Xi'an Jiaotong-Liverpool University
西交利物浦大學

# DTS303TC Big Data Security and Analytics

## Laboratory Session 4 – K Means Clustering using PySpark

(Duration: 2 hours)

**Overview:**

Clustering is an unsupervised learning technique, in short, you are working on data, without having any information about a target attribute or a dependent variable. The general idea of clustering is to find some intrinsic structure in the data, often referred to as groups of similar objects. The algorithm studies the data to identify these patterns or groups such that each member in a group is closer to another member in the group (lower intra-cluster distance) and farther from another member in a different group (higher inter-cluster distance).

In this activity, you will be able to perform the following in PySpark:

- Convert attributes into features using a Vector Assembler
- Use PySpark to build K-means clustering algorithm

**PART 1: K-Means Clustering using PySpark on Big Data**

**Task Instructions**

In this part, you will be programming in PySpark. When building any clustering algorithm using PySpark, one needs to perform a few data transformations.

**Data Preparation**

The dataset consists of 9K active credit cardholders over 6 months and their transaction and account attributes. The idea is to develop a customer segmentation for marketing strategy.
Data download link: https://www.heyibo.net/usr/uploads/2022/09/3335169830.zip

**Using PySpark**

**Step 1:** Import dataset and see schema information.

```
>>> from pyspark.sql import SparkSession

>>> spark = SparkSession.builder.appName('Clustering using K-Mean').getOrCreate()
>>>
>>> data_customer=spark.read.csv('file:///root/CC_GENERAL.csv', header=True, inferSchema=Tru
e)
>>> data_customer.printSchema()
root
 |-- CUST_ID: string (nullable = true)
 |-- BALANCE: double (nullable = true)
 |-- BALANCE_FREQUENCY: double (nullable = true)
 |-- PURCHASES: double (nullable = true)
 |-- ONEOFF_PURCHASES: double (nullable = true)
 |-- INSTALLMENTS_PURCHASES: double (nullable = true)
 |-- CASH_ADVANCE: double (nullable = true)
 |-- PURCHASES_FREQUENCY: double (nullable = true)
 |-- ONEOFF_PURCHASES_FREQUENCY: double (nullable = true)
 |-- PURCHASES_INSTALLMENTS_FREQUENCY: double (nullable = true)
 |-- CASH_ADVANCE_FREQUENCY: double (nullable = true)
 |-- CASH_ADVANCE_TRX: integer (nullable = true)
 |-- PURCHASES_TRX: integer (nullable = true)
 |-- CREDIT_LIMIT: double (nullable = true)
 |-- PAYMENTS: double (nullable = true)
 |-- MINIMUM_PAYMENTS: double (nullable = true)
 |-- PRC_FULL_PAYMENT: double (nullable = true)
 |-- TENURE: integer (nullable = true)
```

**Step 2:** The attributes that are self intuitive can be divided into three broader 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.).

```
>>> data_customer=data_customer.na.drop()
```

**Step 3:** All attributes under consideration are numerical or discrete numeric. Hence we need 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. Since customer id is an identifier that won't be used for clustering, we first extract the required columns using .columns, 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
['CUST_ID', 'BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES', 'ONEOFF_PURCHASES', 'INSTALLMENTS_P
URCHASES', 'CASH_ADVANCE', 'PURCHASES_FREQUENCY', 'ONEOFF_PURCHASES_FREQUENCY', 'PURCHASES_I
NSTALLMENTS_FREQUENCY', 'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX', 'CRED
IT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT', 'TENURE']
>>> 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)
```

```
>>> assembled_data.show(2)
22/09/08 04:04:42 WARN Utils: Truncated the string representation of a plan since it was too
 large. This behavior can be adjusted by setting 'spark.debug.maxToStringFields' in SparkEnv
.conf.
+-------+-----------+-----------------+---------+----------------+----------------------+---
---------+-------------------+--------------------------+-------------------------------+--
--------------------+----------------+-------------+------------+-----------+---------------
-+----------------+------+--------------------+
|CUST_ID|    BALANCE|BALANCE_FREQUENCY|PURCHASES|ONEOFF_PURCHASES|INSTALLMENTS_PURCHASES|CAS
H_ADVANCE|PURCHASES_FREQUENCY|ONEOFF_PURCHASES_FREQUENCY|PURCHASES_INSTALLMENTS_FREQUENCY|CA
SH_ADVANCE_FREQUENCY|CASH_ADVANCE_TRX|PURCHASES_TRX|CREDIT_LIMIT|   PAYMENTS|MINIMUM_PAYMENT
S|PRC_FULL_PAYMENT|TENURE|            features|
+-------+-----------+-----------------+---------+----------------+----------------------+---
---------+-------------------+--------------------------+-------------------------------+--
--------------------+----------------+-------------+------------+-----------+---------------
-+----------------+------+--------------------+
| C10001|  40.900749|         0.818182|     95.4|             0.0|                  95.4|
     0.0|           0.166667|                       0.0|                       0.083333|
               0.0|                0|            2|      1000.0| 201.802084|      139.50978
7|             0.0|    12|[40.900749,0.8181...|
| C10002|3202.467416|         0.909091|      0.0|             0.0|                   0.0| 64
42.945483|                0.0|                       0.0|                        0.0|
              0.25|                4|            0|      7000.0|4103.032597|     1072.34021
7|        0.222222|    12|(17,[0,1,5,9,10,1...|
+-------+-----------+-----------------+---------+----------------+----------------------+---
---------+-------------------+--------------------------+-------------------------------+--
--------------------+----------------+-------------+------------+-----------+---------------
-+----------------+------+--------------------+
only showing top 2 rows
```

Now that all columns are transformed into a single feature vector we need to standardize the data to bring them to a comparable scale. E.g. Balance can have a scale from 10–1000 whereas balance frequency has a scale from 0–1. Euclidean distance is always impacted more by variables on a higher scale, hence it is 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)
+-------+-----------+-----------------+---------+---------------+----------------------+---
---------+------------------+-------------------------+-------------------------------+--
-------------------+---------------+------------+-----------+-----------+---------------
-+---------------+------+--------------------+--------------------+
|CUST_ID|    BALANCE|BALANCE_FREQUENCY|PURCHASES|ONEOFF_PURCHASES|INSTALLMENTS_PURCHASES|CAS
H_ADVANCE|PURCHASES_FREQUENCY|ONEOFF_PURCHASES_FREQUENCY|PURCHASES_INSTALLMENTS_FREQUENCY|CA
SH_ADVANCE_FREQUENCY|CASH_ADVANCE_TRX|PURCHASES_TRX|CREDIT_LIMIT|   PAYMENTS|MINIMUM_PAYMENT
S|PRC_FULL_PAYMENT|TENURE|            features|        standardized|
+-------+-----------+-----------------+---------+---------------+----------------------+---
---------+------------------+-------------------------+-------------------------------+--
-------------------+---------------+------------+-----------+-----------+---------------
-+---------------+------+--------------------+--------------------+
| C10001|  40.900749|         0.818182|     95.4|            0.0|                  95.4|
      0.0|          0.166667|                      0.0|                       0.083333|
               0.0|               0|           2|      1000.0| 201.802084|      139.50978
7|            0.0|    12|[40.900749,0.8181...|[0.01951770812869...|
| C10002|3202.467416|         0.909091|      0.0|            0.0|                   0.0| 64
42.945483|               0.0|                      0.0|                        0.0|
               0.25|               4|           0|      7000.0|4103.032597|     1072.34021
7|        0.222222|    12|(17,[0,1,5,9,10,1...|(17,[0,1,5,9,10,1...|
+-------+-----------+-----------------+---------+---------------+----------------------+---
---------+------------------+-------------------------+-------------------------------+--
-------------------+---------------+------------+-----------+-----------+---------------
-+---------------+------+--------------------+--------------------+
only showing top 2 rows
```

Now that our data is standardized we can develop the K Means algorithm.

**Step 4:** K-means is one of the most commonly used clustering algorithms for grouping data into a predefined number of clusters. The spark.mllib includes a parallelized variant of the k-means++ method called kmeans||. The KMeans function from pyspark.ml.clustering includes the following parameters:

- k is the number of clusters specified by the user
- maxIterations is the maximum number of iterations before the clustering algorithm stops. Note that if the intracluster distance doesn't change beyond the epsilon value mentioned, the iteration will stop irrespective of max iterations
- initializationMode specifies either random initialization of centroids or initialization via k-means|| (similar to K-means ++)

- epsilon determines the distance threshold within which k-means is expected to converge
- initialModel is an optional set of cluster centroids that the user can provide as an input. If this parameter is used, the algorithm just runs once to allocate points to its nearest centroid

train(k=4, maxIterations=20, minDivisibleClusterSize=1.0, seed=-1888008604) are the default values.

```
>>> from pyspark.ml.clustering import KMeans
>>> from pyspark.ml.evaluation import ClusteringEvaluator
>>> silhouette_score=[]
```

```
>>> evaluator = ClusteringEvaluator(predictionCol='prediction', featuresCol='standardized',m
etricName='silhouette')
>>> 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)
...
22/09/08 04:17:27 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.Na
tiveSystemBLAS
22/09/08 04:17:27 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.Na
tiveRefBLAS
Silhouette Score: 0.29921675118619967
Silhouette Score: 0.2689359998887891
Silhouette Score: 0.28100277044184524
Silhouette Score: 0.25920526500243135
Silhouette Score: 0.2932871924946543
Silhouette Score: 0.2898535681866602
Silhouette Score: 0.30371406300895964
Silhouette Score: 0.30362317802382505
```

**Step 5:** Discuss the results achieved by the clustering algorithm.