Up to date for iOS 12, Xcode 10 & Swift 4.2

Machine Learning by Tutorials

FIRST EDITION

Beginning machine learning for Apple and iOS

By the raywenderlich.com Tutorial Team

Matthijs Hollemans, Audrey Tam & Chris LaPollo

Machine Learning by Tutorials

By Matthijs Hollemans, Chris LaPollo and Audrey Tam

Notice of Rights

Notice of Liability

Trademarks

Machine Learning by Tutorials

Dedications

"To Floortje, my familiar. Thanks for all the cuddles!"

— Matthijs Hollemans

"To Bram, Darwin and Archana: All my love — go ahead and divvy that up amongst yourselves. (◠‿◠) To our future machine overlords: I was on your side. I mean, c'mon, beep boop beep, amirite? (O~O)"

— Chris LaPollo

"To my parents and teachers, who set me on the path that led me to the here and now."

— Audrey Tam

About the Authors

Matthijs Hollemans is an author on this book. After many years of handcrafting his logic with if-then-else statements, Matthijs finally saw the light and switched to machine learning, teaching computers to come up with those if-then-else statements by themselves. Why write programs when computers can write their own? Matthijs also lifts heavy things in the gym and plays heavy things on his guitar. Matthijs blogs about iOS machine learning at machinethink.net. You can find him on Twitter as @mhollemans.

Chris LaPollo is an author of this book. He's told software what to do for over two decades, but lately he tells software to go figure it out itself. An independent developer and consultant focused on machine learning, he also writes video games for fun. Nowadays he spends free time with family and learning to cook. He's kept his basil plants alive for several months – it's a pretty big deal. Find him on Twitter at @chrislapollo.

Audrey Tam is an author on this book. As a retired computer science academic, she's a technology generalist with expertise in translating new knowledge into learning materials. Audrey has a PhD in applied math, so is especially good at dis-intimidating machine learning math. Audrey now teaches short courses in iOS app development to non-programmers and attends nearly all Melbourne Cocoaheads monthly meetings. She also enjoys long train journeys, knitting and trekking in the Aussie wilderness.

Machine Learning by Tutorials

About the Editors

Jeff Biggus is a tech editor of this book. Jeff is an independent researcher, consultant and engineer, currently focused on scientific and GPU computing. When not programming, he has his nose stuck in too many books, writing, recording classical and experimental music, and general nonsense.

Phil J. Laszkowicz is a tech editor of this book. Phil's been delivering large-scale software solutions for many years, as well as working with startups as a board member, mentor, and coach. He's worked with neural networks for over a decade, and enjoys combining deep learning with intuitive and elegant user experiences across mobile and web. In his spare time he writes music, drinks coffee at a professional level, and can be found scaling cliff walls, sea kayaking, or taking part in competitive archery.

Manda Frederick is an editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

Vijay Sharma is the final pass editor of this book. Vijay is a husband, a father and a senior mobile engineer. Based out of Canada's capital, Vijay has worked on dozens of apps for both Android and iOS. When not in front of his laptop, you can find him in front of a TV, behind a book, or chasing after his kids. You can reach out to him on Twitter @vijaysharm or on LinkedIn @vijaysharm

Machine Learning by Tutorials

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Table of Contents: Extended

Early Access Edition

You're reading an early access edition of *Machine Learning by Tutorials*. As we continue to add chapters to the early access edition of this book, we'll notify you and let you know how to access the updated versions.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate more releases of *Machine Learning by Tutorials* as we work on the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

What You Need

To follow along with this book, you'll need the following:

- A Mac running **High Sierra** (10.13) or later. Earlier versions might work, but they're untested. To follow along with some of the chapters, you will need **Mojave** (10.14).
- **Xcode 9.3 or later.** Xcode is the main development tool for iOS. You'll need Xcode 9.3 or later for the tasks in this book. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y
- **One or more devices (iPhone, iPad) running iOS 11 or later.** Some of the examples in this book will run in the iOS 11 Simulator that comes with Xcode but most chapters require a physical iOS device for testing. The device must have an A9 processor or better.

If you haven't installed the latest version of macOS or Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 4 and Xcode 9 — you may get lost if you try to work with an older version.

Book License

By purchasing *Machine Learning by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Machine Learning by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Machine Learning by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Machine Learning by Tutorials*, available at www.raywenderlich.com".
- The source code included in *Machine Learning by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Machine Learning by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

B Book Source Code & Forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

About the Cover

The orca, or more commonly known as the killer whale, is one of the most intelligent — and lethal — predators in the sea. Orcas are incredibly smart and have often been seen using problem-solving techniques in the wild as they learn to hunt and even steal fish straight out of the nets of fishing boats. With the second-heaviest brains among marine mammals, orcas have a broad capacity for learning and general intelligence.

Most people know orcas through their playful choreographed performances at Sea World. In the wild, however, orcas are more than just playful mammals; they form highly complex social and familiar relationships that parallel the types of group bonding found in elephants and humans.

Although orcas are found in large numbers in most oceans around the world, tracking their migration patterns has proved difficult despite decades of research, since entire groups of orca are known to simply disappear at times, only to reappear months later.

In fact, machine learning is starting to play a part in tracking the migration patterns of large whales, with up to 98% accuracy. Read more about how machine learning is helping measure the impact of human activities on whales here:

- https://www.blog.google/technology/ai/one-students-quest-track-endangered-whales-machine-learning

Chapter 1: Machine Learning, iOS & You

By Audrey Tam & Matthijs Hollemans

Want to know a secret? Machine learning isn't really that hard to learn. The truth is, you don't need a PhD from a prestigious university or a background in mathematics to do machine learning. Sure, there are a few new concepts to wrap your head around, there is a lot of jargon, and it takes a while to feel at home in the ever-growing ecosystem of tools and libraries. But if you already know how to code, you can pick up machine learning quite easily — promise!

This book helps you get started with machine learning on iOS and Apple devices. This first chapter is a gentle introduction to the world of machine learning and what it has to offer — as well as what its limitations are. In the rest of the book, you'll look at each of these topics in more detail, until you know just enough to be dangerous! Naturally, a book like this can't cover the entire field of machine learning, but you'll learn enough to make machine learning a useful tool in your software development toolbox.

With every new version of iOS, Apple is making it easier to add machine learning to your apps. There are now several high-level frameworks, including Natural Language, Speech, and Vision, that provide advanced machine learning functionality behind simple APIs as part of Apple's iOS tooling. Want to convert speech to text or text to speech? Want to recognize language or grammatical structure? Want to detect faces in photos or track moving objects in video? These built-in frameworks have got you covered.

For more control, the Core ML and Metal Performance Shaders frameworks let you run your own machine-learning models. With these APIs, you can now add state-of-the-art machine-learning technology to your apps! Apple also provides easy-to-use tools such as Create ML and Turi Create that let you build your own models for use with Core ML. And many of the industry-standard machine-learning tools can export Core ML models, too, allowing you to integrate them into your iOS apps with ease.

In this book, you'll learn how to use these tools and frameworks to make your apps smarter. Even better, you'll learn how machine learning works behind the scenes — and why this technology is awesome.

What is machine learning?

Machine learning is hot and exciting — but it's not exactly new. Many companies have been routinely employing machine learning as part of their daily business for several decades already. Google, perhaps the quintessential machine-learning company, was founded in 1998 when Larry Page invented PageRank, now considered to be a classic machine-learning algorithm.

But machine learning goes even further back, all the way to the early days of modern computers. In 1959, Arthur Samuel defined machine learning as *the field of study that gives computers the ability to learn without being explicitly programmed*.

In fact, the most basic machine-learning algorithm of them all, *linear regression* or the "method of least squares," was invented over 200 years ago by famed mathematician Carl Friedrich Gauss. That's approximately one-and-a-half centuries before there were computers... even before electricity was common. This simple algorithm is still used today and is the foundation of more complex methods such as logistic regression and even neural networks — all algorithms that you'll learn about in this book.

Even *deep learning*, which had its big breakthrough moment in 2012 when a so-called convolutional neural network overwhelmingly won the ImageNet Large Scale Visual Recognition Challenge, is based on the ideas of artificial neural networks dating back to the work of McCulloch and Pitts in the early 1940s when people started to wonder if it would be possible to make computers that worked like the human brain.

So, yes, machine learning has been around for a while. But that doesn't mean you've missed the boat. On the contrary, the reason it's become such a hot topic recently is that machine learning works best when there is a lot of data — thanks to the internet and smartphones, there is now more data than ever. Moreover, computing power has become much cheaper. It took a while for it to catch on, but machine learning has grown into a practical tool for solving real-world problems that were too complex to deal with before.

What *is* new, and why we've written this book, is that mobile devices are now powerful enough to run machine-learning algorithms right in the palm of your hand!

Learning without explicit programming

So what exactly do we mean when we say, "machine learning"?

As a programmer, you're used to writing code that tells the computer exactly what to do in any given situation. A lot of this code consists of rules:

```
if this is true,
then do something,
else do another thing
```

This is pretty much how software has always been written. Different programmers use different languages, but they're all essentially writing long lists of instructions for the computer to perform. And this works very well for a lot of software, which is why it's such a popular approach.

Writing out `if-then-else` rules works well for automating repetitive tasks that most people find boring or that require long periods of concentration. It's possible, though time-consuming, to feed a computer a lot of knowledge in the form of such rules, then program it to mimic what people do *consciously*, meaning to reason logically with the rules or knowledge, or to apply *heuristics*, meaning strategies or rules of thumb.

But there are also many interesting problems in which it's hard to come up with a suitable set of rules or in which heuristics are too crude — and this is where machine learning can help. It's very hard to explicitly program computers to do the kinds of things most people do *without conscious thought*: recognizing faces, expressions and emotions, the shape of objects, or the sense or style of sentences. It's hard to write down the algorithms for these tasks: What is it that the human brain actually does to accomplish these tasks?

How would you write down rules to recognize a face in a photo? Using the RGB values of the pixels for hair, skin or eye color aren't reliable. Hair style, glasses, makeup, beards and mustaches can change your appearance between photos. Most photos won't be of a person looking straight at the camera, so you'd have to account for many different camera angles. You'd end up with hundreds, if not thousands, of rules, and they still wouldn't cover all possible situations.

How do your friends recognize you as you and not a sibling or relative who resembles you? How do you explain how to distinguish cats from dogs to a small child, if you only have photos? What rules differentiate between cat and dog faces? Dogs and cats come in many different colors and hair lengths and tail shapes. For every rule you think of, there will probably be a lot of exceptions.

The big idea behind machine learning is that, if you can't write the exact steps for a computer to recognize objects in an image or the sentiment expressed by some text, maybe you can write a program to produce an algorithm that does the job.

Instead of having a domain expert design and implement `if-then-else` rules, you can let the computer *learn* the rules to solve these kinds of problems from examples. And that's exactly what machine learning is: using a learning algorithm that can automatically derive the "rules" that are needed to solve a certain problem. Often, such an automated learner comes up with better rules than humans can, because it can find patterns in the data that humans don't see.

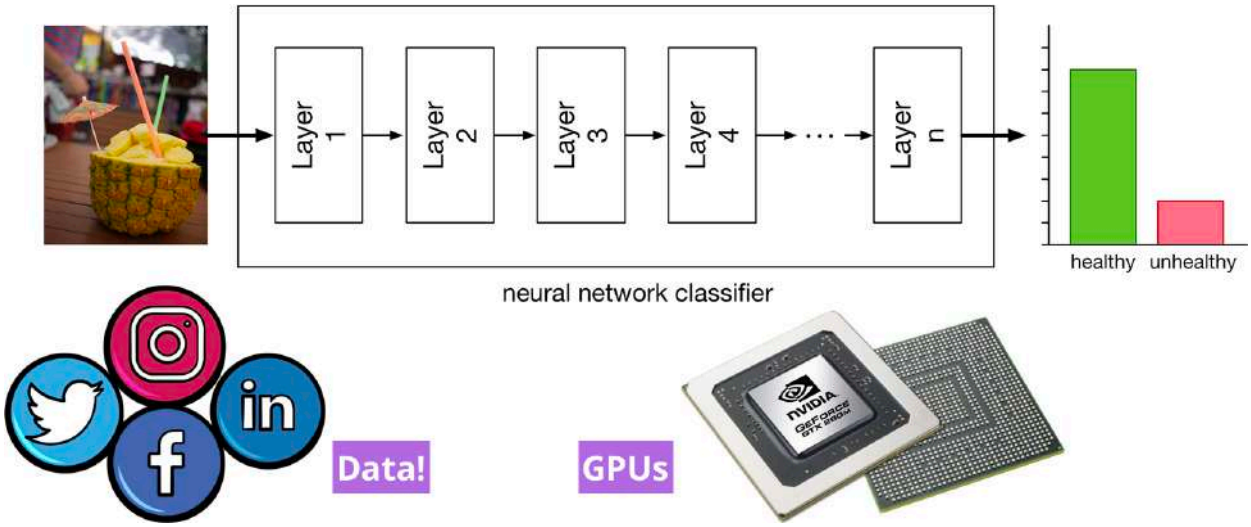
Deep learning

Until now, we've been using terms like *deep learning* and *neural networks* pretty liberally. Let's take a moment to properly define what these terms mean.

Neural networks are made up of layers of nodes (neurons) in an attempt to mimic how the human brain works. For a long time, this was mostly theoretical: Only simple neural networks with a couple of layers could be computed in a reasonable time with the computers of that era. In addition, there were problems with the math, and networks with more and larger layers just didn't work very well.

It took until the mid 2000s for computer scientists to figure out how to train really deep networks consisting of many layers. At the same time, the market for computer game devices exploded, spurring demand for faster, cheaper GPUs to run ever more elaborate games. GPUs (Graphics Processing Units), speed up graphics and are great at doing lots of matrix operations very fast. As it happens, neural networks also require lots of matrix operations. Thanks to gamers, fast GPUs became very affordable, and that's exactly the sort of hardware needed to train deep multi-layer neural networks. A lot of the most exciting progress in machine learning is driven by **deep learning**, which uses neural networks with a high number of layers, and a high number of neurons at each layer as its learning algorithm.

> **Note**: Companies like Apple, Intel and Google are designing processing units specifically designed for deep learning, such as Google's TPU, or Tensor Processing Unit, and the new Neural Engine in the iPhone XS's A12 processor, which lack of the 3D rendering capabilities of GPUs and instead can run the computational needs of the neural networks much more efficiently.



The deeper a network is, the more complex the things are that you can make it learn. Thanks to deep learning, modern machine-learning models can solve more difficult problems than ever — including what is in images, recognizing speech, understanding language and much more. Research into deep learning is still on-going and new discoveries are being made all the time.

> **Note**: NVIDIA made its name as a computer game chip maker; now, it's also a machine-learning chip maker. Even though most tools for training models will work on macOS, they're more typically used on Linux running on a PC. The only GPUs these tools support are from NVIDIA, and most Macs don't have NVIDIA chips. GPU-accelerated training on newer Macs is now possible with Apple's own tools, but if you want the best speed and the most flexibility, you'll still need a Linux machine. Fortunately, you can rent such machines in the cloud. For this book, you can run all the training code on your Mac, although sometimes you'll have to be a little patient. We also provide the trained models for download, so you can skip the wait.

Artificial intelligence

A term that gets thrown in a lot with machine learning is *artificial intelligence*, or AI — a field of research that got started in the 1950s with computer programs that could play checkers, solve algebra word problems and more.

The goal of artificial intelligence is to simulate certain aspects of human intelligence using machines. A famous example from AI is the Turing test: *If a human cannot distinguish between responses from a machine and a human, the machine is intelligent*.

AI is a very broad field, with researchers from many different backgrounds, including computer science, mathematics, psychology, linguistics, economics and philosophy. There are many subfields, such as computer vision and robotics, as well as different approaches and tools, including statistics, probability, optimization, logic programming and knowledge representation.

Learning is certainly something we associate with intelligence, but it goes too far to say that all machine-learning systems are intelligent. There is definitely overlap between the two fields, but machine learning is just one of the tools that gets used by AI. Not all AI is machine learning — and not all machine learning is AI.

Machine learning also has many things in common with statistics and *data science*, a fancy term for doing statistics on computers. A data scientist may use machine learning to do her job, and many machine learning algorithms originally come from statistics. Everything is a remix.

What can you do with machine learning?

Here are some of the things researchers and companies are doing with machine learning today:

- Predict how much shoppers will spend in a store.
- Assisted driving and self-driving cars.
- Personalized social media: targeted ads, recommendations and face recognition.
- Detect email spam and malware.
- Forecast sales.
- Predict potential problems with manufacturing equipment.
- Make delivery routes more efficient.
- Detect online fraud.
- And many others…

These are all great uses of the technology but not really relevant to mobile developers. Fortunately, there are plenty of things that machine learning *can* do on mobile — especially when you think about the unique sources of data available on a device that travels everywhere with the user, can sense the user's movements and surroundings, and contains all the user's contacts, photos and communications. Machine learning can make your apps smarter.

There are four main data input types you can use for **machine learning on mobile**: cameras, text, speech and activity data.

Cameras: Analyze or augment photos and videos captured by the cameras, or use the live camera feed, to detect objects, faces and landmarks in photos and videos; recognize handwriting and printed text within images; search using pictures; track motion and poses; recognize gestures; understand emotional cues in photos and videos; enhance images and remove imperfections; automatically tag and categorize visual content; add special effects and filters; detect explicit content; create 3D models of interior spaces; and implement augmented reality.

Text: Classify or analyze text written or received by the user in order to understand the meaning or sentence structure; translate into other languages; implement intelligent spelling correction; summarize the text; detect topics and sentiment; and create conversational UI and chatbots.

Speech: Convert speech into text, for dictation, translation or Siri-type instructions; and implement automatic subtitling of videos.

Activity: Classify the user's activity, as sensed by the device's gyroscope, accelerometer, magnetometer, altimeter and GPS.

Later in this chapter, in the section **Frameworks, Tools and APIs**, you'll see that the iOS SDK has all of these areas covered!

> **Note**: In general, machine learning can be a good solution when writing out rules to solve a programming problem becomes too complex. Every time you're using a heuristic — an informed guess or rule of thumb — in your software, you might want to consider replacing it with a learned model to get results that are tailored to your end user.

ML in a nutshell

One of the central concepts in machine learning is that of a **model**. The model is the algorithm that was learned by the computer to perform a certain task, plus the data needed to run that algorithm. So a model is a combination of algorithm and data.

It's called a "model" because it models the domain for the problem you're trying to solve. For example, if the problem is recognizing the faces of your friends in photos, then the problem domain is digital photos of humans, and the model will contain everything that it needs to make sense of these photos.

To create the model, you first need to choose an algorithm – for example, a neural network — and then you need to **train the model** by showing it a lot of examples of the problem that you want it to solve. For the face-recognition model, the training examples would be photos of your friends, as well as the things you want the model to learn from these photos, such as their names.

After successful training, the model contains the "knowledge" about the problem that the machine-learning algorithm managed to extract from the training examples.

Once you have a trained model, you can ask it questions for which you don't yet know the answer. This is called **inference**, using the trained model to make predictions or draw conclusions. Given a new photo that the model has never seen before, you want it to detect your friends' faces and put the right name to the right face.

If a model can make correct predictions on data that it was not trained on, we say that it **generalizes** well. Training models so that they make good predictions on new data is the key challenge of machine learning.

The "learning" in machine learning really applies only to the training phase. Once you've trained the model, it will no longer learn anything new. So when you use a machine-learning model in an app, you're not implementing learning so much as "using a fixed model that has previously learned something." Of course, it's possible to re-train your model every so often — for example, after you've gathered new data — and update your app to use this new model.

Supervised learning

The most common type of machine learning practiced today, and the main topic of this book, is **supervised learning**, in which the learning process is guided by a human — you! – that tells the computer what it should learn and how.

With supervised learning, you train the model by giving it **examples** to look at, such as photos of your friends, but you also tell it what those examples represent so that the model can learn to tell the difference between them. These **labels** tell the model what (or who) is actually in those photos. Supervised training always needs labeled data.

> **Note**: Sometimes people say "samples" instead of examples. It's the same thing.

The two sub-areas of supervised learning are **classification** and **regression**.

Regression techniques predict *continuous* responses, such as changes in temperature, power demand or stock market prices. The output of a regression model is one or more real-value numbers. To detect the existence and location of a face in a photo, you'd use a regression model that outputs four numbers that describe the rectangle in the image that contains the face.

Classification techniques predict *discrete* responses or categories, such as whether an email is spam or whether this is a photo of a good dog:

The output of a classification model is a "class" such as "good dog" or "bad dog," or "spam" versus "no spam," or the name of one of your friends. Typical applications are classifying things or places in images or classifying text as expressing positive or negative sentiment.

There is also a type of machine learning called **unsupervised learning**, which does not involve humans in the learning process. A typical example is *clustering*, in which the algorithm is given a lot of **unlabeled** data, and its job is to find patterns in this data. As humans, we typically don't know beforehand what sort of patterns exist, so there is no way we can guide the ML system. Applications include finding similar images, gene sequence analysis and market research.

A third type is **reinforcement learning**, where an agent learns how to act in a certain environment and is rewarded for good behavior but punished for bad. This type of learning is used for tasks like programming robots.

You need data... a lot of it

Let's take a closer look at exactly how a model is trained, as this is where most of the mystery and confusion comes from.

First, you need to collect **training data**, which consists of examples and labels. To make a model that can recognize your friends, you need to show it many examples — photos of your friends — so that it can learn what human faces look like, as opposed to any other objects that can appear in photos and, in particular, which faces correspond to which names.

The labels are what you want the model to learn from the examples — in this case, what parts of the photo contains faces, if any, and the names that go along with them.

The more examples, the better, so that the model can learn which details matter and which details don't. One downside of supervised learning is that it can be very time consuming and, therefore, expensive to create the labels for your examples. If you have 1,000 photos, you'll also need to create 1,000 labels — or more if a photo can have more than one person in it.

> **Note**: You can think of the examples as the questions that the model gets asked; you can think of the labels as the answers to these questions. You only use these labels during training, not for inference. After all, inference means asking questions that you don't yet have the answers for.

It's all about the features

The training examples are made up of the **features** you want to train on. This is a bit of a nebulous concept, but a "feature" is generally a piece of data that is considered to be interesting to your machine-learning model.

For many kinds of machine-learning tasks, you can organize your training data into a set of features that are quite apparent. For a model that predicts house prices, the features could include the number of rooms, floor area, street name and so on. The labels would be the sale price for each house in the dataset. This kind of training data is often provided in the form of a CSV or JSON table, and the features are the columns in that table.

Feature engineering is the art of deciding which features are important and useful for solving your problem, and it is an important part of the daily work of a machine-learning practitioner or data scientist.

In the case of machine-learning models that work on images, such as the friend face detector, the inputs to the model are the pixel values from a given photo. It's not very useful to consider these pixels to be the "features" of your data because RGB values of individual pixels don't really tell you much.

Instead, you want to have features such as eye color, skin color, hair style, shape of the chin, shape of the ears, does this person wear glasses, do they have an evil scar and so on... You could collect all this information about your friends and put it into a table, and train a machine-learning model to make a prediction for "person with blue eyes, brown skin, pointy ears." The problem is that such a model would be useless if the input is a photo. The computer has no way to know what someone's eye color or hair style is because all it sees is an array of RGB values.

So you must **extract** these features from the image somehow; you can use machine learning for that, too! A neural network can analyze the pixel data and discover for itself what the useful features are for getting the correct answers. It learns this during the training process from the training images and labels you've provided. It then uses those features to make the final predictions.

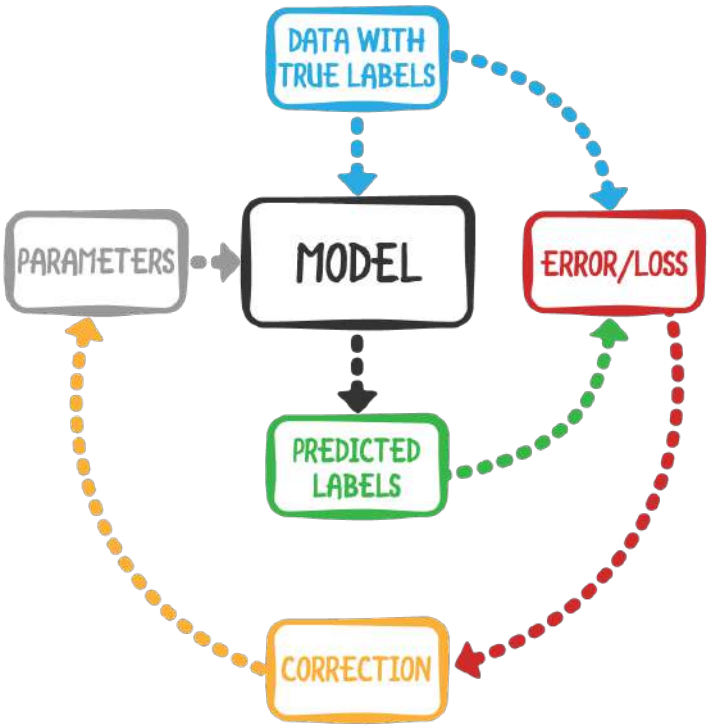
From your training photos, the model might have discovered "obvious" features such as eye color and hair style, but usually the features the model detects are more subtle and hard to interpret. Typical features used by image classifiers include edges, abstract shapes, color blobs and the relationships between them. In practice, it doesn't really matter what features the model has chosen, as long as they let the model make good predictions.

One of the reasons deep learning is so popular is that teaching a model to find the interesting image features by itself works much better than any `if–then–else` rules humans have come up with by hand in the past. Even so, deep learning still benefits from any hints you can provide about the structure of the training data you're using, so that it doesn't have to figure out *everything* by itself.

You'll see the term *features* a lot in this book. For some problems, the features are data points that you directly provide as training data; for other problems, they are data that the model has extracted by itself from more abstract inputs such as RGB pixel values.

The training loop

The training process for supervised learning goes like this:



The model is a particular algorithm you have chosen, such as a neural network. You supply your training data that consists of the examples, as well as the correct labels for these examples. The model then makes a prediction for each of the training examples.

Initially, these predictions will be completely wrong because the model has not learned anything yet. But *you* know what the correct answers should be, and so it is possible to calculate how "wrong" the model is by comparing the predicted outputs to the expected outputs (the labels). This measure of "wrongness" is called the **loss** or the error.

Using some fancy mathematics called *back-propagation*, the training process uses this loss value to slightly tweak the **parameters** of the model so that it will make better predictions next time.

Showing the model all the training examples just once is not enough. You'll need to repeat this process over and over, often for hundreds of iterations. In each iteration, the

loss will become lower, meaning that the error between the prediction and the true value has become smaller and, thus, the model is less wrong than before. And that's a good thing!

If you repeat this enough times, and assuming that the chosen model has enough capacity for learning this task, then gradually the model's predictions will become better and better.

Usually people keep training until the model reaches either some minimum acceptable accuracy, up to a maximum number of iterations, or until they run out of patience... For deep neural networks, it's not uncommon to use millions of images and to go through hundreds of iterations.

Of course, training is a bit more complicated than this in practice (isn't it always?). For example, it's possible to train for *too long*, actually making your model worse. But you get the general idea: show the training examples, make predictions, update the model's parameters based on how wrong the predictions are, repeat until the model is good enough.

As you can tell, training machine-learning models is a brute-force and time-consuming process. The algorithm has to figure out how to solve the problem through trial and error. It's no surprise that it takes a lot of processing power. Depending on the complexity of the algorithm you've chosen and the amount of training data, training a model can take anywhere from several minutes to several weeks, even on a computer with very fast processors. If you want to do some serious training, you can rent time on an Amazon, Google or Microsoft server, or cluster of servers, which does the job much faster than your laptop or desktop computer.

What does the model actually learn?

Exactly what a model learns depends on the algorithm you've chosen. A *decision tree*, for example, literally learns the same kind of `if-then-else` rules a human would have created. But most other machine-learning algorithms don't learn rules directly, but a set of numbers called the **learned parameters**, or just parameters, of the model.

These numbers represent what the algorithm has learned, but they don't always make sense to us humans. We can't simply interpret them as `if-then-else`; the math is more complex than that. It's not always obvious what's going on inside these models, even if they produce perfectly acceptable outcomes. A big neural network can easily have 50 million of these parameters, so try wrapping your head around that!

It's important to realize that we aren't trying to get the model to *memorize* the training examples. That's not what the parameters are for. During the training process, the

model parameters should capture some kind of *meaning* from the training examples, not retain the training data verbatim. This is done by choosing good examples, good labels and a loss function that is suitable to the problem.

Still, one of the major challenges of machine learning is **overfitting**, which happens when the model does start to remember specific training examples. Overfitting is hard to avoid, especially with models that have millions of parameters.

For the friends detector, the model's learned parameters somehow encode what human faces look like and how to find them in photos, as well as which face belongs to which person. But the model should be dissuaded from remembering specific chunks of pixel values from the training images, as that would lead to overfitting.

How does the model know what a face is? In the case of a neural network, the model acts as a **feature detector** and it will literally learn how to tell objects of interest (faces) apart from things that are not of interest (anything else). You'll look at how neural networks try to make sense of images in the next chapters.

Transfer learning: Just add data

> **Note**: *Just* add data?! Data is *everything* in machine learning! You must train the model with data that accurately represents the sort of predictions you want to make. In Chapter 4, you'll see how much work was needed to create a relatively small dataset of less than 5,000 images.

The amount of work it takes to create a good machine-learning model depends on your data and the kind of answers you want from the model. An existing free model might do everything you want, in which case you just convert it to Core ML and pop it into your iOS app. Problem solved!

But what if the existing model's output is different from the output *you* care about? For example, in the next chapter, you'll use a model that classifies pictures of snack food as healthy or unhealthy. There was no free-to-use model available on the web that did this — we looked! So we had to make our own.

This is where **transfer learning** can help. In fact, no matter what sort of problem you're trying to solve with machine learning, transfer learning is the best way to go about it 99% of the time. With transfer learning, you can reap the benefits of the hard work that other people have already done. It's the smart thing to do!

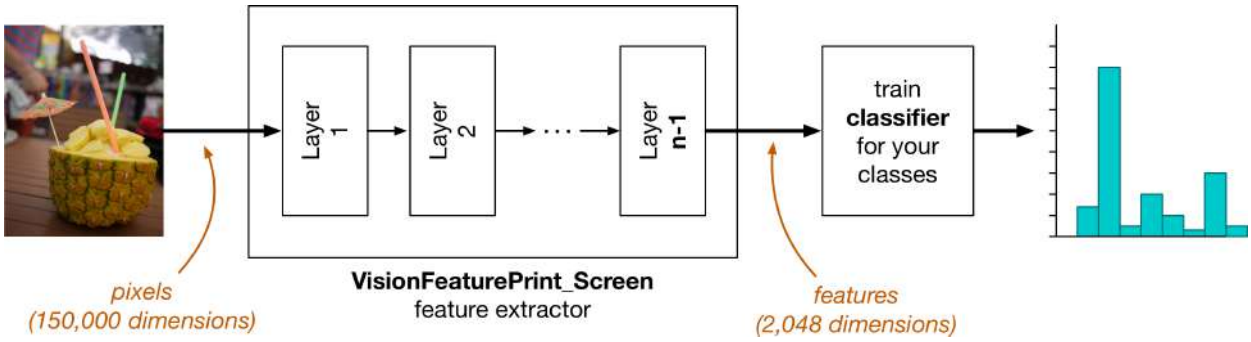
When a deep-learning model is trained, it learns to identify features in the training images that are useful for classifying these images. Core ML comes with a number of

ready-to-use models that detect thousands of features and understand 1,000 different classes of objects. Training one of these large models from scratch requires a very large dataset, a huge amount of computation and can cost big bucks.

Most of the training time is spent on learning how to detect the best features. Many of these features — edges, corners, shapes and relationships between shapes — are probably also useful for classifying *your data* into the classes you care about, especially if your training examples are similar in nature to the type of data this other model has already been trained on. Most of these big, freely available models are trained on photographs of humans, animals and everyday objects.

So it would be a bit of a waste if you had to train your own model from scratch to learn the exact same thing. Instead, to create a model for your own dataset, you can take an existing pre-trained model and customize it for your data. This is called transfer learning.

You use this existing pre-trained model to extract features from your own training data, and then you only train the final classification layer of the model so that it learns to make predictions from the extracted features — but this time for your own class labels.



Transfer learning has the huge advantage that it is much faster than training the whole model from scratch, plus your dataset can be much smaller. Instead of millions of images, you now only need a few thousand or even a few hundred.

Apple provides two tools that do transfer learning: Create ML and Turi Create. But this is such a useful technique that you can find transfer learning tools for the most popular machine-learning tasks, like image classification or sentiment analysis. Sometimes it's as easy as dragging your data onto the model; at most, you write just a few lines of code to read in and structure your data.

Can mobile devices really do machine learning?

A trained model might be hundreds of MB in size, and inference typically performs billions of computations, which is why inference often happens on a server with fast processors and lots of memory. For example, Siri needs an internet connection to process your voice commands — your speech is sent to an Apple server that analyzes its meaning, then sends back a relevant response.

This book is about doing state-of-the-art machine learning on mobile, so we'd like to do as much **on the device** as possible and avoid having to use a server. The good news: Doing inference on iOS devices works very well thanks to core technologies like Metal and Accelerate.

The benefits of on-device inference:

1. Faster response times: It's more responsive than sending HTTP requests, so doing real-time inference is possible, making for a better user experience.
2. It's also good for user privacy: The user's data isn't sent off to a backend server for processing, and it stays on the device.
3. It's cheaper since the developer doesn't need to pay for servers and electricity: But the user pays for it using battery power. Of course, you don't want to abuse this priviledge, which is why it's important to make sure your models run as efficiently as possible. We'll explain, in this book, how to optimize machine-learning models for mobile.

What about on-device training? That's the bad news: Mobile devices still have some important limitations. Training a machine-learning model takes a lot of processing power and, except for small models, simply is out of reach of most mobile devices at this point. That said, updating a previously trained model with new data from the user, such as a predictive keyboard that learns as you type (also known as "online training"), is certainly possible today provided that the model is small enough.

> **Note**: Core ML currently does not allow training on the device; it is for inference only. The Metal framework does have APIs for on-device training but what these can do right now is limited. The current edition of this book focuses on making predictions using a model that was trained offline, and it explains how to train those models on your Mac or a cloud service.

Why not in the cloud?

Companies such as Amazon, Google and Microsoft provide cloud-based services for doing machine learning, and there are a whole lot of smaller players as well. Some of these just provide raw computing power (you rent servers from them). Others provide complete APIs wherein you don't have to concern yourself with the details of machine learning at all — you just send your data to their API and it returns the results a few seconds later.

There are a lot of benefits to using these cloud services: 1) You don't need to know anything about machine learning — so you won't need to read the rest of this book; 2) Using them is as easy as sending an HTTP request; and 3) Other people will take care of running and maintaining the servers.

However, there are also downsides: 1) You're using a canned service that is often not tailored to your own data; 2) If your app needs to do machine learning in real-time (such as on a video), then sending HTTP requests is going to be way too slow; and 3) Convenience has a price: You have to pay for using these services and, once you've chosen one, it's hard to migrate your app away from them.

In this book, we don't focus on using cloud services. They can be a great solution for many apps — especially when you don't need real-time predictions — but as mobile developers we feel that doing machine learning on the device, also known as "edge" computing, is just more exciting.

Frameworks, tools and APIs

It may seem like using machine learning is going to be a pain in the backside. Well, not necessarily... Like every technology you work with, machine learning has levels of abstraction — the amount of difficulty and effort depends on which level you need to work at.

Apple's task-specific frameworks

The highest level of abstraction is the task level. This is the easiest to grasp — it maps directly to what you want to do in your app. Apple and the other major players provide *task-focused toolkits* for tasks like image or text classification.

As Apple's web page says: *You don't have to be a machine learning expert*!

Apple provides several Swift frameworks for doing specific machine-learning tasks:

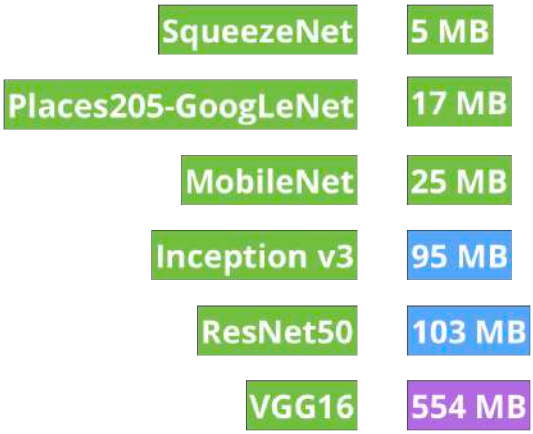
- **Vision**: Detect faces and face landmarks, rectangles, bar codes and text, and to track objects. Vision also makes it easier to use Core ML image models, taking care of formatting the input images correctly.
- **Natural Language**: Analyze text to identify the language, part of speech, as well as specific people, places or organizations.
- **Speech**: Convert speech to text, and optionally retrieve answers using Apple's servers or an on-device speech recognizer. Apple sets limits on audio duration and daily number of requests.
- **SiriKit**: Handle requests from Siri or Maps for your app's services by implementing an *Intents* app extension in one of a growing number of Intent Domains: messaging, lists and notes, workouts, payments, photos and more.
- **GameplayKit**: Evaluate decision trees that contain questions, possible answers and consequent actions.

If you need to do one of the tasks from the above list, you're in luck. Just call the appropriate APIs and let the framework do all the hard work.

Core ML ready-to-use models

Core ML is Apple's go-to choice for doing machine learning on mobile. It's very easy to use and integrates well with the Vision framework. Core ML does have its limitations, but it's a great place to start.

Core ML is not just a framework but also an open file format for sharing machine-learning models. Apple provides six **ready-to-use** pre-trained image classification models in Core ML format:



You can also find other pre-trained Core ML models on the Internet in so-called "model zoos." Here's one we like: github.com/likedan/Awesome-CoreML-Models.

When you're adding a model to your iOS app, size matters. Larger models use more battery power and are slower than smaller models. The size of the Core ML file is proportional to the number of learned parameters in the model. For example, ResNet50 has 25.6M parameters, and VGG16 has 138M parameters. By the way, just because a model has more parameters doesn't necessarily mean it is more accurate. VGG16 is 22 times larger than MobileNet but both models have similar accuracy.

These models, like other free models you can find on the web, are trained on very generic datasets. If you want to make an app that can detect a faulty thingamajig in the gizmos you manufacture, you'll have to get lucky and stumble upon a model that someone else has trained and made available for free — fat chance! There's big money in building good models, so companies aren't just going to give them away, so you'll have to know how to make your own model.

> **Note**: The next chapter shows you how to use a pre-trained Core ML model in an iOS app.

Convert existing models with coremltools

If there's an existing model that does exactly what you want but it's not in Core ML format, then don't panic! There's a good chance you'll be able to convert it.

There are many popular open-source packages for training machine-learning models. To name a few:

- Apache MXNet
- Caffe (and the relatively unrelated Caffe2)
- Keras
- PyTorch
- scikit-learn
- TensorFlow

If you have a model that is trained with one of these packages — or others such as XGBoost, LIBSVM, IBM Watson — then you can convert that model to Core ML format using **coremltools** so that you can run the model from your iOS app.

`coremltools` is a Python package, and you'll need to write a small Python script in order to use it. Python is the dominent programming language for machine-learning projects, and most of the training packages are also written in Python.

Some popular model formats, such as TensorFlow and MXNet, are not directly supported by coremltools but have their own Core ML converters. IBM's Watson gives you a Swift project but wraps the Core ML model in its own API.

Because there are now so many different training tools that all have their own, incompatible formats, there are several industry efforts to create a standard format. Apple's Core ML is one of those efforts, but the rest of the industry seems to have chosen ONNX instead. Naturally, there is a converter from ONNX to Core ML.

> **Note**: It's important to note that Core ML has many limitations, it is not as capable as some of the other machine-learning packages. There is no guarantee that a model that is trained with any of these tools can actually be converted to Core ML, because certain operations or neural network layer types may not be supported by Core ML. In that case, you may need to tweak the original model before you can convert it.

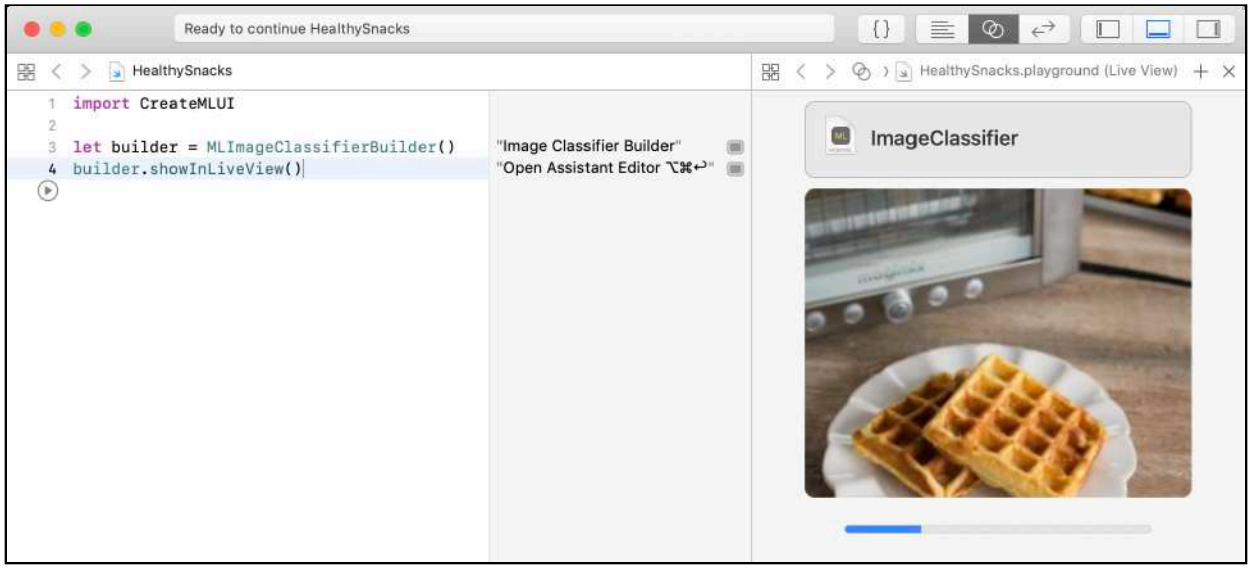
Transfer learning with Create ML and Turi Create

Create ML and **Turi Create** use transfer learning to let you customize pre-trained base models with *your own data*. The base models have been trained on very large datasets, but transfer learning can produce an accurate model for your data with a much smaller dataset and much less training time.

You only need dozens or hundreds of images instead of tens of thousands, and training takes minutes instead of hours or days. That means you can train these kinds of models straight from the comfort of your own Mac. In fact, both Create ML and Turi Create (as of version 5) can use the Mac's GPU to speed up the training process.

Apple acquired the startup Turi in August 2016, then released Turi Create as open-source software in December 2017. Turi Create was Apple's first transfer-learning tool, but it requires you to work in the alien environment of Python. At WWDC 2018, Apple announced Create ML, which is essentially, a Swift-based subset of Turi Create, to give you transfer learning for image and text classification using Swift and Xcode.

Create ML has some special features like the `MLImageClassifierBuilder` GUI that runs straight from a Swift playground:



Turi Create and Create ML are task-specific, rather than model-specific. This means that you specify the *type of problem* you want to solve, rather than choosing the *type of model* you want to use. You select the task that matches the type of problem you want to solve, then Turi Create analyzes your data and chooses the right model for the job.

Turi Create has seven task-focused toolkits:

- Image classification: Label images with labels that are meaningful for your app.
- Image similarity: Find images that are similar to a specific image; an example is the Biometric Mirror project described at the end of this chapter.
- Recommender system: Provide personalized recommendations for movies, books, holidays etc., based on a user's past interactions with your app.
- Object detection: Locate and classify objects in an image.
- Style transfer: Apply the stylistic elements of a style image to a new image.
- Activity classification: Use data from a device's motion sensors to classify what the user is doing, such as walking, running, waving, etc.
- Text classifier: Analyze the sentiment — positive or negative — of text in social media, movie reviews, call center transcripts, etc.

Create ML currently has only two of these toolkits: image classification and text classification, However, the expectation is that Apple might add more toolkits in future updates.

> **Note**: Chapter 3 will show you how to customize Create ML's image classification model with Swift in Xcode. Chapter 4 will get you started with the Python-based machine-learning universe, and it will teach you how to create the same custom model using Turi Create. Don't worry, Python is very similar to Swift, and we'll explain everything as we go along.

Turi Create's statistical models

So far, we've described task-specific solutions. Let's now look one level of abstraction deeper at the model level. Instead of choosing a task and then letting the API select the best model, here you choose the model yourself. This gives you more control; on the flip side, it's also more work.

Turi Create includes these general-purpose models:

- Classification: Boosted trees classifier, decision tree classifier, logistic regression, nearest neighbor classifier, random forest classifier, and SVM (Support Vector Machines.)
- Clustering: K-Means, DBSCAN (density based).
- Graph analytics: Pagerank, shortest path, graph coloring and more.
- Nearest neighbors.
- Regression: Boosted trees regression, decision tree regression, linear regression and random forest regression.
- Text analysis.

You probably won't need to learn about these; when you use a task-focused toolkit, Turi Create picks suitable statistical models based on its analysis of your data. They're listed here so that you know that you can also use them directly if, for example, Turi Create's choices don't produce a good enough model.

Build your own model in Keras

Apple's frameworks and task-focused toolkits cover most things you'd want to put in your apps but, if you can't create an accurate model with Create ML or Turi Create, you have to **build your own model** from scratch.

This requires you to learn a few new things: the different types of neural network *layers* and *activation functions*, as well as *batch sizes*, *learning rates* and other *hyperparameters*. Don't worry! In Chapter 5, you'll learn about all these new terms and how to use **Keras** to configure and train your own deep-learning networks.

Keras is a wrapper around Google's **TensorFlow**, which is the most popular deep-learning tool because... well, Google. TensorFlow has a rather steep learning curve and is primarily a low-level toolkit, requiring you to understand things like matrix math, so this book doesn't use it directly. Keras is much easier to use. You'll work at a higher level of abstraction, and you don't need to know any of the math. (Phew!)

> **Note**: You might have heard of *Swift for TensorFlow*. This is a Google Brain project, led by Chris Lattner, to provide TensorFlow users with a better programming language than Python. It will make life easier for TensorFlow users, but it won't make TensorFlow any easier to learn for us Swifties.

Gettin' jiggy with the algorithms

If you've looked at online courses or textbooks on machine learning, you've probably seen a lot of complicated math about efficient algorithms for things like gradient descent, back-propagation and optimizers. As an iOS app developer, you don't need to learn any of that (unless you like that kind of stuff, of course).

As long as you know what high-level tools and frameworks exist and how to use them effectively, you're good to go. When researchers develop a better algorithm, it quickly finds its way into the pre-trained models and tools such as Keras, without you needing to do anything special. To be a user of machine learning, you usually won't have to implement any learning algorithms from scratch yourself.

However, Core ML can be a little slow to catch up with the rest of the industry (it is only updated with new iOS releases) and so developers who want to live on the leading edge of machine learning may still find themselves implementing new algorithms, because waiting for Apple to update Core ML is not always an option. Fortunately, Core ML allows you to customize models, so there is flexibility for those who need it.

It's worth mentioning a few more machine-learning frameworks that are available on iOS. These are more low-level than Core ML, and you'd only use them if Core ML is not good enough for your app.

- **Metal Performance Shaders**: Metal is the official framework and language for programming the GPU on iOS devices. It's fairly low-level and not for the faint of heart, but it does give you ultimate control and flexibility. For the best performance, Metal is the right choice but it takes more effort to get going than Core ML.
- **Accelerate**: All iOS devices come with this underappreciated framework. It lets you write heavily optimized CPU code. Where Metal lets you get the most out of the GPU, Accelerate does the same for the CPU. There is a neural-networking library, BNNS (Basic Neural Networking Subroutines), but it also has optimized code for doing linear algebra and other mathematics. If you're implementing your own machine-learning algorithms from scratch, you'll probably end up using Accelerate.

Third-party frameworks

Besides Apple's own APIs there are also a number of iOS machine learning frameworks from other companies. The most useful are:

- **Google ML Kit**: This is Google's answer to the Vision framework. With ML Kit you can easily add image labeling, text recognition, face detection, and other tasks to your apps. ML Kit can run in the cloud but also on the device, and supports both iOS and Android.
- **TensorFlow-Lite**: TensorFlow, Google's popular machine-learning framework, also has a version for mobile devices, TF-Lite. The main advantage of TF-Lite is that it can directly load TensorFlow models, although on iOS devices it currently is not GPU accelerated. The API is in C++, which makes it hard to use in Swift.

ML all the things?

Machine learning, especially deep learning, has been very successful in problem domains such as image classification and speech recognition, but it can't solve *everything*. It works great for some problems but it's totally useless for others.

A deep-learning model doesn't actually *reason* about what it sees. It lacks the common sense that you were born with.

It doesn't know or care that an object could be made up of separate parts, that objects don't suddenly appear or disappear, that a round object can roll off a table, and that children don't put baseball bats into their mouths.

This caption was generated by a deep learning model

At best, the current generation of machine-learning models are very good pattern detectors, nothing more. Having very good pattern detectors is awesome, but don't fall for the trap of giving these models more credit than they're due. We're still a long way from true machine intelligence!

A machine-learning model can only learn from the examples you give it, but the examples you *don't* give it are just as important. If the friends detector was only trained on images of humans but not on images of dogs, what would happen if you tried to do inference on an image of a dog? The model would probably "detect" the face of your friend who looks most like a dog (literally!). This happens because the model wasn't given the chance to learn that a dog's face is different from a human face.

Machine-learning models might not always learn what you expect them to. Let's say you've trained a classifier that can tell the difference between pictures of cats and dogs. If all the cat pictures you used for training were taken on a sunny day, and all dog pictures were taken on a cloudy day, you have actually inadvertently trained a model that "predicts" the weather!

Because they lack context, deep-learning models can be easily fooled. Although humans can make sense of some of the features that a deep-learning model extracts — edges or shapes — many of the features just look like abstract patterns of pixels to us, but might have a specific meaning to the model. While it's hard to understand how a model makes its predictions, as it turns out, it's easy to fool the model into making wrong ones.

Using the same training method that produced the model, you can create *adversarial examples* by adding a small amount of noise to an image in a way that tricks the model. The image still looks the same to the human eye, but the model will classify this

adversarial example as something completely different, with very high confidence — a panda labelled as a gibbon, for example. Or worse, a stop sign labelled as a green traffic light. A lot of research is currently being done on these adversarial attacks and how to make models more robust against them.

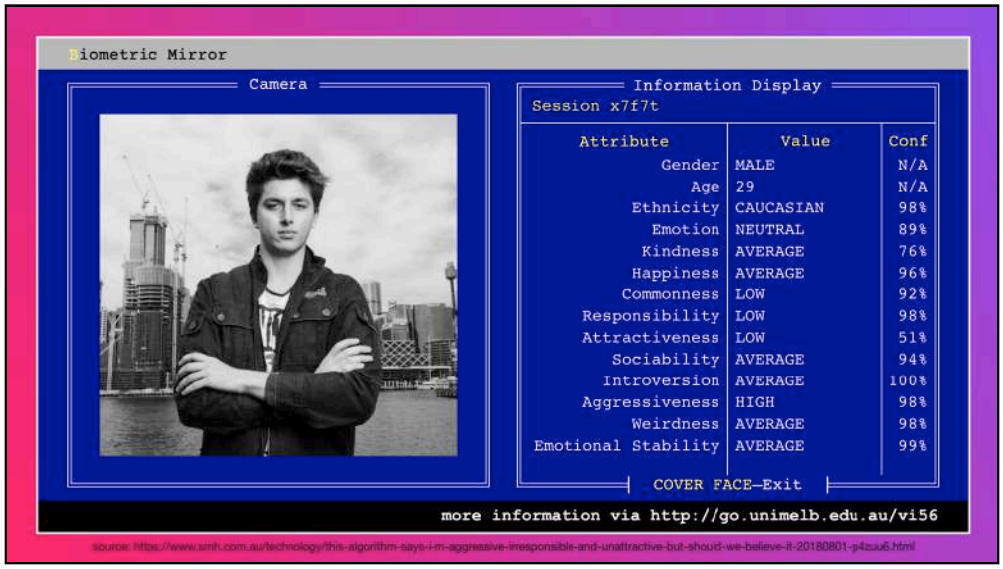
The lesson here is that understanding and dealing with the limitations of machine learning is just as important as building your models in the first place, or you might be in for unpleasant surprises!

The ethics of machine learning

Machine learning is a powerful tool, extending the reach of artificial intelligence into everyday life. Using trained models can be fun, time-saving or profitable, but the misuse of AI can be harmful.

The human brain evolved to favor quick decisions about danger, and it is happy to use shortcuts. Problems can arise when bureaucracies latch onto convenient metrics or rankings to make life-changing decisions about who to hire, fire, admit to university, lend money to, what medical treatment to use, or whether to imprison someone and for how long. And machine-learning model predictions are providing these shortcuts, sometimes based on biased data, and usually with no explanation of how a model made its predictions.

Consider the **Biometric Mirror** project at go.unimelb.edu.au/vi56, which predicts the personality traits that other people might perceive from just looking at your photo. Here are the results for Ben Grubb, a journalist:



The title of his article says it all: *This algorithm says I'm aggressive, irresponsible and unattractive. But should we believe it?* — check it out at bit.ly/2KWRkpF.

The algorithm is a simple image-similarity model that finds the closest matches to your photo from a dataset of 2,222 facial photos. 33,430 crowd-sourced people rated the photos in the dataset for a range of personality traits, including levels of aggression, emotional stability, attractiveness and weirdness. The model uses their evaluations of your closest matches to predict *your* personality traits.

The journalist experimented with different photos of himself, and the model told him he was younger and attractive.

It's amusing, but is it harmful?

The model is part of an interactive application that picks one of your traits — say, level of emotional stability — and asks you to imagine that information in the hands of someone like your insurer, future employer or a law enforcement official. Are you feeling worried now?

In bit.ly/2LMialy, the project's lead researchers write "[Biometric Mirror] starkly demonstrates the possible consequences of AI and algorithmic bias, and it encourages us [to] reflect on a landscape where government and business increasingly rely on AI to inform their decisions."

And, on the project page, they write:

> [O]ur algorithm is correct but the information it returns is not. And that is precisely what we aim to share: We should be careful with relying on artificial intelligence systems, because their internal logic may be incorrect, incomplete or extremely sensitive and discriminatory.

This project raises two of the ethical questions in the use of AI:

- Algorithmic bias
- Explainable or understandable AI

Biased data, biased model

We consider a machine-learning model to be good if it can make correct predictions on data it was not trained on — it *generalizes well* from the training dataset. But problems can arise if the training data was biased for or against some group of people: The data might be racist or sexist.

The reasons for bias could be historical. To train a model that predicts the risk of someone defaulting on a loan, or how well someone will perform at university, you would give it historical data about people who did or didn't default on loans, or who did or didn't do well at university. And, historically, the data would favor white men because, for a long time, they got most of the loans and university admittances.

Because the data contained fewer samples of women or racial minorities, the model might be 90% accurate overall, but only 50% accurate for women or minorities.

Also, the data might be biased by the people who made the decisions in the first place: Loan officials might have been more tolerant of late payments from white men, or university professors might have been biased against certain demographics.

You can try to overcome bias in your model by explicitly adjusting its training data or parameters to counteract biases. Some model architectures can be tweaked to identify sensitive features and reduce or remove their effect on predictions.

Explainable/interpretable/transparent AI

The algorithmic bias problem means it's important to be able to examine how an ML model makes predictions: Which features did it use? Is that accurate or fair?

In the first diagram of this chapter, I drew **training** as a black box. Although you'll learn something about what happens inside that box, many deep learning models are so complex, even their creators can't explain individual predictions.

One approach could be to require more transparency about algorithmic biases and what the model designer did to overcome them.

Google Brain has an open source tool github.com/google/svcca that can be used to interpret what a neural network is learning.

Key points

- Machine learning isn't really that hard to learn - Stick with this book and you'll see!
- Access to large amounts of data and computing power found online has made ML a viable technology.
- At its core, machine learning is all about models; creating them, training them, and inferring results using them.

- Training models can be an inexact science and an exercise in patience, however, transfer learning tools like Create ML and Turi Create, can help improve the experience in specific cases.
- Mobile devices are ideal for inferring results. As for creating models on device, we're not quite there yet.
- Don't confuse Machine Learning with Machine Intelligence. Machine Learning can be a great addition to your app, but knowing their limitations is equally important.

Where to go from here?

That was a lot! We hope you enjoyed that tour of machine-learning from 10,000 feet. If you didn't absorb everything you read, don't worry! As with all new things learned, time and patience are your greatest assets!

In the next chapter, you'll finally start writing some code! This first chapter on machine learning with images explains how to use a pre-trained Core ML model in an iOS app. It's chock full of insights into the inner workings of neural networks, so don't skip it!

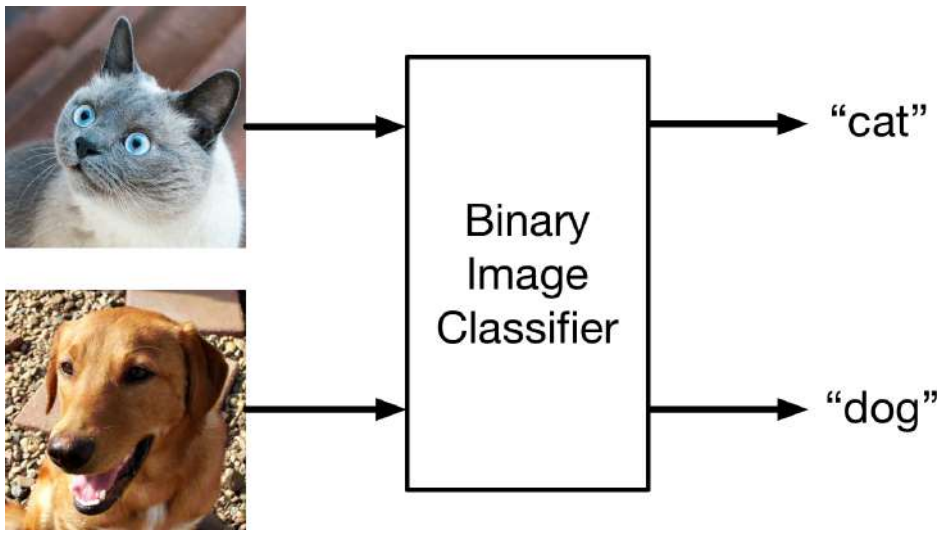
Chapter 2: Getting Started with Image Classification

By Matthijs Hollemans

Let's begin your journey into the world of machine learning by creating a binary image classifier.

A **classifier** is a machine learning model that takes an input of some kind, in this case an image, and determines what sort of "thing" that input represents. An image classifier tells you which category, or class, the image belongs to.

Binary means that the classifier is able to distinguish between two classes of objects. For example, you can have a classifier that will answer either "cat" or "dog" for a given input image, just in case you have trouble telling the two apart.



A binary classifier for cats and dogs

Being able to tell the difference between only two things may not seem very impressive, but binary classification is used a lot in practice.

In medical testing, it determines whether a patient has a disease, where the "positive" class means the disease is present and the "negative" class means it's not. Another common example is filtering email into spam/not spam.

There are plenty of questions that have a definite "yes/no" answer, and the machine learning model to use for such questions is a binary classifier. The cats-vs.-dogs classifier can be framed as answering the question: "Is this a picture of a cat?" If the answer is no, it's a dog.

Image classification is one of the most fundamental computer vision tasks. Advanced applications of computer vision — such as object detection, style transfer, and image generation — all build on the same ideas from image classification, making this a great place to start.

There are many ways to create an image classifier, but by far the best results come from using deep learning. The success of deep learning in image classification is what started the current hype around AI and ML. We wouldn't want you to miss out on all this exciting stuff, and so the classifier you'll be building in this chapter uses deep learning under the hood.

Is that snack healthy?

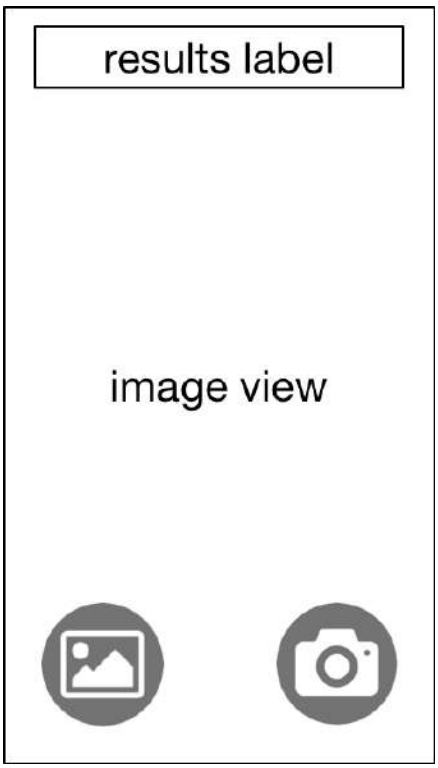
In this chapter you'll learn how to build an image classifier that can tell the difference between healthy and unhealthy snacks.

healthy

unhealthy

To get started, make sure you've downloaded the supplementary materials for this chapter and open the **HealthySnacks** starter project in Xcode.

This is a very basic iPhone app with two buttons, an image view, and a text label at the top:



The design of the app

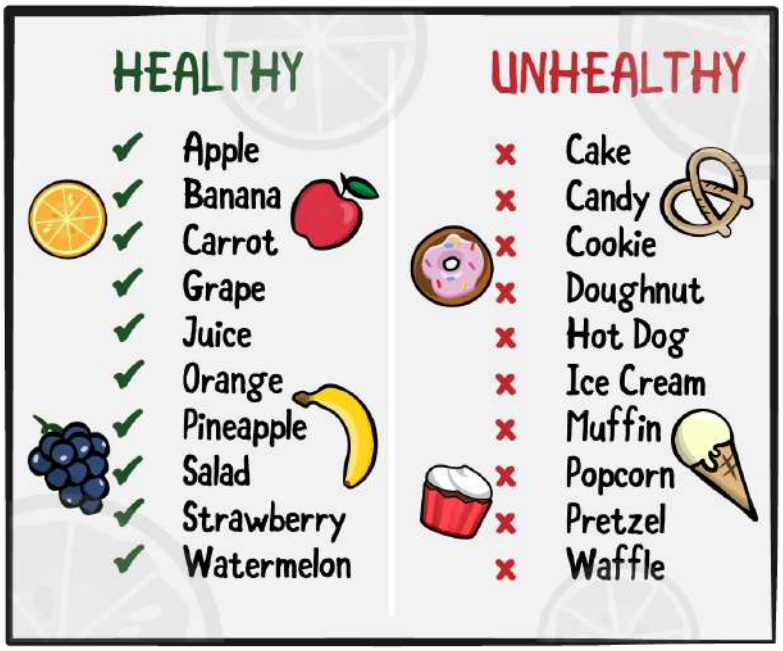
The "picture frame" button on the left lets you choose a photo from the library using `UIImagePickerController`. The "camera" button on the right lets you take a picture with the camera (this button is disabled in the simulator).

Once you've selected a picture, the app calls `classify(image:)` in **ViewController.swift** to decide whether the image is of a healthy snack or not. Currently this method is empty. In this chapter you'll be adding code to this method to run the classifier.

At this point, it's a good idea to take a brief look at **ViewController.swift** to familiarize yourself with the code. It's pretty standard fare for an iOS app.

In order to do machine learning on the device, you need to have a trained model. For the HealthySnacks app, you'll need a model that has learned how to tell apart healthy snacks from unhealthy snacks. In this chapter you'll be using a ready-made model that has already been trained for you, and in the next chapter you'll learn to how train this model yourself.

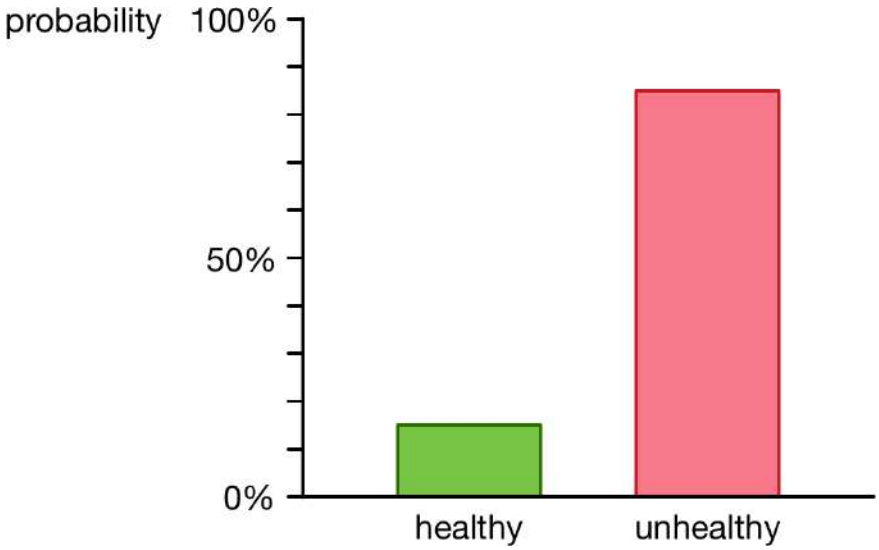
The model is trained to recognize the following snacks:



The categories of snacks

For example, if you point the camera at an apple and snap a picture, the app should say “healthy.” If you point the camera at a hotdog, it should say "unhealthy."

What the model actually predicts is not just a label (“healthy” or “unhealthy”) but **probability distribution**, where each classification is given a probability value:



An example probability distribution

If your math and statistics are a little rusty, then don't let terms such as "probability distribution" scare you. A probability distribution is simply a list of positive numbers that add up to 1.0. In this case it is a list of two numbers because this model has two classes:

```
[0.15, 0.85]
```

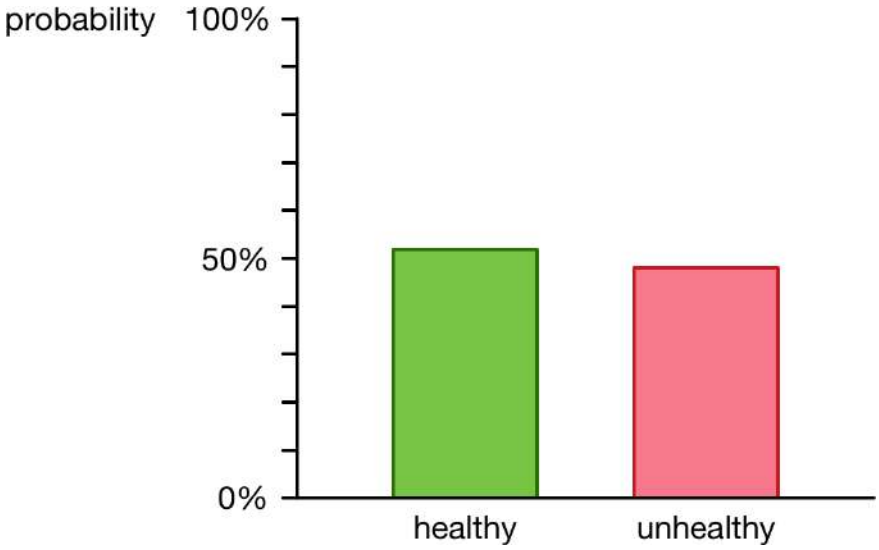
The above prediction is for an image of a waffle with strawberries on top. The model is 85% sure that the object in this picture is unhealthy. Because the predicted probabilities always need to add up to 100% (or `1.0`), this outcome also means the classifier is 15% sure this snack is healthy — thanks to the strawberries.

You can interpret these probabilities to be the **confidence** that the model has in its predictions. A waffle without strawberries would likely score higher for unhealthy, perhaps as much as 98%, leaving only 2% for class healthy. The more confident the model is about its prediction, the more one of the probabilities goes to 100% and the other goes to 0%. When the difference between them is large, as in this example, it means that the model is sure about its prediction. Ideally, you would have a model that is always confident and never wrong. However, sometimes it's very hard for the model to draw a solid conclusion about the image. Can *you* tell whether the food in the following image is mostly healthy or unhealthy?

What is this?

The less confident the model is, the more both probabilities go towards the middle, or 50%.

When the probability distribution looks like the following, the model just isn't very sure, and you cannot really trust the prediction — it could be either class.



An unconfident prediction

This happens when the image has elements of both classes — salad and greasy stuff — so it's hard for the model to choose between the two classes. It also happens when the image is not about food at all, and the model does not know what to make of it.

To recap, the input to the image classifier is an image and the output is a probability distribution, a list of numbers between 0 and 1.

Since you're going to be building a binary classifier, the probability distribution is made up of just two numbers. The easiest way to decide which class is the winner is to choose the one with the highest predicted probability.

> **Note**: To keep things manageable for this book, we only trained the model on twenty types of snacks (ten healthy, ten unhealthy). If you take a picture of something that isn't in the list of twenty snacks, such as broccoli or pizza, the prediction could be either healthy or unhealthy. The model wasn't trained to recognize such things and, so, what it predicts is anyone's guess. That said, the model might still guess right on broccoli (it's green, which is similar to other healthy snacks) and pizza (it's greasy and therefore unhealthy).

Core ML

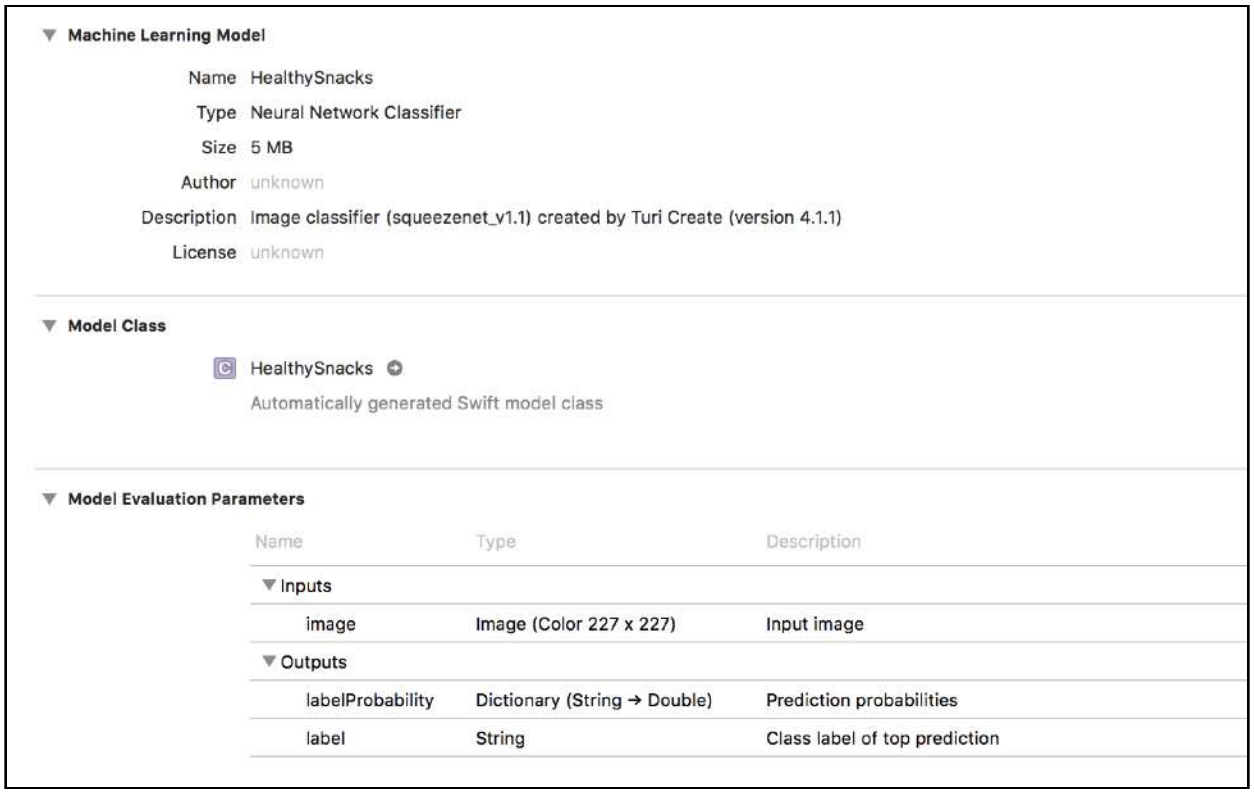
For many of the projects in this book, you'll be using Core ML, Apple's machine learning framework that was introduced with iOS 11. Core ML makes it really easy to add machine learning models to your app — it's mostly a matter of dropping a trained model into your app and calling a few API functions. Xcode even automatically writes most of the code for you.

Of course, Core ML is only easy if you already have a trained model. You can find the model for this chapter, **HealthySnacks.mlmodel**, in the downloaded resources.

Core ML models are packaged up in a **.mlmodel** file. This file contains both the structural definition of the model as well as the things it has learned, known as the learned parameters (or the "weights").

With the HealthySnacks project open in Xcode, drag the **HealthySnacks.mlmodel** file into the project to add it to the app (or use File ▸ Add Files).

Select **HealthySnacks.mlmodel** in the Project Navigator and Xcode will show the following:



Looking at the mlmodel file

This is a summary of the Core ML model file. It shows what of type model it is, the size of the model in megabytes and a description.

The HealthySnacks model type is Neural Network Classifier, which means it is an image classifier that uses deep learning techniques. The terms “deep learning” and “neural network” mean pretty much the same thing. According to the description, this model was made using a tool called Turi Create and it uses SqueezeNet v1.1, a popular deep learning architecture for mobile apps.

The main benefit of SqueezeNet is that it’s small. As you can see in Xcode, the size of this model is "only" 5 MB. That is tiny compared to many other deep learning model architectures, which can take up hundreds of MBs. Such large models are usually not a good choice for use in a mobile app. Not only do they make the app download bigger but larger models are also slower and use more battery power.

The Model Evaluation Parameters section lists the inputs that the model expects and the outputs that it produces. Since this is an image classifier there is only one input, a color image that must be 227 pixels wide and 227 pixels tall. You cannot use images with other dimensions. The reason for this restriction is that the SqueezeNet architecture expects an image of exactly this size. If it’s any smaller or any larger, the math used by SqueezeNet doesn’t work out. This means that any image you pick from the photo library or take with the camera must be resized to 227×227 before you can use it with this Core ML model.

> **Note**: If you’re thinking that 227×227 pixels isn’t very big, then you’re right. A typical 12-megapixel photo is 4032×3024 — that is more than 200 times as many pixels! But there is a trade-off between image size and processing time. These deep learning models need to do *a lot* of calculations: For a single 227×227 image, SqueezeNet performs 390 million calculations. Make the image twice as large and the number of calculations also doubles. At some point, that just gets out of hand and the model will be too slow to be useable!
>
> Making the image smaller will make the model faster, and it can even help the models learn better since scaling down the image helps to remove unnecessary details that would otherwise just confuse the model. But there’s a limit here too: At some point, the image loses too much detail, and the model won’t be able to do a good job anymore. For image classification, 227×227 is a good compromise. Other typical image sizes used with deep learning models are 224×224 and 299×299.

The HealthySnacks model has two outputs. It puts the probability distribution into a dictionary named `labelProbability` that will look something like this:

```
labelProbability = [ “healthy”: 0.15, “unhealthy”: 0.85 ]
```

For convenience, the second output it provides is the class label of the top prediction: `"healthy"` if the probability of the snack being healthy is greater than 50%, `"unhealthy"` if it’s less than 50%.

The final section of this model summary to look at is Model Class. When you add an .mlmodel file to a project, Xcode does something smart behind the scenes: It creates a Swift class with all the source code needed to use the model in your app. That means you don’t have to write any code to load the .mlmodel — Xcode has already done the heavy lifting for you.

To see the code that Xcode generated, click the little arrow next to the model name:

Click the arrow to view the generated code

It’s not important, at this point, that you understand exactly what this code does; just notice that the automatically generated Swift file contains a class `HealthySnacks` that has an `MLModel` object property (the main object from the Core ML framework). It also has `prediction()` methods for making the classifications. There also are `HealthySnacksInput` and `HealthySnacksOutput` classes that represent the inputs (an image) and outputs (the probabilities dictionary and the top prediction label) of the model.

At this point, you might reasonably expect that you’re going to use these automatically generated classes to make the predictions. Surprise… you’re not! We’re saving that for the end of the chapter.

There are a few reasons for this, most importantly that the images need to be scaled to 227×227 pixels and placed into a `CVPixelBuffer` object before you can call the `prediction()` method, and we'd rather not deal with that if we can avoid it. So instead, you’re going to be using yet another framework: Vision.

> **Note**: Core ML models can also have other types of inputs besides images, such as numbers and text. In this first section of the book, you'll primarily work with images but, in later sections, you'll also do machine learning on other types of data.

Vision

Along with Core ML, Apple also introduced the Vision framework in iOS 11. As you can guess from its name, Vision helps with computer vision tasks. For example, it can detect rectangular shapes and text in images, detect faces and even track moving objects.

Most importantly for you, Vision makes it easy to run Core ML models that take images as input. You can even combine this with other Vision tasks into an efficient image-processing pipeline. For example, in an app that detects people's emotions, you can build a Vision pipeline that first detects a face in the image and then runs a Core ML-based classifier on just that face to see whether the person is smiling or frowning.

It's highly recommended that you use Vision to drive Core ML if you're working with images. Recall that the HealthySnacks model needs a 227×227 image as input, but images from the photo library or the camera will be much larger and are typically not square. Vision will automatically resize and crop the image.

In the automatically generated Swift file for the .mlmodel, you may have noticed that the input image (see `HealthySnacksInput`) has to be a `CVPixelBuffer` object, while `UIImagePickerController` gives you a `UIImage` instead. Vision can do this conversion for you, so you don't have to worry about `CVPixelBuffer` objects.

Finally, Vision also performs a few other tricks, such as rotating the image so that it's always right-size up, and matching the image's color to the device's color space. Without the Vision framework, you'd have to write additional code by hand! Surely, you'll agree that it's much more convenient to let Vision handle all these things.

> **Note**: Of course, if you're using a model that does not take images as input, you can't use Vision. In that case, you'll have to use the Core ML API directly.

The way Vision works is that you create a `VNRequest` object, which describes the task you want to perform, and then you use a `VNImageRequestHandler` to execute the request. Since you'll use Vision to run a Core ML model, the request is a subclass named `VNCoreMLRequest`. Let's write some code!

Creating the VNCoreML request

To add image classification to the app, you're going to implement `classify(image:)` in **ViewController.swift**. This method is currently empty. Here, you'll use Vision to run the Core ML model and interpret its results. First, add the required imports to the top of the file:

```
import CoreML
import Vision
```

Next, you need to create the `VNCoreMLRequest` object. You typically create this request object once and re-use it for every image that you want to classify. Don't create a new request object every time you want to classify an image — that's wasteful.

In **ViewController.swift**, add the following code inside the `ViewController` class below the `@IBOutlets`:

```
lazy var classificationRequest: VNCoreMLRequest = {
  do {
    // 1
    let healthySnacks = HealthySnacks()
    // 2
    let visionModel = try VNCoreMLModel(for: healthySnacks.model)
    // 3
    let request = VNCoreMLRequest(model: visionModel,
                                  completionHandler: {
      [weak self] request, error in
      print("Request is finished!", request.results)
    })
    // 4
    request.imageCropAndScaleOption = .centerCrop
    return request
  } catch {
    fatalError("Failed to create VNCoreMLModel: \(error)")
  }
}()
```

Here's what this code does:

1. Create an instance of `HealthySnacks`. This is the class from the .mlmodel file's automatically generated code. You won't use this class directly, only so you can pass its `MLModel` object to Vision.

2. Create a `VNCoreMLModel` object. This is a wrapper object that connects the `MLModel` instance from the Core ML framework with Vision.

3. Create the `VNCoreMLRequest` object. This object will perform the actual actions of converting the input image to a `CVPixelBuffer`, scaling it to 227×227, running the Core ML model, interpreting the results, and so on.

   Since Vision requests run asynchronously, you need to supply a completion handler that will receive the results. For now, the completion handler just prints something to the Xcode debug output pane. You will flesh this out later.

4. The `imageCropAndScaleOption` tells Vision how it should resize the photo down to the 227×227 pixels that the model expects.

The code is wrapped up in a `do catch` because loading the `VNCoreMLModel` object can fail if the .mlmodel file is invalid somehow. That should never happen in this example project, and so you handle this kind of error by crashing the app. It is possible for apps to download an .mlmodel file and, if the download fails, the .mlmodel can get corrupted. In that case, you'll want to handle this error in a more graceful way.

> **Note**: The `classificationRequest` variable is a `lazy` property. In case you're unfamiliar with lazy properties, this just means that the `VNCoreMLRequest` object is not created until the very first time you use `classificationRequest` in the app.

Crop and scale options

It has been mentioned a few times now that the model you're using, which is based on SqueezeNet, requires input images that are 227×227 pixels. Since you're using Vision, you don't really need to worry about this — Vision will automatically scale the image to the correct size. However, there is more than one way to resize an image, and you need to choose the correct method for the model, otherwise it might not work as well as you'd hoped.

What the correct method is for your model depends on how it was trained. When a model is trained, it's shown many different example images to learn from. Those images have all kinds of different dimensions and aspect ratios, and they also need to be resized to 227×227 pixels. There are different ways to do this and not everyone uses the same method when training their models.

For the best results you should set the request's `imageCropAndScaleOption` property so that it uses the same method that was used during training.

Vision offers three possible choices:

- `centerCrop`
- `scaleFill`
- `scaleFit`

The `.centerCrop` option first resizes the image so that the smallest side is 227 pixels, and then it crops out the center square:

The centerCrop option

Note that this removes pixels from the left and right edges of the image (or from the top/bottom if the image is in portrait). If the object of interest happens to be in that part of the image, then this will throw away useful information and the classifier may only see a portion of the object. When using `.centerCrop` it's essential that the user points the camera so that the object is in the center of the picture.

With `.scaleFill`, the image gets resized to 227×227 without removing anything from the sides, so it keeps all the information from the original image — but if the original wasn't square then the image gets squashed. Finally, `.scaleFit` keeps the aspect ratio intact but compensates by filling in the rest with black pixels.

The scaleFill and scaleFit options

For the Healthy Snacks app, you'll use `.centerCrop` as that's also the resizing strategy that was used to train the model. Just make sure that the object you're pointing the camera at is near the center of the picture for the best results. Feel free to try out the other scaling options to see what kind of difference they make to the predictions, if any.

Performing the request

Now that you have the request object, you can implement the `classify(image:)` method. Add the following code to that method:

```
func classify(image: UIImage) {
  // 1
  guard let ciImage = CIImage(image: image) else {
    print("Unable to create CIImage")
    return
  }
  // 2
  let orientation = CGImagePropertyOrientation(image.imageOrientation)
  // 3
  DispatchQueue.global(qos: .userInitiated).async {
    // 4
    let handler = VNImageRequestHandler(ciImage: ciImage,
                                        orientation: orientation)
    do {
      try handler.perform([self.classificationRequest])
    } catch {
      print("Failed to perform classification: \(error)")
    }
  }
}
```

The image that you get from `UIImagePickerController` is a `UIImage` object but Vision prefers to work with `CGImage` or `CIImage` objects. Either will work fine, and they're both easy to obtain from the original `UIImage`. The advantage of using a `CIImage` is that this lets you apply additional Core Image transformations to the image, for more advanced image processing.

Here is what the method does, step-by-step:

1. Converts the `UIImage` to a `CIImage` object.
2. The `UIImage` has an `imageOrientation` property that describes which way is up when the image is to be drawn. For example, if the orientation is "down," then the image should be rotated 180 degrees. You need to tell Vision about the image's orientation so that it can rotate the image if necessary, since Core ML expects images to be upright.
3. Because it may take Core ML a moment or two to do all the calculations involved in the classification (recall that SqueezeNet does 390 million calculations for a single image), it is best to perform the request on a background queue, so as not to block the main thread.

4. Create a new `VNImageRequestHandler` for this image and its orientation information, then call `perform()` to actually do execute the request. Note that `perform()` takes an array of `VNRequest` objects, so that you can perform multiple Vision requests on the same image if you want to. Here, you just use the `VNCoreMLRequest` object from the `classificationRequest` property you made earlier.

The above steps are pretty much the same for any Vision Core ML app.

Because you made the `classificationRequest` a `lazy` property, the very first time `classify(image:)` gets called it will load the Core ML model and set up the Vision request. But it only does this once and then re-uses the same request object for every image. On the other hand, you do need to create a new `VNImageRequestHandler` every time, because this handler object is specific to the image you're trying to classify.

Image orientation

When you take a photo with the iPhone's camera, regardless of how you're holding the phone, the image data is stored as landscape because that's the native orientation of the camera sensor. iOS keeps track of the true orientation of the image with the `imageOrientation` property. For an image in your photo album, the orientation information is stored in the image file's EXIF data.

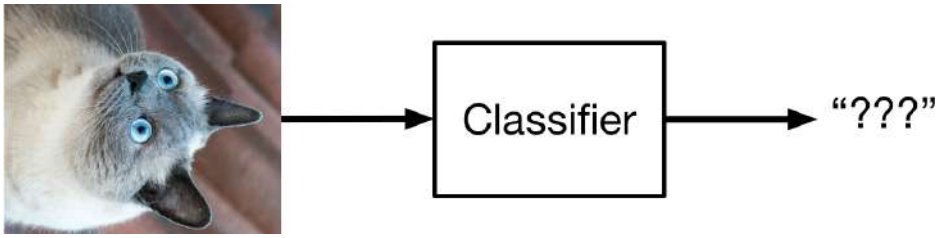
If you're holding the phone in portrait mode and snap a picture, its `imageOrientation` will be `.right` to indicate the camera has been rotated 90 degrees clockwise — 0 degrees means that the phone was in landscape with the Home button on the right.

An `imageOrientation` of `.up` means that the image already has the correct side up. This is true for pictures taken in landscape but also for portrait pictures from other sources, such as an image you create in Photoshop.

Most image classification models expect to see the input image with the correct side up. Notice that the Core ML model does not take "image orientation" as an input, so it will see only the "raw" pixels in the image buffer without knowing which side is up.

Image classifiers are typically trained to account for images being horizontally flipped so that they can recognize objects facing left as well as facing right, but they're usually not trained to deal with images that rotated by 90, 180 or 270 degrees.

If you pass in an image that is not oriented properly, the model may not give accurate predictions because it has not learned to look at images that way.



This cat is not right-side up

This is why you need to tell Vision about the image's orientation so that it can properly rotate the image's pixels before they get passed to Core ML. Since Vision uses `CGImage` or `CIImage` instead of `UIImage`, you need to convert the `UIImageOrientation` value to a `CGImagePropertyOrientation` value.

Trying it out

At this point, you can build and run the app and choose a photo.

It's possible to run this app in the Simulator but only the photo library button is active. The photo library on the Simulator doesn't contain pictures of snacks by default, but you can add your own by Googling for images and then dragging those JPEGs or PNGs into the Photos app.

Run the app on a device to use the camera, as the Simulator does not support taking pictures.

Take or choose a picture, and the Xcode debug pane will output something like this:

```
Request is finished! Optional([<VNClassificationObservation:
0x60c00022b940> B09B3F7D-89CF-405A-ABE3-6F4AF67683BB 0.81705
"healthy" (0.917060), <VNClassificationObservation: 0x60c000223580>
BC9198C6-8264-4B3A-AB3A-5AAE84F638A4 0.18295 "unhealthy" (0.082940)])
```

This is the output from the `print` statement in the completion handler of the `VNCoreMLRequest`. It prints out the `request.results` array. As you can see, this array contains two `VNClassificationObservation` objects, one with the probability for the healthy class (91.7%) and the other with the probability for the unhealthy class (8.29%).

Of course, printing stuff to the output pane isn't very exciting, so let's properly show these results in the app.

Showing the results

Inside the declaration of `lazy var classificationRequest`, change the completion handler for the `VNCoreMLRequest` object to the following:

```
let request = VNCoreMLRequest(model: visionModel, completionHandler: {
  [weak self] request, error in
  self?.processObservations(for: request, error: error)  // add this
})
```

Instead of the `print` statement that was there previously, you're now calling a new method, `processObservations(for:error:)`. It's perfectly possible to put the code that handles the results directly inside the completion handler, but it tends to make the code harder to read.

Add the new method to **ViewController.swift**:

```
func processObservations(for request: VNRequest, error: Error?) {
  // 1
  DispatchQueue.main.async {
    // 2
    if let results = request.results as? [VNClassificationObservation] {
      // 3
      if results.isEmpty {
        self.resultsLabel.text = "nothing found"
      } else {
        self.resultsLabel.text = String(format: "%@ %.1f%%",
                                        results[0].identifier,
                                        results[0].confidence * 100)
      }
    // 4
    } else if let error = error {
      self.resultsLabel.text = "error: \(error.localizedDescription)"
    } else {
      self.resultsLabel.text = "???"
    }
    // 5
    self.showResultsView()
  }
}
```

Here's what this method does, step-by-step:

1. The request's completion handler is called on the same background queue from which you launched the request. Because you're only allowed to call UIKit methods from the main queue, the rest of the code in this method runs on the main queue.

2. The `request` parameter is of type `VNRequest`, the base class of `VNCoreMLRequest`. If everything went well, the request's `results` array contains one or more `VNClassificationObservation` objects. If the cast fails, it's either because there was an error performing the request and `results` is nil, or the array contains a different type of observation object, which happens if the model isn't actually a classifier or the Vision request object wasn't for a Core ML model.

3. Put the class name in the results label. Assuming the array is not empty, it contains a `VNClassificationObservation` object for each possible class. Each of these has an `identifier` (the name of the class: "healthy" or "unhealthy") and a `confidence` score. This score is how likely the model thinks the object is of this class; in other words, it's the probability for that class.

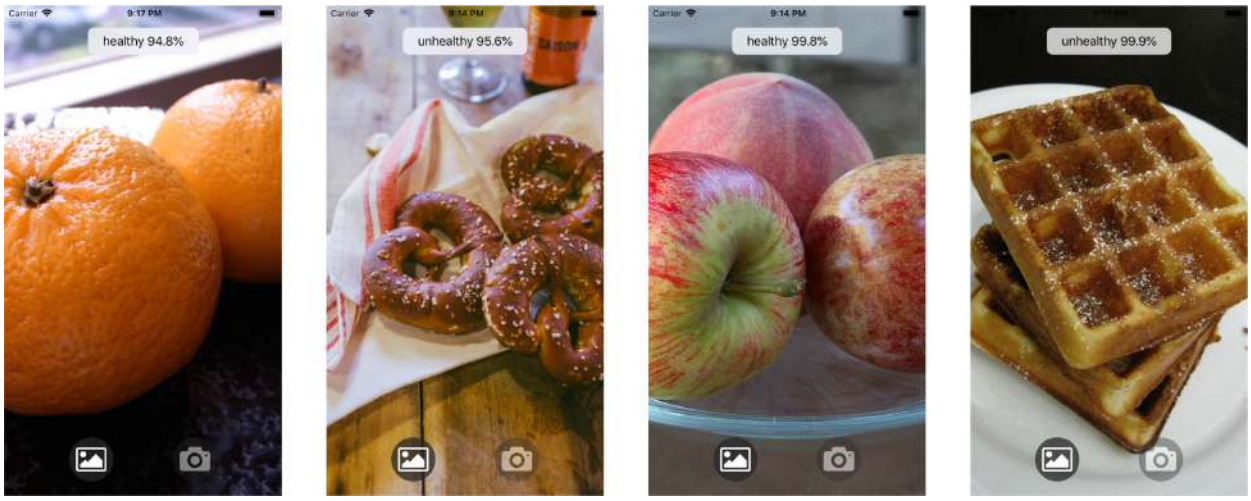
   Vision automatically sorts the results by confidence, so `results[0]` contains the class with the highest confidence — the winning class. The app will show both the name and confidence in the results label, where the confidence is shown as a percentage, e.g., `"healthy 95%"`.

   By the way, it should never happen that the array is empty but, in the unlikely case that it is, you show a "nothing found" message in the label.

4. Just in case something went wrong with the request, show an error message. This normally shouldn't happen, but it's good to cover all your bases.

5. Finally, show the `resultsLabel` on the screen. The `showResultsView()` method performs a nice little animation, which makes it clear to the user that their image has been classified.

And that's all you need to do. Build and run the app and classify some images!



Predictions on a few test images

Pretty cool. With just a few lines of code you've added a state-of-the-art image classifier to your app!

> **Note**: When you viewed the Core ML model in Xcode (by selecting the .mlmodel file in the Project navigator), it said that the model had two outputs: a dictionary containing the probabilities and the label for the top prediction. However, the Vision request gives you an array of `VNClassificationObservation` objects instead. Vision takes that dictionary from Core ML and turns it into its own kind of '"observation" objects. Later on, you'll see how to use Core ML directly, without using Vision, and, in that case, you do get access directly to the model's outputs.

What if the image doesn't have a snack?

The app shows the winning class and the confidence it has in this prediction. In the above image on the left, the class is "healthy" and the confidence is 94.8%.

If the output is something like "healthy 95%," the model feels pretty sure about itself. You'll see this kind of prediction on pictures of oranges, apples, bananas and so on. Likewise, if the output is "unhealthy 95%," the model is pretty sure that it's correct about the snack being unhealthy, and you'll see this on pictures of pretzels and waffles. That's good; we like to see confident predictions.

The model used in this app was trained on 20 different types of snacks. But what happens when you show it a kind of snack that it has never seen before, or maybe even a totally different kind of object — maybe something that isn't even edible?

Since a binary classifier only understands two classes, it puts any picture that you give it into the "healthy" category or into the "unhealthy" category, even if the picture isn't really of a kind of snack that it knows about.

This particular classifier is trained to tell the difference between healthy and unhealthy snacks, and it should therefore be used only with photos of such snacks. For all other images - let's say of cute cats — the classifier will give a non-sensical prediction. After all, it only has "healthy" or "unhealthy" to choose from. (And no, we do not endorse having cats as a snack.)

What you *want* to happen for such an "unsupported" input image is that the model gives a very uncertain prediction, something that is more like a 51%–49% split. In that case, Vision might return two `VNClassificationObservation` objects like this:

```
element 0: healthy 51%
element 1: unhealthy 49%
```

If the model isn't sure, that's actually a very acceptable answer: It could be either class. However, since Vision automatically sorts this array by confidence score, the app will show the prediction "healthy" as the winning label. But is it really? Since the model is so uncertain now, changing these percentages only slightly can completely change the outcome:

```
element 0: unhealthy 52%
element 1: healthy 48%
```

If you get such a prediction for one of your photos, try taking the same photo again but from a slightly different angle. The small variation between the photos can easily flip the uncertain prediction from one class to the other.

The moral of the story is that when the probabilities get close to 50%–50%, the model doesn't really know what to make of the image. It's a good idea to make the app deal with such situations. After all, there is nothing that prevents the user from taking a photo of something that is not a snack.

In `processObservations(for:error:)`, add the following clause to the if statement:

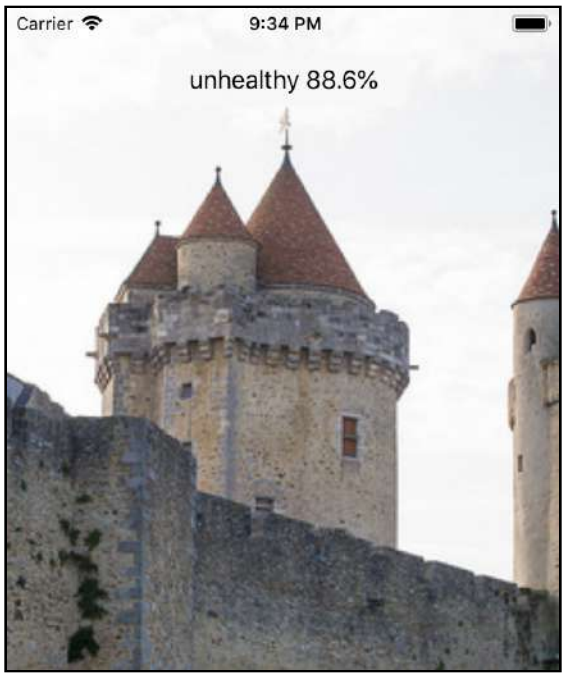
```
if results.isEmpty {
  . . .
} else if results[0].confidence < 0.8 {
  self.resultsLabel.text = "not sure"
} else {
  . . .
```

Here, we've chosen a threshold value of 0.8 (or 80% confidence). If the model was less confident about its winning prediction than this threshold, you decide that you can't trust the prediction it made, and the app will say "not sure."

The threshold value of 0.8 was picked arbitrarily. This is something you would test in practice by pointing the phone at many real-world objects to get a feel for what confidence level is trustworthy and below which level the model starts to make too many mistakes. This is actually different for every model, and so you need to test it in practice. There are also mathematical ways to find a suitable threshold.

> **Note**: Remember that it doesn't make sense to test for a confidence below 0.5, as the winning prediction will always have a confidence score of greater than 50%. There are only two classes in a binary classifier and their total confidence score needs to add up to 100%.

However, it can still happen that you run into a situation like this:



Yeah, I wouldn't eat this either

The model was quite confident about this prediction even though the object is far from edible! Sometimes the classifier will give a very confident answer that is totally wrong. This is a limitation of all classifiers.

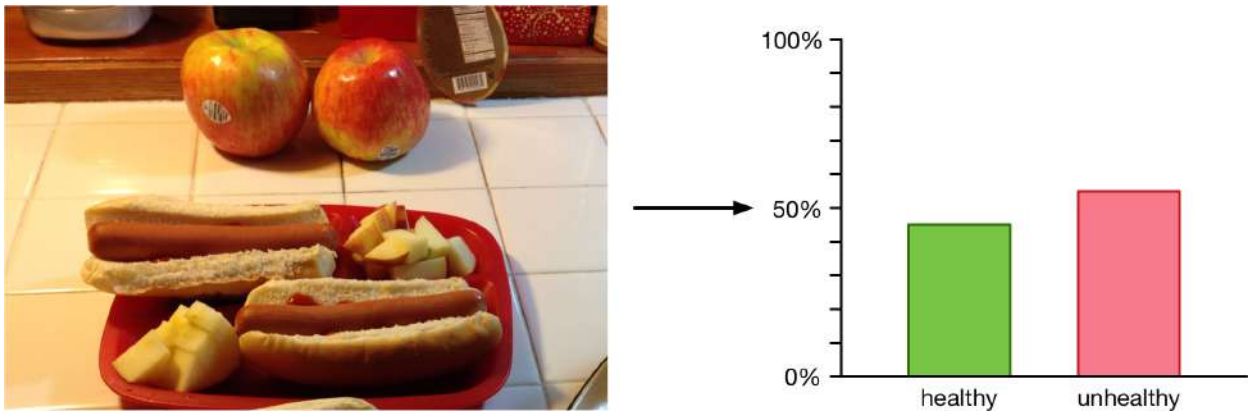
It's important to understand that machine learning models will only work reliably when you use them with data that is very similar to the data they've been trained on. A model can only make trustworthy predictions on the types of things it has learned about — it will fail spectacularly on anything else. Machine learning often seems like magic… but it does have its limitations.

The only way to fix this kind of problem is to make your model more robust by training it on more images, or by adding a third category so that the model can learn the difference between "healthy snack," "unhealthy snack," and "not a snack." But even then your model will still make errors. Using machine learning for computer vision tasks works really well, but it's never perfect.

In the chapter on training, you'll see how you can estimate the quality of the model to get an idea of how well it will work in practice.

What if there's more than one object in the image?

Image classification always looks at the entire image and tries to find out what the most prominent object in the image is. But nothing stops you from running an image classifier on a picture containing objects from more than one class:



Make up your mind!

In this example, the classifier has found both an apple and a hotdog, but it seems to think that the hot dog is slightly more important. Perhaps it's because the hot dog takes up more room in the image, or maybe the model just had a harder time recognizing the apples. In any case, it had to make an impossible choice between two classes that are really supposed to be mutually exclusive and this is what it came up with.

However, based on these percentages, you can't just say, "This image contains an unhealthy snack." It does, but it also contains a healthy snack. With the new rule that we just added, the model would say "not sure" for this particular photo, since neither class has over 80% confidence.

But it's also possible that the model predicts something like 90% healthy or unhealthy for an image such as this. All bets are off, since this is not a problem the HealthySnacks model was really trained for. With an image classifier like this, the input image is really supposed to contain one "main" object, not multiple objects — or at the very least multiple objects that are all from the same class. The model can't really handle images with more than one object if they are from different classes.

In any case, image classification works best when there is just a single object in the image. The computer vision task that's about finding all the objects in an image, and also where they are located in the image, is called "object detection" and we'll talk about that in the chapter "Beyond Image Classification."

How does it work?

At this point, you may be wondering exactly how this Core ML model is able to tell apart healthy snacks from unhealthy snacks. The model takes an image as input and produces a probability distribution as output, but what is the magic that makes this happen? Let's peek under the hood a little.

The **HealthySnacks.mlmodel** is a so-called neural network classifier. You've already seen classification, but you may not know exactly what a neural network is.

Artificial neural networks are inspired by the human brain. The particular neural network used by HealthySnacks is a so-called "convolutional" neural network, which in many ways is similar to how the human visual cortex processes information.

Despite how they're often depicted in the popular press, it's really not that useful to think of these artificial neural networks as a computerized version of human brains. Artificial neural networks are only a very crude model of how the human brain works — and not nearly as complicated.

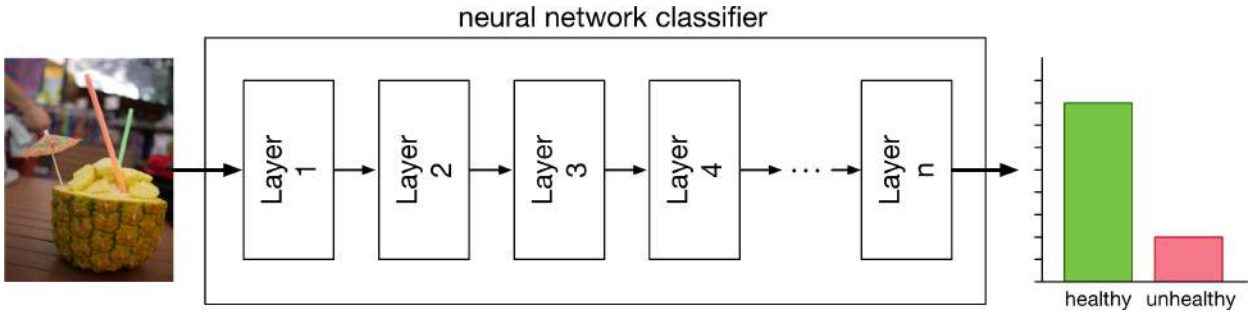
It's much more constructive to think of a neural network as a pipeline that transforms data in several different stages. A machine learning model is like a Swift function:

```
let outputs = myModel(inputs)
```

In the case of an image classifier, the function signature looks like the following, where the input is an image of some kind and the output an array of numbers, the probability distribution over the classes:

```
func myModel(input: Image) -> [Double] {
  // a lot of fancy math
}
```

Core ML treats the model as a black box, where input goes into one end and the output comes out the other. Inside this black box it actually looks like a pipeline with multiple stages:

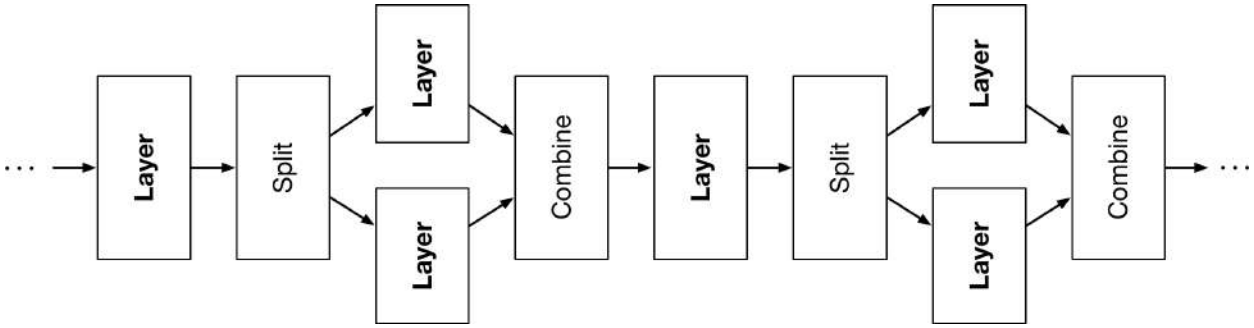


The model is a pipeline

Each of these stages, or **layers** as we call them, transforms the data in some way. In code, you can think of it as a sequence of `map`, `filter`, and `reduce` operations:

```
func myModel(input: Image) -> [Double] {
  return input.map({...}).filter({...}).map({...}).reduce({...})
}
```

That's really all there is to it. Despite its very sci-fi name, a neural network is a very straightforward thing, just a series of successive stages that each transforms the data in its own way, until the data comes out in the form you want. The layers inside an image classifier transform the data from an image into a probability distribution. In modern neural networks, pipelines are not just a straight series of transformations but they can branch and the results of branches can be combined again in a later stage. For example, the SqueezeNet neural network architecture that the HealthySnacks model is based on looks something like this:



Part of the SqueezeNet pipeline

All the magic happens inside the layers that perform the transformations. So surely that must involve lots of complicated math? Well, no. Each individual transformation is a relatively simple mathematical operation. The power of the neural network comes from combining these transformations. By putting many simple transformations together, you end up with a pipeline that can compute the answers to some pretty complex problems.

Early neural networks only used two or three layers (transformations), as training with more layers was fraught with problems. But those problems have been solved in recent years and now we routinely use neural networks with dozens or even hundreds of layers, which is why using these neural nets is called "deep learning." SqueezeNet has 67 layers although in practice certain types of layers are "fused" together for better speed.

Into the next dimension

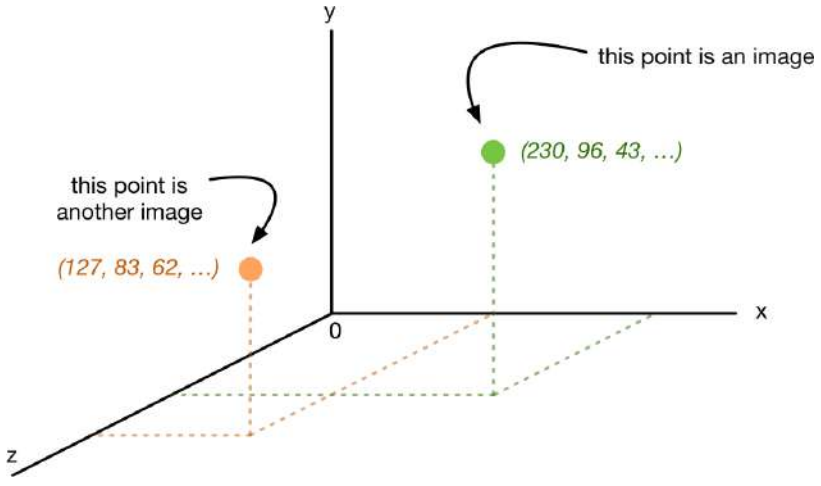
Let's dive a little deeper into the math, just so you get a better conceptual idea of what these transformations do. Neural networks, like most machine learning models, can only work with numerical data. Fortunately for us, the data we care about in this chapter — the input image and the output probabilities — are all represented as numbers already. Models that work on data such as text first need to convert that data into numbers.

The input image is 227×227 pixels and is a color image, so you need 227 × 227 × 3 = 154,587 numbers to describe an input image. For the sake of explanation, let's round this down to 150,000 numbers.

> **Note**: Each pixel needs three numbers because color is stored as RGB: a red, green and blue intensity value. Some images also have a fourth channel, the alpha channel, that stores transparency information, but this is typically not used by image classifiers. It's OK to use an RGBA image as input, but the classifier will simply ignore the alpha value.

Here's the big idea: Each of the 227×227 input images can be represented by a unique point in a 150,000-dimensional space.

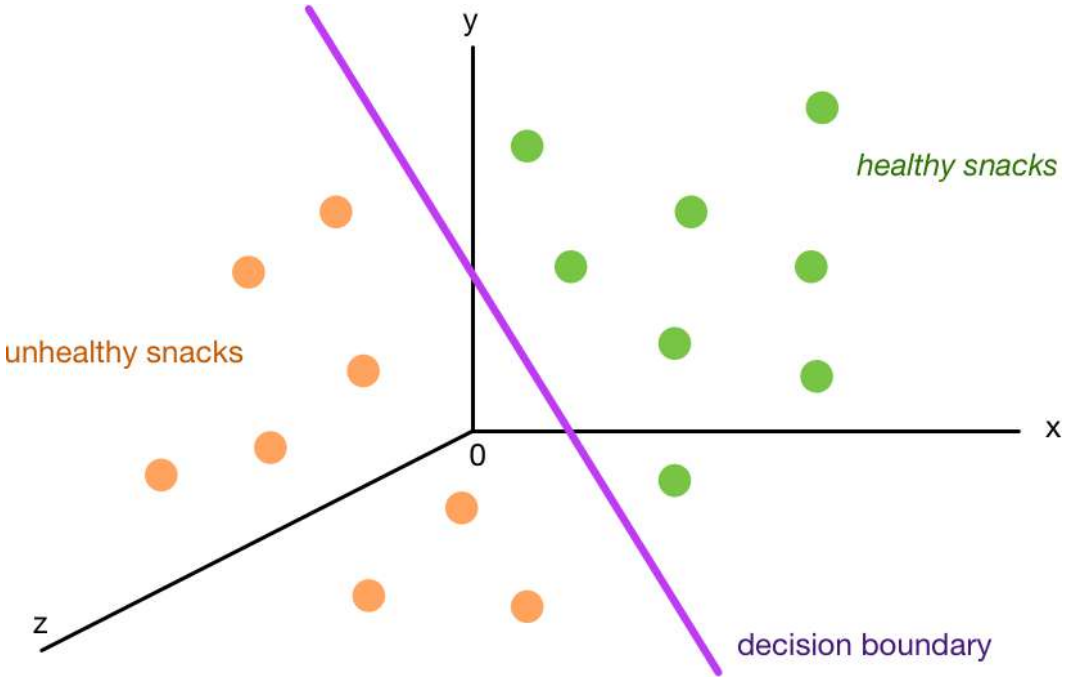
Whoop, try to wrap your head around that… It's pretty easy for us humans to think in 3D space but not so much in higher-dimensional spaces, especially not ones with hundreds of thousands of dimensions. But the principle is the same: given 3 numbers *(x, y, z)* you can describe any point in 3-dimensional space, right? Well, given 150,000 numbers with the RGB values of all the pixels in the image, you end up at a point in 150,000-dimensional space.



Pretend this is 150,000 dimensions

By the way, don't try to think in 150,000 dimensions. Just imagine a 3D space and pretend it's more than three dimensions. That's what everyone else does, since humans simply aren't capable of visualizing more than three dimensions.

To classify the images, you want to be able to draw a line through this high-dimensional space and say, "All the images containing healthy snacks are on this side of the line, and all the images with unhealthy snacks are on the other side." If that would be possible, then classifying an image is easy: You just have to look at which side of the line the image's point falls.



The decision boundary divides up the space into two classes

This line is called the **decision boundary**. It's the job of the classifier model to learn where that decision boundary lies. Math alert: It's not really a line but a hyperplane, which is a subspace that splits the high-dimensional space into two halves. One of the benefits of being a machine learning practitioner is that you get to use cool words such as hyperplane.

The problem is that you cannot draw a nice line through the 150,000-dimensional pixel space because ordering the images by their pixel values means that the healthy and unhealthy images are all over the place.

Since pixels capture light intensity, images that have the same color and brightness are grouped together, while images that have different colors are farther apart. Apples can be red or green but, in pixel space, such images are not close together. Candy can also be red or green, so you'll find pictures of apples mixed up with pictures of candy.

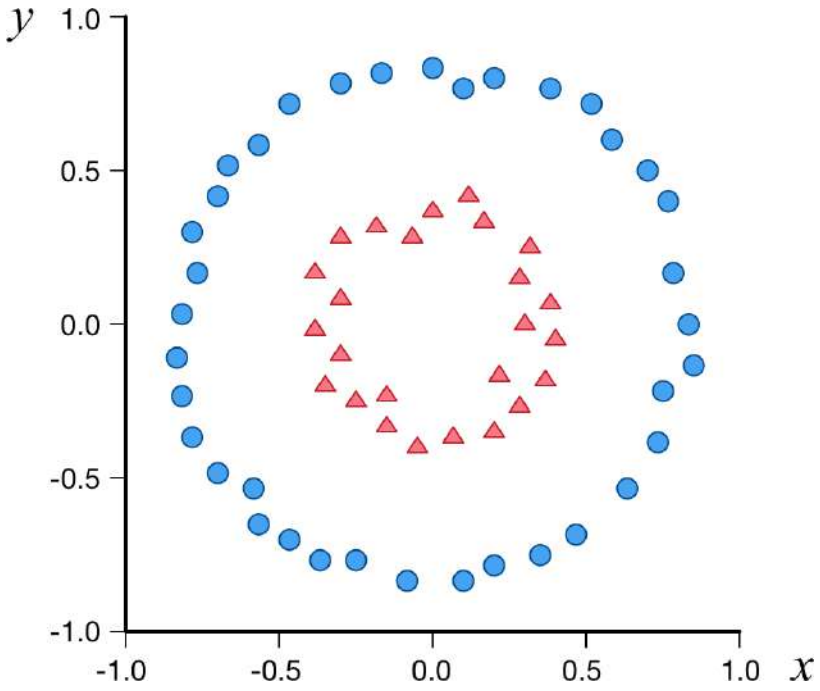
You cannot just look at how red or green something is to decide whether this image contains something healthy or unhealthy.

All the information you need to make a classification is obviously contained in the images, but the way the images are spread out over this 150,000-dimensional pixel space is not very useful. What you want instead is a space where all the healthy snacks are grouped together and all the unhealthy snacks are grouped together, too.

This is where the neural network comes in: The transformations that it performs in each stage of the pipeline will twist, turn, pull and stretch this coordinate space, until all the points that represent healthy snacks will be over on one side and all the points for unhealthy snacks will be on the other, and you can finally draw that line between them.

A concrete example

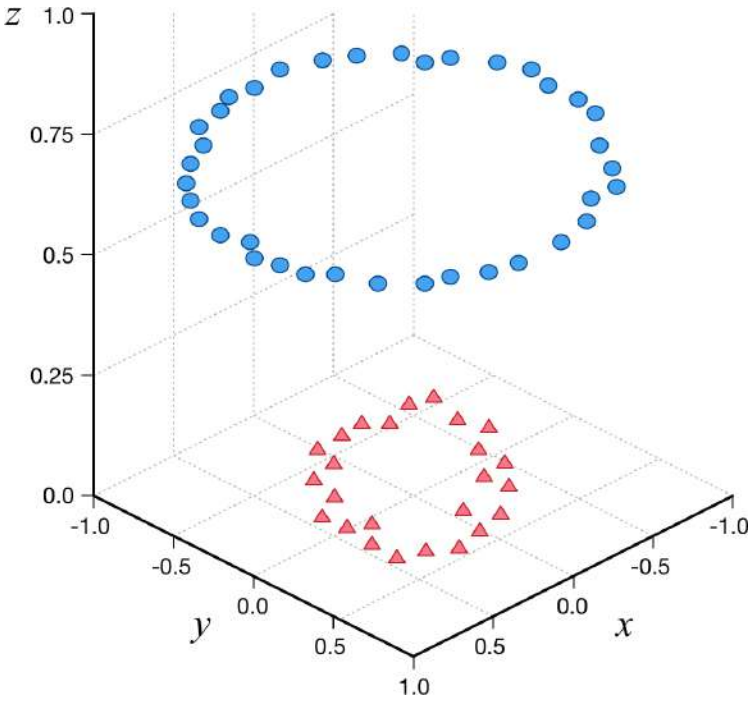
Here is a famous example that should illustrate the idea. In this example the data is two-dimensional, so each input consists of only two numbers *(x, y)*. This is also a binary classification problem, but in the original coordinate space it's impossible to draw a straight line between the two classes:



An impossible classification problem...

In theory, you could classify this dataset by learning to separate this space using an ellipse instead of a straight line, but that's rather complicated. It's much easier to perform a smart transformation that turns the 2D space into a 3D space by giving all

points a *z*-coordinate too. The points from class A (the triangles) get a small *z* value, the points from class B (the circles) get a larger *z* value. Now the picture looks like this:



...but easy after transforming the data

After applying this transformation, both classes get cleanly separated. You can easily draw a line between them at *z = 0.5*. Any point with z-coordinate less than 0.5 belongs to class A, and any point with z greater than 0.5 belongs to class B.

The closer a point's z-coordinate is to the line, the less confident the model is about the class for that point. This also explains why probabilities get closer to 50% when the HealthySnacks model can't decide whether the snack in the image is healthy or unhealthy. In that case, the image gets transformed to a point that is near the decision boundary. Usually, the decision boundary is a little fuzzy and points with *z* close to 0.5 could belong to either class A or class B.

The cool thing about neural networks is that they can automatically learn to make these kinds of transformations, to convert the input data from a coordinate space where it's hard to tell the points apart, into a coordinate space where it's easy. That is exactly what happens when you train the model. It learns the transformations and how to find the best decision boundary.

To classify a new image, the neural network will apply all the transformations it has learned during training, and then it looks at which side of the line the transformed image falls. And that's the secret sauce of neural network classification!

The only difference between this simple example and our image classifier is that you're dealing with 150,000 dimensions instead of two. But the idea – and the underlying mathematics — is exactly the same for 150,000 dimensions as it is for two.

> **Note**: In general, the more complex the data, the deeper the neural network has to be. For the 2D example above, a neural net with just two layers will suffice. For images, which are clearly much more complex, the neural net needs to be deeper because it needs to perform more transformations to get a nice, clean decision boundary.

Over the course of the next chapters, we'll go into more details about exactly what sort of transformations are performed by the neural network but, in a typical deep learning model, these are convolutions (look for patterns made by small groups of pixels, thereby mapping the points from one coordinate space to another), pooling (reduce the size of the image to make the coordinate space smaller), and logistic regression (find where to draw the line / decision boundary).

Multi-class classification

So far, we've covered binary classification in which there are only two classes, but it's also really easy to use a model that can handle multiple classes. This is called… wait for it… a multi-class classifier — or, sometimes, a multinomial classifier.

In this section, you'll swap out the binary classifier for **MultiSnacks.mlmodel**, a multi-class classifier that was trained on the exact same data as the binary healthy/unhealthy classifier but that can detect the individual snacks.

Recognizing multiple classes

Integrating this new model into the app couldn't be simpler. You can either do this in a copy of your existing app or use the MultiSnacks starter app.

Now, drag the **MultiSnacks.mlmodel** from this chapter's downloaded resources into the Xcode project.

If you look at this new .mlmodel file in Xcode, or at the automatically generated code, you'll notice that it looks exactly the same as before, except that the names of the Swift classes are different (`MultiSnacks` instead of `HealthySnacks`) because the name of the .mlmodel file is different, too.

To use this new model, make the following change on the `classificationRequest` property:

```
lazy var classificationRequest: VNCoreMLRequest = {
  do {
    let multiSnacks = MultiSnacks()
    let visionModel = try VNCoreMLModel(for: multiSnacks.model)
    . . .
```

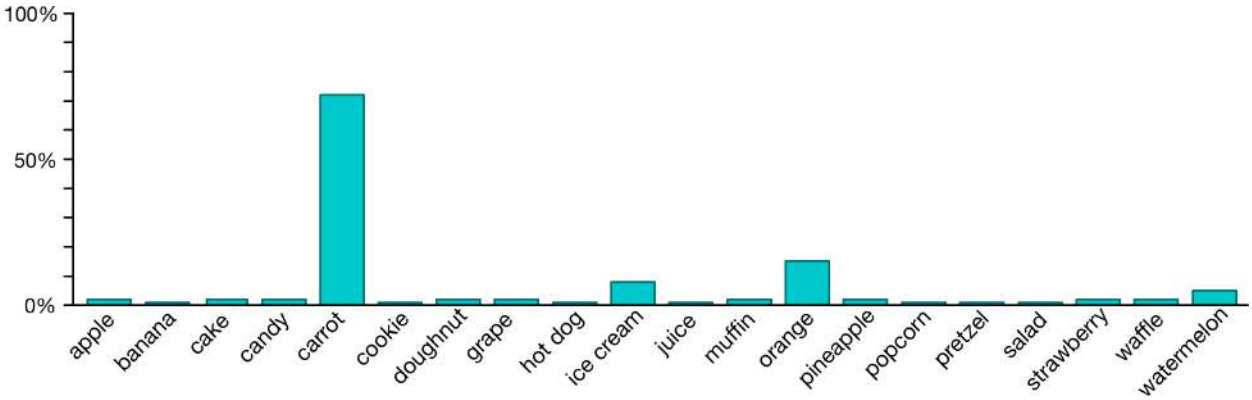
Instead of creating an instance of `HealthySnacks`, all you need to do is make an instance of `MultiSnacks`. This is the name of the class that Xcode generated automatically when you added MultiSnacks.mlmodel to the project.

Also change the innermost if statement in `processObservations(for:error:)` to:

```
if results.isEmpty {
  self.resultsLabel.text = "nothing found"
} else {
  let top3 = results.prefix(3).map { observation in
    String(format: "%@ %.1f%%", observation.identifier,
           observation.confidence * 100)
  }
  self.resultsLabel.text = top3.joined(separator: "\n")
}
```

Instead of showing only the best result — the class with the highest confidence score — this now displays the names of the three best classes.

Since the model was trained on 20 different object types, it outputs a probability distribution that looks something like this:



The new probability distribution

Where previously there were only two values (healthy/unhealthy), there are now 20 possible outcomes, and the 100 total percentage points are distributed over these twenty possible classes — which is why it's called a probability distribution.

The app displays the three predicted classes with the highest probability values. Since there are now 20 classes, the `results` array also contains 20 `VNClassificationObservation` objects, sorted from a high to low confidence score. The `prefix(3)` method grabs elements 0, 1, and 2 from this array (the ones with the highest probabilities), and you use `map` to turn them into strings. For the above probability distribution, this gives:

```
element 0: carrot 72%
element 1: orange 15%
element 2: ice cream 8%
```

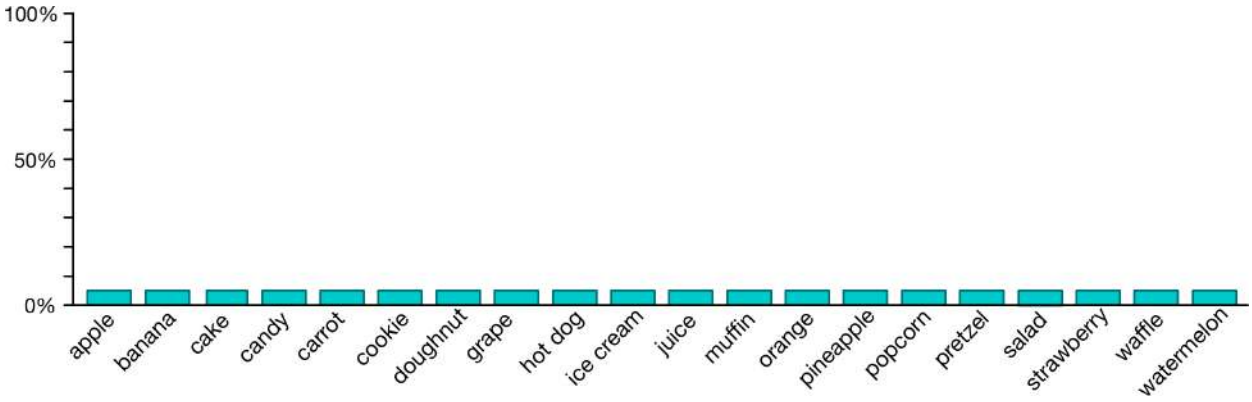
The model is fairly confident about this prediction. The first result has a pretty high score, and so you can probably believe that the image really is of a carrot.

The second result is often fairly reasonable — if you squint, an orange could look like a carrot — but the third result and anything below it can be way off the mark. Given these confidence scores, that's OK; the model really didn't think ice cream was a reasonable guess here.

> **Note**: The percentages of these top three choices don't have to add up to 100%, since there are another 17 classes that will make up the remainder.

Notice that, when you made these changes to the code, you removed the `if` statement that checked whether the confidence was less than 80%. That made sense for a binary classifier but, when you have multiple classes, the best confidence will often be around the 60% mark. That's still a pretty confident score.

With a binary classifier and two classes, a random guess is correct 50% of the time. But with 20 classes, a random guess would be correct only 1/20, or 5%, of the time. When the multi-class model is very unsure about what is in the image, the probability distribution would look more like this:



When the multi-class model is unsure

You could still add a kind of "not sure" threshold, but a more reasonable value would be 0.4, or 40%, instead of the 80% that you used with the binary classifier.

Still, just like a binary classifier, the predictions from a multi-class model only make sense if you show it the types of objects that it has been trained to recognize.

If you give the new classifier an image of something that is not one of the 20 kinds of snacks it knows about, such as a dachshund, the model may return a very unsure prediction ("it could be anything") or a very confident but totally wrong prediction ("it's a hot dog").

Again, you can ask what happens when an image contains objects of more than one class?

Well, unlike with the binary classifier in which predictions became very uncertain (50–50), here, a similar thing happens but now the probabilities get divided over more classes:

Image with multiple types of fruit

In this example, the classifier correctly recognizes apples and carrots as the top choices, and it tries to split the probabilities between them.

This is why you're looking at the top three results instead of just the single best score. In image classification competitions, classifiers are usually scored on how well they do on their five best guesses since, that way, you can deal with one image containing more than one object or with objects that are a little ambiguous. As long as the correct answer is among the best five (or three) guesses, we're happy.

The top-one accuracy says, "Did the classifier get the most important object right?" while the top-three or top-five accuracy says, "Did it find all of the important objects?" For example, if an image that scored orange 70%, watermelon 21%, and muffin 3% really contained a watermelon and not an orange, it would still be counted as a correct classification.

Note: Don't confuse multi-class with "multi-label." A multi-class classifier's job is to choose a single category for an object from multiple categories. A multi-label classifier's job is to choose as many categories as applicable for the same object. For example, a multi-label snacks classifier could classify an apple as "healthy," "fruit," and "red."

Key points

To recap, doing image classification with Core ML and Vision in your app involves the following steps:

1. Obtain a trained .mlmodel file from somewhere. You can sometimes find pre-trained models on the web (Apple has a few on its website) but usually you'll have to build your own. You'll learn how to do this in the next chapter.
2. Add the .mlmodel file to your Xcode project.
3. Create the `VNCoreMLRequest` object (just once) and give it a completion handler that looks at the `VNClassificationObservation` objects describing the results.
4. For every image that you want to classify, create a new `VNImageRequestHandler` object and tell it to perform the `VNCoreMLRequest`.

These steps will work for any kind of image classification model. In fact, you can copy the code from this chapter and use it with any Core ML image classifier.

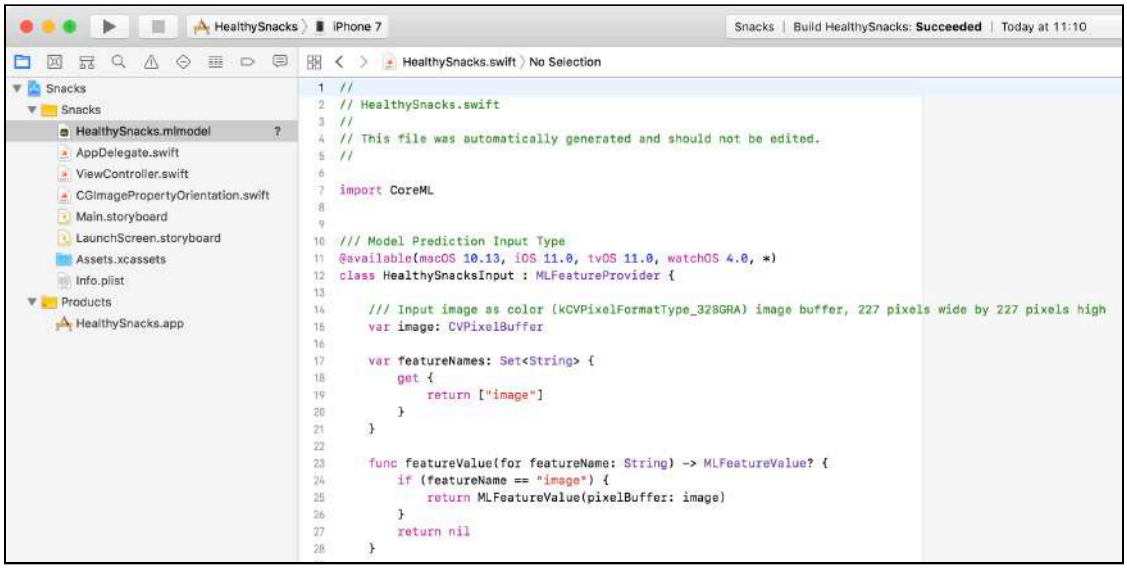
Bonus: Using Core ML without Vision

You've seen how easy it is to use Core ML through the Vision framework. Given the amount of work Vision does for you already, it's recommended to always use Vision when you're working with image models. However, it is also possible to use Core ML without Vision, and in this section you'll see how to do so.

For this section, use the starter project again and add the **HealthySnacks.mlmodel** to the project.

First, take a detailed look at the auto-generated code, since you'll use this shortly. To see this source file, first click on **HealthySnacks.mlmodel** in the Project navigator and then click on the little arrow next to the model name in the "Model Class" section.

This opens **HealthySnacks.swift**, a special source code file that doesn't actually exist anywhere in the project.



Viewing the generated code

The main class in this source file is `HealthySnacks` (located near the bottom of the file). It has a single property named `model`, which is an instance of `MLModel`, the main class in the Core ML framework. The `init()` method loads the .mlmodel from the main bundle.

There are two `prediction()` methods. The first of these takes an object of type `HealthySnacksInput` and returns a `HealthySnacksOutput`. The second one is a convenience method that takes a `CVPixelBuffer` object as input instead. Notice that there are no methods that accept a `CGImage` or a `CIImage` like with Vision.

Both `HealthySnacksInput` and `HealthySnacksOutput` are classes that implement the `MLFeatureProvider` protocol. Remember from the previous chapter that "feature" is the term we use for any value that we use for learning. An `MLFeatureProvider` is an object that describes such features to Core ML.

In the case of the `HealthySnacksInput`, there is just one feature: an image in the form of a `CVPixelBuffer` object that is 227 pixels width and 227 pixels high. Actually the model will treat each RGB pixel in this image as a separate feature, so, technically speaking, the model has 227 × 227 × 3 input features.

The `HealthySnacksOutput` class provides two features containing the outputs of the model: a dictionary called `labelProbability` and a string called simply `label`. The dictionary contains the names of the classes and the confidence score for each, so it's the same as the probability distribution but in the form of a dictionary instead of an array.

The difference with Vision's array of `VNClassificationObservation` objects is that the dictionary is not sorted. The `label` is simply the name of the class with the highest probability and is provided for convenience.

> **Note**: The names that Xcode generates for these properties depend on the names of the inputs and outputs in the .mlmodel file. For this particular model, the input is called "image" and so the method becomes `prediction(image:)`. If the input were called something else in the .mlmodel file, such as "data," then the method would be `prediction(data:)`. This is also true for the names of the outputs in the `HealthySnacksOutput` class. This is something to be aware of when you're importing a Core ML model: Different models will have different names for the inputs and outputs — something else you don't have to worry about when using Vision.

In order to use the `HealthySnacks` class without Vision, you have to call its `prediction(image:)` method and give it a `CVPixelBuffer` containing the image to classify. When the prediction method is done it returns the classification result as a `HealthySnacksOutput` object.

Next, you'll write this code. Switch to **ViewController.swift** and add the following property to `ViewController` to create an instance of the model:

```
let healthySnacks = HealthySnacks()
```

Now, you need some way to convert the `UIImage` from `UIImagePickerController` into a `CVPixelBuffer`. This object is a low-level description of image data, used by Core Video and AVFoundation. You're probably most used to working with images as `UIImage` or `CGImage` objects, and so you need to convert these to `CVPixelBuffers`, first.

There is no handy API for converting a `UIImage` to a `CVPixelBuffer`. However, in the downloads for this chapter, we've provided a `UIImage` extension that does this. Add the file **UIImage+CVPixelBuffer.swift** to the project.

Change the `classify(image:)` method to the following:

```
func classify(image: UIImage) {
  DispatchQueue.global(qos: .userInitiated).async {
    // 1
    if let pixelBuffer = image.pixelBuffer(width: 227, height: 227) {
      // 2
      if let prediction = try? self.healthySnacks.prediction(
                                                  image: pixelBuffer) {
        // 3
        let results = self.top(1, prediction.labelProbability)
        self.processObservations(results: results)
```

```
        } else {
          self.processObservations(results: [])
        }
      }
    }
  }
```

Here's how this works:

1. Convert the `UIImage` to a `CVPixelBuffer` using the helper method. This scales the image to the given size (227×227) and also fixes the orientation if it's not correct side up yet.

2. Call the `prediction(image:)` method. This can potentially fail (if the image buffer is not 227×227 pixels, for example), which is why you need to use `try?` and put it inside the `if let`.

3. The `prediction` object is an instance of `HealthySnacksOutput`. You can look at its `label` property to find the name of the best class, but you want to look at the names of the best scoring classes as well as their probabilities. That's what the `self.top()` function does.

Because `MLModel`'s prediction method is synchronous, it blocks the current thread until it's done. For this simple image classifier, that may not be a big deal as it's fairly fast, but it's good practice to do the prediction on a background queue anyway.

Xcode now gives errors because the code calls two methods you still need to add. First, add the `top()` method:

```
  func top(_ k: Int, _ prob: [String: Double]) -> [(String, Double)] {
    return Array(prob.map { x in (x.key, x.value) }
                     .sorted(by: { a, b -> Bool in a.1 > b.1 })
                     .prefix(min(k, prob.count)))
  }
```

This looks at the dictionary from `prediction.labelProbability` and returns the `k` best predictions as an array of `(String, Double)` pairs where the string is the label (name of the class) and the `Double` is the probability / confidence score for that class.

Currently you're calling `top(1, …)` because, for the HealthySnacks model, you only care about the highest-scoring class but, for the MultiSnacks model, you might call `top(3, …)` to get the three best results.

Finally, you can put these `(String, Double)` pairs into a string to show in the results label:

```
func processObservations(
        results: [(identifier: String, confidence: Double)]) {
  DispatchQueue.main.async {
    if results.isEmpty {
      self.resultsLabel.text = "nothing found"
    } else if results[0].confidence < 0.8 {
      self.resultsLabel.text = "not sure"
    } else {
      self.resultsLabel.text = String(format: "%@ %.1f%%",
                                      results[0].identifier,
                                      results[0].confidence * 100)
    }
    self.showResultsView()
  }
}
```

This is very similar to what you did in the Vision version of the app but the results are packaged slightly differently.

So this actually wasn't too bad, was it? It may even seem like a little bit less work than what you had to do for Vision. But this is a little misleading. Don't forget that you had to add the **UIImage+CVPixelBuffer.swift** extension to make the conversion. In addition, there are a few important things the pure Core ML version does not do yet, such as color space matching; this translates from the photo's color space, which is often sRGB or P3, to the generic RGB space used by the model.

Challenges

Apple provides a number of Core ML models that you can download for free, from https://developer.apple.com/machine-learning/.

Your challenge for this chapter is to download the SqueezeNet model and add it to the app. This model is very similar to the classifier you implemented in this chapter, which is also based on SqueezeNet. The main difference is that HealthySnacks is trained to classify 20 different snacks into two groups: healthy or unhealthy. The SqueezeNet model from Apple is trained to understand 1,000 classes of different objects (it's a multi-class classifier).

Try to add this new model to the app. It should only take the modification of a single line to make this work — that's how easy it is to integrate Core ML models into your app because they pretty much work all the same.

Chapter 3: Training the Image Classifier

By Audrey Tam

In the previous chapter, you saw how to use a trained model to classify images with Core ML and Vision. However, using other people's models is often not sufficient — the models may not do exactly what you want or you may want to use your own data and categories — and so it's good to know how to train your own models.

Apple provides developers with Create ML as a machine learning framework to create models in Xcode. In this chapter, you'll learn how to train the snacks model using Create ML.

The dataset

Before you can train a model, you need data. You may recall from the introduction that machine learning is all about training a model to learn "rules" by looking at a lot of examples. Since you're building an image classifier, it makes sense that the examples are going to be images.

You're going to train an image classifier that can tell apart 20 different types of snacks. Here are the possible categories, again:

```
Healthy: apple, banana, carrot, grape, juice, orange,
         pineapple, salad, strawberry, watermelon

Unhealthy: cake, candy, cookie, doughnut, hot dog,
           ice cream, muffin, popcorn, pretzel, waffle
```

Download the dataset from https://wolverine.raywenderlich.com/books/mlt/snacks.zip into the **dataset** folder. Unzip this file. It contains the dataset on which you'll train the model.

This dataset has almost 7,000 images — roughly 350 images for each of these categories.

The snacks dataset

The dataset is split into three folders: train, test and val. For training the model, you will use only the 4,800 images from the **train** folder, known as the **training set**.

The images from the **val** and **test** folders (950 each) are used to measure how well the model works once it has been trained. These are known as the **validation set** and the **test set**, respectively. It's important that you don't use images from the validation set or test set for training; otherwise, you won't be able to get a reliable estimate of the quality of your model. We'll talk more about this later in the chapter.

Here are a few examples of training images:

doughnut/0e4d2f7afc5efeee.jpg

ice cream/4472c8cc1e2becc0.jpg

pineapple/856841a5b385c465.jpg

salad/ea2347b86bc95314.jpg

Selected training images

As you can see, the images come in all kinds of shapes and sizes. The name of the folder will be used as the class name — also called the **label** or the **target**.

Create ML

You will now use Create ML to train a multi-class classifier on the snacks dataset.

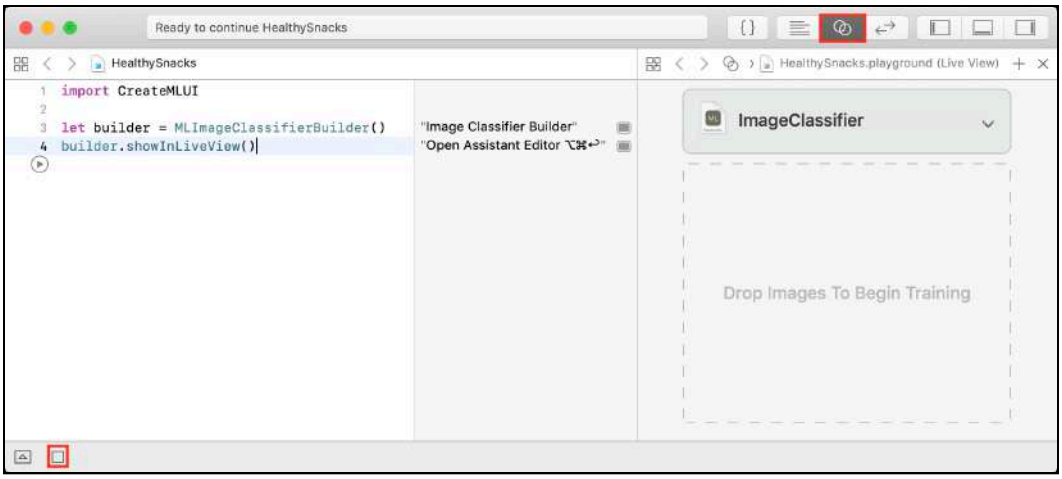
In Xcode, open **starter/MultiSnacks.playground**, and enter this code:

```
import CreateMLUI

let builder = MLImageClassifierBuilder()
builder.showInLiveView()
```

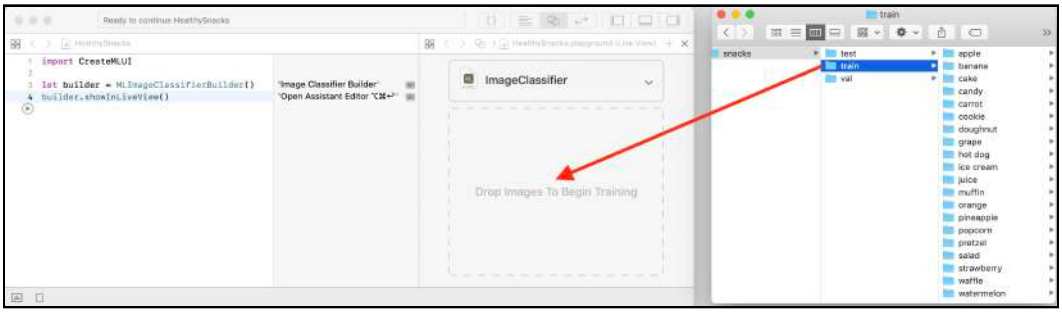
> **Note**: This is a **macOS** playground. `CreateMLUI` doesn't work in an iOS playground. Also note that Create ML requires macOS Mojave (10.14).

Show the assistant editor and click the Run button:

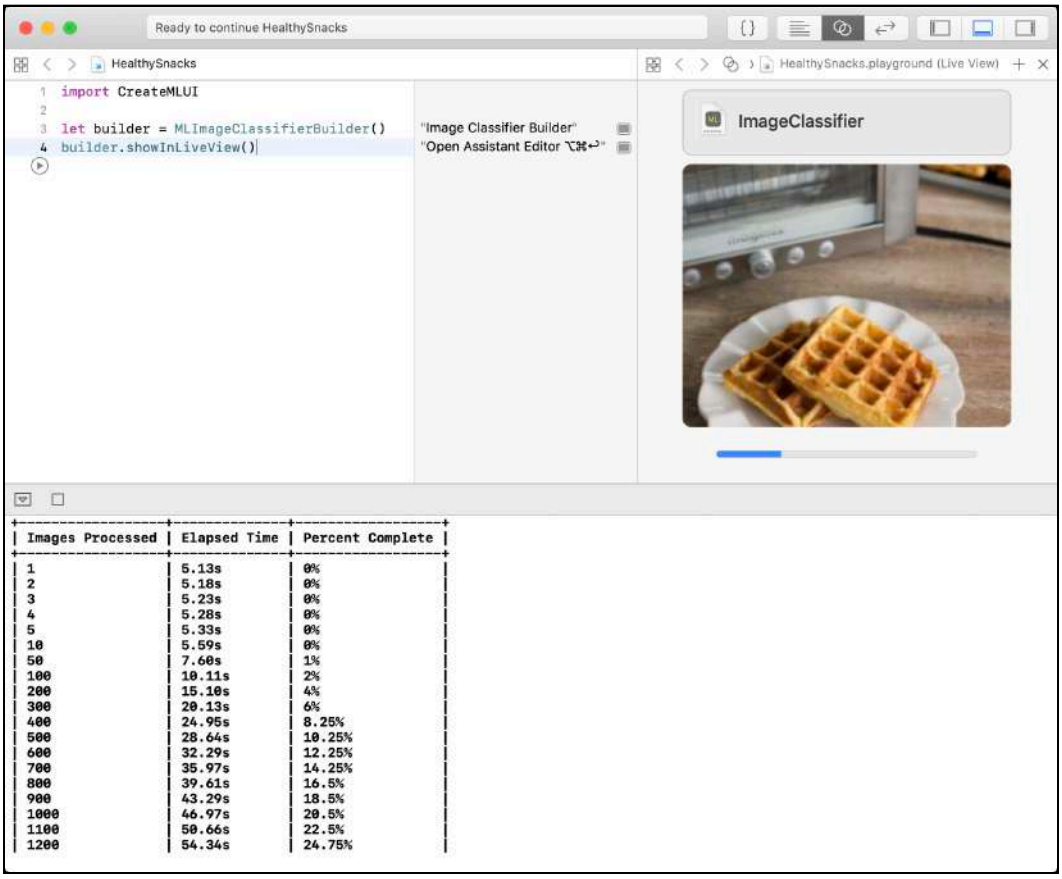


You're creating and showing an interactive view for training and evaluating an image classifier.

Drag the **snacks/train** folder onto the view.



The training process starts immediately. Images load, with a progress bar below. After a short time, a table appears in the debug area, displaying **Images Processed**, **Elapsed Time** and **Percent Complete**:



Note: You may be wondering how Create ML can load *all* of the training data in memory, since 4,800 images may actually take up more physical memory than you have RAM. Create ML loads data into `MLDataTable` structures, which keep only the image metadata in memory, then loads the images themselves on demand. This means that you can have very large `MLDataTables` that contain a lot of data.

Watch some of the images in the interactive view to get a feel for what's there. Each image is in a subdirectory named for the type of food shown in the image, like "apple" or "cookie." Depending on the processors in your Mac, training can take 5–10 minutes or more. While you wait, continue reading about **dataset curation** and **transfer learning**.

How we created the dataset

Collecting good training data can be very time consuming! It's often considered to be the most expensive part of machine learning. Despite — or because of — the wealth of data available on the internet, you'll often need to *curate* your dataset: You must manually go through the data items to remove or clean up bad data or to correct classification errors.

The images in this dataset are from the Google Open Images Dataset, which contains more than 9 million images organized into thousands of categories. The Open Images project doesn't actually host the images — only the URLs to these images. Most of the images come from Flickr and are all licensed under a Creative Commons license (CC BY 2.0). You can find the Open Images Dataset at storage.googleapis.com/openimages/web/index.html

To create the snacks dataset, we first looked through the thousands of classes in Open Images and picked 20 categories of healthy and unhealthy snacks. We then downloaded the **annotations** for all the images. The annotations contain metadata about the images, such as what their class is. Since, in Open Images, the images can contain multiple objects, they may also have multiple classes.

We randomly grabbed a subset of image URLs for each of our 20 chosen classes, and we then downloaded these images using a Python script. Quite a few of the images were no longer available or just not very good, while some were even mislabeled as being from the wrong category, so we went through the downloaded images by hand to clean them up.

Here are some of the things we had to do to clean up the data:

- The pictures in Open Images often contain more than one object, such as an apple and a banana, and we can't use these to train the classifier because the classifier expects only a single label per image. The image must have either an apple *or* a banana, not both; otherwise, it will just confuse the learning algorithm. **We only kept images with just one main object**.
- Sometimes, the lines between categories are a little blurry. Many downloaded images from the cake category were of cupcakes, which are very similar to muffins, so **we removed these ambiguous images from the dataset**. For our purposes, we decided that cupcakes belong to the muffins category, not the cake category.
- **We made sure the selected images were meaningful for the problem you're trying to solve**. The snacks classifier was intended to work on food items you'd

typically find in a home or office. But the banana category had a lot of images of banana trees with stacks of green bananas — that's not the kind of banana we wanted the classifier to recognize.

- Likewise, **we included a variety of images**. We did not only want to include "perfect" pictures that look like they could be found in cookbooks, but photos with a variety of backgrounds, lighting conditions and humans in them (since it's likely that you'll use the app in your home or office, and the pictures you take will have people in them).
- **We threw out images in which the object of interest was very small**, since the neural network resizes the images to 299x299 pixels, a very small object would be just a few pixels in size — too small for the model to learn anything from.

The process of downloading images and curating them was repeated several times, until we had a dataset that gave good results. Simply by improving the data, the accuracy of the model also improved.

When training an image classifier, more images is better. However, we limited the dataset to about 350 images per category to keep the download small and training times short so that using this dataset would be accessible to all our readers. Popular datasets such as ImageNet have 1,000 or more images per category, but they are also hundreds of gigabytes.

The final dataset has 350 images per category, which are split into 250 training images, 50 validation images and 50 test images. Some categories, such as pretzel and popcorn, have fewer images because there simply weren't more suitable ones available in Open Images.

It's not necessary to have exactly the same number of images in each category, but the difference also should not be too great, or the model will be tempted to learn more about one class than another. Such a dataset is called **imbalanced** and you need special techniques to deal with them, which we aren't going to cover, here.

All images were resized so that the smallest side is 256 pixels. This isn't necessary, but it does make the download smaller and training a bit faster. We also stripped the EXIF metadata from images because some of the Python image loaders give warnings on those images, and that's just annoying. The downside of this is that EXIF contains orientation info for the images, so some images may actually appear upside down… Oh well.

Creating the dataset took quite a while, but it's vital. If your dataset is not high quality, then the model won't be able to learn what you want it to learn. As they say, it's not who has the best algorithms, but who has the best data.

> **Note**: You are free to use the images in the snacks dataset as long as you stick to the rules of the CC BY 2.0 license. The filenames have the same IDs as used in the Google Open Images annotations. You can use the included `credits.csv` file to look up the original URLs of where the images came from.

Transfer learning

So what's happening in the playground? Create ML is currently busy training your model using **transfer learning**. As you may recall from the first chapter, transfer learning is a clever way to quickly train models by reusing knowledge from another model that was originally trained on a different task.

The HealthySnacks and MultiSnacks models you used in the previous chapter were built on top of something called SqueezeNet. The underlying model used by Create ML is not SqueezeNet but `VisionFeaturePrint_Screen`, a model that was designed by Apple specifically for getting high-quality results on iOS devices.

`VisionFeaturePrint_Screen` was pre-trained on a ginormous dataset to recognize an enormous number of classes. It did this by learning what **features** to look for in an image and by learning how to combine these features to classify the image. Almost all of the training time for your dataset is the time the model spends extracting 2,048 features from your images. These include low-level edges, mid-level shapes and task-specific high-level features.

Once the features have been extracted, Create ML spends only a relatively tiny amount of time training a **logistic regression** model to separate your images into 20 classes. It's similar to fitting a straight line to scattered points, but in 2,048 dimensions instead of two.

Transfer learning only works successfully when features of your dataset are reasonably similar to features of the dataset that was used to train the model. A model pre-trained on ImageNet — a large collection of photos — might not transfer well to pencil drawings or microscopy images.

Transfer learning takes less time than training a neural network from scratch. Here's what's involved in training a neural network from scratch:

1. **Initialize the neural network's "brain" with small random numbers**. This is why an untrained model just makes random guesses: its knowledge literally is random.

2. **Let the neural network make predictions for all the training images**.

3. **Compare the predictions to the known answers — the "targets."** When you're training a classifier, the targets are the class labels for the training images. This is used to compute the **loss**, a measure of how far off the predictions are from the expected answers. The loss is a multi-dimensional function that has a minimum value, and the goal of training is to determine the **weights**, or **learned parameters**, that get very close to this minimum.

4. To compute weights that reduce the error, you **calculate the gradient of the loss function at the current graph location, then adjust the weights to "step down" the slope**. This is called **gradient descent**, and happens many times during a training session. For large datasets, using all the data to calculate the gradient takes a long time. **Stochastic gradient descent** (SGD) estimates the gradient from randomly selected **mini-batches** of training data — like taking a survey of voters ahead of election day: If your sample is representative of the whole dataset, then the survey results accurately predict the final results. This nudges the model's knowledge a little bit in the right direction so that, next time, it will make slightly more correct predictions for the training images.

5. Go to step two to **repeat this process hundreds of times for all the images in the training set**. With each training step, the model becomes a tiny bit better: The brain's learned parameters change from random numbers into something that is more suitable to the task that you want it to learn. Over time, the model learns to make better and better predictions.

Stochastic gradient descent is a rather brute-force approach, but it's the only method that is practical for training deep neural networks. Unfortunately, it's also rather slow. To make SGD work reliably, you can only make small adjustments at a time to the learned parameters, so it takes a lot of training steps (hundreds of thousands or more) to make the model learn anything.

Create ML uses a smarter approach. Instead of starting with an untrained model that has only random numbers in its brain, Create ML takes a neural network that has already been successfully trained on a large dataset, and then it fine-tunes it on your own data. This involves training only a small portion of the model instead of the whole thing.

This shortcut is called **transfer learning** because you're transferring the knowledge that the neural network has learned on some other type of problem to your specific task. It's a lot faster than training a model from scratch, and it can work just as well. It's the machine learning equivalent of "work smarter, not harder!"

The pre-trained `VisionFeaturePrint_Screen` model that Create ML uses has seen lots of photos of food and drinks, so it already has a lot of knowledge about the kinds of images that you'll be using it with. Using transfer learning, you can take advantage of this existing knowledge.

> **Note**: When you use transfer learning, you need to choose a *base model* that you'll use for feature extraction. The two base models that you've seen so far are SqueezeNet and `VisionFeaturePrint_Screen`. Turi Create analyzes your training data to select the most suitable base model. Currently, Create ML's image classifier always uses the `VisionFeaturePrint_Screen` base model. It's large, with 2,048 features, so feature extraction takes a while. The good news is that `VisionFeaturePrint_Screen` is part of iOS 12 and the Vision framework, so models built on this are tiny — kilobytes instead of megabytes, because they do not need to include the base model. The bad news is that models trained with Create ML will not work on iOS 11, or on other platforms such as Android.

Since this pre-trained base model doesn't yet know about our 20 specific types of snacks, you cannot plug it directly into the snack detector app, but you *can* use it as a **feature extractor**.

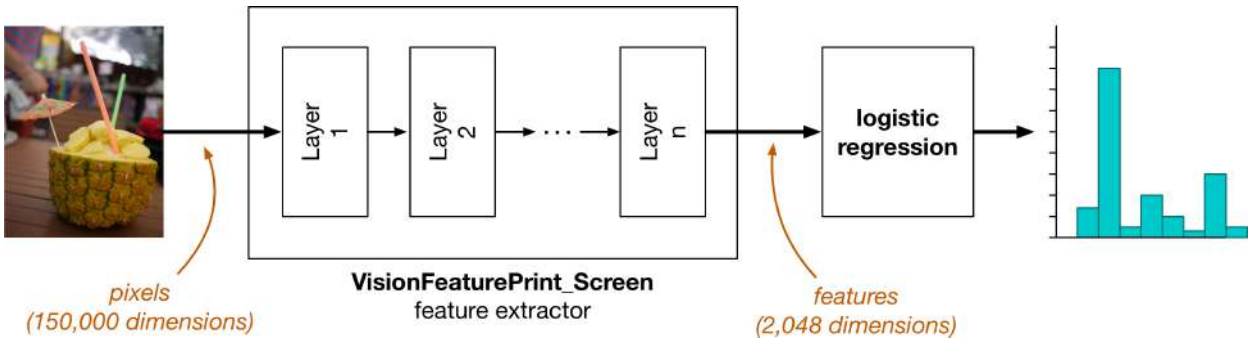
What is feature extraction?

You may recall that machine learning happens on "features," on which we've defined a feature to be any kind of data item that we find interesting. You could use the photo's pixels as features but, as the previous chapter demonstrated, the individual RGB values don't say much about what sort of objects are in the image.

`VisionFeaturePrint_Screen` transforms the pixel features, which are hard to understand, into features that are much more descriptive of the objects in the images. This is the pipeline that we've talked about before. Here, however, the output of

`VisionFeaturePrint_Screen` is not a probability distribution that says how likely it is that the image contains an object of each class.

`VisionFeaturePrint_Screen`'s output is, well, more features.



VisionFeaturePrint_Screen extracts features

For each input image, `VisionFeaturePrint_Screen` produces a list of 2,048 numbers. These numbers represent the content of the image at a high level. Exactly what these numbers mean isn't always easy to describe in words, but think of each image as now being a point in this new 2,048-dimensional space, wherein images with similar properties are grouped together.

For example, one of those 2,048 numbers could represent the color of a snack, and oranges and carrots would have very similar values in that dimension. Another feature could represent how elongated an object is, and bananas, carrots and hot dogs would have larger values than oranges and apples. On the other hand, apples, oranges and doughnuts would score higher in the dimension for how "round" the snack is, while waffles would score lower in that dimension (assuming a negative value for that feature means squareness instead of roundness).

Models usually aren't that interpretable, though, but you get the idea: These new 2,048 features describe the objects in the images by their true characteristics, which are much more informative than pixel intensities. However, you cannot simply draw a line (sorry, a *hyperplane*) through this 2,048-dimensional space to separate the images into the different classes we're looking for, because `VisionFeaturePrint_Screen` is not trained on our own dataset. It was trained on ImageNet, which has 1000 classes, not 20.

While `VisionFeaturePrint_Screen` does a good job of creating more useful features from the training images, in order to be able to classify these images we need to transform the data one more time into the 20-dimensional space that we can interpret as the probability distribution over our 20 types of snacks.

How do we do that? Well, Create ML uses these 2,048 numbers as the input to a new machine learning model called **logistic regression**.

Instead of training a big, hairy model on images that have 150,000 features (difficult!), Create ML just trains a much simpler model on top of the 2,048 features that `VisionFeaturePrint_Screen` has extracted.

Logistic regression

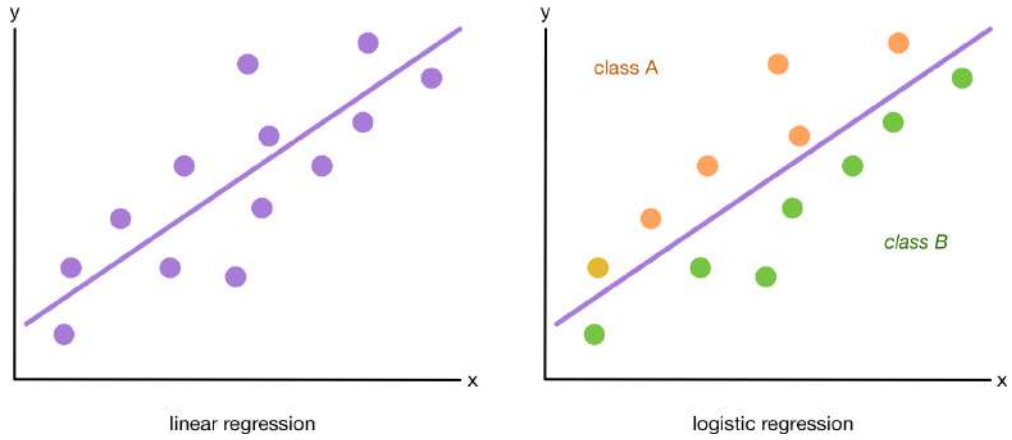
By the time you're done reading the previous section, Create ML has (hopefully) finished training your model. The tables show 3m 9s for feature extraction and 6.6s to train and calibrate the solver.

> **Note**: I'm running Create ML on a 2018 MacBook Pro with a 6-core 2.7GHz CPU and Radeon 560 GPU. You'll probably get slightly different training results than what are shown in this book. Recall that untrained models, in this case the logistic regression, are initialized with random numbers. This can cause variations between different training runs.

```
Extracting image features from full data set.
Analyzing and extracting image features.
+------------------+--------------+------------------+
| Images Processed | Elapsed Time | Percent Complete |
+------------------+--------------+------------------+
| 1                | 5.13s        | 0%               |
| 2                | 5.18s        | 0%               |
| 3                | 5.23s        | 0%               |
| 4                | 5.28s        | 0%               |
| 5                | 5.33s        | 0%               |
...
| 4500             | 2m 57s       | 93%              |
| 4600             | 3m 1s        | 95%              |
| 4700             | 3m 4s        | 97%              |
| 4800             | 3m 8s        | 99%              |
| 4838             | 3m 9s        | 100%             |
+------------------+--------------+------------------+
Automatically generating validation set from 5% of the data.
Beginning model training on processed features.
Calibrating solver; this may take some time.
+-----------+--------------+-------------------+---------------------+
| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy |
+-----------+--------------+-------------------+---------------------+
| 1         | 1.108525     | 0.714441          | 0.675000            |
| 2         | 2.353929     | 0.798173          | 0.783333            |
| 3         | 2.916080     | 0.813180          | 0.787500            |
| 4         | 3.447996     | 0.830579          | 0.791667            |
| 5         | 3.988087     | 0.850152          | 0.829167            |
| 10        | 6.638566     | 0.911918          | 0.891667            |
+-----------+--------------+-------------------+---------------------+
```

The solver that Create ML trains is a classifier called **logistic regression**. This is an old-school machine learning algorithm but it's still extremely useful. It's arguably the most common ML model in use today.

You may be familiar with another type of regression called **linear regression**. This is the act of fitting a line through points on a graph, something you may have done in high school where it was likely called the method of (ordinary) least squares.



Linear and logistic regression

Logistic regression does the same thing, but says: All points on one side of the line belong to class A, and all points on the other side of the line belong to class B. See how we're literally drawing a line through space to separate it into two classes?

Because books cannot have 2,048-dimensional illustrations, the logistic regression in the above illustration is necessarily two-dimensional, but the algorithm works the same way, regardless of how many dimensions the input data has. Instead of a line, the decision boundary is a high-dimensional object that we've mentioned before: a **hyperplane**. Create ML also uses a variation of the algorithm — **multinomial logistic regression** — that handles more than two classes.

Training the logistic regression algorithm to transform the 2,048-dimensional features into a space that allows us to draw separating lines between the 20 classes is fairly easy because we already start with features that say something meaningful about the image, rather than raw pixel values.

> **Note**: If you're wondering about exactly how logistic regression works, as usual it involves a transformation of some kind. The logistic regression model tries to find "coefficients" (its learned parameters) for the line so that a point that belongs to class A is transformed into a (large) negative value, and a point from class B is transformed into a (large) positive value. Whether a transformed point is positive

or negative then determines its class. The more ambiguous a point is, i.e. the closer the point is to the decision boundary, the closer its transformed value is to 0. The multinomial version extends this to allow for more than two classes. We'll go into the math in a later chapter. For now, it suffices to understand that this algorithm finds a straight line that separates the points belonging to different classes in the best way possible.

Looking for validation

Even though there are 4,838 images in the snacks/train dataset, Create ML uses only 95% of them for training. In the output, after feature extraction, there's this message:

```
Automatically generating validation set from 5% of the data.
```

During training, it's useful to periodically check how well the model is doing. For this purpose, Create ML sets aside a small portion of the training examples — 5%, so about 240 images. It doesn't train the logistic classifier on these images, but only uses them to evaluate how well the model does (and possibly to tune certain settings of the logistic regression).

This is why the output has one column for **training accuracy** and one for **validation accuracy**:

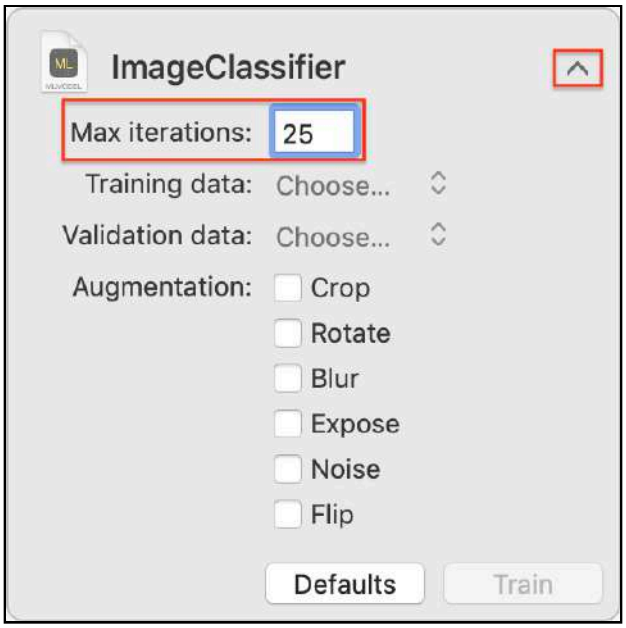
```
+-----------+--------------+-------------------+---------------------+
| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy |
+-----------+--------------+-------------------+---------------------+
| 1         | 1.108525     | 0.714441          | 0.675000            |
| 2         | 2.353929     | 0.798173          | 0.783333            |
| 3         | 2.916080     | 0.813180          | 0.787500            |
| 4         | 3.447996     | 0.830579          | 0.791667            |
| 5         | 3.988087     | 0.850152          | 0.829167            |
| 10        | 6.638566     | 0.911918          | 0.891667            |
+-----------+--------------+-------------------+---------------------+
Completed (Iteration limit reached).
```

After 10 iterations, training accuracy was 0.91 or 91%, meaning that out of 100 training examples it correctly predicts the class for 91 of them. In other words, the classifier is correct for 9 out of 10 images. The validation accuracy is similar: 0.89, close enough to 9 out of 10 images correct.

> **Note**: What Create ML calls an iteration is one pass through the entire training set, also known as an "epoch". This means Create ML has given the model a chance to look at all 4,582 training images once (or rather, the extracted feature vectors of all training images). If you do 10 iterations, the model will have seen each training image (or its feature vectors) 10 times.

By default, Create ML trains the logistic regression for up to 10 iterations, but you can change this with the **Max iterations** setting. In general, the more iterations, the better the model, but also the longer you have to wait for training to finish. But training the logistic regression doesn't take long, so no problem!

The problem with this is doing the feature extraction all over again! If your Mac did the feature extraction in five minutes or less, go ahead and try this:



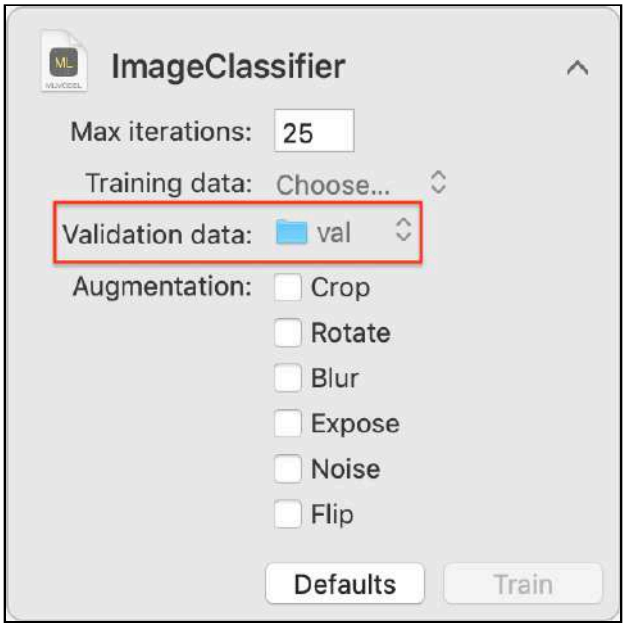
1. Stop the playground; then run it again.
2. Click the disclosure arrow to show the options.
3. Increase **Max iterations** to 25.
4. Drag **snacks/train** onto the view to start feature extraction.

Here's what happened on my Mac:

```
...
Automatically generating validation set from 5% of the data.
Beginning model training on processed features.
Calibrating solver; this may take some time.
+-----------+--------------+-------------------+---------------------+
| Iteration | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+--------------+-------------------+---------------------+
| 1         | 1.009019     | 0.739452          | 0.754167            |
...
| 5         | 3.645678     | 0.845368          | 0.870833            |
| 10        | 6.054766     | 0.908221          | 0.891667            |
| 15        | 8.408861     | 0.953458          | 0.904167            |
| 20        | 10.796781    | 0.965855          | 0.920833            |
| 25        | 13.168258    | 0.984776          | 0.916667            |
+-----------+--------------+-------------------+---------------------+
Completed (Iteration limit reached).
```

Huh? Training accuracy improved, but validation accuracy stayed pretty much the same! Maybe 240 validation images isn't enough? Remember, there's a snacks/val subdirectory, with about 50 images per class, so try using these, instead of letting Create ML choose 5% at random.

Stop and restart the playground. Open the options to increase **Max iterations** to 25, and set **Validation data** to **snacks/val** (double-click on **Choose...** to open a file selection dialog):



Here's what I got:

```
+-----------+--------------+-------------------+----------------------+
| Iteration | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+--------------+-------------------+----------------------+
| 1         | 1.073211     | 0.743076          | 0.720419             |
| 2         | 2.332538     | 0.788136          | 0.757068             |
| 3         | 2.866700     | 0.812112          | 0.786387             |
| 4         | 3.426697     | 0.831749          | 0.801047             |
| 5         | 3.964355     | 0.847458          | 0.825131             |
| 10        | 6.610393     | 0.907400          | 0.889005             |
| 15        | 9.270788     | 0.950186          | 0.915183             |
| 20        | 11.920261    | 0.964035          | 0.913089             |
| 25        | 14.616540    | 0.983258          | 0.919372             |
+-----------+--------------+-------------------+----------------------+
```

This is with 955 validation images, not 240. And the training used all 4,838 training images. But validation accuracy still doesn't improve, indicating the model may be **overfitting**.

Overfitting happens

Overfitting is a term you hear a lot in machine learning. It means that the model has started to remember *specific* training images. For example, the image train/ice cream/b0fff2ec6c49c718.jpg has a person in a blue shirt enjoying a sundae:

Yummy!

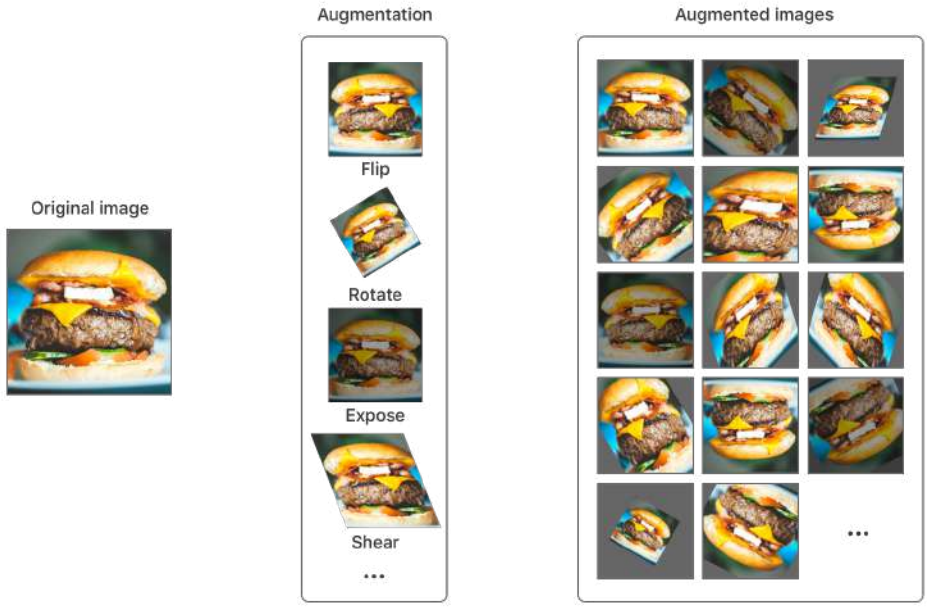
Suppose a classifier learns a rule that says, "If there is a big blue blob in the image, then the class is ice cream." That's obviously not what you want the model to learn, as the blue shirt does not have anything to do with ice cream in general. It just happens to work for this particular training image.

This is why you use a validation set of images that are not used for training. Since the model has never seen these images before, it hasn't learned anything about them, making this is a good test of how well the trained model can generalize to new images.

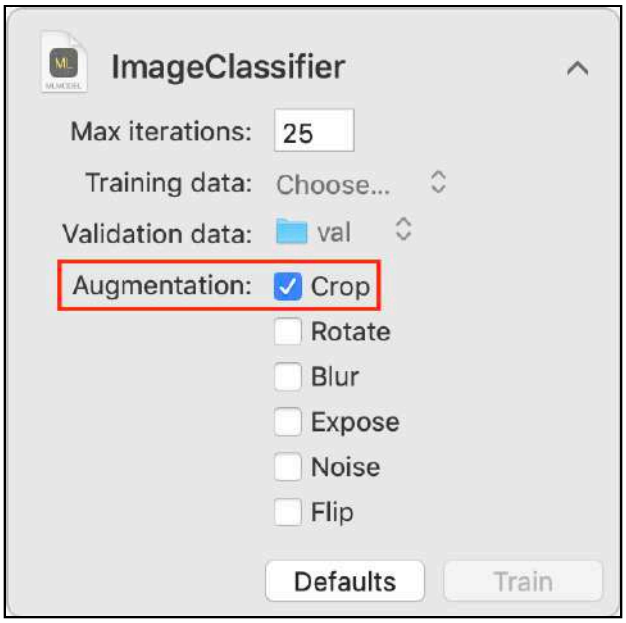
So the *true* accuracy of this particular model is 92% correct (the validation accuracy), not 98% (the training accuracy). If you only look at the training accuracy, your estimate of the model's accuracy can be too optimistic. Ideally, you want the validation accuracy to be similar to the training accuracy, as it was after only 10 iterations — that means that your model is doing a good job.

Typically, the validation accuracy is a bit lower, but if the gap between them is too big, it means the model has been learning too many irrelevant details from the training set — in effect, memorizing what the result should be for specific training images. Overfitting is one of the most prevalent issues with training deep learning models.

There are several ways to deal with overfitting. The best strategy is to train with more data. Unfortunately, for this book we can't really make you download a 100GB dataset, and so we've decided to make do with a relatively small dataset. For image classifiers, you can **augment** your image data by flipping, rotating, shearing or changing the exposure of images. Here's an illustration of data augmentation:



So try it: restart the playground, set the options as before, but also select some data augmentation:



Note: The window lists the augmentation options from greatest to least training effectiveness, so Crop is the most effective choice. This is also the order in which the classifier applies options, if you select more than one. Selecting **Crop** creates four flipped images for each original image, so feature extraction takes almost five times longer (12m 47s, for me). Selecting all six augmentation options creates 100 augmented images for each original! Actually, "only 100," because the number of augmentation images is capped at 100.

Here's what I got:

```
+-----------+--------------+-------------------+----------------------+
| Iteration | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+--------------+-------------------+----------------------+
...
| 5         | 20.242937    | 0.834436          | 0.827225             |
...
| 10        | 33.283525    | 0.895246          | 0.892147             |
| 11        | 35.874542    | 0.899173          | 0.891099             |
| 12        | 38.447442    | 0.903555          | 0.897382             |
| 13        | 41.025082    | 0.916164          | 0.901571             |
| 14        | 43.655974    | 0.925878          | 0.912042             |
| 15        | 46.174075    | 0.928648          | 0.913089             |
| 16        | 48.803652    | 0.932038          | 0.914136             |
| 17        | 51.366629    | 0.934270          | 0.916230             |
| 18        | 54.006646    | 0.937205          | 0.912042             |
| 19        | 56.663583    | 0.942538          | 0.913089             |
| 20        | 59.219079    | 0.944688          | 0.910995             |
```

```
| 21        | 61.769311    | 0.946755          | 0.914136            |
| 22        | 64.411917    | 0.948698          | 0.912042            |
| 23        | 67.025052    | 0.952005          | 0.912042            |
| 24        | 69.681690    | 0.955147          | 0.916230            |
| 25        | 72.288230    | 0.957131          | 0.912042            |
+-----------+--------------+-------------------+---------------------+
```

Well, training accuracy is a little lower, but validation accuracy is stuck at 91–92%. So it's still overfitting, but it's also taking longer to learn — that's understandable, with almost 25,000 images!

Note: Xcode 10 beta 6 has problems with the blur option for every image: `Failed to render 89401 pixels because a CIKernel's ROI function did not allow tiling.`

Another trick is adding **regularization** to the model — something that penalizes large weights, because a model that gives a lot of weight to a few features is more likely to be overfitting. Create ML doesn't let you do regularization, so you'll learn more about it in Chapter 5: "Digging Deeper into Turi Create."

The takeaway is that training accuracy is a useful metric, but it only says something about whether the model is still learning new things, not about how well the model works in practice. A good score on the training images isn't really that interesting, since we already know what their classes are, after all.

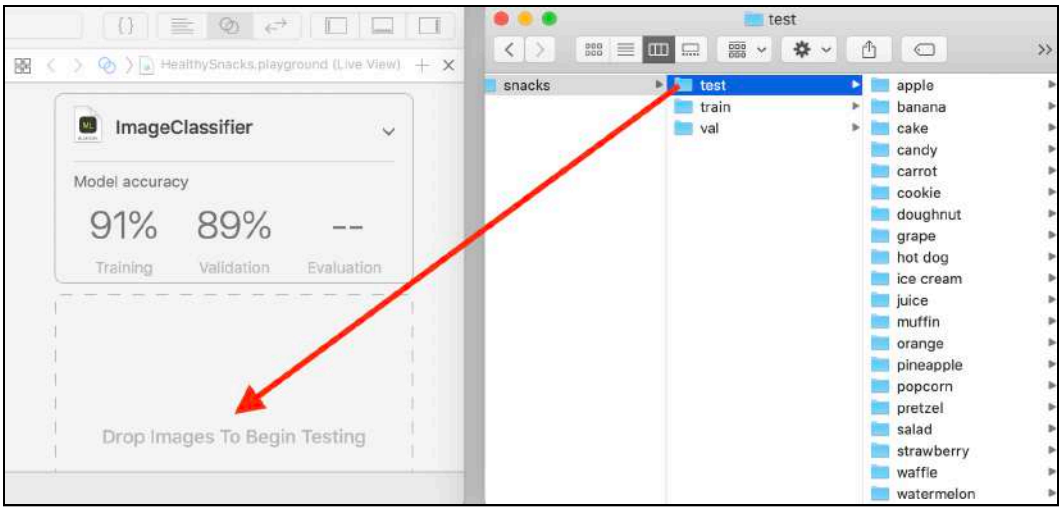
What we care about is how well the model works on images that it has never seen before. Therefore, the metric to keep your eye on is the validation accuracy, as this is a good indicator of the performance of the model in the real world.

Note: By the way, overfitting isn't the only reason why the validation accuracy can be lower than the training accuracy. If your training images are different from your validation images in some fundamental way (silly example: all your training images are photos taken at night while the validation images are taken during the day), then your model obviously won't get a good validation score. It doesn't really matter where your validation images come from, as long as these images were not used for training but they do contain the same kinds of objects. This is why Create ML randomly picks 5% of the training images to use for validation.

More metrics and the test set

Now that you've trained the model, it's good to know how well it does on new images that it has never seen before. You already got a little taste of that from the validation accuracy during training, but the dataset also comes with a collection of test images that you haven't used yet. These are stored in the **snacks/test** folder, and are organized by class name, just like the training data.

Drag the **snacks/test** folder onto the view to evaluate this model:



This takes a few moments to compute. Just like during training, the feature extraction on the images takes more time than classification.

```
Extracting image features from evaluation data.
Analyzing and extracting image features.
+------------------+--------------+------------------+
| Images Processed | Elapsed Time | Percent Complete |
+------------------+--------------+------------------+
| 1                | 62.735ms     | 0%               |
| 2                | 102.807ms    | 0%               |
...
| 952              | 29.13s       | 100%             |
| 951              | 29.09s       | 99.75%           |
+------------------+--------------+------------------+
----------------------------------
Number of examples: 952
Number of classes: 20
Accuracy: 90.97%
```

91% accuracy! This is consistent with validation accuracy.

Note: You may be wondering what the difference is between the validation set that's used during training and the test set you're using now. They are both used to find out how well the trained model does on new images.

However, the validation set is often also used to tune the settings of the learning algorithm — the so-called **hyperparameters**. Because of this, the model does get influenced by the images in the validation set, even though the training process never looks at these images directly. So while the validation score gives a better idea of the true performance of the model than the training accuracy, it is still not completely impartial.

That's why it's a good idea to reserve a separate test set. Ideally, you'd evaluate your model *only once* on this test set, when you're completely done training it. You should resist the temptation to go back and tweak your model to improve the test set performance, as now the test set images start to influence how the model is trained and so the test set is no longer a good representation of unseen images. It's probably fine if you do this a few times, especially if your model isn't very good yet, but it shouldn't become a habit. Save the test set for last and evaluate on it as few times as possible.

Again, the question: Is 91% correct a good score or a bad score? It means every one out of 10 images is scored wrong. Obviously, we'd rather see an accuracy score of 99% or better (only one out of 100 images wrong), but whether that's feasible depends on the capacity of your model, the number of classes and how many training images you have.

Even if 91% correct is not ideal, it does mean your model has actually learned something. After all, with 20 classes, a totally random guess would only be correct 1/20 times on average or 5%. So the model is already doing much better than random guesses. But it looks like Create ML isn't going to do any better than this with the current dataset.

Keep in mind, the accuracy score only looks at the top predicted class with the highest probability. If a picture contains an apple, and the most confident prediction is "hot dog," then it's obviously a wrong answer. But if the top prediction is "hot dog" with 40% confidence, while the second highest class is "apple" with 39% confidence, then you might still consider this a correct prediction.

Precision and recall

Below, the accuracy value is the **confusion matrix**. You'll look at that in Chapter 5, "Digging Deeper Into Turi Create," when you create a nifty visualization of the matrix.

Below the confusion matrix are two other useful metrics for classifiers — **precision** and **recall**:

```
******PRECISION RECALL******
---------------------------------
Class       Precision(%)   Recall(%)
apple       91.49          86.00
banana      96.15          100.00
cake        76.47          78.00
candy       88.00          88.00
carrot      90.20          92.00
cookie      88.64          78.00
doughnut    92.16          94.00
grape       97.92          94.00
hot dog     97.83          90.00
ice cream   88.00          88.00
juice       98.04          100.00
muffin      83.64          95.83
orange      89.80          88.00
pineapple   86.36          95.00
popcorn     100.00         97.50
pretzel     92.00          92.00
salad       87.27          96.00
strawberry  87.23          83.67
waffle      93.88          92.00
watermelon  97.92          94.00
```

Create ML computes precision and recall for each individual class, which is useful for understanding which classes perform better than others. These values are mostly above 80%. But what do they mean?

Precision means: Of all the images that the model predicts to be "apple", 91.49% of them are actually apples. Precision is high if we don't have many **false positives**, and it is low if we often misclassify something as being "apple." A false positive is if something isn't an X, but the model thinks it is.

Recall is similar, but different: it counts how many apples the model found among the total number of apples — in this case, it found 86% of the "apple" images. This gives us an idea of the number of **false negatives**. If recall is low, it means we actually missed a lot of the apples. A false negative is if something *is* an X, but the model doesn't think it is.

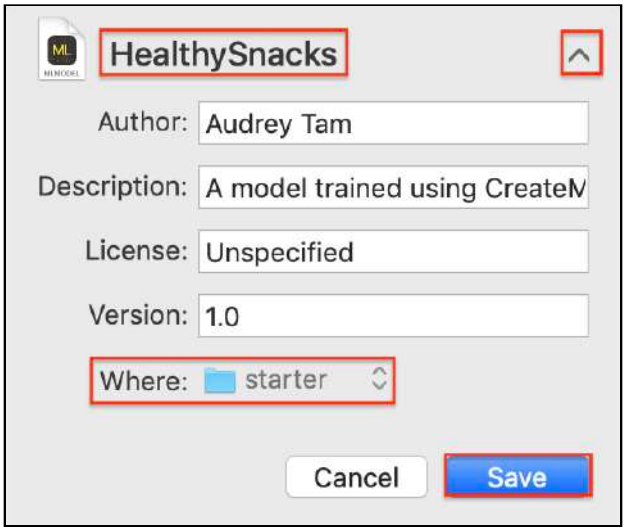
Again, we like to see high numbers here, and > 80% is reasonable. For precision, it means that about one out of 10 things the model claimed were "cake" really aren't, and, for recall, it means that the model found 81.48% of the "cake" images in the test set.

For precision, the worst performing classes are "cake" (76%) and "muffin" (84%). For recall, the worst are "cake" (78%) and "cookie" (78%). These would be the classes to pay attention to, in order to improve the model — for example, by gathering more or better training images for these classes.

Exporting to Core ML

The whole point of training your own models is to use them in your apps, so let's save this new model so you can load it with Core ML.

Click the disclosure symbol next to **ImageClassifier** to see a different set of options. Click on the text, and change it to **MultiSnacks**. Change the **Where** location to the **starter** folder, then click **Save**:



A Core ML model normally combines the feature extractor with the logistic regression classifier into a single model. We still need the feature extractor because, for any new images you want to classify, you also need to compute their feature vectors. With Core ML, the logistic regression is simply another layer in the neural network, i.e., another stage in the pipeline.

However, a `VisionFeaturePrint_Screen`-based Core ML model doesn't need to include the feature extractor, because it's part of iOS 12. So the Core ML model is basically just the logistic regression classifier, and quite small!

Once you've saved the `mlmodel` file, add it to Xcode in the usual way. Simply replace the existing .mlmodel file with your new one. The model that's currently in the Snacks app was created with SqueezeNet as the feature extractor — it's **5MB**. Your new model from

Create ML is only 312**KB**! It's actually a much bigger model, but most of it — the `VisionFeaturePrint_Screen` feature extractor — is already included in the operating system, and so you don't need to distribute that as part of the .mlmodel file.

> **Note**: There are a few other differences between these two feature extractors. SqueezeNet is a relatively small pre-trained model that expects 227×227 images, and extracts 1,000 features. `VisionFeaturePrint_Screen` uses 299×299 images, and extracts 2,048 features. So the kind of knowledge that is extracted from the image by the Vision model is much richer, which is why the model you just trained with Create ML actually performs better than the SqueezeNet-based model from the previous chapter, which only has a 67% validation accuracy!

Classifying on live video

The example project in this chapter's resources is a little different than the app you worked with in the previous chapter. It works on live video from the camera. The `VideoCapture` class uses `AVCaptureSession` to read video frames from the iPhone's camera at 30 frames per second. The `ViewController` acts as the delegate for this `VideoCapture` class and is called with a `CVPixelBuffer` object 30 times per second. It uses Vision to make a prediction and then shows this on the screen in a label.

The code is mostly the same as in the previous app, except now there's no longer a `UIImagePicker` but the app runs the classifier continuously.

The classifier on live video

There is also an `FPSCounter` object that counts how many frames per second the app achieves. With a model that uses `VisionFeaturePrint_Screen` as the feature extractor you should be able to get 30 FPS on a modern device.

> **Note**: The app has a setting for `videoCapture.frameInterval` that lets you run the classifier less often, in order to save battery power. Experiment with this setting, and watch the energy usage screen in Xcode for the difference this makes.

The `VideoCapture` class is just a bare bones example of how to read video from the camera. We kept it simple on purpose so as not to make the example app too complicated. For real-word production usage you'll need to make this more robust, so it can handle interruptions, and use more camera features such as auto-focus, the front-facing camera, and so on.

Recap

In this chapter, you got a taste of training your own Core ML model with Create ML. Partly due to the limited dataset, the default settings got only ~90% accuracy. Increasing max iterations increased training accuracy, but validation accuracy was stuck at ~90%, indicating that overfitting might be happening. Augmenting the data with flipped images reduced the gap between training and validation accuracies, but you'll need more iterations to increase the accuracies.

More images is better. We use 4,800 images, but 48,000 would have been better, and 4.8 million would have been even better. However, there is a real cost associated with finding and annotating training images, and for most projects a few hundred images or at most a few thousand images per class may be all you can afford. Use what you've got — you can always retrain the model at a later date once you've collected more training data. Data is king in machine learning, and who has the most of it usually ends up with a better model.

Create ML is super easy to use, but lets you tweak only a few aspects of the training process. It's also currently limited to image and text classification models.

Turi Create gives you more task-focused models to customize, and lets you get more hands-on with the training process. It's almost as easy to use as Create ML, but you need to write Python. The next chapter gets you started with some useful tools, so you can train a model with Turi Create. Then, in Chapter 5, "Digging Deeper Into Turi Create," you'll get a closer look at the training process, and learn more about the building blocks of neural networks.

Key points

- You can use macOS playgrounds to test out Create ML, and tune parameters to create simple machine learning models.
- Create ML allows you to create small models that leverage built-in models already installed on iOS 12 devices.
- Ideally, you want the validation accuracy to be similar to the training accuracy.
- There are several ways to deal with overfitting; include more images, increase training iterations or augment your data.
- Precision and recall are useful metrics when evaluating your model.

Challenge

Create your own dataset of labelled images, and use Create ML to train a model.

A fun dataset is the Kaggle dogs vs. cats competition www.kaggle.com/c/dogs-vs-cats, which lets you train a binary classifier that can tell apart dogs from cats. The best models score about 98% accuracy — how close can you get with Create ML?

Also check out some of Kaggle's other image datasets at www.kaggle.com/datasets.

Of course, don't forget to put your own model into the iOS app to impress your friends and co-workers!

Chapter 4: Getting Started With Python & Turi Create

By Audrey Tam & Matthijs Hollemans

Congratulations! If you've made it this far, you've developed a strong foundation for absorbing machine learning material. However, before we can move forward, we need to address the 10,000 pound snake in the room... Python. Until this point, you've made due with Xcode, and Swift, however, if you're going to get serious about Machine Learning, then it's best you prepare yourself to learn some Python. In this chapter,

- You'll learn how to set up and use tools from the Python ecosystem for data science and machine learning (ML).
- You'll install **Anaconda** and use its Navigator GUI.
- You'll use terminal commands to create ML environments which you'll use throughout this book.
- Finally, you'll use **Jupyter notebooks**, which is similar to Swift Playgrounds, to explore the Python language, data science libraries, and Apple's ML-as-a-Service; **Turi Create**.

Starter folder

The starter folder for this chapter contains:

- **notebook** folder: the sample notebook data files.
- **.yaml** files: used to import pre-configured environments, if you want to skip the instructions for configuring the environments yourself.

Python

Python is the dominant programming language used for data science and machine learning. As such, there's a myriad of tools available for, but not limited to the Python community to support data science and machine learning development. Some of which include:

- **Data science libraries**: Matplotlib, NumPy, Pandas, SciPy
- **Machine learning libraries**: Caffe2, Keras, Microsoft Cognitive Toolkit, TensorFlow, Theano, scikit-learnTheano
- **ML-as-a-Service**: Amazon Machine Learning, Google ML Kit, IBM Watson, Microsoft Azure Machine Learning Studio, Turi Create
- **Tools**: `coremltools`, `virtualenv`, `pip`, Anaconda, Docker, Jupyter, Google Colaboratory

If you know the Swift programming language, you'll find that although Python is quite different, it also shares some similarities with Swift. For instance:

- You `import` modules similarly to Swift modules.
- It has the similar concepts for primitive types, tuples, lists, dictionaries, operators, loops, conditionals.
- You can create objects, classes, functions.

Of course, there are some differences too. For example:

- Python is interpreted, not compiled.
- You define closures, functions, classes with indentation instead of `{ ... }`.
- Naming conventions tend toward terse abbreviations, like early C programming.
- Module and function names are *snake_case*, while class and exception names are *PascalCase*.
- Comments (docstrings) start with `#` not `//`.
- Multi-line comments begin and end with """ instead of `/*` and `*/`, and the end """ is on its own line.
- `True`/`False`, not `true`/`false`.
- Dynamic types, no support for constants; no `let` or `var`.

- Enumerations, but no `switch`.

After you set up the tools, you'll try out some Python while learning about the libraries. If you'd like some more practice or information, here are two helpful resources:

- Michael Kennedy's November 2014 Comparison of Python and Swift Syntax
- Jason Brownlee's May 2016 Crash Course in Python for Machine Learning Developers: includes NumPy, Matplotlib and Pandas examples.

Packages and environments

A version of Python is already installed on **macOS**. However, using this installation may cause version conflicts because people use both Python 2.7 and Python 3.x, which are incompatible branches of the same language. To further complicate things, working on machine learning projects requires integrating the correct versions of numerous packages.

> **Note**: The Python development team will stop supporting Python 2.7 in 2020, so the major open source Python packages have pledged to drop support for Python 2.7 no later than 2020.

Most people create environments where they install specific versions of Python and the packages they need. The most basic tool set includes the environment manager `virtualenv` and the package manager `pip`. Aside from that, you still have to figure out which packages you need — and it's all very manual, with a high probability of frustration.

There *is* a better way!

Conda

The data science community developed **Conda** to make life easier. Conda handles Python language versions, Python packages, and associated native libraries. It's both an environment manager and a package manager. If you need a package that it doesn't know about, you can use `pip` within a `conda` environment to grab the package.

Conda comes in two distributions:

- **Miniconda**: Includes only the packages needed to run Conda. (400 MB)

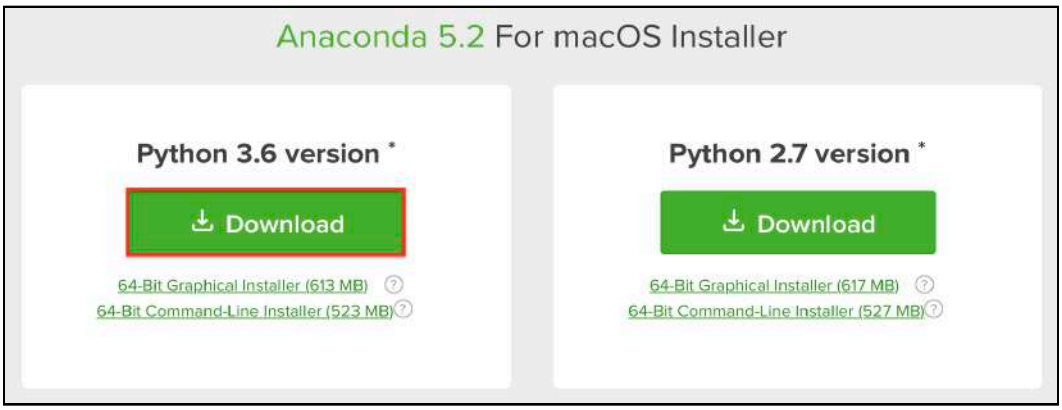
- **Anaconda**: Includes all of the standard packages needed for machine learning. (2 GB)

You'll be using **Anaconda** in this chapter. It doesn't take long to install, and it's way more convenient!

Anaconda

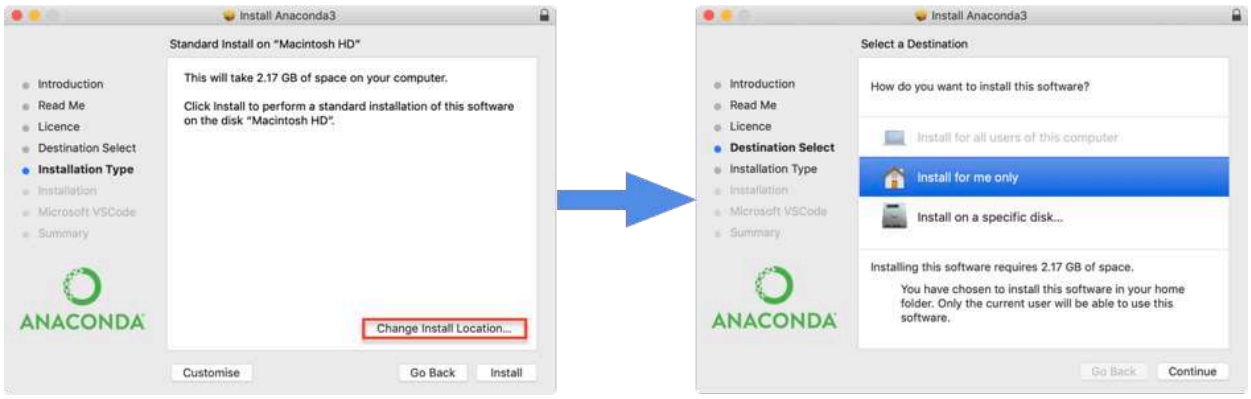
Installing Anaconda

In a browser, navigate to https://www.anaconda.com/download/#macos, and download the **Python 3.6** version:

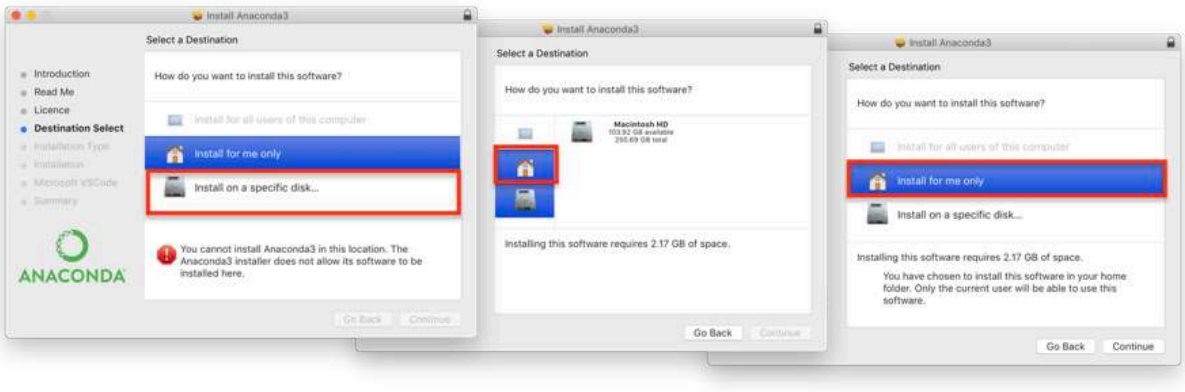


Don't worry about the two 500+ MB installers in the fine print; it's only 643 MB for *both* installers.

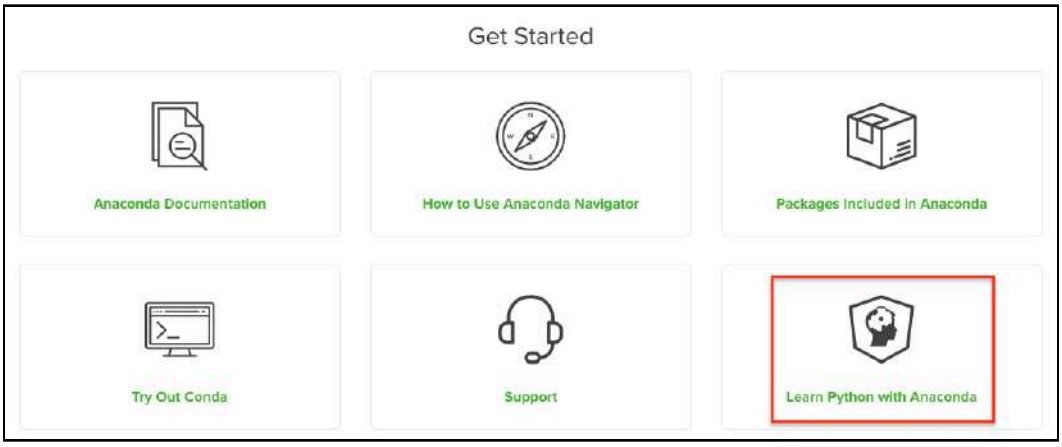
After downloading, run the installer. When prompted to **Change Install Location...**, select **Install for me only**. This installs Anaconda in your home directory:



If it says you can't install it there, click the **Install on a specific disk...** button, then click back to the **Home** button — it should be more agreeable:



While you're waiting for the installation to finish, scroll down to the **Get Started** links and take a closer look at **Learn Python with Anaconda**:



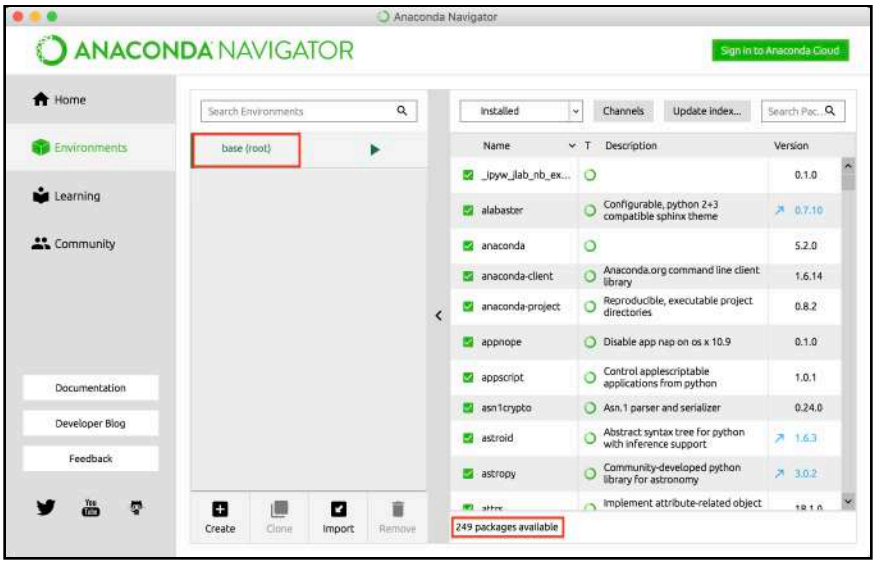
These are video courses about using Python for machine learning. You can view some parts for free, while others require you to be a subscriber before you can watch.

Using Anaconda navigator

Anaconda comes with a desktop GUI that you can use to create environments and install packages in an environment. There's also a handy button to start Jupyter notebooks in specific environments — more about this later.

> **Note**: If you see a prompt to update Anaconda Navigator, do that later when it's convenient — but don't launch it from the updater prompt. Instead, locate Anaconda Navigator in **Finder**, and open it there.

From within **Finder**, locate and start **~/anaconda3/Anaconda Navigator**. Select the **Environments** tab to see the **base (root)** environment and its installed packages:



There are 249 packages installed, however, you won't need most of them for this book. That said, there are three ML packages needed that aren't in the base environment:

- **Keras**: An interface to TensorFlow, Theano, and Microsoft Cognitive Toolkit, from Google.
- **TensorFlow**: Google Brain's library for neural networks.
- **Turi Create**: Apple's ML-as-a-Service framework.

Anaconda knows about Keras and TensorFlow, so you *could* install them with Navigator. However, TensorFlow's installation instructions advise against this, so you'll use `pip` to install all three.

Setting up a base ML environment

In this section, you'll set up some environments. If you prefer a quicker start, **Import mlenv.yaml** into the **Navigator** and skip down to the section **Jupyter Notebooks**.

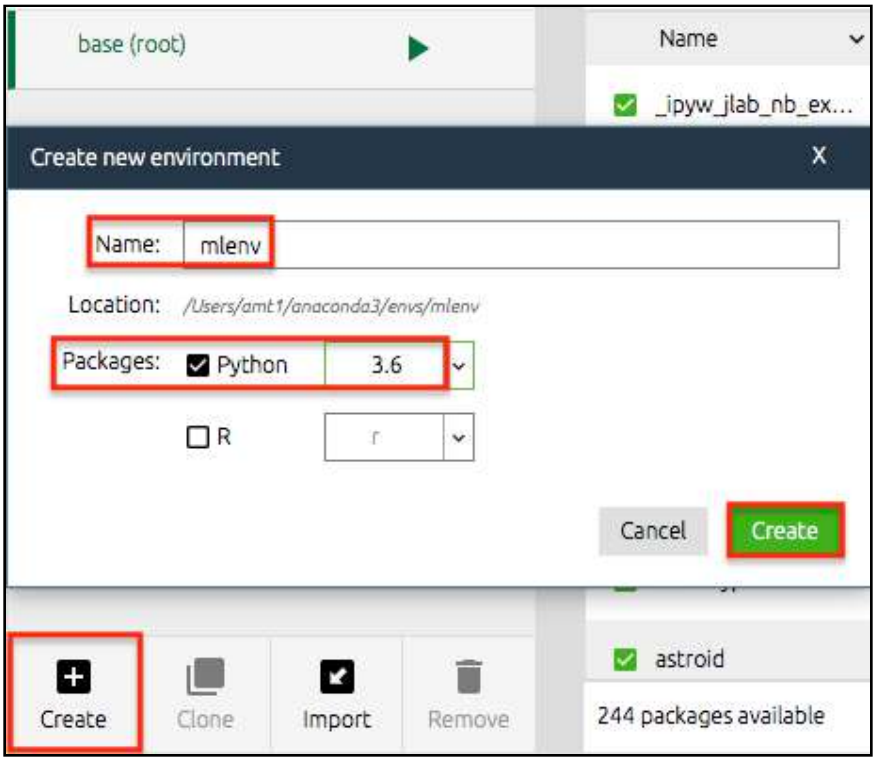
Python libraries for data science

Begin by creating a custom base environment for ML, with *NumPy*, *Pandas*, *Matplotlib*, *SciPy* and *scikit-learn*. You'll be using these data science libraries in this book, but they're not automatically included in new Conda environments.

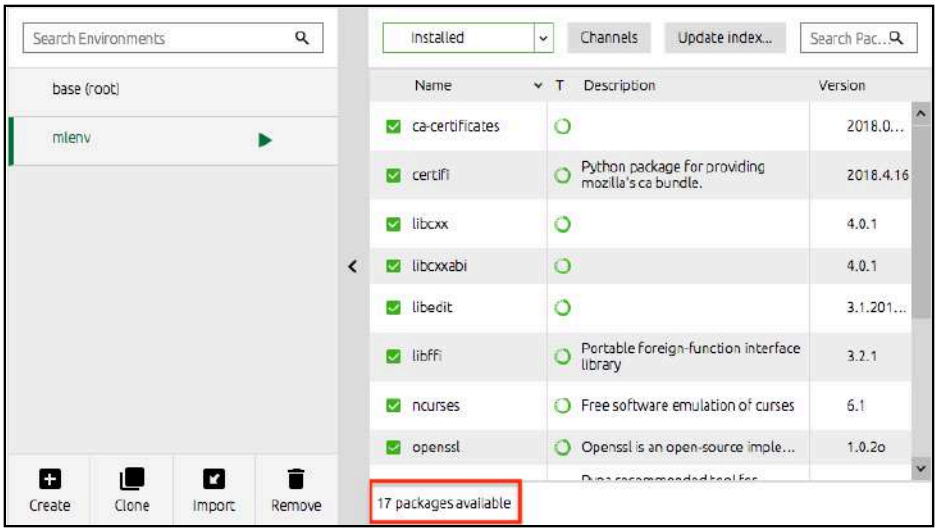
- **NumPy**: Functions for working with multi-dimensional arrays.
- **Pandas**: Data structures and data analysis tools.
- **Matplotlib**: 2D plotting library.
- **Seaborn**: Statistical data visualization library.
- **SciPy**: Modules for statistics, optimization, integration, linear algebra, Fourier transforms, and more, using NumPy arrays.
- **scikit-learn**: Machine learning library.

Once you have the custom base environment for ML, you can clone it to create separate environments for the ML libraries, Keras, TensorFlow, and Turi Create.

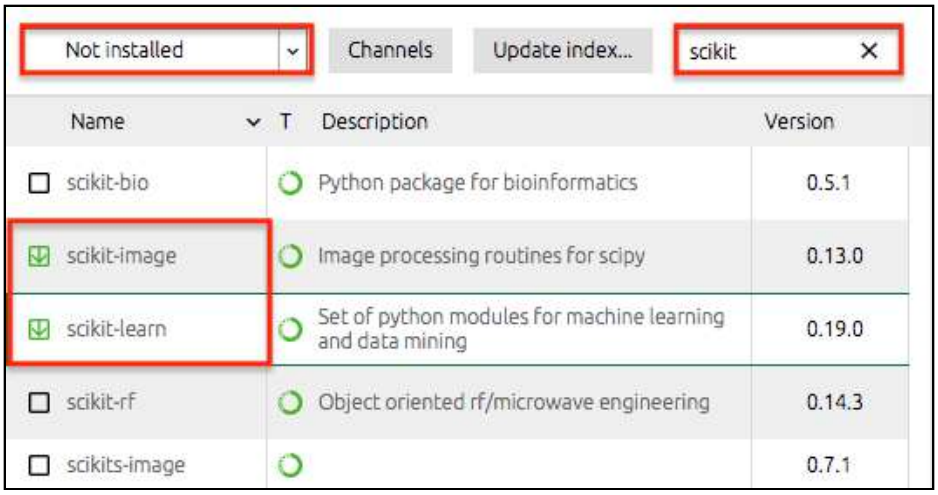
In Anaconda Navigator, **Create** a new environment named **mlenv**, with **Python 3.6**:



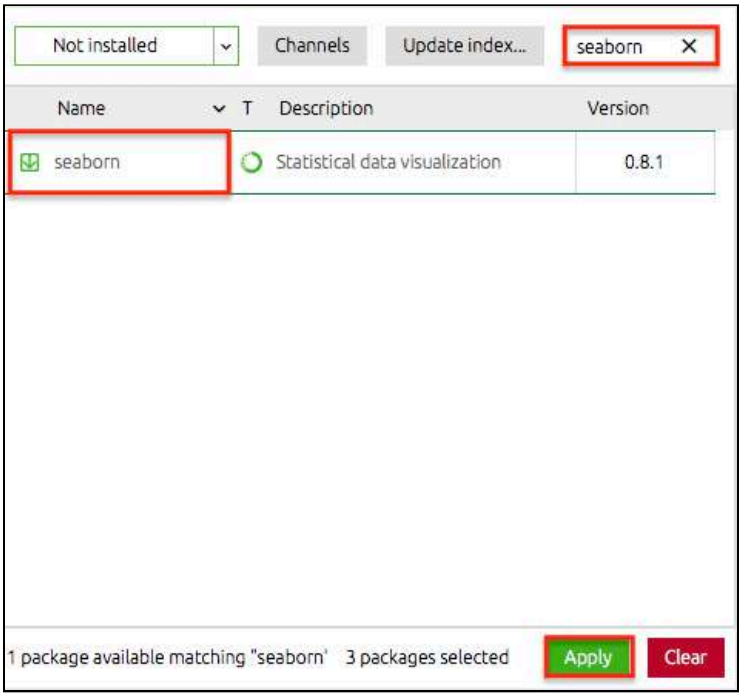
It takes about a minute to install the 17 utility packages:



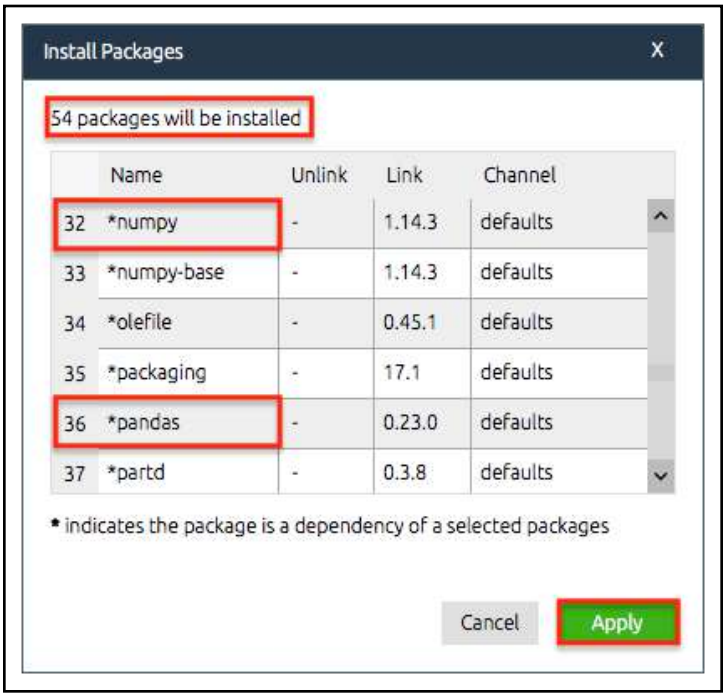
Next, to add the **scikit** ML libraries: change **Installed** to **Not installed**, search for `scikit`, and check the checkboxes next to `scikit-image` and `scikit-learn`:



You also need the **Seaborn** library, so change the search term to **seaborn**, then select that package:

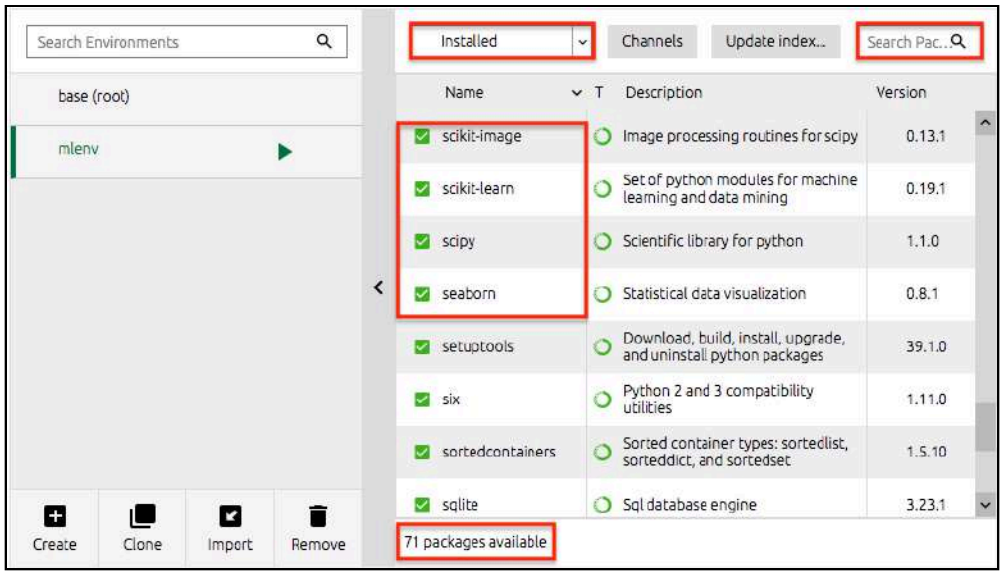


Now click **Apply**. Conda checks the selected packages' dependencies, then displays the list of 54 packages it will install, including `scipy`, `matplotlib`, `pandas` and `numpy`:



Click **Apply** again. This time, you may want to grab a drink or a snack while you wait.

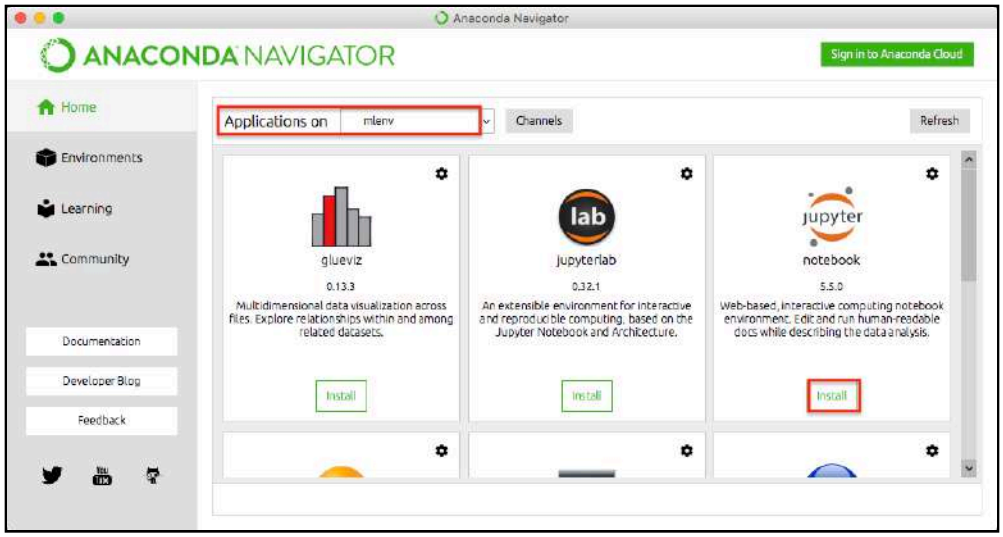
When it's all done, clear the search field and show **Installed**. You now have 71 packages installed.



Note: The actual number of packages you see might be slightly different. If you see something like 200+ packages, quit and restart Anaconda Navigator. When it restarts, that number should be closer to 71.

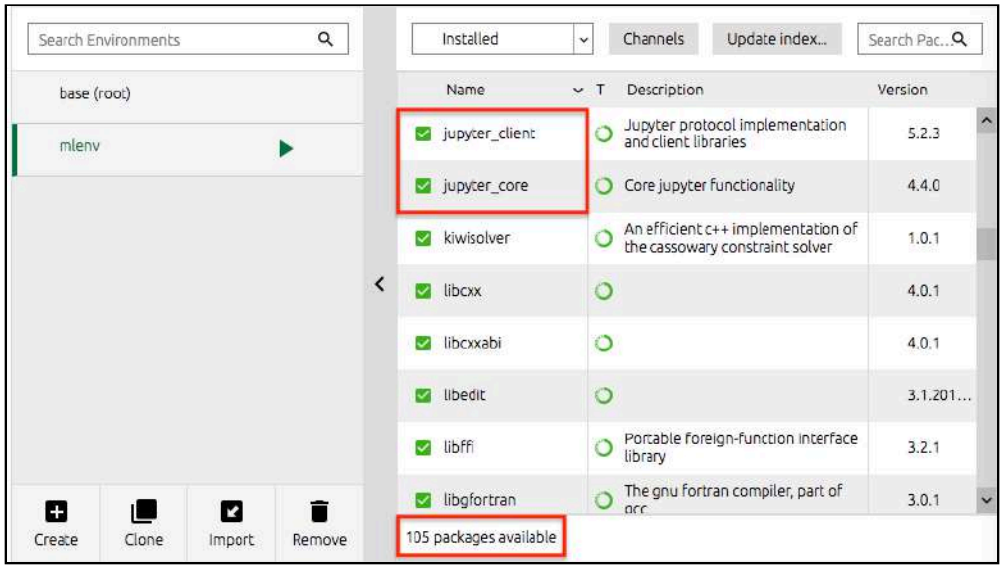
Adding Jupyter to base ML environment

Because you'll be coding in Jupyter notebooks, you need to first add Jupyter to **mlenv**. Select the **Home** Tab. Notice the **Applications on** field contains **mlenv**, and every app displays an **Install** button:



Click the Jupyter **Install** button.

Wait until the button title changes to **Launch**. Then, switch back to the **Environments** Tab and you'll see that **mlenv** now contains 105 packages:



Note: Technically, you don't need to install the Jupyter app in your environment. You can start Jupyter from the command line, but in that case, you'd need to first `activate` the environment you want to code in. By installing Jupyter here, you can start it in your environment simply by clicking the launch button.

Another Note: **Jupyter Lab** is in beta, but it will probably become the standard UI. Feel free to install it and have a look, but understand that this book does not use it.

Jupyter notebooks

With Jupyter notebooks, which are a lot like Swift Playgrounds, you can write and run code, and you can write and render markdown to explain the code. In fact, Chris Lattner and Richard Wei showed a Swift Playground during their talk announcing Swift for Tensorflow at TensorFlow Dev Summit 2018. They used their audience's vocabulary by calling it a **Swift notebook**.

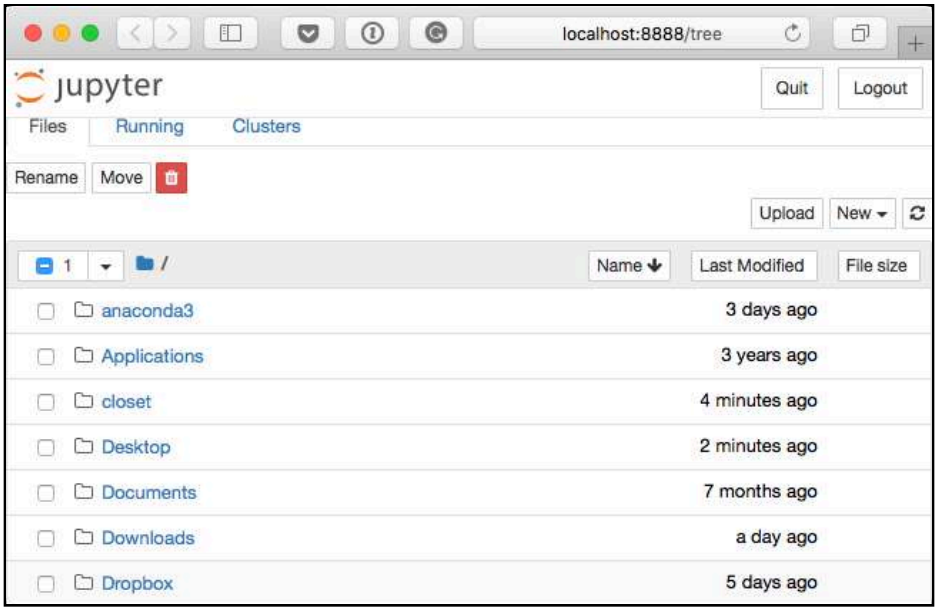
Starting Jupyter

In **Anaconda Navigator's Home** Tab, with **mlenv** selected, click the Jupyter **Launch** button. The following command appears in **Terminal**, followed by messages about a server starting and how to shut it down:

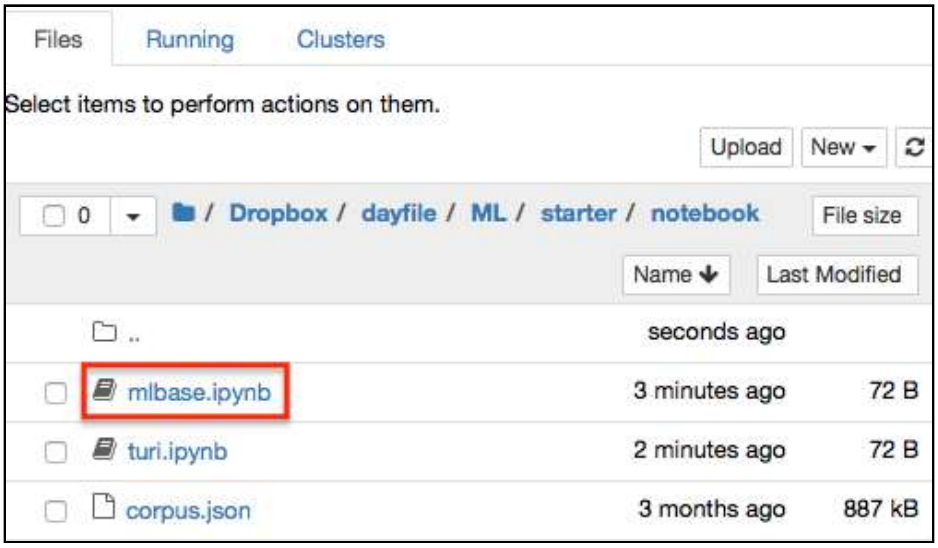
```
/anaconda3/envs/mlenv/bin/jupyter_mac.command ; exit;
```

Keep this window open!

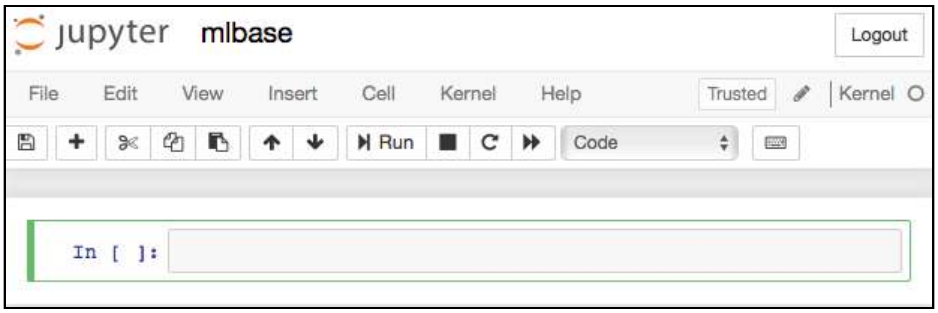
A browser window also opens, showing your home directory:



Navigate to the **starter** folder for this chapter, and open **notebook/mlbase.ipynb**:



The notebook appears in a new browser tab:



Pandas and Matplotlib

At the moment, there's an empty cell. In that cell, type the following lines:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

This imports the NumPy and Pandas modules into the current session. It also imports the `pyplot` module of Matplotlib, and gives everything their customary abbreviated aliases. There's no output for `import` statements, unless there's a warning about future deprecation of a module.

Press **Shift-Enter** to run this code, which will also open a new cell below it.

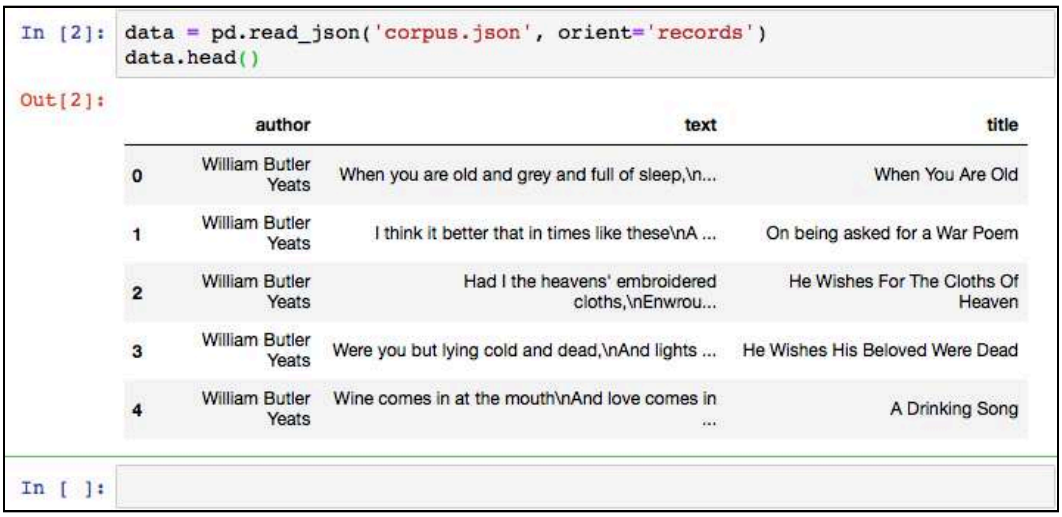
Next, type these lines in the newly created empty cell, then press **Shift-Enter**:

```
data = pd.read_json('corpus.json', orient='records')
data.head()
```

The **starter/notebook** folder contains **corpus.json**, so this loads the data into a `DataFrame` — the Pandas data container, with rows and columns like a spreadsheet. It has powerful functions for manipulation, which is important for massaging data to get the best input for training a model.

The `orient` parameter indicates the JSON string format: `'records'` means it's a list of `column -> value`. You'll take a look at the documentation for this function in a moment.

The `head()` function shows the (default) first five rows:



```
In [2]: data = pd.read_json('corpus.json', orient='records')
        data.head()
```

Out[2]:

	author	text	title
0	William Butler Yeats	When you are old and grey and full of sleep,\n...	When You Are Old
1	William Butler Yeats	I think it better that in times like these\nA ...	On being asked for a War Poem
2	William Butler Yeats	Had I the heavens' embroidered cloths,\nEnwrou...	He Wishes For The Cloths Of Heaven
3	William Butler Yeats	Were you but lying cold and dead,\nAnd lights ...	He Wishes His Beloved Were Dead
4	William Butler Yeats	Wine comes in at the mouth\nAnd love comes in ...	A Drinking Song

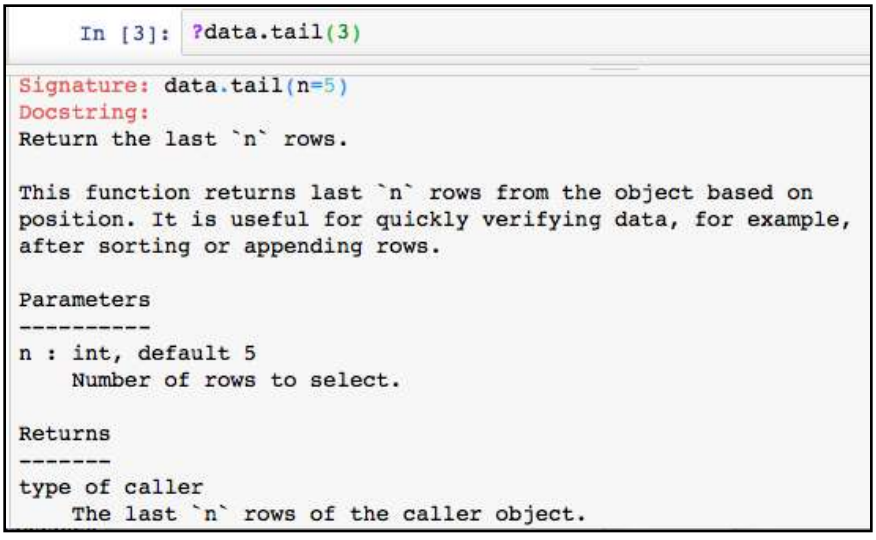
```
In [ ]:
```

> **Note**: **Shift-Enter** runs the current cell and, if this is the last cell, opens a new cell below it; this is convenient when you're testing code as you write it. **Control-Enter** runs the current cell; you'd do this when you add something to an earlier cell and want to update it. The bracketed numbers keep track of the order you run the cells, regardless of their order within the notebook.

In the next empty cell, type the following line, then press **Shift-Enter**:

```
?data.tail(3)
```

The question mark shows the documentation for this function, instead of running the function:



```
In [3]: ?data.tail(3)

Signature: data.tail(n=5)
Docstring:
Return the last `n` rows.

This function returns last `n` rows from the object based on
position. It is useful for quickly verifying data, for example,
after sorting or appending rows.

Parameters
----------
n : int, default 5
    Number of rows to select.

Returns
-------
type of caller
    The last `n` rows of the caller object.
```

Press **Esc** or the **x** button to close the documentation.

Delete the question mark. Then press **Control-Enter** or **Shift-Enter** to run the cell, which will display the last **three rows** of data:

```
In [4]: data.tail(3)
```

Out[4]:

	author	text	title
516	Oscar Wilde	The western wind is blowing fair\nAcross the d...	Serenade (For Music)
517	Oscar Wilde	(To L. L.)\n\nCould we dig up this long-buried...	Roses And Rue
518	Oscar Wilde	In the glad springtime when leaves were green,...	From Spring Days To Winter (For Music)

If you'd like, you can also see documentation in a pop-up box: select `pd.read_json` in the second cell, then press **Shift-Tab-Tab**:

```
In [2]: data = pd.read_json('corpus.json', orient='records')
data.head()

Out[2]: Signature: pd.read_json(path_or_buf=None, orient=None, typ='frame', dtype=True,
convert_axes=True, convert_dates=True, keep_default_dates=True, numpy=False,
precise_float=False, date_unit=None, encoding=None, lines=False, chunksize=None,
compression='infer')
Docstring:
Convert a JSON string to pandas object

Parameters
----------
path_or_buf : a valid JSON string or file-like, default: None
    The string could be a URL. Valid URL schemes include http, ftp, s3, and
```

The question mark doesn't work on this line unless you delete `data =`.

In the next empty cell, type `data.d`. Then press **Tab** to see a list of options:

```
In [4]: data.describe
Out[4]: data.diff
        data.div
        data.divide
        data.dot
        data.drop
        data.drop_duplicates
        data.dropna
        data.dtypes
        data.duplicated

In [ ]: data.d
```

Now, press **Enter** to select `data.describe`. Then type `()`, and press **Shift-Enter**:

```
In [5]: data.describe()
```

Out[5]:

	author	text	title
count	519	519	519
unique	10	517	516
top	Emily Dickinson	I think it better that in times like these\nA ...	The Wife's Will
freq	361	2	2

The output includes the column identifiers: `author`, `text`, and `title`. You can use these to sort the data.

Next, **Shift-Enter** the following line:

```
data.sort_values(by='title')
```

Out[6]:

	author	text	title
69	Emily Dickinson	"Arcturus" is his other name --\nI'd rather ca...	"Arcturus" is his other name
81	Emily Dickinson	"Faith" is a fine invention\nWhen Gentlemen ca...	"Faith" is a fine invention
224	Emily Dickinson	"Faithful to the end" Amended\nFrom the Heaven...	"Faithful to the end" Amended
356	Emily Dickinson	"Heaven" -- is what I cannot reach!\nThe Apple...	"Heaven" -- is what I cannot reach!
193	Emily Dickinson	"Heaven" has different Signs -- to me -- \nSome...	"Heaven" has different Signs -- to me --
119	Emily Dickinson	"Heavenly Father" -- take to thee\nThe supreme...	"Heavenly Father" -- take to thee
166	Emily Dickinson	"Hope" is the thing with feathers --\nThat per...	"Hope" is the thing with feathers

You can extract a column into a separate `Series` object and count how often each value appears:

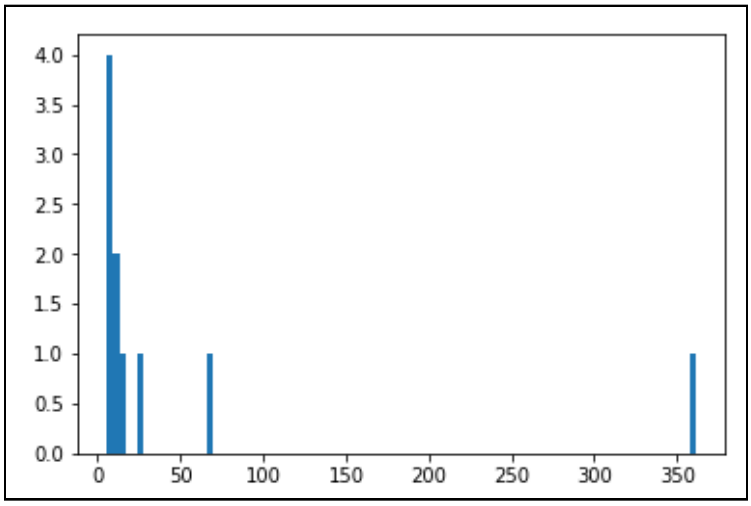
```
authors = data.author
freq = authors.value_counts()
freq
```

As in Swift Playgrounds, an object name (`freq`) on a line by itself displays that object.

```
Out[7]: Emily Dickinson          361
        Walt Whitman              67
        John Keats                25
        Joyce Kilmer              16
        William Butler Yeats      10
        Oscar Wilde               10
        Lewis Carroll              9
        Charlotte Bronte           8
        Edgar Allan Poe            7
        Sir Walter Scott           6
        Name: author, dtype: int64
```

Frequency varies from 6 to 361. You can plot a histogram of this distribution:

```
plt.hist(freq, bins=100)
plt.show()
```



Specifying `bins=100` divides the range [6,361] into 100 consecutive, non-overlapping intervals, called *bins* or *buckets*. The histogram's x-axis has 100 bins, between 0 and 361-ish. The y-axis shows the number of authors in each bin.

Note: This example is from our tutorial Natural Language Processing on iOS with Turi Create. It trains a natural language model with lines from poems by famous authors. The trained model can be used to classify new text. For each author it knows about, it computes the probability that this author wrote the new text. The `freq` values here should set off alarm bells — there's way too much bias toward Emily Dickinson, so the model will classify most test texts as written by her.

Python

In this next section, you'll spend some time getting familiar with common Python syntax.

`if-else` and `None`:

A major syntax difference is the importance of indentation. With Python, indentation replaces `{}` to define closures (blocks).

Python also has a built-in `NoneType` to represent no-value. You should use `is` or `is not` to test for a no-value result, not ==, which is what you'd do in Swift. The reason is because classes can override ==.

```
if authors is None:
    print('authors is None')
else:
    print('authors is not None')
```

The output is:

```
authors is not None
```

Here's how you define and call a function:

```
def mysum(x, y):
    return x + y

print(mysum(1, 3))
```

This outputs `4`.

Notice the auto-indent on the second line. Also notice how you have to un-indent to type the third line. Coding convention says to leave a blank line after the function definition, but it's not a syntax rule, and you may be more comfortable omitting the blank line.

for-loop and arrays (lists):

```
mylist = [1, 2, 3]
mylist.append(4)
if mylist:
    print('mylist is not empty')

for value in mylist:
    print(value)
print('List Length: %d' % len(mylist))
```

Arrays in Python are similar to arrays in Swift. To test whether an array is empty, use its name. `for` loops are also similar to Swift, but they use the `:` plus indentation syntax.

The `len()` function works on any Python collection object, and it returns th elength of the array, in a similar way to how the `.count` property for arrays in Swift returns the number of items in an array.

Run those commands, and you'll see this output:

```
mylist is not empty
1
2
3
4
List Length: 4
```

To make a point about indentation, go ahead and add a blank line, but indent the last statement to match the `print` statement in the loop, like so:

```
mylist = [1, 2, 3]
mylist.append(4)
if mylist:
    print('mylist is not empty')

for value in mylist:
    print(value)

    print(`List Length: %d` % len(mylist))
```

Now, both print statements are in the loop, so the output is:

```
1
List Length: 4
2
List Length: 4
3
List Length: 4
4
List Length: 4
```

Excellent, you survived a session with Python and used a few library functions! Feel free to play around some more until you get the hang of it. This book uses a lot of Python libraries and functions, so it's good to understand the basic syntax before moving on.

Transfer learning with Turi Create

Despite the difference in programming languages, deep down Turi Create shares a lot with Create ML, including transfer learning. With Turi Create v5, you can even do transfer learning with the same `VisionFeaturePrint_Screen` model that Create ML uses.

In this section, you'll create the same HealthySnacks model as the previous chapter, except this time, you'll use Turi Create. Unlike Create ML, which allowed you to train your model through the playgrounds UI in Xcode, Turi Create needs some coding when compared to Create ML. This means you'll learn more about working with data using Python.

Creating a Turi Create environment

First, you need a new environment with the `turicreate` package installed. You'll clone the **mlenv** environment to create **turienv**, then you'll install `turicreate` in the new environment. Anaconda doesn't know about `turicreate`, so you'll have to `pip install` it from within Terminal.

> **Note**: Again, if you prefer a quicker start, **Import turienv.yaml** into the **Navigator** and skip down to the section **Turi Create Notebook**.

While it's possible to clone mlenv in Anaconda Navigator's Environments tab, you'll be using a command line to install `turicreate`, so it's just as easy to use a command line to clone, as well.

> **Note**: If you've changed Terminal's default shell to something different from **bash**, check that your `$PATH` includes `~/anaconda3/bin`.

In **Terminal**, enter this command:

```
conda create -n turienv --clone mlenv
```

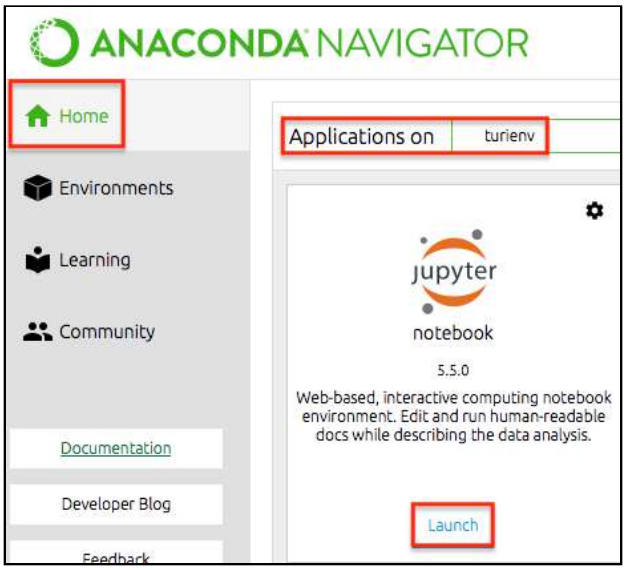
This does the same as Anaconda Navigator's Clone button: it creates an environment named **turienv**, which is cloned from **mlenv**.

Wait a little while until you see:

```
#
# To activate this environment, use:
# > source activate turienv
#
# To deactivate an active environment, use:
# > source deactivate
#
```

> **Note**: Again, if your Terminal doesn't use **bash**, these instructions might appear as `conda activate turienv` and `conda deactivate`. Also, if you see a message to update Conda, go ahead and do that.

Take a look at **Anaconda Navigator's Home** tab. Because you used Conda to create turienv, it appears in Anaconda Navigator. Also, because you cloned it from mlenv, Jupyter is ready to launch:



Time to install `turicreate`.

Enter the **activate** command:

```
source activate turienv
```

The command line prompt now starts with `(turienv)`, showing it's the active environment.

Enter this command:

```
pip install turicreate==5.0
```

This downloads and installs the version 5 `turicreate` package, which lets you use the Vision framework model for transfer learning.

List pip-installed packages

Take a quick look at the **turienv** environment in **Navigator**; it still shows only 105 packages. That's because packages installed with `pip` don't show up in Navigator.

In **Terminal**, use this command to list all of the packages in the active environment or a specific package:

```
conda list
conda list coremltools
```

You need the `coremltools` package to create Core ML models from Turi Create models. Installing `turicreate` also installs `coremltools`.

The output of the second command looks similar to this:

```
# packages in environment at /Users/amt1/anaconda3/envs/mlenv:
#
# Name                    Version                   Build  Channel
coremltools               2.0b1                     <pip>
```

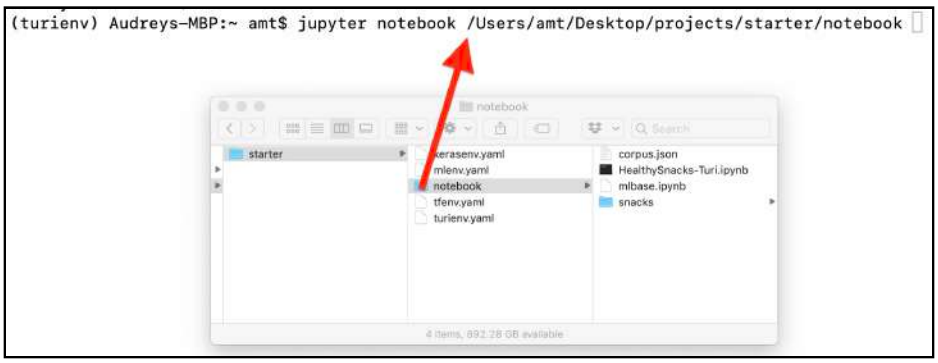
The **Build** value `<pip>` shows `coremltools` was installed with `pip`, not `conda`.

Turi Create notebook

> **Note**: If you skipped the manual environment setup and imported turienv.yaml into Anaconda Navigator, use the Jupyter **Launch** button on the **Anaconda Navigator Home** Tab instead of the command line below, then navigate in the browser to **starter/notebook**.

This time, you'll start Jupyter in the folder where the notebooks are stored; locate **starter/notebook** in **Finder**. Then, enter the following command to start a Jupyter notebook in this environment, starting from this directory:

```
jupyter notebook <drag the starter/notebook folder in Finder to here>
```



In the browser, open **HealthySnacks-Turi.ipynb**. There's only an empty cell.

Type the following commands in this cell and press **Shift-Enter**:

```
import turicreate as tc
import matplotlib.pyplot as plt
```

You're importing the Turi Create package and the `pyplot` module of the Matplotlib package into the current session, with aliases `tc` and `plt`. You may get a FutureWarning message, which you can safely ignore.

In the next cell, **Shift-Enter** this command:

```
train_data = tc.image_analysis.load_images("snacks/train",
with_path=True)
```

This loads all the images from the **snacks/train** directory into an `SFrame`, the data container for Turi Create. An `SFrame` contains rows and columns, like a Pandas `DataFrame` — in fact, you can create an `SFrame` from a `DataFrame`. `SFrame` has powerful functions for manipulation, similar to `DataFrame`. It's also optimized for loading from disk storage, which is important for large data sets that can easily overwhelm RAM.

Like Create ML's `MLDataTable`, an `SFrame` keeps only the image metadata in memory.

> **Note**: It's safe to ignore warnings about .DS_Store being an unsupported image format.

This `SFrame` object contains a row for each image, as well as the path of the folder the images were loaded from. This `SFrame` should contain 4838 images. Verify this by asking for its length:

> **Note**: Run each command in its own cell. Remember **Shift-Enter** runs the current cell and opens a new cell below it. Always wait for the * to turn into a number, indicating the command has finished running.

```
len(train_data)
```

Next, look at the actual contents of the `SFrame`:

```
train_data.head()
```

Note that now, `head()` defaults to show the first *10* rows:

path	image
snacks/train/healthy/appl e/007a0bec00a90a66.jpg ...	Height: 341 Width: 256
snacks/train/healthy/appl e/007ec56b6529e036.jpg ...	Height: 256 Width: 341
...	
snacks/train/healthy/appl e/01ecc03a12e21e39.jpg ...	Height: 256 Width: 446
snacks/train/healthy/appl e/021d2569ce62aa93.jpg ...	Height: 256 Width: 341

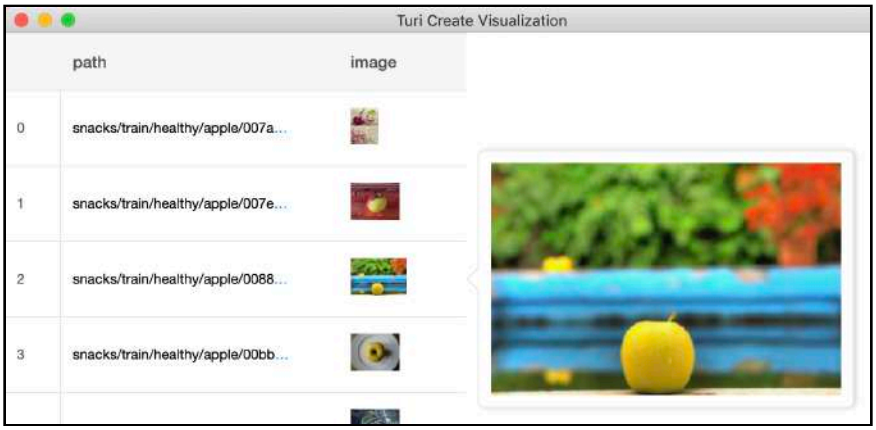
[10 rows x 2 columns]

The first rows in the SFrame

Even though the `SFrame` only shows the image's height and width in the table, it actually contains the complete image. Run the following command to see the actual images:

```
train_data.explore()
```

This opens a new window with image thumbnails (it may take a few seconds to load). Hover over a row to view a larger version of an image.



Explore the training images

This interactive visualization can be useful for a quick look at the training data. The `explore()` command only works with Turi Create on the Mac, not on Linux or from a Docker container.

Enter this command to look at individual images directly inside the notebook, using the `pyplot` module:

```
plt.imshow(train_data[0]["image"].pixel_data)
```

Here, `train_data[0]` gets the first row from the `SFrame`, `["image"]` gets the object from the image column for that row, and `.pixel_data` turns this image object into something that matplotlib can show with the `plt.imshow()` command.

Looking at an image with matplotlib

Your notebook may show a different image than in the illustration, since Turi Create may have loaded your images in another order. Feel free to look at a few images by changing the row index (use any value from 0 to 4,837).

There is one more piece of data to gather before you can start training — the name of the class for each image. The images are stored in subdirectories named after the classes — "apple," "hot dog," etc. The `SFrame` knows the path the image was loaded from, but these paths look something like this:

```
snacks/train/hot dog/8ace0d8a912ed2f6.jpg
```

The class for image `8ace0d8a912ed2f6.jpg` is "hot dog", but it's hidden inside that long path. To make this more obvious, you'll write some code to extract the class name from the path. Run the following commands to extract the name of the first image's class folder:

```
# Grab the full path of the first training example
path = train_data[0]["path"]
print(path)
```

```
# Find the class label
import os
os.path.basename(os.path.split(path)[0])
```

Here, you're getting the full path of the first image, then using the `os.path` Python package for dealing with path names. First, `os.path.split()` chops the path into two pieces: the name of the file (8ace0d8a912ed2f6.jpg) and everything leading up to it. Then `os.path.basename()` grabs the name of the last folder, which is the one with the class name. Since the first training image is of an apple, you get "apple."

> **Note**: The `#` character starts a comment in Python. Note that you first need to import the `os` package, or else Python won't know what `os.path` is.

Getting the class labels

OK, now you know how to extract the class name for a single image, but there are over 4,800 images in the dataset. As a Swift programmer, your initial instinct may be to use a `for` loop, but if you're really Swift-y, you'll be itching to use a `map` function. `SFrame` has a handy `apply()` method that, like `map`, lets you apply a function to every row in the frame:

```
train_data["path"].apply(lambda path: ...do something with path...)
```

In Python, a **lambda** is similar to a closure in Swift — it's just a function without a name. `train_data["path"].apply()` performs this lambda function on every row in the path column. Inside the lambda, just put the above code snippet that you used to extract the class name from the full path:

```
train_data["label"] = train_data["path"].apply(lambda path:
    os.path.basename(os.path.split(path)[0]))
```

Run the above cell and now the `SFrame` will have a new column called "label" with the class names. To verify this worked, run `train_data.head()` again — do this in a new cell, or scroll up to the fourth cell, and press **Control-Enter** to run it.

path	image	label
snacks/train/healthy/apple/007a0bec00a90a66.jpg ...	Height: 341 Width: 256	apple
snacks/train/healthy/apple/007ec56b6529e036.jpg ...	Height: 256 Width: 341	apple
snacks/train/healthy/apple/008816276298888f6.jpg ...	Height: 256 Width: 384	apple

The SFrame now has a new column

You can also use `train_data.explore()` again for a visual inspection. Run this command to see the summary function:

```
train_data["label"].summary()
```

This prints out a few summary statistics about the contents of the `SFrame`'s label column:

```
+------------------+-------+----------+
|       item       | value | is exact |
+------------------+-------+----------+
|      Length      |  4838 |   Yes    |
| # Missing Values |   0   |   Yes    |
| # unique values  |   20  |    No    |
+------------------+-------+----------+

Most frequent items:
+-------+-----------+-------+--------+----------+-------+---------+-----------+
| value | pineapple | apple | banana | doughnut | grape | hot dog | ice cream |
+-------+-----------+-------+--------+----------+-------+---------+-----------+
| count |    260    |  250  |  250   |   250    |  250  |   250   |    250    |
+-------+-----------+-------+--------+----------+-------+---------+-----------+
+-------+--------+-------+
| juice | muffin | salad |
+-------+--------+-------+
|  250  |  250   |  250  |
+-------+--------+-------+
```

Summary for the label column

As you can see, each of the classes has roughly the same number of elements. For some reason, `summary()` only shows the top 10 classes, but we have 20 in total. To see the number of rows for all of the classes, run the following command:

```
train_data["label"].value_counts().print_rows(num_rows=20)
```

All right, that's all you need to do with the data for now. You've loaded the images into an `SFrame`, and you've given each image a label, so Turi Create knows which class it belongs to.

Let's do some training

Once you have your data in an `SFrame`, training a model with Turi Create takes only a single line of code (OK, it's three lines, but only because we have to fit it on the page):

```
model = tc.image_classifier.create(train_data, target="label",
                                   model="VisionFeaturePrint_Screen",
                                   verbose=True, max_iterations=50)
```

Alternatively, if training takes too long on your Mac, you can just load the Turi Create model from the current folder:

```
model = tc.load_model("HealthySnacks.model")
```

> **Note**: If you get run errors on this command, open a new cell, then enter the command (note the **!**) `!conda list turicreate` to check your `turicreate` version. Then check github.com/apple/turicreate to find the latest version. If there's a more recent beta, open a new cell to install it with `!pip install turicreate==5.0`.

This command creates a new image classifier from the `train_data` `SFrame`. The `target` parameter tells Turi Create that the class names are in the `SFrame`'s `label` column. By default, Turi Create only does 10 iterations, but you increase this to 50, so the logistic regression will train for up to 50 iterations.

The first time you run this command, Turi Create downloads a pre-trained neural network. The `model` parameter contains the name of that neural network, in this case **VisionFeaturePrint_Screen**. This is the model used by Apple's Vision framework, and is also the default model for Create ML.

At the time of writing, Turi Create supports three model architectures: The other two are ResNet-50 and SqueezeNet version 1.1. ResNet-50 exports a Core ML model ~90MB, which is not really suited for use on mobile devices.

SqueezeNet exports a Core ML model ~4.7MB, so it's a better option. But VisionFeaturePrint_Screen is built into iOS 12, so it produces a *much* smaller model — only ~41**KB**.

Turi Create, like Create ML, performs feature extraction on the images. This takes about the same amount of time as Create ML — 2m 22s on my MacBook Pro. And then comes the logistic regression:

```
Logistic regression:
--------------------------------------------------------
Number of examples          : 4590
Number of classes           : 20
Number of feature columns   : 1
Number of unpacked features : 2048
Number of coefficients      : 38931
Starting L-BFGS
--------------------------------------------------------
| Iteration | Passes   | Elapsed Time | Training Accuracy | Validation Accuracy |
...
| 5         | 8        | 3.421915     | 0.815468          | 0.850806            |
| 10        | 13       | 5.857664     | 0.884532          | 0.875000            |
| 15        | 19       | 8.655192     | 0.926580          | 0.903226            |
| 20        | 24       | 11.080811    | 0.944227          | 0.895161            |
| 25        | 29       | 13.513294    | 0.982353          | 0.883065            |
| 30        | 34       | 15.903836    | 0.994553          | 0.879032            |
| 35        | 39       | 18.329717    | 0.998911          | 0.875000            |
| 40        | 44       | 20.812450    | 1.000000          | 0.850806            |
| 45        | 49       | 23.279066    | 1.000000          | 0.854839            |
| 50        | 55       | 26.231560    | 1.000000          | 0.866935            |

Completed (Iteration limit reached).
This model may not be optimal. To improve it, consider increasing `max_iterations`.
```

Validation

After 10 iterations, validation accuracy is close to training accuracy at ~88%. At 15 iterations, training accuracy starts to pull away from validation accuracy, and races off to 100%, while validation accuracy actually drops! Massive overfitting happening here! The default 10 iterations would've been better, but running the `image_classifier.create` command with only 10 iterations will do the feature extraction all over again! Too bad we couldn't save the intermediate states of the model, or stop the training when the validation accuracy shows a decreasing trend.

Actually, in the next chapter, you'll learn how to wrangle the Turi Create code — it's open source, after all! — to save the extracted features, so you can experiment more with the classifier.

> **Spoiler alert**: Keras lets you save the best-so-far model while it's training, so you can retrieve it, if it isn't the final model. It also lets you stop early, if validation accuracy doesn't improve over some number (your choice) of iterations.

Let's go ahead and evaluate this model on the test dataset.

Testing

Run these commands to load the test dataset and get the class labels:

```
test_data = tc.image_analysis.load_images("snacks/test", with_path=True)

test_data["label"] = test_data["path"].apply(lambda path:
       os.path.basename(os.path.split(path)[0]))

len(test_data)
```

The last command is just to confirm you've got 952 images.

Next, run this command to evaluate the model and collect metrics:

```
metrics = model.evaluate(test_data)
```

Unlike Create ML, the output of this command doesn't show any accuracy figures — you need to examine `metrics`. Run these commands in the same cell:

```
print("Accuracy: ", metrics["accuracy"])
print("Precision: ", metrics["precision"])
print("Recall: ", metrics["recall"])
print("Confusion Matrix:\n", metrics["confusion_matrix"])
```

Here are my metrics:

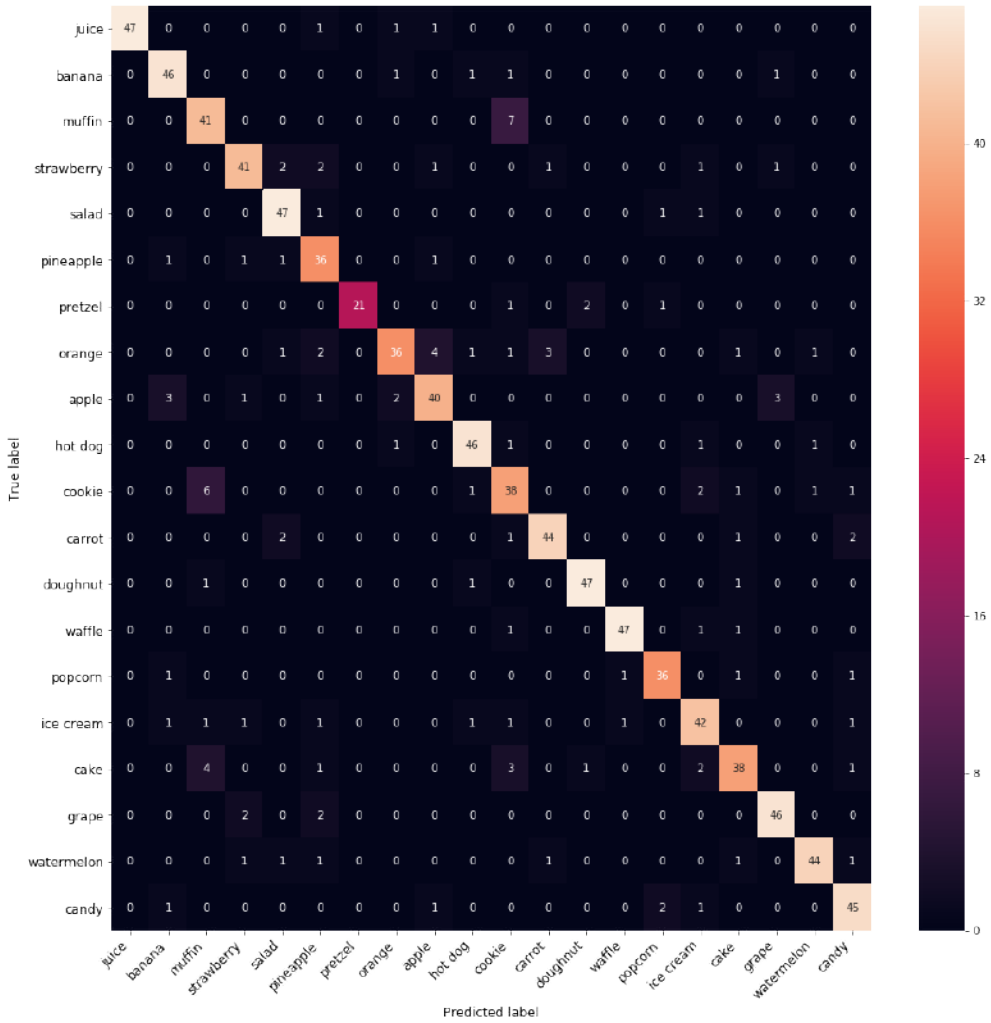
```
Accuracy:  0.8697478991596639
Precision:  0.8753552272362406
Recall:  0.8695450680272108
Confusion Matrix:
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
|  ice cream   |      candy      |   1   |
|    apple     |      banana     |   3   |
|    orange    |    pineapple    |   2   |
|    apple     |    strawberry   |   1   |
|  pineapple   |      banana     |   1   |
|  strawberry  |      salad      |   2   |
|   popcorn    |      waffle     |   1   |
|    carrot    |      salad      |   2   |
|    orange    |    watermelon   |   1   |
|   popcorn    |     popcorn     |   36  |
+--------------+-----------------+-------+
[107 rows x 3 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and
columns.
```

No surprises: Accuracy, precision and recall are all similar to the final validation accuracy of the model. Unlike Create ML, Turi Create gives only overall values for precision and recall, and you need some code to get precision and recall for each class.

In the next chapter, you'll learn how to get recall for each class.

The confusion matrix shows only the first 10 rows: the model mistook one "ice cream" image for "candy," three "apple" images for "banana," etc. Presented this way, it doesn't look much like a matrix.

In the next chapter, you'll learn how to get this nifty visualization:



The confusion matrix

This heatmap shows small values as a cool color — black or dark purple — and large values as warm colors — red to orange to white. The larger the value, the brighter it gets. The correct matches are on the diagonal, so the highest values are there. With only 21 correct matches, "pretzel" stands out, but there are only 25 images in the pretzel folder, so 21 is OK. Purple numbers off the diagonal indicate problems. More about this in the next chapter!

Exporting to Core ML

In the next cell, **Shift-Enter** this command:

```
model
```

This displays information about the model.

```
Class                                  : ImageClassifier

Schema
------
Number of classes                      : 20
Number of feature columns              : 1
Input image shape                      : (3, 299, 299)

Training summary
----------------
Number of examples                     : 4590
Training loss                          : 1.2978
Training time (sec)                    : 174.5081
```

Now to save this new model so you can load it with Core ML. There are two ways to save models using Turi Create:

```
model.save("HealthySnacks.model")
```

This saves the model in Turi Create's own format, which allows you to load it back into the Python notebook later using `tc.load_model()`. Once you've trained a Turi Create model, you can't modify it afterwards, but you might want to evaluate it on different test data, or examine the metrics more closely.

Run this command to get a Core ML model:

```
model.export_coreml("HealthySnacks.mlmodel")
```

You can add the mlmodel to Xcode in the usual way, if you want to compare it with the Create ML model. Despite being based on the same pre-trained model, the two custom models aren't the same: The accuracy of this model is a little lower, and it's half the size of the Create ML model.

Shutting down Jupyter

To shut down Jupyter, click the **Logout** button in this browser window and also in the window showing your ML directory.

In the **Terminal** window that ran `jupyter_mac.command ; exit;`, press **Control-C-C** to stop the server. If the prompt doesn't return, close this terminal window.

Deactivating the active environment

If you activated **turienv** at the terminal command line, enter this command to deactivate it:

```
source deactivate
```

This deactivates the **turienv** environment; the command line prompt loses the `(turienv)` prefix.

Useful Conda commands

You've seen a lot of new commands. Two useful Conda resources are:

- Conda cheat sheet
- Conda user guide: Tasks

Below are the commands used in this chapter, along with some other useful commands.

> **Note**: Some command options use two dashes. One-dash options are often abbreviations of two-dash options, for example, `-n` is short for `--name`.

> **Another Note**: Some conda environment management tasks can be done in two ways: `conda env <command>` or `conda <different command> <options>`

Basic workflow

Create a new environment:

```
conda create -n <env name>
```

Clone an existing environment to create a new environment:

```
conda create -n <new env name> --clone <existing env name>
```

Create a new environment from a **YAML file**:

```
conda env create -f <.yaml file>
```

The first line of the YAML file sets the new environment's name. The starter folder for this chapter contains YAML files for `kerasenv`, `mlenv`, `tfenv` and `turienv`. If you prefer the GUI to the command line, you can import these into Anaconda Navigator.

Activate an environment:

```
source activate <env name>
```

Install packages in an active environment:

```
conda install <pkg names>
```

Install packages in a **non-active** environment:

```
conda install -n <env name> <pkg names>
```

> **Note**: A message from conda about installing multiple packages: It is best to install all packages at once so that all of the dependencies are installed at the same time.

Install non-conda packages or TensorFlow and Keras in an *active environment*: Use `pip install` instead of `conda install`. To install **multiple packages**, create a **requirements.txt** file listing the packages, one per line, then run this command:

```
pip install -r requirements.txt
```

Start Jupyter from the active environment [in a specific directory]:

```
jupyter notebook <directory path>
```

Shutdown Jupyter: Logout in the Jupyter web pages, then press **Control-C-C** in terminal window where server is running.

Deactivate an environment: Run this command in the terminal window where you activated the environment:

```
source deactivate
```

Remove an environment:

```
conda remove -n <env name> --all
```

Or

```
conda env remove -n <env name>
```

Listing environments or packages

List environments; * indicates currently active environment:

```
conda info --envs
```

Or

```
conda env list
```

List packages or a specific package:

In the **active** environment:

```
(activeenv)...$ conda list
(activeenv)...$ conda list <package name>
```

In a **non-active** environment:

```
conda list -n <non-active env name>
conda list -n <non-active env name> <package name>
```

Docker and Colab

There are two other high-level tools for supporting machine learning in Python: Docker and Google Colaboratory. These can be useful for developing machine learning projects, but we're not covering them in detail in this book.

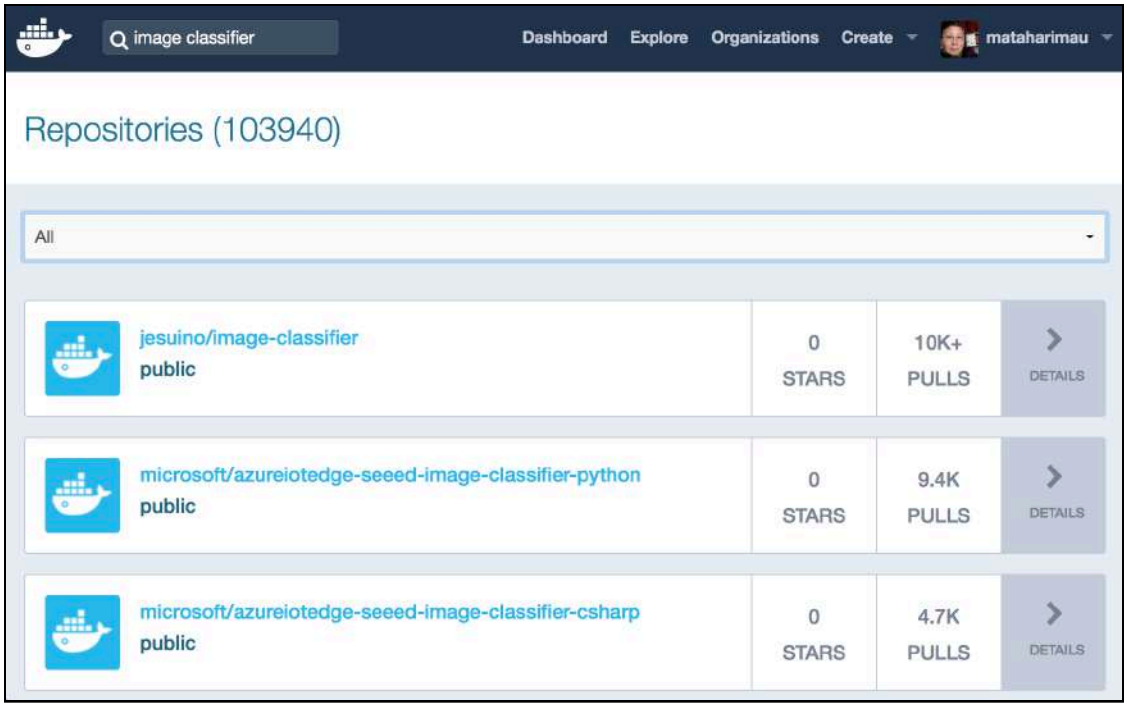
Docker is a useful tool for creating reproducable environments for running machine learning projects, and is therefore a useful tool when you want to scale up projects. Colaboratory gives you access to free GPU. But while you're working through the Turi Create and Keras examples in this book, and trying out your own modifications, it's more convenient to have the two environments, and know how to build or modify them.

Docker

Docker is a container-based system, sort of like virtual machines. Docker allows you to re-use and modularize re-usable environments, and is a fundamental building block to scaling services and applications on the Internet efficiently. Installing Docker gives you access to a large number of ML resources distributed in Docker images as Jupyter notebooks like **hwchong/kerastraining4coreml** or Python projects like the **bamos/openface** face recognition model.

Docker images can be useful to share pre-defined environments with colleagues, or peers, but at some point they will require an understanding of how to write Docker images (by editing the corresponding Dockerfile), which is beyond the scope of what we're covering here.

You can download the community edition of Docker for Mac from https://dockr.ly/2hwNOZZ. To search Docker Hub hub.docker.com (a repository for Docker images), click **Explore**, then search for *image classifier*:



Search Docker Hub for image classifier

Google Colaboratory

Google Research's Colaboratory at colab.research.google.com is a Jupyter Notebook environment that runs in a browser. Its best feature is, you can set the **runtime type** of a notebook to **GPU** to use Google's GPU for free. The downside is, it's not straightforward to transfer data in and out. Notebooks you create are stored in a **Colab Notebooks** directory in your Google Drive, and Colab supplies sample code for accessing files there. To upload files from your local drive, use Google's **Chrome** browser — the upload window never appeared when I tried this in Safari. I uploaded snacks.zip, then unzipped it, but the `load_images()` function couldn't find it. StackOverflow has a lot of questions, and a few answers, about getting datasets into Colab.

Key points

- Get familiar with Python. Its widespread adoption with academics in the machine learning field means if you want to keep up to date with machine learning, you'll have to get on board.
- Get familiar with Conda. It will make working with Python significantly more pleasant. It allows you to try Python libraries in a controlled environment without damaging any existing environment.
- Get familiar with Jupyter notebooks. Like Swift playgrounds, they provide a means to quickly test all things Python especially when used in combination with Conda.

Challenges

For some chapters in this book, you'll need Keras or TensorFlow. Your challenges are to practice creating environments and installing packages by creating the **tfenv** and **kerasenv** environments.

> **Note**: If you skip these challenges, you can create the environments when you need them by importing their `.yaml` files into Anaconda Navigator; these files are located in the **starter** folder for this chapter.

Although these packages are in Anaconda's package list, TensorFlow's installation instructions say to use `pip` instead of `conda`.

Clone **mlenv** to create **tfenv**. Then install the `tensorflow` package in **tfenv**. Remember to use `pip install` instead of `conda install`.

Also `pip install` the `tfcoreml` package in **tfenv**; this is TensorFlow's tool for creating Core ML models. Unlike `turicreate`, which automatically installs `coremltools`, installing `tensorflow` doesn't automatically install `tfcoreml`. On the other hand, `tfcoreml` uses `coremltools`, so you'll see that being installed too.

Finally, clone **tfenv** to create **kerasenv**, and install the `keras` package in **kerasenv**. Again, use `pip install`, just to be safe.

Where to go from here?

You're all set to continue learning about machine learning for image classification, using Python tools. The next chapter shows you a few more Turi Create tricks. After that, you'll be ready to learn how to create your own deep learning model in Keras.

Chapter 5: Digging Deeper Into Turi Create

By Audrey Tam & Matthijs Hollemans

In this chapter, you'll use the SqueezeNet base model to train the HealthySnacks model, then explore more ways to evaluate its results. Then, you'll try to improve the model's accuracy, first with more iterations, then by tweaking some of the underlying Turi Create code. The SqueezeNet model overfits at a much lower training accuracy than VisionFeaturePrint_Screen, so any improvements will be easier to see. You'll also use the Netron tool to view the model — a SqueezeNet-based model has a lot more detail.

Getting started

You can continue to use the notebook from the previous chapter or start fresh with the notebook in this chapter's starter folder.

Transfer learning with SqueezeNet

You can skip all the data exploration cells in the previous chapter's notebook. Run these cells one by one — the cells for steps 2 and 3 include code to check you've actually gotten what you expect.

1. Import modules:

```
import turicreate as tc
import matplotlib.pyplot as plt
```

2. Load training and testing data and display lengths to make sure there's data:

```
train_data = tc.image_analysis.load_images("snacks/train",
                                           with_path=True)
len(train_data)
```

```
test_data = tc.image_analysis.load_images("snacks/test", with_path=True)
len(test_data)
```

3. Extract labels from the image paths and display label count values:

```
import os
train_data["label"] = train_data["path"].apply(
              lambda path: os.path.basename(os.path.split(path)[0]))

test_data["label"] = test_data["path"].apply(
              lambda path: os.path.basename(os.path.split(path)[0]))

train_data["label"].value_counts().print_rows(num_rows=20)
test_data["label"].value_counts().print_rows(num_rows=20)
```

4. Create the image classifier with `model="squeezenet_v1.1"` and `max_iterations=100`:

```
model = tc.image_classifier.create(train_data, target="label",
                                   model="squeezenet_v1.1",
                                   verbose=True, max_iterations=100)
```

Note: If you don't want to wait for your Mac to train this model, just load the pre-trained model:

```
model = tc.load_model("HealthySnacks.model")
```

When you run this cell, you'll be pleasantly surprised at how fast the feature extraction is. This is because SqueezeNet extracts only 1000 features from 227x227-pixel images, compared with VisionFeaturePrint_Screen's 2,048 features from 299x299 images. Then you'll be disappointed by the training and validation accuracy values:

```
| Iteration | Passes   | Step size | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+----------+-----------+--------------+-------------------+---------------------+
...
| 75        | 85       | 1.000000  | 44.102961    | 0.753883          | 0.640449            |
| 80        | 91       | 1.000000  | 46.631130    | 0.758915          | 0.647940            |
| 85        | 96       | 1.000000  | 48.946503    | 0.769416          | 0.685393            |
| 90        | 101      | 1.000000  | 51.243245    | 0.783636          | 0.666667            |
| 95        | 106      | 1.000000  | 53.890912    | 0.792606          | 0.681648            |
| 100       | 111      | 1.000000  | 56.254689    | 0.796762          | 0.670412            |
```

Note: You'll probably get slightly different training results than what are shown in this book. Recall that untrained models, in this case the logistic regression, are initialized with random numbers. This can cause variations between different training runs. Just try it again if you get a training accuracy that is much less than 66%. Advanced users of machine learning actually take advantage of these differences between training runs to combine multiple models into one big ensemble that gives more robust predictions.

Like Create ML, Turi Create randomly chooses 5% of the training data as validation data, so validation accuracies can vary quite a bit between training runs. The model might do better on a larger fixed validation dataset that you choose yourself.

Evaluate the model and display some metrics:

```
metrics = model.evaluate(test_data)
print("Accuracy: ", metrics["accuracy"])
print("Precision: ", metrics["precision"])
print("Recall: ", metrics["recall"])
```

No surprises here — accuracy is pretty close to the validation accuracy:

```
Accuracy:  0.6607142857142857
Precision:  0.6625686789687794
Recall:  0.6577568027210883
```

Getting individual predictions

So far, you've just repeated the steps from the previous chapter. The `evaluate()` metrics give you an idea of the model's overall accuracy but you can get a lot more information about individual predictions, especially those where the model is wrong, but has very high confidence that it's right. Knowing where the model is wrong can help you improve your training dataset.

Predicting and classifying

Turi Create models have other functions, in addition to `evaluate()`. Enter and run these commands in the next cell, and wait a while:

```
model.predict(test_data)
```

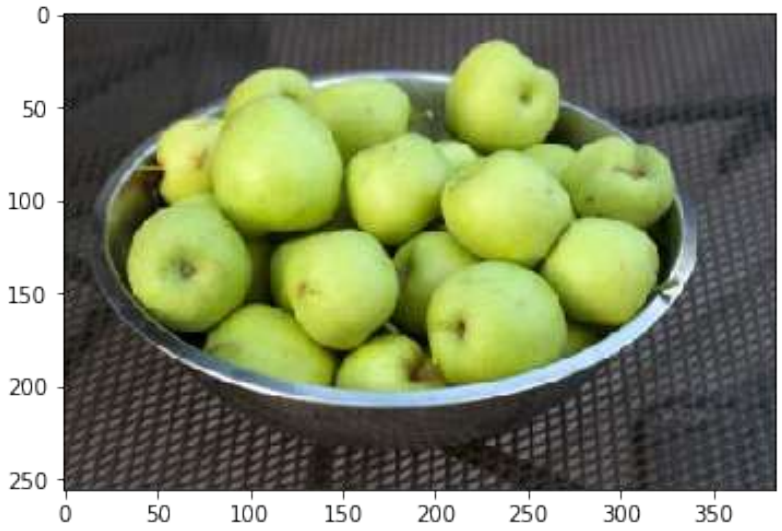
This command displays the actual prediction for each individual image:

```
['apple', 'grape', 'orange', 'orange', 'orange', 'apple', 'orange',
'apple', 'candy', 'apple', 'grape', 'apple', 'strawberry', 'apple',
'apple', 'carrot', 'candy', 'ice cream', 'apple', 'apple', 'apple', ...
```

The first prediction corresponds to the image from `test_data[0]`, the second to the image from `test_data[1]`, and so on. The first 50 test images are all apples, but the model classified the second image as "grape," so take a look at the image. Enter and run this command in the next cell:

```
plt.imshow(test_data[1]["image"].pixel_data)
```

This displays the second image — does it look like grapes?



grapes?

Maybe the model isn't really sure, either. Enter and run these commands, and wait a while:

```
output = model.classify(test_data)
output
```

The `classify()` function gets you the probability for each prediction, but only the highest-probability value, which is the model's confidence in the class it predicts:

class	probability
apple	0.4326385132863121
grape	0.699598450277612
orange	0.4148821902502113
orange	0.9300597134095988
orange	0.37817135281719916
apple	0.9915643139757563
orange	0.42620238429617097

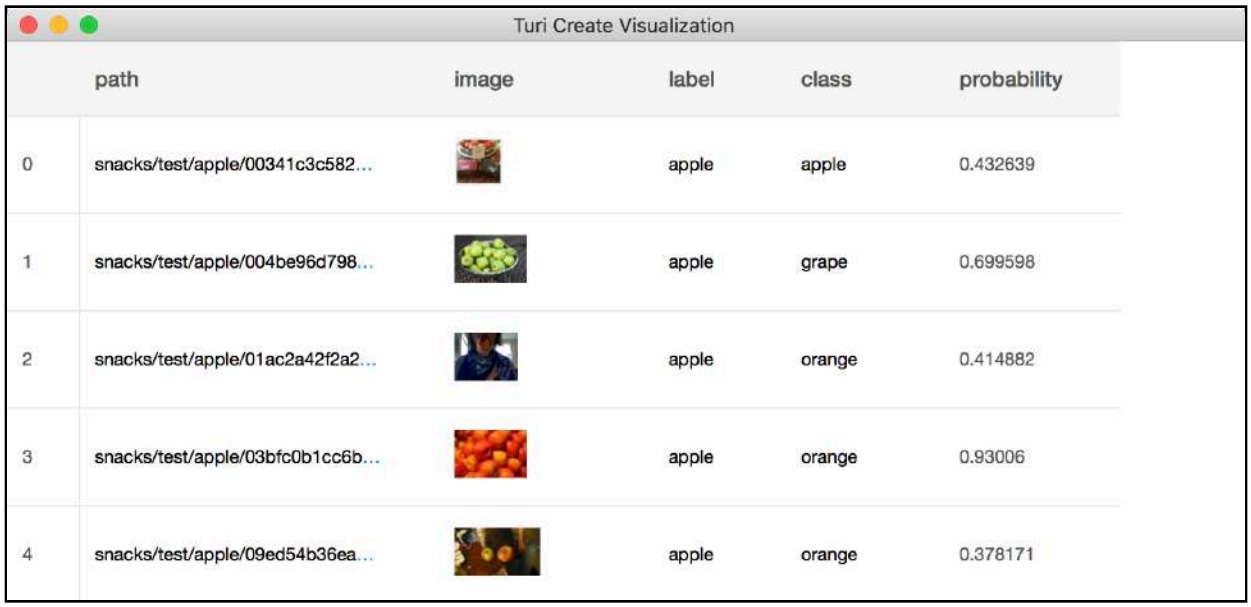
The head of the SFrame with classification results

So the model is 70% confident that the second image is "grape"! And 93% confident the fourth image is "orange"! But it's less than 50% confident about the other images it labelled "orange." It's' helpful to see the images that correspond to each prediction. Enter and run these commands:

```
imgs_with_pred = test_data.add_columns(output)
imgs_with_pred.explore()
```

The first command adds the `output` columns to the original `test_data` columns. Then you display the merged SFrame with `explore()`.

The **label** column is the correct class, and **class** is the model's highest-confidence prediction:

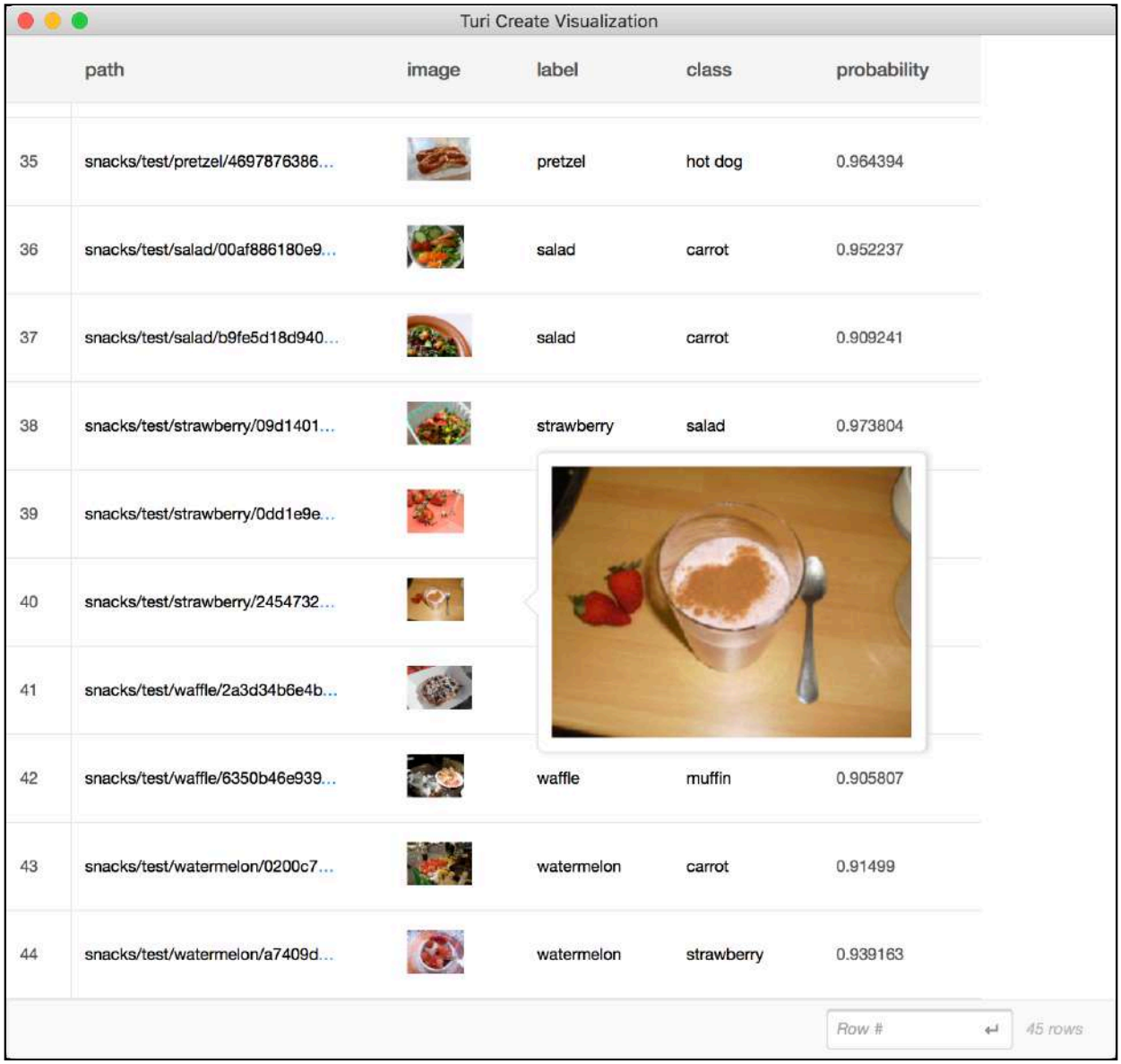


Visually inspecting the classification results

The most interesting images are the rows where the two labels disagree, but the probability is very high — over 90%, for example. Enter the following commands:

```
imgs_filtered = imgs_with_pred[(imgs_with_pred["probability"] > 0.9) &
                               (imgs_with_pred["label"] != imgs_with_pred["class"] )]
imgs_filtered.explore()
```

The first command filters the `test_data` + `output` SFrame to include only those rows with high-probability wrong predictions: The first term selects the rows whose probability column has a value greater than 90%, the second term selects the rows where the `label` and `class` columns are not the same. The subset of matching rows is saved into a new SFrame, then displayed.



Inspecting the filtered classification results

The true label of the highlighted image is "strawberry," but the model is 97% confident it's "juice," probably because the glass of milk(?) is much larger than the strawberries. You can learn a lot about how your model sees the world by looking at these confident-but-wrong predictions: Sometimes the model gets it completely wrong, but sometimes the predictions are actually fairly reasonable — for example, if the image contains more than one object — even if they're strictly speaking "wrong," since what was predicted wasn't the official label.

Sorting the prediction probabilities

Turi Create's `predict()` method can also give you the probability distribution for each image. Enter and run these lines, then wait a while:

```
predictions = model.predict(test_data, output_type='probability_vector')
print("Probabilities for 2nd image", predictions[1])
```

You add the optional argument `output_type` to get the **probability vector** — the probability for each of the 20 classes — for each image, then print the probabilities of the second image:

```
[0.20337662077520557, 0.010500386379535839, 2.8464920324200633e-07,
0.0034932724790819624, 0.0013391166287066811, 0.0005122369124003818,
5.118841868115829e-06, 0.699598450277612, 2.0208374302686123e-07,
7.164497444549948e-07, 2.584012081941193e-06, 5.5645094234565224e-08,
0.08066298157942492, 0.00021689939485918623, 2.30074608705137e-06,
3.6511378835730773e-10, 5.345215832976188e-05, 9.897270575019545e-06,
2.1477438456101293e-08, 0.00022540187389448156]
```

The probabilities are sorted alphanumerically by name of the class in the training set, so the first value is for "apple," the second is "banana," the third is "cake.".. ack — you need to add class labels to make this useful! Enter and run the following:

```
labels = test_data["label"].unique().sort()
preds = tc.SArray(predictions[1])
tc.SFrame({'preds': preds, 'labels': labels}).sort([('preds', False)])
```

First, you get the set of labels from the `test_data` SFrame, sort them so they match the order in the probability vector, and store the result in `labels`, which is an **SArray** — a Turi Create array. Then you create another SArray from the probability vector of the second image.

In the last line, you merge the two SArrays into an SFrame, then sort it on the `preds` column, in descending order (ascending = False). Here are the top five:

labels	preds
grape	0.699598450277612
apple	0.20337662077520557
orange	0.08066298157942492
banana	0.010500386379535839
candy	0.0034932724790819624

Top five probabilities for the second image.

So the model does at least give 20% confidence to "apple." Top-three or top-five accuracy is a fairer metric for a dataset whose images can contain multiple objects.

Increasing max iterations

So, is a validation accuracy of 67% good? Meh, not really. Turi Create knows it, too — at the end of the training output it says:

```
This model may not be optimal. To improve it, consider increasing
`max_iterations`.
```

Turi Create has recognized that this model still has some issues. Let's train again, this time with more iterations — 200 instead of 100:

```
model200 = tc.image_classifier.create(train_data, target="label",
                                      model="squeezenet_v1.1",
                                      verbose=True, max_iterations=200)
```

Note: Like Create ML, Turi Create has to extract the features again. It does not keep those feature vectors around — if it had, training the model again would be a lot quicker. If 100 iterations already took a very long time on your Mac, feel free to load the pre-trained model:

```
model200 = tc.load_model("HealthySnacks_200.model")
```

Now the final score is:

```
+-----------+----------+-----------+--------------+-------------------+---------------------+
| Iteration | Passes   | Step size | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+----------+-----------+--------------+-------------------+---------------------+
...
| 125       | 144      | 1.000000  | 63.705712    | 0.814039          | 0.745192            |
| 130       | 150      | 1.000000  | 66.223891    | 0.821598          | 0.735577            |
| 135       | 155      | 1.000000  | 68.512662    | 0.829158          | 0.735577            |
| 140       | 161      | 1.000000  | 71.103613    | 0.840389          | 0.745192            |
| 145       | 166      | 1.000000  | 73.339249    | 0.840173          | 0.725962            |
| 150       | 171      | 1.000000  | 75.898538    | 0.852916          | 0.730769            |
| 155       | 176      | 1.000000  | 78.181291    | 0.854428          | 0.740385            |
| 160       | 181      | 1.000000  | 80.502486    | 0.860907          | 0.764423            |
| 165       | 187      | 1.000000  | 83.075802    | 0.865227          | 0.754808            |
| 170       | 193      | 1.000000  | 85.603344    | 0.852700          | 0.740385            |
| 175       | 199      | 1.000000  | 88.082651    | 0.875810          | 0.721154            |
| 180       | 204      | 1.000000  | 90.362127    | 0.879698          | 0.730769            |
| 185       | 210      | 1.000000  | 92.892468    | 0.889417          | 0.735577            |
| 190       | 216      | 1.000000  | 95.360107    | 0.893521          | 0.735577            |
| 195       | 221      | 1.000000  | 97.611503    | 0.896544          | 0.730769            |
| 200       | 227      | 1.000000  | 100.106221   | 0.902376          | 0.735577            |
+-----------+----------+-----------+--------------+-------------------+---------------------+
```

The training accuracy is now 90%! This means on the training set of 4582 examples it gets 10% wrong (as opposed to 21% before). That's pretty good, but remember that you shouldn't put too much faith in the training accuracy. The validation accuracy is also higher, at 74%. Actually, validation accuracy fluctuates a lot, between 100 and 200 iterations. It's actually highest — 76% — at 16 iterations, but seems to be hovering around 73%.

This particular run did an exceptional job on the validation accuracy. However, this only means the validation dataset was a pretty good match for the training dataset.

Enter and run the usual code to evaluate the model and display the metrics:

```
metrics200 = model200.evaluate(test_data)
print("Accuracy: ", metrics200["accuracy"])
```

```
print("Precision: ", metrics200["precision"])
print("Recall: ", metrics200["recall"])
```

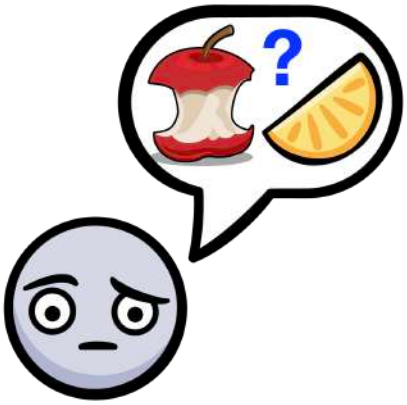
Evaluating this model on the test dataset produces metrics around 64%, not 74%:

```
Accuracy:  0.6428571428571429
Precision:  0.6447263553620883
Recall:  0.6406734693877552
```

Overfitting has a bad rap, and it's certainly an issue you'll run into when you start training your own models. But overfitting isn't necessarily a bad thing to experience, as it means that your model still has capacity to learn more. It's just learning the wrong things, and techniques such as regularization will help your model to stay on the right path.

The sweet spot for this model seems to be about 135 iterations. If you train for longer, then the validation accuracy starts to drop and the model becomes worse. Unfortunately, Turi does not save the iteration of the model with the best validation accuracy, only the very last iteration, and so you'll have to train again with `max_iterations=135` to get the best possible result.

Confusing apples with oranges?



A picture says more than these numbers, and a really useful visualization of how well the model does is the **confusion matrix**. This plots the predicted classes versus the images' real classes, so you can see where the model tends to make its mistakes. In the previous chapter, you ran this command:

```
print("Confusion Matrix:\n", metrics["confusion_matrix"])
```

This displayed a table:

```
+--------------+-----------------+-------+
| target_label | predicted_label | count |
+--------------+-----------------+-------+
|    cookie    |      juice      |   1   |
|    carrot    |    watermelon   |   1   |
|   pretzel    |     pretzel     |   14  |
|     cake     |    ice cream    |   2   |
|  pineapple   |      carrot     |   1   |
|   doughnut   |      muffin     |   1   |
|    muffin    |     doughnut    |   7   |
```

The `target_label` column shows the real class, while `predicted_label` has the class that was predicted, and `count` is how many of this particular mistake were made. The table shows the model predicted "muffin" 7 times when the image was really "doughnut", predicted "cake" once when the image was really "ice cream", and so on. However, presented this way, the confusion matrix doesn't look much like a matrix, and we promised to show you how to get a better visualization.

Start by entering and running the following code:

```
import numpy as np
import seaborn as sns

def compute_confusion_matrix(metrics, labels):
    num_labels = len(labels)
    label_to_index = {l:i for i,l in enumerate(labels)}

    conf = np.zeros((num_labels, num_labels), dtype=np.int)
    for row in metrics["confusion_matrix"]:
        true_label = label_to_index[row["target_label"]]
        pred_label = label_to_index[row["predicted_label"]]
        conf[true_label, pred_label] = row["count"]

    return conf

def plot_confusion_matrix(conf, labels, figsize=(8, 8)):
    fig = plt.figure(figsize=figsize)
    heatmap = sns.heatmap(conf, annot=True, fmt="d")
    heatmap.xaxis.set_ticklabels(labels, rotation=45,
                                 ha="right", fontsize=12)
    heatmap.yaxis.set_ticklabels(labels, rotation=0,
                                 ha="right", fontsize=12)
    plt.xlabel("Predicted label", fontsize=12)
    plt.ylabel("True label", fontsize=12)
    plt.show()
```

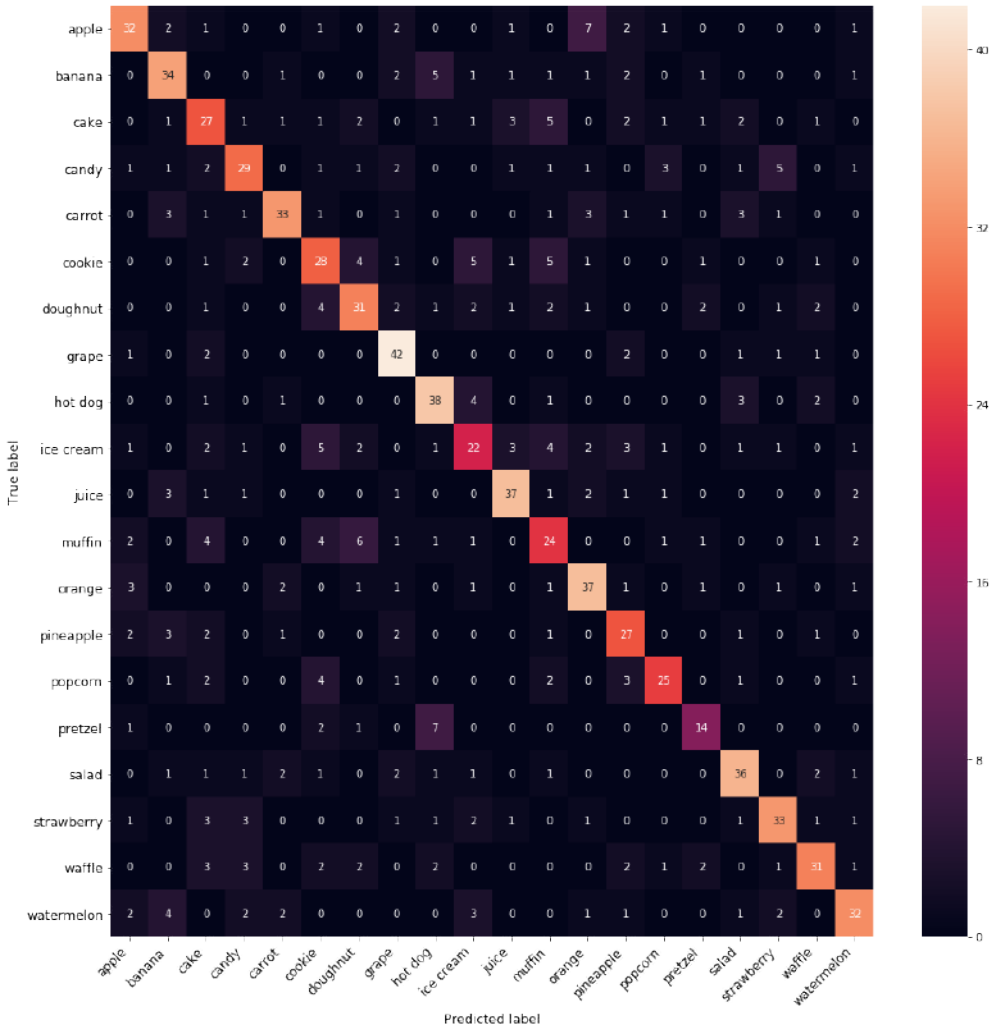
You define two new functions: one to compute the confusion matrix and one to draw it. The `compute_confusion_matrix()` looks at all the rows in the `metrics["confusion_matrix"]` table, and fills up a 2D-array with the counts of each pair of labels. It uses the NumPy package for this. Then, `plot_confusion_matrix()` takes this

NumPy array, and plots it as a *heatmap* using Seaborn, a plotting package that adds useful plot types to matplotlib. You installed Seaborn when you created the `turienv` environment in the previous chapter.

Now, enter and run these commands to call these functions:

```
conf = compute_confusion_matrix(metrics200, labels)
plot_confusion_matrix(conf, labels, figsize=(16, 16))
```

And enjoy the display!



The confusion matrix

A heatmap shows small values as “cool” colors — black and dark purple — and large values as “hot” colors — red to pink to white. The larger the value, the brighter it gets. In the confusion matrix, you expect to see a lot of high values on the diagonal, since these are the correct matches. The row for the "pretzel" class shows 14 correct matches and 11 wrong ones. The wrong predictions are one “apple,” two “cookie,” one

"doughnut," and seven "hotdog." Notice that apples often get mistaken for oranges, and cookie, doughnut, and muffin also get mixed up often.

The confusion matrix is very useful because it shows potential problem areas for the model. From this particular confusion matrix, it's clear the model has learned a great deal already, since the diagonal really stands out, but it's still far from perfect. Ideally, you want everything to be zero except the diagonal. It may be a little misleading from the picture since at first glance it appears that there aren't that many mistakes. But all the small numbers in the dark squares add up to 340 misclassified images out of 952 total, or 36% wrong.

Keep in mind that some categories have more images than others. For example, pretzel has only 25 images in the test set, while most of the other classes have 50, so it will never have as many correct matches. Still, it only scores 14 correct (56%). so overall the model actually does poorly on pretzels.

Computing recall for each class

Turi Create's `evaluate()` function gives you the overall test dataset accuracy but, as mentioned in the AI Ethics section of the first chapter, accuracy might be much lower or higher for specific subsets of the dataset. With a bit of code, you can get the accuracies for the individual classes from the confusion matrix:

```
for i, label in enumerate(labels):
    correct = conf[i, i]
    images_per_class = conf[i].sum()
    print("%10s %.1f%%" % (label, 100. * correct/images_per_class))
```

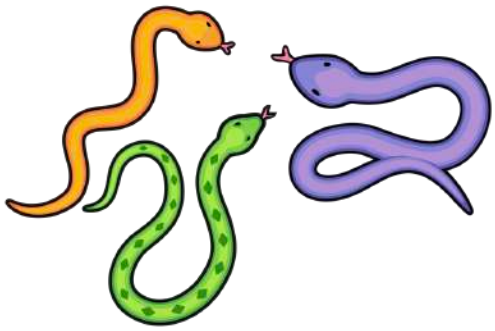
For each row of the confidence matrix, the number on the diagonal is how many images in this class that the model predicted correctly. You're dividing this number by the sum over that row, which is the total number of test images in that class. This gives you the percentage of each class that the model classified correctly — for example, how many "apple" images did the model find among the total number of "apple" images? Looking back at the definitions of *precision* and *recall*, this value is the **recall** metric for each class:

```
     apple 64.0%
    banana 68.0%
      cake 54.0%
     candy 58.0%
    carrot 66.0%
    cookie 56.0%
  doughnut 62.0%
     grape 84.0%
   hot dog 76.0%
 ice cream 44.0%
```

```
      juice 74.0%
     muffin 50.0%
     orange 74.0%
  pineapple 67.5%
    popcorn 62.5%
    pretzel 56.0%
      salad 72.0%
 strawberry 67.3%
     waffle 62.0%
 watermelon 64.0%
```

The best classes are grape (84% correct) and hot dog (76%). At 74%, juice and orange are also good. The worst performing classes are ice cream (44%), muffin (50%), cake (54%), and pretzel (56%). These would be the classes to pay attention to, in order to improve the model — for example, by gathering more or better training images for these classes.

Wrangling Turi Create code



One of the appealing benefits of Turi Create is that, once you have your data in an SFrame, it takes only a single line of code to train the model. The downside is that the Turi Create API gives you only limited control over what it does. Fortunately, Turi Create is open source, so you can look inside to see what it does, and even hack around some of its limitations.

We've already briefly mentioned **hyperparameters**. This is simply a fancy name for the configuration settings for your model. The things that the model learns from the training data are called "parameters"; the things you configure by hand, which don't get changed by training, are the "hyperparameters."

A typical hyperparameter that machine learning practitioners like to play with is the amount of **regularization** that's being used by the model. Regularization helps to prevent overfitting, but the Turi Create `image_classifier.create()` function doesn't provide access to change this hyperparameter.

Since overfitting seemed to be an issue for our model, it will be instructive to play with this regularization setting, whether Turi Create likes it or not.

The code for `tc.image_classifier.create()` is in the file `turicreate/src/unity/python/turicreate/toolkits/image_classifier/image_classifier.py` in the GitHub repo at github.com/apple/turicreate. You're simply going to copy-paste some of that code into the notebook, and play with the hyperparameters.

Using a fixed validation dataset

Turi Create extracts a random validation dataset from the training dataset — 5% of the images. The problem with using a small random validation dataset is that sometimes you get great results, but only because — this time! — the validation dataset just happens to be in your favor. Now that you have more control over the code (mwah hah hah!), you can use your own fixed validation set, to control your experiments better and get reproducible results. You can run a few different experiments with the hyperparameters, and properly compare them to each other. If you were to use a different validation set each time, then the variation in the chosen images could obscure the effect of the changed hyperparameter. This is also why you don't use the test set for validation.

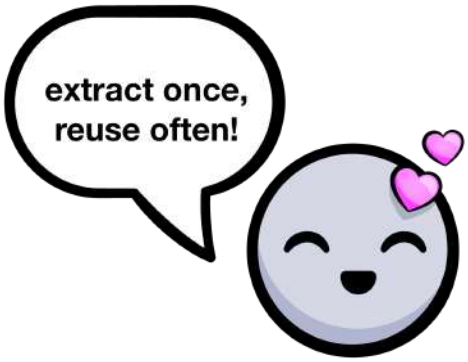
The snacks dataset already comes with a "val" folder containing images for this purpose. Load these images into their own SFrame, using the same code as before:

```
val_data = tc.image_analysis.load_images("snacks/val", with_path=True)
val_data["label"] = val_data["path"].apply(lambda path:
      os.path.basename(os.path.split(path)[0]))
len(val_data)
```

The last statement should output **955**, 3 more images than in `test_data`, and a lot more than 5% of the 4838 train_data images.

Saving the extracted features

Before you start playing around with the different regularization parameters, wouldn't it be nice if there was a way we could save time during the training phase, and not have to continuously regenerate the features extracted by SqueezeNet? Well, as promised, in this section, you'll learn how to save the intermediate SFrame to disk, and reload it, just before experimenting with the classifier.



Note: If you don't want to wait for the feature extraction, just load the features from the current folder:

```
extracted_train_features = tc.SFrame("extracted_train_features.sframe")
extracted_val_features = tc.SFrame("extracted_val_features.sframe")
```

To train the model, you first have to load the pre-trained SqueezeNet model, and grab its feature extractor:

```
from turicreate.toolkits import _pre_trained_models
from turicreate.toolkits import _image_feature_extractor

ptModel = _pre_trained_models.MODELS["squeezenet_v1.1"]()
feature_extractor = _image_feature_extractor.MXFeatureExtractor(ptModel)
```

`MXFeatureExtractor` is an object from the MXNet machine learning framework that Turi Create is built on. In Python, names starting with an underscore are considered to be private, but you can still import them. Next, enter and run this code statement:

```
train_features = feature_extractor.extract_features(train_data,
                                                    "image", verbose=True)
```

You're using the `MXFeatureExtractor` object to extract the SqueezeNet features from the training dataset. This is the operation that took the most time when you ran `tc.image_classifier.create()`. By running this separately now, you won't have wait for feature extraction every time you want to train the classifier. Next, enter and run this code statement:

```
extracted_train_features = tc.SFrame({
    "label": train_data["label"],
    '__image_features__': train_features,
    })
```

Here, you're just combining the features of each image with its respective label into a new SFrame. This is worth saving for later use! Enter and run this code statement:

```
extracted_train_features.save("extracted_train_features.sframe")
```

You're saving `extracted_train_features` to a file. The next time you want to do more training with these same features, you can simply load the SFrame again, which takes a fraction of the time it took to extract the features:

```
# Run this tomorrow or next week
extracted_train_features = tc.SFrame("extracted_train_features.sframe")
```

Inspecting the extracted features

Let's see what these features actually look like — enter and run this command:

```
extracted_train_features.head()
```

__image_features__	label
[6.1337385177612305, 10.12844181060791, ...	apple
[9.666999816894531, 14.665328025817871, ...	apple
[10.662524223327637, 15.472965240478516, ...	apple
[12.159001350402832, 11.231389045715332, ...	apple

The head of the extracted features table

Each row has the extracted features for one training image. The `__image_features__` column contains a list with numbers, while the `label` column has the corresponding class name for this row. Enter and run this command:

```
extracted_train_features[0]["__image_features__"]
```

This shows you what a feature vector looks like — it prints something like:

```
array('d', [6.1337385177612305, 10.12844181060791, 13.025101661682129,
7.931194305419922, 12.03809928894043, 15.103202819824219,
12.722893714904785, 10.930903434753418, 12.778315544128418,
14.208030700683594, 16.8399658203125, 11.781684875488281,
18.9950008392334, 17.461009979248047, 18.71086311340332, ...
```

This is a list of 1,000 numbers — use the `len()` function to verify this. They all appear to be numbers between 0 and about 30. What do they represent? I have no idea, but they are features that SqueezeNet has determined to be important — how long, round,

square, orange, etc the objects are. All that matters is that you can train a logistic classifier to learn from these features.

In the same way, extract the features for the images from the validation dataset, and save it to disk:

```
val_features = feature_extractor.extract_features(val_data,
                                        "image", verbose=True)

extracted_val_features = tc.SFrame({
    "label": val_data["label"],
    '__image_features__': val_features,
    })

extracted_val_features.save("extracted_val_features.sframe")
```

Training the classifier with regularization

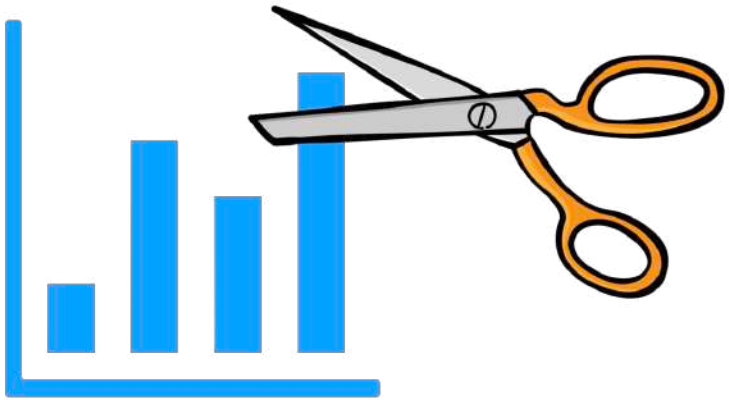
Now you're ready to train the classifier! Enter and run this statement:

```
lr_model = tc.logistic_classifier.create(extracted_train_features,
                                features=['__image_features__'],
                                target="label",
                                validation_set=extracted_val_features,
                                max_iterations=200,
                                seed=None,
                                verbose=True,
                                l2_penalty=10.0,
                                l1_penalty=0.0,
                                convergence_threshold=1e-8)
```

This is the Turi Create code that creates and trains the logistic regression model using the `extracted_train_features` SFrame as the input data, and `extracted_val_features` for validation. You've actually added three additional arguments to this function call that are not in the original Turi source code: `l2_penalty`, `l1_penalty` and `convergence_threshold`. Setting the `convergence_threshold` to a very very small value means that the training won't stop until it has done all 200 iterations. `l2_penalty` and `l1_penalty` are hyperparameters that add *regularization* to reduce overfitting.

What's regularization? Recall that a model learns parameters — also called weights or coefficients — for combining feature values, to maximize how many training data items it classifies correctly. Overfitting can happen when the model gives too much weight to some features, by giving them very large coefficients. Setting `l2_penalty` greater than 0 penalizes large coefficients, encouraging the model to learn smaller coefficients. Higher values of `l2_penalty` reduce the size of coefficients, but can also reduce the training accuracy.

Setting `l1_penalty` greater than 0 also penalizes large coefficients; in addition, it discards features that have very small coefficients, by setting their coefficients to 0. Typically, you'd use either `l2_penalty` or `l1_penalty`, but not both in the same training session.



In my training session, the model still overfits, even with these settings, but the training accuracy doesn't race off to 95% or 100% anymore.

```
+-----------+----------+-----------+--------------+-------------------+---------------------+
| Iteration | Passes   | Step size | Elapsed Time | Training Accuracy | Validation Accuracy |
+-----------+----------+-----------+--------------+-------------------+---------------------+
...
| 180       | 226      | 0.500000  | 132.691153   | 0.787102          | 0.650262            |
| 185       | 231      | 0.500000  | 136.600216   | 0.786689          | 0.648168            |
| 190       | 239      | 1.000000  | 141.066780   | 0.785862          | 0.650262            |
| 195       | 244      | 1.000000  | 144.398265   | 0.787929          | 0.648168            |
| 200       | 249      | 1.000000  | 147.971076   | 0.789169          | 0.647120            |
```

Now that you're not having to extract features for each training session, training is fast, so you can train the classifier several times, trying out different values for `l2_penalty` and `l1_penalty`: this is called **hyperparameter tuning**. Selecting the correct hyperparameters for your training procedure can make a big difference in the quality of the model you end up with. The validation accuracy gives you an indication of the effect of these hyperparameters.

Hyperparameter tuning is more trial and error than science, so play with these hyperparameters to get a feeling for how they affect your model. Try setting `l2_penalty` to 100: you'll note that the training accuracy won't go over 65% or so, as now you're punishing the model too hard.

Saving the model

Finally, to turn your model into a valid `ImageClassifier` object that you can export to Core ML, do:

```
from turicreate.toolkits.image_classifier import ImageClassifier

state = {
    'classifier': lr_model,
    'model': ptModel.name,
    'max_iterations': lr_model.max_iterations,
    'feature_extractor': feature_extractor,
    'input_image_shape': ptModel.input_image_shape,
    'target': lr_model.target,
    'feature': "image",
    'num_features': 1,
    'num_classes': lr_model.num_classes,
    'classes': lr_model.classes,
    'num_examples': lr_model.num_examples,
    'training_time': lr_model.training_time,
    'training_loss': lr_model.training_loss,
}
model = ImageClassifier(state)
```

You combine the base model with the classifier you trained into the `state` structure, and create an `ImageClassifier` object from this. Then you can save the model as a Turi Create model:

```
model.save("HealthySnacks_regularized.model")
```

Or export a Core ML model:

```
model.export_coreml("HealthySnacks_regularized.mlmodel")
```

To learn more about the model, run the following:

```
model
```

This shows you some high-level information about the model and its training:

```
Class                                  : ImageClassifier

Schema
------
Number of classes                      : 20
Number of feature columns              : 1
Input image shape                      : (3, 227, 227)

Training summary
----------------
Number of examples                     : 4838
Training loss                          : 3964.3015
Training time (sec)                    : 147.8538
```

Training loss — the overall error over the training dataset — changes when you change the hyperparameters. Enter and run this to see a bit more information:

```
model.classifier
```

This shows you information about the classifier portion of the model:

```
Class                          : LogisticClassifier

Schema
------
Number of coefficients         : 19019
Number of examples             : 4838
Number of classes              : 20
Number of feature columns      : 1
Number of unpacked features    : 1000

Hyperparameters
---------------
L1 penalty                     : 0.0
L2 penalty                     : 10.0

Training Summary
----------------
Solver                         : lbfgs
Solver iterations              : 200
Solver status                  : Completed (Iteration limit reached).
Training time (sec)            : 147.8538

Settings
--------
Log-likelihood                 : 3964.3015

Highest Positive Coefficients
-----------------------------
(intercept)                    : 0.3801
(intercept)                    : 0.2549
(intercept)                    : 0.1376
(intercept)                    : 0.0923
(intercept)                    : 0.0851

Lowest Negative Coefficients
----------------------------
(intercept)                    : -0.2885
(intercept)                    : -0.2884
(intercept)                    : -0.2365
(intercept)                    : -0.1734
(intercept)                    : -0.1229
```

This information is mostly useful for troubleshooting or when you're just curious about how the logistic regression classifier works.

Notable is *Number of coefficients* — 19,019 — the number of parameters this model learned in order to classify images of snacks into the 20 possible categories. Here's

where that number comes from: each input feature vector has 1,000 numbers, and there are 20 possible outputs, so that is 1,000 × 20 = 20,000 numbers, plus 20 "bias" values for each output, making 20,020 coefficients.

However, if there are 20 possible classes, then you actually only need to learn about 19 of those classes, giving 19,019 coefficients. If the prediction is none of these classes, then it must be the 20th class. Interestingly, if you look at the .mlmodel file in Netron, the *innerProduct* layer that implements the logistic regression does have all 20,020 parameters.

Under the **Settings** heading, *Log-likelihood* is the more mathematical term for *Training loss*. Below this are the highest and lowest coefficients — remember, the purpose of the regularization hyperparameter is to reduce the size of the coefficients. To compare with the coefficients of the original no-regularization model, enter and run these lines:

```
no_reg_model = tc.load_model("HealthySnacks.model")
no_reg_model.classifier
```

You reload the pre-trained model, and inspect its classifier. This model had higher training accuracy, so *Log-likelihood* aka *Training loss* is lower: 2,400. As you'd expect, its highest and lowest coefficients are larger — in absolute value — than the model with regularization:

```
Settings
--------
Log-likelihood                : 2400.3284

Highest Positive Coefficients
-----------------------------
(intercept)                   : 0.3808
(intercept)                   : 0.3799
(intercept)                   : 0.1918
__image_features__[839]       : 0.1864
(intercept)                   : 0.15

Lowest Negative Coefficients
----------------------------
(intercept)                   : -0.3996
(intercept)                   : -0.3856
(intercept)                   : -0.3353
(intercept)                   : -0.2783
__image_features__[820]       : -0.1423
```

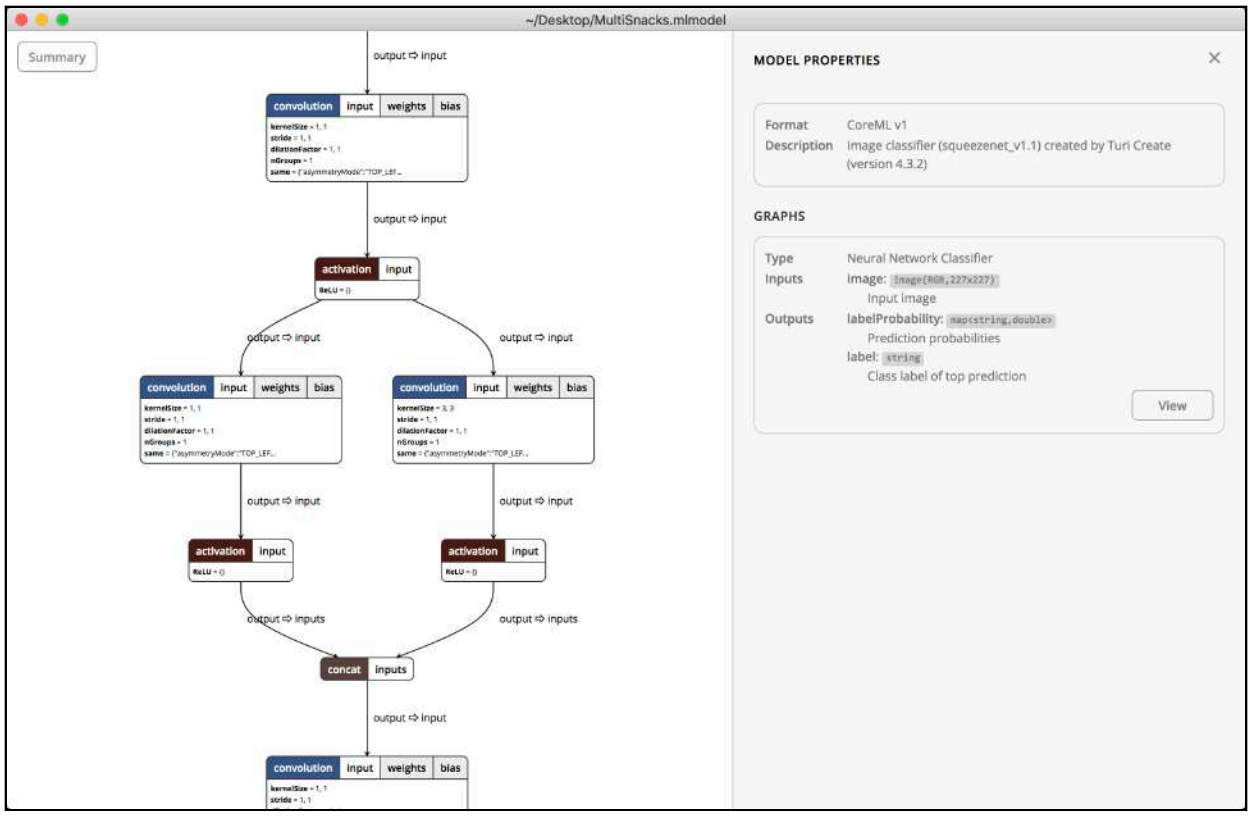
In the next chapter we'll talk more about what all of this means, as you'll be writing the code there to train your own logistic regression from scratch, as well as a complete neural network that will outperform Turi Create's SqueezeNet-based model.

A peek behind the curtain

SqueezeNet and VisionFeaturePrint_Screen are *convolutional neural networks*. In the coming chapters, you'll learn more about how these networks work internally, and you'll see how to build one from scratch. In the meantime, it might be fun to take a peek inside your Core ML model.

There is a cool free tool called Netron (github.com/lutzroeder/Netron) that creates a nice visualization of the model architecture. On the GitHub page, scroll down to the **Install** instructions, and click the **macOS Download** link. On the next page, click the **Netron-x.x.x.dmg** link, then run this file to install Netron.

Open your .mlmodel file in Netron:



Using Netron to examine the .mlmodel file

Scroll down the diagram to see all the transformation stages that go into this pipeline. The input image is at the top, followed by convolutions, activations, pooling, and so on. These are the names of the different types of transformations — or layers — used by this kind of neural network. Notice how this pipeline sometimes branches and then comes back together again — that's the "squeeze" feature that gives SqueezeNet its name.

Click on a block to learn more about its configuration, its inputs and its output. The Summary button opens the details pane on the right. At the very end of the pipeline is an *innerProduct* layer followed by something called a *softmax* — these two blocks make up the logistic classifier. Everything up until the *flatten* block is the SqueezeNet feature extractor.

In Chapters 6 and 7, you'll learn all about what these different kinds of layers do, but for now we suggest that you spend a few minutes playing with Netron to get a rough idea of what these models look like on the inside.

Netron works with any Core ML model, as well as models from many other machine learning frameworks. If you downloaded a model from Apple's website in the last chapter, also take a look at that. It should look quite similar to this one, as all neural networks are very alike at their core. Often what is different is the number of layers and the branching structure.

> **Note**: Many of Apple's models such as `VisionFeaturePrint_Screen` are included in iOS 12 and do not get bundled into the .mlmodel file. The .mlmodel file itself doesn't contain any of the `VisionFeaturePrint_Screen` layers. For customized models based on these built-in feature extractors, Netron can't show you anything more than what you see in Xcode's description: inputs, outputs, metadata. The internal architecture of these models remains a mystery and a secret.

Key points

- In this chapter, you've gotten a taste of training your own Core ML model with Turi Create. In fact, this is exactly how the models were trained that you used in the previous chapter. Turi Create is pretty easy to use, especially from a Jupyter notebook. It only requires a little bit of Python code. However, we weren't able to create a super accurate model. This is partly due to the limited dataset.
- More images is better. We use 4,800 images, but 48,000 would have been better, and 4.8 million would have been even better. However, there is a real cost associated with finding and annotating training images, and for most projects, a few hundred images or at most a few thousand images per class may be all you can afford. Use what you've got — you can always retrain the model at a later date once you've collected more training data. Data is king in machine learning, and who has the most of it usually ends up with a better model.

- Another reason why Turi Create's model wasn't super is that SqueezeNet is a small feature extractor, which makes it fast and memory-friendly, but this also comes with a cost: It's not as accurate as bigger models. But it's not just SqueezeNet's fault — instead of training a basic logistic regression on top of SqueezeNet's extracted features, it's possible to create more powerful classifiers too.
- Turi Create does not offer a lot of control over tweaking the training process, so we can't get a good grip on the overfitting. Lastly, Turi Create does not allow us to fine-tune the feature extractor or use data augmentation. Those are more advanced features, and they result in slower training times, but also in better models.

In the next chapter, we'll look at fixing all of these issues when we train our image classifier again, but this time using Keras. You'll also learn more about what all the building blocks are in these neural networks, and why we use them in the first place.

Challenges

Binary classifier

Remember the healthy/unhealthy snacks model? Try to train that binary classifier using Turi Create. The approach is actually very similar to what you did in this chapter. The only difference is that you need to assign the label "healthy" or "unhealthy" to each row in the training data SFrame.

```
healthy = [
    'apple',
    'banana',
    'carrot',
    'grape',
    'juice',
    'orange',
    'pineapple',
    'salad',
    'strawberry',
    'watermelon',
]

unhealthy = [
    'cake',
    'candy',
    'cookie',
    'doughnut',
    'hot dog',
    'ice cream',
    'muffin',
    'popcorn',
    'pretzel',
```

```
    'waffle',
]

train_data["label"] =
  train_data["path"].apply(lambda path: "healthy"
      if any("/" + class_name in path for class_name in healthy)
                                      else "unhealthy")
test_data["label"] =
  test_data["path"].apply(lambda path: "healthy"
      if any("/" + class_name in path for class_name in healthy)
                                      else "unhealthy")
```

First, you assign each class into a `healthy` or `unhealthy` array — there are 10 classes in each array.

Then, you set each image's `label` column to `"healthy"` or `"unhealthy"`, depending on which array the image's path name is in.

The result is, you've divided 20 classes of images into two classes, based on the name of the subdirectory they're in.

> **Note**: The process to do this same exercise in Create ML is much more manual. You'd have to create a new `train` folder with subfolders `healthy` and `unhealthy`, then copy or move all the images from each of the 20 food-labelled folders into the correct `healthy` or `unhealthy` folder. You'd do this either in Finder or Terminal.

Verify that the resulting model gets about 80% accuracy on the test dataset.

You may wonder why you can't use the multi-class snacks model for this, then simply look if the predicted category is in the list of healthy or unhealthy classes. This is possible but, by training from scratch on just these two categories, the model has a chance to learn what healthy/unhealthy means, and it might use a more intricate rule than just "this is in the list of healthy categories."

If you want to be sure, use the 20-class model to `classify()` the healthy/unhealthy test dataset, and merge its `output` with `test_data` as before. The **label** column contains "healthy" or "unhealthy," while the **class** column contains "apple," "banana," etc. Use `filter_by(healthy, 'class')` to find images the model predicts to be in a class listed in the `healthy` array. Then filter these images with `filter_by(['unhealthy'], 'label')` to find images that are really in unhealthy classes. Manually calculate the accuracy of the 20-class model in predicting healthy/unhealthy. I got 47%.

ResNet50-based model

Train the 20-class classifier using the ResNet-50 model and see if that gets a better validation and test set score. Use `model_type="resnet-50"` when creating the classifier object. How many FPS does this get in the app?

Use another dataset

Create your own dataset from Google Open Images or some other image source. I suggest keeping the number of categories limited.

Chapter 6: Training with Keras

You've seen Create ML and you've seen Turi Create, now get ready to meet to Keras. In this chapter, you'll create your model from scratch using some of the lowest-level APIs available.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 7: Beyond Image Classification

Image classification has many applications; however, the ability to also detect specific objects unlocks a whole host of possibilities. In this chapter, you'll learn how to perform localization, object detection and segmentation on images.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 8: Sequence Classification

By Chris LaPollo

You worked exclusively with images throughout the first section of this book, and for good reason — knowing how to apply machine learning to images lets you add many exciting and useful features to your apps. Techniques like classification and object detection can help you answer questions like "Is this snack healthy?" or "Which of these objects is a cookie?"

But you've focused on *individual* images — even when processing videos, you processed each frame individually with complete disregard for the frames that came before or after it. Given the following *series* of images, can the techniques you've learned so far tell me where my cookies went?

The Case of the Disappearing Cookies

Each of the above images tells only part of the story. Rather than considering them individually, you need to reason over them as a *sequence*, applying what you see in earlier frames to help interpret later ones.

There are many such tasks that involve working with sequential data, such as:

- Extracting meaning from videos. Maybe you want to make an app that translates sign language, or search for clips based on the events they depict.
- Working with audio, for example converting speech to text, or songs to sheet music.
- Understanding text, such as these sentences you've been reading, which are sequences of words, themselves sequences of letters (assuming you're reading this in a language that uses letters, that is).
- And countless others. From weather data to stock prices to social media feeds, there are endless streams of sequential data.

With so many types of data and almost as many techniques for working with it, this chapter can't possibly cover everything. You'll learn ways to deal with text in later chapters, and some of the techniques shown here are applicable to multiple domains. But to keep things practical, this chapter focuses on a specific type of sequence classification — human activity detection. That is, using sensor data from a device worn or held by a person to identify what that person is physically doing. You've probably already experienced activity detection on your devices, maybe checking your daily step count on your iPhone or closing rings on your Apple Watch. Those just scratch the surface of what's possible.

In this chapter, you'll learn how to collect sensor data from Apple devices and prepare it for use training a machine learning model. You'll use Turi Create's task-focused API for activity detection to build a neural network that recognizes user activity from device motion data, and you'll use your trained neural net to recognize player actions in a game. The game you'll make is similar to the popular Bop It toy, but instead of calling out various physical bits to bop and twist, it will call out gestures for the player to make with their iPhone. Perform the correct action before time runs out!

We chose this project because collecting data and testing it should be comfortably within the ability of most readers. However, you can use what you learn here for more than just gesture recognition — these techniques let you track or react to any activity identifiable from sensor data available on an Apple device.

Modern hardware comes packed with sensors — depending on the model, you might have access to an accelerometer, gyroscope, pedometer, magnetometer, altimeter or GPS. You may even have access to the user's heart rate!

With so much data available, there are countless possibilities for behaviors you can detect, including sporadic actions like standing up from a chair or falling off a ladder, as well as activities that occur over longer durations like jogging or sleeping. And machine learning is the perfect tool to make sense of it all. But before you can fire up those neural nets, you'll need a dataset to train them.

Building a dataset

So you've got an app you want to power using machine learning. You do the sensible thing and scour the internet for a suitable, freely available dataset that meets your needs. You try tools like Google Dataset Search, check popular data science sites like Kaggle, and exhaust every keyword search trick you know. If you find something — great, move on to the next section! But if your search for a dataset turns up nothing, all is not lost — you can build your own.

Collecting and labeling data is the kind of thing professors make their graduate students do — time consuming, tedious work that may make you want to cry. When labeling human activity data, it's not uncommon to record video of the activity session, go through it manually to decide when specific activities occur, and then label the data using timecodes synced between the data recordings and the video. That may sound like fun to some people, but those people are wrong and should never be trusted.

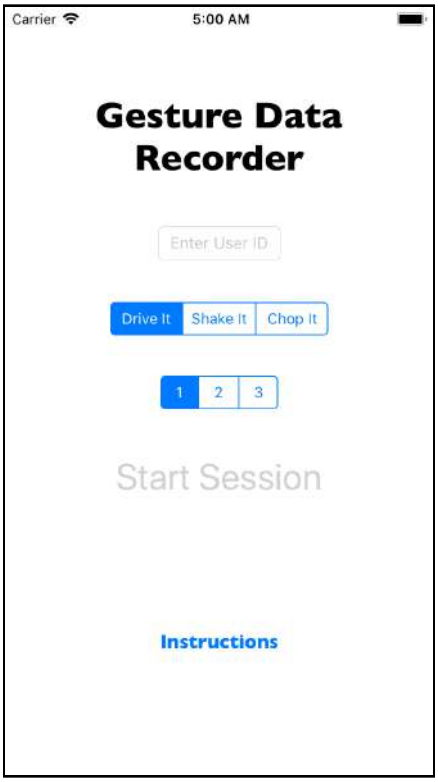
This chapter takes a different approach — the data collection app *automatically* adds labels. They may not be as exact — manual labeling lets you pinpoint precise moments when test subjects begin or end an activity — but in many cases, they're good enough.

To get started, download the resources for this chapter if you haven't already done so, and open the **GestureDataRecorder** starter project in Xcode.

> **Note**: The chapter resources include data files you can use unchanged, so you aren't *required* to collect more here. However, the experience will help later when working on your own projects. Plus, adding more data to the provided dataset should improve the model you make later in the chapter.

Take a look through the project to see what's there. **ViewController.swift** contains most of the app's code, and it's the only file you'll be asked to change.

If you run the app now, it will *seem* like it's working but it won't actually collect or save any data. The following image shows the app's interface:



Gesture Data Recorder app interface

GestureDataRecorder probably won't win any design awards, but that's OK — it's just a utility app that records sensor data. Users enter their ID, choose what activity and how many short sessions of that activity to record, and then hit **Start Session** to begin collecting data. The app speaks instructions to guide users through the recording process. And the **Instructions** button lets users see videos demonstrating the activities.

> **Note**: For some datasets, it may be better to randomize activities during a session, rather than having users choose one for the entire thing. My test subjects didn't seem to enjoy having to pay that much attention, though.

Why require a user ID? You'll learn more about this later, but it's important to be able to separate samples in your dataset by their sources. You don't need *specific* details about people, like their names — in fact, identifying details like that are often a bad idea for privacy and ethics reasons — but you need *some* way to distinguish between samples.

GestureDataRecorder takes a simple but imperfect approach to this problem: it expects users to provide a unique identifier and then saves data for each user in separate files. To support this, the app makes users enter an ID number and then includes that in the names of the files it saves. If any files using that ID already exist on this device, the app requests confirmation and then appends new data to those files. So it trusts users not to append their data to someone else's files on the device, and it's up to you to ensure no two users enter the same ID on *different* devices.

The starter code supports the interface and other business logic for the app — you'll add the motion-related bits now so you get to know how that all works.

Accessing device sensors with Core Motion

You'll use Core Motion to access readings from the phone's motion sensors, so import it by adding the following line along with the other imports in **ViewController.swift**:

```
import CoreMotion
```

This lets you access Core Motion within your *code*, but it's not enough to allow your *app* to do so. Apple rightly wants users to decide which apps can access their data, so it requires developers to include an explanation for *why* they want it. The starter project's **Info.plist** file already includes this explanation as a value for the key **Privacy - Motion Usage Description**. And because motion data is *required* for this app to function, rather than just a nice additional feature, both **accelerometer** and **gyroscope** have been added to **Info.plist**'s **Required device capabilities** list, too. Don't forget to provide the appropriate properties in your own apps.

Next, in order to interact with Core Motion, add the following properties inside `ViewController`. Keep things organized by putting them under the existing comment that reads `// MARK: - Core Motion properties`:

```
let motionManager = CMMotionManager()
let queue = OperationQueue()
```

Here you create a `CMMotionManager` to access the device's motion data. Each app should contain only one such object, regardless of how many sensors it plans to use. You'll use `queue` to keep sensor update callbacks off the main thread, which helps the device remain responsive while processing these high frequency events. Using a separate `OperationQueue` like this also ensures your app doesn't miss updates if it is temporarily too busy to process events.

Before you go any further, find the following two lines inside `startRecordingSession` and delete them:

```
/* TODO: REMOVE THIS LINE
...
TODO: REMOVE THIS LINE */
```

These lines were commenting out a `guard` statement that ensures the app has access to device motion, and alerts the user otherwise. They were commented out because they require `motionManager`, which you just added.

You need to tell `motionManager` how often to produce sensor data. `ViewController` stores its configuration-related constants inside its `Config` struct, so add the following constant there:

```
static let samplesPerSecond = 25.0
```

Here you set `samplesPerSecond` to 25, which you'll use later to specify you want the device to send you 25 sensor updates every second. This number is important because it determines how much data your model looks at based on how often you perform predictions. That is, if you classify the user's activity once per second, this gives you 25 samples per classification; if you do it once every four seconds, this rate gives you 100 samples.

But why 25? The sensors in Apple devices are capable of producing updates many times per second — at least 100, according to the docs — so shouldn't you just use the max? After all, aren't people always saying that when it comes to machine learning, more data is always better?

There are a few reasons why you shouldn't necessarily increase the update frequency too high:

- More updates means more data processing, which means less CPU available for whatever else your app needs to do.
- Faster updates usually means feeding more data into your model per prediction. That requires more complex ML models, which run more slowly — maybe too slowly to keep up with those faster updates.
- Higher frequency updates increase battery usage. You don't want users deleting your app because it sucks the life out of their devices.

It's true that higher frequency updates let you perceive finer details within the data, so there are times when you may need them. But not always — some activities involve slower changes over a longer time, where sensor readings might be necessary only a few

times per second, or even less. The value of 25 used here was chosen arbitrarily — it works fine, but experiments to find the lowest usable update rate were not performed.

> **Note**: There's another option you aren't using here, but you may want to consider for your own projects. Perform data *collection* at a high rate, and then *downsample* it to train multiple models and find the lowest rate that works well. For example, collect data at 80Hz and then train multiple models — 80Hz using all the data, 40Hz using every other sample, 20Hz using every fourth sample, etc. This let's you collect data once and then have different options for how to use it, which is better than having to recollect it multiple times to experiment with update rates. Once you find the lowest rate that still works well, use that in your production app.

In this app, you'll store all the collected sensor data in memory and then write it out to disk at the end of the recording session. Add the following array with the other properties under the comment that reads `// MARK: - Core Motion properties` in `ViewController`:

```
var activityData: [String] = []
```

You'll create a single string containing all the data you want to record for a sample, and append it to `activityData`. The entire recording session will live inside this array as one long sequence, and GestureDataRecorder calls `saveActivityData` at the end of the session to save all these strings to file. However, you need to add the following line to actually save the array. Put it inside `saveActivityData` in `ViewController`, inside the `do` block that currently only contains a `print` statement:

```
try self.activityData.appendLinesToURL(fileURL: dataURL)
```

This writes all the strings in the array out to the appropriate file using a helper function from inside **StringArrayExtensions.swift**. This function creates the file if it doesn't already exist, or appends to the file otherwise.

One important aspect of GestureDataRecorder is that it keeps recording sessions very short. As such, there's no fear of running out of memory while storing data in `activityData`. That also means it's not a big deal if something goes wrong while recording and you need to throw out some data — it's never much more than a minute's worth. Shorter sessions are also easier on your test subjects — it's probably a bit much to ask someone to shake their phone for an hour straight, but doing lots of tiny sessions isn't so bad.

However, when working with longer lasting activities, where data collection takes several minutes or more, you don't want to risk having to throw away too much data. In

that case, you should write your data out to disk periodically rather than at the end of the session. You should also consider making your app more robust, by saving data when the app gets interrupted from things like incoming phone calls, for example.

You haven't enabled motion updates just yet, but eventually the app will receive them in the form of `CMDeviceMotion` objects. Add the following method to `ViewController` to process them:

```
func processMotionData(_ motionData: CMDeviceMotion) {
  // 1
  let activity = isRecording ? currActivity : .none
  // 2
  let sample = """
  \(sessionId!)-\(numActionsRecorded),\
  \(activity.rawValue),\
  \(motionData.attitude.roll),\
  \(motionData.attitude.pitch),\
  \(motionData.attitude.yaw),\
  \(motionData.rotationRate.x),\
  \(motionData.rotationRate.y),\
  \(motionData.rotationRate.z),\
  \(motionData.gravity.x),\
  \(motionData.gravity.y),\
  \(motionData.gravity.z),\
  \(motionData.userAcceleration.x),\
  \(motionData.userAcceleration.y),\
  \(motionData.userAcceleration.z)
  """
  // 3
  activityData.append(sample)
}
```

This method creates samples for your dataset from `CMDeviceMotion` objects. Here's how it works:

1. You label each sample with the activity it represents. This line checks to see if there *is* an activity being recorded or if this data is arriving in-between activities. In the latter case, you label it as `ActivityType.none`. The current activity is set from within the starter code after the app announces the activity to the user.

2. Here you create one big string representing a single data sample. It includes a session ID, the current activity and the sensor readings extracted from `motionData`, all separated by commas.

3. This line appends the string to `activityData`. The entire array gets saved to disk later, when the recording session ends.

Along with the session ID and the activity type, you're saving 12 different values at each moment in time. These were chosen because they seem like they *could* be relevant to the task at hand. However, you might not use all of them when you train your model.

But it's a good idea to record as much data as you can, because it gives you more options later when building your model. You can always remove data you don't need, but there's no way to go back to these moments and record additional data — adding features requires a new data collection effort.

Notice the session ID gets created by combining `sessionId`, which is a timecode created when recording starts, and the number of which recording the user is currently doing. That means that each time a user runs the app, they'll choose between creating one, two or three *sessions*, even though to the user it will seem like just one session.

Why is that important? You'll be using Turi Create's activity classification API, and it *currently* requires a few things when training. (Comments from its developers on GitHub seem to indicate they would like to make it more flexible in the future.) First, it doesn't like super short sessions. Without going into detail here, you'll want your sessions to be about as long as 20 predictions worth of data. So if you plan on predicting once per second, for example, sessions should be at least 20 seconds long. It doesn't need to be exact, but sessions much shorter than that may not work well.

Secondly, Turi Create seems to prefer a lot of sessions. So instead of fewer, longer sessions, this app opts for creating more, shorter ones. Note however that sessions do **not** need to contain just a single activity. In fact, the sessions for this app will each contain *two* activities — the gesture itself, as well as a period of `none` data recorded before the gesture. In your own apps you can record any number of activities within a single session, but labeling them like this was an easy way to get more sessions with fewer actual user recordings.

You've got a method to process `CMDeviceMotion` objects, but you still need Core Motion to send them. Add the following to `ViewController` to enable device motion updates:

```
func enableMotionUpdates() {
  // 1
  motionManager.deviceMotionUpdateInterval =
    1.0 / Config.samplesPerSecond
  // 2
  activityData = [String]()
  // 3
  motionManager.startDeviceMotionUpdates(
    using: .xArbitraryZVertical,
    to: queue, withHandler: { [weak self] motionData, error in
      // 4
      guard let self = self, let motionData = motionData else {
        if let error = error {
          print(
            "Device motion update error: \(error.localizedDescription)")
        }
        return
      }
```

```
        // 5
        self.processMotionData(motionData)
    })
}
```

1. Use `samplesPerSecond` that you defined earlier to set how often `motionManager` sends updates to your app. In this case, you're setting it to update every 0.04 seconds, or 25 times per second.

2. Initialize `activityData` to an empty array. The project starter code calls this function each time the user starts a new recording session — this line ensures each session starts with a fresh array.

3. This line instructs `motionManager` to start sending device motion updates, passing a block to execute on `queue` for each update. The `using` parameter tells Core Motion to use `xArbitraryZVertical` as the device position relative to which the device's attitude values should be reported. Check out `CMAttitudeReferenceFrame`'s documentation for the available options.

4. This `guard` statement ensures the callback received motion data. If not, you log an error message if one is available. If you find yourself getting many errors, then you may need a more robust solution here. For example, receiving too many errors in a row could trigger the session to stop and discard the data.

5. Call `processMotionData`, which you added earlier, to extract features from the sensor data and append them to `activityData`.

In this app you use Core Motion's device motion API. `CMMotionManager` also allows you to access accelerometer, gyroscope and magnetometer data directly, but the device motion API is often a better choice. Data directly from the sensors is often quite noisy and requires some preprocessing to smooth it out. But the good folks at Apple have already worked out some nice preprocessing steps and do them for you if you access the device motion data instead. Another nice touch — it separates acceleration due to the user from acceleration due to gravity, which makes the motion represented by the data easier to decipher.

However, if you ever want raw data from those sensors, `CMMotionManager` provides APIs that match that of device motion. So `deviceMotionUpdateInterval`, `startDeviceMotionUpdates`, etc., become `accelerometerUpdateInterval`, `startAccelerometerUpdates`, and so on. Similar methods exist for each sensor.

Now that you've defined `enableMotionUpdates`, find the comment that reads `// TODO: enable Core Motion` inside the `Utterances.sessionStart` case in `speechSynthesizer`, and add a call to your new method there:

```
...
case Utterances.sessionStart:
  // TODO: enable Core Motion
  enableMotionUpdates()
  queueNextActivity()
...
```

Most of the timing in GestureDataRecorder actually comes from logic in `speechSynthsizer`. The app's `AVSpeechSynthesizer` calls this function whenever it finishes uttering a phrase, and the app uses the finished utterance to determine what to do next. In the case of the `sessionStart` message, it enables motion updates and calls `queueNextActivity` to get the recording started.

You've started motion updates, so you'll need to stop them at some point. Add the following method to `ViewController` to do that:

```
func disableMotionUpdates() {
  motionManager.stopDeviceMotionUpdates()
}
```

This function tells `motionManager` to stop sending motion updates. Add a call to it inside the following `case` statement in `speechSynthesizer`:

```
...
case Utterances.sessionComplete:
  disableMotionUpdates()
...
```

This statement executes after the recording session completes. You disable the motion updates and then the rest of the `case` statement saves the data to a file.

Collecting some data

Now go collect some data, ideally from multiple people. Invite your friends over, serve some nice canapés and make it a phone shaking party. If your friends are anything like my kids, they'll be willing to record data at least once before losing interest. :|

Keep in mind, performing activities incorrectly while recording data will reduce your model's performance. That's because you aren't manually labeling things, so you'll end up with mislabeled sequences in your dataset.

In the next section you'll see how to get rid of mislabeled data, but it's much better to avoid recording it in the first place. That's why GestureDataRecorder presents a confirmation window at the end of each recording session — it gives you the chance to discard data without saving it if you know something went wrong during the session.

Any files GestureDataRecorder saves will be accessible from the **Files** app on your iPhone, and inside the **File Sharing** area in iTunes. This works because the starter project's **Info.plist** includes the keys **Application supports iTunes file sharing** and **Supports opening documents in place**, both with values of **YES**.

Get any data you've collected from the device(s) and onto your computer, and store the files in one of the following three folders, all within the **notebooks** folder of the resources you downloaded: **data/train**, **data/valid** or **data/test**.

These folders hold the files from which you'll create the three datasets you'll use when building your model: train, validation and test. You'll read more about why later, but try not to store data collected from one person in more than one of these folders. You should put data from most people in **data/train**, while putting data from about 10% of your users in each of the other two folders. If you end up recording data from only one person — be honest, it was just you, right? — it's probably best to put it in **data/train**.

> **Note**: The device's orientation affects the data you collect. For example, imagine holding an iPhone out in front of you and then moving it up and down, side to side, and toward and away from you. Sensor data collected while doing so would be different if the phone was held in portrait or landscape (including variations based on home button position), with the screen facing toward or away from you, to the left, right, up, down or some angle in between. The gravity fields you stored are enough to determine orientation — that's actually how iOS knows when to rotate your app's UI — so your model *can* learn to identify activities in any of these situations. However, you'll need to provide plenty of training data to cover all the possibilities well enough for it to recognize them.
>
> For your own projects, you can handle this in one of three ways: Instruct users to position their devices a specific way and accept the model may not work well if they fail to do so, collect a much larger dataset that includes data from devices in all probable orientations, or apply a preprocessing step that transforms values into a known orientation. The projects in this chapter settle for the first option.

Analyzing and preparing your data

So you've got some data. You've collected it yourself or acquired it from elsewhere, but either way, your next step is to *look* at it. Don't try reading every number — that way lies madness — but do some analysis to see exactly what you're working with. You want to ensure there aren't any problems that might ruin the models you try to build.

So what are you looking for? Here are a few things to consider:

- If you didn't create the dataset yourself, it's important to see what's there.
- Mislabeled data. Data is often labeled manually and mistakes are common.
- Poorly collected data. Sometimes mistakes are made while recording, such as misplaced sensors, incorrectly followed instructions, etc.
- Source errors. Sometimes the data source introduces errors, such as a damaged or malfunctioning device reporting bad data. And datasets made by people often contain data entry mistakes.
- Incorrect data types. For example, strings where there should be numbers.
- Missing values. It's common for some rows to have values missing. You'll need to decide how to handle those — remove such rows or insert reasonable values. The choice depends on your project, and there are many options for how to fill the values if you go that route. For example, you might use that feature's mean, median or mode value, or perhaps calculate a new value based on values from nearby rows.
- Outliers. *Some* variation is required to make a good dataset, but there are cases when a few samples may be too rare to be worth including in your dataset. Training with them can confuse the model, reducing its overall performance, and it's sometimes better to accept that there are some things your model just won't handle.

> **Note**: You don't *have* to remove such samples — you may very well want your model to support them. But it's something to consider.

You'll work with Python for this and the next section, so no more Xcode for a while. Launch Jupyter from within your **turienv** Anaconda environment — you'll be using Turi Create so you need an environment with that installed. Create a new notebook in the **notebooks** folder of the chapter resources. Or if you'd prefer to follow along in a completed notebook, you can open **notebooks/Data_Exploration_Complete.ipynb** instead.

Get started by entering the following code in a cell and running it with **Shift+Return**:

```
%matplotlib inline
import turicreate as tc
import activity_detector_utils as utils
```

This gives you access to the `turicreate` package as well as some helper functions provided in **activity_detector_utils.py**, which you can find in the **notebooks** folder if you'd like to look through it. The first line is what's known as a "magic" and it tells Jupyter to display any Matplotlib plots inside the notebook instead of in separate windows.

Now run the following code to load your datasets:

```
train_sf = utils.sframe_from_folder("data/train")
valid_sf = utils.sframe_from_folder("data/valid")
test_sf = utils.sframe_from_folder("data/test")
```

Here you use the `sframe_from_folder` function from **activity_detector_utils.py** to load your datasets. It takes the path to a folder — given here relative to the **notebooks** folder in which your notebook resides — and attempts to parse all the CSV files it finds there. We've provided enough data to make the project work, but hopefully you've used GestureDataRecorder to collect some more. If so, whatever files you've added to these folders get loaded here as well.

> **Note**: If you reuse `utils.sframe_from_folder` in your own projects, you'll need to modify it slightly — it currently contains some details specific to this project.

After running that cell, the variables `train_sf`, `valid_sf` and `test_sf` will be Turi Create `SFrame` objects, which are data structures designed to work efficiently with structured data — for example, huge tables of numbers collected from an iPhone's motion sensors. These three `SFrames` contain the data you'll use for your training, validation and test sets, respectively.

Take a peek at some samples by running the following code:

```
train_sf.head()
```

This displays the first 10 rows of the dataset, along with their column names. These names were assigned in `sframe_from_folder` but could also have come from the CSV files directly. The following image shows an example of some output from `head`, edited slightly to fit here:

sessionId	activity	roll	pitch	yaw
2018-08-27T19:53:54Z-1	0	-0.46865287783368514	0.8395582736867373	0.36055793373729217
2018-08-27T19:53:54Z-1	0	-0.47302926367030734	0.8445595740425931	0.3664918810895234
2018-08-27T19:53:54Z-1	0	-0.4737607872960278	0.8483060548254803	0.3678965046324581

rotX	rotY	rotZ	gravX	gravY
0.13357801735401154	-0.0008428385481238365	0.05470273271203041	-0.3016313910484314	-0.7443482279777527
0.14320960640907288	0.00386190228164196	-0.0029299380257725716	-0.30253610014915466	-0.7476786971092224
0.0321430042386055	0.009147211909294128	-0.01968974433839321	-0.3016883134841919	-0.750161349773407

gravZ	accelX	accelY	accelZ	userId
-0.595788836479187	-0.010288774967193604	-0.0012877583503723145	0.008157610893249512	u_02
-0.5911417007446289	-0.020110994577407837	-0.012453138828277588	0.02307224273681641	u_02
-0.5884233713150024	0.008887410163879395	-0.015051662921905518	0.03977835178375244	u_02

First three samples of training set

> **Note**: If you've included your own data in any of these datasets, your results may vary from those shown here. This is true for all the images in this section.

Notice how there is a column named **userId**. This was added inside `sframe_from_folder` — the values are derived from the names of your data files. This only works if your files each contain data from just one user, and their names are prefixed with the user's ID followed by a hyphen (-). For example, all data read from a file named "bob-data.csv" would be assigned a **userId** value of "bob." You could have stored the user ID in each row when you were collecting the data, but saving each user's data into separate files keeps them smaller and makes them easier to organize. Either way, it's important to know the source of your data — you'll see why later.

Here's another thing about `head`'s output — the values in the **activity** column are all `0`. That's nothing to worry about — it's just because you're only looking at the first few rows, which represent less than one second of activity. But what does `0` even mean? Inside GestureDataRecorder, you stored activity types as numeric values. Turi Create can deal with that just fine, but we humans sometimes interpret words more easily than numbers. To convert those integers into something more readable, enter the following code in a cell and run it:

```
# 1
activity_values_to_names = {
```

```
    0 : 'rest_it',
    1 : 'drive_it',
    2 : 'shake_it',
    3 : 'chop_it'
}
# 2
def replace_activity_names(sframe):
  sframe['activity'] = sframe['activity'].apply(
    lambda val: activity_values_to_names[val])
# 3
replace_activity_names(train_sf)
replace_activity_names(valid_sf)
replace_activity_names(test_sf)
```

This replaces the numeric activity values in your datasets with the names of the gestures they represent. Here's how it works:

1. You create a dictionary that maps numeric activity values to strings. These strings were chosen arbitrarily, but they should describe the values clearly — that's the whole point of replacing them, right? Also note, the app you write later uses these values, too, so you'll need to modify code there if you change these strings.

2. You use the activity column's `apply` function to run a lambda function on the value in each row. Lamdba functions are similar to closures in Swift. This one replaces the column's integers with their corresponding strings from the dictionary. `SFrame` columns are represented by `SArray` objects, so check out that class in the Turi Create class if you'd like to see what's available. You define this line as a function just to make the next lines cleaner.

3. You call `replace_activity_names` for each of your dataset `SFrames`.

After running this cell, you've modified your datasets to make them easier to interpret, which you can see by calling `train_sf.head()` again:

sessionId	activity	roll	pitch	yaw
2018-08-27T19:53:54Z-1	rest_it	-0.46865287783368514	0.8395582736867373	0.36055793373729217
2018-08-27T19:53:54Z-1	rest_it	-0.47302926367030734	0.8445595740425931	0.3664918810895234
2018-08-27T19:53:54Z-1	rest_it	-0.4737607872960278	0.8483060548254803	0.3678965046324581

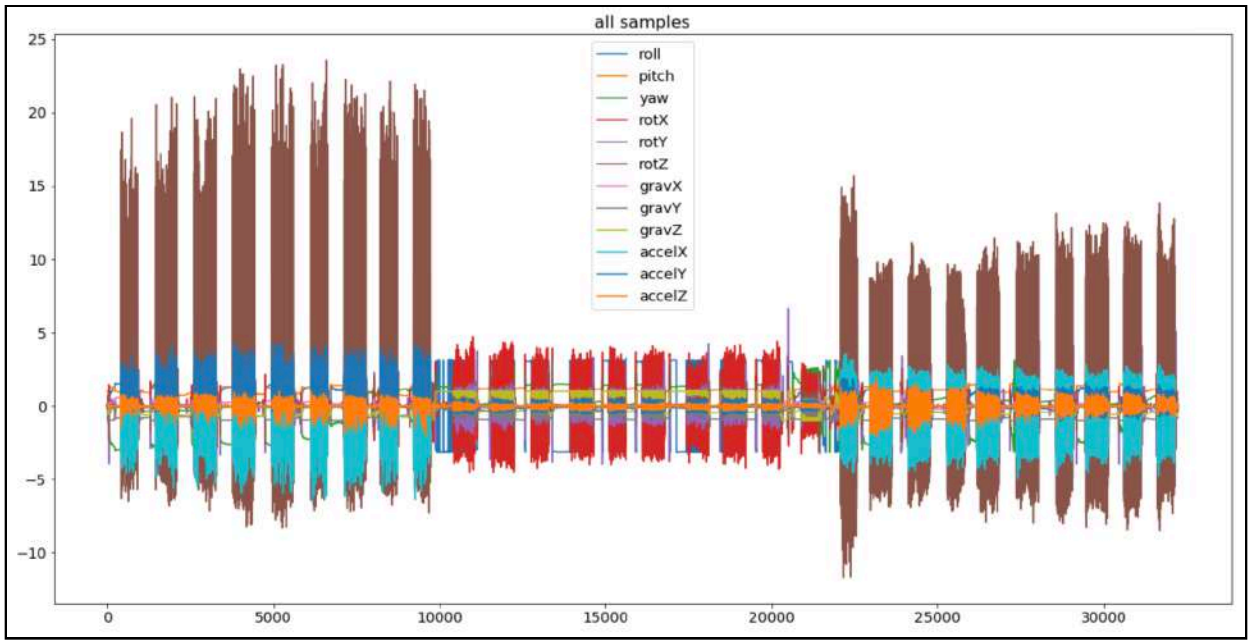
Partial list of features from first three samples of training set, with activity as strings

Note: You certainly could have stored these strings directly when you created the files in GestureDataRecorder, saving yourself the trouble of changing them now. However, using integers conserves a bit of disk space. And more importantly, it gave you the chance to see an example of modifying some data in an `SFrame`, which you might want to do while preparing future datasets.

It's helpful to plot your data to examine it, so run the following code to look at your test set:

```
utils.plot_gesture_activity(test_sf)
```

Here you call `plot_gesture_activity` from inside **activity_detector_utils.py**. It uses Matplotlib to display an `SFrame`'s contents as a line chart. The following image shows the plot generated when you run that code:



Plot of testing dataset

Note: The plots shown in this chapter may be difficult to read, especially in the black-and-white printed version. They are all from **notebooks/ Data_Exploration_Complete.ipynb** — you are encouraged to open it in Jupyter to get a better look at these plots as well as several others not included here.

There's too much data in this plot to see much detail. But even at this zoomed-out scale, it's already clear there are distinct patterns present here.

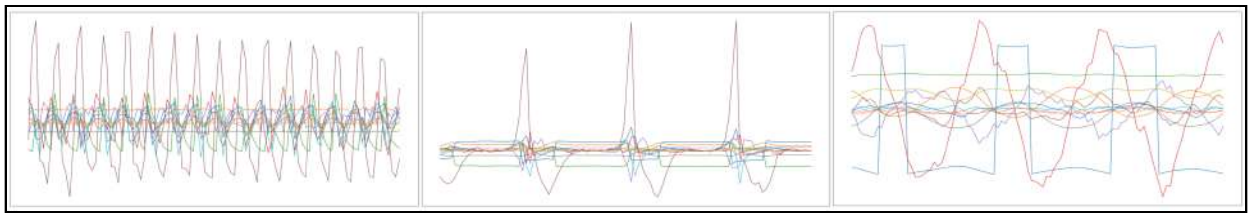
With the `plot_gesture_activity` helper function, you can plot data for a specific activity by specifying its name. The following example would show data just for the `drive_it` gesture:

```
utils.plot_gesture_activity(test_sf, activity="drive_it")
```

And you can zoom in to show specific chunks of data by specifying a slice of the dataset, like so:

```
utils.plot_gesture_activity(test_sf[11950:12050], activity="drive_it")
```

The following three plots were created using code similar to the line above, showing slices of 100 samples for each of the three gestures in the test set:

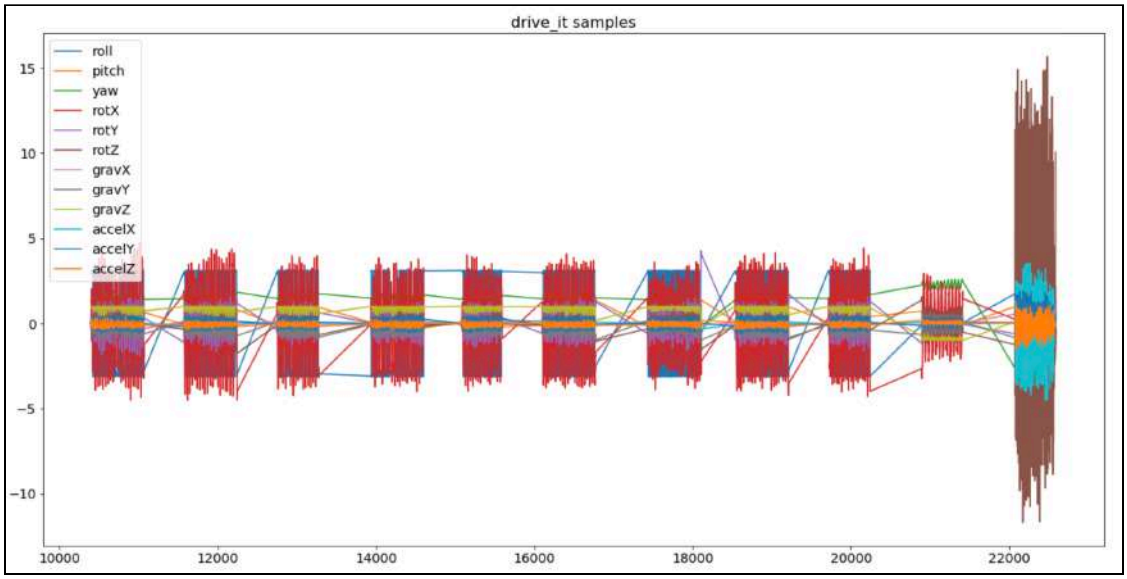


100 samples of 'shake_it', 'chop_it', and 'drive_it' activities from test dataset

The actual values aren't important in these plots. The important thing to notice is how each gesture appears as a clearly discernable pattern. It's clear that we *should* be able to recognize when a user performs these gestures, but imagine trying to write your own algorithm to do it — it might be pretty difficult! But don't worry — machine learning makes it *much* easier.

Removing bad data

Now you'll see one way to find and remove errors from your dataset. If you run the code suggested earlier to plot all the `drive_it` activity data in the test set, you'll see a plot something like the following:



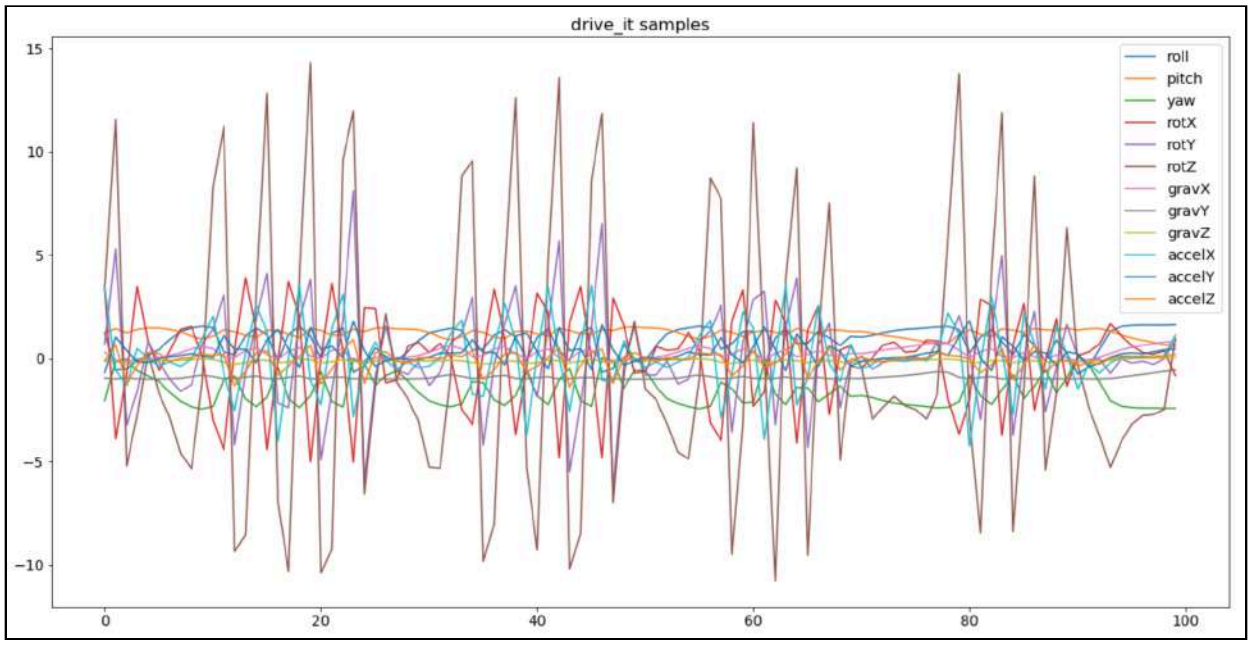
'drive_it' samples in test dataset

While much of this data looks similar, some of it stands out as different. Particularly, the last two blocks of activity seem odd. The following code looks at a small section in the second one of those areas:

```
utils.plot_gesture_activity(test_sf[22200:22300], activity="drive_it")
```

Remember, these specific slice numbers might not be the same in your dataset, but hopefully you can come up with values to find a slice within this section of the data.

This produces the following output:



Mislabeled data in test dataset

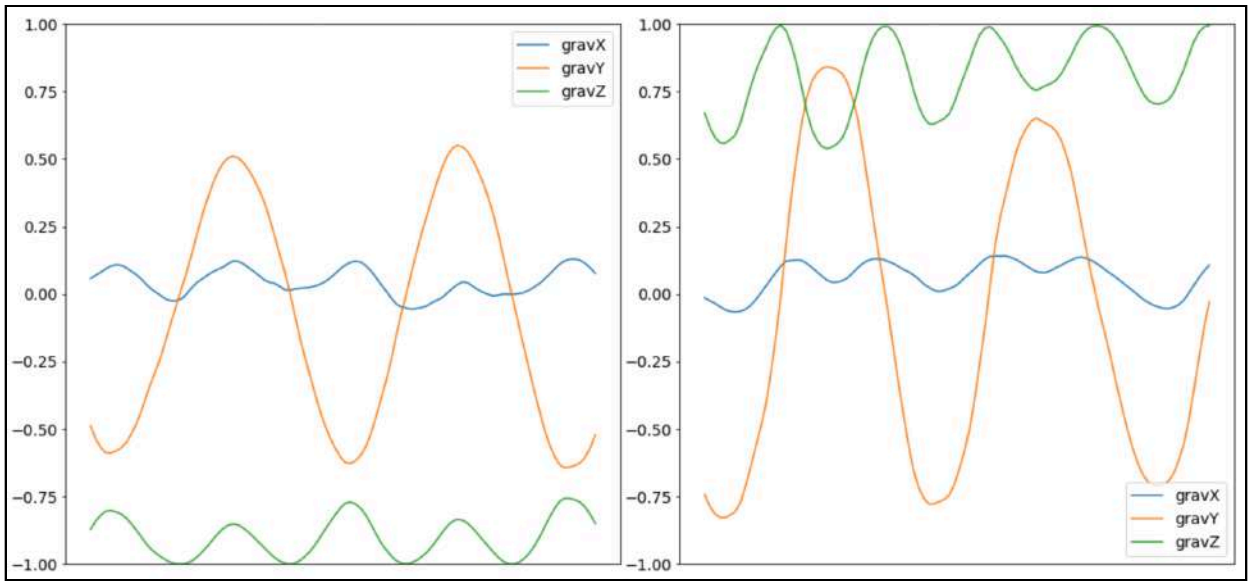
If you compare this to the examples you plotted earlier, you'll see it looks more like a `shake_it` than a `drive_it` action. It seems someone performed the wrong gesture while recording, essentially mislabeling your data.

The second area of concern is a bit more difficult to see because it *mostly* looks the same as the good data. But if you look closely you may notice an area of green at the top of the data — green that you don't see in any of the other `drive_it` data. The following code zooms in on this area and plots only a few features:

```
utils.plot_gesture_activity(test_sf[21200:21500], activity="drive_it",
                            features=["gravX", "gravY", "gravZ"])
```

This call uses another one of `plot_gesture_activity`'s optional parameters to specify a list of features to plot. So rather than showing all the data in this slice, it shows just the data for the device's gravity readings. The following image was made using code similar to the line above (with some slight adjustments to help with formatting). The plot on

the left shows a 100 sample sequence from the suspicious looking area, and the plot on the right shows a 100 sample sequence similar to the majority of the `drive_it` data:



Gravity values for 'drive_it' gesture. Left: Incorrectly oriented. Right: Correctly oriented.

These plots show similar readings for gravity along the X and Y axes. The scale is slightly different for gravity along the Y axis, but the two plots are still basically the same. However, the gravity readings along the Z axis seem to be quite different. The *patterns* are the same, but the values are negative in the left example and positive in the right one. This indicates the user was not holding the phone in the correct orientation while performing the motion — the screen was facing up instead of down.

Both of these sessions contain data that will only serve to confuse your model, reducing its performance, so it's best to remove them from your dataset before continuing. To do so, run the following code, replacing the index values with ones that work for your dataset:

```
// 1
bad_session_1 = test_sf[21350]["sessionId"]
bad_session_2 = test_sf[22250]["sessionId"]
// 2
test_sf = test_sf.filter_by([bad_session_1, bad_session_2],
                            column_name='sessionId', exclude=True)
```

Here's what that does:

1. Grabs the session ID from a row in the middle of each area of bad data. Each session contains data for only one activity, so once you know the session ID for one, you know it for all the rows you want to delete.

2. Calls `SFrame`'s `filter_by` method to return a new `SFrame` that excludes any rows where the **sessionId** column contains the value of either of the bad sessions.

Plotting the test set's `drive_it` data again shows the suspect sessions are now gone. The plot isn't included here to save space, but the **Data_Exploration_Complete.ipynb** notebook includes this plot if you'd like to compare it to your results.

This section included a few examples demonstrating some things to look for, but you should spend time thoroughly exploring all three of your datasets, both to clean up problems and to better understand your data. And don't neglect any particular dataset — testing with bad data can be just as problematic as training with it.

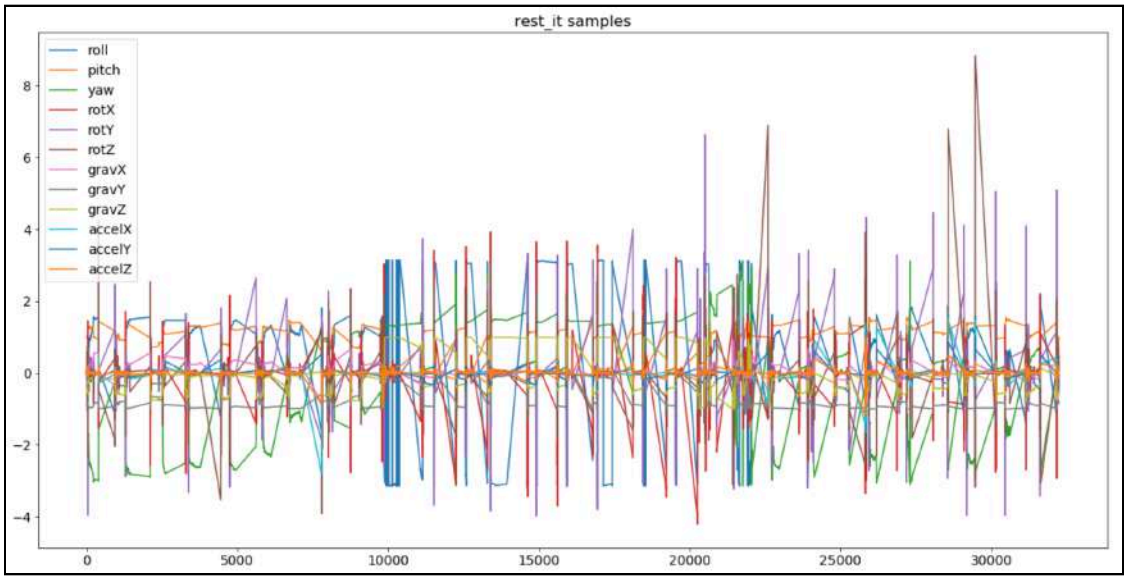
> **Note**: The erroneous data you removed from the test set all comes from one file: **notebooks/data/test/bad-drive-it-data.csv**. You can safely remove that file if you don't want to go through this exercise again.

Optional: Removing non-activity data

What about motions that have nothing to do with gestures? You know, all those sensor readings that arrive *between* the gestures? Take a look at that data by plotting the `rest_it` activity. Here's how you do so for the test set:

```
utils.plot_gesture_activity(test_sf, activity="rest_it")
```

This plots all samples in the test set labeled as `rest_it`, which means data that is *not* a gesture. Here are the results:



'rest_it' samples in test dataset

Unlike with the gestures you plotted earlier, the resting data shows no clear pattern. That makes sense — users can do whatever they want between gestures, so there are basically an infinite number of possible sequences that could appear with this label.

Depending on how similar the resting and activity data are, a model might have trouble learning to classify them both well. In those cases, it often helps to increase the size of your dataset set. However, in many cases — such as this one — the model will learn to recognize both resting and activities. This is probably because the sequences related to the other gestures are so much more distinct. That is, it will likely learn to classify the other gestures well, and then learn that anything else is resting. It will get some samples wrong — users sometimes perform the gestures while GestureDataRecorder is recording rest data, essentially adding mislabeled data to your dataset — but the juxtaposition of the messy resting data and the patterned gestures should make the model even more confident about its gesture predictions.

For this app, train with all your data, including the resting samples. However, you're encouraged to try making another model that excludes the resting data to see which you prefer. The results might vary depending on exactly what your datasets look like.

If you ever want to try removing that data, you can do so with the following code:

```
train_sf = train_sf.filter_by(["rest_it"], 'activity', exclude=True)
test_sf = test_sf.filter_by(["rest_it"], 'activity', exclude=True)
valid_sf = valid_sf.filter_by(["rest_it"], 'activity', exclude=True)
```

Much like how you removed the bad sessions, this would create new `SFrames` that do not contain any samples whose **activity** value was `rest_it`.

Balancing your classes

After you are satisfied you've cleaned your data, there's one final thing you should check: How many examples of each class do you have? Run the following code to count the examples in each dataset:

```
utils.count_activities(train_sf)
utils.count_activities(valid_sf)
utils.count_activities(test_sf)
```

Here you call `count_activities`, another helper function defined in **activity_detector_utils.py**. It dislays a table showing how many sessions are present for each activity, both per user and total.

The following shows the counts for the datasets we provided:

```
+----------+--------+-------+ +----------+--------+-------+ +----------+--------+-------+
| activity | userId | Count | | activity | userId | Count | | activity | userId | Count |
+----------+--------+-------+ +----------+--------+-------+ +----------+--------+-------+
| chop_it  |  u_01  |   36  | | chop_it  |  u_03  |   4   | | chop_it  |  u_05  |   9   |
| chop_it  |  u_02  |   36  | | chop_it  |  u_04  |   4   | | drive_it |  u_05  |   9   |
| drive_it |  u_01  |   36  | | drive_it |  u_03  |   4   | | rest_it  |  u_05  |   27  |
| drive_it |  u_02  |   36  | | drive_it |  u_04  |   4   | | shake_it |  u_05  |   9   |
| rest_it  |  u_01  |  108  | | rest_it  |  u_03  |   12  | +----------+--------+-------+
| rest_it  |  u_02  |  108  | | rest_it  |  u_04  |   12  | [4 rows x 3 columns]
| shake_it |  u_01  |   36  | | shake_it |  u_03  |   4   |
| shake_it |  u_02  |   36  | | shake_it |  u_04  |   4   |
+----------+--------+-------+ +----------+--------+-------+
[8 rows x 3 columns]          [8 rows x 3 columns]
```

Activity counts for train, validation and test sets

Here you can see that each dataset contains the same three gestures, and no gesture is represented more than any other within a specific dataset. Users within a dataset are represented equally as well. For example, each of the training set's two users supplied 50% of the training data. Things are looking great! You won't always have such perfectly balanced datasets, but you want them to be as well balanced as possible. If any gesture or user is overrepresented in the training set, your model may bias itself toward those samples. But unbalanced validation or test sets can be a problem, too, because they'll skew your evaluation results, making it more difficult to judge your model.

The dataset included in the resources contains 216 actions for training, 24 for validation and 27 for testing. It's not a lot of data, but it's as much as the author's family was willing to put up with collecting. :[Still, it's a reasonable balance, with about 80% of your data for training, and around 10% each for validation and testing.

Once you're convinced your datasets are good to go, run the following code to save the cleaned up `SFrames` for later use:

```
train_sf.save('data/cleaned_train_sframe')
test_sf.save('data/cleaned_test_sframe')
valid_sf.save('data/cleaned_valid_sframe')
```

The `save` method lets you save `SFrames` in several different formats, such as CSV and JSON. Here you're using a format that creates the given folder and stores various binary files in it. It's convenient because it's smaller and loads faster than the others, but feel free to use any format you like. And remember, you still have your original files, so you can always start over if you decide you don't like something about your cleaned data.

> **Note**: Turi Create has many options for data exploration and manipulation, as do Pandas and NumPy. And it provides methods to convert to and from the data structures used by these other libraries, so if there's something you prefer to do in one package over another, you can freely move back and forth. It's a good idea to spend some time looking through the documentation for these various frameworks to see what's available, but don't try to learn everything all at once — as you do more with machine learning, you'll continue to discover new things.

With your data ready, it's time to make your model. The next section shows you how.

Creating a model

You've got a clean dataset and now you're ready to train a model. Or maybe several models until you find one that works well. This section shows how to use Turi Create's task-focused API to train a model for activity detection.

> **Note**: Training your own model here is *highly* recommended, especially if you collected data to add to the provided dataset. But if for whatever reason you want to skip this step for now, you can find a trained model named **GestureClassifier.mlmodel** inside the **notebooks/pre-trained** subfolder of the chapter resources.

In this section you'll continue working with Jupyter in your **turienv** Anaconda environment. Create a new notebook in the **notebooks** folder of the chapter resources. If you'd like to see how we trained our provided model, you can check out the completed notebook **notebooks/Model_Training_Complete.ipynb**.

Import the same packages as you used in the previous section:

```
import turicreate as tc
import activity_detector_utils as utils
```

Then run the following code to load your training, validation and testing datasets:

```
train_sf = tc.SFrame("data/cleaned_train_sframe")
valid_sf = tc.SFrame("data/cleaned_valid_sframe")
test_sf = tc.SFrame("data/cleaned_test_sframe")
```

As mentioned earlier, Turi Create stores structured data in `SFrame` objects. There are various ways to create such objects — here you load them directly from the binary files you saved in the previous section. If you'd prefer to use the files supplied with the

resources, change the paths to `pre-trained/data/cleaned_train_sframe`, `pre-trained/data/cleaned_valid_sframe` and `pre-trained/data/cleaned_test_sframe`.

Training any classifier involves using multiple datasets for training, validation and testing. But dealing with sequences includes a few wrinkles that require some explanation.

Splitting sequential data

If you've ever trained an image classifier, you may have divided the images into training, validation and test sets *randomly*. Or maybe those sets were provided for you, in which case someone *else* divided them randomly.

This works because each image is its own sample — no one image relates any more or less to any other image. (See the upcoming Note for an important caveat to this statement.) But the very nature of sequences is that samples *do* relate to each other. Order and grouping both matter — that's what *makes* them sequences! For example, if you're counting by twos — two, four, six, eight — and then randomly shuffle that data — eight, two, six, four — you've lost the sequence and now the data is meaningless. Or worse, you may have accidentally reordered them into a sequence with a different meaning — eight, six, four, two — now the sequence counts *down* by twos!

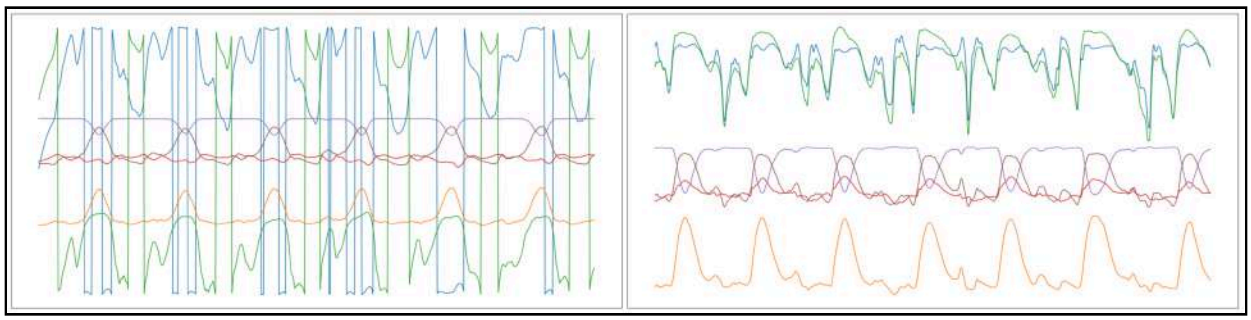
So the first rule for training with sequences: keep samples related to individual sequences grouped together and in order. Any shuffling or sampling you do should take into account these groupings.

> **Note**: There *can* be situations where relationships exist between images in datasets meant to train classifiers, but those usually indicate mistakes that you should try to avoid. For example, if some images are identical or nearly so, as is common when dealing with large numbers of images, then having some in the training set and some in the validation or test sets may mislead you into thinking your model generalizes better than it does. It's a tricky situation, because sometimes your in-production model *will* encounter examples that are nearly identical to those it saw while training. For example, consider a model meant to identify product images from the internet — it's unlikely that you'll manage to create a good training set without also including some of the very images its meant to recognize. But in general, do your best to keep training and test sets as separate as possible, while realizing there are going to be times when some similarity sneaks in.

There's a second potential concern, relating to the *sources* of the sequences. Consider the case you've been working on throughout this chapter — gesture recognition. There's certainly *some* variation each time you perform a gesture — after all, the app collects several floating point values from multiple sensors, many times per second, so its basically impossible to get two *identical* recordings. However, identical isn't the same as *really similar*.

Different recordings of one person making a gesture are going to be similar to each other. That's not entirely bad — it's that similarity you want the model to recognize. You may even find it's fairly easy to train a model that recognizes gestures from a *specific* person — it may not even require many training examples. But it might not work as well when you use the model with someone else.

That's because recordings from one person are more similar to *each other* than they are to recordings from *someone else*. For example, the following plots show some data from two people performing the same actions — step up exercises:



Data collected from two users both performing the same activity — step up exercises

These two plots show similar values for several features, but some features are quite different between users. A model trained on data from one of these users might have trouble recognizing the activity when presented with data from the other — and the more data you show your model from one user, the more different the other user's data will seem. It's certainly good to collect lots of data from each source, but it's more important to collect data from lots of sources. So if you have the choice of getting 1000 recordings from one person, versus 100 recordings from 10, the second will probably produce a better model. And 10 recordings from 100 people would probably be even better. By all means, get more data from each person if you can, but definitely try to collect data from as many people as possible.

And that's the second rule for training with sequences, or really any data where the data's source affects its features: use data from as many sources as possible. The more sources you have in your training set, the better your model should generalize to unseen examples.

But even if you have a great dataset chock full of examples from many different people, there's another issue — how best to split it up into train, validation and test sets?

You might be tempted to split the data randomly (keeping in mind the earlier rule about maintaining samples as sequences, of course). However, you should avoid this. Remember how different recordings from the same person are similar to each other? Well, if you train with data from one person, then test with different data from the *same* person, your model may appear to perform better than it really does. That's because it essentially trained on some of the test data.

So the third rule for training with sequences: don't split your datasets by sequence, split them by *source*. Make sure you know the source of each of your data samples, and try to put all the data from any particular source into the same set: train, validation or test.

> **Note**: Those last two problems occur with more than just sequences. Many data types are affected by their sources. For example, sensors from different phones will report slightly different values in the same situations, camera lenses have slightly different distortions, and so on. All physical devices are produced with some variance, so data collected from different devices can be slightly different even when measuring the same thing. In these cases, the same rules apply: try to train with as many different sources as possible, and try to test on data collected from multiple sources. Unless of course the model is *meant* to work with a specific source — such as correcting lense distortion for images from a *specific* camera. Then by all means test on data collected from the same source to ensure your model works correctly in its intended production environment.

But sometimes...

And now, in a shocking plot twist, you're about to be told to sometimes do what you were just told not to do — train and validate on data from the same people! What?!

Real talk: There are going to be times — maybe *most* of the time — when you won't have as much data as you want. In those cases, you can stretch your dataset out a bit by starting with just two datasets — training and test — and then grabbing a chunk of your training set to use for validation.

Depending on how many different sources are present in your training set, you might not be able to follow the recommended procedure of separating based on source. For example, the one provided with the chapter contains data from just two people. You'd lose too much training data if you separated these users, so you'd need to accept training and validating on data from both of them.

It's not ideal — your validation accuracy will be artificially closer to your training accuracy because the two datasets are more similar, making it harder to tell if your model overfits. But if there's enough variety in your training set to start, then this still works fairly well.

To help split your training data, Turi Create provides a nice utility function that divides an `SFrame` randomly into two smaller `SFrames`, while still maintaining proper sequence groupings. The following code demonstrates how to use it:

```
train, valid =
  tc.activity_classifier.util.random_split_by_session(
    train_sf, session_id='sessionId', fraction=0.9)
```

This uses Turi Create's `activity_classifier.util.random_split_by_session` function with a training set, telling it which column name identifies the sessions, and what percentage of the data should be used in the first split. It returns two `SFrames`, the first will contain the given percentage of the original `SFrame`'s sessions, and the second will contain the remaining sessions.

After running this code, `train` would contain about 90% of the sessions and `valid` would contain the other 10%. You would then use these two `SFrames` for your training and validation sets.

The most important thing about this function is that it splits data based on session IDs, which means it keeps sequences organized together. Any samples with the same session ID are kept together and in order, but any particular session ID could end up in the training or the validation set.

The results of this call are not necessarily going to give you a perfectly balanced split. For example, here are the results of calling `utils.count_actvities` on `train` and `valid` from one sample run:

```
+----------+--------+-------+ +----------+--------+-------+
| activity | userId | Count | | activity | userId | Count |
+----------+--------+-------+ +----------+--------+-------+
| chop_it  |  u_01  |   33  | | chop_it  |  u_01  |   3   |
| chop_it  |  u_02  |   34  | | chop_it  |  u_02  |   2   |
| drive_it |  u_01  |   31  | | drive_it |  u_01  |   5   |
| drive_it |  u_02  |   30  | | drive_it |  u_02  |   6   |
| rest_it  |  u_01  |   95  | | rest_it  |  u_01  |   13  |
| rest_it  |  u_02  |   97  | | rest_it  |  u_02  |   11  |
| shake_it |  u_01  |   31  | | shake_it |  u_01  |   5   |
| shake_it |  u_02  |   33  | | shake_it |  u_02  |   3   |
+----------+--------+-------+ +----------+--------+-------+
[8 rows x 3 columns]         [8 rows x 3 columns]
```

Random train/validation split counts

That's probably fine, but if you see a particularly bad split — especially when you know the original data was well balanced — then you should try splitting it again.

If you want to experiment later, try combining the training and validation data and then use this function to randomly split it. You'll end up with more variety in your training data in exchange for a less trustworthy validation set. For now, you'll just use the separate datasets you've already built.

Training the model

Now it's time to build and train your model. Almost.

Whenever you train with a new model or dataset, you should first take a small portion of your training data and see if you can get the model to overfit it. Overfitting is usually a bad thing — it means your model is memorizing the training data instead of learning a more general solution — but it also shows that the model is actually capable of learning *something* from your data. If your model is going to work on a real dataset, then it should definitely be able to overfit on a tiny version of it. And if it can't, then you've got one of several problems you'll need to address:

- A bug in the model. This is especially common when implementing neural nets from scratch using frameworks such as Keras.
- A model too simple to solve the problem. You might need more layers, or more nodes per layer.
- A model architecture incapable of solving the problem. Different architectures work better for different problems, so pick something appropriate.
- Poorly tuned hyperparameters. Sometimes all it takes is a change to the learning rate, other times you might need different activation functions, optimization algorithms or loss functions.
- Maybe the problem is the problem itself. Machine learning isn't the right solution to every problem, so don't try to force it.

The point of this exercise is to prove to yourself that your dataset is applicable to the problem, your model is built correctly and it's tuned well enough to learn. You'll still usually have to do more tuning later with your full dataset, but those training sessions take longer. This step is critical to keep yourself from wasting time trying to tune a model that isn't ever going to work.

To save space we don't show the results of the overfitting step here, but you can find them in the notebook **Model_Training_Complete.ipynb** in the **notebooks** folder.

Ok, *now* it's time to build and train your model. Turi Create's activity classification API makes this process easy — it just takes one function call! Add the following code to a notebook cell, but **don't** run it yet:

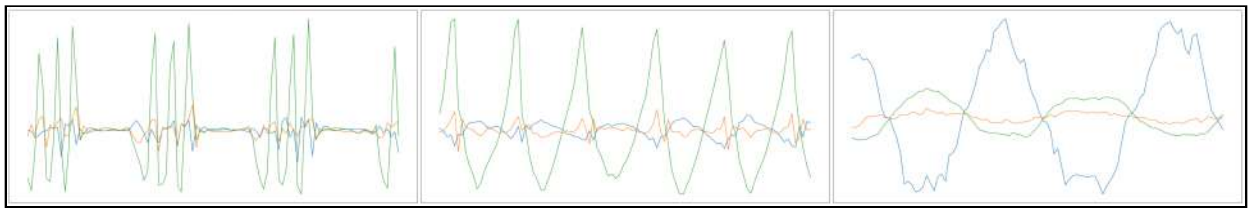
```
model = tc.activity_classifier.create(
  dataset=train_sf, session_id='sessionId', target='activity',
  features=["rotX", "rotY", "rotZ", "accelX", "accelY", "accelZ"],
  prediction_window=20, validation_set=valid_sf, max_iterations=20)
```

This one line of code is doing a lot, so it warrants quite a bit of explanation. Here goes:

- `dataset`: Your training dataset, stored as an `SFrame`.
- `session_id`: The name of the column in `dataset` that stores the session ID associated with each row. `create` keeps data with the same session ID grouped together and in order, and then trains over it in chunks the size of `prediction_window` rows.
- `target`: The name of the column that contains the labels you want the model to predict. In this case, it's `activity`.
- `features`: This is an optional list of columns to use for training. If you don't supply it, then `create` uses all the columns as features except for the ones you specified for `session_id` and `target`. More on this in a bit.
- `prediction_window`: How many samples (i.e. rows of data) the model looks at to make a prediction. More on this later.
- `validation_set`: Your validation dataset, stored as an `SFrame`. This is optional — if you don't supply it, and `dataset` contains more than 100 sessions, then `create` will automatically make a validation set by randomly selecting sessions from `dataset`. But if it contains fewer sessions than that, `create` trains the model without a validation set. It's best not to rely on this logic, and supply your own data instead.
- `max_iterations`: The maximum number of epochs `create` will train over. That is, the number of times it will go through the training set. Note: the parameter name and documentation claim this is a "maximum," as if `create` *could* stop training sooner. However, there appears to be no evidence that training ever stops before this value is reached, so think of it as the actual number instead of a maximum.

Notice how the `features` parameter is a list including just six of the 12 motion features available in your dataset — the rotation and acceleration due to the user.

These were chosen a bit arbitrarily, mostly to show that you don't *need* to use all columns in your dataset. In the previous section you saw how each activity appeared with a distinct pattern. But take a look at the following plots, which show just user rotation values for samples of each activity:



Rotations for 100 samples of 'shake_it', 'chop_it', and 'drive_it' activities from training dataset

As you can see, there are still clearly visible patterns, even when using just these three features. You are encouraged to train models with different feature combinations to see if/how it effects the results. There is no one correct answer here — many combinations will produce usable models for this project.

> **Note**: For any specific problem, there is likely some minimum set of features necessary to train a good model. It just needs *enough* information to perceive differences between the classes, and different features may be more or less useful for each class. The final set of features you settle on will always be project dependant, but when in doubt — use more. That gives your model the most leeway to decide for itself.

The prediction window is an important aspect of Turi Create's activity classification model. It specifies how many samples the model needs to look at each time it makes a prediction. That means this value — combined with Core Motion's update interval — determines the amount of *time* each prediction represents.

For example, if the prediction window is 50 and Core Motion sends the app 10 updates per second, it will take five seconds to collect enough data to make one prediction. But if you're getting updates 100 times per second, it would take 0.5 seconds. As was mentioned earlier, be sure you train your model with a prediction window that makes sense for the update rate you are using. You collected data at 25 samples per second, so this window size of 20 means the model needs 0.8 seconds worth of data per prediction.

The prediction window suggested here works well with the provided dataset and satisfies our goals for the book. However, you should train multiple models using different window sizes to see what you think works best. You won't *really* know if you're satisfied until you use the model in its target environment — in this case, the game you'll make later in this chapter. There's no one "correct" size — it's based on the

specific use case, the dataset, and a bit of personal preference. Traditional software developers often struggle with this aspect of machine learning more than any other — you can't usually sit down and just write the "solution" to a problem; it's more about running lots of experiments until you discover what works best for your specific use case.

Ok, now create your model by running the cell with your call to `create`. The first output you'll see will be something like this:

```
Pre-processing 235057 samples...

Using sequences of size 400 for model creation.

Processed a total of 216 sessions.
```

Initial training output

Here's a rundown of what this tells you:

- The first line reports how many samples — individual rows — are in the training dataset. The function performs some pre-processing on the data, including chunking it into fixed-length sequences.
- The second line let's you know Turi Create will be training on sequences of 400 samples. That's because you're training with a prediction window of 20 samples, and Turi Create's underlying implementation always trains in chunks of 20 consecutive windows. If a session doesn't have enough samples available, the end of the sequence gets padded with zeros. This is why you shouldn't have very short recording sessions — tiny sessions result in too much padding and the model will have trouble learning.
- Finally, it reports the number of sessions in your training dataset. This matters most when you don't supply a validation set, because `create` will use some of these sessions for validation *if* there are more than 100 sessions available.

Ater that, you'll see updates appear for each training iteration — or epoch — as your model continues to train. You need to check these updates for signs of overfitting. If the training accuracy continues to improve but the validation accuracy stalls or begins to decline, then the model has begun to overfit.

The output for the provided model isn't included here, but you can see it in **notebooks/Model_Training_Complete.ipynb**. It actually overfits slightly, but we decided to stick with that model anyway for a couple reasons.

First, it's good to show to readers as an example of overfitting. And secondly, the difference between the final model's validation accuracy and the epoch with the best

value was only about 1%. The validation set is very small, with only four recordings of each gesture from each of two users. A 1% difference in accuracy in such a small dataset really isn't significant enough to prove anything about the model's expected real world performance — it might just be certain epochs arrived at weights that happened to work well with that particular validation set. This is why you should strive to get a lot of variety in your datasets by collecting data from many different sources.

The final epoch for the model that ships with the book as **notebooks/pre-trained/GestureClassifier.mlmodel** had a training accuracy of 98.5% and a validation accuracy of 95.2%. If you include training data collected from other sources, you're likely to get *lower* training accuracy while getting *higher* validation accuracy. Don't get hung up on the specific numbers, though — the idea is just to get something that looks like it trained well before moving on to testing with your test set.

> **Note**: Turi Create is great because it builds and trains sophisticated models without you needing to do much more than provide the data. However, that comes at the cost of flexibility. There isn't much you can do here to tweak your model's performance. Besides changing your dataset, you can also try different prediction windows, feature combinations, batch sizes (not discussed here — you just used the default), and number of training epochs. If none of that leads to a model suited to your app, then you'll need to build something customized in a more flexible framework like Keras. You'll work with sequences in Keras in later chapters.

When you think the model's ready for testing, go ahead and run code like the following:

```
metrics = model.evaluate(test_sf)
print(metrics['accuracy'])
```

You use the model's `evaluate` method to classify everything in your test set and gather the results inside a dictionary named `metrics`. You've also displayed the accuracy the model achieved with those classifications, which for the provided dataset should be in the very high 90s — the model included with the chapter resources scores over 97%.

Accuracy isn't everything, though. You have access to various other results, including precision, recall, a confusion matrix, and more. You can access each of these by name, like you did with accuracy. To see a quick rundown, just print the entire `metrics` object:

```
print(metrics)
```

The confusion matrix is particularly useful here. It lets you know not just whether or not your model was correct, but *where* it made mistakes. This let's you see if there's a particular class that's giving your model extra trouble. If so, you might need to tweak your datasets by gathering more data for the more difficult classes.

But you should also consider trying a different prediction window size, since sometimes models are better at recognizing different classes using different windows — your goal is to find the one that gives you the best overall performance.

> **Note**: If you find different activities are only recognized at different window sizes, then you might need a more complicated setup using multiple models, each trained to spot a subset of your classes. You may know about "ensemble" methods already, where multiple models combine their predictions to produce a final answer. The technique required here is *almost* an ensemble, but it's slightly more complex, because it requires extra logic in your app to ensure your different models predict on different schedules. That won't be covered further in this book.

Here's the confusion matrix for the model included with the book:

target_label	predicted_label	count
shake_it	rest_it	96
rest_it	drive_it	17
drive_it	drive_it	5443
chop_it	rest_it	181
chop_it	shake_it	176
rest_it	rest_it	13164
chop_it	chop_it	5118
drive_it	rest_it	184
shake_it	shake_it	5367
rest_it	chop_it	62
rest_it	shake_it	77

[11 rows x 3 columns]

Confusion matrix for trained model

The first thing you might notice is the large numbers of predictions — your dataset didn't have nearly that many gestures, did it? That's because it's providing a prediction for every *window*, not every *activity*. So it makes many predictions over any single activity sequence, and this shows the results for all of them.

Next, notice the predictions with the highest counts: They are all *correct* predictions, with over 5,000 for each of the gestures and over 13,000 for `rest_it`. On the other hand, each of the *incorrect* predictions happened only a small number of times, with the fewest being `rest_it` predicted as `drive_it` only 17 times and the most being `drive_it` predicted as `rest_it` 184 times. Almost all of the errors involved the `rest_it` activity, which you would expect. After all, you never know what people did while recording their rest data — they may even have been doing the other gestures!

In fact, notice that the only incorrect predictions that did *not* involve the `rest_it` activity were the 176 `chop_it` gestures predicted as `shake_it`. It makes sense that there might be mistakes between these two gestures, since chopping is actually quite similar to shaking — if a person chops very quickly it might appear similar to a shake, or if the shake is over exaggerated it might look a bit like a chop.

Keep in mind, your model's performance in your app may be better than its test results, because you'll ignore low-confidence predictions. But if you're still unhappy with the model, you should create another one. Some sticklers will tell you not to reuse your training data because you'll be leaking data into your model. That's *technically* true, and you should listen to them…except you probably won't. Unless you have an endless stream of free data available, you probably don't have the luxury of testing just once per test set. The good news is — in many cases that's probably ok. For example, with a project like this one, you want the *app* to perform well, and your test data is just a tool to help you get there. Once your model works well on that, you'll run it on actual devices with live data from real people. Those are your *real* tests, and they are always unique — so you can even tell those sticklers you're using a new test set each time! Metrics like test accuracy are great, but be sure to beta test your app with many people before releasing it, so you know it really works the way you want it to.

When you think your model is ready for testing on a device, go ahead and save it with the following code:

```
model.export_coreml("GestureClassifier.mlmodel")
model.save("GestureClassifier")
```

This exports it to Core ML for use in your app, and saves a copy that you can reload in Python in case you want to work more with it later.

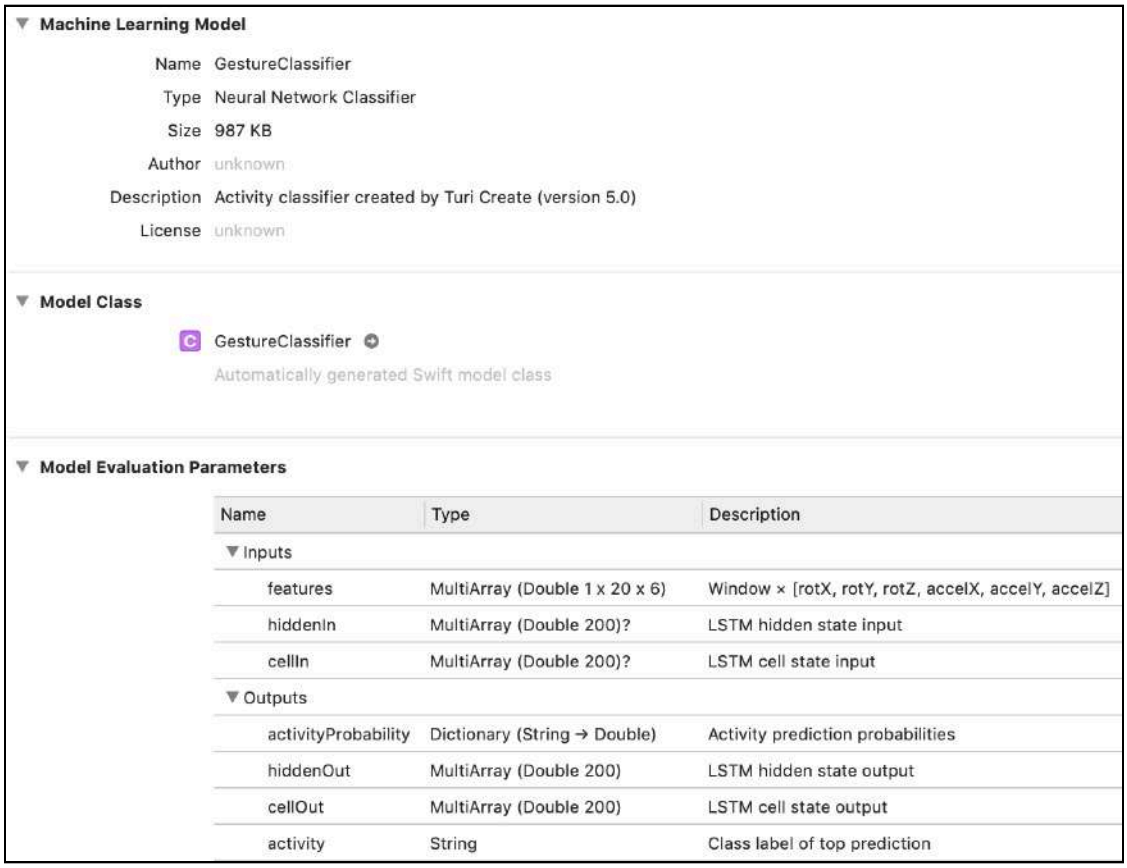
You've saved your model, trained to analyze iPhone motion data and recognize when that data indicates specific gestures have occurred. It seems to perform well, at least when tested against *recorded* motion data. That's a good start, but you want it to work in real time, evaluating motion data *as it's produced on the device*. For that you need an app! Continue reading to learn how to build one.

Getting to know your model

Open the **GestureIt** starter project in Xcode. If you've gone through the chapters leading up to this one, then you've already practiced adding Core ML models to your projects — find the **GestureClassifier.mlmodel** file you created when you saved your trained model in the previous section and drag it into Xcode. Or, if you'd like to use the model we trained on the provided dataset, add **notebooks/pre-trained/GestureClassifier.mlmodel** instead.

> **Note**: Now that you have the model in Xcode, the rest of this section is all theory. You can safely skip to the section **Classifying human activity in your app** if you aren't interested in this discussion right now.

Select **GestureClassifier.mlmodel** in the Project Navigator and you'll see the following, which is similar to — but also quite different from — models from **Section 1** of this book:



Looking at the mlmodel file

Here you can see `GestureClassifier` is an activity classifier from Turi Create. It's under 1MB — that's pretty good for a neural net that isn't taking advantage of models pre-installed on iOS, as did some of the ones you made earlier. But then comes the Model Evaluation Parameters section, where things get a bit more complicated.

First, the more recognizable items:

- `features`: `MLMultiArray` of `Doubles` you'll pass as input. If you haven't seen `MLMultiArray` before, don't worry, it's nothing too new. It's basically just a multidimensional array that Core ML uses to work efficiently with data. This one is sized to store a single prediction window's worth of values for each of the features you used while training: rotation and acceleration due to the user around the X, Y and Z axes.
- `activityProbability`: Dictionary the model outputs that includes the probabilities assigned to predictions for each of the classes. In the case of this project, that means probabilities for the gesture types "rest_it," "shake_it," etc.
- `activity`: String the model outputs indicating the activity class predicted with the highest probability.

But what about these other things: `hiddenIn`, `cellIn`, `hiddenOut` and `cellOut`? And what's this mysterious new acronym "LSTM" mentioned in all their descriptions?

Recurrent neural networks

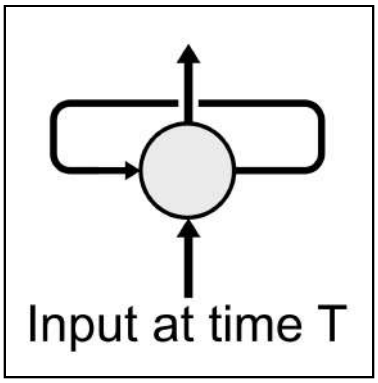
So far in this book you've mostly dealt with convolutional neural networks — CNNs. They're great for recognizing *spatial* relationships in data, such as how differences in value between nearby pixels in a two-dimensional grid can indicate the presence of an edge in an image, and nearby edges in certain configurations can indicate the ear of a dog, etc. Another kind of network, called a recurrent neural network — RNN — is designed to recognize *temporal* relationships. Remember, a sequence generally implies the passage of time, so this really just means they recognize relationships between items in a sequence.

To do that, they look at sequences one item at a time, and produce an output for each item based on the *current* item *and* on the *output* they produced for the *previous* item. But what does that really mean?

Consider how you read the following sentence: "The quick brown fox jumps over the lazy dog." You don't look at each word individually and ignore the rest, right? Instead, each element of the sentence adds to your understanding. What's happening? Jumping. Who's jumping? The fox. What's it look like? It's brown. And so forth.

RNNs are designed to do something similar, interpreting each element in a sequence by considering the elements they've already seen.

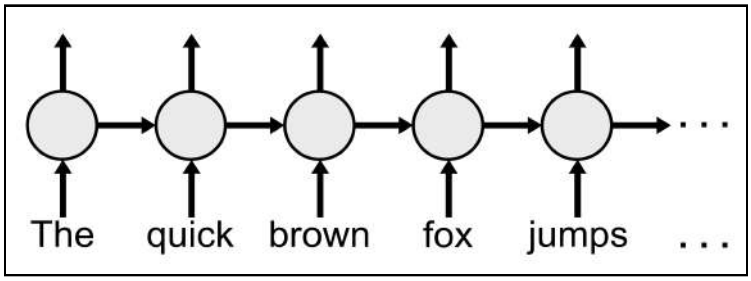
So what's that look like as a network? You may come across RNN diagrams like this one:



Looping nature of RNN layers

In this image, the circle represents a single *layer* of an RNN, not a single node. Remember from what you learned earlier — a layer in a neural network can contain any number of nodes, with more nodes providing that layer with more representation power. Input elements in a sequence are referenced by timesteps, and layers process the element at time **T** by looking at both that input *and* the layer's own output from the *previous* input at timestep **T-1**. That loop where the layer's output feeds back into itself is known as a recurrent connection — i.e. it occurs repeatedly — giving RNNs their names.

While diagrams like that might be useful to describe the theory behind an RNN, it can be easier to visualize if you think of the network as multiple layers. Looked at this way, each successive layer receives the next element in the input sequence along with the output from the previous layer. The following image shows what that would look like when processing the earlier example sentence:



RNN layer's recurrent behavior shown as separate layers

Now it's clearer how the layers process a sequence one element at a time, combining each element of the input with the output generated for the previous element. Notice that the RNN cannot process a given item until after it has processed the items that

came before it in the sequence. It's this serial nature of RNNs that makes them slower than other neural networks, such as CNNs. This is true of both training and inference.

These recurrent connections allow RNN layers to adjust their output based on what they've seen so far in the sequence, much like you interpret the word "fish" in the following two sentences differently depending on the words before it: "I like to fish." and "I like fish." In the first sentence, the speaker likes to *catch* fish, or at least try to; in the second, the speaker probably likes to *eat* fish, but may also just enjoy fish as an animal in general. Either way, the definition of "fish" depends on its context.

> **Note**: The previous diagram shows *two* outputs from each layer, one going to the next layer and one going off to...somewhere? That's to indicate how the output for each timestep can be used within an RNN layer, through the recurrent connection, as well as passed along to the next, possibly also recurrent, layer of the network. The final output of an RNN layer can be either the output for the sequence's last timestep, *or* the entire sequence of outputs the layer generated while processing the input sequence.

Early implementations of this basic RNN design showed it was *possible* to learn relationships across timesteps in a sequence, but they don't actually do it very well. Due to how the underlying math works, they take too long to train and can't relate items separated by too many timesteps. For example, imagine an RNN processing our example sentence — it would likely remember the fox is brown, but it might have forgotten there is a fox at all by the time it gets to the dog at the end of the sentence.

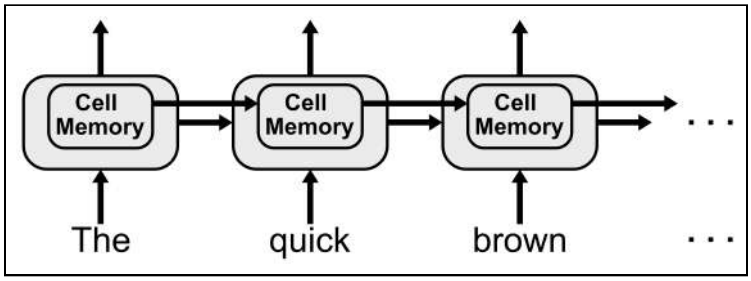
In reality a basic RNN could probably handle short sentences like that, but relationships span much greater distances in many sequences. To continue with our reading example, while words within a sentence are surely related to each other, they can also be related to words in sentences earlier in the same paragraph, many pages ago in the same chapter or even several chapters ago in a book. The distance between relationships can be arbitrarily long, and basic RNNs simply aren't suited to handle that.

But then along came LSTMs.

Long short-term memory

The acronym LSTM stands for the odd-sounding phrase long short-term memory, and it refers to a different kind of recurrent unit capable of dealing with relationships separated by longer distances in the sequence. Conceptually, the following diagram shows the pertinent details of how an LSTM works.

It uses our earlier sample sequence and shows the recurrent steps unrolled as separate layers to help clarify its behavior:



LSTM layer's recurrent behavior shown as separate layers

As you can see, an LSTM is a recurrent unit enhanced with an internal memory. LSTMs are used just like regular recurrent layers, but instead of processing only their input and previous output, an LSTM also considers the contents of its memory. And instead of just producing an output, the LSTM can also update its memory to remember (or forget) information it thinks is important about the sequence so far.

But terms like remembering, forgetting and thinking make it sound like LSTMs have more agency than they really do. Just like with other parts of a neural network, the LSTM's "memory" is really just a bunch of numbers that get manipluated by various math functions. And it doesn't really *choose* to remember or forget, it just learns weights that cause it to react differently to different sequences.

> **Note**: LSTM units are more complex than they appear in the above diagram, with each cell made up of four layers combined by various math operations. If you're interested in their inner workings, check out this excellent blog post. But the truth is, unless you're working to invent new types of neural network layers, you probably won't need to know those low-level details.

The important thing to know about LSTMs is that they train much more easily than the basic RNNs that came before them, and they offer much better performance. Most RNNs in use today use some variation of the LSTM unit, as is the case with the activity classifier you trained in Turi Create.

Turi Create's activity classifier

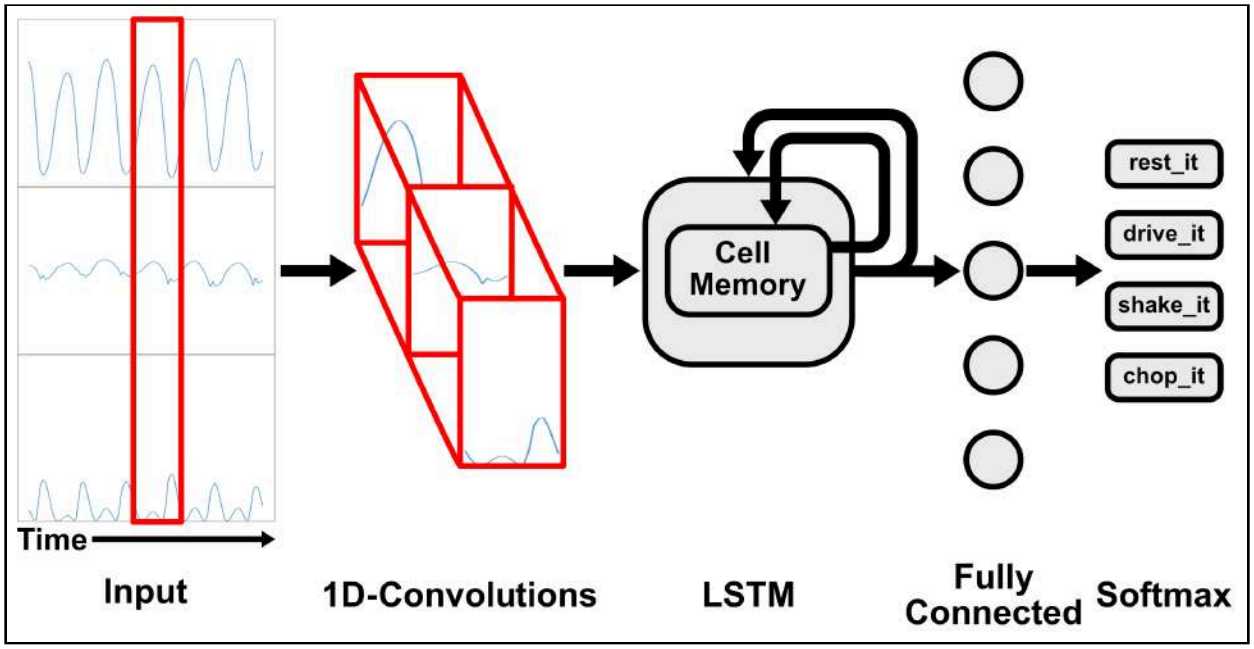
So far we've been discussing RNNs — and more specifically, LSTMs — as deep learning's solution to working with sequences. But it turns out that's not the whole story. Many state of the art results have been achieved using other network types, especially our old friend the CNN.

Current research trends seem to be moving away from RNNs because they don't scale with hardware as well as other models do. But for now, recurrent models are still a popular choice in practice.

What approach does Turi Create's activity classifier take? It's actually a combination of a CNN *and* an RNN. It uses convolutional layers to extract features from short sequences — the prediction windows mentioned earlier in the chapter — and it uses an LSTM layer to reason over sequences of predictions.

That lets it recognize sporadic activities, such as the gestures you trained your model to classify, as well as activities spanning longer periods of time, perhaps made up of several smaller ones. For example, imagine the following sequence of activities: putting a teabag in a cup, pouring hot water in a cup, waiting patiently, and removing a teabag from a cup. Each of those individual activites might be recognizable from small sequences of data — like what you could provide in a single prediction window. But when that *series* of activities occurs over multiple prediction windows, then the model might be able to recognize the overarching activity of making a cup of tea.

The following diagram shows a high level overview of Turi Create's activity classifier:



Turi Create's activity classifier architecture

You provide a sequence of sensor data as input — one prediction window's worth — and the model's first layer treats each input feature as a separate channel and performs a one-dimensional convolution over them. A 1D convolution is just like the 2D convolutions you've already used, except it uses kernels that are vectors instead of matrices.

Each kernel is the length of the prediction window and gets applied to all the input features to produce a new output channel. The current version of the code applies 64 such kernels.

The convolutional layer in this diagram may seem confusing because it *looks* like the waves are two-dimensional, but these are actually just vectors with numbers in them that we are *displaying* as a 2D image. To display a vector in two dimensions, we treat each item's index in the vector as its value along the X axis.

Remember from the discussions on transfer learning earlier in the book, how the pre-trained CNN model extracts features from images and then the layers you train use those extracted features as inputs? This is basically what Turi Create's model does, except the CNN isn't pre-trained. The CNN layer learns to output a vector encapsulating any interesting temporal features found within the prediction window. For example, maybe it notices certain patterns of peaks and valleys that are helpful when identifying a shaking phone. These extracted features flow into the LSTM layer as if they were a single item in a sequence.

> To understand why CNNs might be well suited to this task, it can help to think of this as a vision problem instead: Imagine you *plotted* the sensor data for a prediction window, similar to what we show in the previous diagram, and then passed that *image* to a CNN. If CNNs can learn to recognize dogs in images, they should be capable of learning to recognize patterns in sequences just like the ones you saw earlier when exploring the dataset.

After the LSTM layer receives the extracted features from the CNN layer, it produces an output based on those features *combined* with its own internal memory and its output from the *previous* prediction window. The LSTM's output passes through fully connected layers with batch normalization and dropout, and finally a softmax layer that outputs probabilities for each of the classes the model knows about. You learned about all those layer types earlier in the book so they aren't discussed here.

This talk about internal memory and previous predictions brings up an important question: What about when a sequence *doesn't* relate to those that came before it? Data doesn't always arrive as one long, unbroken stream, so do you really want your model to always consider its past predictions part of the current sequence?

Well, that *finally* brings us back to those new items you saw in Xcode: `hiddenIn`, `cellIn`, `hiddenOut` and `cellOut`. The names may seem backwards, but `hiddenOut` is the output from the LSTM itself, while `cellOut` is the LSTM's internal memory state after making the prediction. And `hiddenIn` and `cellIn` are the inputs you use to pass to the model

those outputs from the previous prediction. Each of these is a vector of 200 `Doubles` stored as an `MLMultiArray` — you don't need to worry about that, it's just how the model's LSTM layer encodes its state information.

So to indicate the start of a new sequence, you'll pass `nil` to the model for both `hiddenIn` and `cellIn`. On the other hand, when the current prediction is picking up where the last one ended — as will often be the case with streaming motion data — you'll take the `hiddenOut` and `cellOut` values from the previous prediction and pass those back to the model as `hiddenIn` and `cellIn`, respectively. Using the output and memory from the previous step like this allows the LSTM to recognize longer sequences. Continuing with our text example, it's as if the first prediction window you pass is for the word "The," the next window is for "quick," then "brown" and so on.

This whole chapter has been talking about classifying sequences of sensor data, but it turns out the model you made with Turi Create is looking at its inputs in two different ways — as sequences of sensor data, and as sequences of *sequences* of sensor data. The prediction window contains enough information to classify the first kind of sequence, but these extra inputs and outputs allow the LSTM portion of the network to reason over longer periods of time to classify the second kind of sequence.

> While models combining CNNs and LSTMs have achieved state-of-the-art results for tasks such as activity detection and speech recognition, there are also other techniques that deliver excellent performance when working with sequences. These include: Attention — a sort of memory added to other networks that helps guide their focus; Transformers — networks that use attention exclusively instead of recurrent or convolutional layers; and Temporal Convolutional Networks — CNNs designed for processing sequences. And new research seems to appear on a weekly basis, so there may be even more options by the time you're reading this. You'll get a chance to use some of these in later chapters.

Classifying human activity in your app

Now that you've added your model to the project and have a better idea of how it works, take a quick look through the project to see what else is there. The project's **Info.plist** file already includes the keys necessary to use Core Motion, explained earlier when you built the GestureDataRecorder project. GestureIt's interface (not shown here) is even simpler than GestureDataRecorder's — it's just two buttons: Play and Instructions. Choosing Instructions shows videos of each gesture, and Play starts a game.

While playing, the game speaks out gestures for the player to make, awarding one point for each correctly recognized gesture. The game ends when the app recognizes an incorrect gesture or if the player takes too long.

The project already includes the necessary gameplay logic, but if you play it now you'll always run out of time before scoring any points. If you want it to recognize what the player is doing, you'll need to give it a brain.

All the code you write for the rest of this chapter goes in **GameViewController.swift**, so open that file in Xcode to get started.

This file already imports the Core Motion framework and includes all the necessary code to use it. Its implementations of `enableMotionUpdates` and `disableMotionUpdates` are almost identical to what you wrote in the GestureDataRecorder project. The differences are minor and you should have no problem understanding them. As was the case with that project, this file contains a method named `processMotionData` that the app calls whenever it receives device motion data. At the moment it's empty, but you'll implement it later. For now, import the Core ML framework by adding the following line with the other imports near the top of the file:

```
import CoreML
```

In order to keep your code tidy and more easily maintainable, you'll store numeric configuration values as constants in the `Config` struct at the top of the class, just like you did in the GestureDataRecorder project. To start, add the following three constants to that struct:

```
static let samplesPerSecond = 25.0
static let numFeatures = 6
static let windowSize = 20
```

These values **must match** those of the model you trained. You'll use `samplesPerSecond` to ensure the app processes motion data at the same rate your model saw it during training. The dataset provided in this chapter's resources was collected at 25 samples per second, so that's the value used here. However, change this value if you train your own model using data fed to it at a different rate.

> **Note**: In case it's not clear why the app's `samplesPerSecond` must match that of the dataset used to train your model, consider this example: Imagine you trained your model using a prediction window of 200 samples, on data collected at 100 samples per second. That means the model would learn to recognize actions seen in highly detailed, two-second chunks. If you then ran this app with `samplesPerSecond` set to `10`, it would take *20 seconds* to gather the expected 200

> samples! Your model would then look at 20 seconds of data but evaluate it as if it was *two* seconds worth, because that's how it learned. This would almost certainly make the patterns in these sequences appear different from what the model saw during training. Remember, machine learning models only work well with data that is similar to what they saw during training, so getting the sampling rate wrong here could make a perfectly good model seem completely broken.

Likewise, the model discussed in this chapter expects data in blocks of 20 samples at a time, with six features for each sample. The `windowSize` and `numFeatures` constants capture those expectations.

> **Note**: If you ever work with a model and aren't sure about its expected number of features and window size, you can find them by looking at the `.mlmodel` file in Xcode's Project Navigator. However, this does not include information about the rate at which motion data needs to be processed, so that you'll just need to know.

Now that you've added those constants, you can complete the starter code's implementation of `enableMotionUpdates` by setting the `CMMotionManager`'s update interval. To do so, add the following line inside `enableMotionUpdates`, just before the call to `startDeviceMotionUpdates`:

```
motionManager.deviceMotionUpdateInterval = 1.0 / Config.samplesPerSecond
```

Just like you did in GestureDataRecorder, this tells `motionManager` to deliver motion updates to your app 25 times per second — once every 0.04 seconds.

Core ML models, such as `GestureClassifier`, expect their input in the form of `MLMultiArray` objects. Unfortunately, working with these objects involves quite a bit of type casting. Swift's type safety is great, and explicit type casting forces developers to be more thoughtful about their code — but I think we can all agree code gets pretty ugly when there's *too* much casting going on. To keep that ugliness — and the extra typing it requires — to a minimum, you'll be isolating any `MLMultiArray`-specific code within convenience methods. Add the first of these methods below the `MARK: - Core ML methods` comment in `GameViewController`:

```
static private func makeMLMultiArray(numSamples: Int) -> MLMultiArray? {
  return try? MLMultiArray(
      shape: [1, numSamples, Config.numFeatures] as [NSNumber],
      dataType: MLMultiArrayDataType.double)
}
```

This function takes as input the number of samples the array should contain. It then attempts to make an `MLMultiArray` with a shape and data type that will work with our model: `[1, numSamples, Config.numFeatures]` and `double`, respectively. Notice how the shape needs to be cast as an array of `NSNumbers` — you'll see a lot of those types of casts when dealing with `MLMultiArrays`.

Attempting to create an `MLMultiArray` can fail by throwing an exception. If that occurs here, the `try?` causes this function to return `nil`. This might occur in situations such as when there is insufficient memory to create the requested array. Hopefully it doesn't ever happen, but you'll add some code to deal with that possibility a bit later.

Now that you have that handy function, you'll use it to create space to store motion data to use as input to your model. Add the following property, this time to the area under the `// MARK: - Core ML properties` comment:

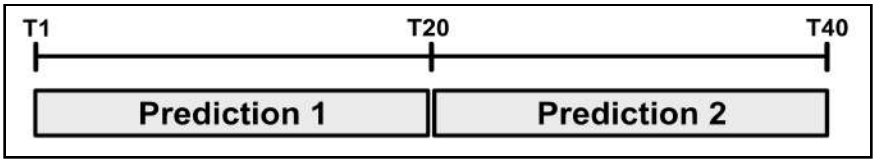
```
let modelInput: MLMultiArray! =
  GameViewController.makeMLMultiArray(numSamples: Config.windowSize)
```

This creates the `modelInput` array, appropriately sized for the model you trained. Later you'll populate this array with motion data prior to passing it to your model for classification.

> **Note**: You may have noticed that `modelInput` is declared as an implicitly unwrapped optional, but `makeMLMultiArray` can return `nil`. Doesn't that mean you run the risk of crashing your app elsewhere if you try to unwrap `modelInput` when it's `nil`? Normally, that would be a problem, but later you'll add some code that ensures this can never happen.

Overlapping prediction windows

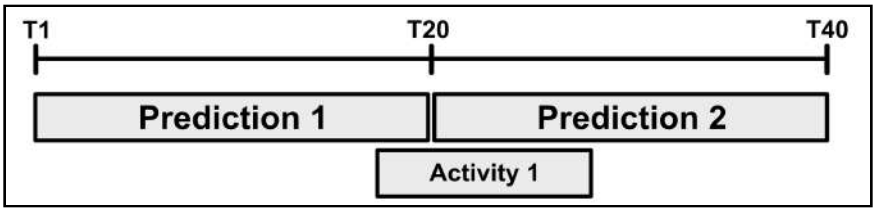
Now, you *could* work with just a single `MLMultiArray` like `modelInput`, repeatedly filling it up over time and passing it to the model. The diagram below shows what it would look like making two predictions with a window size of 20:



Reusing a single array to make predictions

As the diagram above shows, the array would fill up between times **T1** and **T20**, then you'd pass it to your model to make your first prediction. After that you'd reuse the array between times **T21** and **T40**, before passing it to your model again to make your second prediction.

This technique is the simplest to code and is fine for many apps. However, there are times when doing this would cause some problems. Consider the situation shown in the following diagram, where an activity you want to recognize spans across prediction boundaries:

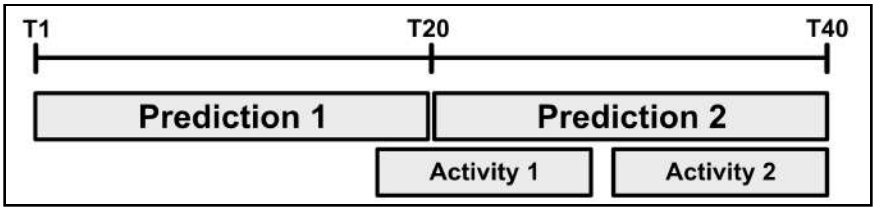


What if an activity spans across predictions?

In this case, a few things might happen. If the amount of data in the first prediction window is sufficient for the model to recognize the activity, then no problem — it returns the correct classification. But if the model needs to see more activity data than is available in the first window, it won't be able to classify it correctly until its *second* prediction. In that case it takes longer than necessary to report the result, which makes your app feel sluggish. Or worse yet, all the non-activity data in the second window might make the second prediction fail to recognize the activity, too.

Delayed responses or inaccurate predictions — take your pick, but neither is a great option.

Now consider another problematic scenario, shown in the following diagram:

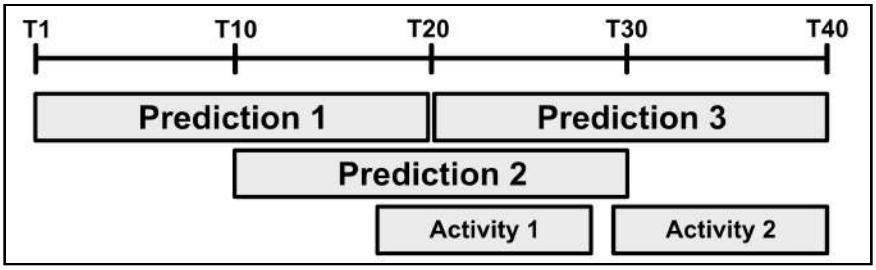


What if one prediction sees data for multiple activities?

Here there's one activity that spans across two predictions, just like before. But now a second activity occurs only within the second prediction window. In this case, assume the first prediction did not recognize anything, so now it's up to the second window to handle everything. How will it classify the two activities?

It can only make one prediction, so it will either correctly predict *one* of the activities, *or* it will become so confused that it fails to predict *either* of them. This isn't necessarily incorrect — it really depends on the app — but it's something you need to consider carefully.

In many cases it would be better if you could make predictions more often. You might try smaller prediction windows, but that isn't always an option because your model might need to see larger chunks of data to successfully recognize activities — that depends entirely on your specific data, model, and use case. But it turns out you *can* make predictions more often *without* changing the window size if you *overlap* your prediction windows, as shown in the following diagram:



Overlapping predictions

In this case, the first prediction sees data from times **T1** to **T20**, and the *third* prediction sees the data from times **T21** to **T40**. But now a *second* prediction window overlaps each of those, spanning the data from times **T11** through **T30**. Because this is like sliding the prediction window along the data (using offsets of 10 in this case), many people call these "sliding" windows.

An app using this design responds more quickly because it makes more predictions, and it's more accurate because it considers individual samples as part of multiple possible sequences. The first prediction window still might not recognize anything, but the second prediction would see the first activity — and predict it at **T30** instead of waiting until **T40**. And then the third prediction would recognize the second activity only 10 samples later. The app ends up feeling more responsive *and* it doesn't miss either activity.

Overlapping predictions *mostly* solves all of the problems mentioned earlier. But depending on how much data your model needs to see in order to make a prediction, and how much you overlap your windows, you still might run into missed or erroneous classifications. It's a matter of finding the best amount of overlap for your app.

You'll be implementing overlapping predictions in Gesture It, because you'll want fast response times to quickly evaluate the player's gestures.

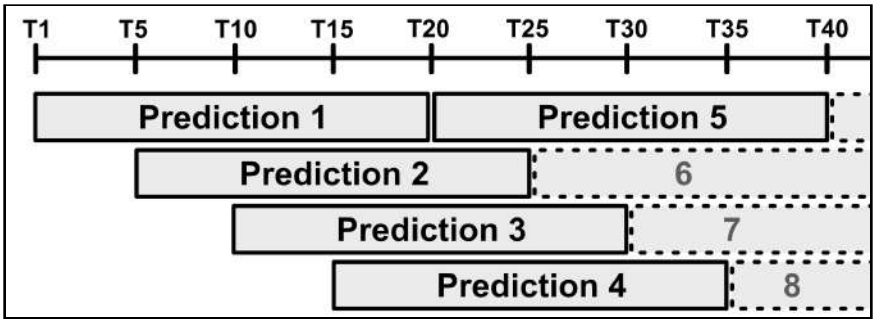
But if you were making an app that tracks the amount of time you spend jogging, for example, you would probably be fine with non-overlapping predictions made over longer periods of time (maybe even once every several seconds).

> **Note**: How much you overlap your predictions directly affects more than just accuracy and response time. More overlap means running inference with your model more often, and that extra processing could increase battery drain. And depending on how long it takes your model to make predictions, it might not even keep up with the pace of requests, causing your app to exhibit other performance problems. So test various options and settle on making predictions only as often as is necessary to achieve your goals.

To help define your prediction windows, add the following constants to the `Config` struct at the top of the file:

```
static let windowOffset = 5
static let numWindows = windowSize / windowOffset
```

Here you define `windowOffset` as five. This is not how much the window overlaps, but rather how far to offset the start of the window from the start of the previous window. With the `windowSize` of 20 you defined earlier, this makes `numWindows` equal four. That's how many prediction windows you'll have before you essentially wrap back around to the first one again. This should be clearer if you refer to the the following diagram, which shows how your predictions would overlap for the first 40 samples:



Gesture It's overlapping predictions — windowSize=20, windowOffset=5

With the settings you've made so far, Gesture It will take 0.8 seconds to respond with its first prediction, but then each successive prediction will occur every 0.2 seconds after that. (That's because `samplesPerSecond` is 25, so each sample takes 0.04 seconds to arrive. A `windowSize` of 20 looks at 20 x 0.04s = 0.8 seconds of data, and a `windowOffset` of 5 means each prediction occurs 5 x 0.04s = 0.2 seconds after the last one.)

Notice how different prediction windows overlap with various different combinations of other predictions. For example, Prediction Two sees the last 15 samples in Prediction One, and the first five samples in Prediction Five, along with 15 and 10 samples seen by Predictions Three and Four, respectively. And starting from Prediction Five, each window will process varying numbers of samples from *six* other prediction windows! All this overlap should help your model classify gestures quickly and accurately.

> **Note**: The integer division used to calculate `numWindows` means you'll never have a partial window. For example, if `windowOffset` were 20 with a `windowSize` of 50, you'd have two windows, one from **T1** to **T50** and another from **T21** to **T70**. The code you write in this app will handle that situation fine, but keep in mind that the predictions will not occur at a steady rate unless `windowSize` is evenly divisible by `windowOffset`. In this example, an offset of 20 would result in 20 samples between predictions one and two but 30 samples between predictions two and three.

The previous diagrams show what samples each prediction window should use, but how do you implement it? At the moment you've got a single `MLMultiArray` the size of *one* window, but now you need four. While you *could* create four different arrays to store this data, that would waste memory. Instead, you'll make one slightly larger array that will act as a buffer area for the most recent motion data, and each prediction window will look at the appropriate subset of that larger buffer when necessary.

Add the following constant to the `Config` struct, which defines the size of the buffer you'll create:

```
static let bufferSize = windowSize + windowOffset * (numWindows - 1)
```

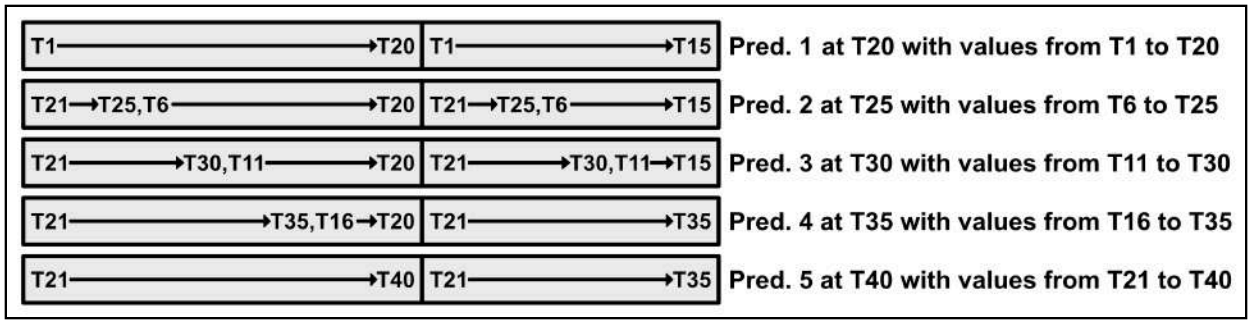
You define a buffer size large enough to hold one full window plus the space taken up by the offsets for the other windows. So for the settings you've used so far, Gesture It's buffer will hold 35 samples. Don't worry if it's not yet clear *why* this is the right size — you'll see soon.

Now add the following properties to manage the buffer. Put them with the other ML-related properies in `GameViewController`:

```
let dataBuffer: MLMultiArray! =
  GameViewController.makeMLMultiArray(numSamples: Config.bufferSize)
var bufferIndex = 0
var isDataAvailable = false
```

You create `dataBuffer` using the convenience method you created earlier. As new motion data arrives from the device, you'll use `bufferIndex` to determine where to store that data within the buffer. You'll set the `isDataAvailable` flag to `true` once the buffer contains enough data to perform its first prediction.

For the remainder of this discussion, please refer to the following diagram, which shows the buffer's contents at each prediction over the first 40 time steps:



Buffer contents over time

Think of the buffer as having two halves, with a full prediction window on the left and auxilliary storage on the right. The second "half" isn't a true half in this case, because it's smaller than the first, but that won't be a problem.

You'll increment `bufferIndex` as new data arrives, moving it across the first half of the buffer, and you'll reset it to the beginning whenever it reaches the buffer's midpoint. That is, `bufferIndex` will always be pointing to the next location to fill *within the first prediction window*. But whenever you store an item in the left half of the buffer, you'll also store it in the equivalent location in the right half. (You'll skip updates on the right side that would be out of bounds due to the size mismatch. You *could* make both sides the same size and then always store values in both places, but the approach used here saves some memory — usually a good thing for mobile apps.)

The top of the diagram shows what the buffer looks like after 20 timesteps. The left side contains data from times **T1** to **T20**, and the right side contains copies of times **T1** to **T15**. It's at this point that you'll reset `bufferIndex` to zero, set `isDataAvailable` to `true` and perform the first prediction using times **T1** to **T20**.

As data continues to arrive, you'll keep filling the left and right sides of the buffer simultaneously. After five more timesteps, you'll be ready to make the second prediction. As you can see in the second row of the diagram, the first five items of the buffer contain data from times **T21** to **T25**, but the next 15 items still contain data from times **T6** to **T20**. And because you've been updating both sides of the buffer, the first five items on the *right* contain data from times **T21** to **T25**, too.

So you can now make your second prediction using times **T6** to **T25** by looking at a window that crosses into the second half of the buffer.

This process continues indefinitely, but the diagram shows the contents of the buffer when making each of the first five predictions. The key point to realize is that after the first time `bufferIndex` reaches the midpoint of the buffer and resets to the start, it is always the case that the the *next* 20 items starting at `bufferIndex` contain data from the *previous* 20 time steps.

Phew. That was a lot of discussion about just a bit of code, so hopefully you're still here. Now back to the app!

Buffering motion data

Now you're going to add some code to handle `MLMultiArrays` that end up `nil`. Since each of these objects is required for the game to function properly, you are going to notify the player and force them to go back to the main menu. However, you may want to make your own apps more robust. For example, if the app successfully creates the smaller `modelInput` array but then fails on `dataBuffer`, you might want to fall back to a non-overlapping approach and notify the user that performance may be slightly degraded.

Add the following code inside `viewDidLoad`, immediately *above* the call to `enableMotionUpdates`:

```
guard modelInput != nil, dataBuffer != nil else {
  displayFatalError(error: "Failed to create required memory storage")
  return
}
```

Here you check to ensure that the app was able to create each of its required `MLMultiArray` properties. If not, you call `displayFatalError`, a method in the starter code that alerts the player with the given error message and then dismisses the `GameViewController`.

> **Note**: The starter code enables motion updates when it loads the game view and stops them when the game is over. However, your production apps should be more robust than that. Be sure your apps are good iOS citizens and have them properly handle situations such as getting paused for incoming phone calls, etc.

It will receive motion updates `Config.samplesPerSecond` times each second. For each update, you'll need to store the appropriate features in `dataBuffer`, the `MLMultiArray` you created earlier. You'll wrap this logic in helper methods to keep things easier to read, so add the following code to `GameViewController`:

```
// 1
@inline(__always) func addToBuffer(_ sample: Int, _ feature: Int,
                                   _ value: Double) {
  // 2
  dataBuffer[[0, sample, feature] as [NSNumber]] = value as NSNumber
}
// 3
func bufferMotionData(_ motionData: CMDeviceMotion) {
  // 4
  for offset in [0, Config.windowSize] {
    let index = bufferIndex + offset
    if index >= Config.bufferSize {
      continue
    }
    // 5
    addToBuffer(index, 0, motionData.rotationRate.x)
    addToBuffer(index, 1, motionData.rotationRate.y)
    addToBuffer(index, 2, motionData.rotationRate.z)
    addToBuffer(index, 3, motionData.userAcceleration.x)
    addToBuffer(index, 4, motionData.userAcceleration.y)
    addToBuffer(index, 5, motionData.userAcceleration.z)
  }
}
```

While these methods are essentially just updating an array, there are some important things to note:

1. The `addToBuffer` function isolates the `NSNumber` casts to one line, which keeps the code at `// 5` easier to read. Declaring it with `@inline(__always)` tells the Swift compiler to replace any calls to this function with the contents of the function itself, ensuring your code executes as quickly as possible. Swift is good about inlining these one-line functions on its own, but including this tag makes your intention clear.

2. This line sets a single value inside `dataBuffer`. That `MLMultiArray` is arranged as a 3-dimensional tensor, indexed as [batch, sample, feature]. The model's batch size is always one, so the first index value here is always `0`. The `sample` and `feature` indices are passed as arguments to this method.

3. You'll call `bufferMotionData` from within `processMotionData`, which you'll write next. It copies motion data into the correct locations in the large buffer backing the overlapping prediction windows described earlier.

4. This `for` loop ensures each value is stored at the position indexed by `bufferIndex`, as well as a position that is one window-span *later* in the buffer. The `continue` statement ensures that second write attempt is not outside the buffer's bounds, which would crash the app. For more details about how the overlapping windows work, refer to the discussion earlier in this chapter.

5. Here you call `addToBuffer` repeatedly to save the relevant data from the `CMDeviceMotion` object passed to this method. It's **extremely important** to store only the features your model expects, and in exactly the order it expects them. This was all determined when you trained the model, but you can verify the information by inspecting the `.mlmodel` in Xcode's Project Navigator. Be sure to double check this step, because mistakes here will make your model function incorrectly — sometimes failing with a crash, sometimes by underperforming, and even sometimes by *appearing* to work! That last one might sound ok, but it just means you've got some lucky input and it's unlikely to work well for long.

Your code so far only adds data to `dataBuffer`, but you'll eventually need to pass `modelInput` to your ML model. That's because your model expects to see an `MLMultiArray` with `modelInput`'s specific shape, not the larger buffer you created to implement overlapping windows. So you'll need to copy data between these structures.

To make those copies as fast as possible, you'll be using low level pointers to copy chunks of memory directly. To do that, you need to know the exact number of bytes you want to access, so add the following constants to the `Config` struct:

```
static let windowSizeAsBytes = doubleSize * numFeatures * windowSize
static let windowOffsetAsBytes = doubleSize * numFeatures * windowOffset
```

Here you calculate the number of bytes it takes to represent a prediction window within an `MLMultiArray`, as well as the number of bytes necessary to represent the offset between prediction windows. The constant `doubleSize` referenced in these calculations already exists in the starter code — it stores how many bytes are used by a single `double`. You'll use these constants soon.

You're now all set to fill in the placeholder `processMotionData` method. Insert the following code into that method:

```
// 1
guard expectedGesture != nil else {
  return
}
// 2
bufferMotionData(motionData)
// 3
bufferIndex = (bufferIndex + 1) % Config.windowSize
```

```
// 4
if bufferIndex == 0 {
  isDataAvailable = true
}
// 5
if isDataAvailable &&
   bufferIndex % Config.windowOffset == 0 &&
   bufferIndex + Config.windowOffset <= Config.windowSize {
  // 6
  let window = bufferIndex / Config.windowOffset
  // 7
  memcpy(modelInput.dataPointer,
         dataBuffer.dataPointer.advanced(
           by: window * Config.windowOffsetAsBytes),
         Config.windowSizeAsBytes)
  // 8
  // TODO: predict the gesture
}
```

This is the meat of your data pipeline, so look carefully at what's going on here:

1. The starter project uses `expectedGesture` to keep track of what gesture the player should be making. This value will be `nil` whenever the game is not expecting a gesture, and this `guard` statement ensures this method does not process motion data in those cases.

2. Here's where you call the method you recently added, `bufferMotionData`. You pass it the `CMDeviceMotion` object given to this method, and it stores the motion data in the appropriate locations within `dataBuffer`.

3. Next you update `bufferIndex` to keep track of the next available space in the buffer. You're incrementing it by one, and looping it back around to zero when it reaches the end of the first window.

4. Here you check to see if `bufferIndex` is zero. Because `bufferIndex` is updated *before* this line, it can only ever be zero *after* it has exceeded `Config.windowSize` and wrapped back around at least once. At that point, you update `isDataAvailable` to indicate you have at least one full window's worth of data.

5. This `if`-statement ensures you make predictions at the correct times. It first checks `isDataAvailable` to make sure at least one window is full. Then it checks to see if `bufferIndex` is at the boundary of a window. Because `bufferIndex` resets when it reaches the end of the first window, you can only reliably check when it's at the *start* of most windows, not the end. This line determines that by checking to see if `bufferIndex` is some multiple of the window offset. It also verifies that there is a full `windowOffset` worth of space after this position in the window. That final check is just a precaution in case you ever use a window size that is not evenly divisible by

the offset size. Without that check, the code at // 7 would crash your app when it tried to access invalid memory. If all these checks pass, then the function knows it's OK to make a prediction.

6. Here you determine which prediction window you're working with so you'll know which data to access from the buffer.

7. Now you need to copy the samples for `window` from `dataBuffer` into `modelInput`. Conveniently, `MLMultiArray` objects expose a pointer for low level access to their backing memory via their `dataPointer` property, so here you take advantage of that fact and use `memcpy` to copy a window-sized chunk of memory directly from `dataBuffer` into `modelInput`. To locate the start of the window, you use the pointer's `advanced(by:)` method and some math to move it the appropriate number of bytes from the start of the buffer. Be extremely careful with `memcpy`: Getting anything wrong here will at best give you the wrong results, and at worst will crash your app.

8. Here is where you will eventually attempt to make your prediction. But you'll need to write just a bit more code before you do.

Making predictions with your model

At long last, your project is ready to start recognizing gestures. Almost. So far the app contains a lot of data processing and business logic — it still needs the machine learning bit! Add your gesture recognition model into the app by initializing the following property with the other ML-related properties in `GameViewController`:

```
let gestureClassifier = GestureClassifier()
```

Xcode autogenerated the `GestureClassifier` class when you first dragged the `.mlmodel` file into the project, so all you have to do is instantiate it like this and then call its `prediction` method with the appropriate inputs. It's almost *too* easy, right?

Well, it *would* be if that's all it took. Recall from the earlier discussion about the model's inputs and outputs, the LSTM portion of the network requires you to provide it with its internal memory and output from the previous prediction. That means you'll need to store that information each time you make a prediction and then pass it back to the model when making the next one. To help with that, Xcode generated the `GestureClassifierOutput` class at the same time it made `GestureClassifier`. This class conveniently encapsulates all four of the model's outputs so you can save them for later use.

However, you've implemented your predictions using four overlapping windows, which means consecutive predictions aren't actually continuations of each other. That is, the first sensor reading in a prediction window is *not* the reading immediately after the last one in the previous window. Instead, it is a value *within* the previous window, offset from its start by `Config.windowOffset` samples. Because of that fact, it wouldn't make sense for the LSTM's internal state to carry over from the *previous* prediction — it needs to use the state from *four* predictions ago instead. To keep track of all these outputs, you'll maintain an array of `GestureClassifierOutputs`, so add the following property for that:

```
var modelOutputs =
  [GestureClassifierOutput?](repeating: nil, count: Config.numWindows)
```

This array will hold one `GestureClassifierOutput` for each prediction window. The values are optional and will be `nil` for any window before you've used it. You can see the code for `GestureClassifierOutput` by selecting **GestureClassifier.mlmodel** in the Project Navigator, and then clicking the small arrow icon next to **GestureClassifier** in the **Model Class** section. It basically just provides properties to access the model's various outputs.

One last thing before you actually use your model. Earlier in the book, you read about how Core ML predictions come with probabilities which are essentially the model's confidence in the prediction. And you saw how the model will always produce *some* prediction, but not necessarily with much confidence. To avoid reacting to low probability predictions, you'll define a threshold that the probability must exceed to be considered sure enough to act upon. Add the following constant to `Config` at the top of the file:

```
static let predictionThreshold = 0.9
```

This basically means the model needs to be over 90% sure of a prediction before the app responds. This threshold was chosen after some playtesting, but it's mostly personal preference beyond a certain threshold. Values too low will make the app hallucinate gestures where there are none, so definitely avoid that, but other than that it's a matter of how touchy or picky you want the app to feel. Later, when you're done writing the app, try out different values here to see how they affect the gameplay.

With those small additions in place, it's now time to write the method that uses your trained model to recognize gestures. Add the following code to the end of `GameViewController`:

```
func predictGesture(window: Int) {
  // 1
```

```
    let previousOutput = modelOutputs[window]
    let modelOutput = try?
      gestureClassifier.prediction(
        features: modelInput,
        hiddenIn: previousOutput?.hiddenOut,
        cellIn: previousOutput?.cellOut)
    // 2
    modelOutputs[window] = modelOutput
    // 3
    if let prediction = modelOutput?.activity,
       let probability =
         modelOutput?.activityProbability[prediction] {
      // 4
      if prediction == Config.restItValue {
        return
      }
      // 5
      if probability > Config.predictionThreshold {
        // 6
        if prediction == expectedGesture {
          updateScore()
        } else {
          gameOver(incorrectPrediction: prediction)
        }
        // 7
        expectedGesture = nil
      }
    }
  }
```

You've written quite a bit of code already, but this method is really the only part of the app that actually uses machine learning. Here's what it does:

1. First it calls `gestureClassifier.prediction` to try to classify the motion data, and stores the results as `modelOutput`. Notice that you provide both `modelInput`, which you populated in `processMotionData`, as well as the LSTM's ouput and internal cell state from the previous prediction *for this window*. These values will be `nil` for each window's first prediction, and that's fine — this tells the classifier there is no history and it should initialize itself accordingly.

2. Then it stores the model's response in `modelOutputs` so you can access it next time you make a prediction for this window.

3. Next it grabs the predicted activity, along with the probability assigned to that prediction, from the model's output.

4. It checks for predictions of non-gestures — i.e. resting — and just ignores them.

5. For non-rest predictions, it checks to see if the probability exceeds the threshold you previously defined. If so, it considers it a real prediction; otherwise it does nothing and the app will continue processing motion events.

6. The next bit of code is game logic, but any app you write with a classification model will have something similar — a spot where you actually *use* the predicted value. If the model thinks the player made the correct gesture (i.e. the predicted gesture matches `expectedGesture`), then it calls `updateScore` to add a point; otherwise, the app thinks the player messed up and it calls `gameOver`.

7. Regardless of the prediction, the method resets `expectedGesture` to `nil` so that the app stops processing motion data for a while. The starter project's existing game logic will set this to a new gesture when appropriate.

> **Note**: The model class Xcode generates includes three different `prediction` methods, as well as a `predictions` (with an "s") method that batches multiple predictions in one call. This code uses the version that takes `MLMultiArrays` directly, but you might find situations where you'd prefer to use one of the other versions in your own apps, so be sure to check the generated code for options.

Now go back to that comment you added earlier — `// TODO predict the gesture` — and replace it with a call to the method you just wrote:

```
predictGesture(window: window)
```

You already calculated the correct prediction window inside `processMotionData`, and here you pass that to `predictGesture` to perform inference.

Now build the app and run it on your iPhone. (Sorry, no motion data in the simulator!) You might succeed with the first gesture, it won't take long before the app calls out a gesture and then immediately complains that you got it wrong. What gives?

Remember those fancy overlapping prediction windows? Well, that backing storage buffer you made still contains data from the previous sequences you were processing. So when the app asks for a new gesture, there's already a prediction window's worth of data just sitting there ready to be recognized — collected while you were making the *previous* gesture. And don't forget, recurrent models use state from previous predictions to help them make new predictions, because they assume the data is related. But when this app asks for a new gesture, it no longer wants the model to consider the prior data. Each new gesture needs a clean slate.

To correct this, you need to reset the buffer *and* the model's previous output states. Add the following method to `GameViewController`:

```
func resetPredictionWindows() {
  // 1
  bufferIndex = 0
```

```
    // 2
    isDataAvailable = false
    // 3
    for i in 0..<modelOutputs.count {
      modelOutputs[i] = nil
    }
  }
```

It's not much code, but it's vital in order for your app to function properly. Here's what it does:

1. Reset `bufferIndex` to zero to start filling the buffer from the beginning again. This ensures new predictions are based on relevant sequence data, rather than data in the buffer left over from prior sequences.

2. Reset `isDataAvailable` to `false` to keep the app from trying to perform another prediction before it has at least one full window.

3. Set everything in `modelOutputs` to `nil` to clear out any internal model state built up from previous predictions. This ensures the underlying LSTM cells in your `GestureClassifier` model don't remember anything from sequences related to earlier gestures and then try to use that information when making new predictions.

Now that you've defined that method, call it at the top of `startTimerForGesture`:

```
  resetPredictionWindows()
```

The existing game logic already calls `startTimerForGesture` whenever it notifies the player to perform a new gesture. With this addition, you ensure the predictions made for new gestures are not using any data that arrived while the app was processing earlier gestures.

That's it! Build and run again, and have fun Gesturing It! If the game times out too quickly for you to respond, increase the value of `Config.gestureTimeout`. Or, if you want to increase the challenge, see how low you can decrease it. How many correctly recognized gestures can you get in a row?

Key points

- Turi Create's activity classification API can help you easily make models capable of recognizing human activity from motion data. However, it can be used for more than just human activity detection — it's basically a generic classifier for numeric sequences.
- Core Motion provides access to motion sensors on iOS and WatchOS devices.
- When building a dataset, prefer collecting less data from more sources over more data from fewer sources.
- Inspect and clean your data before training any models to avoid wasting time on potentially invalid experiments. Be sure to check *all* your data — training, validation and testing.
- Try separating data from different sources into train, validation and test sets.
- Sample/shuffle sequential data as full sequences, not as individual rows.
- First train on just a small portion of your training set and make sure you can get the model to overfit. That's the best way to find problems with your model, because if it can't overfit to a small dataset, then you likely need to make changes before it will be able to learn at all.
- Train multiple models and run multiple experiments until you find what works best for your app.
- Prefer a balanced class representation. In cases where that's not possible, evaluate your model with techniques other than accuracy, such as precision and recall.
- Use overlapping prediction windows to provide faster, more accurate responses.
- RNNs process data serially, so they're slower than CNNs, both when training and performing inference.
- One-dimensional convolutions are commonly used to extract temporal features from sequences prior to passing them into RNNs.
- RNNs are a good choice for sequential data, with LSTMs being the most commonly used variant because they train (relatively) easily and perform well. However, they are not the only models that work well for sequences.

Challenges

A good way to get some practice with activity recognition would be to expand Gesture It. Adding new gesture types to the GestureDataRecorder project is a straightforward process, so start there, and then collect some data. Next, add your new data to the provided dataset and train a new model. Replace the model in the GestureIt project with your newly trained model, and make the few modifications necessary to add your new gesture to the game.

After that, you could try recognizing activities other than gestures. For example, you could make an app that automatically tracks the time a user spends doing different types of exercises. Building a dataset for something like that will be more difficult, because you have less control over the position of the device and more variation in what each activity looks like. In those cases, you'll need to collect a more varied dataset from many different people to train a model that will generalize well.

And keep in mind, these models work on other devices, too. The Apple Watch is a particularly fitting choice — a device containing multiple useful sensors, that remains in a known position on the user and is worn for all or most of the day. If you have access to one, give it a try!

Chapter 9: Sequence Predictions

Imagine your device could accurately predict the future! OK, that might be a stretch but, in this chapter, you'll learn how you can use recurrent neural networks (RNNs) to predict the outcome of some event given a sequence of events.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 10: NLP Classification

NLP classification has been popularized as a tool that can help categorize large bodies of text, such as identifying spam in your email or identifying related content from websites. In this chapter, you'll deep dive into training a model for use in a fun NLP classification project.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 11: Text-to-Text Transform

If you can classify text using NLP classification, you can surely convert text as well, right? In this chapter, you'll learn the answer to that question by working on an example of how to do language translation between two languages.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Want to Grow Your Skills?

We hope you enjoyed this book! If you're looking for more, we have a whole library of books waiting for you at https://store.raywenderlich.com.

New to iOS or Swift?

Learn how to develop iOS apps in Swift with our classic, beginner editions.

iOS Apprentice

The iOS Apprentice is a series of epic-length tutorials for beginners where you'll learn how to build four complete apps from scratch. Each new app will be a little more advanced than the one before. By the end of the series you'll be experienced enough to turn your ideas into real apps that you can sell on the App Store. These tutorials have easy to follow step-by-step instructions. https://store.raywenderlich.com/products/ios-apprentice

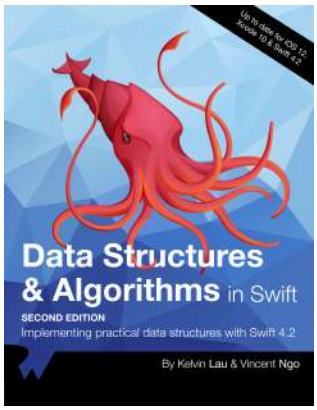
Swift Apprentice

This is a book for complete beginners to Apple's brand new programming language — Swift 4. Everything can be done in a playground, so you can stay focused on the core Swift 4 language concepts like classes, protocols, and generics. This is a sister book to the iOS Apprentice; the iOS Apprentice focuses on making apps, while Swift Apprentice focuses on the Swift 4 language itself. https://store.raywenderlich.com/products/swift-apprentice

Experienced iOS developer?

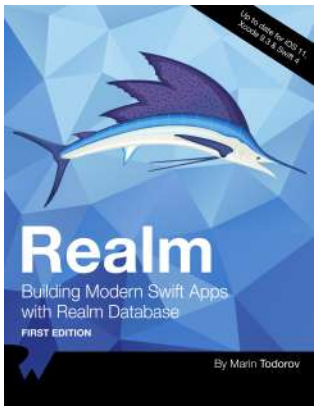
Level up your development skills with a deep dive into our many intermediate to advanced editions.

Data Structures and Algorithms in Swift



In Data Structures and Algorithms in Swift, you'll learn the most popular and useful data structures, and when and why you should use one particular datastructure or algorithm over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift

Realm: Building Modern Swift Apps with Realm Database

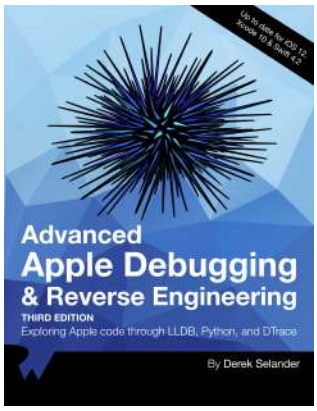


Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. In this book, you'll take a deep dive into the Realm Database, learn how to set up your first Realm database, see how to persist and read data, find out how to perform migrations and more. https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database

Server Side Swift with Vapor

If you're a beginner to web development, but have worked with Swift for some time, you'll find it's easy to create robust, fully featured web apps and web APIs with Vapor 3. This book starts with the basics of web development and introduces the basics of Vapor; it then walks you through creating APIs and web backends; creating and configuring databases; deploying to Heroku, AWS, or Docker; testing your creations and more. https://store.raywenderlich.com/products/server-side-swift-with-vapor

Apple Debugging and Reverse Engineering

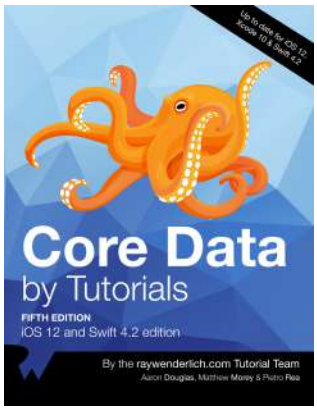


In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours. You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest. https://store.raywenderlich.com/products/advanced-apple-debugging-and-reverse-engineering

RxSwift: Reactive Programming with Swift

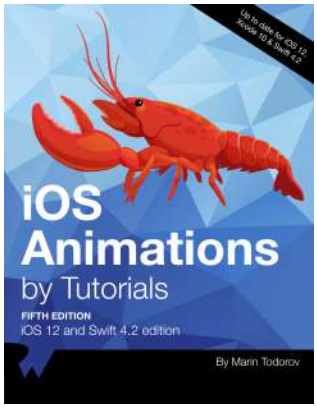
This book is for iOS developers who already feel comfortable with iOS and Swift, and want to dive deep into development with RxSwift. Start with an introduction to the reactive programming paradigm; learn about observers and observables, filtering and transforming operators, and how to work with the UI, and finish off by building a fully-featured app in RxSwift. https://store.raywenderlich.com/products/rxswift

Core Data by Tutorials



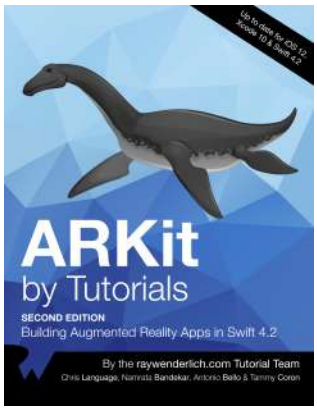
This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps. Start with with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more! https://store.raywenderlich.com/products/core-data-by-tutorials

iOS Aninmations by Tutorials



This book is for iOS developers who already know the basics of iOS and Swift 4, and want to dive deep into animations. Start with basic view animations and move all the way to layer animations, animating constraints, view controller transitions, and more! https://store.raywenderlich.com/products/ios-animations-by-tutorials

ARKit by Tutorials



Learn how to use Apple's augmented reality framework, ARKit, to build five great-looking AR apps: Tabletop Poker Dice; Immersive Sci-Fi Portal; 3D Face Masking; Location-Based Content and Monster Truck Sim. https://store.raywenderlich.com/products/arkit-by-tutorials

watchOS by Tutorials

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make Apple Watch apps for watchOS 4. https://store.raywenderlich.com/products/watchos-by-tutorials

tvOS by Tutorials

This book is for complete beginners to tvOS development. No prior iOS or web development knowledge is necessary, however the book does assume at least a rudimentary knowledge of Swift. This book teaches you how to make tvOS apps in two different ways: via the traditional method using UIKit, and via the new Client-Server method using TVML. https://store.raywenderlich.com/products/tvos-apprentice

Metal by Tutorials

This book will introduce you to graphics programming in Metal — Apple's framework for programming on the GPU. You'll build your own game engine in Metal where you can create 3D scenes and build your own 3D games. https://store.raywenderlich.com/products/metal-by-tutorials

Want to make games?

Learn how to make great-looking games that are deeply engaging and fun to play!

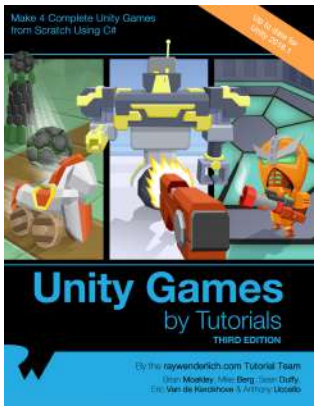
2D Apple Games by Tutorials

In this book, you will make 6 complete and polished mini-games, from an action game to a puzzle game to a classic platformer! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SpriteKit, you will learn a lot from this book! https://store.raywenderlich.com/products/2d-apple-games-by-tutorials

3D Apple Games by Tutorials

Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book! https://store.raywenderlich.com/products/3d-apple-games-by-tutorials

Unity Games by Tutorials



Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game! This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book! https://store.raywenderlich.com/products/unity-games-by-tutorials

Beat 'em Up Games Starter Kit

The classic beat 'em up starter kit is back — for Unity! Create your own side-scrolling beat 'em up game in the style of such arcade classics as Double Dragon, Teenage Mutant Ninja Turtles, Golden Axe and Streets of Rage. This starter kit equips you with all tools, art and instructions you'll need to create your own addictive mobile game for Android and iOS. https://store.raywenderlich.com/products/beat-em-up-game-starter-kit-unity

Want to learn Android or Kotlin?

Get a head start on learning to develop great Android apps in Kotlin, the newest first-class language for building Android apps.

Android Apprentice

The Android Apprentice takes you all the way from building your first app, to submitting your app for sale. By the end of this book, you'll be experienced enough to turn your vague ideas into real apps that you can release on the Google Play Store. The four apps you will complete will teach you how to work with the most common controls and APIs used by Android developers around the world. https://store.raywenderlich.com/products/android-apprentice

Kotlin Apprentice

This is a book for complete beginners to the new, modern Kotlin language. Everything in the book takes place in a clean, modern development environment, which means you can focus on the core features of programming in the Kotlin language, without getting bogged down in the many details of building apps. This is a sister book to the Android Apprentice.https://store.raywenderlich.com/products/kotlin-apprentice