CMP3751M – Machine Learning

Assignment Item 2: Classification of Pressurised Water Reactor Status

Alex Howe – 15618835

Contents

Section 1: Data Import, Summary, Pre-processing and Visualisation 3
Section 2: Discussion on Selecting an Algorithm 6
Section 3: Designing Algorithms 7
Artificial Neural Networks 7
Random Forests 8
Section 4: Model Selection 10
Appendix A: Full Code for Task 12
Bibliography 16

Section 1: Data Import, Summary, Pre-processing and Visualisation

The data required was given to us in a CSV file. This contained all relevant feature variables, along with the resulting status of the reactor – either 'Normal' or 'Abnormal' – with each value separated by a comma or newline character to delimit them. There are just under one thousand entries in this file.

In order to import this file into a python script, the DataFrame feature of the Pandas library was used, in particular the "read_csv()" function, which is dedicated to reading CSV files into a DataFrame. However, an issue with this is that the data is referenced using its header name, for example "Status", or "Vibration_sensor_3". Naturally this is a cumbersome method to select data for each column, and therefore the final variable used for the data was a two-dimensional numpy array, containing each entry and their values, obtained by using the "to_numpy()" function of Pandas. The header values were stored in a separate variable. In order to improve efficiency, a specific function was created which returned both the numpy array of data entries and the feature headers. This decreases memory usage, as the DataFrame which was initially created is not available for use in the rest of the script.

```
def get_data():
    csv = pd.read_csv("ML2.csv")
    data = csv.to_numpy()
    np.random.shuffle(data)
    labels = csv.columns
    return data, labels

data, labels = get_data()
```

Figure 1: Function to return data and header values

Pre-processing is one of the most important steps in creating accurate Machine Learning algorithms. It entails analysing the dataset and removing certain entries – for example impossible data combinations or out-of-range values. Another potential issue which would be resolved during pre-processing is having missing values. This can be remedied by either sourcing the values for these fields elsewhere and updating the dataset, or simply removing this entry altogether. An alternative method is called imputation, where missing data are substituted for other values. One of the most basic imputation methods is to calculate the value based on the mean of its neighbouring values (Chakure, 2019). Fortunately, there were no issues identified with the given file, and thus minimal pre-processing work was required. The Status column consisted of the categorical variables 'Normal' and 'Abnormal', which could have been edited to contain either a 0 or 1 to indicate status, however it was deemed unnecessary to change this, since all functionality remained the same regardless.

Another key feature of pre-processing is the shuffling of the dataset. This ensures that, when splitting the data into training and testing datasets, the data in each is varied, and thus the classification models are slightly different with each run of the program. In addition, shuffling data removes any possibility that there are trends in the data which relate to the order of data entries, for example if there was an issue with the reactor on a certain day (Chicco, 2017). Another step was to split the data based on its status column, allowing graphs to be plotted comparing the Abnormal and Normal data.

A summary of the dataset is a useful step to visualise and comprehend any large amount of raw data. This included the mean, median and standard deviations of every variable in the file. Both 'vibration_sensor_3' and 'pressure_sensor_1' had significantly higher standard deviation values than every other feature, which potentially indicates a fault or external factor impacting the results of the sensor, as the spread of values is much higher. This is further confirmed by the mean and median values being markedly greater than others recording the same type of data.

```
Show stats? y/n y
DATA STATISTICS:
Total count: 996
Normal count: 498
Abnormal count: 498

Feature stats:
Power_range_sensor_1 mean: 4.999573893574293
Power_range_sensor_1 median: 4.8811
Power_range_sensor_1 max: 12.1298
Power_range_sensor_1 min: 0.0082
Power_range_sensor_1 standard deviation: 2.763467190477739

Power_range_sensor_2 mean: 6.3792731526104385
Power_range_sensor_2 median: 6.4704999999999995
Power_range_sensor_2 max: 11.9284
Power_range_sensor_2 min: 0.0403
Power_range_sensor_2 standard deviation: 2.3114075826079636
```

Figure 2: Summary of data values. Continues to describe other feature variables.

Two plots were required to be created in order to visualise the dataset. These included analysing two features in particular – the vibration sensors one and two, respectively. The initial graph, a box plot, shows the range, median and upper- & lower-quartiles for both abnormal and normal entries, and any points considered outliers are individually drawn. One variable in Pyplot, the library used to create the plots, is 'whis', which changes the length of the plot's whiskers, and thus which entries are considered outliers. The default value for this is 1.5, however when viewing this it was felt that there were too many points outside of the whiskers that hadn't been taken into account, and thus it was changed slightly to 1.75. This plot indicates that vibration levels tend to be a small amount higher in tests considered 'normal', although their range is larger, so the higher median & larger interquartile range could be due to this. However, there is one outlier in the abnormal dataset which is particularly high. This is potentially expected due to the nature of abnormal entries, however could be discarded when training the classifier in order to increase accuracy.

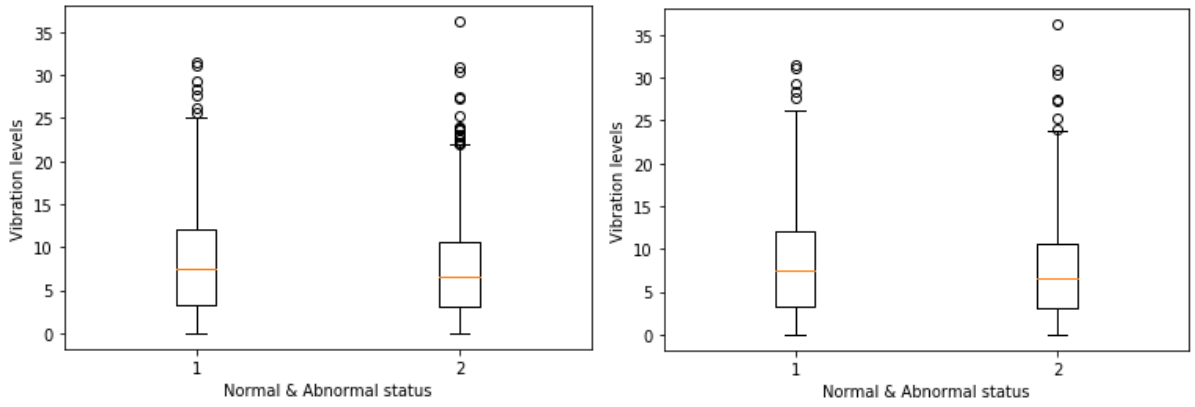

Figure 3: 'whis' = 1.5 (left) vs 1.75 (right)

The second plot was a density plot, or histogram, and demonstrates the amount of entries in given ranges – in this case in intervals of five. This alludes to the fact that generally, lower values from this sensor are often more likely to have a normal status, with the exception of values between 0 and 5. The range of 25-30 had the biggest differential between normal and abnormal classifications, with there being approximately 20 instances of abnormal values in this range, while only around five or six normal. Lower values are the most common, with the majority of entries being between 0 and 15.

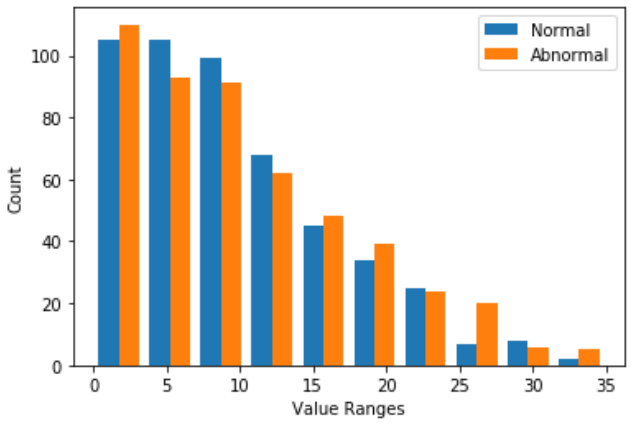

Figure 4: Density plot of 'Vibration_sensor_2'

Section 2: Discussion on Selecting an Algorithm

This section analyses the description of a Machine Learning intern who wants to select an algorithm for this scenario. There are some immediate assumptions that can be made to estimate which algorithms will be the most efficient, fast and effective in creating an accurate classifier. For example, a k-Nearest Neighbour approach may not be the best method due to the model not only being incredibly sensitive to outliers, which as previously discussed there may be a large amount of, but also due to its computational cost: as all training data must be examined each time a prediction is made, this would not be a good fit for a model which frequently makes predictions.

Support Vector Machines (SVMs) are great at generalisation and dealing with high-dimensional data, which may make it suited to this task. However, it is often difficult to fully understand a fully trained SVM model and the effect of changing its parameters. Artificial Neural Networks are one of the best methods for dealing with 'noisy' data, where values are affected by external factors and have a slight amount of randomness to them (Lorena, et al., 2011). This would most definitely prove useful when analysing this dataset as it is stated in the assignment brief that there is an element of stochastic character. Random Forest classifiers are often one of the most popular algorithms to use, given both the lightweight nature of decision trees and the fact that the model will not become overfit as the number of trees in the forest increases.

There are many classifiers used by the intern, but one specifically achieved an accuracy score of 90%. This measurement of scoring a classifier is sometimes known as the 'holdout method' and can be obtained from comparing the predicted values to their actual ground truths, and calculating the percentage of classifications which are correct versus the total datapoints (Raschka, 2018). However, this may not necessarily be the best algorithm to use in this circumstance. For example, when randomly shuffling training and testing data there may be cases where the test data is very similar to the data the classifier was trained upon, and therefore the accuracy score is reported higher as these data will be classified the same as the training data. To ensure this is not the case, accuracy tests must be performed multiple times, recording each score and calculating the mean average from all tests. This will eliminate the chance of the data being randomly skewed.

Another technique to select the best algorithm for a given purpose is to use a different method of verification altogether. One prominent alternative is to use K-Fold Cross Validation. This is a "statistical method used to estimate the skill of machine learning models" (Brownlee, 2018). It's both a simple and effective way to compare algorithms and estimate error and has the benefits of being easy to understand and implement, in addition to being one of the most universal model selection algorithms (He & Fan, 2019). The process of this will be detailed in section four.

Leave One Out (LOO) cross-validation is also an algorithm for estimating which classifier is the best to use in a given scenario. This can be thought of as a special case of K-Fold. Although it is often computationally expensive and doesn't scale well to large datasets, LOO can be useful to use with a small amount of data, especially when removing any amount of data from the training set would be incredibly detrimental to the model (Raschka, 2018).

Section 3: Designing Algorithms

Artificial Neural Networks

An Artificial Neural Network (ANN, or simply NN), as the name suggests, is a method of classifying data using a large array of individual 'neurons', all connected together. Each neuron is given multiple inputs, either from the dataset or from other neurons. Weights are applied to each input and are totalled up in a function known as a summer, along with a bias value which acts as a weight with an input of 1. The output of this is fed into a separate function: the threshold activation function. For a perceptron, this is a simple linear function which outputs a 1 for positive values and 0 for negative.

For other types of neurons there other many others used, for example Sigmoid, TanH and ReLU (Rectified Linear Unit) activation functions. The sigmoid function is the most common in ANN applications, which maps its input between 0 and 1. TanH is very similar however its outputs range from -1 to 1, making it much more useful in some cases. ReLU is often considered the default algorithm in Deep Learning, as it's one of the simplest, while proving effective in many circumstances. This returns zero for all negative values, but purely returns the input value if positive. There are many variations of this, including Parametric and Leaky ReLU, and are mainly focussed on editing the negative output section of the function (Liu, et al., 2019). This output is then fed into one or many neurons, and the process begins again.

Neural networks can have one or more layers; single layer perceptrons are incredibly limited in their functionality, whereas a multi-layered NN using the sigmoid activation function will be able to fit and predict data much more accurately. Layers in between the input and output are called 'Hidden Layers', whose purpose is to transform data received by input neurons into something useful to the output neurons.

Backpropagation is one of the main features of Neural Networks which allows multi-layered nets to calculate which weights require updating. Updating weights, and thus improving efficiency, is an iterative process, where a forward pass is performed, and error calculated (Kulshrestha, 2019). This error is propagated backwards through the network and used for calculating the amount each weight must change by, using gradient descent. The equation to edit a weight is $\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$, where i and j indicate which neuron and weight to adjust, E is the error that's been backpropagated, η is the learning rate, w is the weight, Δw represents the change in weight needed, and finally the ∂ symbols indicate that it's the derivative, with respect to the weight. This method of forward- and backpropagation is repeated multiple times for each value in the dataset. Once this has been iterated over in its entirety, one epoch has occurred. An ANN can often run for well over 100 epochs, though this algorithm is relatively computationally expensive so may take a while to run fully.

After the data pre-processing and reporting of statistics, the python script splits the given data into training and testing datasets. This is where shuffling the data earlier is necessary in order to remove potential biases as previously discussed. The index at which to split the dataset is calculated by multiplying the length of dataset with the ratio of train:test data. In this case 90% of the entire dataset was to be training data, which makes for a 0.9 value of the ratio. The final part of splitting the data uses a Python functionality called slicing, which takes an array's values starting from one index and ending at another. Using the index calculated previously as the value to split to and from, two new arrays containing both training and testing data respectively are created.

```
#**TRAIN/TEST SPLIT**
splitLine = round(len(data) * 0.9)
train = data[:splitLine]
test = data[splitLine:]
```

Figure 5: Code to split data into training and testing data. Assumes the entries have been shuffled previously.

Various ANNs were created when running the script, each with a slightly varied number of epochs, as described above. The accuracy score was different each time the code was run, due to the slight variations of training and testing data however was generally around 85-90% for 200 epochs – the default for the library used in the script. Changing the limit of epochs affected the accuracy of the classifier on the testing data insofar as the greater the number of epochs, the more accurate the model generally was.

```
def ann(epoch):
    print("Training ANN with", epoch, "epochs....")
    ANN = mlp(hidden_layer_sizes=(500,500), max_iter=epoch)
    ANN.fit(train[:, 1:], train[:, 0])
    print("Done!")
    ANNpred = ANN.predict(test[:, 1:])

    accuracy = acc(test[:, 0], ANNpred)
    return accuracy
```

Figure 6: The main code for training an ANN and predicting its accuracy

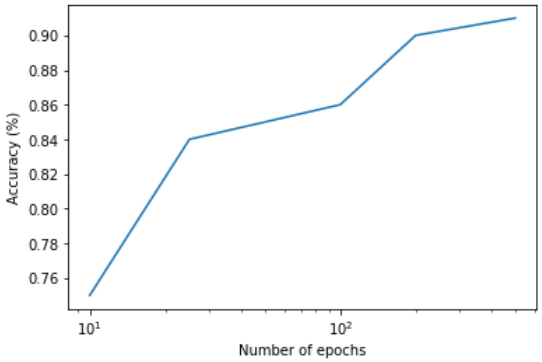

Figure 7: How the number of epochs affects classifier accuracy

Random Forests

Random Forests are a type of classification algorithm which employs the use of a large amount of Decision Trees, or CART (Classification And Regression Trees).

A decision tree involves splitting the data based on a series of conditions, known as internal nodes. These conditions are based on the features, for example “is vibration_sensor_4’s value greater than 12.7?” If the answer to the condition is positive, one path down the tree is taken, and if it’s negative, the other is explored. The main computation for ‘growing’ a decision tree is the process of deciding which features and conditions to use in the tree in order to gain the most efficiency.

The goal of each split is to maximise the information gained by the tree, which requires the minimisation of entropy or impurity. This is calculated using $-\sum_i y_i \log_2 y_i$, where y_i is the ratio of elements from each of the feature variables in the branch (Ricaud, 2017). Another formula is applied to this, in addition to a given attribute we are considering creating a split at, in order to calculate the information gain. The feature with the highest information gain is the best one to create a split at.

Each node is created with a certain number of inputs attached to it, thus increasing the minimum number of samples per node will decrease the total number of possible nodes that can be created. Changing this value can make the tree much more efficient due to ensuring all nodes use a substantial amount of the samples.

In order to ensure that every tree is different, each is given a subset of the dataset with replacement, meaning there may be duplicate instances of the same entry within each tree's sample, or 'bag'. This process is called 'bagging' and has been a key feature of Machine Learning since the 1990's. Once each tree has fully 'grown' the classifier is fitted and ready to make predictions. This involves each tree making its own decision – for a regression tree this will be the output value and for a classification tree, the predicted class of the predicted data point. These values are then either averaged out and given as the entire forest's output (for regressors) or counted, and the class with the most votes from each tree is given as the output (for classifiers).

As the 'SKLearn' library was used in the python script for this assignment, the creation of a Random Forest classifier seems to be a very simple process on the surface, though many more steps are being taken by the library's code. The Forest's accuracy was generally higher than that of the ANN, on occasion reaching up to 96%. As shown by Figure 8, the model's ability to accurately predict an input's class increased every time the total number of trees in the forest increased.

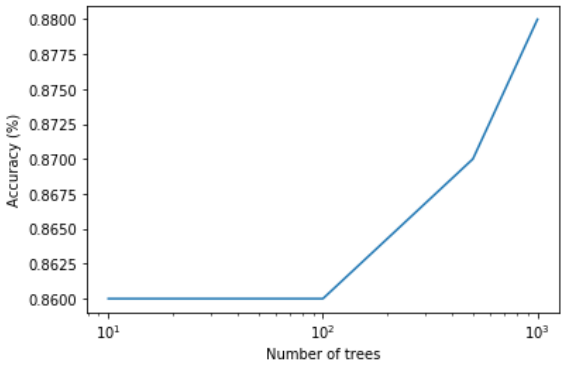

Figure 8: How the number of trees in a Random Forest algorithm affects classifier accuracy

```
def randForest(trees):
    print("Training Random Forest with", trees, "trees....")
    thisRF = rf(n_estimators=trees, min_samples_leaf=5)
    thisRF.fit(train[:, 1:], train[:, 0])
    print("Done!")
    RFpred = thisRF.predict(test[:, 1:])
    accuracy = acc(test[:, 0], RFpred)
    return accuracy
```

Figure 9: Main code for creating Random Forests and measuring accuracy

Section 4: Model Selection

There are many methods to evaluate which models are best to use in any given scenario, in addition to calculating the values of hyperparameters, such as the number of neurons in an ANN, which give the most accurate result. One of the most popular methods is K-Fold Cross Validation. This is a validation technique which splits a dataset into k sections, where k is a value greater than two. One of these sections is taken at random to be the testing set, and the remaining $k - 1$ groups are all combined to serve as the training data.

The model is then trained and evaluated in the regular way, using the new training and testing sets. The above process is repeated k times, with each repetition choosing a different group, or 'fold', to function as the testing data and leaving the rest to train (Brownlee, 2018). All the evaluation scores are averaged when this process is complete, and this gives a much more reliable representation of the accuracy of the model. This can be performed multiple times on a model, changing parameters each iteration, in order to calculate the best version of a given classifier. K-Fold is much more effective than the method initially used, as it removes any risk of what's known as the 'holdout bias', which stems from the evaluated model only being trained on a fraction of the data, with the remaining fraction used for testing.

```
Mean accuracy of 50 neurons: 0.8072828
Mean accuracy of 20 trees: 0.908656565
Mean accuracy of 500 neurons: 0.823333
Mean accuracy of 500 trees: 0.92466666
Mean accuracy of 1000 neurons: 0.81829
Mean accuracy of 10000 trees: 0.928686
Best value of neuron count: 500
Best number of trees to use: 10000
Average for ANN: 0.8163030303030303
Average for RF: 0.9206700336700336
Best model: Random Forest
```

Figure 10: Comparison of ANNs & RFs with number of neurons & trees varied

The features that were required to be varied for this assignment were the number of neurons in the hidden layer of the ANN (50, 500 & 1000), and the number of trees grown in the Random Forest (20, 500 & 10000). K-Fold Cross Validation was performed on each model for every parameter with k as 10, meaning each model and its differing parameters were trained and their accuracy estimated ten times. As shown in Figure 10, the accuracy can vary depending on the number of neurons or trees.

There is a general positive correlation between increasing the number of trees in the forest and an improved accuracy of the model. This is an expected behaviour as a Forest can never be overfit when increasing the total trees; it simply becomes a case of diminishing returns where adding trees takes substantially more time to train, and the accuracy rate barely improves. This can be noticed between the accuracy scores for 500 and 10000 trees, which only increases by 0.002, or 0.2%. The differential in time taken to train these two models may not be worth the very minor increase in accuracy.

The accuracy of the ANN, however, was not quite as simple. As the amount of neurons increases, the network becomes more complex, as each neuron's output feeds into multiple others. This also means that more of the data is 'memorized' by neuron weights, which leads to overfitting. This is the reason the most accurate model was the one with 500 neurons as this balances underfitting and overfitting the best out of the three. While in this case the error between 500 and 1000 neurons in the hidden layer is only around 1.2%, there may be cases where a much more significant error margin is evident. In addition, the extra complexity of the network leads to much longer training times, which is naturally not desirable.

There is a general positive correlation between increasing the number of trees & neurons and an improved accuracy of the model, however the benefit this provides may be negligible, and is can even be lower in some circumstances. The improvement to accuracy may not necessarily be worth the extended time taken to train the classifier, especially if the model may need to be frequently re-trained to account for other data. In terms of selecting the best model, the Random Forests had consistently higher accuracy scores than the ANNs created. In addition, the average of all three variations on Forest size was higher than the average for varying neuron count, thus the Random Forest would be the better algorithm to use for this dataset. It is also the quicker of the two, so if the model did have to be frequently re-trained this would be an added benefit.

```
for i in range(len(neuronArr)):
    for train_index, test_index in kf.split(data):
        kfAccANN[i].append(ann(data[train_index], data[test_index], neurons=neuronArr[i]))
        kfAccRF[i].append(randForest(data[train_index], data[test_index], trees=treeArr[i]))
```

Figure 11: Main code for K-Fold algorithm, using SKLearn python library

Appendix A: Full Code for Task

```
#Import relevant libraries:
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statistics

#Function to get dataset:
def get_data():
    csv = pd.read_csv("ML2.csv") #Reads CSV file into Pandas DataFrame
    data = csv.to_numpy() #Converts to numpy array
    np.random.shuffle(data) #shuffles data
    labels = csv.columns #gives the column names (used for graphing)
    return data, labels

#Function to get accuracy of ANN with given parameters
def ann(train, test, epoch=200, neurons=500):
    ANN = MLPClassifier(hidden_layer_sizes=(neurons), max_iter=epoch, activati
on="logistic") #Initialises a neural net using SKLearn's prebuilt library, wit
h required hyperparameters
    ann.out_activation_ = "logistic" #sets output layer's activation function
to required

    print("Training ANN:", epoch, "epochs,", len(train), "data,", neurons, "ne
urons....") #Prints relevant info
    ANN.fit(train[:, 1:], train[:, 0]) #Trains model using train array
    print("Done!")
    annPred = ANN.predict(test[:, 1:]) #makes predictions using testing datase
t

    acc = accuracy_score(test[:, 0], annPred) #uses SKLearn's library to calcu
late accuracy. the percentage correctly classified, over the total predictions
.
    return acc

#function to get accuracy of Random Forest with given parameters
def randForest(train, test, trees=1000,  msl=1):
    rf = RandomForestClassifier(n_estimators=trees, min_samples_leaf=msl) #Ini
tialises RF using SKLearn
    print("Training Random Forest:", trees, "trees,", len(train), "data, ", ms
l, "min samples per leaf....") #Prints relevant info
    rf.fit(train[:, 1:], train[:, 0]) #Trains model using train array
    print("Done!")
    rfPred = rf.predict(test[:, 1:]) #makes predictions
```

```
    acc = accuracy_score(test[:, 0], rfPred) #calculates accuracy
    return acc

data, labels = get_data() #gets data using above function

norm = data[np.where(data[:, 0] == "Normal")[0]] #splits data into normal & ab
normal, for data summary & graphing only
abnorm = data[np.where(data[:, 0] == "Abnormal")[0]]

#Shows statistics about the dataset. Self explanatory.
if input("Show stats? y/n ") == "y":
    print("DATA STATISTICS:")
    print("Total count:", len(data))
    print("Normal count:", len(norm))
    print("Abnormal count:", len(abnorm))
    print()
    print("Feature stats:")
    for i in range(1, 13):
        print(labels[i], "mean:", np.mean(data[:, i]))
        print(labels[i], "median:", np.median(data[:, i]))
        print(labels[i], "max:", np.max(data[:, i]))
        print(labels[i], "min:", np.min(data[:, i]))
        print(labels[i], "standard deviation:", np.std(data[:, i]))
        print()


print("Box plot: Vibration_Sensor_1 (Normal vs Abnormal):")
plt.boxplot([norm[:, 9], abnorm[:, 9]], whis=1.75) #shows boxplot. whis change
d from 1.5 to 1.75.
plt.xlabel("Normal & Abnormal status")
plt.ylabel("Vibration levels")
plt.show()

print("Density plot: Vibration_Sensor_2:")
plt.hist([norm[:, 10], abnorm[:, 10]]) #shows histogram/density plot
plt.xlabel("Value Ranges")
plt.ylabel("Count")
plt.legend(["Normal", "Abnormal"])
plt.show()

splitLine = round(len(data) * 0.9) #gets index at which to split dataset
train = data[:splitLine] #slices full array until the split index
test = data[splitLine:] #slices full array from split index

epochs = [10, 25, 100, 200, 500] #epochs to compare
trees = [10, 50, 100, 500, 1000] #trees to compare
accANN = []
accRF = []
```

```
for i in range(len(epochs)): #iterates through epochs (and trees) array
    accANN.append(ann(train, test, epoch=epochs[i])) #adds the accuracy for mo
del with given epoch count to array
    accRF.append(randForest(train, test, trees=trees[i])) #adds the accuracy f
or model with given tree count to array

plt.semilogx(epochs, accANN) #plots number of epochs logarithmically against a
ccuracy scores
plt.xlabel("Number of epochs")
plt.ylabel("Accuracy (%)")
plt.show()

plt.semilogx(trees, accRF) #plots number of trees logarithmically against accu
racy scores
plt.xlabel("Number of trees")
plt.ylabel("Accuracy (%)")
plt.show()

print("ANN with 200 epochs is", accANN[3]*100, "percent accurate.") #shows acc
uracy of ANN with defaut number of epochs.
print("Random Forest with 1000 trees is", accRF[4]*100, "percent accurate.")

kf = KFold(10) #Initialises KFold using SKLearn library,

kfAccANN = [[], [], []]
kfAccRF = [[], [], []]
neuronArr = [50, 500, 1000] #neuron counts to compare
treeArr = [20, 500, 10000] #tree counts to compare

for i in range(len(neuronArr)): #iterates through neuron (and tree) counts to
compare
    for train_index, test_index in kf.split(data): #iterates k(10) times, and
sets the two vals to relevant sections of the dataset. test will be 10% of the
 dataset for k=10
        kfAccANN[i].append(ann(data[train_index], data[test_index], neurons=ne
uronArr[i])) #adds accuracy of ANN with given parameters to array. # of neuron
s changes for each iteration of outer for loop
        kfAccRF[i].append(randForest(data[train_index], data[test_index], tree
s=treeArr[i])) #adds accuracy of RF with given parameters.

meanAccANN = []
meanAccRF = []
for i in range(len(neuronArr)):
    meanAccANN.append(statistics.mean(kfAccANN[i])) #adds mean accuracy of all
 10 scores from KF to array.
    meanAccRF.append(statistics.mean(kfAccRF[i])) #adds mean accuracy of all 1
0 scores from KF to array.
```

```
    print("Mean accuracy of", neuronArr[i], "neurons:", meanAccANN[i]) #prints
 mean
    print("Mean accuracy of", treeArr[i], "trees:", meanAccRF[i]) #prints mean

bestANN = neuronArr[meanAccANN.index(max(meanAccANN))] #finds the best value o
f neuron count in mean ANN array
bestRF = treeArr[meanAccRF.index(max(meanAccRF))] #finds the best value of tre
e count in mean RF array

print("Best value of neuron count:", bestANN)
print("Best number of trees to use:", bestRF)

mANN = statistics.mean(meanAccANN) #finds total mean of all ANN models
mRF = statistics.mean(meanAccRF) #finds total mean of all RF models

best = "Artificial Neural Network" if mANN > mRF else "Random Forest" #if mean
 ANN accuracy is higher than mean RF accuracy return ANN, otherwise return Ran
dom Forest

print("Average for ANN:", mANN)
print("Average for RF:", mRF)

print("Best model:", best)
```

Bibliography

Brownlee, J., 2018. *A Gentle Introduction to K-Fold Cross Validation.* [Online]
Available at: https://machinelearningmastery.com/k-fold-cross-validation/
[Accessed 11 December 2019].

Chakure, A., 2019. *Data Preprocessing.* [Online]
Available at: https://towardsdatascience.com/data-preprocessing-3cd01eefd438
[Accessed 17 December 2019].

Chicco, D., 2017. Ten quick tips for machine learning in computational biology. *BioData Mining,* 10(35), pp. 1-17.

He, J. & Fan, X., 2019. Evaluating the Performance of the K-fold Cross-Validation Approach for Model Selection in Growth Mixture Modeling. *Structural Equation Modeling: A Multidisciplinary Journal,* 26(1), pp. 66-79.

Kulshrestha, S., 2019. *Backpropagation – Algorithm For Training A Neural Network.* [Online]
Available at: https://www.edureka.co/blog/backpropagation/
[Accessed 17 December 2019].

Liu, X., Zhou, J. & Qian, H., 2019. *Comparison and Evaluation of Activation Functions in Term of Gradient Instability in Deep Neural Networks: Proceedings of the 31st Chinese Control and Decision Conference, CCDC 2019.* Nanchang, China, IEEE.

Lorena, A. C. et al., 2011. Comparing machine learning classifiers in potential distribution modelling. *Expert Systems with Applications,* 38(5), pp. 5268-5275.

Raschka, S., 2018. *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning.* [Online]
Available at: https://arxiv.org/pdf/1811.12808.pdf
[Accessed 16 December 2019].

Ricaud, B., 2017. *A simple explanation of entropy in decision trees.* [Online]
Available at: https://bricaud.github.io/personal-blog/entropy-in-decision-trees/
[Accessed 16 December 2019].