A Neanderthal’s Guide to Apache Spark in Python

Tutorial on Getting Started with PySpark for Complete Beginners

Evan Heitman
Jun 14, 2019 · 19 min read ★

So You’ve Heard about Apache Spark

If you’re anything like me, you heard about a fancy-sounding technology called Spark and wanted to test your coding mettle to see if you could add another tool to your data-science toolkit. Hopefully you’re not exactly like me because in my case I promptly hit an installation wall, and then a terminology wall, and then a conceptual wall, and four hours later, I hadn’t written a single line of code. And so, after hours of scouring the internet, and more so-called “beginner’s guides” than I care to mention, I decided to write a “neanderthal’s guide” to hopefully spare you some of the hassle that I endured.

. . .

Why read this Guide?

Even a quick search online for learning material on Spark will leave you swimming in documentation, online courses (many of which are *not* cheap), and a menagerie of other resources. From my experience, the majority of these either assumed I knew too much about distributed computing (like assuming I knew what distributed computing meant for example), or they gave high-level or background information without helping me understand how to actually implement anything in Spark.

With that in mind, in this guide I try to do my best to either explain a concept, or direct you somewhere else with an explanation, all with the goal of getting you writing Spark code as quickly as possible. Because I try to do this for as many topics pertaining to Spark as I can, feel free to jump around if you already have a decent grasp of a particular topic. I'll also try to leave you with links to resources that I found helpful as I was diving into Spark.

This is the structure of the guide: I start by explaining some key terminology (i.e. jargon) and concepts so we can be on the same page for the rest of the material and also to lower the barrier to entry to the external resources on Spark you'll find here and elsewhere. Next, I walk through getting a working version of Spark running on your machine using Google Colab. And finally I go through a use case to demonstrate how PySpark is actually implemented and what a first pass through an example problem looks like.

. . .

What You'll Learn

- Enough terminology and concepts to be able to read other Spark resources without being perpetually confused
- A relatively painless way to get PySpark running on your computer
- How to get started with data exploration in PySpark
- Building and evaluating a basic linear regression model in PySpark
- Helpful external resources for the majority of material covered here

. . .

Key Terminology and Concepts

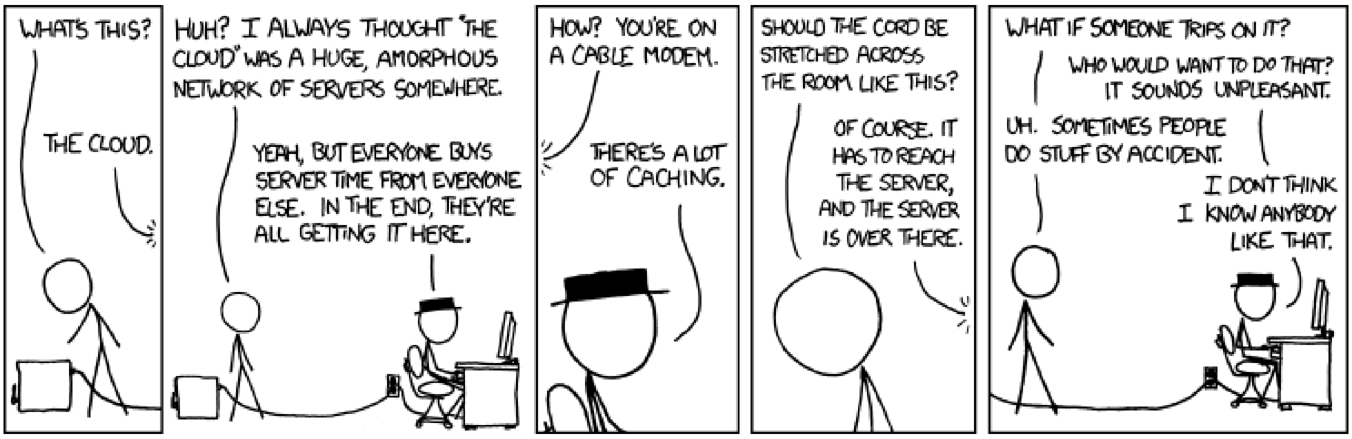
Here is a list of various terms and concepts that will be helpful to know as you delve into the world of Spark.

What is Spark

If you've Googled "what is Spark", there's a chance you've run into the following description, or something like it: *"Spark is a general-purpose distributed data processing engine".* Without a background in Spark or any familiarity with what those terms mean, that definition is rather unhelpful. So let's break it down:

- **Distributed Data/Distributed Computing** — Apache Spark operates in a world that is slightly different from run-of-the-mill computer science. When datasets get too big, or when new data comes in too fast, it can become too much for a single computer to handle. This is where distributed computing comes in. Instead of trying to process a huge dataset or run super computationally-expensive programs on one computer, these tasks can be divided between multiple computers that communicate with each other to produce an output. This technology has some serious benefits, but allocating processing tasks across multiple computers has its own set of challenges and can't be structured the same way as normal processing. When Spark says it has to do with distributed data, this means that it is designed to deal with very large datasets and to process them on a distributed computing system.

> *NOTE: In a distributed computing system, each individual computer is called a* node *and the collection of all of them is called a* cluster


WHAT'S THIS?
THE CLOUD.
HUH? I ALWAYS THOUGHT "THE CLOUD" WAS A HUGE, AMORPHOUS NETWORK OF SERVERS SOMEWHERE.
YEAH, BUT EVERYONE BUYS SERVER TIME FROM EVERYONE ELSE. IN THE END, THEY'RE ALL GETTING IT HERE.
HOW? YOU'RE ON A CABLE MODEM.
THERE'S A LOT OF CACHING.
SHOULD THE CORD BE STRETCHED ACROSS THE ROOM LIKE THIS?
OF COURSE. IT HAS TO REACH THE SERVER, AND THE SERVER IS OVER THERE.
WHAT IF SOMEONE TRIPS ON IT?
WHO WOULD WANT TO DO THAT? IT SOUNDS UNPLEASANT.
UH. SOMETIMES PEOPLE DO STUFF BY ACCIDENT.
I DON'T THINK I KNOW ANYBODY LIKE THAT.

(Comic from xkcd)

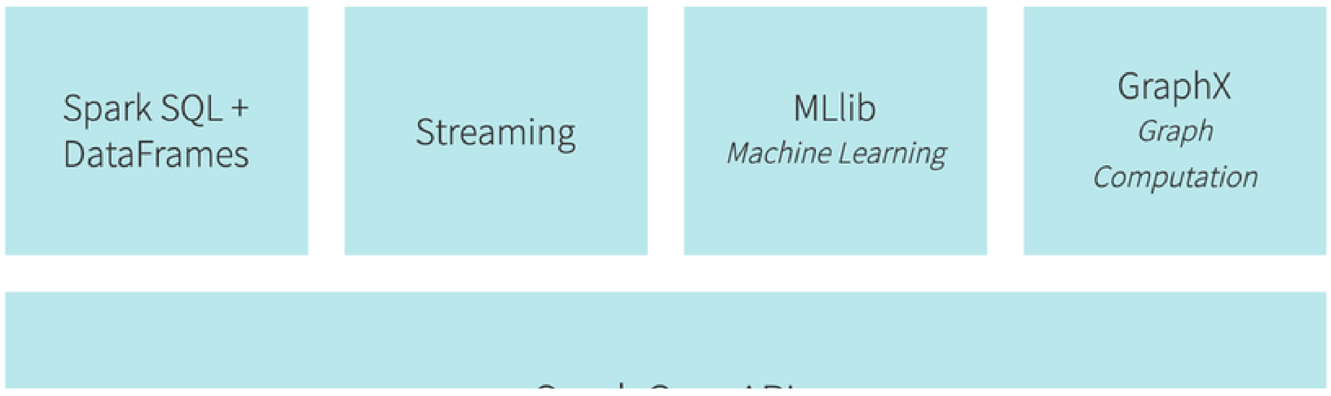
Further Reading — Introduction to distributed computing (8 min read)

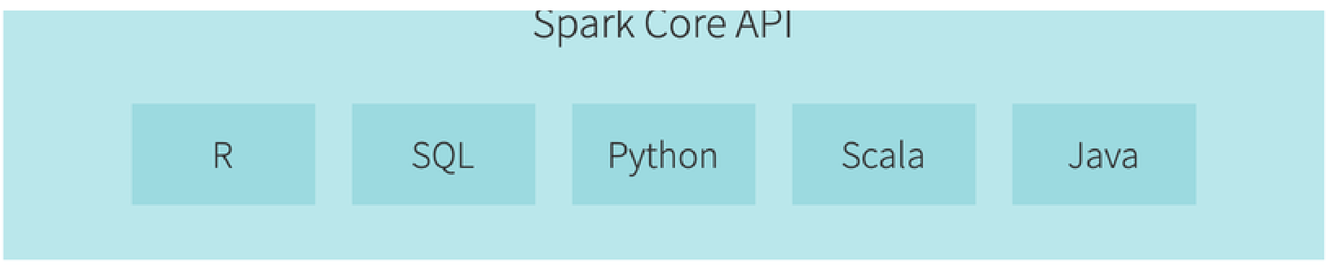
- **Processing Engine/Processing Framework** — A processing engine, sometimes called a processing framework, is responsible for performing data processing tasks (an illuminating explanation, I know). A comparison is probably the best way to understand this. Apache Hadoop is an open source software platform that also deals with “Big Data” and distributed computing. Hadoop has a processing engine, distinct from Spark, called MapReduce. MapReduce has its own particular way of optimizing tasks to be processed on multiple nodes and Spark has a different way. One of Sparks strengths is that it is a processing engine that can be used on its own, or used in place of Hadoop MapReduce, taking advantage of the other features of Hadoop.

(Image from Brad Anderson)

Further Reading — Processing Engines explained and compared (~10 min read)

- **General-Purpose** — One of the main advantages of Spark is how flexible it is, and how many application domains it has. It supports Scala, Python, Java, R, and SQL. It has a dedicated SQL module, it is able to process streamed data in real-time, and it has both a machine learning library and graph computation engine built on top of it. All these reasons contribute to why Spark has become one of the most popular processing engines in the realm of Big Data.





Spark Functionality (from Databricks.com)

Further Reading — 5 minute guide to understanding the significance of Spark (probably more like ~10 min read)

Distributed Computing Terms

- **Partitioned Data** — When working with a computer cluster, you can't just throw in a vanilla dataframe and expect it to know what to do. Because the processing tasks will be divided across multiple nodes, the data also has to be able to be divided across multiple nodes. Partitioned data refers to data that has been optimized to be able to be processed on multiple nodes.

Further Reading — Explanation of Data Partitioning (2 min read)

- **Fault Tolerance** — In short, fault tolerance refers to a distributed system's ability to continue working properly even when a failure occurs. A failure could be a node bursting into flames for example, or just a communication breakdown between nodes. Fault tolerance in Spark revolves around Spark's RDDs (which will be discussed later). Basically, the way data storage is handled in Spark allows Spark programs to function properly despite occurences of failure.

Further Reading — How is Spark fault tolerant (~1 min read)

- **Lazy Evaluation** — Lazy evaluation, or lazy computing, has to do with how code is compiled. When a compiler that is not lazy (which is called strict evaluation) compiles code, it sequentially evaluates each expression it comes across. A lazy compiler on the other hand, doesn't continually evaluate expressions, but rather, waits until it is actually told to generate a result, and then performs all the evaluation all at once. So as it compiles code, it keeps track of everything it will eventually have to evaluate (in Spark this kind of evaluation log, so to speak, is called a lineage graph), and then whenever it is prompted to return something, it performs evaluations according to what it has in its evaluation log. This is useful because it makes programs more efficient as the compiler doesn't have to evaluate anything that isn't actually used.

Further Reading — What is Lazy Evaluation (4 min read)

Spark Terms

- **RDDs, DataFrames, DataSets, Oh My! — Spark RDDs** (Resilient Distributed Datasets) are data structures that are the core building blocks of Spark. A RDD is an immutable, partitioned collection of records, which means that it can hold values, tuples, or other objects, these records are partitioned so as to be processed on a distributed system, and that once an RDD has been made, it is impossible to alter it. That basically sums up its acronym: they are resilient due to their immutability and lineage graphs (which will be discussed shortly), they can be distributed due to their partitions, and they are datasets because, well, they hold data.
  A crucial thing to note is that RDDs *do not have a schema,* which means that they do not have a columnar structure. Records are just recorded row-by-row, and are displayed similar to a list. Enter **Spark DataFrames**. Not to be confused with Pandas DataFrames, as they are distinct, Spark DataFrame have all of the features of RDDs but also have a schema. This will make them our data structure of choice for getting started with PySpark.
  Spark has another data structure, **Spark DataSets**. These are similar to DataFrames but are *strongly-typed,* meaning that the type is specified upon the creation of the DataSet and is not inferred from the type of records stored in it. This means DataSets are not used in PySpark because Python is a dynamically-typed language. For the rest of these explanations I'll be referring to RDDs but know that what is true for an RDD is also true for a DataFrame, DataFrames are just organized into a columnar structure.

Further Reading — RDDs, DataFrames, & DataSets compared (~5 min read)
Further Reading — Pandas v. Spark DataFrames (4 min read)
Further Reading — Helpful RDD Documentation (~5 min read)

- **Transformations** — Transformations are one of the things you can do to an RDD in Spark. They are lazy operations that create one or more new RDDs. It's important to note that Transformations create *new* RDDs because, remember, RDDs are immutable so they can't be altered in any way once they've been created. So, in essence, Transformations take an RDD as an input and perform some function on them based on what Transformation is being called, and outputs one or more RDDs.

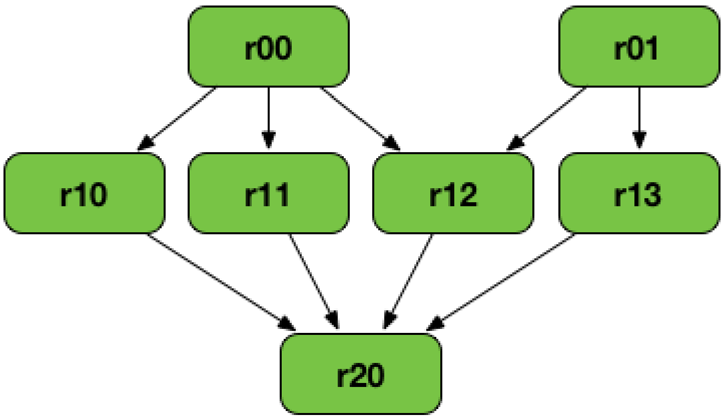
Recalling the section on lazy evaluation, as a compiler comes across each Transformation, it doesn't actually build any new RDDs, but rather constructs a chain of hypothetical RDDs that would result from those Transformations which will only be evaluated once an *Action* is called. This chain of hypothetical, or "child", RDDs, all connected logically back to the original "parent" RDD, is what a lineage graph is.

Further Reading — Helpful Transformation Documentation (~2 min read) *Further Reading* — More in-depth Documentation (5–10 min read; Transformations in first half)

- **Actions** — An Action is any RDD operation that does not produce an RDD as an output. Some examples of common Actions are doing a count of the data, or finding the max or min, or returning the first element of an RDD, etc. As was mentioned before, an Action is the cue to the compiler to evaluate the lineage graph and return the value specified by the Action.

Further Reading — Helpful Action Documentation (~1 min read)
Further Reading — More in-depth Documentation (~5 min read; Actions in second half)

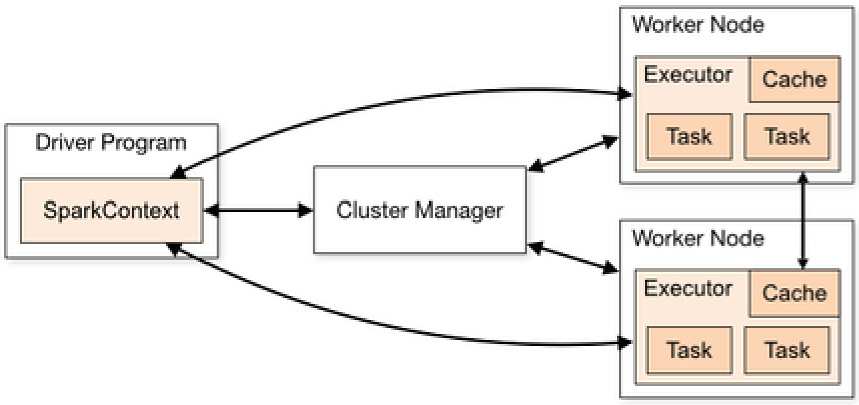
- **Lineage Graph** — Most of what a lineage graph is was described in the Transformations and Actions sections, but to summarize, a lineage graph outlines what is called a "logical execution plan". What that means is that the compiler begins with the earliest RDDs that aren't dependent on any other RDDs, and follows a logical chain of Transformations until it ends with the RDD that an Action is called on. This feature is primarily what drives Spark's fault tolerance. If a node fails for some reason, all the information about what that node was supposed to be doing is stored in the lineage graph, which can be replicated elsewhere.



Visualization of example lineage graph; r00, r01 are parent RDDs, r20 is final RDD (from Jacek Laskowski)

Further Reading — Helpful Lineage Documentation (~2 min read)

- **Spark Applications and Jobs** — There is a lot of nitty gritty when it comes to how a processing engine like Spark actually executes processing tasks on a distributed system. The following is just as much as you'll need to know in order to have a working understanding of what certain snippets of Spark code do. In Spark, when an item of processing has to be done, there is a "driver" process that is in charge of taking the user's code and converting it into a set of multiple tasks. There are also "executor" processes, each operating on a separate node in the cluster, that are in charge of running the tasks, as delegated by the driver. Each driver process has a set of executors that it has access to in order to run tasks. A Spark **application** is a user built program that consists of a driver and that driver's associated executors. A Spark **job** is task or set of tasks to be executed with executor processes, as directed by the driver. A job is triggered by the calling of an RDD Action. This stuff can be rather confusing, so don't sweat it if it doesn't make total sense at first, it's just helpful to be familiar with these terms when they are implemented in code later on. I've included extra resources on this topic if you want more information.



Visualization of Spark Architecture (from Spark API)

Further Reading — Cluster Mode Overview from Spark API (~3 min read)
Further Reading — Helpful Answer on StackOverflow (~2 min read)
Further Reading — Spark Application Overview on Cloudera (~2 min read)

Phew you made it through all the terminology and concepts! Now let's get into implementation!

. . .

Installing Spark

That heading might be a bit of a misnomer, because, strictly speaking, this guide won't show you how to install Apache Spark. Installing Spark can be a pain in the butt. For one, writing Spark applications can be done in multiple languages and each one is installed slightly differently. The underlying API for Spark is written in Scala but PySpark is an overlying API for implementation in Python. For data science applications, using PySpark and Python is widely recommended over Scala, because it is relatively easier to implement. And so instead of installing PySpark, this guide will show you how to run it in Google Colab.

Google Colab

When I was trying to get PySpark running on my computer, I kept getting conflicting instructions on where to download it from (it can be downloaded from spark.apache.org or pip installed for example), what to run it in (it can be run in Jupyter Notebooks or in the native pyspark shell in the command line), and there were numerous obscure bash commands sprinkled throughout. As a data scientist, my reaction to bash commands that aren't pip installs is generally a mix of disgust and despair, and so I turned to Google Colab.

Google Colab is a really powerful interactive python notebook (.ipynb) tool that has a lot data science libraries pre-installed. For more information on what it is and how to run it check out this super helpful article (8 min read).

Once you've got a Colab notebook up, to get Spark running you have to run the following block of code (I know it's not my fault, but I apologize for how ugly it is).

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://www-us.apache.org/dist/spark/spark-2.4.3/spark-
2.4.3-bin-hadoop2.7.tgz
!tar xf spark-2.4.3-bin-hadoop2.7.tgz
!pip install -q findspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.3-bin-hadoop2.7"
import findspark
findspark.init()
```

> *NOTE: When I first ran this block of code it did not run. It was because there had been a new version of Spark released since the code I found was written and I was trying to access an older version of Spark that couldn't be found. So if the above code doesn't run, double check this website to see what the latest version of Spark is and replace everywhere you see "2.4.3" in the above snippet to whatever the newest version is.*

Basically what this block of code does is download the right versions of Java (Spark uses some Java) and Spark, set the PATH to those versions, and to initialize Spark in your notebook.

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget https://mirrors.up.pt/pub/apache/spark/spark-2.4.5/spark-2.4.5-bin-hadoop2.7.tgz
!tar xf spark-2.4.5-bin-hadoop2.7.tgz
!pip install -q findspark
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"
import findspark
findspark.init()
```

. . .

Coding in PySpark

Because we want to be working with columnar data, we'll be using DataFrames which are a part of Spark SQL.

> *NOTE: To avoid possible confusion, despite the fact that we will be working with Spark SQL,* ***none of this will be SQL code****. You can write SQL queries when working with Spark DataFrames but you don't have to.*

Configuring a SparkSession

The entry point to using Spark SQL is an object called `SparkSession`. It initiates a Spark *Application* which all the code for that Session will run on.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local[*]") \
    .appName("Learning_Spark") \
    .getOrCreate()
```

> *NOTE: the "\" character is this context is called a continuation character which is just a helpful wrapping tool for making long lines of code more readable.*

- `.builder` — gives access to Builder API which is used to configure the session .
- `.master()` — determines where the program will run; `"local[*]"` sets it to run locally on all cores but you can use `"local[1]"` to run on one core for example. In this case, our programs will be run on Google's servers.
- `.appName()` — optional method to name the Spark Application
- `.getOrCreate()` — gets an existing `SparkSession` or creates new one if none exists

Check the Builder API for more options when building a `SparkSession`.

Loading Data

To open a local file on Google Colab you need to run the following code which will prompt you to select a file from your computer:

```
from google.colab import files
files.upload()
```

For this guide we'll be working with a dataset on video game sales from Kaggle. It can be found here.

Now load our data into a Spark DataFrame using the `.read.csv()` function: (I shortened the file name for brevity's sake)

```
data = spark.read.csv('Video_Games_Sales.csv',inferSchema=True,
header=True)
```

NOTE: This function is specifically for reading CSV files into a DataFrame in PySparkSQL. It won't work for loading data into an RDD and different languages (besides Python) have different syntax. Exercise caution when searching online for help because many resources do not assume Spark SQL or Python.

Data Exploration

Now let's move into understanding how we can get more familiar with our data!

The first thing we can do is check the shape of of our DataFrame. Unlike Pandas, there is no dedicated method for this but we can use the `.count()` and `.columns()` to retrieve the information ourselves.

```
data.count(), len(data.columns)

>>> (16719, 16)
```

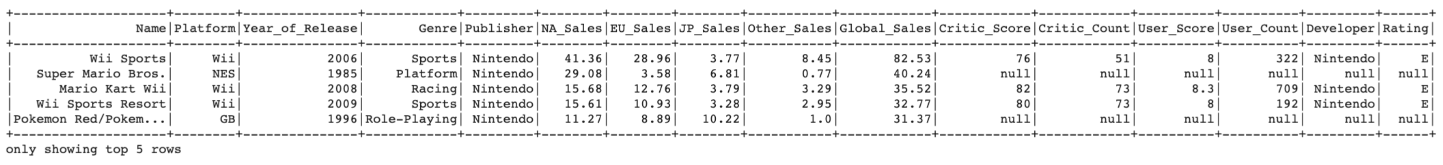
The `.count()` method returns the number of rows in the DataFrame and `.columns` returns a list of column names.

NOTE: We don't have to actually print them because Colab will automatically display the last output of each cell. If you want to show more than one output you will have to print them (unless you use this workaround, which is super nice and works in Jupyter Notebooks as well)

Viewing DataFrames

To view a DataFrame, use the `.show()` method:

```
data.show(5)
```



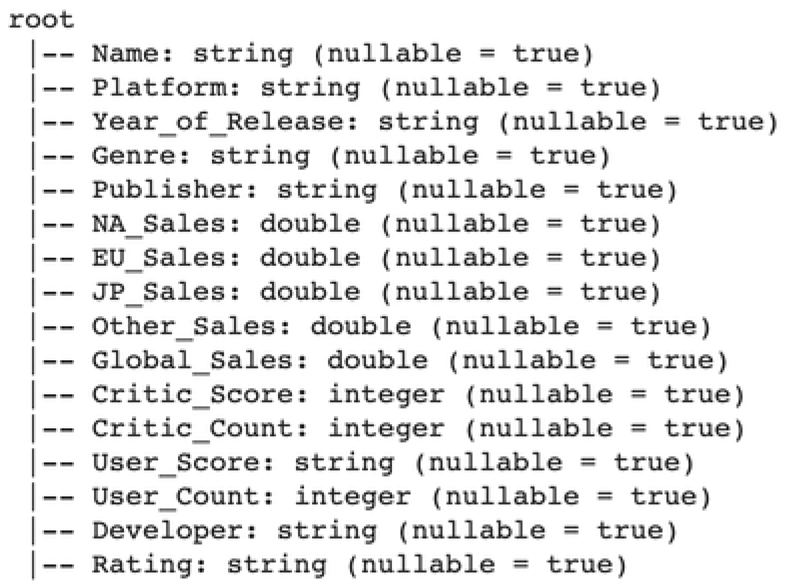
```
+--------------------+--------+---------------+------------+---------+--------+--------+--------+-----------+------------+------------+------------+----------+----------+---------+------+
|                Name|Platform|Year_of_Release|       Genre|Publisher|NA_Sales|EU_Sales|JP_Sales|Other_Sales|Global_Sales|Critic_Score|Critic_Count|User_Score|User_Count|Developer|Rating|
+--------------------+--------+---------------+------------+---------+--------+--------+--------+-----------+------------+------------+------------+----------+----------+---------+------+
|          Wii Sports|     Wii|           2006|      Sports| Nintendo|   41.36|   28.96|    3.77|       8.45|       82.53|          76|          51|         8|       322| Nintendo|     E|
|   Super Mario Bros.|     NES|           1985|    Platform| Nintendo|   29.08|    3.58|    6.81|       0.77|       40.24|        null|        null|      null|      null|     null|  null|
|      Mario Kart Wii|     Wii|           2008|      Racing| Nintendo|   15.68|   12.76|    3.79|       3.29|       35.52|          82|          73|       8.3|       709| Nintendo|     E|
|   Wii Sports Resort|     Wii|           2009|      Sports| Nintendo|   15.61|   10.93|    3.28|       2.95|       32.77|          80|          73|         8|       192| Nintendo|     E|
|Pokemon Red/Pokem...|      GB|           1996|Role-Playing| Nintendo|   11.27|    8.89|   10.22|        1.0|       31.37|        null|        null|      null|      null|     null|  null|
+--------------------+--------+---------------+------------+---------+--------+--------+--------+-----------+------------+------------+------------+----------+----------+---------+------+
only showing top 5 rows
```

Output from data.show(5)

As you can see, running `data.show(5)` displayed the first 5 rows of our DataFrame, along with the header. Calling `.show()` with no parameters will return the first 20 records.

Let's see what our data is comprised of using the `.printSchema()` method (alternatively you can use `.dtypes`):

```
data.printSchema()
```



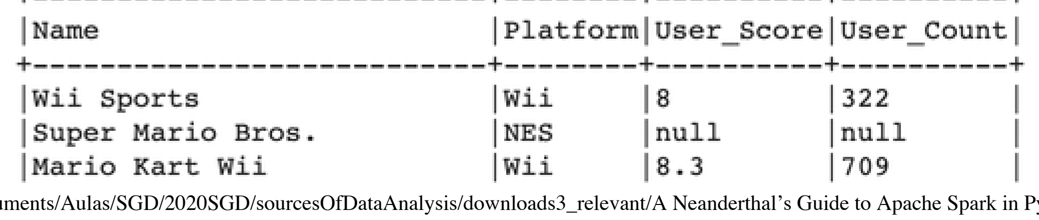
```
root
 |-- Name: string (nullable = true)
 |-- Platform: string (nullable = true)
 |-- Year_of_Release: string (nullable = true)
 |-- Genre: string (nullable = true)
 |-- Publisher: string (nullable = true)
 |-- NA_Sales: double (nullable = true)
 |-- EU_Sales: double (nullable = true)
 |-- JP_Sales: double (nullable = true)
 |-- Other_Sales: double (nullable = true)
 |-- Global_Sales: double (nullable = true)
 |-- Critic_Score: integer (nullable = true)
 |-- Critic_Count: integer (nullable = true)
 |-- User_Score: string (nullable = true)
 |-- User_Count: integer (nullable = true)
 |-- Developer: string (nullable = true)
 |-- Rating: string (nullable = true)
```

Output from data.printSchema()

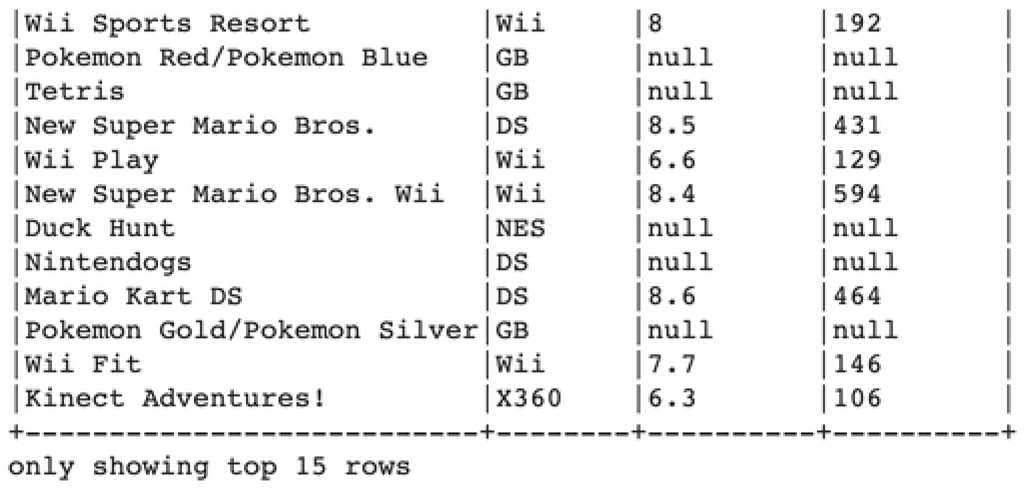
Some takeaways from this output is that Year_of_Release and User_Score have a type of string, despite them being numbers. It also tells us that each of the columns allows null values which can be seen in the first 5 rows.

We can also selectively choose which columns we want to display with the `.select()` method. Let's view only Name, Platform, User_Score, and User_Count:

```
data.select("Name","Platform","User_Score","User_Count") \
.show(15, truncate=False)
```



```
+---------------------------+--------+----------+----------+
|Name                       |Platform|User_Score|User_Count|
+---------------------------+--------+----------+----------+
|Wii Sports                 |Wii     |8         |322       |
|Super Mario Bros.          |NES     |null      |null      |
|Mario Kart Wii             |Wii     |8.3       |709       |
```



```
|Wii Sports Resort         |Wii     |8         |192       |
|Pokemon Red/Pokemon Blue  |GB      |null      |null      |
|Tetris                    |GB      |null      |null      |
|New Super Mario Bros.     |DS      |8.5       |431       |
|Wii Play                  |Wii     |6.6       |129       |
|New Super Mario Bros. Wii |Wii     |8.4       |594       |
|Duck Hunt                 |NES     |null      |null      |
|Nintendogs                |DS      |null      |null      |
|Mario Kart DS             |DS      |8.6       |464       |
|Pokemon Gold/Pokemon Silver|GB     |null      |null      |
|Wii Fit                   |Wii     |7.7       |146       |
|Kinect Adventures!        |X360    |6.3       |106       |
+--------------------------+--------+----------+----------+
only showing top 15 rows
```

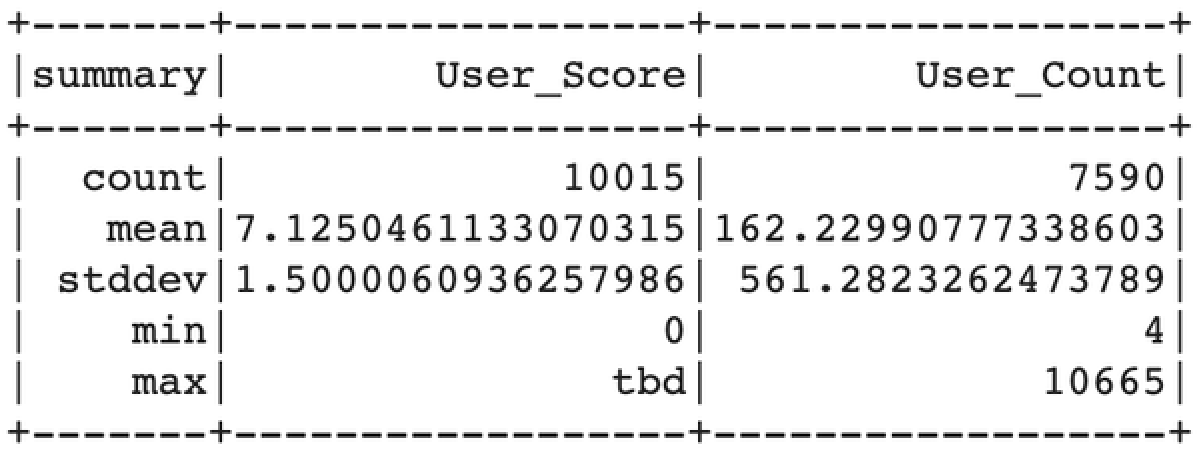
Output from data.select().show()

Included is the `truncate=False` parameter that adjusts the size of columns to prevent values from being cut off.

Summary Statistics/Information

We can use the `.describe()` method to get summary statistics on columns of our choosing:

```
data.describe(["User_Score","User_Count"]).show()
```



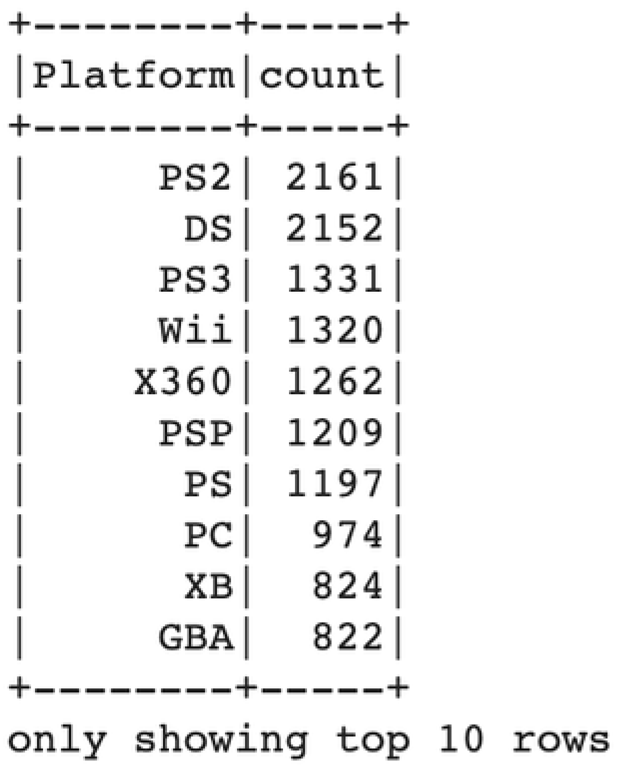
```
+-------+-----------------+-----------------+
|summary|       User_Score|       User_Count|
+-------+-----------------+-----------------+
|  count|            10015|             7590|
|   mean|7.125046113307031|162.2299077733860|
| stddev|1.500006093625798| 561.282326247378|
|    min|                0|                4|
|    max|              tbd|            10665|
+-------+-----------------+-----------------+
```

Output from data.describe().show()

Some takeaways from this output is that there sees to a strange "tbd" value in the User_Score column. The count for User_Score is also higher than User_Count but it's hard to tell if that's because there are actually more values in User_Score or if "tbd" values are artificially raising the count. We'll learn how to filter those values out later on.

We might also want to get some information on what kinds of platforms are in the Platform column and how they are distributed. We can use a `groupBy()` for this and sort it using `.orderBy()`:

```
data.groupBy("Platform") \
.count() \
.orderBy("count", ascending=False) \
.show(10)
```



```
+--------+-----+
|Platform|count|
+--------+-----+
|     PS2| 2161|
|      DS| 2152|
|     PS3| 1331|
|     Wii| 1320|
|    X360| 1262|
|     PSP| 1209|
|      PS| 1197|
|      PC|  974|
|      XB|  824|
|     GBA|  822|
+--------+-----+
only showing top 10 rows
```

Output from data.groupBy().orderBy().show()

Here we're looking at the top 10 most frequent platforms. We can tell this dataset is pretty old because I don't see PS4 anywhere 🤔

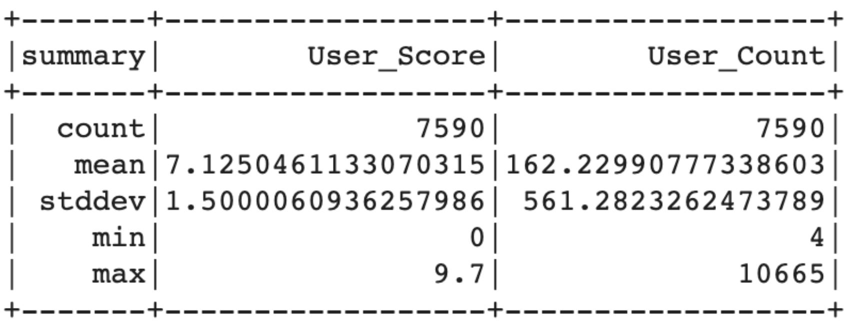
Filtering DataFrames

Lets create a new DataFrame that has the null values for User_Score and User_Count, and the "tbd" values filtered out using the `.filter()` method:

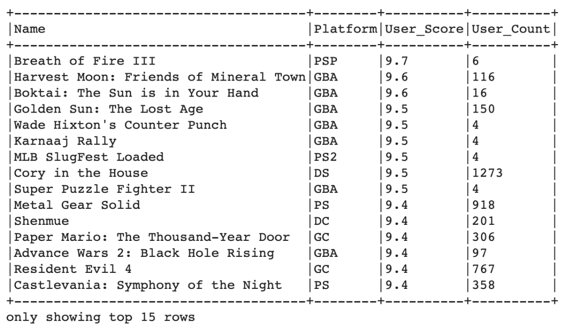
```
condition1 = (data.User_Score.isNotNull()) |
(data.User_Count.isNotNull())

condition2 = data.User_Score != "tbd"

data = data.filter(condition1).filter(condition2)
```

`condition1` returns True for any record that does not have a null value in User_Score or in User_Count. `condition2` returns True for any record that does not have “tbd” in User_Score.

We can double check to see if our filtering worked by reconstructing our earlier visualizations



```
+-------+------------------+------------------+
|summary|        User_Score|        User_Count|
+-------+------------------+------------------+
|  count|              7590|              7590|
|   mean|7.1250461133070315|162.22990777338603|
| stddev|1.5000060936257986| 561.2823262473789|
|    min|                 0|                 4|
|    max|               9.7|             10665|
+-------+------------------+------------------+
```



```
+-------------------------------------+--------+----------+----------+
|Name                                 |Platform|User_Score|User_Count|
+-------------------------------------+--------+----------+----------+
|Breath of Fire III                   |PSP     |9.7       |6         |
|Harvest Moon: Friends of Mineral Town|GBA     |9.6       |116       |
|Boktai: The Sun is in Your Hand      |GBA     |9.6       |16        |
|Golden Sun: The Lost Age             |GBA     |9.5       |150       |
|Wade Hixton's Counter Punch          |GBA     |9.5       |4         |
|Karnaaj Rally                        |GBA     |9.5       |4         |
|MLB SlugFest Loaded                  |PS2     |9.5       |4         |
|Cory in the House                    |DS      |9.5       |1273      |
|Super Puzzle Fighter II              |GBA     |9.5       |4         |
|Metal Gear Solid                     |PS      |9.4       |918       |
|Shenmue                              |DC      |9.4       |201       |
|Paper Mario: The Thousand-Year Door  |GC      |9.4       |306       |
|Advance Wars 2: Black Hole Rising    |GBA     |9.4       |97        |
|Resident Evil 4                      |GC      |9.4       |767       |
|Castlevania: Symphony of the Night   |PS      |9.4       |358       |
+-------------------------------------+--------+----------+----------+
only showing top 15 rows
```

Reconstructed summary statistics and DataFrame with filtered out values

That’s enough Data Exploration to get started, now lets build a model!

Building Models in PySpark

Building models in PySpark looks a little different than you might be used to, and you’ll see terms like *Transformer, Estimator,* and *Param*. This guide won’t go in-depth into what those terms mean but below is a link to a brief description of what they mean.

Further Reading — Machine Learning in Spark (~5–10 min read)

Setup

For an example of linear regression, let’s see if we can predict User_Score from Year_of_Release, Global_Sales, Critic_Score, and User_Count.

First let’s recode all of our predictors to be Doubles (I found that this got rid of some really gnarly errors later on).

```
from pyspark.sql.types import DoubleType

data2 = data2.withColumn("Year_of_Release",
data2["Year_of_Release"].cast(DoubleType()))

data2 = data2.withColumn("User_Score",
data2["User_Score"].cast(DoubleType()))
```

```
data2 = data2.withColumn("User_Count",
data2["User_Count"].cast(DoubleType()))

data2 = data2.withColumn("Critic_Score",
data2["Critic_Score"].cast(DoubleType()))
```

We use the method `withColumn()` , which either creates a new column or replaces one that already exists. So, for example, the Year_of_Release column is replaced with a version of itself that has been cast as doubles .

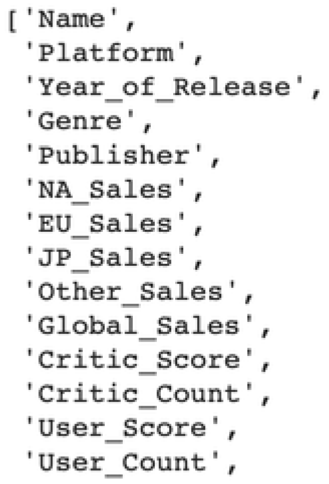
VectorAssembler

The next step is to get our data into a form that PySpark can create a model with. To do this we use something called a `VectorAssembler` .

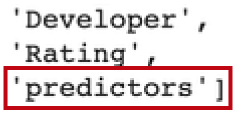
```
from pyspark.ml.feature import VectorAssembler

inputcols = ["Year_of_Release",  "Global_Sales", "Critic_Score",
"User_Count"]

assembler = VectorAssembler(inputCols= inputcols,
                            outputCol = "predictors")
```

Here we've delineated what features we want our model to use as predictors so that `VectorAssembler` can take those columns and transform them into a single column (named "predictors") that contains all the data we want to predict with.

```
predictors = assembler.transform(data)

predictors.columns
```



```
['Name',
 'Platform',
 'Year_of_Release',
 'Genre',
 'Publisher',
 'NA_Sales',
 'EU_Sales',
 'JP_Sales',
 'Other_Sales',
 'Global_Sales',
 'Critic_Score',
 'Critic_Count',
 'User_Score',
 'User_Count',
```



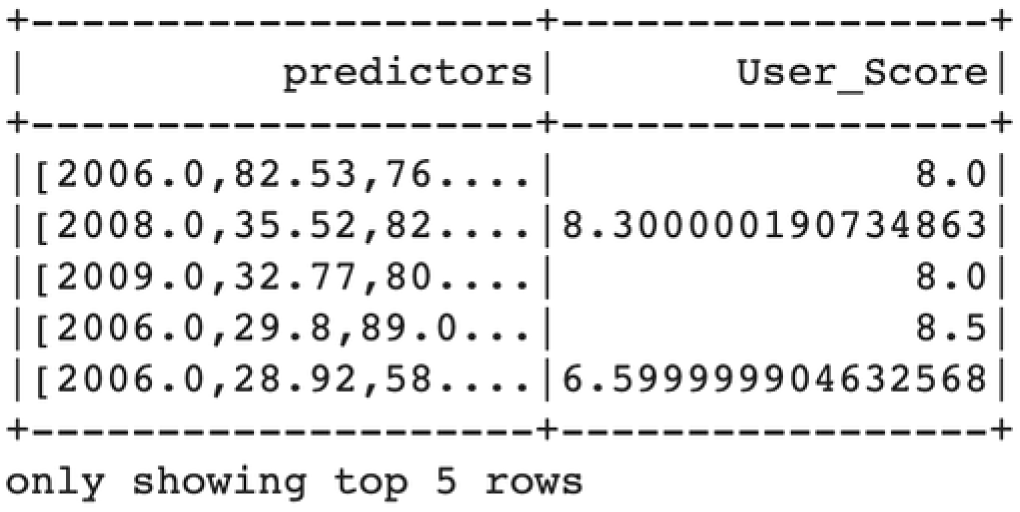
```
'Developer',
'Rating',
'predictors']
```

Output from VectorAssembler.transform().columns

What `VectorAssembler.transform()` does is create a new DataFrame with a new column at the end where each row contains a list of all the features we included in the `inputCols` parameter when we created the assembler.

The final step to getting our data ready to be used in a model is to collect the new predictions column we just made and User_Score (our target variable) by themselves in a DataFrame.

```
model_data = predictors.select("predictors", "User_Score")

model_data.show(5,truncate=False)
```



```
+--------------------+-----------------+
|          predictors|       User_Score|
+--------------------+-----------------+
|[2006.0,82.53,76....|              8.0|
|[2008.0,35.52,82....|8.300000190734863|
|[2009.0,32.77,80....|              8.0|
|[2006.0,29.8,89.0...|              8.5|
|[2006.0,28.92,58....|6.599999904632568|
+--------------------+-----------------+
only showing top 5 rows
```

The final data we will use to build a model

Next is to split `model_data` into a training and testing set:

```
train_data,test_data = model_data.randomSplit([0.8,0.2])
```

Model Training

Now to train the model!

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(
    featuresCol = 'predictors',
    labelCol = 'User_Score')

lrModel = lr.fit(train_data)

pred = lrModel.evaluate(test_data)
```

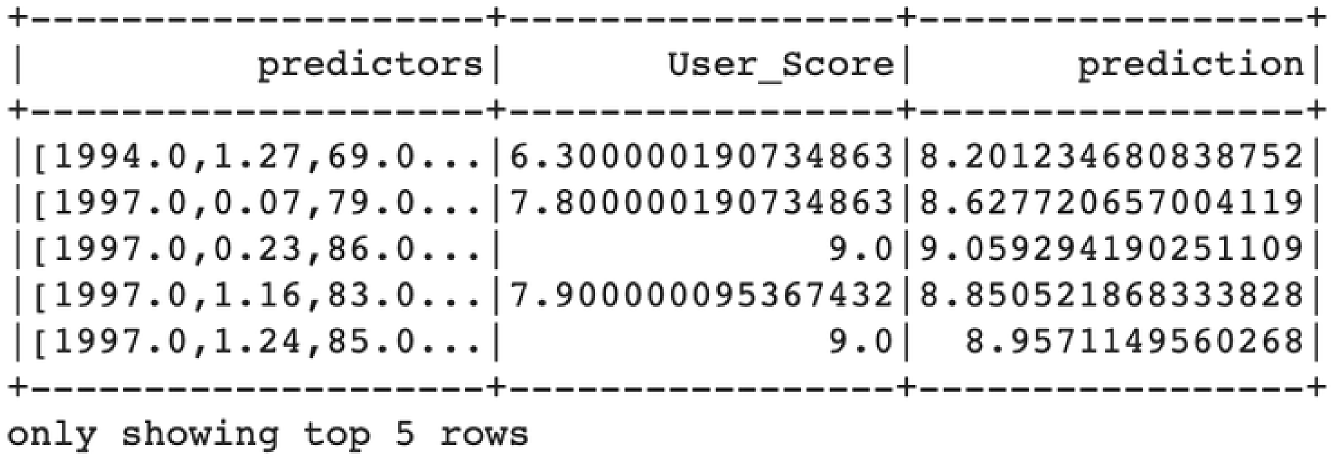
After importing `LinearRegression` from `pyspark.ml.regression`, we construct a regressor and we specify that it should look for a column named "predictors" as the features of the model and a column named "User_Score" as the labels of the model. Next we train it with `.fit()` and finally produce predictions with `.evaluate()`.

We can access the parameters of our model with the following code

```
lrModel.coefficients
>>>
[-0.07872176891379576,-0.0350439561719371,0.06376305861102288,-0.000
2156086537632538]

lrModel.intercept
>>> 160.7985254457876
```

We can also view the final predictions our model made:

```
pred.predictions.show(5)
```



```
+--------------------+-----------------+-----------------+
|          predictors|       User_Score|       prediction|
+--------------------+-----------------+-----------------+
|[1994.0,1.27,69.0...|6.300000190734863|8.201234680838752|
|[1997.0,0.07,79.0...|7.800000190734863|8.627720657004119|
|[1997.0,0.23,86.0...|              9.0|9.059294190251109|
|[1997.0,1.16,83.0...|7.900000095367432|8.850521868333828|
|[1997.0,1.24,85.0...|              9.0|  8.9571149560268|
+--------------------+-----------------+-----------------+
only showing top 5 rows
```

Model predictions

The object named "pred" is a *LinearRegressionSummary* object and so to retrieve the DataFrame with predictions we call `.predictions.show()`

Model Evaluating

To get more detailed information on how our model performed, we can use `RegressionEvaluator` which is constructed like this:

```
from pyspark.ml.evaluation import RegressionEvaluator

eval = RegressionEvaluator(
    labelCol="User_Score",
    predictionCol="prediction",
    metricName="rmse")
```

Let's compute some statistics for the model:

```
rmse = eval.evaluate(pred.predictions)

mse = eval.evaluate(pred.predictions, {eval.metricName: "mse"})

mae = eval.evaluate(pred.predictions, {eval.metricName: "mae"})

r2 = eval.evaluate(pred.predictions, {eval.metricName: "r2"})
```

Which returns

```
rmse
>>> 1.125

mse
>>> 1.266

mae
>>> 0.843

r2
>>> 0.386
```

From this we can interpret that our model tended to be about 1.125 rating points off from the actual User_Score (according to rmse). The r^2 value for tells us that the predictors in our model are able to account for a little under 40% of the total variability in User_Score. This was just a first pass look, and I recommend playing around with the model parameters and features for more practice!

Further Reading — Detailed Code example of Linear Regression (~20+ min to go through the whole thing)
Further Reading — Detailed Code example of Logistic Regression using SQL (~10 minutes)
Further Reading — Example of Linear Regression, Decision Tree, and Gradient-Boosted Tree Regression (6 min read)

. . .

This is just the tip of the iceberg as far as the kind of modeling you can do in PySpark, but I hope this guide has equipped you with enough to get your foot into the door of Big Data!

Conclusion

Wow. Props to you if you made it all the way to the end. You were exposed to a ton of new concepts, from the terminology of distributed computing and Spark, to implementing data exploration and data modeling techniques in PySpark. My hope is that this guide can be a resource for you as you continue working with Spark!

All the code used in this article can be found on GitHub here.

Spark Pyspark Big Data Python Distributed Computing