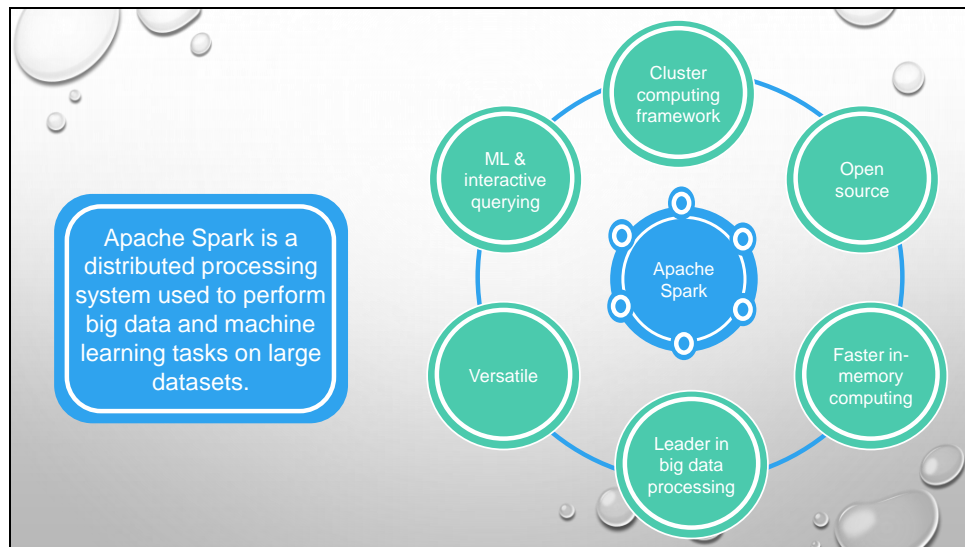


Apache Spark

CONTENTS

- APACHE SPARK
- SPARK ARCHITECTURE
- RDD
- DATAFRAME API
- SPARKSQL
- EDA WITH PYSPARK
- PREDICTIVE ANALYSIS WITH SPARK MLIB



What is Apache Spark?

- Apache Spark is a lightning-fast cluster computing framework designed for real-time processing.
- Spark is an open-source project from Apache Software Foundation.
- Spark overcomes the limitations of Hadoop MapReduce, and it extends the MapReduce model to be efficiently used for data processing.
- Spark is a market leader for big data processing.
- It is widely used across organizations in many ways.
- It has surpassed Hadoop by running 100 times faster in memory and 10 times faster on disks.

Spark is a platform for cluster computing. Spark lets you spread data and computations over *clusters* with multiple *nodes* (think of each node as a separate computer). Splitting up your data makes it easier to work with very large datasets because each node only works with a small amount of data.

As each node works on its own subset of the total data, it also carries out a part of the total calculations required, so that both data processing and computation are performed *in parallel* over the nodes in the cluster. It is a fact that parallel computation can make certain types of programming tasks much faster.

However, with greater computing power comes greater complexity.

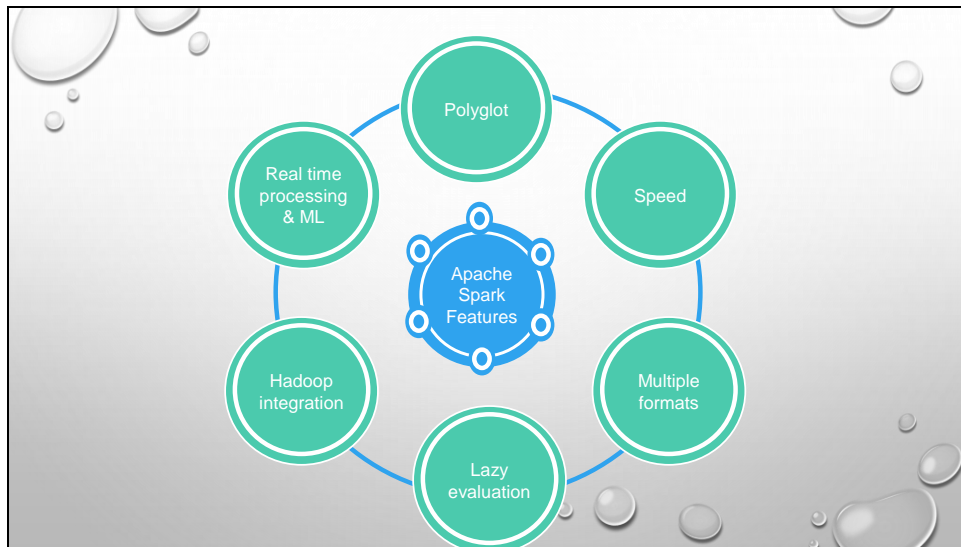
Deciding whether or not Spark is the best solution for your problem takes some experience, but you can consider questions like:

- Is my data too big to work with on a single machine?
- Can my calculations be easily parallelized?

Career paths with Apache Spark

- *Most technology-based companies across the globe have moved toward Apache Spark due to functionalities such as Machine Learning and interactive querying.*

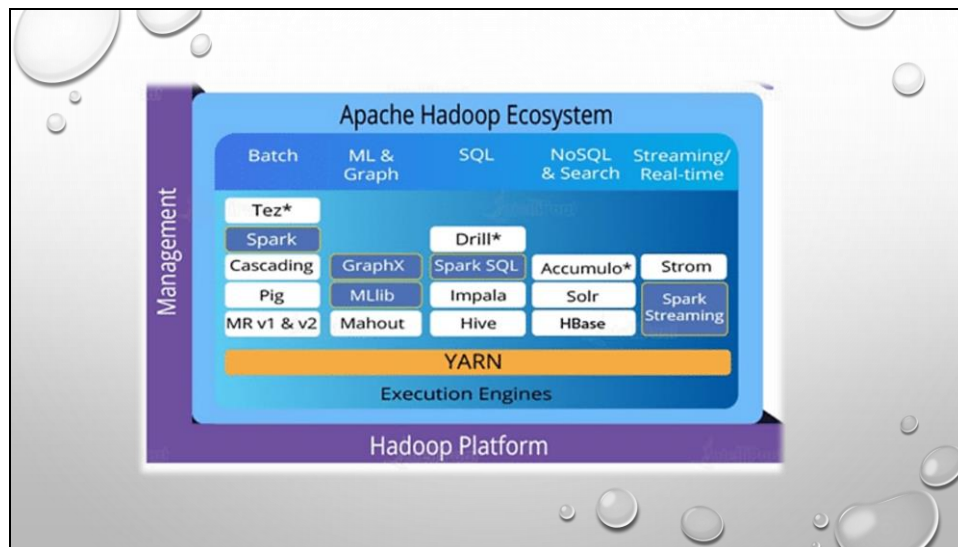
- *Industry leaders such as Amazon, Huawei, and IBM have already adopted Apache Spark.*
- *The firms that were initially based on Hadoop, such as Hortonworks, Cloudera, and MapR, have also moved to Apache Spark.*
- *Big Data Hadoop professionals surely need to learn Apache Spark since it is the next most important technology in Hadoop data processing. Moreover, even ETL professionals, SQL professionals, and Project Managers can gain immensely if they master Apache Spark. Finally, Data Scientists also need to gain in-depth knowledge of Spark to excel in their careers.*
- *Spark can be extensively deployed in Machine Learning scenarios.*
- *Data Scientists are expected to work in the Machine Learning domain, and hence, they are the right candidates for Apache Spark training.*
- *Those who have an intrinsic desire to learn the latest emerging technologies can also learn Spark through this Apache Spark tutorial.*



Features of Apache Spark

Apache Spark has the following features:

- Polyglot – Spark code can be written in Python, R, Java, and Scala. There are shells provided for Scala and Python. These can be accessed from the installed directory.
- Speed – Spark can process large data that is 100 times faster than Hadoop MapReduce. This is possible because of controlled partitioning. Spark can manage data in partitions which help parallelize distributed data processing without excess network traffic.
- Multiple Formats – Spark supports multiple data sources. This makes access easier with the help of the Data Source API.
- Lazy Evaluation – Spark can delay the evaluation unless it's absolutely necessary. This contributes majorly to its high speed.
- Real-Time Computation – Spark can compute in real time. Its latency is low as it can compute in memory. Spark is highly scalable with users running clusters with thousands of nodes.
- Hadoop Integration – Spark can be integrated with Hadoop. It helps all the Big Data Engineers who would have started their career with Hadoop.
- Machine Learning – Spark has a Machine Learning component called MLlib. It comes in handy for processing big data. You don't need to use different tools for processing and machine learning. There are numerous opportunities under Machine Learning.



Apache Spark in the Hadoop ecosystem

- Spark is designed for the enhancement of the Hadoop stack.
- Spark can perform read/write data operations with HDFS, HBase, or Amazon S3.
- Hadoop users can use Apache Spark to enhance the computational capabilities of their Hadoop MapReduce system.
- Apache Spark can be used with Hadoop or Hadoop YARN together.
- It can be deployed on Hadoop in three ways: Standalone, YARN, and SIMR.
- **Standalone Deployment**
 - Spark provides a simple standalone deployment mode. This allows Spark to allocate all resources or a subset of resources in a Hadoop cluster. We can also run Spark in parallel with Hadoop MapReduce. Spark jobs can be deployed easily using HDFS data. Spark's simple architecture makes it a preferred choice for Hadoop users.
- **Hadoop YARN Deployment**
 - Apache Spark contains some configuration files for the Hadoop cluster. These config files can easily read/write to HDFS and YARN Resource Manager. We can easily run Spark on YARN without any pre-installation.
- **Spark in MapReduce (SIMR)**
 - We can easily deploy Spark on MapReduce clusters as well. It will help us start experimenting with Spark to explore more.



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.

What is PySpark used for?

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.

Why PySpark?

Companies that collect terabytes of data will have a big data framework like Apache Spark in place.

To work with these large-scale datasets, knowledge of Python and R frameworks alone will not suffice.

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.

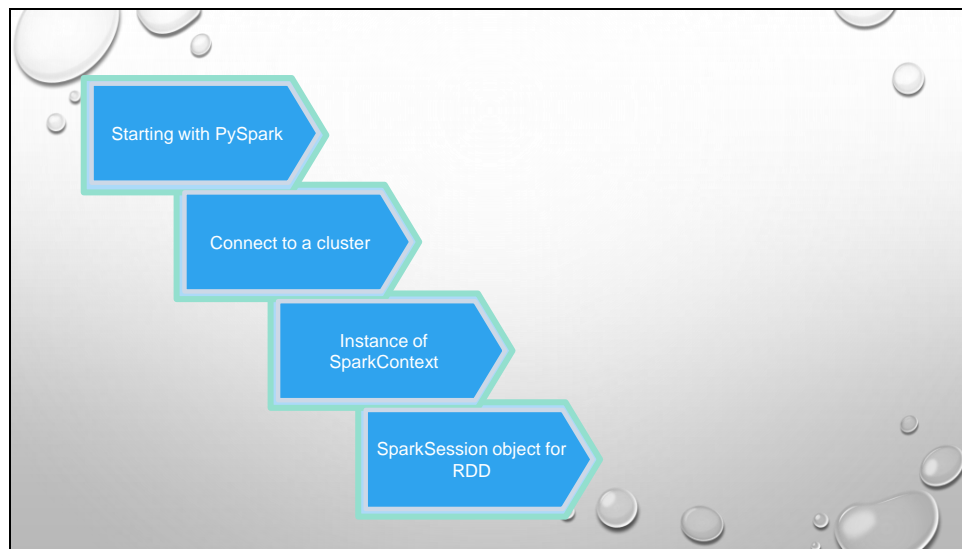
If you had over petabytes of data to process, for instance, Pandas and Dask would fail but PySpark would be able to handle it easily.

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.



Using Spark in Python

- The first step in using Spark is connecting to a cluster.
- In practice, the cluster will be hosted on a remote machine that's connected to all other nodes. There will be one computer, called the *master* that manages splitting up the data and the computations. The master is connected to the rest of the computers in the cluster, which are called *worker*. The master sends the workers data and calculations to run, and they send their results back to the master.
- When you're just getting started with Spark it's simpler to just run a cluster locally.
- Creating the connection is as simple as creating an instance of the SparkContext class. The class constructor takes a few optional arguments that allow you to specify the attributes of the cluster you're connecting to.

Using DataFrames

- Spark's core data structure is the Resilient Distributed Dataset (RDD).
- This is a low level object that lets Spark work its magic by splitting data across multiple nodes in the cluster.
- The Spark DataFrame was designed to behave a lot like a SQL table (a table with variables in the columns and observations in the rows). Not only are they easier to understand, DataFrames are also more optimized for complicated operations than RDDs.
- When you start modifying and combining columns and rows of data, there are many ways to arrive at the same result, but some often take much longer than others.
- When using RDDs, it's up to the data scientist to figure out the right way to optimize the query, but the DataFrame implementation has much of this optimization built in!
- To start working with Spark DataFrames, you first have to create a SparkSession object from your SparkContext. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

How to install PySpark

Pre-requisites:

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.

Dataset

We will be using Datacamp's [e-commerce dataset](#) for all the analysis in this tutorial, so make sure to have it downloaded. We'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.

Installation Guide

Now that you have all the prerequisites set up, you can proceed to install Apache Spark and PySpark.

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:
2. Choose a package type:
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 path
```

OpenAI

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

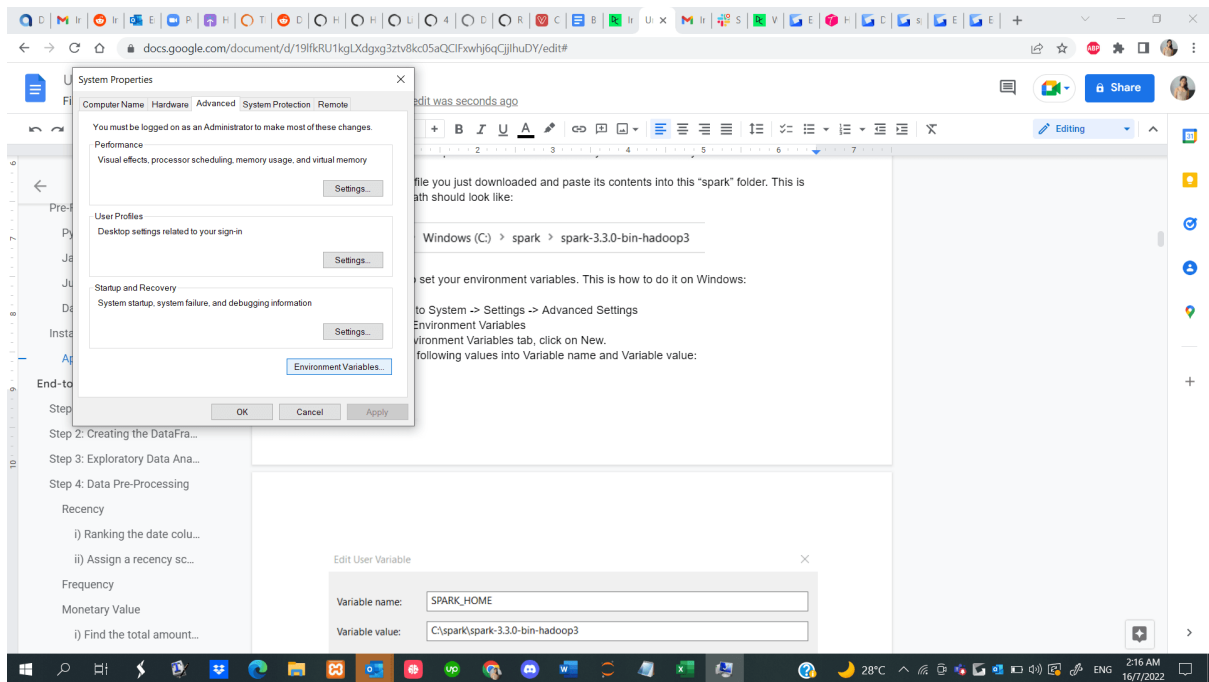
```
setx PATH "C:\spark\spark-3.3.0-bin-hadoop3\bin"
```

OpenAI

Method 2: Changing Environment Variables Manually

Step 1: Navigate to Start -> System -> Settings -> Advanced Settings

Step 2: Click on Environment Variables



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

Browse Directory...

Browse File...

OK

Cancel

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

OpenAI

[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
```

OpenAI

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 dataframes
- The rank() function
- PySpark Machine Learning
- Creating a feature vector
- Standardizing data

- Building a K-Means clustering model
- Interpreting the model

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("spark.memory.offHeap.enabled","true").config("spark.memory.offHeap.size","10g").getOrCreate()
```

OpenAI

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="\")
```

OpenAI

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)
```

OpenAI

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

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 dataframe:

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

OpenAI

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

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

OpenAI

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')).show()
```

OpenAI

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')).orderBy(desc('country_count')).show()
```

OpenAI

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()
```

OpenAI

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()
```

OpenAI

```
+-----+
|                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 dataframe once again to understand how the pre-processing will be done:

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

OpenAI

```
+-----+-----+-----+-----+-----+-----+-----+
|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 dataframe 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 dataframe from the earliest date. This will tell us how recently a customer was seen in the dataframe. 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('recency',col("date").cast("long") - col('from_date').cast("long"))
```

OpenAI

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')),on='recency',how='leftsemi')
```

OpenAI

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

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

OpenAI

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 08:26:00
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	2012-01-10 08:26:00
536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/10 8:26	2.75	17850	United Kingdom	2012-01-10 08:26:00
536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/10 8:26	3.39	17850	United Kingdom	2012-01-10 08:26:00
536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/10 8:26	3.39	17850	United Kingdom	2012-01-10 08:26:00

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()
```

OpenAI

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 bought something on the platform. To do this, we just need to group by each customer ID and count the number of items they purchased:

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

OpenAI

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

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

OpenAI

```
+-----+-----+
|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')
```

OpenAI

Let's print the schema of this dataframe:

```
df3.printSchema()
```

OpenAI

```
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"))
```

OpenAI

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_value'))
```

OpenAI

Merge this dataframe with the all the other variables:

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

OpenAI

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','CustomerID']).distinct()
```

OpenAI

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)
```

OpenAI

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

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

OpenAI

```
+-----+
| 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.

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.

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='standardized',metricName='silhouette', distanceMeasure='squaredEuclidean')

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

```

OpenAI

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')

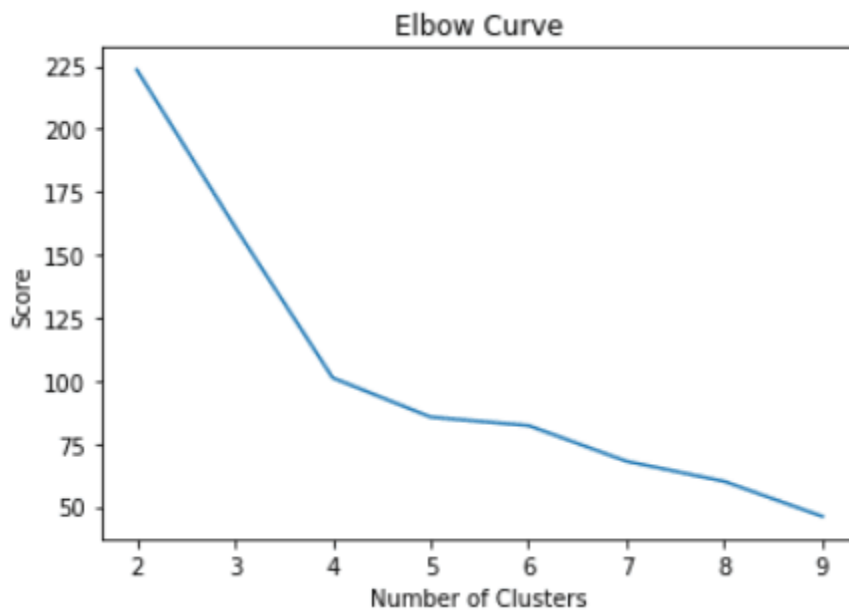
pl.title('Elbow Curve')

pl.show()

```

OpenAI

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)
```

OpenAI

iii) 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)
```

OpenAI

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 the dataframe:

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

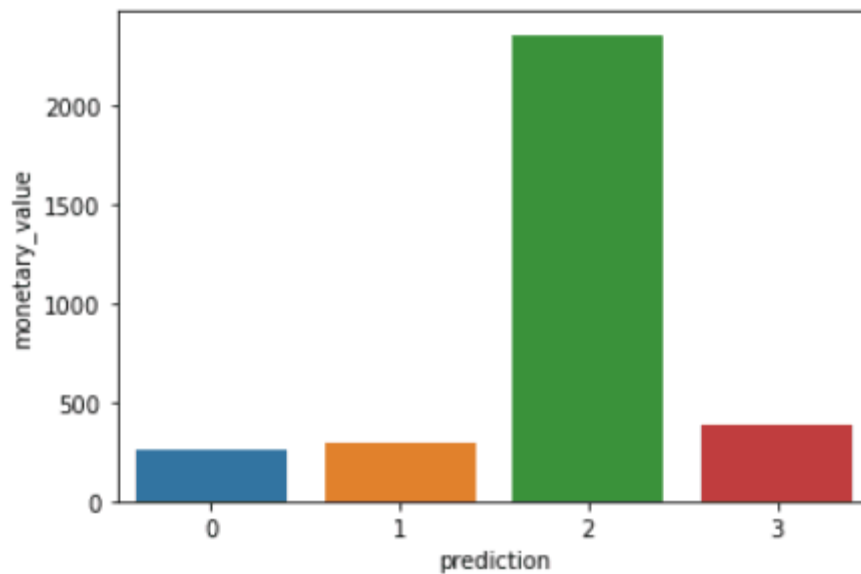
df_viz = preds.select('recency','frequency','monetary_value','prediction')
df_viz = df_viz.toPandas()
avg_df = df_viz.groupby(['prediction'], as_index=False).mean()

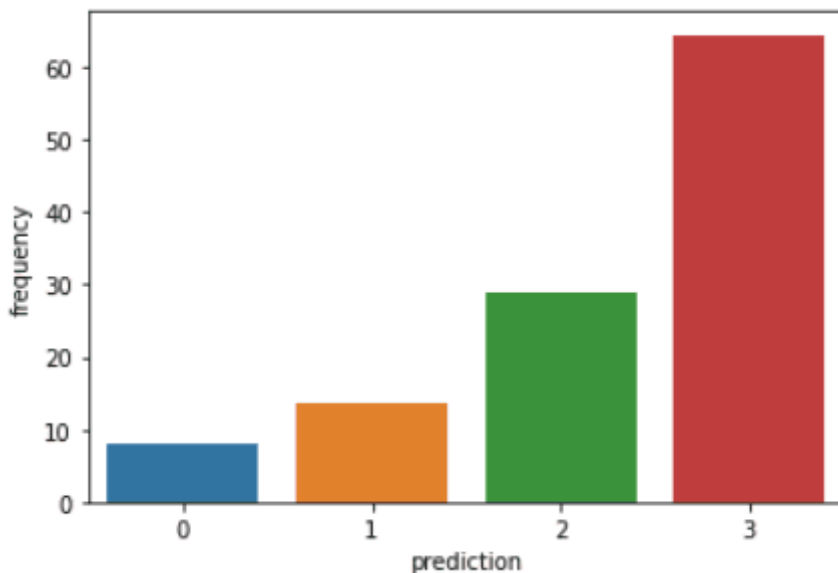
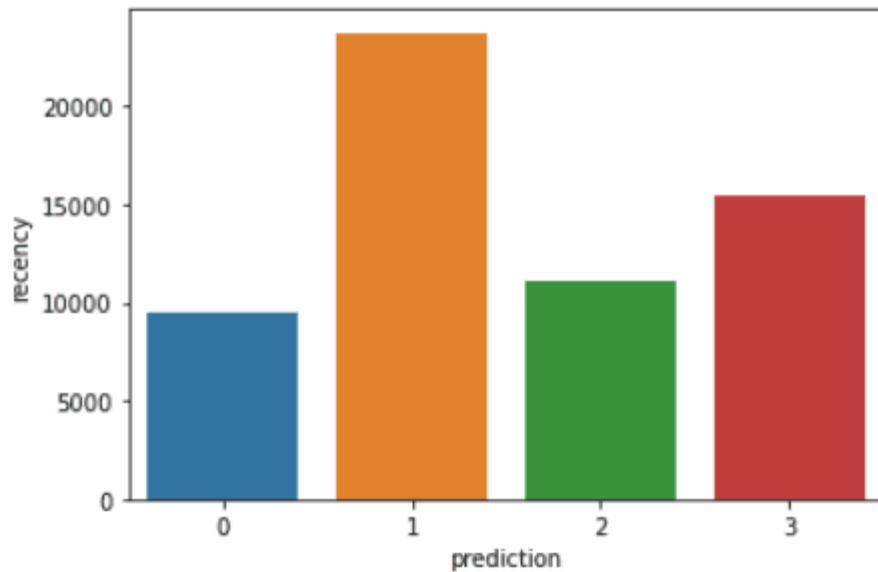
list1 = ['recency','frequency','monetary_value']

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

OpenAI

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.