Big Data Assessment One

Analysis, and Best Practices of, Big Data

1 Task One – Narrative

In this section, I will be discussing the best approaches for big data analysis in the context of Netflix. This scenario will give an overview of an approach to analysing big data from a currently established, large company, that deals with a lot of data worldwide.

In 2019, Netflix had 167 million subscribed users (Ohio University, 2020). Data is collected from each of those users, including their names, contact information, geographical location, device information, and more. All of this data has to be stored, and much of it can be used to by the company in order to either: find current trends, discover what users like, what they may like in the future and predicting watch time/sales, or create a business plan for the future.

The main four approaches to analysing big data are:

1.1 Descriptive Analysis

Descriptive analysis looks at the data that is already there and describes the data in an easier format, such as graphs, easy to read tables, etc. An example of this is shown in section 2, namely during summarisation of the data by finding the maximum, minimum, standard deviation, etc of the data set. Visualising the data helps the human analysts to find trends in the data, like noticing sudden surges in people watching a particular film/series, as well as any anomalies that may either be outliers in the data, or other unsuspected niche trends. These visualisations could take the form of monthly reports for Netflix, outlining the current user base, the most popular shows, which shows are lacking in watch time, etc. Data sources for this could also include demographical data, giving personalised insights into the types of users that could be interested in particular shows on a broad, manual scale.

Descriptive analysis, however, does tend to require a human data analysist to abstract the resulting charts and provide insight into what it means for the company as a whole. Since descriptive analysis is essentially another way of representing already existing data, that data also has to have obvious meaning to it for it to be easily understood by humans. Correlations that may be picked up upon by machine learning techniques may be missed.

1.2 Diagnostic Analysis

Diagnostic analysis takes descriptive analysis a step further by attempting to understand the root cause of the trends shown by descriptive analysis. This often takes the form of techniques such as data mining, where an, often, machine learning model finds correlations in the data. Correlations can also be more manually found using techniques such as creating a Pearson correlation matrix and identifying which features in the data correlate to others, whether positively or negatively. These correlations can then be used by Netflix to, for example, recommend shows to people who like certain categories, or recommend which kinds of shows should receive more funding to management.

Whilst diagnostic analysis gives further insight into existing data, these insights still need to be analysed by analysists in order to come to conclusive predictions. Since predictive analysis can forecast future events based on the statistics of current data with ease, it makes it a much more suitable approach in regards to a company wanting to maximise profits.

1.3 Predictive Analysis

Predictive analysis is used to gain insight into the future through data. This usually is accomplished with the use of machine learning models to classify new data after being trained on pre-existing datasets, giving newly generated outputs of simple insights based on complex data. Netflix may use predictive analysis in this way to, for example, train a model on previous revenue gained by each show, and predict whether a new show will generate money based on key factors such as production cost, genre, current trends in popularity, etc.

Machine learning models are not infallible, and often produce false predictions. This could be due to having errors in the data, anomalies that skew results, or simply a bad seed in the case of models such as k-nearest-neighbour or decision forests. Machine learning models also tend to be computationally expensive and take a long time to process. This could increase costs for Netflix, and they may have to dedicate more of their profits made from using predictive analysis in order to maintain it.

1.4 Prescriptive Analysis

Prescriptive analysis builds upon predictive analysis, specifically focusing on using forecasted trends to produce action plans for how best the company can proceed. In the context of Netflix, this could take the form of using machine learning models to predict the revenue of multiple new theoretical shows, and then selecting which of those would be the best to produce in order to maximise profits.

Since prescriptive analysis builds upon an already slow predictive analysis, it is often even slower to produce results. This may not only take more profits from Netflix in order to maintain but could also take too long and result in them losing an edge over competitors.

1.5 Summary

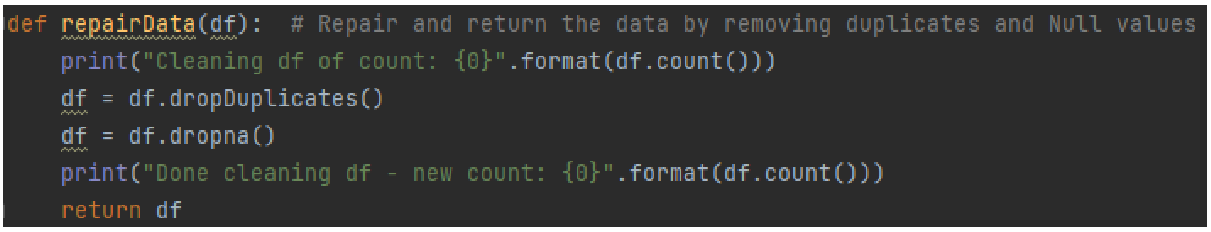
Analysis Type	**Advantages**	**Disadvantages**
Descriptive Analysis	Can provide a summary of current and past data for further analysis.	Is often difficult to make predictions off alone.
Diagnostic Analysis	Can give a deeper insight into why particular trends are forming and help to capitalise on them.	Requires further analysis, making it time consuming and technical.
Predictive Analysis	Can help forecast what data could mean, and predict future trends based on current data.	Takes a lot of cost and time to run machine learning models, especially on large data sets.
Prescriptive Analysis	Can take analysis the entire way and give insight into potential courses of action.	Even slower than predictive analysis, potentially too slow in fast paced environments.

2 Tas2 Two - Analysis

2.1 Section One – Data Summary, Understanding and Visualisation

2.1.1 Task 1

Before analysing the dataset, I first repair any values within it that could interfere with the rest of the process, such as missing values. When the dataset is loaded in, it goes through the `repairData` function which drops any duplicate entries using PySpark's `.dropDuplicates` function, as well as any empty values using the `.dropna` function. This new cleaned data frame is then returned and overwrites the original.

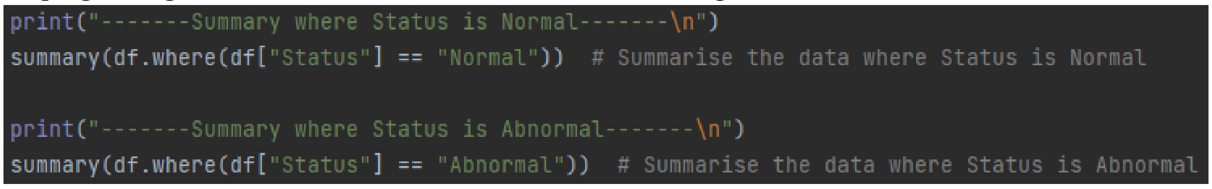


```
def repairData(df):  # Repair and return the data by removing duplicates and Null values
    print("Cleaning df of count: {0}".format(df.count()))
    df = df.dropDuplicates()
    df = df.dropna()
    print("Done cleaning df - new count: {0}".format(df.count()))
    return df
```

Missing values could also have been substituted with averaged data, for instance, however that could lead to innacuracy. Since the dataset is large enough to accommodate a few missing values, I opted to remove them instead.

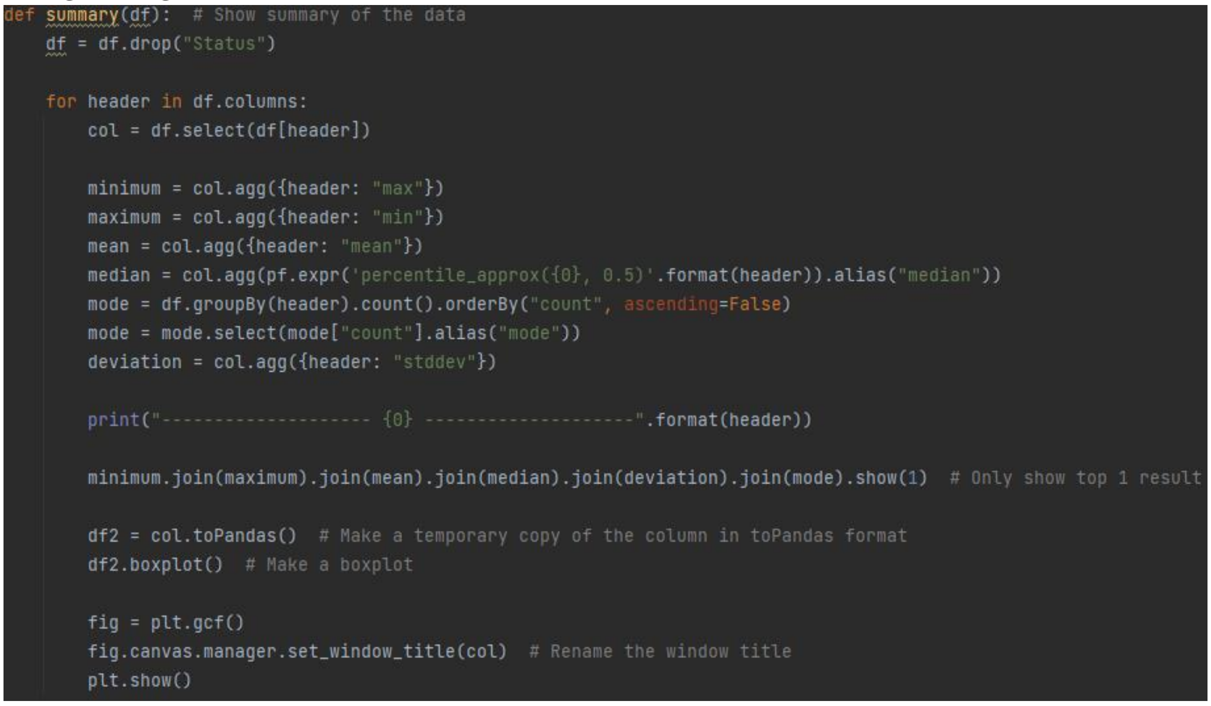
2.1.2 Task 2

The cleaned dataset is then ran through the `summary` function, which returns a summary of each feature and shows the corresponding box plot. I accomplished this by first dropping the 'Status' column in the data frame, as only one of the classes (normal or abnormal) are passed at once, before looping through each column of sensor data and summarising.



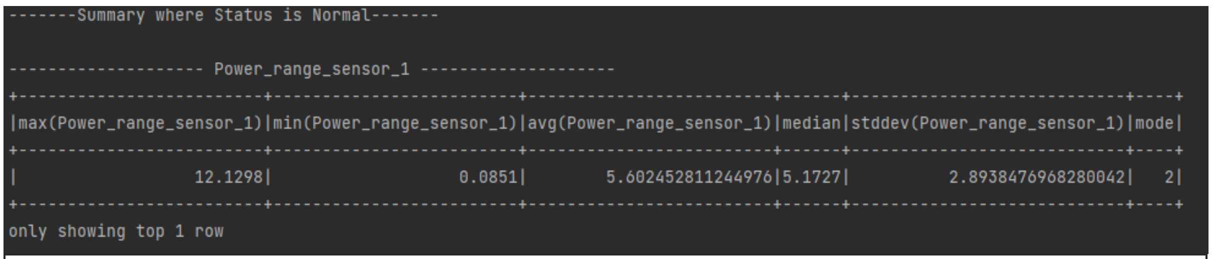
```
print("-------Summary where Status is Normal-------\n")
summary(df.where(df["Status"] == "Normal"))  # Summarise the data where Status is Normal

print("-------Summary where Status is Abnormal-------\n")
summary(df.where(df["Status"] == "Abnormal"))  # Summarise the data where Status is Abnormal
```

The summary is calculated for each category using PySpark aggregate functions to apply statistics functions to the entire column. Median and mode were calculated differently; median made use of the `pyspark.sql.functions` package to find the approximate percentiles of the column, and then the middle percentile was selected. Mode was calculated by grouping, and then ordering, by the count and taking the largest value.

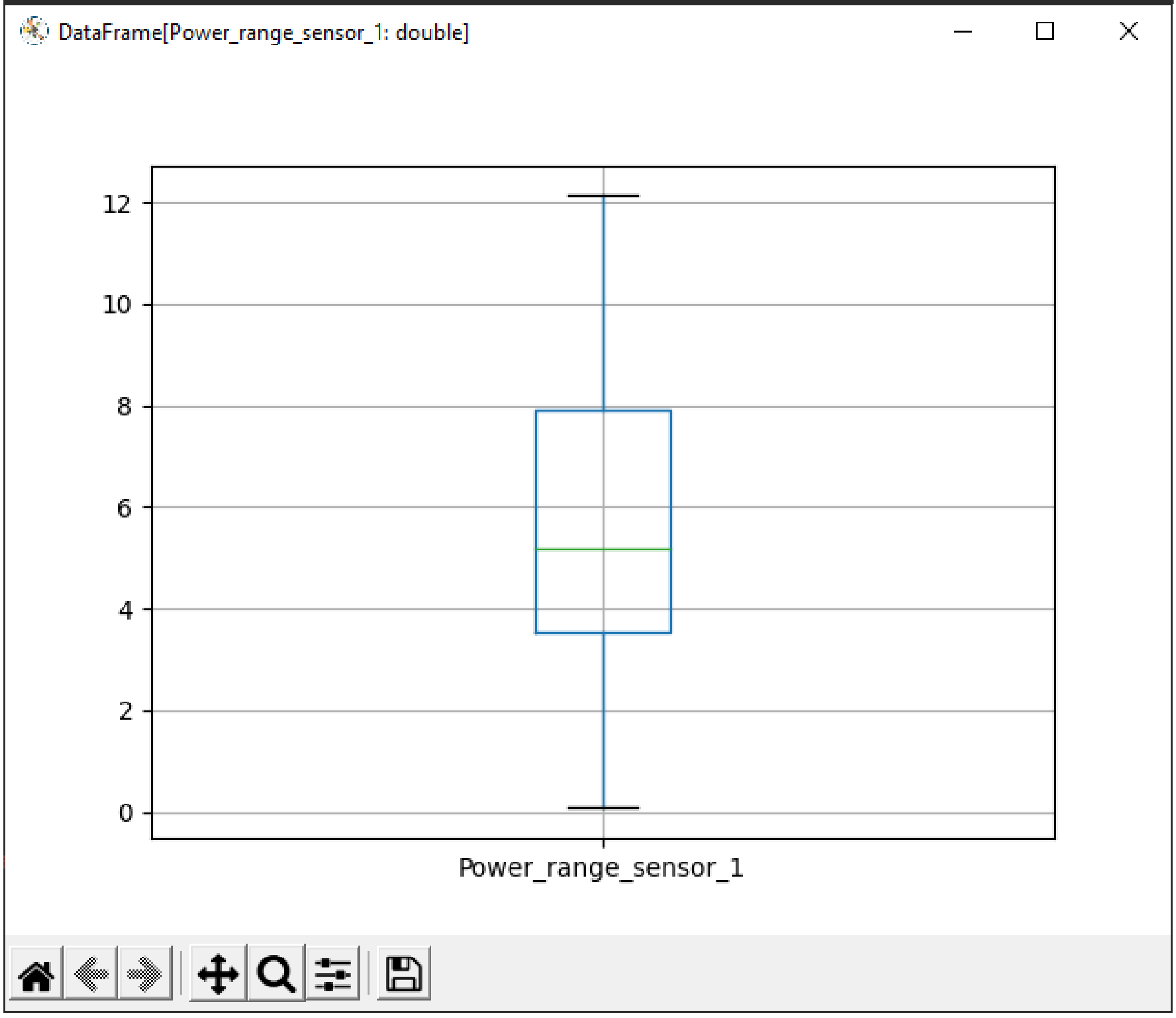


```
def summary(df):  # Show summary of the data
    df = df.drop("Status")

    for header in df.columns:
        col = df.select(df[header])

        minimum = col.agg({header: "max"})
        maximum = col.agg({header: "min"})
        mean = col.agg({header: "mean"})
        median = col.agg(pf.expr('percentile_approx({0}, 0.5)'.format(header)).alias("median"))
        mode = df.groupBy(header).count().orderBy("count", ascending=False)
        mode = mode.select(mode["count"].alias("mode"))
        deviation = col.agg({header: "stddev"})

        print("-------------------- {0} --------------------".format(header))

        minimum.join(maximum).join(mean).join(median).join(deviation).join(mode).show(1)  # Only show top 1 result

        df2 = col.toPandas()  # Make a temporary copy of the column in toPandas format
        df2.boxplot()  # Make a boxplot

        fig = plt.gcf()
        fig.canvas.manager.set_window_title(col)  # Rename the window title
        plt.show()
```

The summary is shown as a table, displaying the summary values for each feature individually, and a box plot for that feature. The box plots are drawn using Matplotlib after converting each summary dataframe table to a Pandas table.



```
-------Summary where Status is Normal-------

------------------- Power_range_sensor_1 -------------------
+-------------------------+-------------------------+-------------------------+------+----------------------------+----+
|max(Power_range_sensor_1)|min(Power_range_sensor_1)|avg(Power_range_sensor_1)|median|stddev(Power_range_sensor_1)|mode|
+-------------------------+-------------------------+-------------------------+------+----------------------------+----+
|                  12.1298|                   0.0851|        5.602452811244976|5.1727|          2.8938476968280042|   2|
+-------------------------+-------------------------+-------------------------+------+----------------------------+----+
only showing top 1 row
```

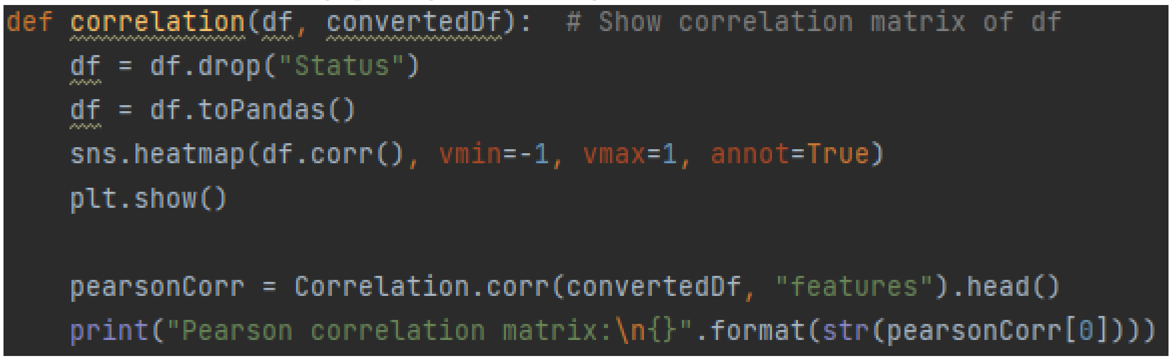


Whilst this table is legible, it could be improved to make it easier for businesses wanting to gain data insight by renaming all of the columns to more appropriate names. This also applies to the title of each box plot window. It is also worth noting that each column does not have a specific degree of accuracy, and hence some results are to a few decimal places, whereas others are far more.

This summary information alone is likely to point to clues of unusual or inconsistent data, and could give insight into the accuracy of predictive models outputting whether readings are normal or abrnormal. Unusually high or low readings from the set of summary statistics are a good sign of abnormal behaviour which can be easily identified.

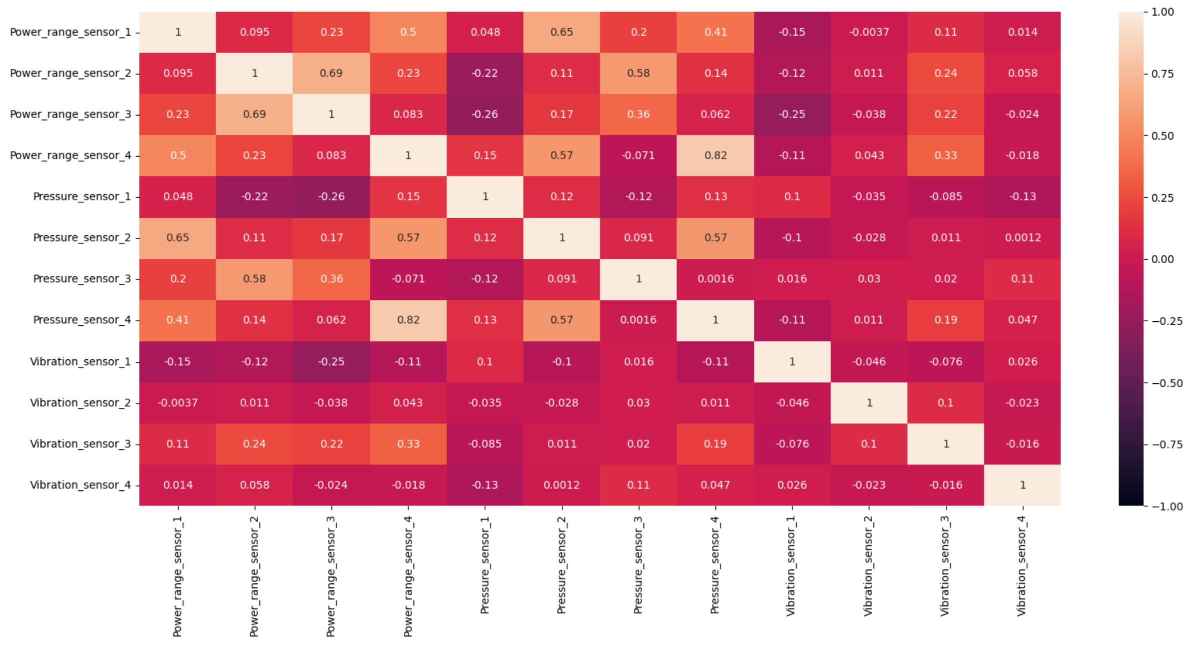
2.1.3 Task 3

Before the correlation matrix is created, I run the data frame through a PySpark `vertexIndexer` in order to use the `pyspark.ml.stat.Correlation` function. I cover the `vertexIndexer` function in more detail in 2.2.1 Task 4 during splitting and shuffling of the data.



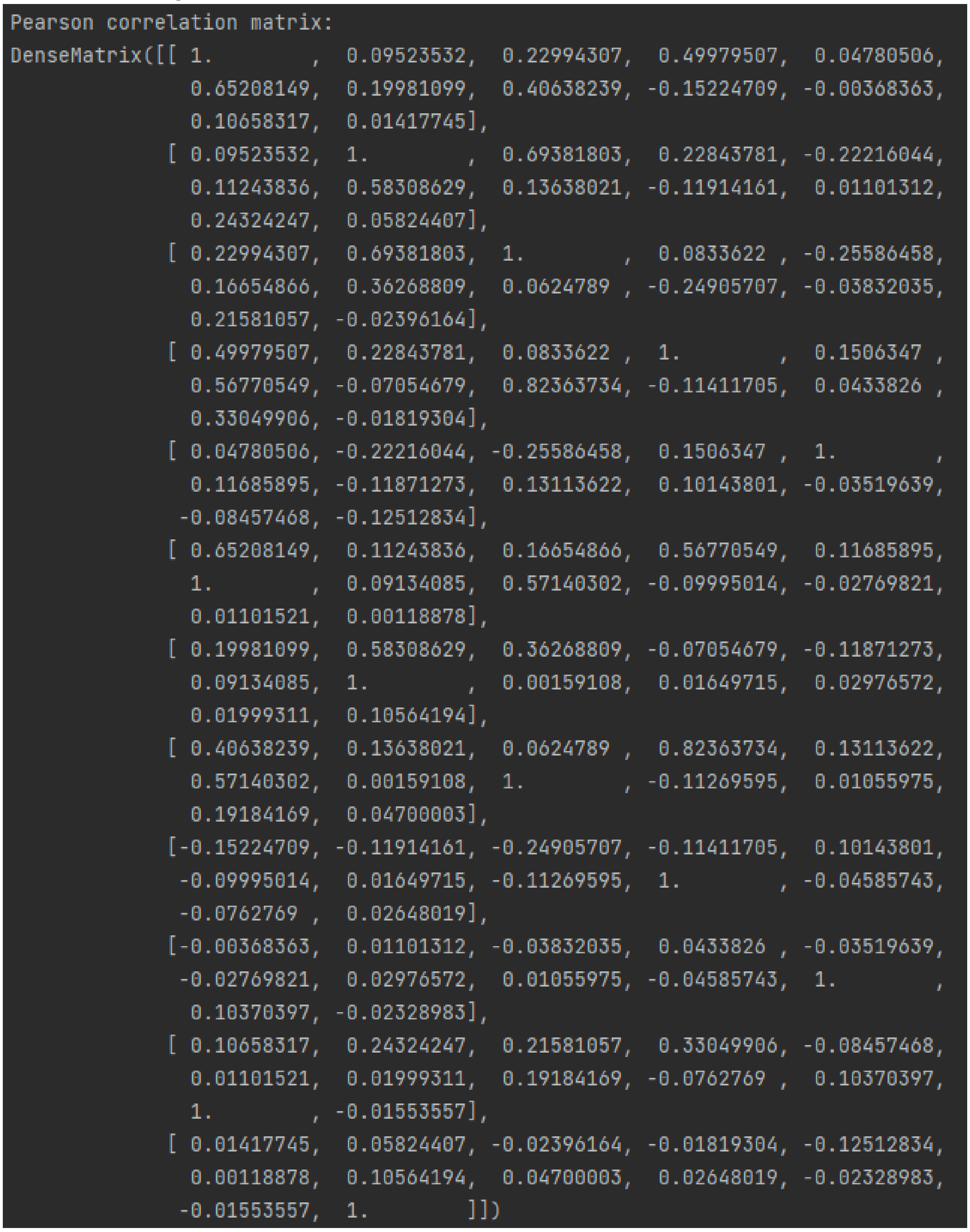
```
def correlation(df, convertedDf):  # Show correlation matrix of df
    df = df.drop("Status")
    df = df.toPandas()
    sns.heatmap(df.corr(), vmin=-1, vmax=1, annot=True)
    plt.show()

    pearsonCorr = Correlation.corr(convertedDf, "features").head()
    print("Pearson correlation matrix:\n{}".format(str(pearsonCorr[0])))
```

Since reading the correlation matrix straight from a PySpark table can often be difficult, especially when the output wraps within the terminal, I also included a heatmap version using `Seaborn`. This heatmap requires use of the original dataframe, not the `convertedDf` that was vectorised, and hence the correlation function takes two inputs. The original dataframe is converted to a Pandas dataframe and plotted.



	Power_range_sensor_1	Power_range_sensor_2	Power_range_sensor_3	Power_range_sensor_4	Pressure_sensor_1	Pressure_sensor_2	Pressure_sensor_3	Pressure_sensor_4	Vibration_sensor_1	Vibration_sensor_2	Vibration_sensor_3	Vibration_sensor_4
Power_range_sensor_1	1	0.095	0.23	0.5	0.048	0.65	0.2	0.41	-0.15	-0.0037	0.11	0.014
Power_range_sensor_2	0.095	1	0.69	0.23	-0.22	0.11	0.58	0.14	-0.12	0.011	0.24	0.058
Power_range_sensor_3	0.23	0.69	1	0.083	-0.26	0.17	0.36	0.062	-0.25	-0.038	0.22	-0.024
Power_range_sensor_4	0.5	0.23	0.083	1	0.15	0.57	-0.071	0.82	-0.11	0.043	0.33	-0.018
Pressure_sensor_1	0.048	-0.22	-0.26	0.15	1	0.12	-0.12	0.13	0.1	-0.035	-0.085	-0.13
Pressure_sensor_2	0.65	0.11	0.17	0.57	0.12	1	0.091	0.57	-0.1	-0.028	0.011	0.0012
Pressure_sensor_3	0.2	0.58	0.36	-0.071	-0.12	0.091	1	0.0016	0.016	0.03	0.02	0.11
Pressure_sensor_4	0.41	0.14	0.062	0.82	0.13	0.57	0.0016	1	-0.11	0.011	0.19	0.047
Vibration_sensor_1	-0.15	-0.12	-0.25	-0.11	0.1	-0.1	0.016	-0.11	1	-0.046	-0.076	0.026
Vibration_sensor_2	-0.0037	0.011	-0.038	0.043	-0.035	-0.028	0.03	0.011	-0.046	1	0.1	-0.023
Vibration_sensor_3	0.11	0.24	0.22	0.33	-0.085	0.011	0.02	0.19	-0.076	0.1	1	-0.016
Vibration_sensor_4	0.014	0.058	-0.024	-0.018	-0.13	0.0012	0.11	0.047	0.026	-0.023	-0.016	1

As can be seen by this heatmap, many of the sensors within this data are uncorrelated, or potentially correlated by chance such as ‘Pressure_sensor_4’ and ‘Power_range_sensor_4’. This correlation could be due to pressure or power being linked in some manner, or by random chance. It is also worth noting that ‘Pressure_sensor_3’ and ‘Power_range_sensor_2’ are highly correlated, and do not share their numeric designation.



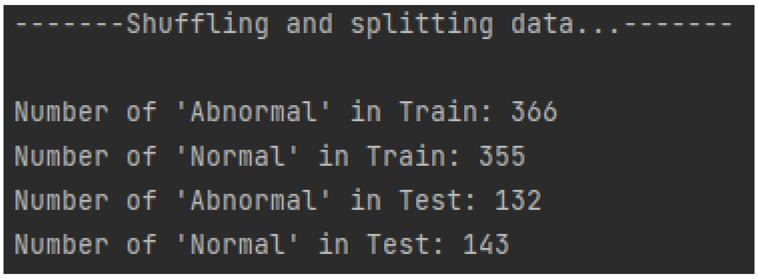
```
Pearson correlation matrix:
DenseMatrix([[ 1.        ,  0.09523532,  0.22994307,  0.49979507,  0.04780506,
               0.65208149,  0.19981099,  0.40638239, -0.15224709, -0.00368363,
               0.10658317,  0.01417745],
             [ 0.09523532,  1.        ,  0.69381803,  0.22843781, -0.22216044,
               0.11243836,  0.58308629,  0.13638021, -0.11914161,  0.01101312,
               0.24324247,  0.05824407],
             [ 0.22994307,  0.69381803,  1.        ,  0.0833622 , -0.25586458,
               0.16654866,  0.36268809,  0.0624789 , -0.24905707, -0.03832035,
               0.21581057, -0.02396164],
             [ 0.49979507,  0.22843781,  0.0833622 ,  1.        ,  0.1506347 ,
               0.56770549, -0.07054679,  0.82363734, -0.11411705,  0.0433826 ,
               0.33049906, -0.01819304],
             [ 0.04780506, -0.22216044, -0.25586458,  0.1506347 ,  1.        ,
               0.11685895, -0.11871273,  0.13113622,  0.10143801, -0.03519639,
              -0.08457468, -0.12512834],
             [ 0.65208149,  0.11243836,  0.16654866,  0.56770549,  0.11685895,
               1.        ,  0.09134085,  0.57140302, -0.09995014, -0.02769821,
               0.01101521,  0.00118878],
             [ 0.19981099,  0.58308629,  0.36268809, -0.07054679, -0.11871273,
               0.09134085,  1.        ,  0.00159108,  0.01649715,  0.02976572,
               0.01999311,  0.10564194],
             [ 0.40638239,  0.13638021,  0.0624789 ,  0.82363734,  0.13113622,
               0.57140302,  0.00159108,  1.        , -0.11269595,  0.01055975,
               0.19184169,  0.04700003],
             [-0.15224709, -0.11914161, -0.24905707, -0.11411705,  0.10143801,
              -0.09995014,  0.01649715, -0.11269595,  1.        , -0.04585743,
              -0.0762769 ,  0.02648019],
             [-0.00368363,  0.01101312, -0.03832035,  0.0433826 , -0.03519639,
              -0.02769821,  0.02976572,  0.01055975, -0.04585743,  1.        ,
               0.10370397, -0.02328983],
             [ 0.10658317,  0.24324247,  0.21581057,  0.33049906, -0.08457468,
               0.01101521,  0.01999311,  0.19184169, -0.0762769 ,  0.10370397,
               1.        , -0.01553557],
             [ 0.01417745,  0.05824407, -0.02396164, -0.01819304, -0.12512834,
               0.00118878,  0.10564194,  0.04700003,  0.02648019, -0.02328983,
              -0.01553557,  1.        ]])
```

The printed table for the pearson correlation matrix is much harder to read, and makes gaining insight difficult. Whilst this could be further process to automatically pertain correlation and potentially spot anomolies, I find the heat map to be a much more usable approach.

2.2 Section Two – Classification and Big Data Analysis

2.2.1 Task 4

When using classification on the dataframe, we first have the train the models. This requires testing them on existing data so that we can know the accuracy of the outputs. To do this, the dataset is split in a 70:30 ratio, with 70% being for training and 30% being for testing. In my implementation, the dataframe loaded is run through the PySpark `vectorAssembler` and `stringIndexer` in order to make it compatible with `pyspark.ml.classification` models first.

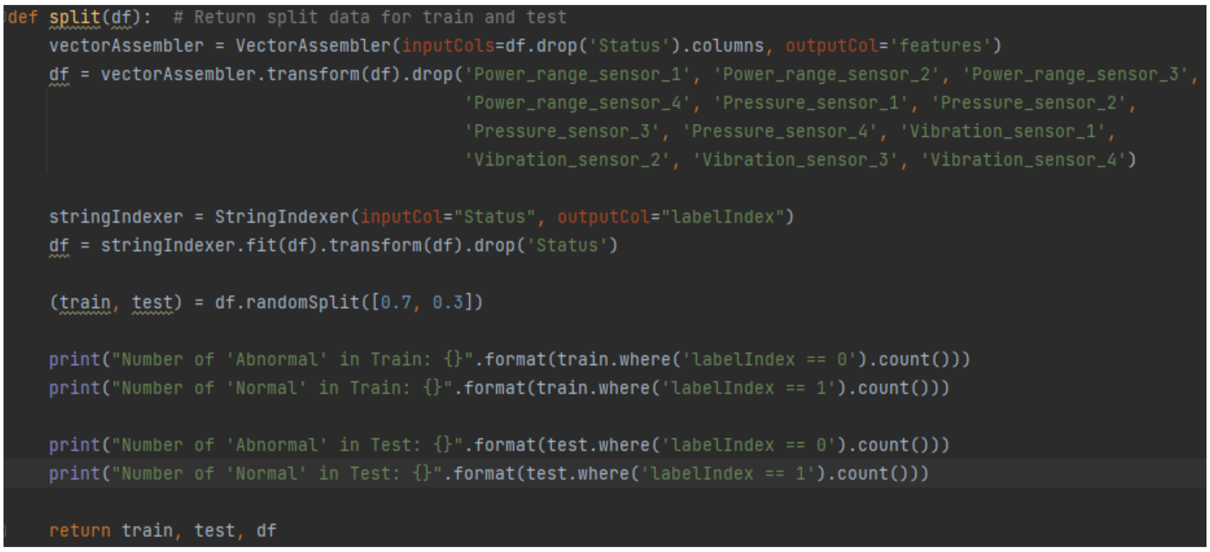


```
-------Shuffling and splitting data...-------

Number of 'Abnormal' in Train: 366
Number of 'Normal' in Train: 355
Number of 'Abnormal' in Test: 132
Number of 'Normal' in Test: 143
```

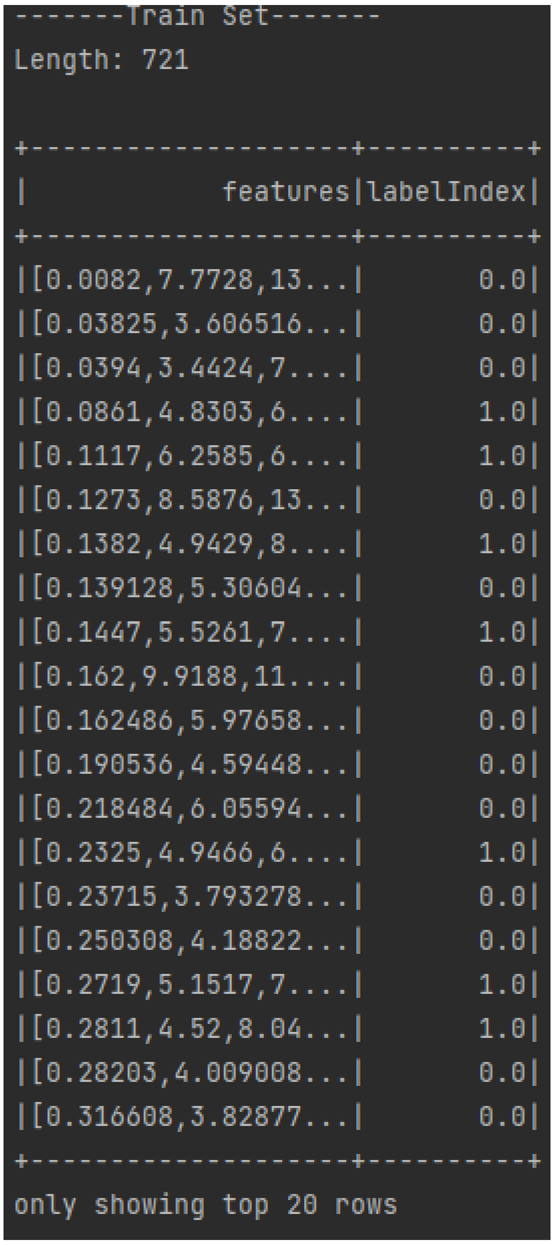
PySpark has a `vectorAssembler` function that takes multiple columns as inputs and outputs a single ‘features’ column to be used for training purposes. The features column contains a vector containing each of the 12 sensor readings, which can then be used by the machine learning models in PySpark.

The features column also needs labelling with classes to be able to classify the data. I used the PySpark `stringIndexer` to accomplish this; the indexer creates a new column called ‘labelIndex’ that takes the string inputs of ‘Status’ (‘Normal’ and ‘Abnormal’) and converts them into float values (1.0 and 0.0).

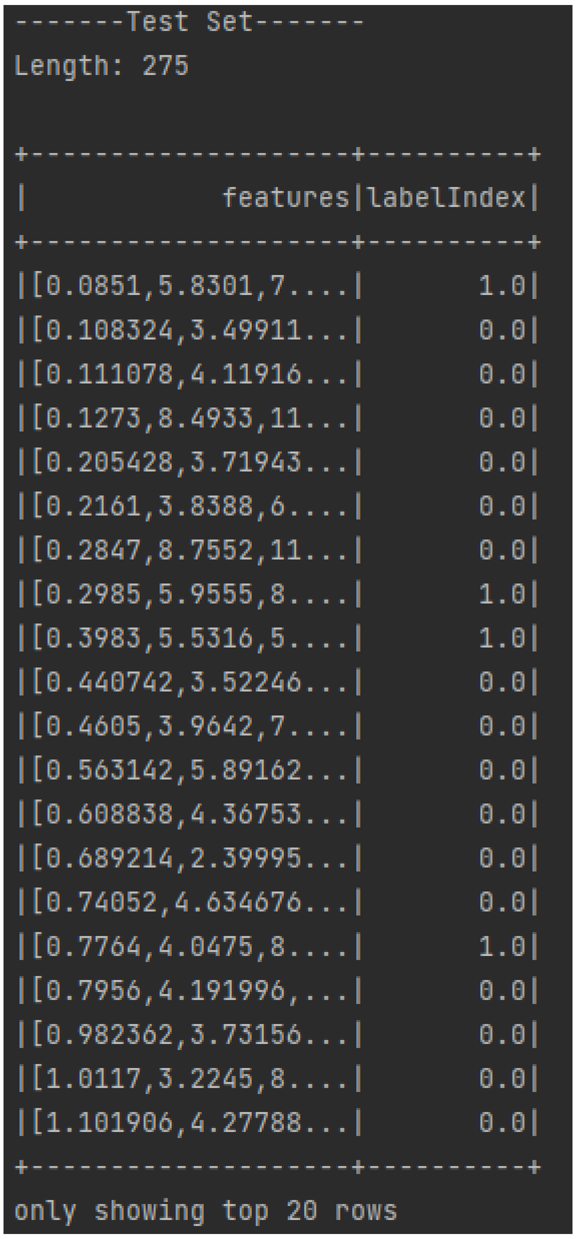
These values result in a table consisting of a vector column ‘features’ and a float columns ‘labelIndex’ which can be split using PySpark’s `randomSplit` function and fed to the machine learning classifiers. Since the correlation matrix required a full dataset in this format, I also returnd the full converted dataframe from this function too.



```
def split(df):  # Return split data for train and test
    vectorAssembler = VectorAssembler(inputCols=df.drop('Status').columns, outputCol='features')
    df = vectorAssembler.transform(df).drop('Power_range_sensor_1', 'Power_range_sensor_2', 'Power_range_sensor_3',
                                            'Power_range_sensor_4', 'Pressure_sensor_1', 'Pressure_sensor_2',
                                            'Pressure_sensor_3', 'Pressure_sensor_4', 'Vibration_sensor_1',
                                            'Vibration_sensor_2', 'Vibration_sensor_3', 'Vibration_sensor_4')

    stringIndexer = StringIndexer(inputCol="Status", outputCol="labelIndex")
    df = stringIndexer.fit(df).transform(df).drop('Status')

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

    print("Number of 'Abnormal' in Train: {}".format(train.where('labelIndex == 0').count()))
    print("Number of 'Normal' in Train: {}".format(train.where('labelIndex == 1').count()))

    print("Number of 'Abnormal' in Test: {}".format(test.where('labelIndex == 0').count()))
    print("Number of 'Normal' in Test: {}".format(test.where('labelIndex == 1').count()))

    return train, test, df
```



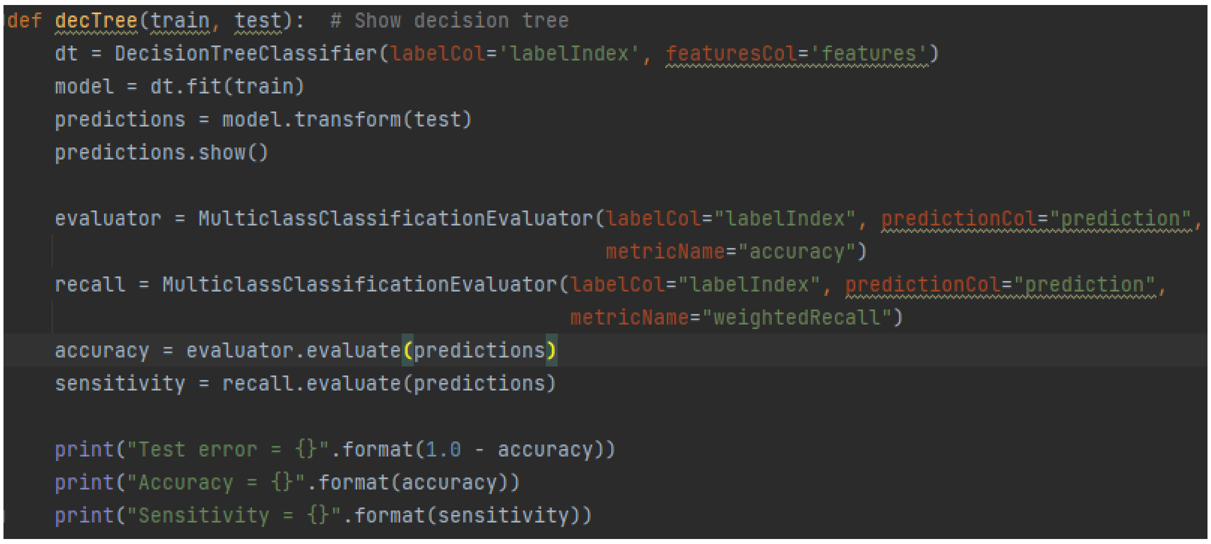
```
-------Train Set-------
Length: 721

+--------------------+----------+
|            features|labelIndex|
+--------------------+----------+
|[0.0082,7.7728,13...|       0.0|
|[0.03825,3.606516...|       0.0|
|[0.0394,3.4424,7....|       0.0|
|[0.0861,4.8303,6....|       1.0|
|[0.1117,6.2585,6....|       1.0|
|[0.1273,8.5876,13...|       0.0|
|[0.1382,4.9429,8....|       1.0|
|[0.139128,5.30604...|       0.0|
|[0.1447,5.5261,7....|       1.0|
|[0.162,9.9188,11....|       0.0|
|[0.162486,5.97658...|       0.0|
|[0.190536,4.59448...|       0.0|
|[0.218484,6.05594...|       0.0|
|[0.2325,4.9466,6....|       1.0|
|[0.23715,3.793278...|       0.0|
|[0.250308,4.18822...|       0.0|
|[0.2719,5.1517,7....|       1.0|
|[0.2811,4.52,8.04...|       1.0|
|[0.28203,4.009008...|       0.0|
|[0.316608,3.82877...|       0.0|
+--------------------+----------+
only showing top 20 rows
```



```
-------Test Set-------
Length: 275

+--------------------+----------+
|            features|labelIndex|
+--------------------+----------+
|[0.0851,5.8301,7....|       1.0|
|[0.108324,3.49911...|       0.0|
|[0.111078,4.11916...|       0.0|
|[0.1273,8.4933,11...|       0.0|
|[0.205428,3.71943...|       0.0|
|[0.2161,3.8388,6....|       0.0|
|[0.2847,8.7552,11...|       0.0|
|[0.2985,5.9555,8....|       1.0|
|[0.3983,5.5316,5....|       1.0|
|[0.440742,3.52246...|       0.0|
|[0.4605,3.9642,7....|       0.0|
|[0.563142,5.89162...|       0.0|
|[0.608838,4.36753...|       0.0|
|[0.689214,2.39995...|       0.0|
|[0.74052,4.634676...|       0.0|
|[0.7764,4.0475,8....|       1.0|
|[0.7956,4.191996,...|       0.0|
|[0.982362,3.73156...|       0.0|
|[1.0117,3.2245,8....|       0.0|
|[1.101906,4.27788...|       0.0|
+--------------------+----------+
only showing top 20 rows
```

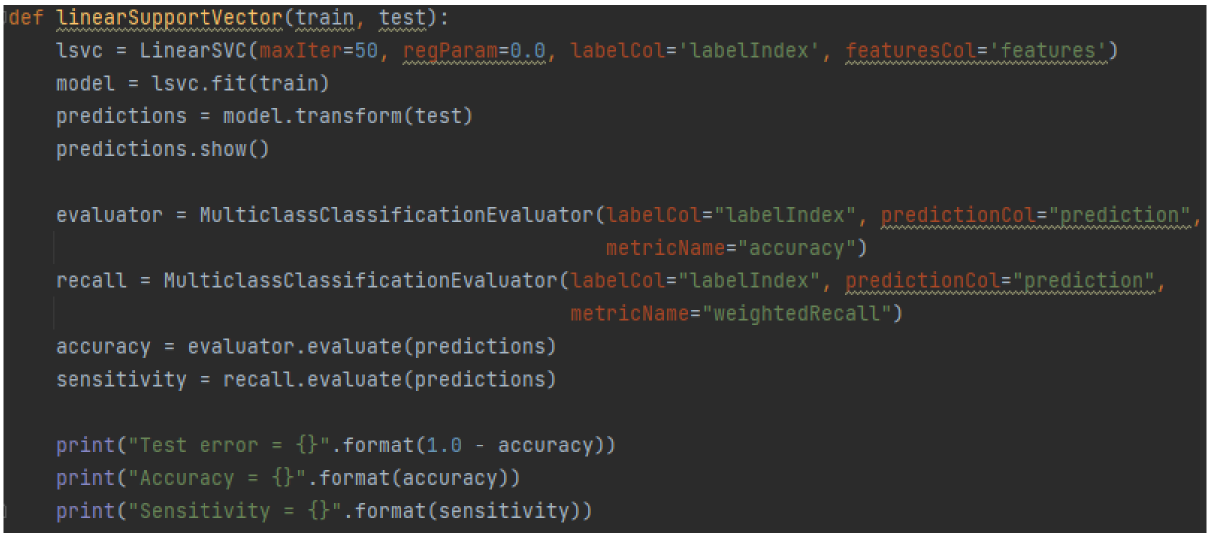
2.2.2 Task 5

Each of the three machine learning algorithms were implemented using `pyspark.ml`, and are implemented in very similar ways.



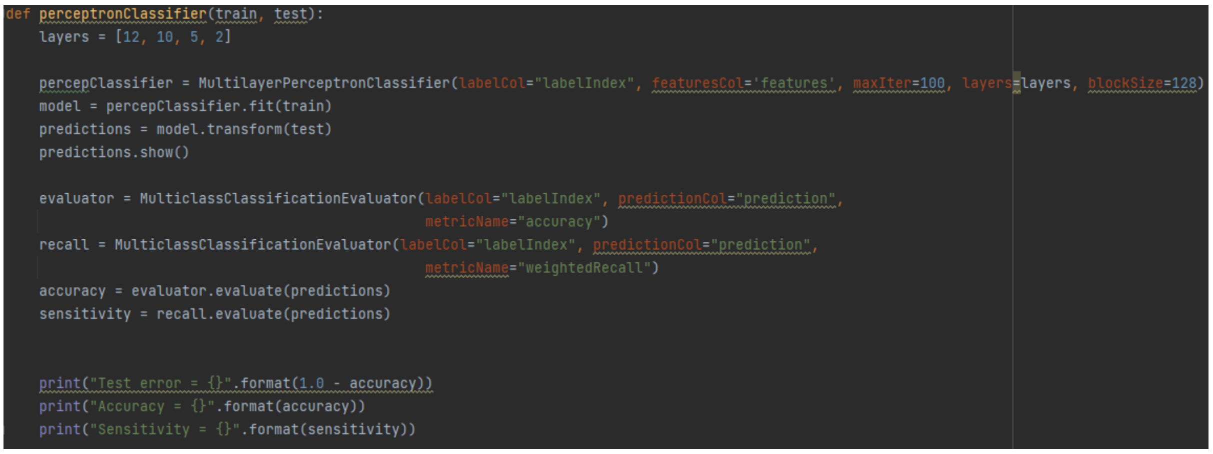
```python
def decTree(train, test):  # Show decision tree
    dt = DecisionTreeClassifier(labelCol='labelIndex', featuresCol='features')
    model = dt.fit(train)
    predictions = model.transform(test)
    predictions.show()

    evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                                  metricName="accuracy")
    recall = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                               metricName="weightedRecall")
    accuracy = evaluator.evaluate(predictions)
    sensitivity = recall.evaluate(predictions)

    print("Test error = {}".format(1.0 - accuracy))
    print("Accuracy = {}".format(accuracy))
    print("Sensitivity = {}".format(sensitivity))
```

The classifier is first initialised and passed the name of the columns headers that will be used for classification. They are also passed any parameters specific to the model, such as max iterations for the linear support vector machine. The model is then fit to the 'train' dataframe portion, and predictions are generated with the newly trained model on the 'test' portion of the dataframe.



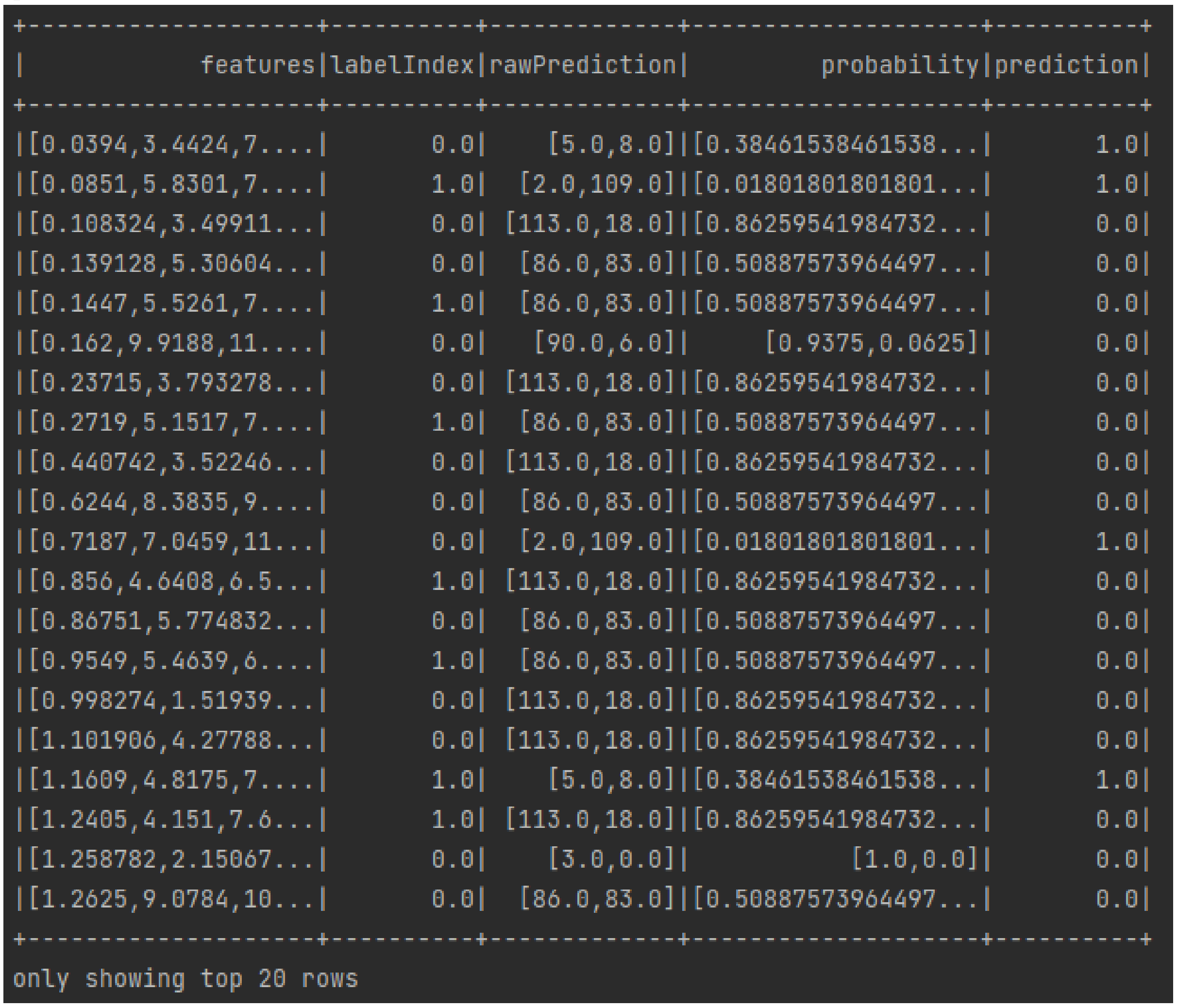
```python
def linearSupportVector(train, test):
    lsvc = LinearSVC(maxIter=50, regParam=0.0, labelCol='labelIndex', featuresCol='features')
    model = lsvc.fit(train)
    predictions = model.transform(test)
    predictions.show()

    evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                                  metricName="accuracy")
    recall = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                               metricName="weightedRecall")
    accuracy = evaluator.evaluate(predictions)
    sensitivity = recall.evaluate(predictions)

    print("Test error = {}".format(1.0 - accuracy))
    print("Accuracy = {}".format(accuracy))
    print("Sensitivity = {}".format(sensitivity))
```

The exception to this is the multilayer percepton classifier (the artificial neural network) which also takes the amount of, and the number of perceptron's in, the layers as an extra paramater.



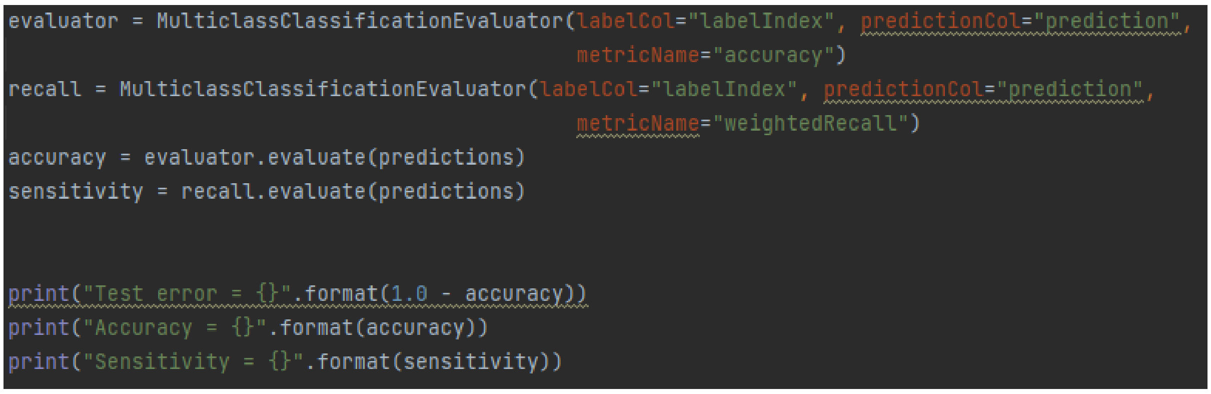
```
def perceptronClassifier(train, test):
    layers = [12, 10, 5, 2]

    percepClassifier = MultilayerPerceptronClassifier(labelCol="labelIndex", featuresCol='features', maxIter=100, layers=layers, blockSize=128)
    model = percepClassifier.fit(train)
    predictions = model.transform(test)
    predictions.show()

    evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                                  metricName="accuracy")
    recall = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                               metricName="weightedRecall")
    accuracy = evaluator.evaluate(predictions)
    sensitivity = recall.evaluate(predictions)


    print("Test error = {}".format(1.0 - accuracy))
    print("Accuracy = {}".format(accuracy))
    print("Sensitivity = {}".format(sensitivity))
```

Once the results have been found, the results table is printed out. The predictions table shows the features in a column, along with the true ‘labelIndex’ that defines what the correct classification should be. The ‘rawPrediction’ column shows a vector with the raw values that were returned by the model for the probability of the features being either one class or the other. This is then converted to a normalised probability in the ‘probability’ column, with the final prediction as to which clas the features belong in being in
‘prediction’.



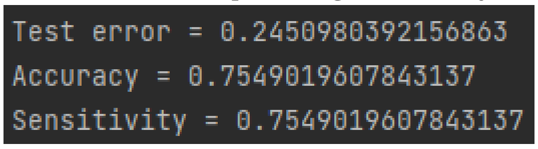
```
+--------------------+----------+-------------+--------------------+----------+
|            features|labelIndex|rawPrediction|         probability|prediction|
+--------------------+----------+-------------+--------------------+----------+
|[0.0394,3.4424,7....|       0.0|    [5.0,8.0]|[0.38461538461538...|       1.0|
|[0.0851,5.8301,7....|       1.0|  [2.0,109.0]|[0.01801801801801...|       1.0|
|[0.108324,3.49911...|       0.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[0.139128,5.30604...|       0.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.1447,5.5261,7....|       1.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.162,9.9188,11....|       0.0|   [90.0,6.0]|     [0.9375,0.0625]|       0.0|
|[0.23715,3.793278...|       0.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[0.2719,5.1517,7....|       1.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.440742,3.52246...|       0.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[0.6244,8.3835,9....|       0.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.7187,7.0459,11...|       0.0|  [2.0,109.0]|[0.01801801801801...|       1.0|
|[0.856,4.6408,6.5...|       1.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[0.86751,5.774832...|       0.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.9549,5.4639,6....|       1.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
|[0.998274,1.51939...|       0.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[1.101906,4.27788...|       0.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[1.1609,4.8175,7....|       1.0|    [5.0,8.0]|[0.38461538461538...|       1.0|
|[1.2405,4.151,7.6...|       1.0| [113.0,18.0]|[0.86259541984732...|       0.0|
|[1.258782,2.15067...|       0.0|    [3.0,0.0]|           [1.0,0.0]|       0.0|
|[1.2625,9.0784,10...|       0.0|  [86.0,83.0]|[0.50887573964497...|       0.0|
+--------------------+----------+-------------+--------------------+----------+
only showing top 20 rows
```

As can be seen from the example above, the models are not 100% accurate. The predictions can be seen compared next to the ground truth values by comparing ‘prediction’ with ‘labelIndex’. This could be useful when classifying new data input, as you can clearly see the model's predictions and compare them against the features to spot obvious anomalies. For instance, the decision tree has one row with a probability of exactly '[1.0, 0.0]'. Whilst this may not be wrong, it is unlike any other results.



```
evaluator = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                              metricName="accuracy")
recall = MulticlassClassificationEvaluator(labelCol="labelIndex", predictionCol="prediction",
                                           metricName="weightedRecall")
accuracy = evaluator.evaluate(predictions)
sensitivity = recall.evaluate(predictions)


print("Test error = {}".format(1.0 - accuracy))
print("Accuracy = {}".format(accuracy))
print("Sensitivity = {}".format(sensitivity))
```

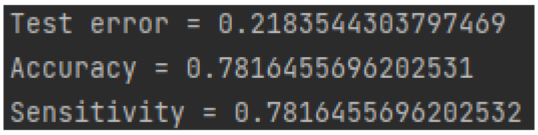
Each model's accuracy and error rate are then calculated using PySpark's inbuilt `MulticlassClassificationEvaluator`. This function calculates the accuracy of the model as a float value, and an error percentage is made by taking the accuracy from one.



```
Test error = 0.2450980392156863
Accuracy = 0.7549019607843137
Sensitivity = 0.7549019607843137
```

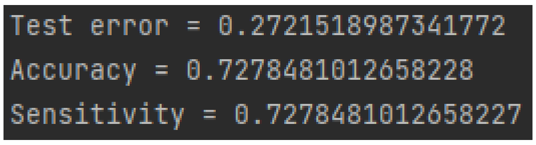
Each model can be compared with the others by comparing their error, accuracy, and sensitvity.

2.2.2.1 ---Decision Tree---



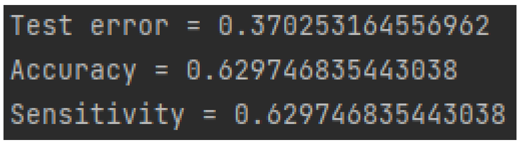
```
Test error = 0.2183544303797469
Accuracy = 0.7816455696202531
Sensitivity = 0.7816455696202532
```

2.2.2.2 ---Linear Support Vector Model---



```
Test error = 0.2721518987341772
Accuracy = 0.7278481012658228
Sensitivity = 0.7278481012658227
```

2.2.2.3 ---Multilayer Perceptron Neural Network---



```
Test error = 0.370253164556962
Accuracy = 0.629746835443038
Sensitivity = 0.629746835443038
```

As can be seen, each of the machine learning models performed similarly. The error was lowest when using the decision tree model, and highest when using the neural network. Similarly, the decision tree model was the most accurate with the neural network being the least. Each sensitivity value was close to the overall accuracy, giving a good indication that false positives could be more prone, but less so than false negatives, a good outcome for predicting potentially dangerous abnormalities in a nuclear power plant.

2.2.3 Task 6

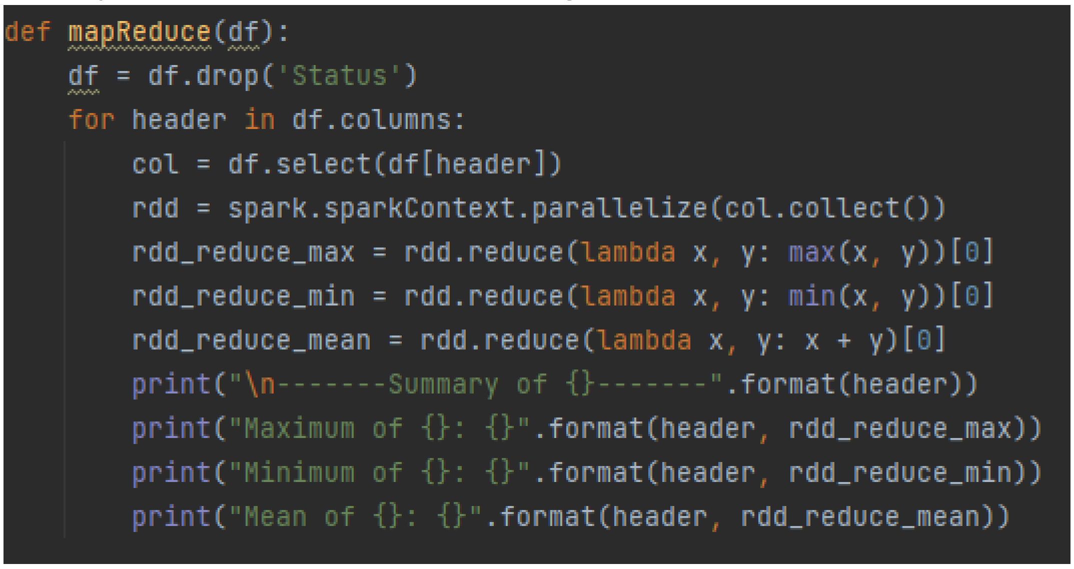
Based on the results for each of the three classifiers in 2.2.2 Task 5, I would suggest that the decision tree classification model would be best suited for classifying normality for readings. The decision tree has consistently higher accuracy than the other models, meaning the predictions are more consistently correct; being correct is vitally important when abnormal readings could result in the loss of power or dangerous effects within the power station. This also applies to the higher sensitivity value, meaning this classification model would be less likely to give false positives, and thus would often mis-identify normal readings but less so abnormal. It is most likely better for a reading to have to be checked by a human operator more frequently than it is for abnormality to continue.

2.2.4 Task 7

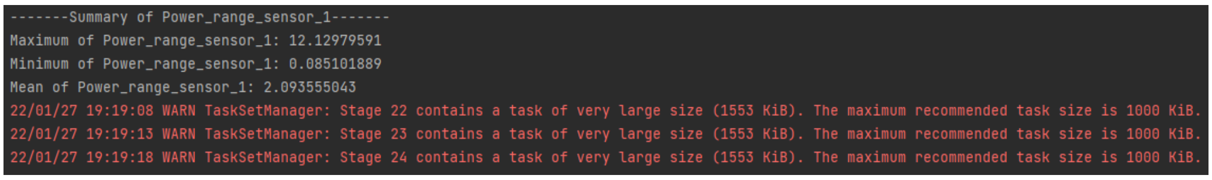
Since the accuracy of the decision tree classifier is ~78%, I believe that these features alone are enough to detect abnormalities in the reactors. Whilst the error rate is still quite high at ~22%, the majority of the time the model classified a correct outcome based on the twelve sensors alone. Since none of the features we're very clearly correlated, it is unlikely that efficiency of classification could be improved by removing some sensors. Fine tuning some of the classifiers, notably the linear support vector machine and the artificial neural network may yield better results, however since the accuracy of all three were so similar, I believe the benefit would be minor.

2.2.5 Task 8

I also implemented a basic version of the summary function using PySpark's inbuilt map reduce functionality for use with the bigger dataset. This function also uses a for loop to isolate each column, apply the map reduce functions to them, and pass onto the next. Each of the columns outputs a small summary list, not in the form of a table, showing the maxmimum, minimum, and mean values.



```python
def mapReduce(df):
    df = df.drop('Status')
    for header in df.columns:
        col = df.select(df[header])
        rdd = spark.sparkContext.parallelize(col.collect())
        rdd_reduce_max = rdd.reduce(lambda x, y: max(x, y))[0]
        rdd_reduce_min = rdd.reduce(lambda x, y: min(x, y))[0]
        rdd_reduce_mean = rdd.reduce(lambda x, y: x + y)[0]
        print("\n-------Summary of {}-------".format(header))
        print("Maximum of {}: {}".format(header, rdd_reduce_max))
        print("Minimum of {}: {}".format(header, rdd_reduce_min))
        print("Mean of {}: {}".format(header, rdd_reduce_mean))
```



```
-------Summary of Power_range_sensor_1-------
Maximum of Power_range_sensor_1: 12.12979591
Minimum of Power_range_sensor_1: 0.085101889
Mean of Power_range_sensor_1: 2.093555043
22/01/27 19:19:08 WARN TaskSetManager: Stage 22 contains a task of very large size (1553 KiB). The maximum recommended task size is 1000 KiB.
22/01/27 19:19:13 WARN TaskSetManager: Stage 23 contains a task of very large size (1553 KiB). The maximum recommended task size is 1000 KiB.
22/01/27 19:19:18 WARN TaskSetManager: Stage 24 contains a task of very large size (1553 KiB). The maximum recommended task size is 1000 KiB.
```

Whilst the maximum and minimum values are correct, the shown mean values are incorrect. I was unable to calucate the mean values correctly, as I was unsure how to implement the PySpark inbuilt `mean` function from `pyspark.sql.functions`, and summing the column and dividing by the count gave incorrect values as well.

As can be seen in the above screenshot, PySpark is fed very large amounts of data. Despite running through map reduce, this procedure still takes a lot of time, and could likely be sped up by implementing a python pooling system to enable multiprocessing, or by implementing a Hadoop system instead of PySpark.

3 References

Ohio University (2020) *How Netflix uses Data to Pick Movies and Curate Content.* Available from https://onlinemasters.ohio.edu/blog/netflix-data/ [accessed 27 January 2022]