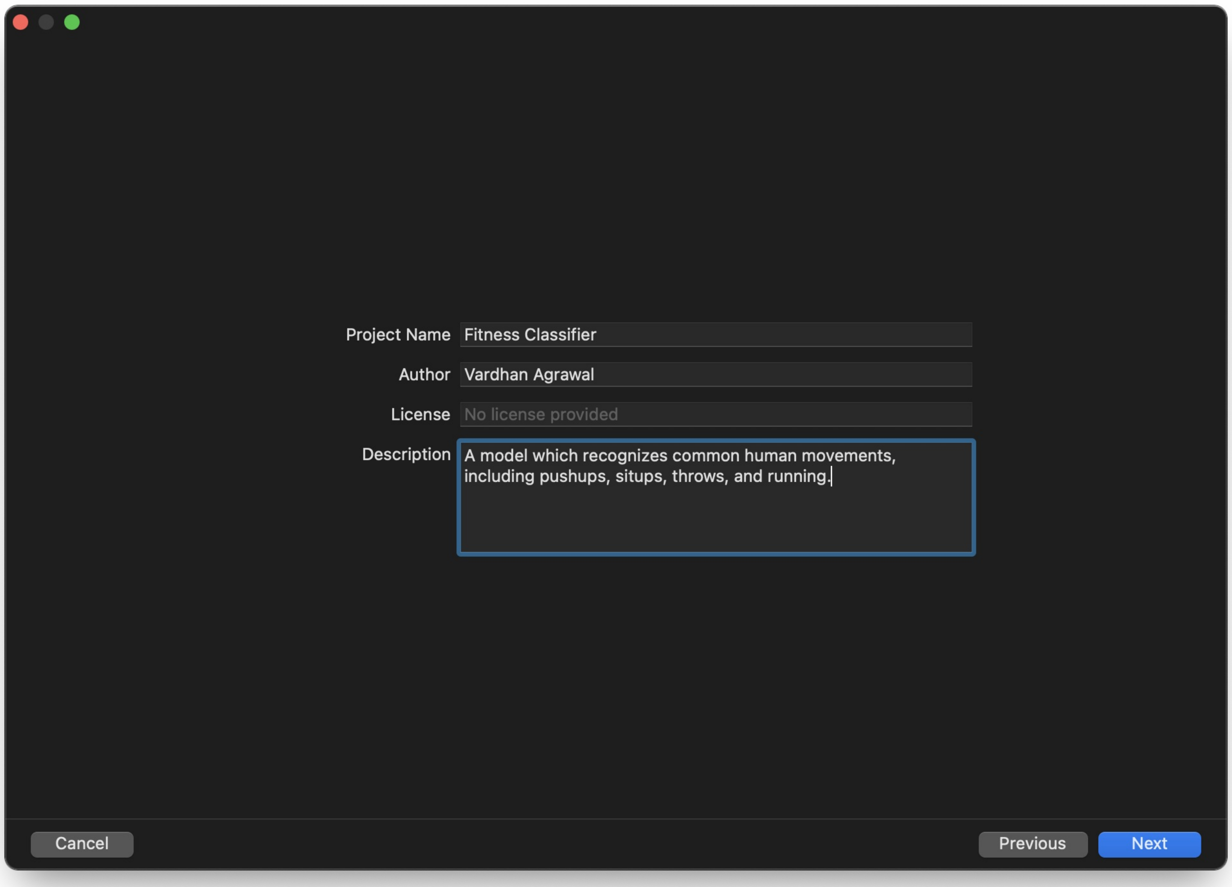

Project Name: Fitness Classifier
Author: Vardhan Agrawal
License: No license provided
Description: A model which recognizes common human movements, including pushups, situps, throws, and running.

Figure 9-2: Naming your Create ML Project

Once you're done naming your project, you'll be asked where you want to store it. Choose a convenient location for your project and then move on. This isn't where your model will be stored, but instead, your Create ML project — a subtle distinction.

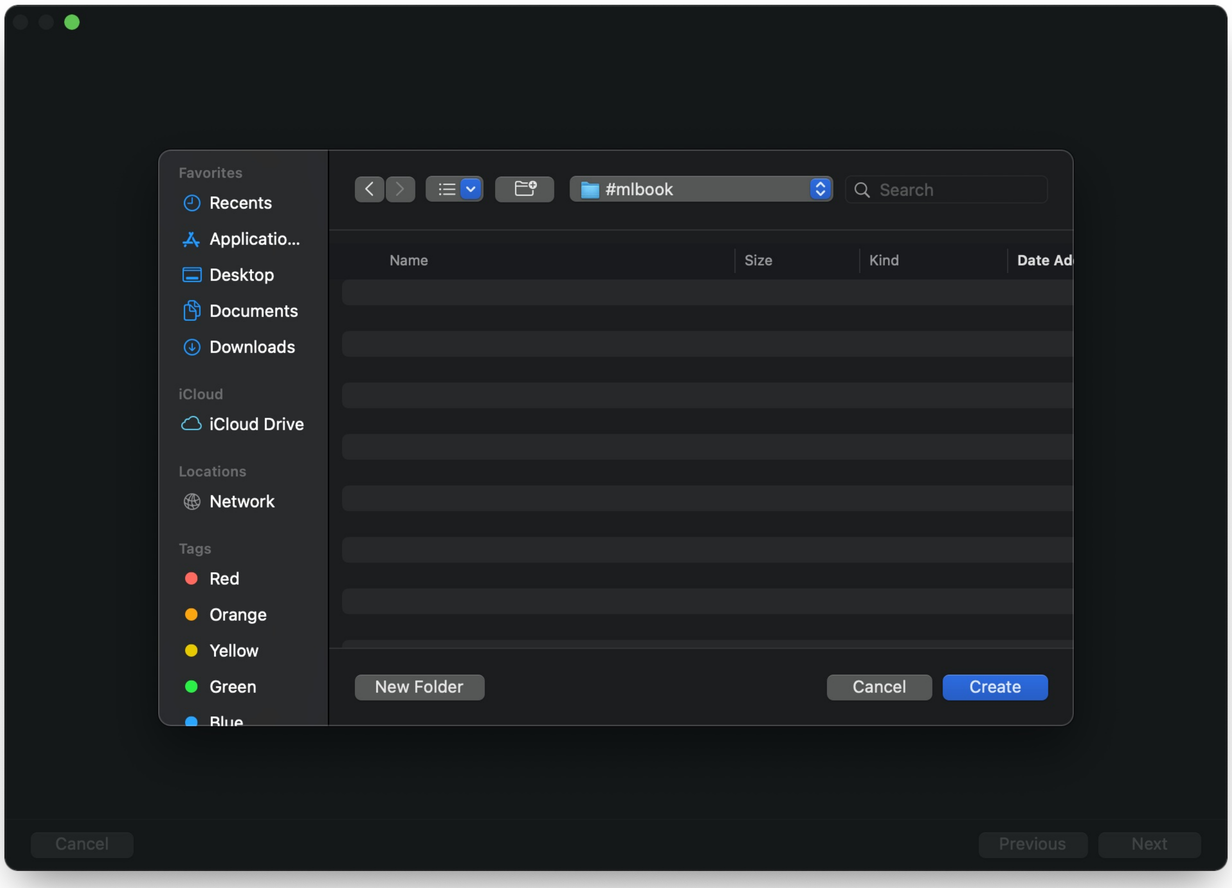
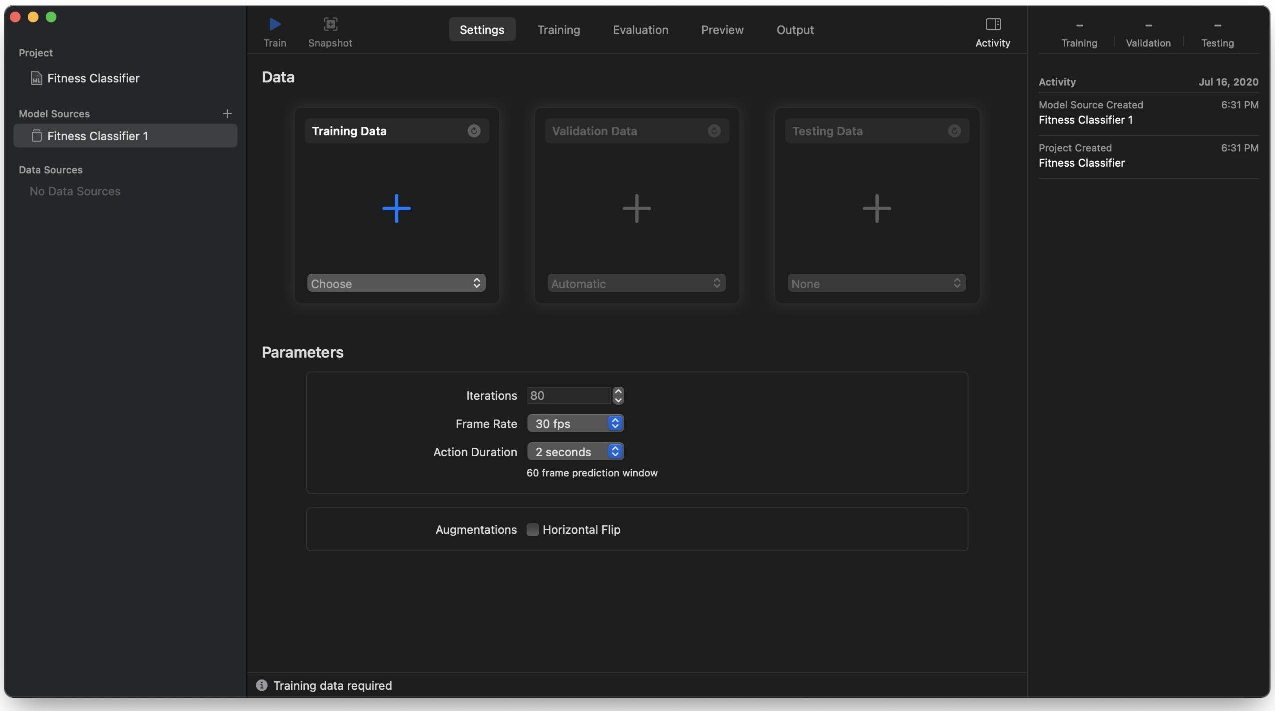


Figure 9-3: Saving your Project

Add Training Data

When your project has been created, you'll get a dashboard which looks something like this:



Iterations 80
Frame Rate 30 fps
Action Duration 2 seconds
60 frame prediction window

Augmentations Horizontal Flip

Training data required

Activity	Jul 16, 2020
Model Source Created Fitness Classifier 1	6:31 PM
Project Created Fitness Classifier	6:31 PM

Figure 9-4: Create ML Dashboard

Here, you'll see three boxes for training, validation, and testing data, respectively. Go ahead and drag in the folder you named "training" into the **Training Data** box.

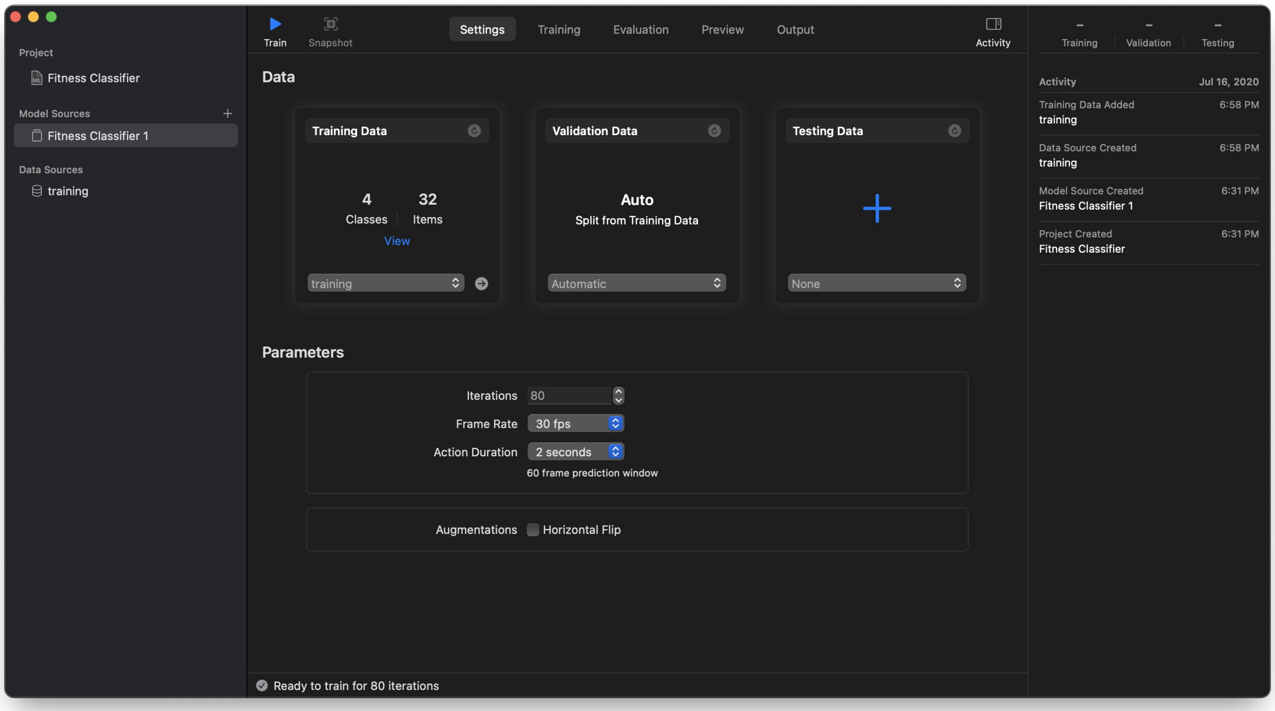


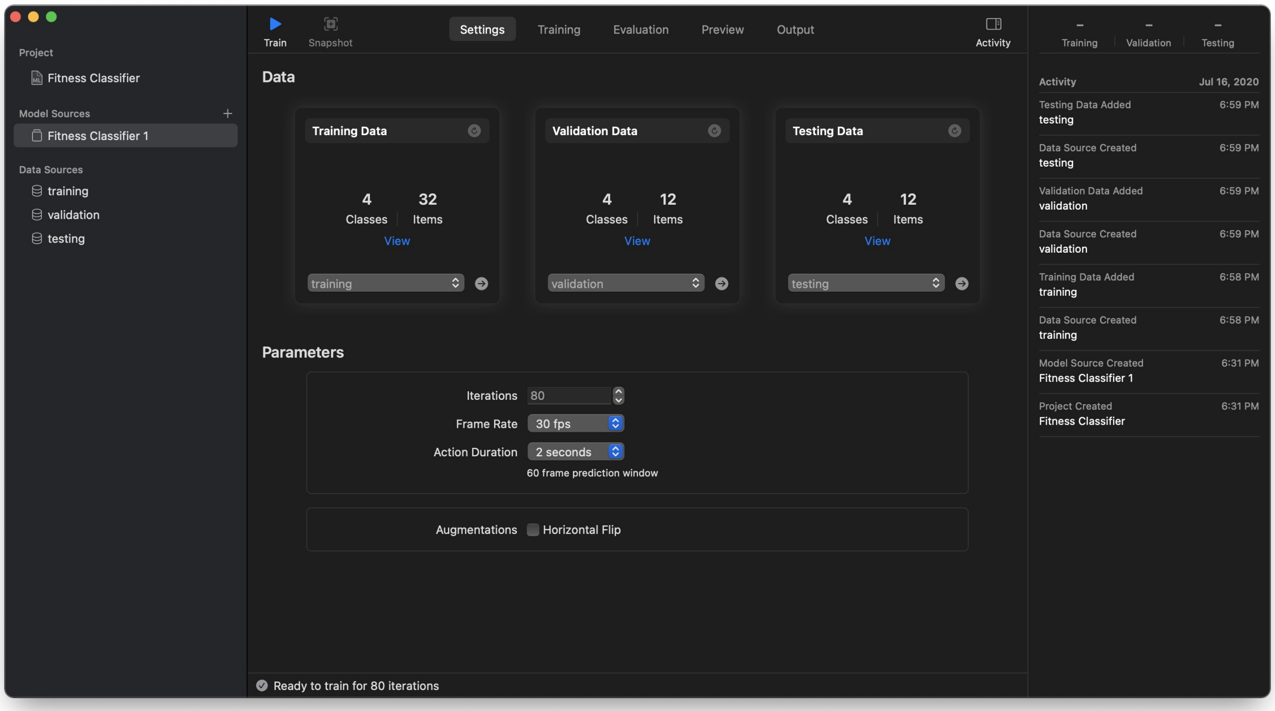
Figure 9-5: Adding Training Data

You will see the number of classes, which, in my case is 4 — and the total number of videos in the training set.

Add Validation and Testing Data

In my case, my validation and testing data is the same size. It's also worth noting that if you don't include validation data, Create ML will automatically use some of your testing data as validation data.

Drag your "validation" folder to the **Validation Data** box and then your "testing" folder to the **Testing Data** box.



Training Data: 4 Classes, 32 Items, training

Validation Data: 4 Classes, 12 Items, validation

Testing Data: 4 Classes, 12 Items, testing

Iterations: 80

Frame Rate: 30 fps

Action Duration: 2 seconds

60 frame prediction window

Augmentations: Horizontal Flip

Ready to train for 80 iterations

Activity — Jul 16, 2020

Testing Data Added — testing — 6:59 PM

Data Source Created — testing — 6:59 PM

Validation Data Added — validation — 6:59 PM

Data Source Created — validation — 6:59 PM

Training Data Added — training — 6:58 PM

Data Source Created — training — 6:58 PM

Model Source Created — Fitness Classifier 1 — 6:31 PM

Project Created — Fitness Classifier — 6:31 PM

Figure 9-6: Adding Validation and Testing Data

Once you've done this, you'll see that all of the boxes say the same number of classes and reflect the number of videos in each folder.

Exploring Model Parameters

Before you train, you'll see some options that you can adjust beforehand. So, let's look at these options and see how they can help you create a better model in the end. You can always use the defaults and try again later, but let's look at what they each mean.

Iterations

By now, you're likely familiar with the concept of iterations. The number of iterations tells Create ML how many times to tweak the model and check its performance with the validation set. Normally, models converge — or stop getting better — after a certain number of iterations. Our goal when setting the number of iterations is to get the smallest number at which we have the best result. Usually, I recommend just going with the default value.

Frame Rate

The frame rate of a video dictates how many frames you see, per second. Your phone's camera normally records between 30 and 60 frames per second — and that's usually plenty. If you have some *really* fast-moving action, you may want to step up the frame rate, but remember: it increases the amount of time needed to train.

Action Duration

Since we're dealing with videos, Create ML asks us to specify an action duration. This is how long each action lasts. For example, if your app detects jump rope (a repetitive motion), you would put in the amount of time it takes for the rope to go once around. Similarly, if you were detecting a golf swing, you would put in the amount of time it takes to finish the entire swing.

Augmentations

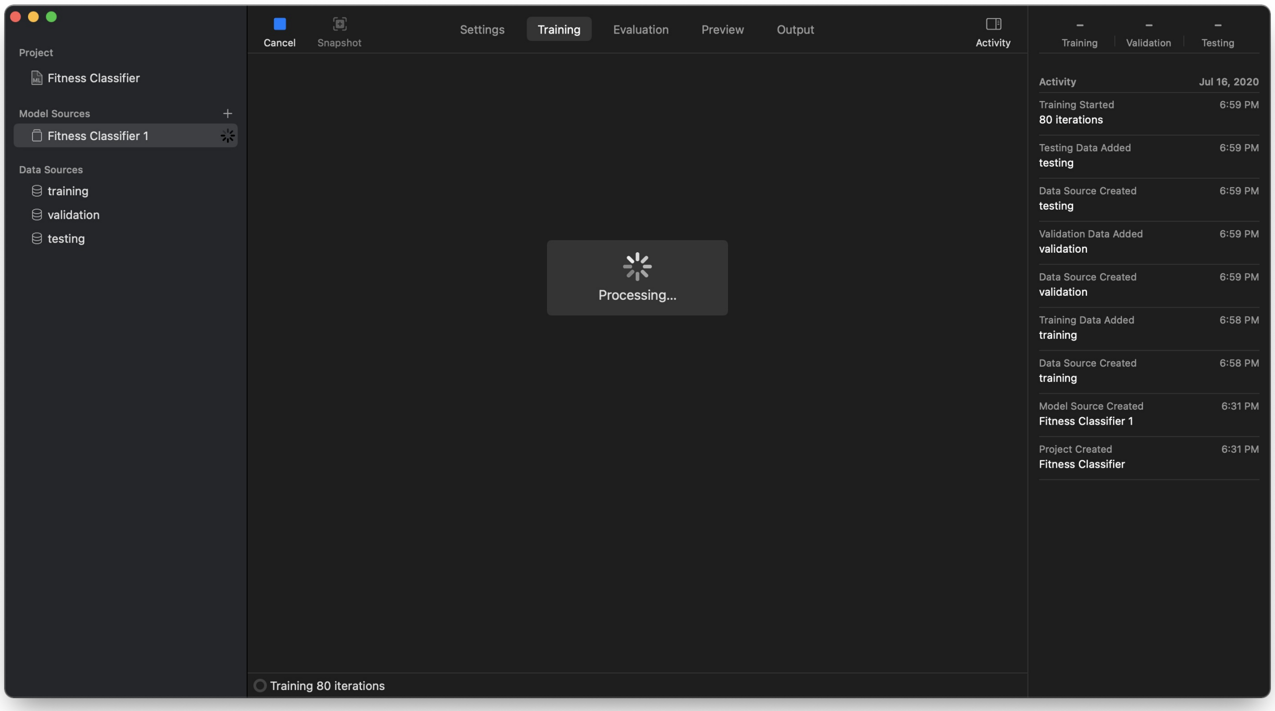
The only augmentation available for video-based models in Create ML is a horizontal flip. While this might make sense for some models, we won't be using it here. Using augmentations are an easy way to increase the size of your dataset, if you have a limited amount of videos to work with.

Model Training and Adjustment

Now, with all of that out of the way, you're ready for the exciting part — model training! Similar to other models we've created in the past, model training is fairly straightforward.

Start Training

To start training, click the blue, triangular **Train** button in the upper-left of your Create ML window. Your videos will then be processed.

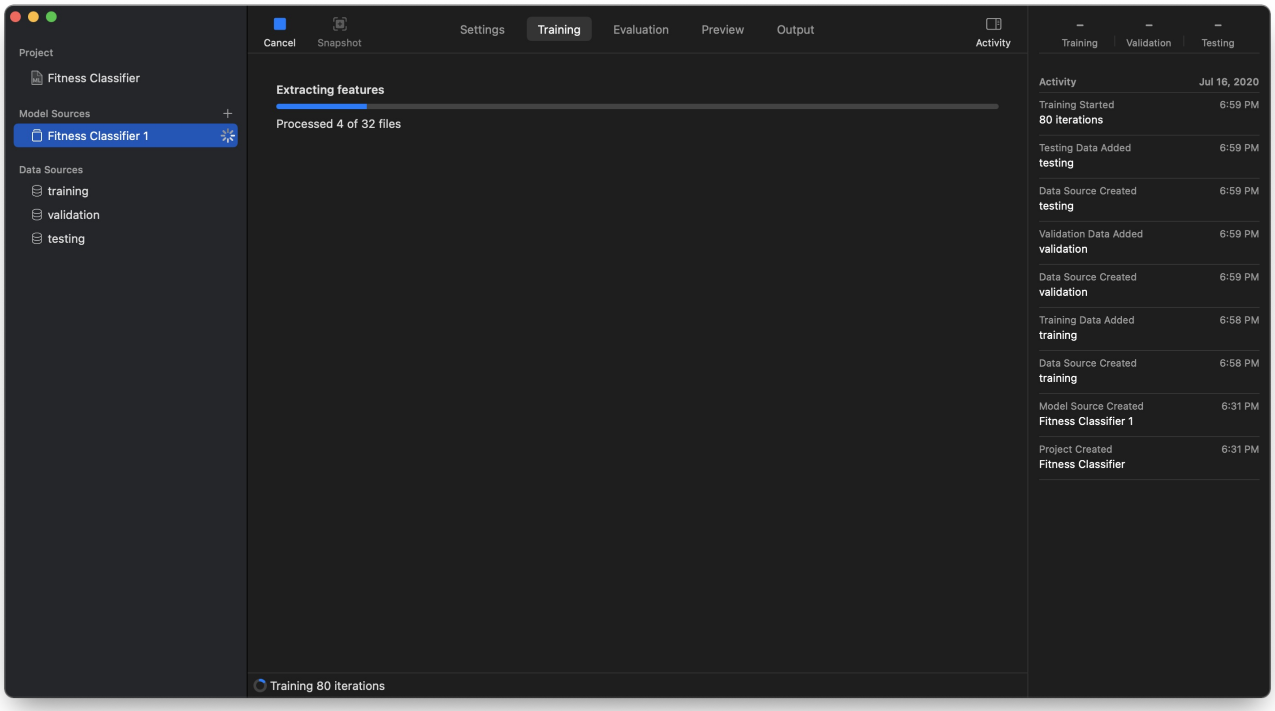


```
Activity                         Jul 16, 2020
Training Started                      6:59 PM
80 iterations
Testing Data Added                    6:59 PM
testing
Data Source Created                   6:59 PM
testing
Validation Data Added                 6:59 PM
validation
Data Source Created                   6:59 PM
validation
Training Data Added                   6:58 PM
training
Data Source Created                   6:58 PM
training
Model Source Created                  6:31 PM
Fitness Classifier 1
Project Created                       6:31 PM
Fitness Classifier
```

Training 80 iterations

Figure 9-7: Starting Model Training

Soon, model training will start, and Create ML will begin extracting the features from each of your training videos.



Extracting features

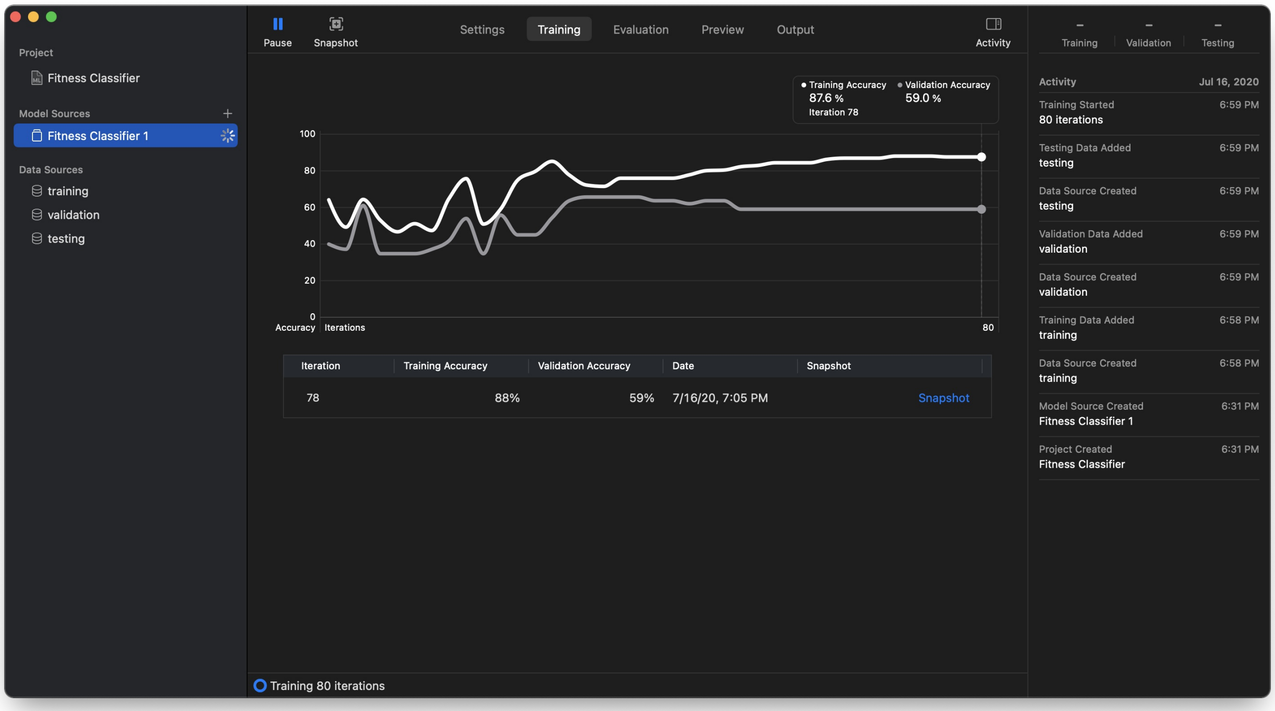
Processed 4 of 32 files

Training 80 iterations

```
Activity                      Jul 16, 2020
Training Started                   6:59 PM
80 iterations
Testing Data Added                 6:59 PM
testing
Data Source Created                6:59 PM
testing
Validation Data Added              6:59 PM
validation
Data Source Created                6:59 PM
validation
Training Data Added                6:58 PM
training
Data Source Created                6:58 PM
training
Model Source Created               6:31 PM
Fitness Classifier 1
Project Created                    6:31 PM
Fitness Classifier
```

Figure 9-8: Model Training Progress Bar

After Create ML is done with this step, you'll see the training and validation accuracies on a graph. The goal is to get these lines as flat as possible, which means that adding more iterations won't make your model much better — which, it looks like I've achieved here:



Iteration	Training Accuracy	Validation Accuracy	Date	Snapshot
78	88%	59%	7/16/20, 7:05 PM	Snapshot

Activity	Jul 16, 2020
Training Started 80 iterations	6:59 PM
Testing Data Added testing	6:59 PM
Data Source Created testing	6:59 PM
Validation Data Added validation	6:59 PM
Data Source Created validation	6:59 PM
Training Data Added training	6:58 PM
Data Source Created training	6:58 PM
Model Source Created Fitness Classifier 1	6:31 PM
Project Created Fitness Classifier	6:31 PM

Figure 9-9: Model Training Accuracy Graph

Pausing and Snapshots

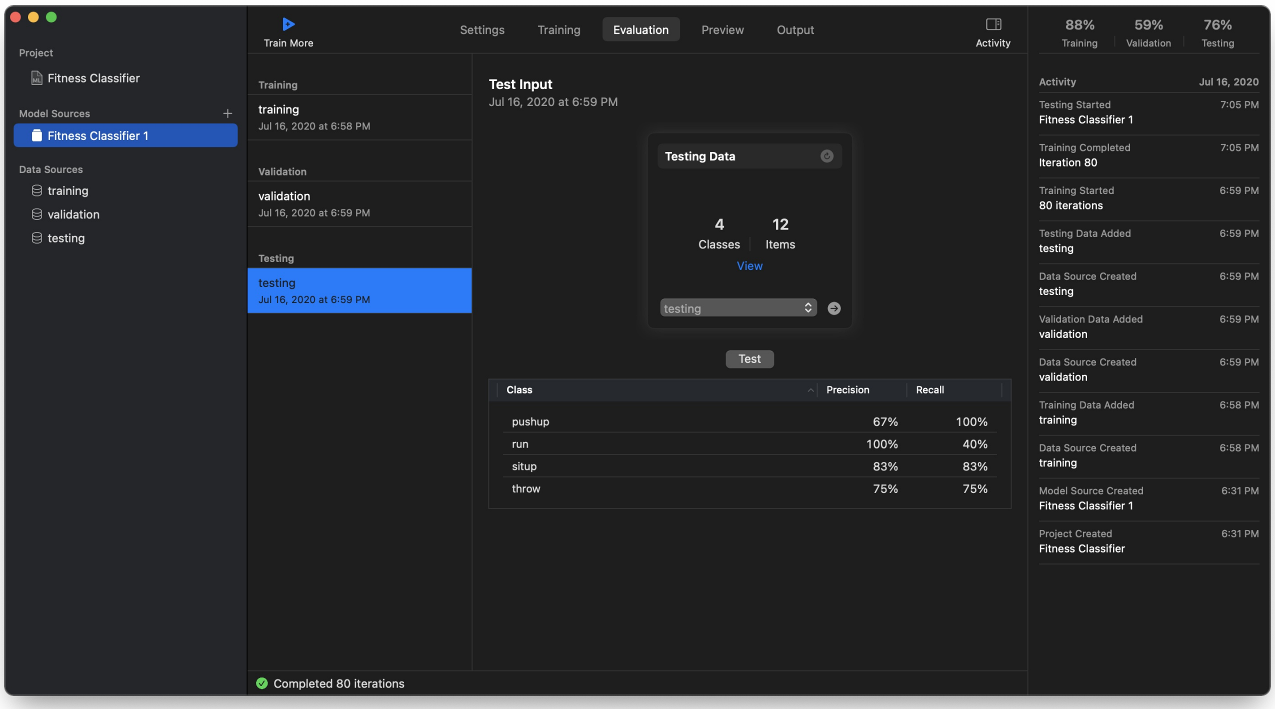
If you feel that you've reached a solid number of iterations, you can pause training early, and if you want to capture a particular iteration, you can take a model snapshot, which you can work with later. Don't worry if this doesn't make complete sense yet — we'll learn more about this later in the chapter.

Project Modes

Once your model is done training, you're presented with a few metrics to evaluate how your model did. Let's break down these metrics and see if you have a sufficiently well-performing model, or not. We'll look at this by examining each of the options in the top menu bar.

Evaluation Tab

If you click on the **Evaluation** tab at the top, you'll see *training*, *testing*, and *validation* on the left column. If you click on each of these, you'll see a detailed breakdown of each class and it's precision-recall values. This will help you understand how your model performed at each stage of the training process for each video type.



Training
training
Jul 16, 2020 at 6:58 PM

Validation
validation
Jul 16, 2020 at 6:59 PM

Testing
testing
Jul 16, 2020 at 6:59 PM

Test Input
Jul 16, 2020 at 6:59 PM

Testing Data
4 Classes
12 Items

Class	Precision	Recall
pushup	67%	100%
run	100%	40%
situp	83%	83%
throw	75%	75%

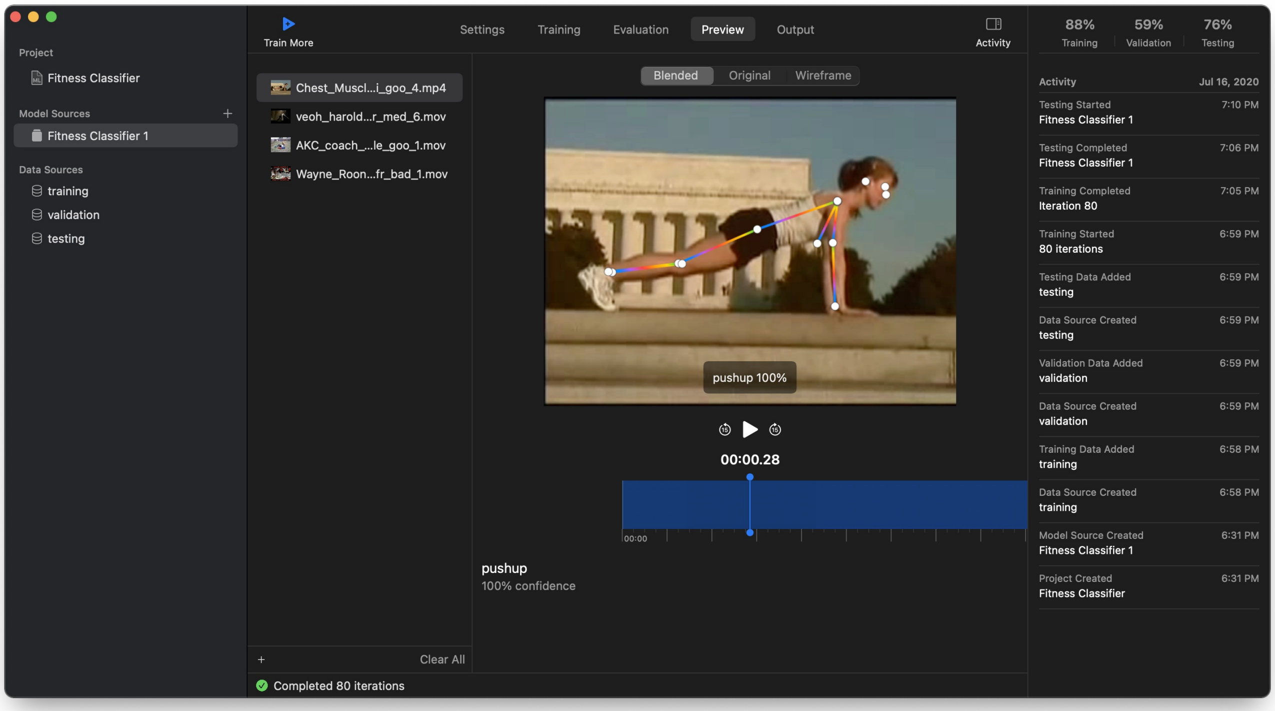
Completed 80 iterations

Activity	Jul 16, 2020
Testing Started Fitness Classifier 1	7:05 PM
Training Completed Iteration 80	7:05 PM
Training Started 80 iterations	6:59 PM
Testing Data Added testing	6:59 PM
Data Source Created testing	6:59 PM
Validation Data Added validation	6:59 PM
Data Source Created validation	6:59 PM
Training Data Added training	6:58 PM
Data Source Created training	6:58 PM
Model Source Created Fitness Classifier 1	6:31 PM
Project Created Fitness Classifier	6:31 PM

Figure 9-10: Evaluating your Action Classifier

Preview Tab

In the **Preview** tab, you'll be able to drag in additional images and see how your model classifies it. You'll also be able to see a really cool wireframe, which Create ML shows you to help visualize what your model sees. I've dragged in some data from my testing folder to see how it looks.



pushup
100% confidence

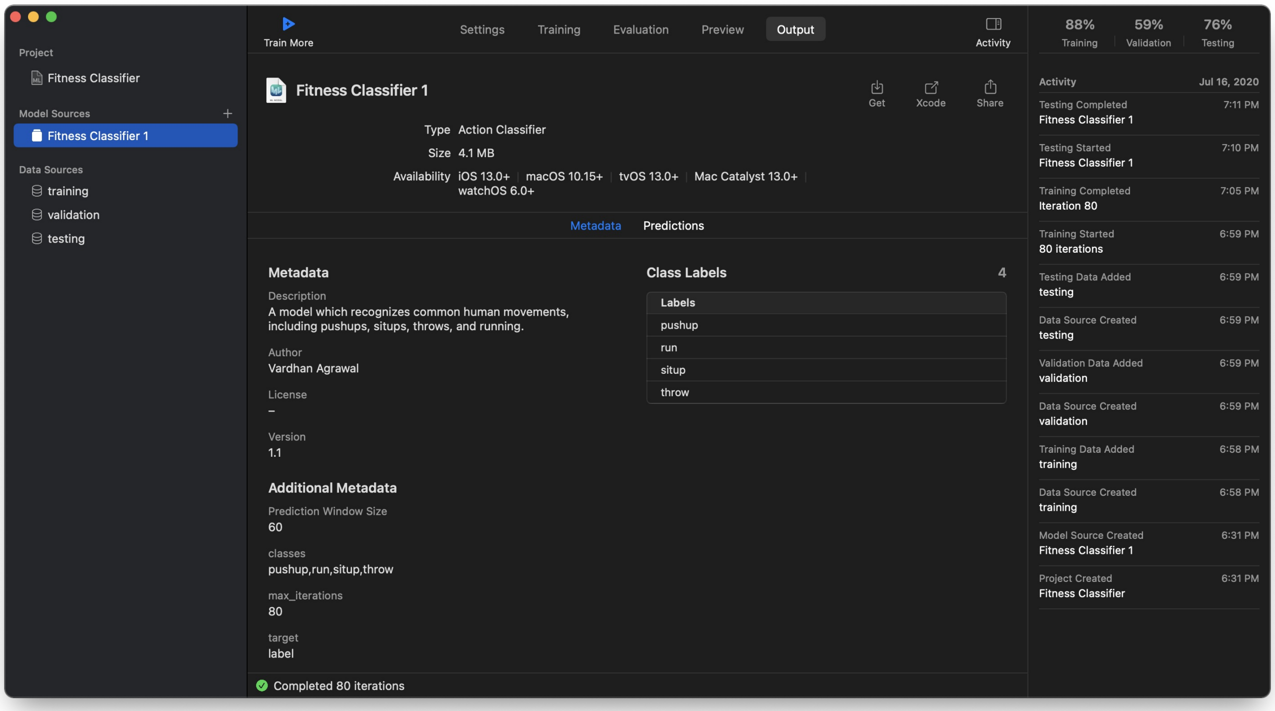
Completed 80 iterations

Activity	Jul 16, 2020
Testing Started Fitness Classifier 1	7:10 PM
Testing Completed Fitness Classifier 1	7:06 PM
Training Completed Iteration 80	7:05 PM
Training Started 80 iterations	6:59 PM
Testing Data Added testing	6:59 PM
Data Source Created testing	6:59 PM
Validation Data Added validation	6:59 PM
Data Source Created validation	6:59 PM
Training Data Added training	6:58 PM
Data Source Created training	6:58 PM
Model Source Created Fitness Classifier 1	6:31 PM
Project Created Fitness Classifier	6:31 PM

Figure 9-11: Previewing Action Classification Model

Exporting your Model

Once everything looks good to you, you can head to the **Output** tab and export your model.



Fitness Classifier 1

Type Action Classifier
Size 4.1 MB
Availability iOS 13.0+ | macOS 10.15+ | tvOS 13.0+ | Mac Catalyst 13.0+ | watchOS 6.0+

Metadata

Description
A model which recognizes common human movements, including pushups, situps, throws, and running.

Author
Vardhan Agrawal

License
–

Version
1.1

Additional Metadata

Prediction Window Size
60

classes
pushup,run,situp,throw

max_iterations
80

target
label

Class Labels 4

Labels
pushup
run
situp
throw

Activity Jul 16, 2020

Testing Completed Fitness Classifier 1	7:11 PM
Testing Started Fitness Classifier 1	7:10 PM
Training Completed Iteration 80	7:05 PM
Training Started 80 iterations	6:59 PM
Testing Data Added testing	6:59 PM
Data Source Created testing	6:59 PM
Validation Data Added validation	6:59 PM
Data Source Created validation	6:59 PM
Training Data Added training	6:58 PM
Data Source Created training	6:58 PM
Model Source Created Fitness Classifier 1	6:31 PM
Project Created Fitness Classifier	6:31 PM

Completed 80 iterations

Figure 9-12: Exporting your Action Classifier

9-2 Image and Video Style Transfer

In the previous section, you learned about an exciting use of machine learning — Action Classification. First, you learned about obtaining the video datasets needed to train such a model, and then, you trained and tested your own action classification model which could accurately analyze body poses and generate a classification result. In this section, I have something just as exciting in store!

Here, you'll learn about style transfer — not only on images, but on videos as well. So, if you have a particular artist you like but can't afford to commission a painting, you can just train a model to paint one in their style! To start, you'll learn about what style transfer is and how it works. Next, you'll choose a source and test image, and then you'll train your model. Let's jump right in!

About Style Transfer

While many people are familiar with style transfer through social media apps and AI-powered filters, few know about style transfer from the lens of a software development standpoint. So, before we create our own style transfer models, let's learn about what style transfer is and how it works under the hood.

Media Types

Style transfer, as the name suggests, is the transfer of styles, colors, and textures from style media to content media. This means that the target media will be altered to match the style of the content image. Let's dissect this a little bit further, to make sure you have a hang of it.

Style Media

The *style media*, normally in the form of an image, dictates the how you want your content media to look. By adjusting certain parameters — as you'll learn about in the next section — you'll be able to control how closely your model matches the appearance of the style media.

You can draw a real-world analogy to understand this a little bit better. Think of yourself as an artist, trying to match the style of another artist. If you were a machine learning model, your reference media would be the art you're trying to replicate.

Content Media

The *content media*, which could be a still image or a video — depending on the complexity of your model and training tools — is the image which you're aiming to adapt to the style media's appearance. In the art example from above, the content media would be the painting you're sitting down to create, in the style of the other artist.

Notable Art

Believe it or not, paintings created using machine learning are recognized as actual works of art in the art community — and, for good reason. If you've ever seen pieces of art created using artificial intelligence, you might not be able to distinguish between the artificially generated piece from ones that the actual artist has created.

Recently, an AI-generated piece was sold for over almost half-a-million dollars. This makes it clear that style transfer is promising technology, and it's at the unexpected intersection between computer science and art. Looking at examples like this, we see how machine learning can quantify and concretize a seemingly qualitative form of expression — art.

Uses of Style Transfer

Clearly, style transfer isn't going anywhere in the next few years, but what are its practical uses? Before we create our own style transfer model, let's take the time to learn more about it in detail.

Photo Apps

Whether it's independent developers or major platforms such as Instagram and Snapchat, there's no shortage of apps which allow you to transform your face — or ones that allow you to create art in the style of someone else. At their core, style transfer is what makes them possible.

Professional Art

As mentioned earlier, art generated by style transfer algorithms is now considered a form of actual art in the art community — after all, they look just like actual art pieces and take skill to make. You'll be learning that skill in this chapter! Aside from auctions and museums, we may see more artists majoring in computer science, bridging the gap between the humanities and STEM.

Games and Augmented Reality

Now, even game developers are exploring the possibility of using style transfer to make their games pop. With an increasing emphasis being placed on how a game feels, game developers are trying new things to make their games stand out, while still being engaging. AR developers are also trying to use style transfer apps to blend real-world objects and virtual objects better.

Preparing Data

In some ways, preparing a dataset for style transfer is easier than datasets we've created in the past. You just need three things: a style image, a validation image, and an optional set of content images. If you want to skip over this step, feel free to use the dataset that I've already created. As always, you can find this in the downloads of this book.

Choosing Content Images

Though it's an optional step, I strongly recommend choosing content images. These should be images which represent what your model's users will want to stylize. For example, if you're making a model for a selfie app, you'd want your content images to be people's faces. I'll be applying Van Gogh's style to landscape images, so my content images are images of mountains and bodies of water.

Choosing a Style Image

This is the easiest — and, arguably the most fun — part of creating style transfer models. Here, simply choose an image or artist who's style you like and pick an image. You'll be training your model to replicate this style, so pick a good one! I really like the style of Van Gogh's "Starry Night," so that's the style image I'll go with.

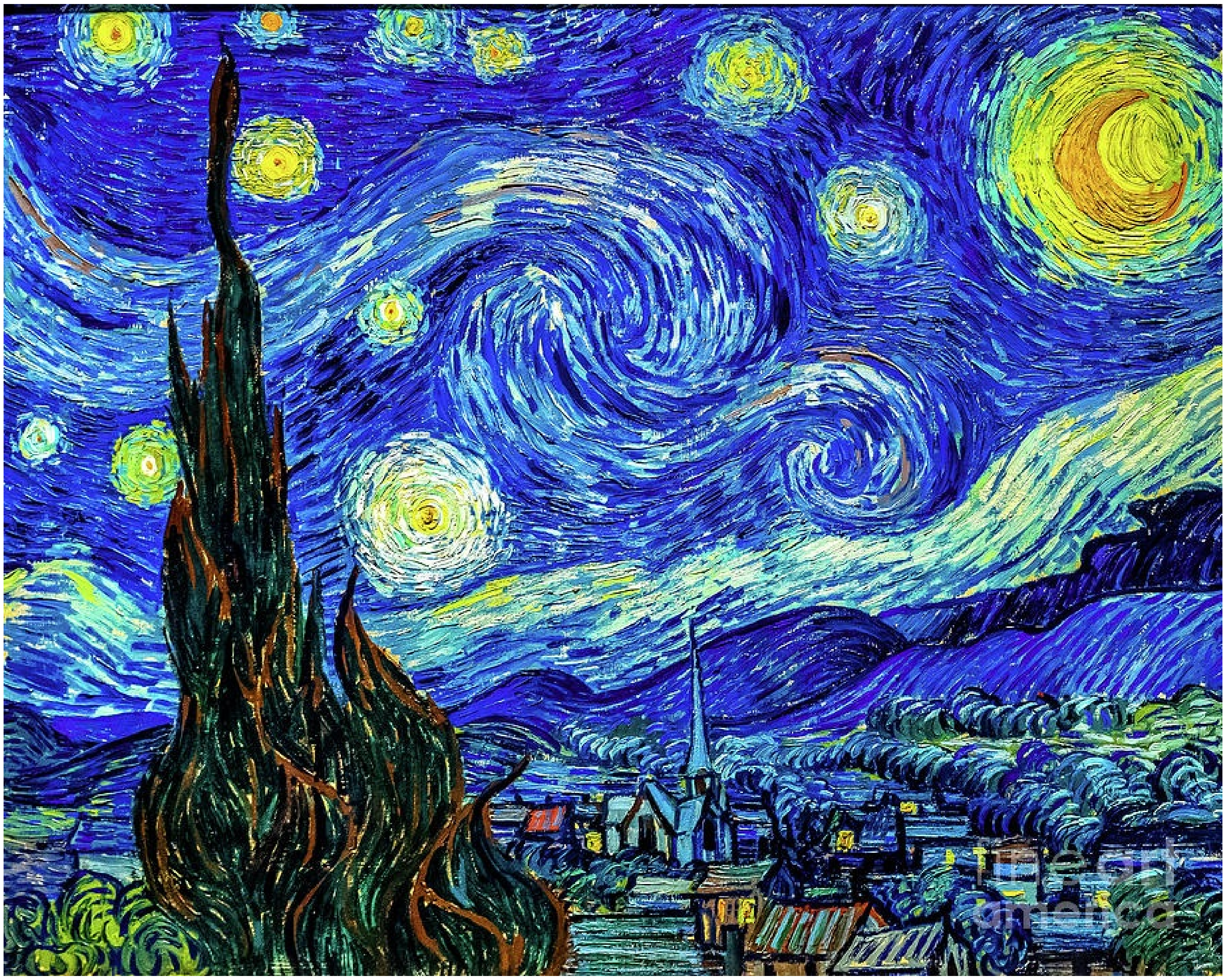


Figure 9-13: A Style Image, Van Gogh's Starry Night

Choosing a Validation Image

Later in this section, we'll learn about fine-tuning your style transfer training process, so the validation image is a visual tool for you to know when your model meets — or doesn't meet — your expectations. Your validation image should well represent your dataset of content images, so choose wisely!

Figure 9-14: A Validation Image

9-3 Training your Model

In the previous section, you learned a little about what style transfer actually is, including some examples of how it's revolutionizing the field of art. You also looked at possible use-cases for style transfer models and prepared data for your own style transfer model.

In this section of the book, we'll roll up our sleeves and dive into the world of fine art. Without any artistic prowess, we'll turn an ordinary photograph into a painting in the style of Vincent Van Gogh's "Starry Night." If you're even remotely familiar with art, you would be familiar with this piece. If you have something else in mind, feel free to follow along with your own data.

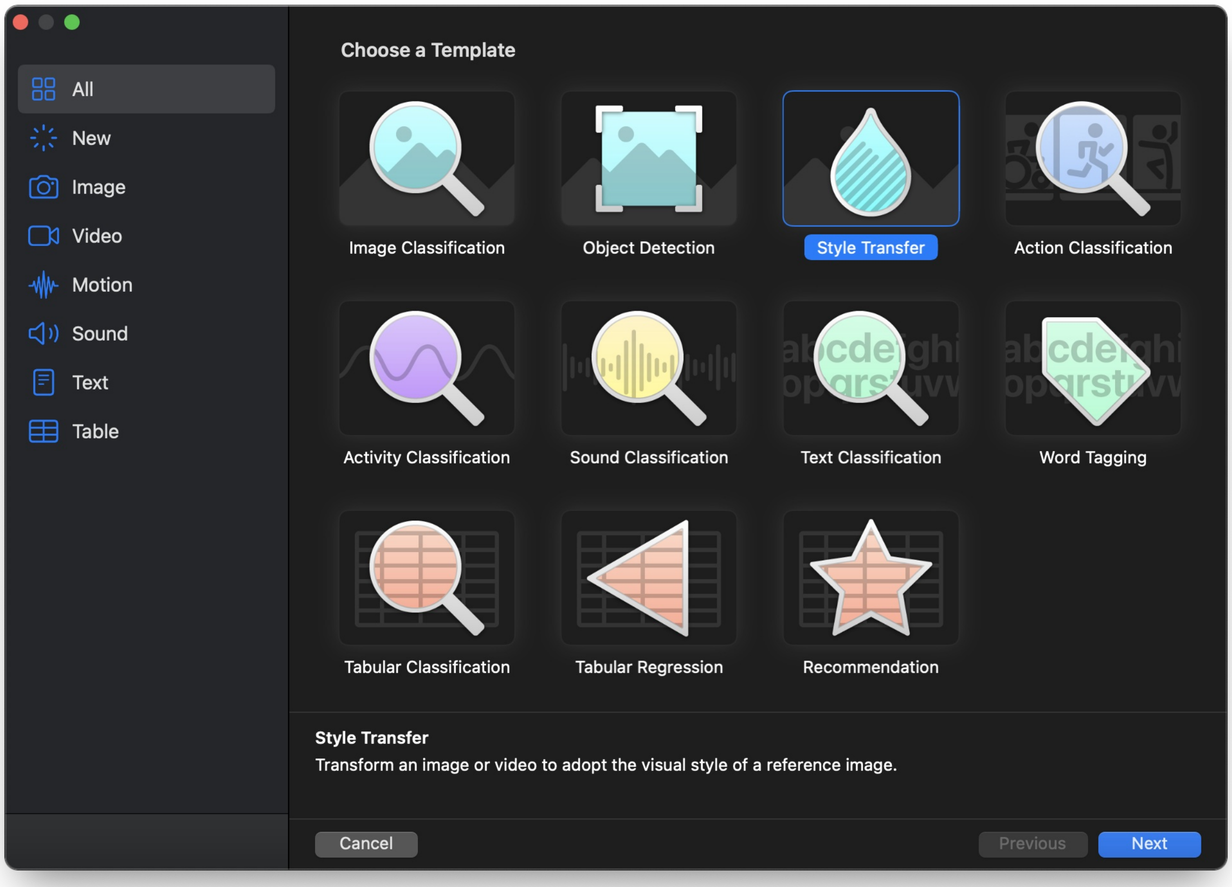
Now that you have a solid understanding of how style transfer works and have your own data to go with it, you can start training your model. By the end, you'll have a finished style transfer model, suited to your preferences!

Inputting Data

Once you're data is ready, you're almost ready to start training. Let's first feed the data into the Create ML App, so that everything is set for training once you're ready. If you're using your own data, pay attention to what I'm putting where. You don't want to mix up your style image and validation image, for instance.

Select Model Type

First and foremost, open the Create ML app. You should be greeted with a screen which asks you what type of model you want to create.

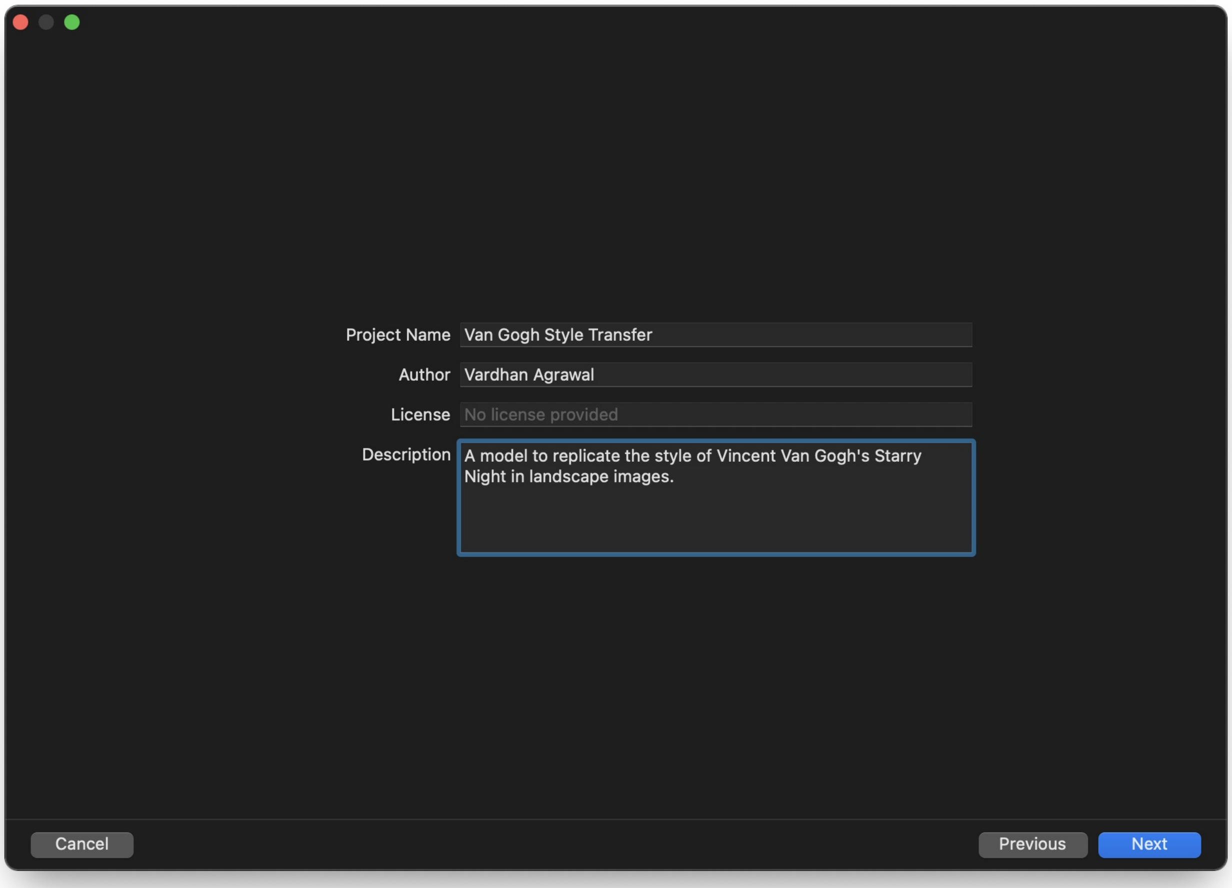


Style Transfer
Transform an image or video to adopt the visual style of a reference image.

Figure 9-15: Choosing a Create ML Template

Save Project

Here, you'll need to select the one which says **Style Transfer** and click **Next**. Then, give your project a name and a detailed description:



Project Name: Van Gogh Style Transfer
Author: Vardhan Agrawal
License: No license provided
Description: A model to replicate the style of Vincent Van Gogh's Starry Night in landscape images.

Figure 9-16: Naming your Create ML Project

You'll then be asked to choose where you want to store your project.

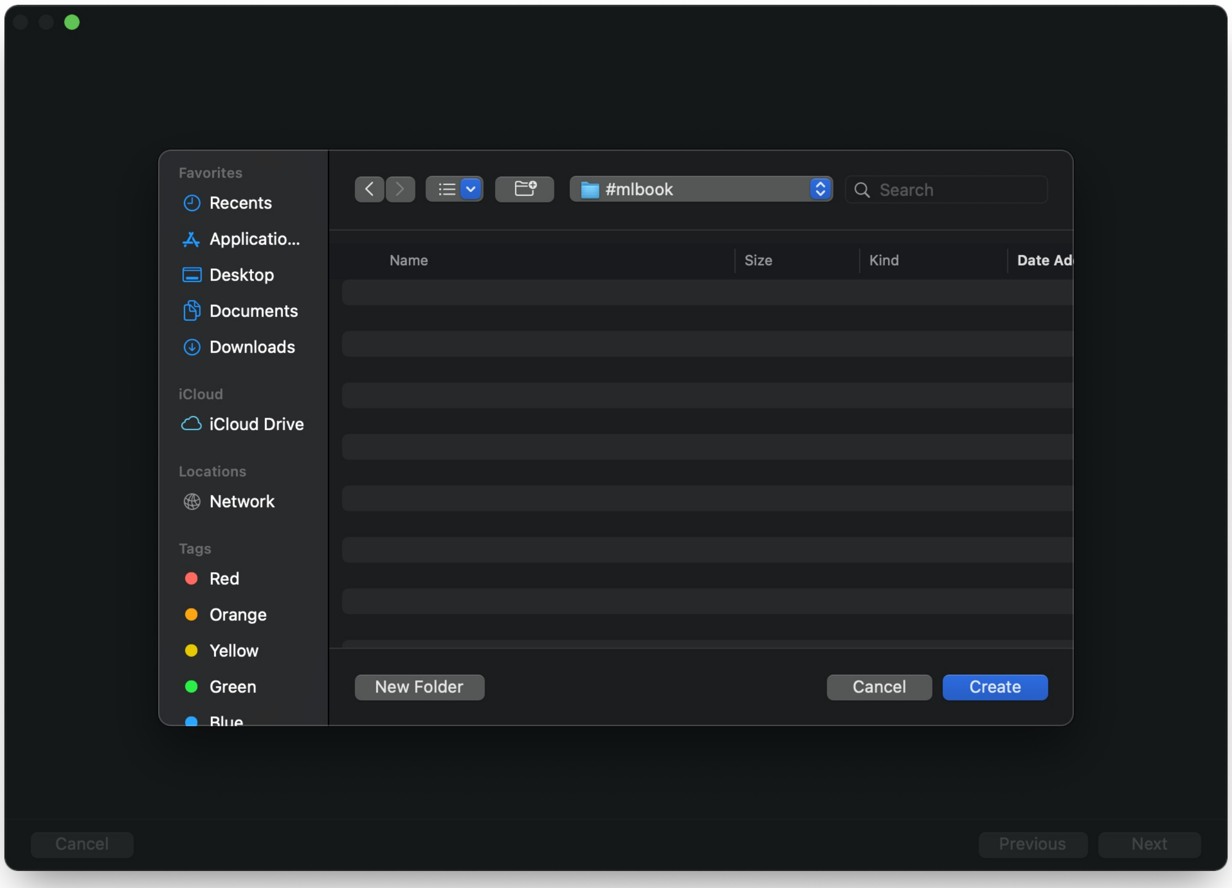
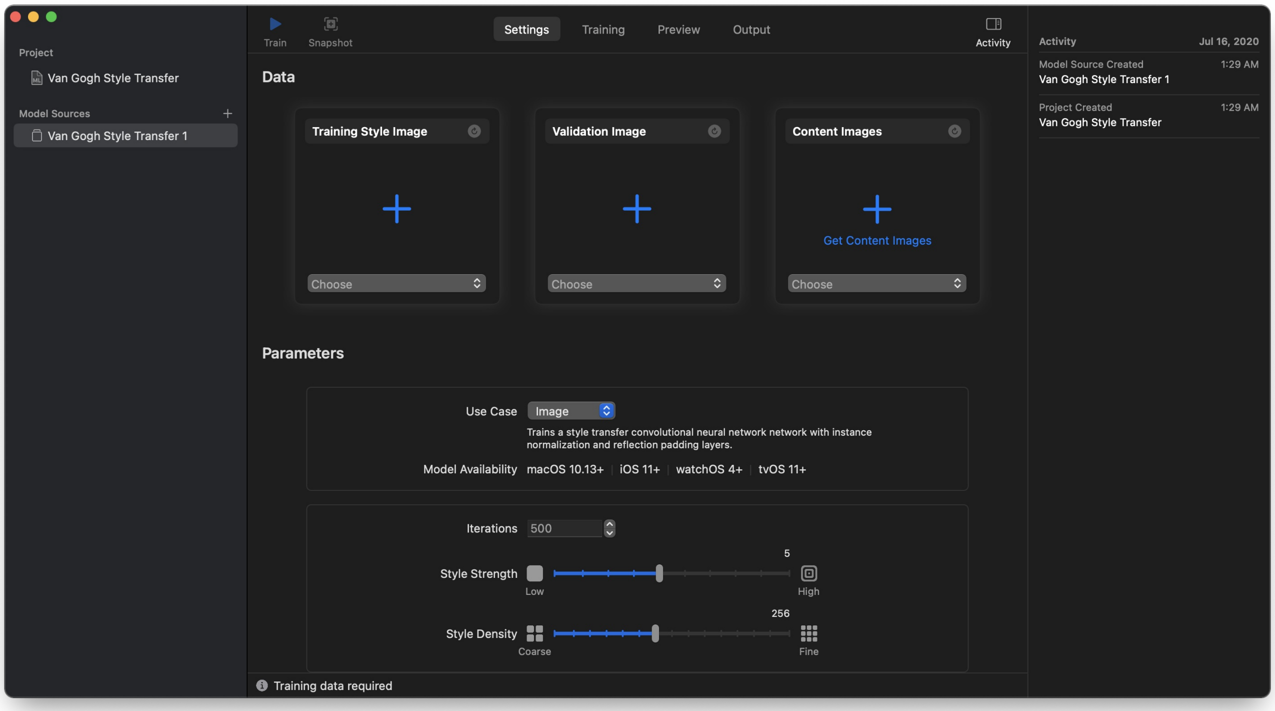


Figure 9-17: Saving your Create ML Project

Here, just choose a convenient location on your file system where you can quickly and easily access your project when you need it.

Add Style and Validation Image

Next, you'll need to input the data you prepared into Create ML. After you create your project, you'll be greeted with this dashboard:



Activity Jul 16, 2020

Model Source Created 1:29 AM
Van Gogh Style Transfer 1

Project Created 1:29 AM
Van Gogh Style Transfer

Use Case: Image

Trains a style transfer convolutional neural network network with instance normalization and reflection padding layers.

Model Availability: macOS 10.13+ | iOS 11+ | watchOS 4+ | tvOS 11+

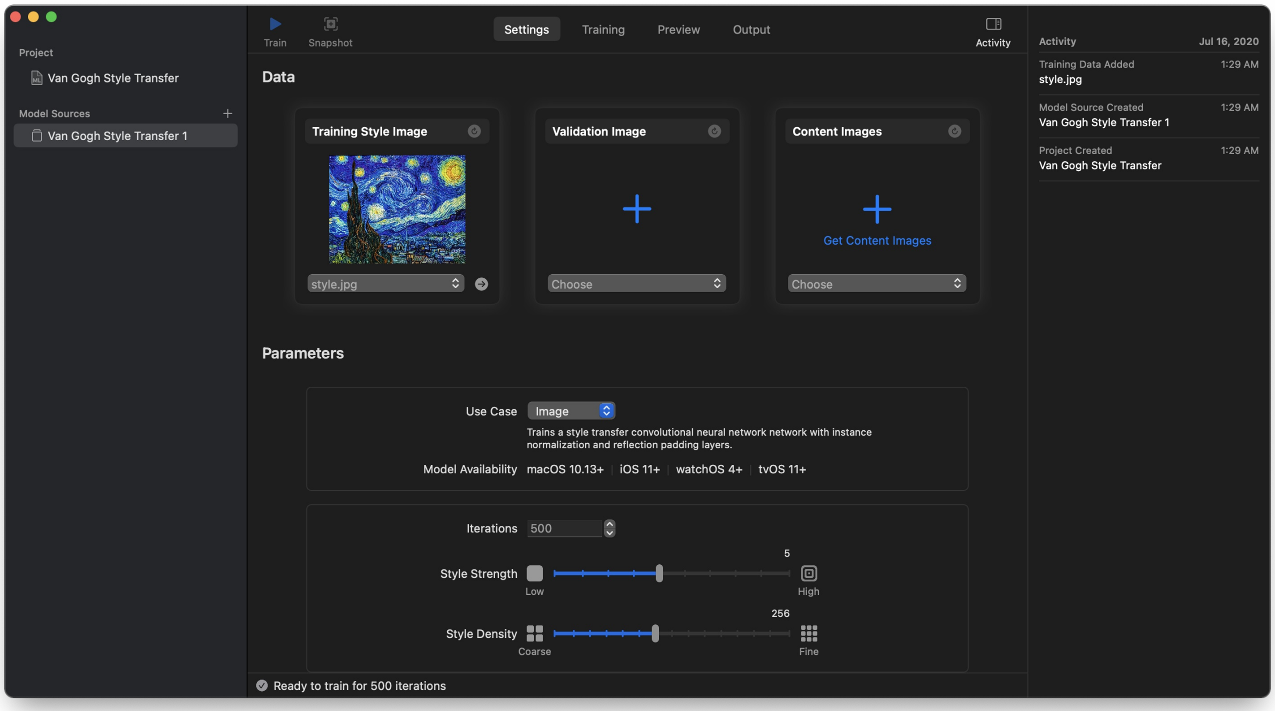
Iterations: 500

Style Strength: 5

Style Density: 256

Figure 9-18: Create ML Dashboard

You'll see three main boxes, where you can drag the corresponding data. First, drag your *style image* into the box which says **Training Style Image**.



Use Case: Image
Trains a style transfer convolutional neural network network with instance normalization and reflection padding layers.
Model Availability: macOS 10.13+ | iOS 11+ | watchOS 4+ | tvOS 11+

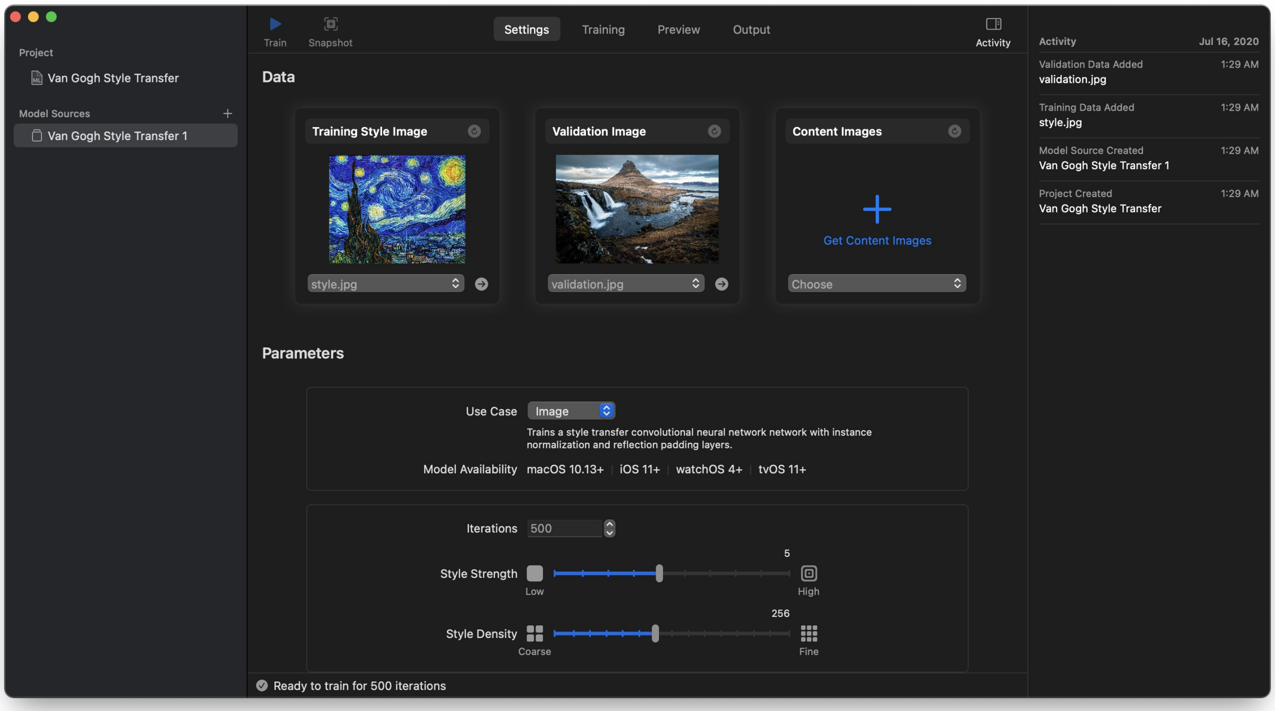
Iterations: 500
Style Strength: 5 (Low – High)
Style Density: 256 (Coarse – Fine)

Ready to train for 500 iterations

Activity	Jul 16, 2020
Training Data Added style.jpg	1:29 AM
Model Source Created Van Gogh Style Transfer 1	1:29 AM
Project Created Van Gogh Style Transfer	1:29 AM

Figure 9-19: Adding Style Image

Then, drag your *validation image* into the **Validation Image** box.



Training Style Image: style.jpg
Validation Image: validation.jpg
Content Images: Choose

Use Case: Image
Trains a style transfer convolutional neural network network with instance normalization and reflection padding layers.
Model Availability: macOS 10.13+ | iOS 11+ | watchOS 4+ | tvOS 11+

Iterations: 500
Style Strength: 5 (Low – High)
Style Density: 256 (Coarse – Fine)

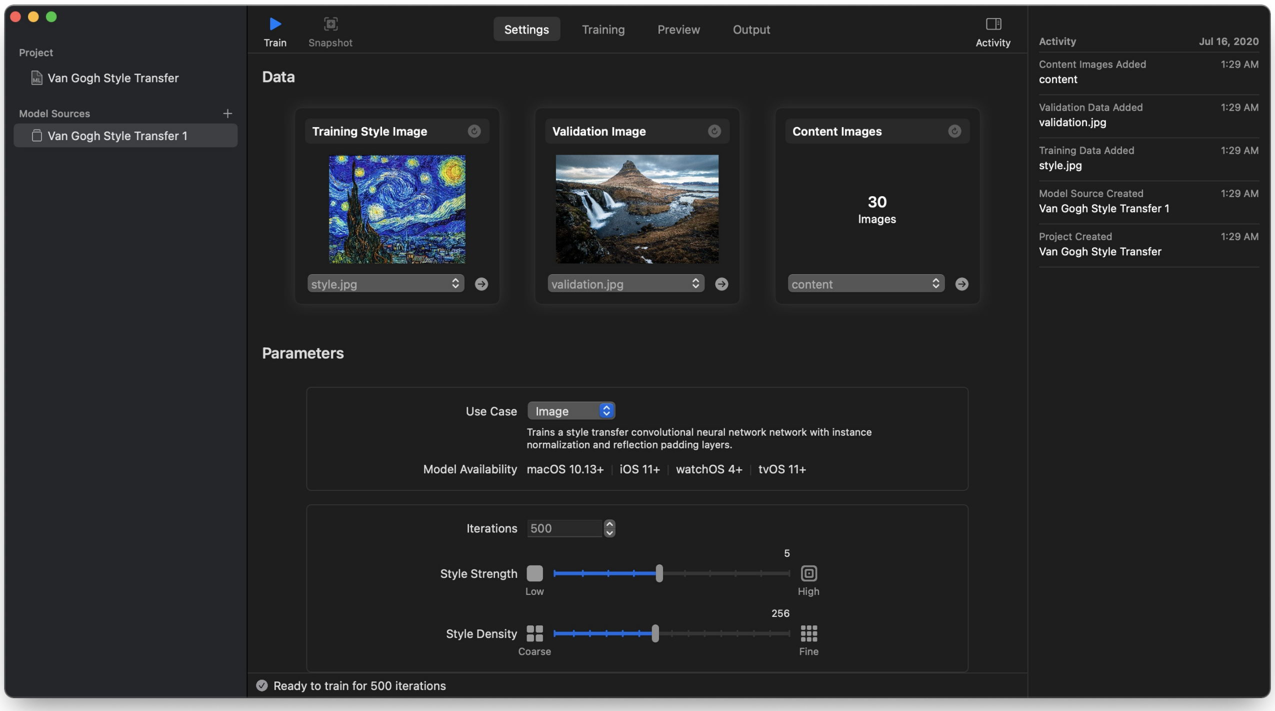
Ready to train for 500 iterations

Validation Data Added 1:29 AM
validation.jpg
Training Data Added 1:29 AM
style.jpg
Model Source Created 1:29 AM
Van Gogh Style Transfer 1
Project Created 1:29 AM
Van Gogh Style Transfer

Figure 9-20: Adding Validation Image

Add Content Images

Once both of these have been dragged in, you'll see previews for each appear in their corresponding boxes. Finally, drag your folder of *content images* into the box which says **Content Images**.



Activity	Jul 16, 2020
Content Images Added content	1:29 AM
Validation Data Added validation.jpg	1:29 AM
Training Data Added style.jpg	1:29 AM
Model Source Created Van Gogh Style Transfer 1	1:29 AM
Project Created Van Gogh Style Transfer	1:29 AM

Training Style Image: style.jpg
Validation Image: validation.jpg
Content Images: 30 Images, content

Use Case: Image
Trains a style transfer convolutional neural network network with instance normalization and reflection padding layers.
Model Availability: macOS 10.13+ | iOS 11+ | watchOS 4+ | tvOS 11+

Iterations: 500
Style Strength: 5
Style Density: 256

Ready to train for 500 iterations

Figure 9-21: Adding Content Images

This time, you'll see the number of images in your folder — instead of a preview of any particular image. Once everything looks right, you know you're all set!

Model Parameters

Before you start training, let's define some of the options you're presented with before you train your model. You can configure how your model will match the style image using both style strength and style density parameters.

Use Case

The use case parameter allows you to either optimize for video style transfer — which supports up to 120 frames per second! Or, you can optimize for image style transfer. For this chapter, we'll be using images; however, if you have videos you want to try this on, feel free to go with the video option!

Iterations

Later, we'll explore how you can use iterations to improve the performance of your model, but for now, you can leave it at the default. If you have an older Mac, which you believe might take too long to train, you may reduce the number of iterations.

Style Strength

As the name suggests, *style strength* defines how much of the content image gets stylized. If you only want your style to be mildly transferred over, you would choose a low style strength. On the flip side, if you want your content image to be completely restyled, you should choose a higher style strength.

Style Density

This parameter, *style density*, provides more nuanced control over your results. With this parameter, you can adjust whether you want a course or fine stylized result.

Depending on what you select for this parameter, Create ML will divide your image into a grid. It will then look at each square in that grid as a reference for the style. If you choose a fine stylization, your model will transfer details like brush strokes and colors. If you choose a course stylization, your model may focus more on shapes and textures.

Training, Adjusting, and Testing

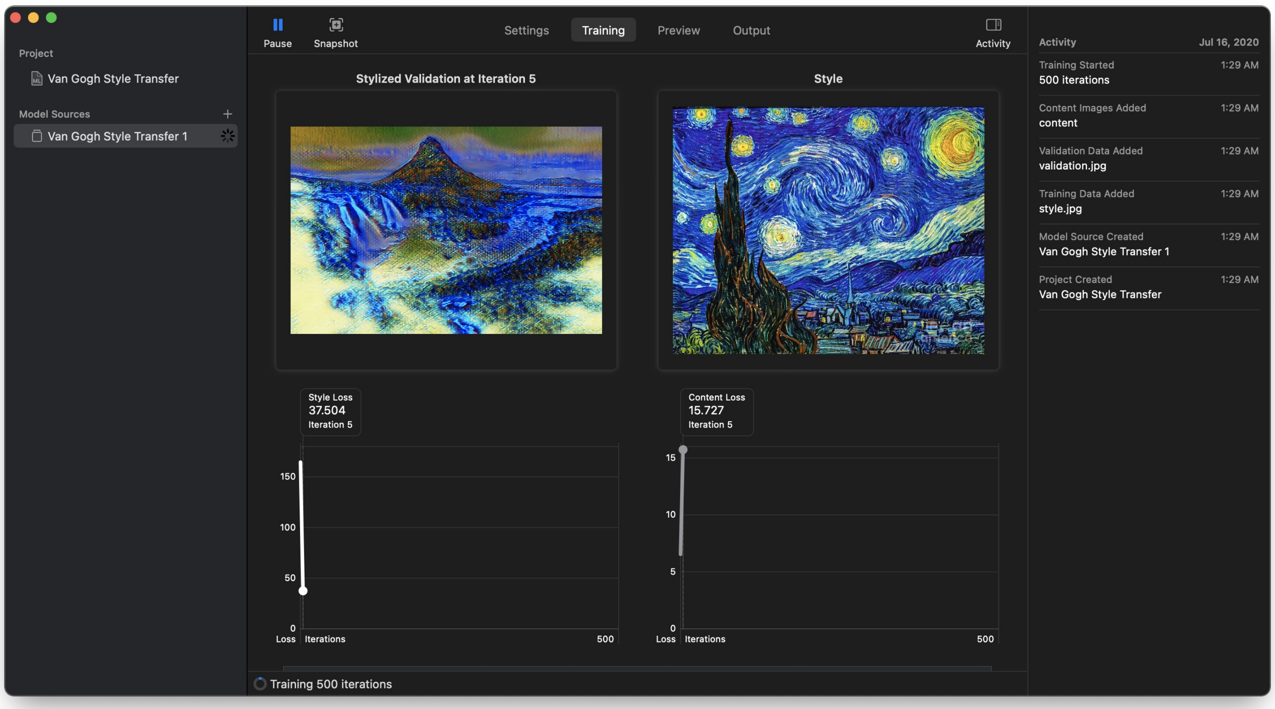
Now that everything is setup and ready-to-go, you're ready to train your model. Along the way, you'll learn how to adjust your model training, including creating model snapshots and pausing training.

Training your Model

After you've inputted everything, you're finally ready to train your model. Here, we'll look at each stage in the model training process and take a moment to look at all of the different options you have while training.

Start Training

Once everything is loaded into the three boxes on the **Settings** page, click the blue **Train** button (shaped like a triangle) in the upper-left of your Create ML window. This will start the training process.



Activity Jul 16, 2020

Training Started 1:29 AM
500 iterations

Content Images Added 1:29 AM
content

Validation Data Added 1:29 AM
validation.jpg

Training Data Added 1:29 AM
style.jpg

Model Source Created 1:29 AM
Van Gogh Style Transfer 1

Project Created 1:29 AM
Van Gogh Style Transfer

Figure 9-22: Training your Style Transfer Model

Creating Model Snapshots

As your model goes through each iteration of training, you'll be able to see how the model evolves — both through graphs and changes to your validation image. If you like how a certain iteration looks, use the **Snapshot** button in the menu bar to save that iteration. You can then go back to that model and tweak it for use later on.

Each model snapshot is its own Core ML model, so if you don't like the result at the end of training, you can simply use one of your snapshots in your apps instead — no need to start training from scratch again!

Pausing Training

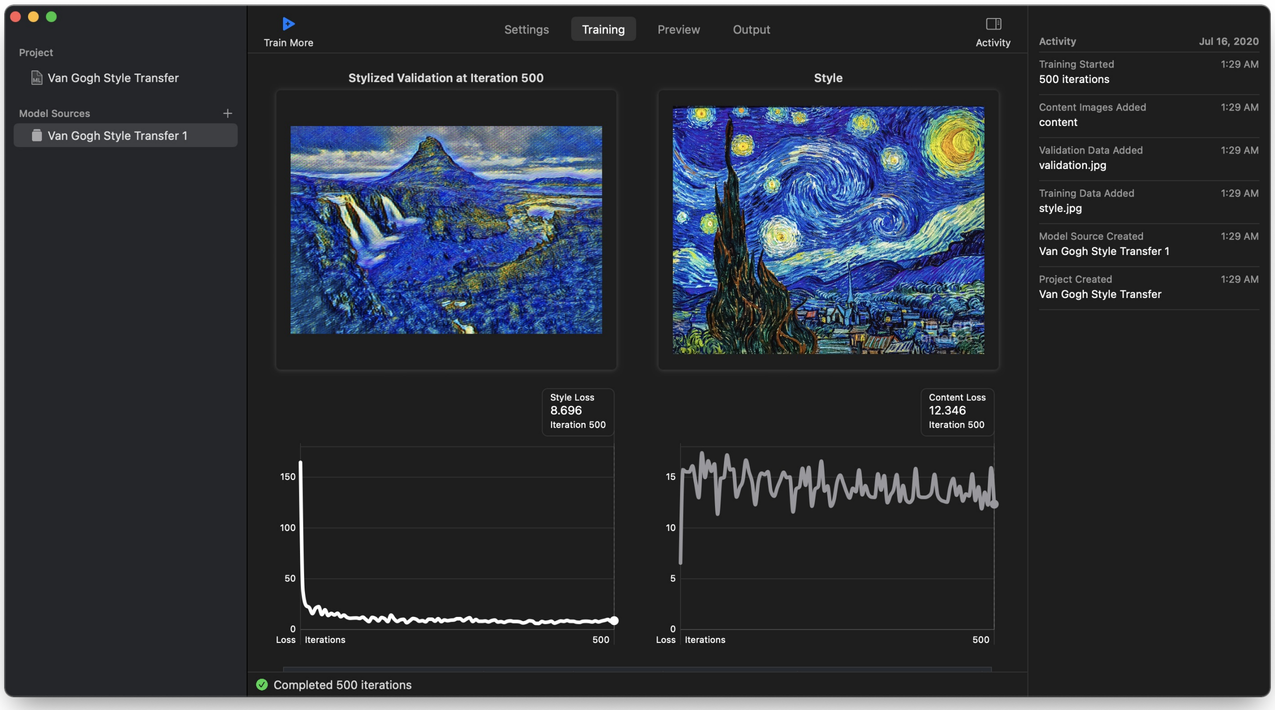
While you'll learn about fine-tuned model control in the next section of this chapter, let's take a moment to cover pausing. If you don't feel like model training needs to continue, you can pause and resume your model training at any time. Each time you pause training, a snapshot gets trained automatically.

Handling Trained Model

Once you've finished training, you have a few options left to improve your model, test your model, or share it with colleagues to provide input. Let's explore these options before we end this section.

Adding Iterations

If you've trained your model with 500 iterations, but you noticed that the style loss graph (left graph) hasn't yet converged to a certain value, you can add more iterations without needing to start training from scratch.

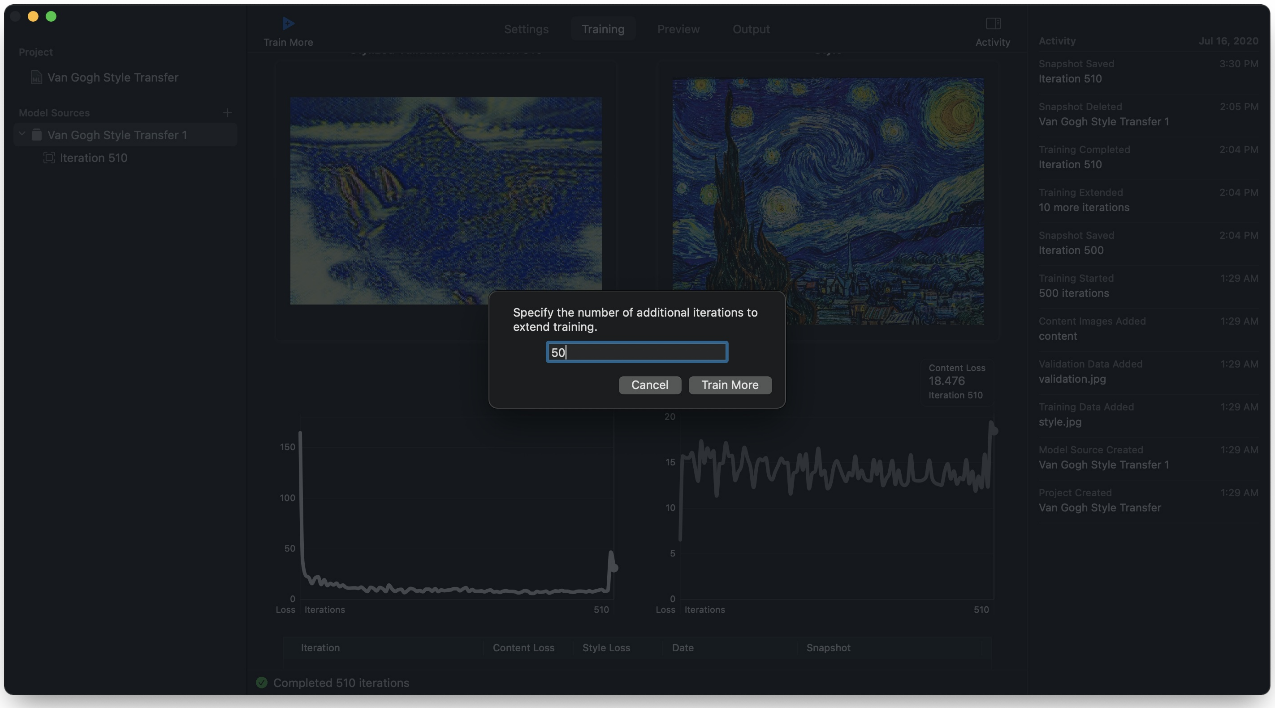


Completed 500 iterations

Activity	Jul 16, 2020
Training Started 500 iterations	1:29 AM
Content Images Added content	1:29 AM
Validation Data Added validation.jpg	1:29 AM
Training Data Added style.jpg	1:29 AM
Model Source Created Van Gogh Style Transfer 1	1:29 AM
Project Created Van Gogh Style Transfer	1:29 AM

Figure 9-23: Adding Training Iterations

In the upper left, you'll notice that the previous **Train** button now has a plus button and the words **Train More** under it. To train more, just click that button and enter the number of additional iterations you want.



Specify the number of additional iterations to extend training.

50

Content Loss
18.476
Iteration 510

Completed 510 iterations

Activity — Jul 16, 2020

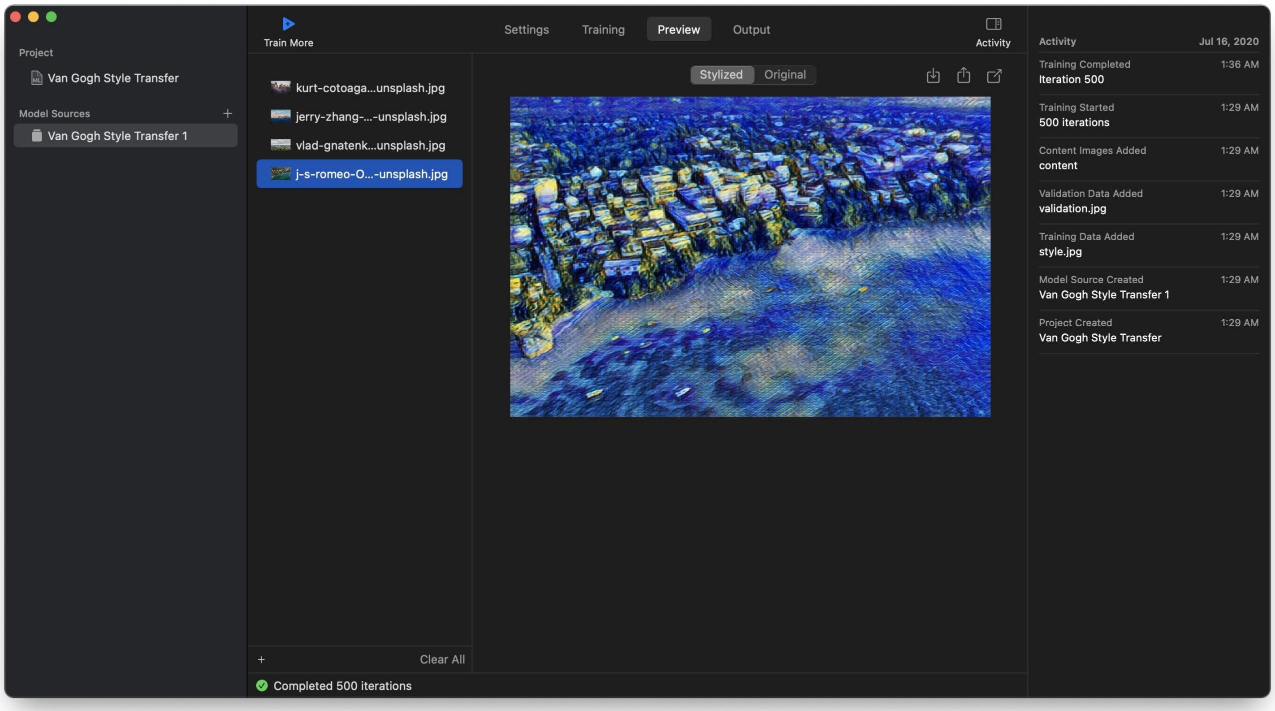
Snapshot Saved — Iteration 510	3:30 PM
Snapshot Deleted — Van Gogh Style Transfer 1	2:05 PM
Training Completed — Iteration 510	2:04 PM
Training Extended — 10 more iterations	2:04 PM
Snapshot Saved — Iteration 500	2:04 PM
Training Started — 500 iterations	1:29 AM
Content Images Added — content	1:29 AM
Validation Data Added — validation.jpg	1:29 AM
Training Data Added — style.jpg	1:29 AM
Model Source Created — Van Gogh Style Transfer 1	1:29 AM
Project Created — Van Gogh Style Transfer	1:29 AM

Figure 9-24: Specifying Additional Iterations

Your model will now continue training for your number of specified additional iterations. Again, you can always revert to a model snapshot or pause training at any time.

Testing the Model

In the dataset I provided, I have four images as testing images. Since style transfer is a very subjective process, it's up to you how you want your results. If you head to the **Preview** tab, you'll be able to drag in your test images and see how your model works!



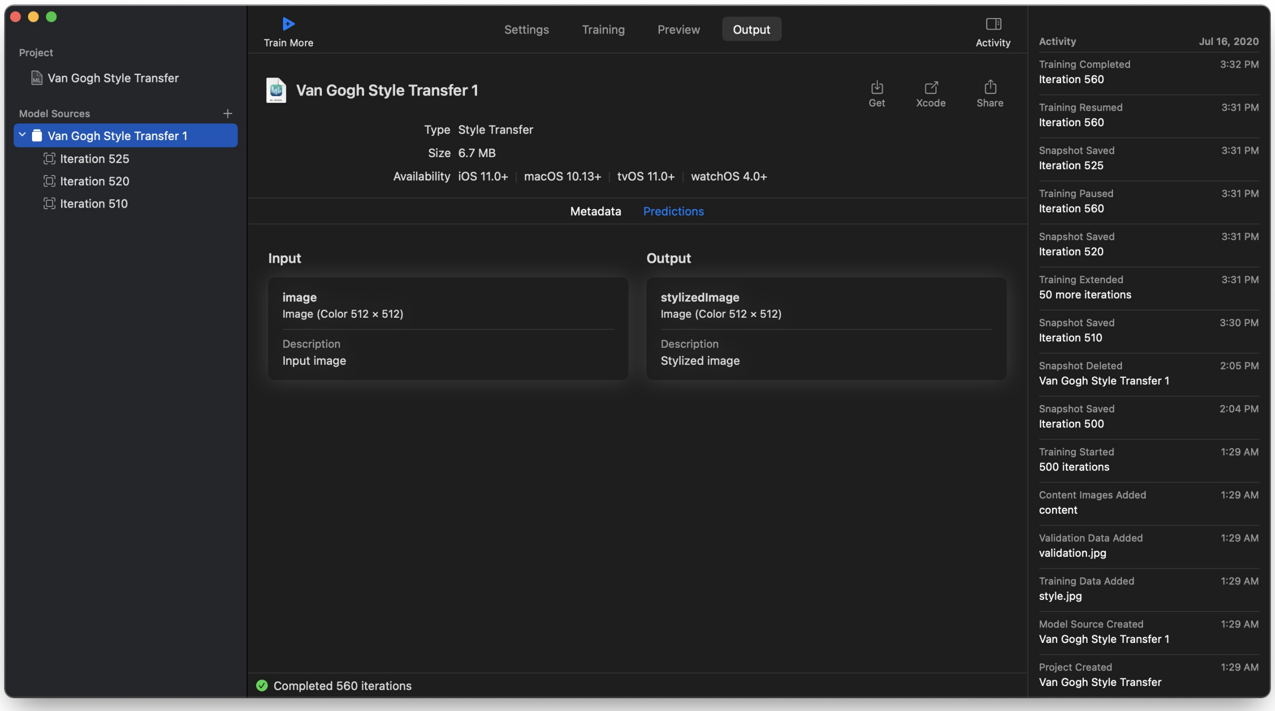
Activity	Jul 16, 2020
Training Completed Iteration 500	1:36 AM
Training Started 500 iterations	1:29 AM
Content Images Added content	1:29 AM
Validation Data Added validation.jpg	1:29 AM
Training Data Added style.jpg	1:29 AM
Model Source Created Van Gogh Style Transfer 1	1:29 AM
Project Created Van Gogh Style Transfer	1:29 AM

Figure 9-25: Previewing your Style Transfer Model

I like my results, so I'll stick with my model as-is; however, you might not like it as much, so you can change your model parameters and train again — or, you can add iterations to your current model and continue training.

Exporting your Model

Once you're done, you can head to the output tab to learn about your final model. In the **Metadata** tab, you can preview the name and description you provided when you created your Create ML project, and in the **Predictions** tab, you can see the input and output the model requires.



Van Gogh Style Transfer 1

Type Style Transfer
Size 6.7 MB
Availability iOS 11.0+ | macOS 10.13+ | tvOS 11.0+ | watchOS 4.0+

Input

image
Image (Color 512 × 512)
Description
Input image

Output

stylizedImage
Image (Color 512 × 512)
Description
Stylized image

Completed 560 iterations

Activity — Jul 16, 2020

Training Completed Iteration 560	3:32 PM
Training Resumed Iteration 560	3:31 PM
Snapshot Saved Iteration 525	3:31 PM
Training Paused Iteration 560	3:31 PM
Snapshot Saved Iteration 520	3:31 PM
Training Extended 50 more iterations	3:31 PM
Snapshot Saved Iteration 510	3:30 PM
Snapshot Deleted Van Gogh Style Transfer 1	2:05 PM
Snapshot Saved Iteration 500	2:04 PM
Training Started 500 iterations	1:29 AM
Content Images Added content	1:29 AM
Validation Data Added validation.jpg	1:29 AM
Training Data Added style.jpg	1:29 AM
Model Source Created Van Gogh Style Transfer 1	1:29 AM
Project Created Van Gogh Style Transfer	1:29 AM

Figure 9-26: Exporting your Trained Model

You also have three options to export your model. The **Get** button will download your model, the **Xcode** button will open it in Xcode, and the **Share** button will allow you to send your trained `.mlmodel` file via Messages, Mail, or AirDrop.

Conclusion

In this chapter, you learned about action classification and style transfer models, which are at the cutting edge of machine learning. I first showed you how to find and source videos for an action classifier and then, you trained your own fitness classifier. Later in the chapter, you learned about style transfer.

Style transfer, a significant part of the chapter, is an emerging field of machine learning is likely to play a major role in many fields currently untouched by computer science. You first learned about the applications of style transfer and then learned about its future merits. We ended the chapter by creating a style transfer app which mimics the style of "Starry Night."

Chapter 10
Supercharge ML Workflow with these Tips and Tricks

In this chapter, I will wrap up this book by giving advice on how to further your machine learning adventures. Hopefully, this book has sparked your curiosity in machine learning, and you will continue learning about the topic. I'll also discuss tips and tricks to help you supercharge your machine learning workflow.

Finally, I'll leave you off with some food for thought and some cutting-edge apps which are making use of machine learning technology to revolutionize their industries. By the end of the chapter, you'll have clear picture about how to proceed with your machine learning endeavors.

10-1 Machine Learning Design

In this section, you'll learn ways to improve your skills and supercharge your development workflows with pro tips and tricks.

Often, when you think of machine learning, it seems like something which happens under-the-hood and doesn't directly concern the user. However, the design of your machine learning models is critical to the experience that the user is going to have. Thinking about the user while creating your machine learning experiences is going to help make your user's experience much more meaningful.

Choose Datasets Carefully

One of the defining aspects of your machine learning model is the data used to power it. For this reason, it's very important to select your dataset to match your user's expected input.

Imagine you're training an image classifier to detect whether the image has a dog or not, and you expect your user to input images with different dog breeds. If you, for example, only trained your model with Golden Retrievers, the model may not recognize your neighbor's Cocker Spaniel as well. Because of this, it's crucial that you think about your user as you're choosing a training and validation dataset.

Use Data Augmentation

Often, if you're creating your own datasets, they won't be large enough to train a robust model. However, as we've discussed throughout the book, data augmentation is a great technique to increase your dataset. Through augmentation, you could significantly increase the amount of data you have, through cropping, flipping, or scaling your images.

Create ML provides you with this option when you first enter your dataset. By checking the boxes, you can choose the type of data augmentation you'd like to use. If you do choose to do this, make sure you test your model's performance with and without it, because in some use cases, augmentation may result in a decreased performance.

Embrace Limitations

Even when you do choose your dataset carefully and use augmentation to increase its size, there will be some cases which aren't covered: it isn't humanely possible. Instead of trying to obscure these limitations from your user, you should make them clear in your app. Your users will thank you for this and trust your app more.

Let's pretend that you've created a fruit detection app. You've used a varied dataset, so it can find all sorts of fruits from apples to bananas. However, your app cannot find fruits easily in the outdoors. If you don't communicate this limitation with your users, your app may come across as finicky or inaccurate if they try and use your app in these situations.

10-2 Model Conversion and Usage

In this book, you first learned about APIs which give you results in the form of a Core ML model, but at the end, you learned to use Firebase, which did not use Core ML at all. Similarly, there are other platforms which give you finer control over certain aspects of model training.

Using Model Conversion

A common “fear” of people starting out with machine learning is converting model formats. While it seems daunting at first, some frameworks such as Caffe and Keras provide much more advanced tools than Turi Create and Create ML. If you ever need these tools, there are ways to convert your models to the Core ML format, even if you can’t export them directly. And, since you know how to code in Python now, it will be much easier to learn such tools.

To convert your models from the supported models to the Core ML format, you can use the built in `coremltools`. Simply add the following in the Python file you use to train your model. Let’s look at an example of how you’d do this in real life.

Import Framework

To start, you’ll need to import the `coremltools` framework, which was installed alongside with the Xcode installation. To import it in your Python file, insert the following import statements:

```
import coremltools
```

Convert a Model

Now, you need to actually convert the model from your source format to the Core ML format. Make sure you add this after your model has been created, since this line of code references a created model. Here is a generalized version, where `<model_format>` is one of the supported models and `<model_name>` is how you're referencing the model in the code.

```
coreml_model = coremltools.converters.<model_format>.convert
    ('<model_name>.<model_format>')
```

Assuming your model was named `my_caffe_model.caffemodel`, and you were using a Caffe model, for example, the same line of code would look like this:

```
coreml_model = coremltools.converters.caffe.convert
    ('my_caffe_model.caffemodel')
```

Export a Model

Finally, you'll just need to export your model. You can name your model whatever you like, and keep the `coreml_model` parameter the same to convert it to a Core ML model. Here's how you do it:

```
coremltools.utils.save_spec
    (coreml_model, 'my_model.mlmodel')
```

Using Models Without Core ML

Whatever you've learned just now was assuming that you want to use Core ML. However, as you've seen in the cloud machine learning chapter, Core ML does have its limitations, especially the lack of cloud computing abilities. For this reason, it may make more sense to explore other options if you're looking for added capabilities such as these. Here are some good ones to consider:

TensorFlow

Through TensorFlow Lite, you can easily connect your iOS app to a powerful, robust machine learning model trained using TensorFlow. This platform is great for training models with fine-tuned control and excellent data visualization tools. It also has a ton of starter projects to help you learn the ropes before you begin.

IBM Watson

IBM has constantly been in the headlines for their cutting-edge machine learning research, and just recently, Apple has announced support for IBM Watson tools in iOS apps. This means that your apps can now take advantage of the robust features that IBM has to offer, while reaping the benefits of Apple's hardware-software integration.

Microsoft Azure

The Microsoft Azure suite also integrates well with Swift apps. Similar to what Firebase offers, Azure allows developers to take advantage of pre-trained models and use cloud computing services. You can also use their easy-to-use machine leaning model training tool called Microsoft Custom Vision, which allows you to create models using a drag-and-drop interface.

Conclusion

In this chapter, you learned some quick tips and tricks, which should help you supercharge your machine learning workflow. Now you should get a clearer picture about how to continue your machine learning endeavors. Hopefully, this book has sparked your curiosity in machine learning, and you will continue learning about the topic in your own time.