16.0.1 Grasping the Scope of Big Data

“Big data” is a popular buzzword in the data industry today. But what does the term mean, and why is it important in the broader context of data science? The video below gives some background about the concept of big data and its impact.

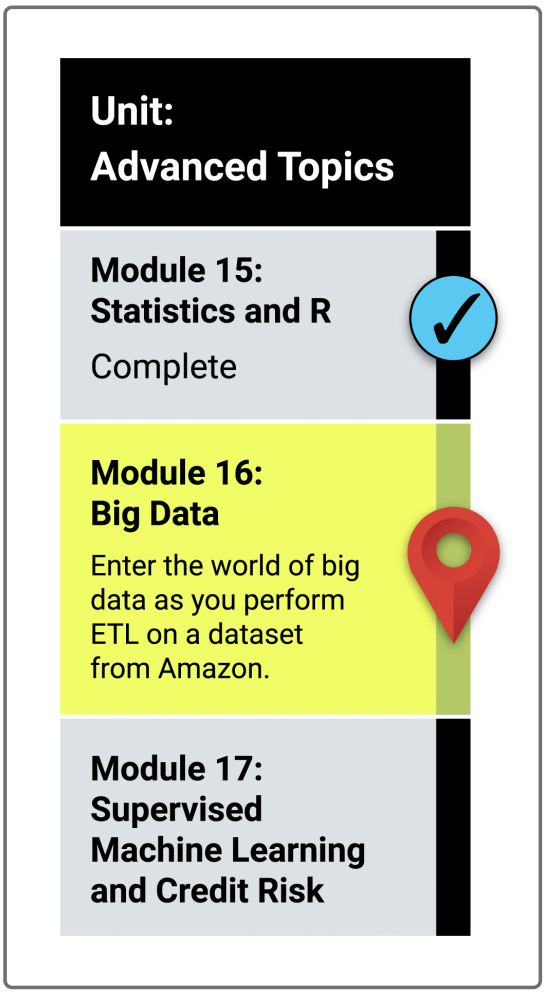
16.0.2 Module 16 Roadmap

Looking Ahead

This week we'll cover what constitutes big data and how it's handled. We'll start by reviewing Hadoop and its ecosystem.

Within this big data and Hadoop context, we'll cover MapReduce and how it has improved the process for handling big data. We'll then move on to PySpark, which has become the leading technology for handling big data.

After diving into some of the technologies used with big data, we'll look at natural language processing (NLP) in relation to big data.



We'll close with an introduction to cloud services. Cloud services let us store large amounts of data at remote locations rather than locally, on top of many other services. This allows for more scalability and performance. We'll use the most popular cloud service available: Amazon Web Services (AWS).

What You Will Learn

By the end of this module, you will be able to:

- Define big data and describe the challenges associated with it.
- Define Hadoop and name the main elements of its ecosystem.
- Explain how MapReduce processes data.
- Define Spark and explain how it processes data.
- Describe how NLP collects and analyzes text data.
- Explain how to use AWS Simple Storage Service (S3) and relational databases for basic cloud storage.
- Complete an analysis of an Amazon customer review.

Planning Your Schedule

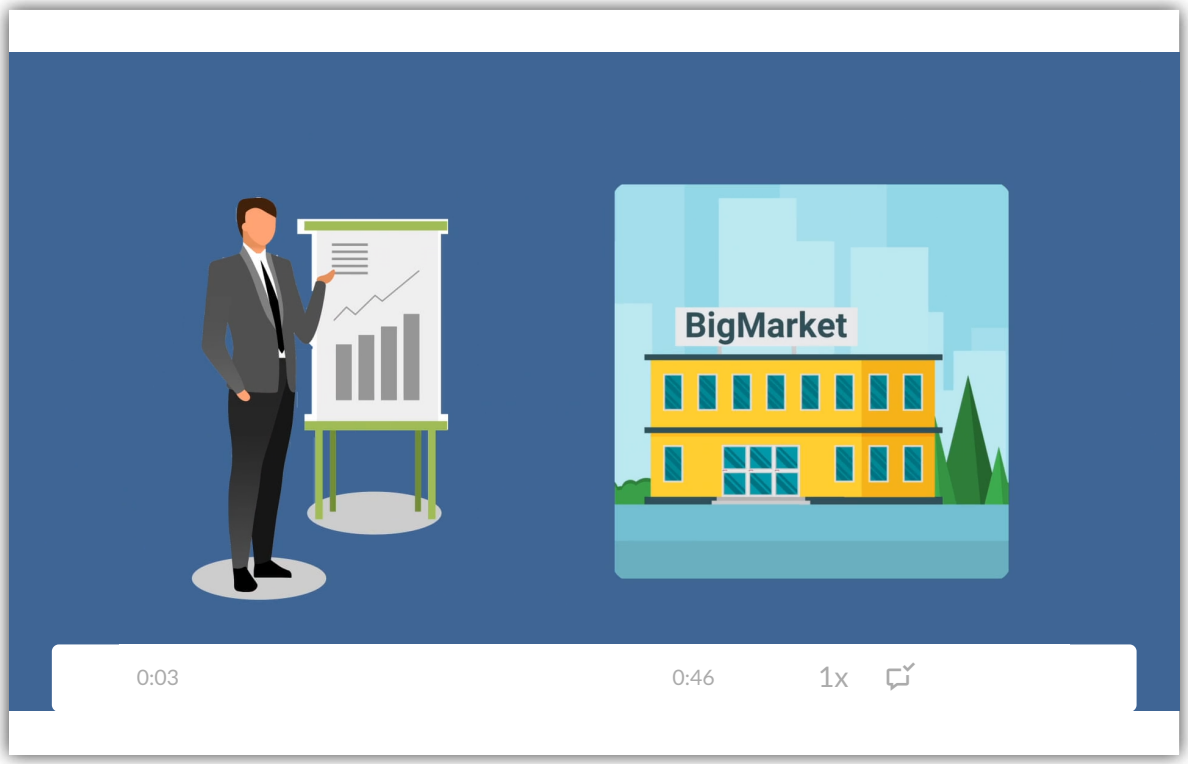
Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule:

- Introduction to Module 16 (15 minutes)
- Overview of Big Data (15 minutes)
- Using MapReduce to Process Data (30 minutes)
- Using Spark to Handle Large Datasets (30 minutes)
- Working with Spark DataFrames and Functions (2 hours)
- Natural Language Processing (1 hour)

- PySpark and Natural Language Processing (2 hours)
- Cloud Databases with Amazon Web Services (2 hours)
- Cloud Storage with S3 on AWS (2 hours)
- ETL in the Cloud (2 hours)
- Application (5 hours)

16.0.3 Welcome to Big Data!

The video below walks you through the project you'll work on in this module, which will require you to analyze product reviews for a marketing company. Get ready to work with some big data!



16.1.1 What Is Big Data?

For this project you'll be partnering with Jennifer, an account manager at BigMarket. SellBy, your client, loves to talk about the power of big data, but Jennifer isn't a data expert. So you start off the project by giving her a quick overview of what big data actually is.

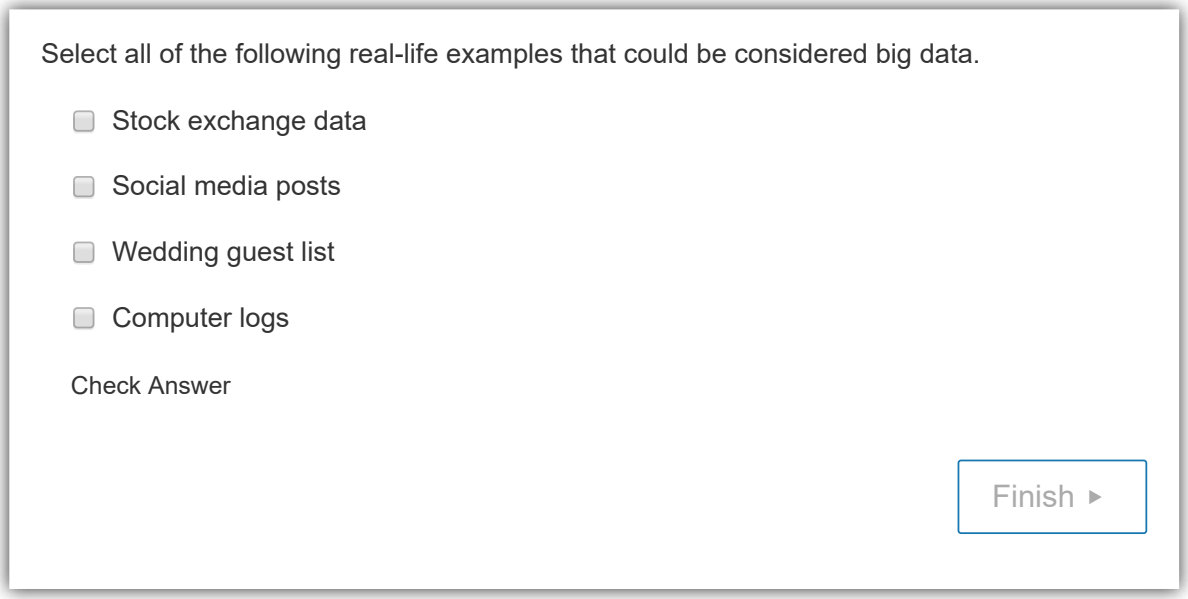
What exactly constitutes big data? At what point does data become "big"? Is it just the size? A good rule of thumb to apply is this: Data is considered big data when it exceeds the capacity of operational databases.

For example, a retail item might have hundreds of reviews but not be big data because a local machine can parse it. However, consider that there are tens of thousands of items for sale on Amazon and each has hundreds of reviews. In this case, you have big data.

Some additional examples of big data are listed below:

- **Financial Industry Regulatory Authority (FINRA)** stores **37 billion financial records (https://aws.amazon.com/solutions/case-studies/finra/)** daily and analyzes trends over a period of days, weeks, and months.

- **McDonald's** collects transactional data as it serves 69 million people each day in more than **100 countries (https://corporate.mcdonalds.com/corpmcd/about-us/around-the-world.html)** .
- **Netflix** collects ratings, searches, watch dates, device information, pause and skip data, repeat views, and additional information for more than **158 million subscribers (https://aws.amazon.com/solutions/case-studies/netflix/)** .



Four Vs of Big Data

There are four characteristics of big data:

- **Volume** refers to the size of data (e.g., terabytes of product information). For instance, a year's worth of stock market transactions is a large amount of data.
- **Velocity** pertains to how quickly data comes in (customers across the world purchasing every second). As an example, McDonald's restaurants are worldwide with customers buying food at a constant rate, so the data comes in fast.

- **Variety** relates to different forms of data (e.g., user account information, product details, etc.). Consider the breadth of Netflix user information, videos, photos for thumbnails, and so forth.
- **Veracity** concerns the uncertainty of data (e.g., reviews might not be real and could come from bots). As an example, Netflix would want to verify whether users are actively watching the shows, falling asleep, or just playing them in the background.

The four Vs of big data will help you determine when to migrate from regular data to big data solutions.

Big Data Problems

Working with datasets of this size creates unique challenges. How will we store all of this data? How can we access it quickly? How do we back up this type of data?

You and Jennifer certainly have your work cut out for you, but you know that learning how to work with datasets of this size will help you give the best recommendations to your client.

We'll cover all of these issues in the module.

16.1.2 Big Data Technologies

You and Jennifer know that this project has outgrown Excel, SQL, and other databases you have used before. This means that you get to explore brand-new tools and think about data in a whole new way. To begin, you decide to explore the technologies that support the big data ecosystem. This way you can decide what technologies you will need to answer SellBy's questions.

Apache Hadoop (Hadoop) is one of the most popular open source frameworks, with numerous technologies for big data. Google developed Hadoop to process large amounts of data by splitting data across a distributed file system.

We'll start with the three main components of Hadoop:

- **Hadoop Distributed File System (HDFS)** is a file system used to **store data** across server clusters (groups of computers). It is scalable (which means it handles influxes of data), fault-tolerant (handles hardware failure), and distributed (spread across multiple servers connected by a common core).

16.2.1 MapReduce Process

Now that you and Jennifer have an understanding of big data, the next step is figuring out how to process it. Jennifer is starting to get excited about big data, so you decide that you will handle half of the dataset while Jennifer will handle the other half. MapReduce is a common tool for splitting up large datasets, so you sit down to find out if it will work for you.

MapReduce is used as a means for distributing and processing data on your cluster. MapReduce is built on the process of **mapping**—the process of assigning the same job to each of the computers—and **reducing,** which is when you come back together to combine the results.

In this case, you and Jennifer have decided to count all the reviews of video games in different categories across a dataset. To save time, you decide to take the first half, and Jennifer takes the second half. There are 2,800 rows of data to comb through, and we want a total for reviews of sports, fantasy, and role-playing games.

By splitting up the data, the time to analyze it has been reduced by half. In the same way, MapReduce works on divided datasets in smaller batches so that we can work faster.

Once you are done, you will come together and combine the tallies for both.

Mapping is taking a small piece of the input and then converting the data into **key-value pairs,** with key identifiers and associated values. For this project, you'll count all of the reviews from the first set of data and store it, as shown below:

Key	Value
sports	500
fantasy	300
role_playing	600

Jennifer does the same for her set and stores it as follows:

Key	Value
sports	1000
fantasy	100
role_playing	300

You have now mapped out the review datasets.

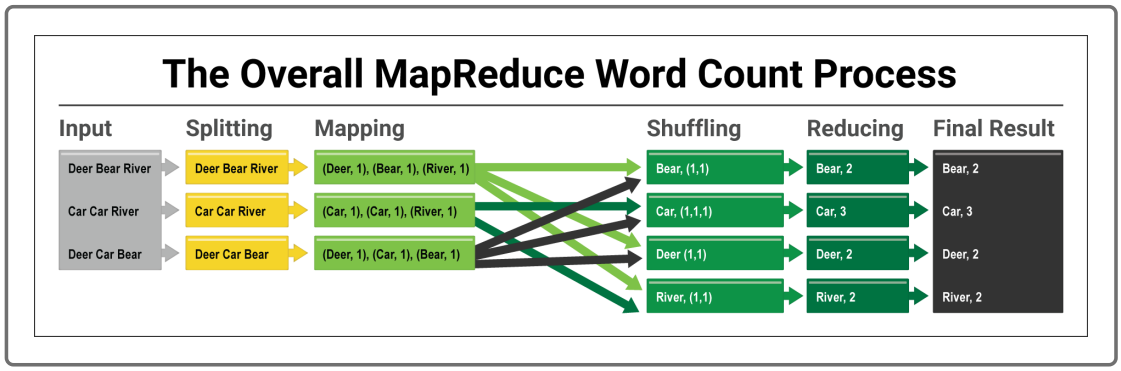
Reducing is when you aggregate the results, in this case, by adding up your figures:

Key	Value
sports	1500
fantasy	400

role_playing	900

You have just reduced the two results into one.

Another example would be running a word count on a document. The following image shows the MapReduce process for running word count on an input:



Let's break down each part of the word count process:

- **Input:** The entire file is fed to the word counter.
- **Splitting:** Each line of text is separated.
- **Mapping:** Each word in every line is assigned a value of 1.
- **Shuffling:** The words are combined and organized alphabetically, creating a list of the words' values.
- **Reducing:** The list of values are summed for each word.
- **Final Result:** The complete list of words and value (counts) are displayed.

MapReduce is an important tool for handling big data. Next, we'll look at a Python library that will allow us to practice this process at a smaller scale.

- **MapReduce** is a programming model and processing technique for big data. MapReduce enables processing the large amount of data spread across the cluster in the HDFS by performing the same task for each file system.
- **Yet Another Resource Negotiator (YARN)** manages and allocates resources across the clusters and assigns tasks.

Hadoop distributes for the storage and processing of data through a cluster, which is a group of connected computers that work together to store and perform tasks on a dataset.

Hadoop is quite difficult to set up. You need to set up all three main components across multiple machines, as well as make sure each one has sufficient resources and is configured for optimal performance. Because of this, it may not be the right technology for your startup. However, you know your client will ask about it—"Hadoop" is a popular buzzword, after all—so it's important to have a baseline knowledge of it.

NOTE

> Visit the **Hadoop official website (https://hadoop.apache.org/)** to see other projects offered by Hadoop.

16.2.2 mrjob Library

To make sure you can answer your client's questions, you and Jennifer decide to get a little practice using MapReduce via Python. For this you'll be using the mrjob library.

Python has a library called mrjob, which stands for "MapReduce job" and it can help us practice MapReduce outside of the Hadoop ecosystem. There is very little setup, and we can run the library from any computer with Python installed.

Let's start by installing mrjob in our Python environment. Open the terminal and activate your PythonData environment. Once activated, run the following command in your terminal:

```
$ pip install MRJob
```

Next, open your text editor and create a new Python file named `bacon_counter.py`. In the file, enter the following code to import mrjob:

```
from mrjob.job import MRJob
```

Create a class called `Bacon_count`, which inherits, or takes properties, from the `MRJob` class. We create this class to be called to run the full MapReduce job with `MRJob`:

```
class Bacon_count(MRJob):
```

Next, create a `mapper()` function that will take `(self, _, line)` as parameters. The `mapper()` function will assign the input to key-value pairs:

```
def mapper(self, _, line):
```

The underscore (`_`) allows methods to be mapped together. Since we are not chaining anything together, we use the Python convention of an underscore to indicate that we won't use this parameter. The `line` parameter will be the line of text taken from the raw input file.

The function will loop through each word in the line of text, as described below:

1. Call the `split()` method on each line to break the text into a list of words.
2. Each word will convert to lowercase.
3. If the words match the search word "bacon," a key-value pair will show as `yield "bacon", 1`.
4. A yield will continue producing a sequence of values until the function has finished running, unlike a return statement, which would exit the function after the first instance of the word. So, for a yield, each time the word "bacon" appears, an output of `"bacon", 1` is produced. If

"bacon" appears three times, then an output of `"bacon", 1` would be produced three times:

```
for word in line.split():
if word.lower() == "bacon":
        yield "bacon", 1
```

There's a shuffle step that occurs after the mapper. There is no code written for this step, and it occurs because the class inherits from the mrjob library. This shuffle step organizes the key-value pairs so that there's only one key for each unique key, and combines the values into a list.

The reducer function might not look like it's doing as much as the mapper function, but it's just as important. The reducer function takes three parameters: `self, key, and values`:

1. The `self` parameter is used in Python to represent the instance of the class.
2. The `key` parameter represents the key of the key-value pair created in the mapper function. In this example, the key is "bacon."
3. The `values` parameter is a list of values created in the mapper function. We want to sum all of these values. Recall that from the mapper function the yield was used to produce multiple outputs. With the reducer we'll produce the key and sum of all the values assigned with it:

```
def reducer(self, key, values):
   yield key, sum(values)
```

The final bit of code, shown below, is conventional Python code for running the program:

```
if __name__ == "__main__":
   Bacon_count.run()
```

All together your code should look like the following:

```
from mrjob.job import MRJob
class Bacon_count(MRJob):
   def mapper(self, _, line):
       for word in line.split():
           if word.lower() == "bacon":
               yield "bacon", 1

   def reducer(self, key, values):
       yield key, sum(values)
if __name__ == "__main__":
   Bacon_count.run()
```

You might have noticed that nowhere in the code is a file imported or opened. The mrjob library works by reading in a file passed to it in the terminal.

Let's use a file full of random words, **input.txt (https://courses.bootcampspot.com/courses/138/files/19401/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/19401/download?wrap=1)**, saved in the same directory as the mrjob file we just created.

Run the following code in the terminal:

```
$ python bacon_counter.py input.txt
```

The output might seem a little confusing at first, but all that the extra output is stating, is that it created a temporary folder, outputted the results

to that folder, and then removed the folder. In the middle of all that, you'll find the result of the `MRJob`.

As mentioned, the mrjob library won't be used in production anywhere, as we'll employ better libraries for that purpose. It's good to practice with smaller files and to try predicting the result, and then confirm with mrjob.

SKILL DRILL

Give it a shot! Create your own sample text file and see if you can find the amount of times different words appear.

16.3.1 Key Features of Spark

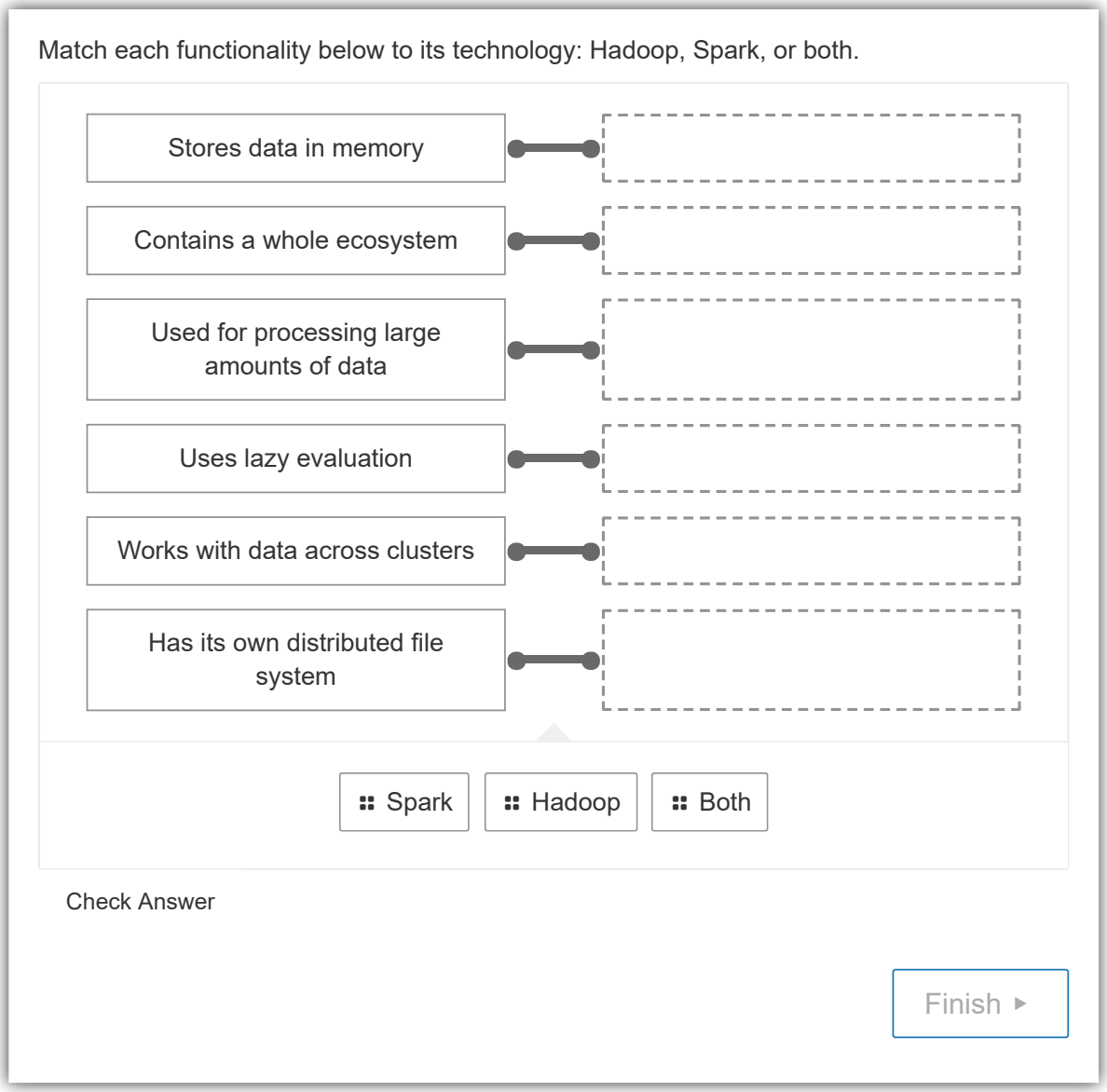
You and Jennifer are really starting to get the hang of this! In your next conversation with SellBy, they tell you they've been using Spark to handle their datasets, and ask if you'll be able to use it, too. Thankfully, you and Jennifer had already set up a working meeting to dig into Spark, so you let them know that you'll be up and running in no time.

Hadoop is an ecosystem for handling big data. Expect to spend significant time configuring multiple servers or computers, as well as researching which technology can best deliver your big data solution. With the growing interest in big data and the ease of access to cloud technology, which we'll cover later, Hadoop is no longer required. New technologies allow more flexibility in data processing. One of these technologies is Spark.

Apache Spark (Spark) is a unified analytics engine for large-scale data processing. Spark lets you write applications in code that can run on Hadoop. However, Spark doesn't have to run on Hadoop, as it can run in stand-alone mode or in the cloud. Spark can be 100 times faster than Hadoop. Just like Hadoop's MapReduce, Spark works with data spread across a cluster, or a group of computers that work together.

Spark uses in-memory computation instead of a disk-based solution, which means it doesn't need to talk to the HDFS each time and can retain as much as HDFS can in-memory. Spark uses lazy evaluation, which delays the evaluation of an expression or command until the value is needed.

For example, when you direct Spark to count all the product reviews and then group them by star rating, Spark is ready to start, but you'll need to initiate the task—at this point, no counting or grouping has been done, only the instructions have been given. Once you give the go-ahead, Spark will then count and group the reviews all at once.



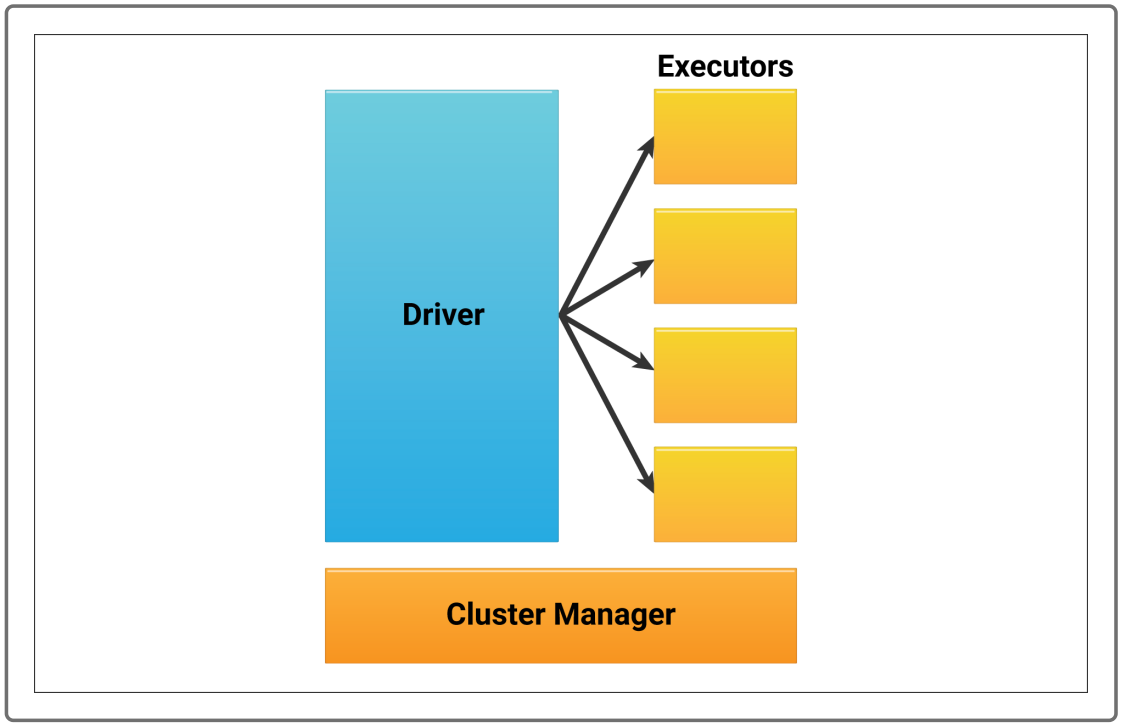
16.3.2 Spark Architecture

Your client is very curious about what goes on behind the scenes. They are familiar with Spark, as it has become so popular in the industry, but they don't understand how it works. In order to be fully prepared for your presentation, the two of you prepare to get into the architecture of how Spark actually works.

The Spark architecture includes the driver, executors, and the cluster manager:

- The **driver** is the heart of the application. It is responsible for maintaining the application information; responding to the code or input; and analyzing, distributing, and scheduling work to the executors.
- The **executors** perform the code assigned by the driver and then report the state of the computation to the driver.
- The **cluster manager** controls the driver and executors and allocates resources to the machines on the Spark applications. The cluster manager is an external service for acquiring resources on the cluster. Spark can either use it's own standalone cluster manager that comes

standard with Spark or another application (e.g., Apache Mesos, Hadoop YARN).



Think of the driver as a manager who assigns work to employees, who then perform the tasks and report back to the manager whether the task was finished or not. The cluster manager functions as a budget manager who allocates the specific amounts to pay employees for performing tasks—the more they are paid, the more work they do.

16.3.3 Spark Parallelism

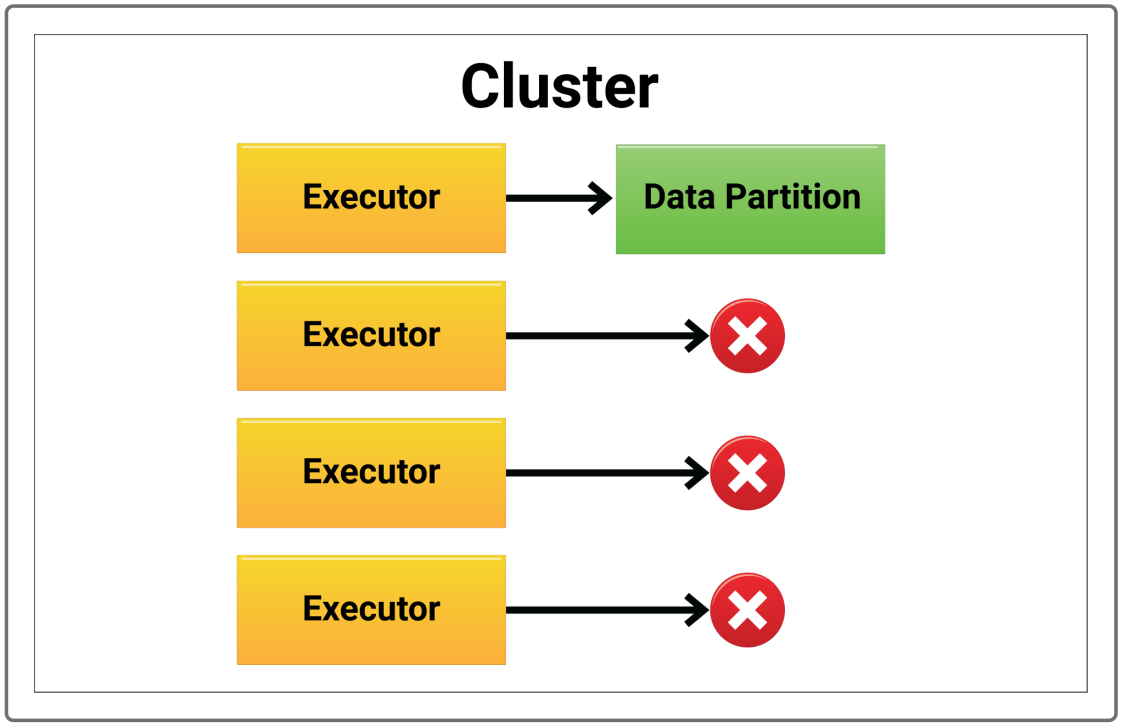
You recall that Hadoop worked on the distributed file system and that Spark can do something similar. Continuing to learn how Spark works, you dive deeper into how Spark can work with data stored in different partitions and operate in parallel.

Parallelism in Spark entails running work through a cluster (group of computers) concurrently, instead of performing all work on a single computer. Recall when you and Jennifer were each counting video game reviews in two separate datasets, and you were working parallel to each other. You did not need to wait for Jennifer to do anything before you could start counting your reviews. This is exactly what Spark is doing.

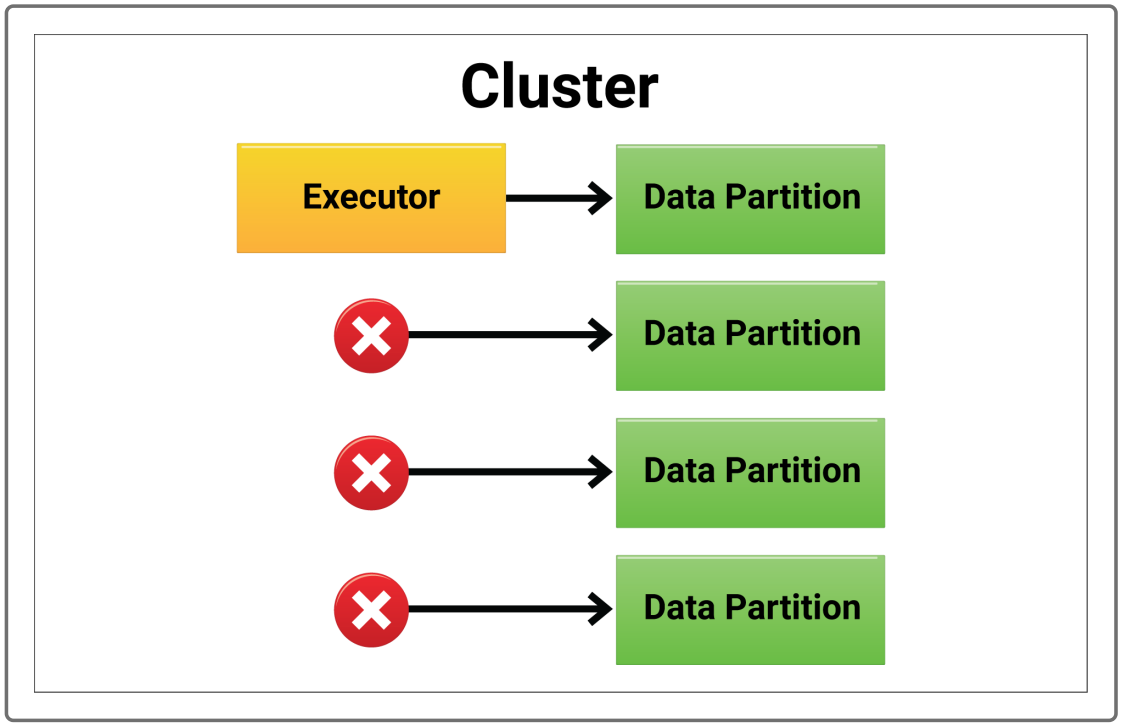
Data is broken into **partitions,** meaning a chunk of your data will sit on a physical machine in your cluster. If your Spark application has three machines, each one of those machines will have a piece of your original dataset. For example, a dataset with 1,000 rows of data might have 334 rows on one machine, 333 on a second, and 333 on a third.

Note the following guidance when running code in parallel: If there is only one partition but many executors, the parallelism is only one. One machine

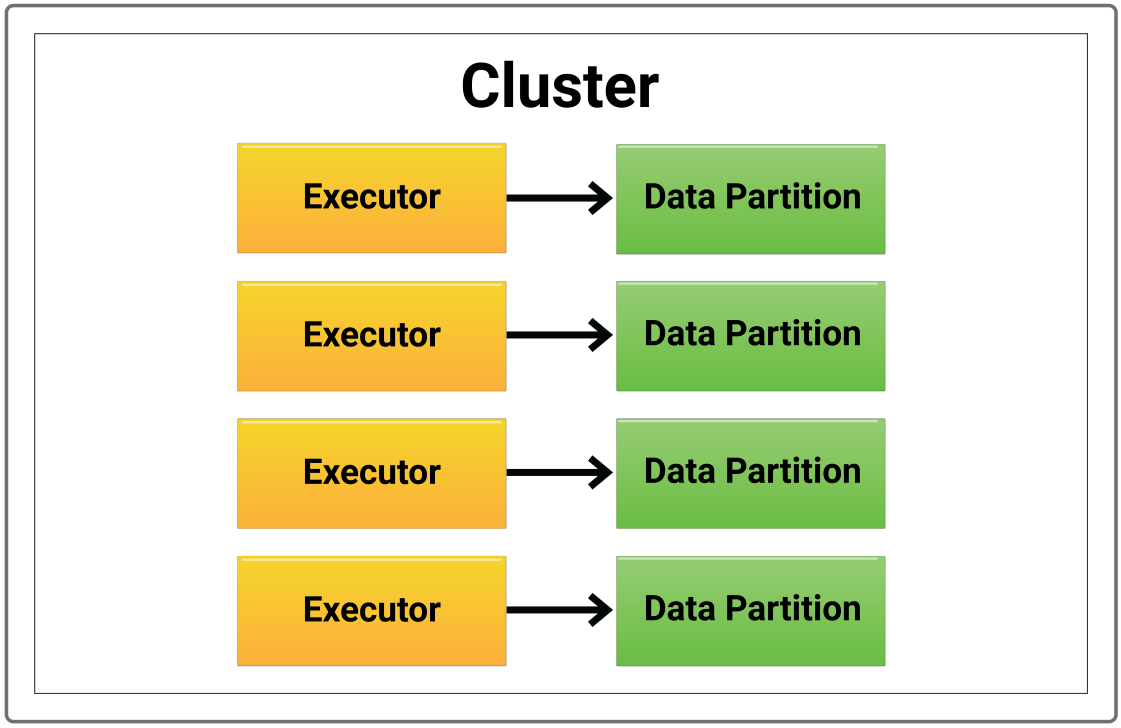
can only work with one executor. The following image visualizes this concept:



If there are many partitions but only one executor, the parallelism is still only one. One executor is assigned to one machine, as shown in the following image:



Each executor needs to work on each partition for **perfect parallelism**, as shown below:



Consider the manager example: If there is only one task but many employees, that task is completed in the time frame that one employee

can accomplish it in, and at the pay rate of one employee.

Next, let’s take a look at Spark’s API.

16.3.4 Spark API

Now that you and Jennifer have an overview of how to use Spark, it's time to dig in and plug in the necessary APIs.

Language APIs

Spark is very accessible through different programming languages. Spark was written in Scala, a tough programming language. However, there is an API that works with many languages to translate the code into Spark code and execute.

Spark has an API that supports the following languages:

- Java
- Python
- SQL
- R

This course will focus on using the Python flavor of Spark, or PySpark.

Data APIs

Spark supports two different sets of data APIs:

- **Low-level unstructured API** is mostly outdated but deals with **resilient distributed datasets (RDDs)**, which are an immutable collection of records that can be operated in parallel. These were part of early versions of Spark.
- **High-level API** consists of structured data such as comma-separated values (CSV).

The high-level API consists of three core forms of data that you'll work with in Spark. Notice that all three contain familiar structures:

- Datasets
- DataFrames
- SQL tables and views

We'll focus on Spark's high level, but it's important to understand that there is some lower level data that Spark can work with. You won't use the low level APIs very often, and generally almost all situations will be better served using structured APIs. However, there are a couple of scenarios where low-level APIs might apply:

- You need finely tuned control over the data in your clusters that high-level APIs can't provide.
- Your role involves maintaining legacy code that uses low-level APIs.

Now that you have some background in Spark, let's set it up on your machine.

Which of the following is true about APIs? (Select all that apply)

- [ ] Allow the use of multiple languages to achieve the same result.
- [ ] A place to store data.
- [ ] A way to interact with a website's endpoint.
- [ ] Allow you to change the way a program works.

Check Answer

Finish ▸

16.4.1 PySpark in Google Colab Notebooks

Your client requests full transparency so that they can eventually maintain this project on their own in the future. Therefore, you'll need to perform your work in a place that is accessible beyond your local laptop. The solution to this will be using cloud-based notebooks.

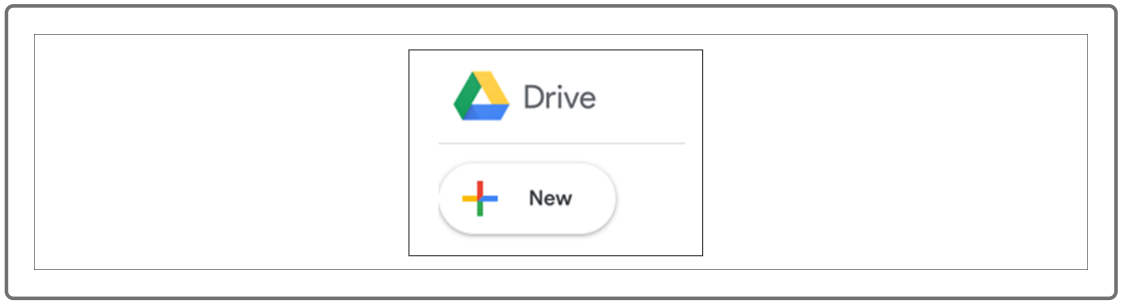
Before we can begin using Spark, we need a place to do so. **Cloud-based notebooks** provide a remote workspace with stronger resources than our local laptop might allow. Cloud notebooks permit us to share our work with others, such as coworkers, similar to GitHub.

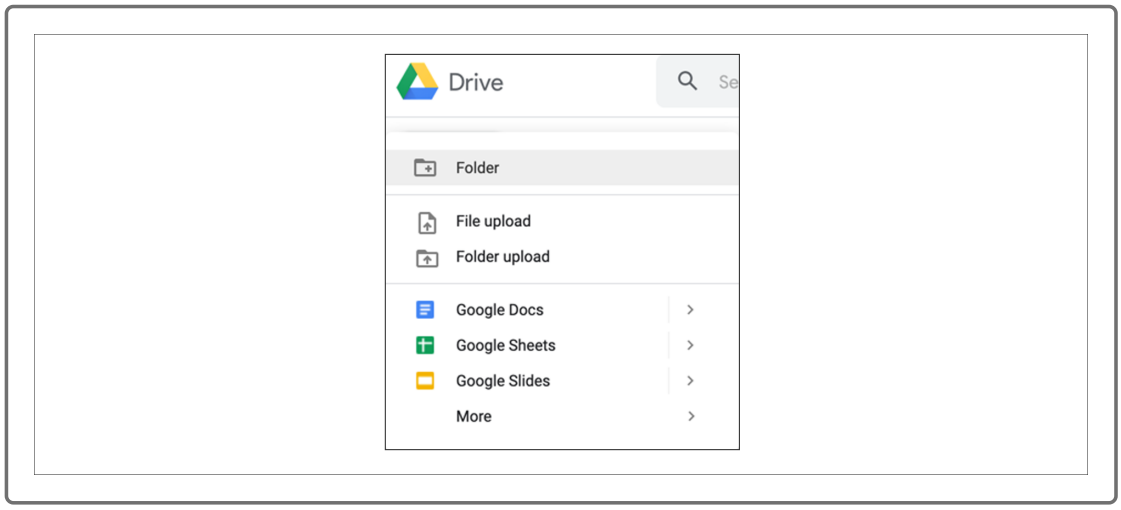
We'll use **Google Colaboratory (https://colab.research.google.com/notebooks/welcome.ipynb)** (Colab), which are Google-hosted notebooks. Some setup is required, so we'll start there before getting back to the basics of PySpark.

First, you'll need a Google account. If you don't already have an account, be sure to sign up for one.

Once you set up an account, navigate to Google Drive and click the "New" button and select "Folder" to create a new folder. Refer to the following

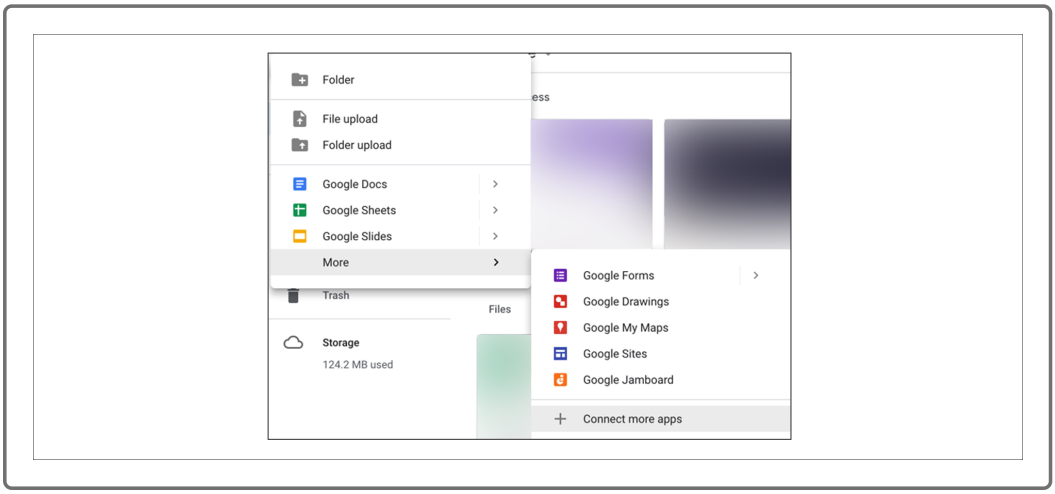
screenshots. Name the folder “DataClassNotebooks.”



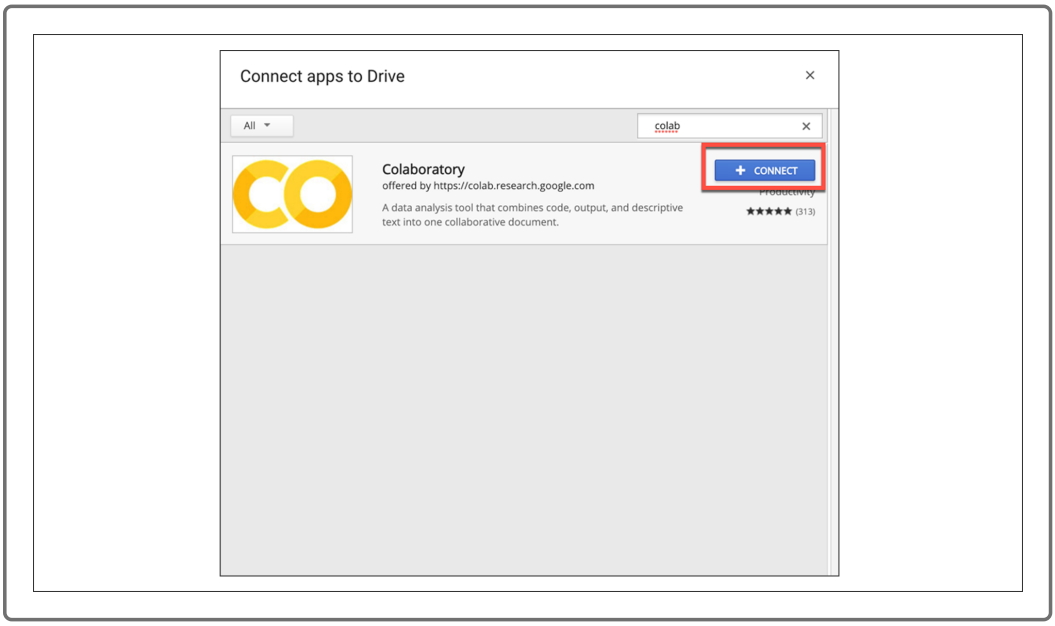


Navigate to the new folder. Once in the notebook, we’ll need to connect (download) our Google Colab application by following these steps:

1. Click “New.”
2. Scroll down to “More” and expand the dropdown menu.
3. At the bottom of the menu, click “Connect more apps.”

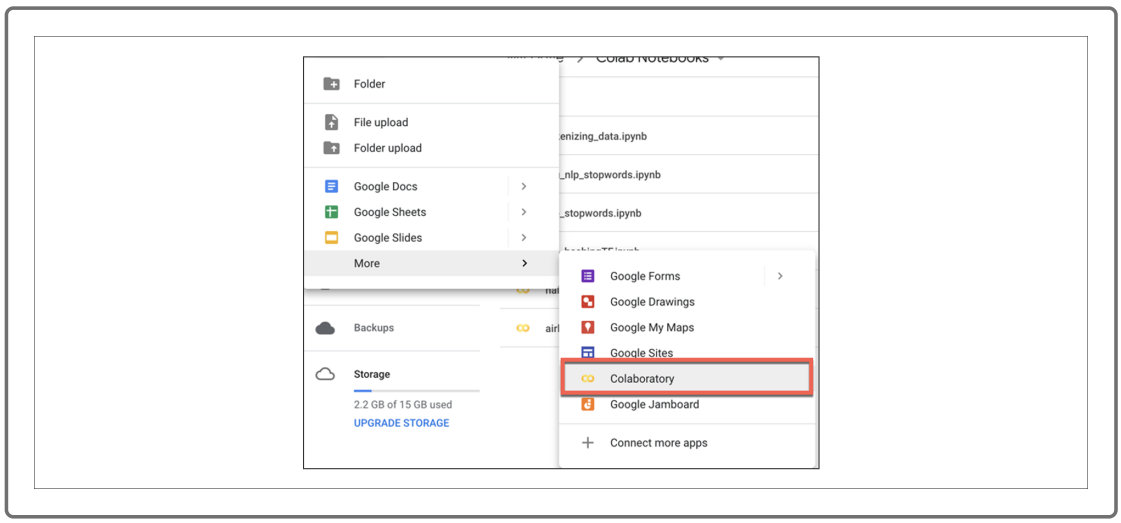


4. Type “colab” in the top-right search field and press Enter to search for the Colaboratory application:



5. Click the “Connect” button to download the Colaboratory application.

Now you can create a Colab Notebook by clicking “New” followed by “More,” and then selecting “Colaboratory.” So let’s do just that!



A new tab will launch with a new notebook. The functionality is very similar to using Jupyter Notebook, except now everything is hosted online.

Just like we do with our local environment, we need to install packages for libraries we want to use. PySpark does not come native to Google Colab and needs to be installed. We'll do so by running the following code in the first cell:

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"

# Start a SparkSession
import findspark
findspark.init()
```

NOTE

> Don’t be scared by this installation code. All this does is install Spark for our notebook to use. It can be copied and pasted into any new notebooks you wish to run.

We are now up and running with cloud notebooks with PySpark.

16.4.2 Spark DataFrames and Datasets

It's time to start working with Spark! You'll start by using DataFrames. You will soon see that Spark has a lot of similarities to the Pandas library, which you have used before.

Working in Spark requires us to put data into DataFrames. If you're wondering if these DataFrames are comparable to those in Pandas, you're correct—Spark DataFrames are very similar. Just as in Pandas, the first step is to load your data into a DataFrame.

What criteria should a DataFrame meet? Select all that apply.

- [ ] The DataFrame must be converted from a CSV.
- [ ] Every column should have the same number of rows in the dataset.
- [ ] Each column should have a data type that is consistent with the column.
- [ ] It needs to have a schema.

Check Answer

Finish ▸

The schema, or structure, for DataFrames and datasets contains the column names and the data types contained within.

The schema can be inferred automatically by letting Spark determine the schema on its own when data is read in or you can define the schema manually and tell spark to use that.

In this module, we'll use the Python version of Spark, PySpark.

Which of the following is true about Spark DataFrames API? Select all that apply.

- [ ] Spark commonly switches between low- and high-level data use.
- [ ] Spark DataFrames are similar to Pandas DataFrames.
- [ ] Mixed data types can be stored in the same column in a DataFrame.
- [ ] Schema can be read when the data is loaded or manually defined.

Check Answer

Finish ▸

Using your Google Colab Notebook, with PySpark installed, follow along with the code.

Create a Spark session by importing the library and setting the `spark` variable to the code below:

```
# Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("DataFrameBasics").getOrCreate()
```

NOTE

> Every time you open a new Google Colab Notebook, run the Spark installation and then start a Spark session. The app name can be different for each notebook.

This creates a Spark application called “DataFrameBasics.”

Spark enables us to create a DataFrame from scratch by passing in a list of tuples to the `createDataFrame` method followed by a list of the column names. The `show` method will display the DataFrame, which is similar to using the `head()` function in Pandas. Enter and run the following code:

```
dataframe = spark.createDataFrame([
                                    (0, "Here is our DataFrame"),
                                    (1, "We are making one from scratch"),
                                    (2, "This will look very similar to a Pandas DataFrame")
], ["id", "words"])

dataframe.show()
```

```
+---+--------------------+
| id|               words|
|  0|Here is out DataF...|
|  1|We are making on...|
|  2|This will look ve...|
+---+--------------------+
```

Spark also lets us import data directly into a DataFrame. To do this, we import `SparkFiles` from the pyspark library that allows us to retrieve files. The next three lines of code tell Spark to pull data from Amazon’s Simple Storage Service (S3), a cloud-based data storage service. This boilerplate

code can be used to read other public files hosted on Amazon's services. We'll dive more into cloud storage later on in the module.

In the next cell in your Colab notebook, run the following code:

```
# Read in data from S3 Buckets
from pyspark import SparkFiles
url = "https://s3.amazonaws.com/dataviz-curriculum/day_1/food.csv"
spark.sparkContext.addFile(url)
df = spark.read.csv(SparkFiles.get("food.csv"), sep=",", header=True)
```

Type the following code to use `show()` again to display the results, as follows:

```
# Show DataFrame
df.show()
```

```
+-------+-----+
|   food|price|
+-------+-----+
|  pizza|    0|
|  sushi|   12|
|chinese|   10|
+-------+-----+
```

Spark will infer the schema from the data, unless otherwise specified. We can check the schema by running the following code:

```
# Print our schema
df.printSchema()
```

Spark also allows users to view columns and a dataset description by running each of the code blocks:

```
# Show the columns
df.columns
```

```
['food', 'price']
```

```
# Describe our data
df.describe()
```

```
DataFrame[summary: string, food: string, price: string]
```

Notice that the DataFrame is claiming that `price` is a string. Generally, `price` is either stored as an integer or floating-point number, so you'll need to change this column.

In this case, we can set our schema and then apply it to the data. We'll start by importing the different types of data with the following code:

```
# Import struct fields that we can use
from pyspark.sql.types import StructField, StringType, IntegerType, StructTy
```

Next, create the schema by creating a `StructType`, which is one of Spark's complex types, like an array or map. The `StructField` will define the column name, the data type held, and a Boolean to define whether null values will be included or not:

```
# Next we need to create the list of struct fields
schema = [StructField("food", StringType(), True), StructField("price", Inte
schema
```

Next, enter the code that will pass the schema just created as fields in a `StructType`. All this will be stored in a variable called `final`:

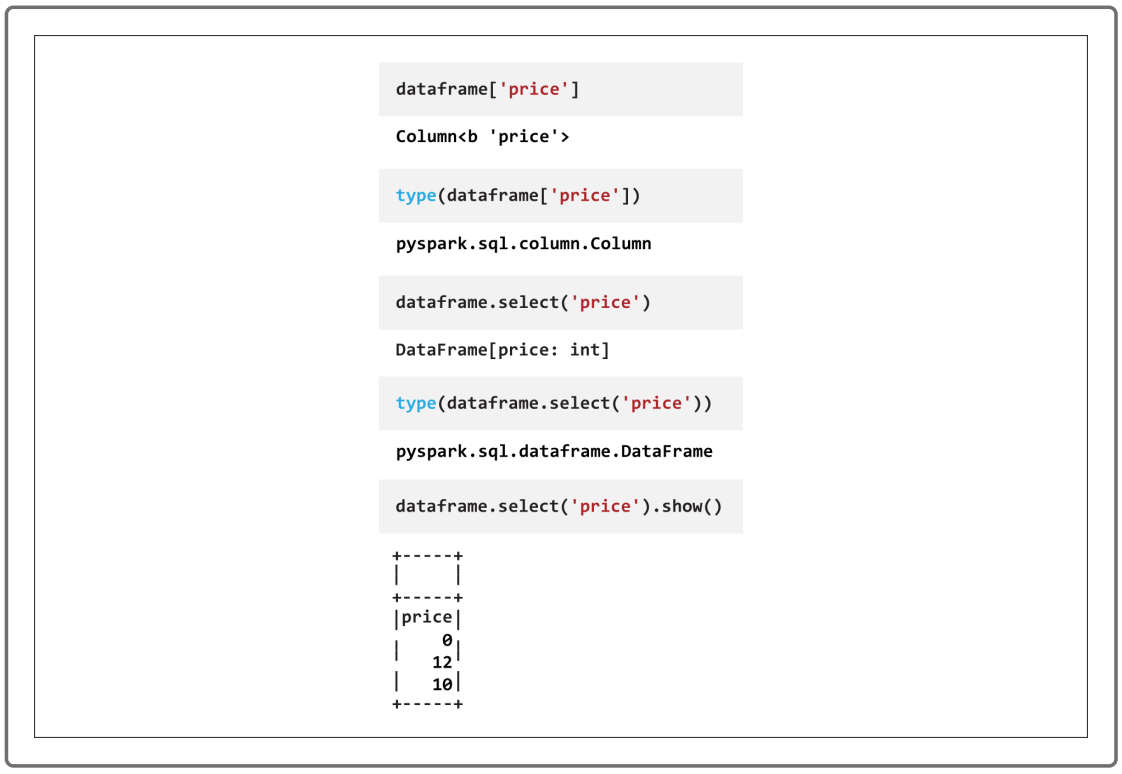
```
# Pass in our fields
final = StructType(fields=schema)
```

```
final
```

Now that we have a predefined schema, we can read in the data again, only this time passing in our own schema. Type and run the following code in a new notebook cell:

```
# Read our data with our new schema
dataframe = spark.read.csv(SparkFiles.get("food.csv"), schema=final, sep=","
dataframe.printSchema()
```

There are a few different ways to access our data with Spark. Run the following commands and look at the results:



Again, you may notice some similarities to Pandas. For example, in both Pandas and Spark, you can select a column using the DataFrame name, followed by the column's name in square brackets. In Pandas, you can

quickly take a look at a DataFrame using `head()`; in Spark, you can do something similar using `show()`.

NOTE

> You might notice that code like `dataframe['price']` isn't performing as expected. After running this code, we get the column name, but no results until the `show()` function runs. `show` is an action, whereas `select` is a transformation. We'll cover what this means in the next section.

We can manipulate columns in Spark as well. Run the code below and guess what will be displayed:

```
# Add new column
dataframe.withColumn('newprice', dataframe['price']).show()
# Update column name
dataframe.withColumnRenamed('price','newerprice').show()
# Double the price
dataframe.withColumn('doubleprice',dataframe['price']*2).show()
# Add a dollar to the price
dataframe.withColumn('add_one_dollar',dataframe['price']+1).show()
# Half the price
dataframe.withColumn('half_price',dataframe['price']/2).show()
```

The first cell created a new column with all the same rows, but it gave a new name to the column. The second cell changed the name of the column without duplicating data:

```
# Add new column
dataframe.withColumn('newprice', dataframe['price']).show()
```

```
+-------+-----+--------+
| food  |price|newprice|
+-------+-----+--------+
|  pizza|    0|       0|
|  sushi|   12|      12|
|chinese|   10|      10|
+-------+-----+--------+
```

```
# Update column name
dataframe.withColumnRenamed('price', 'newerprice').show()
```

```
+-------+-----------+
| food  |newerprice |
+-------+-----------+
|  pizza|        0  |
|  sushi|       12  |
|chinese|       10  |
+-------+-----------+
```

The next three cells created a new column, but they performed some operation on the original:

```
# Double the price
dataframe.withColumn('doubleprice', dataframe['price']*2)show()
```

```
+-------+-----+-----------+
| food  |price|doubleprice|
+-------+-----+-----------+
|  pizza|    0|          0|
|  sushi|   12|         24|
|chinese|   10|         20|
+-------+-----+-----------+
```

```
# Add a dollar to the price
dataframe.withColumn('add_one_dollar',dataframe['price']+1).show()
```

```
+-------+-----+--------------+
| food  |price|add_one_dollar|
+-------+-----+--------------+
|  pizza|    0|             1|
|  sushi|   12|            13|
|chinese|   10|            11|
+-------+-----+--------------+
```

```
# Half the price
dataframe.withColumn('half_price',dataframe['price']/2).show()
```

```
+-------+-----+----------+
| food  |price|half_price|
+-------+-----+----------+
|  pizza|    0|       0.0|
|  sushi|   12|       6.0|
|chinese|   10|       5.0|
+-------+-----+----------+
```

Next, we'll dive into Spark functions and why some of these commands produced actual results while others just relayed information about our data.

16.4.3 Spark Functions

Working with Spark DataFrames seems simple enough. Now you'll run some of PySpark's functions and actually analyze your data.

In the last section, you might have noticed that when calling a column with Spark, no real results were shown. As you might recall, Spark uses lazy evaluation. This means that Spark takes a list of instructions and formulates the best way to fulfill them, and then waits until you tell Spark to complete them. The process of listing and reading the instructions is called **transformation**, and your directive to complete them is called an **action**.

For example, say you have a new coffee maker. First, you read the instructions to learn what you need to do. You don't immediately make coffee, though. Instead, you wait for someone to ask if you can make coffee. Once you receive that request, you follow all the steps to make coffee. The transformation is you reading the instructions, and the action is the request to make coffee.

Let's start with an example.

NOTE

> If you're using a new notebook for the next example, be sure to install Spark in the first cell, as we did in the previous example.

Start by creating a Spark session and loading in data, using the following code:

```
# Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("DataFrameFunctions").getOrCreate()
```

```
# Read in data from S3 Buckets
from pyspark import SparkFiles
url ="https://s3.amazonaws.com/dataviz-curriculum/day_1/wine.csv"
spark.sparkContext.addFile(url)
df = spark.read.csv(SparkFiles.get("wine.csv"), sep=",", header=True)

# Show DataFrame
df.show()
```

Now we'll use this data to identify the difference between transformations and actions.

Transformations

Transformations are the instructions for the computation. With the data loaded in, let's perform some transformations. What do you think will happen after we run the following code?

```
# Order a DataFrame by ascending values
df.orderBy(df["points"].desc())
```

If you guessed that nothing would happen yet, you would be correct. Here we applied the transformation to order the DataFrame by points in descending order. All we're doing is telling Spark that we want this DataFrame to be organized in this particular way, and Spark says, "Okay, got it—just let me know when you want me to do this."

Actions

Actions direct Spark to perform the computation instructions and return a result. What do you think will happen when you add `.show(5)` to the DataFrame?

```
df.orderBy(df["points"].desc()).show(5)
```

```
+-------+--------------------+--------------------+------+-----+----------+
|country|         description|         designation|points|price|  province|
+-------+--------------------+--------------------+------+-----+----------+
|  Italy|Here's a ���wow...|            Messorio|    99|  320|   Tusca
|     US|A stupendous Pino...|Precious Mountain...|    99|   94|California|
|  Italy|The 2007 Ornellai...|           Ornellaia|    99|  200|   Tuscany|
|     US|Depth and texture...|Royal City Stoner...|    99|  140|Washington|
| France|A magnificent Cha...|Dom Prignon Oeno...|    99|  385| Champagne|
+-------+--------------------+--------------------+------+-----+----------+
only showing top 5 rows
```

If you guessed that a DataFrame organized by points would display, you would be correct. The `show(5)` method is an action that tells Spark to show the first five results.

Recap

Let's break down these concepts one more time:

- `orderBy()` and `desc()` are transformations telling Spark how to organize the data. Spark will read these transformations as instructions, but it won't act on them just yet.

- `show()` is an action that gives the go-ahead for Spark to run all of those transformations and to produce a result.

SKILL DRILL

Try to get the top 50 rows for values in ascending order on the points column.

More Functions

Let's take a look at a few more functions that we can use with Spark. As you continue to enter the code in your notebooks, keep in mind which methods you think are transformations and which ones are actions.

Spark can import additional functions, such as averages. Type and run the following code:

```
# Import functions
from pyspark.sql.functions import avg
df.select(avg("points")).show()
```

```
+------------------+
|       avg points |
+------------------+
| 87.88834105383143 |
+------------------+
```

The `avg()` function is the transformation, and `show()` is the action.

Spark can filter on columns by supplying the name of the column and operator and what to compare it against. Refer to the following code and results:

```
# Filter
df.filter("price<20").show(5)
```

```
+--------+--------------------+-----------+------+-----+----------+--------------------+-----------------+--------------+--------------------+
| country|         description|designation|points|price|  province|            region_1|         region_2|       variety|              winery|
+--------+--------------------+-----------+------+-----+----------+--------------------+-----------------+--------------+--------------------+
|Bulgaria|This Bulgarian Ma...|     Bergul|    90|   15|  Bulgaria|                null|             null|       Mavrud|       Villa Melnik|
|   Spain|Earthy plum and c...|     Amandi|    90|   17|    Galicia|       Ribeira Sacra|             null|       Menc�_a|      Don Bernardino|
|      US|There's a lot to ...|       null|    90|   18|California|Russian River Valley|           Sonoma|    Chardonnay|          De Loach|
|      US|Massively fruity,...|       null|    91|   19|    Oregon|   Willamette Valley|Willamette Valley|   Pinot Gris|   Trinity Vineyards|
|Portugal|It is the ripe da...|    Premium|    91|   15|  Alentejo|                null|             null|Portuguese Red|Adega Cooperativa...|
+--------+--------------------+-----------+------+-----+----------+--------------------+-----------------+--------------+--------------------+
only showing top 5 rows
```

`Filter` is the transformation and `show` is the action.

We can also filter and select certain columns:

```
# Filter by price on certain columns
df.filter("price<20").select(['points','country', 'winery','price']).show(5)
```

```
+------+---------+--------------------+-----+
|points|  country|              winery|price|
+------+---------+--------------------+-----+
|    90| Bulgaria|        Villa Melnik|   15|
|    90|    Spain|      Don Bernardino|   17|
|    90|       US|             De Loach|   18|
|    91|       US|   Trinity Vineyards|   19|
|    91| Portugal|Adega Cooperativa...|   15|
+------+---------+--------------------+-----+
```

Both `filter` and `select` are separate transformations, and `show` is again the action.

Spark has multiple ways to perform transformations. For instance, how we were filtering our DataFrame was actually using SQL context with `price<20`.

However, we can also perform the same transformations using Python:

```
# Filter
df.filter("price<20").show(5)
# Filter by price on certain columns
df.filter("price<20").select(['points','country', 'winery','price']).show(5)

# Filter on exact state
df.filter(df["country"] == "US").show()
```

You might notice that there are more transformations than actions. Usually, we want to do several things with a dataset, but we'll only want to see those results in a few ways.

SKILL DRILL

Using both the SQL and Python context, use filtering to find the rows that contain a bottle of wine over $15 and that comes from California.

Now that you have an understanding of using Spark, it's time to dive into the next step of your challenge: learning how computers interpret textual data. For this step we'll explore natural language processing, which we'll use in conjunction with Spark.

16.5.1 Natural Language Processing

You and Jennifer feel you have a good understanding of PySpark and your ability to handle big data. Now it's time to focus on the next part of the challenge, which is handling text data. To start, you'll learn about natural language.

Natural language processing (NLP) is a growing field of study that combines linguistics and computer science for computers to understand written, spoken, and typed natural language. NLP is the process of converting normal language to a machine readable format, which allows a computer to analyze text as if it were numerical data.

While NLP has a wide variety of use cases, and the field is rapidly growing, there are a few use cases that are particularly interesting:

- **Analyzing legal documents:** NLP can be used to analyze many types of legal documents. This can improve the outcome of a given case, as lawyers and staff can find critical information quickly.
- **U.S. Securities and Exchange Commission (SEC) filings:** NLP is used to analyze SEC filings for various businesses. Companies use NLP to analyze filings for real-time business intelligence.

- **Chatbots:** Chatbots are one of the most popular use cases. Chatbots can be used for selling products, customer support, and even medical help.

At this point, you might ask how this relates to big data. Due to the massive amounts of text data needed to drive insights, we'll have to learn how to manage that data. There are a number of important use cases to delve into:

- **Classifying text:** For many of the aforementioned use cases to work, a computer must know how to classify a given piece of text. Classification can mean a few different things in NLP. You can have classification of specific words, even specifying what the part of speech is. You can also classify what the text is as a whole.
- **Extracting information:** Many NLP tasks require the ability to retrieve specific pieces of information from a given document. Think of the case where we are extracting data from law documents. You might want to extract certain aspects of that document to present good cases.
- **Summarizing a document:** Summarization is a key aspect of NLP. It helps solve quite a few different problems. You can essentially create a model that summarizes a given document. This can be helpful to understand the high-level details of law documents, articles, and much more.

Suppose you are developing a system to learn more about calls from a call center. What kind of NLP use case would you use to find the key points from each call? Select all that apply.

- [ ] Classifying text
- [ ] Extracting information
- [ ] Summarizing documents

Check Answer

Finish ▸

Context of Natural Language

Natural language can be complicated because the way it's written is not always how it is intended. Therefore, you might need the full context to understand the meaning.

Sarcasm is a great example. Say you had a bad experience at a restaurant. Your friend asks if you liked your meal and you reply "Oh, yeah, the food was amazing if you like dry, bland food." A friend familiar with your humor would understand your true intentions behind the quip. However, a straight reading, without detecting sarcasm, would give the impression you prefer dry, bland food.

Another challenge is interpreting the tone behind the text. For instance, snidely remarking "Great" and enthusiastically exclaiming "Great!" reveal two distinct tones but, in text, it is the same word.

These are just two examples of the complexity of dealing with natural language. In the next section, we'll show how a computer does its best to interpret language.

16.5.2 Computers Understanding Language

You have been programming for a bit and know that computers like explicit directions to execute programs. With this in mind, you can start to break down the process for a computer to understand language. Since the game reviews you're processing are all in text format, you decide to focus on that aspect of NLP.

Computers comprehend language differently than humans do, so you have to teach the computer how to understand natural language.

On a fundamental level, the computer processes information as 1s and 0s. Machine-level programming languages, which are simple programming languages, process those 1s and 0s and can perform simple operations. Higher-level programming languages execute more complex tasks. Each programming language builds on one another. This high-level overview explains in part what happens when a computer processes language.

We need NLP so that computers can better analyze language and someday communicate seamlessly with humans. Despite significant advancements in NLP, computers still struggle to understand the whole context of a text. For now, they just understand definitions and the literal

meaning of a text. They don’t understand sarcasm or subtext or anything not explicitly defined or expressed.

16.5.3 NLP Core Concepts

You're starting to piece together why it's so important to have NLP. Now Jennifer has asked you to look into some of the core concepts in NLP so that you have a good foundation to build on.

Before we can write algorithms to help computers understand language information, there are some key concepts and use cases that we need to know: tokenization, normalization, part-of-speech tagging, natural language generation, bag-of-words model, n-grams, and text similarity. This section will cover each of these concepts.

Tokenization

Tokenization is the concept of splitting a document or sentence into small subsets of data that can be analyzed. It's essentially the building block of most NLP use cases. Tokenization can be performed by word or sentence.

To tokenize by word, you take a given sentence or document and split it into a list of words, with each sentence or word representing a token. A

sentence tokenized by word would look like the following:

Original sentence: `I am enjoying learning about NLP.`

Tokenized by word: `['I', 'am', 'enjoying', 'learning', 'about', 'NLP', '.']`

To tokenize by sentence, you would provide a document with at least one sentence. Ideally, you would use more than one sentence, as tokenization splits a document up by sentences. Here's an example of a two-sentence document and how it would be split up. Notice how each sentence is a separate string:

Original sentence: `There is a concept in NLP called tokenization. There are two types of tokenization; word and sentence.`

Tokenized by sentence: `['There is a concept in NLP called tokenization.', 'There are two types of tokenization: word and sentence.']`

You can tokenize using NLTK or spaCy libraries, which we'll cover later.

Normalization

Normalization is the concept of taking misspelled words and converting them into their original form. This is another building block in NLP in that it helps get the text to a readable form and allows us to create other use cases on top of it. Essentially, it makes it easier for us to create NLP programs, and it improves the output of those programs. There are numerous ways to accomplish this, but we'll focus on just two practices: stemming and lemmatization. Stemming and lemmatization are similar in that they both remove the suffix from a word, but there are some differences in how smooth or rough the cutoff tends to be:

- **Stemming** removes the suffix from a word and reduces it to its original form. This serves as a "rough" cut off of the end of the word.

- **Lemmatization** removes the suffix from a word and reduces it to its original form. Lemmatization tends to be a “smoother” cut off of the end of the word. It tries to return to the original root word.

Part-of-Speech Tagging

When researching or practicing NLP, you’ll often hear of **part-of-speech tagging** (PoS), which can be helpful for a variety of different models in NLP. PoS tagging is the concept of finding each word’s part of speech in a given document, as the example below illustrates:

Word	PoS and Tags
I	Personal pronoun (PRP)
enjoy	Verb, non-third person (VBP)
biking	Verb, gerund, present participle (VBG)
on	Preposition or subordinating conjunction (IN)
the	Determiner (DT)
trails	Noun, plural (NNS)

If you want to try this, we will do so using the Natural Language Toolkit library. To install follow the steps below.

1. Activate your environment

   ```
   pip install nltk
   ```

2. Install NLTK

3. Install additional NLTK tools

```
python -m nltk.downloader popular
```

Go ahead and create a new Python file. You can add this code to get the PoS tags for the text you provide:

```
import nltk
from nltk import word_tokenize
text = word_tokenize("I enjoy biking on the trails")
output = nltk.pos_tag(text)
print(output)
```

When you run this file, it will print out the PoS tags for each word as shown in the table above.

SKILL DRILL

Go ahead and run this code with a sentence of your choice.

Natural Language Generation

Natural language generation is a growing field in NLP that entails writing code in such a way that it will generate new text. Popular examples include chatbots, automated custom reports, and custom webpage content. In this module, we won't write code to generate new text.

Chatbots are easy to create and so are increasingly used in data science, especially NLP. Most of the technology already exists to create meaningful content from natural language generation.

Bag-of-Words

When we're building datasets for NLP, we need to consider how our program will interact with the text document we provide. If we create a **bag-of-words (BoW)** model (i.e., the most frequent words), we can build models from that.

The basic idea behind this model is this: We have a document of words, but we don't care about the order of the words. We can count these words and create models based on how frequently they appear.

---

n-gram

In NLP, there is an **n-gram** method, which is a sequence of items from a given text. With n-gram, you create groupings of items from the text of size 1 or more. The following n-grams are common:

- **Unigram** is an n-gram of size 1.
- **Bigram** is an n-gram of size 2.
- **Trigram** is an n-gram of size 3.

Of course, you can always use larger groupings than just size 3, such as "four-gram," "five-gram," and so on.

N-grams can be used for a variety of NLP tasks, but most involve text mining or extraction. You can also use n-grams for spellcheck and text summarization.

Text Similarity

Another popular use case for NLP is determining document or sentence similarity. These are important use cases, because they can tell us a lot about a document and its contents. There are a number of ways to do text similarity, with varying levels of difficulty. Many of the technologies that we'll discuss have the ability to compare documents against one another.

What is part-of-speech tagging?

- A process in NLP where you can analyze how similar two documents are.
- A process in NLP where you can identify all important elements in a given piece of text.
- A process in NLP where you can find key pieces of grammar information for each word.

Check Answer

Finish ▸

16.5.4 NLP Use Cases

You've learned a lot of the building blocks required to understand NLP use cases, so keep going. Learning these use cases will really help you and Jennifer become successful.

Within NLP, there are a few use cases that are important to creating programs of value. Each of these builds on what we have previously covered. You can build on each of these to create more advanced programs, but we won't worry about that right now.

NLP Analyses

There are three types of NLP analyses:

- **Syntactic analysis** is essentially checking the dictionary definition of each element of a sentence or document. In this type of analysis, we don't care about the words that come before or after the word in question—we just care about the given word.

- **Sentiment analysis** pertains to what the text means. Is it positive, negative, or neutral? You can come up with a score of how positive or negative the text is using NLP.
- **Semantic analysis** entails extracting the meaning of the text. You want to analyze the meaning of each word, and then relate that to the meaning of the text as a whole.

Named-Entity Recognition

In NLP, **named-entity recognition** (NER) is the concept of taking a document and finding all of the important terms therein. By "important," we mean names of places and people, government organizations, and so forth. Many names are already recognized, but you can always add more names to the list of recognized entities, as necessary.

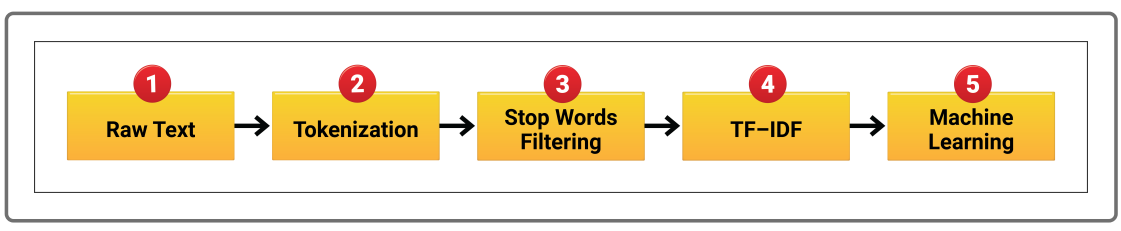
You train a model on data labeled with important entities so that the model can better distinguish which entities should be labeled in a different dataset.

16.5.5 NLP Pipeline

Now that you understand the many steps needed to get regular text into a machine-readable state, it's time to get the process going. This process is called building an NLP pipeline.

NLP is complicated. To manage it, you must **build an NLP pipeline**, a process breaking NLP down into a series of smaller, less complex tasks. Below we'll provide a high overview of this process, and in the next section, we'll dive deeper with the code.

Each step of the NLP pipeline involves a separate task. The output data from one step, in turn, becomes the input data for the next step, with an opportunity to evaluate and refine each task, if needed. A basic NLP pipeline follows:



Here’s a breakdown of each step:

1. **Raw Text:** Start with the raw data.
2. **Tokenization:** Separate the words from paragraphs, to sentences, to individual words.
3. **Stop Words Filtering:** Remove common words like “a” and “the” that add no real value to what we are looking to analyze.
4. **Term Frequency-Inverse Document Frequency (TF-IDF):** Statistically rank the words by importance compared to the rest of the words in the text. This is also when the words are converted from text to numbers.
5. **Machine Learning:** Put everything together and run through the machine learning model to produce an output.

We’ll cover each step of the pipeline in more depth in the next section.

16.6.1 Tokenize Data

NLP seems like a difficult task, and Jennifer is worried if you two can handle this. You know now that NLP can be a complicated task, but you can simplify it by breaking it down into steps. The first step is taking the raw text and separating the words by a process called tokenization.

As we learned in the previous section, tokenizing breaks down sentences into words. To do this with PySpark, we'll use the PySpark Machine Learning (ML) library. Start by creating a new notebook, installing Spark, and creating a SparkSession, as demonstrated in the following code:

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"
```

```
# Start a SparkSession
import findspark
findspark.init()
```

```
# Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Tokens").getOrCreate()
```

Next, we'll import the `Tokenizer` library. Type and run the following code:

```
from pyspark.ml.feature import Tokenizer
```

Spark gives us the ability to create a DataFrame from scratch as well. Although you'll mainly use DataFrames imported from data, the ability to create quick, small DataFrames allows for quick, easy testing. We'll create a small DataFrame that will show the pre-tokenized data, using the following code:

```
# Create sample DataFrame
dataframe = spark.createDataFrame([
    (0, "Spark is great"),
    (1, "We are learning Spark"),
    (2, "Spark is better than hadoop no doubt")
], ["id", "sentence"])

dataframe.show()
```

```
+---+--------------------+
| id|            sentence|
+---+--------------------+
|  0|       Spark is great|
|  1|We are learning S...|
|  2|Spark is better t...|
+---+--------------------+
```

The `tokenizer` function takes input and output parameters. The input passes the name of the column that we want to have tokenized, and the

output takes the name that we want the column called. Type and run the following code:

```
# Tokenize sentences
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
tokenizer
```

The `tokenizer` that we created uses a `transform` method that takes a DataFrame as input. This is a transformation, so to reveal the results, we'll call `show(truncate=False)` as our action to display the results without shortening the output, as shown below:

```
# Transform and show DataFrame
tokenized_df = tokenizer.transform(dataframe)
tokenized_df.show(truncate=False)
```

```
+---+-----------------------------------+-------------------------------------------+
|id | sentence                          |words                                      |
+---+-----------------------------------+-------------------------------------------+
| 0 | Spark is great                    |[spark, is, great]                         |
| 1 | We are learning Spark             |[we, are, learning, spark]                 |
| 2 | Spark is better than hadoop no doubt|[spark, is, better, than, hadoop, no, doubt]|
+---+-----------------------------------+-------------------------------------------+
```

To summarize, we started with a DataFrame that contained full sentences (the big task), then ran the `tokenizer` to break the text down to a list of words (small task).

NOTE

> You may notice that the `tokenizer` looks similar to the `split()` method in Python.

User-defined functions (UDFs) are functions created by the user to add custom output columns.

For the example below, we can create a function that will enhance our `tokenizer` by returning a word count for each line. Start by creating a Python function that takes a list of words as its input, then returns the length of that list. Type and run the following code:

```
# Create a function to return the length of a list
def word_list_length(word_list):
    return len(word_list)
```

Next, we'll import the `udf` function, the `col` function to select a column to be passed into a function, and the type `IntegerType` that will be used in our `udf` to define the data type of the output, as follows:

```
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType
```

Using the `udf` function, we can create our function to be passed in. The `udf` will take in the name of the function as a parameter and the output data type, which is the `IntegerType` that we just imported. Type and run the following code:

```
# Create a user defined function
count_tokens = udf(word_list_length, IntegerType())
```

Now we can redo the `tokenizer` process. Only this time, after the DataFrame has outputted the tokenized values, we can use our own created function to return the number of tokens created. This will give us another data point to use in the future, if needed. Type and run the following code:

```
# Create our Tokenizer
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

# Transform DataFrame
tokenized_df = tokenizer.transform(dataframe)

# Select the needed columns and don't truncate results
tokenized_df.withColumn("tokens", count_tokens(col("words"))).show(truncate=False)
```

```
+---+-----------------------------------+-----------------------------------------+------+
|id | sentence                          |words                                    |tokens|
+---+-----------------------------------+-----------------------------------------+------+
| 0 | Spark is great                    |[spark, is, great]                       |3     |
| 1 | We are learning Spark             |[we, are, learning, spark]               |4     |
| 2 | Spark is better than hadoop no doubt|[spark, is, better, than, hadoop, no, doubt]|7     |
+---+-----------------------------------+-----------------------------------------+------+
```

16.6.2 Stop Words

NLP is starting to seem a little less scary now that each step is being broken down. Now that you have all the words tokenized, you realize that there are some words like “and” and “a” that just don’t really matter in the grand scheme of things. These are called stop words and will be covered next.

Stop words are words that have little or no linguistic value in NLP. Removing these words from the data can improve the accuracy of the language model because it removes inessential words.

While there are common stop words (e.g., “a,” “and,” “the,” etc.), any word can be considered a stop word if it does not contribute to the meaning of the sentence. Stop word removal is a common step in the NLP pipeline, especially for information retrieval, as stop words don’t distinguish between relevant and irrelevant content.

Let’s take a look at the stop word removal code. Start by creating a new notebook. Then install PySpark and start a new Spark session by entering the following code:

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"

# Start a SparkSession
import findspark
findspark.init()
```

```
 # Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("StopWords").getOrCreate()
```

Next, create a DataFrame that's already a list of words. By creating a list of words, we can skip the tokenization step for now. As you recall, tokenization takes input and separates it into a list of words. By creating a DataFrame that already contains a list of words, we are replicating this step:

```
# Create DataFrame
sentenceData = spark.createDataFrame([
    (0, ["Big", "data", "is", "super", "powerful"]),
    (1, ["This", "is", "going", "to", "be", "epic"])
], ["id", "raw"])

sentenceData.show(truncate=False)
```

```
+---+--------------------------------+
|id |raw                             |
+---+--------------------------------+
|0  |[Big, data, is, super, powerful]|
|1  |[This, is, going, to, be, epic] |
+---+--------------------------------+
```

Now import the StopWordsRemover library:

```
# Import stop words library
from pyspark.ml.feature import StopWordsRemover
```

Then, run the `StopWordsRemover()` function, which takes an input column that will be passed into the function, and an output column to add the results. This is stored in a variable for ease of use later:

```
# Run the Remover
remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
```

Now transform the DataFrame by applying `StopWordsRemover` and display the result:

```
# Transform and show data
remover.transform(sentenceData).show(truncate=False)
```

```
+---+-----------------------------+-----------------------------+
|id |raw                          |filtered                     |
+---+-----------------------------+-----------------------------+
| 0 |[Big, data, is, super, powerful]|[Big, data, super, powerful]|
| 1 |[This, is, going, to, be, epic]|[going, epic]               |
+---+-----------------------------+-----------------------------+
```

The outcome shows the raw data, or input, and the result of running the `StopWordsRemover()` function. In the filtered column, all the stop words are removed and the result is displayed.

SKILL DRILL

Combine both `tokenizer` and `StopWordsRemover` on a DataFrame that isn't already broken out into a list of words.

16.6.3 Term Frequency-Inverse Document Frequency Weight

Now that you have our text in a place that is ready for processing, you move on to how computers can determine the importance of certain words. You explain to Jennifer that the computer will do this through term frequency-inverse document frequency (TF-IDF). You assure her that the process may seem slightly complicated, but it's just a way for a computer to rank words.

With the text broken up and excess words eliminated, it's time to assess the value of the remaining text. The "value" of a word is determined by how often it appears in the text. For example, if you were reading an article about technology and the word "Python" appears multiple times, you would assume that Python is important to the subject of the article.

PySpark can apply **TF-IDF**, a statistical weight showing the importance of a word in a document. **Term frequency (TF)** measures the frequency of a word occurring in a document, and **inverse document frequency (IDF)** measures the significance of a word across a set of documents. Multiplying these two numbers would determine the TF-IDF.

Consider the previous example of the technology article. If "Python" appears 20 times in a 1,000-word article, the TF weight would be 20

divided by 1,000 which equals 0.02. Assume that this article resides in a database with other articles, totaling 1,000,000 articles.

IDF takes the number of documents that contain the word Python and compares to the total number of documents, then takes the logarithm of the results. For example, if Python is mentioned in 1000 articles and the total amount of articles in the database is 1,000,000, the IDF would be the log of the total number of articles divided by the number of articles that contain the word Python.

$$\text{IDF} = \log(\text{total articles} / \text{articles that contain the word Python})$$

$$\text{IDF} = \log(1{,}000{,}000 / 1000) = 3$$

Now that we have both numbers, we can determine the TF-IDF which is the multiple of the two.

$$\text{TF-IDF} = \text{TF} * \text{IDF} = 0.2 * 3 = .6$$

As you might recall, computers deal with numbers, not text, so for a computer to determine the TF-IDF, it needs to convert all the text to a numerical format. There are two possible ways to do so.

The first is by `CountVectorizer`, which indexes the words across all the documents and returns a vector of word counts corresponding to the indexes. The indexes are assigned in descending order of frequency. For example, the word with the highest frequency across all documents will be given an index of 0, and the word with the lowest frequency will have an index equal to the number of words in the text.

The second method is `HashingTF`, which converts words to numeric IDs. The same words are assigned the same IDs and then mapped to an index and counted, and a vector is returned. For our Python example, if it gets a numerical ID of 4278 and it appeared 20 times, the vector would be 4278 : 20.

For our project, let's use the `HashingTF` method for your pipeline for ease of use. It's easier to match up a hashed value for a word and the count of

times that word appeared.

Create a new notebook and install PySpark with the following code:

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"

# Start a SparkSession
import findspark
findspark.init()
```

Then, start a new Spark session:

```
# Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("TF-IDF").getOrCreate()
```

Import the following libraries that will be used, and go through both the tokenizer and stop words steps:

```
from pyspark.ml.feature import HashingTF, IDF, Tokenizer, StopWordsRemover
```

This time, we'll work with imported data. This is a dataset of tweets from people directed at an airline. We'll load in the data the same way we did when working with DataFrames earlier:

```
# Read in data from S3 Buckets
from pyspark import SparkFiles
url ="https://s3.amazonaws.com/dataviz-curriculum/day_2/airlines.csv"
spark.sparkContext.addFile(url)
df = spark.read.csv(SparkFiles.get("airlines.csv"), sep=",", header=True)

# Show DataFrame
df.show()
```

We'll tokenize our data first, as follows:

```
# Tokenize DataFrame
tokened = Tokenizer(inputCol="Airline Tweets", outputCol="words")
tokened_transformed = tokened.transform(df)
tokened_transformed.show()
```

Next, remove the stop words:

```
# Remove stop words
remover = StopWordsRemover(inputCol="words", outputCol="filtered")
removed_frame = remover.transform(tokened_transformed)
removed_frame.show(truncate=False)
```

The `HashingTF` function takes an argument for an input column, an output column, and a `numFeature` parameter, which specifies the number of buckets for the split words. This number must be higher than the number of unique words. By default, the value is 2^{18}or 262,144. The power of two should be used so that indexes are evenly mapped. Here we supply the `numFeatures` argument with it's default value for demonstration. This argument can normally be left out and will use the same value by default.

NOTE

> If the `numFeatures` is too low to give each word it's own individual index then we would get what are called collisions. This is beyond the scope of this class but if you wish to learn more about this please see this wikipedia article: **Collision (computer science) (https://en.wikipedia.org/wiki/Collision_(computer_science))** .

If you scroll to the right and down, you'll see the `hashedValues` result, which contains the index and term frequency of the corresponding index for each word.

```
# Run the hashing term frequency
hashing = HashingTF(inputCol="filtered", outputCol="hashedValues", numFeatures=pow(2,18))

# Transform into a DF
hashed_df = hashing.transform(removed_frame)
hashed_df.show(truncate=False)
```

```
-----------------------------------------------------+------------------------------------------------------------------------------------------------------------
                                                     |hashedValues
-----------------------------------------------------+------------------------------------------------------------------------------------------------------------
experience..., tacky.]                               |(262144,[99916,139943,141811,177269,203416,244945],[1.0,1.0,1.0,1.0,1.0,1.0])
t, seats, playing., really, bad, thing, flying, va]  |(262144,[14,56056,95547,96638,99549,107499,118368,152814,171117,227630,244945,249145],[1.0,1.0,1.0,1.
, soon.]                                             |(262144,[19871,34748,57178,147224,232735,244945],[1.0,1.0,1.0,1.0,1.0,1.0])
o, posted, online, current?]                         |(262144,[9103,9641,19136,63409,83569,188384,192171,244945],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])
all,, prefer, use, online, self-service, option, :(] |(262144,[19136,50528,67607,96822,116873,127390,186712,199981,219828,225517,244945],[1.0,1.0,1.0,1.0,1
-----------------------------------------------------+------------------------------------------------------------------------------------------------------------
```

With our words successfully converted to numbers, we can plug it all into an IDFModel, which will scale the values while down-weighting based on document frequency. If you scroll to the right on the output, you'll see the result of the TF-IDF:

```
# Fit the IDF on the data set
idf = IDF(inputCol="hashedValues", outputCol="features")
idfModel = idf.fit(hashed_df)
rescaledData = idfModel.transform(hashed_df)

# Display the DataFrame
rescaledData.select("words", "features").show(truncate=False)
```

```
----+------------------------------------------------------------------------------------------------------------------------------
    |features
----+------------------------------------------------------------------------------------------------------------------------------
    |(16,[1,3,5,7,8,12],[0.0,1.0986122886681098,1.0986122886681098,0.6931471805599453,0.1823215567939546,0.6931471805599453])
va] |(16,[0,1,8,9,11,13,14],[0.4054651081081644,0.0,0.1823215567939546,0.4054651081081644,1.3862943611198906,1.3862943611198906,2.772588722239781])
    |(16,[1,8,10,12,15],[0.0,0.1823215567939546,1.0986122886681098,0.6931471805599453,1.3862943611198906])
    |(16,[0,1,9,11,15],[0.8109302162163288,0.0,0.4054651081081644,0.6931471805599453,0.6931471805599453])
    |(16,[0,1,4,6,7,8,9,13,14],[0.8109302162163288,0.0,1.0986122886681098,1.0986122886681098,0.6931471805599453,0.1823215567939546,0.40546510810816
----+------------------------------------------------------------------------------------------------------------------------------
```

Remember that computers can't just read text and analyze it. With the process we have just completed, we have now given values from raw text that a computer can work with. The next section will put all of this

together and show how we can use text data to determine the accuracy of the corresponding rating.

16.6.4 Pipeline Setup to Run the Model

Great, you have all the steps of the pipeline down! Now it's time to put it all together and create a pipeline that will link everything and allow for the process to be reproduced.

Set Up the Pipeline

A pipeline enables us to store all of the functions we have created in different stages and run only once. Each stage that is passed in won't run until the previous stage has been completed, which means the output of one stage is passed on to the next one.

We'll put the pipeline to use by employing a sample Yelp dataset. This will allow you to practice with a smaller set of reviews using similar data that you'll be working on for your client.

Open up a new notebook and run the usual setup.

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
```

```
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"

# Start a SparkSession
import findspark
findspark.init()
```

Then start a new Spark session.

```
# Start Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Yelp_NLP").getOrCreate()
```

Then load in the following dataset into a DataFrame by entering the following code.

```
# Read in data from S3 Buckets
from pyspark import SparkFiles
url ="https://s3.amazonaws.com/dataviz-curriculum/day_2/yelp_reviews.csv"
spark.sparkContext.addFile(url)
df = spark.read.csv(SparkFiles.get("yelp_reviews.csv"), sep=",", header=True

# Show DataFrame
df.show()
```

Type the code to import all the functions that will be used in our NLP process, or pipeline, as follows:

```
# Import functions
from pyspark.ml.feature import Tokenizer, StopWordsRemover, HashingTF, IDF,
```

Next, create a new column that uses the `length` function to create a `future feature` with the length of each row. This is similar to the `tokenizer` phase when we created our own `udf` to do the same thing. A `udf` could still be used here, but PySpark makes it easier by supplying a ready-to-use function.

Type and run the following code.

```
from pyspark.sql.functions import length
# Create a length column to be used as a future feature
data_df = df.withColumn('length', length(df['text']))
data_df.show()
```

Now we'll create all the transformations to be applied in our pipeline. Note that the `StringIndexer` encodes a string column to a column of table indexes. Here we are working with positive and negative game reviews, which will be converted to 0 and 1. This will form our labels, which we'll delve into in the ML unit. The label is what we're trying to predict: will the review's given text let us know if it was positive or negative?

Also note that we don't need to run all of these completely as we did before. By creating all the functions now, we can then use them all in the pipeline later.

Type and run the following code.

```
# Create all the features to the data set
pos_neg_to_num = StringIndexer(inputCol='class',outputCol='label')
tokenizer = Tokenizer(inputCol="text", outputCol="token_text")
stopremove = StopWordsRemover(inputCol='token_text',outputCol='stop_tokens')
hashingTF = HashingTF(inputCol="token_text", outputCol='hash_token')
idf = IDF(inputCol='hash_token', outputCol='idf_token')
```

We'll create a feature vector containing the output from the IDFModel (the last stage in the pipeline) and the length. This will combine all the raw features to train the ML model that we'll be using. Enter the code for this as follows:

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vector
# Create feature vectors
clean_up = VectorAssembler(inputCols=['idf_token', 'length'], outputCol='fea
```

Now it's time to create our pipeline, the easiest step. We'll import the pipeline from `pyspark.ml`, and then store a list of the stages created earlier. It's important to list the stages in the order they need to be executed. As we mentioned before, the output from one stage will then be passed off to another stage.

Run the following code:

```
# Create and run a data processing Pipeline
from pyspark.ml import Pipeline
data_prep_pipeline = Pipeline(stages=[pos_neg_to_num, tokenizer, stopremove,
```

Nice work! All the stages of the pipeline have been set up. You may have noticed that each individual step didn't need to be set up, which is the perk of setting up the pipeline! Now your data is ready to be run through the pipeline. Then we can run it through a machine learning model.

16.6.5 Run the Model

You know that ML is needed to run NLP. However, ML is its own complex topic to master. As you and Jennifer are on a deadline, you decide to put your curiosity on hold for now and work with some basics to get your NLP to work. You plan to revisit ML once this project is done.

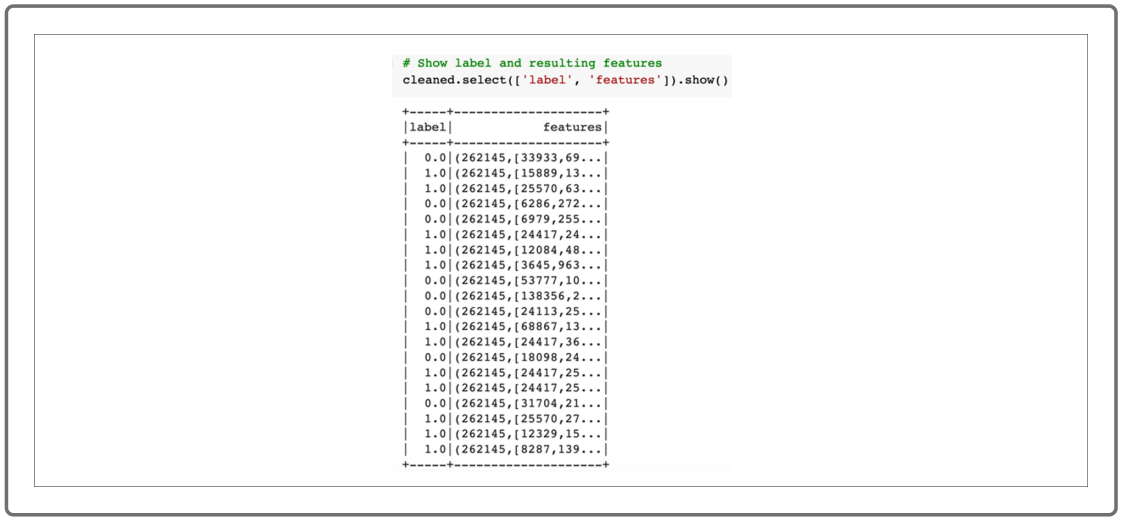
After our pipeline has been set up, we'll fit the outcome with our original DataFrame and transform it.

Type and run the following code:

```
# Fit and transform the pipeline
cleaner = data_prep_pipeline.fit(data_df)
cleaned = cleaner.transform(data_df)
```

As you can see in the following image, our labels and features that we created early on in the process are numerical representations of positive and negative reviews. The features will be used in our model and predict

whether a given review will be positive or negative. These features are the result of all the work we have been doing with the pipeline.



```
# Show label and resulting features
cleaned.select(['label', 'features']).show()
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(262145,[33933,69...|
|  1.0|(262145,[15889,13...|
|  1.0|(262145,[25570,63...|
|  0.0|(262145,[6286,272...|
|  0.0|(262145,[6979,255...|
|  1.0|(262145,[24417,24...|
|  1.0|(262145,[12084,48...|
|  1.0|(262145,[3645,963...|
|  0.0|(262145,[53777,10...|
|  0.0|(262145,[138356,2...|
|  0.0|(262145,[24113,25...|
|  1.0|(262145,[68867,13...|
|  1.0|(262145,[24417,36...|
|  0.0|(262145,[18098,24...|
|  1.0|(262145,[24417,25...|
|  1.0|(262145,[24417,25...|
|  0.0|(262145,[31704,21...|
|  1.0|(262145,[25570,27...|
|  1.0|(262145,[12329,15...|
|  1.0|(262145,[8287,139...|
+-----+--------------------+
```

IMPORTANT

> You may not have seen the terms “fit,” “predict,” and “transform” before in this context. These terms will make more sense when we learn about machine learning:

Now let’s run our ML model on the data. One of the basics of ML is that data gets broken into training data and testing data. **Training data** is the data that will be passed to our NLP model that will train our model to predict results. The **testing data** is used to test our predictions. We can do this with the `randomSplit` method, which takes in a list of the percent of data we want split into each group. Standard conventions use 70% with training and 30% with testing.

To split the data into a training set and a testing set, run the following code:

```
# Break data down into a training set and a testing set
training, testing = cleaned.randomSplit([0.7, 0.3], 21)
```

The array supplied to `randomSplit` is the percentage of the data that will be broken into training and testing respectively. So 70% to training and 30% to testing. The second number supplied is called a seed. The seed assures that this random split will also be the same, this is good for when we want to make sure we all have similar results.

The ML model we'll use is Naive Bayes, which we'll import and then fit the model using the training dataset. **Naive Bayes** is a group of classifier algorithms based on Bayes' theorem. Bayes theorem provides a way to determine the probability of an event based on new conditions or information that might be related to the event.

NOTE

> For a deeper dive into Naive Bayes, refer to this article on **Naive Bayes Classifiers (https://www.geeksforgeeks.org/naive-bayes-classifiers/)** and the **Naive Bayes Wikipedia page (https://en.wikipedia.org/wiki/Naive_Bayes_classifier)**.

Run the following code:

```
from pyspark.ml.classification import NaiveBayes
# Create a Naive Bayes model and fit training data
nb = NaiveBayes()
predictor = nb.fit(training)
```

Once the model has been trained, we'll transform the model with our testing data. Run the following code, and then look at the results:

```
# Tranform the model with the testing data
test_results = predictor.transform(testing)
test_results.show(5)
```

```
+--------+--------------------+------+-----+--------------------+--------------------+--------------------+--------------------+--------------------+
|   class|                text|length|label|          token_text|         stop_tokens|          hash_token|           idf_token|            features|
+--------+--------------------+------+-----+--------------------+--------------------+--------------------+--------------------+--------------------+
|negative|"The servers went...|    97|  1.0|["the, servers, w...|["the, servers, w...|(262144,[50940,67...|(262144,[50940,67...|(262145,[50940,67...|
|negative|"like the other r...|    82|  1.0|["like, the, othe...|["like, reviewer,...|(262144,[22808,61...|(262144,[22808,61...|(262145,[22808,61...|
|negative|"the food is not ...|    80|  1.0|["the, food, is, ...|["the, food, tast...|(262144,[15889,65...|(262144,[15889,65...|(262145,[15889,65...|
|negative|AN HOUR... seriou...|    21|  1.0|[an, hour..., ser...|[hour..., serious...|(262144,[64527,18...|(262144,[64527,18...|(262145,[64527,18...|
|negative|AVOID THIS ESTABL...|    25|  1.0|[avoid, this, est...|[avoid, establish...|(262144,[45245,10...|(262144,[45245,10...|(262145,[45245,10...|
+--------+--------------------+------+-----+--------------------+--------------------+--------------------+--------------------+--------------------+
only showing top 5 rows
```

At first glance, it may seem like the model isn't showing us anything useful. From your output, scroll all the way to the right to view the prediction column:

```
+--------------------+--------------------+--------------------+----------+
|            features|       rawPrediction|         probability|prediction|
+--------------------+--------------------+--------------------+----------+
|(262145,[50940,67...|[-1081.3695985467...|[6.80452613078075...|       1.0|
|(262145,[22808,61...|[-926.89178182297...|[1.64532098805577...|       1.0|
|(262145,[15889,65...|[-937.84361288546...|[8.55165743652179...|       1.0|
|(262145,[64527,18...|[-243.95752739931...|[0.35089668070171...|       1.0|
|(262145,[45245,10...|[-226.57284862787...|[1.12816468099250...|       1.0|
+--------------------+--------------------+--------------------+----------+
```

This prediction column will indicate with a 1.0 if the model thinks this review is negative and 0.0 if it thinks it's positive. Future data sets can now be run with this model and determine whether a review was positive or negative without having already supplied in the data.

How useful is this model? Should we just blindly trust that it will be right every time? There is one last step in the process to answer these questions.

The last step is to import the `MulticlassClassificationEvaluator`, which will display how accurate our model is in determining if a review will be positive or negative based solely on the text within a review. Run the following code:

```
# Use the Class Evaluator for a cleaner description
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

acc_eval = MulticlassClassificationEvaluator()
acc = acc_eval.evaluate(test_results)
print("Accuracy of model at predicting reviews was: %f" % acc)
```

```
Accuracy of model at predicting reviews was: 0.718984
```

The accuracy of the model isn't perfect, but it's not too low either: 0.741000. Machine learning isn't a guarantee, and tweaking our models as well as the data used is part of the process. One of the ways to do this is to add more data; when you keep adding data, eventually you grow from your local storage to something much larger—thus leading to big data!

Which of the following is the term used to describe words that are common and do not offer much value to the meaning of a text?

- Tokens
- Terms
- Vectors
- Stop words

Check Answer

Finish ▸

16.7.1 Evaluate Amazon Web Services

Cloud computing is becoming the future for almost every aspect of business, but it's especially important with data. Your company works with Amazon Web Services (AWS) and has suggested that you use that for your client's requested database. AWS is a big platform. Where do you even begin? You decide to start with an introduction to AWS and cloud services.

"In the cloud" is an oft-repeated phrase these days, and nearly every technology used promises cloud access. What does it mean? It's not technology "floating in the clouds," but rather computing hosted on a shared virtual environment, interconnected to a massive storage facility with contained servers, storage, and different sites.

Amazon Web Services (AWS) is the largest cloud provider in the market today. It's not the only one—many companies, such as Google and Microsoft, also have their own cloud platforms. We'll focus on AWS, but the services offered will be similar across all solutions.

Cloud services, such as AWS, allow a company to scale easily. Before cloud services, you would need to buy and store your own servers, which is costly, time-consuming, and requires physical space. A few

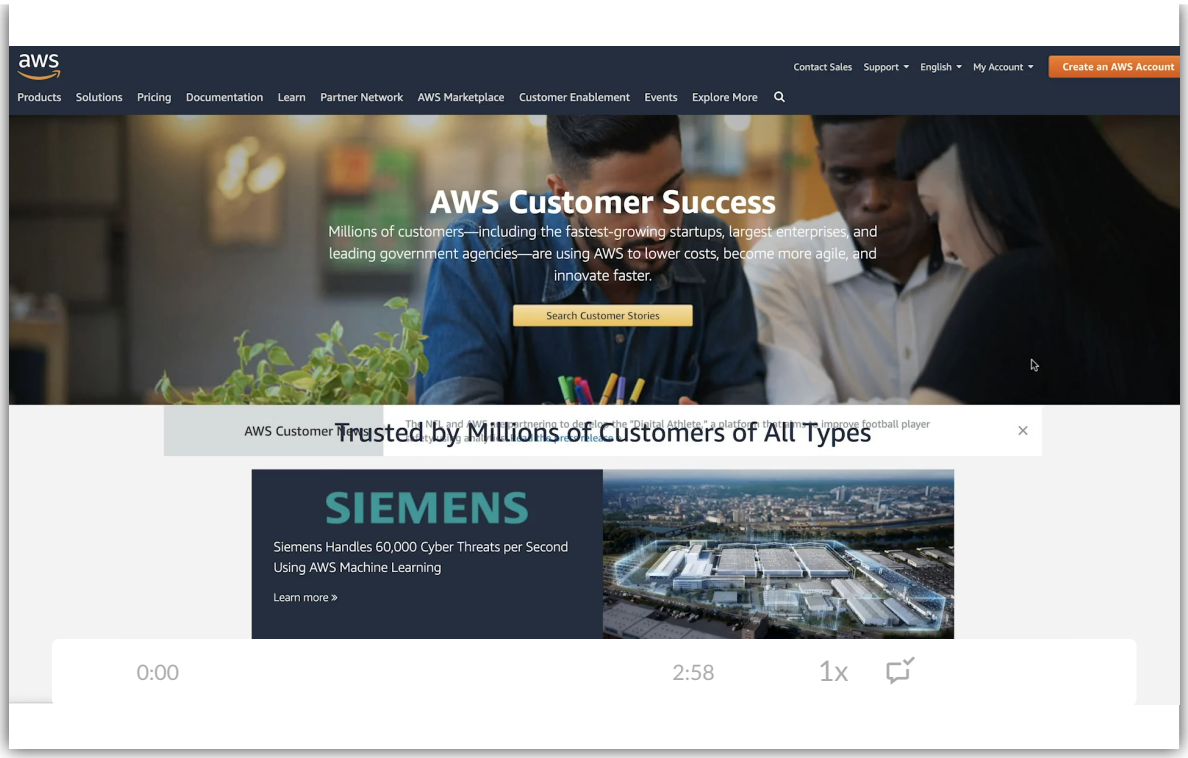
considerations: How much to buy? What if you miscalculated your growth rate and spent too much or too little on servers? Cloud services allow the flexibility to adjust on the fly for budget and resources.

Before we begin, understand we'll be using AWS's set of free tier options. This way, we can learn and practice on the platform without incurring costs. AWS and other cloud services charge sneaky costs that quickly add up, so it's important to always follow the directions and use the free tier services. At the end of this module, we'll perform a full cleanup of all the services we used to ensure nothing is left in the cloud.

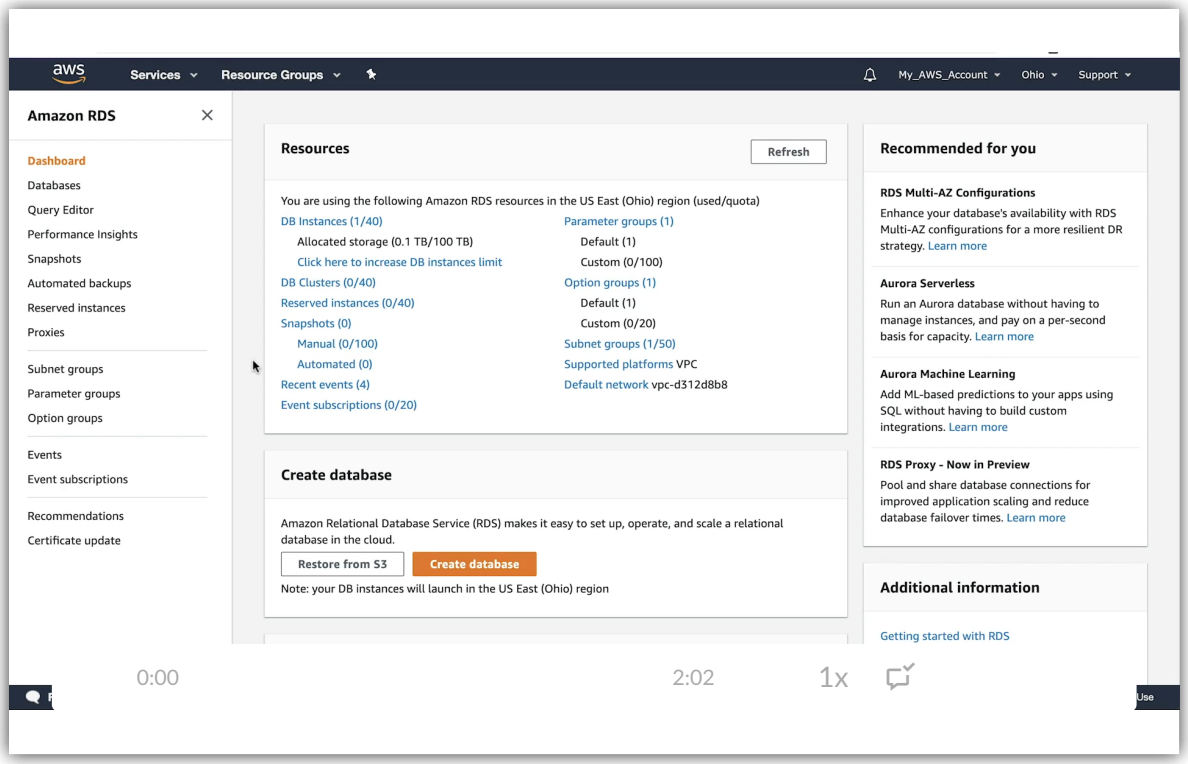
16.7.2 Create an AWS Relational Database

Now that you have a good idea of what AWS does, it's time to use it for hosting the database for your client. You decide to go with a Postgres database for the many benefits and host it on AWS.

AWS offers a wide variety of storage options on its platform, including both structured and unstructured databases. We'll go through setting up a Postgres database using AWS's relational database service (RDS), but note that there are many databases to choose from.



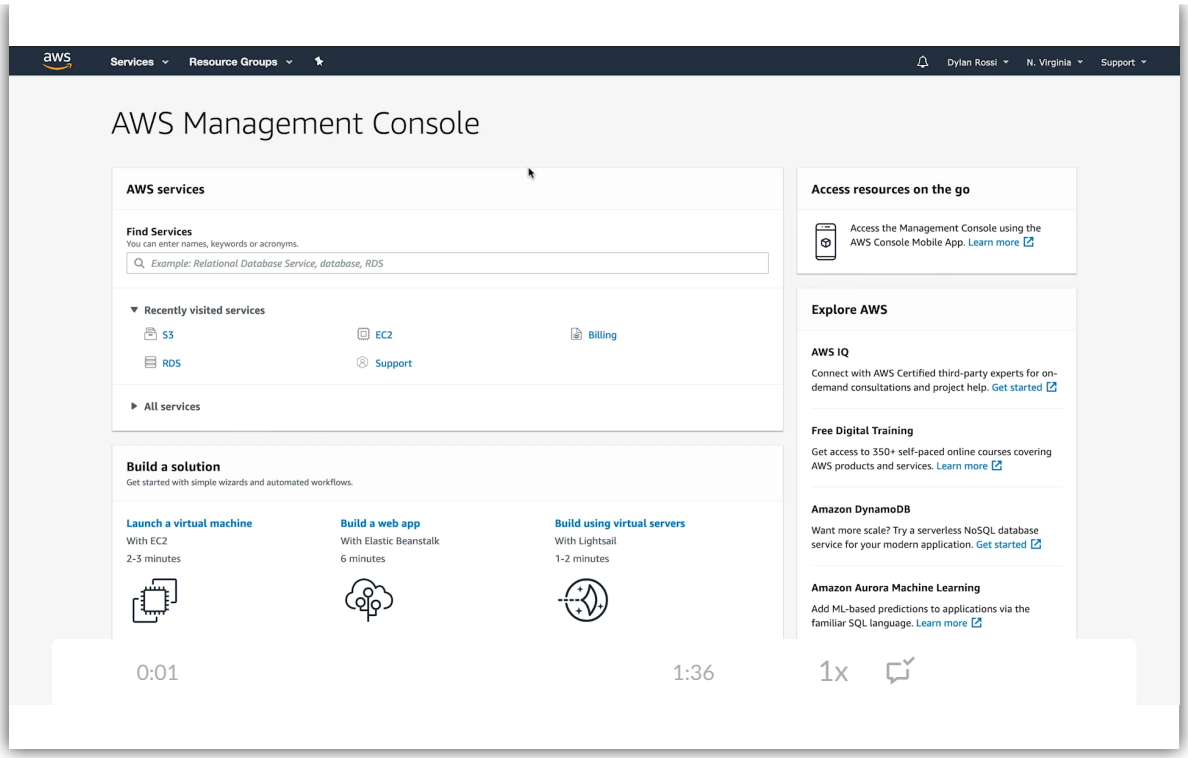
After setting up your RDS there are some quick adjustments to the settings to make sure your database is available everywhere.



16.7.3 Connect pgAdmin to Your RDS Instance

Now that you are all set up on AWS, it's time to actually connect. Luckily, you already have experience using pgAdmin for your SQL queries, so you can simply connect pgAdmin to the RDS instance you just created.

The Postgres database you just created is hosted on the cloud, so it can be accessed by anyone with credentials using whatever platform you prefer. Since we have experience with pgAdmin, we'll connect that.



16.7.4 Test with Create, Read, Update, and Delete

With your database fully set up and accessible, you decide that in order to test everything, it would also be a good time to cover Create, Read, Update, and Delete (CRUD) with Jennifer. This is an important part of database management.

Before we test our connection with our RDS instance and pgAdmin, let's learn about persistent data storage. **Persistent data storage** is where data is saved even when a machine's power is off (i.e., your computer's hard drive). SQL databases, such as the one we just connected to with AWS, is persistent storage.

The four basic functions of persistent data storage are **Create, Read, Update,** and **Delete (CRUD).** Let's test our connection to the RDS instance by performing some simple CRUD operations. This is review material, but it's important to understand which queries and operations belong to which part of CRUD.

Before we begin, from pgAdmin, create a new database within our RDS instances called "medical."

Create

The first function of CRUD is creating data. We'll make tables and insert data into them. Inserting is the main part of creating data to be stored within the database. Let's run the following query.

```
CREATE TABLE doctors (
 id INT PRIMARY KEY NOT NULL,
 speciality TEXT,
 taking_patients BOOLEAN
);
CREATE TABLE patients (
 id INT NOT NULL,
 doctor_id INT NOT NULL,
 health_status TEXT,
 PRIMARY KEY (id, doctor_id),
 FOREIGN KEY (doctor_id) REFERENCES doctors (id)
);

INSERT INTO doctors(id, speciality, taking_patients)
VALUES
(1, 'cardiology', TRUE),
(2, 'orthopedics', FALSE),
(3, 'pediatrics', TRUE);
INSERT INTO patients (id, doctor_id, health_status)
VALUES
(1, 2, 'healthy'),
(2, 3, 'sick'),
(3, 2, 'sick'),
(4, 1, 'healthy'),
(5, 1, 'sick');
```

Great! We have just created some data. Although you have seen this before, it's important to understand where these queries fall in CRUD whenever you see it referenced.

Read

The second function is reading our data. We'll run our `SELECT` statements for the data we want to retrieve from our tables. Let's run the following query to confirm our data has been successfully inserted.

```
-- Read tables
SELECT * FROM doctors;
SELECT * FROM patients;
```

Update

The third function is updating data that is currently stored. We'll run the following query to update our data.

```
-- Update rows
UPDATE doctors
SET taking_patients = FALSE
WHERE id = 1;
UPDATE patients
SET health_status = 'healthy'
WHERE id = 1;
```

To confirm our update, run a `SELECT` statement, which as you recall is the read function of CRUD.

Delete

The final function is deleting data. We'll run the following query to delete data.

```
-- Delete row
DELETE FROM patients
WHERE id = 1;
```

You're not likely to use the `DELETE` statement because we're in the business of collecting data, not removing it. You should always be very careful when deleting data—often a system or rules are in place to secure your database and make data removal difficult to prevent mistakes, which can happen.

You now have an RDS instance running and tested by the CRUD functions within persistent data storage. It's time to learn how raw data is stored.

16.8.1 Database Versus Data Storage

Your client has all the data already, but they do not have it uploaded to a database. They maintain all their raw data in AWS's S3. You explain to Jennifer that data storage is a great place for data collection, because it allows many types and formats of data without having to slow down the intake for processing.

Data storage holds raw data such as CSVs, Excel files, and JavaScript Object Notation (JSON) files. Think of your own computer file system where you keep a ton of files as data storage. This data doesn't need to be queried and analyzed for business decisions. The files still have structure and can be reviewed, but not nearly as efficiently as a database.

A database contains cleaned, related information in tabular form. This database has been carefully planned and structured so that data can be analyzed efficiently through queries. Doing so comes at a cost of processing data to fit all the rules and structures.

Data storage is a place where large amounts of raw data can be kept without any munging or curating. Data storage allows us to keep data of different types or data we might want to parse in the future.

The benefit of having dedicated data storage is that nothing limits the intake of data. Data can flow in constantly and be saved without having to worry if it fits the criteria of the database. We have seen this with our extract, transform, and load (ETL) process—the data storage can hold raw files, such as CSV or JSON, for different needs.

AWS's S3 is a popular data storage service that we'll cover next.

16.8.2 AWS's Simple Storage Service

When you use a cloud provider such as AWS, all of your data needs can be taken care of in one spot. You won't need to worry that one person has all of the CSVs on one machine while someone else has a different set of data on another machine. You decide that your best bet is to move all of your client's incoming data into S3 buckets on AWS.

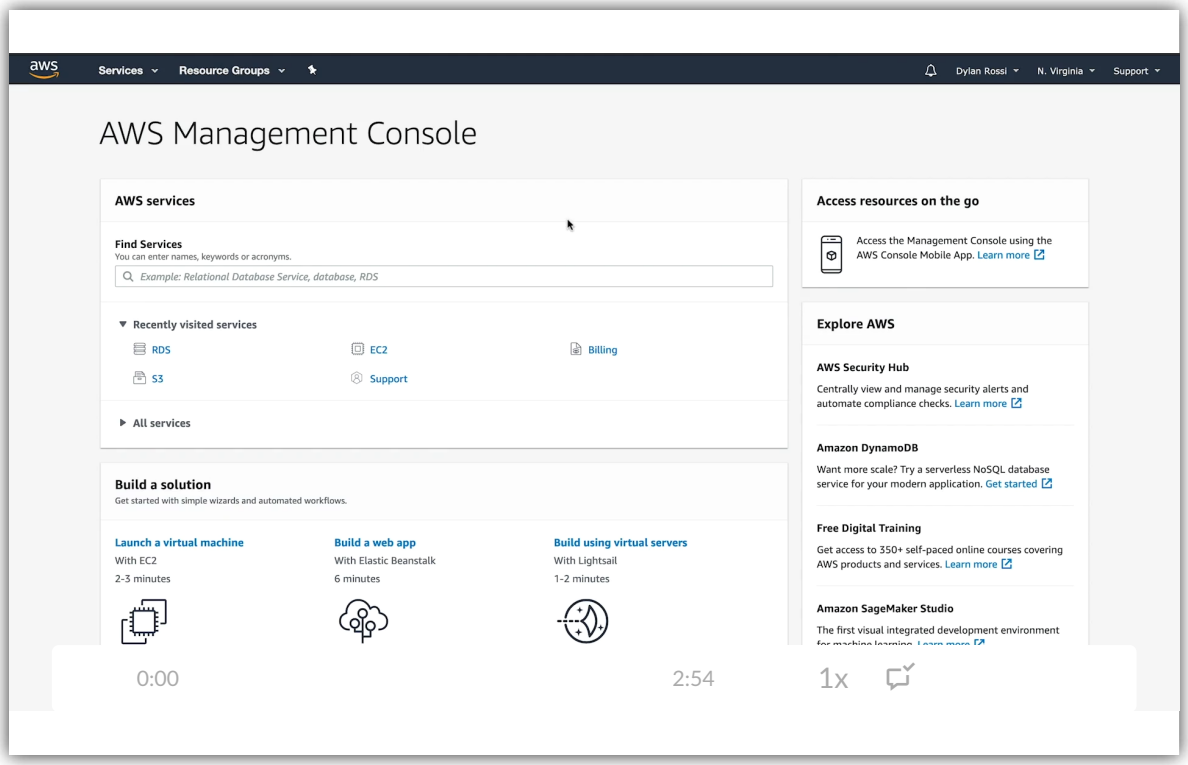
S3 is Amazon's cloud file storage service that uses key-value pairs. Files are stored on multiple servers and have a high rate of availability of more than 99.9%. To store files, S3 uses **buckets**, which are similar to folders or directories on your computer. Buckets can contain additional folders and files. Each bucket must have a unique name across all of AWS.

One of S3's perks is its fine-grained control over files. Each file or bucket can have different read and write permissions, which helps regulate what can be done with each file.

S3 is also very scalable—you are not limited to the memory of one computer. As data flows in, more and more can be stored, as opposed to a local computer that is limited by available memory. Additionally, it offers

availability—several team members can access massive amounts of data from one central location.

Follow along with the video to create your S3 buckets for data storage:



16.8.3 PySpark and S3 Stored Data

Jennifer now understands the difference between databases and data storage and is really enjoying using S3 for storage. She's now wondering how to load in your data from storage. You remind her where we began with big data and explain that PySpark is the perfect tool for loading in stored data.

Since PySpark is a big data tool, it has many ways of reading in files from data storage so that we can manipulate them. We have decided to use S3 as our data storage, so we'll use PySpark for all data processing.

Using PySpark is how we've been reading in our data into Google Colab so far. The format for reading in from S3 is the S3 link, followed by your bucket name, folder by each folder, and then the filename, as follows:

For US East (default region)

```
template_url = "https://<bucket-name.s3.amazonaws.com/<folder-name>/<file-na

example_url = "https://dataviz-curriculum.s3.amazonaws.com/data-folder/data.
```

For other regions

```
template_url = "https://<bucket-name.s3-<region>.amazonaws.com/<folder-name>

example_url =" https://dataviz-curriculum.s3-us-west-1.amazonaws.com/data-fo
```

SKILL DRILL

Load in a dataset to one of your S3 buckets, and then, using Google Colab, load in your dataset to a DataFrame.

16.9.1 PySpark ETL

You and Jennifer are excited about all the new technologies you're ready to use. You explain to Jennifer that you can now move your ETL process to the cloud to take the load off your local laptops, use strictly cloud resources, and eventually lead to automation. Let's get started with extraction!

Let's run through a mock scenario using two different types of raw data stored in S3. Our goal is to get this raw data from S3 into an RDS database. Let's start with uploading files to your own S3 bucket:

Create an S3 bucket, and then load the following files into the bucket:

- **user_data.csv (https://courses.bootcampspot.com/courses/138/files/19405/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/19405/download?wrap=1)**
- **user_payment.csv (https://courses.bootcampspot.com/courses/138/files/19409/download?wrap=1)** 

(https://courses.bootcampspot.com/courses/138/files/19409/download?wrap=1)

IMPORTANT

> Remember to make the bucket and files public.

Assume your company already has three tables set up in the RDS database and would like to get the raw data from S3 into the database. Create a new database in pgAdmin called “my_data_class_db.” We'll have it represent the company database by first running the following schema in pgAdmin for our RDS:

```
-- Create Active User Table
CREATE TABLE active_user (
 id INT PRIMARY KEY NOT NULL,
 first_name TEXT,
 last_name TEXT,
 username TEXT
);

CREATE TABLE billing_info (
 billing_id INT PRIMARY KEY NOT NULL,
 street_address TEXT,
 state TEXT,
 username TEXT
);

CREATE TABLE payment_info (
 billing_id INT PRIMARY KEY NOT NULL,
 cc_encrypted TEXT
);
```

NOTE

> Table creation is not part of the ETL process. We’re creating the tables to represent a pre-established database you need for the raw data. In a

> real-life situation, databases will already have a well-defined schema and tables for you, as the engineer, to process data into.

Start with creating a new notebook, installing Spark:

```
# Install Java, Spark, and Findspark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-had
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark

# Set Environment Variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.5-bin-hadoop2.7"

# Start a SparkSession
import findspark
findspark.init()
```

We'll use Spark to write directly to our Postgres database. But in order to do so, there are few more lines of code we need.

First, enter the following code to download a Postgres driver that will allow Spark to interact with Postgres:

```
!wget https://jdbc.postgresql.org/download/postgresql-42.2.9.jar
```

Then, start a Spark session with an additional option that adds the driver to Spark:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("CloudETL").config("spark.driver.extraC
```

We have performed the first two steps of the ETL process before with PySpark, so let's quickly review those.

Extract

We can connect to data storage, then extract that data into a DataFrame. We'll do this on two datasets, and be sure to replace the bucket name with one of your own.

We'll start by importing SparkFiles from PySpark into our notebook. This will allow Spark to add a file to our Spark project.

Next, the file is read in with the `read` method and combined with the `csv()` method, which pulls in our CSV stored in SparkFiles and infers the schema. SparkFiles.get() will have Spark retrieve the specified file, since we are dealing with a CSV. The "," is the chosen separator, and we will have Spark determine the head for us. Enter the following code:

```
# Read in data from S3 Buckets
from pyspark import SparkFiles
url ="https://YOUR-BUCKET-NAME.s3.amazonaws.com/user_data.csv"
spark.sparkContext.addFile(url)
user_data_df = spark.read.csv(SparkFiles.get("user_data.csv"), sep=",", head

# Show DataFrame
user_data_df.show()
```

Finally, an action is called to show the first 10 runs and confirm our data extraction by entering the following code:

```
# Show DataFrame
user_data_df.show()
```

Repeat a similar process to load in the other data. Enter the code:

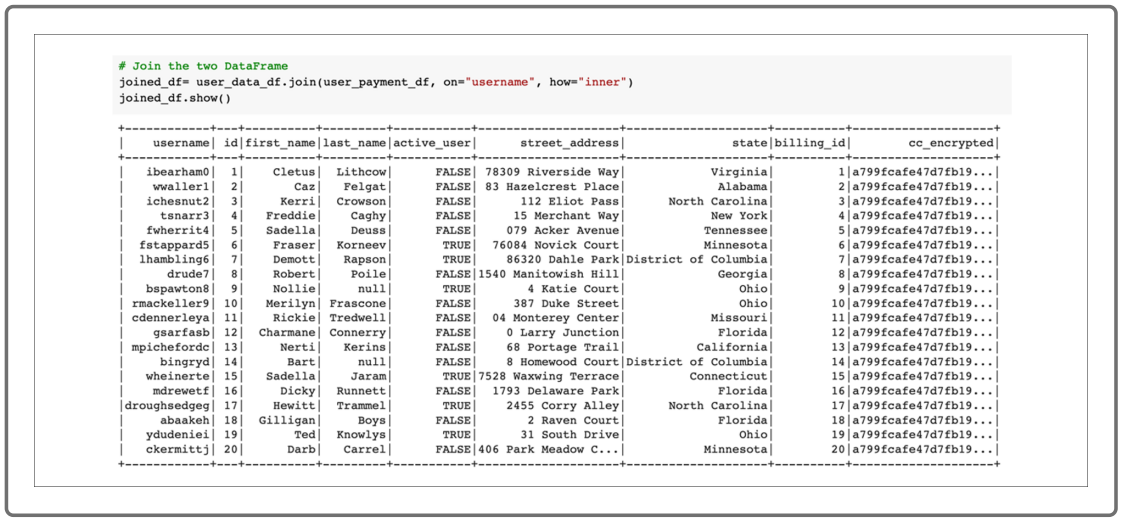
```
url ="https://YOUR-BUCKET-NAME.s3.amazonaws.com/user_payment.csv"
spark.sparkContext.addFile(url)
user_payment_df = spark.read.csv(SparkFiles.get("user_payment.csv"), sep=","

# Show DataFrame
user_payment_df.show()
```

Transform

Now that the raw data stored in S3 is available in a PySpark DataFrame, we can perform our transformations.

First, join the two tables:



```
# Join the two DataFrame
joined_df= user_data_df.join(user_payment_df, on="username", how="inner")
joined_df.show()
```

username	id	first_name	last_name	active_user	street_address	state	billing_id	cc_encrypted
ibearham0	1	Cletus	Lithcow	FALSE	78309 Riverside Way	Virginia	1	a799fcafe47d7fb19...
wwaller1	2	Caz	Felgat	FALSE	83 Hazelcrest Place	Alabama	2	a799fcafe47d7fb19...
ichesnut2	3	Kerri	Crowson	FALSE	112 Eliot Pass	North Carolina	3	a799fcafe47d7fb19...
tsnarr3	4	Freddie	Caghy	FALSE	15 Merchant Way	New York	4	a799fcafe47d7fb19...
fwherrit4	5	Sadella	Deuss	FALSE	079 Acker Avenue	Tennessee	5	a799fcafe47d7fb19...
fstappard5	6	Fraser	Korneev	TRUE	76084 Novick Court	Minnesota	6	a799fcafe47d7fb19...
lhambling6	7	Demott	Rapson	TRUE	86320 Dahle Park	District of Columbia	7	a799fcafe47d7fb19...
drude7	8	Robert	Poile	FALSE	1540 Manitowish Hill	Georgia	8	a799fcafe47d7fb19...
bspawton8	9	Nollie	null	TRUE	4 Katie Court	Ohio	9	a799fcafe47d7fb19...
rmackeller9	10	Merilyn	Frascone	FALSE	387 Duke Street	Ohio	10	a799fcafe47d7fb19...
cdennerleya	11	Rickie	Tredwell	FALSE	04 Monterey Center	Missouri	11	a799fcafe47d7fb19...
gsarfasb	12	Charmane	Connerry	FALSE	0 Larry Junction	Florida	12	a799fcafe47d7fb19...
mpicheford	13	Nerti	Kerins	FALSE	68 Portage Trail	California	13	a799fcafe47d7fb19...
bingryd	14	Bart	null	FALSE	8 Homewood Court	District of Columbia	14	a799fcafe47d7fb19...
wheinerte	15	Sadella	Jaram	TRUE	7528 Waxwing Terrace	Connecticut	15	a799fcafe47d7fb19...
mdrewetf	16	Dicky	Runnett	FALSE	1793 Delaware Park	Florida	16	a799fcafe47d7fb19...
droughsedgeg	17	Hewitt	Trammel	TRUE	2455 Corry Alley	North Carolina	17	a799fcafe47d7fb19...
abaakeh	18	Gilligan	Boys	FALSE	2 Raven Court	Florida	18	a799fcafe47d7fb19...
ydudeniei	19	Ted	Knowlys	TRUE	31 South Drive	Ohio	19	a799fcafe47d7fb19...
ckermittj	20	Darb	Carrel	FALSE	406 Park Meadow C...	Minnesota	20	a799fcafe47d7fb19...

Next, drop any rows with null or “not a number” (NaN) values:

```
# Drop null values
dropna_df = joined_df.dropna()
dropna_df.show()
```

```
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
|    username| id|first_name|last_name|active_user|      street_address|               state|billing_id|        cc_encrypted|
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
|   ibearham0|  1|     Cletus|  Lithcow|      FALSE| 78309 Riverside Way|            Virginia|         1|a799fcafe47d7fb19...|
|    wwaller1|  2|       Caz|   Felgat|      FALSE| 83 Hazelcrest Place|             Alabama|         2|a799fcafe47d7fb19...|
|   ichesnut2|  3|     Kerri|  Crowson|      FALSE|      112 Eliot Pass|      North Carolina|         3|a799fcafe47d7fb19...|
|     tsnarr3|  4|   Freddie|    Caghy|      FALSE|     15 Merchant Way|            New York|         4|a799fcafe47d7fb19...|
|   fwherrit4|  5|   Sadella|    Deuss|      FALSE|    079 Acker Avenue|           Tennessee|         5|a799fcafe47d7fb19...|
|  fstappard5|  6|    Fraser|  Korneev|       TRUE|  76084 Novick Court|           Minnesota|         6|a799fcafe47d7fb19...|
|  lhambling6|  7|    Demott|   Rapson|       TRUE|    86320 Dahle Park|District of Columbia|         7|a799fcafe47d7fb19...|
|      drude7|  8|    Robert|    Poile|      FALSE|1540 Manitowish Hill|             Georgia|         8|a799fcafe47d7fb19...|
| rmackeller9| 10|   Merilyn| Frascone|      FALSE|     387 Duke Street|                Ohio|        10|a799fcafe47d7fb19...|
| cdennerleya| 11|    Rickie| Tredwell|      FALSE|  04 Monterey Center|            Missouri|        11|a799fcafe47d7fb19...|
|    gsarfasb| 12|  Charmane| Connerry|      FALSE|    0 Larry Junction|             Florida|        12|a799fcafe47d7fb19...|
| mpichefordc| 13|     Nerti|   Kerins|      FALSE|    68 Portage Trail|          California|        13|a799fcafe47d7fb19...|
|   wheinerte| 15|   Sadella|    Jaram|       TRUE|7528 Waxwing Terrace|         Connecticut|        15|a799fcafe47d7fb19...|
|   mdrewetf| 16|     Dicky|  Runnett|      FALSE|  1793 Delaware Park|             Florida|        16|a799fcafe47d7fb19...|
|droughsedgeg| 17|    Hewitt|  Trammel|       TRUE|    2455 Corry Alley|      North Carolina|        17|a799fcafe47d7fb19...|
|     abaakeh| 18|  Gilligan|     Boys|      FALSE|       2 Raven Court|             Florida|        18|a799fcafe47d7fb19...|
|  ydudeniei| 19|       Ted|  Knowlys|       TRUE|      31 South Drive|                Ohio|        19|a799fcafe47d7fb19...|
|  ckermittj| 20|      Darb|   Carrel|      FALSE|406 Park Meadow C...|           Minnesota|        20|a799fcafe47d7fb19...|
|    ipowisk| 21|   Diandra|Cancellor|      FALSE|      1 Fisk Parkway|      North Carolina|        21|a799fcafe47d7fb19...|
|   dtaltonl| 22|    Ulrika|  Itzhayek|      FALSE|  890 Lakewood Alley|          California|        22|a799fcafe47d7fb19...|
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
```

Filter for active users:

```
# Load in a sql function to use columns
from pyspark.sql.functions import col

# Filter for only columns with active users
cleaned_df = dropna_df.filter(col("active_user")  == True)
cleaned_df.show()
```

```
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
|    username| id|first_name|last_name|active_user|      street_address|               state|billing_id|        cc_encrypted|
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
|  fstappard5|  6|    Fraser|  Korneev|       TRUE|  76084 Novick Court|           Minnesota|         6|a799fcafe47d7fb19...|
|  lhambling6|  7|    Demott|   Rapson|       TRUE|    86320 Dahle Park|District of Columbia|         7|a799fcafe47d7fb19...|
|   wheinerte| 15|   Sadella|    Jaram|       TRUE|7528 Waxwing Terrace|         Connecticut|        15|a799fcafe47d7fb19...|
| droughsedgeg| 17|    Hewitt|  Trammel|       TRUE|    2455 Corry Alley|      North Carolina|        17|a799fcafe47d7fb19...|
|   ydudeniei| 19|       Ted|  Knowlys|       TRUE|      31 South Drive|                Ohio|        19|a799fcafe47d7fb19...|
|    fmyttonm| 23|  Annmarie|   Lafond|       TRUE|     35 Oriole Place|             Georgia|        23|a799fcafe47d7fb19...|
|  bfletcherr| 28|      Toma|   Sokell|       TRUE|39641 Eggendart Hill|            Maryland|        28|a799fcafe47d7fb19...|
|     gturleyt| 30|       Ram|  Lefever|       TRUE|   9969 Laurel Alley|               Texas|        30|a799fcafe47d7fb19...|
|   calyukinu| 31|    Raddie|  Heindle|       TRUE|   811 Talmadge Road|                Ohio|        31|a799fcafe47d7fb19...|
| ckleinlererw| 33|    Wallie|     Caws|       TRUE|   9999 Kenwood Pass|              Oregon|        33|a799fcafe47d7fb19...|
|  pshanklandx| 34|    Derril|Varfolomeev|       TRUE|     4 Jenifer Court|             Florida|        34|a799fcafe47d7fb19...|
|   enelanel2| 39|     Kelcy|   Wheway|       TRUE|93207 Morningstar...|             Florida|        39|a799fcafe47d7fb19...|
|   sfollet13| 40|    Dorree|  Rookeby|       TRUE|       2 Troy Circle|          California|        40|a799fcafe47d7fb19...|
|     mteshl4| 41|    Martyn|     Tott|       TRUE|      728 Muir Lane|             Florida|        41|a799fcafe47d7fb19...|
|  tseyfart16| 43|     Cally|   Thody|       TRUE|  1 Graceland Plaza|             Florida|        43|a799fcafe47d7fb19...|
|  hfarrier18| 45|       Ted| Pittaway|       TRUE|767 Little Fleur ...|      North Carolina|        45|a799fcafe47d7fb19...|
|    nabbie1b| 48|      Fifi|  Lidgley|       TRUE|6744 Sutherland Road|      South Carolina|        48|a799fcafe47d7fb19...|
| ystadding1d| 50|    Ashely|  O'Hern|       TRUE|   929 Scoville Park|             Florida|        50|a799fcafe47d7fb19...|
|hhallgalley1g| 53|  Diannne|Osbaldeston|       TRUE|       0 Mesta Pass|           Tennessee|        53|a799fcafe47d7fb19...|
|   ageavenyln| 60|     Sonny|   Jeskin|       TRUE| 50 Sutherland Drive|       Massachusetts|        60|a799fcafe47d7fb19...|
+------------+---+----------+---------+-----------+--------------------+--------------------+----------+--------------------+
```

Next, select columns to create three different DataFrames that match what is in the AWS RDS database. Create a DataFrame to match the active_user table:

```
# Create user dataframe to match active_user table
clean_user_df = cleaned_df.select(["id", "first_name", "last_name", "username"])
clean_user_df.show()
```

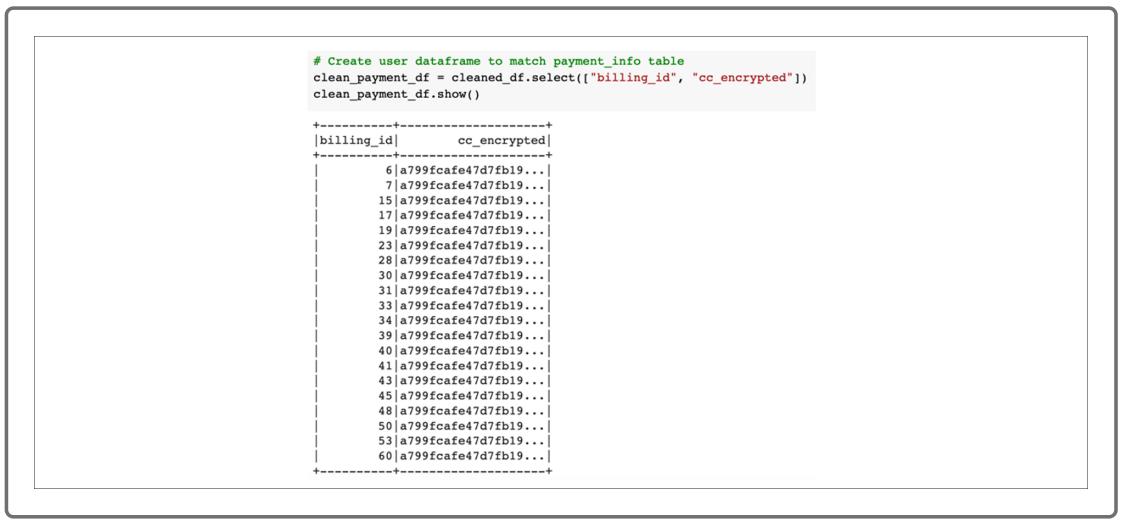
```
+---+----------+-----------+-------------+
| id|first_name|  last_name|     username|
+---+----------+-----------+-------------+
|  6|    Fraser|    Korneev|   fstappard5|
|  7|    Demott|     Rapson|   lhambling6|
| 15|   Sadella|      Jaram|    wheinerte|
| 17|    Hewitt|    Trammel| droughsedgeg|
| 19|       Ted|    Knowlys|    ydudeniei|
| 23| Annmarie|     Lafond|     fmyttonm|
| 28|      Toma|     Sokell|   bfletcherr|
| 30|       Ram|    Lefever|     gturleyt|
| 31|    Raddie|    Heindle|    calyukinu|
| 33|    Wallie|       Caws| ckleinlererw|
| 34|    Derril|Varfolomeev|  pshanklandx|
| 39|     Kelcy|     Wheway|    enelane12|
| 40|    Dorree|    Rookeby|    sfollet13|
| 41|    Martyn|       Tott|     mteshl4|
| 43|     Cally|      Thody|   tseyfart16|
| 45|       Ted|   Pittaway|   hfarrier18|
| 48|      Fifi|    Lidgley|     nabbielb|
| 50|    Ashely|     O'Hern|  ystaddingld|
| 53|   Diannne|Osbaldeston|hhallgalleylg|
| 60|     Sonny|     Jeskin|   ageavenyln|
+---+----------+-----------+-------------+
```

Next, create a DataFrame to match the billing_info table:

```
# Create user dataframe to match billing_info table
clean_billing_df = cleaned_df.select(["billing_id", "street_address", "state", "username"])
clean_billing_df.show()
```

```
+----------+--------------------+--------------------+-------------+
|billing_id|      street_address|               state|     username|
+----------+--------------------+--------------------+-------------+
|         6|  76084 Novick Court|           Minnesota|   fstappard5|
|         7|    86320 Dahle Park|District of Columbia|   lhambling6|
|        15|7528 Waxwing Terrace|         Connecticut|    wheinerte|
|        17|    2455 Corry Alley|      North Carolina| droughsedgeg|
|        19|      31 South Drive|                Ohio|    ydudeniei|
|        23|     35 Oriole Place|             Georgia|     fmyttonm|
|        28|39641 Eggendart Hill|            Maryland|   bfletcherr|
|        30|   9969 Laurel Alley|               Texas|     gturleyt|
|        31|   811 Talmadge Road|                Ohio|    calyukinu|
|        33|   9999 Kenwood Pass|              Oregon| ckleinlererw|
|        34|     4 Jenifer Court|             Florida|  pshanklandx|
|        39|93207 Morningstar...|             Florida|    enelane12|
|        40|       2 Troy Circle|          California|    sfollet13|
|        41|       728 Muir Lane|             Florida|     mteshl4|
|        43|   1 Graceland Plaza|             Florida|   tseyfart16|
|        45|767 Little Fleur ...|      North Carolina|   hfarrier18|
|        48|6744 Sutherland Road|      South Carolina|     nabbielb|
|        50|   929 Scoville Park|             Florida|  ystaddingld|
|        53|        0 Mesta Pass|           Tennessee|hhallgalleylg|
|        60| 50 Sutherland Drive|       Massachusetts|   ageavenyln|
+----------+--------------------+--------------------+-------------+
```

Finally, create a DataFrame to match the payment_info table:



```
# Create user dataframe to match payment_info table
clean_payment_df = cleaned_df.select(["billing_id", "cc_encrypted"])
clean_payment_df.show()
```

```
+----------+--------------------+
|billing_id|        cc_encrypted|
+----------+--------------------+
|         6|a799fcafe47d7fb19...|
|         7|a799fcafe47d7fb19...|
|        15|a799fcafe47d7fb19...|
|        17|a799fcafe47d7fb19...|
|        19|a799fcafe47d7fb19...|
|        23|a799fcafe47d7fb19...|
|        28|a799fcafe47d7fb19...|
|        30|a799fcafe47d7fb19...|
|        31|a799fcafe47d7fb19...|
|        33|a799fcafe47d7fb19...|
|        34|a799fcafe47d7fb19...|
|        39|a799fcafe47d7fb19...|
|        40|a799fcafe47d7fb19...|
|        41|a799fcafe47d7fb19...|
|        43|a799fcafe47d7fb19...|
|        45|a799fcafe47d7fb19...|
|        48|a799fcafe47d7fb19...|
|        50|a799fcafe47d7fb19...|
|        53|a799fcafe47d7fb19...|
|        60|a799fcafe47d7fb19...|
+----------+--------------------+
```

Once our data has been transformed to fit the tables in our database, we're ready to move on to the "Load" step.

Load

The final step is to get our transformed raw data into our database. PySpark can easily connect to a database to load the DataFrames into the table. First, we'll do some configuration to allow the connection with the following code:

```
# Configure settings for RDS
mode = "append"
jdbc_url="jdbc:postgresql://<endpoint>:5432/<database name>"
config = {"user":"root",
          "password": "<password>",
          "driver":"org.postgresql.Driver"}
```

Let's break down what's happening here:

- `Mode` is what we want to do with the DataFrame to the table, such as `overwrite` or `append`. We'll append to the current table because every time we run this ETL process, we'll want more data added to our database without removing any.

- The `jdbc_url` is the connection string to our database.
- A dictionary of configuration that includes the `user`, `password`, and `driver` to what type of database is being used.

The cleaned DataFrames can then be written directly to our database by using the `.write.jdbc` method that takes in the parameters we set:

- The connection string stored in `jdbc_url` is passed to the URL argument.
- The corresponding name of the table we are writing the DataFrame to.
- The mode we're using, which is "append."
- The connection configuration we set up passed to the properties.

The code is as follows:

```
# Write DataFrame to active_user table in RDS
clean_user_df.write.jdbc(url=jdbc_url, table='active_user', mode=mode, prope
```

```
# Write dataframe to billing_info table in RDS
clean_billing_df.write.jdbc(url=jdbc_url, table='billing_info', mode=mode, p
```

```
# Write dataframe to payment_info table in RDS
clean_payment_df.write.jdbc(url=jdbc_url, table='payment_info', mode=mode, p
```

Let's wrap up by double-checking our work and running queries in pgAdmin on our database to confirm that the load did exactly what we wanted:

```
-- Query database to check successful upload
SELECT * FROM active_user;
```

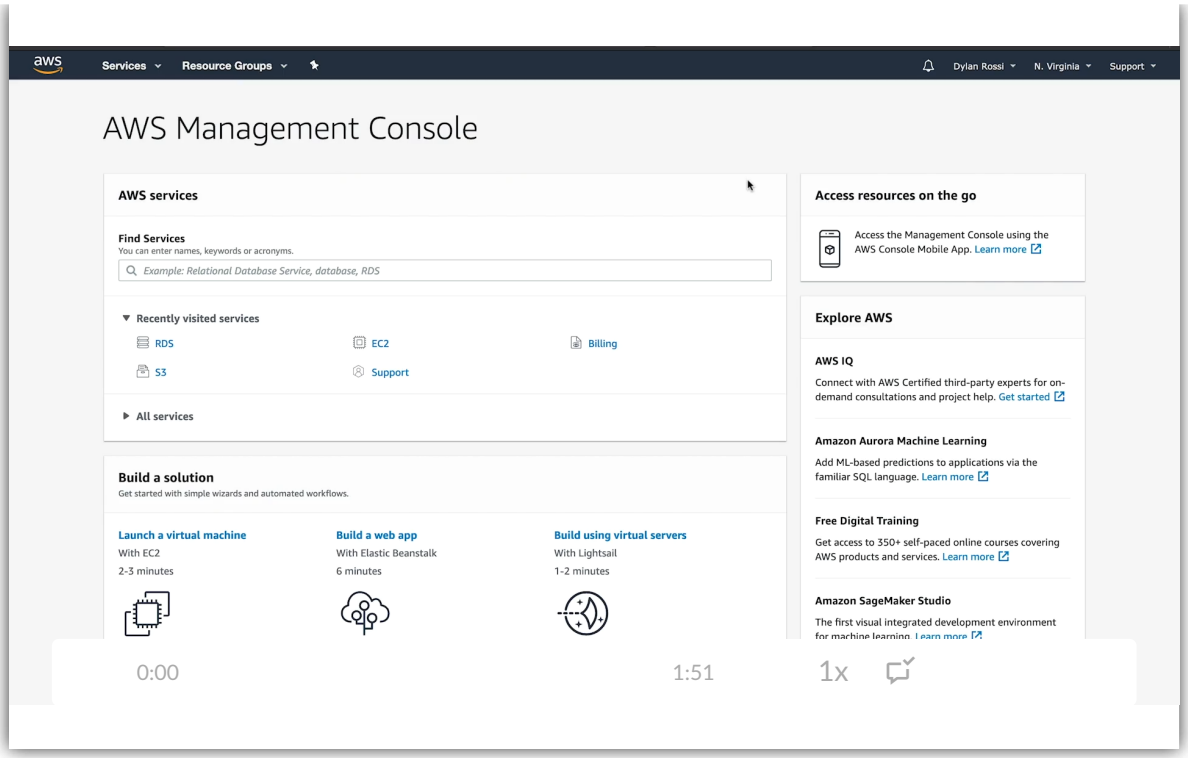
```
SELECT * FROM billing_info;
SELECT * FROM payment_info;
```

Nice work! You now have enough knowledge and practice with PySpark and AWS to begin your client project.

16.9.2 Shut Down Your Instances

Everything you have done with AWS so far was under the free tier. However, just to be sure that nothing accidental happened to accrue charges, you'll need to clean up everything you created.

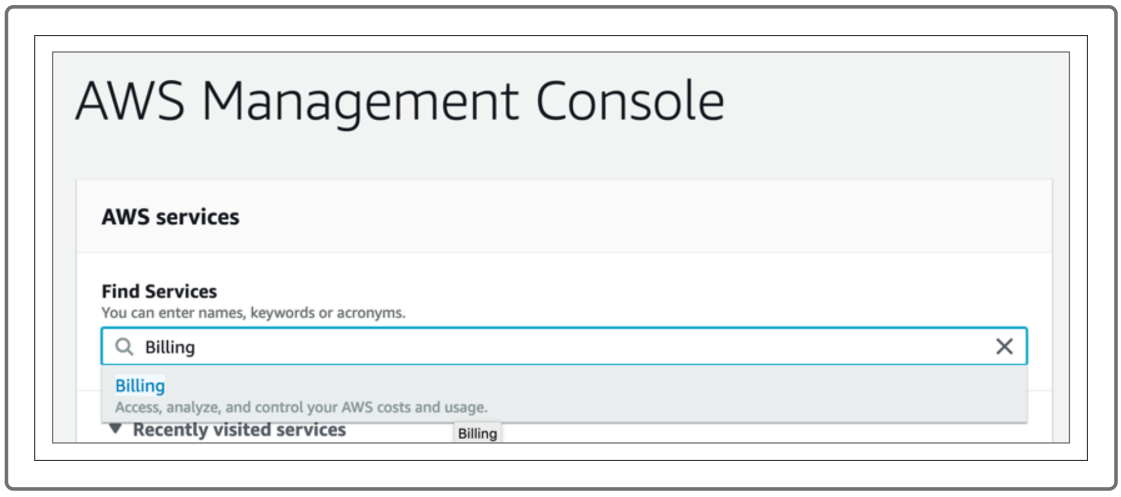
The following video walks you through how to remove all the resources we have created throughout the module:



16.9.3 Check AWS billing

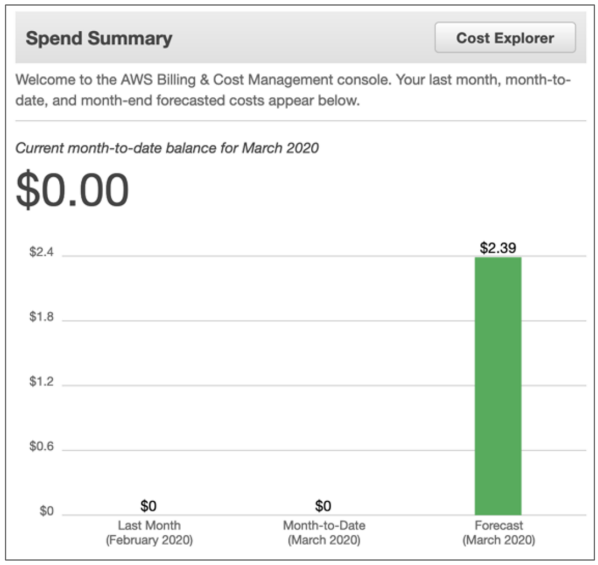
With everything shut down in AWS you should no longer accrue any costs. To be sure, you will check your billing dashboard to confirm nothing falls through.

From the AWS Management Console search for "Billing" and click on the result:

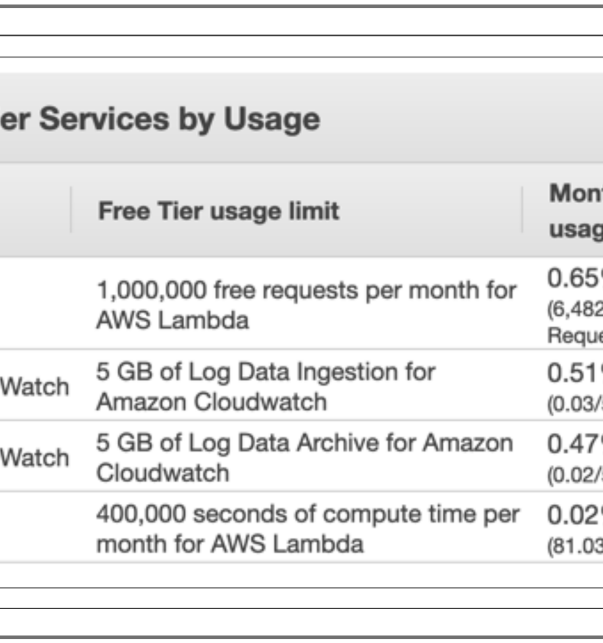


This will bring you to your Billing & Cost Management Dashboard. If you scroll down you will see your "Spend Summary." This is only a forecast and

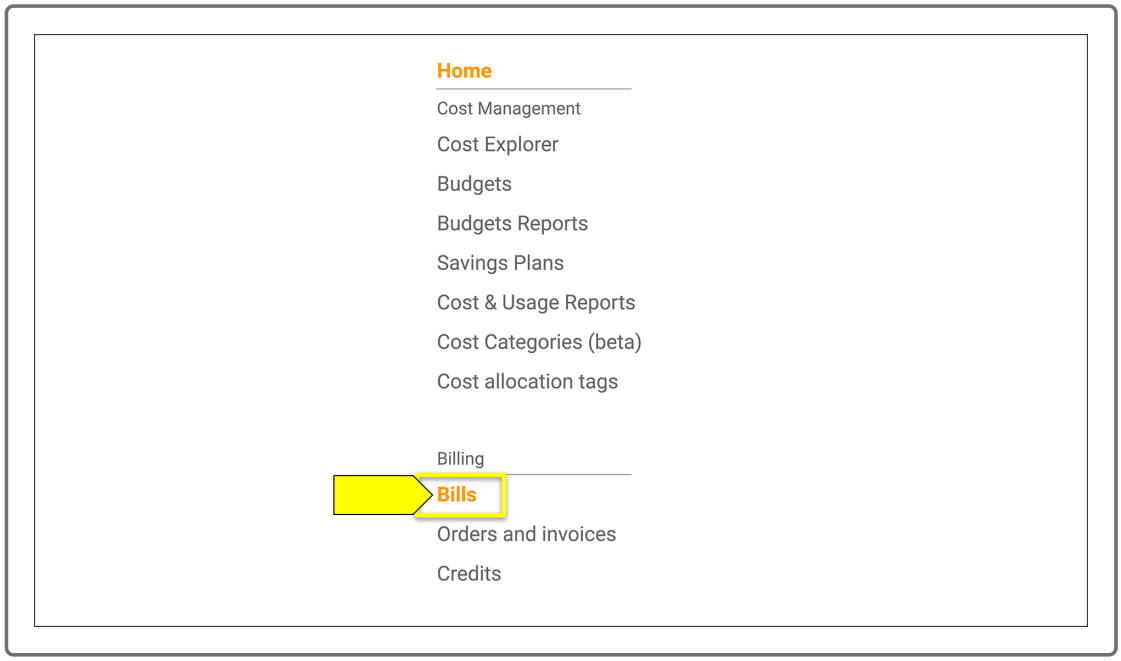
not what you will actually be charged, and this doesn't apply your AWS free usage credits. However, this should be no more than a few dollars:



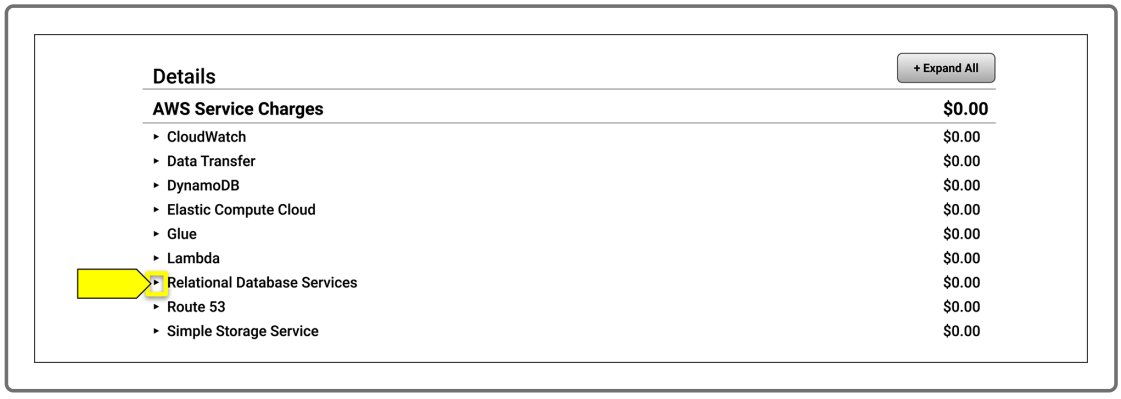
If you keep scrolling down you will find the "Top Free Tier Services by Usage" section. Double check to make sure all your services are within the Free Tier Usage. If they are close to full or higher than expected be sure to delete that service and limit future usage for the month:



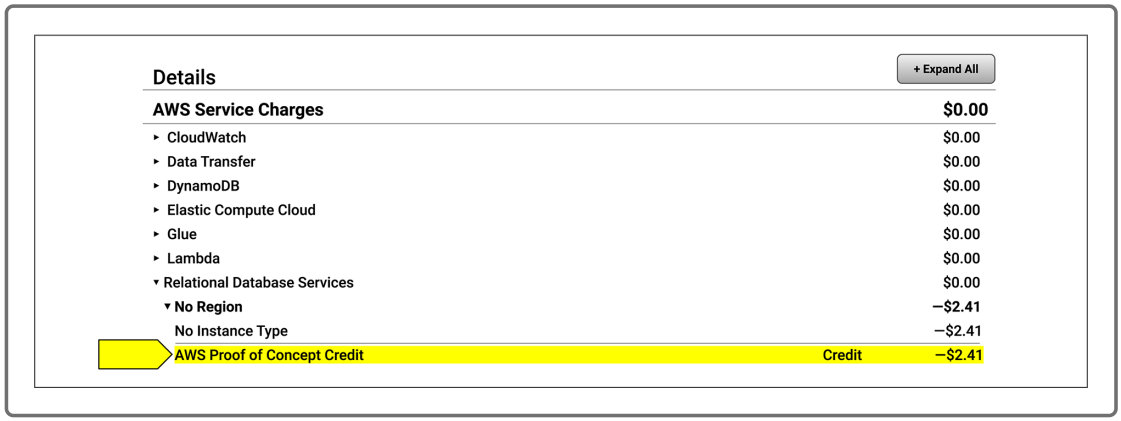
Scroll back up and select "Bills" on the navigation menu located on the left hand side of the page:



The dashboard that this brings you to will show you what you are actually being charged. You should see "$0.00" listed next to each service. You can explore individual services by selecting the arrow to the left of the service:



This page will also display where the AWS Free Tier credits will be applied. Clicking the arrow next to "Relational Database Services" will show the charge you accrued and the AWS credits applied to it:



Be sure to constantly check your billing to make sure surprise costs are not happening. For more information on handling Free Tier, please checkout this **AWS article (https://aws.amazon.com/premiumsupport/knowledge-center/stop-future-free-tier-charges/)** .

Module 16 Challenge

Submit Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Since your work with Jennifer on the SellBy project was so successful, you've been tasked with another, larger project analyzing Amazon reviews. This project will require you to once again perform ETL in the cloud, upload a DataFrame to an RDS instance, and perform statistical analysis of the data.

In this challenge, you have two main goals. First, you'll need to perform the ETL process completely in the cloud and upload a DataFrame to an RDS instance. The second goal is to use PySpark or SQL to perform a statistical analysis of selected data.

NOTE

> Be sure to constantly monitor your AWS usage so you don't go above Free Tier. Consult the "Shutting Down Your AWS instances" and "AWS Billing" for more information.

Instructions

The first step of this challenge is to create DataFrames to match production-ready tables for one of the large Amazon customer review

datasets.

1. Use the furnished schemata to create tables in your RDS database.
2. Create a Google Colab Notebook and extract any dataset from the list of **review datasets** (https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt) , into a notebook.

   Hint: Be sure to handle the header correctly. If you read the file without the header parameter, you might find that the column headers are included in the table rows.

3. For the notebook, complete the following:
   - **Extract** the dataset from the S3 bucket and load into a DataFrame. Note that these are tab-separated files, and the parameter `sep="\t"` should be used.
   - Count the number of records (rows) in the dataset.
   - **Transform** the dataset to fit the tables in the **schema file**  . Be sure the DataFrames match in both data type and column name.
   - **Load** the DataFrames that correspond to tables into an RDS instance.

     Note: This process can take up to 10 minutes for each. Be sure that everything is correct before uploading.

     IMPORTANT

     > Be sure that you don't leave your RDS instance up too long. Try to get all your work done for parts 3 and 4 in one sitting, then shut down your instance. Please consult the AWS clean up videos for shutting down your RDS instance. You will not be

> graded on anything contained strictly in your RDS so be sure to shut them down.

4. Use either PySpark or, for an extra challenge, SQL to analyze the data and determine if the Vine reviews are biased.
    - If you choose to use SQL, use the `vine_table` from the result of the previous step. Perform your analysis with SQL queries on RDS.
    - If you choose to use PySpark create a new notebook and perform your analysis there.
    - While there are no hard requirements for the analysis, consider steps you can take to reduce noisy data (e.g., filtering for reviews that meet a certain number of helpful votes, total votes, or both).

Background

Many of Amazon's shoppers depend on product reviews to make a purchase. Amazon makes these datasets publicly available. However, they are quite large and can exceed the capacity of local machines to handle. One dataset alone contains more than 1.5 million rows. With more than 40 datasets, this can be quite taxing on the average local computer.

Objectives

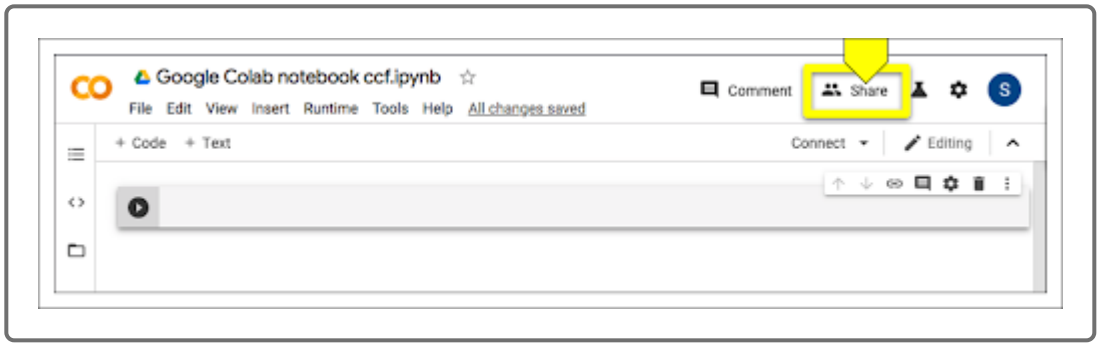
- Perform ETL on one of the review datasets.
- Store your results on an AWS RDS database.

- Determine if reviews are biased using PySpark or SQL with the appropriate statistical methods.

Submission

Submit a link to your Google Colab notebook for the ETL process on the review datasets. If you used PySpark for your analysis, submit a link for that notebook as well. If you used SQL, save your queries to a `.sql` file and submit it in Canvas.

Before submitting a link, make sure to make your notebook public by selecting "Share" in the top right, then "more" from the drop down. From the new screen select "ON - Anyone with the link."



For your written analysis, choose one of the following options:

- Include the analysis in a text cell within your notebook.
- Submit via the text box in Canvas.
- Save to a text file and upload it to Canvas.

Rubric

Please **download the detailed rubric** to access the assessment criteria.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 16 Rubric

Criteria	Ratings					Pts
Extract Please see detailed rubric linked in Challenge description.	**30.0 pts Mastery**	**23.0 pts Approaching Mastery**	**16.0 pts Progressing**	**7.0 pts Emerging**	**0.0 pts No Marks**	30.0 pts
Transform & Load Please see detailed rubric linked in Challenge description.	**30.0 pts Mastery**	**23.0 pts Approaching Mastery**	**16.0 pts Progressing**	**7.0 pts Emerging**	**0.0 pts No Marks**	30.0 pts
Analysis Please see detailed rubric linked in Challenge description.	**40.0 pts Mastery**	**30.0 pts Approaching Mastery**	**20.0 pts Progressing**	**10.0 pts Emerging**	**0.0 pts No Marks**	40.0 pts
						Total Points: 100.0