Project Report

Project 2

Big Data Analytics

Kulsoom Khan

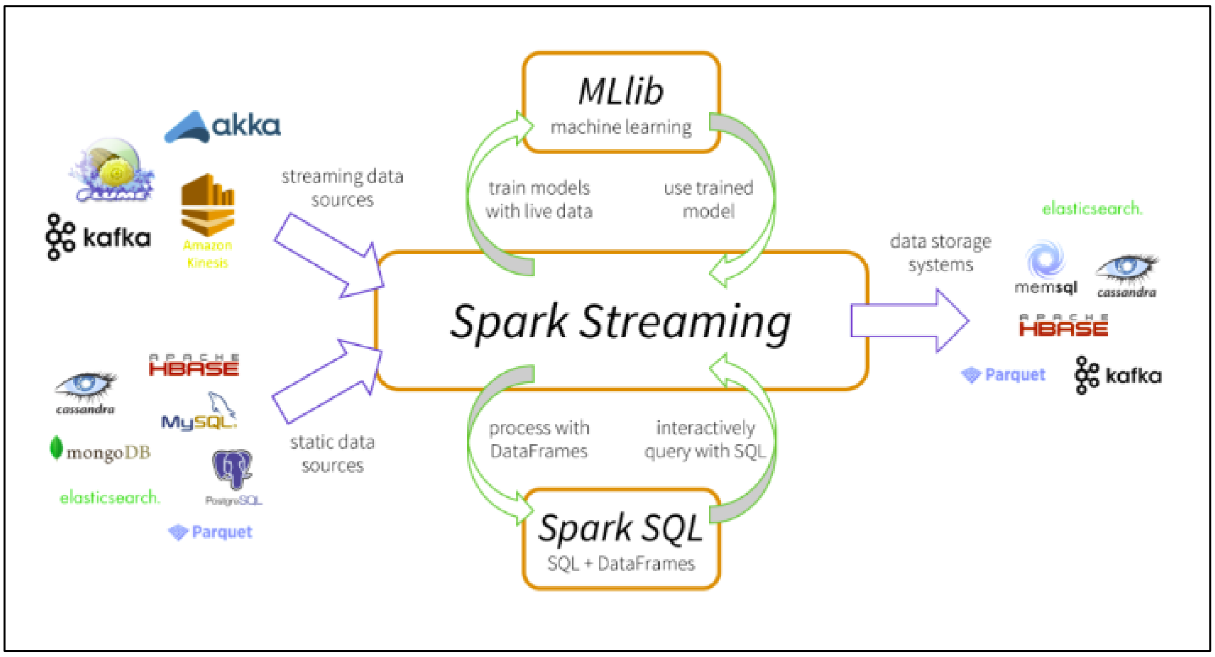
25375

Introduction

This report discusses the findings of the project has been done as part of my course for MS Data Science at IBA. The goal was to compare the performance of Spark Mllib and Apache Mahout.

Apache Spark

Apache Spark is a data processing framework that can quickly perform processing tasks on very large data sets and can also distribute data processing tasks across multiple computers, either on its own or in tandem with other distributed computing tools. It is a lightning-fast unified analytics engine for big data and machine learning. Apache Spark is the most developed library that you can utilize for many of your Machine Learning applications. It provides the users with the ease of developing ML-based algorithms in data scientist's favorite scientific prototyping environment Jupyter Notebooks.



What is Mllib?

Spark has the ability to perform machine learning at scale with a built-in library called MLlib. Spark MLlib is used to perform machine learning in Apache Spark. MLlib consists of popular algorithms and utilities. MLlib in Spark is a scalable Machine learning library that discusses both high-quality algorithm and high speed. The machine learning algorithms like regression, classification, clustering, pattern mining, and collaborative filtering. Lower level machine learning primitives like generic gradient descent optimization algorithm are also present in MLlib.

Spark.ml is the primary Machine Learning API for Spark. The library Spark.ml offers a higher-level API built on top of DataFrames for constructing ML pipelines. The MLlib API, although not as inclusive as scikit-learn, can be used for classification, regression and clustering problems. In the proceeding report, we'll train machine learning models Spark.

Spark MLlib & The Types of Algorithms That Are Available

We can do both supervised and unsupervised machine learning operations with Spark. Supervised Learning has labelled data already whereas unsupervised learning does not have labelled data and thus it is more akin to seeking patterns in chaos.

- Supervised Learning Algorithms → Classification, Regression, Gradient Boosting. The algorithms in this category have both the features column and the labels columns.
- Unsupervised Learning Algorithms → K-means clustering, Value Decomposition, Self-organizing maps, Nearest Neighbours Mapping. With this class of algorithms you only have the features column available and no labels.

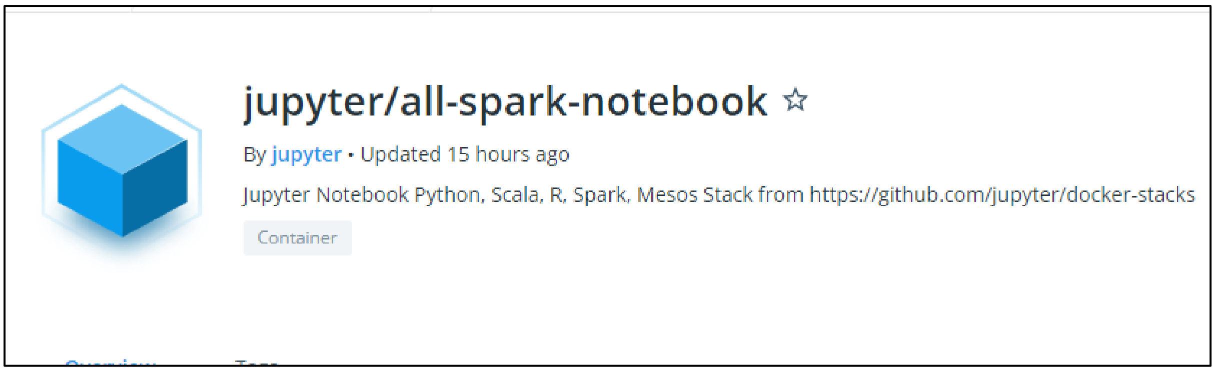
Basics of Model Building

MLlib (Machine Learning library) is heavily based on Scikit-learn's ideas on pipelines. In this library to create an ML model the basics concepts are:

- ***DataFrame***: This ML API uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions.
- ***Transformer***: A Transformer is an algorithm that can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer that transforms a DataFrame with features into a DataFrame with predictions.
- ***Estimator***: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model
- ***Pipeline***: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow
- ***Parameter***: All Transformers and Estimators now share a common API for specifying parameters.

Setup

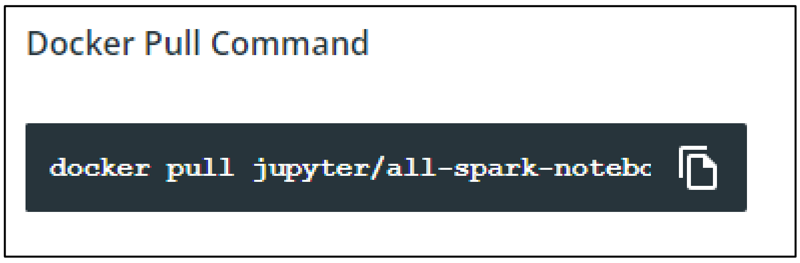
The easiest way to start using Spark is to use the Docker image provided by Jupyter.


jupyter/all-spark-notebook ☆

By jupyter • Updated 15 hours ago

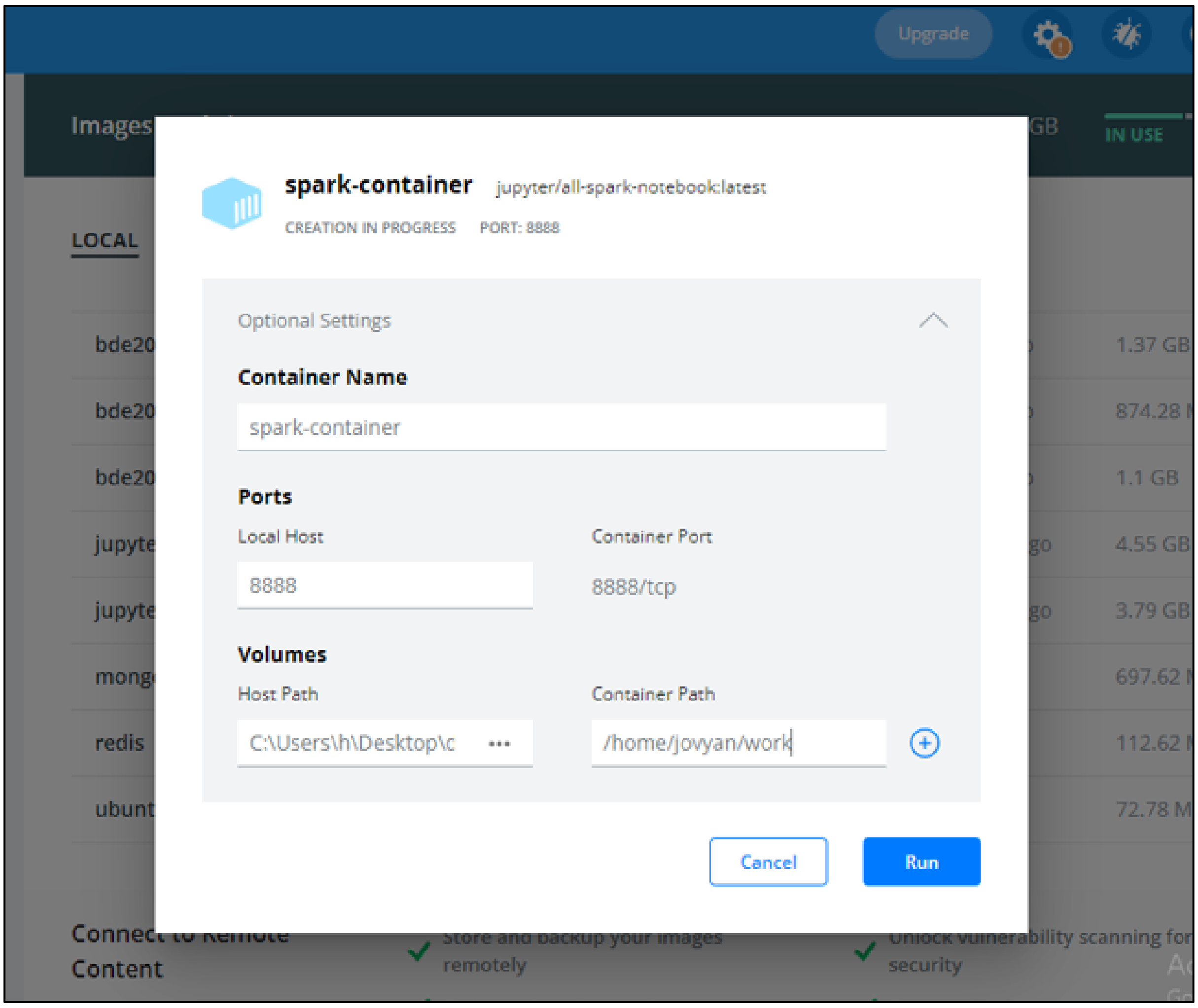
Jupyter Notebook Python, Scala, R, Spark, Mesos Stack from https://github.com/jupyter/docker-stacks

We will pull the docker image and then run a container with our specifications.


Docker Pull Command

```
docker pull jupyter/all-spark-notebc
```

Once we have pulled the image we will make run our container.


spark-container jupyter/all-spark-notebook:latest

CREATION IN PROGRESS PORT: 8888

Optional Settings

Container Name: spark-container

Ports

Local Host	Container Port
8888	8888/tcp

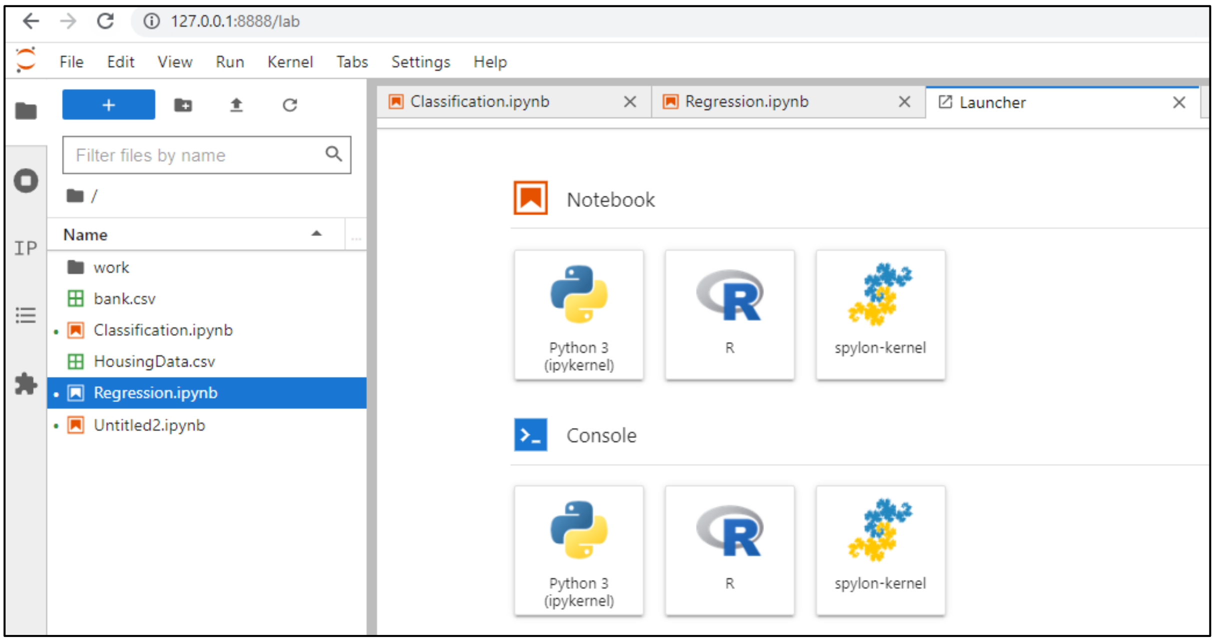
Volumes

Host Path	Container Path
C:\Users\h\Desktop\c	/home/jovyan/work

You can name the container whatever you want. The host path should be on your local machine while the container path should remain the same.

Make sure to modify the host path to match the directory that contains the data you want to work with.

To access the Jupyter Notebook, open a browser and go to localhost:8888. The following window should open up:

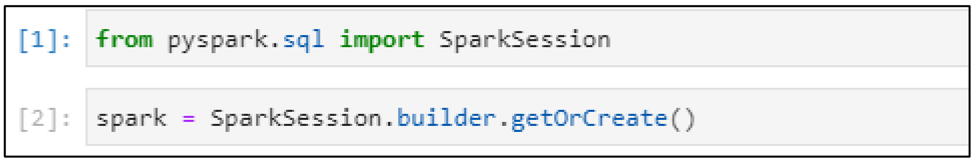


You can create a notebook or a console from this page.

Machine Learning with Spark Mllib

Loading the data into Spark

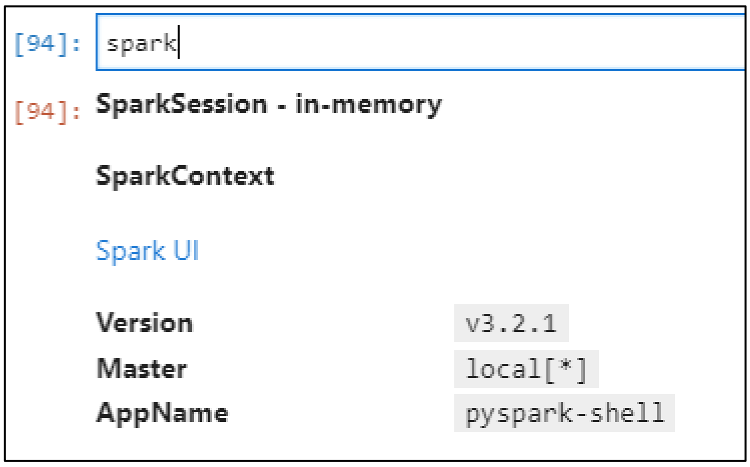
To load the data we are using Spark DataFrames. Spark it’s a little bit more complicated than Pandas. You can’t just do “import -> read_csv()”. You first need to start a Spark Session, to do that write:



```
[1]: from pyspark.sql import SparkSession

[2]: spark = SparkSession.builder.getOrCreate()
```

Import the SparkSession module from pyspark.sql and build a SparkSession with the builder() method. Afterwards, you can set the master URL to connect to, the application name, add some additional configuration like the executor memory and then lastly, use getOrCreate() to either get the current Spark session or to create one if there is none running. We can use this to read multiple types of files, such as CSV, JSON, TEXT, etc. This enables us to save the data as a Spark dataframe.

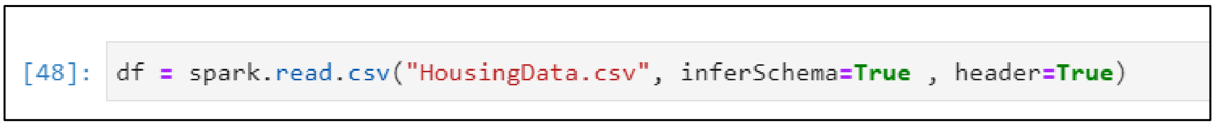
Now if you write “spark” on your notebook, you should get:



```
[94]: spark

[94]: SparkSession - in-memory

      SparkContext

      Spark UI

      Version      v3.2.1
      Master       local[*]
      AppName      pyspark-shell
```

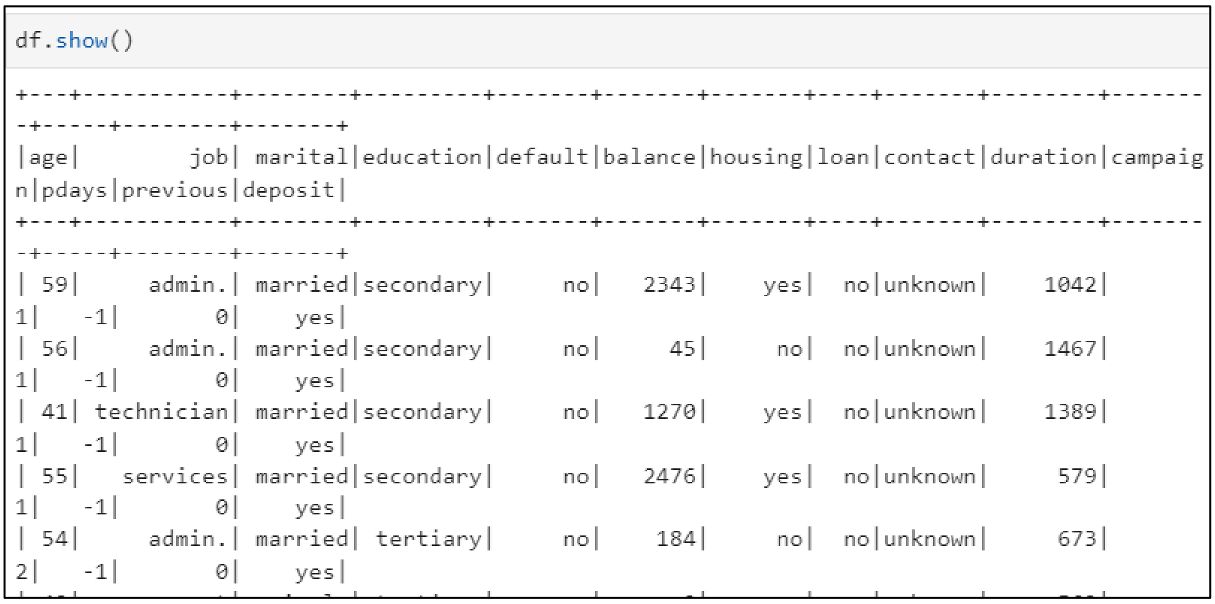
That means that you are using Spark locally with all the cores (that’s the *), with version 3.2.1 and the name of the session is “pyspark-shell”.

Like Pandas, Spark provides an API for loading the contents of a csv file into our program. So now we have everything in place to read the data. To do so write:



```
[48]: df = spark.read.csv("HousingData.csv", inferSchema=True , header=True)
```

We have created a first Spark DataFrame. To see the internals of the DataFrame write:



```
df.show()

+---+----------+-------+---------+-------+-------+-------+----+-------+--------+-------
-+-----+-------+-------+
|age|       job| marital|education|default|balance|housing|loan|contact|duration|campaig
n|pdays|previous|deposit|
+---+----------+-------+---------+-------+-------+-------+----+-------+--------+-------
-+-----+-------+-------+
| 59|    admin.| married|secondary|     no|   2343|    yes|  no|unknown|    1042|
1|   -1|       0|    yes|
| 56|    admin.| married|secondary|     no|     45|     no|  no|unknown|    1467|
1|   -1|       0|    yes|
| 41| technician| married|secondary|     no|   1270|    yes|  no|unknown|    1389|
1|   -1|       0|    yes|
| 55|   services| married|secondary|     no|   2476|    yes|  no|unknown|     579|
1|   -1|       0|    yes|
| 54|    admin.| married| tertiary|     no|    184|     no|  no|unknown|     673|
2|   -1|       0|    yes|
```

You can check the data types by using the printSchema function on the dataframe:

```
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- deposit: string (nullable = true)
```

Handling Categorical Variables

Most machine learning algorithms accept the data only in numerical form. The Spark ML library also only works with numeric data. So, it is essential to convert any categorical variables present in our dataset into numbers.

String Indexing

String Indexing is similar to Label Encoding. It assigns a unique integer value to each category. 0 is assigned to the most frequent category, 1 to the next most frequent value, and so on. We have to define the input column name that we want to index and the output column name in which we want the results.

```
indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
             for c in categoricalCols ]
```

One-Hot Encoding

Spark's OneHotEncoder does not directly encode the categorical variable. First, we need to use the String Indexer to convert the variable into numerical form and then use OneHotEncoderEstimator to encode multiple columns of the dataset. It creates a Sparse Vector for each row.

```
# default setting: dropLast=True
encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
             outputCol="{0}_encoded".format(indexer.getOutputCol()))
             for indexer in indexers ]
```

Vector Assembler

A vector assembler combines a given list of columns into a single vector column. This is typically used at the end of the data exploration and pre-processing steps. At this stage, we usually work with a few raw or transformed features that can be used to train our model. The Vector Assembler converts them into a single feature column in order to train the machine learning model (such as Logistic Regression). It accepts numeric, boolean and vector type columns.

```
assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                            + continuousCols, outputCol="features")
```

We have made a user defined function that implements all three techniques mentioned above at once. But first we separate the continuous, categorical and target variable in our dataset.

```
catcols = ['job','marital','education','default',
           'housing','loan','contact']

num_cols = ['balance', 'duration','campaign','pdays','previous',]
labelCol = 'deposit'
```

Building Machine Learning Pipelines using PySpark

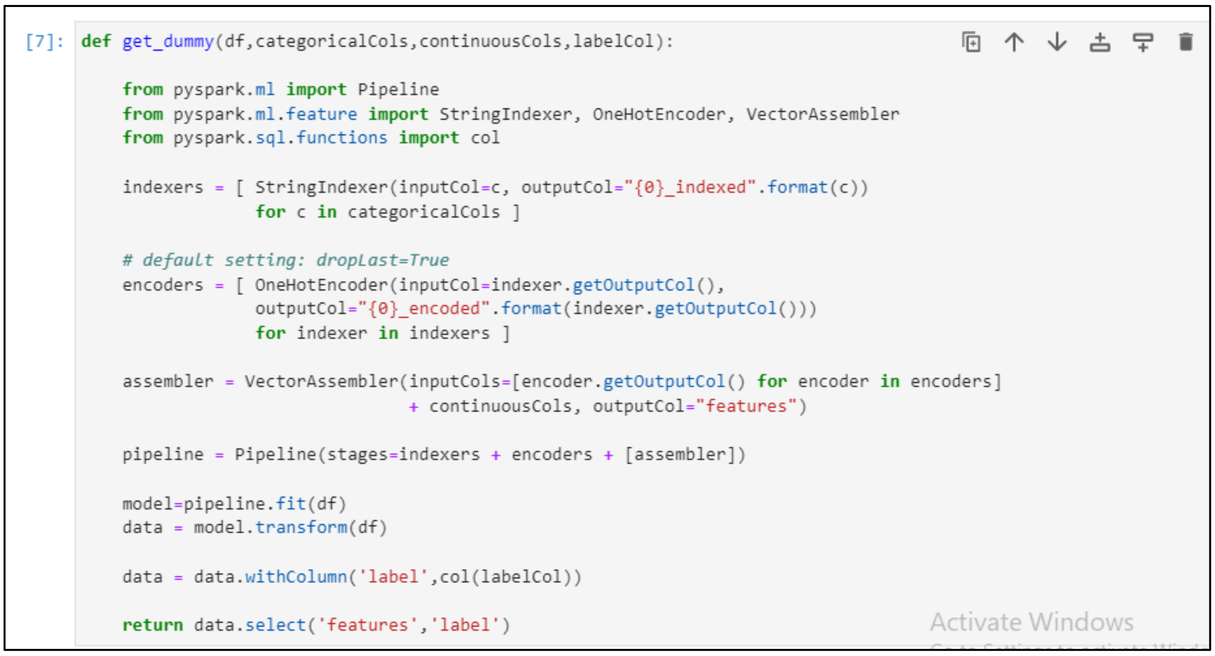
In order to implement the above mentioned code we will be using a pipeline. A machine learning project typically involves steps like data preprocessing, feature extraction, model fitting and evaluating results. We need to perform a lot of transformations on the data in sequence. As you can imagine, keeping track of them can potentially become a tedious task.

This is where machine learning pipelines come in. A pipeline allows us to maintain the data flow of all the relevant transformations that are required to reach the end result. We need to define the stages of the pipeline which act as a chain of command for Spark to run. Here, each stage is either a Transformer or an Estimator. For now we will be using a transformer.

Transformers

As the name suggests, Transformers convert one dataframe into another either by updating the current values of a particular column (like converting categorical columns to numeric) or mapping it to some other values by using a defined logic.

Below is the snapshot of our function:



```
[7]: def get_dummy(df,categoricalCols,continuousCols,labelCol):

         from pyspark.ml import Pipeline
         from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
         from pyspark.sql.functions import col

         indexers = [ StringIndexer(inputCol=c, outputCol="{0}_indexed".format(c))
                      for c in categoricalCols ]

         # default setting: dropLast=True
         encoders = [ OneHotEncoder(inputCol=indexer.getOutputCol(),
                      outputCol="{0}_encoded".format(indexer.getOutputCol()))
                      for indexer in indexers ]

         assembler = VectorAssembler(inputCols=[encoder.getOutputCol() for encoder in encoders]
                                     + continuousCols, outputCol="features")

         pipeline = Pipeline(stages=indexers + encoders + [assembler])

         model=pipeline.fit(df)
         data = model.transform(df)

         data = data.withColumn('label',col(labelCol))

         return data.select('features','label')
```

Our continuous, target and categorical features are passed as parameters to the function. We have to transform the data in the below order:

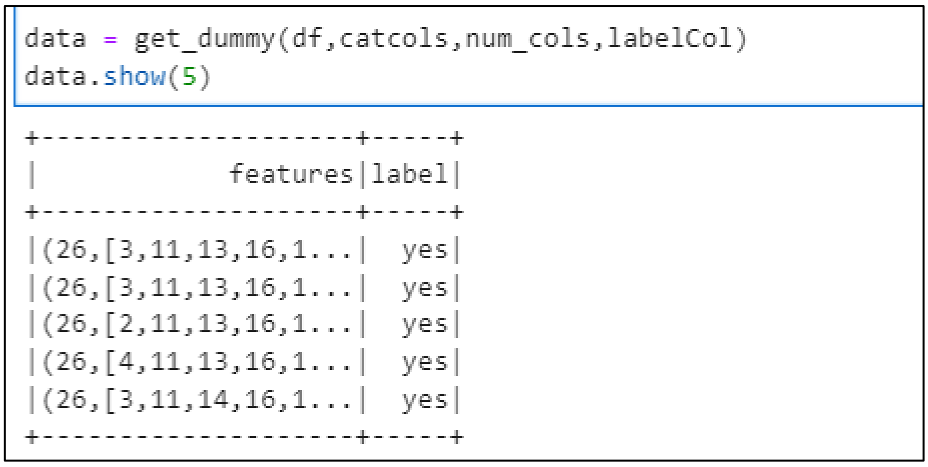
stage_1: String Index our categorical columns

stage_2: One-Hot encode the indexed column

stage_3: Assemble the encoded column

Spark actually works to predict with a column with all the features smashed together into a list-like structure. To do that in Spark we use the VectorAssembler.

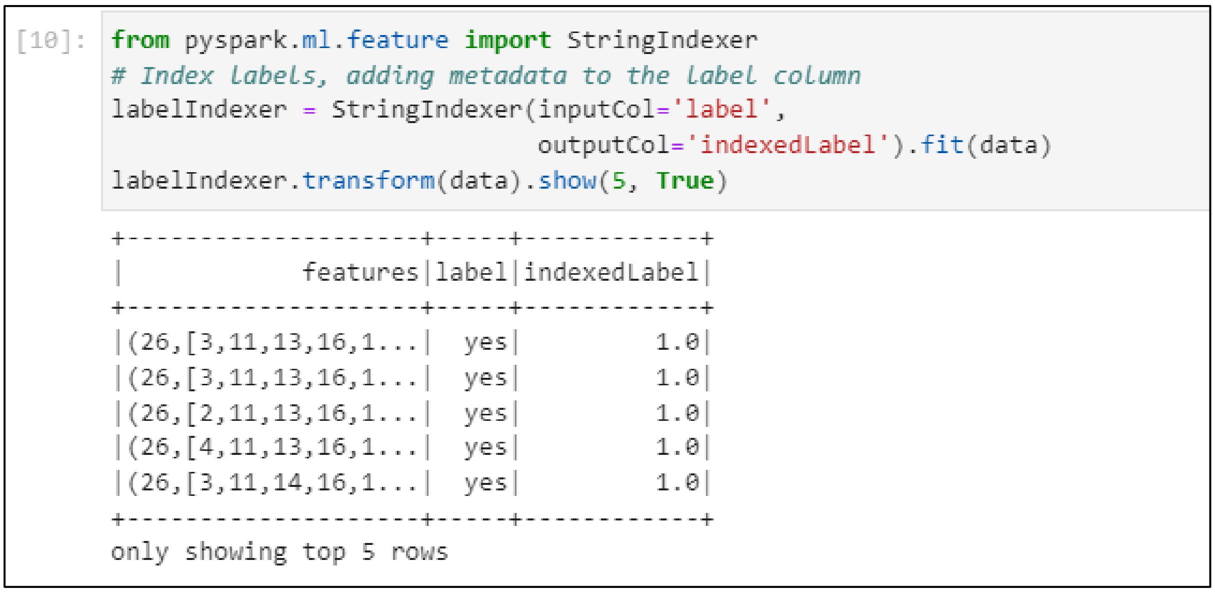
At each stage, we will pass the input and output column name and setup the pipeline by passing the defined stages in the list of the Pipeline object. The pipeline model then performs certain steps one by one in a sequence and gives us the end result.



```
data = get_dummy(df,catcols,num_cols,labelCol)
data.show(5)
```

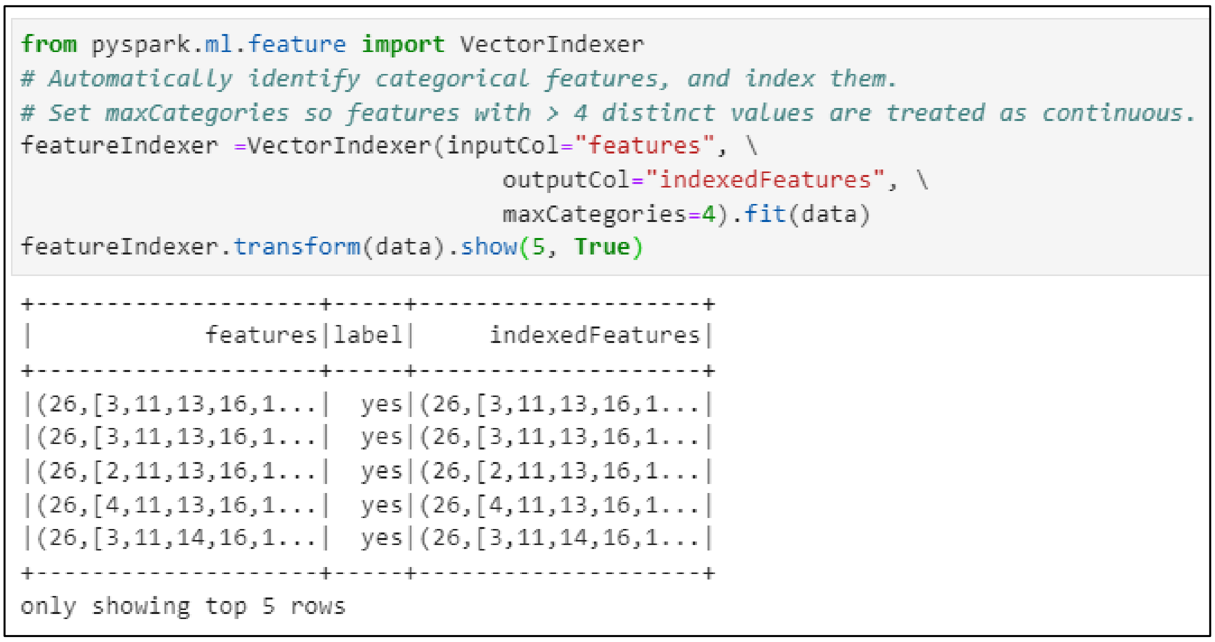
```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|(26,[3,11,13,16,1...|  yes|
|(26,[3,11,13,16,1...|  yes|
|(26,[2,11,13,16,1...|  yes|
|(26,[4,11,13,16,1...|  yes|
|(26,[3,11,14,16,1...|  yes|
+--------------------+-----+
```

Up till now we have only handled our categorical features but not our target variable (which is also categorical). We will now do the following:



```
[10]: from pyspark.ml.feature import StringIndexer
      # Index labels, adding metadata to the label column
      labelIndexer = StringIndexer(inputCol='label',
                                   outputCol='indexedLabel').fit(data)
      labelIndexer.transform(data).show(5, True)

      +--------------------+-----+------------+
      |            features|label|indexedLabel|
      +--------------------+-----+------------+
      |(26,[3,11,13,16,1...|  yes|         1.0|
      |(26,[3,11,13,16,1...|  yes|         1.0|
      |(26,[2,11,13,16,1...|  yes|         1.0|
      |(26,[4,11,13,16,1...|  yes|         1.0|
      |(26,[3,11,14,16,1...|  yes|         1.0|
      +--------------------+-----+------------+
      only showing top 5 rows
```

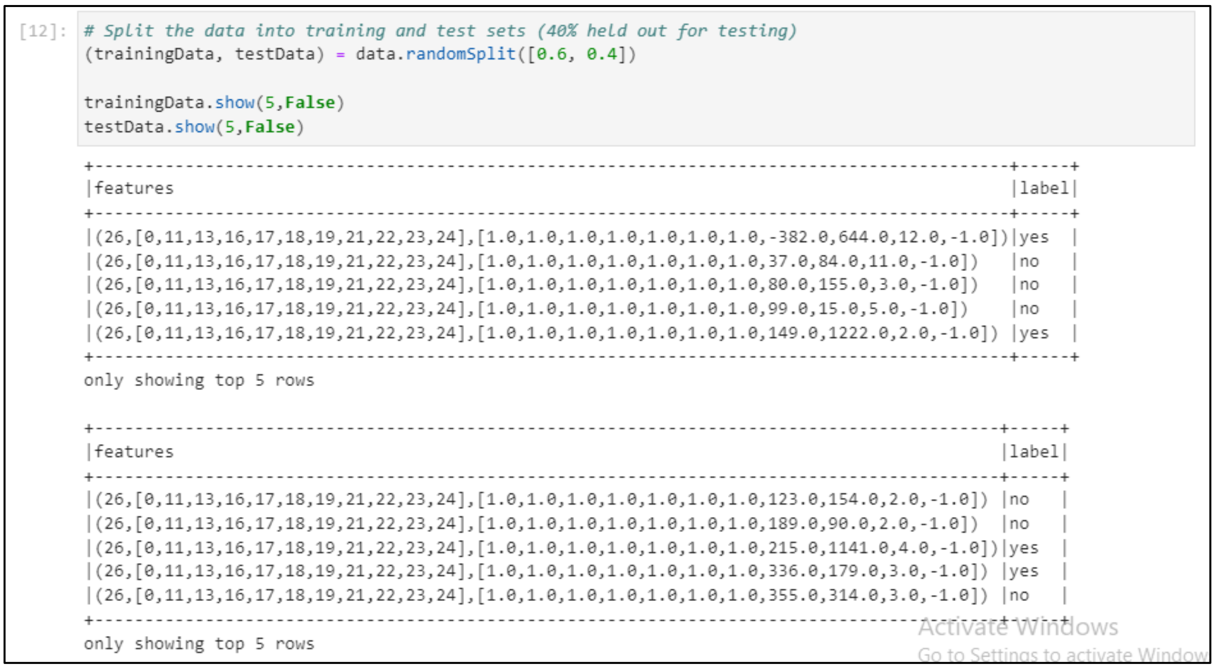
We also apply VectorIndexer to the features. This helps process a dataset of unknown vectors into a dataset with some continuous features and some categorical features. The choice between continuous and categorical is based upon a maxCategories parameter.



```
from pyspark.ml.feature import VectorIndexer
# Automatically identify categorical features, and index them.
# Set maxCategories so features with > 4 distinct values are treated as continuous.
featureIndexer =VectorIndexer(inputCol="features", \
                              outputCol="indexedFeatures", \
                              maxCategories=4).fit(data)
featureIndexer.transform(data).show(5, True)

+--------------------+-----+--------------------+
|            features|label|     indexedFeatures|
+--------------------+-----+--------------------+
|(26,[3,11,13,16,1...|  yes|(26,[3,11,13,16,1...|
|(26,[3,11,13,16,1...|  yes|(26,[3,11,13,16,1...|
|(26,[2,11,13,16,1...|  yes|(26,[2,11,13,16,1...|
|(26,[4,11,13,16,1...|  yes|(26,[4,11,13,16,1...|
|(26,[3,11,14,16,1...|  yes|(26,[3,11,14,16,1...|
+--------------------+-----+--------------------+
only showing top 5 rows
```

Train-Test Split

Now that all the preprocessing is done we can implement train and test split on the dataset.



```
[12]: # Split the data into training and test sets (40% held out for testing)
      (trainingData, testData) = data.randomSplit([0.6, 0.4])

      trainingData.show(5,False)
      testData.show(5,False)
```

```
+-------------------------------------------------------------------------------------------+-----+
|features                                                                                   |label|
+-------------------------------------------------------------------------------------------+-----+
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,-382.0,644.0,12.0,-1.0])|yes  |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,37.0,84.0,11.0,-1.0])   |no   |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,80.0,155.0,3.0,-1.0])   |no   |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,99.0,15.0,5.0,-1.0])    |no   |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,149.0,1222.0,2.0,-1.0]) |yes  |
+-------------------------------------------------------------------------------------------+-----+
only showing top 5 rows

+------------------------------------------------------------------------------------------+-----+
|features                                                                                  |label|
+------------------------------------------------------------------------------------------+-----+
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,123.0,154.0,2.0,-1.0]) |no   |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,189.0,90.0,2.0,-1.0])  |no   |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,215.0,1141.0,4.0,-1.0])|yes  |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,336.0,179.0,3.0,-1.0]) |yes  |
|(26,[0,11,13,16,17,18,19,21,22,23,24],[1.0,1.0,1.0,1.0,1.0,1.0,1.0,355.0,314.0,3.0,-1.0]) |no   |
+------------------------------------------------------------------------------------------+-----+
only showing top 5 rows
```

Classification Models

Logistic Regression

Now we can build and fit an ML model to our dataset to predict the target columns with all the other ones. First we will be using Logistic Regression. This is actually an estimator that we have to fit.

Estimator

An Estimator implements the *fit()* method on a dataframe and produces a model. For example, *LogisticRegression* is an Estimator that trains a classification model when we call the *fit()* method.

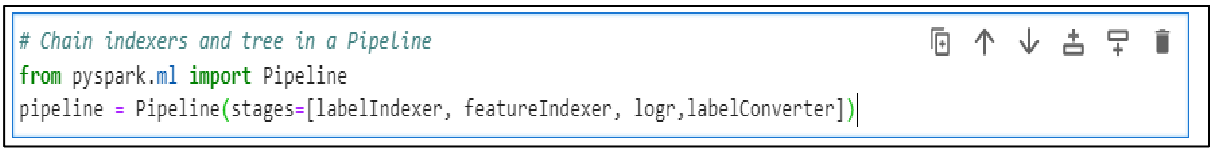
First we import the logistic regression method:

```
from pyspark.ml.classification import LogisticRegression
logr = LogisticRegression(featuresCol='indexedFeatures', labelCol='indexedLabel')
```

Now we convert our labels from index to string to better understand the predictions:

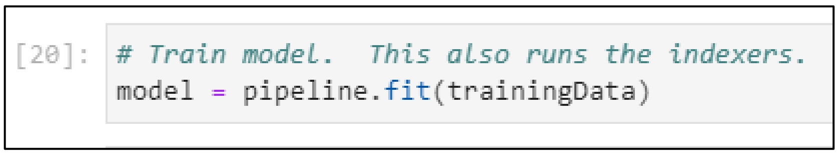
```
# Convert indexed labels back to original labels.
from pyspark.ml.feature import IndexToString
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)
```

Now we build a pipeline so we can fit it to our model. All the stages in the pipeline have already been discussed above.



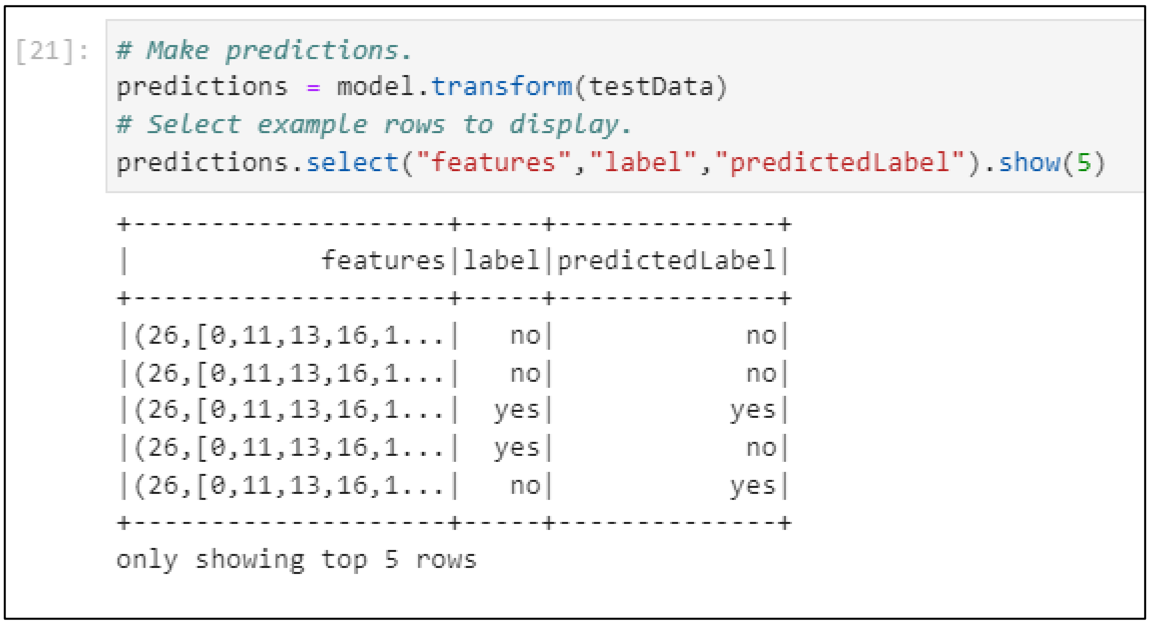
```python
# Chain indexers and tree in a Pipeline
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, logr,labelConverter])
```

Now that our pipeline is ready we can train our model:



```python
[20]: # Train model.  This also runs the indexers.
      model = pipeline.fit(trainingData)
```

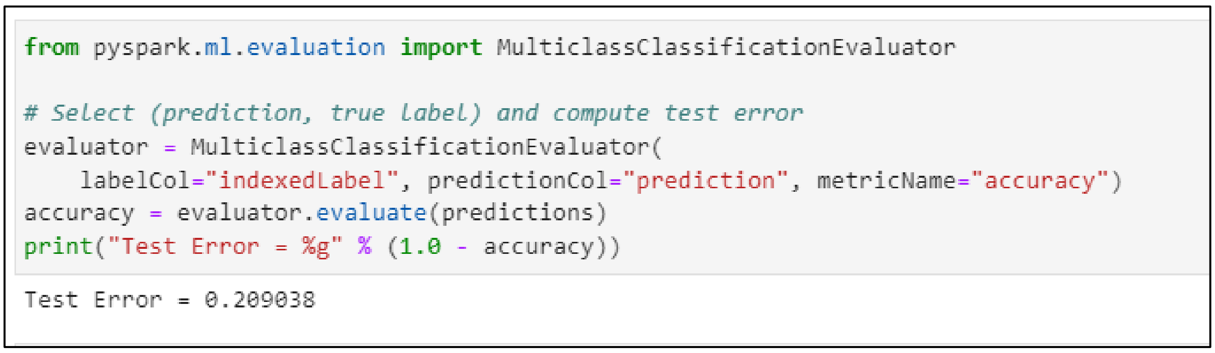
This will give us a transformer. And finally, we predict using the test dataset:



```python
[21]: # Make predictions.
      predictions = model.transform(testData)
      # Select example rows to display.
      predictions.select("features","label","predictedLabel").show(5)
```

```
+--------------------+-----+--------------+
|            features|label|predictedLabel|
+--------------------+-----+--------------+
|(26,[0,11,13,16,1...|   no|            no|
|(26,[0,11,13,16,1...|   no|            no|
|(26,[0,11,13,16,1...|  yes|           yes|
|(26,[0,11,13,16,1...|  yes|            no|
|(26,[0,11,13,16,1...|   no|           yes|
+--------------------+-----+--------------+
only showing top 5 rows
```

In order to see how well our model performed we can evaluate our model. For that, we will calculate the test error:



```python
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

```
Test Error = 0.209038
```

We can also observe roc, FPR and TPR:

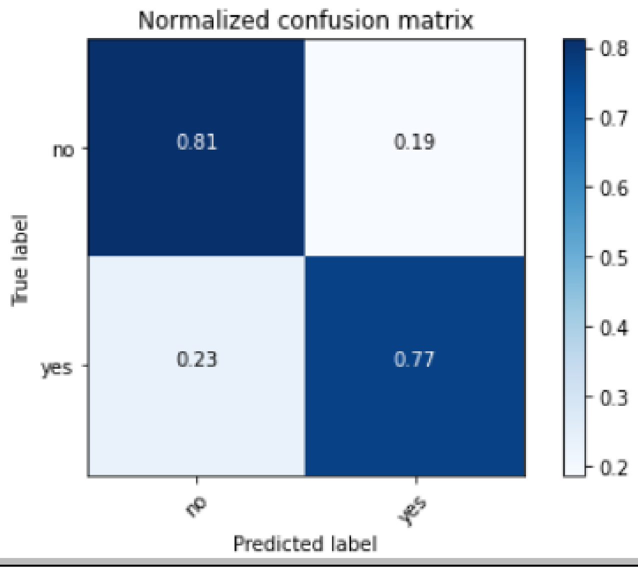
```
# Obtain the receiver-operating characteristic as a dataframe and areaUnderROC.
trainingSummary.roc.show(5)
print("areaUnderROC: " + str(trainingSummary.areaUnderROC))

# Set the model threshold to maximize F-Measure
fMeasure = trainingSummary.fMeasureByThreshold
maxFMeasure = fMeasure.groupBy().max('F-Measure').select('max(F-Measure)').head(5)
```

```
/usr/local/spark/python/pyspark/sql/context.py:125: FutureWarning: Deprecated in 3.
er.getOrCreate() instead.
  warnings.warn(
+--------------------+--------------------+
|                 FPR|                 TPR|
+--------------------+--------------------+
|                 0.0|                 0.0|
|2.838489923360772E-4|0.001588814744200...|
|2.838489923360772E-4|0.003495392437241...|
|8.515469770082316E-4|0.004766444232602479|
|8.515469770082316E-4| 0.00667302192564347|
+--------------------+--------------------+
only showing top 5 rows

areaUnderROC: 0.8769102419336883
```

In order to visualize the results we can plot the confusion matrix:

```
[36]: # Plot normalized confusion matrix
      plt.figure()
      plot_confusion_matrix(cnf_matrix, classes=class_temp, normalize=True,
                            title='Normalized confusion matrix')

      plt.show()
```

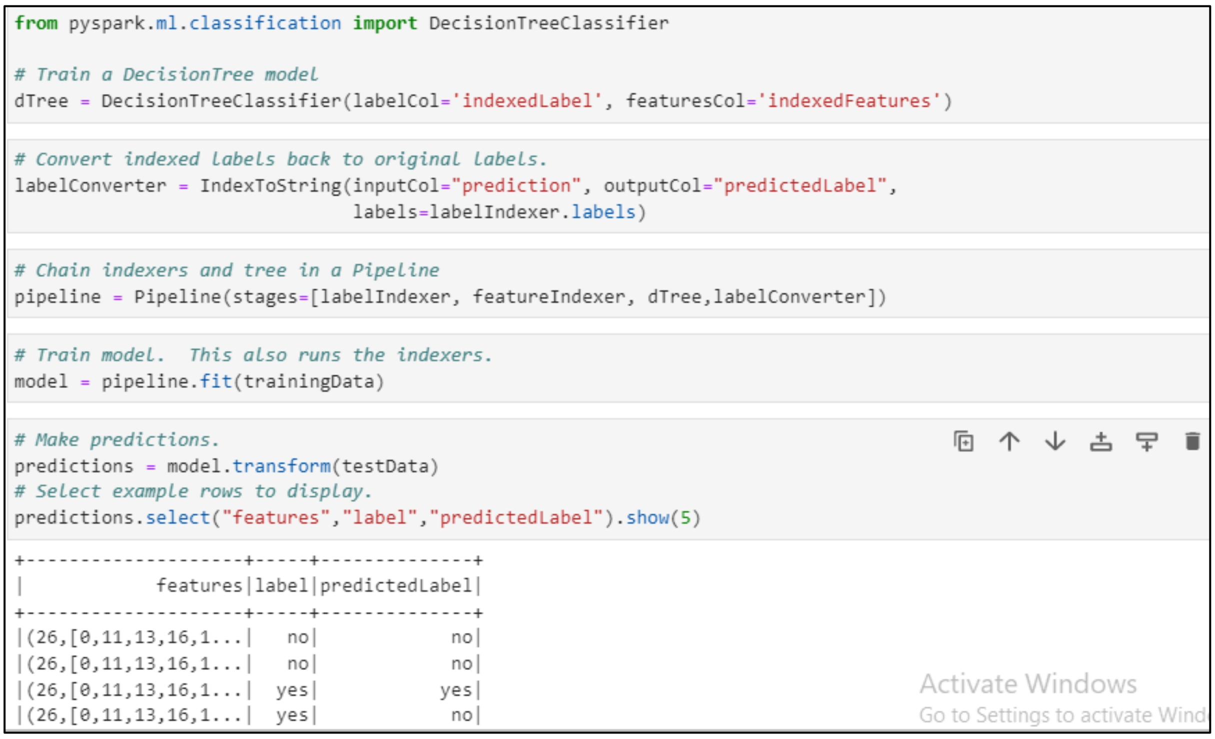
```
Normalized confusion matrix
[[0.81148936 0.18851064]
 [0.23155929 0.76844071]]
```



Now that we have covered all the basics we can fit other classification models to our data and most of the process will be the same.

Decision Tree

Now we will try decision tree for our data. The process will be the same as mentioned above we will just change the model which we are fitting.



```python
from pyspark.ml.classification import DecisionTreeClassifier

# Train a DecisionTree model
dTree = DecisionTreeClassifier(labelCol='indexedLabel', featuresCol='indexedFeatures')
```

```python
# Convert indexed labels back to original labels.
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)
```

```python
# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dTree,labelConverter])
```

```python
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

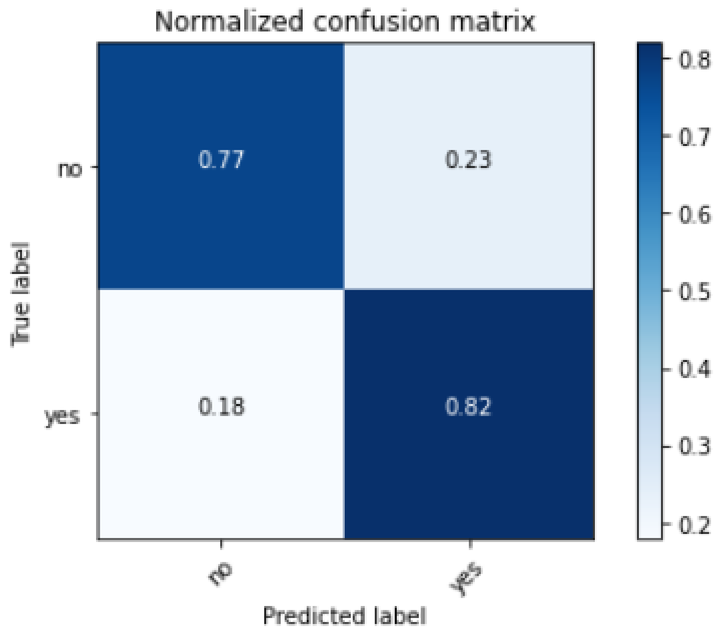
```python
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("features","label","predictedLabel").show(5)
```

```
+--------------------+-----+--------------+
|            features|label|predictedLabel|
+--------------------+-----+--------------+
|(26,[0,11,13,16,1...|   no|            no|
|(26,[0,11,13,16,1...|   no|            no|
|(26,[0,11,13,16,1...|  yes|           yes|
|(26,[0,11,13,16,1...|  yes|            no|
```

The confusion matrix:

```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_temp, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

```
Normalized confusion matrix
[[0.76595745 0.23404255]
 [0.18067227 0.81932773]]
```



Random Forest:

Importing the model:

```
from pyspark.ml.classification import RandomForestClassifier

# Train a RandomForest model.
rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numTrees=10)
```

Creating pipeline:

```
[50]: # Chain indexers and tree in a Pipeline
      pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf,labelConverter])
```

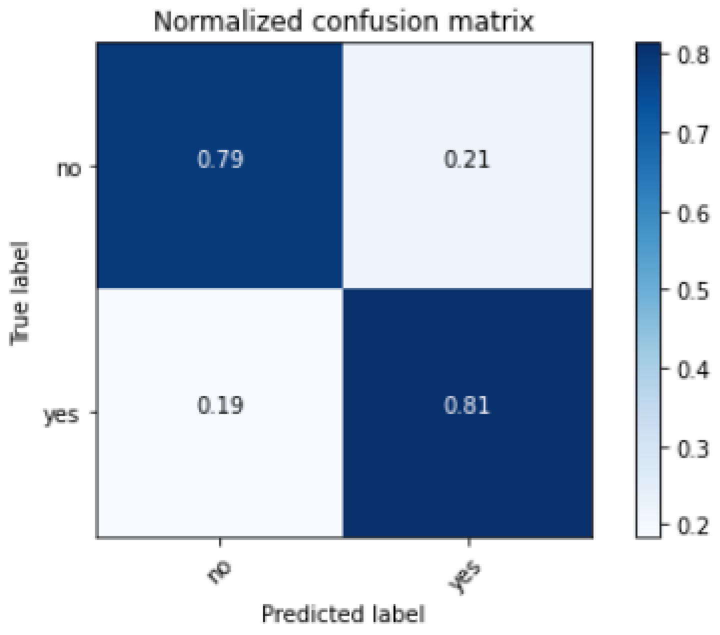
Fitting the model:

```
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

Confusion matrix:

```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_temp, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

```
Normalized confusion matrix
[[0.78808511 0.21191489]
 [0.18580766 0.81419234]]
```



GradientBoosting:

Importing the model:

```
from pyspark.ml.classification import GBTClassifier

gbt = GBTClassifier(labelCol='indexedLabel', featuresCol='indexedFeatures')
```

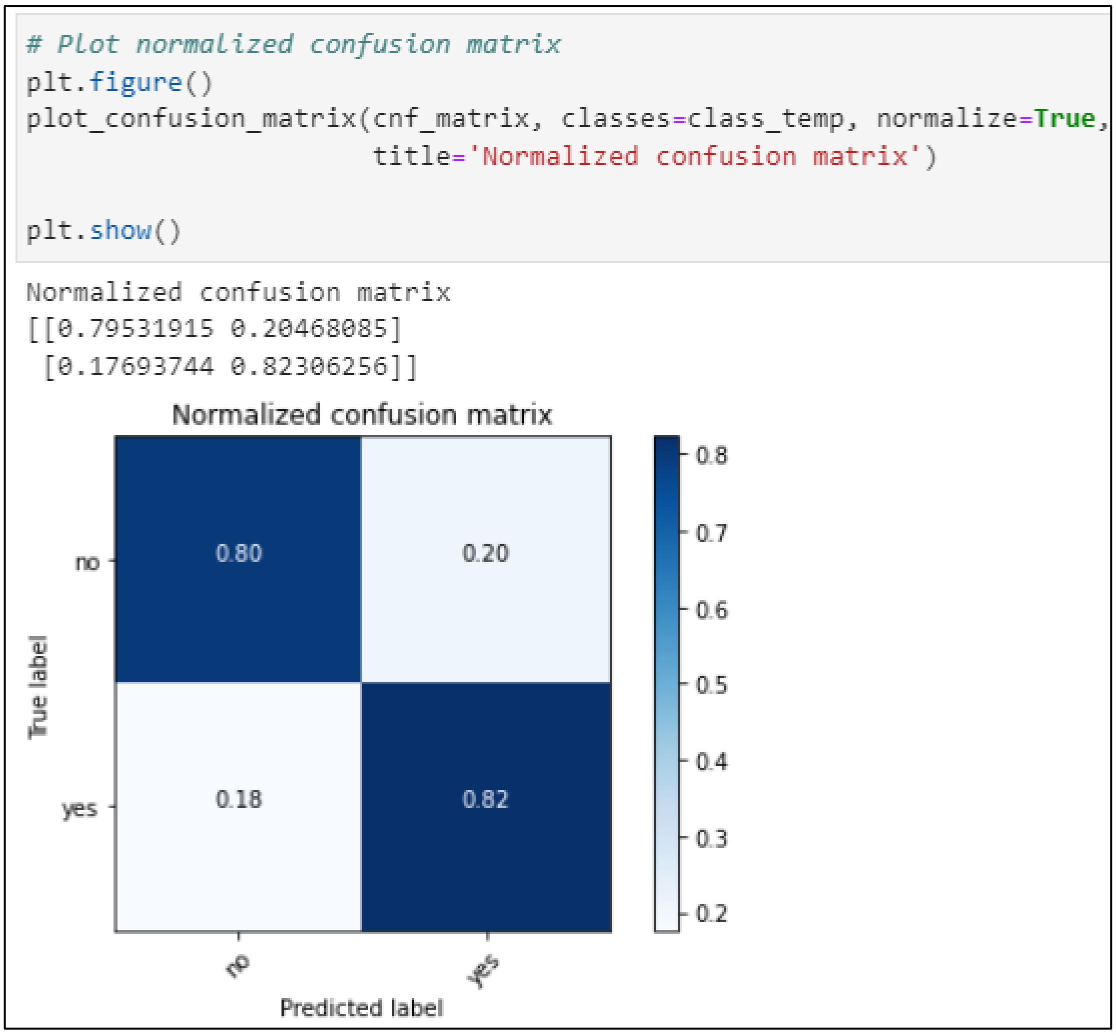
Creating pipeline:

```
# Chain indexers and tree in a Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, gbt,labelConverter])
```

Fitting the model:

```
# Train model.  This also runs the indexers.
model = pipeline.fit(trainingData)
```

Confusion matrix:



```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_temp, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

```
Normalized confusion matrix
[[0.79531915 0.20468085]
 [0.17693744 0.82306256]]
```

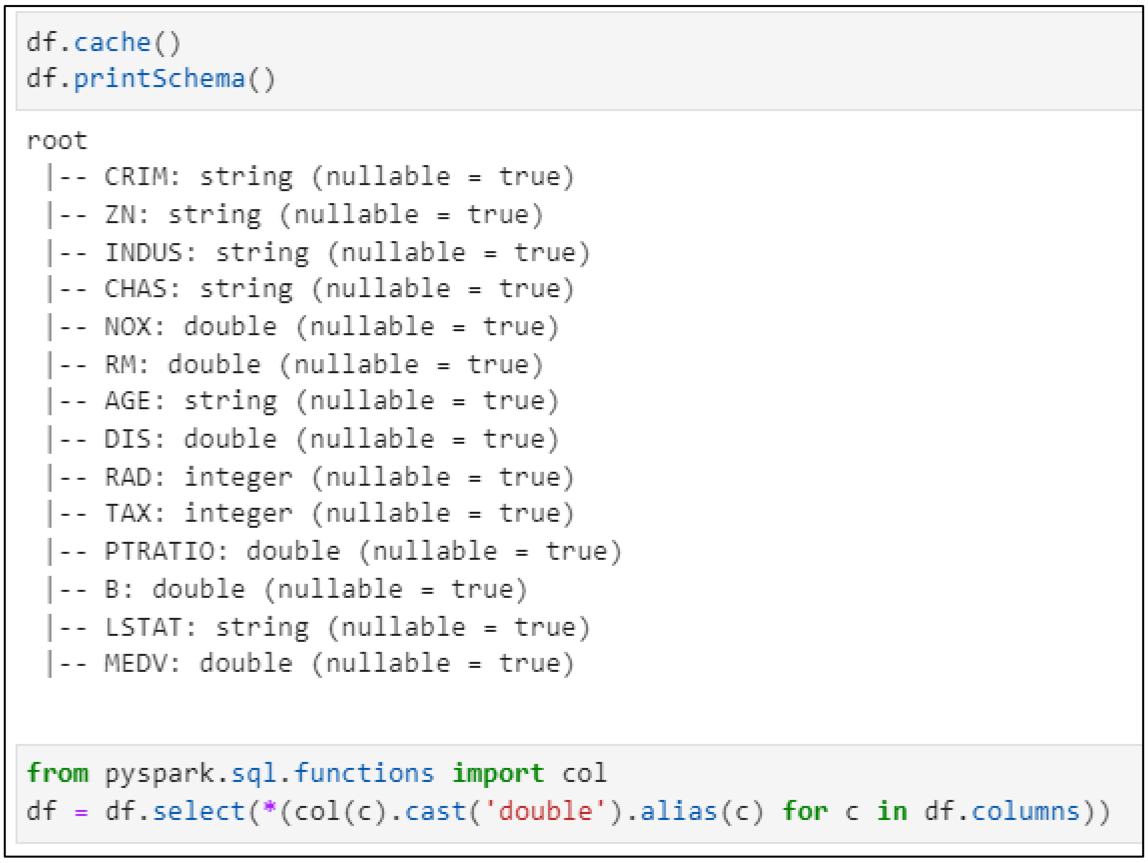
Regression Models

The data processing performed above will be followed for regression as well the difference will be that we are now using regressors instead of classifiers. Hence we will not apply string indexer on the target variable. Repeating the whole process (discussed above) in the report would be redundant so we will only be sharing the models and their results. Both notebooks for regression and classification have been shared along with the project.

Some of the few steps done differently are shared here:

Casting Variables

Some the columns were of String type even though they should be numeric. So we are going to cast them.



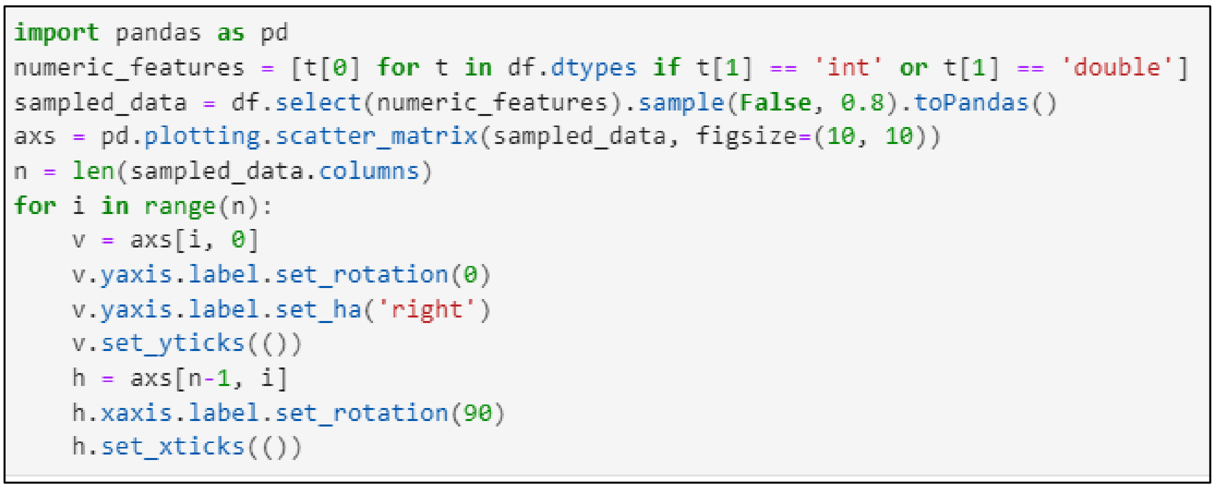
```
df.cache()
df.printSchema()
```

```
root
 |-- CRIM: string (nullable = true)
 |-- ZN: string (nullable = true)
 |-- INDUS: string (nullable = true)
 |-- CHAS: string (nullable = true)
 |-- NOX: double (nullable = true)
 |-- RM: double (nullable = true)
 |-- AGE: string (nullable = true)
 |-- DIS: double (nullable = true)
 |-- RAD: integer (nullable = true)
 |-- TAX: integer (nullable = true)
 |-- PTRATIO: double (nullable = true)
 |-- B: double (nullable = true)
 |-- LSTAT: string (nullable = true)
 |-- MEDV: double (nullable = true)
```

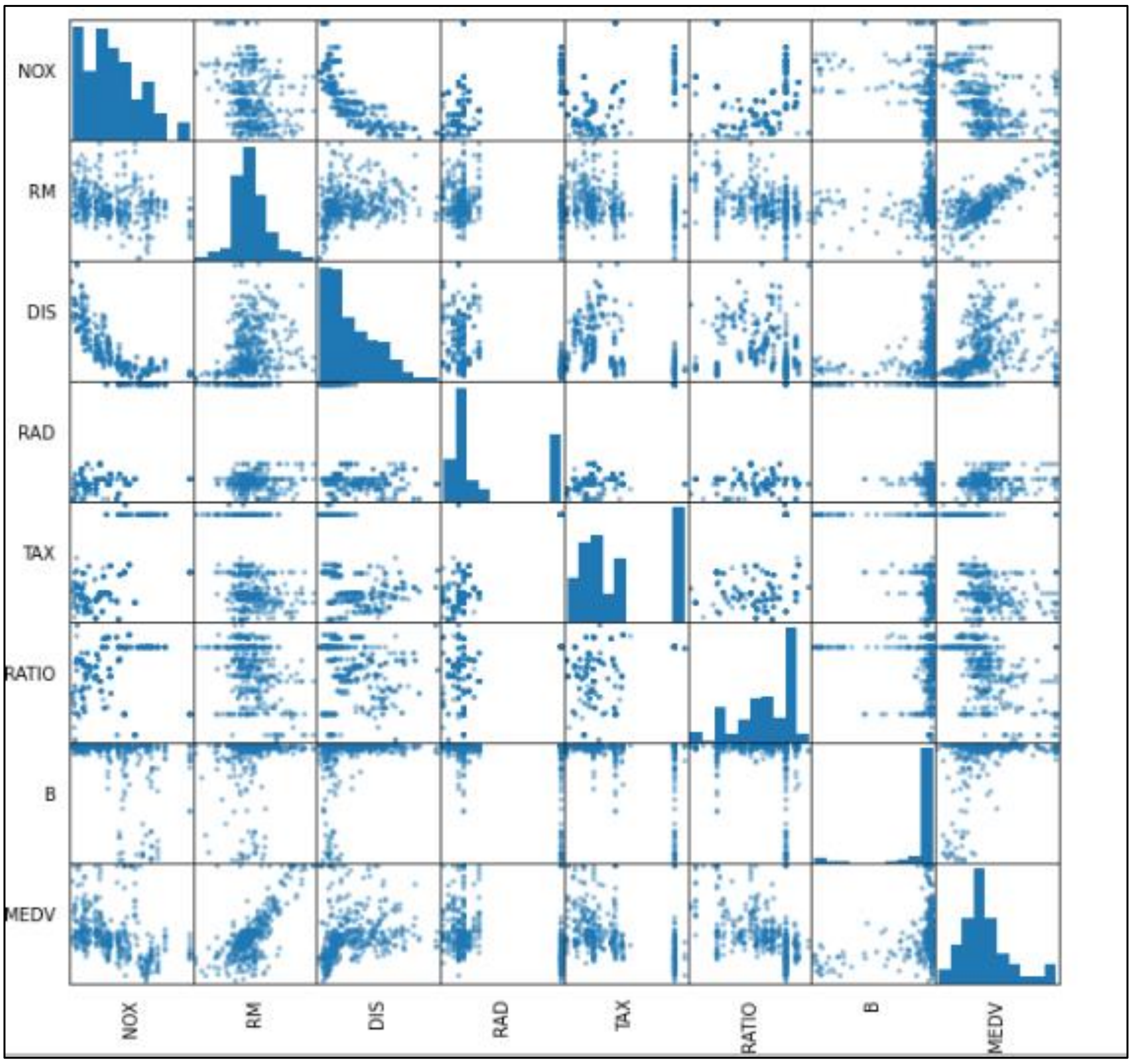
```
from pyspark.sql.functions import col
df = df.select(*(col(c).cast('double').alias(c) for c in df.columns))
```

Data Exploration:

Scatter matrix is a great way to roughly determine if we have a linear correlation between multiple independent variables.

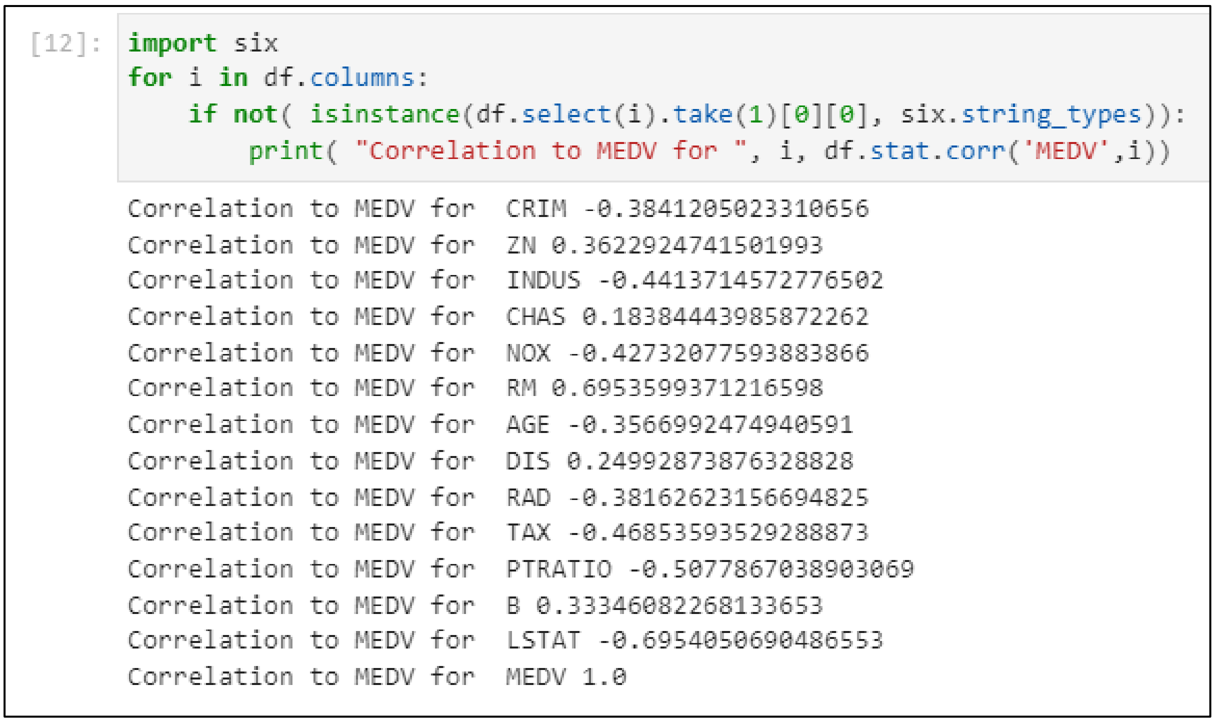


```
import pandas as pd
numeric_features = [t[0] for t in df.dtypes if t[1] == 'int' or t[1] == 'double']
sampled_data = df.select(numeric_features).sample(False, 0.8).toPandas()
axs = pd.plotting.scatter_matrix(sampled_data, figsize=(10, 10))
n = len(sampled_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```



Correlation

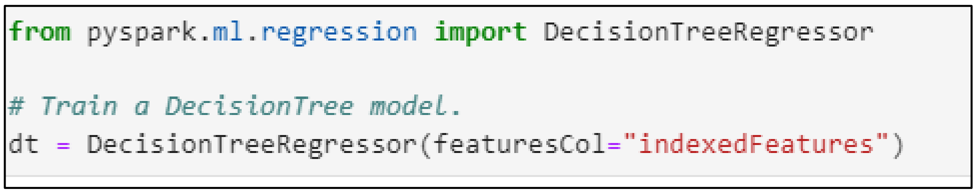
We can also find correlation between independent variables and target variable.



```
[12]: import six
      for i in df.columns:
          if not( isinstance(df.select(i).take(1)[0][0], six.string_types)):
              print( "Correlation to MEDV for ", i, df.stat.corr('MEDV',i))

      Correlation to MEDV for  CRIM -0.3841205023310656
      Correlation to MEDV for  ZN 0.3622924741501993
      Correlation to MEDV for  INDUS -0.4413714572776502
      Correlation to MEDV for  CHAS 0.18384443985872262
      Correlation to MEDV for  NOX -0.42732077593883866
      Correlation to MEDV for  RM 0.6953599371216598
      Correlation to MEDV for  AGE -0.3566992474940591
      Correlation to MEDV for  DIS 0.24992873876328828
      Correlation to MEDV for  RAD -0.38162623156694825
      Correlation to MEDV for  TAX -0.46853593529288873
      Correlation to MEDV for  PTRATIO -0.5077867038903069
      Correlation to MEDV for  B 0.33346082268133653
      Correlation to MEDV for  LSTAT -0.6954050690486553
      Correlation to MEDV for  MEDV 1.0
```

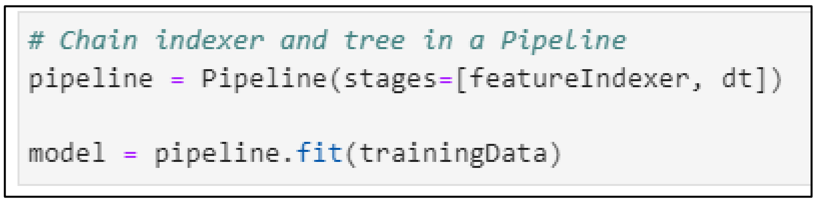
Decision Tree Regressor

Importing Model:



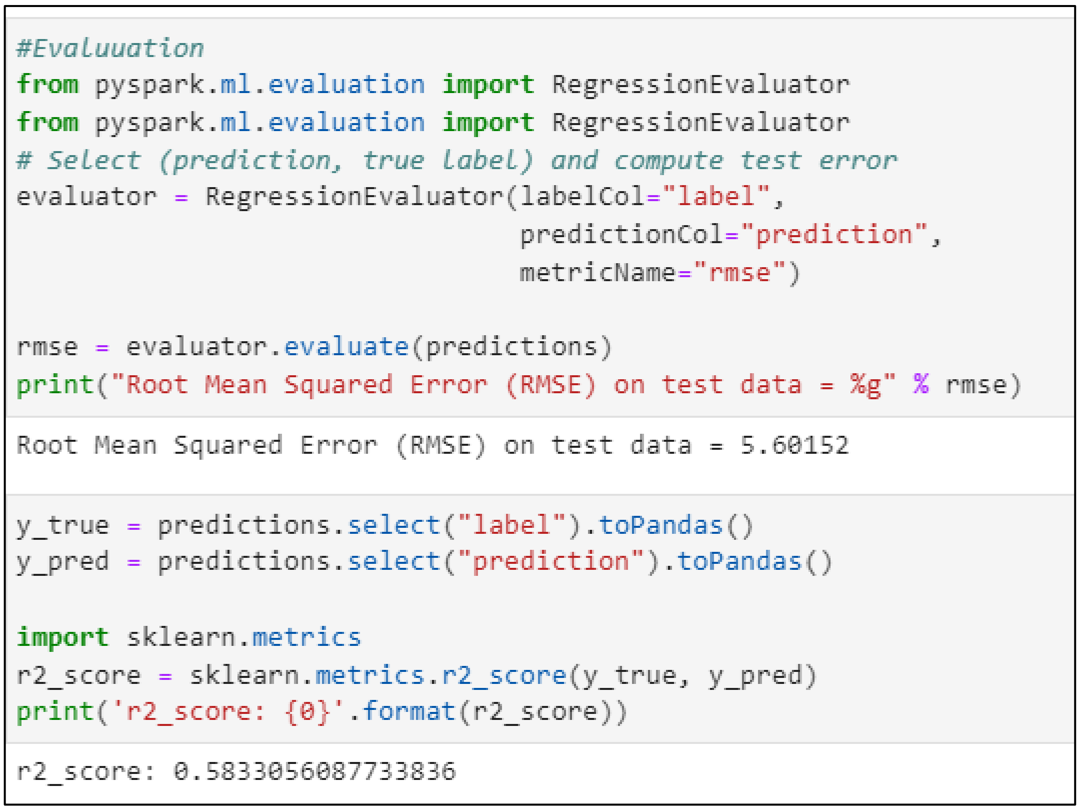
```
from pyspark.ml.regression import DecisionTreeRegressor

# Train a DecisionTree model.
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")
```

Building Pipeline and Fitting model:



```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, dt])

model = pipeline.fit(trainingData)
```

Scores:



```
#Evaluuation
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.evaluation import RegressionEvaluator
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="label",
                                predictionCol="prediction",
                                metricName="rmse")

rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

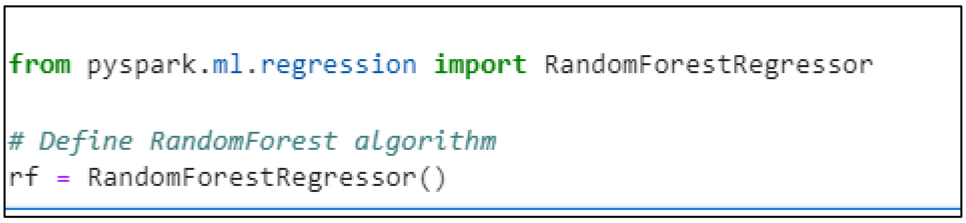
```
Root Mean Squared Error (RMSE) on test data = 5.60152
```

```
y_true = predictions.select("label").toPandas()
y_pred = predictions.select("prediction").toPandas()

import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {0}'.format(r2_score))
```

```
r2_score: 0.5833056087733836
```

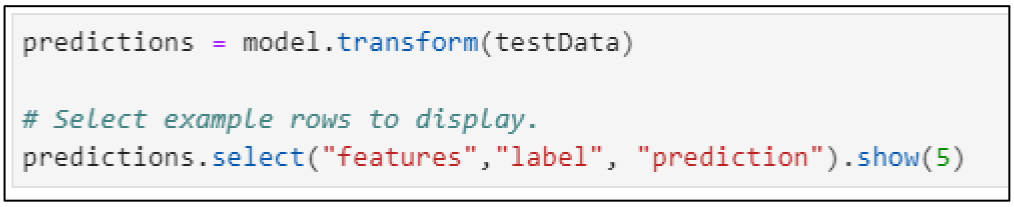
Decision Tree Regressor

Importing Model:



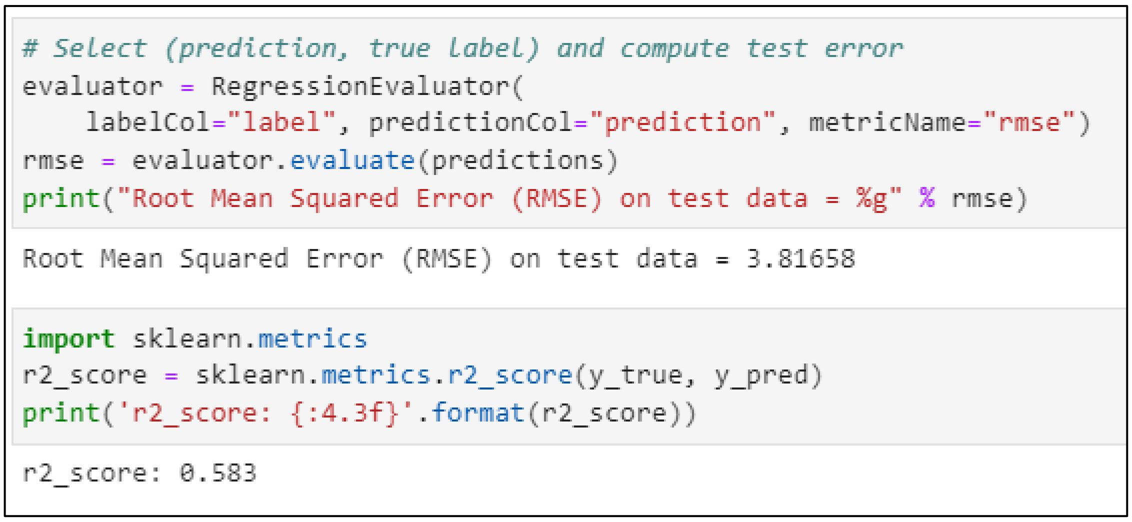
```
from pyspark.ml.regression import RandomForestRegressor

# Define RandomForest algorithm
rf = RandomForestRegressor()
```

Building Pipeline and Fitting mode:



```
predictions = model.transform(testData)

# Select example rows to display.
predictions.select("features","label", "prediction").show(5)
```

Scores:



```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

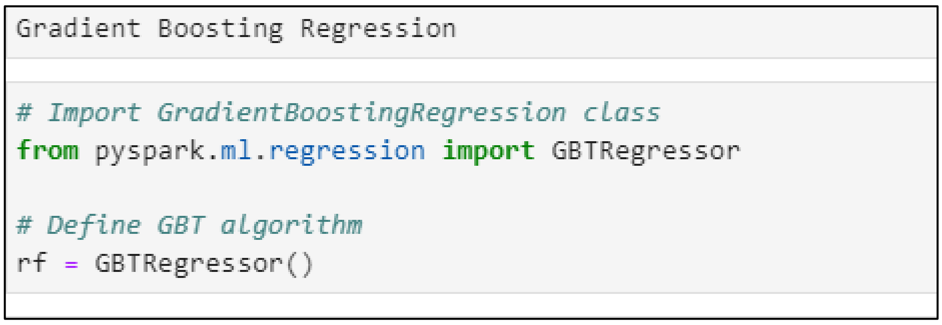
```
Root Mean Squared Error (RMSE) on test data = 3.81658
```

```
import sklearn.metrics
r2_score = sklearn.metrics.r2_score(y_true, y_pred)
print('r2_score: {:4.3f}'.format(r2_score))
```

```
r2_score: 0.583
```

Decision Tree Regressor

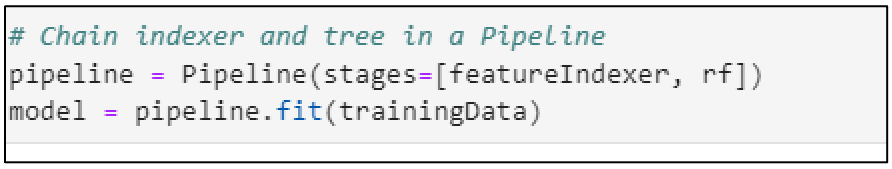
Importing Model:



```
Gradient Boosting Regression
```

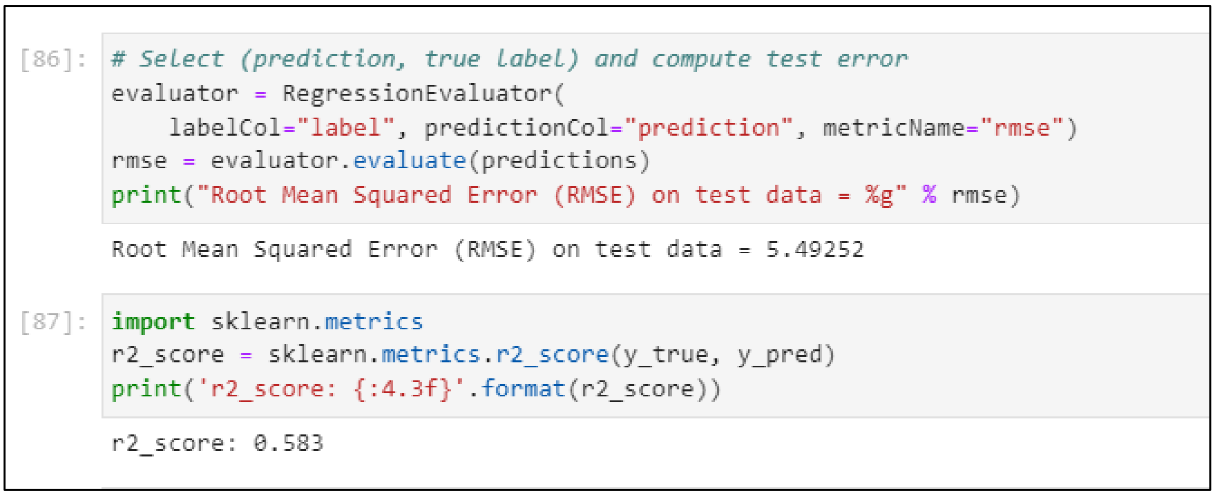
```
# Import GradientBoostingRegression class
from pyspark.ml.regression import GBTRegressor

# Define GBT algorithm
rf = GBTRegressor()
```

Building Pipeline and Fitting mode:



```
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, rf])
model = pipeline.fit(trainingData)
```

Scores:


```
[86]: # Select (prediction, true label) and compute test error
      evaluator = RegressionEvaluator(
          labelCol="label", predictionCol="prediction", metricName="rmse")
      rmse = evaluator.evaluate(predictions)
      print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)

      Root Mean Squared Error (RMSE) on test data = 5.49252

[87]: import sklearn.metrics
      r2_score = sklearn.metrics.r2_score(y_true, y_pred)
      print('r2_score: {:4.3f}'.format(r2_score))

      r2_score: 0.583
```

Apache Mahout

Mahout is an open source machine learning library from Apache. Mahout primarily implements clustering, recommender engines (collaborative filtering), classification, and dimensionality reduction algorithms but is not limited to these.

Difficulties in the project

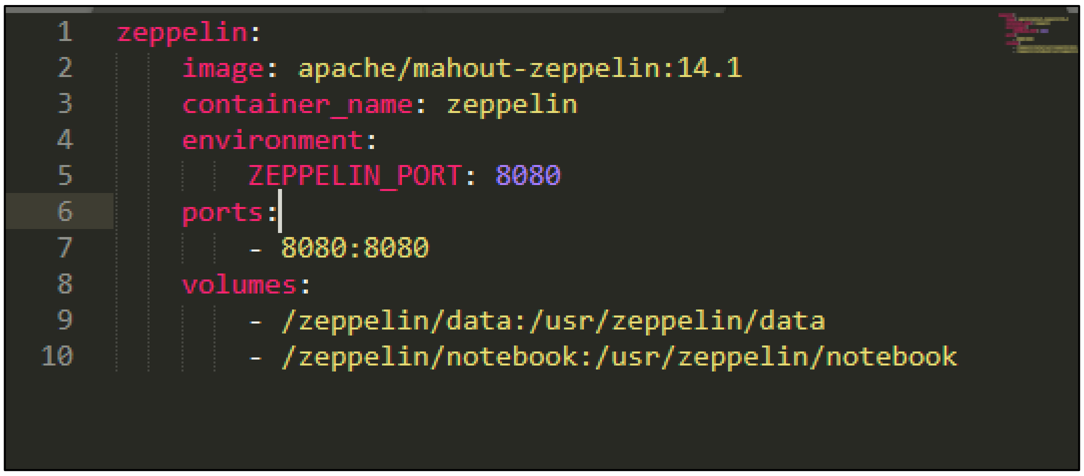
While researching on mahout there were not many useful resources found which could help in completing the project. A lot of the technologies mentioned on the online resources were either outdated or the methods mentioned were depreciated. While online resources proved to be of no help, I turned towards books. There were five books that I skimmed through in order to find the solution. The name of the boos are as follows:

- Mahout in Action
- Mahout Cookbook
- Apache Mahout Essentials
- Learning Apache Mahout Classification
- Learning Apache mahout

In some books there were codes given which I implemented on Zeppelin. However most of the classes were depreciated. Which in turn led me to this page which shows that a lot of classes in mahout are depreciated.

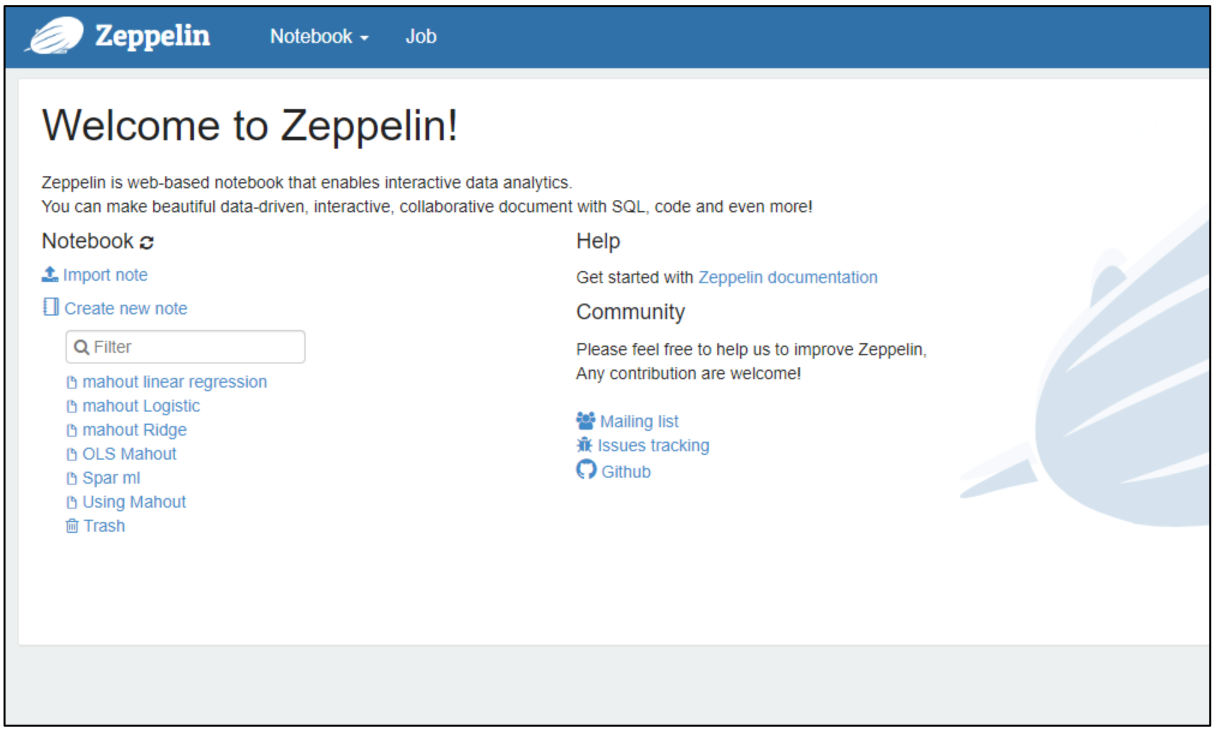
Setup

In this project I have tried to implement mahout algorithms using the Zeppelin notebook. We used the following docker-compose file.



```yaml
zeppelin:
    image: apache/mahout-zeppelin:14.1
    container_name: zeppelin
    environment:
        ZEPPELIN_PORT: 8080
    ports:
        - 8080:8080
    volumes:
        - /zeppelin/data:/usr/zeppelin/data
        - /zeppelin/notebook:/usr/zeppelin/notebook
```

The file uses the apache/mahout-zeppelin:14.1 image which has a a preconfigured Mahout+Spark interpreter. It defines two docker volumes which keeps the datasets and notebooks. I can start Zeppelin container via docker-compose up -d zeppelin command. Then I can browse the Zeppelin web in http:localhost:8080 url. Zeppelin web provides interface to create notebooks, import notebooks, data visualizations.



Welcome to Zeppelin!

Zeppelin is web-based notebook that enables interactive data analytics.
You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!

Notebook

- Import note
- Create new note
- Filter
- mahout linear regression
- mahout Logistic
- mahout Ridge
- OLS Mahout
- Spar ml
- Using Mahout
- Trash

Help

Get started with Zeppelin documentation

Community

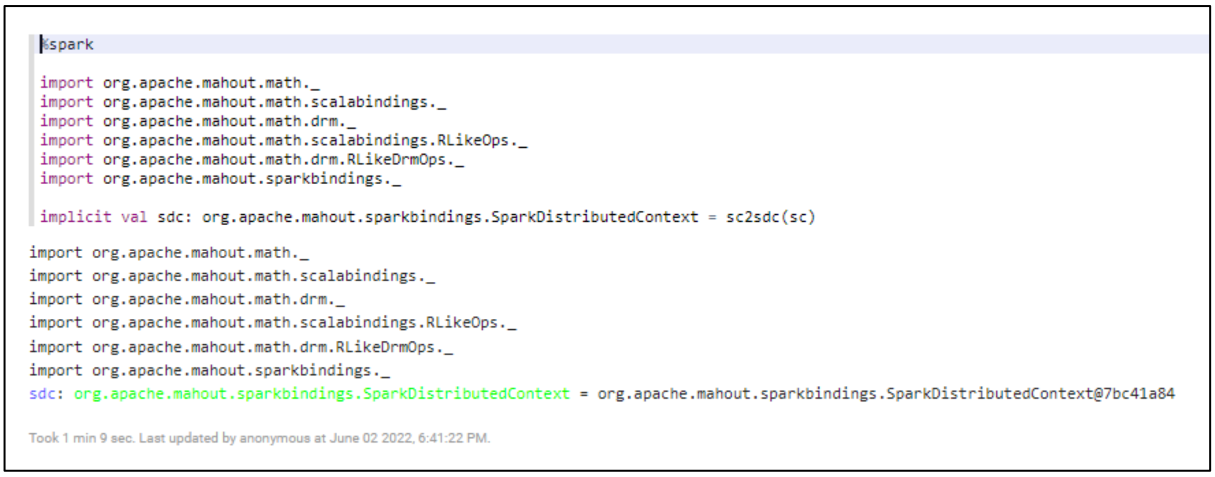
Please feel free to help us to improve Zeppelin,
Any contribution are welcome!

- Mailing list
- Issues tracking
- Github

How To Use

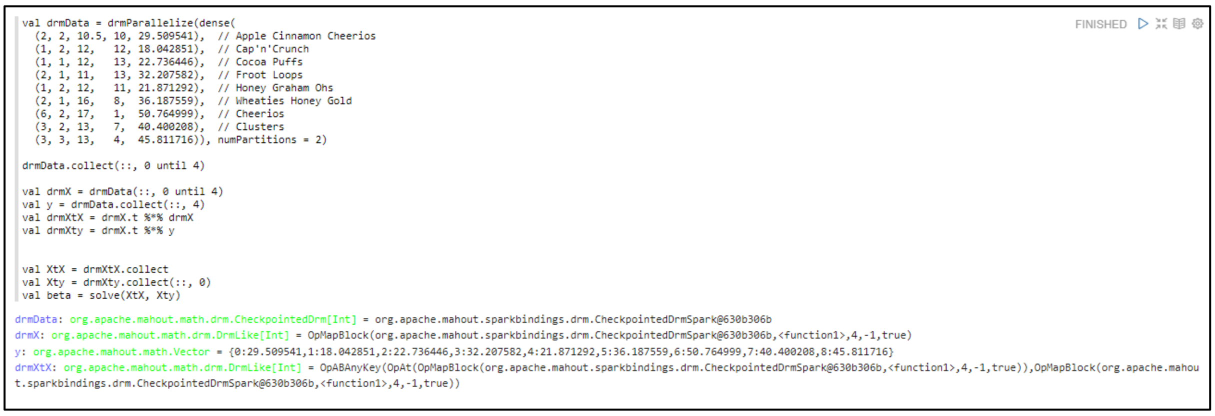
I followed this tutorial to use the mahout interpreter on zeppelin.

When starting a session with Apache Mahout, depending on which engine you are using (Spark or Flink), a few imports must be made and a Distributed Context must be declared. Copy and paste the following code and run once to get started.



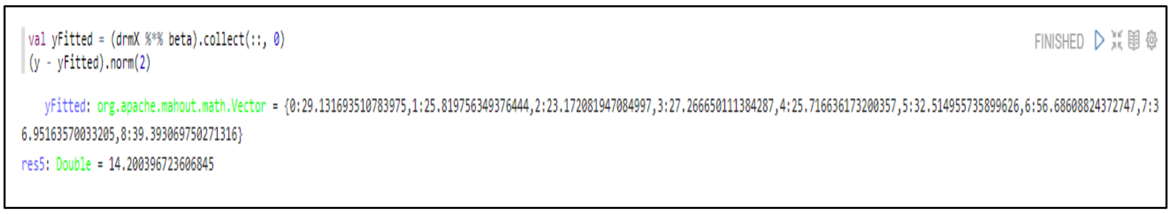
```
%spark

import org.apache.mahout.math._
import org.apache.mahout.math.scalabindings._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.scalabindings.RLikeOps._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.sparkbindings._

implicit val sdc: org.apache.mahout.sparkbindings.SparkDistributedContext = sc2sdc(sc)

import org.apache.mahout.math._
import org.apache.mahout.math.scalabindings._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.scalabindings.RLikeOps._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.sparkbindings._
sdc: org.apache.mahout.sparkbindings.SparkDistributedContext = org.apache.mahout.sparkbindings.SparkDistributedContext@7bc41a84

Took 1 min 9 sec. Last updated by anonymous at June 02 2022, 6:41:22 PM.
```

After importing and setting up the distributed context, the Mahout R-Like DSL is consistent across engines. The following code will run in both %flinkMahout and %sparkMahout.



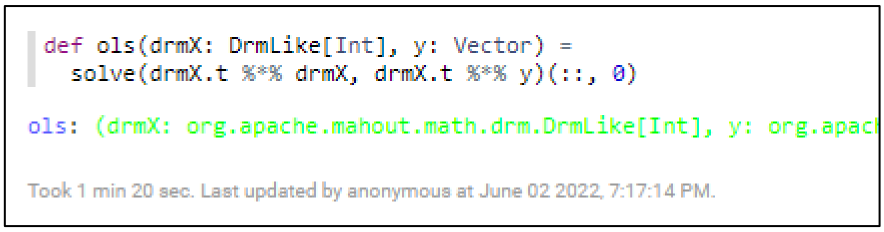
```
val drmData = drmParallelize(dense(
  (2, 2, 10.5, 10, 29.509541),  // Apple Cinnamon Cheerios
  (1, 2, 12,   12, 18.042851),  // Cap'n'Crunch
  (1, 1, 12,   13, 22.736446),  // Cocoa Puffs
  (2, 1, 11,   13, 32.207582),  // Froot Loops
  (1, 2, 12,   11, 21.871292),  // Honey Graham Ohs
  (2, 1, 16,   8,  36.187559),  // Wheaties Honey Gold
  (6, 2, 17,   1,  50.764999),  // Cheerios
  (3, 2, 13,   7,  40.400208),  // Clusters
  (3, 3, 13,   4,  45.811716)), numPartitions = 2)

drmData.collect(::, 0 until 4)

val drmX = drmData(::, 0 until 4)
val y = drmData.collect(::, 4)
val drmXtX = drmX.t %*% drmX
val drmXty = drmX.t %*% y


val XtX = drmXtX.collect
val Xty = drmXty.collect(::, 0)
val beta = solve(XtX, Xty)
drmData: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@630b306b
drmX: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@630b306b,<function1>,4,-1,true)
y: org.apache.mahout.math.Vector = {0:29.509541,1:18.042851,2:22.736446,3:32.207582,4:21.871292,5:36.187559,6:50.764999,7:40.400208,8:45.811716}
drmXtX: org.apache.mahout.math.drm.DrmLike[Int] = OpABAnyKey(OpAt(OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@630b306b,<function1>,4,-1,true)),OpMapBlock(org.apache.mahou
t.sparkbindings.drm.CheckpointedDrmSpark@630b306b,<function1>,4,-1,true))
```

We can now check how well our model fits its training data. First, we multiply the feature matrix \(\mathbf{X}\) by our estimate of \(\boldsymbol{\beta}\). Then, we look at the difference (via L2-norm) of the target variable \(\mathbf{y}\) to the fitted target variable:



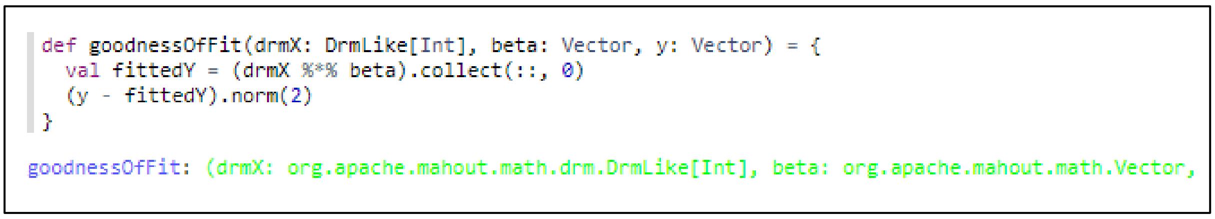
```
val yFitted = (drmX %*% beta).collect(::, 0)
(y - yFitted).norm(2)

   yFitted: org.apache.mahout.math.Vector = {0:29.131693510783975,1:25.819756349376444,2:23.172081947084997,3:27.266650111384287,4:25.716636173200357,5:32.514955735899626,6:56.68608824372747,7:3
6.95163570033205,8:39.393069750271316}
res5: Double = 14.200396723606845
```

We can now refactor a little by wrapping our statements into easy-to-use functions. The definition of functions follows standard scala syntax. We put all the commands for ordinary least squares into a function ols.



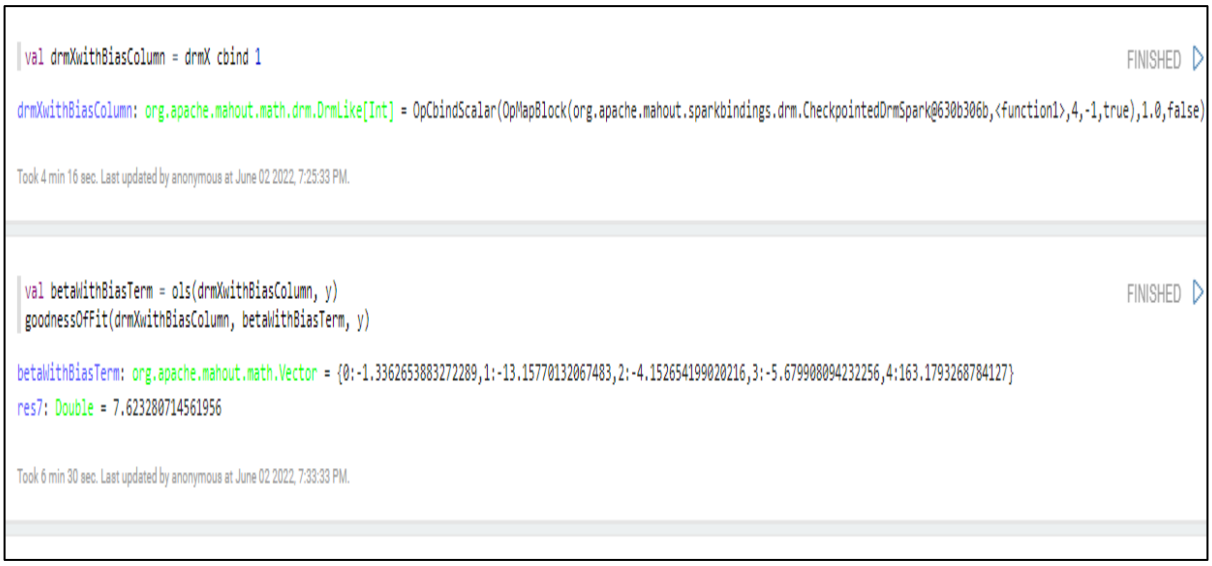
```
def ols(drmX: DrmLike[Int], y: Vector) =
  solve(drmX.t %*% drmX, drmX.t %*% y)(::, 0)

ols: (drmX: org.apache.mahout.math.drm.DrmLike[Int], y: org.apac

Took 1 min 20 sec. Last updated by anonymous at June 02 2022, 7:17:14 PM.
```

Next, we define a function goodnessOfFit that tells how well a model fits the target variable:



```
def goodnessOfFit(drmX: DrmLike[Int], beta: Vector, y: Vector) = {
  val fittedY = (drmX %*% beta).collect(::, 0)
  (y - fittedY).norm(2)
}

goodnessOfFit: (drmX: org.apache.mahout.math.drm.DrmLike[Int], beta: org.apache.mahout.math.Vector,
```

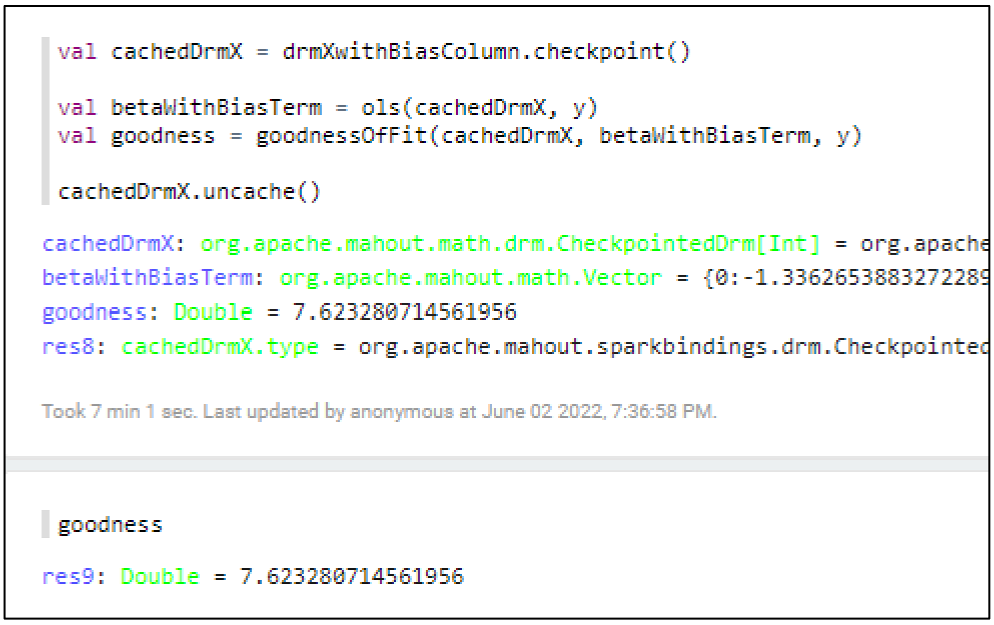
An easy way to add such a bias term to our model is to add a column of ones to the feature matrix \(\mathbf{X}\). The corresponding weight in the parameter vector will then be the bias term. Here is how we add a bias column:



```
val drmXwithBiasColumn = drmX cbind 1                                          FINISHED

drmXwithBiasColumn: org.apache.mahout.math.drm.DrmLike[Int] = OpCbindScalar(OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@630b306b,<function1>,4,-1,true),1.0,false)

Took 4 min 16 sec. Last updated by anonymous at June 02 2022, 7:25:33 PM.
```

```
val betaWithBiasTerm = ols(drmXwithBiasColumn, y)                              FINISHED
goodnessOfFit(drmXwithBiasColumn, betaWithBiasTerm, y)

betaWithBiasTerm: org.apache.mahout.math.Vector = {0:-1.3362653883272289,1:-13.15770132067483,2:-4.152654199020216,3:-5.679908094232256,4:163.1793268784127}
res7: Double = 7.623280714561956

Took 6 min 30 sec. Last updated by anonymous at June 02 2022, 7:33:33 PM.
```

As a further optimization, we can make use of the DSL's caching functionality. We use drmXwithBiasColumn repeatedly as input to a computation, so it might be beneficial to cache it in memory. This is achieved by calling checkpoint(). In the end, we remove it from the cache with uncache:



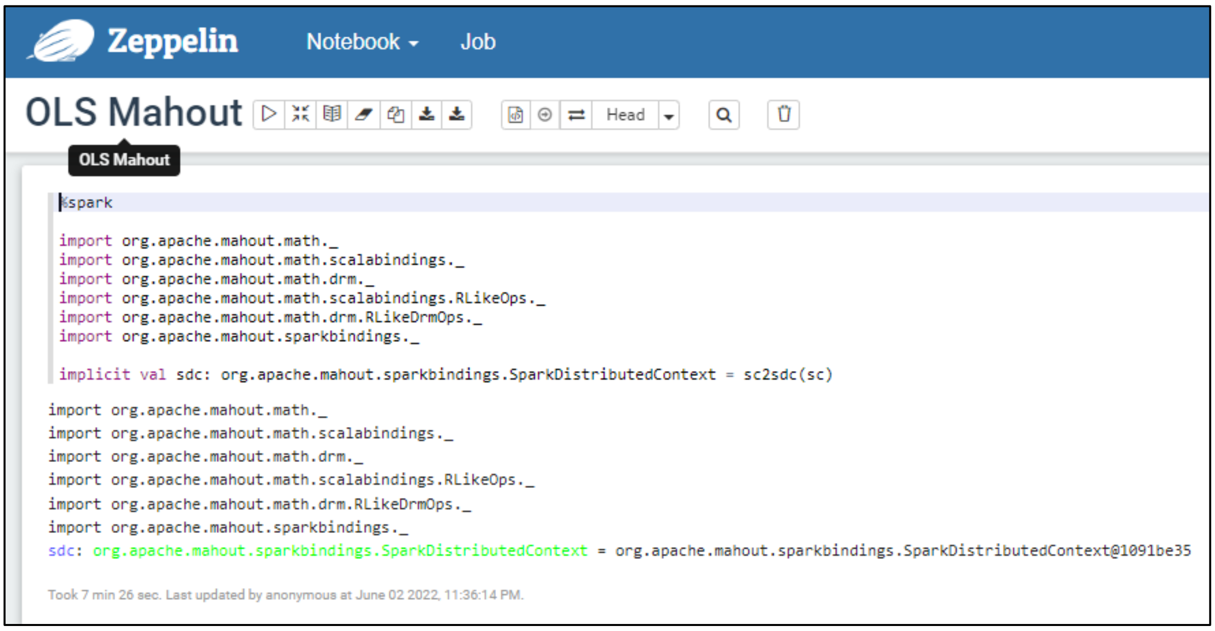
```
val cachedDrmX = drmXwithBiasColumn.checkpoint()

val betaWithBiasTerm = ols(cachedDrmX, y)
val goodness = goodnessOfFit(cachedDrmX, betaWithBiasTerm, y)

cachedDrmX.uncache()

cachedDrmX: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.apache
betaWithBiasTerm: org.apache.mahout.math.Vector = {0:-1.336265388327228
goodness: Double = 7.623280714561956
res8: cachedDrmX.type = org.apache.mahout.sparkbindings.drm.Checkpointed

Took 7 min 1 sec. Last updated by anonymous at June 02 2022, 7:36:58 PM.
```

```
goodness

res9: Double = 7.623280714561956
```

Ordinary Least Squares

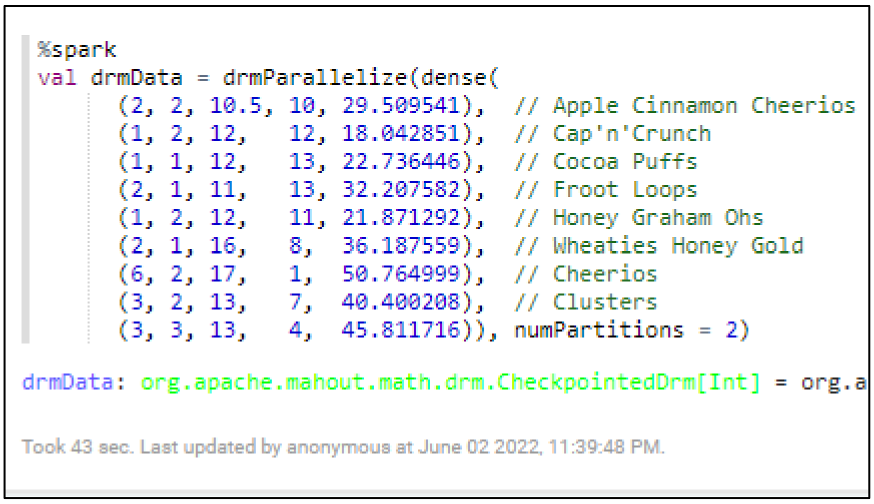
The OrinaryLeastSquares regressor in Mahout implements a closed-form solution to Ordinary Least Squares.

Making the necessary imports:



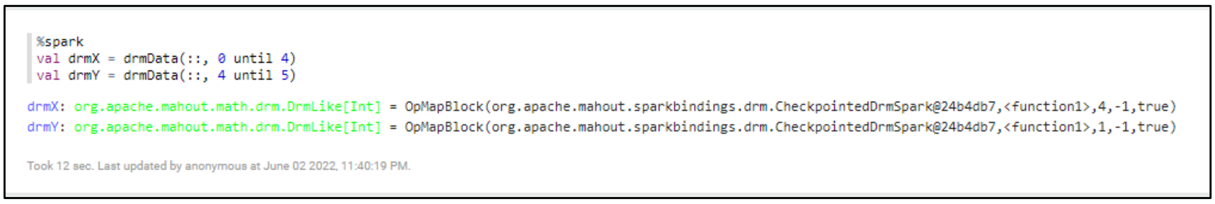
```
%spark

import org.apache.mahout.math._
import org.apache.mahout.math.scalabindings._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.scalabindings.RLikeOps._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.sparkbindings._

implicit val sdc: org.apache.mahout.sparkbindings.SparkDistributedContext = sc2sdc(sc)

import org.apache.mahout.math._
import org.apache.mahout.math.scalabindings._
import org.apache.mahout.math.drm._
import org.apache.mahout.math.scalabindings.RLikeOps._
import org.apache.mahout.math.drm.RLikeDrmOps._
import org.apache.mahout.sparkbindings._
sdc: org.apache.mahout.sparkbindings.SparkDistributedContext = org.apache.mahout.sparkbindings.SparkDistributedContext@1091be35

Took 7 min 26 sec. Last updated by anonymous at June 02 2022, 11:36:14 PM.
```

Loading Data:



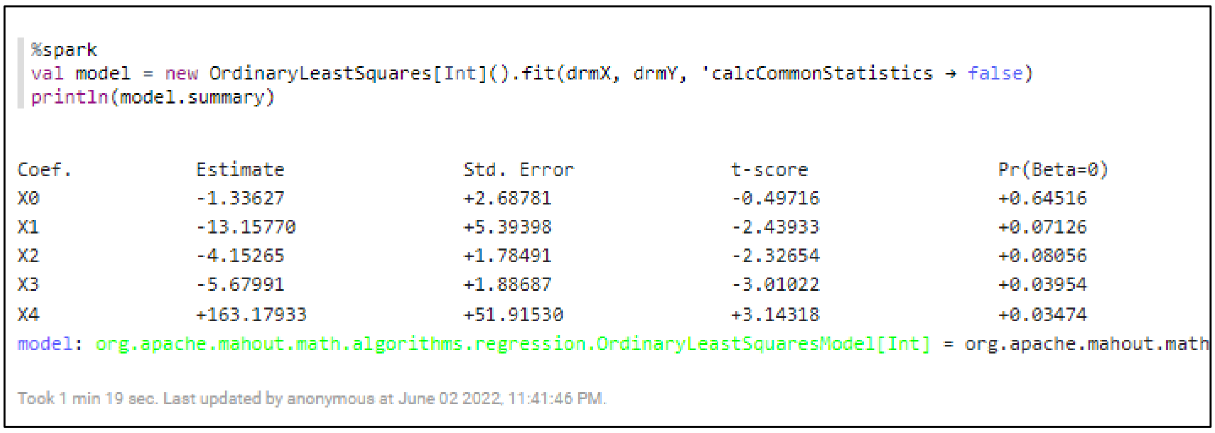
```
%spark
val drmData = drmParallelize(dense(
      (2, 2, 10.5, 10, 29.509541),  // Apple Cinnamon Cheerios
      (1, 2, 12,   12, 18.042851),  // Cap'n'Crunch
      (1, 1, 12,   13, 22.736446),  // Cocoa Puffs
      (2, 1, 11,   13, 32.207582),  // Froot Loops
      (1, 2, 12,   11, 21.871292),  // Honey Graham Ohs
      (2, 1, 16,   8,  36.187559),  // Wheaties Honey Gold
      (6, 2, 17,   1,  50.764999),  // Cheerios
      (3, 2, 13,   7,  40.400208),  // Clusters
      (3, 3, 13,   4,  45.811716)), numPartitions = 2)

drmData: org.apache.mahout.math.drm.CheckpointedDrm[Int] = org.a

Took 43 sec. Last updated by anonymous at June 02 2022, 11:39:48 PM.
```

Declaring Variables:



```
%spark
val drmX = drmData(::, 0 until 4)
val drmY = drmData(::, 4 until 5)

drmX: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@24b4db7,<function1>,4,-1,true)
drmY: org.apache.mahout.math.drm.DrmLike[Int] = OpMapBlock(org.apache.mahout.sparkbindings.drm.CheckpointedDrmSpark@24b4db7,<function1>,1,-1,true)

Took 12 sec. Last updated by anonymous at June 02 2022, 11:40:19 PM.
```

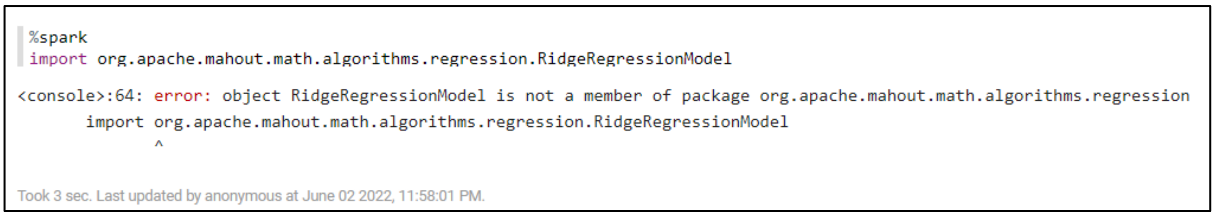
Fitting OLS:



```
%spark
val model = new OrdinaryLeastSquares[Int]().fit(drmX, drmY, 'calcCommonStatistics → false)
println(model.summary)


Coef.          Estimate          Std. Error          t-score          Pr(Beta=0)
X0             -1.33627          +2.68781            -0.49716         +0.64516
X1             -13.15770         +5.39398            -2.43933         +0.07126
X2             -4.15265          +1.78491            -2.32654         +0.08056
X3             -5.67991          +1.88687            -3.01022         +0.03954
X4             +163.17933        +51.91530           +3.14318         +0.03474
model: org.apache.mahout.math.algorithms.regression.OrdinaryLeastSquaresModel[Int] = org.apache.mahout.math

Took 1 min 19 sec. Last updated by anonymous at June 02 2022, 11:41:46 PM.
```

Ridge Regression

I also tried this example mentioned on Apache Mahout Website. However it semmed like the method did not exist anymore.



```
%spark
import org.apache.mahout.math.algorithms.regression.RidgeRegressionModel

<console>:64: error: object RidgeRegressionModel is not a member of package org.apache.mahout.math.algorithms.regression
       import org.apache.mahout.math.algorithms.regression.RidgeRegressionModel
              ^

Took 3 sec. Last updated by anonymous at June 02 2022, 11:58:01 PM.
```

Conclusion:

The main goal of the project was to compare the performance of Apache Mahout and Spark Mllib, however due to lack of valid resources on Mahout implementation, all the goals of the project were not met. However the project does a deep exploration of the classification and regression technologies from Mlib.

Work To be Done

Implement a front-end for the winning model.

Work on a bigger dataset which was not possible due to the specifications of the local machine.

Research more on Mahout