Data Import, Summary, Visualisation and Pre-processing

Data Summary

The summary of nuclear reactor data looks at 9 different components to allow for adequate feature and label pre-processing. The data set summary can be seen in figure 1.1. Inspections into the summary have led to concrete conclusions about the data and its structure. Observations show that there is 1 column of categorical data (the representation of a non-numerical character). ANN (Artificial Neural Networks) aren't able to process categorical data, evidencing their need to be converted. (SEGER, 2018). The columns categorical presence, suggests it to be the label, because it's seen to be located in the first column. The labels have been identified to 2 different classes, (1) Normal and (2) Abnormal; providing these are the only classes within the entirety of the dataset, we can confirm that this will require a binary classification (the number of classes can be represented as binary, 0 or 1, for the classification problem). Not only this, but the classes have equal number of data (Class 1: 498 and Class2: 498), allowing for a balanced training/testing space. Balancing training and testing data in binary classifications is evident to provide higher accuracies in ML (Machine Learning) models, including Neural Nets and Decision Trees used in the proceeding sections for the classification task. The use of unbalanced data sets provides a majority classification, where the number of a singular class is seen to be higher than another, dictating optimism towards that class. (Wei & Dunbrack Jr , 2013). The proceeding 12 columns represent each samples features; each feature was analysed for categorical data where none to be found, figure 1.2 and 1.2. Along with this, a checked function for each samples was applied to find any NULL, NONE, NAN or empty features. The analysis found that each sample was fully represented by 12 features. However, in the instance that a feature was categorical, encoding that data allows for a feasible contribution to be derived from the model during the training process. There are many techniques for encoding, (1) binary, (2) on-hot, and (3) feature hashed; a study provided evidence that one-hot encoding gave the best results when compared on multiple ML algorithms. One-hot encoding represents each category as a vector of N (number of categories), with the vector being filled with all 0's, excluding a singular 1 used for that category. Intern, this allows the sample to be interpreted correctly, but does lead to increased data complexity and model computational cost. (SEGER, 2018).

Fig 1.

```
Statistic Summaries:
--------------------

Number of Element in Dataset: 12948
Number of Features: 12
Number of Samples: 996
Number of Continuous Columns in Dataset:  12
Number of Categorical Columns in Dataset:  1
Number of Categorical Variables in Dataset:  2
Number of Samples in for Categorical status 'NORMAL':  498
Number of Samples in for Categorical status 'ABNORMAL':  498
Missing Values in Dataset: False.
```

```
Statistic Summaries:
--------------------

Status                    object
Power_range_sensor_1     float64
Power_range_sensor_2     float64
Power_range_sensor_3     float64
Power_range_sensor_4     float64
Pressure_sensor_1        float64
Pressure _sensor_2       float64
Pressure _sensor_3       float64
Pressure _sensor_4       float64
Temperature_sensor_1     float64
Temperature_sensor_2     float64
Temperature_sensor_3     float64
Temperature_sensor_4     float64
dtype: object
```

Fig 1.1. Statistical Summary, (1) Elements, (2) No. Features, (3) No. Features, (4) Continuous Columns, (5) Categorical Columns, (6) Categorical Variables within Column, (7) (8) No. samples within categorical column (with names), and (9) Missing values. **Fig 1.2.** Features Datatype.

Another method used to analyse and evaluate the features is through statistical extraction, figure 2, which helps to identify different feature traits and whether data normalisation is required. The description includes: (1) Count, showing the number of samples within the entirety of the

dataset. (2) Mean represents the sum of the all occurrences of each feature, divided by the count. (3) STD (Standard Deviation) shows how each feature differs from the mean. (4) Min, shows the lowest value in each feature. (5) 25% (Lower quartile) signifies the percentage value for each feature column. (6) 50% (Medium) the middle value in each feature column. (7) 75% (Upper quartile) characterises the percentage value for each feature. (8) Max represents the highest value in each feature. Conclusions from the data statistical extraction shows that minimum, maximum, and the range between them prove that each feature is scaled differently. This provides useful in identifying whether or not the features need scaling down, allowing normalisation.

Fig 2.

	count	mean	std	min	25%	50%	75%	max
Power_range_sensor_1	996.0	4.996993	2.762409	0.008200	2.892120	4.881100	6.775377	12.129800
Power_range_sensor_2	996.0	6.378542	2.313596	0.040300	4.931750	6.470500	8.104500	11.928400
Power_range_sensor_3	996.0	9.227265	2.532658	2.583966	7.500425	9.348000	11.046800	15.759900
Power_range_sensor_4	996.0	7.354094	4.356061	0.062300	3.438141	7.071550	10.917400	17.235858
Pressure_sensor_1	996.0	14.199127	11.680045	0.024800	5.014875	11.716802	20.280250	67.979400
Pressure _sensor_2	996.0	3.077681	2.126752	0.008262	1.412600	2.674746	4.502500	10.242738
Pressure _sensor_3	996.0	5.748279	2.526864	0.001224	4.003975	5.741357	7.503578	12.647500
Pressure _sensor_4	996.0	4.997002	4.165490	0.005800	1.581625	3.859200	7.599900	16.555620
Temperature_sensor_1	996.0	8.155479	6.174639	0.000000	3.190292	6.734450	11.246400	36.186438
Temperature_sensor_2	996.0	10.001593	7.336233	0.018500	4.004200	8.793050	14.684055	34.867600
Temperature_sensor_3	996.0	15.186910	12.159565	0.064600	5.508900	12.185650	21.835000	53.238400
Temperature_sensor_4	996.0	9.933125	7.282817	0.009200	3.842675	8.853050	14.357400	43.231400

Statistical Summary for each Feature; (1) Count, (2) Mean, (3) STD, (4) Min, (5) 25%, (6) 50%, (7) 75%, and (8) Max.

Visualisations

The interpretation of the box plot shows a distribution in a five number summary (similar to that provided in the statistical extraction in visual form), which assists in identifying outliers and their values. The plot shows that the data doesn't have any outliers (usually found below the lower or upper quartile). The inclusion and exclusion of outliers provides both benefits and drawbacks; including them can indicate behaviours which normalised data can't identify, but excluding them can improve accuracy of estimators research suggests. (Acuna & Rodriguez , 2019). The box plot shown in figure 3. Image A. Represents the original data for status and power range sensor 1. Furthermore, normal status represents the power range sensor 1 has a wider distribution, leading to more scattered data. In contrast, the abnormal power range sensor 1 converges its distributions closer to its minimum, suggesting the majority of the samples are smaller. Overall, normal and abnormal classes appear to be on different scales based on their min and maxes, but both evidence similarities because both classes mediums overlap their respective boxes.

A density graph is more visually stimulating than the common histogram, where a kernel density estimation is applied to produce a continuous curve. The density graph shown in figure 4. Image A characterised the original data for status and pressure sensor 1. The curve estimates that more pressure accumulates between 0-30 in the abnormal data than normal data. Whereas, between 0-10, normal has more data compared to abnormal in this range. Overall, both graphs show similarities (seen in figure 4. Image A. where the shaded regions between both lie), however, deductions can be portrayed that normal has a smaller range of data, where abnormal has more variance (spread out further along x axis). The more variance between the two classes shows the data is non-normalised.

Figure 5 represents the code used to plot the graphs seen in figures 3 and 4. The plot takes the normal and abnormal statuses and plots the 5 summary statistics from the matplotlib library, using the boxplot function. Similarly, the density plot uses sns plot within matplotlib for plotting a histogram with a Gaussian kernel density estimate for both normal and abnormal curves. The ranges for the density graphs have been limited to the minimum and maximum number for statuses in the x-axis.

Fig 3.

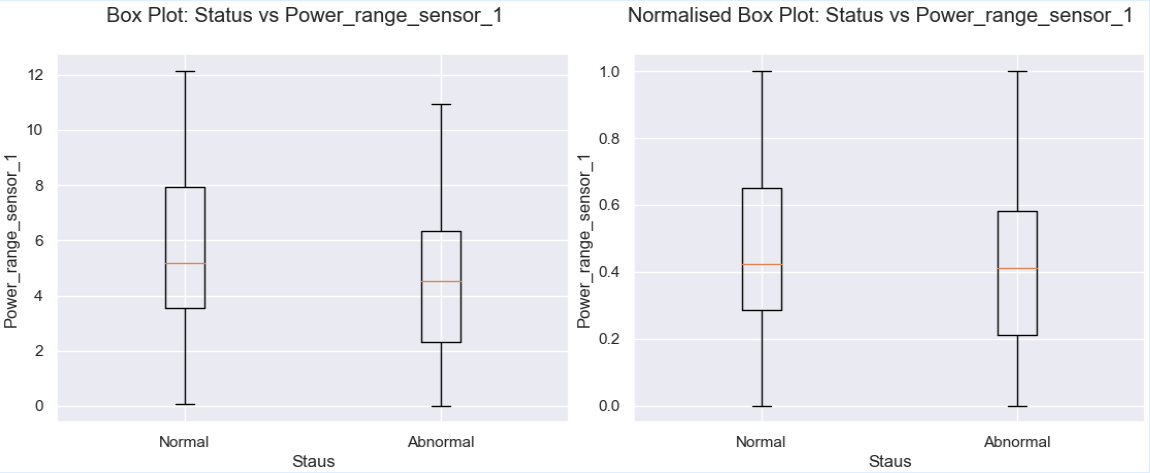


Fig 3. Image A; Original data box plot, class (status) against Power Range Sensor 1.
Image B; Normalised data box plot, class (status) against Power Range Sensor 1.

Fig 4.

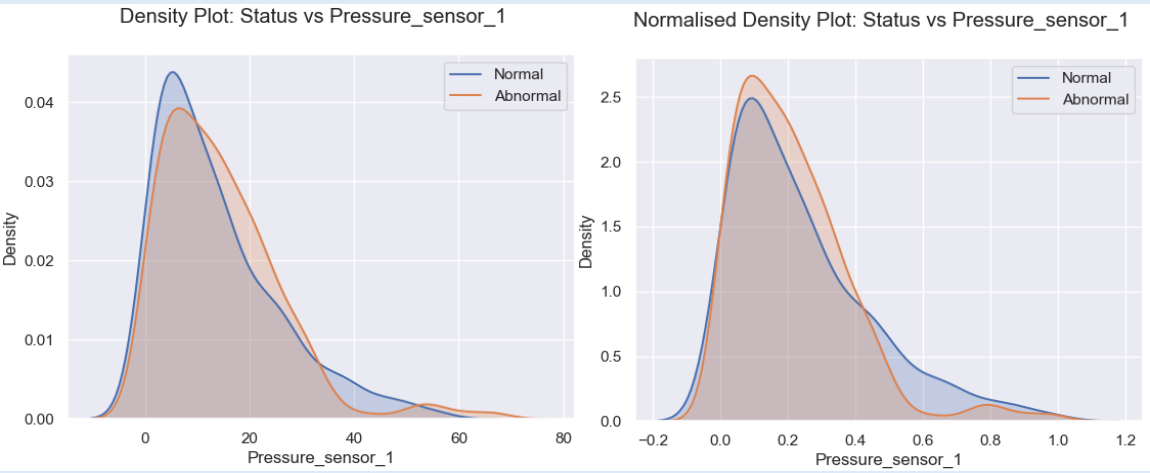


Fig 4. Image A; Original data density graph, class (status) against Power Sensor 1. **Image B**; Normalised data density graph, class (status) against Pressure Sensor 1.

Fig 5.

```
# Subplot the data to a box plot for both
plt.figure(1)
plt.boxplot(data_power_range_sensor1_original, labels=["Normal", "Abnormal"])
plt.suptitle('Box Plot: Status vs Power_range_sensor_1')
plt.xlabel('Staus')
plt.ylabel('Power_range_sensor_1')

# Subplot the density plot of the Status and Pressure_sensor_1
plt.figure(3)
pt = sns.kdeplot(Normal.Pressure_sensor_1, shade=True, label="Normal")
pt = sns.kdeplot(Abnormal.Pressure_sensor_1, shade=True, label="Abnormal")
plt.xlim(0, 67.97)
x = [14.99, 14.99]
y = [0, 0.045]
plt.plot(x, y)
plt.suptitle('Density Plot: Status vs Pressure_sensor_1')
plt.xlabel('Pressure_sensor_1')
plt.ylabel('Density')
```

Overall, no trends can be identified from all the features when plotting their respective features together, seen within each set of temperatures, pressures and powers. This may be due to the data sets size, resulting in an unfair trend evaluation because there isn't a larger enough sample to understand the true connections between each respective feature (1, 2, and 3).

Pre-processing

DPP (Data Pre-Processing) is a useful technique within ML, as data may need to be fine-tuned to allow algorithms to train efficiently. Research concluded that having a DPP plan to mitigate the effect of anomalies on the generalisation performance of SL (Supervised Learning) algorithms. (Alexandropoulos, et al., 2019). Techniques for DPP include, (1) Normalisation, (2) Feature Selection, (3) Noise Reduction, (4) Outlier Detection, (5) Instance Selection, and (6) Missing value Imputation. After evaluation of each feature, the difference between them has been identified as the scale. Each can be seen to represent different minimums, maximums and clustered areas within certain ranges.

Other techniques are analysed for normalisation; such as, missing values and outliers which neither to be found. However, consideration to feature selection was made but due to lack of samples importance wasn't obtained. With this being said, applying a min-max scaling normalisation technique to generalise the data has been proven to be advantageous; retaining relationships between the original values. (Singh, et al., 2015). It was also evidenced to slightly worsen the accuracy that the ML algorithm predicted for each class in this study, proving that normalisation may something effect the generalisation too much. This min-max normalisation technique achieves linear transformations on original data with equation 1. Further understanding will applied in 'Algorithm Design' Section. For this example all sample features have all been normalised between 0-1, potentially helping the model by having consistent scales between features. Figure 3. Image B shows the normalisation box plot; proving that the ranges of data are still the same but their values now are represented between 0-1. This can also be seen in Figure 4. Image B where the scales have been reduced according to min-max and the density's normalised within pressure sensor 1 samples. Figure 5 shows the code used for min-max normalisation provide through the Sklearn library for converting the scales between 0-1 for all features with the data. Equation 1 represents the formula used within the MinMaxScaler fit function, which is applied to both normal and abnormal. Both class are then split separately for plotting purposes.

(1) $$Z = \frac{x - Min(x)}{Max(x) - Min(x)}$$

The min and max formula used to calculate the new values for each feature.

Fig 6.

```
# import the data set into a np array
data_set = (np.loadtxt('Data.csv', delimiter=",")).transpose()
# extract the the 12 features and no the label
whole_set = (data_set[1:13, [i for i in range(996)]]).transpose()
# create the scaler class with a range of 0 and 1
_min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
# transform the whole set features to new scale
features_scale = _min_max_scaler.fit_transform(whole_set)

# transpose the data back to column 12 rows 996
data_set = features_scale.transpose()
# split the data into normal and abnormal
normal = (data_set[:, [i for i in range(498)]]).transpose()
abnormal = (data_set[:, [i + 498 for i in range(498)]]).transpose()
```

Sklearn MinMaxScaler normalisation

Discussion for Selecting an Appropriate Model

When considerations are being made about model optimality, multiple factors are required to make an accurate choice. The brief identifies the student splits his dataset into: (1) Train, (2) Validation, and (3) Test. This evidences that the validation and test data have different samples from that used within the train environment. The exclusion of resubstitution validation, in practice, provides realistic performance metrics because it's tested on unseen data. Whereas, resubstitution validation shows to provide heavy optimistic bias, resulting from memorisation of training data.

However, using 30% of the data on validation and testing can lead to inaccurate generalisation estimates because of the large hold-off of the data from training.

Furthermore, the use of K-fold CV (Cross Validation) hasn't been included in the student's model selection. The use of a CV estimator over the single hold-out approach produces a lower variance, which proves to be important when the available data is limited. Single hold-out where 70% is used for training and 10% for testing, shows a small amount of samples, which can produce a massive amounts of variances in performance when using different samples of data or samples for training and test data. This variance can be minimised through averaging over k different folds/partitions, producing less sensitive performance estimates than simply partitioning the data. This process can be taken further by repetition of CV using different data samples in each partition. However, with the split of data being in train, validation and testing; an approach to train each model using the training data, whilst hyper-tuning them using the validation set. This method is applied to each model until optimal parameters have been identified and the test set is used to evaluate them all. On the other hand, implementation of CV on top of CV to test the averages of each model against multiple variations of data ordering. This allows the model to be evaluated fully through the use of all permutations.

Moreover, data balancing hasn't been declared for train, validation and test to prove whether the student has applied equal sized samples for each class. Research indicates that using a 50/50 split of each class for a binary classification problem proves to have highest balanced accuracy; the average of TP (True Positive Rate) and TN (True Negative Rate). (Wei & Dunbrack Jr, 2013). The accuracy of 90% for testing may not show true if the model has been trained on a dominate class, which then dictates the high accuracy found in test set because only that class can be predicted. With this being said, confusions matrixes allow for valid descriptive analysis of the performance of the classification model, by analysing test input results. This approach takes into consideration the number of each class and works out TP, TN, FP (False Positive), and FN (False Negative). This misclassification rate (Error Rate) is another common approach for evaluating a model on how well it classified individual classes. This is usually expressed as a confusion matrix, including for each class a TP and TN, allowing true performance to be analysed, through having high precision and recall; found on all good classification algorithms. With this being said, the models specificity should be considered as this is the true negative, which is the classification rate of abnormality. This class is more important when compared to misclassifying normal, because in reality this could lead to reactor destruction.

Also, the accuracy of the model is an extremely important factor when monitoring nuclear reactor status, however, considerations may need to apply to the time and computational cost for model train and estimation. This takes into account the best ratio of model in terms of performance metrics and time/computation, as this data is time sensitive reporting.

Feature selection is an important concept in model selection. It provides efficiency increases in time and computation complexity, through removal of redundant features such as ones that provide any considerable input into training the model. An approach to this selection can come from model testing through iterative feature removal, where comparisons are made to output performance of not having said feature. This allows analysis of the feature importance to understand whether a feature has any contribution to actual prediction of the class. Once no relationship has been found from this evaluation, these features can be removed. This may lead to increased performance and time & computation costs, which, dependent upon the true metric (accuracy or time complexity) required from the classification task, may provide increased efficiency. Overall, the lacking evidence of the above implemented standards for adequate model selection, proves the classier chosen may not represent the most optimal choice.

Machine Learning Algorithm Designs

Dataset Manipulation

Figure 7. Shows the reports code on balancing the two status classes (Normal and Abnormal). The csv file is read into an np array where normal and abnormal class get split into separate arrays. These two arrays are seeded and shuffled according to the seed for reproducible results. The two arrays then merge together equally through alternating passing a single value from each into the single array. Balancing the data as mentioned previously will help to maximise accuracies as there won't be bias amounts of one class. Furthermore, the random seed used within this function allows the data to be shuffled the same way for all call, which allows both the ANN and RFT models to use the same order of data. This also allows Sklearns cross fold validation function to have balanced amounts of each class in both the 90% training data and 10 testing data for each K-fold. Along with this, Sklearn CV will use the same splits of data for each of the K-folds for training and testing, allowing accurate comparisons of data between models.

Fig 7.

```
# read the dataset into a np array
DATA_SET = balanced_normal_abnormal('Data.csv')

# split dataset into sets for testing and training
X = DATA_SET[:, 1:13]  # takes all the 12 features
Y = DATA_SET[:, 0:1]  # takes the label column
# balances the data set
def balanced_normal_abnormal(FILE_PATH):
    data_set = (np.loadtxt(FILE_PATH, delimiter=",")).transpose()  # read the dataset into a np array, transposes
    normal = (data_set[:, [i for i in range(498)]]).transpose()  # extracts the normal statuses
    abnormal = (data_set[:, [i+498 for i in range(498)]]).transpose()  # extracts the abnormal statuses
    np.random.seed(0), np.random.shuffle(normal)  # shuffles both the classes in the same way
    np.random.seed(0), np.random.shuffle(abnormal)  # creates a new array to store the balanced data
    k_fold = np.zeros((996, 13))
    count = 0  # used for new array
    step = 0  # used for looping the class arrays
    # loops through to the number of samples
    while count < 996:
        k_fold[count] = normal[step]  # the class arrays pass one each time loop into new balances array.
        k_fold[count+1] = abnormal[step]  # the class arrays pass one each time loop into new balances array.
        count += 2  # increments count
        step += 1  # increments step
    return k_fold  # returns the balanced data set
```

Representing the Grid search for keras model of 100 neurons found: Hyper-parameters: {'Nodes': 100, 'batch size': 512, 'epochs': 2000, 'optimizer': 'Adam'}

Artificial Neural Network

Brief Overview

ANN (Artificial Neural Network) is a classification device, used to differentiate between classes during the training phase to allow for meaningful predictions to be concluded. With the name it's given, you can expect the classification model to have high relations to that of neural connections within the human brain. These neurons are represented as circles in figure 8.1 and include: (1) Input nodes, (2) Hidden nodes, and (3) Bias Nodes; each being found within ANN's layering structure. The complete architecture can be seen in figure 9 representing; (1) An input layer, (2) A hidden layer, and (3) An Output layer. The input layer represents the parsing of input nodes (a single input neuron represents a single feature from the sample) into the first hidden layer of the network, seen as circles in figure 8.2. The Hidden Layer represents how the model learns to classify through the input training samples. The hidden layer receives incoming weights from the previous layer and creates an activation based upon these weights. The weights are represented in figure 9 as branches from each node in each layer, known as the formation of weighted connections. These weights are randomly initialised before the first sample is passed through and manipulated throughout the training process. Also, within each hidden layer, a bias is associated with each

neuron (also all randomised upon initialisation), allowing for increased flexibility through controlling if, or by how much a neuron is activated. The use of a bias stops certain neurons from not being activated, which has large importance in model training, because these can affect later neurons ability to activate. Finally, the output layer similar to its sibling, hidden, outputs the number of neurons based on the number of classes used within the classification. This whole training process can be summed through the use of 2 main concepts, forward and backward propagation.

Forward and Backward Propagation

Forward propagation is the process of activating each neuron based on its input weights, and predicting the output class from the given weights and biases. Figure 8 dictates the individual elements found within a small section of network, alongside, giving a visual representation of where each of the equations below will find their parameters. For each neuron in each layer (minus the input layer nodes) a total net sum is calculated from the weights with the previous layers out and the bias, which is repeated for each neuron, seen in equation 2. The activation (logistic) function is then computed from the total net, which determines the activeness of the node with respect to its following neuron, seen in equation 3. The final step is to calculate the total error by comparing the models output against desired for each output neuron, equation (4).

(2) $net_{hj} = (w_i * i_i + b_i)$, ***(3)*** $out_{h1} = 1(\frac{1}{1+e^{-x}})$, ***(4)*** $E_{total} = 1(\frac{1}{1+e^{-x}})$

Backward propagation can be summarised as the process of updating weights and biases, through the use of minimising the error for each output neuron. The derivative of the above functions measures the rate of change from each parameter weighting to minimise the cost function. Equation 8 is the chain rule used for computing how much the weight effects the overall error and which direction needs to updated the weight to reduce the error. Equations 5-7 are expressed through equation 8's chain rule, which all share a common relationship: Equation (5) looks at the rate of change of error with respect to the neurons activation out, Equation (6) looks at the rate of change of activation out with respect to the net sum, and Equation (7) looks at the rate of change in total net sum with respect to the weights. Once these equations have been computed, equation 9 shows how a single weight can be updated to minimise the error through using the old weight, bias and the rate of change for a weight against the error. The same can be computed with the rearrangement of the chain to look at the bias instead of weight, equation (10) is the update step for a single bias, similarly to the new weight the use of old bias and learning rate and the total error with respect to bias is used in this computation.

(5) $\frac{\partial E_{total}}{\partial out_o} = -(target_o - out_o)$, **(6)** $\frac{\partial out_o}{\partial net_o} = out_o(1 - out_o)$, **(7)** $\frac{\partial net_o}{\partial w_o} = \left(out_o * w_o^{(1-1)}\right)$,

(8) $\frac{\partial E_{total}}{\partial w_i} = \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial w_i}$, **(9)** $w_{new} = w_{old} - n * \frac{\partial E_{total}}{\partial w_{old}}$, **(10)** $b_{new} = b_{old} - n * \frac{\partial E_{total}}{\partial b_{old}}$

Furthermore, the activations function is important factor to consider, because it decides whether, or by how much a node should fire. The sigmoid activation function is applied in this NN. The non-linear smooth gradient outputs either a 0 or 1, stopping the magnification of activations, compared to them in linear functions. The graph plot of this functions activations shows steepness within smaller ranges, producing small changes within that region has significant affects to values of y. However, tendencies can be identified to bring the Y values to the end of the curve. This leads to the vanishing gradient problem, which produces such small enough values, which the network may drastically slow down its learning or stop all together.

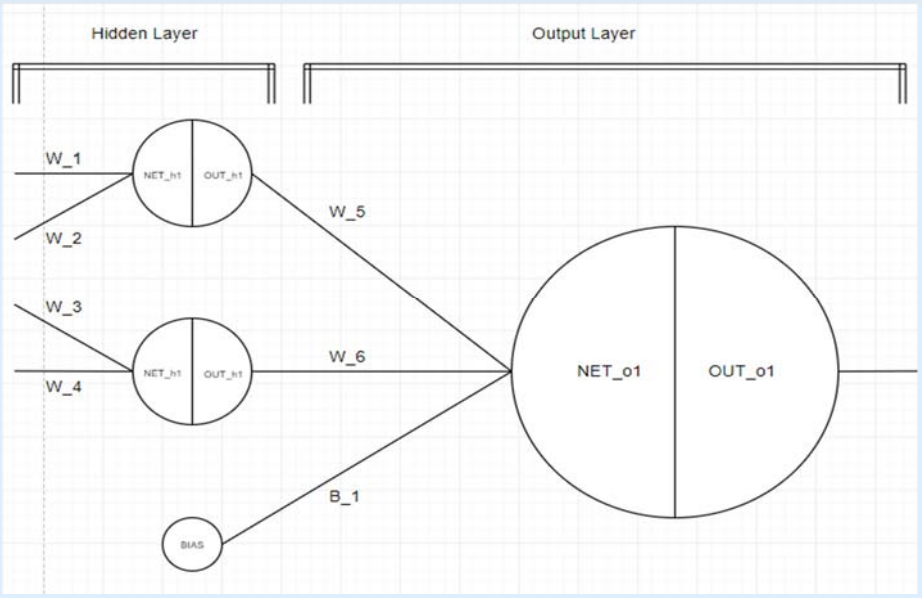


Fig 8.1. The hidden layers connecting into the output node, which represents the net and out properties of each neuron.

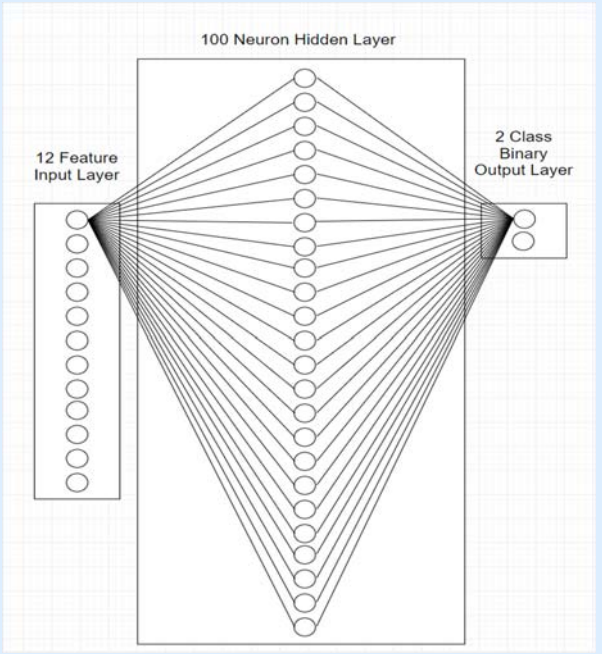


Fig 8.2. Diagram of an Artificial Neural Network. (1) Input Layer is the 12 features from the data set, (2) Hidden Layer of 100 neurons, and (3) Output Layer (Binary Classification: either 0, 1).

Figure 9. Represents the ANN implementation within the keras library, conforming to the briefs specification; including: (1) Using a binary loss function for the classification of two statuses, (2) Using the data samples provided, (3) Has a hidden layer of 100 neurons including a sigmoid activation function, and (4) An output layer with a logistic (sigmoid) function. Using Keras allows building personal networks through the use of their simplistic API, which uses tensorflow backend. Two important components within the keras library, being able to add the number of layers and assigning the number of neurons to that layer, shown within figure 10s code snippet, through the use of add and dense. The input layer is initialised by defining the number of dimensions required from each sample (the 12 features). The hidden layer neurons will activate based on these inputs, through the use of forward propagation using the equations 2-4. The output layer only has 1 output which can be represented through binary cross-entropy function. During each training step a single sample will be passed through the network, allowing forward and backpropagation to calculate their individual parameter changes. Within the keras model, batch sizes allows groups of data to be forwardfeed into the network and the error calculates across the whole batch. Upon batch completion, back propagation updates all parameters within the network, commonly known as mini-batch gradient descent. Investigations into batch sizes have proved that they can have crucial effects on accuracy; two sizes were identified as inefficient, 1 and N (data size). (Radiuk, 2017). Whilst this descent has a different name, it still applies SGD (Stochastic Gradient Descent) on the finished batch result. SGD is a fast approach, but does provide frequent updates with high variance, leading to heavy fluctuations in objective function.

Research was made into optimisation techniques such as hyper-tuning. A grid search was applied to the keras model to optimise the hyper-parameters seen in Figure 10, which outline the parameters required for this task. This method is Sklearn importation that allows the model to exhaustedly search based on the parameters passed into the model. The parameters can be seen being passed into the param_grid dictionary to be used later within the Grid Search CV (Cross Validation), which applies 10 CV to each model search. The application of CV identified concrete parameters for the use in a network as sensitive as nuclear reactor data. The model was tested on: (1) epochs, (2) batches, and (3) optimisers. Each hyper-parameter has its own benefit to the model,

by testing each parameter against every other parameter. The grid search applied 360 model tests on 36 different model configurations. The output research is seen in figure 10, but will be discussed further in the next paragraph. Along with this, an early exit clause has been implemented to analyse the computational cost against the validation accuracy increase. This is a form of regularisation, which, due to its effectiveness and simplicity is commonly used in stopping criteria. The implementation uses the test data as the validation set to comply with the assignments brief; error is measured progressively from validation data accuracy, where comparisons are made to previous batch and epoch errors to identify whether increases were found. Figure 10 shows how it's been implemented on the keras ANN.

Furthermore, research has proven that the min-max scaling normalisation decreases the performance metrics for each ANN classifier. The noticeable difference between original data and scaled showed a 20-28% drop, evidencing that their original scale is important to classification. The features with high values may identify as a reason to why the model does worse with normalised values. These features may show dominance in the network and scaling those values removes their diversity, resulting in worse accuracy. This project also looked into other normalisation techniques to understand their effect, but research concluded the same results for all approaches selected. This may be related to the removal of scales, which destroys the diversity, resulting in less discrimination within training data. Research identified good performance may be related to how each model is able to capture unique parameters and compliment the information within the individual features.

Fig 9.

```
# function for model creation, number of neurons in the hidden layer| optimiser used with the hidden layer
def create_model(nodes=100, optimizer='adam'):
    CLASSIFICATIONS = 1  # number of reactor classes
    LOSS_FUNCTION = 'binary_crossentropy'  # used for categorical classes 2 or more
    HIDDEN_LAYER_ACTIVATION = 'sigmoid'  #
    OUTPUT_LAYER_ACTIVATION = 'sigmoid'  # logistic function
    INPUT_DIMENSION = 12  # number of features input

    # creating model
    model = Sequential()
    # input data into hidden layer
    model.add(Dense(nodes, input_dim=INPUT_DIMENSION, activation=HIDDEN_LAYER_ACTIVATION))
    # output classification layer using logistic activation function
    model.add(Dense(CLASSIFICATIONS, activation=OUTPUT_LAYER_ACTIVATION))

    # compile model
    model.compile(loss=LOSS_FUNCTION, optimizer=optimizer, metrics=['accuracy'])
    return model
```

Fig 10.

```
# creates model
model = KerasClassifier(build_fn=create_model, verbose=0)
# assigns different parameters for the grid search
epochs = np.array([250, 500, 1000, 2000])
nodes = np.array([100])
batches = np.array([128, 256, 512])
optimizer = np.array(['adam', 'RMSprop', 'SGD'])
# assigneds the above parameters to a gride search dict
param_grid = dict(epochs=epochs, batch_size=batches, optimizer=optimizer)
# applies the grid search based on setup parameters
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=10)
grid_result = grid.fit(x_train, y_train)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

Representing the Grid search for keras model of 100 neurons found: Hyper-parameters: {'Nodes': 100, 'batch size': 512, 'epochs': 2000, 'optimizer': 'Adam'}

A single test was applied to ANN of a single hidden layer of 100 neurons. The results showed that on test accuracy it was able to classify 89% of the samples, whilst training showed 100% accuracy. Along with this, sensitivity and specificity calculated to understand the individual class distribution of correct prediction. Sensitivity (the quantity of actual positives that got predicted

positive) showed 88%, proving that 1 in every 9.8 will be miss classified for normal samples. Specificity (the quantity of actual negativities that got predicted negative) showed 90%, proving, 1 in every 10 will be misclassified for abnormal. With consideration to these figures, our networks main focus is to identify when the reactor is running abnormally, which is shown in specificity and shows to have higher correctly classifier samples than normal.

Random Forest Trees

RFT (Random Forest Trees) are a supervised approach to ML. This algorithm is used within this assignment for classification, but can also be used for regression. Initially RFT takes random samples from the original data (around 2/3) which is known as bootstrapping. This can lead to duplications of the samples, however, it allows each tree to be trained on different samples, increasing the local tree variance. On the other hand, this is compensated with lower variance in the entire forest, without the trade-off of increased bias. A by-product, OOB (Out-of-Bag) is created to validate the performance of the decision trees using the remaining 1/3 of the data. The OOB samples are individually passed through each decision tree, on the condition they weren't used in its formation. From the initial bootstrap, a decision tree is formed to classify the data from these random samples. Decision trees are made up of 4 sections, (1) a root node to initialise the tree, (2) 2 branches to represent ranges of values, (3) internal nodes, which represent a certain characteristic of the data set, and (4) a leaf node used to end the chain and represent a classification. (Ali, et al., 2015). N number of random features are chosen and their impurities evaluated, allowing the smallest to be chosen for the root node. Once a feature node has been identified as the most optimal, a range within that feature needs to be calculated. This calculation on numerical attributes includes : (1), order the data lowest to highest; (2), calculate the average for each adjacent value; (3), calculate the impurity value for each of the average weight, and (4) select the lowest weight as this is the cut-off for the best split of data. This procedure is done for each following node, where, random feature subsets are chosen and their impurities evaluated against the impurity of not using that feature to split. For example, Ln (Left node) from the prior split, may have a lower impurity than all of the features chosen to split that sample further, in turn changing it into a leaf node, which represents the class with the highest percentage. Each of these trees estimates the label of passed input features, which allows for bagging (bootstrap aggregation), where each trees output is counted and the highest percent of output class will be chosen as the prediction. RFT will usually have a terminator for each tree node and the model itself, common approaches use (1) A number of minimum samples allowed for continuous splitting of parent node by more features and (2) A N number of trees allowed to populate the forest. Finally, the previously mentioned OOB samples are passed through each decision tree which didn't include their sample, each tree will be used to calculate the MPE (Mean Prediction Error). This approach attempts at removing the need for cross validation because as dataset and N-features grow, CV becomes computational onerous. (James, et al., 2013). In conjunction, a paper considering the effects of OOB, evidenced that MPE shows a pessimistic approach to testing; understood from the limited decision trees used within the bagging process. (Ishwaran, 2015).

Figure 11 represents the code used to create my RFT using the Sklearn library. We split the data into features and labels for both training and testing. This spilt uses 90% of the samples as the training data and 10% as the test. The brief specified the model needed to create 100 decision trees within the model, which is applied through 'n_estimators' parameter. Within the RT classifier, the default impurity calculator is Gini, seen in equation 4. Gini identifies the probability of N number of randomly selected samples being incorrectly classified based on the distribution of samples within the node. This formula looks to achieve a good split point by increasing the decrease in node impurity. As previously mentioned, this will determine whether the feature is optimal against not splitting and which feature is best for splitting and finding the best range within the chosen feature.

However, a grid search is applied to the RTF to identify the best parameters for data which can be seen in figure 12 along with the code used. The code checks all the main hyper parameters within the RF classifier to find the most optimal. These were then applied to try and obtain the highest accuracy with the parameters specified already.

(11) $Gini(E) = 1 - \sum_{j=1}^{c} p_j \log p_j$

Fig 11.

```
# splits the features and labels into train and test data
# randomises the data with random seed
x_train, x_test, y_train, y_test = train_test_split(Scaled_X, Y, test_size=0.10, shuffle=False)
# setting the RFC class for the all the hyperparameters
RFC = RandomForestClassifier(bootstrap=bootstrap, class_weight=None, criterion='gini',
                             max_depth=max_depth, max_features=max_features, max_leaf_nodes=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=min_samples_leaf, min_samples_split=min_samples_split,
                             min_weight_fraction_leaf=0.0, n_estimators=no_trees, n_jobs=None,
                             oob_score=True, random_state=None, verbose=0, warm_start=False)
```

Fig 11. The code for RFT model architecture using the Sklearn library.

Fig 12.

```
    # empty random forest class
    RFC = RandomForestClassifier()
    # ranges for all hyper testing
    n_estimators = [int(x) for x in np.linspace(start=1, stop=2000, num=10)]
    max_features = ['auto', 'sqrt']
    max_depth = [int(x) for x in np.linspace(10, 110, num=11)]
    max_depth.append(None)
    min_samples_split = [1, 2, 5, 10]
    min_samples_leaf = [1, 2, 10]
    bootstrap = [True, False]
    # the grid for all the parameters to be tested
    random_grid = {'n_estimators': n_estimators,
                   'max_features': max_features,
                   'max_depth': max_depth,
                   'min_samples_split': min_samples_split,
                   'min_samples_leaf': min_samples_leaf,
                   'bootstrap': bootstrap}
    # repeated until optimal parameters are found
    # start hyper-tuning
    rf_random = RandomizedSearchCV(estimator=RFC, param_distributions=random_grid, n_iter=100,
                                   cv=10, verbose=2, n_jobs=-1)
    # test data against parameter and runs next
    rf_random.fit(x_train, y_train)
```

The model architecture for RFT. 'n_estimators': 223, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': None, 'bootstrap': False

Two performance tests have been carried out to understand the effects of minimum number of samples per leaf (1 and 10). These tuning parameters allow either >=1 or >=10 samples to be in a leaf node increasing the depth for 1 and decreasing the depth of 10. The results from the min being 1 produced the following performance metrics: (1) Train **100%** (using 90% of the dataset), (2) Test **97%** (using 10% of the dataset) and (3) OOB **92.08%** (approximately using 1/3 train data). As research previously showed the OOB has lower accuracy than test data due to it only using certain trees for classification. However, testing accuracy proved to be high, showing that a model with 100 tree terminate clause and minimum leaf samples at 1 can classify 97/100 correctly. Another test carried out with the same train/test conditions but minimum leaf samples at 10 proved to be worse at predicting. The produced performance metrics were: (1) Train **95.65%**, (2) Test **89%**, and (3) OOB **87.05%.** With minimum samples as 10, results identify abnormality to have a weaker classification than normality, evidenced by 84% for specificity and 94% for sensitivity. Also, specificity at 91% has shown to be less than sensitivity at 93% with minimum leaf samples as 1. This implies that normal samples have a higher classification rate than abnormal. This is a concern between both tests as in reality abnormality needs to have a higher classification, if not the same, as normality. However, the increase in leaf minimum samples shows to increase the performance between 4.35-8%. The

evidence provided, shows that trees with larger depths have greater ability to split the tree further (if impurity allows it), allowing classification to be more accuracy within this problem spaces domain.

Model Selection

For both ANN and RFT 10 fold cross validation has been applied, where the dataset has been split 10% chunks for testing and the other 90% for training. The models will be trained and tested 10 times using different data for both sets. This approach evaluates the datasets and models performance through each fold of data, providing an average accuracy. To allow for fair comparisons the same randomisation seed has been used with reading and balancing the data. The code used has been imported from the Sklearn library and splits the data accordingly, the first 6 folds have 897 train & 99 test samples and the last 4 folds have 898 train & 98 test samples, which compensates for the unequal split of 10 folds. In each iteration of K, train and test are split further into features and labels, seen in figure 13. The parameters identified through grid searching have been used for both models in the final CV performance analysis. Seen in figure 10 for ANN and figure 12 for RFT. The model implementations stem from the initial implementation in figures 9-11; not requiring further evidencing within this section. However, for each model (ANN & RFT) average confusion matrixes are computed from each 10 fold cross validation, for TP and TN evaluation seen in the conclusion.

Fig 13.

```
# initialises the number of folds
kf = KFold(n_splits=10)
# loops through 10 fold cross validations
for train, test in kf.split(DATA_SET):
    # splits the features and labels into train and test data
    train_data = np.array(DATA_SET)[train]
    test_data = np.array(DATA_SET)[test]
    x_train, y_train = train_data[:, 1:13], train_data[:, 0:1]  # takes all the 12 features, takes the label column
    x_test, y_test = test_data[:, 1:13], test_data[:, 0:1]  # takes all the 12 features, takes the label column
```

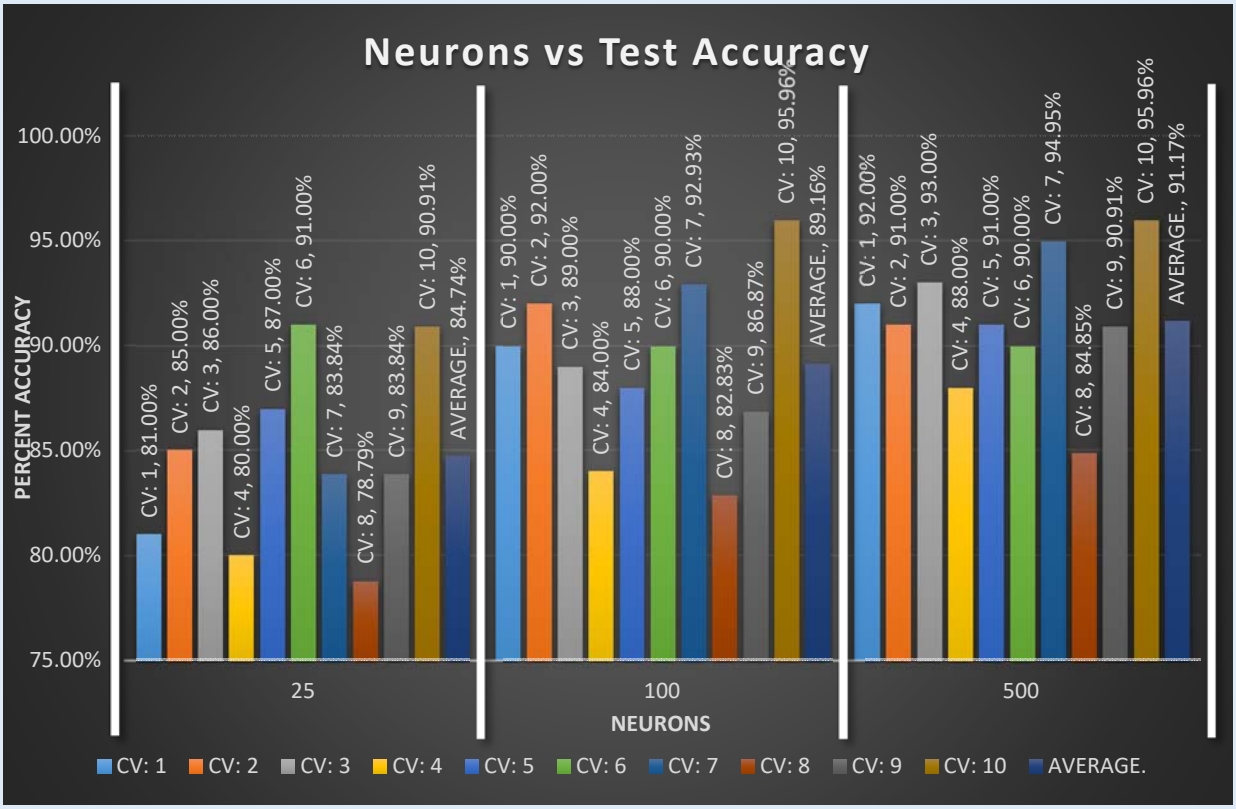
Artificial Neural Network:

Figure 15 shows the each folds test accuracy along with the average between them, which is then applied for 3 different model neurons: (1) 25, (2), 100, and (3) 500. The results prove that the formation of the data does play a significant role on the performance of the models, seen in ranges of peaks in the bar chart. 25 neurons can be seen as the worst classification, followed by 100 and then 500. The model with the lowest test performance, showed to be 7% lower than the best, which shows that the increase in hidden nodes does increases the overall performance of the NN. The cause of this maybe directly related to the number activations available for the classification, allowing more information to be learnt from the training samples. Furthermore, specificity and sensitivity was evaluated for each model, seen in figure 14. The results from the 500 neuron model showed to have higher sensitivity than specificity; having 1% higher misclassifications in abnormal than normal. This could be related to the abnormal data having similar properties to normal, allowing for misclassification. The implementation of this algorithms can be referenced in figure 10 as the same model will be used with just different parameters used for neurons within the CV tests.

Fig 14.

	Sensitivity	Specificity	Train	Test
25 Neurons	85%	84%	94.61%	84.74%
100 Neurons	88%	89%	98.97%	89.16%
500 Neurons	91%	90%	99.94%	91.17%

Fig 15.



Random Forest Trees:

Figure 16 shows the implementation of 10 fold cross validation for the RTF, where they are then passed into the already hyper-tuned Random forest classifier, which showed using 1 as the minimum number of samples at a leaf to be optimal. The train, test and OBB accuracies are stored into a list for print once all 10 iterations are complete. Alongside this, an average confusion matrix was calculated to provide evidence of each models ability to classify each class.

Fig 16.

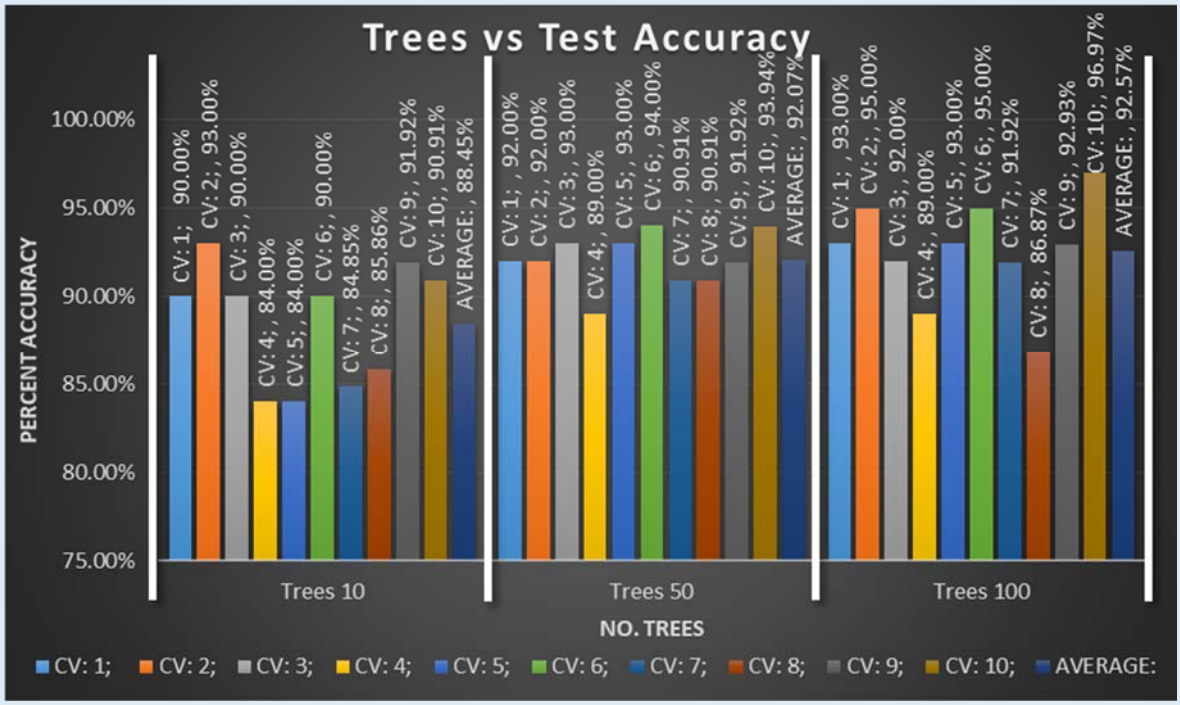
```
# loops through 10 fold cross validations
for train, test in kf.split(DATA_SET):
    train_data = np.array(DATA_SET)[train]  #
    test_data = np.array(DATA_SET)[test]  #
    x_train, y_train = train_data[:, 1:13], train_data[:, 0:1]  # takes all the 12 features | takes the label column
    x_test, y_test = test_data[:, 1:13], test_data[:, 0:1]  # takes all the 12 features | takes the label column
    CV_no += 1  # increments the cv count
    # setting the RFC class for the all the hyperparameters
    RFC = RandomForestClassifier(bootstrap=bootstrap, class_weight=None, criterion='gini',
                                 max_depth=max_depth, max_features=max_features, max_leaf_nodes=None,
                                 min_impurity_decrease=0.0, min_impurity_split=None,
                                 min_samples_leaf=min_samples_leaf, min_samples_split=min_samples_split,
                                 min_weight_fraction_leaf=0.0, n_estimators=no_trees, n_jobs=None,
                                 oob_score=True, random_state=None, verbose=0, warm_start=False)
    # fit the data to the model architecture
    RFC.fit(x_train, y_train.ravel())
    # stores the accuracies for each cross validation for train and test
    train_accuracy_cv.append(RFC.score(x_train, y_train))
    test_accuracy_cv.append(RFC.score(x_test, y_test))
    oob_accuracy_cv.append(RFC.oob_score_)
```

Showing the code used to implement a K-fold CV for Random Forest Trees

Figure 17 plots a bar chart with the test iteration in each fold and the average within each hype-parameter (Trees: 10, 50 and 100). Results are detailed in figure 13 but will now be discussed further. The results identified 10 trees to be least effective model parameter, Showing the following performance metrics: (1) Train 99.61%, (2) Test 88.45%, and (3) OOB 84.08% (Warning from Sklearn providing evidence the OOB didn't have enough trees to reliably score OOB). Using 50 trees showed

the following performance metrics: (1) Train 100%, (2) Test 92.07%, and (3) OOB 90.7%. The final parameter of 100 trees showed the following performance metrics: (1) Train 100%, (2) Test 92.57%, and (3) OOB 91.7%. The comparison between each tree shows that an increasing the number of trees results in an increase in performance. This shows that increasing trees, increases the probability that certain samples or features will be used in the bootstrap or split each node, leading to higher performances because more classifications can be made accurately. When considerations were made to model specificity and sensitivity performance for 100 trees (figure 18), results showed to have to have lower true negative (Specificity) than true positive (Sensitivity). This means that the model is 2% more likely to classify abnormalities as normalities than normalities as abnormalities. This causes issues within this problem space, because misclassification of abnormalities can cause fatalities within the reactor.

Fig 17



Showing the results of 10 fold CV on random forest trees

Fig 18.

	Sensitivity	Specificity	Train	Test
10 Trees	88%	90%	99.61%	88.45%
50 Trees	93%	90%	100%	92.07%
100 Trees	93%	91%	100%	92.57%

Conclusion for Model Selection

Reactors within nuclear power satiations play a critical role in today society. These same reactors also need continuous monitoring and evaluating to ensure safety status stay within normal ranges. The normal and abnormal samples given for this assignment have proven to be classifiable, and achieved high performance accuracies within both artificial models. However, the data samples when restructured have the ability to provide low to high ranges within both models, seen in figures 15-17. This being said, a neural networks algorithm has the ability to mode linear and nonlinear relations, but shows to have lower accuracies as a whole compared to the lowest RFT. Alongside this, the complexity of a neural network has the ability to be over sensitive to inputs, leading to higher risks deviations. Furthermore, the time and cost complexity of each model have clear comparisons; NNs have increased time and cost complexities compared to that of a RFT. This can be seen through pre-processing and training, for example requiring extra time normalising and encoding. However, RFT has comparatively low training speeds and no need to normalise data,

resulting in increased algorithm completion. Within NN testing, using a 500 neuron model outperformed the 25 and 100 neurons; having higher overall performance and higher specificity over sensitivity, required in this classification task. The random forest tree hyper-tuning showed higher performances in 100 decision tree ensemble, but had higher sensitivity than specificity compared to ANN. Overall, ANNs had slightly worse performance when compared to the lowest performing model within RFT.

References

Acuna, E. & Rodriguez , C., 2019. *On detection of outliers and their effect in supervised classification,* Puerto Rico: ResearchGate.

Alexandropoulos, S., Kotsiantis, S. & Vrahatis, M., 2019. Data preprocessing in predictive data mining.. *The Knowledge Engineering Review,* 34(10).

Ali, J., Khan, R., Ahmad, N. & Maqsood, I., 2015. Random Forests and Decision Trees. *International Journal of Computer Science,* 9(5).

Ishwaran, H., 2015. The Effect of Splitting on Random Forests. *Journal of Machine Learning,* Volume 1, pp. 75-118.

James, G., Witten, D., Hastie, T. & Tibshirani, R., 2013. *An Introduction to Statistical Learning with Applications in R.* London: Springer.

Radiuk, P. M., 2017. Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets. *Information Technology and Management Science,* 20(3), pp. 20-24.

SEGER, C., 2018. *An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing.* [Online]
Available at: http://www.diva-portal.se/smash/get/diva2:1259073/FULLTEXT01.pdf
[Accessed 19 04 2019].

Singh, B. K., Verma, K. & Thoke, A. S., 2015. Investigations on Impact of Feature Normalization Techniques on Classifier's Performance in Breast Tumor Classification. *International Journal of Computer Applications,* 116(19).

Wei, Q. & Dunbrack Jr , R. L., 2013. *The Role of Balanced Training and Testing Data Sets for Binary Classifiers in Bioinformatics.* [Online]
Available at: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0067863#abstract0
[Accessed 19 04 2019].

Wei, Q. & Dunbrack Jr, R. L., 2013. The Role of Balanced Training and Testing Data Sets for Binary Classifiers in Bioinformatics. *PLOS ONE,* 8(7).