Kmeans Clustering in Pyspark

May 13, 2017

1 Introduction

Identifying similar customers or users having similar patterns is one of the challenges faced in today's world. Segmenting or grouping such customers can lead to developing new strategies which are specically created to target these users. One such algorithm which can do clustering is called as K-means algorithm. This algorithm uses distance metrics to find distances between observations and group the similar observations. This is however just a short overview, but there is a lot of math involved in this algorithm. The objective of this project would be to cluster the household holds having similar power usage pattern so that the power companies can develop efficient strategies for them. Also any anamoly or users misusing the resources can also be detected. Such a segmenting or clustering can thus help the business in a variety of ways and hence improve the efficiency of the business model. We would be using the housing dataset obtained from UCI web repository. This dataset has more than 2 million records and represents the power consumpton patterns collected at a minute interval.

2 Motivation

With this project we aim to demonstrate the power of machine learning on apache spark and how it can be used in developing a clustering algoritm which will cluster all the similar users. We aim to optimize the business model of the power companies by giving them the information about their users. We also aim to convert the numbers stored by the business into real insights and solid patterns about their users.

3 Design

The design of the report can be split into following steps, 1. Importing libraries and creating spark session. 2. Loading the data and pre-processing it. 3. Explorartory analysis. 4. Model building, optimizing and evaluating. 5. Inferences.

3.1 Step 1: Importing libraries and creating spark session

In the below steps we will load the required libararies and create a spark session

```
In [1]: #Loading libraries required in our code.
        from pyspark.sql import SQLContext
        sqlContext = SQLContext(sc)
```

```python
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.sql.types import DoubleType
```

In [2]:
```python
#Creating the spark session
if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Kmeans") \
        .getOrCreate()
```

3.2 Step 2: Loading the data and pre-processing it.

We will use spark sqlContext to load the data in pyspark. We have a colon delimited file and hence we will explicitly define the separator in the below code. Spark sqlContext was not able to accurately infer the schema of the data, hence we manually defined the schema for each column in the dataset.

In [3]:
```python
#Loading the dataset
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true',delimiter=
    .load('/home/meaww/Downloads/household_power_consumption.txt')
```

In [4]:
```python
#Changing data types of the columns.
df=df.withColumn("Global_active_power", df.Global_active_power.cast(DoubleType()))
df=df.withColumn("Global_reactive_power", df.Global_reactive_power.cast(DoubleType()))
df=df.withColumn("Voltage", df.Voltage.cast(DoubleType()))
df=df.withColumn("Global_intensity", df.Global_intensity.cast(DoubleType()))
df=df.withColumn("Sub_metering_1", df.Sub_metering_1.cast(DoubleType()))
df=df.withColumn("Sub_metering_2", df.Sub_metering_2.cast(DoubleType()))
df=df.withColumn("Sub_metering_3", df.Sub_metering_3.cast(DoubleType()))
```

In [5]:
```python
#Removing NA records and unwanted columns
df=df.na.drop()
df=df.drop('Date')
df=df.drop('Time')
```

3.3 Step 3: Explorartory analysis

We will do some descriptive analysis of the datain the below steps. These will include getting the number of records in the data, viewing first n records of the data, getting summary dtas for the data etc.

In [6]:
```python
#getting number of observations
df.count()
```

Out[6]: 2049280

In [7]:
```python
#Viewing first 5 rows of the data
df.show(5)
```

```
+-------------------+--------------------+-------+----------------+--------------+------------
|Global_active_power|Global_reactive_power|Voltage|Global_intensity|Sub_metering_1|Sub_metering_
+-------------------+--------------------+-------+----------------+--------------+------------
|              4.216|               0.418| 234.84|            18.4|           0.0|           1.
|               5.36|               0.436| 233.63|            23.0|           0.0|           1.
|              5.374|               0.498| 233.29|            23.0|           0.0|           2.
|              5.388|               0.502| 233.74|            23.0|           0.0|           1.
|              3.666|               0.528| 235.68|            15.8|           0.0|           1.
+-------------------+--------------------+-------+----------------+--------------+------------
only showing top 5 rows
```

In [8]:
```
#Getting summary of the dataset/
df.describe().toPandas().transpose()
```

Out[8]:
```
                             0                    1                   2  \
summary                  count                 mean              stddev
Global_active_power    2049280   1.0916150365007122  1.0572941610939701
Global_reactive_power  2049280  0.12371447630388838  0.1127219795507155
Voltage                2049280    240.8398579745544  3.2399866790098937
Global_intensity       2049280    4.627759310588417   4.444396259786192
Sub_metering_1         2049280   1.1219233096502186    6.15303108970134
Sub_metering_2         2049280   1.2985199679887571   5.822026473177461
Sub_metering_3         2049280     6.45844735712055   8.437153908665614

                           3       4
summary                  min     max
Global_active_power    0.076  11.122
Global_reactive_power    0.0    1.39
Voltage                223.2  254.15
Global_intensity         0.2    48.4
Sub_metering_1           0.0    88.0
Sub_metering_2           0.0    80.0
Sub_metering_3           0.0    31.0
```

In [9]:
```
#Viewing the schema of the dataset.
df.printSchema()
```

```
root
 |-- Global_active_power: double (nullable = true)
 |-- Global_reactive_power: double (nullable = true)
 |-- Voltage: double (nullable = true)
 |-- Global_intensity: double (nullable = true)
 |-- Sub_metering_1: double (nullable = true)
 |-- Sub_metering_2: double (nullable = true)
 |-- Sub_metering_3: double (nullable = true)
```

3.4 Step 4: Model building, optimizing and evaluating.

We would be building a kmeans model here, however there are some important points which should be considered before building th model. As mentioned in the introduction that kmeans uses distance meaures to find similar users. This assumes that all the columns have a same scale. If the scale is not same, it should be normalized so that they are on same scale and the distances measured can be compared across columns. The model also takes input in dense vector format and hence proper conversions are also done. We will use a assempler to create the dense vector.

In [10]:
```
#Assempling and creating a dense vector of inputs.
featuresUsed = df.columns
assembler = VectorAssembler(inputCols=featuresUsed, outputCol="features_unscaled")
assembled = assembler.transform(df)
```

In [11]:
```
#Scaling and normalizing the data.
scaler = StandardScaler(inputCol="features_unscaled", outputCol="features", withStd=Tru
scalerModel = scaler.fit(assembled)
scaledData = scalerModel.transform(assembled)
```

In [12]:
```
scaledData = scaledData.select("features")
scaledData.persist()
```

Out[12]:
```
DataFrame[features: vector]
```

In [13]:
```
#Viewing first 5 rows of scaled data
scaledData.head(5)
```

Out[13]:
```
[Row(features=DenseVector([2.9551, 2.6107, -1.8518, 3.0988, -0.1823, -0.0513, 1.2494]))
 Row(features=DenseVector([4.0371, 2.7704, -2.2253, 4.1338, -0.1823, -0.0513, 1.1309]))
 Row(features=DenseVector([4.0503, 3.3204, -2.3302, 4.1338, -0.1823, 0.1205, 1.2494])),
 Row(features=DenseVector([4.0636, 3.3559, -2.1913, 4.1338, -0.1823, -0.0513, 1.2494]))
 Row(features=DenseVector([2.4349, 3.5866, -1.5926, 2.5138, -0.1823, -0.0513, 1.2494]))
```

Kmeans algorithm requires the number of clusters to preknown or to be assumed and finding can be done by some calculations. The algorithm calculates distance of the points from the initially randomly selected centroids. Then we will group and form a cluster of records which are closest to each other. Then based on the new groups we get a new adjusted centroid and distances are calculated again. This process continues for multiple iterations and the end result are the clusters having minimum within sum of squared errors. However finding optimum number of clusters is also a challenge. To solve this, we will build the kmeans model on multiple number of cluster values and find the one which has the optimum value of within sum of squared errors.

In [14]:
```
#Building model for different cluster values
for i in xrange(2,18):
    kmeans = KMeans().setK(i).setSeed(1+i)
    model = kmeans.fit(scaledData)
    wssse = model.computeCost(scaledData)
    print("Within Set Sum of Squared Errors for " + str(i) + " clusters is: " + str(wss
```

```
Within Set Sum of Squared Errors for 2 clusters is: 9877808.13851
Within Set Sum of Squared Errors for 3 clusters is: 7359092.92477
Within Set Sum of Squared Errors for 4 clusters is: 5667438.62157
Within Set Sum of Squared Errors for 5 clusters is: 5023874.78373
Within Set Sum of Squared Errors for 6 clusters is: 4358179.92543
Within Set Sum of Squared Errors for 7 clusters is: 3865183.53805
Within Set Sum of Squared Errors for 8 clusters is: 3649760.92402
Within Set Sum of Squared Errors for 9 clusters is: 3568468.36212
Within Set Sum of Squared Errors for 10 clusters is: 3529721.33258
Within Set Sum of Squared Errors for 11 clusters is: 3110280.56329
Within Set Sum of Squared Errors for 12 clusters is: 2841578.99354
Within Set Sum of Squared Errors for 13 clusters is: 2715241.06545
Within Set Sum of Squared Errors for 14 clusters is: 2694933.19454
Within Set Sum of Squared Errors for 15 clusters is: 2658764.28829
Within Set Sum of Squared Errors for 16 clusters is: 2433062.12566
Within Set Sum of Squared Errors for 17 clusters is: 2428000.93015
```

We can see from the above set of values that the wsse doesn't decrease much after 13 clusters. Hence we would choose our cluster count as 13. It can also be said that we are splitting our user base into 13 categories which can be inferred by looking at the records. It is also highly possible that these may not be the total number of categories and there might be more such categories. These can be identified by getting the understanding of the business domain and checking the wsse on more number of clusters. We will now build the final model on 13 clusters again and append the cluster value to each record.

In [15]:
```
#Buildinfg final model and appending the predictions/categories
kmeans = KMeans().setK(13).setSeed(14)
model = kmeans.fit(scaledData)
transformed = model.transform(scaledData)
```

In [16]:
```
#Viewing first 5 rows of the data
transformed.head(5)
```

Out[16]:
```
[Row(features=DenseVector([2.9551, 2.6107, -1.8518, 3.0988, -0.1823, -0.0513, 1.2494]),
 Row(features=DenseVector([4.0371, 2.7704, -2.2253, 4.1338, -0.1823, -0.0513, 1.1309]),
 Row(features=DenseVector([4.0503, 3.3204, -2.3302, 4.1338, -0.1823, 0.1205, 1.2494]),
 Row(features=DenseVector([4.0636, 3.3559, -2.1913, 4.1338, -0.1823, -0.0513, 1.2494]),
 Row(features=DenseVector([2.4349, 3.5866, -1.5926, 2.5138, -0.1823, -0.0513, 1.2494]),
```

3.5 Step 5: Inferences

We have now idnetified the optimum number of clusters and found out the categories of each user. In the below step we will look at the cluster centroids. Since we built 13 clusters we will have 13 cluster centers. Each cluster center will have dimensions equal to that of the input data. These values in a way represents the mean values of all features for each cluster. And any new observation having values close to these centers, will have the category assigned of the nearest cluster.

```
In [17]: centers = model.clusterCenters()
         print("Cluster Centers: ")
         for center in centers:
             print(center)
```

```
Cluster Centers:
[-0.52079573  1.1100625   0.28579226 -0.48186692 -0.17264899 -0.07286142
 -0.6523427 ]
[-0.6614767  -0.48314765  1.38469467 -0.66955756 -0.17923476 -0.18529062
 -0.72472727]
[ 4.93454693  0.98571463 -1.89174055  5.02327482  3.72143501  7.97252492
  0.83800104]
[ 2.1983827  -0.02917821 -0.84190595  2.20231466  5.7107185  -0.13598369
  0.27553774]
[ 2.33007943  0.56888218 -0.91045987  2.35493009 -0.13521022  5.65007765
  0.47542549]
[ 0.8214675   2.15979215 -0.34314917  0.83928564 -0.10378456 -0.02395986
  1.11847144]
[ 0.45608216 -0.34862445  0.4073854   0.41265118 -0.16520296 -0.16826356
  1.4020373 ]
[ 1.93828693  0.10043299 -0.97279     1.94157731 -0.09448841 -0.10766511
  1.26863977]
[ 3.36576756  1.73872278 -1.50040016  3.41434325  6.00549301 -0.02008365
  0.99364119]
[ 0.62832283 -0.16538037 -0.34701093  0.62167322 -0.13529261 -0.0822579
 -0.72941109]
[ 0.43558148 -0.29243143 -1.06128706  0.42328072 -0.16309598 -0.16112524
  1.3238972 ]
[-0.72647213 -0.56193807  0.16411659 -0.72836206 -0.17940623 -0.18718808
 -0.71286533]
[-0.66555302 -0.2666868  -1.34120734 -0.65268031 -0.17648226 -0.16651239
 -0.73700124]
```

4 Challenges faced

Folowing were the challenges faced, 1. Installing spark and integrating it with jupyter. 2. Defining the problem statement and getting the data. 3. Since the data was large, the compute time and compute was very intensive 4. Getting the data in required format and model building

5 Conclusion

We were successfull in clustering the households based on the power consumption patterns. Also the business was made aware about insights which were not possible to get earlier. This unsupervised learning approach has thus helped in improving the efficiency of the business.