**Big Data Analytics - CSE6034 Digital-Assignment - 1:**

**Assignment Question :**

**Implement K-means clustering algorithm in Spark by utilizing transformations and actions. Explain your understanding how this algorithm executes in a spark cluster. Take any sample data set (twitter/e-commerce..etc**

**Submitted By : 20MAI0001 - NIHARIKA MAITRA**

**Git Repo link :**

Link for the Input dataset used in this Assignment :

<https://github.com/Niharika-20-MAI-01/Winter-2021-Big-Data-Analytics-LabCSE6034-Niharika-20MAI01/blob/main/CC%20GENERAL.csv>

<https://github.com/Niharika-20-MAI-01/Winter-2021-Big-Data-Analytics-LabCSE6034-Niharika-20MAI01>

**Implementation of K-means Clustering Algorithm in Spark by utilizing Transformations and Actions, on a Dataset**

**Solution :-**

**A Brief Introductory Note on the Input Dataset :**

**i) The Input dataset used in this Assignment for the implementation of the K-means Clustering Algorithm on it is : ‘CC General.csv’.**

**ii) This dataset : ‘CC General.csv’ , consists of 9K active credit cardholders history / Record for over 6 months of each of their transactions and account attributes.**

**iii) The main aim of this Assignment is to develop a Customer Segmentation Model by the implementation of the K-means Clustering Algorithm so that it can be used in the marketing strategy.**

**Spark Program Code snippets and their corresponding Outputs obtained on executing those Spark Program codes for implementation of K-means Clustering Algorithm on a dataset:**

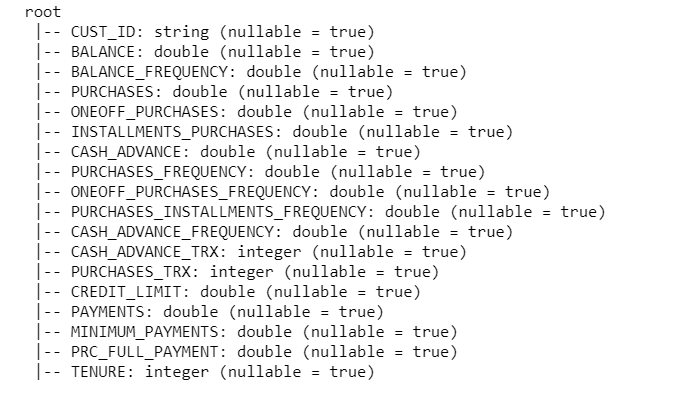
(The attributes in the Input dataset : **‘CC General.csv’ ,** can be divided into three basic Categories :- Customer Information (Primary Key as CUST\_ID), Account Information (balance, balance frequency, purchases, credit limit, tenure, etc.), and Transactions (purchase frequency, payments, cash advance, etc.)

**from pyspark.sql import SparkSession**

**spark = SparkSession.builder.appName(‘Clustering using K-Means’).getOrCreate()**

**data\_customer=spark.read.csv('CC General.csv', header=True, inferSchema=True)**

**data\_customer.printSchema()**

****

**data\_customer=data\_customer.na.drop()**

All attributes are numerical or discrete numeric, it is required to convert them into features using a Vector Assembler.

A vector assembler is a transformer that converts a set of features into a single vector column often referred to as an array of features. Features here are columns.

As customer id is an identifier that won’t be used for clustering, need to extract the required columns using .columns, then pass it as an input to Vector Assembler, and then use the transform() to convert the input columns into a single vector column called a feature.

**from pyspark.ml.feature import VectorAssembler**

**data\_customer.columns**

**assemble=VectorAssembler(inputCols=[**

**'BALANCE',**

**'BALANCE\_FREQUENCY',**

**'PURCHASES',**

**'ONEOFF\_PURCHASES',**

**'INSTALLMENTS\_PURCHASES',**

**'CASH\_ADVANCE',**

**'PURCHASES\_FREQUENCY',**

**'ONEOFF\_PURCHASES\_FREQUENCY',**

**'PURCHASES\_INSTALLMENTS\_FREQUENCY',**

**'CASH\_ADVANCE\_FREQUENCY',**

**'CASH\_ADVANCE\_TRX',**

**'PURCHASES\_TRX',**

**'CREDIT\_LIMIT',**

**'PAYMENTS',**

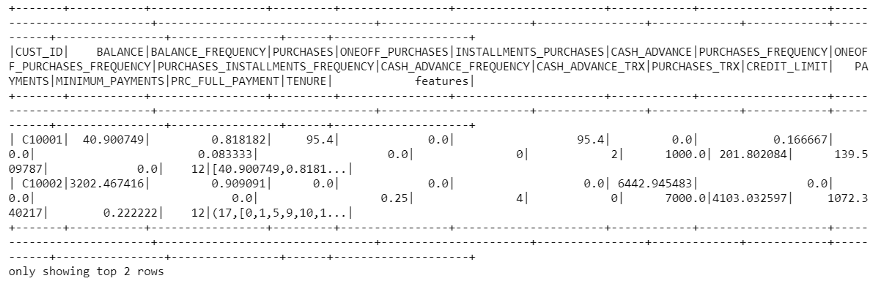
**'MINIMUM\_PAYMENTS',**

**'PRC\_FULL\_PAYMENT',**

**'TENURE'], outputCol='features')**

**assembled\_data=assemble.transform(data\_customer)**

**assembled\_data.show(2)**

****

As now all columns are transformed into a single feature vector , it is required to standardize the data to bring them to a comparable scale.

Balance can have a scale from 10–1000 whereas balance frequency has a scale from 0–1.

As Euclidean distance is always impacted more by variables on a higher scale, hence it’s important to scale the variables out.

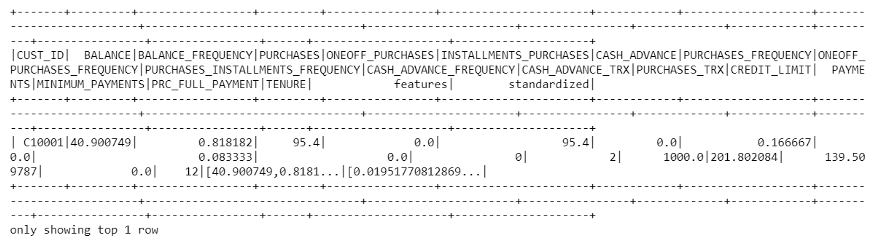
**from pyspark.ml.feature import StandardScaler**

**scale=StandardScaler(inputCol='features',outputCol='standardized')**

**data\_scale=scale.fit(assembled\_data)**

**data\_scale\_output=data\_scale.transform(assembled\_data)**

**data\_scale\_output.show(2)**

****

**Now that the Input data is standardized , the K-Means Clustering Algorithm can be effectively implemented on the data :**

**from pyspark.ml.clustering import KMeans**

**from pyspark.ml.evaluation import ClusteringEvaluator**

**silhouette\_score=[]**

**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)**

**score=evaluator.evaluate(output)**

**silhouette\_score.append(score)**

**print("Silhouette Score:",score)**

Visualizing the silhouette score it can be observed that the previous versions of the K Means had computeScore() that calculated the sum of intracluster distance

but got deprecated.

Silhouette Score using ClusteringEvaluator() measures how close each point in one cluster is to the points in the neighboring clusters , thus helping in figuring out clusters that are compact and well-spaced out.

**#Visualizing the silhouette scores in a plot**

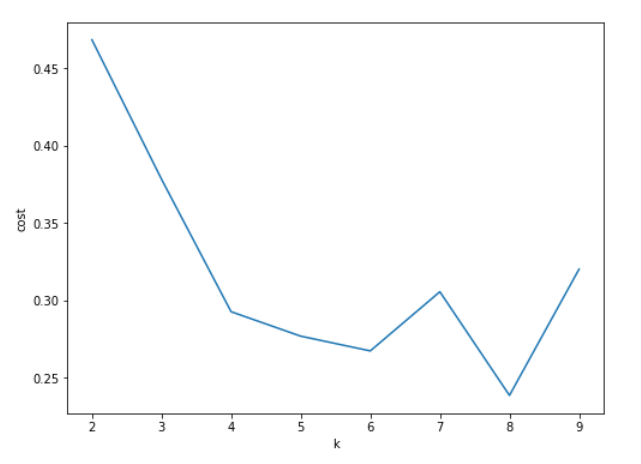
**import matplotlib.pyplot as plt**

**fig, ax = plt.subplots(1,1, figsize =(8,6))**

**ax.plot(range(2,10),silhouette\_score)**

**ax.set\_xlabel(‘k’)**

**ax.set\_ylabel(‘cost’)**

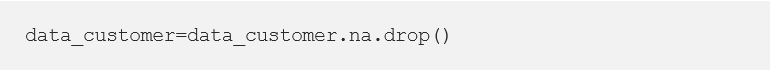
****

**It is preferred, proceeding ahead with K=7 where a local maxima of Silhouette Score is observed.**

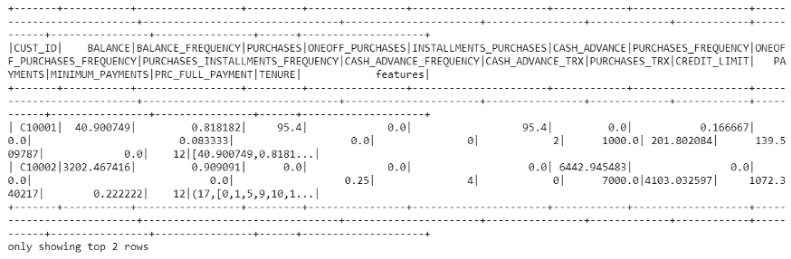
**Since there is no correct answer for what value of K is good, it can be restored to descriptive statistics and plots to check the distribution of customers.**

**Screenshots of the Spark Program Code snippets and their corresponding Outputs obtained on executing those Spark Program codes for implementation of K-means Clustering Algorithm on a dataset:**

****

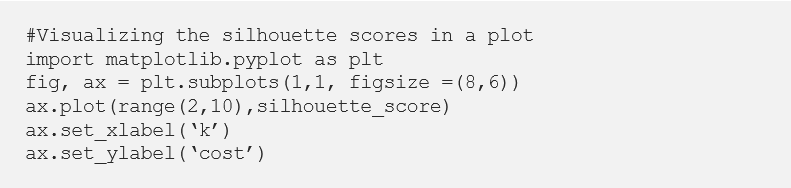
****

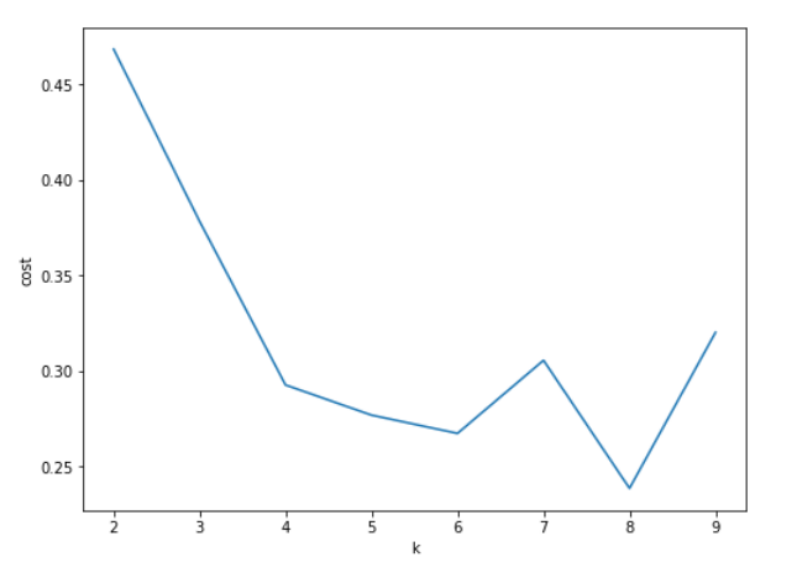
****

****

****

****

****

****

**A brief observatory Note to Explain, the basic understanding on how the K-Means Clustering algorithm executes in a Spark Cluster :**

1. **K-Means** is one of the most commonly used Clustering algorithms ,

for grouping data into a predefined number of Clusters.

**2) The ‘spark.mllib’** includes a parallelized variant of the **K\_Means++ method** called **KMeans||.**

**3) The KMeans function** from **‘pyspark.ml.clustering’** includes the following **parameters:**

1. **k** is the number of clusters specified by the user.
2. **maxIterations** is the maximum number of iterations before the clustering algorithm stops. If the intracluster distance doesn’t change beyond the epsilon value mentioned, the iteration will stop irrespective of max iterations.
3. **initializationMode** specifies either random initialization of centroids or initialization via **k-means|| (similar to K-means ++).**
4. **epsilon** determines the distance threshold within which **k-Means is expected to converge.**
5. **initialModel** is an optional set of cluster centroids that the user can provide as an input**. On using this parameter, the algorithm just runs once , to allocate points to its nearest centroid.**

The default values are as follows : **train(k=4, maxIterations=20, minDivisibleClusterSize=1.0, seed=-1888008604).**