



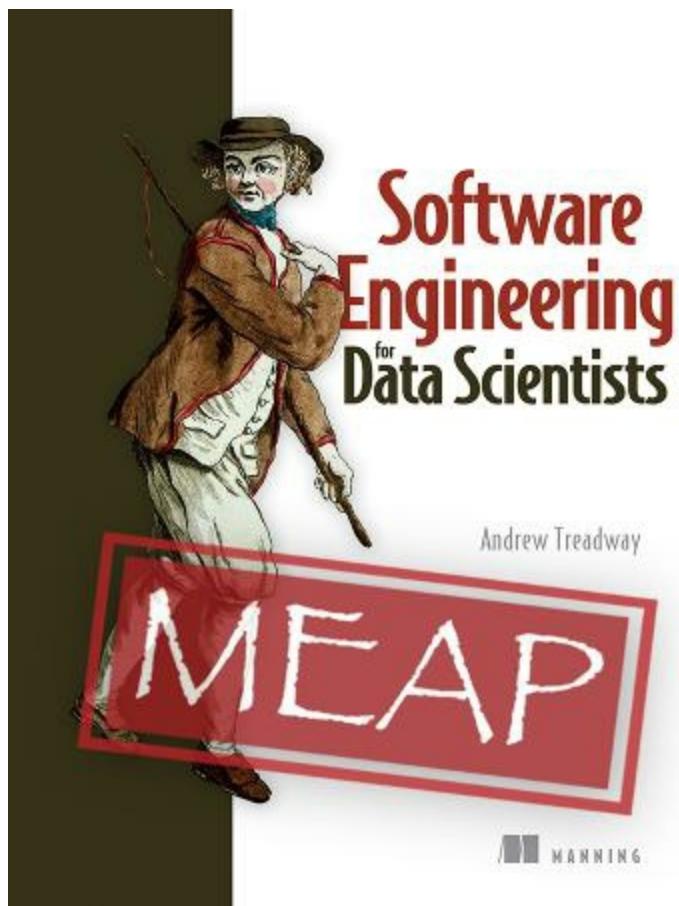
Software Engineering for **Data Scientists**

Andrew Treadway

MEAP

Software Engineering for Data Scientists MEAP V03

1. [MEAP VERSION 3](#)
2. [Welcome](#)
3. [1 Introducing engineering principles](#)
4. [2 Source control for data scientists](#)
5. [3 How to write robust code](#)
6. [4 Object-oriented programming for data scientists](#)
7. [5 Creating progress bars and time-outs in Python](#)
8. [6 Making your code faster and more efficient](#)
9. [7 Memory management with Python](#)
10. [8 Alternatives to Pandas](#)
11. [9 Putting your code into production](#)



MEAP VERSION 3

 MANNING PUBLICATIONS

Welcome

Thanks for purchasing the MEAP for *Software Engineering for Data Scientists*. This book is written for readers looking to learn how to apply software engineering concepts to data science.

The book is split into four parts:

- Part 1 – Getting started
 - This part will cover topics such as source control, exception handling, better structuring your code, object-oriented programming (OOP) for data science, and monitoring the progress of your code (such as model training or data extraction)
- Part 2 – Scaling
 - Part 2 covers scaling your code effectively. For example – how do you deal with larger datasets? We'll cover both the computational and memory components of scaling
- Part 3 – Scheduling, testing, and deployment into production
 - Part 3 details how to rigorously test your code, protecting your credentials (for example when connecting to a database to query data, scheduling models and data pipelines to run automatically, and packaging data analytics code into a portable library that can be shared with and downloaded by others)
- Part 4 – Monitoring your data processing and modeling code
 - Lastly, Part 4 will teach you how to effectively monitor your code in production. This is especially relevant when you deploy a machine learning model to make predictions on a recurring or automated basis. We'll cover logging, automated reporting, and how to build dashboards with Python.

In addition to the direct topics we cover in the book, you'll also get hands-on experience with the code examples. The code examples in the book are meant to be runnable on your own with downloadable datasets, and you'll find corresponding files available in the Github repository. Besides the examples laid out in the book, you'll also find *Practice on your own* sections at the end of most chapters so that you can delve further into the material in a practical way.

The book covers an extensive set of topics, and I hope you find it helpful in your technical journey. If you have any questions, comments, or suggestions, please share them in Manning's [liveBook Discussion forum](#)

— Andrew Treadway

In this book

[MEAP VERSION 3](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents 1](#)
[Introducing engineering principles 2](#) [Source control for data scientists 3](#) [How to write robust code 4](#) [Object-oriented programming for data scientists 5](#)
[Creating progress bars and time-outs in Python 6](#) [Making your code faster and more efficient 7](#) [Memory management with Python 8](#) [Alternatives to Pandas 9](#) [Putting your code into production](#)

1 Introducing engineering principles

This chapter covers

- What data scientists need to know about software engineering
- Why data pipelines are important
- How machine learning (ML) pipelines are used
- Putting models into production

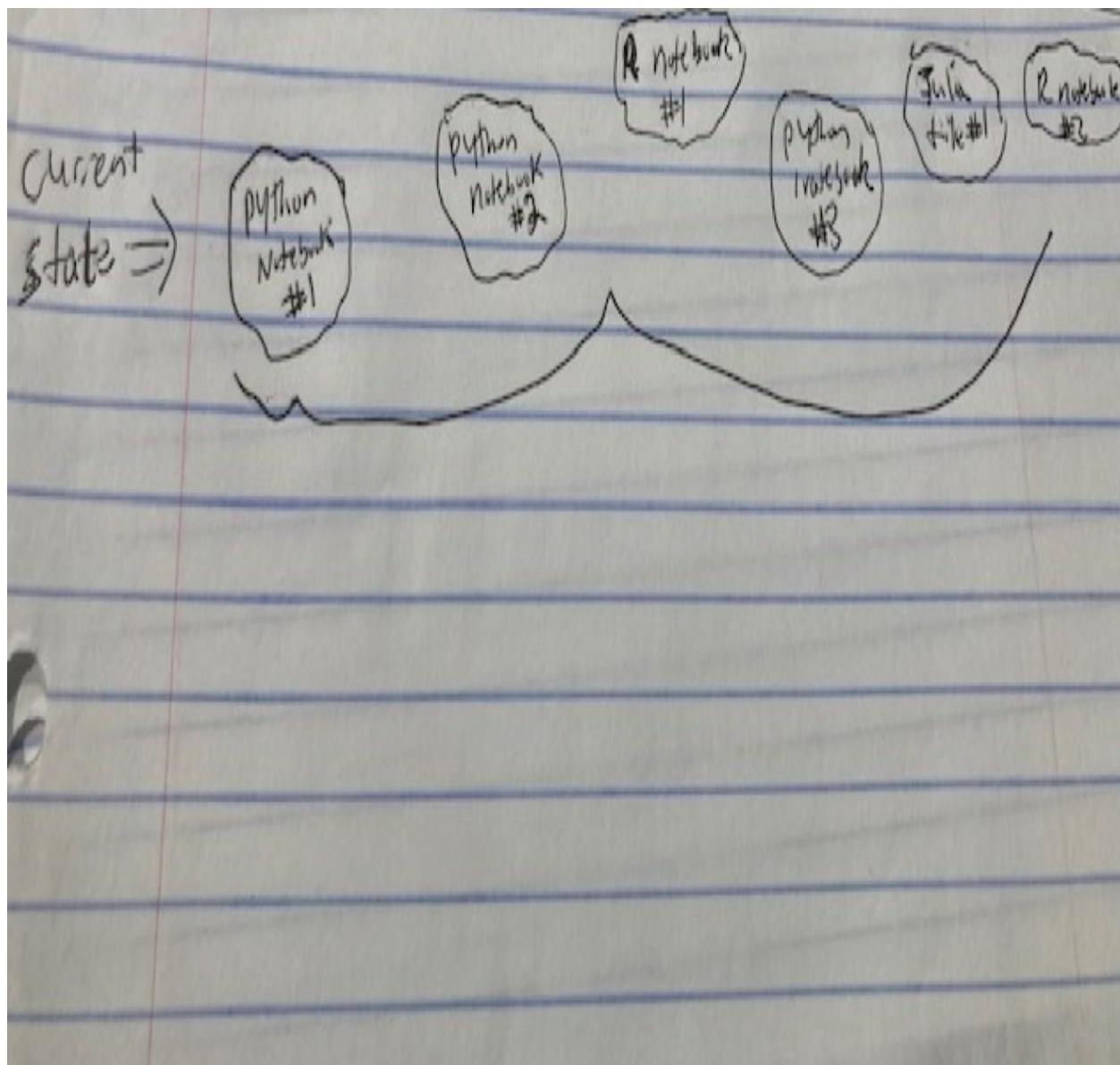
Suppose you're working on a project with several others (could be data scientists, software engineers, etc.). How do you handle modifying the same code files? What about testing out new features or modeling techniques? What's the best way to track these experiments or to revert changes? Often times, data scientists will use tools like Jupyter Notebook (a software tool that for writing code and viewing its results in a single integrated environment). Because Jupyter Notebook allows for easy viewing of code results (such as showing charts or other visuals, for example), it's a popular tool for data scientists. However, working on these *notebook* files can often get messy quickly (you may hear the term *spaghetti code*). This is generally because part of being a data *scientist* is experimentation and exploration - trying out various ideas, creating visualizations, and searching for answers in data. Applying software engineering principles, such as reducing redundancies, making code more readable, or using object-oriented programming (a topic we'll introduce later), can vastly improve your workflow even if your direct involvement with software engineers is limited. These principles make your code easier to maintain and to understand (especially if you're looking at it some length of time after you've written it).

Additionally, being able to pass your code to someone else (like a software engineer, or even another data scientist) can be very important when it comes to getting your code or model to be used by others. Having messy code spread across Jupyter Notebook files, or perhaps scattered across several programming languages or tools makes transitioning a code base to someone else much more painful and frustrating. It can also cost more time and resources in order to re-write a data scientist's code into something that is

more readable, maintainable, and able to be put into production. Having a greater ability to think like a software engineer can greatly help a data scientist minimize these frustrations.

The diagram in Figure 1.1 shows an example of a codebase where a data scientist's code may be scattered across several notebooks, potentially in multiple languages. This lack of a cohesive structure makes it much more difficult to integrate the code (for example, a model) into another codebase. We'll revisit this diagram later in the chapter within an updated, improved codebase example.

Figure 1.1. For data scientists, the state of their codebase is often a scattered collection of files. These might be across multiple languages, like Python or R. The code within each file might also have little structure, forming what is commonly known as spaghetti code.



Next, let's delve into what data scientists need to know about software engineering.

1.1 What do data scientists need to know about software engineering?

Before we go further, it may be helpful to briefly define software engineering. Software engineering is the application of *engineering principles* to developing software applications. Just like a civil engineer helps

to ensure the reliability and effectiveness of bridges or other constructs, a software engineer develops applications that are reliable, scalable, and efficient. A few of the key principles of software engineering as applied to data science are below:

- Better-structured code to minimize errors (both in terms of bugs in the code, but also in terms of inputs into code functions, such as the features being fed into a model)
- Collaboration among co-workers is a key part of any data science team. Software engineering principles can and should be applied to make collaboration and working together on the same code base seamless and effective
- Scaling code to be able to process large datasets efficiently and effectively is also very important in modern data science.
- Putting models into production (as mentioned above)
- Effectively testing your code to reduce future issues

We'll cover each of these points in more detail in the next section. First, let's briefly discuss the intersection of data science and software engineering.

Josh Wills (a former director of data engineering at Slack) once said that to be a data scientist, you need to be better at statistics than a software engineer, and better at software engineering than a statistician. One thing is for certain - the skills that a data scientist is expected to have has grown much over the years. This greater skillset is needed as both technology and business needs have evolved. For example, there's no recommending posts or videos to users if there's no internet. Platforms, like Facebook, TikTok, Spotify, etc. also benefit from advanced technology and hardware available in modern times that allow them to process and train models on massive datasets in relatively short periods of time. Models like predicting customer churn or potential fraud are much more prevalent nowadays because more data being collected, and more companies looking to data scientists to provide solutions for these problems.

Additionally, if you're interviewing for a data scientist position nowadays, chances are you'll run into questions around programming, deploying models, and model monitoring. These are in addition to being tested on more traditional statistics and machine learning questions. This makes the

interview process more challenging, but also creates more opportunities for those knowledgeable in both data science and software engineering. Data scientists need to have a solid knowledge of several areas in software engineering in their day-to-day work. For example, one key area where software engineering comes into play is around *implementation*. The example models we've mentioned so far, like recommendation systems, customer churn prediction, or fraud models all need to be *implemented* in production in order to provide value. Otherwise, those models are just existing in a data scientist's code files, never making predictions on new data.

Before we delve more deeply in the engineering principles mentioned above, let's walk through a few more examples of common issues in a data scientist's work where software engineering can help!

1.2 When do we need software engineering principles?

Let's walk through a few real-life scenarios for a data scientist. These scenarios will highlight several key issues data scientists face in their work. The motivation for walking through these problems is to illustrate the usefulness of incorporating software engineering knowledge, which we'll further introduce as we go through this chapter.

Better stuctured code

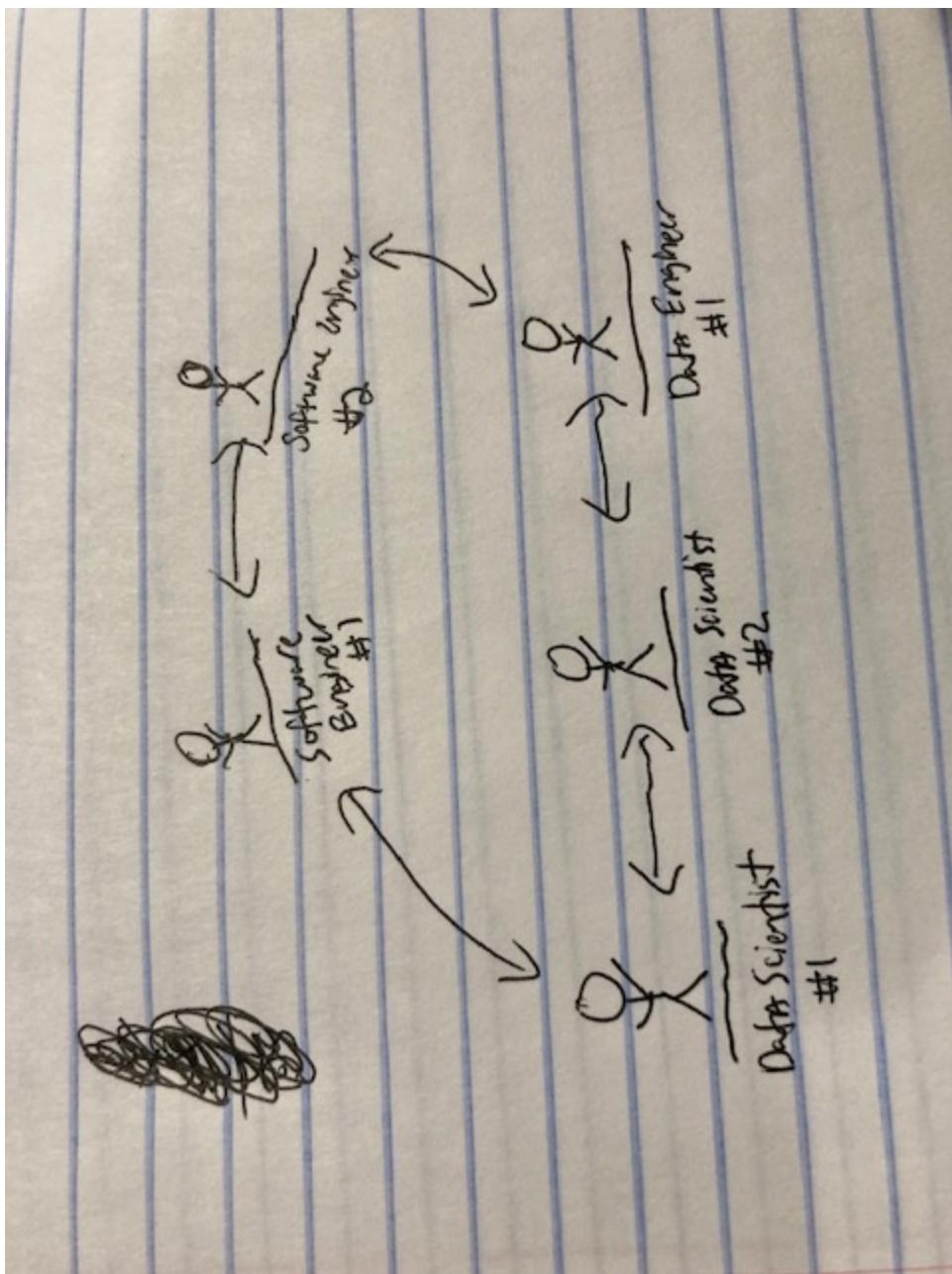
We covered this point already earlier in the chapter, so we'll just briefly rehash that improving the structure of your code can greatly help for several key reasons, including sharing your code with others, and integrating the code into other applications.

Improving coding collaboration

Extending on the earlier scenario, collaborating on the same code base with others is crucial across almost any data science organization or team. Applying software engineering principles through *source control* allows you to track code changes, revert to previous code file versions, and (importantly) allows for multiple people to easily change the same code files without threat

of losing someone else's changes. These benefits of source control can also be useful even if you're working alone on a project because it makes it much easier to keep track of the changes or experiments that you may have tried.

Figure 1.2. Collaboration is an important part of coding in many companies. Data scientists, data engineers, and software engineers are three common roles that often interact with each other, and share code with each other. Working effectively with a shared codebase across multiple (or many) users is a topic we'll delve into in the next chapter.



Scaling your code to handle more data efficiently

Another common scenario involves scaling. *Scaling* involves improving your code's ability to handle larger amounts of data from both a memory perspective, as well as an efficiency point of view. Scaling can come up in many different scenarios. For instance, even reading in a large file might take precious time when your compute resources are constrained. Data processing and cleaning, such as merging datasets together, transforming variables, etc. can also take up a lot of time and potentially memory. Luckily, there exists many techniques for handling these problems on a large scale, which we'll cover in more detail later in this book. It is quite common for data science code to be initially written inefficiently. Again, this is often because data scientists spend a lot of time exploring data and experimentally trying out new features, models, etc. By applying software engineering concepts and tools available, you can transform your code to run on a more robust basis, scaling to larger numbers of observations much more efficiently.

Putting models into production to make them usable by others

Once you've developed a model, you need to put it to use in order to provide value for your team or company. This is where putting your model into *production* comes into play, which as mentioned above, essentially means scheduling your model to make predictions on a recurring (or potentially real-time) basis. Putting code in production has long been a software engineering task. It's the heart of software engineering work at many different companies. For example, software engineers are heavily involved in creating and putting apps on your phone into production, which allows you to use the apps in the first place. The same principles behind doing this can also be applied to data science in order to put models into production, making them usable by others, operating anywhere from a small number of users to billions (like predicting credit card fraud), depending on the use case.

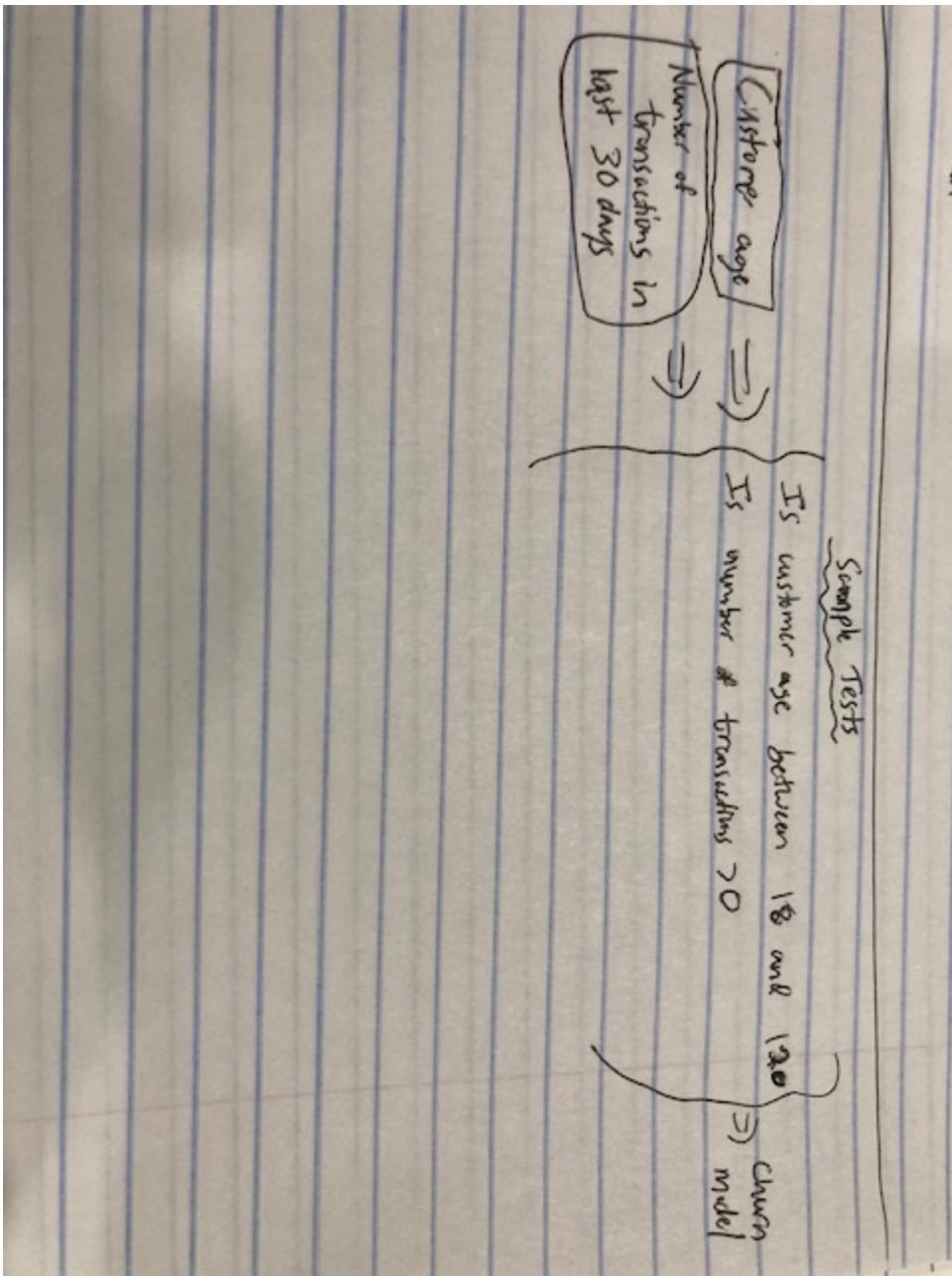
Effectively testing code to reduce future issues

So you've created a model and it's soon going to be making new predictions. Maybe it's a model to predict which customers are going to churn in the next month. Maybe it's predicting how much new insurance claims will cost. Whatever it is, how do you know the model will continue to perform

adequately? Even before that, how can you ensure the code base extracting, processing, and inputting data into the model doesn't fail at some point? Or how can you mitigate the results of the code base failing? Variations of these issues are constantly faced by software engineers with respect to code they're writing. For example, an engineer developing a new app for your phone is going to be concerned with making sure the app is rigorously tested to make sure it performs like it is supposed to. Similarly, by thinking as a software engineer, you can develop tests for your data science code to make sure it runs effectively and is able to handle potential errors.

In Figure 1.3, we show an example of inputting features into a customer churn prediction model. Here, we might add tests to ensure the inputs to the model, like customer age or number of transactions over last 30 days, are valid values within pre-defined ranges.

Figure 1.3. This snapshot shows a sample of tests that could be performed when deploying a customer churn model. For example, if the model has two inputs - customer age and number of transactions over the last 30 days, we could perform checks to make sure those inputs values are within set ranges prior to inputting into the model.



Let's summarize these principles in a table.

Engineering principle	Advantages
Better structured code	Makes code more easily integratable, easier to maintain, and helps improve coding collaboration.

Improving code collaboration	In addition to better structured code, we can use additional tools - such as source control - to make it easy to work together on the same codebase across many users or teams.
Scaling your code	Making your code robust enough to handle large volumes of data and generalizable enough to deal with new variations of inputs, data, or various errors that may occur.
Deploying models into production	Make the application of your code usable or accessible by others. This can be anything from a customer churn model making predictions to an app on your phone tracking your fitness goals.
Effective testing	Any model or application that will be handling data or being used by others needs to be rigorously tested to ensure it can handle potential issues.

Next, let's go through a sample data science workflow using a specific example. This will help to tie the above scenarios to a common data science use case.

1.2.1 Sample data science workflow

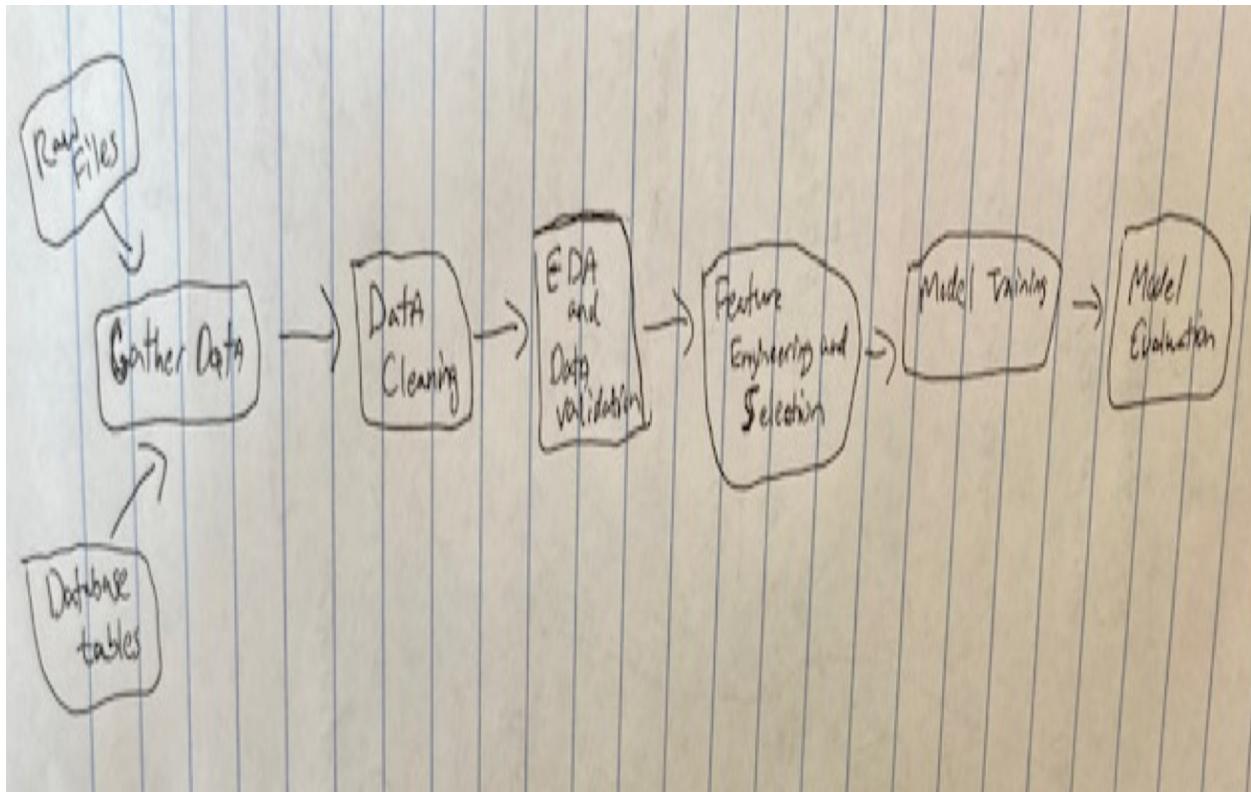
Suppose you're working for an e-commerce company and you want to predict whether a customer will churn in the next 30 days. Oftentimes, a data scientist working on a problem like this would roughly follow these steps:

- Gather data
 - Data can be collected in several ways:
 - Writing SQL to extract data from various tables / databases (MySQL, SQL Server, etc.).
 - Scraping data from documents (e.g. CSV, excel files, or even PDF / Word documents in some cases)
 - Extracting data from webpages or through web APIs
- Exploratory data analysis (EDA) / data validation
 - EDA involves steps like checking the distributions of key variables, investigating missing values, looking at correlation plots, and checking descriptive statistics (such as the median values for numeric variables or most common value for categorical features). Data validation might involve checking the results of EDA against domain knowledge or across multiple data sources to ensure that the data is accurate and reasonable to use for analysis and modeling.
- Data cleaning
 - Data cleaning involves minimizing the number of issues in a dataset, including the following:
 - Replacing missing values
 - Removing highly correlated variables
 - Treating highly skewed variables (potentially transforming certain features)
 - Dealing with an imbalanced dataset (think about predicting ad clicks, for example, where the vast majority of users never click on an ad)
- Feature engineering
 - Feature engineering is the process of developing new features from existing variables. This is generally done in an effort to improve

model performance. For example, certain machine learning models will perform better when the inputs follow a normal distribution, so there are existing techniques that make these transformations. In other instances, feature engineering is absolutely necessary to get anything useful out of a variable. This is typical of date variables, for instance. For example, in the credit fraud use case, transaction date could be a raw variable, but cannot be input directly into a machine learning model. Instead, we parse out new features, such as the day of the week, hour of the day, month, etc.

- Model training
 - Model training involves inputting data into machine learning algorithms like logistic regression, random forests, etc. so that the algorithm can learn the patterns in the data in order to be able to make predictions on new data. This process may involve testing out several different models, performing fine-tuning the parameters of the models, and perhaps selecting a subset of the more important features relevant to a model.
- Model evaluation
 - Model evaluation is a key component where data scientists need to check how well the model performs. This is usually done by evaluating the model on a fresh, or hold-out, dataset that was not used for model development. There are a variety of metrics that may be used to assess performance, such as accuracy (number of examples where the model was correct / total number of predictions) or correlation score (sometimes used to evaluate models outputting a continuous prediction).

Figure 1.4. A sample data science workflow, as described above. Data science workflows involve key steps like gathering data, exploring and cleaning the data, feature engineering, and model development. The model evaluation piece is also a highly important step, that we will cover in detail when we discuss model monitoring later in this book.



1.2.2 How does software engineering come into the picture?

The process above presents several potential problems from a software engineering perspective.

- How do we allow easier coding collaboration between data scientists, data engineers, software engineers, and whomever else may be working on the project?
 - While data scientists ultimately use data to perform analysis and modeling, data engineers are more heavily involved in creating new tables, managing databases, and developing workflows that bring data from some source (like raw logging on a website) to a more easily ingestible place where data scientists can query a table - or small number of tables - in order to get the data needed for modeling or analysis. Software engineers, as mentioned above, help to create reliable applications that might be used either by internal employees or external consumers. These three roles often work closely together, and their exact responsibilities may overlap

depending on different companies.

- How do we use the developed model to predict churn for customers on an ongoing basis?
 - We can think of this question as an extension of the last point in our data science workflow concerning model evaluation. Once we're satisfied with a model's performance, how do we go from a trained model to one that is making predictions on a regular cadence? This heavily involves one of the main topics of this book, which is putting models into production.
- How can we have fresh data ready for the model to use in production?
 - This question is partially related to the first piece of the data science workflow - gathering data. Essentially, ensuring fresh data is available in production involves automating the *gathering data* process and *hardening* it to reduce the possibility of errors or data issues.
- How do we handle invalid inputs into the model or other errors?
 - Handling invalid inputs into the model is often needed once the model is developed and ready to be deployed into production. In a production environment with new data coming in, we may need to create checks like making sure the data types of each feature input is correct (for example, no character inputs when a numeric value is expected) or that numeric values are in an expected range (such as avoiding negative values when a positive number is expected). Other types of errors may also occur in the workflow above when we are automating those steps for fresh incoming data. For instance, whatever code is being used to extract the new data may fail for some reason (such as the server hosting a database going down), so you could develop logic to retry running the code after a few minutes.
- How can we effectively monitor the model's performance once it's making predictions on a recurring basis?

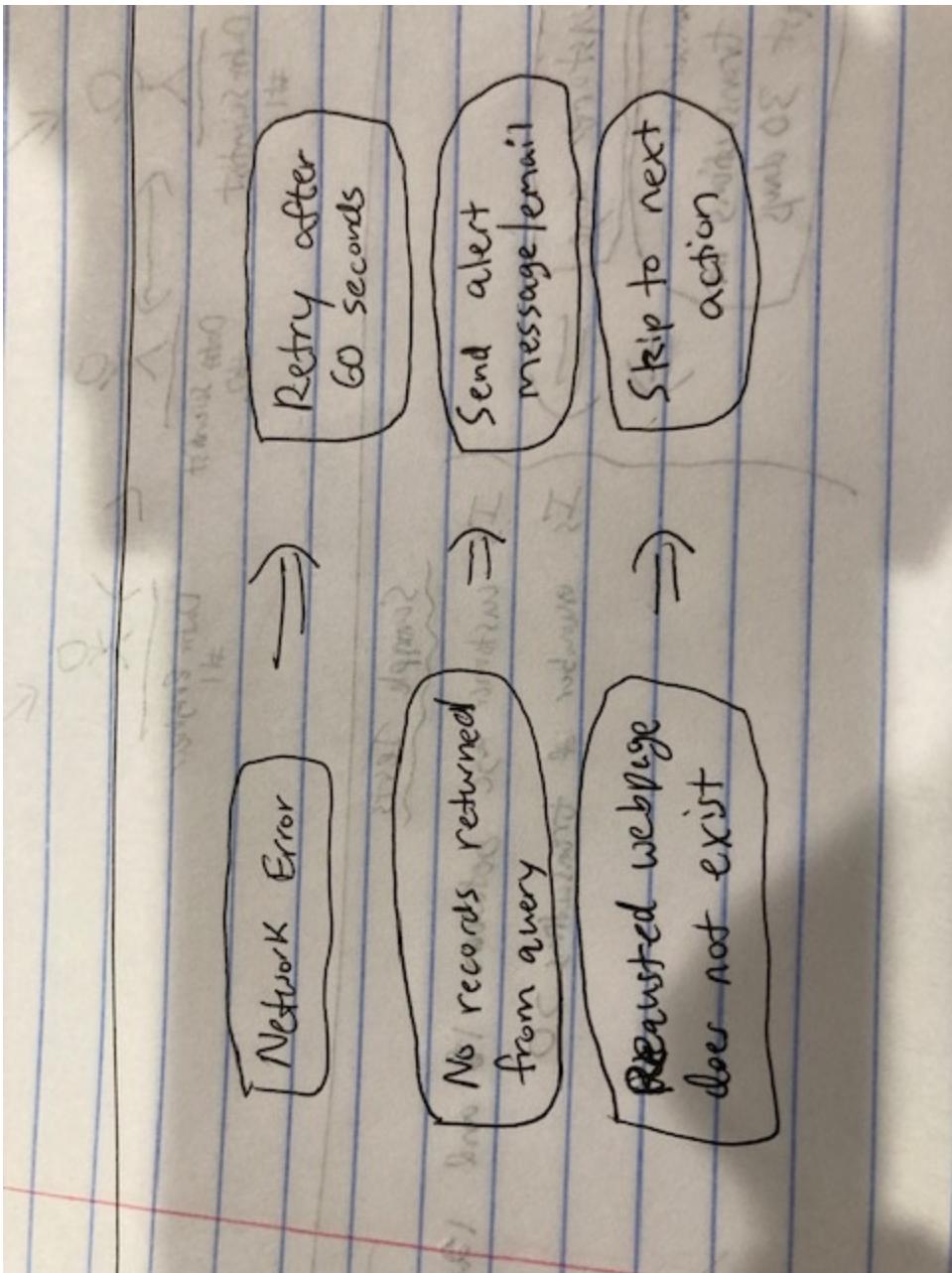
- Model monitoring can be thought of as an extension of the model evaluation step in our data science workflow above. Effective monitoring of a model is necessary to ensure confidence in a model’s performance over time. The exact way we monitor a model will depend upon what the model is actually doing. In our example above, we would know after 30 days whether a customer actually churned. We could use this to then chart the accuracy of the model over time, along with other metrics like precision and recall (which may actually be more important depending on the balance of the dataset). These metrics can be tracked using a dashboard. Additionally, we might monitor information like the distributions of the features in the model. This can help to debug potential issues, such as dips in model performance. We will discuss Python tools for introductory web development and dashboarding later in this book.
- Do we need to re-train the model on an ongoing schedule?
 - Re-training a model on a recurring cadence is tied to the performance of the model over time. The short response to this question is...it depends. If the model performance is dropping a week after you’ve trained it, then you might need to re-train the model frequently (or consider different features). If the model performance is stable over time, you might not need to re-train the model very often, though it’s a must to monitor the model to ensure it is meeting the standards expected.
- How can we scale the model to millions of users?
 - Scaling can be a fairly in-depth topic, but we can broadly think of it in terms of memory and efficiency. Many machine learning applications can be extremely intensive in terms of both CPU (or GPU) cycles, as well as memory. Enabling ML and data workflows to handle larger-scale data is an important task, and one that is likely to grow in importance as datasets get larger and more diverse. Scaling code is heavily a software engineering topic. When applied to data science, it can involve areas like parallelizing code or using advanced data structures.

We will dive into more detailed approaches to each of these concerns in later chapters, but for now let's introduce a few summary points. Broadly speaking, software engineering provides solutions to these issues. Software engineering helps to fortify our modeling code to reduce errors and increase reliability. It can integrate a model trained in a data scientist's development environment into a robust application making predictions on millions of observations.

There are a few software engineering concepts we can apply to make our data science workflow more robust and to handle these issues.

- Source control
 - *Source control* (sometimes called *version control*) refers to a set of practices to manage and monitor changes to a collection of code files.
- Exception handling
 - *Exception handling* involves developing logic to handle errors or cases where a piece of code may fail. For instance, this would come up in the example mentioned earlier where a section of code retrieving data from a database might fail and exception handling logic could be implemented to retry retrieving the data after several minutes. A few examples of exception handling can be seen in Figure 1.5.

Figure 1.5. Exception handling can take many forms. A few examples are shown in this figure. For example, we may need to query data on a regular basis for model training (for instance, updating the customer churn model). What happens if the query returns zero rows one day? There could be multiple solutions, but one could be sending an alert/email to an oncall data scientist (or engineer) about the problem. Or what if you're scraping data from a collection of webpages and a request fails due to a non-existing webpage? We might want to skip over the webpage without ending the program in error. We'll delve into exception handling more fully in Chapter Three.



- Putting a model into production
 - As mentioned earlier, putting a model in *production* involves enabling the model to make predictions on an automated basis, perhaps on a recurring schedule or in real-time.
- Object-oriented programming (OOP)
 - *Object-oriented programming (OOP)* is a type of coding paradigm

that revolves around *objects* which have corresponding functions and features known as *attributes*. To take a real-life analogy, consider a car as an *object*. A car has many attributes like a make and model, color, number of doors, etc. It also has associated functions, such as *starting*, *stopping*, *driving*, etc.

- Automated testing
 - *Automated testing* refers to rigorously checking each component of code that will need to run in production to ensure there are no issues - all in as automated a way as possible. This might involve validating a model's predictions on test inputs, for example.
- Scale
 - *Scale* generally refers to data size and is usually thought of in terms of memory or efficiency. Larger scale problems might have millions (or billions) of observations and thousands of columns.

These software engineering concepts are not utilized in a vacuum. Any ML model or data science workflow (even basic analysis) involves data. Where do we get the data from? How do we make sure the data source and any aggregations, data merging, or pre-processing done prior to being ingested by a model is reliable? The key idea is to construct a data pipeline, which we'll introduce shortly. Another key concept when it comes to deploying machine learning models is a machine learning pipeline, which we'll discuss after data pipelines. Before we dive into data pipelines, however, let's summarize what we've learned in this section.

- Software engineering involves applying *engineering principles* to software in order to make it more scalable, reliable, and efficient. These same principles can also be applied to make data science applications have improved reliability, able to scale to larger datasets, and more efficient.
- These principles include well-structured code, which helps make debugging and collaboration much easier. They also involve dealing with larger and larger datasets in a world where data is continually

growing. Depending on your data science application, engineering principles can also be applied to put your code into production (such as a model that can run on millions - even billions - of observations).

- A typical data science workflow involves several steps, including gathering data, data cleaning, and model training / evaluation.
- There are several key components of software engineering that can enhance the data science workflow. These include exception handling, source control, object-oriented programming (OOP), and scale (among others).

Now, let's dive into data pipelines!

1.3 What are the components of a data pipeline?

A data pipeline is a set of actions to process data. Processing data means any action taken to a collection of data such as the following:

- Extracting the data from some source (database, webpage, set of documents, etc.)
- Merging and aggregating features (i.e. columns) of data from different sources (e.g. customer-level vs. transaction-level data)
- Logging a model's predictions to a new table

In the workplace, data pipelines are often created by data engineers. However, depending on a company's structure, data scientists may also write data pipelines. In general, if you're a data scientist, it's recommended to know about data pipelines and understand the source of the data you're using for your modeling or analysis projects. This is important for several reasons, but especially because it helps you to build confidence in knowing where the data you're using is coming from. Software engineers can also be involved in data pipelines. For example, at tech companies, it is common for software engineers to write the code that logs data from a particular web application (a simple example would be logging whether someone clicks on an ad).

Data pipelines are necessary because they ensure that data is served reliably for a variety of data science applications. Even if you're working on an insight analysis, rather than a model, for example, you need to be able to

retrieve reliable data. Data pipelines are used in these cases to bring data from between different tables and sources. They can also be used for logging data, such as storing predictions from a model.

Sometimes you may see data pipelines referred in the context of reporting and analytics, as well, in addition to being used for machine learning models. The main summary point to keep in mind is that data pipelines are ultimately used to flow data from a source (or collection of sources) into finalized outputs, usually in the form of structured tables with collections of rows and columns. The exact application of ultimately using the data can vary. Let's give a real-world example of a data pipeline.

1.3.1 Real-world example: Building a model to predict customer churn

Taking the example mentioned earlier, suppose you're building a model to predict whether an e-commerce customer will churn in the next 30 days. Sample inputs into the model might include:

- Total amount of transactions over last 3 months
- Number of times customer logged onto the e-commerce website
- Length of time since customer first registered
- Number of times customer has deactivated account in last 12 months

The data points above may be derived from several different tables. Let's list those out below:

Table 1.1 Transaction table

Customer ID	Transaction ID	Transaction amount	Number of items purchased
123	999	100	4

Table 1.2 Login record table

Customer ID	Login timestamp	Landing page
123	2022-04-27 08:30:57	example.ecommerce.com

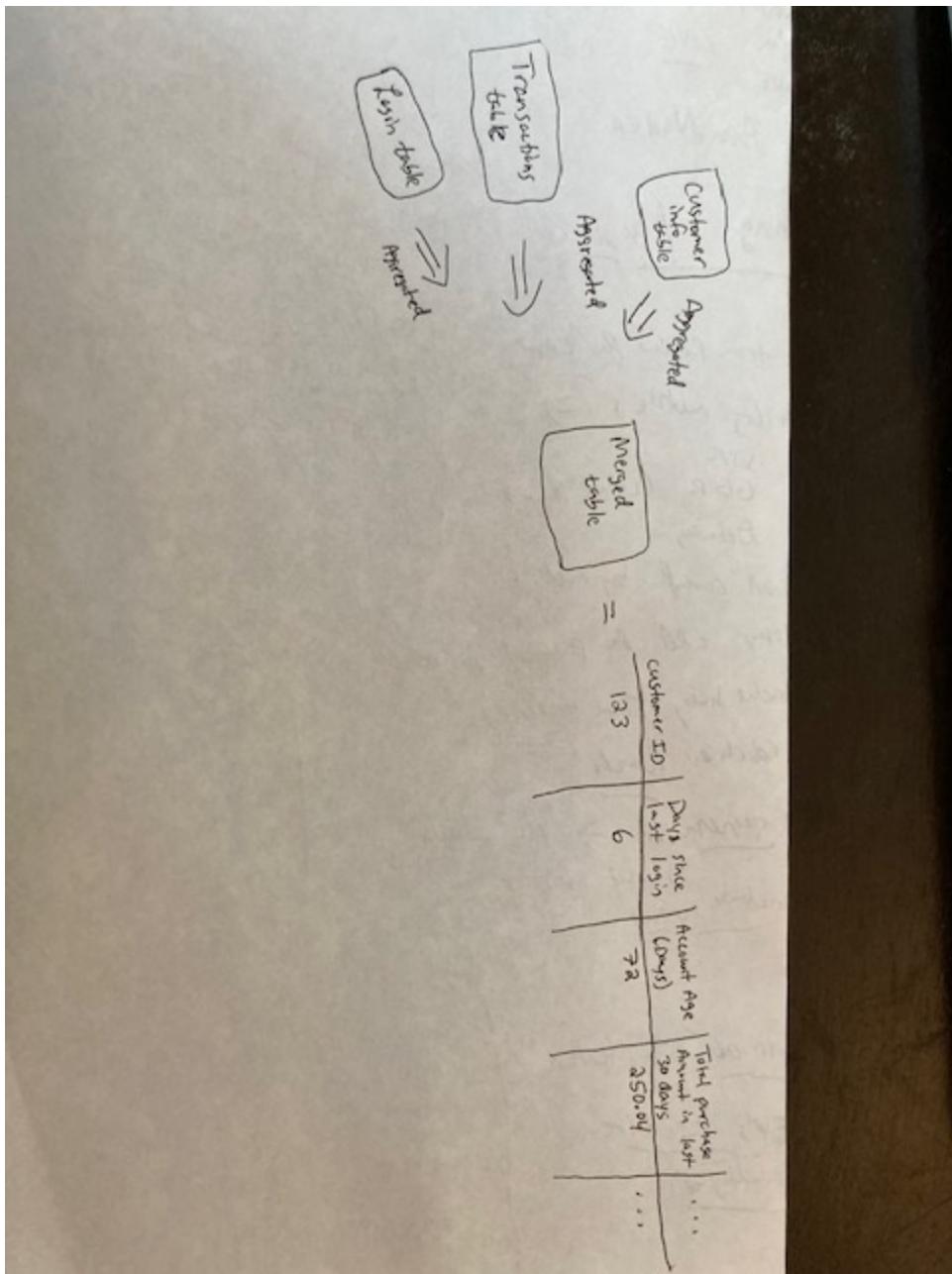
Table 1.3 Customer information table (e.g. dim_customers)

Customer ID	Initial registration timestamp	Age	Deactivation timestamp
123	2022-02-20 15:30:00	34	NULL

In order to train a model in the first place, we need to have a method of combining and aggregating the data inputs we need from the above tables. This process might involve multiple sources of data and in some cases, may involve using various languages or frameworks. A common language used in extracting data from tabular sources is SQL. SQL, or *Structured Query Language*, is basically a programming language designed to extract data from databases.

The final dataset may include additional pre-processing and feature engineering prior to the actual model development. Depending on the setup, some of these components may be handled in the data pipeline or in the modeling code component, which we'll cover next. For instance, suppose that a person's age is being used as a feature in predicting customer churn. Rather than inputting age directly into a model, we might want to apply a transformation to age, like bucketing it into different groups (for instance, under 18, 18 - 24, 25 - 30, 31 - 40, etc.). This bucketing could be handled directly in a data pipeline by creating a table that has a column (*bucketed_age*) with those categories, or it could be handled after the data (including the age variable) has been passed to the model, with some pre-processing code to create that bucketed feature prior to inputting it into a model.

Figure 1.6. An example flow of data from several source tables to a final output dataset used for predicting customer churn.



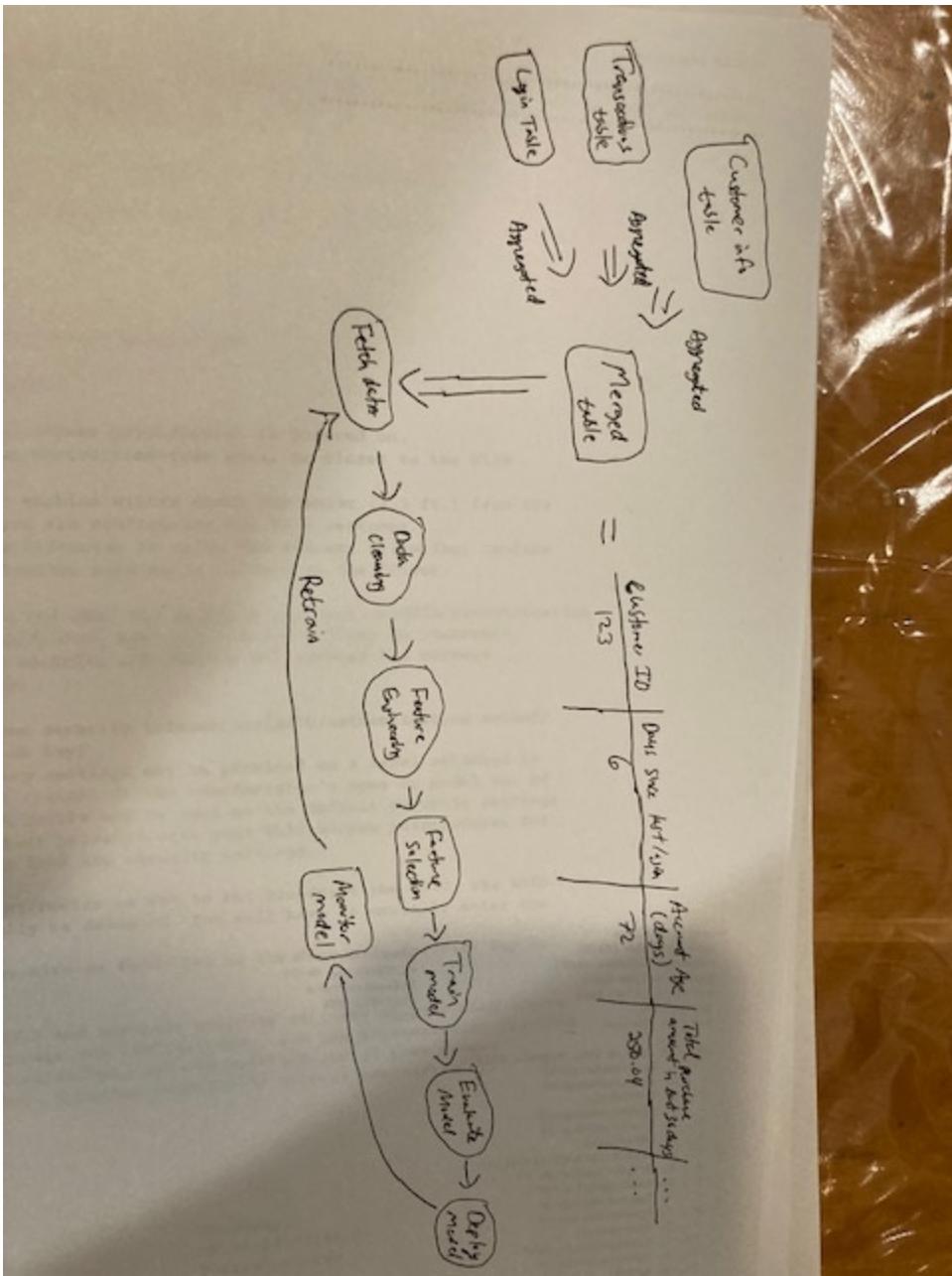
Now that we've covered the flow of data from several potential sources to a finalized dataset, what happens to the data? This is where machine learning pipelines come into the picture, as we will discuss next.

1.4 Deploying models with machine learning

pipelines

A machine learning pipeline is essentially a setup for automating and hardening a workflow to deploy a machine learning (ML) model. In this context, an ML pipeline can be thought of as an extension of (or encompassing) a data pipeline. A data pipeline is necessary to have a scheduled and fortified (minimizing errors) workflow for providing the data for an ML model. The rest of the ML pipeline feeds the data into an algorithm (think random forest, logistic regression, etc.). We can break the model structure component into several sub-components, which we break out below. Before we dive into the details, let's take a look at an overview of an ML pipeline, as you can see in the below diagram. Here, we show how an ML pipeline is connected to a data pipeline. All ML pipelines start with and depend upon data. This data typically comes as the result of a data pipeline. Additionally, the data must undergo processing, such as creating new features for a machine learning model, selecting which features to use in a model, evaluation of a model's performance, and more.

Figure 1.7. A sample workflow of a machine learning pipeline extending from a data pipeline. This diagram shows an outline of a typical machine learning pipeline. Keep in mind that the starting point is data. Data is always needed for any machine learning pipeline. The final part is monitoring the model to validate its performance on an ongoing basis. These and the in-between components are covered in detail below.



Now, let's dive into the details of the ML pipeline components displayed above.

1.4.1 Data ingestion

Data ingestion involves ingesting the data needed for the machine learning model. This data can be ingested from the last step of a corresponding data pipeline, which outputs a table with the columns needed for the model. There

are a variety of ways the data ingestion component can be setup, including fetching features from a *key-value store* (think of a key-value store has a fast lookup table to map a key - such as a customer ID - to a set of features for that customer, such as age or account history information), writing SQL to pull the needed data, and many more, which will be further discussed later in this book. Once data is ingested, there may be additional processing that needs to be done prior to using the data for model training. We'll explain what this processing involves next.

1.4.2 Pre-processing

Pre-processing, broadly speaking, involves performing any additional cleaning or feature engineering prior to model training. For example, the cleaning component might involve replacing missing values or capping outliers. As stated earlier, feature engineering could be implemented in the data pipeline by creating the new features directly into table columns, which are ingested by a model. Alternatively, feature engineering could be done as part of the pre-processing step after the data has already been extracted. When the pre-processing step is complete, the next component involves training a model on the data.

1.4.3 Model training

There are two main types of model training when it comes to ML pipelines, regardless of the type of model you're using.

- Train a model once, and use the static model for fetching predictions on an ongoing basis.
 - This type of model structure is useful in cases where the data does not change very rapidly. For instance, building a model to predict insurance claim cost from the time a claim is filed might be an example where the underlying data doesn't change from week to week (the relationship of the variables in the model vs. actual claim cost could be relatively static for shorter durations), but rather it might only need to be updated every 6-12 months.
- Re-train the model on a periodic basis, such as monthly, weekly, or even

more frequently if needed. This setup is useful when the performance of the model drops by a high-enough margin over shorter time periods to warrant updating it on an ongoing basis. This is related to the topic of model monitoring, which we'll cover in a later chapter. An example of this might be a recommendation system, for instance, where new and more varied types of content are frequently being made available, and the relationships between the inputs and the target that is being predicted changes fairly rapidly, requiring more frequent re-training.

After a model is trained, we need to evaluate the performance of the model. The main reason for this is that we need to be confident that the model is performing adequately for our standards, which we'll discuss next.

1.4.4 Model evaluation

Model evaluation comes after a model has been trained on a dataset. There may be a variety of evaluation metrics and breakdowns to help ensure confidence in the model performance. The evaluation of the model will often be monitored on an ongoing basis. For instance, in the customer churn example, we should be able find out in the available data whether a customer actually churned i.e. whether our model prediction was correct. This ground truth could be compared to our model prediction to calculate metrics like precision or recall. In this case, *precision* would be the proportion of cases where the model predicted a customer would churn and that the model was successful in that prediction. *Recall* would be the proportion of customers that actually churned which were successfully captured by the model. The model monitoring component may consist of dashboards and potentially a notification system to alert if the performance of the model drops by a significant amount. Model evaluation is highly important to any business use case because if a model is not performing adequately, then it could end up costing a company money, wasting resources, or having a negative impact on customers relying on the model predictions (such as credit card users depending on accurate predictions preventing malicious actors from secretly using their credit).

Once we are confident in a model's performance, it is time for putting the model into production order to make predictions on an ongoing basis.

1.4.5 Model prediction / deployment

On the model prediction side, there are also two main model structures (again, this is regardless of the underlying model being used).

- **Real-time models**

- Real-time models return predictions in real-time. Predicting whether a credit card transaction is fraudulent is a high-impact example where a real-time model can be critical. These predictions are often fetched via an API call, which we'll discuss in more detail later in this book.

- **Offline models**

- Offline models make predictions on a *scheduled* basis, such as daily, weekly, hourly, etc. Our example of predicting customer churn could potentially be an offline model, running daily.

After the model is deployed and able to make predictions on new data, it is important to monitor the model's performance on an ongoing basis. This allows us to be alerted if there are any issues that arise in terms of the model's performance or changes in the data being used for the model.

Before we go into model monitoring, let's summarize a few examples based on real-time vs. offline prediction and recurring training vs. single (or infrequent) model training.

Model prediction	Training frequency	Prediction type
Customer churn	Evaluate based on need, but potentially monthly, or every several months	Offline - update model prediction based on data ingested each day
	Patterns related to ad clicks can change very	

Ad click prediction	fast, so training might be done on a weekly, daily, or even real-time basis	Real-time
Credit card fraud prediction	The factors used to identify fraud may change relatively rapidly, so training frequency should account for this	Real-time
Property insurance claim cost	Potentially every few months unless there are major changes that will impact the model's predictive power	Offline - update predictions with new information daily
Music genre prediction (for example, Spotify)	Depending on how often new genres are added, this could be every few weeks/months or on more frequent basis	Could be real-time or offline

Now, let's discuss model monitoring.

1.4.6 Model monitoring

As just mentioned, it is important to monitor a model in order to ensure its performance is adequate. This was briefly discussed earlier in the chapter, and to re-iterate, usually involves tracking model performance metrics and potentially feature distributions.

Before we close out the chapter, let's revisit Figure 1. Since we now know

about data pipelines and ML pipelines, we can think about restructuring the scattered notebooks/files shown in Figure 1 into a more sequential collection of files. Ideally, these would be standardized across languages as much as possible (for example, keeping code files only in Python rather than across Python, R, Julia, etc.). For example, this might look like the following table:

Sample file name	Description
1_fetch_dataset.py	Python script to extract data needed for model training
2_clean_dataset.py	A Python file to clean the raw dataset
3_feature_engineering.py	Perform feature engineering, such as calculating number of transactions over last 30 days
4_feature_selection.py	Reduce the features to only those needed
5_train_model.py	Train the model and store the model object
6_generate_evaluation_metrics.py	Generate a collection of evaluation metrics to measure the model's performance

This tables show a few sample code files corresponding to different steps in the ML pipeline. There could be additional code files as well. For example - feature engineering might be divided into several components in different files. We'll go further into the possible structure of files in the data science

workflow when we get to Chapter Three. Now, let's close out this chapter with a summary of what we've covered.

1.5 Summary

In this chapter, we discussed data pipelines, ML pipelines, and provided an overview of putting models into production.

- Data scientists need software engineering concepts in order to make code easier to maintain, allow for more seamless collaboration, implement models into production, scaling, and rigorously testing your code.
- Data pipelines are used for merging, aggregating, and extracting data from some underlying source into a final output (typically a table or collection of tables).
- Machine learning (ML) pipelines are used to deploy machine learning models to make predictions on an automated basis. This might be making predictions in real-time or recurring on a schedule, such as daily or weekly predictions.
- Key software engineering concepts that are used in fortifying data pipelines and ML pipelines include source control, object-oriented programming, scale, and exception handling.
- Putting a model in production requires fortification of code like handling errors / exceptions, scaling as necessary, and structuring code to be more readable and allow for easier collaboration

2 Source control for data scientists

This chapter covers

- What is source control
- Why do data scientists need to know source control
- What is git
- What are the main git commands you need to know as a data scientist

In the last chapter, we introduced several key software engineering concepts that will improve your life as a data scientist. These included:

- Source control
- Exception handling
- Putting a model into production
- Object-oriented programming (OOP)
- Automated testing
- Scale

In this chapter, we're going to delve deeper into the first of these - namely, *source control*. Source control (also called *version control*) is basically a way of tracking changes to a codebase. As the number and size of codebases has grown indescribably over the years, the need for monitoring code changes, and making it easier for various developers to collaborate is absolutely crucial. Because software engineering has existed longer than modern data science, source control has been a software engineering concept longer than a data science one. However, as we'll demonstrate in this chapter, source control is an important tool to learn for any data scientist. Before we delve into using source control for data science, however, let's discuss how source control fits into the picture of applying software engineering to data science. Recall in the last chapter, we discussed several key concepts of software engineering, such as better structured code, object-oriented programming, exception handling, etc.

Source control can (and should) be used from the very beginning and

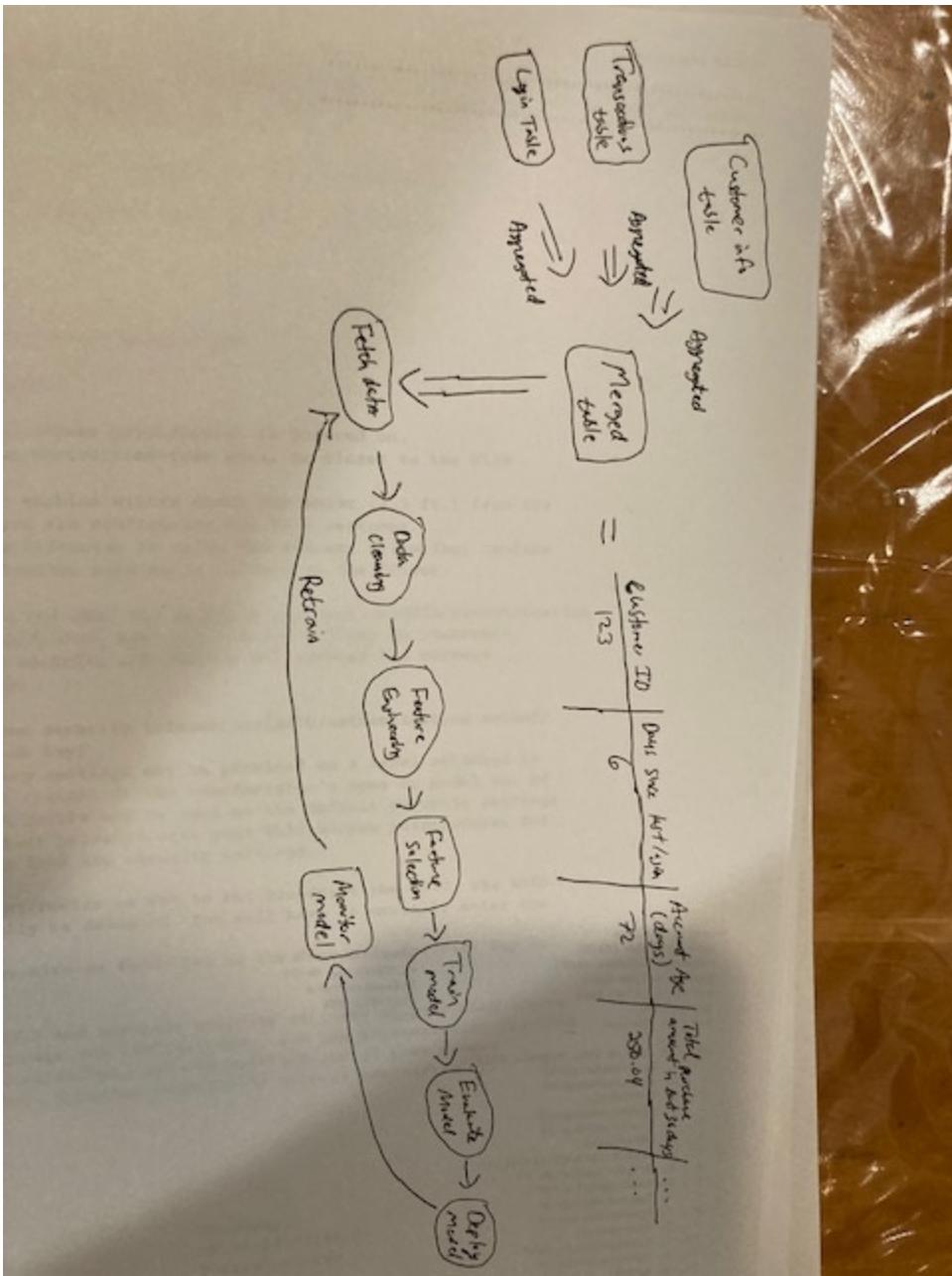
throughout a project. That project could be a purely software engineering application, a data science project, or some combination. For example, the project could be developing a new machine learning model, like our example in the last chapter around predicting customer churn. It could also be a purely engineering project, like code to create a new app for your phone. There is a consistent theme between source control, better structured code, and object-oriented programming (explained in a later chapter) in that each of these software engineering concepts make collaboration between developers (or *data scientists*) much easier. This chapter will focus specifically on common software for using source control.

Next, let's explore how source control will help you in your projects. Going back to the customer churn example from the first chapter, suppose you and a colleague are working on a data science project together to predict whether a customer will churn. To make this concrete, we will use the customer churn dataset available from Kaggle here:

<https://www.kaggle.com/competitions/customer-churn-prediction-2020/data>.

This dataset involves predicting whether a customer from a Telecom company will churn. The workflow of this project can be structured similarly to the data science life cycle that we discussed in the first chapter. As a refresher, we've repeated the chapter one diagram showing the combined data pipeline/ML pipeline view in Figure 2.1.

Figure 2.1. Adding to what we covered in the previous chapter, we can use source code at almost any step of of a data pipeline or ML pipeline. This is especially useful when multiple coworkers are collaborating together on the same codebase, but can also be helpful for tracking changes in a codebase even if you are the only one working on it.



In the following steps, we tie the pipeline components to our specific Kaggle dataset:

- Gathering data
 - Fetch the data needed to build the customer churn model. In the Kaggle dataset we'll be using, this includes information like length of account (account age), number of day calls, area code, etc.

- Exploratory data analysis (EDA) / data validation
 - Analyzing the data for patterns, distributions, correlations, missing values, etc. For example, what's the proportion of churn to non-churn? What's the association with total day minutes used (total_day_minutes) and churn? Etc.
- Data cleaning
 - Handle issues, such as missing values, outliers, dirty data, etc.
- Feature engineering
 - Create new features for the models you both will build. For example, you could create a new feature based on the average total day charge for the state the user resides in.
- Model training
 - Develop models, such as logistic regression or random forest to predict churn.
- Model evaluation.
 - Evaluate the performance of the models. Since churn vs. non-churn is a classification problem, we might use metrics like precision or recall here.
- Model deployment
 - Make the finalized model accessible to others. For this case, that might mean prioritizing individuals predicted to churn for marketing or lower price incentives.

Each of these steps might involve collaboration between you and your colleague. Consider a few example scenarios:

- When gathering data, you might write code to extract data from the last year. Your colleague, wishing to get more data, modifies the codebase to

extract data from the last three years.

- Your colleague adds code that creates several visualizations of the data, like the distribution of daytime calls. You find a bug in your colleague's code and want to correct it, so you modify the shared codebase.
- The two of you are working on cleaning the data. One colleague adds code to replace missing values and handle outliers. The other one realizes that one of the fields has a mix of numeric and string values, and writes code to clean this issue. Both sets of changes go into the same file.
- One of you works on developing a logistic regression model, while the other wants to try out a random forest. But you both want to share code with each other and potentially make modifications (like use different parameters or features)

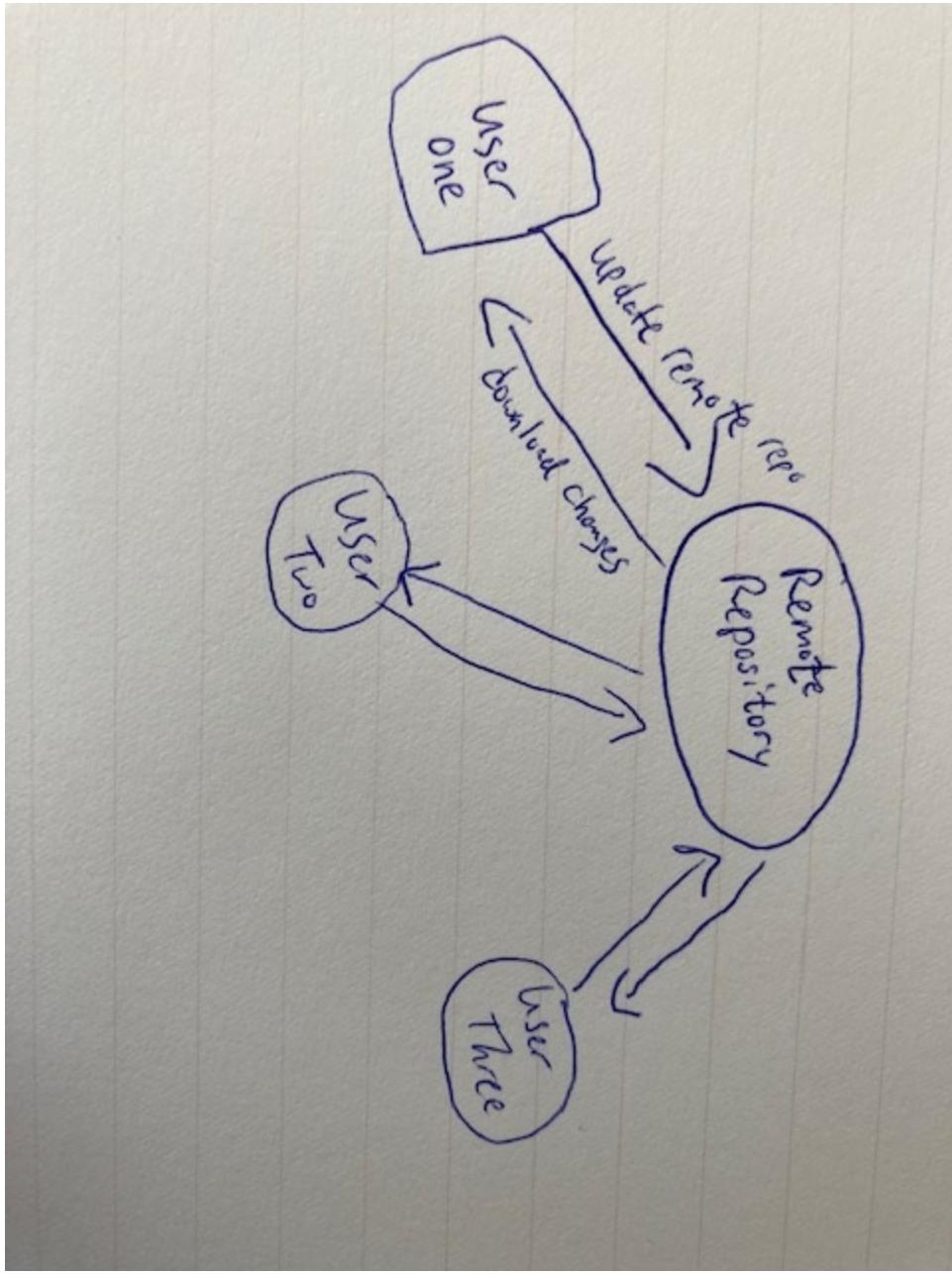
Source control makes these types of collaborations much easier and trackable. Next, let's dive into how source control would help you and your colleague in this situation, or in general for any data science project you may work on.

2.1 What is source control?

Source control, as mentioned above, is a system for tracking code changes across a collection of users (though that collection of users could also just be a single individual). Source control has long been used by software engineers, though as we'll see below is also invaluable for data scientists, as well. When speaking about source control, you'll often encounter the term *repository* (or *repo* for short). A repository is simply a collection of files, often hosted on a remote server. Typically, the source control workflow involves multiple people contributing to the same repository. Each person will download a copy of the repository's codebase to a local directory. By *local*, we mean whatever server or personal environment someone is using to create and write new code files. You may hear the term *local repository* to refer your copy of the collection of files. A developer can then make whatever changes needed to the codebase, such as creating new files (like a new Python script, for example) or modifying existing files. After changes are made, the updates can be pushed to the repository so that other users can download the most up-to-date code. Additionally, any changes pushed to the remote repository can

be tracked, so that you can trace who made particular changes.

Figure 2.2. Source control enables easier coding collaboration between different developers of the same codebase, often stored on a shared remote repository



An alternative to using source control would be to simply use a shared network directory where different developers could add or modify files to a centralized location. However, this has several key problems:

- How do we track changes based by different users? In other words, how can I easily tell who made what change? With the methodology above, this is very difficult.
- Difficult to revert changes. This can be especially important in cases where a new change causes an issue, like an app to crash, for instance.
- Easy to overwrite others' changes.
- *Merging* changes from different users working on the same code file is a challenge
 - Taking our customer churn dataset example, suppose you write code that creates a collection of new features for the churn model. Your colleague wants to modify the same code file(s) that you created. This process is called *merging*. Let's look at an example below.

Merging example:

Data scientist (DS) #1 changes

Listing 2.1. DS #1 creates a new Python file

```
import pandas as pd #1

p50_account_length_by_state = train.\ #2
    groupby("state").\ #2
    median()["account_length"] #2
```

Data scientist (DS) #2 changes

Listing 2.2. DS #2 modifies the same file DS #1 created.

```
import pandas as pd #1

p50_account_length_by_state = train.groupby("state").\ #2
    median()["account_length"] #2

p50_day_minutes_by_state = train.\ #3
    groupby("state").\ #3
    median()["total_day_minutes"] #3
```

In the first code snippet, DS #1 creates a code file that calculates median account length values by state. Another data scientist (DS #2) changes the code to calculate median total daytime minutes by state (adding an extra line of code). DS #2 can now merge the changes made into a shared repository where DS #1 can download DS #1's contributions. Now, let's move into how source control helps with the above-mentioned problems.

Broadly speaking, source control offers many benefits, including:

- Tracking who changed what. Source control makes it easy to track who created or modified any file in a repository.
- Undoing changes to the repository is straightforward. Depending on the specific software you're using for source control, this may even be as simple as executing one line of code to revert the changes.
- Provides a system for merging changes together from multiple users.
- Provides a backup for the codebase. This can be useful even if you're the only person contributing code. It's all too easy to accidentally overwrite a file or potentially lose work if a system crashes. Source control helps to mitigate these issues, in addition to delivering the benefits listed above.
- Code consistency. Too often, different members of a team may follow various styles or have different sets of functions, which may perform overlapping or similar actions. Source control allows for easier code-sharing among team members, enabling greater consistency in the codebase used across team members.

The benefits of using source control can be applied to any codebase, regardless of the application, environment, company, etc. Though source control has long been used by software engineers, it is also an important tool for data scientists to learn, as we'll explain below.

2.2 Why do data scientists need to know source control?

As data science projects scale in terms of the code base, as well as the number of people working together on the code, source control is crucial in keeping track of all the changes that are made. Almost any software engineer

should be familiar with using source control for projects. However, with data science, there can be a tendency to write *spaghetti code*. This phrase generally refers to code that is disorganized, contains redundancies, and is difficult to track changes or contributions from different people working on the code base. A core reason for this is that data science tends to be more experimental in the way code is written vs. pure software engineering. For example, when you're working on a project to develop a model, you might try out various combinations of features, different treatments of the data (like replacing missing values, capping outliers, etc.), or different types of models. Source control offers several benefits for this type of scenario:

- Easily track any experimental changes done by either an individual or a group of data scientists working on the same project
- Enables data scientists, software engineers, data engineers, etc. to modify the same code base in parallel, without fear of overwriting anyone's changes
- Allows for more easily packaging of the code base and hand-off to software engineers. This can be very important for putting models, data pipelines, etc. into production
- Makes it easier to merge changes to the same underlying files

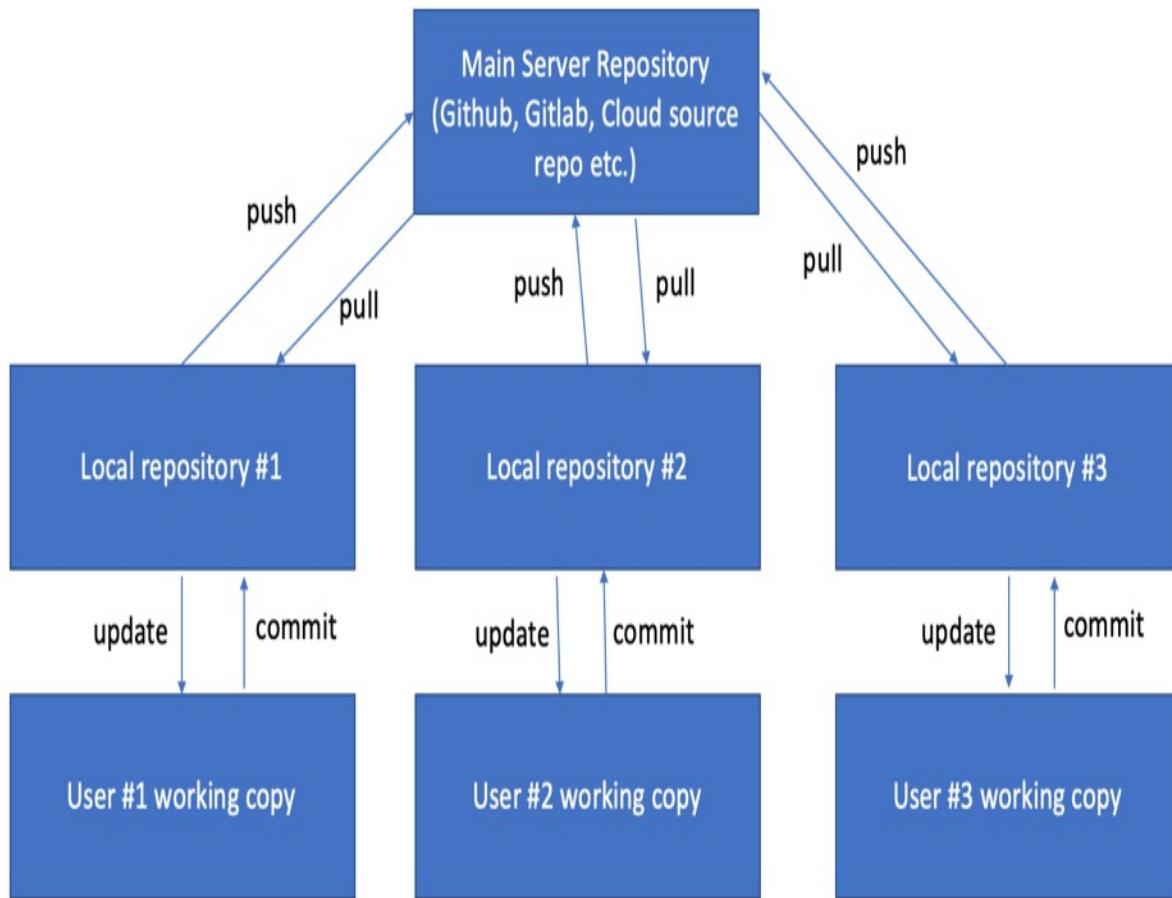
To make the concept of source control less abstract, let's use a concrete example of a version control software called *Git*.

2.3 Introducing git

Git is an open source version control tool, which is also freely available. *Git* is the software behind popular sites like *Github* and *Gitlab*. Because it is free and open source, in addition to being straightforward to use, *Git* is popular not only for software engineers, but is also used for many data science projects. We will be using *git* throughout this book for version control, so we will give an overview of how *git* works in the following sections.

Git can be used via the command line (terminal) or through a UI, like *Github*'s web UI (*SourceTree* and *GitKraken* are other UI's for *Git*). We can think of a typical *Git* workflow as shown in the below diagram.

Figure 2.3. This diagram is an extension of the one shown earlier. Developers push and pull code to and from, respectively, a remote repository. Each user must first commit his or her code to a local repository before pushing the code to the remote repo.



Next, let's get hands-on experience using Git so that this workflow will become more clear.

2.3.1 Basic git commands

Installing Git

If you're using a Mac, Git comes pre-installed. On Windows, you'll need to install Git. A common way to get Git setup on Windows is to go to Git's website (<https://git-scm.com/>) and download the installer for Windows.

Once you have Git installed, you can get started with it by opening a

terminal. Any Git command you use will be comprised of *git* followed by some keyword or other parameters. Our first example of this is using Git to download a remote repo for the first time.

Downloading a repository

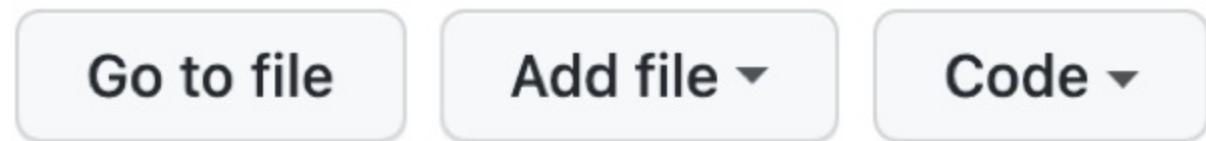
To download an existing remote repository, we can use the *git clone* command, like this:

Listing 2.3. Use the git clone command followed by the URL to the repo you want to download.

```
git clone [URL of repository] #1
```

In this command, we just need to write *git clone* followed by the URL of the repository we want to download. For example, suppose you want to download the popular Python **requests** library repository from Github. On Github, you can find the link you need to use by going to the repo's main page on Github.

Figure 2.4. Snapshot of Github repo page.



Clicking on *Code* should bring up a view like below, where you can see and copy the *HTTPS* URL (<https://github.com/psf/requests.git> in this case).

Figure 2.5. Snapshot of Github repo page.

 **Clone**



HTTPS **SSH** **GitHub CLI**

<https://github.com/kennethreitz-archive>



Use Git or checkout with SVN using the web URL.

 **Open with GitHub Desktop**

 **Download ZIP**

Next, you can download the contents of the repo like this:

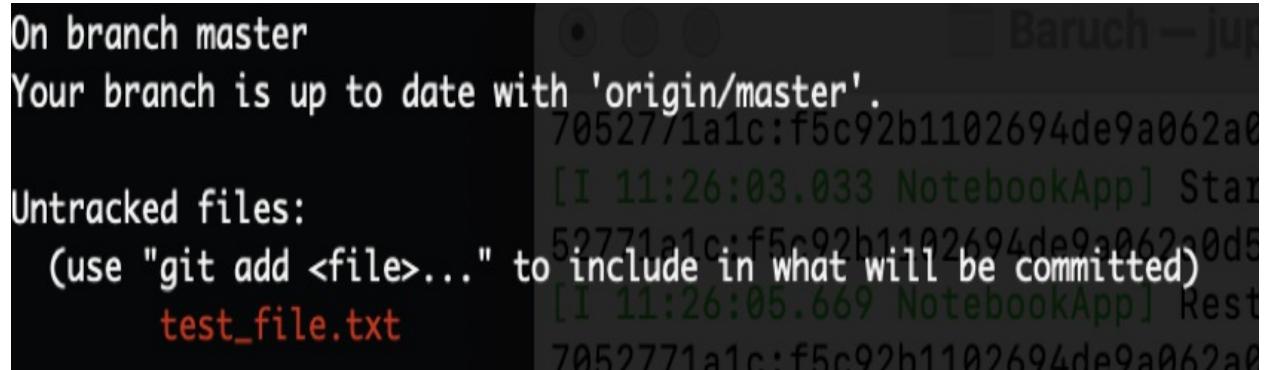
Listing 2.4. git clone command

```
git clone https://github.com/  
kennethreitz-archive/requests3.git #1
```

A benefit of using *git clone* is that it automatically links to the downloaded repo to being tracked by git for you. For example, running the above command will download a folder (in this case, called requests3) from the Github repository. But now, any changes you make in this folder will be automatically tracked. For example, let's suppose we create a new file within the downloaded directory called *test_file.txt*. Then, running *git status* in the

terminal shows the following message:

Figure 2.6. Running `ls -a` in the terminal will show the newly created hidden git files



The screenshot shows a terminal window with the following text:
On branch master
Your branch is up to date with 'origin/master'.
Untracked files:
(use "git add <file>..." to include in what will be committed)
 test_file.txt

From this message we can see that there is one file (the one we just created) that is currently untracked. Additionally, we can see our local repository is up-to-date with the remote repository on Github. This means no else has made any changes to the remote repository since we've downloaded the local repository.

git clone is a useful command when there's an already-existing remote repository that you're planning to modify. However, what if a remote repository doesn't exist? That's where *git init* comes in handy, which we'll discuss next.

Downloading the repo for this book!

Now that we walked through how to clone the repository for a sample repo, let's download the repo for this book!

Listing 2.5. Use the `git clone` command to download the repo for this book!

```
git clone https://github.com/atreadw1492/  
software_engineering_for_data_scientists.git #1
```

Now, you should have all the files from the book's repository in whatever directory you selected in your computer.

2.4 Git workflow from scratch

Creating a new repository

What if you want to version control a local collection of files? For instance, let's go through how we created the book repository in the first place. To start with, we will use the *git init* command. If you enter *git init* in the terminal, a new local repository will be created in the current working directory. As an example, if you're currently in the /this/is/an/example/folder, simply running *git init* will create a local repository in /this/is/an/example/folder.

Listing 2.6. *git init* is a command to create a new local repository

```
git init #1
```

To create a repository in another folder, you just need to specify the name of the that directory, like in the example below, where we create a repository in /some/other/folder.

Listing 2.7. Use *git init [directory name]* to create a new repository in another folder

```
git init /some/other/folder #1
```

When you use the *git init* command, git will create several hidden files in the input directory. If you're using a Mac or Linux, you can see this by running *ls -a* in the terminal. Similarly, you can use the same command in Bash on Windows, as well.

Figure 2.7. Running *ls -a* in the terminal will show the newly created hidden git files



git init can be run in a directory either before or after you've created files that you want to backup via version control. Let's suppose you've already setup a collection of sub-directories and files corresponding to what you can see in this book's remote repository.

- ch2/...
- ch3/...
- ch4/...

- Etc.

Next, we need to tell git to track the files we've created. We can do that easily enough by running `git add .`. The period at the end of this line tells git to track *all* of the files within the directory.

Listing 2.8. Use `git add .` to tell Git to track all files that were changed

```
git add .
```

Now, we can *commit* our changes. This means that we want to save a snapshot of our changes. We do this by writing `git commit` followed by a message that is associated with the commit. To specify the message, you need to type the parameter `-m` followed by the message in quotes.

Listing 2.9. `git commit` saves the changes you've made as a snapshot of the repo. The parameter `-m` is used to include a message / description for the commit.

```
git commit -m "upload initial set of files" #1
```

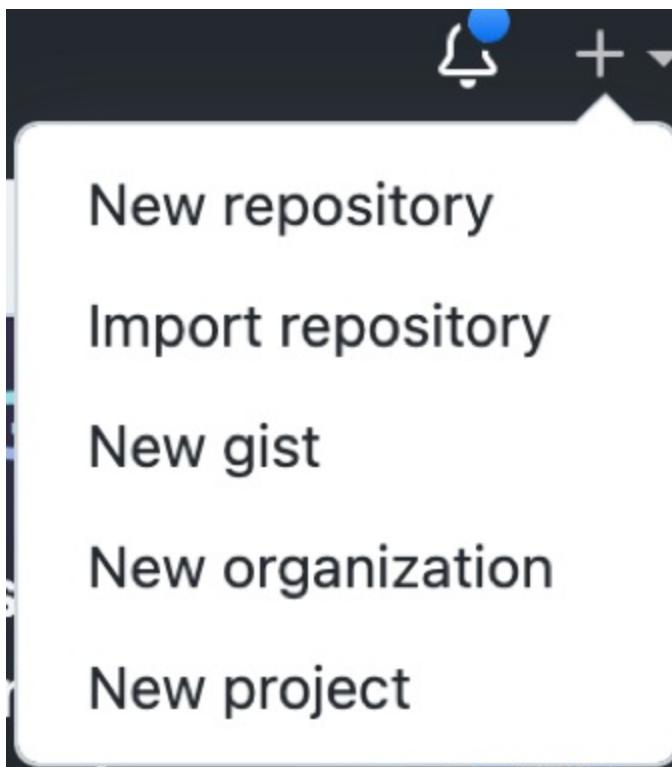
Now that we've committed our changes, we need to create a remote repository in order to upload our committed changes. This remote repository will be where other users can download or view your changes.

2.4.1 Uploading local repository changes to a remote repository

There's several sites for hosting remote repositories, but a popular one (as mentioned above) is Github. To create a remote repo on Github, you should be able to follow these steps:

- Create a Github account if you don't already have one
- On most Github pages, you should see a plus sign that you can click to create a new repo

Figure 2.8. To create a new repo on Github, look for the plus sign (top right corner of the webpage) to click and create a new repo.



- Give a name to your new repository. It's recommended that this name matches the name of the folder you're storing the local copies of the files you're dealing with. For example, if *sample_data_science_project* is the name of your local folder, then you could also name your repository *sample_data_science_project*. For the book's codebase, our repository name is *software_engineering_for_data_scientists*.
- Next, you're ready to push the local repository to the remote one you just created.

To actually push your local repository to the one on Github, you can run a command similar to this:

Listing 2.10. Use `git remote add origin [remote repo URL]` to enable pushing your changes to a remote repository.

```
git remote add origin #1
https://github.com/USERNAME/ #1
sample_data_science_project.git #1
```

USERNAME will be replaced with your username, while *sample_data_science_project* will be replaced with whatever name you

choose for the repository (like *software_engineering_for_data_scientists*, for example).

Next, run the line below to tell git that you want to use the main branch. Usually, the main branch is called *master* or *main*. This branch should be considered the *source of truth*. In other words, there might be other branches that deal with experimental code (for instance, *feature_x* branch), but the main branch should use code that has closer to being production-ready, or at least passed off to software engineers for production. In some cases, if the codebase is small, or there are only a few contributors, you might decide to just use a single main branch. However, as codebases get larger, and more people get involved in contributing code to the repository, then creating separate branches can help keep the main branch clean from messy *spaghetti* code that frequently changes.

Listing 2.11. git branch -M main tells Git that you want to use the main branch when you push or pull changes.

```
git branch -M main #1
```

Lastly, you can run *git push -u origin main* to push your local repo changes to the remote repository. After running this line, you should be able to see the changes in the remote repository.

Listing 2.12. git push -u origin main will push your local changes to the remote repo.

```
git push -u origin main #1
```

As you and your potential colleagues make changes to the same repo, it is convenient to be able to easily tell who made which changes. Fortunately, there's a straightforward way to do this, which we'll cover next.

Modifying a Git repo

Let's suppose now that we want to add a new file to our working directory called **1_process_data.py**. This file could be a Python script that reads in data from a database and performs basic processing / cleaning of the data. As a naming convention, we might add "1_" to the front of the script name in order to convey that this is the first script in a collection of potential files that

needs to be run. For now, though, let's suppose we've just created this **1_process_data.py** file. In order to backup our new file via git's source control system, we'll need to *commit* the file (more on this in just a moment). First, however, let's run *git status*. This command, as described above, is a simple way of checking what files have been modified or created, but are not currently being tracked in git's version control system.

Listing 2.13. Use *git status* to check the status of the current directory, which will show any files that have been created or modified, but not yet committed.

```
git status #1
```

Figure 2.9. Running *git status* shows what files, if any, are not currently being tracked via git

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    1_process_data.py
nothing added to commit but untracked files present (use "git add" to track)
```

Next, let's tell Git to track our new file, **1_process_data.py**. Again, we can do that easily enough by running *git add .* to add all changed/new files for tracking.

Listing 2.14. Use *git add [file_name]* to tell git to track a specific file. In this case, we will track our new file, **1_process_data.py**.

```
git add 1_process_data.py #1
```

You can also add multiple files by running the *git add* command separately. For example, after running the command above, you could run *git add 2_generate_features.py* and Git will prepare *2_generate_features.py* to be

committed. This sample file is available in the Git repo, along with **1_process_data.py**, so that you can practice on your own.

Now, we can *commit* our new file. The **-m** parameter adds a message in the commit providing a short description.

Listing 2.15. git commit saves the changes you've made as a snapshot of the repo. The parameter -m is used to include a message / description for the commit.

```
git commit -m "create 1_process_data.py  
for processing data" #1
```

Lastly, we can push the changes to our remote repo.

Listing 2.16. git push -u origin main will push your local changes to the remote repo.

```
git push -u origin main #1
```

Next, let's take a look at how we can see who made commits.

2.4.2 How to see who made commits

We stated previously that Git helps with tracking who made specific changes. To actually see who made specific changes, you can run the *git log* command.

Listing 2.17. git log will print out what commits have been made in the repo.

```
git log #1
```

Figure 2.10. Running git log will show the commit history for the repo

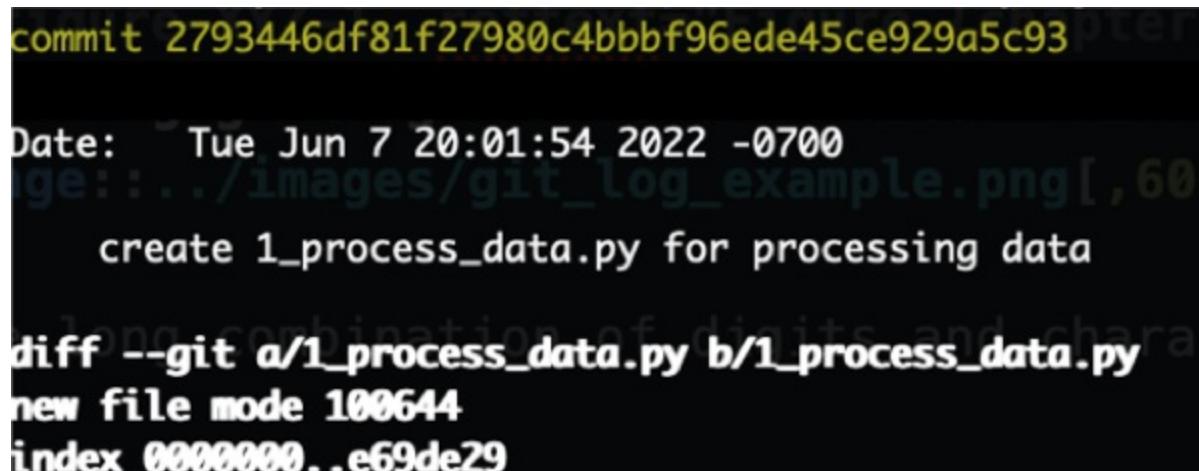
```
commit 2793446df81f27980c4bbbbf96ede45ce929a5c93  
Date: Tue Jun 7 20:01:54 2022 -0700  
create 1_process_data.py for processing data
```

The long combination of digits and characters after *commit* is the commit hash ID. It's a unique identifier for a specific commit. You can also see the contents of the commit (the actual code change) by using *git show*, followed by the a hash commit ID.

Listing 2.18. Use *git show* to see the contents of a commit

```
git show 2793446df81f27980c4bbb96ede45ce929a5c93 #1
```

Figure 2.11. Running *git show* will show the contents of a specific commit, as can be seen in this example snapshot.



A screenshot of a terminal window displaying the output of the command `git show 2793446df81f27980c4bbb96ede45ce929a5c93 #1`. The output shows a single commit with a long hash ID. The commit message is "create 1_process_data.py for processing data". Below the commit message, there is a diff section showing the creation of a new file named `1_process_data.py` with mode `100644`. The index line is `index 000000..e69de29`.

```
commit 2793446df81f27980c4bbb96ede45ce929a5c93
Date: Tue Jun 7 20:01:54 2022 -0700
  create 1_process_data.py for processing data

diff --git a/1_process_data.py b/1_process_data.py
new file mode 100644
index 000000..e69de29
```

In this case, *git show* doesn't display any code because our sample file we created was empty. But, let's suppose we have created and committed another file for feature creation. If we run *git show* for this other commit, we might get something like the snapshot below showing lines of code that have been added to a file.

Figure 2.12. This time *git show* displays the code changes to a specific file

```
commit 488f4cd290102d1b74ea2d1d3c3667588f091f1a (HEAD -> main, origin/main)
Date: Sat Jun 11 09:43:09 2022 -0700
t show 2793446df81f27980c4bbbbf96ede45ce929a5c93
    update feature creation script

diff --git a/2_create_features.py b/2_create_features.py
index e69de29..99238ae 100644
--- a/2_create_features.py
+++ b/2_create_features.py
@@ -0,0 +1,5 @@
+import pandas as pd
+import matplotlib.pyplot as plt
+from IPython.display import Image
+customer_data = pd.read_csv("customer_data.csv")
+
+
```

In addition to committing your own changes, it is important to be able to get the latest updates from the remote repository. Let's dive into how to get the latest changes from the remote repository next!

2.5 Getting the latest changes from a remote repository

As you collaborate on your data science project, it's important to keep up to date with the latest code changes. For example, a member of your team might add a new file for cleaning a collection of features. To get the latest changes, you can use the *git pull* command. *git pull* will fetch the remote changes and automatically try to merge them with your local repository. If there are no conflicts between the remote repo and your own, Git will merge the remote repository's updates into your local repo. However, if there are conflicts, they will need to be handled separately, as we'll discuss in the next section.

Listing 2.19. Use the *git pull* command to get the latest changes from a repo

```
git pull origin #1
```

In the above command, *origin* refers the remote repository (you can think of it as the *original* repo). Alternatively, you can just run *git pull*, which will pull the changes from the same remote repository by default.

Listing 2.20. We can also omit the "origin" snippet in our previous git pull command.

```
git pull #1
```

Next, let's show how to deal with conflicts that can arise when merging.

2.6 Conflicts and merging changes from different users

Let's go back to our example of two data scientists (you are your colleague) working on the customer churn project. What happens if you and your colleague make changes to the exact same line of code in the same file? This can lead to a conflict. Let's walk through an example.

Conflict example

Suppose the `1_process_data.py` file mentioned previously contains a simple Python function, like this:

Listing 2.21. Sample Python function to demonstrate what happens when two users update the same line of code

```
def read_data(file_name): #1
    df = pd.read_csv(file_name) #1
    return df #1
```

Let's say that you change the name of the function to *read_file*. However, your colleague changed the name to *read_info* in a local repo and then pushed those changes to the remote repository that both of you are working with. In this case, running *git pull* will result in an error message that looks like this:

Listing 2.22. Run git diff to show the conflict differences between your repo vs. the remote one.

```
Automatic merge failed; fix conflicts #1
and then commit the result. #1
```

Listing 2.23. Run git diff to show the conflict differences between your repo vs. the remote one.

```
diff --cc 3_sample_file.py #1
index 48eb46a,43e4e41..0000000 #1
--- a/3_sample_file.py #1
+++ b/3_sample_file.py #1
@@@ -1,4 -1,4 +1,8 @@@ #1
+<<<<<< HEAD #1
+def read_data(file_name): #1
+===== #1
+ def read_info(file_name): #1
+>>>>>> #1
c4fa0bff6539ee4182f2dc296a47fe1de51fbac2 #1
```

In this case, there's two main options.

1. Keep the remote changes
2. Keep your local changes

We describe how to handle these options next.

Keeping remote changes

To keep the remote changes, you can simply change the line of code causing the conflict, then commit your changes. In this case, that means changing the function name to match what is in the remote repo: *read_info*. Then, you can simply commit your changes like before, and run *git pull* again.

Alternatively, you can run a shortcut command if you want to keep all the changes from the remote repo that conflict with your own. To do this, you can use the *git checkout* command, like below. The *--theirs* parameter tells Git you want to keep the remote changes. Secondly, we specify the name of the file with conflicted changes that we want to maintain as the remote changes.

Listing 2.24. Use git checkout --theirs to keep remote changes

```
git checkout --theirs 1_process_data.py #1
```

As mentioned above, keeping remote changes is one option of dealing with conflicts. Another option is to keep the local changes, which we cover next.

Keeping local changes

Similar to keeping all remote changes for a file, you can run the below command to keep the local changes. Here, we just need to put in the name of the file containing the changes we want to keep (in this case, *1_process_data.py*).

Listing 2.25. Use git checkout --ours to keep your local changes

```
git checkout --ours 1_process_data.py #1
```

Now, let's talk about *branching*. Suppose that you want to try out a rules-based model for predicting customer churn, but your colleague wants to test out a random forest approach. This can be a useful scenario to use *branching*. Think about *branches* as different snapshots of the same codebase that can be worked on in parallel. Branching is usually used when you want to test out changes to a codebase without pushing them to the main (*source of truth*) branch. In our example, you might create a new branch to work on the rules-based model, while your colleague uses a different branch to test out a random forest approach. Ultimately, the two of you might decide to go with the random forest approach. In this case, you can use *git merge* (discussed in the next section) to merge your random forest branch into the main branch.

Now, let's talk more about branches and how to work with them using Git.

2.7 How to work with branches in Git

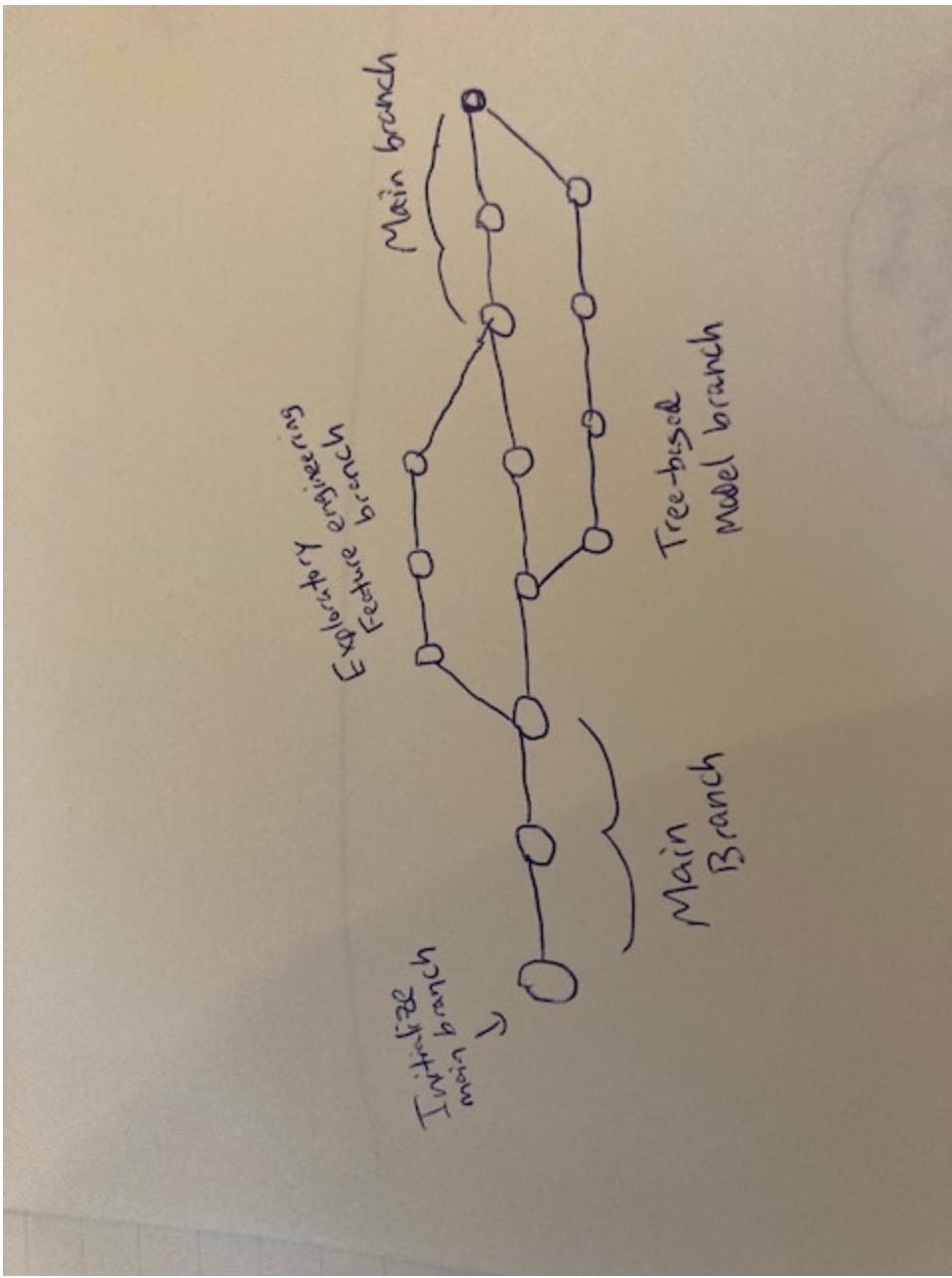
As mentioned earlier, a *branch* is essentially a snapshot of a codebase. A repository might have multiple snapshots (branches). There's several ways to think about using branches.

- Using the main branch as the final source of truth. Whenever you're confident in the changes you want to make and persist, push them to the

main branch. Otherwise, you can create separate branches to experiment and test out code.

- Potentially split development vs. main. The development branch can act as the branch that is mostly used when you and your potential colleagues are working together on your data science project. For example, if you're developing a new model, you might have separate code files for ingesting data, performing data cleaning and processing, feature engineering, and separate scripts for testing out different models. Once a model is finalized, you can push the finalized codebase to the main branch, where you might work with software engineers to put your model into production.
- In other cases, using just a single main branch can work as well. This can be the case if only a few people are working together and decide that using a single branch works for their particular use case. It can also be the case that if you're working in an environment where only production-code is pushed to a repository, it might make sense to use just a single main branch.
- Ultimately, the decision on how to use branches is up you, the team you work with, or the company you work for

Figure 2.13. Branching can be thought of as having different snapshots of the same codebase. Each branch might have its own alternate set of changes. However, each branch can ultimately be merged back together, or back into the single main branch.



Now, let's cover a few Git commands related to branching.

2.7.1 Git commands for branches

There's several Git commands related to branching. Firstly, if you want to confirm what branch you're currently working with, you can use the *git branch* command. This command will print out all the branches in your local repository. An asterisk will be printed next to the branch that is currently

active (the branch you're currently working with).

Listing 2.26. git branch command

```
git branch #1
```

The result of running the command is below. In this case, the result shows we're on the main branch.

```
* main
```

To see all branches - including remote ones, you can add the -a parameter to the command above.

Listing 2.27. git branch -a command

```
git branch -a #1
```

Again, the result shows the main branch, along with the remote main branch.

```
* main  
remotes/origin/main
```

Creating a new branch

If we create a new branch, we'll be able to see that one in our list of branches, as well. To create a new branch, use the *git branch* command, followed by the name you want to give to the new branch.

Listing 2.28. git branch [new_branch_name] command

```
git branch [new_branch_name] #1
```

For example, to create a new branch to handle the rules based model (which can be called *rules_based_model*), you can type the below command into the terminal.

Listing 2.29. git branch allows you to create new branches by specifying the name of the branch you want to create (in this example, rules_based_model)

```
git branch rules_based_model #1
```

Switching to a new branch

You can switch to a different branch by entering *git checkout* followed by the name of the branch you want. Let's switch to the new branch we just created.

Listing 2.30. *git checkout* [different branch] will switch to a different branch so any commits you make will be made to this branch

```
git checkout rules_based_model #1
```

Running *git branch* again will now show two branches, with an asterisk next to *rules_based_model*, telling us that is the current branch we are working with.

```
main
* rules_based_model
```

Pushing changes to a new branch

Now that *rules_based_model* is checked out, any changes you commit will be reflected only to that this branch, and not the main branch (or any other branch for that matter).

Since we created a new branch locally, but the same branch doesn't exist in the remote repository, we need to add extra parameters to *git push* before we push any changes to the remote repo.

Let's suppose, for example, that we created a new file called **3_churn_rules_based_model.py**. This file can be found in the ch2 directory in the book's Github repo.

Listing 2.31. *git push --set-upstream origin rules_based_model* will push the changes made to the *rules_based_model* branch. When using *git push* on a new branch for the first time, we need to add the extra parameter *--set-upstream origin* so that the remote repository will now reflect the new branch.

```
git push --set-upstream origin rules_based_model #1
```

In the above command, we're essentially telling Git to create the new branch

in the remote repository and to push any changes that were made from the local branch (*rules_based_model*) to the new remote one.

In Github, you can see the new branch by clicking on the dropdown box labeled *main*:

Figure 2.14. Any Github repo should show the name of the active branch - in this case, the main branch.



Once you click on the dropdown box, you should see other available branches (if indeed, there are other branches):

Figure 2.15. Clicking on the main dropdown box will show any other available branches, like the rules-based model branch.

Switch branches/tags

X

Find or create a branch...

Branches

Tags

✓ main

default

rules_based_model

[View all branches](#)

Now, let's walk through how to merge our two branches - main and rules_based_model.

Merging two branches

As mentioned above, branches are often used for fixing bugs for or testing out experiments, like new features or models. Take our previous example where we created a new branch called *rules_based_model*. Let's suppose we are satisfied with our code changes for this branch and we want to merge it back to the main branch. Git makes this fairly straightforward.

First, switch back to the main branch by using *git checkout*.

Listing 2.32. Running git checkout main will switch us back to the main branch

```
git checkout main #1
```

Second, use the *git merge* command to merge the *rules_based_model* branch

into the main branch. If there are no conflicts, the merge will be automatic. If there are conflicts, you will have to manually resolve them just like we did above.

Listing 2.33. When in the main branch, we can use git merge rules_based_model to merge the rules_based_model branch into the main branch.

```
git merge rules_based_model #1
```

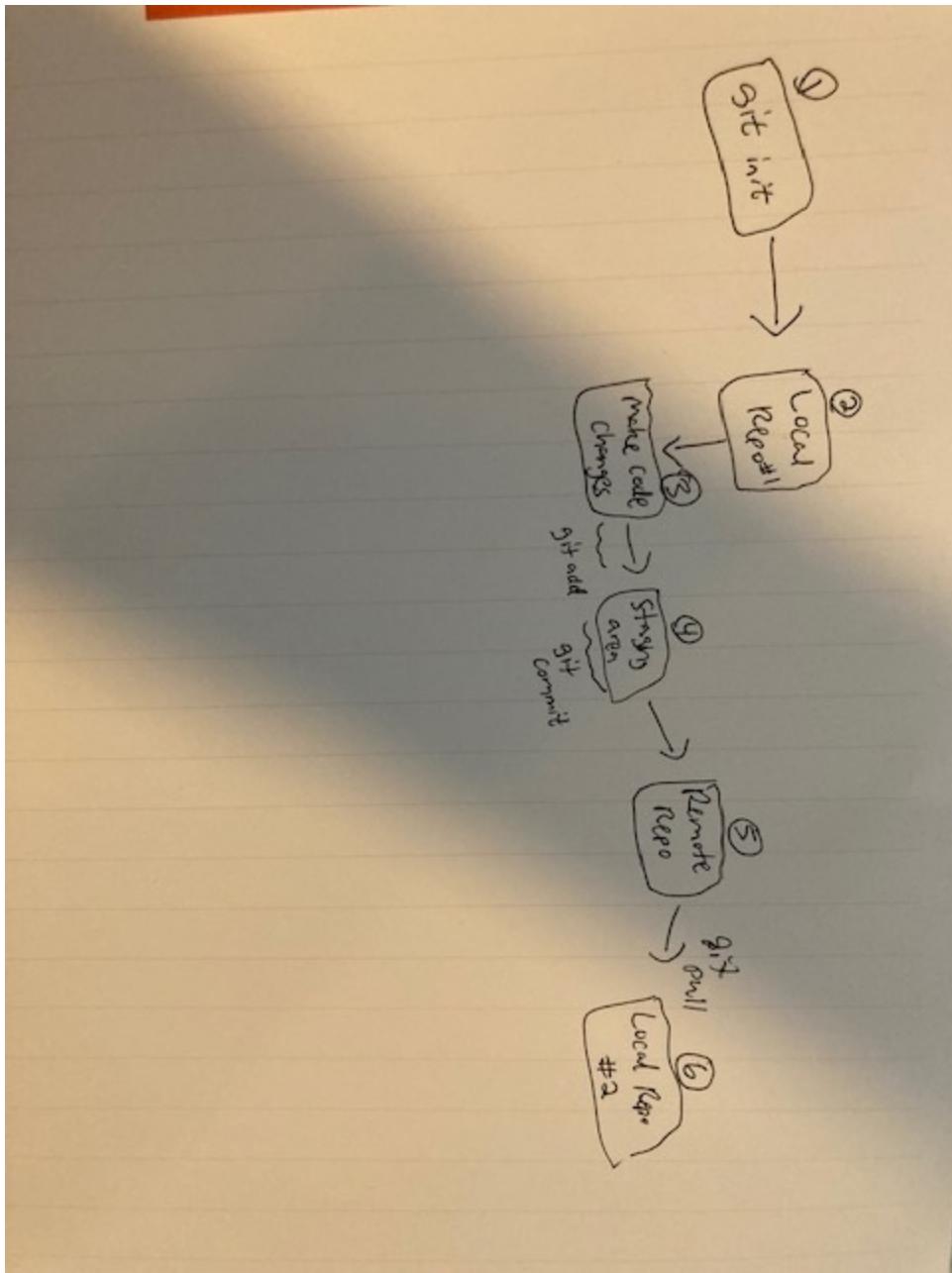
Next, let's summarize the primary Git commands we've learned.

2.7.2 Summarizing Git commands

That covers the Git section. To summarize Git commands, remember this workflow and the corresponding diagram below:

- Use git clone to download an existing repository.
- Use git init to initialize a new repository (#1 in the below diagram)
- Use git add to add changed files to a staging area, ready to be committed (between points #3 and #4 in the diagram)
- Use git commit to create new milestones in your code's life cycle (after adding changes to the staging area in #4)
- Use git push to submit local repo changes to a remote repository (changes are pushed to the remote repo - diagram point #5)
- Use git pull to download and merge the most up-to-date code in the remote repository with your local repository (#6)
- With different branches, you can use the git merge command to merge one branch into another

Figure 2.16. This diagram shows a sample Git workflow from using git init to initialize a new local repo, to git push to update a remote repository with local changes, and git pull to get the latest updates from the remote repo.



As you practice using Git, it may be helpful to use a cheatsheet at first. A sample one can be found on Github's site here:

<https://education.github.com/git-cheat-sheet-education.pdf>.

Now that we've summarized the Git workflow, let's cover a few best practices for using source control.

2.7.3 Best practices for using source control

Let's go through a few best practices when it comes to source control. These apply no matter what type of version control software you're using.

- It's a good idea to start using version control early on in your project. That means even if you've only created one file with a few lines of code.
- *Commit* your code often. *Committing* code means to send the code's latest changes to a repository, generally hosted on a remote server such that multiple (or potentially many) people can access, download the latest code base, or make commits themselves. Usually committing code daily is a good practice, though you can commit your code even more frequently, as well.
- Whenever you commit your code, you should include a brief message describing what changes have been made. We'll explain later in this chapter how to do this.
- Have a consistent policy around *branches*. A *branch* in source control terminology refers to a copy of a codebase. We'll talk more about branches later, but for now, when we say *consistent policy*, we're referring to having an understanding among the contributors to the repository how branches will be used. This will become more clear in the section on branches below.

In the next section, we will cover a way to compare the diffs between two Jupyter Notebook files.

2.8 Comparing Jupyter Notebook files with *nbdime*

Jupyter Notebook is a popular tool for data scientists because it integrates coding with being able to visualize the results of code, such as plots or tables, all in one seamless environment. While you can commit Jupyter Notebook files, just like most other files, it can be more difficult to handle merge conflicts when two users modify the same notebook. This is because Jupyter Notebook files are more complex than simple Python files or R files (these are not much different than plain text files). Jupyter Notebook files are comprised of HTML, markdown, source code, and potentially images all embedded inside JSON. Thus, trying to programmatically identify the differences between files using Git is quite challenging. However, there are a few alternatives to easily identify the differences between two Jupyter Notebook files. One alternative is a Python package called *nbdime*, which we'll dive into next.

2.8.1 Using the *nbdime* package

nbdime can be installed using pip.

Listing 2.34. Use pip to install the *nbdime* library

```
pip install nbdime #1
```

Next, let's take a look at two sample notebook files. These files are both available in the ch2 folder in the book's repo. In the first file below, we import a few packages and create a simple function for cleaning data. The second snapshot shows essentially the same file, but with an extra line of code normalizing an input dataset.

create_plots.ipynb

Figure 2.17. Sample file with a simple Python function

```
import pandas as pd
import matplotlib.pyplot as plt
import random
```

```
def clean_data(df):
    # fill missing values with zero
    df = df.fillna(0)

    # return data frame
    return df
```

create_plots_v2.ipynb

Figure 2.18. A slight modification of the above file, with an extra line of code

```
import pandas as pd
import matplotlib.pyplot as plt
import random
```

```
def clean_data(df):

    # fill missing values with zero
    df = df.fillna(0)

    # normalize data
    df = (df - df.min()) / (df.max() - df.min())

    # return data frame
    return df
```

Now, let's use *nbdime* to show the difference between the two notebook files. From the terminal, you just need to type *nbdime* followed by the names of the two files (see below).

Listing 2.35. Use *nbdime* to show the difference between two notebook files

```
nbdime diff clean_data.ipynb clean_data_v2.ipynb #1
```

Running the above command will generate the following output:

```
-- create_data.ipynb 2022-06-11 19:38:52.936719
+++ create_data_v2.ipynb 2022-06-11 19:43:30.380621
## replaced /cells/1/execution_count:
- 3
+ 1

## modified /cells/3/source:
@@ -3,6 +3,9 @@ def clean_data(df):
    # fill missing values with zero
    df = df.fillna(0)

+    # normalize data
+    df = (df - df.min()) / (df.max() - df.min())
+
# return data frame
return df

## deleted /cells/4-5:
- code cell:
- code cell:
```

In summary, using *nbdime* can be a great way to identify the differences between two Jupyter Notebook files. Once you figure out the differences, you can make adjustments as necessary to one or both files.

2.9 Summary

In this chapter, we discussed source control, Git, common Git commands a data scientist should know, and nbdime.

- Source control enables easier collaboration between different developers (like data scientists) of the same codebase.
- Git is a popular open-source version control software, commonly used by data scientists and software engineers
- To push local changes to a remote repository, you will generally use git add / commit / push as the workflow.
- Branches provide a way of having multiple snapshots of the same repository. They are commonly used to test out experimental features.

- The nbdime Python library can be used to easily check the differences between two Jupyter Notebook files.

2.10 Practice on your own

Now that you've read the chapter, try going through a few git exercises on your own!

- Can you initialize a local repo?
- After initialization, can you push the changes to a remote repo?
- Practice with the git add command. Can you change multiple files, but only commit one?
- How's your merge handling? Try creating a conflict (or work with a friend to make changes that conflict). Can you keep your local changes? Keep only the remote changes?

3 How to write robust code

This chapter covers

- Types of errors in Python
- How to effectively handle exceptions
- Restricting function input data types
- Making your code cleaner

These topics are important for several reasons:

- They are useful (and can be necessary) components of implementing code in production, including machine learning models, data pipelines, or other analytics tools that you may work on as a data scientist
- Regardless of whether you're putting code in production, these topics can make your life easier. For example, what if you're scraping data from various webpages, and you need a way to keep going in case you run into an error? What if you want to easily integrate your code with a separate collection of code files and functions? After this chapter, you'll be covered on how to handle these situations.
- Building on a theme from the first two chapters, several of these topics help make collaboration easier - whether that's between data scientists, data engineers and software engineers, or essentially any developer.

Let's consider how these topics fit into the broader scope of this book.

- Error / exception handling
 - Errors happen in code all the time. Invalid inputs, network failure, memory issues, etc. There are many types of errors. Some are fatal and require stopping code execution entirely, while others can (and should) be handled differently in order to avoid a disruption (such as an application going down, or more simply a loop of API requests failing). An important point to emphasize here is that these issues can occur regardless of whether you are putting code into

production. Now, if you are deploying code into production (such as a machine learning model or data pipeline), then there are specific types of errors you may have to be prepared for, and the handling for these will have to be implemented into the codebase being pushed into production. If you are coding on your own without any need of deploying code into production, you may still have to deal with error handling, like deciding what to do when requesting data from a collection of webpages fails for a particular webpage (we'll see an example of this later in the chapter). Being able to handle errors properly is easier when you're familiar with the main types of errors that can occur - a topic we will cover first.

- Cleaner code
 - Having *clean* code means to make your code easy to read, more easily portable (for example, someone else can easily move or integrate blocks of code from your codebase to a new codebase without breaking anything), and providing a clear understanding of the purpose for each variable, function, file, etc. This idea is especially useful when it comes to collaboration and maintenance of your code. When working on the same codebase with others (see the previous chapter), having clean, easy-to-read code is essential to making your life and the lives of your co-workers easier. One example of cleaner code is being more explicit about the inputs into functions (such as the data types), which is an area we'll tackle in this chapter

Using our customer churn dataset, the table in Table 3.1 shows how these issues can appear in a specific dataset we are already familiar with.

Table 3.1. Below are a few examples using the customer churn dataset of issues you'll learn to handle in this chapter.

High-level problem	Customer churn example issue
Exception handling	Error reading customer churn data from database

Repetitive code	Generate the same evaluation metrics from multiple models predicting customer churn
Restricting function inputs	Making it clear what the data types should be for function inputs, like creating a function to cap the outliers for various columns in the customer churn dataset
Standardizing code formatting	When writing code to process the customer churn dataset, you want to make your code clean and easy to understand.

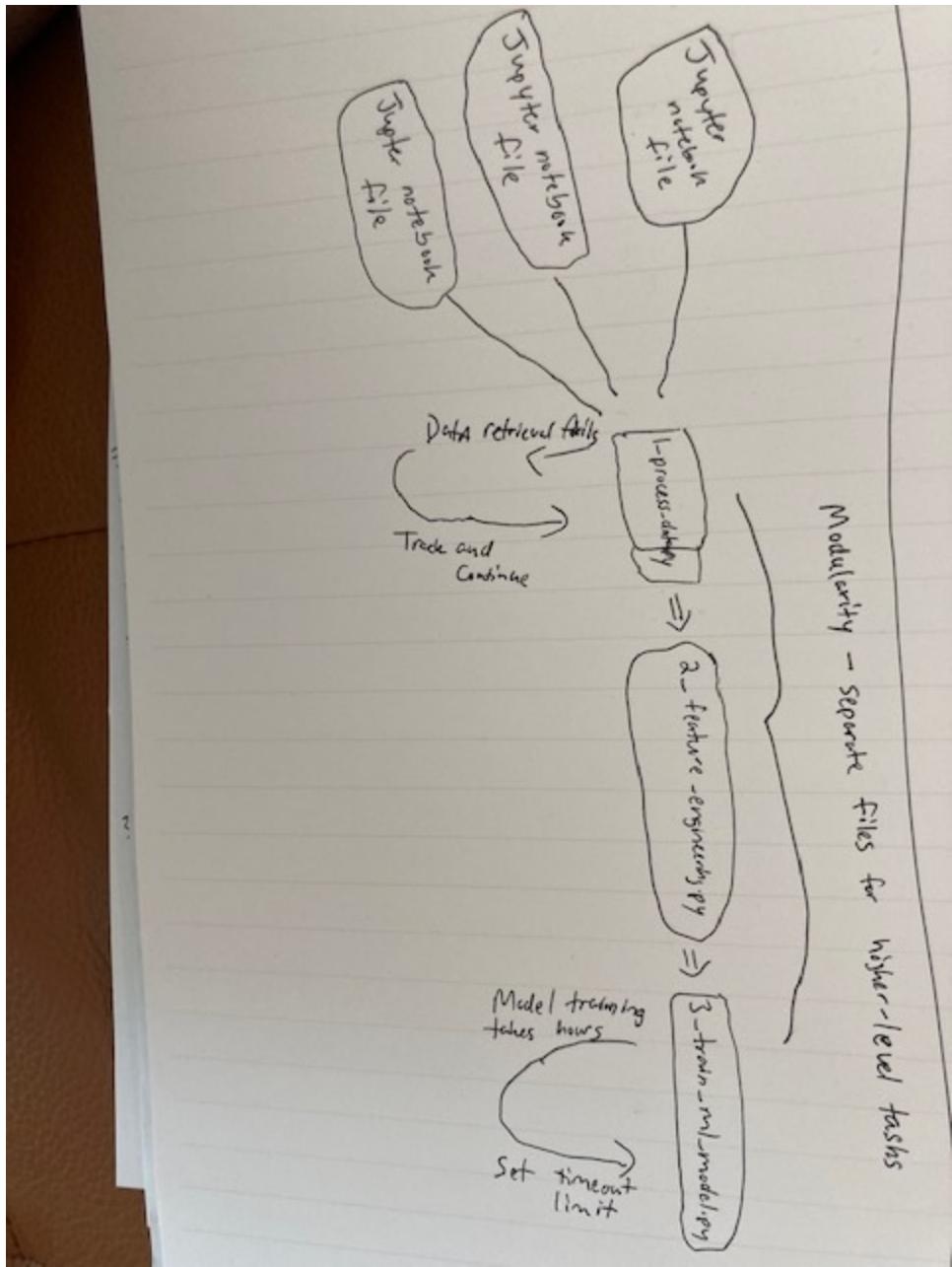
Oftentimes, in a data scientist workflow, you may have a collection of notebook files (Jupyter Notebook). These may not handle various errors that could arise (like network request failure or a database going down preventing you from being able to read data). They may also be difficult to read, and not easily portable to someone else that wants to use your code. This can especially be the case when you're trying to put the code you have across notebook files into production. But it also can be the case when you're working with your co-workers and you want to be able to easily share code with others. In the last chapter, we talked about using *Git* and *source control* as a method of enhancing collaboration between different data scientists, developers, or software engineers. As a next step, we need to learn how to make our code more modular, cleaner, and easier to maintain, along with being able to automatically handle many issues when they arise. The below diagram shows an overview of the process of creating *modular* code.

Modular code is code that is broken into multiple files (and multiple functions) for different tasks, incorporating exception-handling logic, and potentially setting timeouts for model training.

As we go through this chapter, these topics will become more clear, and you'll see hands-on examples of how to deal with them.

In the diagram below, we provide a hint of how modular code could be used. For example, recall the ML pipelines we discussed in chapter one. There are several sequential steps in these pipelines, like processing data, feature engineering, model training, etc. Taking our previous example of predicting customer churn, our modular file structure might look like this:

Figure 3.1. The above diagram demonstrates how we might organize a sequence of modular files, each performing a separate task as part of an overall goal - in this case, implementing a machine learning model.



Each file covers a particular step in the ML pipeline sequence. We could potentially break these further down as well. For instance, in a more complex pipeline, you may have data being consumed from several different sources, such databases, unstructured documents, image data, etc. Depending on the complexity of the features being used and the scope of the code required for extraction, you might split this code into multiple files in order to have a better organized structure. Converting a long Python file working on many tasks in the ML pipeline into a sequence of organized Python files makes the

codebase much more easily maintainable and less prone to bugs. This is true regardless of whether you’re working on an ML pipeline or even a data science application at all, which is why this concept of modularity is also frequently found in software engineering.

Additionally, we may have to incorporate logic to handle errors that may arise. These errors may include:

- Network requests failing
- Database errors (unable to connect to a database or a database query returning zero records)
- Mathematical operation errors (division by zero, for instance)
- Using a package on an unsupported operating system

We can also add logic to set timeouts for code execution, like model training. These topics will be covered in detail as we go through this chapter and the next.

Next, let’s talk about improving the structure of your code. This will prove important as you we walk through making your code more modular and (later in the chapter) how to handle errors effectively.

3.1 Improving the structure of your code

As mentioned earlier in the book, it is often natural for data scientists to write *spaghetti code*, or code that is relatively disorganized, less structured, and sometimes inefficient. This is often because data scientists write code in an exploratory or experimental manner, testing out features, exploring datasets, etc. However, this often creates quite a headache when passing your code to someone else (like a software engineer, for example). In this section, we’re going to walk through a few key points to help you avoid *spaghetti code*. First, let’s give a brief summary of those points.

- Use a standardized style guide, such as PEP8 or Google’s Python Style Guide to make your code more easily readable by others
- Don’t repeat yourself (DRY) is a key software engineering principle to use, which basically means avoiding redundant code (more on this later)

- Modularize your code

We'll explore each of these points in detail in the following sections.

3.1.1 PEP8 standards

PEP8 is a set of standards for how to style your Python code (see Python's official website for the full guidelines: <https://peps.python.org/pep-0008/>). This style guide is good practice for putting your code in a state that is more easily digestible by others. Google's Python Style Guide (available on Google's Github repo: <https://google.github.io/styleguide/pyguide.html>) is another similar stylistic guide that also provides recommendations on how to style your code. Both of these involve a wide collection of standards, so we'll stick with covering a few key points in this section. The short summary is that both of these guidelines provide recommendations for how to style your code. *Styling* your code refers how you structure your code in terms of importing packages, variable names, the way you write common blocks of code (such as if-else statements), and more. Popular IDEs like VS Code and PyCharm often have styling recommendations built-in. We're going to cover a few key styling examples below, but you can search online for the PEP8 standards or the Google Style Guide to learn more.

A few of the key styling standards that we're going to delve into involve:

- Importing packages
- Avoiding compound statements
- Naming conventions
- Avoiding builtin functions or keywords as variable names
- Why to avoid global variables

In the below sections, we're going to start with a poorly-written snippet of code, discuss the code's issues, and then provide alternative ways of styling your code that are more in line with the styling guidelines mentioned above.

Example of a poorly written script

Suppose that you want to write a piece of code to process the customer churn data. We'll start by writing a short snippet that handles reading in the data,

replacing missing values with the median, and taking a sample of the data. The below snippet of code shows an example of a poorly-written way of performing these tasks. You can find this same script in the **ch3** directory in the book's Github repository (look for the file named **1_process_data_before_fix.py**). Now, let's dive into the issues present in this code and how to fix them!

Listing 3.1. Poorly written code example

```
import pandas, numpy #1
num_states = 50 #2
def PROCESSDATA(file_name, int): #3
    global df #4
    df = pd.read_csv(file_name) #5
    df = df.fillna(df.median()) #6
    if int > 0: df = df.sample(int) #7
```

For the first issue, let's discuss package imports.

Importing packages

Many Python scripts start by importing packages. PEP8 standards say that each package import should be done on a separate line. This makes it clear what libraries are being used in the script.

Listing 3.2. When importing packages, it's good practice to put each package import on a separate line

```
# Don't multiple import #1
# packages on the same line #1
import pandas as pd, numpy as np #1

# Instead, separate the package #2
# imports onto multiple lines #2
import pandas as pd #2
import numpy as np #2
```

Additionally, there may be common aliases used in package imports. While using package aliases are not an official part of the above-mentioned style guidelines, using common aliases (like *pd* for *pandas* or *np* for *numpy*) can help avoid confusion. You can typically find the common alias used for a specific package by looking through that package's documentation.

Next, let's discuss the formatting of the if-else clause in our example code snippet.

Avoiding compound statements

Compound statements are lines of code that have multiple statements on a single line.

For instance, taking the *if clause* from our snippet, we're using a function from *numpy* to apply a logarithm transformation to the variable *num_orders* if the value is greater than zero. This code has both the *if* condition and the corresponding logic when the *if* condition is satisfied on the same line. A better approach is to separate these into multiple lines for better readability. This alternative is shown below.

Listing 3.3. It's a good practice to split compound statements into multiple lines. This makes your code more easily readable by others.

```
if int > 0: df = df.sample(int) #1

if int > 0: #2
    df = df.sample(int) #2
```

Next, let's talk about naming conventions when it comes to writing your code.

Naming conventions

PEP8 offers several standards for naming constants, variables, functions, etc.

- Constants should be defined using in all upper-case letters. Example:
SPEED_LIMIT = 55.
- Variables should typically be defined in all lower-case letters. Example:

num_orders = 10.

- Similarly, functions should typically be created using all lower-case letters. Example: *process_data*.

Keeping with our example, suppose that the `num_states` variable is actually a constant that will not change in value throughout our script. Maintaining the guidelines we just described, we can simply change the name of this constant to be in all uppercase letters.

Listing 3.4. Be clear with how you're defining constants, variables, and functions. Following the PEP8 guidelines helps to make these common programming constructs more standardized across any code you or your co-workers may write.

```
# Original #1
num_states = 50 #1

# Updated #2
NUM_STORES = 50 #2
```

Similarly, we should keep function names with the style guidelines by changing the *PROCESSDATA* function to all lower-case. We can also add an underscore for improved readability.

Listing 3.5. Be clear with how you're defining constants, variables, and functions. Following the PEP8 guidelines helps to make these common programming constructs more standardized across any code you or your co-workers may write.

```
def PROCESSDATA(file_name, int): #1

    global df #1
    df = pd.read_csv(file_name) #1

    df = df.fillna(df.median()) #1

    if int > 0: #1
        df = df.sample(int) #1

def process_data(file_name, int): #2

    global df #2
    df = pd.read_csv(file_name) #2
```

```
df = df.fillna(df.median()) #2  
if int > 0: #2  
    df = df.sample(int) #2
```

Now, let's discuss using builtin functions / keywords as variable names.

Never use a builtin function or keyword as a variable name

Another common styling tip is - *never use a builtin function or keyword as a variable name*.

In our `process_data` function above, we use `int` as a parameter, which is also a keyword in Python. Let's replace this parameter name to both avoid this issue and it to make it more clear the purpose of the parameter.

Listing 3.6. Never use a builtin function or keyword as a variable name.

```
def process_data(file_name, int): #1  
  
    global df #2  
    df = pd.read_csv(file_name) #2  
  
    df = df.fillna(df.median()) #2  
    df = df.sample(int) #2  
  
  
def process_data(file_name, num_samples): #3  
  
    global df #4  
    df = pd.read_csv(file_name) #4  
  
    df = df.fillna(df.median()) #4  
  
    if num_samples > 0:  
        df = df.sample(num_samples) #4
```

The difference between the two versions of our `process_data` function comes in the first line of each function definition by changing `int` to `num_samples`.

```
def process_data(file_name, int):
```

vs.

```
def process_data(file_name, num_samples):
```

As an extra example, another common scenario where a keyword is often used as a variable name is the builtin *sum* function to keep track of a sum total. We can fix this issue by replacing *sum* with another variable name, like *total*.

Listing 3.7. It's not uncommon to see the keyword *sum* used as a variable name. However, since *sum* is a Python keyword, this is not a good practice.

```
sum = 0 #1
for i in range(5): #1
    sum += i #1

total = 0 #2
for i in range(5): #2
    total += i #2
```

Next, let's cover *global variables*.

Avoid global variables

Global variables are variables that are *visible* from any other scope. In other words, a *global variable* can be referenced anywhere in the code file that it is defined, regardless of whether it's defined in a function, class (we'll define *classes* in another chapter), or outside of either. The use of global variables should generally be minimized. The reason for this is that global variables can have unintended *side effects* when writing code. At a high level, the term *side effect* has an official meaning in computer science (and software engineering) that refers to situations where a function modifies a variable outside of its environment. In other words, if a function modifies a variable that was originally defined outside of the function, this would be an example of a *side effect*. These *side effects* can make your code more difficult to debug, harder to understand, and less portable. *Global constants*, however, are not bad and are generally fine to be used. As mentioned above, however, constants should typically be defined in all upper-case letters to distinguish them from variables.

Global variables are defined using the *global* keyword, like we've done in our *process_data* function. Placing this *global* keyword in front of the variable *df* makes this variable accessible from outside of the *process_data* function, so we could potentially call or manipulate this variable anywhere else in the code after it is defined. As mentioned, having *global variables* is considered bad practice from a styling perspective because they can make your code messier and more difficult to debug. In our example, we can avoid the use of a global variable, by using the *return* statement instead. This way, our *process_data* function simply returns the data frame, *df*.

Listing 3.8. Global variables should generally be avoided in your code.

```
def process_data(file_name, num_samples): #1

    global df #2
    df = pd.read_csv(file_name) #3

    df = df.fillna(df.median()) #3

    if num_samples > 0:
        df = df.sample(num_samples) #3


def process_data(file_name, num_samples): #4

    df = pd.read_csv(file_name) #5

    df = df.fillna(df.median()) #5

    if num_samples > 0:
        df = df.sample(num_samples) #5

    return df #6
```

To double check our understanding, let's cover another example with global variables. In the below snippet, we've defined a variable named *num_orders*. However, we use the *global* keyword inside of the *update_orders* function in order to

Listing 3.9. Try to minimize the use of global variables in your code. This code snippet is an example of using a function to modify a global variable, when a better alternative is to pass a parameter to the function instead.

```
num_orders = 5 #1  
  
def update_orders(): #2  
  
    global num_orders #3  
  
    num_orders += 1 #4
```

Instead of doing the above, we can pass a parameter to our function, like below. Now, with our updated function below, we could just copy-and-paste this function somewhere else, or import it into another script easily. The original method of using a global variable would not work unless we also moved / imported the global definition of *num_orders* to the other script. If you create many global variables, you can see how this can cause confusion, issues in portability, or difficulties in maintenance of your code.

Listing 3.10. In this example, we update the above function to use a parameter instead of a global variable.

```
def update_orders(num_orders): #1  
  
    num_orders += 1 #2  
  
    return num_order #3
```

There are other tips we will discuss for avoiding global variables when we cover object-oriented programming in a future chapter. To be clear, however, *global constants* are generally fine to use in your code. Since Python doesn't have a *constant* keyword to define constants, like some other languages, you can define a global constant the same way that you define a global variable, except that, as mentioned earlier, you should use all caps when defining your global constants (e.g. MY_CONSTANT = 5).

As mentioned earlier, there are quite a few styling standards that can be applied to your code. You can always check out more by searching for the PEP8 or Google Python Style Guide standards. Additionally, as we'll cover below, there are automated ways to check the styling of your code!

3.1.2 Using pylint to automatically check the formatting and style of your code

It might be difficult to always keep in mind a set of standards when it comes to writing code. That's why the *pylint* library (see <https://pylint.pycqa.org/en/latest/>) was created! This Python package will automatically check your code for potential issues that go against common styling standards. This Python library will perform automated styling checks against your code. In general, a *linter* is a tool that makes stylistic checks against your code (hence, the name *pylint*). To get started with *pylint*, you'll need to install it using pip:

Listing 3.11. Use pip to install the pylint library

```
pip install pylint #1
```

Let's take the example code we went through above and use *pylint* on the sample file to see what happens.

Listing 3.12. Below is the same poorly-written script from above, which we will use with pylint.

```
import pandas as pd, numpy as np #1

num_states = 50 #2

if int > 0: trans_orders = np.log(int) #3

def PROCESSDATA(file_name, int): #4

    global df #5
    df = pd.read_csv(file_name) #6

    df = df.fillna(df.median()) #6

    if int > 0: df = df.sample(int) #6
```

Next, to use *pylint* for our example script, you can open up the terminal (or command line) and execute *pylint* followed by the name of the script (in this case, *1_process_data_before_fix.py*):

Listing 3.13. To run pylint on our script, we just need to execute pylint followed by the name of the script in the terminal.

```
pylint ch3/1_process_data_before_fix.py #1
```

Running the above command will generate the following output in this case. Notice how issues we discussed earlier to avoid are present in the output, such as re-defining a built-in function and having multiple imports on the same line. In this example, since we're not even using those imports, *pylint* will also provide alerts of the libraries not being used. As can be seen in the output below, *pylint* will also show the line numbers corresponding to the stylistic violations to help you quickly know where to make changes. An example of *pylint* not being quite perfect is that it interprets the variable *df* as a constant (which is not the case). However, *pylint* does pick up on the issue that *df* is defined globally.

Listing 3.14. Notice how pylint generates several issues, such as multiple imports in the same line and re-defining the built-in function, int.

```
***** Module 1_process_data_before_fix
ch3/1_process_data_before_fix.py:12:0:
    C0304: Final newline missing (missing-final-newline)
ch3/1_process_data_before_fix.py:1:0: C0103:
    Module name "1_process_data_before_fix"
    doesn't conform to snake_case
    naming style (invalid-name)
ch3/1_process_data_before_fix.py:1:0: C0114:
    Missing module docstring (missing-module-docstring)
ch3/1_process_data_before_fix.py:1:0: C0410:
    Multiple imports on one line
    (pandas, numpy) (multiple-imports)
ch3/1_process_data_before_fix.py:3:0: C0103:
    Constant name "num_states" doesn't conform
    to UPPER_CASE naming style (invalid-name)
ch3/1_process_data_before_fix.py:5:27: W0622:
    Redefining built-in 'int' (redefined-builtin)
ch3/1_process_data_before_fix.py:5:0: C0103:
    Function name "PROCESSDATA" doesn't
    conform to snake_case naming style (invalid-name)
ch3/1_process_data_before_fix.py:5:0: C0116:
    Missing function or method
    docstring (missing-function-docstring)
ch3/1_process_data_before_fix.py:7:4: W0601:
    Global variable 'df' undefined at
    the module level (global-variable-undefined)
ch3/1_process_data_before_fix.py:7:4: C0103:
    Constant name "df" doesn't conform
    to UPPER_CASE naming style (invalid-name)
ch3/1_process_data_before_fix.py:12:16: C0321:
```

```
More than one statement on a
single line (multiple-statements)
ch3/1_process_data_before_fix.py:1:0:
    W0611: Unused numpy
          imported as np (unused-import)
```

While *pylint* is certainly useful in identifying potential styling issues in your code, it doesn't perform automated changes. There are, however, several Python packages available for attempting to make automated changes - including *autopep8* (<https://pypi.org/project/autopep8/>) and *yapf* (<https://github.com/google/yapf>). In the next section, we'll cover an example using the *autopep8* package.

3.1.3 Auto-formatting your code to meet styling guidelines

To get started with *autopep8*, we can again install it using pip:

Listing 3.15. Use pip to install autopep8

```
pip install autopep8 #1
```

autopep8 is used similarly to *pylint* via the terminal. To make automated changes to your code, you need to run *autopep8* followed by the *in-place* parameter and the name of the file (*sample_bad_program.py*).

Listing 3.16. Below we use autopep8 to auto-format our code

```
autopep8 --in-place
ch3/1_process_data_before_fix.py #1
```

In our case, the auto-changes will reformat the package imports and will separate the *if* statement into multiple lines. However, the naming convention for our function and the global variable *df* are not automatically changed.

Updated script:

Listing 3.17. Our updated script can be seen below, after applying autopep8.

```
import pandas as pd #1
import numpy as np #1
```

```
num_states = 50 #2

def PROCESSDATA(file_name, int): #3

    global df #4
    df = pd.read_csv(file_name) #5

    df = df.fillna(df.median()) #5

    if int > 0: #6
        df = df.sample(int) #6
```

If you’re not comfortable using *autopep8* to auto-format your code, you can always use *pylint* to perform the automated checks, and then manually make the changes yourself.

3.2 Avoiding repetitive code

As mentioned earlier, there is a principle in software engineering known as *DRY*, meaning *Don’t repeat yourself*. Avoiding repetitive code is beneficial no matter what situation you’re in. It saves time for you and can make your code more digestible. A key way to avoid repetitive code is to use functions to organize code. Let’s go through a few examples.

Replacing missing values in a data frame

In our first example, there are several lines of code involved in replacing individual column missing values with the corresponding column medians. This can be simplified by using just a single *fillna* method in *pandas* on the dataset, rather than on each individual column (provided that each column’s set of missing values just need to be replaced with its corresponding median). This is a simple way of understanding the functionality of *pandas*, one of the most common Python packages for data wrangling and analysis. The single line of code at the bottom of this snippet performs the same job as the other lines of code put together. To try this code out for yourself, you can see the *simplify_imputation.py* script in the ch3 directory.

Listing 3.18. Replacing missing values on a Python dataset (data frame) can be done using the *fillna* method. This method can be applied to an entire data frame at once, or potentially a

column at a time.

```
import pandas as pd #1

customer_data = pd.read_csv("train.csv") #2

customer_data["total_day_minutes"] =
    customer_data["total_day_minutes"].\ #3
    fillna(customer_data.total_day_minutes.median()) #3

customer_data["total_day_calls"] =
    customer_data["total_day_calls"].\ #3
    fillna(customer_data.total_day_calls.median()) #3

customer_data["total_day_charge"] = #3
    customer_data["total_day_charge"].\ #3
    fillna(customer_data.total_day_charge.median()) #3

customer_data["total_eve_minutes"] = #3
    customer_data["total_eve_minutes"].\ #3
    fillna(customer_data.total_eve_minutes.median()) #3

customer_data = customer_data.\ #4
    fillna(customer_data.median()) #4
```

Another common example of repetitive code in data science comes up with generating metrics for models, which we'll cover next.

Generate metrics for multiple classification models

When developing a model, it's common for data scientists to try out several different methods or models. For instance, if you're building a model to predict customer churn (like our working example), you might try out logistic regression, random forest, gradient boosting, etc. For each of these, you might want to print out evaluation metrics to compare the performance of each model. One way to do this is to manually write the performance checks for each model, like below. As you can see, the code below is quite repetitive. Essentially, we are performing the same operations for each model - random forest, logistic regression, and gradient boosting. A better alternative would be to create a single function that can be applied toward any of the models. Let's show how to do this next.

Listing 3.19. Example code generating evaluation metrics for a collection of models. This code is highly repetitive, which is a sign we should use functions to make our code more clear and less redundant. For this example, we are just using default hyperparameters for the models, but you could also have a process that tunes the parameters for each model as well. The full code for this snippet is available in the ch3 Python file `generate_model_metrics_without_function.py`. For brevity, we've left out part of the repetitiveness of calculating the same metrics of precision and accuracy

```
[import packages] #1

customer_data = #2
    pd.read_csv("../data/customer_churn_data.csv") #2

train_data,test_data = train_test_split(customer_data, #3
    train_size = 0.7, #3
    random_state = 0) #3

feature_list = ["total_day_minutes", #4
    "total_day_calls", #4
    "number_customer_service_calls"] #4

train_features = train_data[feature_list] #5
test_features = test_data[test_list] #5

train_labels = train_data.\ #6
    churn.\ #6
    map(lambda key: 1 if key == "yes" else 0) #6
test_labels = test_data.\ #6
    churn.\ #6
    map(lambda key: 1 if key == "yes" else 0) #6

forest_model = RandomForestClassifier( #7
    random_state = 0).\ #7
    fit(train_features, #7
    train_labels) #7

logit_model = LogisticRegression().\ #8
    fit(train_features, #8
    train_labels) #8

boosting_model = GradientBoostingClassifier().\ #9
    fit(train_features, train_labels) #9

train_pred = forest_model.\ #10
    predict(train_features) #10
```

```

test_pred = forest_model.\ #10
    predict(test_features) #10

print("Train Precision = ", #11
      metrics.\ #11
      precision_score(train_labels, train_pred)) #11
print("Train Recall = ", #11
      metrics.\ #11
      precision_score(train_labels, train_pred)) #11
print("Train Accuracy = ", #11
      metrics.\ #11
      accuracy_score(train_labels, train_pred)) #11

[...additional metrics...]

```

Now, let's build a function to help reduce the amount of code we write. Here, the function will take a model object as input, along with the training / testing features and labels. Setting up the function this way allows us to easily adjust the training / testing sets (like if we wanted to get the performance of a subset of the testing set, for instance). This way, the function will be able to handle essentially any classification model. That means if we add new models, our function should be able to handle those as well. Once we've developed the function, we can either write a single line of code for each model to output the metrics, or we can use a for loop to iterate over each of the possible models we have (see the application of our helper function in Listing 3.21). You can run this code snippet for yourself by checking out the generate_model_metrics_using_function.py file in the ch3 directory.

Listing 3.20. This code snippet shows how we can create a helper function to get the performance metrics for any of the classification models we're testing.

```

def get_metrics(model, #1
               train_features, #2
               test_features, #2
               train_labels, #2
               test_labels): #2

    train_pred = model.predict(train_features) #3
    test_pred = model.predict(test_features) #3

    print("Printing metrics summary...") #4

    print("Train Precision = ", #5

```

```

metrics.\ #5
precision_score(train_labels, train_pred)) #5

print("Train Recall = ", #5
metrics.\ #5
precision_score(train_labels, train_pred)) #5

print("Train Accuracy = ", #5
metrics.\ #5
accuracy_score(train_labels, train_pred)) #5

print("Test Precision = ", #6
metrics.\ #6
precision_score(test_labels, test_pred)) #6

print("Test Recall = ", #6
metrics.\ #6
precision_score(test_labels, test_pred)) #6

print("Test Accuracy = ", #6
metrics.\ #6
accuracy_score(test_labels, test_pred)) #6

return "...Metrics summary complete" #7

```

Next, let's use the helper function created in Listing 3.20 to calculate evaluation metrics based on several customer churn ML models (see Listing 3.21).

Listing 3.21. This code snippet shows how we can create a helper function to get the performance metrics for any of the classification models we're testing.

```

customer_data = pd.read_csv("../data/ #1
    customer_churn_data.csv") #1

train_data, test_data = train_test_split( #2
    customer_data, #2
    train_size = 0.7, #2
    random_state = 0) #2

feature_list = ["total_day_minutes", #3
    "total_day_calls", #3
    "number_customer_service_calls"] #3

train_features = train_data[feature_list] #4

```

```

test_features = test_data[feature_list] #4

train_labels = train_data.\ #5
  churn.\ #5
    map(lambda key: 1 if key == "yes" else 0) #5
test_labels = test_data.\ #5
  churn.\ #5
    map(lambda key: 1 if key == "yes" else 0) #5

forest_model = RandomForestClassifier( #6
  random_state = 0).\ #6
  fit(train_features, #6
  train_labels) #6

logit_model = LogisticRegression().\ #7
  fit(train_features, #7
  train_labels) #7

boosting_model = GradientBoostingClassifier().\ #8
  fit(train_features, #8
  train_labels) #8

get_metrics(forest_model, #9
  train_features, test_features, #9
  train_labels, train_features) #9

get_metrics(logit_model, train_features, #9
  test_features, #9
  train_labels, train_features) #9
get_metrics(boosting_model, #9
  train_features, test_features, #9
  train_labels, train_features) #9

for model in [forest_model, #10
  logit_model, #10
  boosting_model]: #10
  get_metrics(model, #10
    train_features, test_features, #10
    train_labels, train_features) #10

```

Next, let's talk about long if-else blocks and how to avoid them.

Avoiding long if-else blocks

A common scenario where code can get very redundant is when it comes to if-else blocks. A common example of this is the problem of mapping a day

index to the corresponding day of the week. Operations like this are common in data wrangling scenarios. One way to handle this is with a series of if-else statements, like below. However, this has two major drawbacks:

1. It's difficult to read, and will become even less readable as more if-else statements are added (imagine if we had 20, 30, 40 or more if-else statements)
2. It's easy to make a typo or other mistake, causing you headaches as you try to debug the code

The code snippet in Listing 3.22 shows an example of this issue. The corresponding code file for this snippet is called *if_statements_long_sequence.py* in the book repo's ch3 directory.

Listing 3.22. This example shows a series of if-else statements to map the day-index of the week to its corresponding name.

```
if day_index = 1: #1
    day = "Sunday" #1
else if day_index = 2: #1
    day = "Monday" #1
else if day_index = 3: #1
    day = "Tuesday" #1
else if day_index = 4: #1
    day = "Wednesday" #1
else if day_index = 5: #1
    day = "Thursday" #1
else if day_index = 6: #1
    day = "Friday" #1
else if day_index = 7: #1
    day = "Saturday" #1
else: #1
    pass #1
```

Since the previous example has several issues, what's the solution? An alternative to using a large collection of if-else statements is to use a dictionary instead. Recall that a Python dictionary works as a key-value lookup. Here, the keys of the dictionary are the day-indexes of the week - 1, 2, 3, ..., 7 and the values are the day-names of the week - Sunday, Monday, ..., Saturday. We simply need to create the lookup dictionary, and then we can pass the day-index to this dictionary to get the mapped value (day-name

of the week in this case). A similar concept could be used in almost any situation where you need to map some variable (a *key*) to another value. In our example in Listing 3.23, this cuts the code down to two lines, and makes it very clear what's happening. The shortened code can be found in ch3's *replacing_if_else_with_dict.py*.

Listing 3.23. Instead of a series of if-else statements, we can just use a dictionary as a key-value lookup.

```
days_of_week = {1 : "Sunday", #1
                2: "Monday", 3: "Tuesday", 4: "Wednesday", #1
                5: "Thursday", 6: "Friday", 7: "Saturday"} #1

day = days_of_week[day_index] #2
```

Avoiding redundant code, making clear the data types expected for functions, and following standard formatting guidelines all make your code more readable, digestable, and easily portable. Another aspect to add is the concept of *code modularization*, a similar extension to the concept of functions. We'll explain what *modularization* is and how it works next.

3.2.1 Modularizing your code

Modularizing your code is also a good way of making your codebase better structured and easier to understand. *Modularization* essentially means to break your code into blocks such that each block revolves around completing a particular task. This is highly related to the concept of using functions to avoid repetitive code that we discussed in the previous section. However, we can also take it a step further. Instead of just functions, modularization can also mean organizing code into separate files based on the tasks blocks of code are performing. For example,

- Use indexes to mark the order each file should be run. Example:
 - 1 - process_data.py
 - 2 - feature_engineering.py
 - 3 - train_ml_model.py
 - 4 - generate_evaluation_metrics.py
 - Etc.

One advantage of having separate functions for different tasks is that it makes those blocks of code more easily portable. For example, in the below snippet of code, we've defined several functions for processing and cleaning data, such as reading in a file and capping outliers. There could potentially be many other functions, as needed, depending on the range of cleaning needed for your use case. Here, we have a separate function for reading the data, a function for capping outliers, and a function that works as a higher-level function that calls other functions.

Listing 3.24. It's a good idea to break your code execution workflow into multiple functions (or multiple files). This helps improve portability, readability, and if you're pushing code into production, these concepts will greatly help reduce any need for code refactoring by software engineers to get incorporate your code into a production pipeline (like the machine learning pipelines or data pipelines that we covered in the first chapter). This listing's code can be found in the `code_modularization_example.py` file from ch3.

```
import pandas as pd #1

def read_data(file_name): #2
    df = pd.read_csv(file_name) #2
    return df #2

def cap_outliers(feature, #3
                 upper_percentile, #3
                 lower_percentile): #3

    # get the lower and upper #3
    # thresholds based on the input percentiles #3
    lower_thresh = feature.quantile(lower_percentile) #3
    upper_thresh = feature.quantile(upper_percentile) #3

    # cap any outliers at the #3
    # lower / upper threshold bounds #3
    feature = feature.map(lambda val: \
        lower_thresh if val < lower_thresh \ #3
        else upper_thresh if val > upper_thresh \ #3
        else val) #3

    return feature #3

def clean_data(df): #4
    df = df.fillna(df.\ #4
```

```
median(numeric_only=True)) #4

for col in df.columns: #4
    df[col] = cap_outliers(df[col], #4
                           upper_percentile, #4
                           lower_percentile) #4

# additional cleaning... #4
# [code block] #4

return df #4

if __name__ == "__main__": #5

    customer_data = read_data("customer_data.csv") #5

    cleaned_data = clean_data(customer_data) #5

    # additional code #5
    # [code block] #5
```

Notice how we add an extra line in our script:

```
if __name__ == "__main__":
```

This line of code may seem mysterious at first, but it turns out to be very useful. Suppose, for example, that the script we created in Listing 3.24 is called `code_modularization_example.py`. Now, since we've added in this line of code, we can import any of the functions (or all of them) from the script without executing the code block within this statement's indentation. This makes it easy to use functions from other scripts (modules) you create without needing to execute all of the code in the script.

To import a function from our new script, we can write the code in Line 1 from Listing 3.25.

Listing 3.25. Example of importing a function from a script and using it to clean the customer churn dataset. You can run this code for yourself from the `call_clean_function.py` file in ch3.

```
from code_modularization_example import clean_data #1
import pandas as pd #1

customer_data = pd.read_csv("../data/ #2
    customer_churn_data.csv") #2
```

```
customer_data = clean_data(customer_data) #3
```

So far we've tackled several styling issues and how to deal with them in your code. Next, we're going to talk about another aspect of clean code, which is making it clear what the expected inputs and outputs of your functions are.

3.3 Restricting inputs to functions

Unlike languages like Java or C++, Python doesn't require you to specify the data type of a variable when it's being defined. This also allows you to easily re-define a variable to have a completely different data type. While this flexibility can be very convenient, it can also make it more difficult to understand what certain parameters should be when reading through someone else's code. Luckily, more recent versions of Python allow you to provide *type hints* for the parameters in a function that you define. Let's start an example with a simple function that doesn't specify data types. In the snippet in Listing 3.26, we've defined a function to add two numbers. However, technically this code could take two strings as inputs and would concatenate those inputs together without throwing an error.

Listing 3.26. Specifying parameter data types in a function

```
def add(a, b): #1
    return a + b #1
add("x", "y") #2
# "xy" #2
```

To specify data types for the parameters, we just need to add a colon (:), followed by the data type to each parameter. For example, in Listing 3.27 we specify the parameter *a* to be an integer by writing *a:int*. We can also specify the output to be an integer, as well, by appending → *int* to the first line of the function definition. Note that this does not absolutely restrict the data types of the function, but it does provide an easy-to-read hint of what the data types *should* be.

Listing 3.27. Specifying data types in function parameters can be done by adding a colon followed

by the needed data type for each parameter. The expected output data type can be implied using the → symbol followed by the desired output type. Try out this code for yourself in the add_function_with_type_hints.py file from ch3.

```
def add(a:int, b:int) -> int: #1  
    return a + b #1
```

Additionally, we can add extra logic to check the data types. Here, we use Python's built-in *isinstance* function to check whether the given inputs are actually integers. *isinstance* takes two parameters - the variable you want to check and the corresponding data type that you want to match the variable's actual data type against. Here, the function will return True if *a* is an int (likewise for the check on *b*).

The purpose of using *isinstance* is to directly return whether a data type meets some specification. In other words, providing the data type hints doesn't actually restrict the parameter inputs. For example, even though we specify that *a* and *b* should be integers, nothing in the code actually *prevents* you from inputting non-integer values. However, we *can* force a check against the parameter types that validates whether the parameter types are indeed integers. To do this, we use *isinstance*. If running *isinstance* returns False, we can raise a *TypeError* that alerts the user that the inputs must be integers.

Listing 3.28. Use *isinstance* to check whether a variable has a specific data type. The updated function can also be found in the *add_function_with_type_hints.py* file.

```
def add_v2(a:int, b:int) -> int: #1  
    if not isinstance(a, int) and #2  
        not isinstance(b, int): #2  
            raise TypeError("Inputs must be integers") #2  
    return a + b #3
```

Using the coding paradigm in Listing 3.28 can be especially useful if a function has many parameters. This may happen, for instance, if you're creating a function to handle fetching model predictions and you want to make it clear what the model input data types should be. Let's walk through an example like this. In the below code, we're inputting a collection of

features into a function that will then process those features and input them into a model to get predictions. However, by looking at this function definition, how can you tell the correct data types for the input parameters? For example, is customer_age numeric or a bucketed-categorical variable ("young adult" vs. "retired" etc. rather than 18 vs. 60 etc.)? The same question could be applied to the other input features. Like above, we can make the expected data types clear by adding them to the function definition.

Listing 3.29. Creating a function without specifying the parameters

```
# define function without expected parameter data types #1
def get_model_predictions(days_since_registration, #1
    days_since_last_order, #1
    num_orders_last_90_days, #1
    customer_age, #1
    avg_order_last_90_days, #1
    num_logins_last_30_days, #1
): #1

# [code block here] #2
```

Now, let's add the expected parameter types to our sample function. Like Listing 3.29, we just add the expected parameter type for each individual parameter, separating each parameter from its expected type using a colon.

Listing 3.30. Specifying data types in function parameters

```
# add expected parameter data types #1
# in this case, customer_age may be #1
# input as a categorical feature #1
def get_model_predictions(days_since_registration:int, #1
    days_since_last_order:int, #1
    num_orders_last_90_days:int, #1
    customer_age:str, #1
    avg_order_last_90_days:int, #1
    num_logins_last_30_days:int, #1
) -> int: #1

# [code block here] #2
```

Next, let's revisit the earlier example on modularizing our code into functions. This time, we'll add in type hints.

Adding type hints to our modularized code

Copying our earlier example, we can now add type hints to the functions for reading and cleaning data, and capping outliers. Note how in the update code, we specify the following:

- The *read_data* function takes a single string as input and returns a pandas data frame (*pd.DataFrame*).
- *cap_outliers* takes a pandas Series (*pd.Series*), and two float parameters (*upper_percentile* and *lower_percentile*) as inputs, while returning a pandas Series
- The *clean_data* function takes as input and returns a pandas data frame.

Listing 3.31. The below code is a copy of our earlier example in Listing 3.24, except now we add in type hints for each function. This code is also available in the `code_modularization_with_type_hints.py` file from ch3.

```
import pandas as pd #1

def read_data(file_name: str) -> pd.DataFrame: #2

    df = pd.read_csv(file_name) #2
    return df #2

def cap_outliers(feature:pd.Series,
                 upper_percentile: float, #3
                 lower_percentile: float) -> pd.Series: #3

    # get the lower and upper thresholds #3
    # based on the input percentiles #3
    lower_thresh = feature.quantile(lower_percentile) #3
    upper_thresh = feature.quantile(upper_percentile) #3

    # cap any outliers at the lower
    # and upper threshold bounds #3
    feature = feature.\ #3
              map(lambda val: \ #3
                  lower_thresh if val < lower_thresh \ #3
                  else upper_thresh if val > upper_thresh \ #3
                  else val) #3
    return feature #3

def clean_data(df: pd.DataFrame,
```

```

        upper_percentile: float, #4
        lower_percentile: float) -> pd.DataFrame: #4

df = df.fillna(df.median(numeric_only=True)) #4

for col in df.columns: #4
    df[col] = cap_outliers(df[col], #4
                           upper_percentile, #4
                           lower_percentile) #4

# additional cleaning... #4
# [code block] #4

return df #4

if __name__ == "__main__": #5

    customer_data = read_data("customer_data.csv") #5

    cleaned_data = clean_data(customer_data) #5

    # additional code #5
    # [code block] #5

```

That covers going through data type specifications. Next, let's summarize what we've learned about developing clean code.

3.4 Clean code summary

In the above sections, we've covered several key points (also summarized in the diagram below) to get your code into a cleaner, more modular, state:

- Follow guidelines such as PEP8 or the Google Python Style Guide and have a policy to enforce these to make collaboration easier, your code more readable, and easily portable.
- Avoid redundant code by using functions, dictionaries to replace long if-else statements, and paying close attention when you're repeating code.
- Make your code modular by breaking tasks into functions or multiple files if needed. This is especially useful when it comes to deploying code into production as it makes code much more portable, but it also useful in making your own code easier to maintain or understand months in the future.

- Use tools like pylint or autopep8 to automatically find styling issues in your code (or automatically correct them in the case of autopep8)

Table 3.2 shows a summary table of the main points around ensuring you have clean, modular code.

Table 3.2. This table shows a summary of the clean code topics, including reducing repetitive code, avoiding long sequences of if-else statements, and splitting package imports onto multiple lines.

Lots of repetitive code

Create new function(s)

Long sequence of if-else statements

Use a dictionary

import pandas as pd,sys,pyodbc

Separate multiple imports into different lines

def list(...)

Don't use builtin keywords or function names as variable names

File with thousands of lines of code

Separate into multiple files

Now that we've covered how to make your code cleaner and more modular, let's dive into how to automatically handle errors in your coding applications, and how to raise your own errors. Having clean, modular code is one step to making your code more portable and less error-prone, but inevitably errors will arise in your code. For example, you may try to connect to a database, but the connection fails, or you may try to scrape data from a webpage that doesn't exist, and you need a way to handle the issue so your code workflow continues to run. We will tackle these types of issues in the next section.

3.5 How to implement exception handling in Python

Exception handling is the process of adding logic to your code that takes care of errors or warnings that may occur. For example, if your code makes a request to extract data from a webpage, and the request fails, how should your code proceed? Or if a function receives unexpected inputs, what should happen? These questions are ones we need to consider before putting our code in production, but as alluded to above, can also be very useful in everyday coding. Before we can talk about how to *handle* errors in Python, we need to cover the various *types* of errors you may encounter.

Understanding the types of errors you can encounter will help you in your decision-making around how to treat different errors that may arise. Let's discuss the main types of errors in Python next.

3.5.1 What types of errors can we get in Python?

There's quite a few possible error types available in Python. We won't cover all of them here, but will stick with covering several common ones, starting with *type errors* (`TypeError`). To get a full list of exceptions, you can check out Python's exceptions documentation on Python's official website (<https://docs.python.org/3/library/exceptions.html>).

TypeError

One of the most common types of errors is a `TypeError`. A `TypeError` occurs when Python expects a certain data type, but gets another. For example, running the below code will result in a `TypeError` because Python expects numeric inputs when dividing two values.

Listing 3.32. A `TypeError` occurs when an unexpected data type is used for a particular operation, such as attempting to divide two strings.

```
"a" / "b" #1  
  
# Result of running above line of code: #2  
# TypeError: unsupported operand type(s) #2  
#           for /: 'str' and 'str' #2
```

IndexError

Another type of error is an `IndexError`. An `IndexError` occurs when an invalid index is referenced in a sequential object, like a list or tuple. Let's create a small list and try to reference an invalid index. Recall that Python is zero-indexed, meaning that in our 3-element list in Listing 3.33, the last valid index is 2.

Listing 3.33. An `IndexError` occurs when an invalid index is referenced, often in a list or tuple

```
sample_customer_states = ["OH", "PA", "NY"] #1  
# now, try to reference an invalid index #2  
sample_customer_states[3] #2  
  
# Result of running above line of code: #3  
# IndexError: list index out of range #3
```

NameError

Another common type of error is a `NameError`. This error occurs when you reference a non-existent variable name. For instance, if you reference the variable `test`, but haven't previously defined `test`, then Python will throw a `NameError` (see Listing 3.34).

Listing 3.34. `NameErrors` occur when an undefined variable is referenced because Python doesn't recognize the variable name.

```
print(test) #1  
  
# Result of running above line of code: #2  
# NameError: name 'test' is not defined #2
```

KeyError

A `KeyError` occurs when you try to reference a non-existent key in a dictionary. Suppose, for example, we have a dictionary (like in Listing 3.35) storing basic customer information, such as customer ID, credit score, and age.

Listing 3.35. Below we create a simple dictionary, which we'll use to demonstrate a `KeyError`

further below.

```
customer_info = {"customer_id": 12345, #1
                 "state": "OH", #1
                 "area_code": "area_code_415", #1
                 } #1
```

If we try to reference a non-existent key, we will get a `KeyError`. Below, we try to reference `"customer_balance"`, which is an invalid key.

Listing 3.36. `KeyErrors` occur when an invalid (non-existent) key is referenced in a dictionary

```
customer_info["number_customer_service_calls"] #1

# Running the above snippet will #2
# result in the KeyError below: #2
# KeyError: 'account_balance' #2
```

SyntaxError

Syntax errors (`SyntaxError`) occur when improper syntax is attempted to be executed. For example (see Listing 3.37), if we execute a single line of code containing a quotation mark, Python will return a syntax error (because this is invalid syntax).

Listing 3.37. `SyntaxErrors` occur when you attempt to execute invalid syntax.

```
# Executing a line of code with a
# single quotation mark is invalid, #1
# resulting in an error #1
' #1

# Running the above snippet will result #2
# in the SyntaxError below: #2
# SyntaxError: EOL while scanning string literal #2
```

ImportError

Import errors (`ImportError`) occur when you try to import a package or module that doesn't exist. This can happen if you haven't installed the package / module on the machine you're using, or if it just doesn't exist at all. In the code snippet in Listing 3.38, we try to import `doesnt_exist` from

pandas, resulting in an `ImportError`.

Listing 3.38. `ImportErrors` occur when you attempt to import a non-existent package (or an existing package that hasn't been downloaded).

```
# Try to import doesnt_exist #1
# from the pandas library #1
from pandas import doesnt_exist #1

# Running the above snippet will #2
# result in the ImportError below: #2
# ImportError: cannot import name #2
# 'doesnt_exist' from 'pandas' #2
```

AssertionError

An `AssertionError` is a type of error raised by an `assert` statement. An `assert` statement is basically a statement that checks whether a condition is true and raises an error when the condition is false. We're going to describe `assert` statements in more detail in the next section, so for now we'll just show a brief example (Listing 3.39).

Listing 3.39. An `AssertionError` occurs via the `assert` statement

```
# Asserting whether num1 is equal to num2 will return #1
# an error since num1 != num2 #1
num1 = 5 #1
num2 = 10 #1

assert num1 == num2 #2

# Running the above snippet will
# result in the AssertionError below: #3
# -----
# AssertionError Traceback (most recent call last) #3
# <ipython-input-52-b36f64ef1bf0> in <module> #3
#      2 num2 = 10 #3
#      3 #3
# ----> 4 assert num1 == num2 #3
```

Table 3.3 shows a summary of the common error types that we covered above.

Table 3.3. There are many different error types in Python. A few of the most common ones are summarized in this table.

Error type	Description	Example cause of error
AssertionError	Generated from the assert statement when a given condition is false.	assert 1 == 0
ImportError	Occurs when you try to import a package that doesn't exist or can't be found	import invalid_library
IndexError	Generated when you reference an invalid index for an array-type object, like a list or tuple	sample_list = [1, 2, 3] # print(sample_list[10])
KeyError	Occurs when reference a non-existent key in a dictionary	sample_dict = {"a": 5, "b": 10} # sample_dict["c"]
NameError	Generated when you reference a variable that hasn't been defined	print(sample_undefined_variable)

SyntaxError	Occurs when you try to execute code with invalid syntax	'test # error because Python would expect a quotation mark on both sides of 'test'
TypeError	Occurs when you use an invalid data type, such as trying to multiply two strings	"a" * "b"

Now that we've covered a few common types of errors, let's walk through how to handle them when they arise in your code.

3.5.2 Using `try/except` to bypass errors

To handle errors that may arise in your code, it's general practice to use *try/except* code blocks. Before we explain that these code blocks mean, let's setup an example where it would be useful to have a *try/except* block. In the following example, we will use the *yahoo_fin* package to extract stock price data. Error handling is a common need when dealing with network-related operations, such as downloading data from various webpages. For this reason, we will focus our first example on using error handling to deal with issues that may occur when downloading external data from webpages.

Firsly, you can install *yahoo_fin* using pip:

Listing 3.40. Use pip to install yahoo_fin

```
pip install yahoo_fin #1
```

To make sure the package is working properly, you can try downloading data from a known stock ticker, like "AMZN", for instance (see Listing 3.41). To do that, we'll use the *stock_info* module available in *yahoo_fin*. From this module, we'll use the *get_data* method to download the historical price data for AMZN. Running this line of code should return a few thousand rows of

data (each row corresponding to an individual day's stock price).

Listing 3.41. Test to make sure `yahoo_fin` is working by downloading data for AMZN. This code is available in the `test_yahoo_fin.py` file from ch3.

```
from yahoo_fin import stock_info as si #1

# download data for AMZN #2
amzn_data = si.get_data("AMZN") #2

# show the first few result records #3
amzn_data.head() #3
```

Figure 3.2. This snapshot shows the first few records of running the above code.

	open	high	low	close	adjclose	volume	ticker
1997-05-15	0.121875	0.125000	0.096354	0.097917	0.097917	1443120000	AMZN
1997-05-16	0.098438	0.098958	0.085417	0.086458	0.086458	294000000	AMZN
1997-05-19	0.088021	0.088542	0.081250	0.085417	0.085417	122136000	AMZN
1997-05-20	0.086458	0.087500	0.081771	0.081771	0.081771	109344000	AMZN
1997-05-21	0.081771	0.082292	0.068750	0.071354	0.071354	377064000	AMZN

Now that we've tested out `yahoo_fin`, let's use it in the code in Listing 3.42 to download historical price data for a collection of stock tickers. However, if you run this snippet of code, you will run into an `AssertionError` because one of the elements in the ticker list is not a valid stock symbol (the one named "`not_a_real_ticker`").

In our code Listing 3.42, we are using a Python dictionary to store the stock price data so that each ticker will map to its corresponding downloaded data frame (similar to our AMZN snapshot above). Because `not_a_real_ticker` is a fake ticker, we will get an error from the library when we try to download

data from this invalid stock symbol. The error is shown below the code snippet (in this case, we get an *AssertionError* stating that no data was found for this symbol).

Listing 3.42. In this example, we download stock price data based on a collection of stock ticker symbols (plus a fake one). The code will fail when it hits the invalid ticker, without downloading the data from the tickers remaining after the invalid one. You can try running this code yourself using the `download_tech_stock_prices_without_error_handling.py` file. Just remember that you should expect an error when running this code!

```
from yahoo_fin import stock_info as si #1

tickers = ["amzn", "meta", "goog", #2
           "not_a_real_ticker", #2
           "msft", "aapl", "nflx"] #2

all_data = {} #3
for ticker in tickers: #3
    all_data[ticker] = si.get_data(ticker) #3
```

Running the above code will generate the following error:

```
# AssertionError: {'chart': {'result': None, 'error': {'code': 'N
# 'description': 'No data found, symbol may be delisted'}}}
```

Ideally, we'd like to be able run the above code and at least pull the data for any valid ticker we have, while tracking the tickers that failed so that we might investigate those further. This is where the *try/except* clause comes in. The code below shows how to implement the *try/except* clause. Note that instead of directly calling the function to download price data, we add the *try:* statement above. Adding this line of code tells Python that we want to attempt to run the following code within the indentation - in this case, the line of code setting *all_data[ticker]* equal to the result of downloading the price data for that ticker. However, if there is an *AssertionError*, like we saw above, then the *except* clause will *catch* this error, and proceed with the code within the *exception* block, rather than generating an error and stopping code execution. Within the *exception* block, we add a piece of code to append the failed ticker to the *failures* list so that we can later see which tickers failed to retrieve data.

We can also add a print statement to log each attempted download. The print

statement will print out a success message if the download was successful - otherwise, it will print a failure message.

Listing 3.43. This time we use a try/except block to handle the `AssertionError` that arises when an invalid stock ticker is passed to the web request. The exception block here will bypass the error, tracking the stock ticker in a failures list, and the code execution will proceed to the next ticker in the list (rather than exiting the loop with an error). You can run this code yourself using ch3's `download_tech_stock_prices_including_error_handling.py` file.

```
all_data = {} #1
failures = [] #2

for ticker in tickers: #3

    try: #4
        all_data[ticker] = si.get_data(ticker) #4
        print(ticker + " downloaded SUCESSFULLY") #4
    except AssertionError: #5
        failures.append(ticker) #5
        print(ticker + " download FAILED") #5
```

Running this code will generate the following output showing each stock symbol that was successfully downloaded and the one (*not_a_real_ticker*) that failed.

```
amzn downloaded SUCESSFULLY
meta downloaded SUCESSFULLY
goog downloaded SUCESSFULLY
not_a_real_ticker download FAILED
msft downloaded SUCESSFULLY
aapl downloaded SUCESSFULLY
nflx downloaded SUCESSFULLY
```

Additionally, the *failures* list will contain the fake ticker since it was not able to be downloaded.

Figure 3.3. Printing out the *failures* list will show the fake ticker since it was not able to be downloaded.

failures

```
[ 'not_a_real_ticker' ]
```

What if we want to protect our code execution against any type of error, rather than just assertion errors? You can do this easily enough by tweaking the exception clause, like the example in Listing 3.44. Notice how we change `AssertionError` to `Exception`. By making this change, any error will get caught by the `except` clause, and the code within the `except` clause will then be executed. This is useful when you're not sure what type of errors you may encounter, and you just want to protect against any error.

Listing 3.44. We can change `AssertionError` to `Exception` to catch any type of error.

```
all_data = {} #1
failures = [] #2

for ticker in tickers: #3
    try: #4
        all_data[ticker] = si.get_data(ticker) #4
    except Exception: #5
        failures.append(ticker) #5
```

What if we want to execute a piece of code regardless of whether the code within the `try` block succeeded or failed? That's where the `finally` statement comes in, which we'll cover next.

The `finally` clause

In addition to `try` and `except`, we can also add a `finally` clause. The purpose of the `finally` clause is to execute code regardless of whether the code in the `try` clause succeeded. To reiterate, the code block within the `except` clause will only be executed in case the code block within the `try` clause fails. In the `finally` clause, however, the code will execute regardless of whether the code in the `try` clause executed without error. Let's add a `finally` clause to our earlier example (see Listing 3.45 for the updated example).

Listing 3.45. Here, we add a finally block to go along with the try/except blocks. Code within a finally clause is executed regardless of whether an error occurs within the try block.

```
all_data = {} #1
failures = [] #2

for ticker in tickers: #3

    try: #4
        all_data[ticker] = si.get_data(ticker) #4
    except Exception: #5
        failures.append(ticker) #5
    finally: #6
        print(ticker) #6
```

Getting the error type from Exception

Before we close out this section, let's consider the above code snippet where we use *Exception* to catch any type of error that may occur. It might be useful to know what types of errors are occurring, while still having the flexibility to keep attempting our code on multiple tickers. We can figure this out by modifying our code like in Listing 3.46.

Listing 3.46. Below, we tweak our code to record the type of error that occurs whenever an exception is raised. This code is available in the `download_stock_prices_try_except_finally.py` file from ch3.

```
all_data = {} #1
failures = {} #2

for ticker in tickers: #3

    try: #4
        all_data[ticker] = si.get_data(ticker) #4
    except Exception as error: #5
        failures[ticker] = type(error) #5
    finally: #6
        print(ticker) #6
```

Note that we change *failures* to be a dict which maps each failed ticker to its corresponding error. To get the type of error that occurs, we modify the *except* clause to include *as error*:

```
exception Exception as error
```

Doing this allows us to capture the type of error by passing *error* to Python's *type* function. For example, we can see the failed tickers with their corresponding errors by simply printing out the *failures* dictionary:

```
print(failures)
```

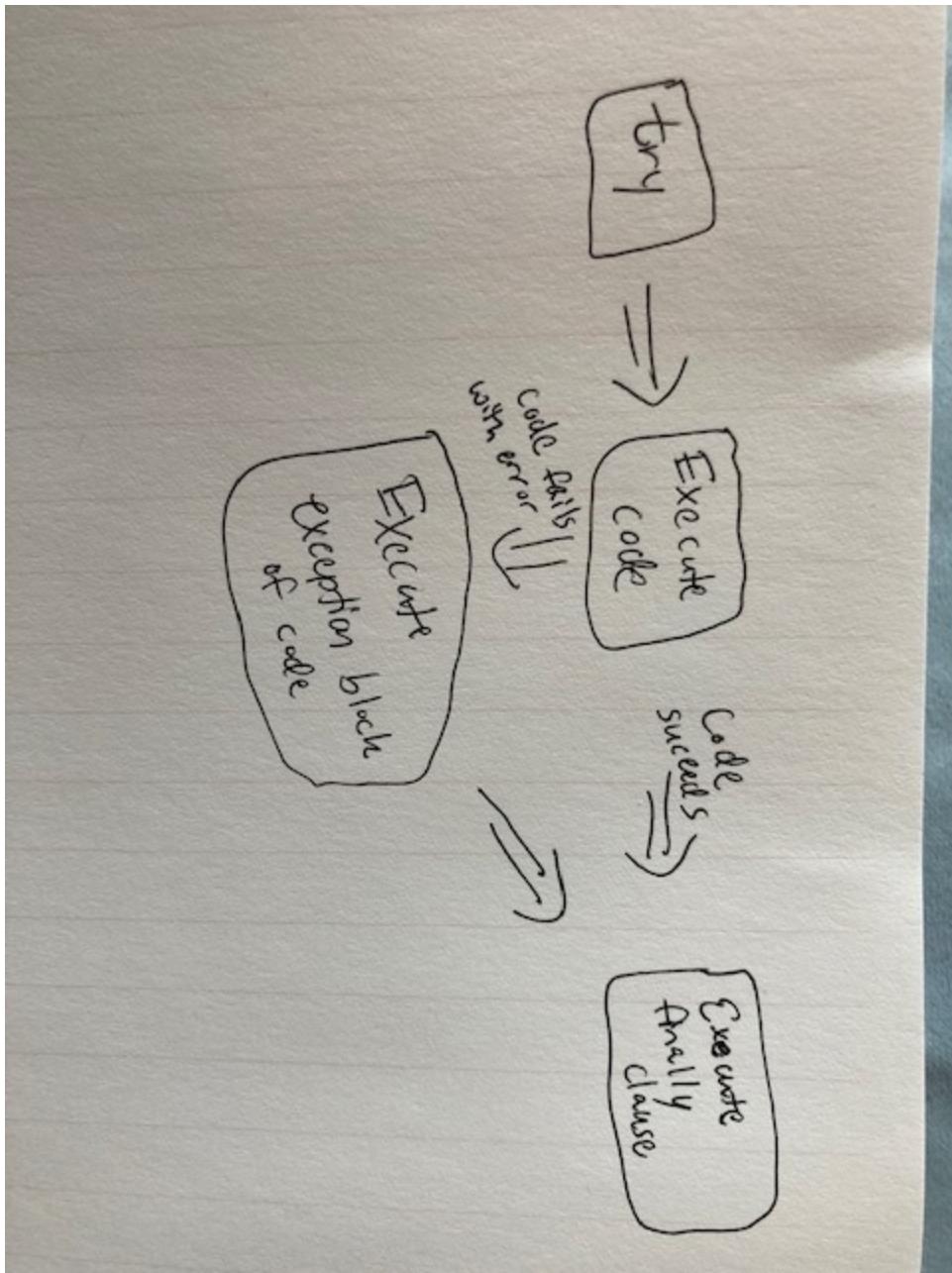
Printing out the dictionary will return the following:

```
{"not_a_real_ticker": <class 'AssertionError'>}
```

Now, let's summarize the error handling workflow.

Error handling workflow summary The below workflow shows a summary of the try/except/finally logic we walked through above.

Figure 3.4. A workflow of the try/except/finally logic showing how code within the try block is executed. If this code fails, the exception block will be triggered to run. In either case, the finally clause will run either after the exception clause, or directly after the try clause if the exception clause is never triggered (no error occurs).



Let's summarize what we've learned so far in error handling:

- Handling potential errors that can arise in code is generally done via *try/except/finally* blocks.
- The *try* clause attempts to execute a block of code.
- If the code in the *try* clause fails with an error, the code in the *except* clause will then be executed. This is in contrast to the code stopping completely.
- The *finally* clause can be used to execute a block of code that will run

regardless of whether the code within the *try* block succeeded or failed.

Now that we know how to use the try / except / finally logic to handle errors that may arise, how do we go about raising error messages of our own? Let's delve into this question in the next section!

3.5.3 How to raise errors

In addition to handling errors that may occur when executing code, it is also possible (and sometimes desirable) to raise errors or to stop execution of code. For example, suppose you're writing code to extract data from a database, followed by processing the data (cleaning, creating new features, etc.). What happens if you extract data from the database, but no records are returned? In this case, you might want to immediately stop code execution immediately, rather than using additional compute resources and time running code when needed data doesn't exist. This might be especially helpful, for instance, if you have a collection of code files (or even just additional code in the same file) that would be executed, depending on non-empty data.

Let's suppose you're working with a database called *customers.db* using the code below. This sample database might include a collection of weather-related data. The data that we want is in a table called *weather_data*. This table has historical information about daily temperatures, rainfall, wind, humidity, etc. In Python, we can retrieve data from this table using the *pyodbc* package. This package allows you to open a connection to a database, which we can use to retrieve data. To do that, we use the *pyodbc.connect* method. This method takes a *connection string* containing several components of information. Firstly, we specify the *driver*, which corresponds to the type of database system we're connecting to (in this case that's SQLite, but could potentially be SQL Server, MySQL, Oracle, etc.). Next, we specify the server name where the database resides. Thirdly, we specify the name of the database (*weather.db*). Lastly, we state that this connection should be a *trusted connection*, which means the connection will automatically use the system credentials (your system username and password) rather than specifying them manually in the connection string.

Once we've created the connection, we have a function that reads data from the *customer_data* table using *pandas.read_sql* (*pd.read_sql*). This method simply takes the connection object (*_conn*) and a SQL query as inputs. It runs the SQL code and returns the result as a data frame in Python. There might also be additional code, such as cleaning, handling missing values, outliers, etc. What if the query returns zero records? In this case, we might want to stop any further code execution and alert the user that no records were returned. This process of alerting the user can be referred to as *raising* an error.

Listing 3.47. The sample code below is retrieving data from a database table and then performing some additional processing. To execute this code yourself, you'll need to setup your own SQLite database. For instructions on how to do this, see the Appendix. The code in this listing is available in the ch3 file `read_from_customer_database_without_error_handling.py`.

```
import pandas as pd #1
import pyodbc #1

conn = pyodbc.connect("DRIVER={SQLite3 ODBC Driver}#2
                      ;SERVER=remotehost123;DATABASE=customers.db;
                      Trusted_connection=yes") #2

def process_data(conn = conn): #3

    customer_key_features = pd.read_sql(conn, #3
                                         """SELECT churn, #3
                                                total_day_charge, #3
                                                total_intl_minutes, #3
                                                number_customer_service_calls #3
                                         FROM customer_data""") #3

    [...additional code] #3
```

To stop code execution when no results are returned, and to provide a custom error message, we can use the *raise* statement. This statement works by using the keyword *raise* followed by a specific type of error (like *Exception*, *AssertionError*, *KeyError*, etc.) and custom error message. For instance, in the example below, we modify our above code to use the *raise* statement. This example raises an *Exception* error with the message "*Empty data returned*" if the queried dataset has zero rows returned. An *Exception* error, as mentioned earlier, represents a general type of error. We could potentially use an *AssertionError* if we generally expect non-zero rows to be returned.

Listing 3.48. Modify process_data function to raise an Exception error if no rows are returned.
This updated script is named `read_from_customer_database_using_error_handling.py` in the ch3 directory.

```
import pandas as pd #1
import pyodbc #1

conn = pyodbc.connect("DRIVER={SQLite3 ODBC Driver}; #2
                      SERVER=remotehost123; #2
                      DATABASE=customers.db;Trusted_connection=yes") #2

def process_data(conn = conn):

    customer_key_features = pd.read_sql(conn, #3
                                         """SELECT churn, #3
                                                total_day_charge, #3
                                                total_intl_minutes, #3
                                                number_customer_service_calls #3
                                         FROM customer_data""") #3

    # if zero rows are returned,
    # raise an Exception #4
    if customer_key_features.shape[0] == 0: #4
        raise Exception("Empty data returned") #4

    [...additional code] #5
```

Next, let's compare using the `assert` statement as an alternative to what we just covered with the `raise` statement.

The assert statement

Another way to raise errors is via the `assert` statement. This statement specifically raises an `AssertionError` if the condition in the statement is not met. For example, we can modify our code snippet from above to use the `assert` statement. A benefit of using `assert` is that it simplifies the validation check we want to perform. On the other hand, `assert` statements will always return an `AssertionError` if the `assert` condition fails. This makes it less flexible than the `raise` statement, which can be used to return any type of Python error. If there isn't a particular type of error that you want to return, then it is generally safe to use either `assert` or `raise`. Otherwise, it is better to use the `raise` statement.

Listing 3.49. In this listing, we modify the earlier code to use an assert statement that checks for whether non-zero rows are returned from the customer data table.. This code is available in the `read_from_customer_database_with_assert.py` ch3 file.

```
def process_data(): #1

    customer_key_features = pd.read_sql(conn, #2
    """SELECT churn, #2
        total_day_charge, #2
        total_intl_minutes, #2
        number_customer_service_calls #2
    FROM customer_data""") #2

    assert customer_key_features.shape[0] > 0 #3

    [... additional code] #4
```

To add a custom error message using the `assert` statement, you just need to add the message following the `assert` condition, followed by a comma. Let's modify our earlier example to use a custom error message.

Listing 3.50. Below, we can modify our previous example to also generate a custom error message in the event that no data is returned via the query. The code file for this snippet is called `custom_assert_message.py` and is in the ch3 directory.

```
def process_data(conn): #1

    # read weather data from database table #2
    customer_key_features = pd.read_sql(conn, #2
    """SELECT churn, #2
        total_day_charge, #2
        total_intl_minutes, #2
        number_customer_service_calls #2
    FROM customer_data""") #2

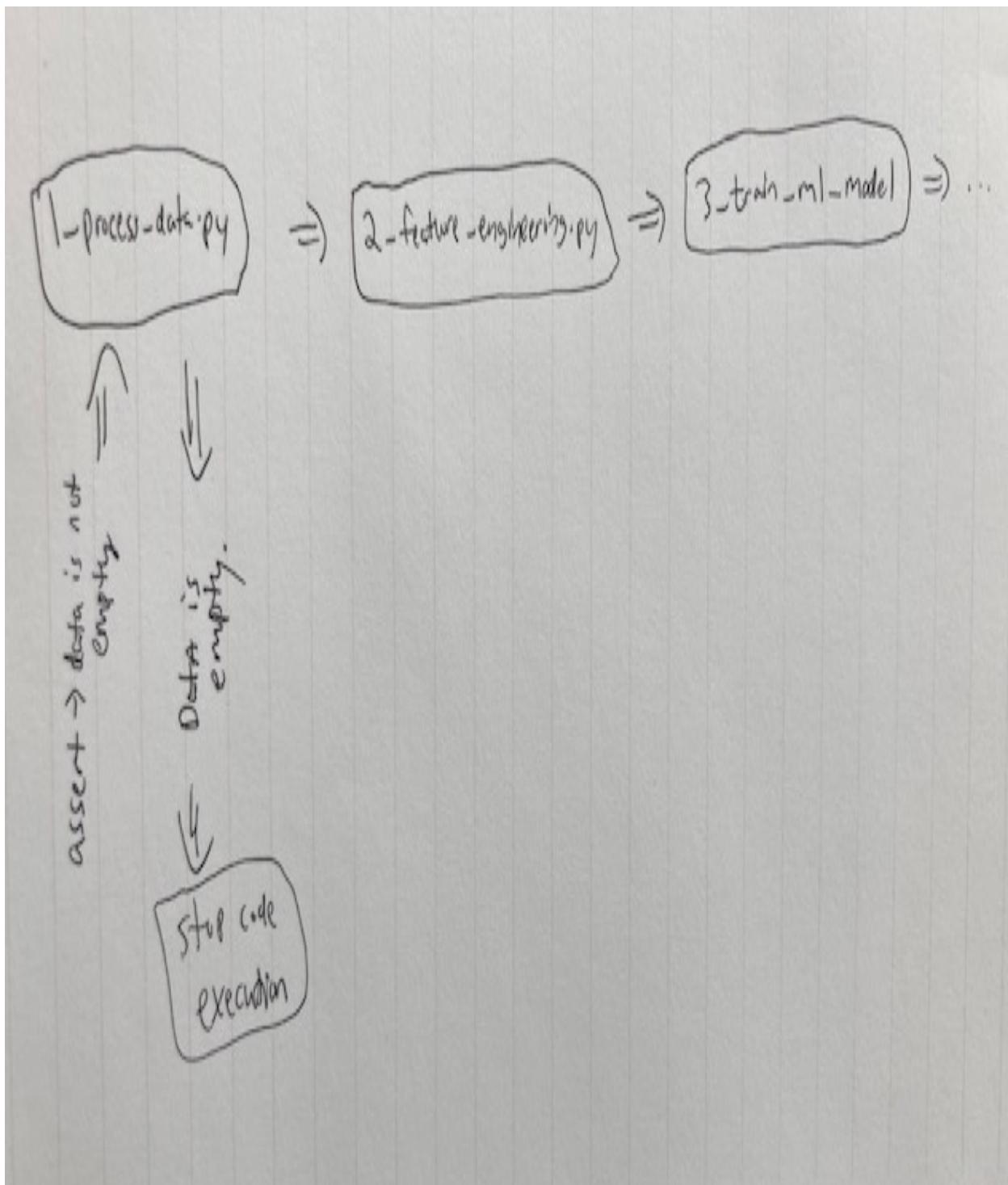
    # if zero rows are returned, raise an AssertionError #3
    # (this time with a custom error message) #3
    assert weather_data.shape[0] > 0, "Empty data returned" #3

    [... additional code] #4
```

The below snapshot summarizes our use case here with the `assert` statement.

Figure 3.5. This workflow shows an example of how we might use the `assert` statement, similar to our example above. We could for example, have a sequence of files that need to be executed.

However, if the data extraction process returns zero records, then none of the remaining files need to run, so we can add an assert statement to stop any remaining code execution on the spot.



That covers *assert* statements. Before we close out this chapter, let's walk through adding documentation to code.

3.6 Documentation

An additional (and important) part of making your code more robust is to document your code. While writing comments like this

```
# read in data
```

can be useful one-liners, there are a few other techniques for documenting your code that come in handy.

3.6.1 Using docstrings

In Python, a *docstring* is a way of documenting functions, packages, classes (covered in a future chapter), or other sections of code. Docstrings are created by adding text between a pair of triple quotation marks. Let's revisit the earlier code modularization example by adding a docstring to our *read_data* function (see Listing 3.51).

Listing 3.51. This code snippet modifies the *read_data* function we created earlier by adding a docstring. This docstring explains the inputs to the function (in this case, that's just the input file name), what the function does, and the object that's returned (the data frame created from reading the source dataset).

```
def read_data(file_name: str) -> pd.DataFrame: #1
    """Inputs:
        file_name: str
            Takes a file name as input and reads in data from the file
            Returns a data frame (pd.DataFrame)
    """ #1
    df = pd.read_csv(file_name) #1
    return df #1
```

We can extend this docstring idea to our other functions in the *code_modularization* example.

Listing 3.52. Extending on the earlier code modularization example, we can now add docstrings to each our functions.

```

import pandas as pd #1

def read_data(file_name: str) -> pd.DataFrame: #2

    """Inputs: file_name: str
        Takes a file name as input and
        reads in data from the file.
        Returns a data frame (pd.DataFrame)"""\#2

    df = pd.read_csv(file_name) #2
    return df #2

def cap_outliers(feature:pd.Series, #3
                 upper_percentile: float, #3
                 lower_percentile: float) -> pd.Series: #3

    """Inputs: feature: pd.Series
               upper_percentile: float
               lower_percentile: float
        Takes a file name as input and
        reads in data from the file.
        Returns a pandas Series (pd.Series)
    """ #3

    lower_thresh = feature.quantile(lower_percentile) #3
    upper_thresh = feature.quantile(upper_percentile) #3

    feature = feature.map(lambda val: \
        lower_thresh if val < lower_thresh \ #3
        else upper_thresh if val > upper_thresh \ #3
        else val) #3

    return feature #3

def clean_data(df: pd.DataFrame, #4
               upper_percentile: float, #4
               lower_percentile: float) -> pd.DataFrame: #4

    """Inputs: df: pd.DataFrame #4
               upper_percentile: float
               lower_percentile: float
        Cleans an input data frame, including
        replacing missing values and capping outliers.
        Returns the processed data frame after cleaning.
    """

    df = df.fillna(df.median(numeric_only=True)) #4
    for col in df.columns: #4

```

```

df[col] = cap_outliers(df[col], #4
    upper_percentile, #4
    lower_percentile) #4

# [additional cleaning...] #4
return df #4

if __name__ == "__main__": #5
    customer_data = read_data("customer_data.csv") #5
    cleaned_data = clean_data(customer_data) #5

    # [additional code block] #5

```

Once you've added docstrings to your code, it's possible to auto-create an HTML file using a Python package called *pdoc*. Let's cover pdoc next.

3.6.2 Using pdoc to auto-create documentation

pdoc is a great Python library for auto-creating HTML documentation files. pdoc uses docstrings along with the code itself to create the documentation file. No knowledge of HTML is required! Let's use pdoc to create an HTML documentation file based on our updated script, `code_modularization_with_docstrings.py`. Again, this file is available in the book repo's ch3 directory.

Before we use pdoc, we need to install it using pip (Listing 3.53).

Listing 3.53. Use pip to install pdoc

```
pip install pdoc #1
```

Now that we have pdoc installed, we can test it out! pdoc can be used directly from the command line. If you have the book repository downloaded on your machine, you can go to the ch3 directory on the command line. From this directory, we can run the line of code in Listing 3.54 on the command line:

Listing 3.54. This code snippet will create an HTML file documenting the input file - `code_modularization_with_docstrings.py`.

```
pdoc --html code_modularization_with_docstrings.py #1
```

Running this command will create a new subdirectory called *html* (also available in the Github repo). Within this folder, you'll find a file called `code_modularization_with_docstrings.html`. This is the new documentation file. Figure 3.6 shows a partial view of the documentation. For the full view, you can open up the file in a web browser.

Figure 3.6. This snapshot shows part of the documentation generated by `pdoc`. In this view, we can see the function definition for `cap_outliers` and its corresponding docstring.

Functions

```
def cap_outliers(feature: pandas.core.series.Series, upper_percentile: float,  
                  lower_percentile: float) -> pandas.core.series.Series
```

Inputs

`feature: pd.Series`

`upper_percentile: float`

`lower_percentile: float`

Takes a file name as input and reads in data from the file.

Returns a pandas Series (`pd.Series`)

The full file shows the following for each function:

- Function parameters
- Full function code
- The docstring defined within each function

That covers `pdoc`. Now, let's summarize the topics we've covered in this chapter.

3.7 Summary

In this chapter, we covered several key topics that help to bridge the gap between data science and software engineering:

- PEP8 and the Google Style Guideline provide outlines for how to best style your code.
- Type hints can be used to provide a developer hints as to what the data types should be for a given function. Use *isinstance* to directly return whether a variable has a specified data type.
- Think before you write repetitive code. As alternatives to redundant code, consider writing functions, using builtin methods (like the pandas `fillna` method), or utilizing dictionaries to replace long if-else blocks.
- Use *try/except/finally* to handle errors that occur in your code, and to add logic that allows your code to continue executing past errors that arise.
- Use *raise* or *assert* to alert the user of an error and stop execution of a code file immediately
- Utilize the pdoc library to auto-create documentation for your code.

3.8 Practice on your own

In this section, spend some time practicing the contents of this chapter on your own.

1. Suppose you have a function called `create_plots`, taking a pandas data frame, a plot type (specified by an integer), and a color (red, orange, yellow, etc.). The output is a list. How could you specify the input data types when defining this function? See this reference: https://www.w3schools.com/python/python_datatypes.asp for a list of common data types in Python.
2. Now, add logic to your function so that if an incorrect data type is entered, the function will raise an error before attempting any code execution.
3. Look up one of the free APIs here: <https://github.com/public-apis/public-apis>, and use one of them in a loop to collect data. Then, add *try/except* to your code to handle any error that may occur. For example, similar to our stock tickers example earlier, you can test what happens if you have invalid inputs and how to handle them using the examples in

this chapter as a reference.

4. Can you refactor the following code snippet to follow standards similar to those covered in this chapter?

```
import sklearn, os, math
```

```
INDEX = 5
for i in range(10):
    INDEX += i
```

```
INDEX = math.log(INDEX)
```

4 Object-oriented programming for data scientists

This chapter covers

- Introducing object-oriented programming (OOP) for data scientists
- Creating your own classes
- Defining class methods

In this chapter, we're going to introduce object-oriented programming (OOP) for data scientists. Object-oriented programming (OOP) can be a confusing and abstract topic, especially if you're reading about it for the first time. For software engineers, OOP is common practice. Data scientists, however, tend to have less exposure to OOP, which is one of the motivations for this chapter. Building on the material we've covered so far, OOP allows for better structure and adaptive flexibility in your code. In a nutshell, OOP also makes your code more easily portable, meaning that rather than having convoluted notebooks of code, you have a codebase that easily be integrated somewhere else. This saves time for you as a data scientist, as well time for others working with your code. It's also useful when you're putting your code into production.

So, what exactly *is* OOP? First things first, object-oriented programming (OOP) refers to a coding paradigm that revolves around *objects* and *classes*. For now, you can loosely think of a *class* as a collection of functions and attributes (data). A common analogy in the physical world is a car. A car has a set of functions, like driving, parking, or backing up. It also has a set of attributes (sometimes called fields), such as make, model, or color. An *object* is an instance of class, just like a Dodge Charger is an instance (or type) of car. As we'll see in the next several sections, applying these concepts to programming can help create better structured and more flexible code.

- Improves the structure and flexibility of code
- Makes your code more easily portable

- Combine functions with data
 - Classes combine functions with data so that only those functions can directly access the data
- More easily create new Python libraries

Now, let's dive in by formally introducing *classes* and *objects*.

4.1 Introducing classes and objects

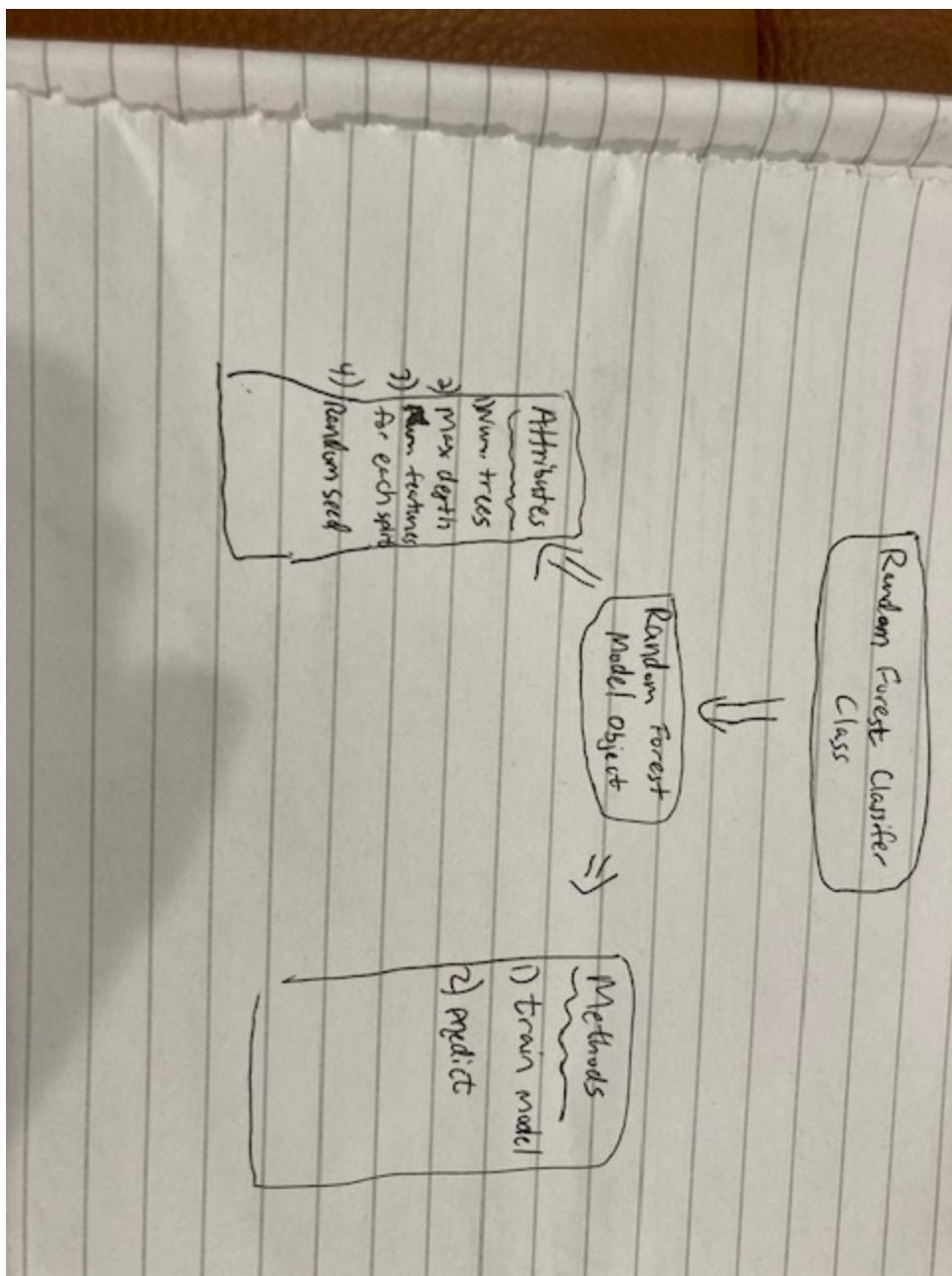
As mentioned above, a *class* is essentially a collection of functions and attributes. The functions within a class are typically called *methods* (we'll dig more into methods shortly). In data science and coding, an example of a class could be a classifier (a random forest, for example), such as a decision tree. The model class might have attributes:

- Number of trees
- Number of features to use for each split
- Max depth of each tree

The classifier will also have functions:

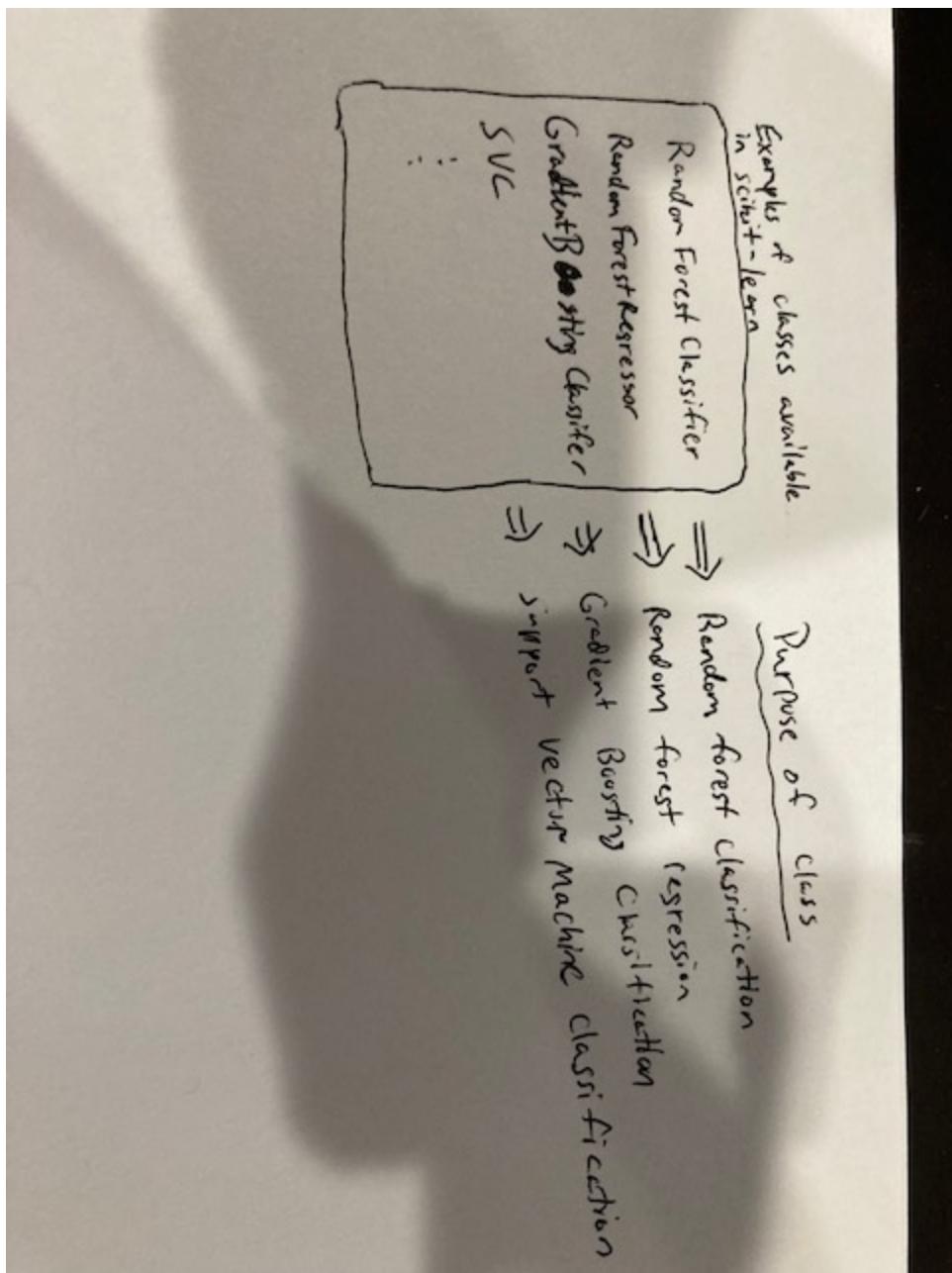
- Train (train model)
- Predict (get predictions)

Figure 4.1. This diagram shows a commonly used class in Python's core machine learning library (scikit-learn). This class represents a random forest classifier. We can create an object of this class, which corresponds to a specific random forest model. Like the random forest algorithm itself, we need to input parameters (number of trees, max depth, etc.). These parameters form the attributes of the random forest model object. Additionally, the object has methods for training and predicting.



In fact, Python's scikit-learn library is structured like this, where each of the popular machine learning algorithms form various classes (see Figure 5.2). Classes allow for better organizational structure. Consider how in the previous chapter, we discussed how functions can help you to avoid repetitive code. As a codebase gets more complex, with a high number of lines of code or many functions, classes are a next step in better organizing those functions. Additionally, creating classes can make your code more easily portable.

Figure 4.2. Python's scikit-learn library has many classes, including the examples shown in this figure. The examples in this figure each correspond to types of machine learning models.



Next, let's walk through how you may be using object-oriented programming without realizing it!

4.1.1 You're using OOP without knowing it...

Python is full of objects. Let's suppose you define a variable called

sample_num. Below, we've set *sample_num* to be equal to the number 5. By doing this, we've actually created an *object*. Why? Because *sample_num* is an *instance* of the *integer* (or *int*) class. If we had set it equal to 10 or 20 etc. our variable would be another *instance* of the *int* class.

Listing 4.1. Define an integer, called sample_num.

```
sample_num = 5
```

Now, let's look at a more data science specific example. If you're a data scientist using Python, you've most likely used packages like pandas or scikit-learn. These packages are also full of object-oriented concepts. For example, as mentioned above, you can think of a random forest classifier in terms of objects and classes.

For example, below we import the *RandomForestClassifier* class from scikit-learn. Secondly, we define an *object* of this class, which we call *forest_model*. This object takes several attributes:

- *n_estimators*
- *max_depth*
- *max_features*
- *random_state*

Listing 4.2. This code creates a RandomForestClassifier object and specifies several key parameters, including the number of trees (*n_estimators* = 100) and *max_depth*.

```
from sklearn.ensemble import RandomForestClassifier #1  
  
forest_model = RandomForestClassifier( #2  
    n_estimators = 100, #2  
    max_depth = 4, #2  
    max_features = "sqrt", #2  
    random_state = 0) #2
```

Additionally, the *RandomForestClassifier* class has several methods, which we can use for our object. For example, we could use our object to train the random forest model:

Listing 4.3. Example of training a random forest model using scikit-learn.

```
forest_model.train(X, y) #1
```

In addition to methods, our random forest object also has instance variables. *Instance variables* are variables that are attached to a specific object of a class. These instance variables generally correspond to the *attributes* of an object. Often times, they are initially defined as inputs when creating an object.

In our example, the values we chose for n_estimators, max_depth, max_features, and random_state correspond to instance variable values for the *forest_model* object. You can reference these variables using dot notation, like below. By *dot notation* here, we mean the object name followed by a period (dot), and followed in turn by the name of the instance variable you want to reference. To get the value of max_features, for example, you would write the code in Listing 5.4:

Listing 4.4. Reference the max_features attributes of our forest_model object

```
forest_model.max_features #1
```

Similarly, you could get the values of other instance variables like this:

Listing 4.5. Reference the max_features attributes of our forest_model object

```
forest_model.n_estimators #1
```

```
forest_model.max_depth #2
```

In the example we've just walked through, we used a class that already exists from scikit-learn. Now, let's walk through how to create your own *new* classes in Python!

4.2 Creating a class in Python

Classes in Python can be created using the *class* keyword. It is common practice (think PEP8 from Chapter 3) to name a class starting with an uppercase letter. This helps to distinguish classes from functions when they are referenced at various points in your code. Additionally, styling standards

recommend using *camelcase* (in other words, avoiding underscores) for the class name. To create our first class, let's first recall a few lines of code from earlier in this book.

In Listing 4.5, we have a few lines of code that are similar to what we've shown in previous chapters. This code reads in the customer churn dataset that we've grown familiar with. It then splits the dataset into train vs. test. Additionally, the code reduces the train and test datasets down to a subset of input columns. Instead of rewriting this code everytime we want to use the customer churn dataset, let's create a class that handles the initial processing of the data. We can also add in additional methods.

Listing 4.6. This code is a collection of tasks we will convert into a class (see Listing 4.6). Currently, this code reads in the customer churn data, gets a subset of features needed, splits the data into train and test, and encodes the label for both the train and test datasets.

```
customer_data = pd.read_csv("../data/  
    customer_churn_data.csv") #1  
  
train_data, test_data = train_test_split(  
    customer_data, #2  
    train_size = 0.7, #2  
    random_state = 0) #2  
  
  
feature_list = ["total_day_minutes", #3  
    "total_day_calls", #3  
    "number_customer_service_calls"] #3  
  
train_features = train_data[feature_list] #4  
test_features = test_data[feature_list] #4  
  
train_labels = train_data.\ #5  
    churn.\ #5  
    map(lambda key: 1 if key == "yes" else 0) #5  
test_labels = test_data.\ #5  
    churn.\ #5  
    map(lambda key: 1 if key == "yes" else 0) #5
```

In Listing 4.6, we define a class called *CustomerData*. Let's break down what's going on in the code:

- First, we define the class name (*CustomerData*)

- Second, we define a *constructor* method. The constructor method is a special method (function) that is always called whenever a new object (or instance) of the class is created. Generally, the purpose of the constructor method is to instantiate variables that can be accessed by other methods in the class. The constructor method is always created using a double underscore, followed by the keyword *init*, and again followed by a double underscore. This can be seen in Listing 4.6.
- Thirdly, within the constructor method, we define a collection of *instance variables*. These are the variables noted by the keyword *self*, followed by a variable name. *self* is Python's way of referencing an object, or instance, of a class. It allows you to access variables or methods attached to a class. In our example, that means accessing variables like *customer_data* or *train_data*, which are defined within the constructor method. These variables can be accessed by other methods in our same class (which we'll see in a later example shortly). You'll also notice how we input *self* as a parameter into the constructor method. This will be the case for any method we add to our class. Again, this allows us to reference any variables we've created that are attached to the class.

Listing 4.7. In this code, we create an initial class called **CustomerData**. Currently, the class handles reading in data from the customer churn dataset, splitting the data into train/test, and initial pre-processing of the data. This code can be found in the `first_class.py` file in the `ch4` directory.

```
import pandas as pd #1
from sklearn.model_selection import train_test_split #1

class CustomerData: #2

    def __init__(self, #3
                 feature_list: list): #3

        self.customer_data = pd.read_csv( #4
            "../data/customer_churn_data.csv") #4

        self.train_data, #5
        self.test_data = train_test_split( #5
            self.customer_data, #5
            train_size = 0.7, #5
            random_state = 0) #5
```

```

    self.train_data = self.\ #6
        train_data.\ #6
            reset_index(drop = True) #6

    self.test_data = self.\ 
        test_data.\ 
            reset_index(drop = True) #6

    self.feature_list = feature_list #7

    self.train_features = self.\ 
        train_data[feature_list] #8

    self.test_features = self.\ 
        test_data[feature_list] #8

    self.train_labels = self.\ #9
        train_data.\ #9
            churn.\ #9
                map(lambda key: 1 if key == "yes" else 0) #9

    self.test_labels = self.\ #9
        test_data.\ #9
            churn.\ #9
                map(lambda key: 1 if key == "yes" else 0) #9

```

Before we add more methods to our class, let's first consider that we can improve our class by making it more generalizable. For example, instead of hard coding the customer churn dataset, we can pass a data file as a parameter to the constructor method, allowing this class to be used for *any* dataset, rather than just the example one we're working with. Additionally, we'll need to update the logic currently processing the label, as well (section 9 in Listing 4.6).

The updated code is in Listing 4.7.

Listing 4.8. This code generalizes our initial class to handle inputting and processing any dataset (assuming a binary target variable). This code is available in the `generalized_dataset_class.py` file from the `ch4` directory.

```

import pandas as pd #1
from sklearn.model_selection import train_test_split #1

class DataSet: #2

```

```

def __init__(self, #3
            feature_list: list, #3
            file_name: str, #3
            label_col: str, #3
            pos_category: str): #3

    self.customer_data = pd.read_csv(file_name) #4

    self.train_data, #5
    self.test_data = train_test_split( #5
        self.customer_data, #5
        train_size = 0.7, #5
        random_state = 0) #5

    self.train_data = self.\
        train_data.\
        reset_index(drop = True) #6

    self.test_data = self.\
        test_data.\
        reset_index(drop = True) #6

    self.feature_list = feature_list #7

    self.train_features = self.\
        train_data[feature_list] #8

    self.test_features = self.\
        test_data[feature_list] #8

    self.train_labels = self.\
        train_data[label_col].\
        map(lambda key: 1 if key == pos_category #9
            else 0) #9

    self.test_labels = self.\
        test_data[label_col].\
        map(lambda key: 1 if key == pos_category #9
            else 0) #9

```

Next, how do we create a new object from our class? Let's do that in Listing 4.8. To create a new object, we need to use the variable name DataSet (the name our class), and input the required parameters that were defined in the

constructor method.

Listing 4.9. This code creates a new object of our `DataSet` class, called `customer_obj`.

```
customer_obj = DataSet( #1
    feature_list = ["total_day_minutes", #1
    "total_day_calls", #1
    "number_customer_service_calls"], #1
    file_name = "../data/customer_churn_data.csv", #1
    label_col = "churn", #1
    pos_category = "yes" #1
) #1
```

Now, we can reference attributes of our class. For example, if you want to access the `train_data` variable we define within the constructor method, you just need to use dot notation, like Listing 4.9.

Listing 4.10. Access the `train_data` variable that was defined when we created the instance of our `DataSet` class.

```
customer_obj.train_data #1
```

Any other attribute of the object can be accessed the same way - using the object name, followed by a dot, then followed by the name of the attribute.

In the next section, let's add methods to our class!

4.2.1 Creating your own methods

Now, let's add a new method to our class. This method will create visuals of the input features.

Listing 4.11. This code adds a new method (called `get_summary_plots`) to our class. This method will generate a histogram of each input feature in the `feature_list` variable.

```
import pandas as pd #1
from sklearn.model_selection import train_test_split #1
import matplotlib.pyplot as plt #1

class DataSet: #2

    def __init__(self, #3
```

```
feature_list: list, #3
file_name: str, #3
label_col: str, #3
pos_category: str): #3

    self.customer_data = pd.read_csv(file_name) #4

    self.train_data, #5
    self.test_data = train_test_split( #5
        self.customer_data, #5
        train_size = 0.7, #5
        random_state = 0) #5

    self.train_data = self.\ #6
        train_data.\ #6
        reset_index(drop = True) #6
    self.test_data = self.\ #6
        test_data.\ #6
        reset_index(drop = True) #6

self.feature_list = feature_list #7

self.train_features = self.\ #8
    train_data[feature_list] #8

self.test_features = self.\ #8
    test_data[feature_list] #8

self.train_labels = self.\ #9
    train_data[label_col].\ #9
    map(lambda key: 1 if key == pos_category #9
        else 0) #9

self.test_labels = self.\ #
    test_data[label_col].\ #9
    map(lambda key: 1 if key == pos_category #9
        else 0) #9

def get_summary_plots(self): #10

    for feature in self.feature_list: #10
        self.train_data[feature].hist() #10
        plt.title(feature) #10
        plt.show() #10
```

We can call our new method using dot notation, similar to how we access attributes of our object. See Listing 4.11 for an example.

Listing 4.12. This code adds a new method (called `get_summary_plots`) to our class. This method will generate a histogram of each input feature in the `feature_list` variable.

```
customer_obj.get_summary_plots()
```

The snapshot in Figure 4.3 shows an example of one of the plots that is generated by running the code in Listing 4.11.

Figure 4.3. This snapshot is an example of one of the exhibits generated by calling the `get_summary_plots` method.



Now, let's add another method. This time, our method will generate evaluation metrics for a model that predicts the target label of an input dataset. For now, our evaluation metrics will just print out the precision based on the train and test datasets against the input prediction labels. When you practice on your own, you can add in additional metrics.

Listing 4.13. Now, we add in a new method for our class that calculates evaluation metrics (currently, precision) based on input prediction columns. This code is available in the `dataset_class_final.py` file from the `ch4` directory. Package imports are omitted for brevity.

```
class DataSet: #1
    def __init__( #2
        self, #2
        feature_list: list, #2
        file_name: str, #2
        label_col: str, #2
        pos_category: str): #2

        self.customer_data = pd.\ #3
            read_csv(file_name) #3

        self.train_data, #4
            self.test_data = train_test_split( #4
                self.customer_data, #4
                train_size = 0.7, #4
                random_state = 0) #4

        self.train_data = self.\ #5
            train_data.\ #5
            reset_index(drop = True) #5
        self.test_data = self.\ #5
            test_data.\ #5
            reset_index(drop = True) #5

        self.feature_list = feature_list #6

        self.train_features = self.\ #7
            train_data[feature_list] #7
        self.test_features = self.\ #7
            test_data[feature_list] #7

        self.train_labels = self.train_data[label_col].\ #8
            map(lambda key: 1 if key == pos_category #8
                else 0) #8
```

```

    self.test_labels = self.test_data[label_col].\ #8
        map(lambda key: 1 if key == pos_category #8
            else 0) #8

def get_summary_plots(self): #9
    for feature in self.feature_list: #9
        self.train_data[feature].hist() #9
        plt.title(feature) #9
        plt.show() #9

def get_model_metrics( #10
    self, #10
    train_pred: pd.Series, #10
    test_pred: pd.Series): #10

    print("Train precision = ", #10
        metrics.precision_score( #10
            self.train_labels, #10
            train_pred)) #10

    print("Test precision = ", #10
        metrics.precision_score( #10
            self.test_labels, #10
            test_pred)) #10

```

To call our method, let's create a simple rules-based prediction that predicts whether a customer in our dataset will churn. This rules-based prediction will be based on the values of two columns in the dataset - `high_service_calls` and `has_international_plan`.

- First, we create a new column called `high_service_calls` based on the value of the number of customer service calls.
- Second, we create a column called `has_international_plan`, which is just a numeric, 1/0, indicator version of the `international_plan` column.
- Third, we use the created columns to create a separate column called `rules_pred`, which gives a 1/0 prediction of whether a given customer will churn.

Listing 4.14. Call the new method by creating a simple prediction for customer churn.

```

customer_obj.\ #1
train_data["high_service_calls"] = customer_obj.\ #1
    train_data.\ #1

```

```

number_customer_service_calls.\ #1
map(lambda val: 1 if val > 3 else 0) #1

customer_obj.\ #2
train_data["has_international_plan"] = customer_obj.\ #2
  train_data.\ #2
    international_plan.\ #2
      map(lambda val: 1 if val == "yes" else 0) #2

customer_obj.\ 
  train_data["rules_pred"] = [max(calls, plan) for calls,plan in
zip(customer_obj.\ #3
  train_data.high_service_calls, #3
  customer_obj.train_data.\ #3
  has_international_plan)] #3

customer_obj.\ #4
  test_data["high_service_calls"] = customer_obj.\ #4
  test_data.\ 
    number_customer_service_calls.\ 
      map(lambda val: 1 if val > 3 else 0) #4

customer_obj.\ #5
  test_data["has_international_plan"] = customer_obj.\ #5
  test_data.\ #5
    international_plan.\ #5
      map(lambda val: 1 if val == "yes" else 0) #5

customer_obj.\ #6
  test_data["rules_pred"] = [ #6
  max(calls, plan) for calls,plan in #6
  zip(customer_obj.\ #6
  test_data.high_service_calls, #6
  customer_obj.test_data.\ #6
  has_international_plan)] #6

customer_obj = DataSet(
  feature_list = ["total_day_minutes",
  "total_day_calls",
  "number_customer_service_calls"],
  file_name = "../data/customer_churn_data.csv",
  label_col = "churn",

```

```
    pos_category = "yes"
)

customer_obj.\
    get_model_metrics(customer_obj.\
        train_data.rules_pred,
        customer_obj.test_data.rules_pred)
```

The result of running this code will print the following:

```
Train precision = 0.44129554655870445
Test precision = 0.458128078817734
```

While the precision results are low because we are only using a simple rules-based model, you can try to improve this result by building a better model in the Practice on your Own section.

That wraps up covering our initial class example. Next, let's walk through another OOP example where we will build a class to help us perform hyperparameter tuning.

4.2.2 Creating a new ML model class

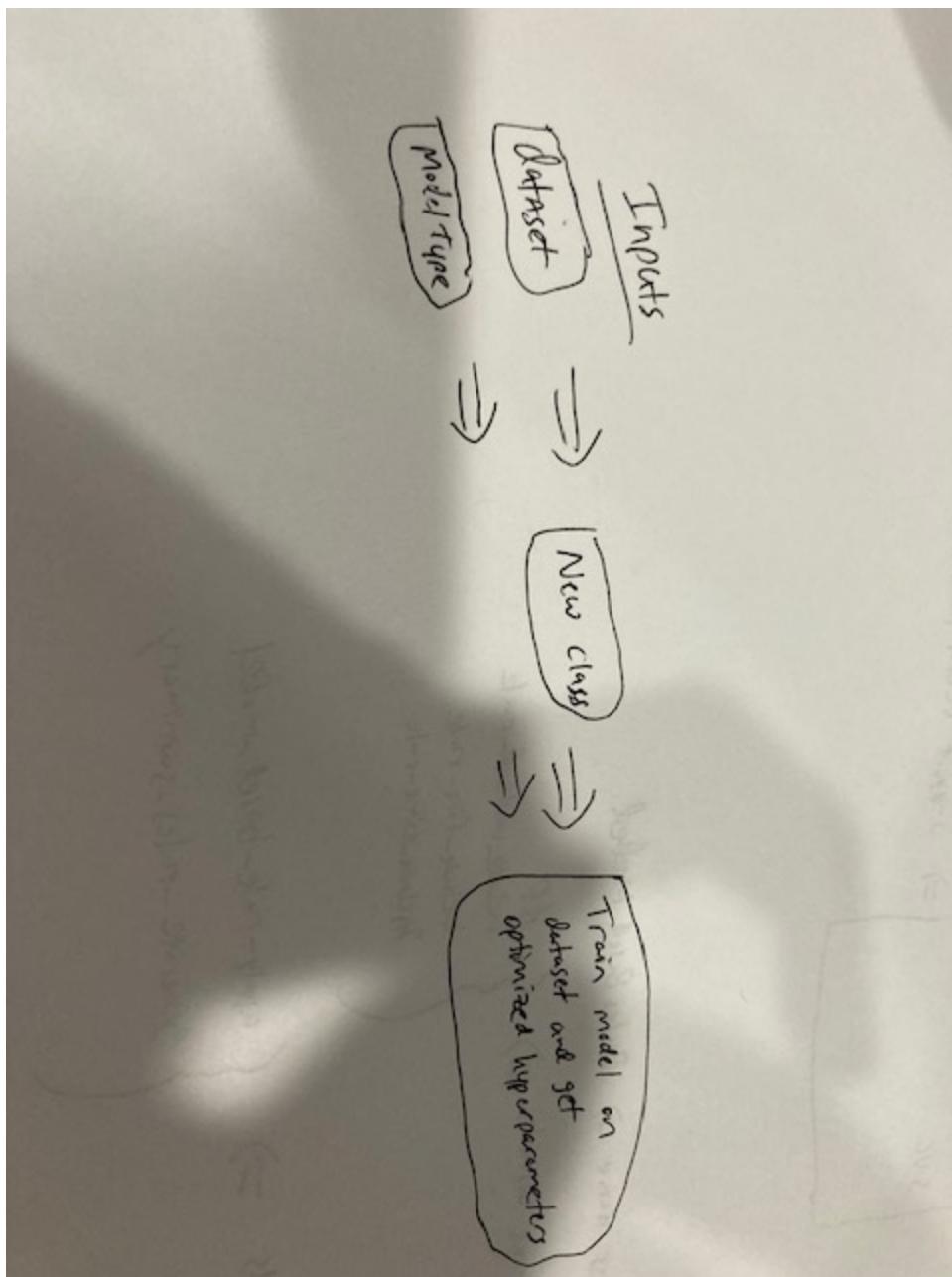
Let's suppose that we wanted to create a web application where a user can do the following:

1. Upload a dataset
2. Select a machine learning model (random forest, gradient boosting, etc.) to use the dataset for training

For example, if you were dealing with the customer churn dataset with this application, you would be able to upload the dataset, select random forest, and train the model to predict whether a customer will churn. We're going to cover how to build lightweight web applications in a later chapter, so for now let's focus on the idea of writing a program that will take a dataset and model type as input, while returning a trained model. Additionally, for our model, we want to be able to automatically perform hyperparameter tuning to get an optimized model for the one selected.

Figure 4.4. This is a overview summary of what we're looking to do in the current section. The

goal is to create a new class that can handle training a model on a dataset and getting the optimized hyperparameters, while allowing the user to pick what type of model to use.



Let's break this problem into a few steps.

1) Create class that performs hyperparameter tuning for a random forest

In the first step, let's create a class that trains a specific model (random

forest), and then we'll follow this up by generalizing the class to handle other models.

Our code to do this is in Listing 5.16.

- First, we start by importing the RandomizedSearchCV class from scikit-learn. This class is often used for hyperparameter tuning to select the optimized parameters for machine learning models.
- Second, we define a new class called *RandomForestModel*.
 - Within this class, we first create the *init* method. As above, this takes the inputs to our class. This time, the inputs will be the values we need to pass to ultimately train the random forest model with hyperparameter tuning. This includes a dictionary of hyperparameters, the number of jobs we want to run in parallel, the metric used for evaluation in the tuning, number of iterations to test out various combinations of parameters, and a random state value to ensure repeatable results.
 - Next, we define a method called *tune*. This method serves as a wrapper function to handle the hyperparameter tuning piece. It takes the inputs to the class and trains an optimized model based on the hyperparameter grid that is input to the class.
 - Lastly, we create a method to handle fetching predictions for the trained model.

Listing 4.15. In this code, we create a new class that can handle hyperparameter tuning for a random forest.

```
from sklearn.model_selection import RandomizedSearchCV #1

class RandomForestModel: #2

    def __init__( #3
        self, parameters, #3
        n_jobs, #3
        scoring, #3
        n_iter, #3
        random_state): #3

        self.parameters = parameters #4
```

```

        self.n_jobs = n_jobs #4
        self.scoring = scoring #4
        self.n_iter = n_iter #4
        self.random_state = random_state #4

    def tune(self, X_features, y): #5

        self.clf = RandomizedSearchCV( #6
            RandomForestClassifier(), #6
            self.parameters, #6
            n_jobs=self.n_jobs, #6
            scoring = self.scoring, #6
            n_iter = self.n_iter, #6
            random_state = self.random_state) #6

        self.clf.fit(X_features, y) #7

    def predict(self, X_features): #8

        return self.clf.predict(X_features) #8

```

Next, let's test out our class. To do this, we're going to use the customer churn dataset.

Listing 4.16. This code uses the random forest-based class that we defined in Listing 4.14 to run hyperparameter tuning and train a random forest model on the customer churn dataset.

```

from sklearn.ensemble import RandomForestClassifier #1
from dataset_class_final import DataSet #1

customer_obj = DataSet(
    feature_list = ["total_day_minutes",
    "total_day_calls", #2
    "number_customer_service_calls"], #2
    file_name = "../data/customer_churn_data.csv", #2
    label_col = "churn", #2
    pos_category = "yes" #2
)

parameters = {"max_depth":range(2, 6), #3
    "min_samples_leaf": range(5, 55, 5), #3
    "min_samples_split": range(10, 110, 5), #3
    "max_features": [2, 3], #3
    "n_estimators": [50, 100, 150, 200]} #3

```

```
forest = RandomForestModel(parameters = parameters, #4
    n_jobs = 4, #4
    scoring = "roc_auc", #4
    n_iter = 10, #4
    random_state = 0) #4

forest.\ #5
    tune(customer_obj.\ #5
        train_features, #5
        customer_obj.\ #5
        train_labels) #5
```

For a combined view of the code implementing the `RandomForestModel` class and testing it out, you can check out the `rf_class_example.py` file in the ch4 directory.

After running the code in the previous listing, we can see the optimal hyperparameters selected by running the code in Listing 4.16.

Listing 4.17. Examine the optimized hyperparameters.

```
forest.clf.best_estimator_ #1
```

For example, the result of Listing 4.16 may look like this:

```
RandomForestClassifier(
    max_depth=3,
    max_features=3,
    min_samples_leaf=5,
    min_samples_split=80, n_estimators=200)
```

We could also use the result to get model predictions for a dataset.

Listing 4.18. Now, we can get the predictions on the train and test sets.

```
train_pred = forest.\ #1
    predict(customer_obj.\ #1
        train_features) #1

test_pred = forest.\ #2
    predict(customer_obj.\ #2
        test_features) #2
```

Now that we've created and used a class to perform hyperparameter tuning for a random forest, let's generalize this to handle other machine learning models.

2) Generalize class to handle other ML models

The code structure we have so far for our class is not much different than what we need to handle other models.

- First, we will change the class name from RandomForestModel to MLModel since we will be creating a class that can handle multiple ML models
- Second, we add an extra input to the new class called *ml_model*. This input will correspond to the type of model that the user will be able to input. For example, this could be the RandomForestClassifier(). Or, it could be RandomForestRegressor() or GradientBoostingClassifier() etc.
- Third, we replace the RandomForestClassifier() input in section 6 of the code for the class (see Listing 4.14) with *self.ml_model*, again corresponding to the type of model that you want to use

Listing 4.19. In this code, we make a few minor adjustments to our previous class in order to make it handle a variety of machine learning models available in scikit-learn. This code is available in the *ml_model.py* file from the *ch4* directory.

```
from sklearn.model_selection import RandomizedSearchCV #1

class MLModel: #2

    def __init__(self, #3
                 ml_model, #3
                 parameters, #3
                 n_jobs, #3
                 scoring, #3
                 n_iter, #3
                 random_state): #3

        self.parameters = parameters #4
        self.n_jobs = n_jobs #4
        self.scoring = scoring #4
        self.n_iter = n_iter #4
        self.random_state = random_state #4
        self.ml_model = ml_model #4
```

```

def tune(self, X_features, y): #5

    self.clf = RandomizedSearchCV(self.ml_model, #6
        self.parameters, #6
        n_jobs=self.n_jobs, #6
        scoring = self.scoring, #6
        n_iter = self.n_iter, #6
        random_state = self.random_state) #6

    self.clf.fit(X_features, y) #7

def predict(self, X_features): #8

    return self.clf.predict(X_features) #8

```

Now, we can repeat a similar process to earlier, and create objects for our new class. To repeat the hyperparameter tuning of the random forest, we can write the code in Listing 4.20. In this case, we just modify the previous code to use `MLModel` rather than `RandomForestModel`, and then input `RandomForestModel()` as an extra parameter into our `forest` object.

Listing 4.20. This code is essentially a repeat of our earlier code to get an optimized random forest model, except now we use the more generalized `MLModel` class rather than the `RandomForestModel` class.

```

parameters = {"max_depth":range(2, 6), #1
    "min_samples_leaf": range(5, 55, 5), #1
    "min_samples_split": range(10, 110, 5), #1
    "max_features": [2, 3], #1
    "n_estimators": [50, 100, 150, 200]} #1

forest = MLModel( #2
    ml_model = RandomForestClassifier(), #2
    parameters = parameters, #2
    n_jobs = 4, #2
    scoring = "roc_auc", #2
    n_iter = 10, #2
    random_state = 0) #2

forest.\ #3
    tune(customer_obj.\ #3
        train_features, #3
        customer_obj.\ #3

```

```
train_labels) #3
```

To use a different model type, like GradientBoostingClassifier, for instance, we just need to change the hyperparameters and tweak the object definition to use that model type rather than a random forest.

As you can see, our code is very similar to the previous examples, with two primary differences. First, we modify the hyperparameter search grid to be fine with gradient boosting models (for example, including learning rate as one of the hyperparameters). Secondly, we directly specify that we want to use the GradientBoostingClassifier model class from scikit-learn, rather than a random forest.

Listing 4.21. This code is essentially a repeat of our earlier code to get an optimized random forest model, except now we use the more generalized MLModel class rather than the RandomForestModel class.

```
from sklearn.ensemble import GradientBoostingClassifier #1

parameters = {"max_depth":range(2, 6), #2
              "min_samples_leaf": range(5, 55, 5), #2
              "min_samples_split": range(10, 110, 5), #2
              "subsample": [0.6, 0.7, 0.8], #2
              "max_features": [2, 3], #2
              "n_estimators": [50, 100, 150, 200], #2
              "learning_rate": [0.1, 0.2, 0.3]} #2

gbm = MLmodel(ml_model = GradientBoostingClassifier(),
               parameters = parameters, #3
               n_jobs = 4, #3
               scoring = "roc_auc", #3
               n_iter = 10, #3
               random_state = 0) #3

gbm.tune(customer_obj.train_features, #4
          customer_obj.train_labels) #4
```

As one additional point, note that the class we've created not only handles classification models, but could also handle regression models. For example, if you wanted to use the RandomForestRegressor (random forest regression) model class from scikit-learn, you could easily do that using the code we have here. To illustrate this, here's what that would look like (assuming we

have a new dataset and continuous label):

Listing 4.22. Now, we just modify our code to use the RandomForestRegressor instead. The rest of the code is essentially identical to when we used the RandomForestClassifier. The new_train_data and new_train_label variables represent a new training set with a different training label.

```
parameters = {"max_depth":range(2, 6), #1
              "min_samples_leaf": range(5, 55, 5), #1
              "min_samples_split": range(10, 110, 5), #1
              "max_features": [2, 3, 4, 5, 6], #1
              "n_estimators": [50, 100, 150, 200]} #1

forest_regressor = MlModel(ml_model = RandomForestRegressor(), #2
                           parameters = parameters, #2
                           n_jobs = 4, #2
                           scoring = "roc_auc", #2
                           n_iter = 10, #2
                           random_state = 0) #2

forest_regressor.tune(new_train_data, #3
                      new_train_label) #3
```

That's covers object-oriented programming for now. There's more to OOP, and we'll get more practice with it in future chapters.

4.3 Summary

In this chapter, we introduced object-oriented programming and discussed how it can be used for data science.

- Object-oriented programming is all around you in Python. Everything is an object.
- A class is a collection of functions and variables together.
- An object is an instance of a class. For example, the integer 10 is an instance of the integer class.
- Methods are basically functions defined inside of classes.
- Instance variables are variables (attributes) attached to an object of a class. They can take different values depending on different objects of and inputs for the same class.

4.4 Practice on your own

1. Can you add additional ML evaluation metrics to the DataSet class we defined earlier?
2. Can you train a better customer churn prediction model than the rules-based one we did earlier? Feel free to explore additional features available in the dataset. Then, use the more robust metrics method you created in step 1 to evaluate the new model.
3. Can you modify the MLModel class to also handle automatically splitting an input dataset into train / test? How would you make the train and test sets accessible to the user after inputting a dataset?
4. Can you add a new method (or multiple methods) to MLModel to create visualizations of an input dataset?

5 Creating progress bars and timeouts in Python

This chapter covers

- How to create a progress bar to monitor code execution (such as time-consuming loops)
- How to monitor the progress of training machine learning (ML) models
- How to create a timer to auto-stop long-running code

The purpose of this chapter is to serve as a key foundation for the next several chapters on scaling. We defined scaling back in Chapter 1, and will visit it in more detail next chapter. For now, consider of the components within scaling is to deal with larger amounts of data from an efficiency (less time) perspective. If you've been working in data science for some time, you're well aware some types of code execution (such as model training or data extraction) can take a long time. For example, maybe you're training a machine learning model on a dataset. How long do you have to wait? A few seconds? Minutes? Hours? Without a progress bar, it may be difficult to know. Or what if you're scraping weather data from a collection of hundreds or thousands of locations? How would you know if your code became stuck or was still progressing? Having a way to monitor the progress of the running code or stopping long-running code automatically once a time limit is reached can very useful in these situations. For example, you might want to use a progress bar to track as you loop through this collection of weather-related webpages to extract data. Or you might want to know how far long the training has come when training an ML model.

Let's discuss how these topics fit into the bigger picture of this book.

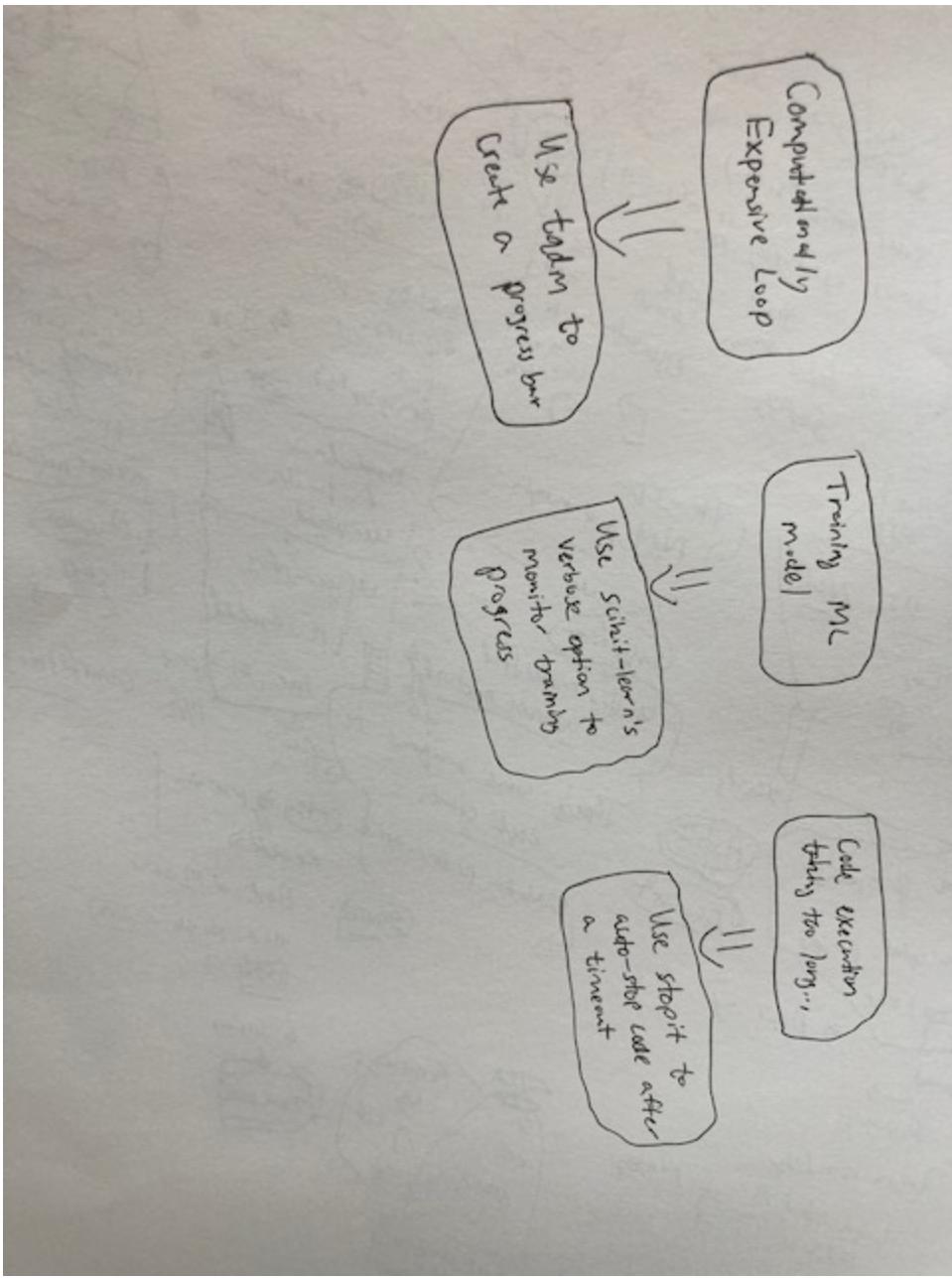
- Progress bars are invaluable tools for letting you know how far along a piece of code has executed. This can be helpful if you're not sure how long code will run so that you're not in the dark while it is executing. Python has several packages available to create progress bars, including

tqdm and *progressbar2*. We'll be delving into the *tqdm* library shortly.

- You're training a machine learning model (like predicting customer churn), and you're uncertain how long it will take given your dataset. Having a progress monitor in this situation can help you know whether you should expect to wait only seconds, hours, or somewhere in between.
- You may have code running for a long period of time and you want to add a time-out in case it runs past a certain point. This could be useful in variety of situations, such as avoiding have code run for too long in a production setting or simply if you're testing out training a model and you want to check if it will complete in under a specified time frame. For this, you can use a handy package called *stopit*, which we'll explain in more detail later in this chapter.

These issues correspond to the diagram in Figure 5.1. These items have something in common - they all deal with an efficiency-related issue. Being able to better monitor the progress of your code will be valuable once we are working through examples of improving the runtime of your code (see the next chapter, for instance).

Figure 5.1. This diagram shows three common problems you may encounter in your data science work, and the Python packages we can use to solve them. In the next few sections, we'll explain what these packages do, and how to use them to monitor code progress and to auto-stop long-running code.



In the next few sections, we'll cover the issues and proposed solutions in the above diagram. First, let's start with the *tqdm* package for creating progress bars in Python.

5.1 Creating progress bars and timing Python processes

The *tqdm* package is a popular Python library for creating progress bars on executing code. *tqdm* can be useful when you’re interesting in monitoring how far along you are when running code. For example, suppose you build a tool to download data from a collection of webpages. Depending on the number of webpages, this process might take a while. *tqdm* will tell you exactly how many iterations you’ve completed (and how many are left). Without further ado, let’s get started!

You can install *tqdm*, like the other libraries mentioned in this chapter, using pip:

Listing 5.1. Install *tqdm* using pip

```
pip install tqdm #1
```

The main way *tqdm* is used involves iterations, like *for loops*. After running *from tqdm import tqdm*, we just need to wrap the keyword *tqdm* around the *iterable* that we are looping over. Let’s extend an example from Chapter 3 where we downloaded stock price data. This time, though, we’ll use *tqdm* to monitor the progress of the downloads.

Listing 5.2. Use *tqdm* to monitor the progress of download data from a sequence of webpages

```
from tqdm import tqdm #1
from yahoo_fin import stock_info as si #1

tickers = ["amzn", "meta", "goog", #2
           "not_a_real_ticker", #2
           "msft", "aapl", "nflx"] #2

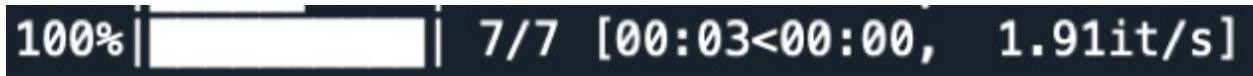
all_data = {} #3
failures = [] #4

for ticker in tqdm(tickers): #5
    try: #6
        all_data[ticker] = si.get_data(ticker) #6

    except Exception: #7
        failures.append(ticker) #7
        print(ticker) #7
```

If you execute the above code in a terminal, or in some other environment (like a notebook or IDE), you should see a progress bar being printed out, like below, which shows the total number of iterations we covered and the amount of time it took to handle going through them (just one second in this case).

Figure 5.2. Example of a complete progress bar being printed using tqdm



Part way through the code execution, you'll see output like below, which shows how far long we've come - in this case, the progress bar tells us we've covered the first 4 iterations out of 7 in total.

Figure 5.3. Example of a partially complete progress bar being printed using tqdm



tqdm can also be used in list comprehensions or dictionary comprehensions, as well. Recall that a list comprehension is a compact way of executing a for loop inside of a list (and likewise, a dictionary comprehension is a compact way of using a for loop to create a new dict). The result is a new list (or dict in the case of a dictionary comprehension).

For example, we could run the code broken out in Listings 5.3 and 5.4 using a *list comprehension* to get the evaluation metrics for each model. Listing 5.3 shows functions that we use to calculate common classification metrics, like precision and recall. The list comprehension in Listing 5.4 creates the list of evaluation metrics by using a for loop over each model. Much of the code below is similar to what we saw in Chapter Three examples, except now we are modifying the code to use a list comprehension and *tqdm*.

Listing 5.3. This code listing creates a function called `get_metrics` that we can use to extract metrics like precision or recall for a model. Listing 5.4 uses this function to get metrics on several different customer churn models, including logistic regression and boosting.

```
from sklearn import metrics #1
from sklearn.ensemble import #1
```

```
RandomForestClassifier, #1
GradientBoostingClassifier #1

from sklearn.linear_model import LogisticRegression #1
from tqdm import tqdm #1
from ch4.dataset_class_final import DataSet #1

def get_metrics(model, #2
    train_features, #2
    test_features, #2
    train_labels, #2
    test_labels): #2

    train_pred = model.\ #2
        predict(train_features) #2
    test_pred = model.\ #2
        predict(test_features) #2

    train_precision = metrics.\ #2
        precision_score(train_labels, #2
            train_pred) #2
    train_recall = metrics.\ #2
        recall_score(train_labels, #2
            train_pred) #2
    train_accuracy = metrics.\ #2
        accuracy_score(train_labels, #2
            train_pred) #2

    test_precision = metrics.\ #2
        precision_score(test_labels, #2
            test_pred) #2

    test_recall = metrics.\ #2
        recall_score( #2
            test_labels, #2
            test_pred) #2

    test_accuracy = metrics.\ #2
        accuracy_score( #2
            test_labels, #2
            test_pred) #2

    return train_precision, #2
        train_recall, #2
        train_accuracy, #2
        test_precision, #2
        test_recall, #2
```

```
    test_accuracy #2
```

Listing 5.4. Here, we use tqdm to monitor the progress of getting evaluation metrics for each model. This makes use of tqdm's ability to monitor the progress of list comprehensions. The code for this example is available in the `tqdm_with_list_comprehension_file.py` file in the `ch5` directory. Make sure to run this code from the parent directory (as you can see based on the code snippets in this listing).

```
customer_obj = DataSet(feature_list = [ #1
    "total_day_minutes", "total_day_calls", #1
    "number_customer_service_calls"], #1
    file_name = "data/customer_churn_data.csv", #1
    label_col = "churn", #1
    pos_category = "yes" #1
) #1

forest_model = RandomForestClassifier( #2
    random_state = 0).\\ #2
    fit(customer_obj.\\ #2
        train_features, #2
        customer_obj.train_labels) #2

logit_model = LogisticRegression().\\ #3
    fit(customer_obj.\\ #3
        train_features, #3
        customer_obj.\\ #3
        train_labels) #3

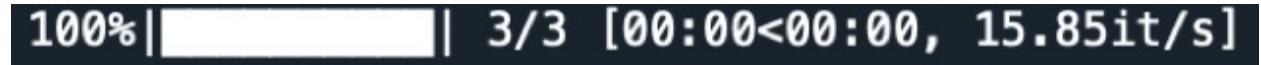
boosting_model = GradientBoostingClassifier().\\ #4
    fit(customer_obj.\\ #4
        train_features, #4
        customer_obj.\\ #4
        train_labels) #4

model_list = [forest_model, #5
    logit_model, #5
    boosting_model] #5

all_metrics = [get_metrics(model, #6
    customer_obj.train_features, #6
    customer_obj.test_features, #6
    customer_obj.train_labels, customer_obj.test_labels) #6
    for model in tqdm(model_list)] #6
```

The result of running the code in Listings 5.3 and 5.4 will generate a progress bar like that in Figure 5.4:

Figure 5.4. Example of a `tqdm` progress bar from executing a list comprehension.



That covers `tqdm`. In summary, `tqdm` allows us to easily create progress bars to monitor how long it is taking for our code to execute. Next, let's discuss how to monitor the progress of ML model training.

5.2 Monitoring the progress of training ML models

While `tqdm` is great for monitoring the progress of iterating over loops, it's also a common desire to monitor the progress of an ML model being trained. Luckily, several types of models available in Python's scikit-learn library do allow for this. Let's walk through an example using a gradient boosting model (GBM). We'll use the customer churn dataset we're already familiar with from previous examples.

Listing 5.5. The code below trains a GBM model on the customer churn dataset.

```
from sklearn.ensemble import GradientBoostingClassifier #1
from ch4.dataset_class_final import DataSet #1

customer_obj = DataSet( #2
    feature_list = ["total_day_minutes", #2
    "total_day_calls", #2
    "number_customer_service_calls"], #2
    file_name = "data/customer_churn_data.csv", #2
    label_col = "churn", #2
    pos_category = "yes" #2
) #2

gbm_model = GradientBoostingClassifier(learning_rate = 0.1, #3
    n_estimators = 300, #3
    subsample = 0.7, #3
    min_samples_split = 40, #3
    max_depth = 3) #3

gbm_model.fit(customer_obj.\ #4
    train_features, #4
    customer_obj.\ #4
    train_labels) #4
```

The issue with the code we have above is that the model will train without giving us any progress update while training. Fortunately, there's a simple fix to this. All we need to do is to add an extra parameter called *verbose* to the *GradientBoostingClassifier* object.

Listing 5.6. This code is almost exactly the same as Listing 5.5, except now we add the *verbose = 1* argument to the *GradientBoostingClassifier* object. This will print out a progress log as the model trains.

```
gbm_model = GradientBoostingClassifier(learning_rate = 0.1, #1
                                         n_estimators = 300, #1
                                         subsample = 0.7, #1
                                         min_samples_split = 40, #1
                                         max_depth = 3, #1
                                         verbose = 1) #1

gbm_model.fit(customer_obj.\ #2
               train_features, #2
               customer_obj.train_labels) #2
```

Now, if we run the above block of code, a progress log will be generated, like in Figure 4.5. To run the full code for yourself, check the *train_gbm_model.py* script available in the ch5 directory.

Figure 5.5. Sample output from training the GBM model using the *verbose* argument set equal to 1

Iter	Train Loss	OOB Improve	Remaining Time
1	1.3205	0.0606	1.60s
2	1.2645	0.0488	1.64s
3	1.2200	0.0410	1.62s
4	1.1845	0.0365	1.59s
5	1.1509	0.0327	1.57s
6	1.1200	0.0270	1.54s
7	1.0968	0.0257	1.58s
8	1.0666	0.0190	1.86s
9	1.0447	0.0185	1.92s
10	1.0328	0.0128	2.18s
20	0.9267	0.0047	1.69s
30	0.8568	0.0011	1.45s
40	0.8433	0.0006	1.32s
50	0.8462	-0.0003	1.28s
60	0.7934	-0.0004	1.19s
70	0.7833	-0.0010	1.12s
80	0.7862	-0.0011	1.04s
90	0.8094	-0.0013	0.99s
100	0.7629	-0.0002	0.94s
200	0.6901	-0.0011	0.44s
300	0.6328	-0.0011	0.00s

This log will print out as the model is training. In this case, since we are using GBM, the log prints out as iterations in the model complete (as each tree finishes training).

The *verbose* option is available for several other model types, as well. To know if a scikit-learn model supports the *verbose* argument for progress monitoring, you can check out that model's documentation on scikit-learn's website. For example, if you wanted to monitor the progress of a random forest model, you could follow the code example in Listing 5.7.

Listing 5.7. Use the verbose option to monitor the progress of a random forest model in sklearn

```
from sklearn.ensemble import RandomForestClassifier #1

forest_model = RandomForestClassifier(verbose = 1, #2
    random_state = 0, #2
    n_estimators = 500) #2

forest_model.fit(customer_obj.\ #3
    train_features, #3
    customer_obj.train_labels) #3
```

Running the code in Listing 5.7 will generate the output in Figure 5.6:

Figure 5.6. Sample output from training the random forest model using the verbose argument

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 500 out of 500 | elapsed:      3.1s finished
```

The above examples cover monitoring the progress of training a model. However, what about hyperparameter tuning? Hyperparameter tuning is a common process in machine learning. In the next section, we'll cover how to monitor the progress of hyperparameter tuning.

5.2.1 Monitoring hyperparameter tuning

A related area in machine learning where having a progress monitor is useful involves hyperparameter tuning. Recall that tuning hyperparameters is analogous to tuning a radio to have optimal signal. Hyperparameter tuning, such as selecting the number of iterations or max depth of a tree in GBM, is used to optimize the hyperparameters for optimal model performance. This process can also take a long time, depending on the size of your dataset and

scope of the tuning involved. Python's scikit-learn library also offers the ability to monitor the progress of hyperparameter tuning. Extending on our example from above, let's use the same dataset for tuning our GBM model.

First, we will start by importing *RandomizedSearchCV*, which will randomly select combinations of parameters from the grid we create and use them to train the model. For large datasets and search grids, randomized search is much faster than a standard grid search. For our purposes here, you could potentially use either a randomized search or standard grid search (you can create a progress bar for either). In our *RandomizedSearchCV* object, we specify several components:

- The model we want to use for hyperparameter tuning (in this case, the *GradientBoostingClassifier*)
- The search grid, *parameters*
- *n_jobs* = the number of cores on our computer we want to use in parallel while running the search
- The metric we used to determine the *optimal* model (in this case, ROC-AUC)
- The number of iterations - or *combinations* of parameters we want to try out (200 in our example)
- The *random_state*, which is a typical parameter for setting the random seed
- The same *verbose* argument from above (set equal to 1 in this example)

Again, setting *verbose* = 1 in our example will print out the progress of the hyperparameter tuning.

Listing 5.8. Use the verbose argument to monitor the progress of the hyperparameter tuning process.

```
from sklearn.model_selection import RandomizedSearchCV #1
from sklearn.ensemble import GradientBoostingClassifier #1
from ch4.dataset_class_final import DataSet #1

customer_obj = DataSet( #2
    feature_list = ["total_day_minutes", #2
    "total_day_calls", #2
    "number_customer_service_calls"], #2
    file_name = "data/customer_churn_data.csv", #2
```

```

label_col = "churn", #2
pos_category = "yes" #2
) #2

parameters = {"max_depth":range(2, 8), #3
              "min_samples_leaf": range(5, 55, 5), #3
              "min_samples_split": range(10, 110, 5), #3
              "max_features": [2, 3], #3
              "n_estimators": [100, 150, 200, #3
                               250, 300, 350, 400]} #3

clf = RandomizedSearchCV( #4
    GradientBoostingClassifier(), #4
    parameters, #4
    n_jobs=4, #4
    scoring = "roc_auc", #4
    n_iter = 200, #4
    random_state = 0, #4
    verbose = 1) #4

clf.fit(train_features, #5
        train_labels) #5

```

Running the code in Listing 5.8 with our *verbose* parameter will generate the output in Figure 5.7:

Figure 5.7. This snapshot shows the output of using the *verbose* parameter when conducting hyperparameter tuning. Notice how it culminates with displaying the total amount of time taken to run the hyperparameter search.

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

```
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:   21.8s
[Parallel(n_jobs=4)]: Done 192 tasks      | elapsed:  1.3min
[Parallel(n_jobs=4)]: Done 442 tasks      | elapsed:  3.5min
[Parallel(n_jobs=4)]: Done 792 tasks      | elapsed:  6.0min
[Parallel(n_jobs=4)]: Done 1000 out of 1000 | elapsed:  9.1min finished
```

If we remove the *verbose* argument, nothing will be printed.

Thus far, we've covered:

- How to create progress bars in loops
- How to monitor model training progress
- How to monitor the progress of hyperparameter tuning

Next, let's talk about how to automatically *stop* code execution once it has been running beyond a timeout period.

5.3 How to auto-stop long-running code

What if you have code which may run for a long time, but you want to stop it in case it goes past a certain duration? This is where the *stopit* package comes in handy. The *stopit* library allows you to specify a time-out limit for code execution. This can be useful in many cases, including when you're deploying code into production or when you just want to test running your code and you're not sure whether it will take a long time (where *long* is defined by your particular use case). For example, in a production setting, you might need to fetch a model prediction in a low-latency amount of time, so you could set a timeout on how long it takes. Or, if you might want to set a limit on how long it takes to train a given model in order to better spend your time on another method or make changes to the current training process.

To start using *stopit*, you first need to install it using pip.

Listing 5.9. Run pip install stopit to setup the stopit package

```
pip install stopit #1
```

Once you've installed *stopit*, you're ready to start using it for your tasks! Let's go through a few examples.

Example 1) Set a time-out on a loop

A common example where running code might take longer than desired is when looping over a collection of values or performing a series of tasks over a loop. Let's start with a simple example of iterating over the integers between 1 and 100 million. To set a timeout limit, we will use the

ThreadingTimeout method available in *stopit*. The only parameter we need to give this method is the number of seconds that we want to set as the timeout period. In this first case, we'll use 5 seconds as the timeout. The code that we want to execute within this timeout limit will go inside of the *with* block. Once you execute the code below, any code within the *with* block will start executing, but will stop once the timeout limit of 5 seconds has been reached. If you want to set a higher limit, you just need to change 5 to another value, like 30 seconds, 60 seconds, etc.

The *context_manager* variable we created in the code snippet below stores the *state* of the code execution. We can check whether *context_manager.state* = *context_manager.EXECUTED*. If this condition holds, then our desired code block finished executing before the timeout limit. Otherwise, the code did not finish before the timeout was reached. In the second case, we could print a message to the user stating that the code did not finish, or potentially return an error message saying the same.

Listing 5.10. Use stopit to limit how much time a for loop takes to execute. This code is available in the ch5/stopit_for_loop.py file.

```
import stopit #1

with stopit.ThreadingTimeout(5) as context_manager: #2

    # sample code we want to run... #2
    for i in range(10**8): #2
        i = i * 2 #2

if context_manager.state == context_manager.EXECUTED: #3
    print("COMPLETE...") #3

elif context_manager.state ==
      context_manager.TIMED_OUT: #4

    print("DID NOT FINISH...") #4

    # or raise an error if desired #4
    raise AssertionError("DID NOT FINISH") #4
```

The essential workflow when using *stopit* is similar across many applications.

As we went through above in the first example, the general workflow goes like this:

1. Create a *with* block using the *ThreadingTimeout* method. This method takes the number of seconds you want to use as a time limit
2. Check whether the code within the *with* block completed successfully (this is in section three above)
3. If the code did not finish, alert the user (this can be printing a message or raising an error)

Putting the workflow into code looks like Listing 5.11 (similar to our first example). The two main pieces that will change depending on your scenario are the code block within the *with* block and the NUM_SECONDS value, representing the number of seconds you want to use for the time limit.

Listing 5.11. The code sample below shows a skeleton of how to use stopit to set time limits for your code.

```
import stopit #1

with stopit.\#2
    ThreadingTimeout(NUM_SECONDS) as context_manager: #2
        [CODE BLOCK HERE] #2

if context_manager.state == context_manager.EXECUTED: #3
    print("COMPLETE...") #3

elif context_manager.state ==
      context_manager.TIMED_OUT: #4
    print("DID NOT FINISH...") #4

# or raise an error if desired #4
raise AssertionError("DID NOT FINISH") #4
```

Next, let's set a timeout for reading data from a database!

Example 2) Set a time-out on reading data from a database

Now, what if we change our above example to read in data from a database? This could be useful if we want our code to be able to read in data within a

given timeframe. Again, this could be useful in both production and non-production settings, such as enforcing a timeout on reading in data needed for model predictions in real-time application, or just wanting a limit on how long it takes to get data in case there are issues with the database. Let's suppose our timeout limit in this case will be 5 minutes.

Listing 5.12. Using a timeout limit to read data from a database can be done similarly to our above example. Essentially, we're changing the contents of the code going into our with statement block to reflect pulling data from a database table. Check the code out for yourself in the ch5/stopit_read_from_database.py file.

```
import stopit #1
import pandas as pd #1
import pyodbc #1

conn = pyodbc.connect("DRIVER={SQLite3 ODBC Driver}; #2
                      SERVER=localhost; #2
                      DATABASE=customers.db;Trusted_connection=yes") #2

with stopit.ThreadingTimeout(300) as context_manager: #3

    customer_key_features = pd.read_sql(
        conn, #3
        """SELECT churn, #3
               total_day_charge, #3
               total_intl_minutes, #3
               number_customer_service_calls #3
          FROM customer_data"""") #3

# Did code finish running in #4
# under 300 seconds (5 minutes)? #4
if context_manager.state ==
    context_manager.EXECUTED: #4
    print("FINISHED READING DATA...") #4

# Did code timeout? #5
elif context_manager.state ==
    context_manager.TIMED_OUT: #5
    raise Exception("""DID NOT FINISH
                     READING DATA WITHIN TIME LIMIT""") #5
```

Next, let's cover an example of using a time-out when training a machine learning model.

Example 3) Set a time-out on training a machine learning model

In the below example, we use *stopit* to set a timeout limit for training a random forest model. The principle here is the same as the above examples. Effectively, we just need to replace the code within the same *with* block with the new code to train the random forest model.

Listing 5.13. Using a timeout limit when training a machine learning model. Run the code for yourself using the ch5/stopit_train_model.py file.

```
import stopit #1
import pandas as pd #1
from sklearn.ensemble import RandomForestClassifier #1
from sklearn.model_selection import train_test_split #1

with stopit.ThreadingTimeout(180) as context_manager: #2

    forest_model = RandomForestClassifier( #2
        n_estimators = 500).\
        fit(customer_obj.train_features, #2
            customer_obj.train_labels) #2

    # Did code finish running in
    # under 180 seconds (3 minutes)? #3
    if context_manager.state == context_manager.EXECUTED: #3
        print("FINISHED TRAINING MODEL...") #3

    # Did code timeout? #4
    elif context_manager.state == context_manager.TIMED_OUT: #4

        # or raise an error if desired #4
        raise AssertionError("""DID NOT FINISH MODEL
TRAINING WITHIN TIME LIMIT""") #4
```

Let's go through one more example with *stopit*. This time, we're going to show a way of setting a timeout without using a *with* block.

Example 4) Using decorators

All of the examples above place the code we want to set a timeout limit for the inside of a *with statement* block. However, it's also possible to set a timeout limit when defining a function. The key to doing this is to using an

advanced concept called a *decorator*. *Decorators* can be a confusing and fairly extensive topic, so we're just going to provide a highlighted view of them here. Essentially, think of a decorator as a statement written on the line above a function definition that modifies the inputs to the function. That sounds...abstract. Alright, then - let's do an example using the *stopit* package. Then, we'll explain why using a decorator with stopit can be beneficial.

First, we need to import the *threading_timeoutable* method, which we'll import simply as *timeoutable*. Then, we add this method as a *decorator* to a function that trains the random forest model on the training dataset (similar to above except now we're just wrapping the model training code inside a function). The decorator is specified by the @ symbol in the line immediately above the function definition. As mentioned above, decorators modify the inputs of a function. In this case, our function - *train_model*, takes a single input, the training dataset. However, adding this decorator allows us to call the same function with two inputs. The first input will be the number of seconds we want to set as a timeout limit, while the second parameter is the same input as the original function (the training dataset). Below, we set a timeout limit of 300 seconds, similar to the earlier example.

Listing 5.14. This listing's code uses a timeout limit whenever the *train_model* function is called. Try the code out for yourself in the ch5/stopit_with_decorator.py file.

```
from stopit import threading_timeoutable as timeoutable #1
from sklearn.ensemble import RandomForestClassifier #1
from ch4.dataset_class_final import DataSet #1

@timeoutable() #2
def train_model(features, labels): #2

    forest_model = RandomForestClassifier( #2
        n_estimators = 500).\#2
        fit(features, labels) #2

    return forest_model #2

customer_obj = DataSet( #3
    feature_list = [ #3
        "total_day_minutes", #3
        "total_day_calls", #3
```

```

"number_customer_service_calls"], #3
file_name = "data/customer_churn_data.csv", #3
label_col = "churn", #3
pos_category = "yes" #3
)

forest_model = train_model(timeout = 180, #4
    features = train_features, #4
    labels = train_labels) #4

# Did code finish running in under
# 180 seconds (3 minutes)?
if forest_model: #5
    print("FINISHED TRAINING MODEL...") #5

# Did code timeout?
else: #6
    raise AssertionError("""DID NOT FINISH
        MODEL TRAINING WITHIN TIME LIMIT""") #6

# [Additional code block...] #7

```

If the model doesn't finish executing before the timeout limit, then *model* will be set to Python's *None* value because the function never returns the trained model object. Thus, we can raise an error when this occurs, stating that the model did not complete training before the time limit was up (like below).

```
AssertionError: DID NOT FINISH MODEL TRAINING WITHIN TIME LIMIT
```

A key reason why using a decorator instead of the *with* block approach is beneficial is this way provides you with more flexibility and portability in your code. Using the decorator approach allows you to easily add an additional parameter to any function you create in order to time its execution. These are useful benefits from a software engineering perspective because they make your code more generalizable. However, a potential downside could be if your codebase doesn't already make use of decorators, this approach could become less clear for those less familiar vs. just using a *with* block.

Now, let's summarize the *stopit* workflow using decorators:

1. Import the `threading_timeoutable` method from *stopit*

2. Add the method to whatever function you want to modify to have a time limit
3. Check if the function returns a value. If it doesn't, then the timeout limit was reached

Again, we can view this summary in terms of code like Listing 5.15 (similar to our above example). To use *stopit* decorators in your code, you should be able to follow a similar code flow to the skeleton below. The main differences will be the function definition and the value of NUM_SECONDS for the timeout limit.

Listing 5.15. Using a timeout limit whenever a specific function is called

```
from stopit import threading_timeoutable as timeoutable #1

@timeoutable() #2
def sample_function(parameter1, #3
                   parameter2, #3
                   parameter3, #3
                   ...): #3

    [CODE BLOCK HERE]

model = sample_function( #4
                        timeout = NUM_SECONDS, #4
                        parameter1, #4
                        parameter2, #4
                        parameter3, #4
                        ... ) #4

if not model: #5
    raise AssertionError("Timeout error") #5
```

That covers the *stopit* package! Now, let's recap what we covered this chapter.

5.4 Summary

In this chapter we covered monitoring the progress of loops, model training, and auto-stopping code based on a pre-defined timer.

- *tqdm* is a Python package that lets you generate progress bars to monitor code execution.
- The *tqdm* library is useful in monitoring the progress of loops, such as for loops or list comprehensions.
- Use scikit-learn's verbose argument to monitor the progress of model training, such as monitoring the progress of a random forest model.
- The *stopit* library can be used to set timeout limits for your code execution.
- Decorators allow you to alter a function without directly modifying its code. Use *stopit*'s *threading_timeoutable* method to set a timeout limit for any function's execution.

5.5 Practice on your own

1. Download the Spotify hit zip file from Kaggle (<https://www.kaggle.com/datasets/theoverman/the-spotify-hit-predictor-dataset>). This file contains a collection of datasets across multiple song-release decades (90s, 00s, 10s, etc). Use *tqdm* to monitor the progress of reading each in as part of a loop.
2. Using the Spotify dataset file *dataset-of-00s.csv* (from the zip file you downloaded in step 1), can you monitor the progress of tuning the hyperparameters for a random forest model? The label of the dataset is in the *target* column.
3. Can you set a timeout of 5 minutes for the hyperparameter tuning process using a *with* block?
4. Can you repeat #3 using a decorator instead?

6 Making your code faster and more efficient

This chapter covers

- What is scaling
- How to understand why your code is running slowly
- How to make code faster with parallelization, including vectorization and multiprocessing
- What is caching and how can it help you improve computational efficiency
- Making use of Python's data structures to optimize your code

Scalability of a codebase is essentially the ability of the code to handle large amounts of data or requests. For example, scaling a web application means being able to handle a large amount of user traffic. This might mean going from hundreds of users to millions (or even billions) of users.

In data science, scaling is often important as well. For data science, scaling usually refers to one of two main components:

1. Optimizing code to run faster in order to handle a large number of operations in a shorter period of time
2. Handling large datasets, including cleaning, feature engineering, and building models when your dataset may not fit in memory (or consumes a high amount of existing memory)

This chapter will focus on the first of these points, while the second point will be delved into during the next chapter. Optimizing code to run faster becomes more and more important as the data size grows. There are many possibilities where this issue involving data size arises, including:

- Writing code to scrape text from a large collection of raw files (for example, image files)

- Feature engineering – creating a new column (or collection of columns) in a large dataset based on other column values. Additionally, data analysis of large datasets, such as generating a correlation matrix, can take much longer with a sizable dataset than with a small one
- Fetching predictions from a trained ML model. Though computing predictions from an ML model is generally much faster than training a model, being able to fetch predictions in very low latency can be crucial to the user’s experience (think what would happen if YouTube’s recommended video list took 20 seconds to load, for instance)
- Downloading data or files from a large number of URLs

For a specific example, let’s consider the *artists* dataset, which is available at this URL: <https://www.kaggle.com/datasets/yamaerenay/spotify-dataset-19212020-600k-tracks>. This dataset contains several features related to artists, such as follower counts. Suppose we want to do an analysis on the data to figure out which artists are in the 99.9%-tile for follower counts. Let’s break down the computational process for tackling this task.

- Calculate the 99.9%-tile of the follower counts
- Iterate over each artist, comparing an artist’s follower count to the 99.9%-tile threshold.
- If the follower count is above the threshold, output the artist name (for example, print the artist name or append the name to the list of high-follower artists)

There are several ways to approach this problem programmatically, but they vary widely in terms of computational speed. In the next section of this chapter, we will walk through three methods to tackle this example, starting with the computationally slowest approach.

Since this chapter is focused primarily on computational efficiency, we will use this example (and others throughout this chapter) to show how to find the slow-running points in your code.

Before we do that, however, let’s briefly discuss computational efficiency in a computer science context. From a computer science viewpoint, computational efficiency is often expressed in *big O* notation. For example, the expression $O(N)$ means that an expression requires N operations.

Considering the artists dataset - if we already know the 99.9%-tile follower count, then, it requires N iterations to check whether a given artist has a follower count greater than this threshold where N is equal to the number of artists. In *big O* notation, the computational efficiency (also called *time complexity*) is $O(n)$. To compute the 99.9%-tile requires sorting the follower counts, which has a time complexity of $O(n * \log(n))$ (it's beyond the scope of this book to prove this, but suffice to say the most optimal sorting algorithms are $O(n * \log(n))$). Thinking of your code in terms of time complexity can be helpful when identifying points in your code that are inefficient or slow-performing. Later in this chapter, we'll delve into a Python package that can help find these slow-performing points in your code automatically.

NOTE: Throughout this chapter, we will give sample runtimes for examples we walk through. One additional point to keep in mind throughout this chapter is that whenever a time taken to complete is given for a piece of code, keep in mind that the exact runtime will vary between different machines, and you may get slightly different runtimes when you run the code for yourself. The examples in this chapter are mostly run on a Mac with 8GB of RAM and 4 CPU cores.

6.1 Slow code walk through

In the first section of this chapter, we will walk through an example of a piece of code that takes quite a while to run. This code is **not** optimized. We will go through how to figure out the slow points of the code, and to improve its runtime efficiency (in other words, make the code run faster). For this example, we'll be using a new dataset called *artists.csv*, which is available in the book repo's data directory. You can also access the dataset from this URL: <https://www.kaggle.com/datasets/yamaerenay spotify-dataset-19212020-600k-tracks>.

6.1.1 Don't repeat yourself (DRY)

In a previous chapter, we discussed how it is important to avoid repeating yourself in code. For example, if you're writing similar code over and over again, it's probably beneficial to look into using a function or potentially a class. This principle also applies *computationally*. For instance, if you're

using a loop, you should make sure that you are not repeating the same exact computations over and over again. Let's look at an example.

In the code in Listing 6.1, we make use of the artists dataset. After reading in this dataset, we loop through the rows of the data and print out each artist name that has a follower count above the 99.9%-tile. Running the code in Listing 6.1 times out after 60 seconds (due to using the stopit package). After 60 seconds, only around 2000 iterations are completed out of over 1.1 million! This is incredibly slow, and it would take quite a bit of time to finish if we allowed the code to continue.

Listing 6.1. The code in this listing is available in the dont_repeat_yourself_bad_version.py file.

```
import pandas as pd #1
from tqdm import tqdm
import stopit

artists = pd.read_csv("data/artists.csv") #2

with stopit.ThreadingTimeout(60) as context_manager: #3

    for index in tqdm(range(artists.shape[0])):

        if artists.followers[index] >
            artists.followers.quantile(.999):
            print(artists.name[index])

    if context_manager.state ==
        context_manager.EXECUTED: #4
        print("FINISHED...")

elif context_manager.state ==
    context_manager.TIMED_OUT: #5

    print("DID NOT FINISH WITHIN TIME LIMIT")
```

Next, let's discuss how to use a profiling tool to analyze your code for computational efficiency. This will help us to speed up the example in Listing 6.1.

6.1.2 Line profiler

One of the main ways to improve the efficiency of your code is to understand what lines or sections of your code are causing your code to run slower. Luckily, there are several Python packages available for this. Let's walk through a popular one called *line-profiler* (see its documentation here: https://github.com/pyutils/line_profiler). You can install this library using pip (see Listing 6.2).

Listing 6.2. Install the line-profiler package

```
pip install line-profiler #1
```

line-profiler will calculate the compute time for each line of code in a function. Let's use line-profiler on the *dont_repeat_yourself_function_bad_version.py* file (available in the ch6 directory).

Listing 6.3. The code here is similar to the code in Listing 6.1, except we wrap the code to get the artists with high follower counts in a function and add a decorator. This decorator, named `profile`, will be accessible if we analyze the code via a command in the terminal (provided you've downloaded line-profiler). The decorator will allow line-profiler to analyze the speed of your code.

```
import pandas as pd #1
from tqdm import tqdm #1
import stopit #1

artists = pd.read_csv("data/artists.csv") #2

# add a profile decorator so we can use the
# line-profiler package to analyze
# the computational speed
@profile #3
def get_artists_with_high_followers( #4
    artists = artists): #4

    # wrap the previous code we wrote in a function #4
    # Set a 60-second timeout
    with stopit.ThreadingTimeout(60) as context_manager: #4

        # Loop through each artist
        for index in tqdm(range(artists.shape[0])): #4
```

```

# Check whether the artist's #4
# follower count is #4
# greater than the 99%-tile. #4
# This calculates the #4
# 99.9%-tile in each iteration! #4
if artists.followers[index] >
    artists.followers.quantile(.999):
    print(artists.name[index])

if context_manager.state ==
    context_manager.EXECUTED:
    print("FINISHED...")

elif context_manager.state ==
    context_manager.TIMED_OUT:

    print("DID NOT FINISH WITHIN TIME LIMIT")

get_artists_with_high_followers(artists = artists) #5

```

line-profiler can be run from the command line (terminal), as can be seen in Listing 6.4. To do this, we use the *kernprof* command, followed by the name of the input Python file. The *kernprof* command should be available after installation. Additionally, we need to add *@profile* to the top of the function that we want to profile (as we did in Listing 6.3).

Listing 6.4. Using the *kernprof* command, we generate a binary file containing the compute profile of the input script - *dont_repeat_yourself_function_bad_version.py*.

```
#1
kernprof -l ch6/dont_repeat_yourself_
function_bad_version.py
```

After the binary file is generated, you can print a more easily viewable version using the command in Listing 6.5:

Listing 6.5. Running this command will print a readable output to the terminal summarizing time spent by line.

```
#1
python -m
    line_profiler
    dont_repeat_yourself_function_bad_version.py.lprof
```

The output of running the command in Listing 6.5 is shown below.

Line #	Hits	Time	Per Hit	% Time
<hr/>				
9				
10				
11				
12	1	256.0	256.0	0.0
13				
14	1949	332082.0	170.4	0.6
15				
16	1948	59637641.0	30614.8	99.4
17				
18				
19				
20	1	4.0	4.0	0.0
21				
22				
23	1	1.0	1.0	0.0
24				
25	1	55.0	55.0	0.0
<hr/>				
Line #	Line Contents			
9				
10	def get_artists_with_high_followers(artists = artists			
11	with stopit.ThreadingTimeout(60) as context_manager			
12				
13	for index in tqdm(range(artists.shape[0])):			
14				
15	if artists.followers[index] > artists.followers.quant			
16	print(artists.name[index])			
17				
18				
19				
20	if context_manager.state == context_manager.EXECUTED:			
21	print("FINISHED...")			
22				
23	elif context_manager.state == context_manager.TIMED_O			
24				
25	print("DID NOT FINISH WITHIN TIME LIMIT")			

Based on the output, we can see:

- Hits = Number of times this line was run

- Time = How much time (in microseconds) spent on each line in total (across all executions of that line)
- Per Hit = Amount of time (again, in microseconds) per hit (on average per each time the line was executed)
- % Time = The percentage of time consumed by the line of code across all executions of the line
- Line Contents = The associated line of code

For our example, 99.4% of the time spent executing the function is on the line calculating the 99.9%-tile value (note: the exact percentage will vary between different machines)! This is an immediate giveaway that this line of code needs to be optimized if possible. Let's make an attempt to speed up our code example in Listing 6.6.

In contrast to Listing 6.1, Listing 6.6 achieves the same original goal, but completes in around 11 seconds! The only change we make is to calculate the 99.9%-tile just once and store this value as a variable called *high_followers*. Then, we just reference this variable within the for loop.

Listing 6.6. The code in this listing is available in the `dont_repeat_your_self_better_version.py` file from the `ch6` directory.

```
import pandas as pd #1
from tqdm import tqdm
import stopit

artists = pd.read_csv("data/artists.csv") #2

with stopit.ThreadingTimeout(60) as context_manager: #3

    high_followers = artists.followers.quantile(.999)
    for index in tqdm(range(artists.shape[0])):

        if artists.followers[index] > high_followers:
            print(artists.name[index])

if context_manager.state ==
    context_manager.EXECUTED: #4
print("FINISHED...")
```

```
elif context_manager.state ==  
    context_manager.TIMED_OUT: #5  
  
    print("DID NOT FINISH WITHIN TIME LIMIT")
```

This code is **still not** optimal. In fact we can do quite a bit better. Next, we'll discuss how to further improve the speed of this code.

6.1.3 Reducing loops

Let's continue the earlier example of identifying artists with high follower counts. As mentioned, we can still do better than the second improved version.

The code in Listing 6.7 completes our task in ~0.21 seconds (this is printed in Line 7), which is still much faster than Listing 6.6, and significantly faster than the first version we tried in Listing 6.1 (or Listing 6.3). In this case, we still use a loop to print out the name of each high-follower-count artist, but avoid using a loop to get the collection of such artists. We get this collection of artists using pandas to directly filter the source dataset. Then, we cycle through each of the artists, printing out their names. Alternatively, we could also avoid printing out the names, and just keep the result as a pandas Series or list. This would be even faster than printing out each name, and would avoid using a loop all together.

Listing 6.7. In this listing, we improve upon the previous versions of this code (Listings 6.1 / 6.3 and 6.6). This version also calculates the 99.9%-tile of the follower counts just once, similar to Listing 6.6. Then, we use pandas to filter the artists dataset based on this value, rather than looping through each artist sequentially (resulting in a much faster code execution for the same task).

```
import pandas as pd #1  
import stopit  
import time  
  
artists = pd.read_csv("data/artists.csv") #2  
  
def get_artists_with_high_followers( #3  
    artists): #3  
  
    with stopit.ThreadingTimeout(60) as context_manager:
```

```

top_percentile = artists.\
    followers.\
    quantile(.999)

names = artists[
    artists.\
    followers > top_percentile].\
    name

for name in names:
    print(name)

if context_manager.state == 
    context_manager.EXECUTED: #4
    print("FINISHED...")

elif context_manager.state == 
    context_manager.TIMED_OUT: #5

    print("""DID NOT FINISH
WITHIN TIME LIMIT""")

start = time.time()
get_artists_with_high_followers( #6
    artists = artists) #6
end = time.time()
print(end - start) #7

```

Listing 6.7 is much faster than using a for loop in part because pandas data frames use NumPy arrays under the hood to store columns. We'll discuss more about NumPy arrays later during this chapter.

For now, let's delve into parallelization and how it can help us greatly speed up a variety of different tasks.

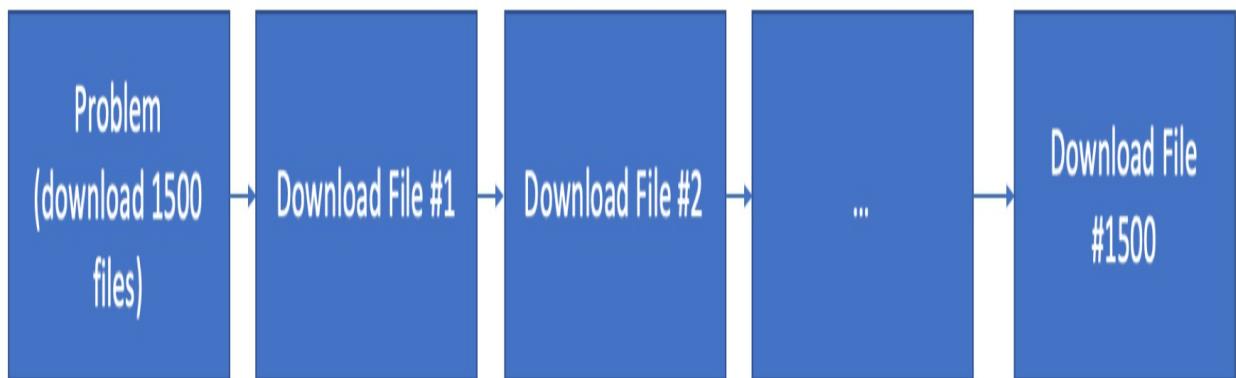
6.2 Parallelization

Another way to improve the computational efficiency of your code is to use parallelization. *Parallelization* essentially means to break a problem into sub-problems that can be executed simultaneously. There are many examples where parallelization can be useful:

- Downloading a collection of files
- Training ML models. A random forest, for example, is readily parallelizable because each tree in the forest is independent of each other tree. This means multiple trees can be constructed simultaneously. Additionally, using a grid search to find optimal hyperparameters is another example where parallelization could greatly speed up the process.
- Scraping text from a set of files
- Performing operations across entire columns in a dataset at once. This can be useful, for instance, when you are creating new features in a dataset based on existing variables.

Let's contrast using parallelization versus *not* using parallelization. Taking the first example of downloading a collection of files - imagine we want to download 1500 files. If you don't use parallelization, your code would download each file in a sequence, one at a time. An individual file in the sequence cannot be downloaded until the files prior to it in the sequence have already been downloaded. A visualization of this idea is in Figure 6.1.

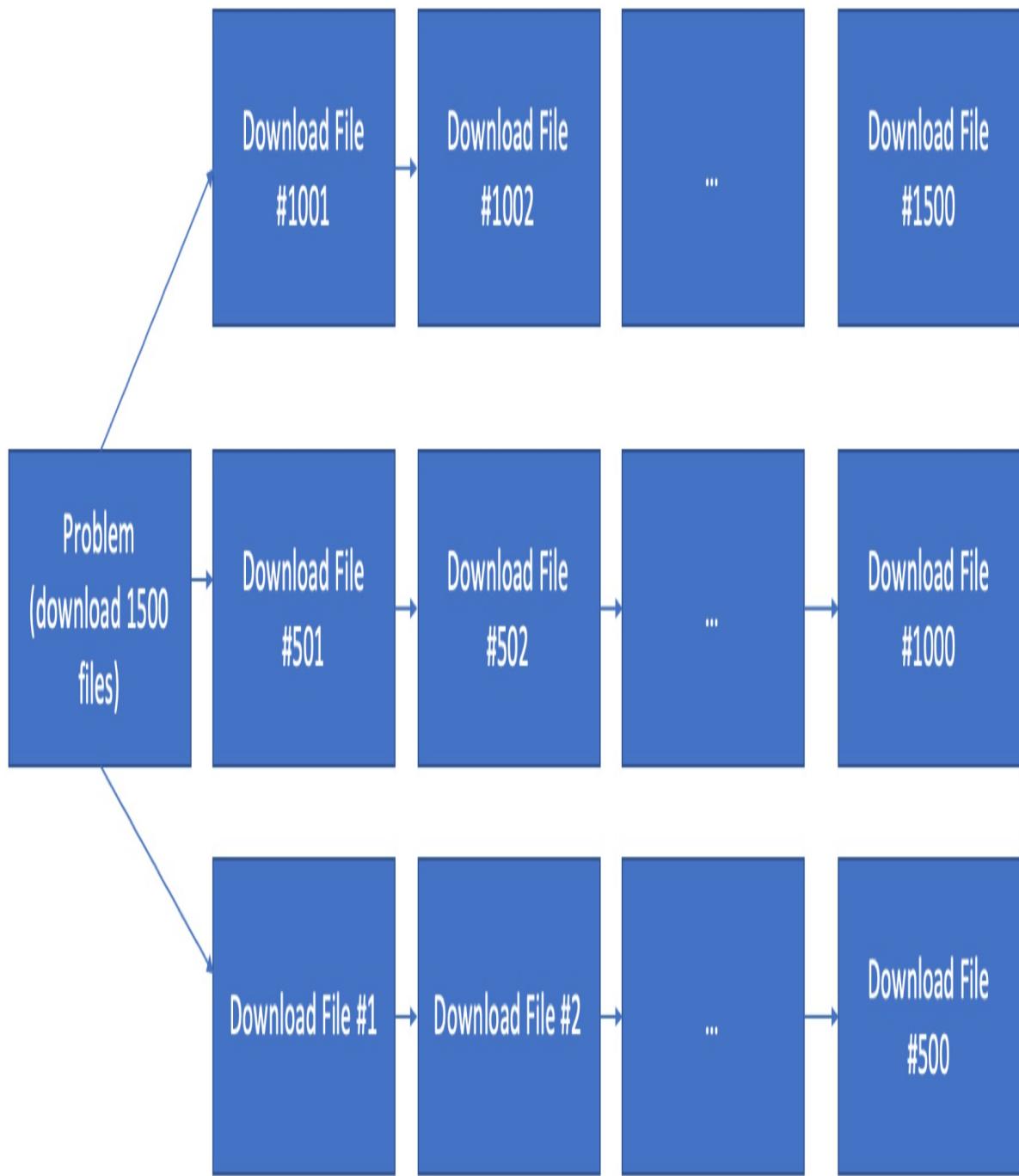
Figure 6.1. This is an example of downloading a collection of files without using parallelization. In this setting, you would sequentially download each file one at a time.



In contrast, using parallelization allows you to split this task into multiple download operations occurring simultaneously. Figure 6.2 shows an example of what this would look like.

Figure 6.2. This view shows an example of a parallelization workflow for downloading the same

example 1500 files. Here, we can split up the downloads into separate workflows that occur at the same time. The benefit is that the original task of getting all of the files downloaded occurs much faster because multiple file downloads occur simultaneously in parallel.



Parallelization comes in several primary forms:

- Vectorization
 - Vectorization involves operating on a collection of values at once, rather than processing values one element at a time. We'll walk through an example further below.
- Multiprocessing
 - A *process* is an instance of an application or program. *Multiprocessing* refers to running multiple instances of the same application or program. In the case of Python, this means running multiple Python instances to accomplish a given task faster.
- Multithreading
 - A *thread* is a sub-unit within a process. *Multithreading* means having multiple threads operating at the same time, while sharing resources from the same process.
- Asynchronous programming
 - Asynchronous programming involves initiating events that run separately (in the background) apart from the primary application.

We will tackle each of these in turn in the next few sections. First, let's start with vectorization.

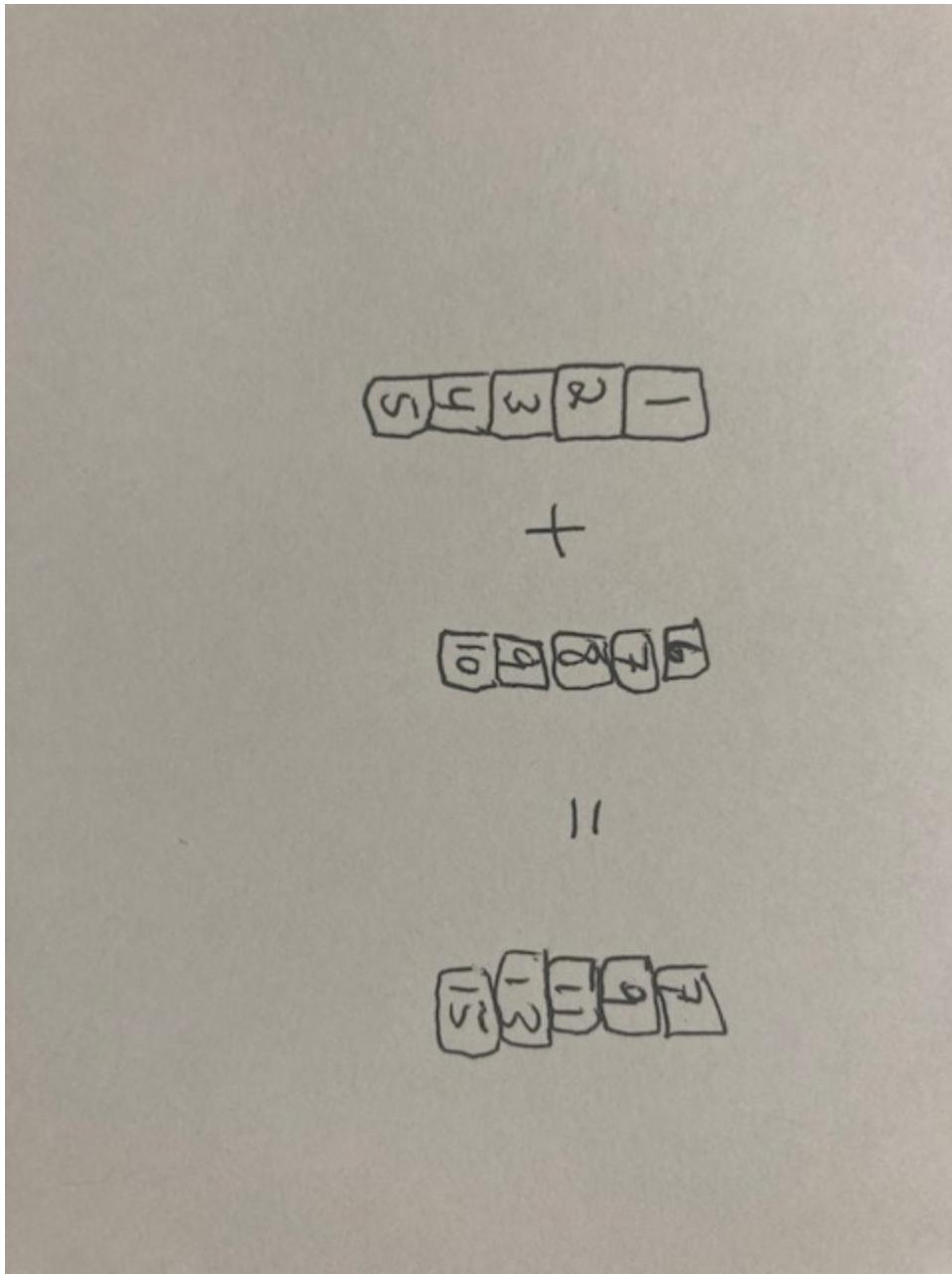
6.2.1 Vectorization

Vectorization is type of programming where operations are applied across entire arrays at once, rather than individually one element at a time. This allows for a type of parallelization, which can often speed up your code.

A diagram showing vectorization at work is shown in Figure 6.3. In this diagram, we have two arrays - one storing the integers 1 - 5, and the second one storing the integers 6 - 10. With vectorization, we can apply

mathematical operators, like addition, and pairwise add the two arrays in a single step. This avoids the use of a loop, and results in much faster code for large collections of values.

Figure 6.3. Example of using vectorization to pairwise add two arrays



A simple example of vectorization can be shown using Python's NumPy package.

For example, in NumPy we can create two arrays and then perform operations like addition, multiplication, etc. An example of creating NumPy arrays is shown in Listing 6.8.

Listing 6.8. The code here creates two NumPy arrays, which are vectorizable.

```
import numpy as np #1  
  
sample_nums1 = np.arange(1, 6) #2  
sample_nums2 = np.arange(6, 11) #2
```

Next, we can add, multiply, or subtract these arrays (see Listing 6.9). Notice how we don't need to loop through each element of the two arrays, but instead we can use a single "+", "*", or "-" operator to perform the operation we want between the arrays. This operation is then performed pairwise between the arrays. *Pairwise* operations between two arrays mean that the resultant array will have the same dimensions as the two arrays and the same operation is applied between the first elements of each array, between the second elements of each array, between the third elements of each array, and so on. See Listing 6.9 and the resulting output for performing pairwise addition.

Listing 6.9. This code snippets here show examples of how to perform math operations between two NumPy arrays using vectorization.

```
sample_nums1 + sample_nums2 #1  
  
sample_nums1 * sample_nums2 #2  
  
sample_nums1 - sample_nums2 #3
```

For example, when adding the two arrays, the result would look like this:

```
array([ 7,  9, 11, 13, 15])
```

Before we delve into a more advanced usage of vectorization, let's start with another example of sub-optimal code and then show we can optimize it using vectorization.

In the sub-optimal example, we'll be using the apply method from pandas.

Avoiding pandas apply

The `apply` method in pandas is commonly used. However, it can be quite slow for medium to larger datasets. The `apply` method works by executing a function to every row in a data frame. It executes each iteration sequentially, and each function call requires several steps under the hood, such as storing local variables within each function call, executing the function, and performing cleanup (for instance, removing memory allocation for variables defined within the function). This causes `apply` to perform slowly when executed across a high number of observations. Additionally, the `apply` method has extra overhead under the hood that makes processing slower - often slower than using a list comprehension, for instance.

Running the code in listing 6.10 takes ~27 seconds (this time may be different from your computer) to create the `high_followers_has_genre` field. That's quite a bit of time just to create a single field.

Listing 6.10. This code uses the pandas `apply` function to create a new column called `high_followers_has_genre`.

```
import pandas as pd #1
import time #1

artists = pd.read_csv("data/artists.csv") #2

start = time.time()
artists["high_followers_has_genre"] = artists.\ #3
    apply(lambda df: 1 if df.followers > 10
        and len(df.genres) > 0 else 0,
        axis = 1)
end = time.time()
print(end - start) #4
```

Using a list comprehension instead

In contrast, we can create the field in ~0.781 seconds using a list comprehension (see Listing 6.11). In this case, we loop over the `followers` and `genres` fields simultaneously (using Python's `zip` function) and check the needed criteria.

Listing 6.11. The code in this listing also creates the `high_followers_has_genre` field, but does so

using a list comprehension. This results in a much faster computation than the previous version using the apply method.

```
import pandas as pd #1
import time #1

artists = pd.read_csv("data/artists.csv") #2

artists["high_followers_has_genre"] = [ #3
    1 if followers > 10
    and len(genres) > 0
    else 0
    for followers, genres in
    zip(artists.followers,
        artists.genres)]
```

Next, let's further improve the speed of calculating *high_followers_has_genre* using vectorization!

Numpy's vectorize method

Another way to improve the speed is to use numpy's *vectorize* method. This method converts a function to a vectorizable function so that it can be applied across an entire array (or data frame column) at once.

Listing 6.12. Instead of using apply, we can use NumPy's vectorize method to vectorize a function that creates the new column for artists having a follower count greater than 10 and also having genres associated with them in the dataset.

```
import pandas as pd #1
import time
import numpy as np

artists = pd.read_csv("data/artists.csv") #2

def create_follower_genre_feature(
    followers, genres): #3
    return 1 if followers > 10 and
        len(genres) > 0 else 0

vec_create_follower_genre_feature = np.\ #4
    vectorize(create_follower_genre_feature) #4
```

```

col = "high_followers_has_genre" #5
artists[col] = vec_create_follower_genre_feature(
    artists.followers,
    artists.genres)

```

Table 6.1 summarizes the runtimes for each method by varying the number of rows we read in from the artists dataset. As you can see in the table, the difference between the methods becomes more and more noticeable as the number of rows grows.

Table 6.1. This table summarizes the runtimes of each of the methods discussed in this section to create the high_followers_has_genre column. We also test the runtime by varying number of rows for reference.

Method	Number of rows	Time (seconds)
Apply method	1000	0.03
Apply method	10,000	0.23
Apply method	100,000	1.74
Apply method	1,000,000	17.88
Apply method	Full dataset (1.16M rows)	22.17
List comprehension	1000	0.002
List comprehension	10,000	0.008

List comprehension	100,000	0.069
List comprehension	1,000,000	0.582
List comprehension	Full dataset (1.16M rows)	0.740
Vectorization	1000	0.002
Vectorization	10000	0.004
Vectorization	100,000	0.029
Vectorization	1,000,000	0.249
Vectorization	Full dataset (1.16M rows)	0.301

That's all for vectorization. Let's continue our parallelization discussion by delving into multiprocessing next.

6.2.2 Multiprocessing

As mentioned at the start of the Parallelization section, multiprocessing involves launching multiple instances of Python to tackle a task simultaneously.

One of the most common Python packages for multiprocessing is the *multiprocessing* library. Before you get started setting up a parallelized job,

it's a good idea to check how many CPUs are available on your computer (or server). You can do this using `multiprocessing`'s `cpu_count` method, like in Listing 6.13. The number of processes you run in parallel is limited by the number of CPUs you have available.

Listing 6.13. To check the number of CPUs the computer/server you're using has, just import the `cpu_count` function from `multiprocessing`. Calling this function will tell you the number of CPUs you have.

```
from multiprocessing import cpu_count #1  
cpu_count() #2
```

For the next example, we'll be scraping text from a collection of images (this process is known as OCR, or Optical Character Recognition). We'll be using a dataset available here: <https://expressexpense.com/blog/free-receipt-images-ocr-machine-learning-dataset/>.

Before we use `multiprocessing` to parallelize this task, let's tackle this *without* using parallelization. To scrape the text, we will use the `pytesseract` package. This is a popular Python library for scraping text from images. To install this package, you'll need to follow the steps at this link: <https://pypi.org/project/pytesseract/>. This includes installing the library using pip, as you can see in Listing 6.14 (you'll also need to separately install Tesseract, as referenced in the previous link).

Listing 6.14. Use pip to install pytesseract

```
pip install pytesseract #1
```

Now that we have `pytesseract` installed, we can write the code in Listing 6.15 to scrape the text from the collection of images in our example. This takes ~185 seconds to run.

Listing 6.15. This code uses `pytesseract` to scrape text from a collection of images.

```
import cv2 #1  
import pytesseract  
import pathlib  
import time
```

```

path = pathlib.\
    Path("data/large-receipt-image-dataset-SRD") #2
files = list(path.rglob("*"))
files = [str(file) for file in files
        if ".jpg" in file.name]

def scrape_text(file): #3

    image = cv2.imread(file)

    return pytesseract.image_to_string(image)

start = time.time()
all_text = [scrape_text(file) for file in files] #4
end = time.time()
print(end - start)

```

Now, let's use the multiprocessing library to parallelize the OCR task. In Listing 6.16, we start by replicating the code in Listing 6.12. Instead, of using a single process to handle the OCR task, we use the Pool class available in multiprocessing to setup a pool of Python processes. The input number three into the Pool class means we will create a pool of three Python processes. Next, we use *pool.map* to apply the *scrape_text* function to each element in the files list. Lastly, we shut down the Python processes by running *pool.close()*. If you're running on certain operating systems, like Windows, you'll need to add the *if name == "main":* line. To make your code more easily portable between different operating systems, you should add this line and your program should work correctly regardless of the operating system someone is using to execute the code. In a previous chapter, we mentioned that this line is used so that any code within this block will not be run if the module is imported by another module. However, setting up a Pool object will import the module within which that object is defined. The result of this is to recursively keep creating new processes - that is, unless we use the *_if name == "main"* block to make sure that only the set number of processes (in this case, three) are created. Even if you're not on Windows, it's still a good idea to use this structure as it allows for better generalizability across different operating systems (including Linux, for instance).

Running the code in Listing 6.16 takes ~91 seconds, which is less than half the time it takes to achieve the same result in Listing 6.15 (we saved ~94 seconds). Consider we are only looping over 200 image files. If we were looping over thousands or millions, the time difference between using multiprocessing vs. not would become very large.

Listing 6.16. Similar to Listing 6.15, this code also uses pytesseract to scrape text from a collection of images. In this example, however, we use Python's multiprocessing package to parallelize the text scraping, resulting in the code running in roughly half the time!

```
import cv2 #1
import pytesseract
import pathlib
from multiprocessing import Pool
import time

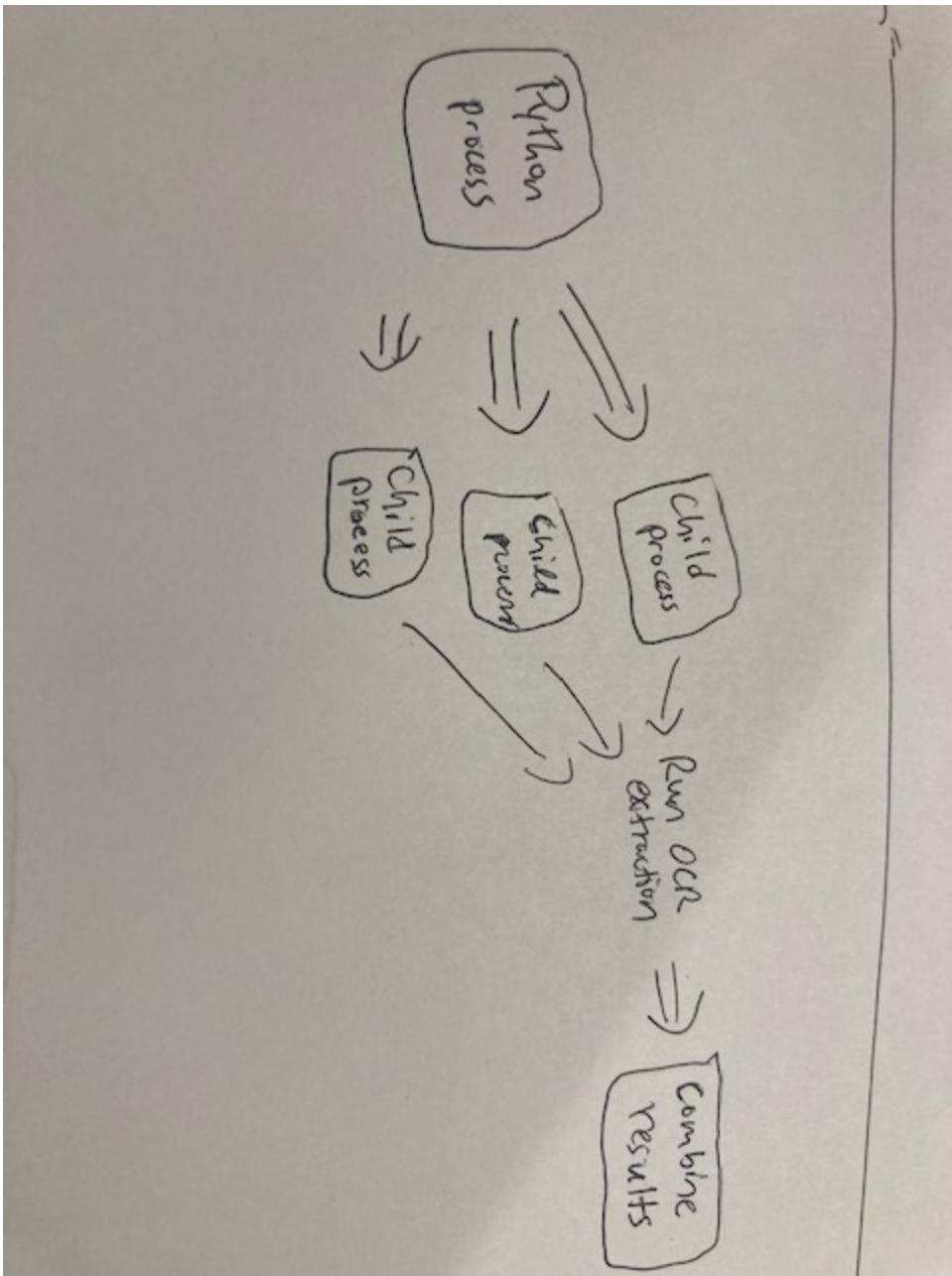
filename = "data/large-receipt-image-dataset-SRD"
path = pathlib.Path(filename) #2
files = list(path.rglob("*"))
files = [str(file) for file in files if ".jpg" in file.name]

def scrape_text(file): #3
    image = cv2.imread(file)
    return pytesseract.image_to_string(image)

if __name__ == "__main__": #4
    start = time.time()
    cores_pool = Pool(3) #5
    cores_pool.map(scrape_text, files) #6
    cores_pool.close() #7
    end = time.time()
    print(end - start) #8
```

The diagram in Figure 6.4 summarizes how multiprocessing works with the Pool.map method.

Figure 6.4. This diagram visualizes how Python's multiprocessing package creates separate Python processes to tackle the OCR task, followed by combining the results from each individual process. The result of doing this for the OCR task is much faster than using just a single process.



Now that we've covered a simpler case of scraping data in parallel, let's discuss how we can train ML models with parallelization!

6.2.3 Training ML models with parallelization

Parallelization can also be used when training an ML model. For example, using the artists dataset, we can build a model to predict an artist's popularity score based on the number of followers and genres associated with the artist.

First, let's train a random forest model using just a single process. After that, we'll show the speed difference using parallelization.

Listing 6.17. This code trains a simple random forest model on the artists dataset. Here we're predicting the artist popularity using the follower count and associated genres the artist has.

```
import pandas as pd #1
from sklearn.ensemble import RandomForestRegressor
import time

artists = pd.read_csv("data/artists.csv") #2

artists = artists.\ #3
    dropna().\ #3
    reset_index(drop = True) #3

artists['num_genres'] = artists.genres.map(len) #4

features = ["followers", "num_genres"] #5

start = time.time() #6
forest_model = RandomForestRegressor(
    n_estimators = 300,
    min_samples_split = 40,
    max_depth = 3,
    verbose = 1,
    n_jobs = 1
)

forest_model.fit(artists[features], #7
                 artists.popularity)
end = time.time()
print(end - start)
```

Running this code takes ~172.4 seconds. Next, let's use parallelization to speed up the model training. To do that, we just need to increase the value of `n_jobs`. For example, let's change it to 3 (alternatively, you could also change the value to -1, which will be evaluated as one less than the number of cores on your machine). Again, just like in multiprocessing, you generally don't want to use a higher value than the number of CPUs available on your machine.

The parallelized model training code is available in the

train_ml_model_parallel.py file in the book's ch6 directory.

Listing 6.18. This code modifies the previous listing just by increase the number of cores used to 3.

```
import pandas as pd #1
from sklearn.ensemble import RandomForestRegressor
import time

artists = pd.read_csv("data/artists.csv") #2

artists = artists.\ #3
    dropna().\
    reset_index(drop = True)

artists['num_genres'] = artists.genres.map(len) #4

features = ["followers", "num_genres"] #5

start = time.time() #6
forest_model = RandomForestRegressor(n_estimators = 300,
    min_samples_split = 40,
    max_depth = 3,
    verbose = 1,
    n_jobs = 3
)

forest_model.fit(artists[features], #7
    artists.popularity) #7
end = time.time()
print(end - start)
```

The updated code takes ~106.8 seconds, which is quite an improvement versus the previous version! Another aspect of training ML models is performing hyperparameter tuning. Let's cover how to parallelize hyperparamter tuning next.

Hyperparameter tuning in parallel

Parallelization can also be used when doing hyperparameter tuning. In fact, we've already seen this in Chapter 4 (Section 2.2).

Using the *MlModel* class we defined in that section, we can use

parallelization to find the optimal hyperparameters when training a random forest model on the artists dataset. We'll stick with a few of the most common hyperparameters for random forests, like the number of estimators, max depth of each tree, and minimum samples needed per tree split. The code to implement this parallelized hyperparameter search is in Listing 6.19. You can also find the code in the *parallelize_hyperparameter_tuning.py* file in the ch6 directory.

Listing 6.19. This code setups up a hyperparameter search grid, and then uses the `MLModel` class we defined back in Chapter 4 (Section 2.2) to search for the optimal hyperparameters based on the input grid and the artists dataset.

```
from ch4.ml_model import MLModel #1
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
import pandas as pd
import time

parameters = {"n_estimators": (50, 100, 150, 200), #2
              "max_depth": (1, 2),
              "min_samples_split": (100, 250, 500, 750, 1000)
              }

forest = MLModel(ml_model = RandomForestRegressor(), #3
                 parameters = parameters,
                 n_jobs = 1,
                 scoring = "neg_mean_squared_error",
                 n_iter = 5,
                 random_state = 0)

artists = pd.read_csv("data/artists.csv") #4

artists = artists.\n    dropna().\n    reset_index(drop = True) #5

artists['num_genres'] = artists.\n    genres.\n    map(len) #6

features = ["followers", "num_genres"] #7
```

```

start = time.time()
forest.tune(artists[features],
            artists.popularity) #8
end = time.time()
print("Completed in ", end - start, " seconds") #9

```

Table 6.2 shows estimated runtimes using varying number of cores (just changing the n_jobs parameter to be 1, 2, or 3) in Listing 6.19's code. To reiterate, the exact runtimes will vary depending on your machine specifications and other applications that may be running on your computer. We can get an estimate from this table, though, that using 3 cores versus just 1 can save at least half the runtime.

Table 6.2. This table summarizes the runtimes for performing the hyperparameter search using varying number of cores.

Number of cores	Runtime (seconds)
1	678.4
2	561.3
3	291.4

Next, let's walk through how to use caching to speed up operations when the same computation is needed multiple (or many) times..

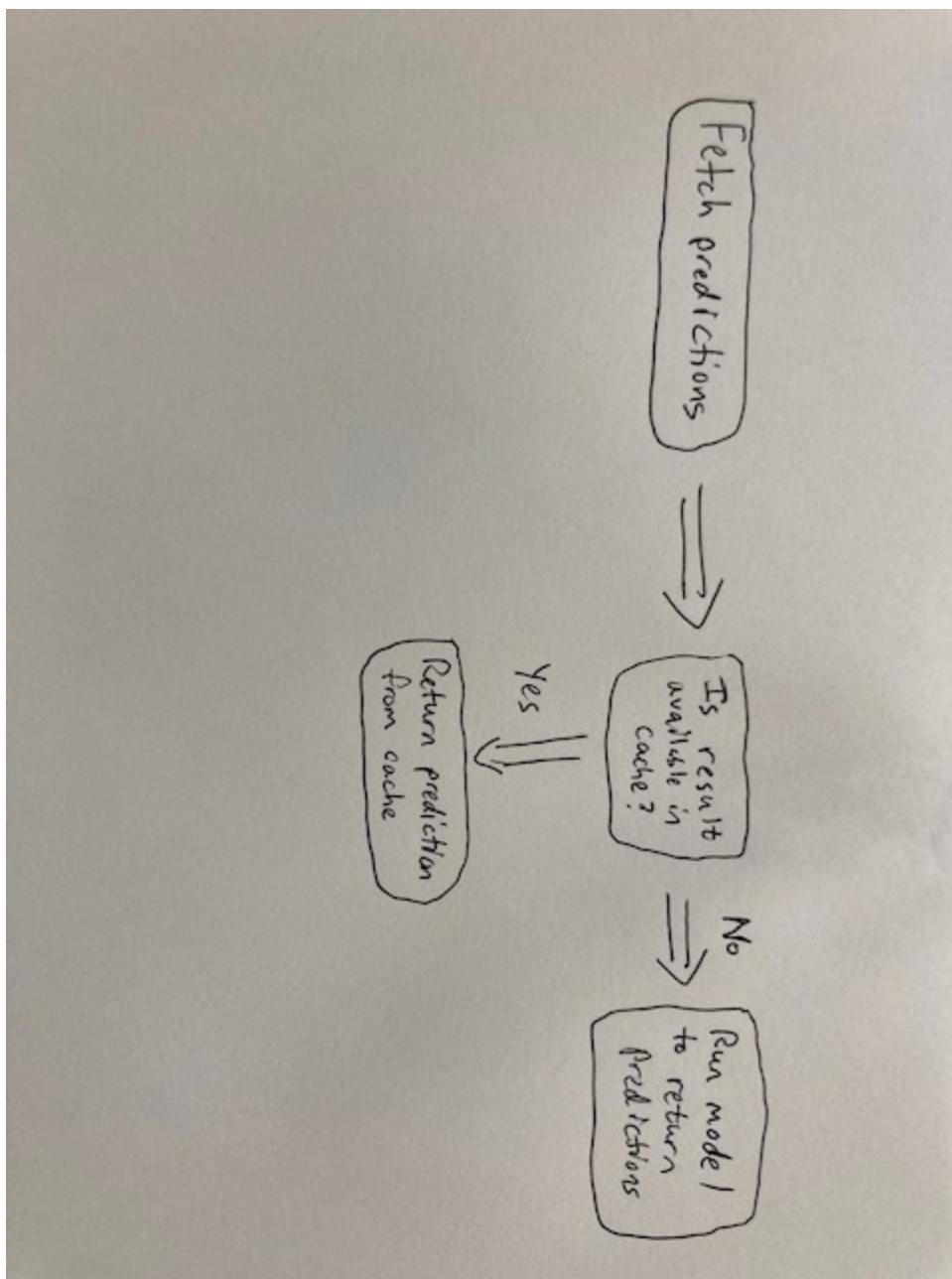
6.3 Caching

There is sometimes a tradeoff between computational efficiency and memory. Imagine you have a function that will repeatedly be called. As a specific example, suppose you've deployed a machine learning model, like the random forest we just trained on the artists dataset. In computer science, there is a concept known as memoization (note: *not memorization*). *Memoization* essentially means to store previous function call results in order to return the

results for the same inputs faster. For our example, let's suppose you want to build a web application that allows users to check predictions for particular artists. The application might be used to check how predictions might change with various inputs.

An example of a cache can be seen in the diagram in Figure 6.5.

Figure 6.5. Example of a complete progress bar being printed using tqdm



Let's implement a cache for the random forest artists model. To do that, we can make use of Python's *functools* package. This package is part of Python's standard library, so it should be available by default when you install Python.

To create the cache, we'll start by importing the *lru_cache* method from *functools*. Next, the only piece we need to add is a decorator with this method name on top of the function that for which we want to enable caching. In this case, we create a function that handles taking a tuple of inputs, and return the model prediction. In the Practice on your Own section, you'll get a chance to generalize this function to work for almost any supervised ML model.

After implementing the cache, we create a test tuple for inputs to the model, and then fetch the predictions twice. The first time will be the first time we make the predictions call for this input value, while in the second time the function will be able to find the result in the cache.

The *lru* in *lru_cache* stands for *Least Recently Used*. This is a special and very common type of cache that will store up to N items in the cache. If more than N items are added to the cache, the least recently used result will be cast out of the cache to make room for the next new function call result. By default, *lru_cache* doesn't have a max limit size. This means, in theory, you could continue adding many, many result to the cache. However, this may not be optimal as your cache can consume more and more memory as it gets larger. We'll address this issue in just a moment - for now, let's run the code to see what the performance difference is between making a function call on the same inputs for the first time vs. being able to fetch the results from the cache.

Listing 6.20. This code is available as part of *cache_ml_model_call.py* in the *ch6* directory. To create the cache, we just need to add the *lru_cache* method as a decorator to the function that we want to cache. Next, we write the function we're caching (in this case, a function that fetches model predictions). Lastly, we time the difference between running the model to get the prediction vs. fetching it from the cache (after it's available).

```
from functools import lru_cache #1

@lru_cache #2
def get_model_predictions(model, inputs): #3

    formatted_inputs = pd.DataFrame.\
```

```

        from_dict({"followers": inputs[0],
                    "num_genres": inputs[1]},
                    orient = "index").\
                    transpose()

    return model.predict(formatted_inputs)

inputs = (2, 4) #4

start = time.time()
get_model_predictions(forest_model, inputs) #5
end = time.time()
print("Model predictions runtime (w/o) cache: ", end - start)

start = time.time()
get_model_predictions(forest_model, inputs) #6
end = time.time()
print("Model predictions runtime (w/) cache: ", end - start)

```

Running the code prints the following results:

```

Model predictions runtime (w/o) cache:  0.07513427734375
Model predictions runtime (w/) cache:  0.00012421607971191406

```

Fetching the result from the cache is more than 600x times faster than calculating the result from the random forest directly! Again, the exact timing result will vary depending on your machine specifications, but this is a significant difference.

Now, what if we wanted to limit the size of the cache? This might be useful to mitigate memory consumption if we expect the cache to grow to a large size based on many inputs.

Limiting the max size is as simple as passing an extra parameter to the lru_cache decorator. Just change the line:

```
@lru_cache
```

to include the max size you want.

For example, if we want to set the limit to be 10, just specify maxsize = 10 as a parameter in lru_cache. Setting the cache size varies on several factors.

Generally, you'll want to balance it with the memory consumption (using a cache size of 1 *million* will take far less memory than a cache size of 1 *billion*) and the average speed it takes to get a result from the function that caches previous results.

```
@lru_cache(maxsize = 10)
```

Now that we've finished discussing caching, let's discuss how several of Python's data structures relate to computational efficiency.

6.4 Data structures at scale

Before we close out this chapter, let's briefly walk through how using the right data structures can help speed up certain tasks, such as searching for a value. We'll be focusing on the following data structures:

- Sets
- Priority Queues
- NumPy arrays

Let's start by discussing how sets can improve your computational efficiency.

6.4.1 Sets

Both sets and dictionaries allow for efficient lookups. In computational efficiency theory, we would say that they each allow for *constant* (or $O(1)$) lookups. If you have a set with 100 million elements, and you want to check whether an element exists in the set, you can get your answer in a fraction of a second. In contrast, if you have a *list* with 100 million elements, Python will check through every element in the list to compare whether it matches the element you are searching for. This means there are up to 100 million different comparison operations occurring under the hood. To see what the time difference looks like between checking whether a value exists in a list vs. set, let's do a quick test in Python.

In Listing 6.22, we create a sample list with 100 million elements. Then, we do a single check to determine whether a value exists in the list. Lastly, we

print out the time it takes to do this.

Listing 6.21. This code creates a sample list with 100 million values. Then, it checks whether a sample value exists within the list, printing out the time it takes to perform the check.

```
import time #1

nums_list = list(range(100000000)) #2

start = time.time()
print(1000 in nums_list) #3
end = time.time()
print(end - start) #4
```

Running the code in Listing 6.22 takes ~0.003479 seconds, as can be seen in the printout:

```
0.00347900390625
```

In contrast, we can check whether the same value exists in a set of the same 100 million numbers. The code for this is in Listing 6.23.

Listing 6.22. This code creates a set of the same 100 million integers from Listing 6.22. Then, it checks whether a value exists in this set (also printing out the time it takes to perform this lookup).

```
import time #1

nums_set = set(range(100000000)) #2

start = time.time()
print(1000 in nums_set) #3
end = time.time()
print(end - start) #4
```

Running this listing's code prints out the following:

```
0.00043773651123046875
```

Taking ~0.0004377 seconds, this result is almost 8 times faster than the previous version using a list. If you have to repeatedly check whether a value exists in a static collection of items, then using a set is generally a much better option than a list.

Next, let's discuss priority queues.

6.4.2 Priority Queues

A queue is a data structure, perhaps less known in data science, but common in software engineering. A queue offers the FIFO (first-in, first out principle). In other words, if you remove an element from the queue (called *popping* the queue), the earliest entrant to the queue will get removed.

One of the key reasons queues as a data structure are used is for their computational efficiency. If you append a new value to a queue or remove (pop) a value from the queue, it takes just a single operation ($O(1)$ time). As a comparison, a list with 100 million values would take approximately 100 million operations to insert a value at the beginning of the list because each element in the list would need to be shifted over by one. More generally N operations would be required for a list with N elements.

A *priority queue* is a special type of queue that arranges elements based on their priority. If you remove, or *pop*, an element from the queue, the value that is removed has the highest priority of the values in the queue. There are many real-world examples of priority queues. Below, we highlight a few examples that correspond to data science applications.

- Prioritizing comments on a live video. A live stream may have many comments coming in during the broadcast, and prioritizing them based on relevance, likelihood of being spam, etc. could be useful. NLP (Natural Language Processing)-based models might be useful to make these assessments. The models could output a priority score associated with each comment. Then, a process that makes the comments visible could use the priority queue in showing the highest priority comments.
- Handling customer complaints. Consider the customer churn scenario. If we had additional data around customer complaints (such as audio or text), we could develop a model that scores complaints as they come in (real-time), and stores the priorities along with the complaints in a priority queue.
- Showing popular songs on a music streaming platform could potentially use a priority queue as well. For example, you could prioritize songs

based on predicted attributes related to the songs, such as their categories, and current popularity/relevance. There could potentially be priority scores associated with the songs based on analysis or models related to the music, artists, location, or other factors.

Priority queues can be created in Python using the Queue package. A quick example is in Listing 6.23. In this example, we start by importing the queue package (which should come by default with most Python installations). Next, we create the priority queue. Then, we use the *put* method to add elements to the queue.

Listing 6.23. This code is available in the `queue_example.py` file in the `ch6` book directory. The code handles creating a queue, plus adding a few elements to the queue.

```
import queue #1

complaints = queue.PriorityQueue() #2

complaints.put((30, "Price is too high")) #3
complaints.put((10,
    """Service is terrible!
    I'm definitely not renewing my account!"""))
complaints.put((20, """
    Not sure if I want to
    renew my account"""))
```

We can retrieve values from the queue one at a time using the *get* method. Running this method will return the item with the highest priority. Since we are storing tuples in the queues, the first element of the queue will correspond to how each item is ranked. In this case, the first element of each item in the queue is the score we assign to each complaint. The highest priority complaint is the item with the lowest score. Since, we assigned the worst complaint the lowest score (in this case, 10), that complaint will get retrieved first.

Listing 6.24. This code runs the *get* method to retrieve the highest-priority item from the queue.

```
complaints.get() #1
```

The result of running this code is below:

```
(10, """
    Service is terrible!
```

```
I'm definitely not renewing my account!""")
```

So far we've covered sets and priority queues. Another popular data structure, and one that we've already touched upon, is the NumPy array. In the next section, let's recap why NumPy arrays are useful and how they help with computational speed.

6.4.3 NumPy arrays

We already talked about NumPy arrays briefly earlier in this chapter. Just to re-emphasize, NumPy arrays inherently use vectorization, so mathematical operations can be much faster than using lists.

As an additional example, let's time how long it takes to do pairwise addition across elements of two standard lists vs. NumPy arrays (see Listing 6.25).

Listing 6.25. This code times doing pairwise addition across values in two sample lists vs. the same values in two NumPy arrays.

```
import numpy as np #1
import time #1

sample_list1 = list(range(1000000)) #2
sample_list2 = list(range(1000000))

start = time.time()
combined_sum = [ #3
    num1 + num2 for
    num1, num2 in
    zip(sample_list1, sample_list2)]
end = time.time()
print("""Pairwise list addition takes """,
      end - start, " seconds")

sample_array1 = np.arange(1000000) #4
sample_array2 = np.arange(1000000)

start = time.time()
combined_sum = sample_array1 + sample_array2 #5
end = time.time()
```

```
print("Pairwise NumPy array addition takes ",  
      end - start, " seconds")
```

Printing out the time results after running Listing 6.25 shows that the NumPy approach is about 5x faster (on average) than using lists! Feel free to run the code to see for yourself. Similarly, for other operations, such as multiplication, division, etc. NumPy arrays will be much faster than standard lists.

```
Pairwise list addition takes 0.09517073631286621 seconds  
Pairwise NumPy array addition takes 0.019664764404296875 second
```

To learn more about NumPy, check out Manning's *Data Processing Tools with NumPy*, available here: <https://www.manning.com/liveproject/data-processing-tools-with-numpy>.

Now, let's briefly point out a few items we haven't yet discussed concerning computational efficiency.

6.5 What's next for computational efficiency?

There's a few points concerning computational efficiency that we haven't addressed yet, but these will be covered in the next two chapters. In the next chapter we'll be delving into memory management. A key point involving computational efficiency that we haven't fully gone into yet is improving computational efficiency in situations that also directly involve memory management. An example of this could be reading or processing a large dataset - perhaps one that doesn't fit in memory or consumes most of the memory available. Handling datasets too large for memory will be covered in Chapter 7. Beyond these concerns, most of what we covered in this chapter involved using pandas, NumPy, or other standard Python packages. There are also alternatives to these that we will walk through in Chapter 8. The alternative packages, such as Dask or Modin, can also be used to improve the computational speed of tasks in Python.

That's all for this chapter. Next, let's summarize what've learned.

6.6 Summary

In this chapter we covered the DRY principle, profiling for computational inefficiencies, vectorization, multiprocessing, parallelization in training ML models and hyperparameter tuning, caching, and utilizing better data structures for computational efficiency.

- The DRY principle means - don't repeat yourself. If your code is taking a long time to run, make sure you are not repeatedly doing the same computations. You can use the line-profiler package to quickly spot which points in your code are taking the longest to execute.
Additionally, you can think of your code in terms of its time complexity in order to more easily think about how to improve its efficiency for your own specific usecases.
- Vectorization is a method for applying operations across entire arrays at once. You can use NumPy's vectorization method to speed up tasks, such as performing operations across multiple columns in a data frame (as a faster alternative to using the pandas apply method).
- Multiprocessing means to run multiple instances of an application (for instance, Python) to hopefully accomplish a task faster. Python's multiprocessing package utilizes this paradigm to run multiple Python processes in parallel, which can often result in faster runtimes for various tasks (such as data collection, for instance).
- Caching can be a useful way to trade memory for computational efficiency. You can use caching to store a function call's results (such as an ML model's predictions).
- If you need to repeatedly lookup whether a value exists in a collection, then it is much more computiontally faster to use sets rather than lists. Sets have an $O(1)$ lookup runtime versus lists, which have an $O(n)$ runtime.
- Priority queues can be used to store values by their priority, where priority can be defined by the user - most important customer complaints, most useful comments on a livestream, etc. Fetching the item with the highest priority can also be done in $O(1)$ time. This means priority queues are very efficient in terms of figuring out what value has the highest prioity.
- Using the right data structures can be helpful for computational efficiency. For example, searching for a value in a set is much faster than a list for a large collection of values.

6.7 Practice on your own

1. Try implementing a cache from scratch yourself. The cache could be an LRU cache or another type of cache. Test your cache to get predictions with the model we trained earlier.
2. Can you do more complex NumPy vectorization tasks with arrays? Try timing them, and then implementing versions of your code using lists. What's the compute time difference?
3. Time how long it takes to find all the artist names in the artists dataset that start with the letters a, e, i, o, or u. Try doing this with a list and with a set for comparison.
4. Test doing hyperparameter tuning with different values for n_jobs. You can use the class we created in chapter 4 to handle hyperparamter tuning. Try out the class using the artists dataset that was introduced in this chapter. What's the compute time for each value you try?

7 Memory management with Python

This chapter covers

- How to profile your code for memory usage and issues
- Handling and consuming large datasets
- Optimizing data types for memory
- Training an ML model when your data doesn't fit in memory
- Making use of Python's data structures for memory efficiency

For this chapter, our primary dataset will be the ad clicks dataset available at this link: <https://www.kaggle.com/competitions/avazu-ctr-prediction/data>.

The training dataset available here has over 40 million rows. Many of the techniques we will discuss in this chapter are also applicable for even larger datasets, such as ones in the billions of rows.

In Chapter 6 (Section 1.1.2), we covered using *line-profiler* to profile your code for computational speed / efficiency issues. This helped us to easily identify what points in our code are taking the longest to run. We're going to get started in this chapter by discussing memory profiling, which is a similar mechanism for identifying what points in your code cause the highest amount of memory consumption.

7.1 Memory profiler

A *memory profiler* is a tool that allows you to identify how much memory is being consumed in various actions in your code. Similar to what we covered in the last chapter around computational profiling, we can perform an analogous check for memory.

Python has several packages that offer memory profiling. These libraries cover various tasks, such as line-by-line profiling similar to line-profiler or

calculating how much memory different data types are consuming. First, let's discuss the *guppy* package. We'll cover this library first because it provides a simple, but useful way to figure out your memory consumption by data type. This doesn't require running your code, as a line-by-line profiler does, so it can be a quick way to identify if you're taking up more memory than you may have realized. For example, using this package lets you easily see how much memory all of the data frames in your current Python session are consuming.

7.1.1 High-level memory summaries with guppy

We can install the Python 3 specific version (there is still a Python 2 version available, but it is highly recommended to use the Python 3 version) of guppy using pip (see Listing 7.1).

Listing 7.1. Use pip to install the guppy package. Make sure to specify guppy3 to get the Python 3 version.

```
pip install guppy3 #1
```

To see how much memory each data type your Python session is using, we can use the code in Listing 7.2. In the code, we start by importing the *hpy* method from guppy. The *hpy* method gives you access to the heap, where is an area where memory is stored and can be referenced by Python. This allows us to generate the memory usage by each data type currently in use during our Python session.

Listing 7.2. Use guppy to check the memory consumption by kind of object (where "kind" is a data type or class).

```
from guppy import hpy #1
data_type_summary = hpy() #2
print(data_type_summary.heap()) #3
```

An example printout of running the code in Listing 7.2 is below:

```
Partition of a set of 910281 objects. Total size = 6532736502 byte
Index  Count    %     Size    % Cumulative  % Kind (class / dict o
```

```

0      23    0 4786211179   73 4786211179   73 pandas.core.frame
1     370    0 1310176775   20 6096387954   93 numpy.ndarray
2      32    0 324541602    5 6420929556   98 pandas.core.series
3 378435    42 44166696    1 6465096252   99 str
4 159387    18 12246376    0 6477342628   99 tuple
5  44322     5 7967590    0 6485310218   99 types.CodeType
6  89041    10 7721792    0 6493032010   99 bytes
7  41079     5 5586744    0 6498618754   99 function
8  16178     2 5110360    0 6503729114  100 dict (no owner)
9   5093     1 5026016    0 6508755130  100 type
<2360 more rows. Type e.g. '_._more' to view.>

```

The results in this view are sorted by memory consumption, with all data frames currently in memory (in this example) taking up 4786211179 bytes (or over 4GB). We can see that this accounts for 73% of memory consumption. Running this code yourself will result in a different output depending on whatever objects you currently have in memory. The printout is limited to the top ten classes or data types by memory consumption, but you can easily see more by following the instructions at the bottom of the printout (type `_._more` and enter). Using guppy to generate the memory consumption view can be useful to get a high-level understanding where the memory usage is coming from.

To get a line-by-line analysis of your memory consumption, we can use the memory-profiler package. Let's discuss this library next!

7.1.2 Analyzing your memory consumption line by line with memory-profiler

The memory-profiler package works similarly to the line-profiler library that was discussed in Chapter 6 (Section 1.1.2). To get started, let's install memory-profiler using pip:

Listing 7.3. Install memory-profiler using pip.

```
pip install memory-profiler #1
```

We can use memory-profiler very similarly to line-profiler. First, we just need to add the `@profile` decorator above any function we want to profile. Then, we will run our code from the terminal using the profiler.

In Listing 7.4, we present a problem of handling a large dataset. In this code snippet, we perform the following general steps:

1. Read in the full ads dataset (with over 40 million rows and 25 columns)
2. Do basic feature engineering to get one-hot-encoded columns for the site_category field, and combine this with the banner_pos column to get the input features for a model to predict ad clicks
3. Train a logistic regression model on the dataset and return this ad-click model as the output of a function

The code in Listing 7.4 is available in the train_model_without_optimization.py file in the ch7 directory.

Listing 7.4. This code is a simple example of reading in a sample of the ads data, and then return a filtered version of the dataset (where the field, click = 0). We use the @profile decorator above the function so that memory-profiler will profile this function when calling the library from the terminal.

```
import pandas as pd #1
from sklearn.linear_model import LogisticRegression
import time

@profile #2
def train_logit_model(filename: str): #3

    ad_frame = pd.read_csv(filename) #4
    print("Finished reading dataset...\\n\\n") #4

    input_features = ad_frame[[ #5
        "site_category",
        "banner_pos"]]

    sc_features = pd.\
        get_dummies(input_features.\
        site_category)

    final_features = pd.concat([ #6
        sc_features,
        ad_frame.banner_pos])

    logit_model = LogisticRegression( #7
        random_state = 0)

    print("Start training...\\n\\n") #8
```

```

logit_model.fit(ad_frame[final_features],
ad_frame.click)

return logit_model #9

start = time.time()
logit_model = train_logit_model( #10
    "data/ads_train_data.csv")
end = time.time()
print("Finsihed running in ",
      end - start, " seconds") #11

```

Now, we can profile the code in Listing 7.4 by running the command in Listing 7.5.

Warning: running this code may take ~2-3 hours to run depending on your machine specifications. If you'd like to try it out on a smaller dataset, you can replace *ch7/train_model_without_optimization.py* with *ch7/sample_train_model_without_optimization.py* file instead.

Listing 7.5. This code executes memory-profiler on our example script from Listing 7.4.

```
python -m memory_profiler
    ch7/train_model_without_optimization.py #1
```

The result of running Listing 7.5 should generate the following output:

Line #	Mem usage	Increment	Occurrences	Line Contents
6	119.098 MiB	119.098 MiB	1	
7				
8				
9	1893.219 MiB	1774.121 MiB	1	
10	1893.414 MiB	0.195 MiB	1	
11				
12	2672.891 MiB	779.477 MiB	1	
13				
14	2477.320 MiB	-836.914 MiB	3	
15	2477.320 MiB	0.008 MiB	2	
16				
17				
18	1836.211 MiB	-641.109 MiB	1	
19	1836.219 MiB	0.008 MiB	1	

```

20 3383.680 MiB 1547.461 MiB           1
21
22 3383.727 MiB      0.047 MiB           1

Line #  Line Contents
=====
6  @profile
7  def train_logit_model(filename: str):
8
9      ad_frame = pd.read_csv(filename)
10     print("Finished reading dataset...\n\n")
11
12     input_features = ad_frame[["site_category", "banner_pos"]]
13
14     final_features = pd.concat([pd.get_dummies(
15         input_features.site_category),
16         ad_frame.banner_pos], axis = 1)
17
18     logit_model = LogisticRegression(random_state = 0)
19     print("Start training...\n\n")
20     logit_model.fit(final_features, ad_frame.click)
21
22     return logit_model

```

Let's break down the output:

- Line 6 corresponds to the `@profile` decorator. Using this decorator consumes around 119MB of memory. This can be seen under the Increment column.
- Reading the ads dataset consumes ~1774 MB in Line 9. The cumulative memory at this point is 1893 MB (under the Mem Usage column).
- Creating the `input_features` data frame adds an extra ~779 MB of memory
- The peak memory usage hit during the function call is ~3384 MB (by Line 22), or ~3 GB

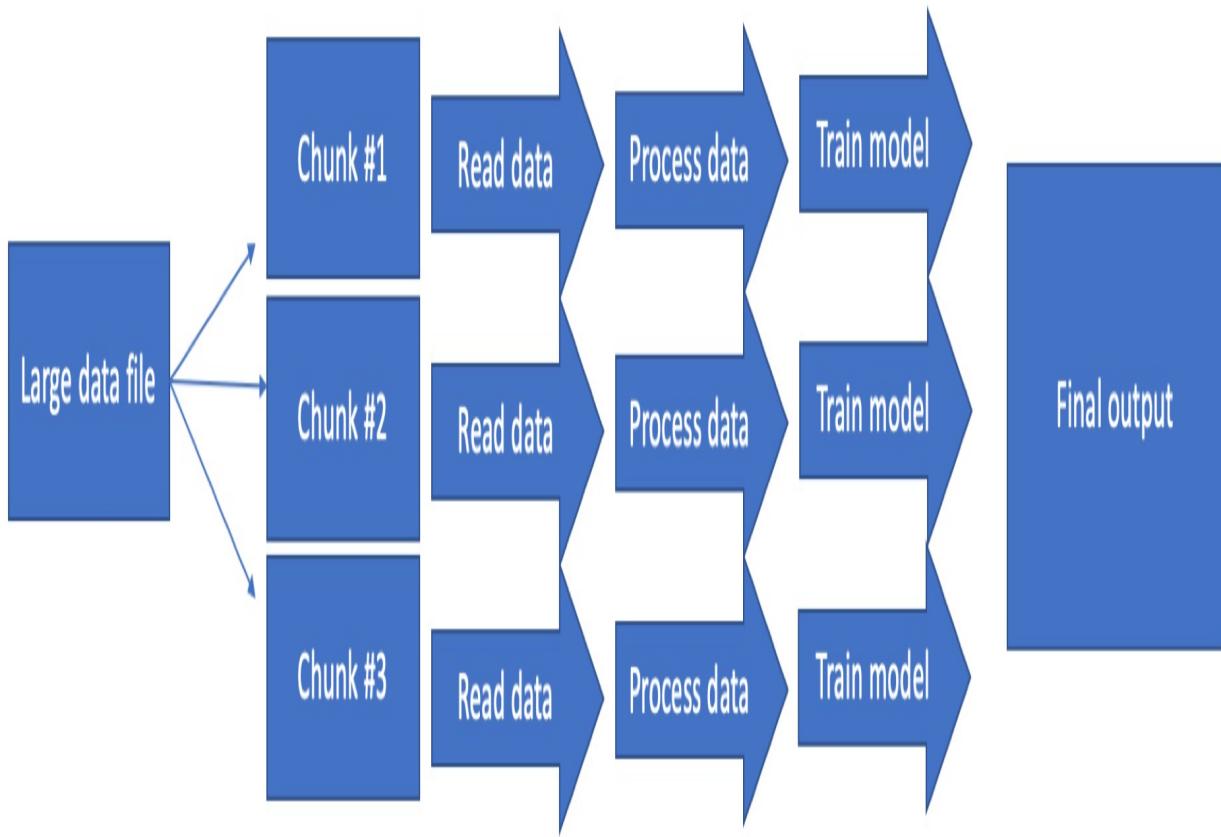
Given the amount of time it takes to run the code in Listing 7.4 (as mentioned, this can be 2-3 hours depending on our machine specifications), and the peak memory usage of over 3 GB - how can we do better? In the next few sections, we'll delve into various ways to reduce memory consumption. Let's start by talking about how to handle processing larger datasets in Python. This will allow us to process the full ads dataset, with over 40

million rows, while utilizing less memory and speeding up the operations of reading in the dataset, performing feature engineering, and training a model. The techniques we cover will also be able to scale to even larger datasets (hundreds of millions or billions of rows).

7.2 Sampling and chunking large datasets

A common way to deal with large datasets is to utilize chunking or sampling to reduce the amount of data consumed at once. *Chunking* means to break a dataset into pieces (chunks) so that each individual chunk can be processed separately. This can be especially helpful if your memory resources are limited and you're dealing with a large (potentially larger-than-memory) dataset. The idea of how this works is summarized in Figure 7.1.

Figure 7.1. This is an example of a workflow utilizing chunking where we split a large dataset into chunks. Each chunk is read in, processed, and potentially even input into a training model separately.



Now, let's discuss how we can use pandas to read from a large CSV file (including one too large to fit in memory) using chunks.

7.2.1 Reading from a large CSV file using chunks

In this section we will discuss a simple way of using pandas to process datasets that are too large to fit into memory. We can do that using *chunks*. For example, suppose we want to get a sample of the 40-million row ad clicks dataset. Pandas has a useful parameter called *chunksize*, which we can use to read in N rows at a time, where N is equal to the number of rows you want to read in at one time. For example, let's write a snippet of code to read in the first million rows of the dataset (see Listing 7.6). Running the code in Listing 7.6 takes about ~0.1 seconds.

Listing 7.6. The code reads in the first million rows the ad clicks dataset. By specifying chunksize = 1000000, everytime we call the get_chunk method, we will fetch the next million rows of the dataset.

```
import pandas as pd #1
import time

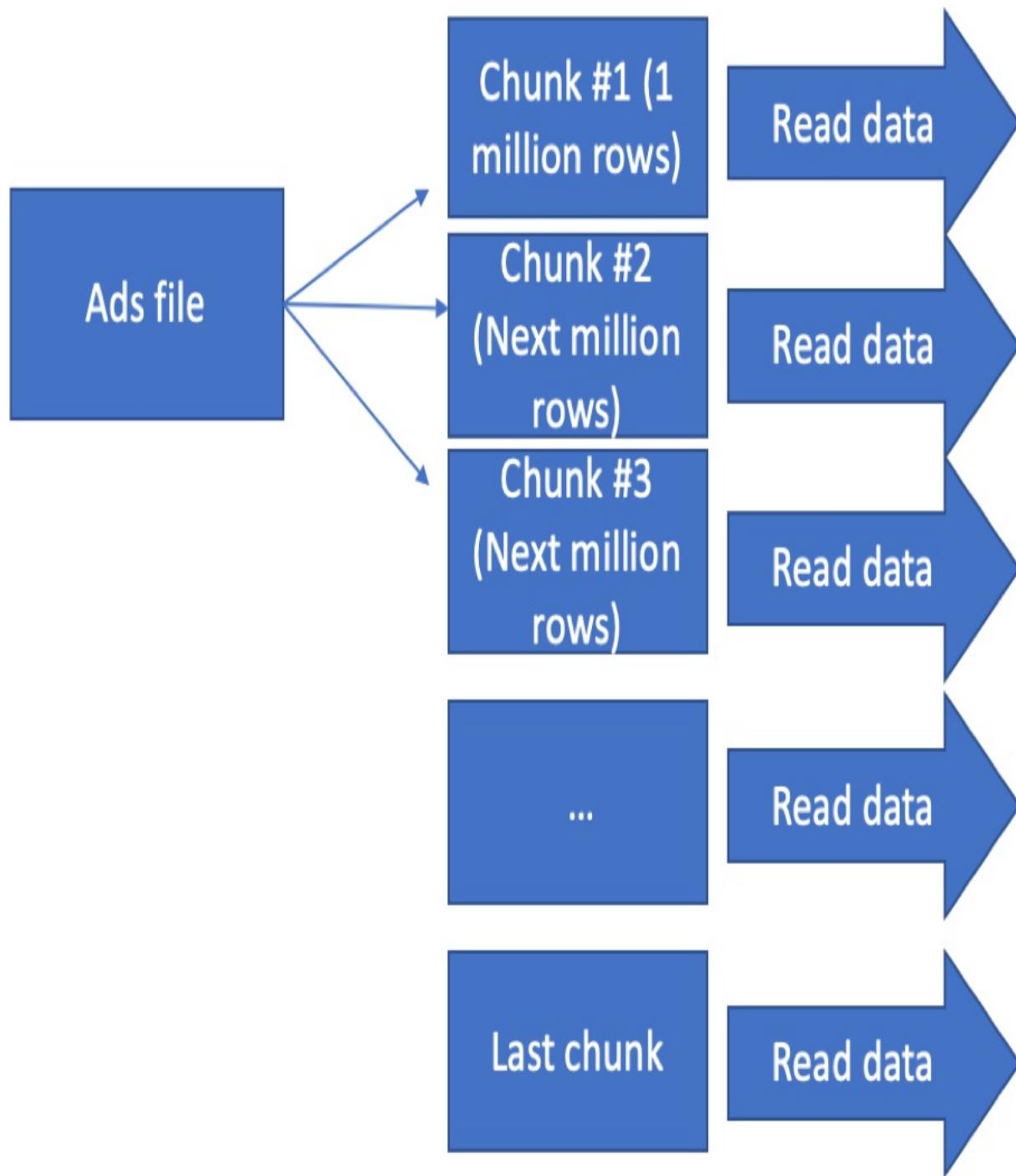
start = time.time()
ad_data = pd.read_csv("data/ads_train_data.csv", #2
                      chunksize=1000000) #2

ad_data.get_chunk() #2

end = time.time()
print(end - start)
```

A visualization of the chunking process for the ads dataset is in Figure 7.2. This visual shows how we break the ads file into chunks of 1 million rows, reading in the data, one chunk at a time.

Figure 7.2. Similar to Figure 7.1, this views summarizes the chunking process for the ads file. For now, this allows us to read in the file one chunk at a time. In Section 1.4, we will expand the example to include processing the data and training a model using the chunks.



An important point to keep in mind for the example in Listing 7.6 is that this same principle of chunking will scale regardless of whether your dataset is in the tens of millions of rows, or even tens of *billions* of rows.

Now that we've covered one key way of reading in data from a large file, let's discuss how to randomly select data from a large dataset (using the ads dataset as our example).

7.2.2 Random selection

One drawback with the chunking example we just went through is that the selection is not random. Randomizing data is often crucial for training machine learning models. In our next example, we'll randomly sample from the ads dataset.

The code in Listing 7.7 reads a randomized sample of 10% of the ads dataset, while avoiding ever reading the full dataset into memory.

Listing 7.7. The code randomly samples 10% of the ads dataset without ever reading the full dataset into memory.

```
import pandas as pd #1
import random
import time

def get_random_sample(
    filename: str, #2
    sample_rate: float):

    total_rows = sum(1 for line #3
        in open(filename)) - 1

    num_samples = round( #4
        total_rows * sample_rate)

    skip = random.sample(range(1, total_rows + 1), #5
        total_rows - num_samples)

    skip = sorted(skip)

    return pd.read_csv(filename, #6
        skiprows=skip)

start = time.time() #7
sampled_data = get_random_sample(
    "data/ads_train_data.csv",
```

```

    0.1)
end = time.time()

print("Read in dataset in ",
      end - start,
      " seconds")

```

Currently, running the code in Listing 7.8. takes over two minutes (exact time may vary depending on your machine specifications). We'll discuss how to reduce this time further in the next chapter when we cover alternatives to pandas.

Before we move into other ways of reducing memory, let's briefly discuss how you can use chunking when reading data from a database.

7.2.3 Chunking when reading from a database

Using chunks works well if your data is stored in a CSV format. However, it is also a very common need to extract data from databases. We can also use chunking to avoid reading too much data into memory at once. For example, suppose the ads dataset is stored in a SQLite database table called *core_ads_data* (see Appendix for how to setup this database). The database could simply be called *ads*. We can use the pandas *read_sql* method to ingest the data in chunks from this database table. The example would look like the code in Listing 7.8.

Listing 7.8. The code reads the ads data from a SQLite database.

```

import pandas as pd #1
import pyodbc

#2
conn = pyodbc.connect("DRIVER={SQLite3 ODBC Driver};
                      SERVER=localhost;DATABASE=ads.db;
                      Trusted_connection=yes")

ads_data = pd.read_sql(conn,
                       """SELECT click, #3
                          site_category,
                          banner_pos
                         FROM core_ads_data
                       """,

```

```
chunksize = 1000000)
```

Next, let's discuss how to reduce memory consumption by using optimal data types for your dataset.

7.3 Optimizing data types for memory

The data type of individual columns in a pandas data frame can have a significant impact on the amount of memory used to store a column's values. Before we delve into checking memory usage for data frames or columns, let's do a quick review of how to check the data type of individual columns in a pandas data frame.

7.3.1 Checking data types

Pandas has a method called *dtypes* that can be used to check the data type of each of the variables in a data frame. The code in this listing is available in *check_data_types_of_df.py* file in the *ch7* directory, but is also shown in Listing 7.9.

Listing 7.9. This code reads in the first million rows of the ad dataset. Then, it checks the data type for each column in the data frame.

```
import pandas as pd #1

ad_data = pd.read_csv("data/ads_train_data.csv", #2
                      chunksize=1000000)

ad_frame = ad_data.get_chunk() #2

print(ad_frame.dtypes) #3
```

The result of running this code will print the following:

id	float64
click	int64
hour	int64
C1	int64
banner_pos	int64
site_id	object
site_domain	object

site_category	object
app_id	object
app_domain	object
app_category	object
device_id	object
device_ip	object
device_model	object
device_type	int64
device_conn_type	int64
C14	int64
C15	int64
C16	int64
C17	int64
C18	int64
C19	int64
C20	int64
C21	int64

Now, let's walk through how to check the memory usage of data frames and individual columns.

7.3.2 How to check the memory usage of a data frame

You can see the memory usage of a data frame using the *memory_usage* method. This can be seen in Listing 7.10.

Listing 7.10. The code reads in the first million rows of the ads dataset, followed by checking the memory usage of the data frame.

```
import pandas as pd #1

ad_data = pd.read_csv("data/ads_train_data.csv", #2
                      chunksize=1000000)

ad_frame = ad_data.get_chunk() #2

print(ad_frame.memory_usage(deep = True)) #3
```

Running the code in Listing 7.10 will print out the following:

Index	128
id	8000000
click	8000000
hour	8000000

C1	8000000
banner_pos	8000000
site_id	65000000
site_domain	65000000
site_category	65000000
app_id	65000000
app_domain	65000000
app_category	65000000
device_id	65000000
device_ip	65000000
device_model	65000000
device_type	8000000
device_conn_type	8000000
C14	8000000
C15	8000000
C16	8000000
C17	8000000
C18	8000000
C19	8000000
C20	8000000
C21	8000000

This output shows the number of bytes used in memory for each column in the dataset. If we compare this view vs. the earlier one showing the data type of each column, we can see that the variables with the *object* data type consume the most memory. As we'll see later in this section, we can often save a large chunk of memory by converting these *object* columns to other data types if possible.

Now, let's show how to check the memory usage of individual columns, rather than the entire data frame at once.

7.3.3 How to check the memory usage of a column

In the code in Listing 7.11, we check the memory usage of just the `banner_pos` column in the `ads` dataset.

Listing 7.11. The code reads in the `ads` dataset, followed by checking the memory usage of the `banner_pos` column.

```
import pandas as pd #1
ad_data = pd.read_csv("data/train", #2
```

```
    chunksize=1000000) #2  
  
ad_frame = ad_data.get_chunk() #2  
  
print(ad_frame.\ #3  
      banner_pos.\ #3  
      memory_usage(deep = True)) #3
```

Running Listing 7.11 will show the following:

```
8000128
```

This means the single banner_pos column consumes around 8MB of memory! Now, let's explain how to convert numeric columns to be more efficient.

7.3.4 Converting numeric data types to be more memory-efficient

The result of running the previous listing shows the banner_pos column consumes around 8000000 bytes, or roughly 8 bytes per element in the column.

Next, let's convert the banner_pos column to be int8 (see Listing 7.12). The max value of an int8 value can be 255. Because int8 variables have a smaller numeric range than int64 (or int16, int32, etc.), they consume less memory than these other int-based data types. In our example, banner_pos is a binary variable, so having a max possible value of 255 isn't an issue. Next, we can check the updated memory usage - again, using the *memory_usage* method. The result of converting this column to int8 shows the memory consumption at closer to 1000000 bytes, or approximately 1/8 the previous memory usage! That's a significant savings and we're only dealing with one column here. We can apply optimizations to other columns in our dataset, as well.

Listing 7.12. This code converts the banner_pos column to the int8 data type. Then, we check the memory usage of the banner_pos variable with the new data type.

```
import numpy as np #1  
  
ad_frame["banner_pos"] = ad_frame["banner_pos"].\ #2
```

```
astype(np.int8)

print(ad_frame.\ #3
      banner_pos.\ 
      memory_usage(deep=True))
```

Since we know how to handle optimizing numeric data types now, let's go through an analogous example for text columns.

7.3.5 Category data type

One of the largest consumers of memory can be string fields in a data frame. Pandas may show these columns as *object* data types. If a string field is representing a categorical variable, then you can often save memory by converting the column to a pandas *Categorical* type.

In Listing 7.13, we check the memory usage of the site_category field. The result is roughly 65MB.

Listing 7.13. The code here checks the memory usage of the site_category feature.

```
print(ad_frame.\ #1
      site_category.\ 
      memory_usage(deep = True))
```

Now, let's convert the site_category field to a categorical variable. After doing this (in Listing 7.14), the updated memory usage is only ~1MB. This means the previous data type version of this column was consuming ~65-times the amount of memory vs. after the update!

Listing 7.14. This code converts the site_category feature to a category (categorical) data type. Then, we check the updated memory used to store the variable.

```
ad_frame["site_category"] = ad_frame.\ #1
      site_category.\ 
      astype("category")

print(ad_frame.\ #2
      site_category.\ 
      memory_usage(deep = True))
```

Next, let's discuss how to use sparse data types.

7.3.6 Sparse data type

Pandas also offers a sparse data type that allows for more optimal memory usage when a data frame or column is comprised mostly of a single value. For example, a column would be mostly zeros or mostly have missing values. Using a sparse data type compresses the memory needed for such columns.

In Listing 7.15 (also available in the `convert_column_to_sparse_dtype.py` file), we create a new column called `click_pos_interaction`, which equals 1 when both the `click` and `banner_pos` fields are 1 - otherwise, it equals 0. This field is largely zero (~96% for the first chunk of ads data).

Listing 7.15. This code creates a new column called `click_pos_interactions`, based on the `click` and `banner_pos` fields. This column is mostly zero, which means we can compress the memory usage using the `SparseDtype` option available in pandas. After we've created the column, we convert it to this sparse data type, and compare the memory usage before and after.

```
import pandas as pd #1
import numpy as np #1

#2
ad_frame['click_pos_interaction'] =
    ad_frame.click * ad_frame.banner_pos

num_bytes = ad_frame.\ #3
    click_pos_interaction.\ 
        memory_usage(deep = True)
print(f"""Bytes used before
transformation: {num_bytes}""")

ad_frame['click_pos_interaction'] = ad_frame.\ #4
    click_pos_interaction.\ 
        astype(pd.SparseDtype("float", 0))

num_bytes = ad_frame.\ #5
    click_pos_interaction.\ 
        memory_usage(deep = True)
print(f"""Bytes after before
```

```
transformation: {num_bytes}""") #5
```

The result of running Listing 7.15 is shown below:

```
Bytes used before transformation: 8000128
Bytes after before transformation: 508556
```

As you can see, the memory consumption after transforming the new feature to a sparse data type is almost 16-times less than what is was before!

Next, let's discuss how to specify data types when reading in a dataset.

7.3.7 Specifying data types when reading in a dataset

In the previous examples, we assumed we've already read in a dataset and modified the data types of specific columns. It's also possible to specify the data types directly when reading a dataset into Python. This avoids consuming initial additional memory when importing the dataset. This will be especially useful if you already know the proper data types for the fields you want. Let's walk through an example using the ads dataset.

Listing 7.16. This code is available in the `specify_data_type_reading_file.py` file in the `ch7` directory. It reads in the first million rows of the ads dataset, while specifying the data types of a few features. Then, it prints the memory usage of each column in the dataset.

```
import pandas as pd #1
import numpy as np

ad_data = pd.read_csv("data/ads_train_data.csv", #2
                      chunksize=1000000,
                      dtype = {"site_category": "category",
                               "banner_pos": np.int8,
                               "click": np.int8})

ad_frame = ad_data.get_chunk() #2

print(ad_frame.memory_usage(deep = True)) #3
```

Next, let's summarize data types and memory consumption.

7.3.8 Summary of data types and memory

Table 7.1 shows a summary of when to use specific data types for optimizing memory consumption. For example (as discussed in this section), if you have a column that has a discrete collection of categories, it will be beneficial to convert the column to a *category* data type, rather than the *object* data type.

Table 7.1. This table summarizes a few key data types and when to use them for improving memory performance.

Data type	When to use
category	Column contains a set of discrete categories
SparseDtype	Column is mostly one value. Can be mostly zeros, NA values, or any other arbitrary value as long as it is mostly that value
int8, int16, int32	Depends on the range of your numeric field. int8 ranges between -128 and +127 ($2^8 - 1$). int16 ranges between -32768 and +32767 ($2^{16} - 1$). int32 ranges between -2,147,483,648 and +2,147,483,647 ($2^{32} - 1$)

That covers optimizing for data types. Beyond data types, another way to save memory is to directly limit the columns in your dataset when reading it in. Let's cover this in the next section.

7.3.9 Limiting number of columns

One fairly easy, but important, way to limit memory is to restrict the columns in the dataset you read in when importing a dataset. Often, if you're

processing or performing analysis on a dataset, it may have additional columns that you care less about. Fortunately, pandas has a built-in parameter in the `read_csv` method to handle this. See Listing 7.17 for an initial example. This code snippet combines reading in a subset of columns with Listing 7.16 showing how to specify data types when ingesting a dataset. Notice this example is similar to the earlier Listing 7.4. That listing uses the same columns we read in now to train a logistic regression model. The key differences are that Listing 7.17 *only* reads in the few columns we need (`click`, `site_category`, and `banner_pos`) and specifies the data types of those fields on ingestion to reduce the memory footprint.

Listing 7.17. Available in the `limit_columns_example.py` file in the `ch7` directory, this code reads in the first million rows of the ads dataset, but only includes four sample columns.

```
import pandas as pd #1

ad_data = pd.read_csv("data/ads_train_data.csv", #2
    chunksize=1000000,
    usecols = ["click", "site_category", "banner_pos"],
    dtype = {"site_category": "category",
        "banner_pos": np.int8,
        "click": np.int8})

ad_frame = ad_data.get_chunk()
```

Next, let's walk through a sample workflow that combines a few of the techniques we've covered in this chapter. This workflow will essentially convert Listing 7.4 into a workflow using chunks, much less peak memory usage, and computes in a much faster time (read on to see the difference!). After we walk through this workflow, we'll re-run memory-profiler to contrast the difference in memory usage vs. the original task of reading in the ads dataset, encoding the `site_category` field, and training a logistic regression model.

7.4 Processing workflow for individual chunks of data

Thus far, we've discussed limiting rows and columns, as well as optimizing for data types to limit the memory consumption. While each of these methods

are great for reducing memory, what would a workflow look like after we've followed these techniques to get a dataset? As we've already discussed in earlier chapters around data and ML pipelines, we would also need to be able to clean the data, perform feature engineering, and train a model, as a few examples. Let's walk through this process in the next few sub-sections. We will focus on the following tasks:

- Cleaning and feature engineering for large datasets
- Training an ML model when your data doesn't fit into memory

For these examples, we will focus on the original task in Listing 7.4 to extract a few features from the ads dataset and train a model.

Let's start by discussing doing cleaning and feature engineering on the ads dataset.

7.4.1 Cleaning and feature engineering for big datasets

Consider the problem of training a logistic regression model to predict ad clicks using our ads dataset. When working with a larger dataset, it's usually better to start on the simpler side - so let's do this by using just two variables from our dataset. We did this already in Listing 7.4. However, in that code listing, we still read in the full ads dataset, and didn't perform any memory optimizations.

In Listing 7.18, we create a function called *process_df* that handles reading in the ads dataset, chunk by chunk. It reads in the three specific columns we need - click (the label), site_category, and banner_pos.

Listing 7.18. This code processes the ads dataset chunk by chunk. Currently, the processing involves one-hot encoding the site_category field and combining it with the banner_pos variable. These will be used later in the chapter to train a model on the ads data.

```
import pandas as pd #1
from tqdm import tqdm #1

def process_df(): #2
    filename = "data/ads_train_data.csv" #3
    fields = ["click", #3
```

```

"site_category",
"banner_pos"]

ad_data = pd.read_csv(filename, #4
                      chunksize=1000000,
                      usecols = fields)

for ad_frame in tqdm(ad_data): #5
    features = pd.concat([ad_frame["banner_pos"],
                          pd.get_dummies(ad_frame.site_category,
                                         prefix = "site_category")],
                          axis = 1)

process_df()

```

Currently, the function in this listing doesn't actually return anything. It just sets up a loop to go through each chunk of the ads dataset, one-hot encoding the site_category field and combining it with banner_pos. This gets the input features for each chunk. In the next section, we will add a piece of code to train the ad-click prediction model chunk by chunk.

7.4.2 Training a logistic regression model when the data doesn't fit in memory

In this section, we will use scikit-learn's SGDClassifier class to train a logistic regression model on the ads dataset, one chunk at a time. This method allows for *partial fitting*. Recall that because logistic regression is an iterative method that can optimize its objective function (log loss) using stochastic gradient descent (the *SGD* in SGDClassifier), this means we can train a model on sequential batches of data. There are several types of ML models that can operate on batches of data, including:

- Logistic regression
- SVM (Support Vector Machine)
- Perceptron algorithm
- K-means (using a different class called *MiniBatchKMeans*). See this link for more details: <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html>

Now, let's use SGDClassifier to train a logistic regression model on the ads dataset. To use SGDClassifier for other types of models, you can check out the scikit-learn documentation for it here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html.

Training a logistic regression model for the ads dataset

To start, we'll write our code specifically for the ads dataset, and then in the immediate next example, we'll generalize our code to handle other datasets. The goal of the logistic regression model is to predict whether an ad will be clicked by a user.

Our initial code to train the model is in Listing 7.19.

Listing 7.19. The code in this listing trains a logistic regression model on the ads dataset. For now, we narrow the list of variables we're using to two inputs (site_category and banner_pos), along with the label (click) that indicates whether an ad click occurred.

```
import pandas as pd #1
from tqdm import tqdm
from sklearn.linear_model import SGDClassifier

def train_model(): #2

    filename = "data/ads_train_data.csv" #3
    inputs = ["click", #3
              "site_category",
              "banner_pos"]

    ad_data = pd.read_csv(filename, #4
                          chunksize=1000000,
                          usecols = inputs)

    sgd_model = SGDClassifier(random_state = 0, #5
                              loss = "log")

    fields = [] #6
    for ad_frame in tqdm(ad_data): #7

        features = pd.concat([
            ad_frame["banner_pos"],
            pd.get_dummies(ad_frame.site_category,
                           prefix = "site_category")],
```

```

    axis = 1)

if not fields: #8

    top_site_categories = set(ad_frame.\ #9
        site_category.\ 
        value_counts().\ 
        head().index.tolist())

fields = [field for field in #10
    features.columns.tolist()
    if "site_category" in field]

fields = [field for field in fields #11
    if "site_category_" + field
    in top_site_categories]

fields.append("banner_pos") #12

sgd_model.partial_fit(features[fields], #13
    ad_frame.click,
    classes = (0, 1))

return sgd_model #14

sgd_model = train_model() #15

```

Now, let's generalize the code in Listing 7.19 to handle any arbitrary dataset.

Generalize the model framework to an arbitrary dataset

In Listing 7.20, we modify our code to use a class that takes a filename as input, along with several other parameters such as what numeric and categorical signals you wish to include when reading/processing the dataset. In part 6 of the code, we add the *profile* decorator so that you can use memory-profiler on the main method that handles reading in, processing, and training the logistic regression model on the ads dataset. The full code is available in the ch7/sgd_model_chunking.py file.

Listing 7.20. The code in this listing trains a logistic regression model on the ads dataset. For now, we narrow the list of variables we're using to two inputs (site_category and banner_pos), along

with the label (click) that indicates whether an ad click occurred.

```
class Chunk_pipeline: #1
    def __init__(self, #2
                 filename: str, label: str, numeric_signals: list,
                 cat_signals: list, chunksize: int):

        self.filename = filename #3
        self.label = label
        self.numeric_signals = numeric_signals
        self.cat_signals = cat_signals
        self.keep_cols = [self.label] + \
                         self.numeric_signals + \
                         self.cat_signals
        self.chunksize = chunksize

    def get_chunks(self): #4
        return pd.read_csv(self.filename,
                           chunksize=self.chunksize,
                           usecols = self.keep_cols)

    @profile #5
    def train_model(self): #5
        data_chunks = self.get_chunks() #6
        self.sgd_model = SGDClassifier( #7
            random_state = 0,
            loss = "log")

        keep_fields = [] #8
        for frame in tqdm(data_chunks): #9
            features = pd.concat([
                frame[self.numeric_signals],
                pd.get_dummies(frame[self.cat_signals]),
                axis = 1])

            if not keep_fields: #10
                for signal in self.cat_signals: #11
                    top_categories = set(
                        frame[signal].value_counts().\
                        head(5).index.tolist())

                    fields = [field for field in
                              features.columns.tolist()
                              if field in top_categories]

                    keep_fields.extend(fields)
```

```

        keep_fields.\ #12
        extend(self.\
            numeric_signals)

    self.sgd_model.\ #13
        partial_fit(features[keep_fields],
            frame[self.label],
            classes = (0, 1))

pipeline = Chunk_pipeline( #14
    filename = "data/ads_train_data.csv",
    label = "click", numeric_signals=["banner_pos"],
    cat_signals=["site_category"],
    chunksize=1000000)
pipeline.train_model() #15

```

Now, let's use memory-profiler on our updated code from Listing 7.20. Using memory-profiler here will show us the updated memory performance (how much memory is used per line in the code, as well as peak memory usage). We can use this to compare Listing 7.20 to Listing 7.4, which was our initial attempt at using the ads dataset to train a model.

Running memory-profiler on the updated workflow

The command to run memory-profiler on our updated script is in Listing 7.21.

Listing 7.21. This code runs memory-profiler on the sgd_model_chunking script.

```
python -m memory_profiler ch7/sgd_model_chunking.py #1
```

Running memory-profiler on the code from Listing 7.20 shows a peak memory usage of 267MB (see line 57 below). In contrast, Listing 7.4 had a peak memory usage of 3384MB! Additionally, we used *tqdm* to monitor the progress of looping through each chunk. The entire process of reading in each chunk, creating the features we needed, and training the model took around 70 seconds (1 minute, 10 seconds)! This is in comparison to over 2 hours in Listing 7.4!

```
41it [01:10, 1.73s/it]
Filename: ch7/sgd_model_chunking.py
```

Line #	Mem usage	Increment	Occurrences
24	119.312 MiB	119.312 MiB	1
25			
26			
27	122.910 MiB	3.598 MiB	1
28			
29	122.918 MiB	0.008 MiB	2
30	122.910 MiB	0.000 MiB	1
31			
32			
33	122.918 MiB	0.000 MiB	1
34	215.691 MiB	-4141.797 MiB	42
35			
36	264.207 MiB	-6364.523 MiB	123
37	264.207 MiB	-2056.062 MiB	41
38	264.207 MiB	-2152.941 MiB	41
39			
40			
41	263.707 MiB	-2172.930 MiB	41
42			
43	231.133 MiB	0.000 MiB	2
44	230.820 MiB	-0.312 MiB	1
45			
46			
47			
48	230.820 MiB	0.000 MiB	25
49	230.820 MiB	0.000 MiB	22
50			
51			
52	230.820 MiB	0.000 MiB	1
53			
54	230.820 MiB	-0.312 MiB	1
55			
56			
57	267.277 MiB	-4161.613 MiB	82
58	263.707 MiB	-2292.168 MiB	41
59	263.707 MiB	-2152.926 MiB	41
60			
61			
62	150.895 MiB	-64.797 MiB	1

Line #	Line Contents
24	@profile
25	def train_model(self):
26	
27	data_chunks = self.get_chunks()

```

28 sgd_model = SGDClassifier(random_state = 0,
29     loss = "log")
30
31
32 keep_fields = []
33 for frame in tqdm(data_chunks):
34
35     features = pd.concat([frame[self.numeric_signals],
36         pd.get_dummies(frame[self.cat_signals])],
37         axis = 1)
38
39
40     if not keep_fields:
41
42         for signal in self.cat_signals:
43             top_categories = set(frame[signal].value_counts().\
44                 head().index.tolist())
45
46
47             fields = [field for field in features.columns.tolist(
48                 if field in top_categories]
49
50
51         keep_fields.extend(fields)
52
53     keep_fields.extend(self.numeric_signals)
54
55
56     sgd_model.partial_fit(features[keep_fields],
57         frame[self.label],
58         classes = (0, 1))
59
60
61
62 self.sgd_model = sgd_model

```

Now, let's walk through a few additional ways of optimizing for memory in your code.

7.5 Additional tips for saving memory with Pandas and Python

In this section we will introduce three additional ways to save memory in Python:

- Avoiding creating extraneous variables
- Staying away from gloabl variables
- Comparing NumPy arrays vs. lists for memory usage
- Alternatives to pandas

Let's get started by discussing the effect of unnecessary variables on memory.

7.5.1 Avoid creating extraneous variables

Creating unnecessary variables can take up wasted memory. Let's walk through an example using a sampled version of the ads dataset. In Listing 7.22, we create a function that does the following:

- Reads in the sampled ads file and stores the data into a variable called *ad_frame*
- Creates a new data frame (called *non_clicks*) to store only the records where the ad was *not* clicked
- Returns the *non_clicks* data frame

Listing 7.22. This code (available in the ch7/get_non_clicks_not_optimized.py file) reads in the sampled ads dataset, filters it down to non-clicks only, and returns a data frame storing the non-click observations.

```
import pandas as pd #1

@profile #2
def get_non_clicks(filename: str): #3

    ad_frame = pd.read_csv(filename) #4

    non_clicks = ad_frame[ad_frame.click == 0] #5

    return non_clicks #6

get_non_clicks("data/ads_data_sample.csv") #7
```

Now, let's run memory-profiler on Listing 7.22 so we can see how much memory is used in our *get_non_clicks* function.

Listing 7.23. Running the command in this listing executes memory-profiler on the code in

Listing 7.22.

```
python -m memory_profiler  
ch7/get_non_clicks_not_optimized.py #1
```

The result of memory-profiler is below:

Line #	Mem usage	Increment	Occurrences
4	88.648 MiB	88.648 MiB	1
5			
6			
7	458.305 MiB	369.656 MiB	1
8			
9	607.348 MiB	149.043 MiB	1
10			
11	607.348 MiB	0.000 MiB	1

Line #	Line Contents
4	@profile
5	def get_non_clicks(filename: str):
6	
7	ad_frame = pd.read_csv(filename)
8	
9	non_clicks = ad_frame[ad_frame.click == 0]
10	
11	return non_clicks

Based on the results, we can see that executing the *get_non_clicks* function has a peak memory usage of ~607MB.

Our updated code (in Listing 7.24) is almost identical to Listing 7.22, except now we replace *non_clicks* with *ad_frame* in line 5.

Listing 7.24. This code (available in the ch7/get_non_clicks_improved.py file) is an almost-duplicate of Listing 7.22. The only difference is in line 5 where we replace *non_clicks* with *ad_frame* when creating the filtered-down dataset.

```
import pandas as pd #1  
  
@profile #2  
def get_non_clicks(filename: str): #3  
  
    ad_frame = pd.read_csv(filename) #4
```

```

ad_frame = ad_frame[ad_frame.click == 0] #5
return ad_frame #6

get_non_clicks("data/ads_data_sample.csv") #7

```

Now, let's use memory-profiler again, but this time we will profile the updated *get_non_clicks* function. This will help us compare the memory usage between the two versions of *get_non_clicks*. Running the code in Listing 7.24 through memory-profiler prints the following output:

Line #	Mem usage	Increment	Occurrences	Line Contents
4	88.602 MiB	88.602 MiB	1	
5				
6				
7	458.211 MiB	369.609 MiB	1	
8				
9	576.723 MiB	118.512 MiB	1	
10				
11	576.723 MiB	0.000 MiB	1	

Line #	Line Contents
4	@profile
5	def get_non_clicks(filename: str):
6	
7	ad_frame = pd.read_csv(filename)
8	
9	ad_frame = ad_frame[ad_frame.click == 0]
10	
11	return ad_frame

As you can see (Line 11), this reduces the peak memory usage of the *get_non_clicks* function by around 30MB.

In general, if you create a variable and then no longer need it (or only use it once), then you should consider modifying your code to either avoid using the variable or minimizing its memory consumption if possible. Now, let's revisit global variables.

7.5.2 Avoiding global variables

In Chapter 3, we discussed how it is generally better to avoid using global variables unless necessary (for example, when creating a constant). Global variables will consume memory until your program execution is complete. This means using global variables can also be worse in terms of memory consumption. Keeping with the ads dataset, it is better to use functions or potentially classes to handle data processing, for example. This way, we can avoid allocating memory for global variables.

Next, let's walk through the memory usage of NumPy arrays vs. lists.

7.5.3 NumPy arrays vs. Lists

NumPy arrays are not only more computationally efficient than standard Python lists, they can also save memory resources. Let's take a look at the example in Listing 7.25. Here, we create a list of 100 million integers and a NumPy array of the same 100 million integers.

Listing 7.25. The code here generates a list of 100 million integers and an array of the same 100 million integers. We will use this to compare how much memory creating this list consumes vs. a NumPy array.

```
import numpy as np #1

@profile #2
def create_list(limit): #2

    return list(range(limit)) #2

@profile #3
def create_array(limit): #3

    return np.arange(limit) #3

limit = 100000000 #4
create_list(limit) #5
create_array(limit) #6
```

Now, let's compare the memory usage of creating the NumPy array vs. the list in order to see how NumPy arrays can be more memory-efficient than lists. In Listing 7.26 we run memory-profiler on the code from Listing 7.25.

Listing 7.26. Executing the command here will run memory-profiler on the list_vs_array_compare.py file.

```
python -m memory_profiler ch7/list_vs_array_compare.py
```

The results of the running memory-profiler are below:

Line #	Mem usage	Increment	Occurrences	Line Contents
4	55.840 MiB	55.840 MiB	1	@profile
5				def create_list(li
6				
7	1686.543 MiB	1630.703 MiB	1	return list(ra

Filename: ch7/list_vs_array_compare.py

Line #	Mem usage	Increment	Occurrences	Line Contents
9	61.137 MiB	61.137 MiB	1	@profile
10				def create_array(l
11				
12	824.078 MiB	762.941 MiB	1	return np.array

As you can see, creating the list uses roughly twice as much memory as creating the NumPy array! In general, NumPy arrays will be more memory-efficient than lists, particularly when the number of values being stored is large. The reason for the memory savings is that NumPy arrays only store homogenous values (the elements must be of the same data type, unlike lists) and the values are stored in memory in a continuous block. In Table 7.2, we summarize the memory usage of the code in Listing 7.25 with varying values for the *limit* variable (the size of the list or array). These values may vary slightly depending on your machine specifications, but the general trend should be the same - as a list gets larger, it becomes more and more memory-efficient to use a NumPy array if you need to store the values in memory.

Table 7.2. This table compares the memory usage (in MB) between lists and NumPy arrays based on the number of integers stored in each.

Size of list/array	List memory usage (MB)	Array memory usage (MB)

100,000	59	57
1,000,000	94	66
10,000,000	442	135
100,000,000	1687	824

Next, let's discuss an alternative file format that can greatly improve memory savings.

7.5.4 The parquet file format

In many of our examples, we've been reading in CSV files as the data source. An alternative file format to CSV is the *parquet* file format. While CSV files are still very common, *parquet* files are generally more memory efficient for large files. For large files, they can also be ingested faster than CSVs. Parquet files use a more optimized encoding scheme than CSV files (which helps with memory efficiency), and are also stored as binary files, rather than plain text. They also embed metadata about the data when stored, unlike CSV files. For example, parquet files will store the fact that a specific column has an *int* data type versus a string, etc.

Pandas can read in parquet files with syntax similar to reading in CSV files. Additionally, they can write data frames out to parquet files as well. Listing 7.27 shows an example where we read in data from a CSV file containing ads data, write it back out to a parquet file, and then ingest the data from the parquet file back into pandas. The method reading in a parquet file is `pd.read_parquet` (rather than `pd.read_csv`) and we use the *columns* parameter, rather than *usecols*, to specify individual columns we want to read in (if desired). Writing data our to a parquet file also has similar syntax to writing to a CSV file in that we use the `to_parquet` method (instead of the `to_csv` method).

Listing 7.27. This code reads in a few columns from the ads dataset, writes the results back out to a parquet file, and then reads the data into pandas from that new file. The time it takes to perform each of these tasks is printed out for comparison.

```
import pandas as pd #1
import time #1

start = time.time() #2
ad_frame = pd.read_csv("data/ads_train_data.csv",
    usecols = ("click", "banner_pos", "site_category"))
end = time.time()
print("Read in CSV file in ", end - start, " seconds")

start = time.time() #3
ad_frame.to_csv("ad_frame_train.csv",
    index = False)
end = time.time()
print("Write out to CSV file in ",
    end - start, " seconds")

start = time.time() #4
ad_frame.to_parquet("data/ads_train_data.parquet")
end = time.time()
print("Write out to parquet file in ",
    end - start, " seconds")

start = time.time() #5

ad_frame = pd.read_parquet(
    "data/ads_train_data.parquet",
    columns = ("click", "banner_pos",
    "site_category"))

end = time.time()

print("Read in parquet file in ",
    end - start,
    " seconds")
```

The results of running Listing 7.27 is shown below:

```
Read in CSV file in 50.98247313499451 seconds
Write out to CSV file in 68.19436407089233 seconds
Write out to parquet file in 6.056212902069092 seconds
Read in parquet file in 4.290560960769653 seconds
```

As we can see, reading in the same columns (with same number of rows) of the ads dataset is about 12x faster when the file format is a parquet file versus a CSV! That's a huge difference! Writing the same data *out* to parquet file is also much faster than writing to a CSV file. Additionally, the data stored on disk for the parquet file (~38MB) is much smaller than the CSV file (~526MB).

In the next chapter, we'll use the parquet file we just created to explore alternatives to pandas.

Lastly, let's briefly introduce those alternatives to pandas, which will be covered in more detail during the next chapter.

7.5.5 Alternatives to pandas

In the next chapter, we're going to cover alternatives to pandas. Pandas is a powerful package, and in this chapter we discussed several ways to optimize it for better memory consumption. However, there are alternatives which can more robustly handle large sets of data, depending on your machine specifications. For example, the Dask package is a popular alternative that can parallelize many operations in pandas. Another possibility is the modin library, which has very similar syntax to pandas, but allows for parallelization under the hood. An older library (older in 2023!) is PySpark, which leverages the Spark ecosystem to handle huge datasets. We'll discuss each of these in turn during the next chapter.

Now, let's summarize what we've learned this chapter.

7.6 Summary

In this chapter, we covered:

- Guppy and memory-profiler are two examples of profilers to analyze your memory usage. Additionally, we used the pandas builtin method `memory_usage` to check the memory consumption of data frames and individual columns.
- Use chunking or sampling (rows or columns) to reduce the amount of

memory consumed at once

- Choosing the right data types can have a significant impact on memory, especially when you have a large number of observations or large number of columns with suboptimal data types.
- Utilize scikit-learn's partial_fit functionality to train a logistic regression model in chunks
- Avoid creating unneeded variables - particularly global variables - that take up wasted space.

7.7 Practice on your own

1. Can you add a method to the Chunk_pipeline class that calculates evaluation metrics for the model?
2. Run memory-profiler on the new method. How does it perform?
3. Add additional feature engineering steps to the chunking workflow for the ads dataset. How do these extra features affect your memory consumption?

8 Alternatives to Pandas

This chapter covers

- Parallel machine learning training with Dask
- Exploratory data analysis and processing with PySpark
- Machine learning models training with PySpark
- Other alternatives to pandas

In the last two chapters, we've discussed scaling Python code in terms of both computational speed and dealing with memory challenges. However, many of the examples in Chapters 6 and 7 still rely largely on standard data science packages like pandas or sklearn. In this chapter, we will focus on taking scaling to the next level by learning about packages that can distribute many of the tasks in pandas or scikit-learn across either a single machine or a cluster of machines. This will allow for both computational speed improvements in terms of parallelization, and also the ability to process very large datasets by scaling the appropriate number of machines you need by the size of your data. The two primary packages we'll focus on are *Dask* and *PySpark*. We will also touch on a few others near the end of the chapter. There are several reasons why we need to consider moving beyond standard pandas or scikit-learn:

- Pandas uses a single CPU core by default. This means if you're processing 40 rows or 40 *million* rows of data, you'll have to deal with only using a single core. Even with the techniques we covered in the last two chapters, this can make it more difficult to continue to scale as data gets larger and larger.
- Alternatives to pandas, like Dask or PySpark, allow you to scale data operations across *machines*, as well as utilizing more of the cores on a single machine if you're just running them on your local computer or single remote server. This means continuing to scale out your data processing or model training operations is simply a matter of increasing the number of machines you need.
- Additionally, when pandas reads in a dataset, it directly stores the data

into your machine’s RAM. Even if you split a 40 million-row file into 1-million row chunks, each individual chunk will be read into memory at once.

Now that we have a better understanding why we need to seek tools that let us distribute common data science tasks in parallel, let’s get started by discussing how to use Dask. We’ll start by going through a few basics such as reading in and filtering a dataset. Then, we’ll use Dask to train a machine learning model to predict ad clicks, using the same dataset used in Chapter 7. For most of our examples in this chapter, we will use Google Colab. Google Colab makes it easy to setup PySpark, and we can use it for both Dask and PySpark (which requires Spark, a tool we’ll discuss in more detail in the PySpark sections this chapter, to be installed) so that we will have a fair comparison in terms of performance between the two. This tool functions similarly to Jupyter Notebook, but allows you to leverage Google’s servers, rather than relying on your own. You can get started using Google Colab for free through this link: <https://colab.research.google.com/>. Before we delve into using Dask or PySpark, we’ll also need to upload the ads dataset to Google Colab, which we can do by running the code in Listing 8.1. This code snippet creates a new folder called *data* on Google’s server, and then prompts the user to upload a file. From here, you can upload the ads dataset (found in *data/ads_train_data.parquet* in this book’s repo).

Listing 8.1. This is an example code snippet for uploading data to Google Colab. Running this code will show an button that you can use to upload a dataset (the ads data in this case).

```
import os #1
from google.colab import files
import shutil

new_folder = "data" #2
os.mkdir(new_data)

uploaded = files.upload() #3
for filename in uploaded.keys(): #4
    dst_path = os.path.join(new_folder, filename)
    shutil.move(filename, dst_path)
```

Running the code in Listing 8.1 will show the upload button (*Choose Files*) seen in Figure 8.1.

Figure 8.1. This snapshot shows the UI generated when you run the code in Listing 8.1 within Google Colab.

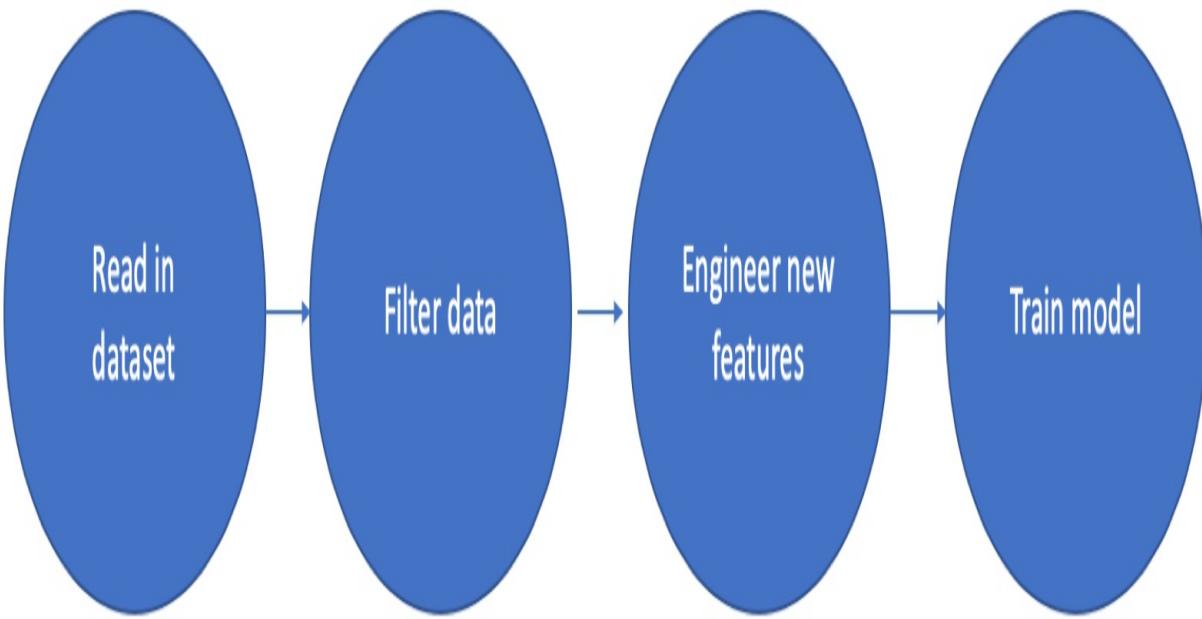


Now that we've uploaded the dataset to Google Colab, let's learn how to use Dask to scale our data science workflows.

8.1 Dask

Dask (<https://www.dask.org/>) is a popular alternative to pandas. It is a Python package that allows you to handle data processing, analysis, and model training on *large* datasets. Dask can work on datasets that do not fit in your RAM, unlike pandas. One of the key utilities of Dask is *lazy evaluation*. Lazy evaluation is a mechanism in programming where an expression is not immediately executed until the results of the expression are needed. For example, suppose you're dealing with the ads dataset from the previous chapter. We discussed ways of reading in the dataset using chunking, but an alternative to reading and processing the dataset is using lazy evaluation via Dask. Dask uses lazy evaluation in the form of a DAG (directed acyclic graph), as can be seen in Figure 8.2. In this snapshot, we see a workflow of reading in a dataset, applying a filter, creating new features, and then training a model. This workflow could be applied to the ads dataset, for instance. The DAG in Figure 8.2 represents how each component is stored as part of a workflow, but is not actually computed until needed.

Figure 8.2. This diagram shows a sample Dask workflow DAG, starting with reading in a dataset, followed by a few steps ending in training a model. Dask can store the steps in a workflow like this as a DAG. When a step in the DAG needs to output the results from previous steps, then the preceding steps in the DAG workflow will be executed.



Dask can also distribute processes across multiple machines. For example, it is possible to train a machine learning model on a large dataset across multiple machines, leveraging parallelization to make the jobs run faster (and potentially feasible to begin with, since some datasets would be too large to fit into memory on a single machine).

Now, let's install Dask using pip (see Listing 8.2). We will also install the *dask_ml* library, which is part of the Dask framework and provides additional functionality related to machine learning (ML).

Listing 8.2. Use pip to install dask and dask_ml.

```
pip install dask[complete] dask_ml #1
```

Now, let's read in the ads dataset using Dask in Listing 8.3. In our example, we'll limit the fields we need to a few key columns, such as *click*, *banner_pos*, and *site_category*.

Listing 8.3. This code snippet reads in the ads dataset using Dask.

```
import dask.dataframe as dd #1
```

```

import time

start = time.time()
ad_frame = dd.read_parquet( #2
    "data/ads_train_data.parquet",
    usecols=("click", "banner_pos",
    "site_category"))

end = time.time()
print(end - start)

```

Running the code in Listing 8.3 should take just a fraction of a second, as can be seen below. The exact runtime will vary between different machines and on the state of running processes on your machine, so you may get slightly different results when executing this code on your own.

0.021186113357543945

The caveat is that Dask doesn't actually read the dataset into memory when calling the *read_csv* method. It will not actually read the data into memory until you need to access the data. This is the concept of lazy evaluation in action! The opposite of lazy evaluation is known as *eager evaluation*, which is when an expression is evaluated as soon as it is called. Pandas generally uses eager evaluation, which is on display in Listing 8.4. There are some exceptions, such as when we used chunking to only read in subsets of the ads dataset as each subset is needed.

Listing 8.4. This code snippet reads in the ads dataset using pandas.

```

import pandas as pd #1
import time #1

start = time.time() #2
ad_frame = pd.read_parquet(
    "data/ads_train_data.parquet",
    columns=("click", "banner_pos",
    "site_category"))
end = time.time()
print("Read data in ", end - start, " seconds")

```

This code uses pandas to read in the ads dataset, rather than Dask. This code takes over four seconds to run on Google Colab, and around seven seconds

on local machine (using a mac with four cores). As an additional note, ingesting the CSV file version of the ads dataset takes almost a full minute to run on a Mac with four cores (running Listing 8.4 will print out the following):

```
Read data in 4.33516263961792 seconds
```

Now, let's suppose we want to get the subset of observations where a click occurred. In Listing 8.5, we use Dask to read in the ads dataset, and filter it down to only click-observations. Again, since Dask applies lazy evaluation, this code will run in a fraction of a second.

Listing 8.5. Filter ads dataset to get only clicks

```
import dask.dataframe as dd #1
import time

start = time.time() #2
ad_frame = dd.read_parquet(
    "data/ads_train_data.parquet",
    columns=("click", "banner_pos",
    "site_category"))

ad_frame = ad_frame[ad_frame.click == 1]

end = time.time()

print(end - start)
```

Running Listing 8.5 prints the following output (again, you may get slightly different values on different machines):

```
0.011932373046875
```

Next, let's go through a few data analysis examples using Dask.

8.1.1 Exploratory data analysis with Dask

Let's suppose we want to get the average click value by site_category (this will show the proportion of times an ad was clicked by site_category). We'll do that in Listing 8.6.

Listing 8.6. This code snippet reads in the ads dataset and calculates the proportion of ad clicks by site_category.

```
import dask.dataframe as dd #1
import time
from dask.distributed import LocalCluster, Client

cluster = LocalCluster() #2
client = Client(cluster) #3

start = time.time() #4
ad_frame = dd.read_parquet(
    "data/ads_train_data.parquet",
    columns=("click", "banner_pos",
              "site_category"))

print(ad_frame[[
    "click", "site_category"]].\
    groupby("site_category").\
    mean().\
    compute())

end = time.time()

print("Time to execute code: ",
      end - start)
```

Let's break down the code in Listing 8.6:

- First, we import the packages we need.
- Next, we create a LocalCluster object. A local cluster allows you to use the CPU cores available on your machine. Creating the client object (Line 3) lets you connect to this cluster. Any subsequent dask code that gets executed will be applied to this cluster. It is also possible to create a remote cluster, rather than a local one. A remote cluster allows you to scale your data science code across multiple machines.
- After our initial setup, we read in the ads dataset (Line 4) and filter it to only clicks (Line 5)
- Lastly, we use the *groupby* method to group the data by site_category and get the mean of the click field. Since this column is binary (1/0), taking the mean (average) will tell us the proportion of ad clicks by site category. Notice, that we append the *compute* method to the end of the

groupby statement. Using the compute method will calculate and return the groupby results at once, rather than applying lazy evaluation. If you don't want to see the results right away, you can avoid using the *compute* method so that Dask will still rely on lazy evaluation.

Executing the code in Listing 8.6 takes ~13 seconds to run. The code should output the following:

```
site_category
0569f928      0.053958
110ab22d      0.000000
28905ebd      0.208019
335d28a8      0.093644
3e814130      0.283003
42a36e14      0.231014
50e219e0      0.128580
5378d028      0.095238
70fb0e29      0.136146
72722551      0.057627
75fa27f6      0.111284
76b2941d      0.030261
8fd0aea4      0.014836
9ccfa2ea      0.012579
a818d37a      0.004025
bcf865d9      0.045933
c0dd3be3      0.111998
dedf689d      0.514000
e787de0e      0.073615
f028772b      0.179579
f66779e6      0.039687
74073276      0.142857
c706e647      0.000000
6432c423      0.000000
a72a0145      0.000000
da34532e      0.000000
```

Similar to pandas, we can also use the *describe* method to get descriptive statistics about a data frame, as you can see in Listing 8.7.

Listing 8.7. Similar to pandas, we can use Dask to calculate descriptive statistics about a data frame.

```
ad_frame.describe().compute() #1
```

We can also get counts of categorical variables, like `site_category`, similar to pandas as well. See Listing 8.8 for an example.

Listing 8.8. Similar to pandas, we can use Dask to calculate descriptive statistics about a data frame.

```
ad_frame.site_category.\ #1  
value_counts().compute()
```

Likewise, we can get the mean of a column almost just like pandas (see Listing 8.9). The only difference, again, is in using the `compute` method if we want to show the results right away.

Listing 8.9. Similar to pandas, we can use Dask to calculate descriptive statistics about a data frame.

```
ad_frame.click.\ #1  
mean().compute()
```

In addition to performing data analysis tasks, it's also common to create new columns in a data frame, or to randomly sample rows of data, prior to training a model. For example, you might want to create a new feature as a model input, or split your dataset into train and test. We can accomplish these tasks with Dask, as well, in a similar fashion to pandas.

8.1.2 Creating new features and random sampling with Dask

New columns can be created in Dask in basically the same way as pandas. Listing 8.10 shows an example where we create two new columns based on the `site_category` field.

Listing 8.10. We can create new columns in Dask the same way we do in pandas.

```
#1  
ad_frame["site_category_28905ebd"] =  
    ad_frame.site_category.map(lambda val:  
        1 if val == "28905ebd" else 0)  
  
ad_frame["site_category_50e219e0"] =  
    ad_frame.site_category.map(lambda val:  
        1 if val == "50e219e0" else 0)
```

Additionally, we can split our data frame into train and test using Dask, as well (see Listing 8.11). The process of splitting data into train and test is almost identical to scikit-learn’s *train_test_split* method.

Listing 8.11. We can split the ads dataset into train and test, very similarly to how we split data using scikit-learn’s `train_test_split` method.

```
from dask_ml.model_selection #1
    import train_test_split

train, test = train_test_split(ad_frame, #2
    train_size = 0.7,
    shuffle = True,
    random_state = 0)
```

The last few examples have covered common tasks that data scientists often need to perform, such as exploratory analysis, creating new features, and splitting a dataset into train and test. Now that we’ve gone through these code snippets, let’s delve into how to train a machine learning model with Dask! Our goal will be to build an ML model to predict whether an ad will be clicked.

8.1.3 Training a model with Dask locally

Dask is not just an alternative to pandas, it can also be used to train machine learning models. In this way, Dask serves as an extension of scikit-learn, allowing you to train ML models in a parallelizable and distributed fashion. In Chapter 7, we covered training a logistic regression model on the ads dataset by using chunking. While chunking is a great away to handle large datasets when you’re restricted to pandas and scikit-learn, not every algorithm can utilize chunking. For example, what if you want to train a random forest or gradient boosting model? The previous methods described in Chapter 7 will not work for these types of models. This is where Dask comes into the picture. In this section, we’ll introduce training a random forest model on the ads dataset. We’ll start by doing this locally (using a single machine). In the next section, we’ll show how to generalize the code to work across a collection of machines. The code for the initial model example is in Listing 8.12.

Listing 8.12. In this code, we train a random forest model on the ads dataset using Dask. This code combines previous examples of reading in the ads dataset, connecting to a local Dask cluster, creating new features, splitting the ads data into train and test, and finally training a random forest model to predict ad clicks.

```
from dask.distributed import LocalCluster, Client #1
import dask.dataframe as dd
from sklearn.ensemble import RandomForestClassifier
import joblib
from dask_ml.model_selection import train_test_split
import time

cluster = LocalCluster() #2
client = Client(cluster) #3

ad_frame = dd.read_parquet( #4
    "data/ads_train_data.parquet",
    columns=("click", "banner_pos",
              "site_category"))

ad_frame["site_category_28905ebd"] = #5
    ad_frame.site_category.map(lambda val:
        1 if val == "28905ebd" else 0)

ad_frame["site_category_50e219e0"] =
    ad_frame.site_category.map(lambda val:
        1 if val == "50e219e0" else 0)

inputs = ["banner_pos", #6
          "site_category_28905ebd",
          "site_category_50e219e0"]

train, test = train_test_split(ad_frame, #7
    train_size = 0.7,
    shuffle = True,
    random_state = 0)

with joblib.parallel_backend("dask"): #8
    forest = RandomForestClassifier(
        n_estimators = 100,
        max_depth = 2,
        min_samples_split = 40,
        verbose = 1)

    start = time.time()
    forest.fit(train[inputs], train.click)
```

```
end = time.time()
print("Time to train model = ",
      end - start, " seconds")
```

Let's break the code in Listing 8.8 down:

- First, we import the packages we'll need, including functionality from Dask and sklearn
- Next, we create the LocalCluster and Client objects, similar to our previous example (Lines 2 and 3)
- Then, we read in the ads dataset (Line 4)
- Create two new columns based on site_category (Line 6). These are binary fields based on two of the categories within the *site_category* field. For example, *site_category_28905ebd* will equal 1 if *site_category* equals *28905ebd* - otherwise, it will equal zero.
- Define the input features for the model (Line 7)
- Setup a parallelization job using the joblib package and Dask as the backend (Line 8)
- Create a RandomForestClassifier object with a few standard hyperparameter values
- Train the random forest model to predict ad clicks

Running this code in Google Colab's free tier with two available cores takes around 10 minutes. If you have a powerful machine, such as four, eight, or more cores, this code should finish faster.

Training other machine learning models using Dask is very similar to what we see in Listing 8.8. Now that we know how to train a random forest model using Dask on a single machine, let's walk through how to do it on a remote cluster. This will allow us to scale the size of the data we use for training.

8.1.4 Training a machine learning model on a remote Dask cluster

To train an ML model on a remote Dask cluster, you first need to have a Dask cluster setup and a cloud provider. AWS (Amazon Web Services), Azure, or Google Cloud are examples of common cloud providers. To setup the Dask cluster on your cloud provider, there are also several options. Two

popular options are Fargate and Coiled. Fargate is provided by AWS (<https://aws.amazon.com/fargate/>). Coiled is another service that offers the ability to setup Dask clusters through external cloud providers (like AWS or Azure, for example). In this example, we will use Coiled (see <https://www.coiled.io/> for additional reference). Appendix A has additional details on setting up a Coiled account and connecting it to AWS. The general workflow of the Dask code to train the random forest model will be similar regardless of which provider you use. See Listing 8.13 for an example of modifying our previous code in Listing 8.12 to work for a remote Dask cluster.

Listing 8.13. In this code, we train a random forest model on the ads dataset using Dask. This time, we use Coiled's service to train the model across a Dask cluster.

```
from dask.distributed import Client #1
import dask.dataframe as dd
import dask
import coiled
from sklearn.ensemble import RandomForestClassifier
import joblib
import time

cluster = coiled.Cluster( #2
    name="new",
    n_workers=8,
    worker_memory='8Gib',
    shutdown_on_close=False,
    package_sync = keep
)

client = Client(cluster) #3

ad_frame = dd.read_csv("data/ad_needed_columns.csv") #4

#5
ad_frame["site_category_28905ebd"] =
    ad_frame.site_category.map(lambda val:
        1 if val == "28905ebd" else 0)

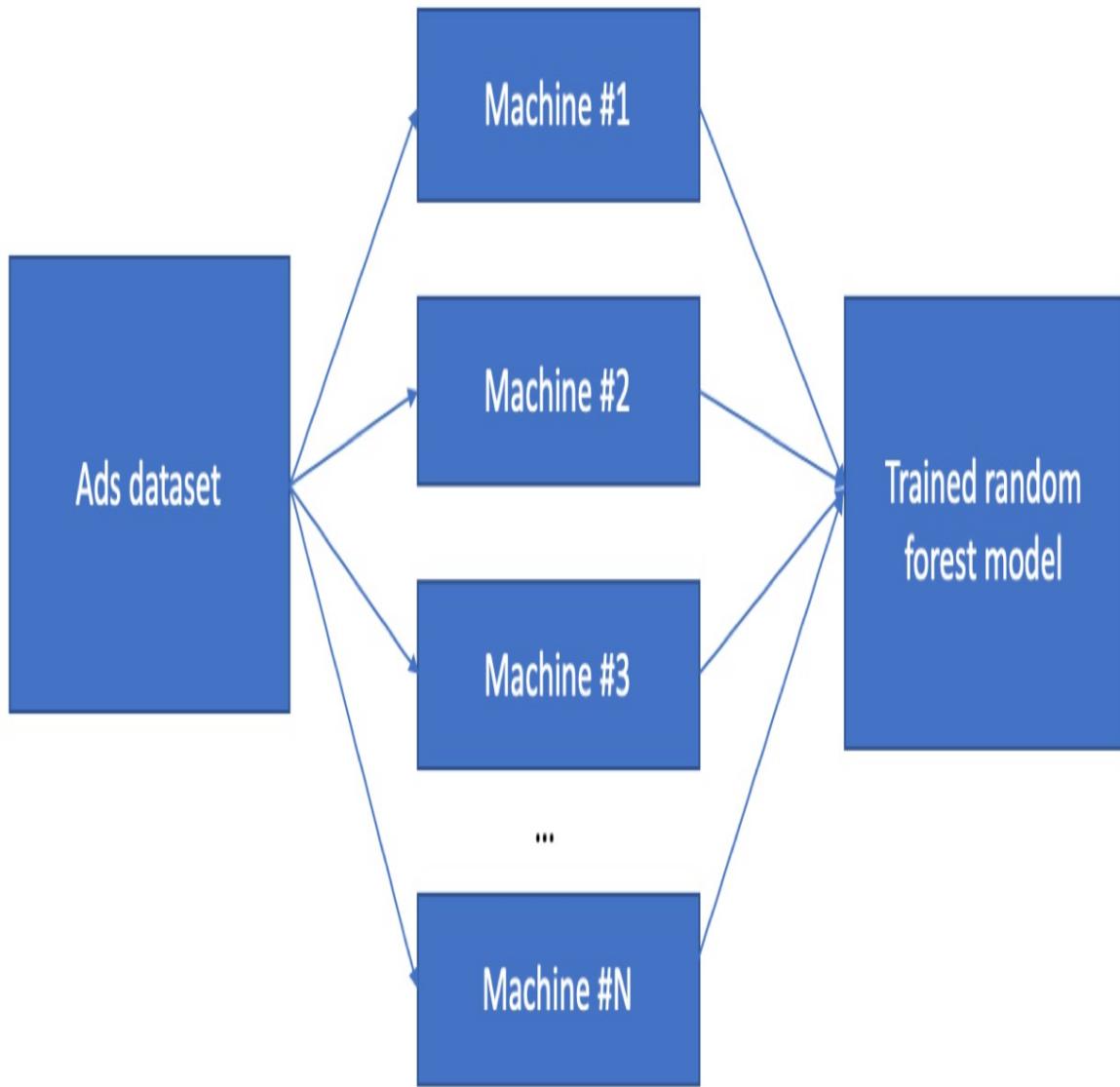
ad_frame["site_category_50e219e0"] =
    ad_frame.site_category.map(lambda val:
        1 if val == "50e219e0" else 0)
```

```
inputs = ["banner_pos", #6
          "site_category_28905ebd",
          "site_category_50e219e0"]

with joblib.parallel_backend("dask"): #7
    forest = RandomForestClassifier(n_estimators = 100,
        max_depth = 2,
        min_samples_split = 40,
        verbose = 1)
    forest.fit(ad_frame[inputs],
        ad_frame.click)
```

We can summarize the workflow of Listing 8.13 in Figure 8.2. Basically, the only difference in our code in Listing 8.13 vs. in Listing 8.12 is using Coiled's library to establish the remote Dask cluster.

Figure 8.3. This diagram visualizes the Dask workflow of consuming the ads dataset, and then launching a cluster of machines to train a random forest model on the data.



Now, let's recap what we've covered with Dask.

8.1.5 Summarizing Dask

To quickly recap, Dask offers the following key benefits:

- Dask can utilize parallelization, even if you're only using a single machine (on a single machine, Dask can still leverage all CPU cores)

available). However, it can also be scaled to use parallelization across a cluster of machines

- Dask is not only an alternative to pandas, but can also be used to parallelize the training of various machine learning models, including random forests and gradient boosting machines (GBMs)
- The difference between prototyping a code workflow using a local Dask cluster vs. a remote one across many machines is essentially one line - using *LocalCluster* versus leveraging a remote cluster object
- Dask uses lazy evaluation, meaning that expressions are only executed once they're needed

Now, let's delve into using PySpark. PySpark is an older alternative to Dask that can also help speed up your data science workflows. It has also more functionality than Dask, as we'll see in the next section.

8.2 PySpark

PySpark (<https://spark.apache.org/docs/latest/api/python/>) is older than Dask, but is still in use today. It allows you to leverage *Apache Spark* using Python syntax. *Spark* is an open-source tool that is used to scale data processing and modeling to large datasets. Spark allows you to run data processing code across a cluster of machines, rather than a single computer or server. Similar to Dask, this means you can typically process a large dataset much faster than relying only on pandas or standard Python. Spark was initially created in 2009.

In this section, we will use PySpark to train a random forest model on the ads dataset. We'll start by learning how to read in data with PySpark and using it to perform exploratory data analysis (EDA) on the ads dataset. First, let's go through our setup for getting PySpark working.

8.2.1 Setting up PySpark

Similar to our Dask examples, we will use PySpark on Google Colab. If you are looking to run PySpark in your local environment or remote server, you can follow the instructions found in Appendix A.

To install PySpark in a new Google Colab notebook, run the command in Listing 8.15.

Listing 8.14. Use pip to install PySpark.

```
!pip install pyspark #1
```

Now, let's get started with using PySpark and reading in the ads data.

8.2.2 Reading in data with PySpark

Before we can read in the ads dataset using PySpark, we need to connect to Spark within Python. We do that by creating a `SparkSession` object, as you can see in Listing 8.15.

Listing 8.15. This code creates a `SparkSession` object, which allows you to connect to Spark from within a Python script/session.

```
from pyspark.sql import SparkSession #1

spark = SparkSession \
    .builder \
    .getOrCreate()
```

Now that we've created the `SparkSession` object, we can use it to read in the ads dataset. The syntax for this is fairly similar to pandas - we'll just use `spark.read.parquet` and input the filename (see Listing 8.16). Reading in a CSV file would be very similar, except you would use `spark.read.csv` instead.

Listing 8.16. Use the `SparkSession` object we created in Listing 8.15 to read in the ads dataset.

```
#1
ad_frame = spark.\
    read.\
    parquet("./data/ad_sample_data.parquet",\
    header = True)
```

Similar to Dask, PySpark uses lazy evaluation. This means when you run the code in Listing 8.16, the dataset is not immediately read into memory. Now, let's walk through a few examples of exploratory analysis using PySpark.

8.2.3 Exploratory data analysis with PySpark

First, let's get the overall proportion of clicks. To do this, we first read in the ads dataset. Then, we use the `select` method to refer to only the `click` column, and use the `mean` method to calculate the mean of the column. The `collect` method we append on the end of Line 3 will force the evaluation of the mean click calculation. If we don't use this, PySpark will utilize lazy evaluation, and the expression in Line 3 will not be evaluated unless later called upon to be computed (for example, using the `collect` method in a later line of code, which works similarly to Dask's `compute` method).

Listing 8.17. Use the `SparkSession` object we created in Listing 8.15 to read in the ads dataset.

```
from pyspark.sql import SparkSession #1
from pyspark.sql.functions import mean

ad_frame = spark.read.parquet( #2
    "./data/ad_sample_data.parquet",
    header = True)

ad_frame.select(mean("click")).collect() #3
```

To group one variable by another, you can use similar syntax to pandas. In Listing 8.18, we group the ads data by `site_category`, and then calculate the mean of the `click` field by each category. The `.show` method at the end of Line 1 is necessary to generate the output. If you don't append the `show` method, PySpark will use lazy evaluation, and will not compute the group by results right away. You can also use `collect` as well, as mentioned previously. The difference in this case is that the `show` method will only print the output, whereas `collect` will

Listing 8.18. This code snippet calculates the average click value (proportion of clicks) by `site_category`.

```
ad_frame.\ #1
groupBy("site_category").\
mean("click").\
show()
```

Similarly, we can also group a field by multiple variables. In Listing 8.19, we

group the ads data by site_category and banner_pos.

Listing 8.19. This code snippet calculates the average click value (proportion of clicks) by site_category and banner_pos.

```
ad_frame.\ #1
groupBy(["site_category", "banner_pos"]).\
mean("click").\
show()
```

We can also get descriptive statistics using the *describe* method, similar to pandas. An example of this is in Listing 8.20.

Listing 8.20. Similar to pandas, we can get descriptive stats for a data frame using the describe method.

```
ad_frame.\ #1
describe().\
show()
```

Now that we've covered a few ways of doing basic data analysis, let's walk through how to create new columns using PySpark. Creating new columns is a very common task in data science applications, such as when you need to create new features.

8.2.4 Creating new columns with PySpark

Creating new columns for your data frame is a frequent task. There are several ways to create new data frame columns in PySpark. One fairly common method for doing so is using SparkSQL. SQL (Structured Query Language) is a common language used for extracting data from database tables. SparkSQL provides a mechanism for querying data within a Spark environment using SQL-like syntax, such as the ads data frame that we are currently working with.

Listing 8.21. In PySpark, we can use SQL syntax to create new columns. In this example, we create two new binary columns based on the site_category field.

```
ad_frame.createOrReplaceTempView("ad_frame_table") #1
```

```

sql_command = """SELECT *, #2
CASE WHEN site_category = '28905ebd' THEN 1
ELSE 0
END AS site_category_28905ebd,
CASE WHEN site_category = '50e219e0' THEN 1
ELSE 0
END AS site_category_50e219e0
FROM ad_frame_table"""

new_ad_frame = spark.sql(sql_command)
new_ad_frame.show() #3

```

The first few rows of running Listing 8.21 looks like below:

click	banner_pos	site_category
0	0	28905ebd
0	1	0569f928
0	0	f028772b

site_category_28905ebd	site_category_50e219e0
1	0
1	0
1	0
1	0
0	0
0	0

Another way of creating new columns is using the `withColumn` method. For example, suppose we want to create a new column that is the product of the `click` and `banner_pos` fields. Listing 8.22 shows an example of doing this.

Listing 8.22. An alternative to using SparkSQL to create columns is using the `withColumn` method.

```

ad_frame = ad_frame.\ #1
    withColumn("click_banner_pos_interaction",
               ad_frame.click * ad_frame.banner_pos)
ad_frame.show() #2

```

The results of running Listing 8.22 is below (restricted to the first few rows):

click	banner_pos	site_category	click_banner_pos_interaction
0	0	28905ebd	0
0	0	28905ebd	0
0	0	28905ebd	0
0	0	28905ebd	0
0	1	0569f928	0

That covers creating new columns in PySpark. Another common data science task is to randomly sample datasets. We can also leverage PySpark to do this for large datasets, like the ads dataset.

8.2.5 Randomly sample large datasets with PySpark

PySpark also supports randomly sampling data frames, which can be very helpful when you need to randomly sample large datasets (for example when training a machine learning model). An example of randomly sampling the ads dataset is shown in Listing 8.23.

Listing 8.23. PySpark data frames can be randomly sampled similar to pandas, using the `sample` method as shown in this code snippet. Using `fraction = 0.1` will randomly sample 10% of the dataset. You can adjust this value between 0 and 1 to get whatever proportion of the data you want to sample.

```
ad_subset = ad_frame.sample( #1
    seed = 123,
    fraction = 0.1).\
    collect()
```

One key difference between using the `sample` method in PySpark versus using pandas is that the PySpark will return a list of Row objects, rather than directly returning a data frame (like in pandas). This output from PySpark looks like below (showing the first five elements of `ad_subset`):

```
[Row(click=0, banner_pos=0,
      site_category='76b2941d',
      click_banner_pos_interaction=0),
Row(click=1, banner_pos=0,
      site_category='50e219e0',
```

```

    click_banner_pos_interaction=0),
Row(click=0, banner_pos=0,
    site_category='f66779e6',
    click_banner_pos_interaction=0),
Row(click=0, banner_pos=1,
    site_category='50e219e0',
    click_banner_pos_interaction=0),
Row(click=0, banner_pos=1,
    site_category='f028772b',
    click_banner_pos_interaction=0)]

```

Each Row object is analogous to a row in a data frame. To convert to a proper data frame, we can use the *createDataFrame* method, as shown in Listing 8.24. To do this, we just need to input the *ad_subset* variable (list of Row objects) into the *createDataFrame* method.

Listing 8.24. Use the `createDataFrame` method to convert the list of Row objects sampled in Listing 8.19 to a PySpark data frame.

```

ad_subset_frame = spark.createDataFrame( #1
    ad_subset)
ad_subset_frame.show() #2

```

The first few records of the results of Listing 8.24 is shown below.

click	banner_pos	site_category	click_banner_pos_interaction
0	0	76b2941d	0
1	0	50e219e0	0
0	0	f66779e6	0
0	1	50e219e0	0
0	1	f028772b	0

Another key difference in random sampling using PySpark versus pandas is that it is easier (with simpler syntax) to scale PySpark sampling to large datasets versus using chunking in pandas. In addition to the method we just covered, a second way of random sampling in PySpark is to directly split a dataset into subsets - such as splitting a data frame into train and test. We can do this using the *randomSplit* method, as you can see in Listing 8.25.

Listing 8.25. This code snippet will randomly split the ads dataset into two subsets, analagous to scikit-learn's `train_test_split` method.

```
train_data, test_data = ad_frame.\ #1
    randomSplit([0.7, 0.3],
    seed = 123)
```

Now that we've covered a few key topics around exploratory data analysis, creating new features, and random sampling, let's delve into training machine learning models with PySpark. We'll leverage what we just learned to randomly split the ads dataset into train and test so that we can train a random forest model using PySpark.

8.2.6 Training a machine learning model with PySpark

In this section, we will walk through how to train a random forest model using PySpark. Our goal will be to train a random forest to predict ad clicks using the ads dataset. First, let's cover how to prepare the dataset for training the model.

To prepare the input dataset needed for training (see Listing 8.22), we will use the *VectorAssembler* class to merge multiple numeric columns from our ads data frame into a single vector column. A *vector column* is a column that stores a collection of features as an array (or vector) for each observation. For example, instead of having three separate columns corresponding to three individual features, a vector column will consist of a single column made up of three-element arrays (vectors). This vector column will then be input into the random forest model. Listing 8.21 shows how to use the *VectorAssembler* class to create the new vector column. In the code, we specify the input columns we want to include and a name for the output vector column (which we call *features* in this example). Then, we use the *transform* method to apply the *VectorAssembler* to the ads data frame. Using the *show* method will print out the first few rows of the updated data frame.

Listing 8.26. In this code snippet, we use the *VectorAssembler* class from PySpark that lets you merge multiple columns into a single vector object. This vector object will be used as input into the random forest model.

```
from pyspark.ml.feature import VectorAssembler #1
assembler = VectorAssembler( #2
    inputCols=["banner_pos",
    "site_category_28905ebd",
```

```

    "site_category_50e219e0"],
outputCol="features")

ad_trans_frame = assembler.\ #3
    transform(new_ad_frame)

ad_trans_frame.show() #4

```

Running the code in Listing 8.26 will show the following output:

```

+-----+-----+-----+
|click|banner_pos|site_category|site_category_28905ebd|
+-----+-----+-----+
|  0|      0| 28905ebd|          1|
|  0|      1| 0569f928|          0|
+-----+-----+-----+
|site_category_50e219e0|      features|
+-----+-----+
1|          0|[0.0,1.0,0.0]|
1|          0|[0.0,1.0,0.0]|
1|          0|[0.0,1.0,0.0]|
1|          0|[0.0,1.0,0.0]|
0|          0|[1.0,0.0,0.0]|

```

Now, let's split the dataset into train and test using the *randomSplit* method. Listing 8.27 splits out 70% of the data for training and 30% for testing.

Listing 8.27. This code splits the ads data into train and test using the randomSplit method. We'll use 70% for training data, and 30% for test.

```

train_data, test_data = ad_trans_frame.\ #1
    randomSplit([0.7, 0.3], seed = 123)

```

Now, let's use the training dataset to train a random forest model to predict whether an ad will be clicked. We can break the code to do this like so (see Listing 8.28):

- First, create a RandomForestClassifier object. This object will take in the input features and label for our dataset. Additionally, you can set hyperparameters for the random forest. In our example, we use 100 trees

and a max depth of 4.

- Second, we use the RandomForestClassifier object to train a random forest model using the train dataset.
- Lastly, we use the fitted model to fetch predictions based on the test data. These are stored in a variable called *test_pred*.

Listing 8.28. This code starts by creating a RandomForestClassifier object. This takes a few inputs, such as the features and label for the model, along with hyperparameters like the number of trees and max tree depth. Then, we train the random forest model and fetch the predictions from it based on the test set.

```
from pyspark.ml.classification
    import RandomForestClassifier #1
import time #1

#2
random_forest = RandomForestClassifier(
    featuresCol = "features",
    labelCol = "click",
    numTrees = 100,
    maxDepth = 2)

start = time.time()
forest_model = random_forest.fit(train_data) #3
end = time.time()
print("Time to train model = ",
      end - start,
      " seconds")

test_pred = forest_model.transform(test_data) #4
```

Use the *test_pred* variable we just created, we can take a peak at the predictions versus the ground truth values from the click field (see Listing 8.29).

Listing 8.29. This code shows the first few predictions along with the predicted probabilities and actual values for the click field.

```
test_pred.select("click", #1
                 "probability",
                 "prediction").\
show(5)
```

That covers PySpark! You can learn more about PySpark from Manning's

book *Data Analysis with Python and PySpark* (see here: <https://www.manning.com/books/data-analysis-with-python-and-pyspark>).

In the next section, let's briefly introduce a few other alternatives to pandas.

8.3 Other alternatives to pandas

While Dask and PySpark are two of the most popular alternatives to pandas, there are a few other packages available that also serve as alternatives. For example, *modin* (<https://modin.readthedocs.io/en/stable/>) is another pandas alternative to speed up functions while keeping the syntax largely the same. Modin works by utilizing the multiple cores available on a machine (like your laptop or a remote server, for instance) to run pandas operations in parallel. Since most laptops have between four and eight cores, this means you can still have a performance boost on a local machine even without using a more powerful server. Let's discuss modin in more detail next.

8.3.1 Using modin to parallelize pandas

First, let's install modin using pip (see Listing 8.30). For this step, we're going to install all the dependencies, which includes Dask and Ray. These will not be installed if you leave out the "[all]" piece of the installation command.

Listing 8.30. Use pip to install modin. Make sure to use modin[all] to setup the dependencies needed.

```
pip install modin[all] #1
```

Once you have modin installed, you can use it almost just like pandas. The main difference in terms of syntax is during package import. We can use pandas syntax via modin by running the package import in Listing 8.24.

Listing 8.31. To use modin's pandas-like syntax, simply change your usual pandas import to the one in this listing.

```
import modin.pandas as pd
```

After running this import, most pandas operations will now be parallelized using either Dask or Ray (another pandas alternative: <https://www.ray.io/>) as the backend. For example (Listing 8.32), you can read in a parquet file with the same syntax as pandas.

Listing 8.32. This code snippet reads in the ads dataset using modin. Note that this is essentially identical to using pandas (the only difference is the import statement).

```
import modin.pandas as pd #1

#2
ad_frame = pd.read_parquet(
    "data/ads_train_data.parquet",
    columns=("click",
              "banner_pos",
              "site_category"))
```

Similarly, if you want to perform basic data analysis, you could write the same syntax as pandas. Listing 8.33 shows a few examples:

- Line 1 calculates the mean of the click column (again, this will equate to the proportion of clicks)
- Line 2 calculates the proportion of clicks by site_category
- Line 3 computes descriptive statistics about the ads dataset

Listing 8.33. This code snippet reads in the ads dataset using modin. Note that this is essentially identical to using pandas (the only difference is the import statement).

```
ad_frame.click.mean() #1

ad_frame.groupby("site_category").mean()["click"] #2

ad_frame.describe() #3
```

Besides pandas functionality, modin also has experimental functionality replicating parts of scikit-learn. You can learn more about modin through its documentation here: <https://modin.readthedocs.io/en/stable/>.

Now, let's briefly discuss a few other libraries.

8.3.2 Ray, Polars, and beyond

As mentioned during the modin discussion, *Ray* is another alternative to pandas designed so that you can write similar code for dealing with small datasets (thousands of rows) to large datasets (millions or billions of rows). *Ray* can also serve as the backend of *modin*.

Another alternative is the *Polars* library, which is written in Rust. This package is designed for working with very large data frames.

Ultimately, the choice of what package or tool you use is dependent on several factors:

- Are you handling small datasets with little chance of scaling up? Then, pandas might be the most useful.
- Do you have a cluster of machines like AWS or Azure at your disposal? Then, Dask, PySpark, or Ray could be good choices.
- Do you want to stick with pandas syntax, but also enable parallelization in your code? Modin could be a good choice.

8.4 Summary

In this chapter, we covered the following key points:

- Dask, PySpark, Modin, and Ray are potential alternatives to pandas. Each allow you to utilize all the cores on your machine, and can also work across clusters of machines. This means you can scale up any data operations you have depending on your needs.
- PySpark allows you to utilize the power of Spark using Python syntax. PySpark leverages Apache Spark as its backend, while letting the user write Pythonic syntax rather than Scala (which is the native language associated with Spark).
- Dask and PySpark offer alternatives not just to pandas, but also have quite a bit of functionality for machine learning development. For example, you can use either of these tools to train models such as random forests, GBMs, or neural networks.
- Dask and PySpark both use lazy evaluation, meaning that expressions are generally not evaluated unless needed or explicitly called to evaluate. This saves memory resources and can save compute time as well.

- Modin's syntax is almost identical to pandas, except that it allows you to run pandas operations using parallelization

8.5 Practice on your own

1. Can you use Dask to train a GBM on the ads dataset to predict ad clicks?
2. Using PySpark, can you get the counts of each categorical variable in the ads dataset?
3. Try creating a few new features from the ads data using PySpark. Then train a random forest on the updated feature set. What's the model performance?

9 Putting your code into production

This chapter covers

- Pipeline to store model predictions in a database table
- Credentials protection in code
- Application Programming Interfaces

The concepts of this chapter are important for many applications where you need to put data science-related code into production. Consider the following scenarios:

- Revisiting the customer churn problem from earlier chapters, suppose you want to run a machine learning (ML) model predicting whether a customer will churn as a daily job. Each day, the model will run across the set of customers, and output the associated predictions. A common way to integrate this model would be to output the predictions to a database table. We'll cover how to do that in the next section.
- Your code makes use of a password or secret key. In this case, it is a bad idea to ever expose credentials directly in your code. Instead, we can utilize a few tools to protect your credentials.
- You need to execute Python code from another language. For example, you might train a machine learning model in Python, and need to fetch predictions from the model in another language (such as Java, for example). This is an example where could convert your code into an Application Programming Interface (API) or command-line interface (CLI) application. This can be especially useful if the model you need to call and the application you are using to call it from live on different servers.

Let's get started by delving into more details around putting machine learning models into production. As we've discussed in earlier chapters, being able to do this is often critical for making sure your work as a data scientist or machine learning engineer is fully utilized. Deploying models into production means enabling them to be used by others in a (ideally) reliable way. Now,

let's discuss a few of the common methods for putting models, data or ML pipelines, and other types of code into production.

9.1 Putting a machine learning model into production

As we discussed in Chapter 1, as a data scientist, you often want to see your ML model or application used in production. There are several ways of implementing a machine learning model in production. The methods listed below are among the most common:

- Schedule a recurring job that runs the model on a set of data, and outputs the predictions to a database table. For example, suppose you want to deploy an ML model that predicts the final cost of an insurance claim. Each day, you may receive additional information about the claim, which will be input into the model to get an updated prediction. The prediction with the most recent date could be used for tracking high priority claims.
- Create an API that handles fetching predictions from the model. This is one of the most common ways of handling situations where you need to fetch model predictions in real-time. For example, if you need to predict whether someone will click on an ad, having a real-time model is essential.
- Create a CLI application for your code. Similar to creating APIs, this can be useful in situations where you are working with multiple programming languages. An example would be if you develop a machine learning model in Python, but need to call it from Java. Creating CLI applications can also be useful when you need to run a separate script from within a function.
- Designing a desktop application that other users can download. The application can hide the code from the end users, but allow them to use its benefits such as fetching model predictions. An example of this could be creating an application that performs image recognition.
- Packaging your code into a downloadable library that can be accessed by others. For instance, if you developed a codebase that has specific functions training or evaluating machine learning models (like scikit-

learn), you could convert this code into a package and upload the package to PyPI so that it can be downloaded and used by others.

We will cover the first three of these items in this chapter. The remaining two items will be tackled in Chapter 12. To be clear, the points we have raised here can apply across many different types of applications - not restricted to ML models or ML pipelines only. For example, setting up a data pipeline purely for reporting purposes is also a very valid and common task that may be automated to run on a daily basis. Or, creating a desktop application like Excel or PowerPoint, which doesn't directly relate to data science, is a form of putting a codebase in production where it is used by millions of people.

Let's get started by covering how to create a prototype machine learning pipeline.

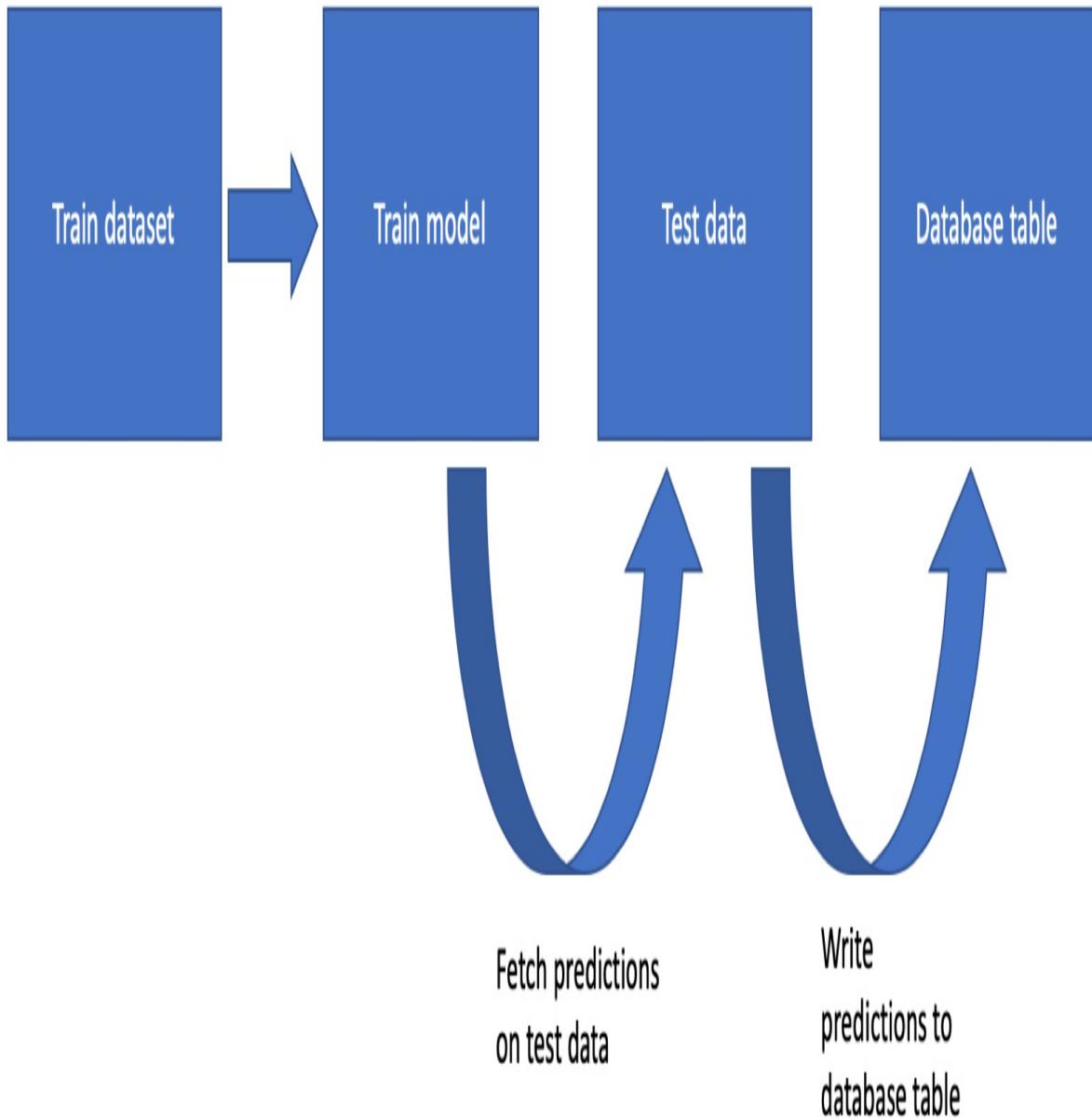
9.2 Creating a machine learning pipeline

In this section, we will discuss how to output model predictions to a database table. The examples in this section will use the customer churn dataset utilized earlier in the book. Our initial goal will start simpler. We will setup a pipeline that does the following:

- Ingests the customer churn data
- Trains a gradient boosting model (GBM) on a training dataset
- Fetches the model predictions on a separate (test) dataset
- Writes the predictions out to a database table

After we create this pipeline, we will work toward generalizing it so that we can have a proper functioning ML pipeline. In Chapters 10 and 11, we will also expand upon the work here to develop rigorous tests for this pipeline and to schedule it to run automatically. A workflow of our initial goal is in Figure 9.1:

Figure 9.1. This diagram demonstrates our initial workflow goal where we will create a pipeline that trains a model based on an input dataset. Then, we will fetch predictions on a separate test dataset, and write the predictions out to a database table.



Now, let's break down the process we need to go through to create the ML pipeline corresponding to the customer churn prediction problem.

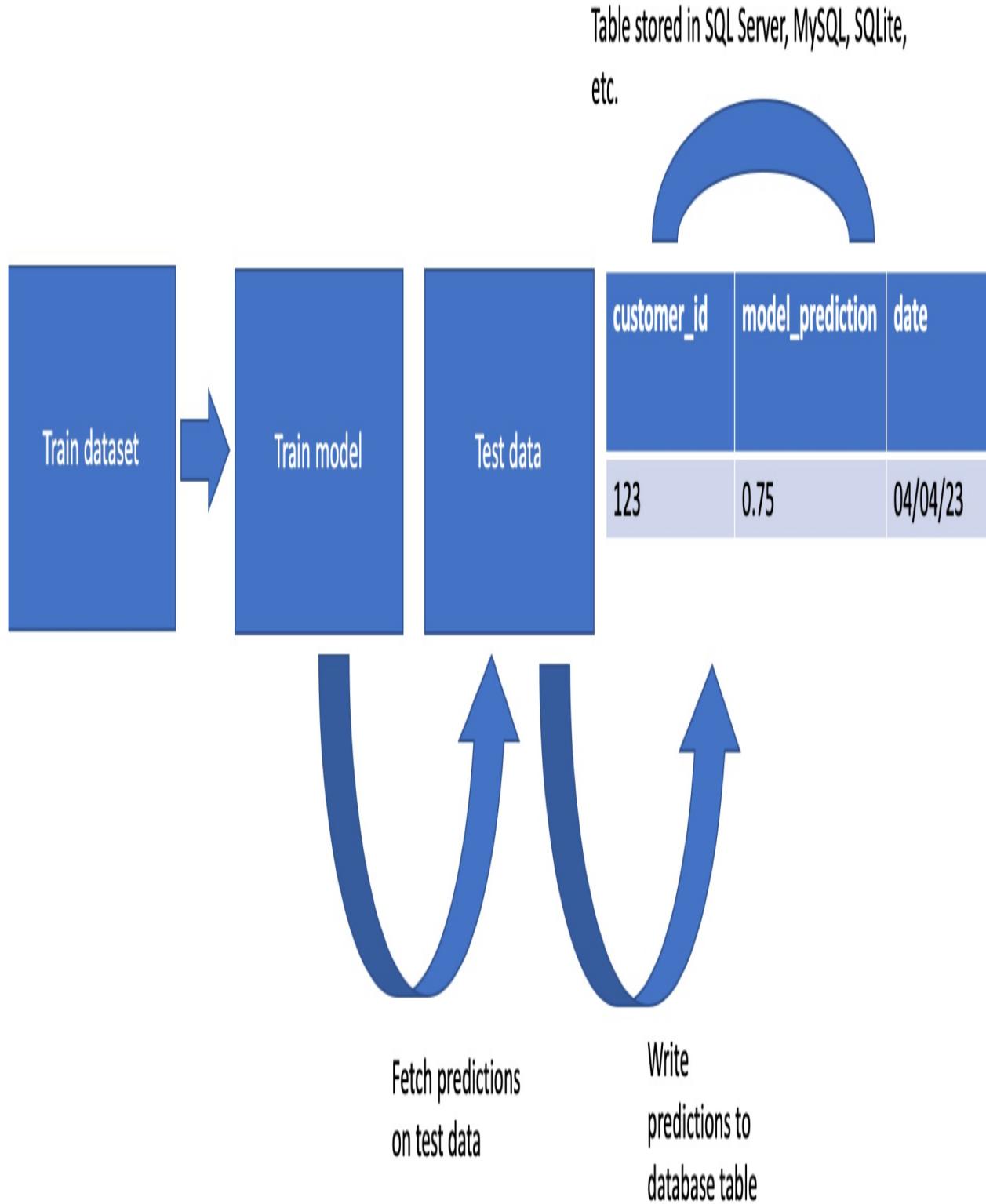
9.2.1 Creating a prototype pipeline

As we create the ML pipeline, our final product will be to upload the

customer churn model predictions into a database table. Let's first give a brief overview of databases and how you might interact with them in production use cases.

A *database* is a structured system to store data. Depending on the size of the data, the database could be stored entirely locally or (as in many real-world cases) distributed across a cluster of servers. A *Relational Database Management System (RDMS)* is a database system that structures data in a tabular, relational format. If you've used services like SQL Server, MySQL, Oracle, PostgreSQL, or SQLite, then you've already interacted with an RDMS. Let's update Figure 9.1 to include the workflow of writing out to one of these sample database systems. The updated diagram (see Figure 9.2) also shows the sample tabular structure of the output dataset, which includes columns for the date, model prediction, and customer ID.

Figure 9.2. This diagram extends Figure 9.1 to also show the sample output table, which can be stored in an RDMS, like MySQL, SQL Server, SQLite, etc. The view shows a sample record of the data, with values for customer_id, model_prediction, and date (prediction date).



In a production setting, we would probably have a scheduled job that fetches model predictions on a recurring basis. For example, each day the ML pipeline runs and uploads the latest customer churn predictions to the output database table. We’re going to tackle scheduling in Chapter 11, but for now let’s focus on setting up the offline pipeline structure.

Before we write the prototype pipeline code, let’s first create an output table for use in SQLite. As mentioned previously, SQLite is an example RDMS. We will use SQLite for the examples in this section because it is free and easy to setup. It is very lightweight, and in most production settings, you will most likely use a different database system (again, this could be MySQL, SQL Server, etc.), but using SQLite for these examples is easier because we don’t need to worry about installing or setting up one of the other database systems. Additionally, you could also follow a similar process for this section’s examples with other database systems like MySQL, SQL Server, or Oracle as well (assuming you have access to one of these). In other database systems, the syntax and data types might be slightly different, but the overall logic of what we will walk through will be very similar. If you’re using a Mac to run these examples, it should come with SQLite installed by default. Otherwise (for example, if you’re using Windows), you can install SQLite from this link: <https://www.sqlitetutorial.net/download-install-sqlite/>.

For now, the table we create will be empty, but we will use it to upload the customer churn predictions data after we train an ML model. In SQLite, we can do that either by using a UI for SQLite, or directly from the SQLite command line in the terminal. For example, to create the table from the terminal, you can type in *sqlite3* followed by the name of the database you want to connect to. If you enter a database name that doesn’t exist, SQLite will create that database.

Listing 9.1. Running this code from the terminal will create the customers database if it doesn’t already exist. If it does exist, SQLite will merely open a connection to the database.

```
sqlite3 customers #1
```

Next, we can create a new table to store the model predictions using the *CREATE TABLE* SQL command. In this command, we specify the name of the output table, the columns we want the table to have, and the

corresponding data types for those columns. We will specify that customer_id needs to be an integer (or *INT*) data type, that model_prediction needs to be a *REAL* (this may be a *double* in other database systems), and that the output_date field needs to be a *TEXT* data type (other database systems may have a formal *date* type).

Listing 9.2. This code snippet will create the `customer_churn_predictions` table

```
CREATE TABLE customer_churn_predictions( #1
    customer_id INT,
    model_prediction REAL,
    prediction_date TEXT
)
```

Now that we've created an empty table to store our model predictions, we can go on with creating our first version of an ML pipeline.

The code in Listing 9.3 shows our initial attempt at creating an ML pipeline.

- We start by reading in the customer churn dataset. Then, we create a DataSet object and specify the label and positive category using the DataSet class we created back in Chapter 4.
- Next, we create a GradientBoostingClassifier object to handle training the GBM model. For now, we setup a few reasonable hyperparameters
- Then, we train the GBM model
- Next, we create an output data frame. This output data frame stores the model predictions on the test dataset, along with a date field that will represent the upload date when we store the data in the *customers* database.
- Lastly, we upload the data frame to the customers SQLite database.

Listing 9.3. This code starts by reading in the customer churn dataset. Then it creates a DataSet object using the class we created back in Chapter 4. Next, we train a GBM on the dataset, and fetch the predictions for a separate test set. After this, we create an output data frame with the predictions, customer ID, and date of prediction. Lastly, we upload this output data frame to SQLite so that the results could be queried outside of our Python session.

```
import pandas as pd #1
import sqlite3
from sklearn.ensemble import GradientBoostingClassifier
from ch4.dataset_class_final import DataSet
```

```
customer_data = pd.read_csv("data/
    customer_churn_data.csv") #2

customer_obj = DataSet( #3
    feature_list =
        ["total_day_minutes",
        "total_day_calls",
        "number_customer_service_calls"],
    file_name = "data/customer_churn_data.csv",
    label_col = "churn",
    pos_category = "yes"
)

gbm_model = GradientBoostingClassifier( #4
    learning_rate = 0.1,
    n_estimators = 300,
    subsample = 0.7,
    min_samples_split = 40,
    max_depth = 3)

gbm_model.fit(customer_obj.train_features, #5
    customer_obj.train_labels)

output = pd.DataFrame([index for index in #6
    range(customer_obj.test_features.shape[0]) ],
    columns=["customer_id"])

output["model_prediction"] = gbm_model.\ #7
    predict_proba(customer_obj.\
    test_features)[:,1]

output["prediction_date"] = "2023-04-01" #8

sqliteConnection = sqlite3.connect('data/customers') #9
cursor = sqliteConnection.cursor() #10

insert_command = """insert into #11
    customer_churn_predictions(
    customer_id,
    model_prediction,
    prediction_date)
    values (?,?,?)"""

cursor.executemany(insert_command, #12
```

```
    output.values)

sqliteConnection.commit() #13
```

The output data frame object from Listing 9.3 is uploaded into a table called `customer_churn_predictions` (see Line 11). To upload the data to the SQLite database, we first need to connect to the database (Line 9). Then we create a `cursor` object (Line 10). A `cursor` is an object that lets you iterate over a collection of rows for a database table. In Line 11, we define a SQL statement to insert rows into the `customer_churn_predictions` table. Line 12 uses this statement to write the rows of the output data frame to the SQLite table. Lastly, we use the `commit` method in Line 13 to save our changes to the database table. **If you don't include this `commit` line, the updates you make to the table will be lost.**

Now that we've written the test output data to SQLite, we can query the data either in Python or directly from SQLite. Listing 9.4 shows how to query the data from within Python:

Listing 9.4. This code reads in the model predictions from SQLite, still using the `sqlite3` package.

```
select_query = """SELECT * FROM #1
                  customer_churn_predictions"""
cursor.execute(select_query) #2
customer_records = cursor.fetchall() #3
cursor.close() #4
```

Printing the first few elements of `customer_records` will show the following:

```
[(0, 0.38438860309249445, '2023-04-01'),
 (1, 0.06202837584700288, '2023-04-01'),
 (2, 0.31594775024747573, '2023-04-01'),
 (3, 0.02697966903960774, '2023-04-01'),
 (4, 0.024514897613872592, '2023-04-01')]
```

Querying the results using the method in Listing 9.4 returns a list of tuples containing the `customer_churn_predictions` data. We can also read the data directly into a data frame using pandas. The example to do this is in Listing 9.5.

Listing 9.5. This code reads in the model predictions from SQLite using pandas.

```
pd.read_sql("""SELECT * #1  
    FROM customer_churn_predictions""",  
    sqliteConnection)
```

Alternatively, and perhaps most powerfully, we can query the model output data directly from SQLite *outside* of Python. For example, using a SQLite browser UI or the `sqlite3` command from a terminal, we can execute the following query:

Listing 9.6. Alternatively to Listings 9.4 and 9.5, we can query the customer churn predictions data from a SQLite environment itself.

```
SELECT * FROM customer_churn_predictions; #1
```

This means you don't have to be familiar with the model code or machine learning to be able to get the predictions. The predictions can be retrieved using a SQL query and used for monitoring or analysis.

Now that we've created a prototype pipeline and tested to make sure we can retrieve the data we write out to the SQLite database, let's summarize a few issues with our current approach.

- While we use the `DataSet` class, our code is still relatively tailored to the customer churn dataset.
- We directly hard-code the SQLite database and table names.
- Additionally, we assume the dataset we want to fetch predictions for is a test set carved out from the input customer churn dataset.
- We train the model and output predictions on the test set in the same code file. Ideally, the model training should be a separate process or module. The key reason for this is that we may not need to re-train the model as often as we fetch predictions. For example, you could setup an ML pipeline such that predictions are fetched daily, while the model is re-trained every 6 months (if your data doesn't change much) or more frequently (weekly, monthly, etc.) depending on the data itself and your need.

In the next few sections, let's work on fixing these issues. We'll start by separating out the model training component of the pipeline.

9.2.2 Generalizing the pipeline: training the machine learning model

Let's create a new file called *ml_pipeline_train_model.py* (available in the ch9 directory). This file will handle training a GBM model. Our goal is to create a Python file that can be run from the terminal (command line), and is generalized well so that the file is not tailored specifically to the customer churn dataset or specific columns. Additionally, running this script from the terminal will require a config argument. The purpose of the config (or *configuration*) argument is to point to a JSON file with the information we need when running this code. JSON is lightweight data format for storing key-value pairs of information. Before we delve into the code, let's show how to run it first. This will help make the logic in the code make more sense. To run this file from the command line, open up a terminal window and navigate to the *Code* directory corresponding to this book. Then, type in the command in Listing 9.7 and press enter.

Listing 9.7. Run this command from the terminal (you need to be the Code directory for this). This should train a GBM model on the customer churn data, and output the saved model. The ch9/ml_pipeline_config.json file specified in the config argument provides details on what parameters to use when training the model and where to output the saved model file.

```
python ch9/ml_pipeline_train_model.py  
--config ch9/ml_pipeline_config.json
```

The command in Listing 9.7 takes the Python filename as its first parameter. Additionally, it takes a parameter called *config*, which points to a config JSON file. The JSON contains parameter options that a user can specify, such as the name of an input data file to use for training the GBM model, what columns you want to include, what field is the label, etc. The contents of the config file are below (also available in ch9/ml_pipeline_config.json):

```
{  
    "input_data": {  
        "filename": "data/customer_churn_data.csv",  
        "needed_fields": ["total_day_minutes", "total_day_calls",  
                          "number_customer_service_calls"],  
        "label_col": "churn",  
        "pos_category": "yes"
```

```

},
"model_parameters": {
    "max_depth": [2, 3, 4, 5],
    "min_samples_leaf_lower": 5,
    "min_samples_leaf_upper": 55,
    "min_samples_leaf_inc": 5,
    "min_samples_split_lower": 10,
    "min_samples_split_upper": 110,
    "min_samples_split_inc": 10,
    "subsample": [0.6, 0.7, 0.8],
    "max_features": [2, 3],
    "n_estimators": [50, 100, 150, 200],
    "learning_rate": [0.1, 0.2, 0.3]
},
"hyperparameter_settings": {
    "n_jobs": 4,
    "scoring": "roc_auc",
    "n_iter": 10,
    "random_state": 0
},
"model_filename": "ch9/model_outputs/churn_model.sav"
}

```

As you can see, the config file contains four main pieces of information:

- input_data
 - The input_data parameters contain the configurations for the input filename, what fields to include as model features, the label column, and what category should be the positive category for the label.
- model_parameters
 - The model_parameters contain the hyperparameter search grid we want to use to optimize the GBM model
- hyperparameter_settings
 - These are the additional settings that get passed into the

hyperparameter tuning component of our code, including number of iterations we want to use in the grid search (in this sample case, just 10) and what metric we want to use to optimize the model ("roc_auc").

- `model_file`
 - This simply points to the output filename the user wishes to give to the trained model.

Using this config file, the code in Listing 9.8 will consume the file, train a GBM model based on the specifications of the config, and output the trained model to a file path based on the config `model_file` parameter.

The code for this file is shown in Listing 9.8.

Let's break it down:

- We start by importing the packages we'll need, including previously used modules from Chapter 4 (`ch4`)
- Then, we use `argparse` to parse the command line arguments from our input. `argparse` is a popular library that comes pre-packaged with most Python installations. It is used to parse command line arguments when running Python from the terminal. In this case, our only command-line argument is the `config` parameter. Again, this is specifying the configurations we want for our sample use case - training a model to predict customer churn utilizing the same dataset we're familiar with from earlier chapters.
- Next, we use Python's `json` library to read in the (JSON) config file.
- Using the config file, we create a `DataSet` object called `customer_obj`. This will store the training and test datasets. For now, we'll only use the training dataset, but in the *Practice on your own* section, you'll get a chance to extend this code and also make use of the test dataset.
- Then we set the hyperparameter search grid using `model_parameters` from the config file.
- Next, we create an `MLModel` object (from the class we defined back in Chapter 4).
- After this, we run the hyperparameter tuning process to get the optimized

GBM model.

- Lastly, we save the model object to the output file based on the config file's specifications.

Listing 9.8. The code in this listing consumes the input config file so that it knows what specifications to use for training the GBM model, including the hyperparameter search grid and input file. Then, it processes the input file to get only the needed features for inputting into the model. Next, it performs hyperparameter tuning using the MLModel class we created back in Chapter 4 (see Listing 4.18). Lastly, the optimized model is saved to a file location based on the specification given in the config file.

```
import sys #1
sys.path.append(".")
from sklearn.ensemble import GradientBoostingClassifier
from ch4.dataset_class_final import DataSet
from ch4.ml_model import MLModel
import json
import joblib
import argparse

parser = argparse.ArgumentParser() #2
parser.add_argument("-c", "--config") #3
args = parser.parse_args() #4

with open(args.config, "r") as config_file: #5
    config = json.load(config_file) #5

customer_obj = DataSet( #6
    feature_list = config["input_data"]["needed_fields"],
    file_name = config["input_data"]["filename"],
    label_col = config["input_data"]["label_col"],
    pos_category = config["input_data"]["pos_category"]
)

model_parameters = config["model_parameters"] #7

model_parameters["min_samples_leaf"] = #8
    range(model_parameters["min_samples_leaf_lower"],
          model_parameters["min_samples_leaf_upper"],
          model_parameters["min_samples_leaf_inc"])

model_parameters["min_samples_split"] = #9
    range(model_parameters["min_samples_split_lower"],
          model_parameters["min_samples_split_upper"],
```

```

model_parameters["min_samples_split_inc"])

parameter_check = {"max_depth", #10
                   "subsample", "max_features",
                   "n_estimators", "learning_rate",
                   "min_samples_leaf", "min_samples_split"}

model_parameters = {key : val for key, val in #11
                    model_parameters.items() if
                    key in parameter_check}

gbm = MLModel( #12
               ml_model = GradientBoostingClassifier(),
               parameters = model_parameters,
               n_jobs = config["hyperparameter_settings"]["n_jobs"],
               scoring = config["hyperparameter_settings"]["scoring"],
               n_iter = config["hyperparameter_settings"]["n_iter"],
               random_state = 0)

gbm.tune(customer_obj.train_features, #13
          customer_obj.train_labels)

joblib.dump(gbm, config["model_filename"]) #14
print("Model trained and saved to ", #15
      config["model_filename"])

```

Now that we have code that handles training the customer churn prediction model, let's write the next part of the pipeline to handle fetching predictions for new data and writing the predictions out to a database table.

9.2.3 Fetching predictions offline

In this section, we will create a new generalized Python file to handle fetching predictions from the customer churn model we created in the last section. The purpose of creating this file to have a generalized script that can handle many different types of models and datasets. While our specific example in this case is the customer churn dataset, this same script could be applied to many other scenarios as well. Similar to the previous section, let's start by creating a config file. This config file will specify key information such as the locations of the input file and model. The config file is shown below (also available in the `ml_offline_predictions_config.json` file in the `ch9` directory):

```

{
    "model_filename": "ch9/model_outputs/churn_model.sav",
    "id_field": "customer_id",
    "input_filename": "data/customer_sample.csv",
    "input_database": "data/customers.db",
    "output_database": "data/customers.db",
    "output_table": "customer_churn_predictions",
    "input_features": ["total_day_minutes", "total_day_calls",
                       "number_customer_service_calls"]
}

```

Next, let's walk through the code that will consume this config file and fetch the model predictions. This code is in Listing 9.9.

- First, we import the necessary packages
- The next few lines are similar to Listing 9.8, which allow us to consume the config file
- Then (Line 6), we get a list of the columns we will need from the input dataset. This list comes from the config file plus the name of the ID field. The ID field is a column that provides a unique ID for each row in the input dataset. For our example, that will be the *customer_id* field.
- Next (Line 7), we read in the input file, with the restriction that we only bring into memory the columns specified in the previous step
- In Line 8 we load the model file into memory
- Then, we create the output data frame, which will store the model predictions, along with the ID field (again, that's the *customer_id* for this specific use case), and a date field specifying the upload date
- Next, we connect to SQLite and create a cursor object
- In Lines 16 and 17 we get the names of the database table we will upload the data to, and the name of the ID field (*customer_id*)
- Line 18 provides the SQL insert statement we'll need to upload the data
- Lines 19 uploads the output data frame into the database table (*customer_churn_predictions*). Using the *commit* method in Line 20 ensures the updated data is saved in the database table.
- Lines 22 and 23 close the cursor object and the SQLite connection

Listing 9.9. This code (also available in the `ml_pipeline_offline_predictions.py` file located in the `ch9` directory) will handle reading in an input dataset, running the model on the data, and storing the predictions to a database table. This code is meant to be generalizable, so that you can use it with little modification on many other datasets or ML models.

```

import sys #1
sys.path.append(".")
import sqlite3
import argparse
import pandas as pd
import joblib
from datetime import datetime
import json

parser = argparse.ArgumentParser() #2
parser.add_argument("-c", "--config") #3
args = parser.parse_args() #4

with open(args.config, "r") as config_file: #5
    config = json.load(config_file)

include_cols = config["input_features"] + \
    [config["id_field"]] #6

new_data = pd.read_csv(config["input_filename"], #7
                      usecols = include_cols)

model = joblib.load(config["model_filename"]) #8

output = pd.DataFrame() #9
output[config["id_field"]] = new_data[config["id_field"]].copy()
output["model_prediction"] = model.\ #11
    clf.\
    predict_proba(new_data[config["input_features"]])[::,1]

output["prediction_date"] = datetime.\ #12
    date(datetime.now()).\
    strftime("%Y-%m-%d")

sqliteConnection = sqlite3.connect(
    config["output_database"]) #13
cursor = sqliteConnection.cursor() #14
print("Connected to SQLite...") #15

table = config["output_table"] #16
id_field = config["id_field"] #17

insert_command = f"""
    INSERT INTO {table}({id_field}, #18
    model_prediction,
    prediction_date)

```

```

values (?, ?, ?)"""

cursor.executemany(insert_command, #18
                    output.values)

sqliteConnection.commit() #19
print("Committed model predictions...")

cursor.close() #20
sqliteConnection.close()

```

At this point, we have a generalized file that can handle training a GBM classifier for any arbitrary dataset and saving the model object out to a filename of the user's choice. Additionally, we have a file that can read in input data from a CSV file or database table, run an input model on the data, and stores the predictions to a database table. In practice, you might split this two functionalities into more files. For example, if your feature engineering process is very complex, you might break it into multiple files. If you need to ingest data from many sources, it is reasonable to have a separate code file solely dedicated to data ingestion, while feature engineering and model training are handled in other files. Beyond these practical considerations, there are a few key points we should also consider:

- What if your database connection requires username and password? We used SQLite for our examples in this section, but using other databases like SQL Server, Oracle, or MySQL could require entering credentials to connect to a database.
- What if you need to fetch predictions real-time or one at a time?
- The code in our examples work fine if you know exactly what you need to input. But what happens if a user incorrectly inputs some values in the config file? Or what if the config file is missing entirely? Furthermore, how can we be confident that our code works properly all the time? These are practical considerations that we will tackle in Chapter 10.

That covers this section. Now, let's cover the previously mentioned point about protecting your credentials when connecting to a database. This will help to expand our current example to be able to handle any database connection where you need to input a username and password.

9.2.4 How to protect your credentials in your code

Storing credentials directly in your code, particularly passwords, is a very bad idea. Doing so can lead to information leakage, or allow malevolent actors to steal credentials and gain unwarranted access. Fortunately, there are several ways to avoid this issue. Python offers several popular libraries related to protecting credentials, including the following:

- keyring (<https://pypi.org/project/keyring/>)
- passlib (<https://passlib.readthedocs.io/en/stable/>)
- argon2-cffi (<https://argon2-cffi.readthedocs.io/en/stable/>)
- bcrypt (<https://github.com/pyca/bcrypt/>)
- cryptography (<https://github.com/pyca/cryptography>)

Let's start by discussing the *keyring* library. Keyring is an easy-to-use Python package that works by allowing you to store a password in your operating system's credential store. It's great in that it can work across multiple different operating systems. We'll start with keyring because it is easy to setup and use as a starting point for protecting your credentials. To get started, just use pip to install (see Listing 9.10).

Listing 9.10. Use pip to install keyring

```
pip install keyring #1
```

Once we've imported keyring, we can store a username / password combination using the *set_password* method, as seen in Listing 9.11. This method has three parameters. First, it takes a parameter called *servicename*. This is the name we choose for whatever service our username / password is associated with e.g. email, database, etc. In our example, we'll just call it *database_conn*. Next, the second and third parameters are the username and password, respectively.

Listing 9.11. Use keyring's set_password method to store a username and password. The first parameter of this method is an arbitrary name that you want to associate with the username/password combination. In this case, we store the username and password as database_credentials.

```
import keyring #1
```

```
keyring.set_password("database_credentials", #2  
    "secret_username",  
    "secret_password")
```

Once we've set the password, it remains stored by our operating system – so if you start a new Python session, you'll be able to retrieve it just the same. To retrieve the password, we just need to use the `get_password` method with the `servicename` value and username.

Listing 9.12. To retrieve the stored credentials, you can use the `get_password` method from `keyring`. For this method, we just need to input the name associated with the credentials we want to retrieve, along with the username.

```
keyring.get_password("database_credentials", #1  
    "secret_username")
```

Let's suppose, for example, that we're connecting to a database, where we need to pass credentials. For example, when using SQL Server, a popular database system, you may need to enter a username and password. An example of this is in Listing 9.13:

Listing 9.13. This code snippet is an example of passing a username and password into a database connection string.

```
import pyodbc #1  
import keyring #1  
  
password = keyring.get_password("database_credentials", #2  
    "secret_username")  
  
conn_str = ("Driver={SQL Server};"  
            "Server=servername"  
            "Database=databasename;"  
            "UID=secret_username;"  
            f"PWD={password};")  
  
conn = pyodbc.connect(conn_str) #3
```

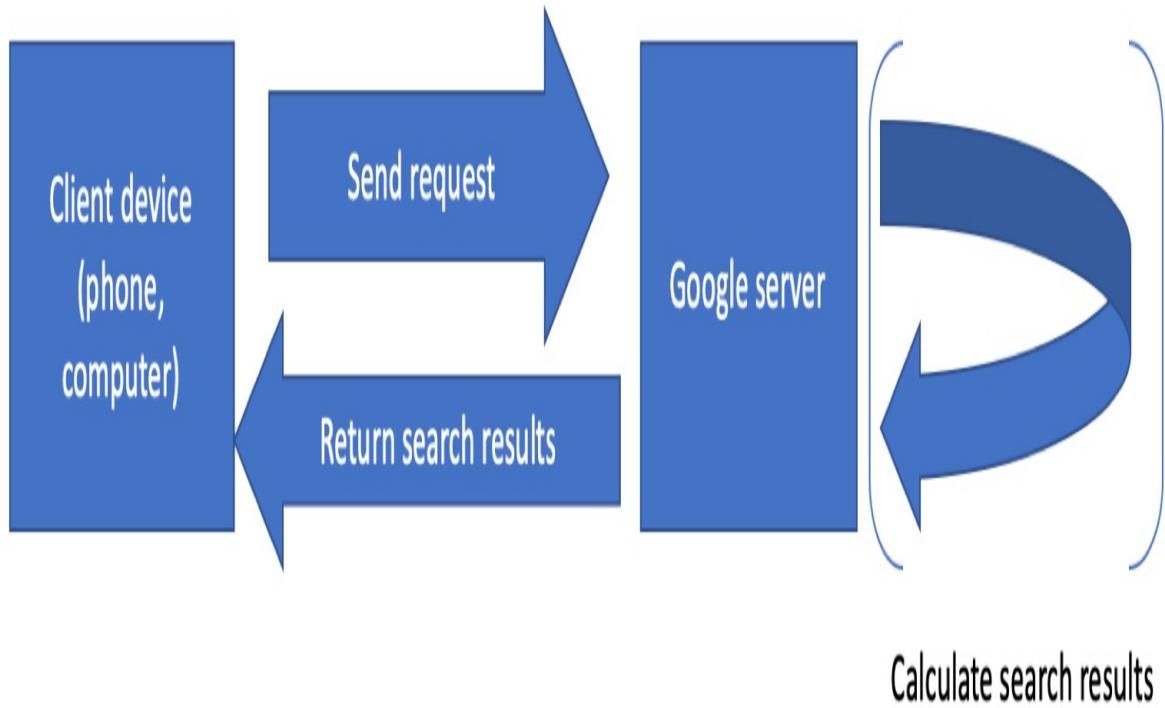
That covers the `keyring` package. In the next section, we will cover how to convert Python code into an API. This will allow us to deploy an ML model in production so that you can fetch predictions from the model in real-time without ever interacting with Python directly to get the predictions.

9.3 How to convert your code into an API

In this section, we will use Python's FastAPI package to convert the model prediction code from the previous section into an API.

First off, what exactly *is* an API? An API is a mechanism that allows you to communicate with a service without needing to know the details of how that service works under the hood. Let's take a common example of making a request to a website. When you search for a term in Google, and press enter, your device (phone, computer, etc.) makes a request to one of Google's servers. The request contains key information such as term that you are searching for. Then, there are various processes that occur on Google's servers to handle ranking and returning the search results to you. These might involve complex algorithms that the user doesn't need to know or understand in order to actually get the results. The process of determining how to return the results is handled by an *API*. When you make a request to search for the term on Google, your device is actually sending a request to this API. The API will then return the results to you. The diagram in Figure 9.3 demonstrates a simplistic version of this.

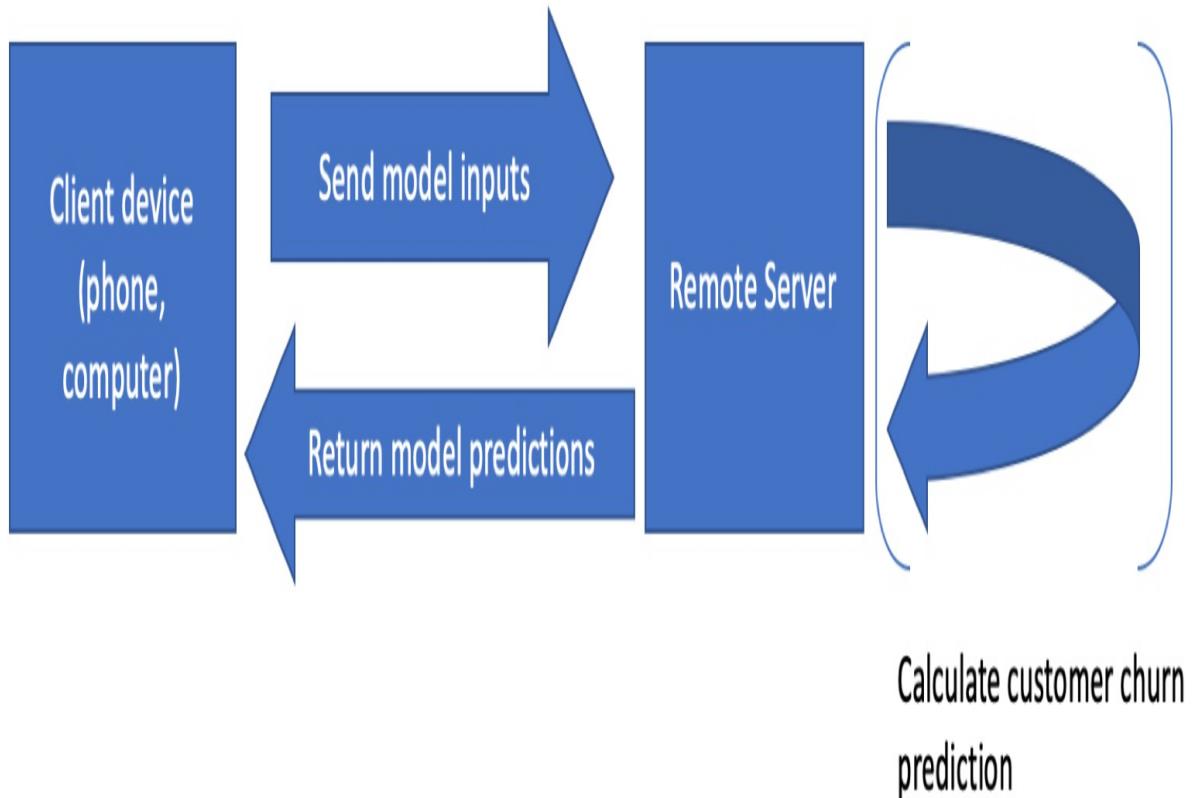
Figure 9.3. This diagram shows a simplified version of the API request made when you search for a term on Google. Here, the client represents your device, such as phone or computer. The client sends a request to Google's server(s), which then computes the search results and returns them to the user. The user never directly interacts with or runs any code.



We can extend the same idea from Figure 9.3 to creating an API that calls a machine learning model and returns the results to the user without the user ever needing to interact or even know anything about the code computing the model result.

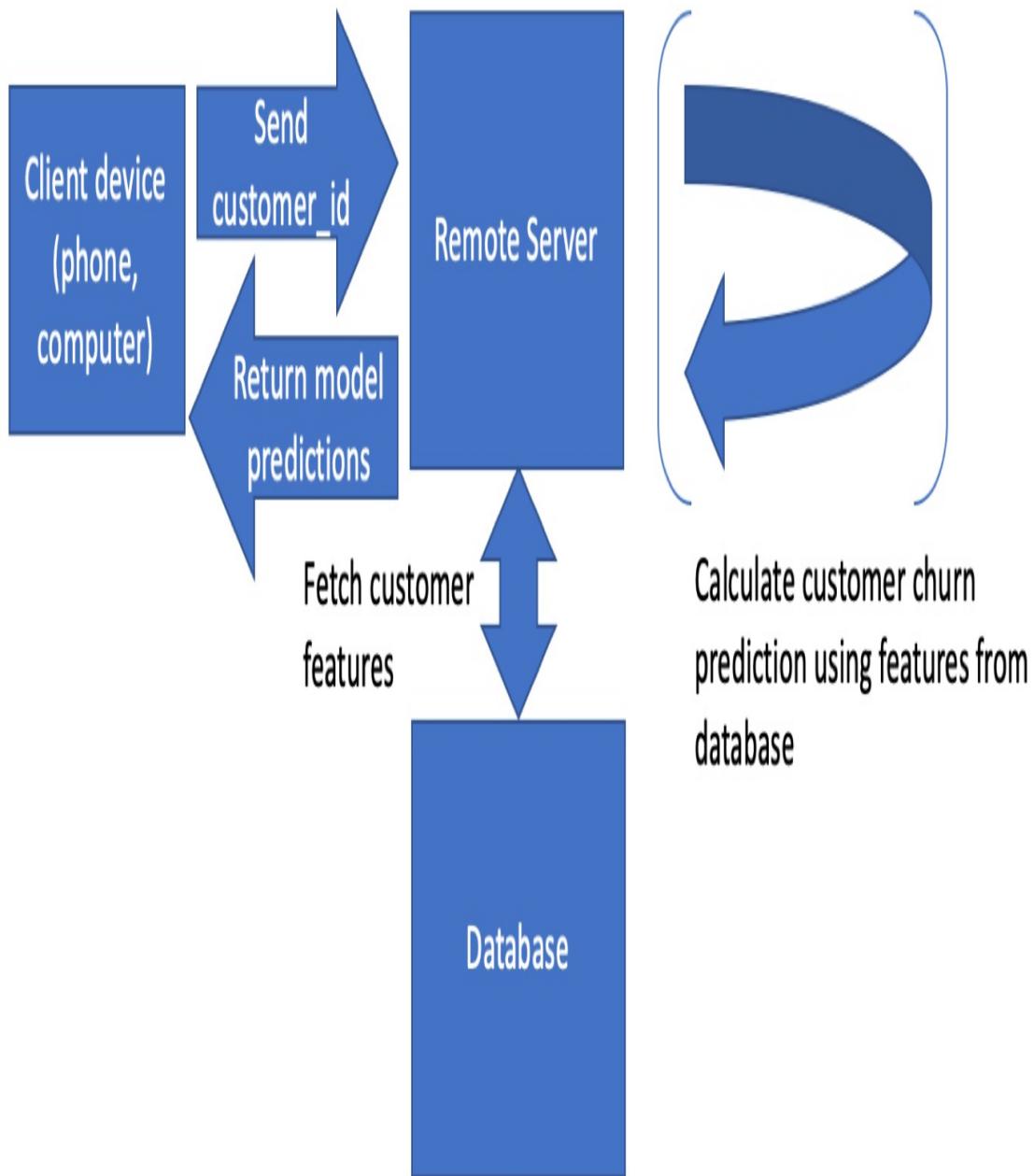
In Figure 9.4, we modified Figure 9.3 to demonstrate how we can construct an API so that a user can send the inputs to the customer churn model, and have the API handle computing the prediction, and then returning the result to the user.

Figure 9.4. This diagram shows a simplified version of the API request made when you search for a term on Google. Here, the client represents your device, such as phone or computer. The client sends a request to Google's server(s), which then computes the search results and returns them to the user. The user never directly interacts with or runs any code.



We could go one step further, as well, where the user doesn't even need to know what the model inputs are. For example, we could setup an API where the user just needs to pass the customer_id. Then, the API will handle fetching the model features corresponding to that customer_id, passing the features to the model, and then returning the model prediction to the user. Figure 9.5 shows a visualization of this.

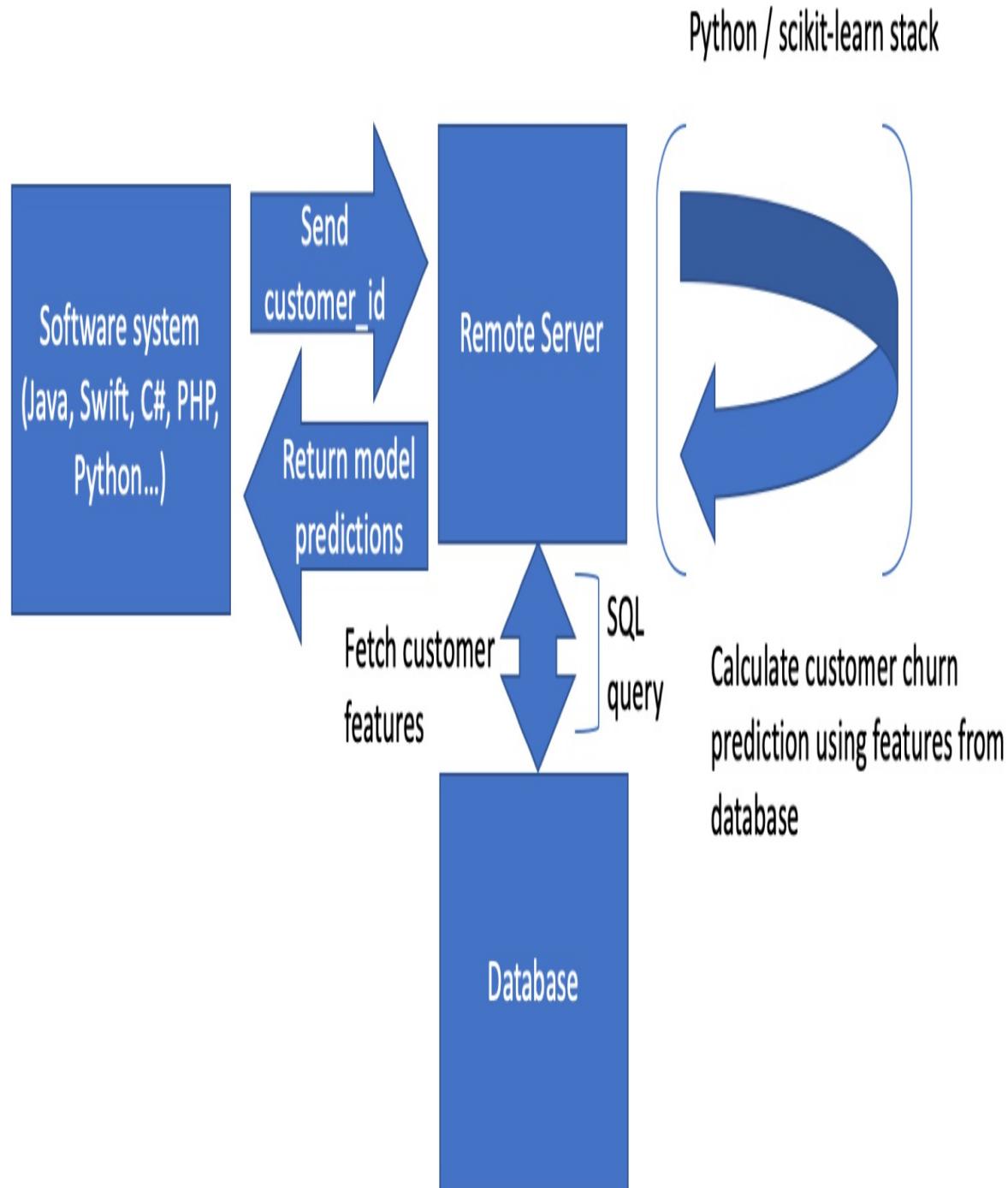
Figure 9.5. This diagram shows a simplified version of the API request made when you search for a term on Google. Here, the client represents your device, such as phone or computer. The client sends a request to Google's server(s), which then computes the search results and returns them to the user. The user never directly interacts with or runs any code.



To make the power of APIs more clear, we should consider that the request can be made from many different platforms. For example, if you have a system that is written in Java, but you want to utilize the customer churn prediction model, you could simply call the API handling interaction with the model from the Java code. The same applies for virtually any other language,

such as PHP, Perl, C#, etc. See Figure 9.6 for this visualization.

Figure 9.6. This diagram shows a simplified version of the API request made when you search for a term on Google. Here, the client represents your device, such as phone or computer. The client sends a request to Google's server(s), which then computes the search results and returns them to the user. The user never directly interacts with or runs any code.



There are several different types of API requests. Table 9.1 summarizes several of the main ones. In the example APIs we will build, we will mostly be concerned with GET and POST requests:

- GET requests are used to read or retrieve data from a server. For example, whenever you navigate to a webpage, a GET request is made to the server hosting the corresponding website to show the data on the webpage
- POST requests are used to send data to a server. For example, if we create an API to handle fetching the churn prediction for a customer, we could use a POST request sending the model features to a server, and then the server will handle computing and returning the model prediction

Table 9.1. Common types of API requests include GET, POST, PUT, etc. shown in this view. For instance, GET requests are used when retrieving data from a server.

Type of API request	When to use	Example use case
GET	GET requests are used to retrieve data from a server	Navigating your browser to https://www.google.com
POST	POST requests are used to send data to a server	Entering and submitting credentials for a webpage
PUT	PUT requests modify data on a server	Could be used to update the data corresponding to a particular customer
	PATCH requests modify	Similar to the PUT use case, except only the specific data that needs

PATCH	part of a piece of content to be updated must be passed with the PATCH request
DELETE	DELETE requests delete a piece of data at a specified location on a server Sending a request to deleting a record corresponding to a customer

Now that we have a better understand of APIs, let's get started with creating our own API by installing FastAPI (see Listing 9.14). FastAPI (see full documentation here: <https://fastapi.tiangolo.com/lo/>) is a very popular package for deploying APIs for Python-based code. It can be used as a single cohesive framework for both prototyping APIs and putting APIs into production. It is also built on another Python library called *pydantic* (full documentation is available here: <https://docs.pydantic.dev/latest/>) that makes it easy to validate the correctness of inputs into the API. Other Python libraries for creating APIs include Flask and Django REST Framework (DRF). FastAPI is very flexible and is easy to get started when creating APIs for the first time. Flask and DRF are both associated with Python web frameworks. We will introduce Flask in Chapter 14.

While we use pip to install FastAPI, we will also install *uvicorn*. In simplistic terms, *uvicorn* is a Python-based web server that allows FastAPI to handle requests. Essentially, uvicorn will function as the server program from the previous diagrams (see Figures 9.2-4 for instance).

Listing 9.14. Use pip to install FastAPI and uvicorn

```
pip install fastapi uvicorn[standard] #1
```

Now that we have FastAPI installed, let's test it out using a simple script. Once we make sure our test file is working, we'll get back to converting the customer churn model prediction code into an API. In Listing 9.17 we create a script called *main.py*. This script will be used to generate a webpage that

prints the message *Testing...FastAPI at work!*

Let's break down what this code does:

- First, we import the FastAPI class
- Second, we create an instance of the FastAPI class called *app*. This will be the main object we use to create our API.
- Thirdly, we create a decorator that specifies that the function below it will handle any requests that go the URL set in the decorator. In this case, our URL is simply "/", which means that this points to the host server address where the API will live.
- Next, we create the function mentioned in the third step to handle any requests to the URL in its corresponding decorator
- Lastly, we return a message stating *Testing...FastAPI at work!*. This message will be printed when you navigate to the host URL (assuming the server is currently running the API)

Listing 9.15. Use pip to install FastAPI and uvicorn

```
from fastapi import FastAPI #1

app = FastAPI() #2

@app.get("/") #3
async def root(): #4
    return {"message": #5
        """Testing...FastAPI at work!"""}#6
```

To launch the API code, we can use *uvicorn*, following the command in Listing 9.18. This command should be run from the terminal.

Listing 9.16. Use uvicorn to launch the test API

```
uvicorn main:app --reload #1
```

Now that we've covered a basic test example using FastAPI, let's do something more useful and work on creating an API for our customer churn model prediction code. We will design this API in the form of a POST request (see Listing 9.17). To make the POST request, you will need to pass the input features for the model. The API will then handle passing those

features to the model and returning the customer churn predictions. A common way of formatting data when being sent for a POST request is to send the data in JSON. JSON is a lightweight method for transferring data between client and server or between two different servers. It structures data in the form of key-value pairs, similar to Python dictionaries. After we walk through the code in Listing 9.17, we'll show an example sending JSON with our POST request to retrieve the model's predictions.

Listing 9.17. This code will launch an API to handle fetching predictions for the custom churn prediction model.

```
import sys #1
sys.path.append("../..")

from fastapi import FastAPI
from pydantic import BaseModel
import pandas as pd
import joblib

class ModelFeatures(BaseModel): #2
    total_day_minutes: float
    total_day_calls: int
    num_customer_service_calls: int

model = joblib.load("../model_outputs/churn_model.sav") #3

app = FastAPI() #4

@app.post("/churn/") #5
async def get_churn_predictions( #6
    features: ModelFeatures):

    model_inputs = pd.DataFrame([features.total_day_calls, #7
        features.total_day_minutes,
        features.num_customer_service_calls]).transpose()

    pred = model.clf.predict_proba(model_inputs)[0,1] #8

    return {"prob_of_churn": pred} #9
```

Breaking down the code in Listing 9.17, we created a single code file that handles creating an API that can fetch predictions from the customer churn model. Let's delve into what each of piece of the code is doing.

- First, we import the packages we need. This time we'll include a new package call *pydantic*. We will use the *BaseModel* class from this library.
- Second, we create an object of the *BaseModel* class called *ModelFeatures*. The purpose of this class is to define the input features into the customer churn prediction model. These are the features that will need to be passed when someone makes a request to the API.
- Next, we load the customer churn model using the *joblib* library
- Then, we create a FastAPI object called *app*. Again, this will serve as our main object handling interaction with the API
- Next, we define the decorator detailing what URL endpoint to use to fetch the customer churn predictions. We also specify that a *post* request needs to be made. The function that handles the request is defined just underneath this decorator (Line 6)
- In Line 6, we start the definition for the function handling requests to the API. This function takes a single parameter called *features*. This parameter is meant to be the inputs to the model, which will be passed in JSON form when a request is made to the API
- Within the *get_churn_predictions* function, we convert the inputs to a data frame, and then pass the single-row data frame to GBM model. This model will return the probability of a customer churning.
- Lastly, we return the result from the model to the user making the request

Now that we've gone through the code, let's test out using the new API! There are many ways to do this. First, we need to launch unicorn so that we'll be able to make requests to the API. To do that, we just need to run the same command from Listing 9.18, but pointing to our new *main.py* file.

Listing 9.18. Use unicorn to launch the test API

```
uvicorn main:app --reload #1
```

Next, let's use Python's *requests* library to submit a POST request.

Listing 9.19. Use unicorn to launch the test API

```
import requests #1
```

```

inputs = {"total_day_minutes": 10.2, #2
          "total_day_calls":4,
          "num_customer_service_calls": 1}

resp = requests.post( #3
    "http://127.0.0.1:8000/churn",
    json = inputs)

print(resp.json()) #4

```

The result of running Listing 9.19 is below:

```
{'prob_of_churn': 0.021225141894258632}
```

In order to make an API you create with FastAPI available to other users on your network, you need to add an extra parameter called *host* when creating the *FastAPI* object. This parameter should be set to *0.0.0.0*. Listing 9.20 shows the updated code.

Listing 9.20. This code will launch an API to handle fetching predictions for the custom churn prediction model, similar to Listing 9.19. The only difference between Listing 9.19 and 9.20 is that in this updated code, we add the parameter for *host* = "*0.0.0.0*" to make the API available to other users in the same network.

```

import sys #1
sys.path.append("../..")

from fastapi import FastAPI
from pydantic import BaseModel
import pandas as pd
import joblib

class ModelFeatures(BaseModel): #2
    total_day_minutes: float
    total_day_calls: int
    num_customer_service_calls: int

model = joblib.load("../model_outputs/ #3
                    churn_model.sav")

app = FastAPI(host = "0.0.0.0") #4

@app.post("/churn/") #5
async def get_churn_predictions(features: ModelFeatures): #6

```

```

model_inputs = pd.DataFrame([ #7
    features.total_day_calls,
    features.total_day_minutes,
    features.num_customer_service_calls]).transpose()

pred = model.clf.predict_proba(model_inputs)[0,1] #8

return {"prob_of_churn": pred} #9

```

It is also possible to deploy a FastAPI application publicly on the internet. This can be done using AWS (Amazon Web Services), or other tools like Space (<https://deta.space/>) or Fly.io (<https://fly.io/>).

That's all for now on FastAPI. Next, let's summarize what we've learned this chapter.

9.4 Summary

In this chapter we covered the following key points:

- One way of putting a model into production is to setup an offline pipeline where model predictions are computed and stored in a database table. The general offline workflow looks like this:
 - Fetch data from database needed for model inference
 - Run model on the data (or on the data after cleaning and processing)
 - Save the model predictions in a database table so that they can be fetched using SQL queries
- The keyring package can be used to protect your credentials. Use the *set_password* and *get_password* methods to save or retrieve your credentials.
- An alternative way of putting a model into production is to create an API. A key use case of APIs for data science cases is that they allow you to call a model living in a different server or environment.
- FastAPI is a powerful library for creating an API for your Python code.

9.5 Practice on your own

1. In our pipeline examples that write to a database table, we still read from a CSV file. Can you modify the code to read from a database table?
Hint: you'll first have to upload a subset of the customer churn dataset to a SQLite database.
2. Modify the example in Listing 9.8 to also generate performance metrics, such as precision and recall for the trained model. Can you include these metrics into a database table?
3. Can you update the API example we went through using a random forest model instead of a GBM?