DATA AND AI SKILLS 50% OFF

Sale ends in
0d 23h 52m 59s

EN

TUTORIALS

category

Pyspark Tutorial: Getting Started with Pyspark

Discover what Pyspark is and how it can be used while giving examples.

Contents

Updated Nov 21, 2024 · 10 min read

Natassha Selvaraj
Data Consultant who works at the intersection of data science and marketing.

TOPICS

Data Visualization

Data Science

Python

As a data science enthusiast, you are probably familiar with storing files on your local device and processing them using languages like R and Python. However, local workstations have their limitations and cannot handle extremely large datasets.

This is where a distributed processing system like Apache Spark comes in. Distributed processing is a setup in which multiple processors are used to run an application. Instead of trying to process large datasets on a single computer, the task can be divided between multiple devices that communicate with each other.

All this represents an exciting innovation. To follow up on this article, you can practice hands-on exercises with our Introduction to PySpark course, which will open doors for you in the area of parallel computing. The ability to analyze data and train machine learning models on large-scale datasets is a valuable skill to have, and having the expertise to work with big data frameworks like Apache Spark will set you apart from others in the field.

What is Apache Spark?

Apache Spark is a distributed processing system used to perform big data and machine learning tasks on large datasets. With Apache Spark, users can run queries and machine learning workflows on petabytes of data, which is impossible to do on your local device.

This framework is even faster than previous data processing engines like Hadoop, and has increased in popularity in the past eight years. Companies like IBM, Amazon, and Yahoo are using Apache Spark as their computational framework.

Become a Data Engineer

Build Python skills to become a professional data engineer.

Get Started for Free

What is PySpark?

PySpark is an interface for Apache Spark in Python. With PySpark, you can write Python and SQL-like commands to manipulate and analyze data in a distributed processing environment. Using PySpark, data scientists manipulate data, build machine learning pipelines, and tune models.

Most data scientists and analysts are familiar with Python and use it to implement machine learning workflows. PySpark allows them to work with a familiar language on large-scale distributed datasets. Apache Spark can also be used with other data science programming languages like R. If this is something you are interested in learning, the **Introduction to Spark with sparklyr in R** course is a great place to start.

Why Use PySpark?

The reason companies choose to use a framework like PySpark is because of how quickly it can process big data. It is faster than libraries like Pandas and Dask, and can handle larger amounts of data than these frameworks. If you had over petabytes of data to process, for instance, Pandas and Dask would fail but PySpark would be able to handle it easily.

While it is also possible to write Python code on top of a distributed system like Hadoop, many organizations choose to use Spark instead and use the PySpark API since it is faster and can handle real-time data. With PySpark, you can write code to collect data from a source that is continuously updated, while data can only be processed in batch mode with Hadoop.

Apache Flink is a distributed processing system that has a Python API called PyFlink, and is actually faster than Spark in terms of performance. However, Apache Spark has been around for a longer period of time and has better community support, which means that it is more reliable.

Furthermore, PySpark provides fault tolerance, which means that it has the capability to recover loss after a failure occurs. The framework also has in-memory computation and is stored in random access memory (RAM). It can run on a machine that does not have a hard-drive or SSD installed.

How to Install PySpark

Before installing Apache Spark and PySpark, you need to have the following software set up on your device:

Python

If you don't already have Python installed, follow our Python developer set-up guide to set it up before you proceed to the next step.

Java

Next, follow this tutorial to get Java installed on your computer if you are using Windows. Here is an installation guide for MacOs, and here's one for Linux.

Jupyter Notebook

A Jupyter Notebook is a web application that you can use to write code and display equations, visualizations, and text. It is one of the most commonly used programming editors by data scientists. We will use a Jupyter Notebook to write all the PySpark code in this tutorial, so make sure to have it installed.

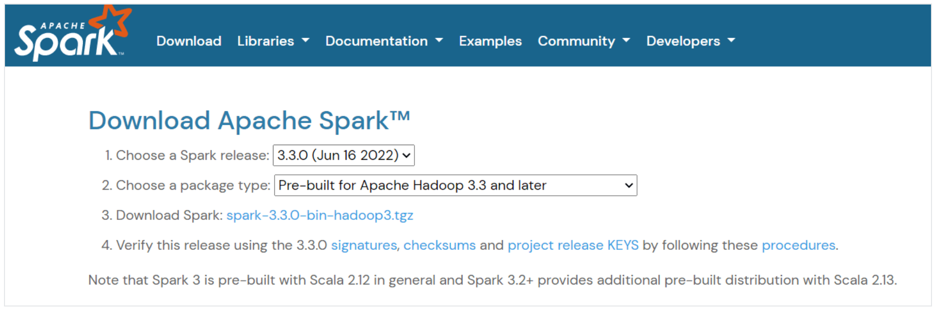
You can follow our tutorial to get Jupyter up and running on your local device or use an online, cloud-based IDE like DataLab built by DataCamp, which comes with PySpark preinstalled. I will pull an e-commerce dataset from DataLab for all the analysis in this tutorial, so make sure to have it downloaded. I've renamed the file to "datacamp_ecommerce.csv" and saved it to the parent directory, and you can do the same so it's easier to code along.

PySpark Installation Guide

Now that you have all the prerequisites set up, you can proceed to install Apache Spark and PySpark. You can skip this step if you're using DataLab.

Installing Apache Spark

To get Apache Spark set up, navigate to the download page and download the .tgz file displayed on the page:



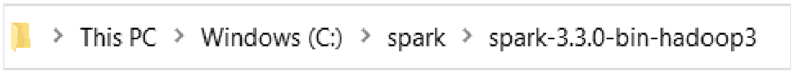
Download Apache Spark™

1. Choose a Spark release: 3.3.0 (Jun 16 2022)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.3.0-bin-hadoop3.tgz
4. Verify this release using the 3.3.0 signatures, checksums and project release KEYS by following these procedures.

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

Then, if you are using Windows, create a folder in your C directory called "spark." If you use Linux or Mac, you can paste this into a new folder in your home directory.

Next, extract the file you just downloaded and paste its contents into this "spark" folder. This is what the folder path should look like:



This PC > Windows (C:) > spark > spark-3.3.0-bin-hadoop3

Now, you need to set your environment variables. There are two ways you can do this:

Method 1: Changing Environment Variables Using Powershell

If you are using a Windows machine, the first way to change your environment variables is by using Powershell:

- **Step 1:** Click on **Start** -> **Windows Powershell** -> **Run as administrator**
- **Step 2:** Type the following line into Windows Powershell to set `SPARK_HOME`:

```
setx SPARK_HOME "C:\spark\spark-3.3.0-bin-hadoop3" # change this to your pa
```

Explain code Run code POWERED BY datalab

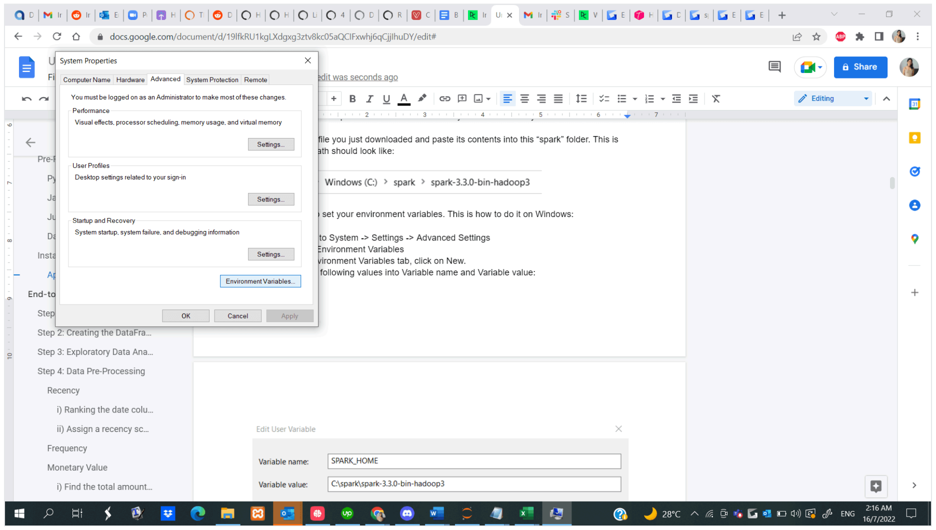
Step 3: Next, set your Spark bin directory as a path variable:

```
setx PATH "C:\spark\spark-3.3.0-bin-hadoop3\bin"
```

Explain code Run code POWERED BY datalab

Method 2: Changing Environment Variables Manually

- **Step 1:** Navigate to **Start** -> **System** -> **Settings** -> **Advanced Settings**
- **Step 2:** Click on **Environment Variables**



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

Environment Variables...

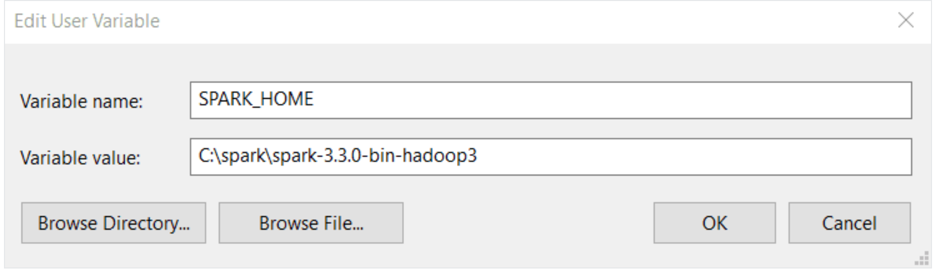
Edit User Variable

Variable name: SPARK_HOME

Variable value: C:\spark\spark-3.3.0-bin-hadoop3

docs.google.com/document/d/19lfkRU1kgLXdgxg3ztv8kc05aQClFxwhj6qCjjlhuDY/edit#
Settings...
OK
Cancel
Apply
Windows (C:) > spark > spark-3.3.0-bin-hadoop3
Step 2: Creating the DataFra...
Step 3: Exploratory Data Ana...
Step 4: Data Pre-Processing
Recency
i) Ranking the date colu...
ii) Assign a recency sc...
Frequency
Monetary Value
i) Find the total amount...
28°C

- **Step 3:** In the **Environment Variables** tab, click on **New**.
- **Step 4:** Enter the following values into **Variable name** and **Variable value**. Note that the version you install might be different from the one shown below, so copy and paste the path to your Spark directory.


Edit User Variable

Variable name: SPARK_HOME

Variable value: C:\spark\spark-3.3.0-bin-hadoop3

Step 5: Next, in the **Environment Variables** tab, click on **Path** and select **Edit**.

Step 6: Click on **New** and paste in the path to your Spark bin directory. Here is an example of what the bin directory looks like:

```
C:\spark\spark-3.3.0-bin-hadoop3\bin
```

Explain code Run code

POWERED BY datalab

Here is a guide on setting your environment variables if you use a Linux device, and here's one for MacOS.

Installing PySpark

Now that you have successfully installed Apache Spark and all other necessary prerequisites, open a Python file in your Jupyter Notebook and run the following lines of code in the first cell:

```
!pip install pyspark
```

Explain code Run code

POWERED BY datalab

Alternatively, you can follow along to this end-to-end PySpark installation guide to get the software installed on your device.

End-to-end Machine Learning PySpark Tutorial

Now that you have PySpark up and running, we will show you how to execute an end-to-end customer segmentation project using the library.

Customer segmentation is a marketing technique companies use to identify and group users who display similar characteristics. For instance, if you visit Starbucks only during the summer to purchase cold beverages, you can be segmented as a "seasonal shopper" and enticed with special promotions curated for the summer season.

Data scientists usually build unsupervised machine learning algorithms such as K-Means clustering or hierarchical clustering to perform customer segmentation. These models are great at identifying similar patterns between user groups that often go unnoticed by the human eye.

In this tutorial, we will use K-Means clustering to perform customer segmentation on the e-commerce dataset we downloaded earlier.

By the end of this tutorial, you will be familiar with the following concepts:

- Reading csv files with PySpark
- Exploratory Data Analysis with PySpark
- Grouping and sorting data
- Performing arithmetic operations
- Aggregating datasets
- Data Pre-Processing with PySpark
- Working with datetime values
- Type conversion
- Joining two data frames
- The rank() function
- PySpark Machine Learning
- Creating a feature vector
- Standardizing data
- Building a K-Means clustering model
- Interpreting the model

Run and edit the code from this tutorial online

Run code

Step 1: Creating a SparkSession

A SparkSession is an entry point into all functionality in Spark, and is required if you want to build a dataframe in PySpark. Run the following lines of code to initialize a SparkSession:

```
spark = SparkSession.builder.appName("Datacamp Pyspark Tutorial").config("s
```

Explain code Run code

POWERED BY datalab

Using the codes above, we built a spark session and set a name for the application. Then, the data was cached in off-heap memory to avoid storing it directly on disk, and the amount of memory was manually specified.

Step 2: Creating the DataFrame

We can now read the dataset we just downloaded:

```
df = spark.read.csv('datacamp_ecommerce.csv',header=True,escape="\"")
```

Explain code Run code

POWERED BY datalab

Note that we defined an escape character to avoid commas in the .csv file when parsing.

Let's take a look at the head of the DataFrame using the show() function:

```
df.show(5,0)
```

Explain code Run code

POWERED BY datalab

The DataFrame consists of 8 variables:

1. InvoiceNo : The unique identifier of each customer invoice.
2. StockCode : The unique identifier of each item in stock.
3. Description : The item purchased by the customer.
4. Quantity : The number of each item purchased by a customer in a single invoice.
5. InvoiceDate : The purchase date.
6. UnitPrice : Price of one unit of each item.
7. CustomerID : Unique identifier assigned to each user.
8. Country : The country from where the purchase was made.

Step 3: Exploratory data analysis

Now that we have seen the variables present in this dataset, let's perform some exploratory data analysis to further understand these data points:

1. Let's start by counting the number of rows in the DataFram :

```
df.count()  # Answer: 2,500
```

Explain code Run code

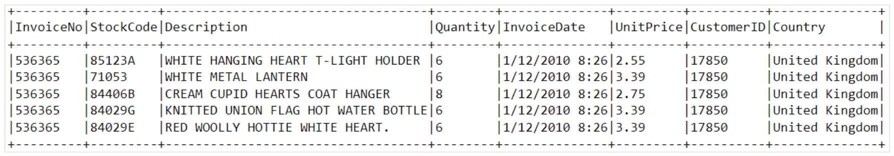
POWERED BY datalab

1. How many unique customers are present in the DataFrame?

```
df.select('CustomerID').distinct().count() # Answer: 95
```

Explain code Run code

POWERED BY datalab



```
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
|InvoiceNo|StockCode|Description                        |Quantity|InvoiceDate   |UnitPrice|CustomerID|Country       |
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
|536365   |85123A   |WHITE HANGING HEART T-LIGHT HOLDER |6       |1/12/2010 8:26|2.55     |17850     |United Kingdom|
|536365   |71053    |WHITE METAL LANTERN                |6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
|536365   |84406B   |CREAM CUPID HEARTS COAT HANGER     |8       |1/12/2010 8:26|2.75     |17850     |United Kingdom|
|536365   |84029G   |KNITTED UNION FLAG HOT WATER BOTTLE|6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
|536365   |84029E   |RED WOOLLY HOTTIE WHITE HEART.     |6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
```

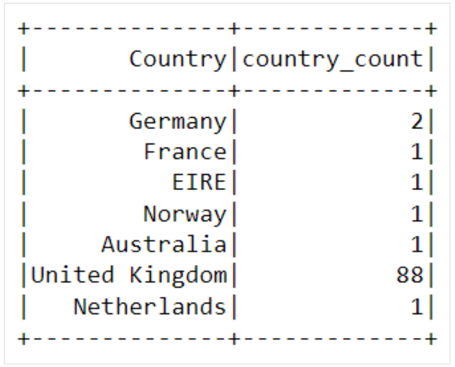
1. What country do most purchases come from?

To find the country from which most purchases are made, we need to use the groupBy() clause in PySpark:

```
from pyspark.sql.functions import *
from pyspark.sql.types import *

df.groupBy('Country').agg(countDistinct('CustomerID').alias('country_count')).sho
```

POWERED BY datalab

The following table will be rendered after running the codes above:



```
+--------------+-------------+
|       Country|country_count|
+--------------+-------------+
|       Germany|            2|
|        France|            1|
|          EIRE|            1|
|        Norway|            1|
|     Australia|            1|
|United Kingdom|           88|
|   Netherlands|            1|
+--------------+-------------+
```

Almost all the purchases on the platform were made from the United Kingdom, and only a handful were made from countries like Germany, Australia, and France.

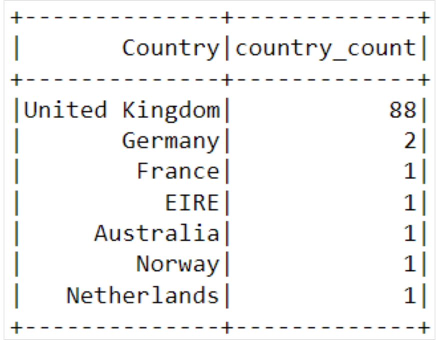
Notice that the data in the table above isn't presented in the order of purchases. To sort this table, we can include the orderBy() clause:

```
df.groupBy('Country').agg(countDistinct('CustomerID').alias('country_count' rd
```

Explain code Run code

POWERED BY datalab

The output displayed is now sorted in descending order:



```
+--------------+-------------+
|       Country|country_count|
+--------------+-------------+
|United Kingdom|           88|
|       Germany|            2|
|        France|            1|
|          EIRE|            1|
|     Australia|            1|
|        Norway|            1|
|   Netherlands|            1|
+--------------+-------------+
```

1. When was the most recent purchase made by a customer on the e-commerce platform?

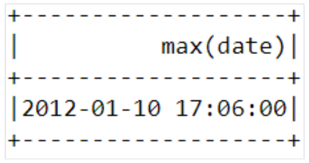
To find when the latest purchase was made on the platform, we need to convert the InvoiceDate column into a timestamp format and use the max() function in Pyspark:

```
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")
df = df.withColumn('date',to_timestamp("InvoiceDate", 'yy/MM/dd HH:mm'))
df.select(max("date")).show()
```

Explain code   Run code

POWERED BY datalab

You should see the following table appear after running the code above:



```
+-------------------+
|          max(date)|
+-------------------+
|2012-01-10 17:06:00|
+-------------------+
```

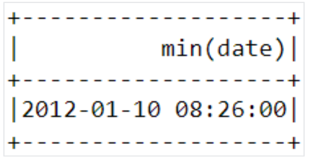
1. When was the earliest purchase made by a customer on the e-commerce platform?

Similar to what we did above, the min() function can be used to find the earliest purchase date and time:

```
df.select(min("date")).show()
```

Explain code   Run code

POWERED BY datalab



```
+-------------------+
|          min(date)|
+-------------------+
|2012-01-10 08:26:00|
+-------------------+
```

Notice that the most recent and earliest purchases were made on the same day just a few hours apart. This means that the dataset we downloaded contains information of only purchases made on a single day.

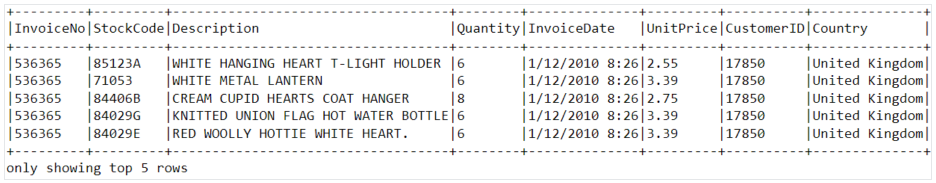
Step 4: Data pre-processing

Now that we have analyzed the dataset and have a better understanding of each data point, we need to prepare the data to feed into the machine learning algorithm.

Let's take a look at the head of the data frame once again to understand how the pre-processing will be done:

```
df.show(5,0)
```

Explain code Run code POWERED BY datalab



```
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
|InvoiceNo|StockCode|Description                        |Quantity|InvoiceDate   |UnitPrice|CustomerID|Country       |
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
|536365   |85123A   |WHITE HANGING HEART T-LIGHT HOLDER |6       |1/12/2010 8:26|2.55     |17850     |United Kingdom|
|536365   |71053    |WHITE METAL LANTERN                |6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
|536365   |84406B   |CREAM CUPID HEARTS COAT HANGER     |8       |1/12/2010 8:26|2.75     |17850     |United Kingdom|
|536365   |84029G   |KNITTED UNION FLAG HOT WATER BOTTLE|6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
|536365   |84029E   |RED WOOLLY HOTTIE WHITE HEART.     |6       |1/12/2010 8:26|3.39     |17850     |United Kingdom|
+---------+---------+-----------------------------------+--------+--------------+---------+----------+--------------+
only showing top 5 rows
```

From the dataset above, we need to create multiple customer segments based on each user's purchase behavior.

The variables in this dataset are in a format that cannot be easily ingested into the customer segmentation model. These features individually do not tell us much about customer purchase behavior.

Due to this, we will use the existing variables to derive three new informative features - recency, frequency, and monetary value (RFM).

RFM is commonly used in marketing to evaluate a client's value based on their:

1. **Recency**: How recently has each customer made a purchase?
2. **Frequency**: How often have they bought something?
3. **Monetary Value**: How much money do they spend on average when making purchases?

We will now preprocess the data frame to create the above variables.

Recency

First, let's calculate the value of recency - the latest date and time a purchase was made on the platform. This can be achieved in two steps:

i) Assign a recency score to each customer

We will subtract every date in the data frame from the earliest date. This will tell us how recently a customer was seen in the data frame. A value of 0 indicates the lowest recency, as it will be assigned to the person who was seen making a purchase on the earliest date.

```
df = df.withColumn("from_date", lit("12/1/10 08:26"))
df = df.withColumn('from_date',to_timestamp("from_date", 'yy/MM/dd HH:mm'))
```

```
df2 = df.withColumn('from_date',to_timestamp(col('from_date'))).withColumn('recen
```

POWERED BY datalab

ii) Select the most recent purchase

One customer can make multiple purchases at different times. We need to select only the last time they were seen buying a product, as this is indicative of when the most recent purchase was made:

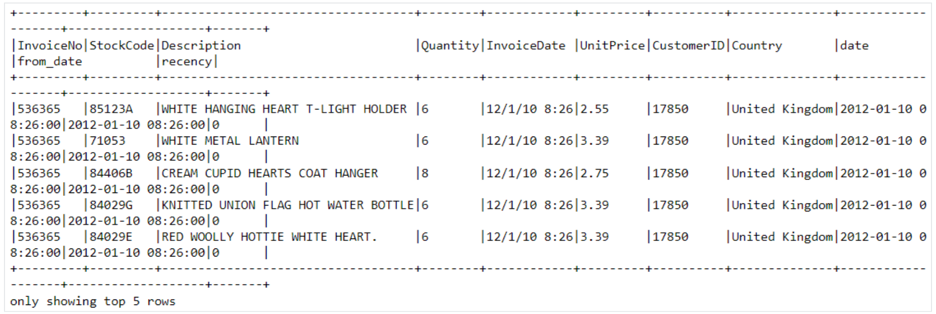
```
df2 = df2.join(df2.groupBy('CustomerID').agg(max('recency').alias('recency')  n=
```

Explain code Run code POWERED BY datalab

Let's look at the head of the new data frame. It now has a variable called "recency" appended to it:

```
df2.show(5,0)
```

Explain code Run code POWERED BY datalab



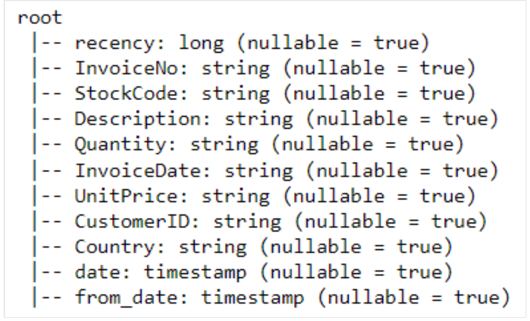
```
+---------+---------+-----------------------------------+--------+------------+--------+----------+--------------+------------
-------+-------------------+-------+
|InvoiceNo|StockCode|Description                        |Quantity|InvoiceDate |UnitPrice|CustomerID|Country       |date
|from_date          |recency|
+---------+---------+-----------------------------------+--------+------------+--------+----------+--------------+------------
-------+-------------------+-------+
|536365   |85123A   |WHITE HANGING HEART T-LIGHT HOLDER |6       |12/1/10 8:26|2.55     |17850     |United Kingdom|2012-01-10 0
8:26:00|2012-01-10 08:26:00|0      |
|536365   |71053    |WHITE METAL LANTERN                |6       |12/1/10 8:26|3.39     |17850     |United Kingdom|2012-01-10 0
8:26:00|2012-01-10 08:26:00|0      |
|536365   |84406B   |CREAM CUPID HEARTS COAT HANGER     |8       |12/1/10 8:26|2.75     |17850     |United Kingdom|2012-01-10 0
8:26:00|2012-01-10 08:26:00|0      |
|536365   |84029G   |KNITTED UNION FLAG HOT WATER BOTTLE|6       |12/1/10 8:26|3.39     |17850     |United Kingdom|2012-01-10 0
8:26:00|2012-01-10 08:26:00|0      |
|536365   |84029E   |RED WOOLLY HOTTIE WHITE HEART.     |6       |12/1/10 8:26|3.39     |17850     |United Kingdom|2012-01-10 0
8:26:00|2012-01-10 08:26:00|0      |
+---------+---------+-----------------------------------+--------+------------+--------+----------+--------------+------------
-------+-------------------+-------+
only showing top 5 rows
```

An easier way to view all the variables present in a PySpark DataFrame is to use its printSchema() function. This is the equivalent of the info() function in Pandas:

```
df2.printSchema()
```

Explain code Run code POWERED BY datalab

The output rendered should look like this:


```
root
 |-- recency: long (nullable = true)
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: string (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- CustomerID: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- date: timestamp (nullable = true)
 |-- from_date: timestamp (nullable = true)
```

Frequency

Let's now calculate the value of frequency - how often a customer buys something on the platform. To do this, we just need to group by each CustomerID and count the number of items they purchased:

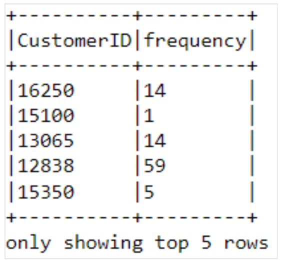
```
df_freq = df2.groupBy('CustomerID').agg(count('InvoiceDate').alias('frequen  )
```

Explain code Run code POWERED BY datalab

Look at the head of this new DataFrame we just created:

```
df_freq.show(5,0)
```

Explain code Run code POWERED BY datalab


```
+----------+---------+
|CustomerID|frequency|
+----------+---------+
|16250     |14       |
|15100     |1        |
|13065     |14       |
|12838     |59       |
|15350     |5        |
+----------+---------+
only showing top 5 rows
```

There is a frequency value appended to each customer in the DataFrame. This new DataFrame only has two columns, and we need to join it with the previous one:

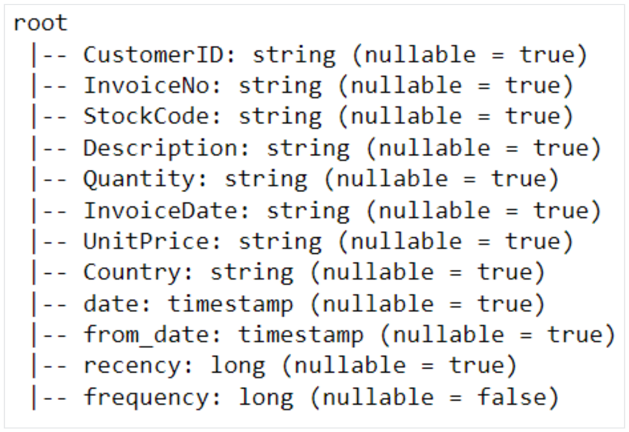
```
df3 = df2.join(df_freq,on='CustomerID',how='inner')
```

Explain code Run code POWERED BY datalab

Let's print the schema of this DataFrame:

```
df3.printSchema()
```

Explain code Run code POWERED BY datalab



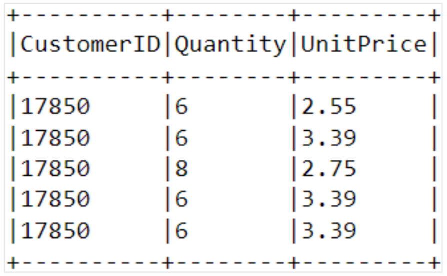
```
root
 |-- CustomerID: string (nullable = true)
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: string (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- date: timestamp (nullable = true)
 |-- from_date: timestamp (nullable = true)
 |-- recency: long (nullable = true)
 |-- frequency: long (nullable = false)
```

Monetary Value

Finally, let's calculate monetary value - the total amount spent by each customer in the DataFrame. There are two steps to achieving this:

i) Find the total amount spent in each purchase:

Each CustomerID comes with variables called Quantity and UnitPrice for a single purchase:



```
+----------+--------+---------+
|CustomerID|Quantity|UnitPrice|
+----------+--------+---------+
|17850     |6       |2.55     |
|17850     |6       |3.39     |
|17850     |8       |2.75     |
|17850     |6       |3.39     |
|17850     |6       |3.39     |
+----------+--------+---------+
```

To get the total amount spent by each customer in one purchase, we need to multiply Quantity with UnitPrice:

```
m_val = df3.withColumn('TotalAmount',col("Quantity") * col("UnitPrice"))
```

Explain code Run code POWERED BY datalab

ii) Find the total amount spent by each customer:

To find the total amount spent by each customer overall, we just need to group by the CustomerID column and sum the total amount spent:

```
m_val = m_val.groupBy('CustomerID').agg(sum('TotalAmount').alias('monetary_[illegible]e'
```

Explain code Run code POWERED BY datalab

Merge this DataFrame with the all the other variables:

```
finaldf = m_val.join(df3,on='CustomerID',how='inner')
```

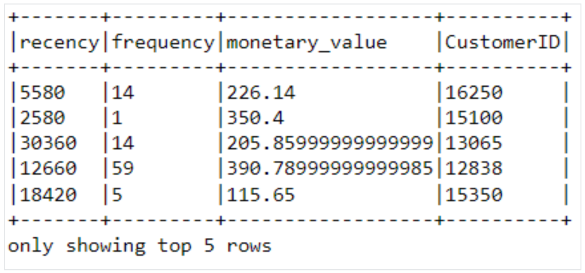
Explain code Run code POWERED BY datalab

Now that we have created all the necessary variables to build the model, run the following lines of code to select only the required columns and drop duplicate rows from the DataFrame:

```
finaldf = finaldf.select(['recency','frequency','monetary_value','CustomerI .d
```

Explain code Run code POWERED BY datalab

Look at the head of the final DataFrame to ensure that the pre-processing has been done accurately:



```
+-------+---------+------------------+----------+
|recency|frequency|monetary_value    |CustomerID|
+-------+---------+------------------+----------+
|5580   |14       |226.14            |16250     |
|2580   |1        |350.4             |15100     |
|30360  |14       |205.85999999999999|13065     |
|12660  |59       |390.78999999999985|12838     |
|18420  |5        |115.65            |15350     |
+-------+---------+------------------+----------+
only showing top 5 rows
```

Standardization

Before building the customer segmentation model, let's standardize the DataFrame to ensure that all the variables are around the same scale:

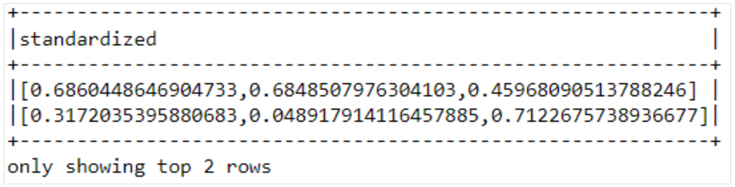
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler

assemble=VectorAssembler(inputCols=[
    'recency','frequency','monetary_value'
], outputCol='features')

assembled_data=assemble.transform(finaldf)

scale=StandardScaler(inputCol='features',outputCol='standardized')
data_scale=scale.fit(assembled_data)
data_scale_output=data_scale.transform(assembled_data)
```

Explain code Run code POWERED BY datalab

Run the following lines of code to see what the standardized feature vector looks like:

```
data_scale_output.select('standardized').show(2,truncate=False)
```

Explain code Run code POWERED BY datalab



```
+-------------------------------------------------------------+
|standardized                                                 |
+-------------------------------------------------------------+
|[0.6860448646904733,0.6848507976304103,0.45968090513788246] |
|[0.3172035395880683,0.048917914116457885,0.7122675738936677]|
+-------------------------------------------------------------+
only showing top 2 rows
```

These are the scaled features that will be fed into the clustering algorithm.

If you'd like to learn more about data preparation with PySpark, take this feature engineering course on Datacamp.

Step 5: Building the machine learning model

Now that we have completed all the data analysis and preparation, let's build the K-Means clustering model.

The algorithm will be created using PySpark's machine learning API.

i) Finding the number of clusters to use

When building a K-Means clustering model, we first need to determine the number of clusters or groups we want the algorithm to return. If we decide on three clusters, for instance, then we will have three customer segments.

The most popular technique used to decide on how many clusters to use in K-Means is called the "elbow-method."

This is done simply running the K-Means algorithm for a wide range of clusters and visualizing the model results for each cluster. The plot will have an inflection point that looks like an elbow, and we just pick the number of clusters at this point.

Read this Datacamp K-Means clustering tutorial to learn more about how the algorithm works.

Let's run the following lines of code to build a K-Means clustering algorithm from 2 to 10 clusters:

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
import numpy as np

cost = np.zeros(10)

evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='standard
for i in range(2,10):
    KMeans_algo=KMeans(featuresCol='standardized', k=i)
    KMeans_fit=KMeans_algo.fit(data_scale_output)
    output=KMeans_fit.transform(data_scale_output)
    cost[i] = KMeans_fit.summary.trainingCost
```

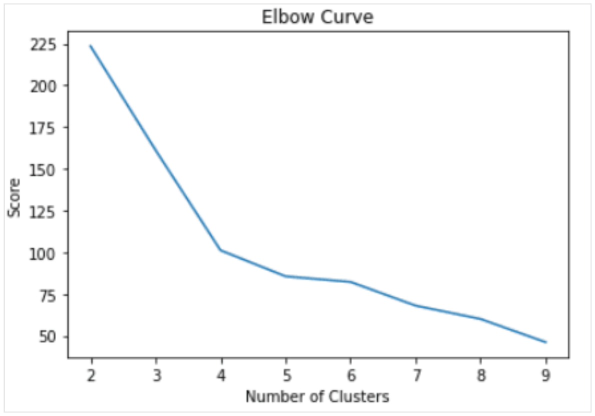
Explain code Run code POWERED BY datalab

With the codes above, we have successfully built and evaluated a K-Means clustering model with 2 to 10 clusters. The results have been placed in an array, and can now be visualized in a line chart:

```
import pandas as pd
import pylab as pl
df_cost = pd.DataFrame(cost[2:])
df_cost.columns = ["cost"]
new_col = range(2,10)
df_cost.insert(0, 'cluster', new_col)
pl.plot(df_cost.cluster, df_cost.cost)
pl.xlabel('Number of Clusters')
pl.ylabel('Score')
pl.title('Elbow Curve')
pl.show()
```

Explain code Run code POWERED BY datalab

The codes above will render the following chart:



ii) Building the K-Means Clustering Model

From the plot above, we can see that there is an inflection point that looks like an elbow at four. Due to this, we will proceed to build the K-Means algorithm with four clusters:

```
KMeans_algo=KMeans(featuresCol='standardized', k=4)
KMeans_fit=KMeans_algo.fit(data_scale_output)
```

Explain code Run code POWERED BY datalab

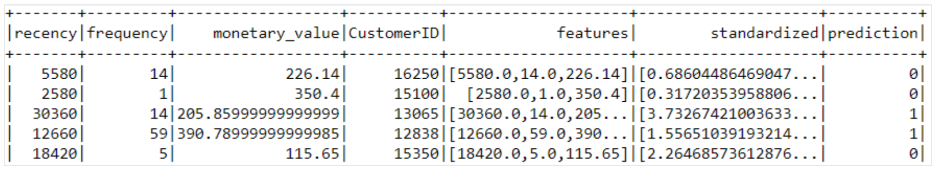
3) Making Predictions

Let's use the model we created to assign clusters to each customer in the dataset:

```
preds=KMeans_fit.transform(data_scale_output)

preds.show(5,0)
```

Explain code Run code POWERED BY datalab

Notice that there is a "prediction" column in this DataFrame that tells us which cluster each CustomerID belongs to:



```
+-------+---------+------------------+----------+-------------------+--------------------+----------+
|recency|frequency|     monetary_value|CustomerID|           features|        standardized|prediction|
+-------+---------+------------------+----------+-------------------+--------------------+----------+
|   5580|       14|             226.14|     16250|[5580.0,14.0,226.14]|[0.68604486469047...|         0|
|   2580|        1|              350.4|     15100|  [2580.0,1.0,350.4]|[0.31720353958806...|         0|
|  30360|       14|205.85999999999999|     13065|[30360.0,14.0,205...|[3.73267421003633...|         1|
|  12660|       59|390.78999999999985|     12838|[12660.0,59.0,390...|[1.55651039193214...|         1|
|  18420|        5|             115.65|     15350|[18420.0,5.0,115.65]|[2.26468573612876...|         0|
```

Step 6: Cluster Analysis

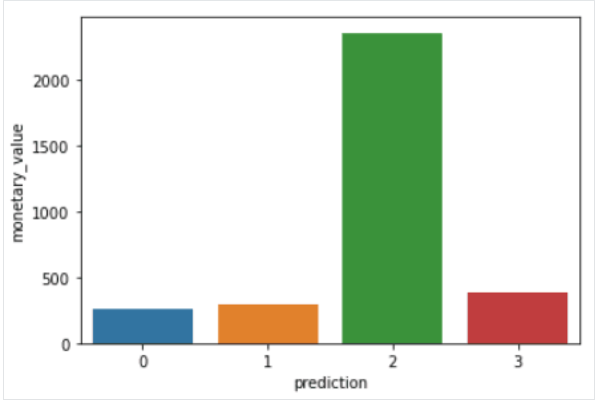
The final step in this entire tutorial is to analyze the customer segments we just built.

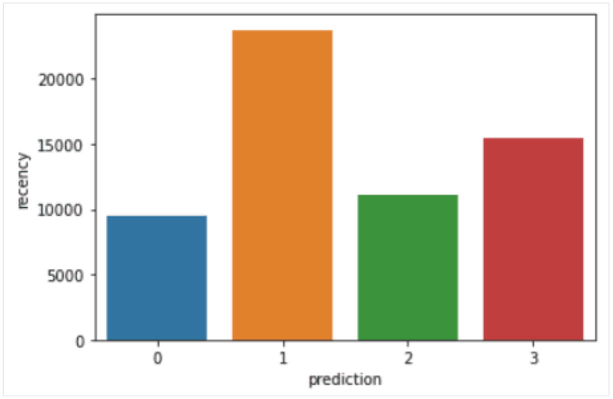
Run the following lines of code to visualize the recency, frequency, and monetary value of each CustomerID in theDataFrame:

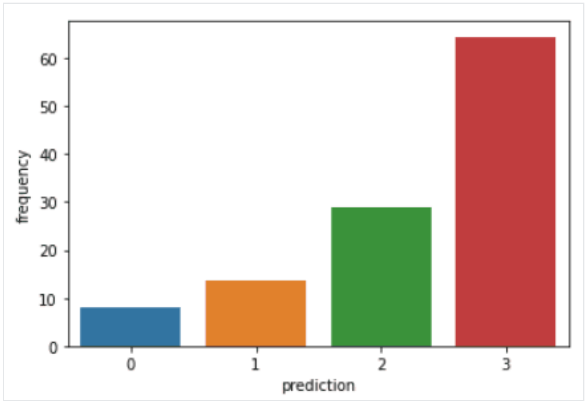
```
import matplotlib.pyplot as plt
import seaborn as sns

df_viz = preds.select('recency','frequency','monetary_value','prediction')
df_viz = df_viz.toPandas()
avg_df = df_viz.groupby(['prediction'], as_index=False).mean()

list1 = ['recency','frequency','monetary_value']

for i in list1:
    sns.barplot(x='prediction',y=str(i),data=avg_df)
    plt.show()
```

Explain code Run code POWERED BY datalab

The codes above will render the following plots:







Here is an overview of characteristics displayed by customers in each cluster:

- **Cluster 0**: Customers in this segment display low recency, frequency, and monetary value. They rarely shop on the platform and are low potential customers who are likely to stop doing business with the ecommerce company.
- **Cluster 1**: Users in this cluster display high recency but haven't been seen spending much on the platform. They also don't visit the site often. This indicates that they might be newer customers who have just started doing business with the company.
- **Cluster 2**: Customers in this segment display medium recency and frequency and spend a lot of money on the platform. This indicates that they tend to buy high-value items or

make bulk purchases.

- **Cluster 3**: The final segment comprises users who display high recency and make frequent purchases on the platform. However, they don't spend much on the platform, which might mean that they tend to select cheaper items in each purchase.

To go beyond the predictive modeling concepts covered in this course, you can take the **Machine Learning with PySpark** course on Datacamp.

Learning PySpark From Scratch - Next Steps:

If you managed to follow along with this entire PySpark tutorial, congratulations! You have now successfully installed PySpark onto your local device, analyzed an e-commerce dataset, and built a machine learning algorithm using the framework.

One caveat of the analysis above is that it was conducted with 2,500 rows of ecommerce data collected on a single day. The outcome of this analysis can be solidified if we had a larger amount of data to work with, as techniques like RFM modeling are usually applied onto months of historical data.

However, you can take the principles learned in this article and apply them to a wide variety of larger datasets in the unsupervised machine learning space.

Check out this **cheat sheet** by Datacamp to learn more about PySpark's syntax and its modules.

Finally, if you'd like to go beyond the concepts covered in this tutorial and learn the fundamentals of programming with PySpark, you can take the **Big Data with PySpark** learning track on Datacamp. This track contains a series of courses that will teach you to do the following with PySpark:

- Data Management, Analysis, and Pre-processing
- Building and Tuning Machine Learning Pipelines
- Big Data Analysis
- Feature Engineering
- Building Recommendation Engines

Upskilling Your Team in PySpark

As you've seen throughout this tutorial, mastering PySpark and distributed data processing is essential for handling large-scale datasets that are increasingly common in the modern world. For companies managing terabytes or even petabytes of data, having a team proficient in PySpark can significantly enhance your ability to derive actionable insights and maintain a competitive edge.

However, staying updated with the latest technologies and best practices can be challenging, especially for teams working in fast-paced environments. This is where **DataCamp for Business** can make a difference. DataCamp for Business provides your team with the tools and training they need to stay on the cutting edge of data science and engineering.