APACHE SPARK---WITH MACHINE LEARNING LIBRARY --- CRIME DATA ANALYSIS

Detailed Description of the Project

As mentioned earlier in proposal, The goal behind the project is to analyze the crime dataset of a particular city(region) with the help of machine learning algorithms namely regression(linear), clustering, classification(non-quantitative approach), collaborative filtering etc. where the data from the previous momentous crimes will be analyzed and considering the goal of prediction, model will be built using analyzed significant results which can predict the crime rates for the future purposes.

Here I have already imported the dataset and have performed certain operations on it like data cleaning, removing spurious characters and duplicates from it. Further I will be converting my data into labeled points which will be used as prescribed format for the data to be present for the mllib library. Then this data will be loaded into the spark system and with the help of Decision trees, Linear regression, k means clustering, Naive Bayes, Random Forest, F- P Growth and Hashing methods, analysis will be carried out to see how our system responds in providing the results.

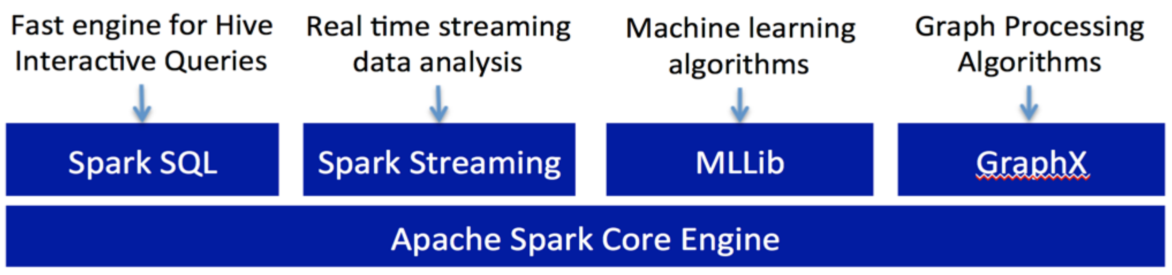
Literature Review

Apache Spark

Apache Spark is an open-source powerful distributed querying and processing engine. It provides flexibility and extensibility of MapReduce but at significantly higher speeds: Up to 100 times faster than Apache Hadoop when data is stored in memory and up to 10 times when accessing disk.

Apache Spark allows the user to read, transform, and aggregate data, as well as train and deploy sophisticated statistical models with ease. The Spark APIs are accessible in Java, Scala, Python, R and SQL. Apache Spark can be used to build applications or package them up as libraries to be deployed on a cluster or perform *quick* analytics interactively through notebooks (like, for instance, Jupyter, Spark-Notebook, Data bricks notebooks, and Apache Zeppelin).

Apache Spark can easily run locally on a laptop, yet can also easily be deployed in standalone mode, over YARN, or Apache Mesos - either on your local cluster or in the cloud. It can read and write from a diverse data sources including (but not limited to) HDFS, Apache Cassandra, Apache HBase, and S3:


Fast engine for Hive Interactive Queries
Real time streaming data analysis
Machine learning algorithms
Graph Processing Algorithms
Spark SQL
Spark Streaming
MLLib
GraphX
Apache Spark Core Engine

Spark Core

Spark Core is the foundation of the overall project. It provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) centered on the RDD abstraction (the Java API is available for other JVM languages, but is also usable for some other non-JVM languages, such as Julia, that can connect to the JVM). This interface mirrors a functional/higher-order model of programming: a "driver" program invokes parallel operations such as map, filter or reduce on an RDD by passing a function to Spark, which then schedules the function's execution in parallel on the cluster. These operations, and additional ones such as joins, take RDDs as input and produce new RDDs. RDDs are immutable and their operations are lazy; fault-tolerance is achieved by keeping track of the "lineage" of each RDD (the sequence of operations that produced it) so that it can be reconstructed in the case of data loss. RDDs can contain any type of Python, Java, or Scala objects.

Spark Streaming

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD transformations on those mini-batches of data. This design enables the same set of application code written for batch analytics to be used in streaming analytics, thus facilitating easy implementation of lambda architecture.[16][17] However, this convenience comes with the penalty of latency equal to the mini-batch duration. Other streaming data engines that process event by event rather than in mini-batches include Storm and the streaming component of Flink.[18] Spark Streaming has support built-in to consume from Kafka, Flume, Twitter, ZeroMQ, Kinesis, and TCP/IP sockets.

Spark SQL

Spark SQL is a component on top of Spark Core that introduced a data abstraction called Data Frames,[a] which provides support for structured and semi-structured data. Spark SQL provides a domain-specific language (DSL) to manipulate Data Frames in Scala, Java, or Python. It also provides SQL language support, with command-line interfaces and ODBC/JDBC server. Although Data Frames lack the compile-time type-checking afforded by RDDs, as of Spark 2.0, the strongly typed Dataset is fully supported by Spark SQL as well.

MLlib Machine Learning Library

Spark MLlib is a distributed machine learning framework on top of Spark Core that, due in large part to the distributed memory-based Spark architecture, is as much as nine times as fast as the disk-based implementation used by Apache Mahout (according to benchmarks done by the MLlib developers against the Alternating Least Squares (ALS) implementations, and before Mahout itself gained a Spark interface), and scales better than Vowpal Wabbit.[21] Many common machine learning and statistical algorithms have been implemented and are shipped with MLlib which simplifies large scale machine learning pipelines.

Overview of Mlib:

-Transformer

The Transformer class, like the name suggests, *transforms* your data by (normally) appending a new column to your DataFrame.

At the high level, when deriving from the Transformer abstract class, each and every new Transformer needs to implement a .transform(...) method. The method, as a first and normally the only obligatory parameter, requires passing a DataFrame to be transformed. This, of course, varies *method-by-method* in the ML package: other *popular* parameters are inputCol and outputCol; these, however, frequently default to some predefined values, such as, for example, 'features' for the inputCol parameter.

There are many Transformers offered in the spark.ml.feature and we will briefly describe them here (before we use some of them later in this chapter):

Bucketizer: Similar to the Binarizer, this method takes a list of thresholds (the splits parameter) and transforms a continuous variable into a multinomial one.

Binarizer: Given a threshold, the method takes a continuous variable and transforms it into a binary one.

ChiSqSelector: For the categorical target variables (think classification models), this feature allows you to select a predefined number of features (parameterized by the numTopFeatures parameter) that explain the variance in the target the best. The selection is done, as the name of the method suggests, using a Chi-Square test. It is one of the two-step methods: first, you need to .fit(...) your data (so the method can calculate the Chi-square tests). Calling the .fit(...) method (you pass your DataFrame as a parameter) returns a ChiSqSelectorModel object that you can then use to transform your DataFrame using the .transform(...) method.
DCT: The Discrete Cosine Transform takes a vector of real values and returns a vector of the same length, but with the sum of cosine functions oscillating at different frequencies. Such transformations are useful to extract some underlying frequencies in your data or in data compression.

ElementwiseProduct: A method that returns a vector with elements that are products of the vector passed to the method, and a vector passed as the scalingVec parameter. For example, if you had a [10.0, 3.0, 15.0] vector and your scalingVec was [0.99, 3.30, 0.66], then the vector you would get would look as follows: [9.9, 9.9, 9.9].

NGram: This method takes a list of tokenized text and returns *n-grams*: pairs, triples, or *n-mores* of subsequent words. For example, if you had a ['good', 'morning', 'Robin', 'Williams'] vector you would get the following output: ['good morning', 'morning Robin', 'Robin Williams'].
Normalizer: This method scales the data to be of unit norm using the p-norm value (by default, it is L2).

IDF: This method computes an **Inverse Document Frequency** for a list of documents. Note that the documents need to already be represented as a vector (for example, using either the HashingTF or CountVectorizer).

IndexToString: A complement to the StringIndexer method. It uses the encoding from the StringIndexerModel object to reverse the string index to original values. As an aside, please note that this sometimes does not work and you need to specify the values from the StringIndexer.

Estimators

Estimators can be thought of as statistical models that need to be estimated to make predictions or classify your observations.

If deriving from the abstract Estimator class, the new model has to implement the .fit(...) method that fits the model given the data found in a DataFrame and some default or user-specified parameters.

There are a lot of estimators available in PySpark and we will now shortly describe the models available in Spark 2.0.

Model Review:

Linear regression:

Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable. For example, a modeler might want to relate the weights of individuals to their heights using a linear regression model.

K-means clustering:

K-means algorithm identifies *k* number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.The *'means'* in the K-means refers to averaging of the data; that is, finding the centroid.

Random Forest Classifier:

This model produces multiple decision trees (hence the name—forest) and uses the mode output of those decision trees to classify observations. The RandomForestClassifier supports both binary and multinomial labels.

Naïve Bayes:

Based on the Bayes' theorem, this model uses conditional probability theory to classify observations. The NaiveBayes model in PySpark ML supports both binary and multinomial labels.

Dataset Description

https://catalog.data.gov/dataset/statisticsmajor-crimes-2014-by-precinct-and-beat

Here I will be using the above-mentioned dataset which has been downloaded from the repository data.gov.

The dataset comprises of the statistics about the crimes that took place in 2014, which are recorded based on the precinct and the beats.

Here the Number of variables are: **- 13**

Number of Instances are:- **1048576**

Description of attributes:

Below are the attributes through which analysis will be carried around: -

Precinct : Location where the crime has been recorded.

Beat : The beat where the crime has been recorded

Homicide : Number of Homicides that had materialize in that precinct.

Rape : Number of Rapes that happened in that particular precinct.

Robbery : Number of robberies that took place in that precinct

Assault : Number of assault cases that has happened in that particular precinct.

Violent_Crimes_Total : Number of violent crimes cases that occurred in that precinct.

Burglary : Number of burglary cases that took place during that precinct.

Larceny_Theft : Number of larceny theft cases that materialize in that precinct.

Vehicle_Theft : Number of cases where vehicle thefts took place during that precinct.

Property_Major_theft : Number of Property thefts cases that occurred in that precinct.

Major_crime_Total : Number of Major crimes that has been reported in that precinct.

Report_Date : The date on which the particular crime has been reported.

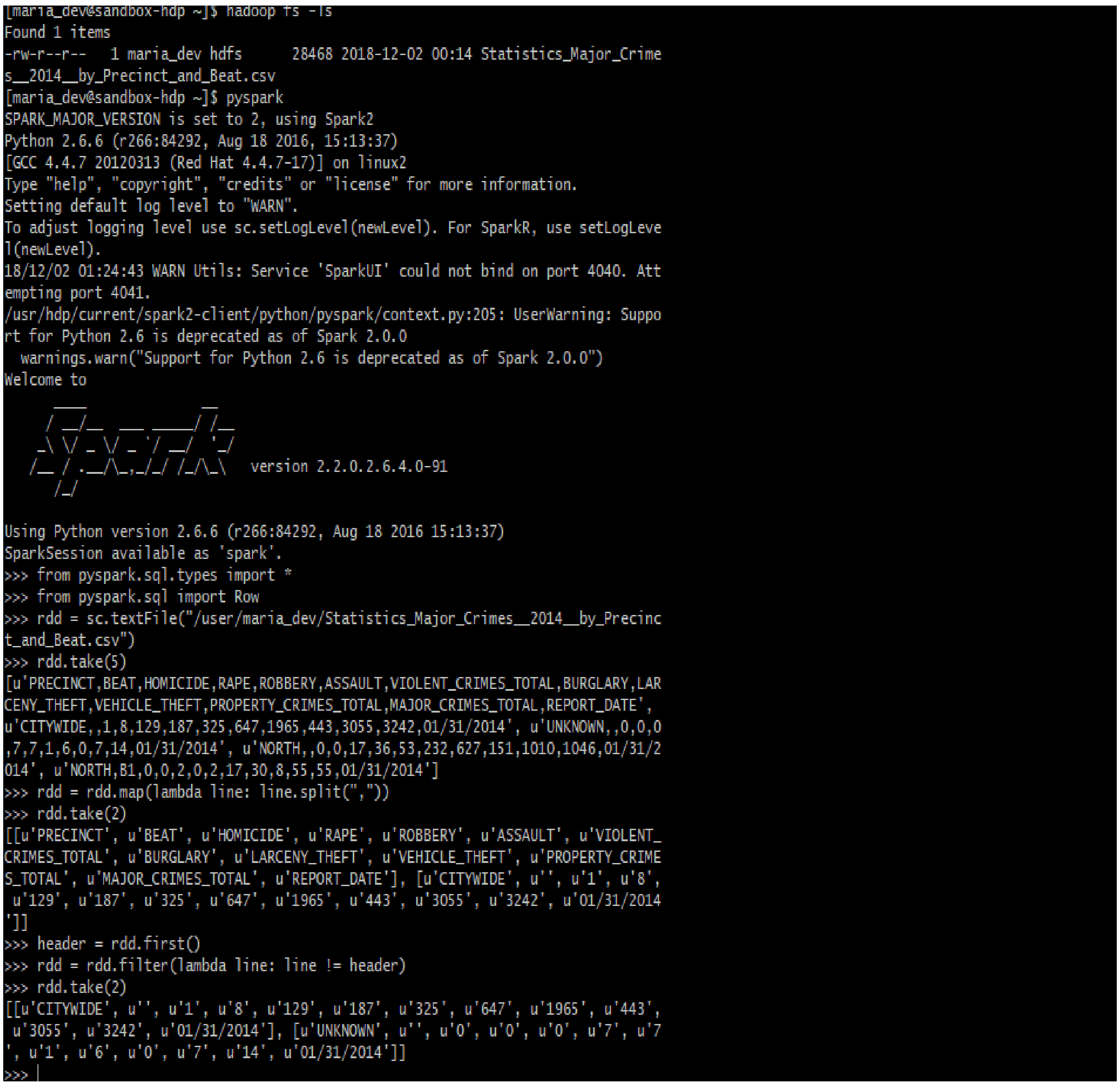
Detailed Description of Project

- **Data Preparation and Data cleaning**

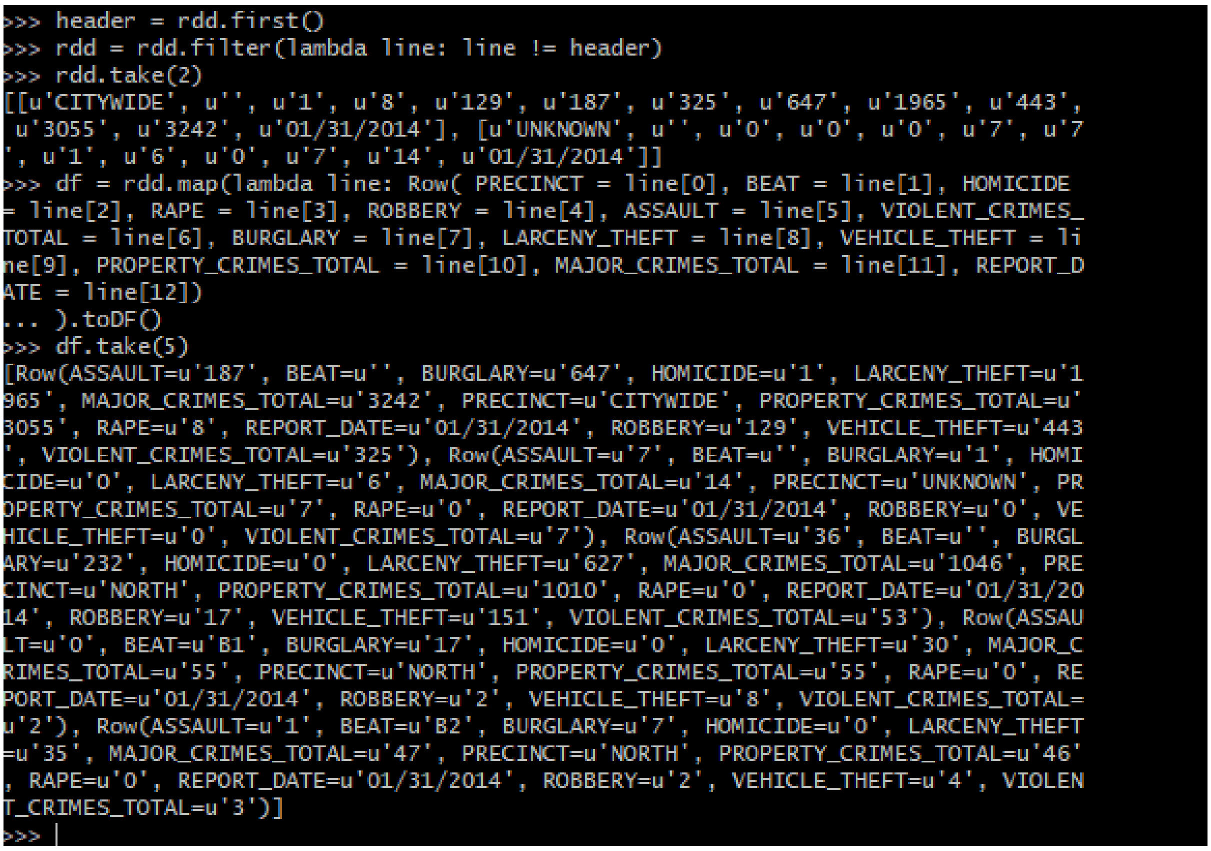
  -First of all, we will load out dataset (.xlsx) file to cluster using scp command.

  - Here the data is further lade into a variable and splitting is done on it based on the " , ", then we will read the headers from the file to delete them.

  -Also mlib libraries are used cause our data must be in the form of Labeled Points in order to perform analysis and regression on it.

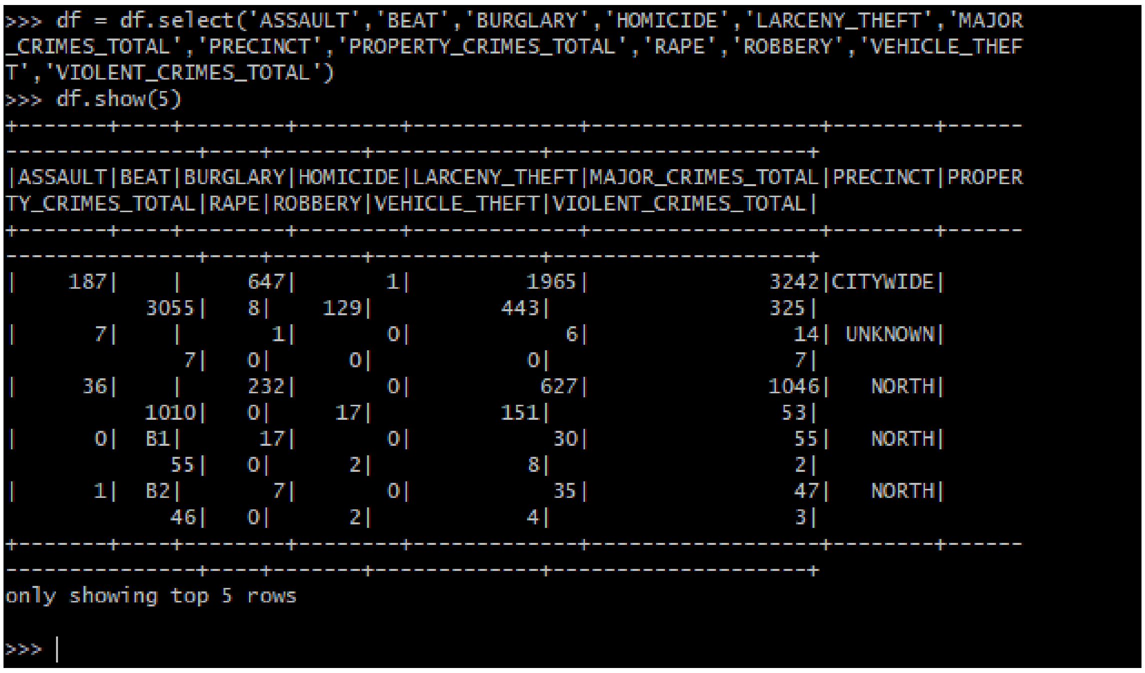


```
[maria_dev@sandbox-hdp ~]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 maria_dev hdfs      28468 2018-12-02 00:14 Statistics_Major_Crime
s__2014__by_Precinct_and_Beat.csv
[maria_dev@sandbox-hdp ~]$ pyspark
SPARK_MAJOR_VERSION is set to 2, using Spark2
Python 2.6.6 (r266:84292, Aug 18 2016, 15:13:37)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-17)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLeve
l(newLevel).
18/12/02 01:24:43 WARN Utils: Service 'SparkUI' could not bind on port 4040. Att
empting port 4041.
/usr/hdp/current/spark2-client/python/pyspark/context.py:205: UserWarning: Suppo
rt for Python 2.6 is deprecated as of Spark 2.0.0
  warnings.warn("Support for Python 2.6 is deprecated as of Spark 2.0.0")
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.4.0-91
      /_/

Using Python version 2.6.6 (r266:84292, Aug 18 2016 15:13:37)
SparkSession available as 'spark'.
>>> from pyspark.sql.types import *
>>> from pyspark.sql import Row
>>> rdd = sc.textFile("/user/maria_dev/Statistics_Major_Crimes__2014__by_Precinc
t_and_Beat.csv")
>>> rdd.take(5)
[u'PRECINCT,BEAT,HOMICIDE,RAPE,ROBBERY,ASSAULT,VIOLENT_CRIMES_TOTAL,BURGLARY,LAR
CENY_THEFT,VEHICLE_THEFT,PROPERTY_CRIMES_TOTAL,MAJOR_CRIMES_TOTAL,REPORT_DATE',
u'CITYWIDE,,1,8,129,187,325,647,1965,443,3055,3242,01/31/2014', u'UNKNOWN,,0,0,0
,7,7,1,6,0,7,14,01/31/2014', u'NORTH,,0,0,17,36,53,232,627,151,1010,1046,01/31/2
014', u'NORTH,B1,0,0,2,0,2,17,30,8,55,55,01/31/2014']
>>> rdd = rdd.map(lambda line: line.split(","))
>>> rdd.take(2)
[[u'PRECINCT', u'BEAT', u'HOMICIDE', u'RAPE', u'ROBBERY', u'ASSAULT', u'VIOLENT_
CRIMES_TOTAL', u'BURGLARY', u'LARCENY_THEFT', u'VEHICLE_THEFT', u'PROPERTY_CRIME
S_TOTAL', u'MAJOR_CRIMES_TOTAL', u'REPORT_DATE'], [u'CITYWIDE', u'', u'1', u'8',
 u'129', u'187', u'325', u'647', u'1965', u'443', u'3055', u'3242', u'01/31/2014
']]
>>> header = rdd.first()
>>> rdd = rdd.filter(lambda line: line != header)
>>> rdd.take(2)
[[u'CITYWIDE', u'', u'1', u'8', u'129', u'187', u'325', u'647', u'1965', u'443',
 u'3055', u'3242', u'01/31/2014'], [u'UNKNOWN', u'', u'0', u'0', u'0', u'7', u'7
', u'1', u'6', u'0', u'7', u'14', u'01/31/2014']]
>>> 
```



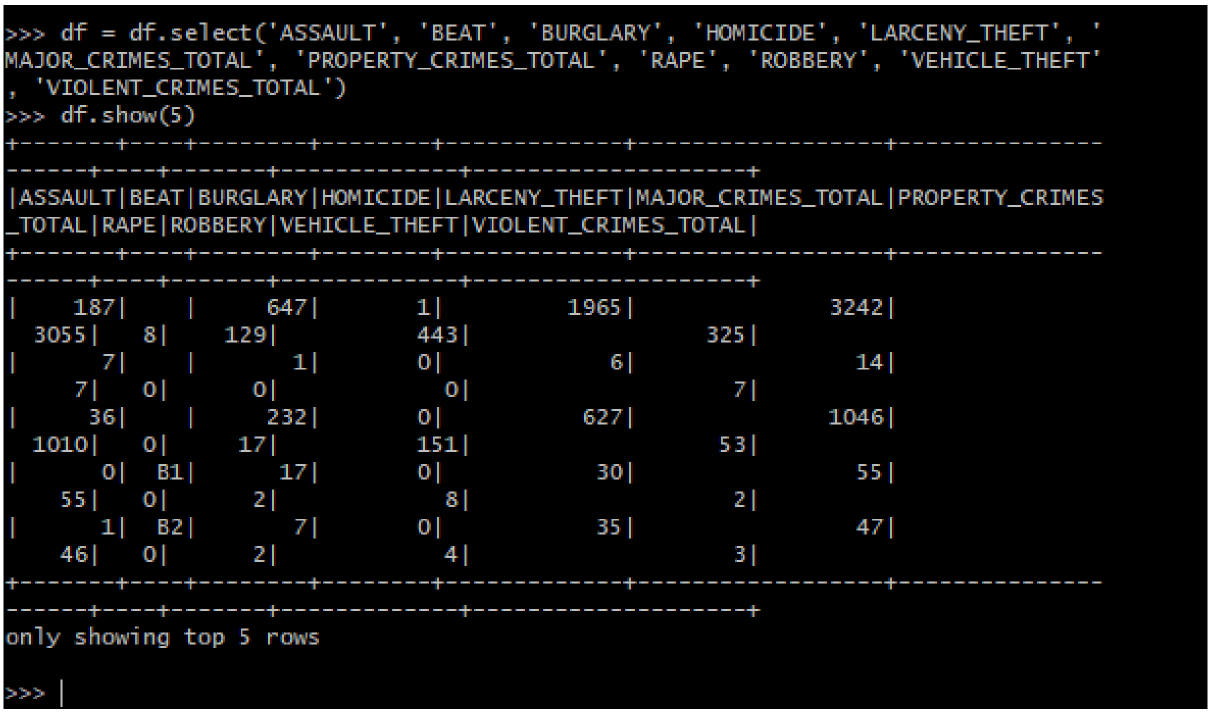
```
>>> header = rdd.first()
>>> rdd = rdd.filter(lambda line: line != header)
>>> rdd.take(2)
[[u'CITYWIDE', u'', u'1', u'8', u'129', u'187', u'325', u'647', u'1965', u'443',
 u'3055', u'3242', u'01/31/2014'], [u'UNKNOWN', u'', u'0', u'0', u'0', u'7', u'7
', u'1', u'6', u'0', u'7', u'14', u'01/31/2014']]
>>> df = rdd.map(lambda line: Row( PRECINCT = line[0], BEAT = line[1], HOMICIDE
= line[2], RAPE = line[3], ROBBERY = line[4], ASSAULT = line[5], VIOLENT_CRIMES_
TOTAL = line[6], BURGLARY = line[7], LARCENY_THEFT = line[8], VEHICLE_THEFT = li
ne[9], PROPERTY_CRIMES_TOTAL = line[10], MAJOR_CRIMES_TOTAL = line[11], REPORT_D
ATE = line[12])
... ).toDF()
>>> df.take(5)
[Row(ASSAULT=u'187', BEAT=u'', BURGLARY=u'647', HOMICIDE=u'1', LARCENY_THEFT=u'1
965', MAJOR_CRIMES_TOTAL=u'3242', PRECINCT=u'CITYWIDE', PROPERTY_CRIMES_TOTAL=u'
3055', RAPE=u'8', REPORT_DATE=u'01/31/2014', ROBBERY=u'129', VEHICLE_THEFT=u'443
', VIOLENT_CRIMES_TOTAL=u'325'), Row(ASSAULT=u'7', BEAT=u'', BURGLARY=u'1', HOMI
CIDE=u'0', LARCENY_THEFT=u'6', MAJOR_CRIMES_TOTAL=u'14', PRECINCT=u'UNKNOWN', PR
OPERTY_CRIMES_TOTAL=u'7', RAPE=u'0', REPORT_DATE=u'01/31/2014', ROBBERY=u'0', VE
HICLE_THEFT=u'0', VIOLENT_CRIMES_TOTAL=u'7'), Row(ASSAULT=u'36', BEAT=u'', BURGL
ARY=u'232', HOMICIDE=u'0', LARCENY_THEFT=u'627', MAJOR_CRIMES_TOTAL=u'1046', PRE
CINCT=u'NORTH', PROPERTY_CRIMES_TOTAL=u'1010', RAPE=u'0', REPORT_DATE=u'01/31/20
14', ROBBERY=u'17', VEHICLE_THEFT=u'151', VIOLENT_CRIMES_TOTAL=u'53'), Row(ASSAU
LT=u'0', BEAT=u'B1', BURGLARY=u'17', HOMICIDE=u'0', LARCENY_THEFT=u'30', MAJOR_C
RIMES_TOTAL=u'55', PRECINCT=u'NORTH', PROPERTY_CRIMES_TOTAL=u'55', RAPE=u'0', RE
PORT_DATE=u'01/31/2014', ROBBERY=u'2', VEHICLE_THEFT=u'8', VIOLENT_CRIMES_TOTAL=
u'2'), Row(ASSAULT=u'1', BEAT=u'B2', BURGLARY=u'7', HOMICIDE=u'0', LARCENY_THEFT
=u'35', MAJOR_CRIMES_TOTAL=u'47', PRECINCT=u'NORTH', PROPERTY_CRIMES_TOTAL=u'46'
, RAPE=u'0', REPORT_DATE=u'01/31/2014', ROBBERY=u'2', VEHICLE_THEFT=u'4', VIOLEN
T_CRIMES_TOTAL=u'3')]
>>>
```

-Here the RDD is being converted into a Data Frame by naming the fields earlier that is with the help of .toDF(), we converted the loaded RDD to dataframe.



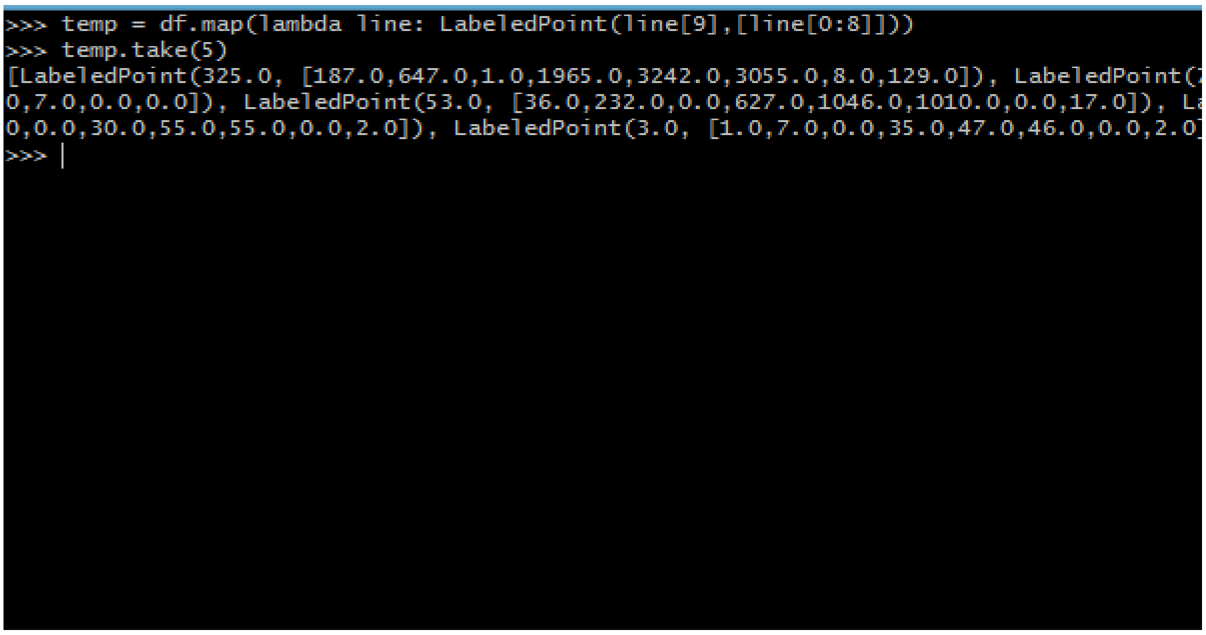
```
>>> df = df.select('ASSAULT','BEAT','BURGLARY','HOMICIDE','LARCENY_THEFT','MAJOR
_CRIMES_TOTAL','PRECINCT','PROPERTY_CRIMES_TOTAL','RAPE','ROBBERY','VEHICLE_THEF
T','VIOLENT_CRIMES_TOTAL')
>>> df.show(5)
+-------+----+--------+--------+-------------+------------------+--------+------
---------------+----+-------+-------------+--------------------+
|ASSAULT|BEAT|BURGLARY|HOMICIDE|LARCENY_THEFT|MAJOR_CRIMES_TOTAL|PRECINCT|PROPER
TY_CRIMES_TOTAL|RAPE|ROBBERY|VEHICLE_THEFT|VIOLENT_CRIMES_TOTAL|
+-------+----+--------+--------+-------------+------------------+--------+------
---------------+----+-------+-------------+--------------------+
|    187|    |     647|       1|         1965|              3242|CITYWIDE|
           3055|   8|    129|          443|                 325|
|      7|    |       1|       0|            6|                14| UNKNOWN|
              7|   0|      0|            0|                   7|
|     36|    |     232|       0|          627|              1046|   NORTH|
           1010|   0|     17|          151|                  53|
|      0|  B1|      17|       0|           30|                55|   NORTH|
             55|   0|      2|            8|                   2|
|      1|  B2|       7|       0|           35|                47|   NORTH|
             46|   0|      2|            4|                   3|
+-------+----+--------+--------+-------------+------------------+--------+------
---------------+----+-------+-------------+--------------------+
only showing top 5 rows

>>> |
```

-Here the Data Frame is being filtered based on our analysis requirement - (use of select command).



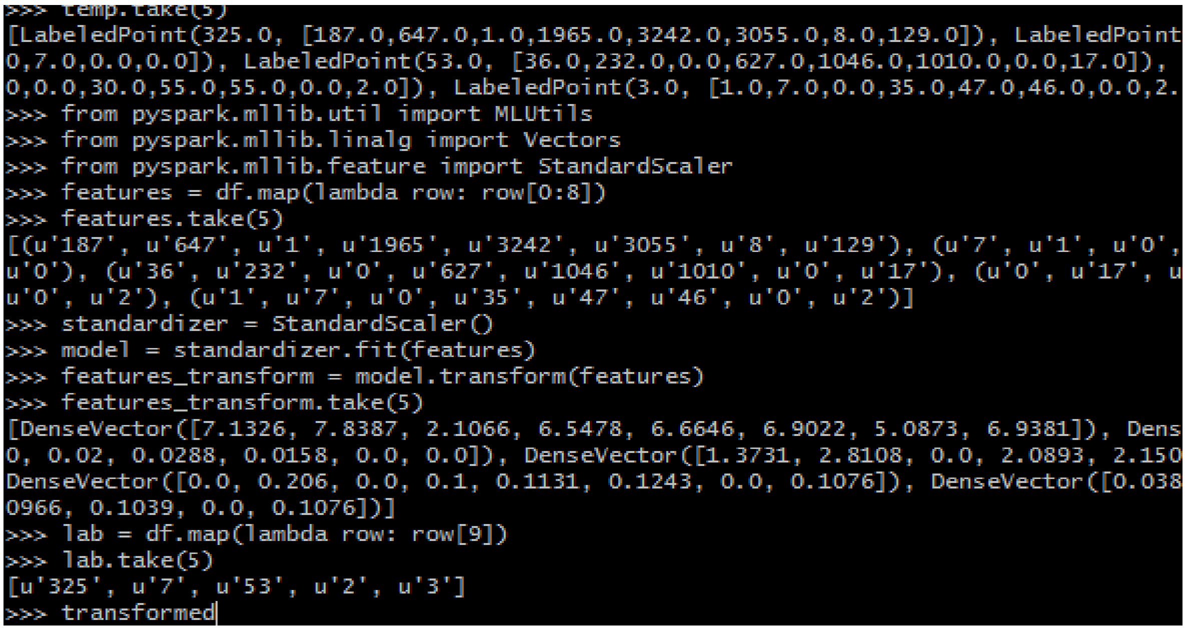
```
>>> df = df.select('ASSAULT', 'BEAT', 'BURGLARY', 'HOMICIDE', 'LARCENY_THEFT', '
MAJOR_CRIMES_TOTAL', 'PROPERTY_CRIMES_TOTAL', 'RAPE', 'ROBBERY', 'VEHICLE_THEFT'
, 'VIOLENT_CRIMES_TOTAL')
>>> df.show(5)
+-------+----+--------+--------+-------------+------------------+---------------
------+----+-------+-------------+--------------------+
|ASSAULT|BEAT|BURGLARY|HOMICIDE|LARCENY_THEFT|MAJOR_CRIMES_TOTAL|PROPERTY_CRIMES
_TOTAL|RAPE|ROBBERY|VEHICLE_THEFT|VIOLENT_CRIMES_TOTAL|
+-------+----+--------+--------+-------------+------------------+---------------
------+----+-------+-------------+--------------------+
|    187|    |     647|       1|         1965|              3242|
   3055|   8|    129|          443|                 325|
|      7|    |       1|       0|            6|                14|
      7|   0|      0|            0|                   7|
|     36|    |     232|       0|          627|              1046|
   1010|   0|     17|          151|                  53|
|      0|  B1|      17|       0|           30|                55|
     55|   0|      2|            8|                   2|
|      1|  B2|       7|       0|           35|                47|
     46|   0|      2|            4|                   3|
+-------+----+--------+--------+-------------+------------------+---------------
------+----+-------+-------------+--------------------+
only showing top 5 rows

>>> |
```

- During Data frame filtering, we observed that, column PRECINCT comprises of Null value, so we removed that column.



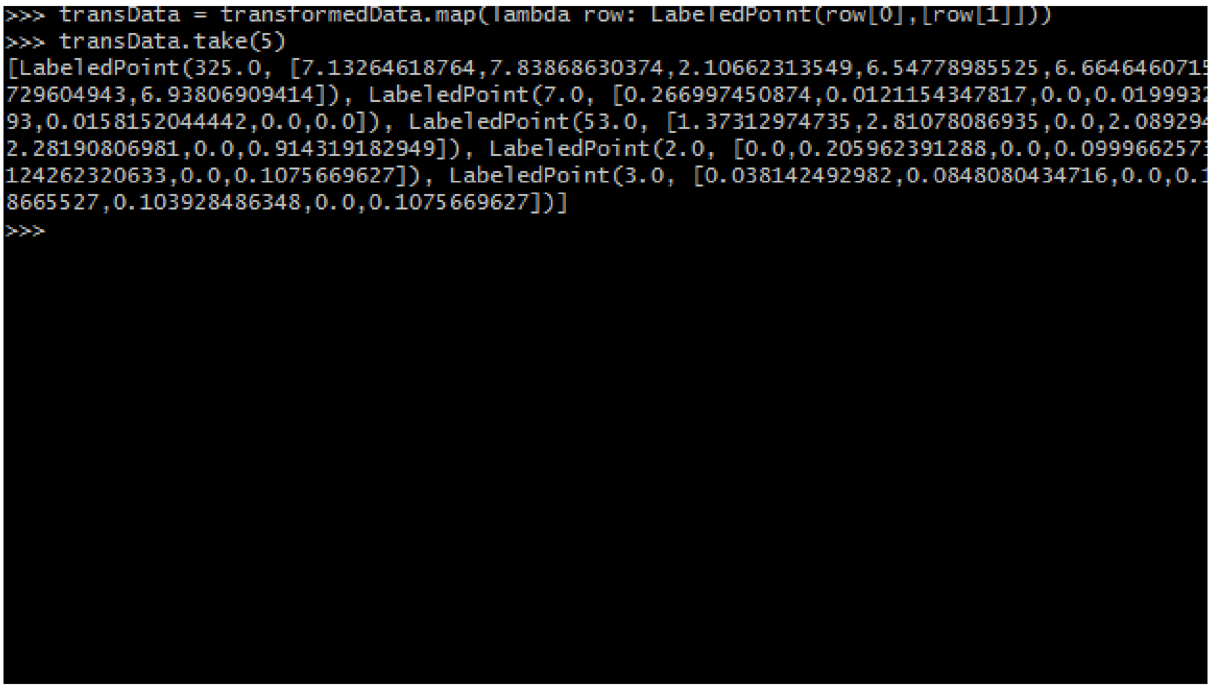
```
>>> temp = df.map(lambda line: LabeledPoint(line[9],[line[0:8]]))
>>> temp.take(5)
[LabeledPoint(325.0, [187.0,647.0,1.0,1965.0,3242.0,3055.0,8.0,129.0]), LabeledPoint(7
0,7.0,0.0,0.0,0.0]), LabeledPoint(53.0, [36.0,232.0,0.0,627.0,1046.0,1010.0,0.0,17.0]), La
0,0.0,30.0,55.0,55.0,0.0,2.0]), LabeledPoint(3.0, [1.0,7.0,0.0,35.0,47.0,46.0,0.0,2.0]
>>> |
```

-Then the Data Frame is being mapped and stored as Labeled Points where line[9] is the label and the list of all other variables.



```
>>> temp.take(5)
[LabeledPoint(325.0, [187.0,647.0,1.0,1965.0,3242.0,3055.0,8.0,129.0]), LabeledPoint
0,7.0,0.0,0.0,0.0]), LabeledPoint(53.0, [36.0,232.0,0.0,627.0,1046.0,1010.0,0.0,17.0]),
0,0.0,30.0,55.0,55.0,0.0,2.0]), LabeledPoint(3.0, [1.0,7.0,0.0,35.0,47.0,46.0,0.0,2.
>>> from pyspark.mllib.util import MLUtils
>>> from pyspark.mllib.linalg import Vectors
>>> from pyspark.mllib.feature import StandardScaler
>>> features = df.map(lambda row: row[0:8])
>>> features.take(5)
[(u'187', u'647', u'1', u'1965', u'3242', u'3055', u'8', u'129'), (u'7', u'1', u'0',
u'0'), (u'36', u'232', u'0', u'627', u'1046', u'1010', u'0', u'17'), (u'0', u'17', u
u'0', u'2'), (u'1', u'7', u'0', u'35', u'47', u'46', u'0', u'2')]
>>> standardizer = StandardScaler()
>>> model = standardizer.fit(features)
>>> features_transform = model.transform(features)
>>> features_transform.take(5)
[DenseVector([7.1326, 7.8387, 2.1066, 6.5478, 6.6646, 6.9022, 5.0873, 6.9381]), Dens
0, 0.02, 0.0288, 0.0158, 0.0, 0.0]), DenseVector([1.3731, 2.8108, 0.0, 2.0893, 2.150
DenseVector([0.0, 0.206, 0.0, 0.1, 0.1131, 0.1243, 0.0, 0.1076]), DenseVector([0.038
0966, 0.1039, 0.0, 0.1076])]
>>> lab = df.map(lambda row: row[9])
>>> lab.take(5)
[u'325', u'7', u'53', u'2', u'3']
>>> transformed
```

-Here the predictors were computed.



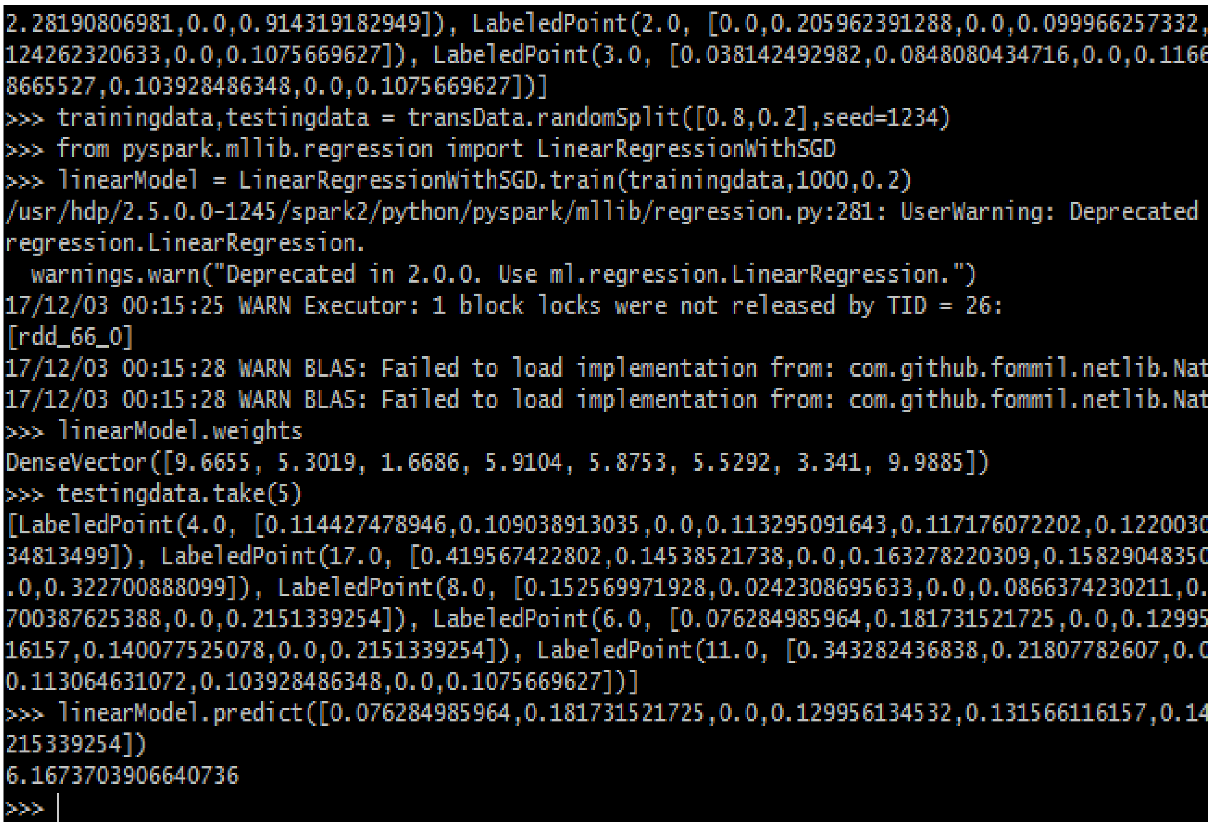
```
>>> transData = transformedData.map(lambda row: LabeledPoint(row[0],[row[1]]))
>>> transData.take(5)
[LabeledPoint(325.0, [7.13264618764,7.83868630374,2.10662313549,6.54778985525,6.6646460715
729604943,6.93806909414]), LabeledPoint(7.0, [0.266997450874,0.0121154347817,0.0,0.0199932
93,0.0158152044442,0.0,0.0]), LabeledPoint(53.0, [1.37312974735,2.81078086935,0.0,2.089294
2.28190806981,0.0,0.914319182949]), LabeledPoint(2.0, [0.0,0.205962391288,0.0,0.099966257
124262320633,0.0,0.1075669627]), LabeledPoint(3.0, [0.038142492982,0.0848080434716,0.0,0.1
8665527,0.103928486348,0.0,0.1075669627])]
>>>
```

-Finally we received the final labeled points in transData which we will use for analysis.

ANALYSIS:

1) Linear regression:

- Here for Linear regression, we imported the package LinearRegressionWithSGD.
- Also the functions here used are LinearRegressionWithSGD.train( )
- When fitting Linear Regression Model without intercept on dataset with constant nonzero column by “l-bfgs” solver, we observed that Spark MLlib outputs zero coefficients for constant nonzero columns.
- This kind of behavior is quite similar to R glmnet but different from LIBSVM.
- Further we computed the weights of the linear model which resulted into a dense vector.
- And at last we predicted the linear model using predict( ) function.



```
2.28190806981,0.0,0.914319182949]), LabeledPoint(2.0, [0.0,0.205962391288,0.0,0.099966257332,
124262320633,0.0,0.1075669627]), LabeledPoint(3.0, [0.038142492982,0.0848080434716,0.0,0.1166
8665527,0.103928486348,0.0,0.1075669627])]
>>> trainingdata,testingdata = transData.randomSplit([0.8,0.2],seed=1234)
>>> from pyspark.mllib.regression import LinearRegressionWithSGD
>>> linearModel = LinearRegressionWithSGD.train(trainingdata,1000,0.2)
/usr/hdp/2.5.0.0-1245/spark2/python/pyspark/mllib/regression.py:281: UserWarning: Deprecated
regression.LinearRegression.
  warnings.warn("Deprecated in 2.0.0. Use ml.regression.LinearRegression.")
17/12/03 00:15:25 WARN Executor: 1 block locks were not released by TID = 26:
[rdd_66_0]
17/12/03 00:15:28 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.Nat
17/12/03 00:15:28 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.Nat
>>> linearModel.weights
DenseVector([9.6655, 5.3019, 1.6686, 5.9104, 5.8753, 5.5292, 3.341, 9.9885])
>>> testingdata.take(5)
[LabeledPoint(4.0, [0.114427478946,0.109038913035,0.0,0.113295091643,0.117176072202,0.1220030
34813499]), LabeledPoint(17.0, [0.419567422802,0.14538521738,0.0,0.163278220309,0.1582904835
.0,0.322700888099]), LabeledPoint(8.0, [0.152569971928,0.0242308695633,0.0,0.0866374230211,0.
700387625388,0.0,0.2151339254]), LabeledPoint(6.0, [0.076284985964,0.181731521725,0.0,0.12995
16157,0.140077525078,0.0,0.2151339254]), LabeledPoint(11.0, [0.343282436838,0.21807782607,0.0
0.113064631072,0.103928486348,0.0,0.1075669627])]
>>> linearModel.predict([0.076284985964,0.181731521725,0.0,0.129956134532,0.131566116157,0.14
215339254])
6.1673703906640736
>>> |
```

-Since we observed that, there were certain more predictors, so rather than concluding from single predictor, we further carried out our analysis over other linear and non-linear models.

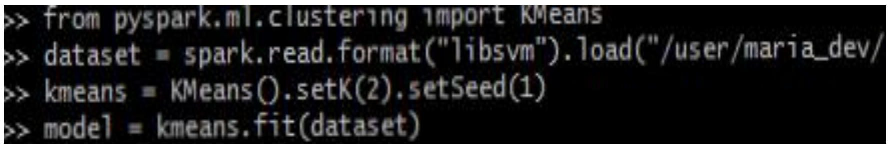
2) K-Means Clustering:

- As Described earlier K-means algorithm identifies *k* number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible

- The MLlib implementation of k-means includes a parallelized variant of the k-means++ method called kmeans||.

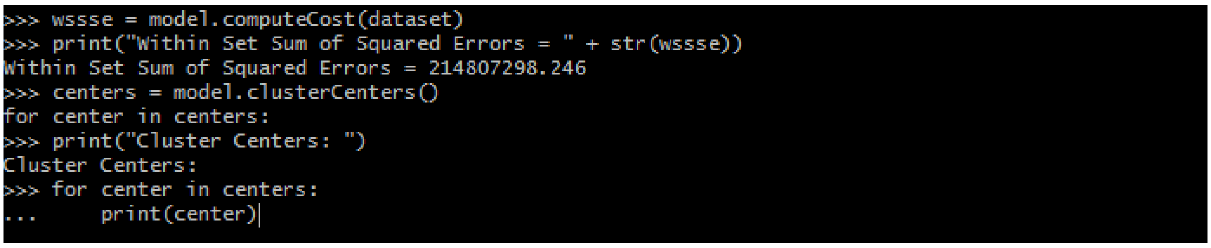
-KMeans is implemented as an Estimator and generates a KMeansModel as the base model.

- The libraries imported here are KMeans.



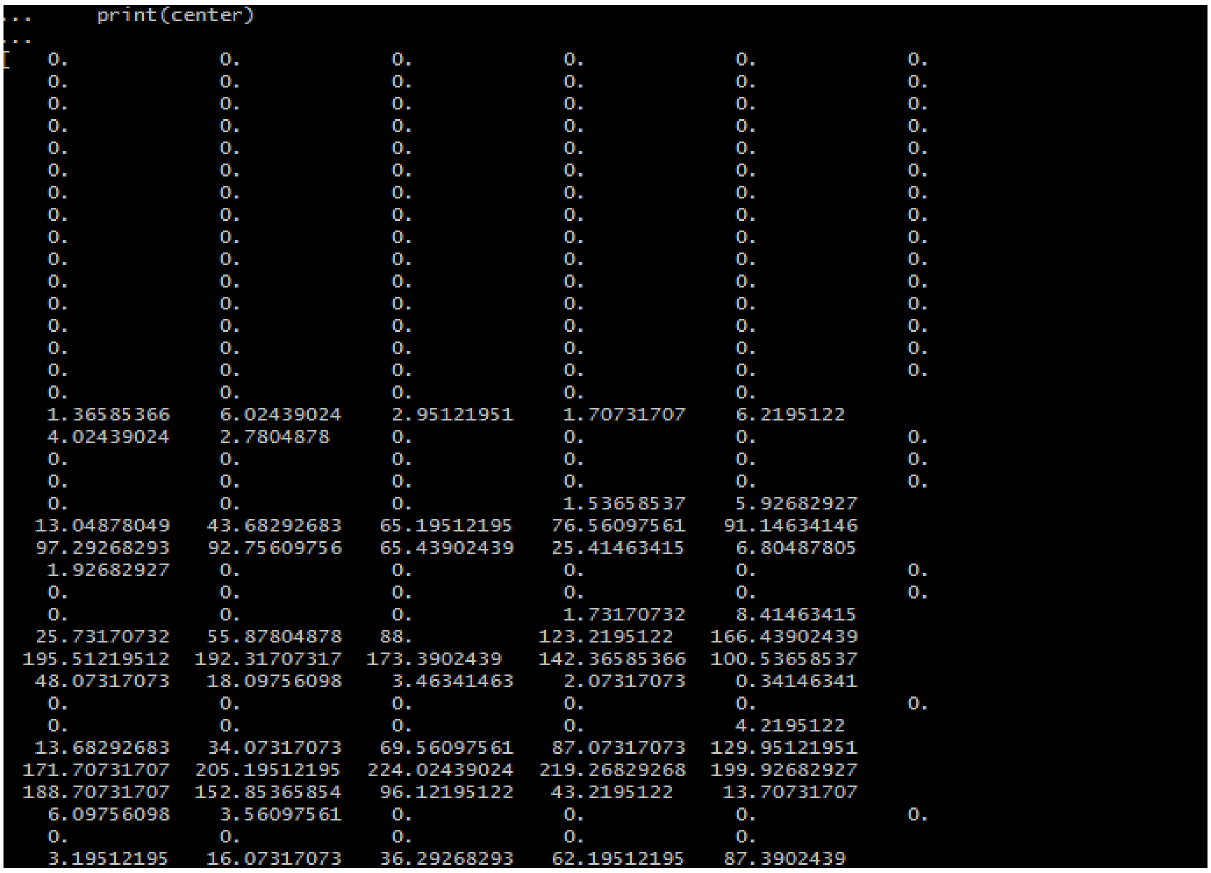
```
>> from pyspark.ml.clustering import KMeans
>> dataset = spark.read.format("libsvm").load("/user/maria_dev/
>> kmeans = KMeans().setK(2).setSeed(1)
>> model = kmeans.fit(dataset)
```

-The functions used in the above method are KMeans( ).setK( ).setSeed( ), Kmeans.fit( )



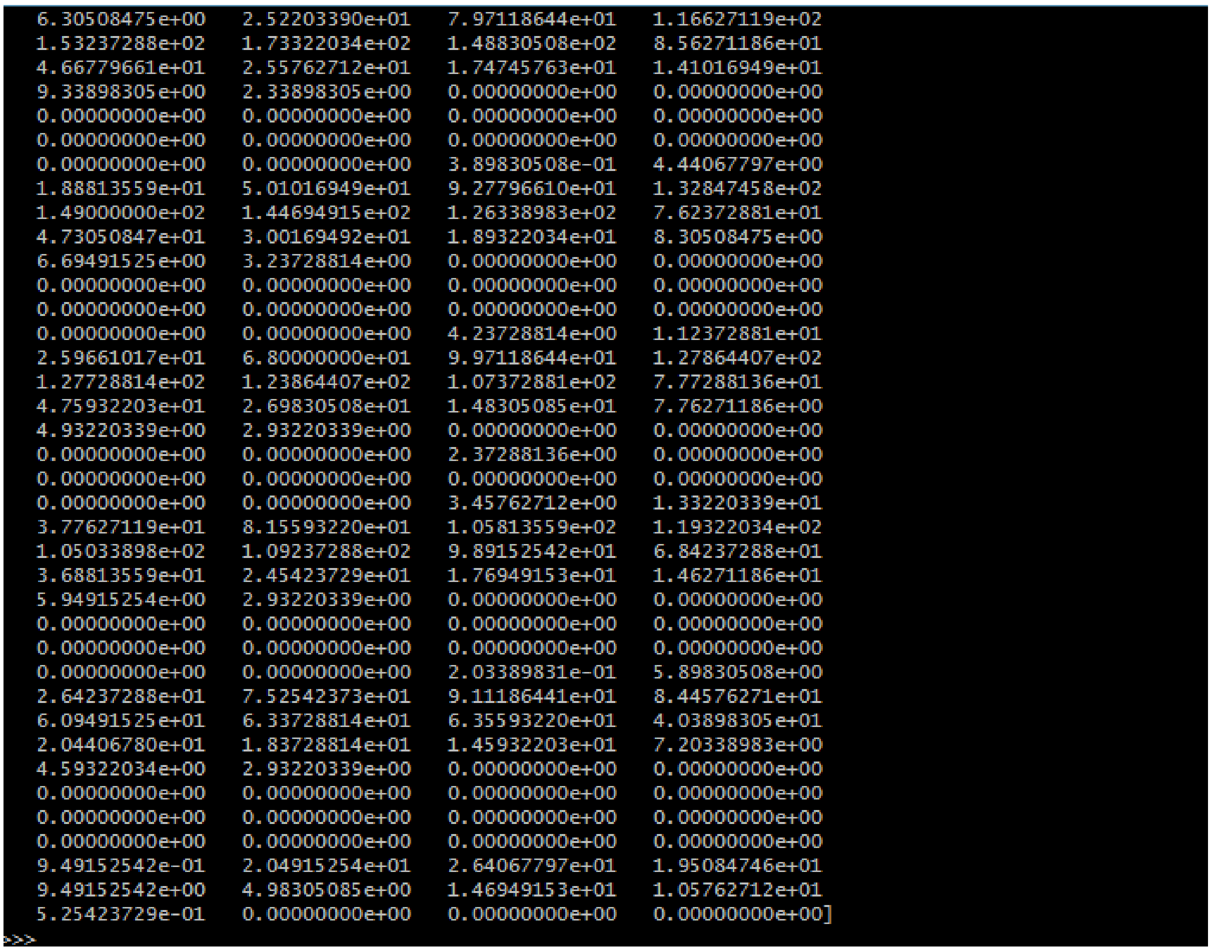
```
>>> wssse = model.computeCost(dataset)
>>> print("Within Set Sum of Squared Errors = " + str(wssse))
Within Set Sum of Squared Errors = 214807298.246
>>> centers = model.clusterCenters()
for center in centers:
>>> print("Cluster Centers: ")
Cluster Centers:
>>> for center in centers:
...     print(center)
```

-Then we are computing the wssse cost and printing the sum of squared errors.



```
...     print(center)
...
[   0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.
    1.36585366    6.02439024    2.95121951    1.70731707    6.2195122
    4.02439024    2.7804878     0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            1.53658537    5.92682927
   13.04878049   43.68292683   65.19512195   76.56097561   91.14634146
   97.29268293   92.75609756   65.43902439   25.41463415    6.80487805
    1.92682927    0.            0.            0.            0.            0.
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            1.73170732    8.41463415
   25.73170732   55.87804878   88.          123.2195122   166.43902439
  195.51219512  192.31707317  173.3902439   142.36585366  100.53658537
   48.07317073   18.09756098    3.46341463    2.07317073    0.34146341
    0.            0.            0.            0.            0.            0.
    0.            0.            0.            0.            4.2195122
   13.68292683   34.07317073   69.56097561   87.07317073  129.95121951
  171.70731707  205.19512195  224.02439024  219.26829268  199.92682927
  188.70731707  152.85365854   96.12195122   43.2195122    13.70731707
    6.09756098    3.56097561    0.            0.            0.            0.
    0.            0.            0.            0.            0.
    3.19512195   16.07317073   36.29268293   62.19512195   87.3902439
```

-Here we circumscribe the model.clusterCenters( ) to find the cluster centers and print them.



```
   6.30508475e+00   2.52203390e+01   7.97118644e+01   1.16627119e+02
   1.53237288e+02   1.73322034e+02   1.48830508e+02   8.56271186e+01
   4.66779661e+01   2.55762712e+01   1.74745763e+01   1.41016949e+01
   9.33898305e+00   2.33898305e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   3.89830508e-01   4.44067797e+00
   1.88813559e+01   5.01016949e+01   9.27796610e+01   1.32847458e+02
   1.49000000e+02   1.44694915e+02   1.26338983e+02   7.62372881e+01
   4.73050847e+01   3.00169492e+01   1.89322034e+01   8.30508475e+00
   6.69491525e+00   3.23728814e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   4.23728814e+00   1.12372881e+01
   2.59661017e+01   6.80000000e+01   9.97118644e+01   1.27864407e+02
   1.27728814e+02   1.23864407e+02   1.07372881e+02   7.77288136e+01
   4.75932203e+01   2.69830508e+01   1.48305085e+01   7.76271186e+00
   4.93220339e+00   2.93220339e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   2.37288136e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   3.45762712e+00   1.33220339e+01
   3.77627119e+01   8.15593220e+01   1.05813559e+02   1.19322034e+02
   1.05033898e+02   1.09237288e+02   9.89152542e+01   6.84237288e+01
   3.68813559e+01   2.45423729e+01   1.76949153e+01   1.46271186e+01
   5.94915254e+00   2.93220339e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   2.03389831e-01   5.89830508e+00
   2.64237288e+01   7.52542373e+01   9.11186441e+01   8.44576271e+01
   6.09491525e+01   6.33728814e+01   6.35593220e+01   4.03898305e+01
   2.04406780e+01   1.83728814e+01   1.45932203e+01   7.20338983e+00
   4.59322034e+00   2.93220339e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   9.49152542e-01   2.04915254e+01   2.64067797e+01   1.95084746e+01
   9.49152542e+00   4.98305085e+00   1.46949153e+01   1.05762712e+01
   5.25423729e-01   0.00000000e+00   0.00000000e+00   0.00000000e+00]
>>>
```

- Finally we printed the predictive estimates of the value of the crimes.

3) Naïve Bayes:

- Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem.

- Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers.

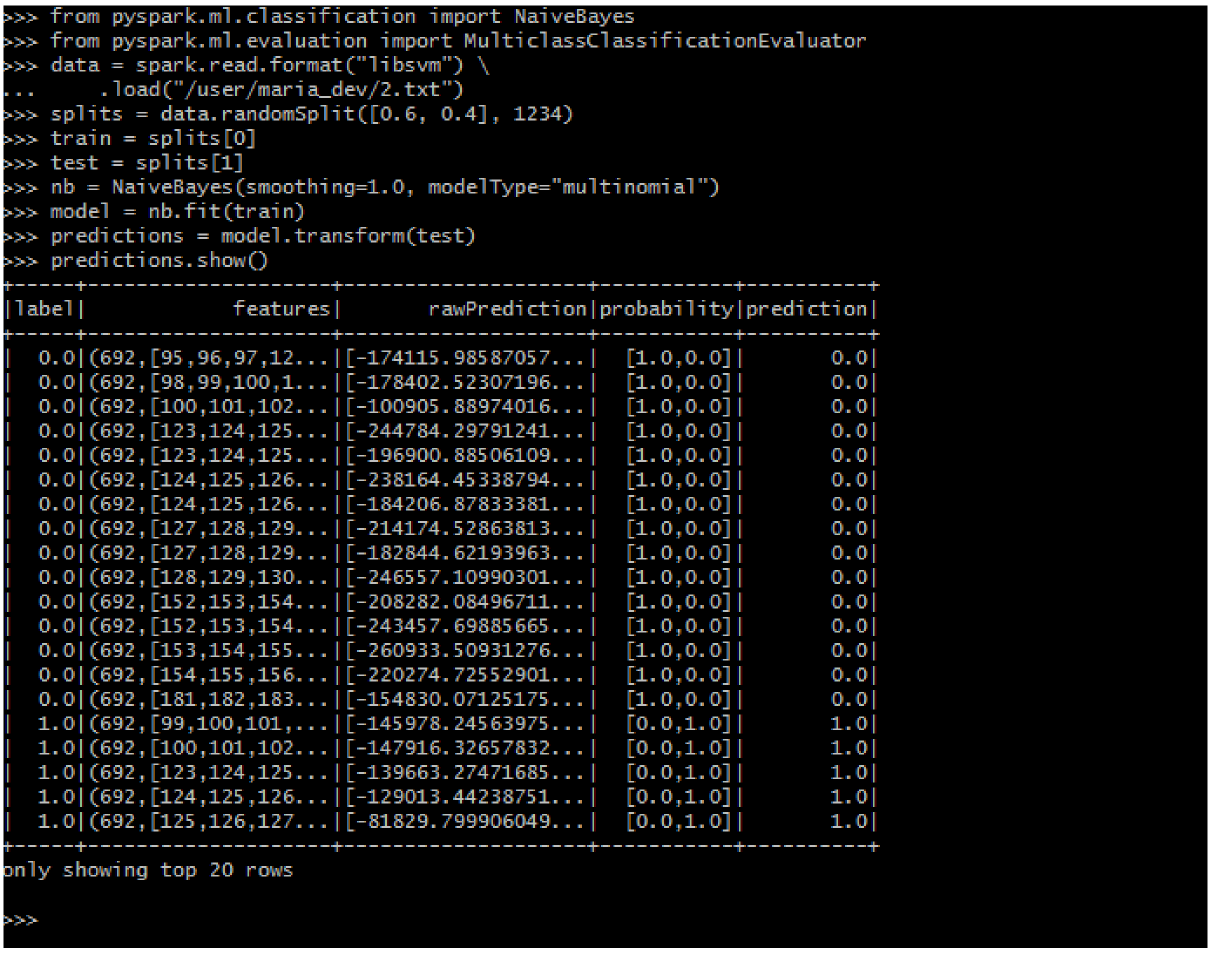
- Based on the Bayes' theorem, this model uses conditional probability theory to classify observations. The NaiveBayes model in PySpark ML supports both binary and multinomial labels

- With appropriate pre-processing, it is competitive in this domain with more advanced methods including support vector machines. It also finds application in automatic medical diagnosis.

- In the statistics and computer science literature, naive Bayes models are known under a variety of names, such as independence Bayes and simple Bayes.

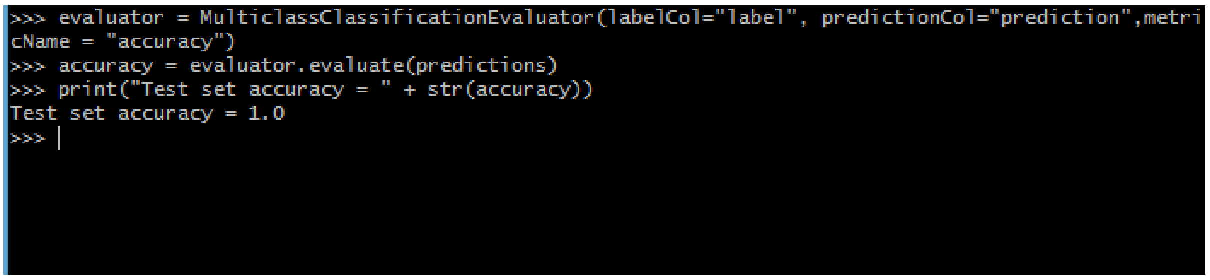
- The packages imported here are NaiveBayes, MulticlassClassificationEvaluator

-The functions which are used here : NaiveBayes( ), model.fit( ), model.transform( )



```
>>> from pyspark.ml.classification import NaiveBayes
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator
>>> data = spark.read.format("libsvm") \
...     .load("/user/maria_dev/2.txt")
>>> splits = data.randomSplit([0.6, 0.4], 1234)
>>> train = splits[0]
>>> test = splits[1]
>>> nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
>>> model = nb.fit(train)
>>> predictions = model.transform(test)
>>> predictions.show()
+-----+--------------------+--------------------+-----------+----------+
|label|            features|       rawPrediction|probability|prediction|
+-----+--------------------+--------------------+-----------+----------+
|  0.0|(692,[95,96,97,12...|[-174115.98587057...|  [1.0,0.0]|       0.0|
|  0.0|(692,[98,99,100,1...|[-178402.52307196...|  [1.0,0.0]|       0.0|
|  0.0|(692,[100,101,102...|[-100905.88974016...|  [1.0,0.0]|       0.0|
|  0.0|(692,[123,124,125...|[-244784.29791241...|  [1.0,0.0]|       0.0|
|  0.0|(692,[123,124,125...|[-196900.88506109...|  [1.0,0.0]|       0.0|
|  0.0|(692,[124,125,126...|[-238164.45338794...|  [1.0,0.0]|       0.0|
|  0.0|(692,[124,125,126...|[-184206.87833381...|  [1.0,0.0]|       0.0|
|  0.0|(692,[127,128,129...|[-214174.52863813...|  [1.0,0.0]|       0.0|
|  0.0|(692,[127,128,129...|[-182844.62193963...|  [1.0,0.0]|       0.0|
|  0.0|(692,[128,129,130...|[-246557.10990301...|  [1.0,0.0]|       0.0|
|  0.0|(692,[152,153,154...|[-208282.08496711...|  [1.0,0.0]|       0.0|
|  0.0|(692,[152,153,154...|[-243457.69885665...|  [1.0,0.0]|       0.0|
|  0.0|(692,[153,154,155...|[-260933.50931276...|  [1.0,0.0]|       0.0|
|  0.0|(692,[154,155,156...|[-220274.72552901...|  [1.0,0.0]|       0.0|
|  0.0|(692,[181,182,183...|[-154830.07125175...|  [1.0,0.0]|       0.0|
|  1.0|(692,[99,100,101,...|[-145978.24563975...|  [0.0,1.0]|       1.0|
|  1.0|(692,[100,101,102...|[-147916.32657832...|  [0.0,1.0]|       1.0|
|  1.0|(692,[123,124,125...|[-139663.27471685...|  [0.0,1.0]|       1.0|
|  1.0|(692,[124,125,126...|[-129013.44238751...|  [0.0,1.0]|       1.0|
|  1.0|(692,[125,126,127...|[-81829.799906049...|  [0.0,1.0]|       1.0|
+-----+--------------------+--------------------+-----------+----------+
only showing top 20 rows

>>>
```

-Here we are calculating the predictions.



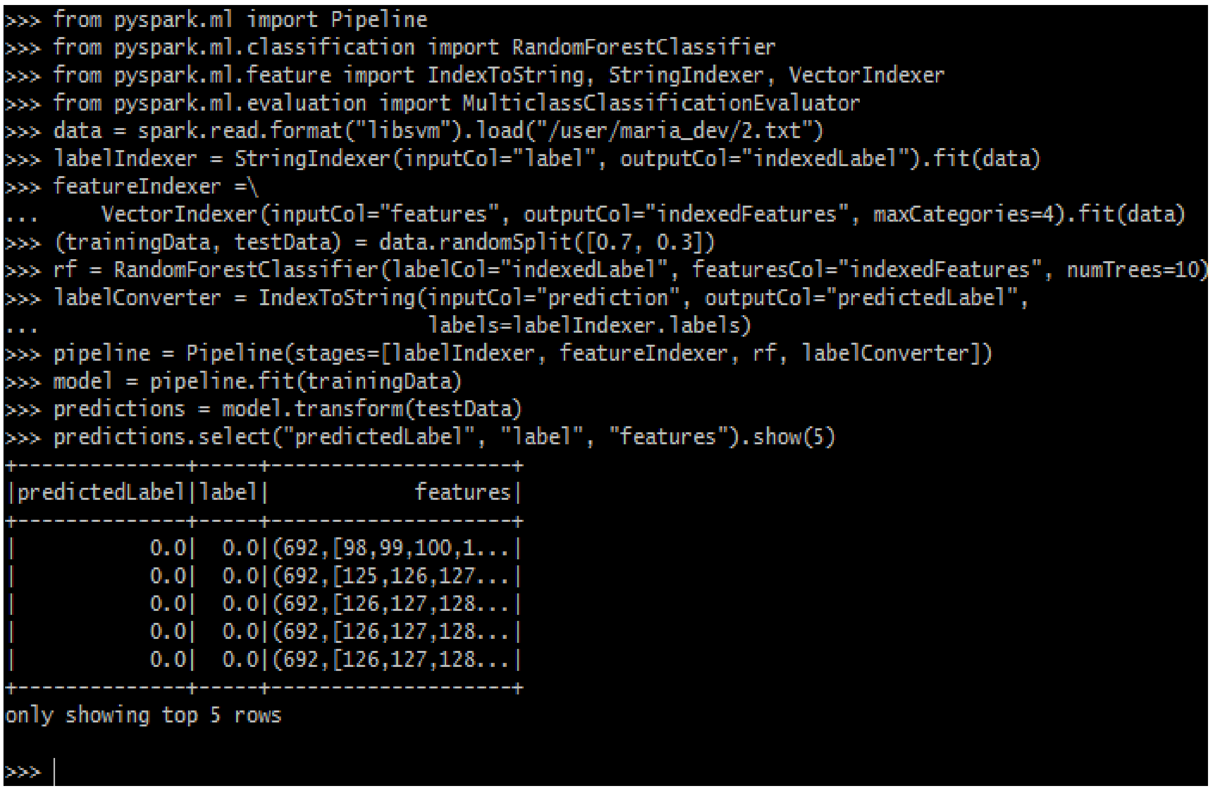
```
>>> evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction",metri
cName = "accuracy")
>>> accuracy = evaluator.evaluate(predictions)
>>> print("Test set accuracy = " + str(accuracy))
Test set accuracy = 1.0
>>> |
```

-Then we are using the evaluator function.

-Furth calculating the accuracy of the predictions made and printing the test set accuracy which turned out to be 1.0.

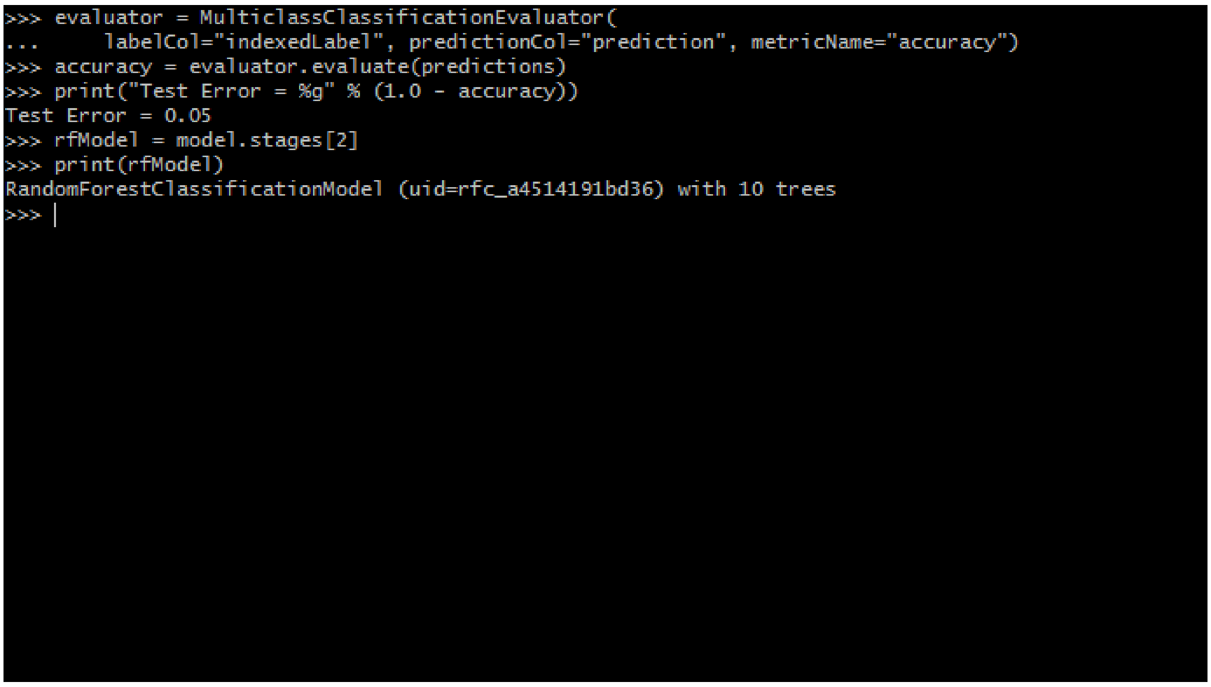
4) Random Forest Classifier

-Random forests are a popular family of classification and regression methods.
-This model produces multiple decision trees (hence the name—forest) and uses the mode output of those decision trees to classify observations. The RandomForestClassifier supports both binary and multinomial labels
- Here we first load our dataset in LibSVM format, split it into two sets : training and test sets, then we trained our dataset, and then evaluated it on the held-out test set.
-Here we used two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the DataFrame which the tree-based algorithms can recognize.
- The packages used here are Pipeline, RandomForestClassifier, IndexToString, StringIndexer, VectorIndexer.
-The functions used here are RandomClassifier( ), IndexToClassifier( ), Pipeline.fit( )



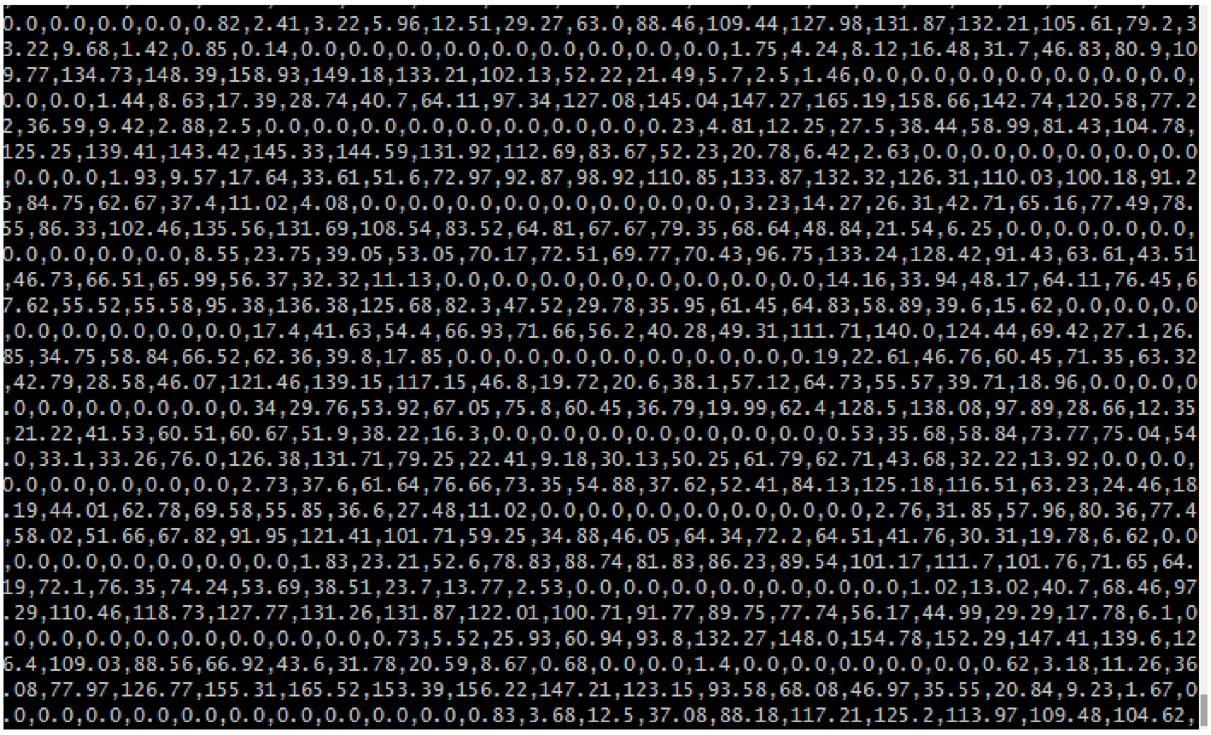
```
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.classification import RandomForestClassifier
>>> from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
>>> from pyspark.ml.evaluation import MulticlassClassificationEvaluator
>>> data = spark.read.format("libsvm").load("/user/maria_dev/2.txt")
>>> labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
>>> featureIndexer =\
...     VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)
>>> (trainingData, testData) = data.randomSplit([0.7, 0.3])
>>> rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)
>>> labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
...                                labels=labelIndexer.labels)
>>> pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])
>>> model = pipeline.fit(trainingData)
>>> predictions = model.transform(testData)
>>> predictions.select("predictedLabel", "label", "features").show(5)
+--------------+-----+--------------------+
|predictedLabel|label|            features|
+--------------+-----+--------------------+
|           0.0|  0.0|(692,[98,99,100,1...|
|           0.0|  0.0|(692,[125,126,127...|
|           0.0|  0.0|(692,[126,127,128...|
|           0.0|  0.0|(692,[126,127,128...|
|           0.0|  0.0|(692,[126,127,128...|
+--------------+-----+--------------------+
only showing top 5 rows

>>> 
```

- Here we calculated the LabelIndexer, using the classifier function for prediction.



```
>>> evaluator = MulticlassClassificationEvaluator(
...     labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
>>> accuracy = evaluator.evaluate(predictions)
>>> print("Test Error = %g" % (1.0 - accuracy))
Test Error = 0.05
>>> rfModel = model.stages[2]
>>> print(rfModel)
RandomForestClassificationModel (uid=rfc_a4514191bd36) with 10 trees
>>> 
```

- Further in order to calculate accuracy we used evaluation index and the printed the test error. - Further we calculated the stages using rf ( ).



```
0.0,0.0,0.0,0.0,0.82,2.41,3.22,5.96,12.51,29.27,63.0,88.46,109.44,127.98,131.87,132.21,105.61,79.2,3
3.22,9.68,1.42,0.85,0.14,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.75,4.24,8.12,16.48,31.7,46.83,80.9,10
9.77,134.73,148.39,158.93,149.18,133.21,102.13,52.22,21.49,5.7,2.5,1.46,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,1.44,8.63,17.39,28.74,40.7,64.11,97.34,127.08,145.04,147.27,165.19,158.66,142.74,120.58,77.2
2,36.59,9.42,2.88,2.5,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.23,4.81,12.25,27.5,38.44,58.99,81.43,104.78,
125.25,139.41,143.42,145.33,144.59,131.92,112.69,83.67,52.23,20.78,6.42,2.63,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0,1.93,9.57,17.64,33.61,51.6,72.97,92.87,98.92,110.85,133.87,132.32,126.31,110.03,100.18,91.2
5,84.75,62.67,37.4,11.02,4.08,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,3.23,14.27,26.31,42.71,65.16,77.49,78.
55,86.33,102.46,135.56,131.69,108.54,83.52,64.81,67.67,79.35,68.64,48.84,21.54,6.25,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,8.55,23.75,39.05,53.05,70.17,72.51,69.77,70.43,96.75,133.24,128.42,91.43,63.61,43.51
,46.73,66.51,65.99,56.37,32.32,11.13,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,14.16,33.94,48.17,64.11,76.45,6
7.62,55.52,55.58,95.38,136.38,125.68,82.3,47.52,29.78,35.95,61.45,64.83,58.89,39.6,15.62,0.0,0.0,0.0
,0.0,0.0,0.0,0.0,0.0,17.4,41.63,54.4,66.93,71.66,56.2,40.28,49.31,111.71,140.0,124.44,69.42,27.1,26.
85,34.75,58.84,66.52,62.36,39.8,17.85,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.19,22.61,46.76,60.45,71.35,63.32
,42.79,28.58,46.07,121.46,139.15,117.15,46.8,19.72,20.6,38.1,57.12,64.73,55.57,39.71,18.96,0.0,0.0,0
.0,0.0,0.0,0.0,0.0,0.34,29.76,53.92,67.05,75.8,60.45,36.79,19.99,62.4,128.5,138.08,97.89,28.66,12.35
,21.22,41.53,60.51,60.67,51.9,38.22,16.3,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.53,35.68,58.84,73.77,75.04,54
.0,33.1,33.26,76.0,126.38,131.71,79.25,22.41,9.18,30.13,50.25,61.79,62.71,43.68,32.22,13.92,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,2.73,37.6,61.64,76.66,73.35,54.88,37.62,52.41,84.13,125.18,116.51,63.23,24.46,18
.19,44.01,62.78,69.58,55.85,36.6,27.48,11.02,0.0,0.0,0.0,0.0,0.0,0.0,0.0,2.76,31.85,57.96,80.36,77.4
,58.02,51.66,67.82,91.95,121.41,101.71,59.25,34.88,46.05,64.34,72.2,64.51,41.76,30.31,19.78,6.62,0.0
,0.0,0.0,0.0,0.0,0.0,0.0,1.83,23.21,52.6,78.83,88.74,81.83,86.23,89.54,101.17,111.7,101.76,71.65,64.
19,72.1,76.35,74.24,53.69,38.51,23.7,13.77,2.53,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.02,13.02,40.7,68.46,97
.29,110.46,118.73,127.77,131.26,131.87,122.01,100.71,91.77,89.75,77.74,56.17,44.99,29.29,17.78,6.1,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.73,5.52,25.93,60.94,93.8,132.27,148.0,154.78,152.29,147.41,139.6,12
6.4,109.03,88.56,66.92,43.6,31.78,20.59,8.67,0.68,0.0,0.0,1.4,0.0,0.0,0.0,0.0,0.0,0.62,3.18,11.26,36
.08,77.97,126.77,155.31,165.52,153.39,156.22,147.21,123.15,93.58,68.08,46.97,35.55,20.84,9.23,1.67,0
.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.83,3.68,12.5,37.08,88.18,117.21,125.2,113.97,109.48,104.62,
```

CONCLUSION:

-The project's objective was to analyse the crime data set and provide productive information in order to reduce crime rates.

-This analysis is also useful for the department of security, so they can protect those affected regions and reduce crime rates by taking some strict actions.

-Here, we carried out our analysis using linear models like regression, non linear models like naïve bayes and Random forest and also using classification (unsupervised model) models like clustering.

-From the analysis perspective, We found that random forest and naïve bayes models came up with good amount of accuracy and these models fit our data with less error rate while linear model didn't perform so well.

- Thus, at last , I could discern that if proper pattern recognition tools and learning can be applied over our analysis, one can get enormous constructive results.

REFRENCES:

http://digitalsteampunk.blogspot.com/2015/11/k-nearest-neighbour-classification-in.html

https://mapr.com/blog/churn-prediction-pyspark-using-mllib-and-ml-packages/

https://spark.apache.org/docs/latest/ml-guide.html

https://stackoverflow.com/questions/43920111/convert-dataframe-to-libsvm-format

https://spark.apache.org/docs/latest/mllib-ensembles.html

https://spark.apache.org/docs/2.2.0/mllib-naive-bayes.html