BIG DATA ANALYTICS

Final Project

Project 2

Syed Najeeb Iqbal – 21205

Table of Contents

Introduction to Apache Spark

Apache spark is an analytics engine that can be used for big data processing and machine learning. It was developed by UC Berkeley in 2009. It has been used in wide range of industries such as Netflix, Yahoo and eBay.

Advantages:

- It provides much faster processing speeds as compared to Hadoop with Map reduce as it makes use of in memory computing.
- It is open source with over 1000 contributors from more than 250 organizations. Hence its resources are easily available.
- Spark has libraries that support SQL queries, streaming data, machine learning and graph processing.
- Spark also supports multiples programming languages such as R, scala, Python, SQL and java.

Configuration of the machine used:

Windows edition: Windows 10 Home

Processor: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz

RAM: 24.0 GB

Storage: 1Tb hard disk, 512Gb SSD

System type: 64-bit operating system, x64-based processor

Project introduction

The purpose of this project was to compare the processing times and accuracy for training and testing a machine learning model across two platforms. I.e. apache spark and apache mahout. For spark the coding was done using the jupyter notebook. The pyspark notebook image can be pulled from the docker hub to make use of the jupyter notebook.

In general, the machine learning algorithms performed much faster using apache spark as compared to mahout. One of the reasons for that is Sparks MlLib is built on top of spark which means that it does all the processing in memory. Whereas Mahout is built atop MapReduce, and therefore constrained by disk accesses it slow and does not handle iterative jobs very easily.

Dataset:

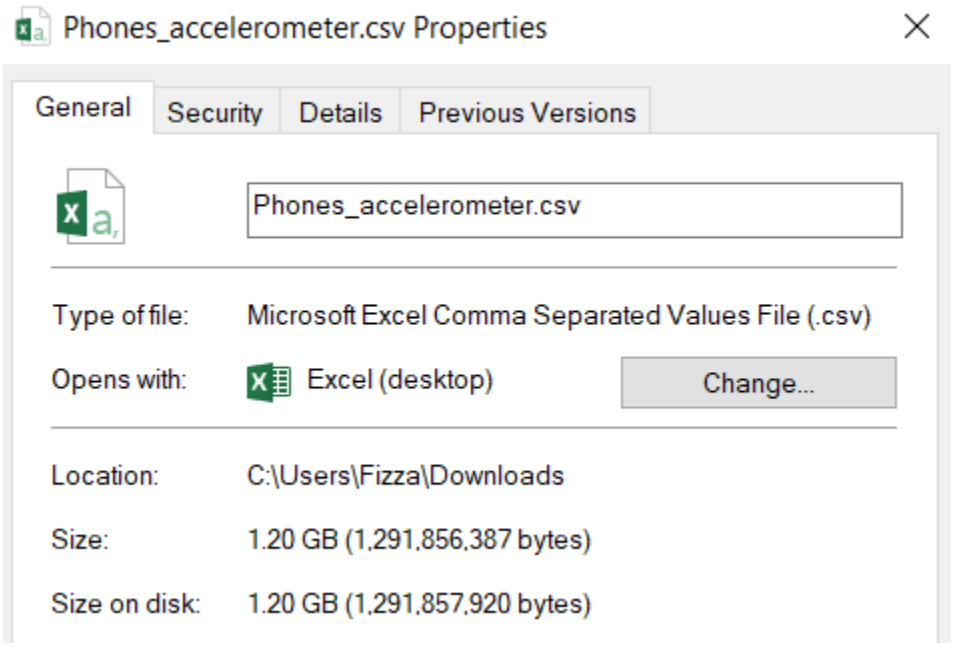
The dataset has been chosen from the UCI machine learning repository. The Heterogeneity Dataset for Human Activity Recognition has data collected from Smartphone and Smart watch sensors and is used to investigate sensor heterogeneities' impacts on human activity recognition.

Human Activity Recognition: Physical activities play a very important role in our physical and mental well-being. The lack of physical activities can negatively affect our well-being. Though people know the importance of physical activities, still they need regular motivational feedback to remain active in their daily life. In order to give them proper motivational feedback, we need to recognize their physical activates first (in our case, the main target group is knowledge workers). Therefore, this research is about recognizing human context (condition, activity and situation etc.) using heterogeneous sensors. If recognized reliably, this context can enable novel wellbeing applications in different fields, for example, healthcare. As a first step to achieve this goal, we recognize some physical activities using smartphone sensors like accelerometer, gyroscope, and magnetometer.

No of rows: 13million

No of columns: 10

Size of data: 1.2 GB



Arrival_Time	Creation_Time	x	y	z	gt
1.424696633908E12	1.424696631913248...	-5.958191	0.6880646	8.135345	stand
1.424696633909E12	1.424696631918284...	-5.95224	0.6702118	8.136536	stand
1.424696633918E12	1.424696631923288...	-5.9950867	0.6535492	8.204376	stand
1.424696633919E12	1.424696631928385...	-5.9427185	0.6761627	8.128204	stand
1.424696633929E12	1.424696631933420...	-5.991516	0.64164734	8.135345	stand
1.424696633929E12	1.424696631938456...	-5.965332	0.6297455	8.128204	stand
1.424696633938E12	1.424696631943522...	-5.991516	0.6356964	8.16272	stand

The six axes from the accelerometer is the x,y,z columns.

Gt = ground truth. This represents activities of the user. Activities include: ‘Biking’, ‘Sitting’, ‘Standing’, ‘Walking’, ‘Stair Up’ and ‘Stair down’. This is the target variable with multiple classes.

Step-by-step instructions for the whole project:

Opening pyspark jupyter notebook:

To open jupyter notebook using docker, we have to make use of the "Jupyter/pyspark-notebook" docker image, and run it on the cli.

```
docker run -it --rm -p 8888:8888 jupyter/pyspark-notebook
```

"-it" is used the run the container in interactive mode.
"–p" is used to define the port.
In the end we have the docker image name that will be used to open the notebook.

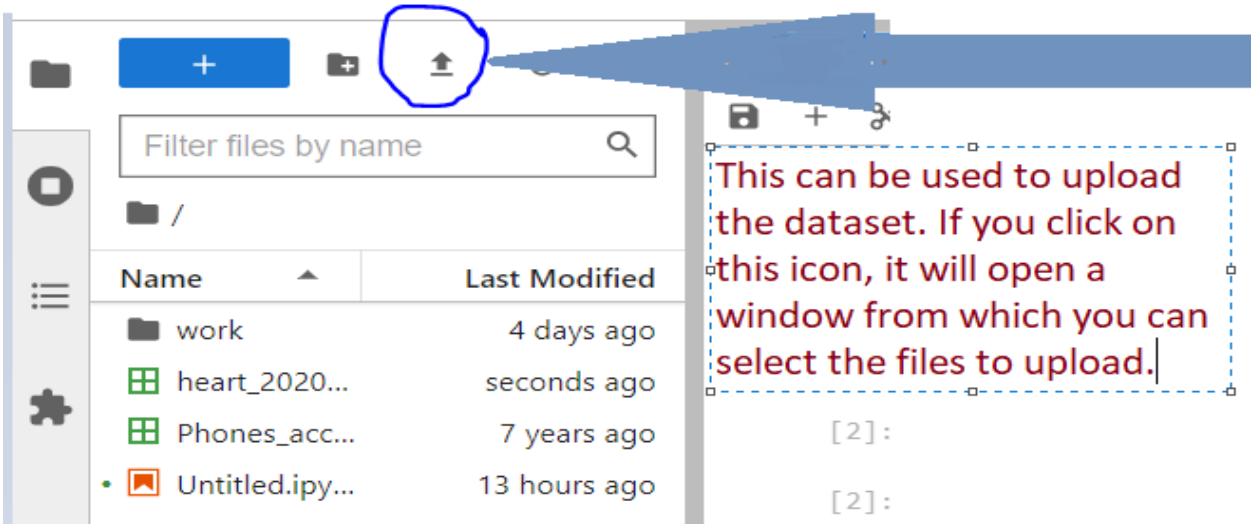
Loading the dataset:

Data has to be loaded in the container for the user to be able to access it to build machine learning models.

- One way is to load the data using this command

  ```
  docker cp Phones_accelerometer.csv stoic_shirley:\home\jovyan
  ```

  "cp": copy
  "Phones_accelerometer.csv": csv file
  "stoic_shirley": name of the container
  "\home\jovyan": file directory where notebook is running
- The other way is to load the dataset directly from the jupyter notebook.



Spark session:

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

The entry point to programming Spark with the Dataset and DataFrame API. A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

To create a SparkSession, we use the following builder pattern:

```
spark = SparkSession \
    .builder \
    .appName('human activity recognition') \
    .getOrCreate()
```

Builder : It is used to construct SparkSession instances

.getOrCreate(): Gets an existing SparkSession or, if there is no existing one, creates a new one based on the options set in this builder.

```
df1 = (spark.read.format("csv").option('header', 'true').load("Phones_accelerometer.csv"))
df1.show()
```

This is used to load the dataset inside the jupyter notebook.

```
+-----------------+--------------------+----------+----------+---------+-----+
|     Arrival_Time|       Creation_Time|         x|         y|        z|   gt|
+-----------------+--------------------+----------+----------+---------+-----+
|1.424696633908E12|1.424696631913248...| -5.958191| 0.6880646| 8.135345|stand|
|1.424696633909E12|1.424696631918284...|  -5.95224| 0.6702118| 8.136536|stand|
|1.424696633918E12|1.424696631923288...|-5.9950867| 0.6535492| 8.204376|stand|
|1.424696633919E12|1.424696631928385...|-5.9427185| 0.6761627| 8.128204|stand|
|1.424696633929E12|1.424696631933420...| -5.991516|0.64164734| 8.135345|stand|
|1.424696633929E12|1.424696631938456...| -5.965332| 0.6297455| 8.128204|stand|
|1.424696633938E12|1.424696631943522...| -5.991516| 0.6356964|  8.16272|stand|
|1.424696633939E12|1.424696631948496...| -5.915344|0.63093567| 8.105591|stand|
|1.424696633951E12|1.424696631953592...| -5.984375| 0.6940155| 8.067505|stand|
|1.424696633952E12|1.424696631960428...| -5.937958|0.71543884| 8.090118|stand|
|1.424696633959E12|1.424696631963663...| -5.902252| 0.6678314| 8.069885|stand|
| 1.42469663396E12|1.424696631968912...|-5.9498596|0.68092346| 8.119873|stand|
|1.424696633966E12|1.424696631973734...|-5.9796143| 0.7416229|8.0841675|stand|
|1.424696633972E12|1.424696631978769...|-5.9617615|0.71424866| 8.155579|stand|
|1.424696633978E12|1.424696631983805...|  -5.95343| 0.7130585| 8.153198|stand|
|1.424696633981E12|1.424696631988840...|-5.8665466| 0.7344818|  8.10083|stand|
|1.424696633989E12|1.424696631993875...| -5.901062| 0.7582855| 8.081787|stand|
|1.424696633991E12|1.424696631999064...|-5.8713074| 0.7190094| 8.192474|stand|
|1.424696634003E12|1.424696632003946...| -5.932007|0.67259216| 8.185333|stand|
|1.424696634004E12|1.424696632010447...| -5.895111| 0.6797333| 8.132965|stand|
+-----------------+--------------------+----------+----------+---------+-----+
only showing top 20 rows
```

Datatypes

```
df1.dtypes
```

```
[('Index', 'string'),
 ('Arrival_Time', 'string'),
 ('Creation_Time', 'string'),
 ('x', 'string'),
 ('y', 'string'),
 ('z', 'string'),
 ('User', 'string'),
 ('Model', 'string'),
 ('Device', 'string'),
 ('gt', 'string')]
```

Arrival_time, Creation_time, x, y and z columns are being shown as string type but in fact they are of integer type. There datatype has to be changed.

Type Casting:

```
from pyspark.sql.functions import col
dataset1 = df1.select(col('Arrival_Time').cast('double'),
                      col('Creation_Time').cast('double'),
                      col('x').cast('float'),
                      col('y').cast('float'),
                      col('z').cast('float'),
                      col('gt'),
                      )
dataset1.show()
```

This will change the datatypes of the columns to our desired choice. It is important to change the datatype, if we don't do it we will not be able to scale the data since the mathematical functions don't work on string datatype.

Checking for Imbalanced Data

```
dataset1.groupBy('gt').count().orderBy('count', ascending=False).show()
```

```
+----------+-------+
|        gt|  count|
+----------+-------+
|      walk|2192401|
|       sit|1991919|
|     stand|1851492|
|      bike|1845557|
|      null|1783200|
|  stairsup|1782010|
|stairsdown|1615896|
+----------+-------+
```

This code will group and count the unique entries in the 'gt' column which is the target variable.

Based on the table on right we can observe that the data is almost balanced with a small difference between the max and min values.

Scaling of data:

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import MinMaxScaler
columns_to_scale = ["Arrival_Time", "Creation_Time", "x","y","z"]
assemblers = [VectorAssembler(inputCols=[col], outputCol=col + "_vec") for col in
columns_to_scale]
scalers = [MinMaxScaler(inputCol=col + "_vec", outputCol=col + "_scaled") for col in
columns_to_scale]
pipeline = Pipeline(stages=assemblers + scalers)
scalerModel = pipeline.fit(dataset1)
scaledData = scalerModel.transform(dataset1)
```

For scaling we first have to convert each column into vector form, then the scaler is applied.

Vector assembler: A feature transformer that merges multiple columns into a vector column.

Scaler: the min max scaler has been used

Pipeline: it is used to merge the vectorization and the scaling process.

Label encoding:

```
from pyspark.ml.feature import StringIndexer
scaledData = StringIndexer(
   inputCol='gt',
   outputCol='gt_index',
   handleInvalid='keep').fit(scaledData).transform(scaledData)
```

The ‘gt’ (ground truth) column which was the target variable, had multiple values that had to be encoded.

InputCol is the name of the column that we have to encode

outputCol is the name that the column will get after the encoding. Spark is going to keep both columns. Which means that the input column will stay as it is and the output column will be added to the dataframe.

Dropping columns:

```
scaledData.columns
```

```
['Arrival_Time',
 'Creation_Time',
 'x',
 'y',
 'z',
 'gt',
 'Arrival_Time_vec',
 'Creation_Time_vec',
 'x_vec',
 'y_vec',
 'z_vec',
 'Arrival_Time_scaled',
 'Creation_Time_scaled',
 'x_scaled',
 'y_scaled',
 'z_scaled']
```

After scaling and label encoding was performed, there were a few extra columns that were added because spark converts each column to vectorized form before scaling and label encoding. These columns are not needed since we are only going to deal with the scaled and the label encoded columns. Therefore those extra columns will be dropped from the dataframe.

```
scaledData = scaledData.drop('gt')
scaledData = scaledData.drop('x')
scaledData = scaledData.drop('y')
scaledData = scaledData.drop('z')
scaledData = scaledData.drop('x_vec')
scaledData = scaledData.drop('y_vec')
scaledData = scaledData.drop('z_vec')
scaledData = scaledData.drop('Arrival_Time')
scaledData = scaledData.drop('Creation_Time')
scaledData = scaledData.drop('Arrival_Time_vec')
scaledData = scaledData.drop('Creation_Time_vec')
```

Features:

In spark, we need to make a separate array named features. This array will contains all the values of different columns in the array form. We will see its implementation in machine learning models. That is one of the difference between pyspark and python programming language. In python we can simply pass the training data for model training without having to make the features column.

```
# Assemble all the features with VectorAssembler
required_features = ['Arrival_Time_scaled',
                     'Creation_Time_scaled',
                     'x_scaled',
                     'y_scaled',
                    'z_scaled'
                   ]
from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler(inputCols=required_features, outputCol='features')
transformed_data = assembler.transform(scaledData)
```

Decision tree classifier:

```
import time
start_time = time.perf_counter()
from pyspark.ml.classification import DecisionTreeClassifier
dtc = DecisionTreeClassifier(labelCol='gt_index',
                             featuresCol='features',
                            )

model = dtc.fit(training_data)
predictions = model.transform(test_data)
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol='gt_index',
    predictionCol='prediction',
    metricName='accuracy')
accuracy = evaluator.evaluate(predictions)
print('Test Accuracy = ', accuracy)
end_time = time.perf_counter()

training_time = end_time - start_time


print("The time taken to train the data is: %0.3f seconds" %training_time)
```

```
Test Accuracy =  0.4260588555872221
The time taken to train the data is: 119.447 seconds
```

There are a few common commands that we are going to see in each classifier.

1. labelCol: this is target column
2. featuresCol: this is the name of the features column. Since I set the name to ‘features’, hence I have features over here. You can use any other name that you like when defining the features column. The code is given in the previous page.
3. Dtc.fit: this command is used to fit the machine learning model on our training data.
4. For evaluation we make use of the multiclass evaluator, since the target variable has multiple classes.
5. That metric that has been used to evaluate the model performance is accuracy, since the data was balanced.
6. The time that it takes for model training and testing will printed in the end

Random Forrest Classifier:

There were three types of random forrest models that were used, with different hyper parameters. The one with a max depth of 10 gave the most optimum results.

Model 1:

```
import time
start_time = time.perf_counter()
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol='gt_index',
                            featuresCol='features',
                            )

model = rf.fit(training_data)
predictions = model.transform(test_data)
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol='gt_index',
    predictionCol='prediction',
    metricName='accuracy')
accuracy = evaluator.evaluate(predictions)
print('Test Accuracy = ', accuracy)
end_time = time.perf_counter()

training_time = end_time - start_time

print("The time taken to train the data is: %0.3f seconds" %training_time)
```

```
Test Accuracy =  0.48132164066618083
The time taken to train the data is: 156.649 seconds
```

This particular model was trained with the default hyper parameters.

Model 2:

```
import time
start_time = time.perf_counter()
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol='gt_index',
                            featuresCol='features',
                            maxDepth = 5)

model = rf.fit(training_data)
predictions = model.transform(test_data)
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
   labelCol='gt_index',
   predictionCol='prediction',
   metricName='accuracy')
accuracy = evaluator.evaluate(predictions)
print('Test Accuracy = ', accuracy)
end_time = time.perf_counter()

training_time = end_time - start_time

print("The time taken to train the data is: %0.3f seconds" %training_time)
```

```
Test Accuracy =  0.48132164066618083
The time taken to train the data is: 167.276 seconds
```

For this model, the maxDepth was kept as 5. This model took more time than the previous one, will almost no improvement in accuracy.

Model 3:

```
import time

start_time = time.perf_counter()
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol='gt_index',
                            featuresCol='features',
                            maxDepth = 10)

model = rf.fit(training_data)
predictions = model.transform(test_data)
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol='gt_index',
    predictionCol='prediction',
    metricName='accuracy')
accuracy = evaluator.evaluate(predictions)
print('Test Accuracy = ', accuracy)
end_time = time.perf_counter()

training_time = end_time - start_time

print("The time taken to train the data is: %0.3f seconds" %training_time)
```

```
Test Accuracy =  0.657033277133314
The time taken to train the data is: 271.433 seconds
```

For this all the parameters were kept to default other than the maxDepth. Which was kept as 10.

This model took the most time but gave the best accuracy across all the models that were tried.

Logistic Regression:

```
import time
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(featuresCol = 'features', labelCol = 'gt_index', maxIter=10)

start_time = time.perf_counter()
lrModel = lr.fit(training_data)
predictions = lrModel.transform(test_data)

# Evaluate our model
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(
   labelCol='gt_index',
   predictionCol='prediction',
   metricName='accuracy')

accuracy = evaluator.evaluate(predictions)
print('Test Accuracy = ', accuracy)

end_time = time.perf_counter()
training_time = end_time - start_time
print("The time taken to train the data is: %0.3f seconds" %training_time)
```

```
Test Accuracy =  0.3269231161852496
The time taken to train the data is: 349.731 seconds
```

APACHE MAHOUT:

Mahout image:

```
docker run -it michabirklbauer/mahout:latest
```

This will run the latest mahout image using docker.

Uploading the dataset:

In the command line type this command. This will copy the csv file to the container

```
docker cp Phones_accelerometer.csv festive_lederberg:\apache
```

Now from the powershell, we can upload the dataset to mahout. Before uploading the dataset to mahout, we need to run this command since it is used to load dataframes.

```
val spark = org.apache.spark.sql.SparkSession.builder
```

Now to upload the dataframe, use this command

```
val df = spark.read.format("csv").option("header", "true").option("mode",
"DROPMALFORMED").load("Phones_accelerometer.csv")
```

```
      df.show()
+-----+-------------+------------------+------------------+------------------+-----------------+----+------+--------+-----
|Index| Arrival_Time|     Creation_Time|                 x|                 y|                z|User| Model|  Device|   gt
+-----+-------------+------------------+------------------+------------------+-----------------+----+------+--------+-----
|    0|1424696633908|1424696631913248572|         -5.958191|         0.6880646|         8.135345|   a|nexus4|nexus4_1|stand
|    1|1424696633909|1424696631918283972|          -5.95224|         0.6702118|         8.136536|   a|nexus4|nexus4_1|stand
|    2|1424696633918|1424696631923288855|        -5.9950867|0.6535491999999999|         8.204376|   a|nexus4|nexus4_1|stand
|    3|1424696633919|1424696631928385290|        -5.9427185|0.6761626999999999|         8.128204|   a|nexus4|nexus4_1|stand
|    4|1424696633929|1424696631933420691| -5.991516000000001|        0.64164734|         8.135345|   a|nexus4|nexus4_1|stand
|    5|1424696633929|1424696631938456091|         -5.965332|         0.6297455|         8.128204|   a|nexus4|nexus4_1|stand
|    6|1424696633938|1424696631943522009| -5.991516000000001|0.6356963999999999|          8.16272|   a|nexus4|nexus4_1|stand
|    7|1424696633939|1424696631948496374|         -5.915344|        0.63093567|         8.105591|   a|nexus4|nexus4_1|stand
|    8|1424696633951|1424696631953592810|         -5.984375|         0.6940155|         8.067505|   a|nexus4|nexus4_1|stand
|    9|1424696633952|1424696631960428747|         -5.937958|        0.71543884|8.090117999999999|   a|nexus4|nexus4_1|stand
|   10|1424696633959|1424696631963663611|         -5.902252|0.6678314000000001|8.069885000000001|   a|nexus4|nexus4_1|stand
```

Dropping columns:

```
import org.apache.spark.sql.Column
val colsToRemove = Seq("Index","User","Model","Device")

val filteredDF = df.select(df.columns .filter(colName =>
!colsToRemove.contains(colName)).map(colName => new Column(colName)): _*)
```

Datatypes:

```
import spark.sqlContext.implicits._
filteredDF.dtypes.foreach(f=>println(f._1+","+f._2))
```

```
Arrival_Time,StringType
Creation_Time,StringType
x,StringType
y,StringType
z,StringType
gt,StringType
```

Type casting:

```
val dfX = filteredDF.select(filteredDF.columns.map(x => col(x).cast("float")): _*)
```

This is used to change the datatype from string to float