CS551G - Data Mining and Visualisation. Assessment 2 report.

Task 1: Description of Distributed Learning Big Data Ecosystem (10/50) – (~max 600 words)

Using your own words, the lecture material and any other relevant sources, explain the distributed big data processing ecosystem. Your description should cover the following points at a technical level:

- Distributed File Systems (e.g. HDFS)
- Resource Manager and Scheduler (e.g. YARN)
- Volume, Velocity, Variety, Value, Veracity (Big Data Characteristics)
- Fault Tolerance and Resilience
- Apache Spark
- Spark ML/PySpark
- SparkSQL
- Docker containers (e.g. Kubernetes)

Answer: Every data processing starts from obtaining the data, initial analysis on what type of data we obtain, what structure does it have, and how we are going to store this data. Most of the companies use Distributed File Systems where employees from many different departments (HR, Management, Sales, Development etc) have access and store the relevant information (data sets) on commodity hardware. An example of the distributed file system is Hadoop Distributed File System (HDFS). (IBM, analytics, available at https://www.ibm.com/analytics/hadoop/hdfs , 2021). The storage cannot be local as we need to process huge amounts of data at the same time (example: millions of tweets every hour). The type of data can be different like videos, images, documents, tables etc.

Another major component of Apache Hadoop (like HDFS) is YARN. YARN is the distributed operating system managing big data. The YARN Resource Manager arbitrates the resources between all applications in the system and has two main components Scheduler and ApplicationsManager. The Scheduler allocates resources to the various running applications subjects. The ApplicationsManager accepting job submissions provide service on restarting the application on failure and negotiating the first container for executing. (Apache, Apache Hadoop YARN, available at https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html , 2021).

The approach to show what constitutes the data is defined with the 3 Vs. The volume shows how much data we have and the processing capacity of the data. Velocity shows how quick the data come. Variety shows what type of data we have (images, metadata etc). More definitions to

describe the data can be used. Veracity describes the quality of the data, its reliability. Value of the data describes the amount of material value associated with the data for the company. The idea of describing the big data that we cannot limit this data to one specific concept. The big data description is not about the size of the data, but about what tools we need to use to analyse this data.

The fault tolerance and resilience of distributed file system is a core architectural goal of HDFS. Heartbeat messages help to find and isolate dead DataNodes on hundreds of thousands of server machines. Checksum checking on the content of HDFS files helps data integrity verification of HDFS files. The synchronous updating of multiple copies of central data structures of HDFS helps to prevent these structures from corruption. The distributed edit log (journal) is one of the options used to increase resilience against failures. The snapshot feature stores a copy of data at a particular instance of time. The snapshot feature used to roll back a corrupted HDFS to a previously known good point in time. (Apache, HDFS Architecture, available at https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html , 2021).

Apache Spark is a data processing tool that can process large data sets or can distribute the data processing tasks across multiple computers. The Apache Spark can work in a tandem with other distributed computing tools. (Apache, Apache Spark, available at http://spark.apache.org/ , 2021)

Spark ML is the package available in Spark that helps users to create and tune practical machine learning pipelines (sequences of consecutive algorithms and methods to process data and learn from them). PySpark is an API to support Python while working in Spark.

Spark SQL is a Spark module for structured data (such as Relational Database Data) processing. Spark SQL allows querying data stored both in Spark's distributed data sets (Resilient Distributed Datasets RDDs) and in external sources. (Databricks, Spark SQL, available at https://databricks.com/glossary/what-is-spark-sql#:~:text=Spark%20SQL%20is%20a%20Spark,a%20distributed%20SQL%20query%20engine.&text=It%20also%20provides%20powerful%20integration,query%20processing%20with%20machine%20learning , 2021).

The Docker container is a standard unit of software that packages code and all its dependencies into the unit, so the application runs from one computing environment to another. The Kubernetes for example takes care of scaling and wrapping for your application, provides deployment patterns and more. The Kubernetes is the framework that helps to run distributed system resiliently. (Kubernetes, Overview, available at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ , 2021).

Task 2: Develop distributed models in apache spark to classify gas turbines (40/50) – (~max 1800 words)

The problem we aim at tackling has been clearly described and defined earlier. This task includes ***five*** subtasks, each of which bears its own marks.

Subtasks:

1. Create an apache spark environment and load the dataset provided. You may use CoLab or a jupyter notebook. Please create a table providing summary statistics of this dataset, i.e. mean values, range, standard deviations, min/max values, median values and 25%/50%/75% percentile values. Comment on whether there are any missing values present throughout. The column that contains the labels (normal/abnormal) should not be part of the table (**3 marks**).

Answer: From the start following steps from practical. Setting the Java environment, downloading and installing spark libraries, assigning, and checking GPU, setting the spark environment variable, setting the Spark session, uploading the dataset file provided. The steps performed are the same as on practical except adding a config line when setting the Spark session, the code can be found in the python notebook provided with this document. The Spark is not working with the Excel spreadsheets directly, so we need to upload an external library. I have used spark-excel library, the description can be found at https://index.scala-lang.org/crealytics/spark-excel/spark-excel/0.8.4?target=_2.11. I have downloaded the jar file from https://jar-download.com/artifacts/com.crealytics/spark-excel_2.11/0.8.2/source-code, and copied the "spark-excel_2.11-0.8.2.jar" file into the collab spark folder: '/content/spark-2.4.7-bin-hadoop2.7/jars'(once I did it I don't need to copy it again to make it work).

The next chunk of code is to load the data from an Excel file with a header and inferring the schema. After that, I created a new data frame with no labels (normal/abnormal). Displayed the length of the data in the table which is 996 rows, and a number of columns in the table which is 12. Command display shows the column name and type of data this column holds (all 12 columns have a type double). Then I showed the top five rows of the table and printed the schema of the data frame. Command 'describe' shows the information we need to display but does not show the percentile values, which another command 'summary' does.

```
#read the file using spark read command
data = spark.read.format("com.crealytics.spark.excel").option("useHeader",
"true").option("inferSchema",
"true").load("CS551G_DMV_Assessment_2_Dataset.xlsx")

# dropping columns
copy_data = data.drop('Status')
print(copy_data.count(), len(copy_data.columns))
display(copy_data)

# print out the schema
copy_data.show(5)
copy_data.printSchema()

#show the data description
```

```
copy_data.describe().show()
copy_data.summary().show()
```

```
996 12
DataFrame[Temperature_Sensor_1: double, Temperature_Sensor_2: double, Temperature_Sensor_3: double, Flow_Rate_Sensor_1: double, Flow_Rate_Sensor_2: double, Flow_Rate_Sensor_3: double,
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
|Temperature_Sensor_1|Temperature_Sensor_2|Temperature_Sensor_3|Flow_Rate_Sensor_1|Flow_Rate_Sensor_2|Flow_Rate_Sensor_3|Pressure _sensor_1|Pressure _sensor_2|Pressure _sensor_3|Vibra
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
|            4.5044|            0.7443|              6.34|           1.9052|          29.5315|           0.8647|           2.2044|            6.048|           14.4659|
|            4.4284|            0.9073|            5.6433|           1.6232|          27.5032|           1.4704|           1.9929|           5.9856|           20.8356|
|            4.5291|            1.0199|             6.113|           1.0565|          26.4271|           1.9247|            1.942|           6.7162|            5.3358|
|            5.1727|            1.0007|            7.8589|           0.2765|          25.1576|            2.609|           2.9234|           6.7485|            1.9017|
|            5.2258|            0.6125|            7.9504|           0.1547|          24.0765|           3.2113|           4.4563|           5.8411|            0.5077|
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
only showing top 5 rows

root
 |-- Temperature_Sensor_1: double (nullable = true)
 |-- Temperature_Sensor_2: double (nullable = true)
 |-- Temperature_Sensor_3: double (nullable = true)
 |-- Flow_Rate_Sensor_1: double (nullable = true)
 |-- Flow_Rate_Sensor_2: double (nullable = true)
 |-- Flow_Rate_Sensor_3: double (nullable = true)
 |-- Pressure _sensor_1: double (nullable = true)
 |-- Pressure _sensor_2: double (nullable = true)
 |-- Pressure _sensor_3: double (nullable = true)
 |-- Vibration_Sensor_1: double (nullable = true)
 |-- Vibration_Sensor_2: double (nullable = true)
 |-- Vibration_Sensor_3: double (nullable = true)
```

```
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
|summary|Temperature_Sensor_1|Temperature_Sensor_2|Temperature_Sensor_3|Flow_Rate_Sensor_1|Flow_Rate_Sensor_2|Flow_Rate_Sensor_3|Pressure _sensor_1|Pressure _sensor_2|Pressure _sensor_3|Pressure _sensor
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
|  count|               996|               996|               996|               996|               996|               996|               996|               996|               996|          9
|   mean| 4.9995738935742935|  6.379273152610439|  9.228112114457828| 7.355272086345389|14.199126817269088| 3.077957961847389| 5.749233845381521| 4.997002126506022| 8.164562875502
| stddev|  2.764855518719287|  2.312568802265547|  2.532172890894333| 4.354778181006016|11.680045382938975|2.1260909561041483|2.5261356617270914| 4.165489943788825| 6.173261395362
|    min|            0.0082|            0.0403|          2.583966|            0.0623|            0.0248|          0.008262|          0.001224|            0.0058|                 0
|    max|           12.1298|           11.9284|           15.7599|          17.235858|           67.9794|          10.242738|           12.6475|          16.55562|          36.1864
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
```

```
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
|summary|Temperature_Sensor_1|Temperature_Sensor_2|Temperature_Sensor_3|Flow_Rate_Sensor_1|Flow_Rate_Sensor_2|Flow_Rate_Sensor_3|Pressure _sensor_1|Pressure _sensor_2|Pressure _sensor_3|Pressure _sensor
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
|  count|               996|               996|               996|               996|               996|               996|               996|               996|               996|          9
|   mean| 4.9995738935742935|  6.379273152610439|  9.228112114457828| 7.355272086345389|14.199126817269088| 3.077957961847389| 5.749233845381521| 4.997002126506022| 8.164562875502
| stddev|  2.764855518719287|  2.312568802265547|  2.532172890894333| 4.354778181006016|11.680045382938975|2.1260909561041483|2.5261356617270914| 4.165489943788825| 6.173261395362
|    min|            0.0082|            0.0403|          2.583966|            0.0623|            0.0248|          0.008262|          0.001224|            0.0058|                 0
|    25%|            2.8894|            4.9283|            7.5018|          3.425364|            5.0037|            1.4149|            4.0054|            1.5721|              3.18
|    50%|            4.8788|            6.4691|            9.3347|            7.0633|           11.7085|           2.671788|            5.7394|            3.8586|              6.74
|    75%|            6.7877|            8.0988|           11.0468|           10.9103|           20.2629|            4.5025|            7.4988|            7.5993|             11.25
|    max|           12.1298|           11.9284|           15.7599|          17.235858|           67.9794|          10.242738|           12.6475|          16.55562|          36.1864
+-------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+-----------
```

```
from pyspark.sql.functions import col,sum
copy_data.select(*(sum(col(c).isNull().cast("int")).alias(c) for c in copy_data.columns)).show()
```

```
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
|Temperature_Sensor_1|Temperature_Sensor_2|Temperature_Sensor_3|Flow_Rate_Sensor_1|Flow_Rate_Sensor_2|Flow_Rate_Sensor_3|Pressure _sensor_1|Pressure _sensor_2|Pressure _sensor_3|Vibra
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
|                 0|                 0|                 0|                0|                0|                0|                0|                0|                0|
+------------------+------------------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
```

The last chunk of code is to display missing values, as we can see there are no missing values in our data set. In the code 2 cells below I double checked the missing values with Handyspark function isnull, no missing values found.

2. Visualise the data as follows: create <u>two</u> plots, i.e. box plot and a scatter plot. The box plot shall include the two classes (normal/abnormal) in the x-axis, and the “Vibration_Sensor_1” in the y-axis. The scatter plot shall include the feature “Vibration_Sensor_2” with the graphs of both classes appearing in the same plot (different colour or symbol). Please elaborate on what information one can obtain from each of these two plots (**hint: handyspark)** (**7 marks**).

<u>Answer:</u> Before we make a boxplot we need to prepare the data to use it with Handyspark library. First we need to install the handyspark into our Colab environment.

```
!pip install handyspark
```

After Handyspark installed we need to restart the runtime and rerun all steps from the start. Now when the handyspark is set into the environment, we can start adjusting the data set to plot the data. Import handispark and transfer the data into the handyspark format.

```
from handyspark import *

hdf = data.toHandy()
hdf
```

For the box plot we need the 'Vibration_Sensor_1' column data, which we extracting from the data frame and filtering into the two sets ('Normal'and 'Abnormal') then join these sets into one HandyFrame (the code shown below).

```
vib_sens1_norm = hdf.filter(hdf['Status'] == 'Normal')
vib_sens1_norm = vib_sens1_norm.select("Vibration_Sensor_1")
vib_sens1_norm = vib_sens1_norm.selectExpr("Vibration_Sensor_1 as Normal")
print(vib_sens1_norm.count(), len(vib_sens1_norm.columns))
display(vib_sens1_norm)
vib_sens1_norm.show(5)

vib_sens1_abnorm = hdf.filter(hdf['Status'] == 'Abnormal')
vib_sens1_abnorm = vib_sens1_abnorm.select("Vibration_Sensor_1")
vib_sens1_abnorm = vib_sens1_abnorm.selectExpr("Vibration_Sensor_1 as
Abnormal")
print(vib_sens1_abnorm.count(), len(vib_sens1_abnorm.columns))
display(vib_sens1_abnorm)
vib_sens1_abnorm.show(5)

import pyspark.sql.functions as F
from pyspark.sql.functions import monotonically_increasing_id, row_number
from pyspark.sql.window import Window

# since there is no common column between these two dataframes add row_index so
that it can be joined
vib_sens1_norm=vib_sens1_norm.withColumn('row_index',
row_number().over(Window.orderBy(monotonically_increasing_id())))
vib_sens1_abnorm=vib_sens1_abnorm.withColumn('row_index',
row_number().over(Window.orderBy(monotonically_increasing_id())))

vib_sens1 = vib_sens1_norm.join(vib_sens1_abnorm,
on=["row_index"]).drop("row_index")
print(vib_sens1.count(), len(vib_sens1.columns))
display(vib_sens1)
vib_sens1.show(5)
```

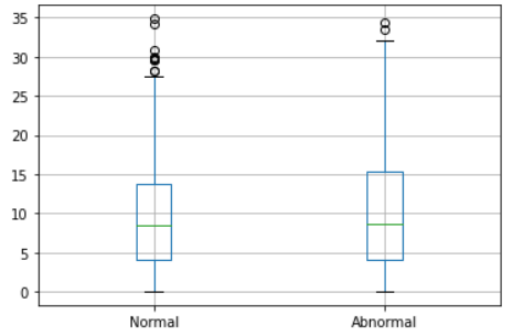
The output of this code is:

```
498 1
HandyFrame[Normal: double]
+-------+
| Normal|
+-------+
| 21.648|
| 0.0646|
|11.0779|
| 1.8463|
|   9.37|
+-------+
only showing top 5 rows

498 1
HandyFrame[Abnormal: double]
+--------+
|Abnormal|
+--------+
| 17.5954|
| 18.3339|
| 10.9209|
|  10.884|
|  3.8588|
+--------+
only showing top 5 rows

498 2
HandyFrame[Normal: double, Abnormal: double]
+-------+--------+
| Normal|Abnormal|
+-------+--------+
| 21.648| 17.5954|
| 0.0646| 18.3339|
|11.0779| 10.9209|
| 1.8463|  10.884|
|   9.37|  3.8588|
+-------+--------+
only showing top 5 rows
```

Now we can make a box plot.

```
vib_sens1.cols[['Normal', 'Abnormal']].boxplot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd2609f8450>
```

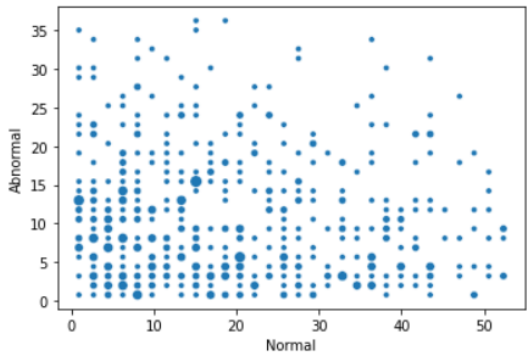


From the box plot, we can see that the lower quartile and median have not much visible difference for both normal and abnormal data, but the upper quartile is above 15 for Abnormal when for Normal the data is above 15. There are more outliers on the top for Normal data than for Abnormal, which means that in the Abnormal set we have a more dense spread for the top values than in the Normal set or outliers have higher values.

The same technique is used to prepare the `"Vibration_Sensor_2"` data for scatter plot, the code can be found in the notebook supplied for this assessment. The resulting graph is:

```
vib_sens2.cols[['Normal', 'Abnormal']].scatterplot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd25e67dc90>
```



From the scatter plot we can see that Abnormal data spreads higher above 50 when the top value of data in Normal is just above 35. This supports our conclusions made on the box plot graph. The data in the Abnormal set spread more with the higher values.

3. Using the apache spark ecosystem, such as pyspark, sparksql or any other component needed, please develop and train a random forest model with a binary output to classify the condition of gas turbines, i.e. normal or abnormal, based on how they operate. Split the dataset provided into training (70%) and test (30%) sets. Please use the training set to train your developed model, keeping the test set only for evaluating its performance in unseen data (**15 marks**).

For this task, we start by assembling the features vector from our data set. The data in this vector are all data columns from the data set given except for the ‘Status’ column. The ‘Status’ column we will use to label the data into the classes.

```
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols =
['Temperature_Sensor_1','Temperature_Sensor_2','Temperature_Sensor_3','Flow_Rat
e_Sensor_1','Flow_Rate_Sensor_2','Flow_Rate_Sensor_3','Pressure
_sensor_1','Pressure _sensor_2','Pressure
_sensor_3','Vibration_Sensor_1','Vibration_Sensor_2','Vibration_Sensor_3'],
outputCol = 'features')
```

Then we need to transform and index our data into a format we can use to build the Random tree model.

```
output = assembler.transform(data)
```

```
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol = 'Status', outputCol = 'StatusIndex')
outputFixed = indexer.fit(output).transform(output)
outputFixed.printSchema()
```

```
root
 |-- Status: string (nullable = true)
 |-- Temperature_Sensor_1: double (nullable = true)
 |-- Temperature_Sensor_2: double (nullable = true)
 |-- Temperature_Sensor_3: double (nullable = true)
 |-- Flow_Rate_Sensor_1: double (nullable = true)
 |-- Flow_Rate_Sensor_2: double (nullable = true)
 |-- Flow_Rate_Sensor_3: double (nullable = true)
 |-- Pressure _sensor_1: double (nullable = true)
 |-- Pressure _sensor_2: double (nullable = true)
 |-- Pressure _sensor_3: double (nullable = true)
 |-- Vibration_Sensor_1: double (nullable = true)
 |-- Vibration_Sensor_2: double (nullable = true)
 |-- Vibration_Sensor_3: double (nullable = true)
 |-- features: vector (nullable = true)
 |-- StatusIndex: double (nullable = false)
```

Finally, we are creating the data frame with features and labels (indexed 'Status' -> `'StatusIndex'`), and creating the training set 'train' (randomly selected 70% of data) and the testing set 'test' (randomly selected 30% of data).

```
final_df = outputFixed.select('features', 'StatusIndex')
final_df.show(3)
```

```
+--------------------+-----------+
|            features|StatusIndex|
+--------------------+-----------+
|[4.5044,0.7443,6....|        1.0|
|[4.4284,0.9073,5....|        1.0|
|[4.5291,1.0199,6....|        1.0|
+--------------------+-----------+
only showing top 3 rows
```

```
train, test = final_df.randomSplit([0.7, 0.3])
```

We are importing the Random forest classifier from the ml pyspark library, creating classifier for our data with the random forest model, and training this model with the 'train' data.

```
train, test = final_df.randomSplit([0.7, 0.3])
```

```
from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(labelCol = 'StatusIndex', featuresCol = 'features')
```

```
rf_model = rf.fit(train)
```

4. Use the following three metrics to report the model's performance, i.e. Precision/Recall, Accuracy and Area under the curve (AUROC). When reporting performance, please only use the test set created by yourselves (**5 marks**).

Before the evaluation, we need to make the predictions on our 'test' data.

```
rf_predictions = rf_model.transform(test)
# Select example rows to display.
rf_predictions.show(5)
```

```
+--------------------+-----------+--------------------+--------------------+----------+
|            features|StatusIndex|       rawPrediction|         probability|prediction|
+--------------------+-----------+--------------------+--------------------+----------+
|[0.0851,5.8301,7....|        1.0|[5.12753749512918...|[0.25637687475645...|       1.0|
|[0.2325,4.9466,6....|        1.0|[4.86397857699482...|[0.24319892884974...|       1.0|
|[0.2811,4.52,8.04...|        1.0|[15.2584437405230...|[0.76292218702615...|       0.0|
|[0.3983,5.5316,5....|        1.0|[10.5161675758050...|[0.52580837879025...|       0.0|
|[0.7059,8.4814,8....|        1.0|[12.8212715902953...|[0.64106357951476...|       0.0|
+--------------------+-----------+--------------------+--------------------+----------+
only showing top 5 rows
```

I used the '`BinaryClassificationEvaluator`' pyspark.ml library to find the Area under the ROC value.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

binary_evaluator = BinaryClassificationEvaluator(labelCol = 'StatusIndex')
#print('Random Forest areaUnderROC:' , binary_evaluator.evaluate(rf_predictions))
print("Test Area Under ROC: " + str(binary_evaluator.evaluate(rf_predictions, {binary_evaluator.metricName: "areaUnderROC"})))
```

```
Test Area Under ROC: 0.9223132290891051
```

The '`MulticlassClassificationEvaluator`' can be used to find accuracy.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

acc_evaluator = MulticlassClassificationEvaluator(
    labelCol="StatusIndex", predictionCol="prediction", metricName="accuracy")
accuracy = acc_evaluator.evaluate(rf_predictions)
print("Test Accuracy = %g" % (accuracy))
```

```
Test Accuracy = 0.868167
```

The area under the Precision/Recall curve can be found with the '`BinaryClassificationEvaluator`'.

```
print("Test Area Under PR: " + str(binary_evaluator.evaluate(rf_predictions, {binary_evaluator.metricName: "areaUnderPR"})))
```

```
Test Area Under PR: 0.9381752169280615
```

5. Repeat steps 3 and 4 but using a multilayer perceptron classifier this time. (**10 marks**)

For this task, I renamed the 'StatusIndex' column into the 'label'. The train/test split stays the same at 0.7 and 0.3 respectively.

```
final_df.show(3)
mpc_final_df = final_df.withColumnRenamed('StatusIndex', 'label')
train, test = mpc_final_df.randomSplit([0.7, 0.3])
train.show(3)
test.show(3)
```

```
+--------------------+-----------+
|            features|StatusIndex|
+--------------------+-----------+
|[4.5044,0.7443,6....|        1.0|
|[4.4284,0.9073,5....|        1.0|
|[4.5291,1.0199,6....|        1.0|
+--------------------+-----------+
only showing top 3 rows

+--------------------+-----+
|            features|label|
+--------------------+-----+
|[0.0851,5.8301,7....|  1.0|
|[0.0861,4.8303,6....|  1.0|
|[0.1117,6.2585,6....|  1.0|
+--------------------+-----+
only showing top 3 rows

+--------------------+-----+
|            features|label|
+--------------------+-----+
|[0.1382,4.9429,8....|  1.0|
|[0.2325,4.9466,6....|  1.0|
|[0.2985,5.9555,8....|  1.0|
+--------------------+-----+
only showing top 3 rows
```

Next, I imported the necessary libraries for the multilayer perceptron classifier model. I specified the layers for the neural network. The first (input) layer takes as an input 12 columns (features). I have specified four intermediate (hidden) layers with the size 16, 12, 8 and 4. The output layer is the size of 2 to predict normal or abnormal status. These values can be changed to improve the performance of the model the same as the number of layers may affect performance as well. The trainer has a limit of 1000 iterations, the block size is 128, and seed 1234 for the experiment could be repeated. The iterations limit and block size values can be adjusted to improve model performance.

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# specify layers for the neural network:
# input layer of size 12 (features), 4 intermediate of size 16, 12, 8 and 4
# and output of size 2 (classes)
layers = [12, 16, 12, 8, 4, 2]

# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=1000, layers=layers, blockSize=128, seed=1234)

# train the model
mpc_model = trainer.fit(train)
```

The prediction accuracy on the test data is 82.5%.

```
# compute accuracy on the test set
mpc_pred = mpc_model.transform(test)
print(mpc_pred)
mpc_pred.show(5)

mpc_predictionAndLabels = mpc_pred.select("prediction", "label")
mpc_evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
mpc_accuracy = mpc_evaluator.evaluate(mpc_predictionAndLabels)
print("Test Accuracy = %g" % (mpc_accuracy))
```

```
DataFrame[features: vector, label: double, rawPrediction: vector, probability: vector, prediction: double]
+--------------------+-----+--------------------+--------------------+----------+
|            features|label|       rawPrediction|         probability|prediction|
+--------------------+-----+--------------------+--------------------+----------+
|[0.1382,4.9429,8....|  1.0|[-2.8394140962509...|[0.00305533310976...|       1.0|
|[0.2325,4.9466,6....|  1.0|[-0.8471929374748...|[0.16563450508847...|       1.0|
|[0.2985,5.9555,8....|  1.0|[-2.9054848565520...|[0.00268980259709...|       1.0|
|[0.4724,4.9612,8....|  1.0|[-2.0863172219983...|[0.01526287462594...|       1.0|
|[0.7828,6.2814,7....|  1.0|[1.88298584051804...|[0.98193172051776...|       0.0|
+--------------------+-----+--------------------+--------------------+----------+
only showing top 5 rows

Test Accuracy = 0.824916
```

The area under the ROC calculation, and Precision/Recall values:

```
mpc_binary_evaluator = BinaryClassificationEvaluator(labelCol = 'label')
#print('Random Forest areaUnderROC:' , binary_evaluator.evaluate(rf_predictions))
print("Test Area Under ROC: " + str(mpc_binary_evaluator.evaluate(mpc_pred, {mpc_binary_evaluator.metricName: "areaUnderROC"})))
```

```
Test Area Under ROC: 0.8073290346017611
```

```
print("Test Area Under Precision/Recall: " + str(mpc_binary_evaluator.evaluate(mpc_pred, {mpc_binary_evaluator.metricName: "areaUnderPR"})))
```

```
Test Area Under Precision/Recall: 0.770047581329278
```

The code inspired with the Spark Apache documentation available at
https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier

****

Bonus – Optional: Should you decide to try an alternative task, there will be a bonus of **5 marks.** The maximum overall mark for this assessment remains at 50/50; however, attempting the bonus exercise will a) make you practice with an alternative distributed library and b) enhance your chances of getting a higher mark overall.

Description: Using the package elephas and getting inspiration from the colab notebook that has been provided to you, create a distributed keras model - (model = sequential ()) – with:

a) three dense layers,

b) relu activation function, and

c) dropout layers after each dense layer.

The last layer will have a softmax activation. Please use categorical_cross entropy and adam as optimizer. Learning rate can be chosen by you, which can be used to finetune/optimise performance.

Report performance in terms of accuracy only (test set).

****

The bonus task.

First, we need to install the necessary libraries and restart the runtime.

```
!pip install -q elephas
!pip install --ignore-installed PyYAML
```

```
     |████████████████████████████████| 337kB 11.1MB/s
     |████████████████████████████████| 51kB 7.4MB/s
  Building wheel for elephas (setup.py) ... done
Collecting PyYAML
  Downloading https://files.pythonhosted.org/packages/7a/a5/393c087efdc7809
     |████████████████████████████████| 645kB 7.4MB/s
Installing collected packages: PyYAML
Successfully installed PyYAML-5.4.1
WARNING: The following packages were previously imported in this runtime:
  [yaml]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

Then I created a new data frame and split it into train and test sets.

```
bonus = outputFixed.select('features', 'Status')
bonus.show(3)
```

```
+--------------------+------+
|            features|Status|
+--------------------+------+
|[4.5044,0.7443,6....|Normal|
|[4.4284,0.9073,5....|Normal|
|[4.5291,1.0199,6....|Normal|
+--------------------+------+
only showing top 3 rows
```

```
train_df, test_df = bonus.randomSplit([0.7, 0.3])
```

Then we are indexing the data frame into the new scaled data frame as in the example provided.

```
string_indexer = StringIndexer(inputCol="Status", outputCol="index_status")
fitted_indexer = string_indexer.fit(train_df)
indexed_df = fitted_indexer.transform(train_df)
```

```
from pyspark.ml.feature import StandardScaler

scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
fitted_scaler = scaler.fit(indexed_df)
scaled_df = fitted_scaler.transform(indexed_df)
```

```
scaled_df.show(10)
```

```
+--------------------+------+------------+--------------------+
|            features|Status|index_status|     scaled_features|
+--------------------+------+------------+--------------------+
|[0.0851,5.8301,7....|Normal|         1.0|[-1.7612346158876...|
|[0.0861,4.8303,6....|Normal|         1.0|[-1.7608774039677...|
|[0.1382,4.9429,8....|Normal|         1.0|[-1.7422666629412...|
|[0.1447,5.5261,7....|Normal|         1.0|[-1.7399447854618...|
|[0.2325,4.9466,6....|Normal|         1.0|[-1.7085815788951...|
|[0.2719,5.1517,7....|Normal|         1.0|[-1.6945074292512...|
|[0.2811,4.52,8.04...|Normal|         1.0|[-1.6912210795882...|
|[0.2985,5.9555,8....|Normal|         1.0|[-1.6850055921820...|
|[0.3983,5.5316,5....|Normal|         1.0|[-1.6493558425765...|
|[0.4724,4.9612,8....|Normal|         1.0|[-1.6228864393122...|
+--------------------+------+------------+--------------------+
only showing top 10 rows
```

Then we are creating and compiling the model.

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers.core import Dense, Dropout, Activation
from tensorflow.python.keras.utils import np_utils, generic_utils

nb_classes = train_df.select("Status").distinct().count()
input_dim = len(train_df.select("features").first()[0])
print(nb_classes, input_dim)

model = Sequential()
model.add(Dense(512, input_shape=(input_dim,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')
```

```
2 12
```

Then we are initializing the SparkML Estimator and setting all relevant properties. I had to change the import of Keras optimizers, as it did not work for me.

```
from elephas.ml_model import ElephasEstimator
#from tensorflow.python.keras import optimizers
from tensorflow import keras

adam = keras.optimizers.Adam(lr=0.003)
opt_conf = optimizers.serialize(adam)

# Initialize SparkML Estimator and set all relevant properties
estimator = ElephasEstimator()
estimator.setFeaturesCol("scaled_features")             # These two come directly from pyspark,
estimator.setLabelCol("index_status")                 # hence the camel case. Sorry :)
estimator.set_keras_model_config(model.to_yaml())       # Provide serialized Keras model
estimator.set_categorical_labels(True)
estimator.set_nb_classes(nb_classes)
estimator.set_num_workers(1)  # We just use one worker here. Feel free to adapt it.
estimator.set_epochs(20)
estimator.set_batch_size(128)
estimator.set_verbosity(1)
estimator.set_validation_split(0.15)
estimator.set_optimizer_config(opt_conf)
estimator.set_mode("synchronous")
estimator.set_loss("categorical_crossentropy")
estimator.set_metrics(['acc'])
```

```
ElephasEstimator_45abf125b283
```

Then we are creating a pipeline to index and make use of data and making predictions on the test set.

```
[85] from pyspark.ml import Pipeline

     pipeline = Pipeline(stages=[string_indexer, scaler, estimator])
```

```
[91] from pyspark.mllib.evaluation import MulticlassMetrics

     fitted_pipeline = pipeline.fit(train_df) # Fit model to data

     prediction = fitted_pipeline.transform(test_df) # Evaluate on test
     #prediction = fitted_pipeline.transform(test_df) # <-- The same co
     pnl = prediction.select("index_status", "prediction")
     pnl.show(10)
```

```
     >>> Fit model
     >>> Synchronous training complete.
     +------------+--------------------+
     |index_status|          prediction|
     +------------+--------------------+
     |         1.0|[0.01479736622422...|
     |         1.0|[0.42840147018432...|
     |         1.0|[0.23763343691825...|
     |         1.0|[0.00111016177106...|
     |         1.0|[0.00121192040387...|
     |         1.0|[0.00418216455727...|
     |         1.0|[2.26613861741498...|
     |         1.0|[0.14957541227340...|
     |         1.0|[0.44166645407676...|
     |         1.0|[0.03204603493213...|
     +------------+--------------------+
     only showing top 10 rows
```

As we can see from the result table the predictions are in the list format, where our labels are in the categorical. I do not have enough information on the data set provided in the example, and I can only guess why I have got the list of values. I guess this is because of the nature of the categorical classification which we are using. I mean here that we are getting raw prediction for each feature we have. Unfortunately can not provide the evaluation as labels and predictions are in a different format.