Task 1

Section 1: Data import, summary, pre-processing and visualisation

Shown below is the initial importation of the file before being converted to a DataFrame. The pandas functions linked to this container will be useful for future data manipulation, this is then passed into the summary function.

```
#Importing
nuclear_inp = pd.read_csv('CMP3744M_ADM_Assignment 2-dataset-nuclear_plants.csv', index_col=None)
#pd.read_csv('CMP3744M_ADM_Assignment 2-dataset-nuclear_plants.csv', index_col=None)
nuclear_inp_df = pd.DataFrame(nuclear_inp)

#Section 1: Summary
(c,r) = summary(nuclear_inp_df)

#Section 1: Visualising
statpow1 = pd.DataFrame(nuclear_inp_df.iloc[:,0:2])# columns = nuclear_inp_df.groupby(["Status"]))
statpress1 = nuclear_inp_df[['Status', 'Pressure _sensor_1']].copy()
visualise(statpow1, statpress1)
```

Initially, the summary function calculated all the necessary values individually but the information can be more easily displayed using the describe function, this was then transposed to make the data more readable and compact. The data was also tested for missing values, the DataFrame shape and any categorical features, output is shown below code:

```
def summary(input_d):
    mean = input_d.mean()
    std = input_d.std()
    minim = input_d.min()
    maxim = input_d.max()

    cols = input_d.shape[1] #len(input_d.columns)
    rows = input_d.shape[0]
    dty = input_d.dtypes

    #print("Means:")
    #print(mean)
    #print("stds:")
    print("Data Summary:")
    #print(input_d.describe(include = 'all'))
    desc = input_d.describe().transpose()
    print(desc)
    print("Empty values: {}.".format(input_d.isnull().values.any()))
    print("Shape of the data: columns = {}, rows = {}".format(cols, rows))
    #print(input_d.select_dtypes(exclude=["number","bool_","object_"]))
    #nuclear_inp_df.select_dtypes(include=["category"])
    #summar = pd.DataFrame(mean, std, minim, maxim)
    #print(mean, std, minim, maxim)
    #print(cols, rows)
    #print(input_d.dtypes)
    cols2 = input_d.columns
    num_cols = input_d._get_numeric_data().columns
    print ("Categorical Features: {}".format(list(set(cols2) - set(num_cols))))
    return cols, rows
```

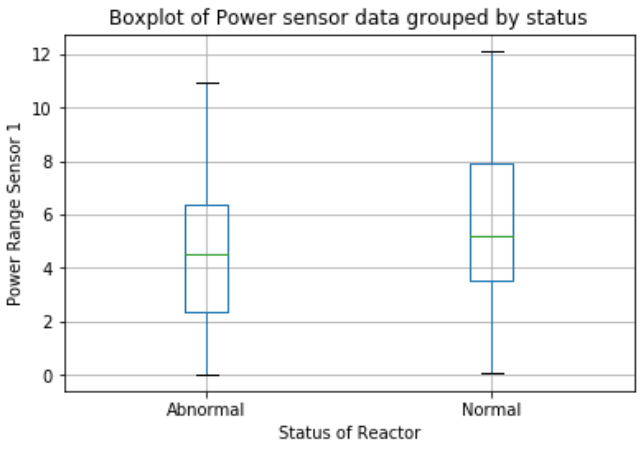
The only data returned is the shape of the DataFrame stored as columns and rows. The host code is returned to and the DataFrame is passed to the visualise function to be displayed.

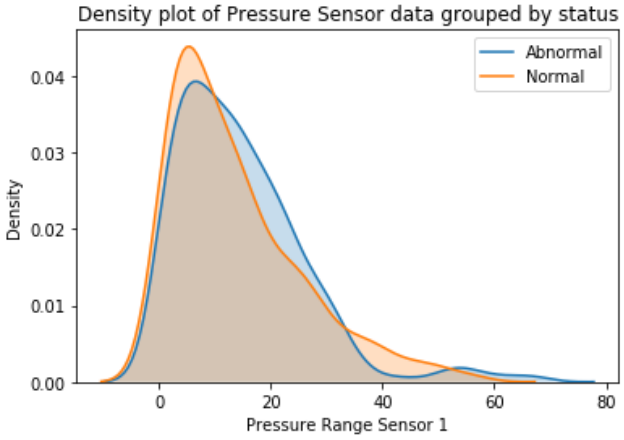
```
def visualise(boxd, dens):
    #nuclear_inp_df.groupby(["Status"]).size()
    #t2 = t.groupby(["Status"])
    #print(t2)
    #t.plot.box(by='Status')
    #plt.set_title('Boxplot Status')
    #t2.plot.box()
    #plt.plot(stat, power_sens_1)
    #t.groupby('Status').size().plot(kind='bar')
    #nuclear_inp_df.boxplot(column="Status")
    #t.plot(x='Status',y = 'Power_range_sensor_1', kind='box')
    #plt.xlabel("test")
    #nuclear_inp_df.boxplot(column="Status")

    plt.figure()
    #p1 = plt.subplot(211)
    bplot = boxd.boxplot(by = 'Status')
    plt.title("Boxplot of Power sensor data grouped by status")
    plt.suptitle("")
    bplot.set_xlabel("Status of Reactor")
    bplot.set_ylabel("Power Range Sensor 1")
    plt.show()


    plt.figure()
    #p2 = plt.subplot(212)
    abnorm = dens[dens.Status == "Abnormal"]
    norm = dens[dens.Status == "Normal"]
    plt.title("Density plot of Pressure Sensor data grouped by status")
    dplot = sbn.kdeplot(abnorm['Pressure _sensor_1'], shade=True, label="Abnormal")
    dplot = sbn.kdeplot(norm['Pressure _sensor_1'], shade=True, label="Normal")
    dplot.set_xlabel("Pressure Range Sensor 1")
    dplot.set_ylabel("Density")
    plt.show()

    #plt.hist(press_sens_1, density = True, histtype = 'step')
    #press_sens_1_df = pd.DataFrame(press_sens_1)
    #press_sens_1.plot.kde()
    #for ti, gr in dens.groupby('Status'):
    #dplot = dens.plot.density()
    #dplot = dens.hist(by = 'Status', sharex=True, sharey=True)
    #for d in dplot.flatten():
```

The outputs for both of these functions are then visualised below:

	count	mean	std	min	25%	50%	75%	max
Status	996.0	0.500000	0.500251	0.000000	0.000000	0.500000	1.000000	1.000000
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

```
Empty values: False.
Shape of the data: columns = 13, rows = 996
Categorical Features: ['Status']
```





This data summary and visualisation shows that the data is complete and contains some interesting values. This includes the spread of the data displayed in the power sensor boxplot, normal status data has a wider range than the abnormal data, the mean is very similar but the quartiles are very different. The max values of both Pressure_sensor_1 and Temperature_sensor_3 being much higher than the other pressure sensors is also of interest as this could mean they could heavily bias information within the models (for example the weighted sum with a ANN) and could mean normalising is needed to solve this. Further review of the density plot shows

that this may not be an issue as both categories within the feature have values within at that value. The density plot does show that specifically for the pressure sensor data, abnormal data is only very rarely within the values of 35 to 45.

Section 2: Discussion on selecting an algorithm

When discussing the process chosen by student and the conclusion given, several things must be discussed. This discussion includes the initial split of data, the training set of 70% is considered a normal size (Bronshtein, 2017) but testing using a size of only 10% may not have been viable without a further 20% for validation, ensuring that the data doesn't suffer from overfitting.

When checking the different models and recording their accuracy, a number of other factors within each classifier would need to be manipulated and tested to allow for a better representation of the best accuracy of each model. Examples of this is include learning rate for an ANN or number of trees for Random Forest Classifiers, both have a large effect on accuracy and changing them can drastically effect results, because it is not explicitly stated that these factors were also being changed, the student may have just used the default values and received vastly underperforming results.

Recording the accuracy of each model says nothing about the sensitivity and resiliency they have. The chosen model could be 100% accurate to the data given but unable to accurately define outliers and new data as normal or abnormal.

Overall, for an incredibly vital and potentially destructive system such as nuclear safety, 90% accuracy would not be considered good enough as a model on its own, the use of separate models, fail safes or other security measures may make this a viable option however. The speed of computation would also need to be taken into account to allow these additional systems or the operator time to validate the output of the model.

Section 3: Designing algorithms

To initially randomise the data, the DataFrame is passed into a randomising function that simply shuffles the data before splitting it into the percentage data asked for in the brief.

```
#Section 3: Splitting Data
(out_tr, out_te) = randomsplit(nuclear_inp_df)
```

```
def randomsplit(input_d): #Randomises which part of the input file will be used for training or testing

    #First convert Status from string to int/bool
    input_d['Status'] = (input_d['Status'] != 'Abnormal').astype(int)

    #Then Split data 90:10 for training and test
    length = len(input_d)
    output = input_d.sample(frac=1)
    ninety = int(round(0.9*(length)))
    output_train = output[0:ninety]
    output_test = output[ninety:length]
    return output_train, output_test
```

The status feature of the data set is also converted to Boolean integers at this time so that the values can work with the classifiers being used.

```
#Section 3: ANN
ann1 = artneurnet(r, O_nodes, Max_Iters, L_Rate)
ANN_error_tot = ann1.process(out_tr, out_te, H_nodes, True, ANN_error_tot)
```

As shown above, an object is created to facilitate the initial ANN. This object will initialise and then pass-through data into a Multi-Layer Perceptron Classifier, this is initially defined with a number of factors set out such as using the sigmoid activation function defined as 'logistic' within the library function, these will be discussed in more detail below.

Once defined the training portion of the data is then used for fitting the classifier to the data using backpropagation. The feed forward portion is carried out and initially random weights will be chosen for each input node to hidden node connection. Each hidden node then performs a weighted sum where the value within each input node connected to the hidden node is multiplied by the weight assigned to the connection and then added together before being passed through an activation function. As discussed before the activation function used is the sigmoid function, this maps the weighted sum values onto the range of 0 to 1 to reduce the bounds of the data to something manageable, the equation used for this is:

$$f(x) = \frac{1}{1 + e^{-x}}$$

These values will then be stored in a matrix and added to the bias, this bias is to ensure that when classifying the values there can' be a zero value achieved. The steps leading to this matrix from the input to hidden nodes will be repeated again from the hidden to output nodes with new random weights and a bias added. This leads into backpropagation and examining the output and tuning weights based on previously acquired error/cost while using this algorithm. Error is calculated from the difference between the outputted value given by this iteration of the neural network and the actual classification labels given, in this case status. The Backward pass of Backpropagation is then used to calculate the error given from any neuron in the network and update weights accordingly to reduce total error. This is done by calculating the derivative of the error total with respect to each individual weight, this uses the chain rule to calculate this error as shown below:

$$\frac{\partial E_{total}}{\partial w1} = \overset{\mathbf{a}}{\frac{\partial E_{total}}{\partial out_{h1}}} \times \overset{\mathbf{b}}{\frac{\partial out_{h1}}{\partial net_{h1}}} \times \overset{\mathbf{c}}{\frac{\partial net_{h1}}{\partial w1}}$$

(Leontidis, 2019)

Once the weights are updated, the stages are iterated until convergence or other stopping criteria are reached.

```
class artneurnet:
    #Sigmoid: 1/1+e^-z = 1/1+e^(- sum(wi*xi-b))
    def __init__(self, input_nodes, output_nodes, maximum_iterations, learning_rate):
        print("-- Initialising ANN --")
        self.inpn = input_nodes
        self.outn = output_nodes
        self.maxi = maximum_iterations
        self.lear = learning_rate

    def process(self, train, test, hidden_nodes, disp, total_accuracy):
        if (disp == True):
            print("Artificial Neural network: ")
        network = MLPClassifier(hidden_layer_sizes = hidden_nodes, activation = 'logistic', learning_rate_init = self.lear, max_iter = self.maxi)
        train_labels = train.iloc[:,0]
        train_samp = pd.DataFrame(train.iloc[:,1:13])
        test_labels = test.iloc[:,0]
        test_samp = pd.DataFrame(test.iloc[:,1:13])
        network.fit(train_samp, train_labels)
        train_prediction =  network.predict(train_samp)
        test_prediction = network.predict(test_samp)
        #print(network.score(test_labels,test_prediction))
        train_acc = accuracy_score(train_labels, train_prediction)
        test_acc = accuracy_score(test_labels, test_prediction)
        conf_mat = confusion_matrix(test_labels, test_prediction)
        if (disp == True):
            print("Number of hidden nodes: {}".format(hidden_nodes))
            print("Training Accuracy: {}".format(train_acc))
            print("Testing Accuracy: {}".format(test_acc))
            print("Confusion Matrix: {} , {}".format(conf_mat[0], conf_mat[1]))
            return total_accuracy
        else:
            total_accuracy[0] = total_accuracy[0] + train_acc
            total_accuracy[1] = total_accuracy[1] + test_acc
            return total_accuracy
```

A second object was then created to work with the random forests, random forests are built from decision trees and again using the library involves defining the classifier and then applying it to the data set. The data set is initially bootstrapped, where a sample of the original dataset is randomly reallocated to a new dataset, the same row of data can be added more than once. From this dataset several decision trees are then created to model all possible combinations of features. We have previously selected the number of decision trees to be one hundred, this is to attempt to get as full a coverage of the sample data chosen within the bootstrapped dataset as possible.

The minimum number of samples required for a leaf node was also initially chosen to be one and ten for these tests, this states that when splitting the data within decision trees each split must have at least one or ten samples to finish that route.

Once all decision trees are created, the test data can then be used for predictions. The features are fed into the random forest and in this case the status is determined based on the number of decision trees that support the outcome of normal or abnormal, this is completed for the whole test set and the outcomes can then be compared to the actual results, providing the amount of error within the accuracy score function and receiving the true negative, false positive, false negative and true positive values respectively from the confusion matrix function.

```
class randforests:
    def __init__(self):#, estim):
        print("-- Initialising Random Forest --")
        self.meanacc = []
        #self.num_estim = estim

    def classify(self, minsamp, train, test, num_estim, disp, totacc):
        if (disp == True):
            print("Random Forest Classifier: ")
        forest = RandomForestClassifier(n_estimators=num_estim, min_samples_leaf = minsamp)
        #print(train)
        train_labels = train.iloc[:,0]
        train_samp = pd.DataFrame(train.iloc[:,1:13])
        test_labels = test.iloc[:,0]
        test_samp = pd.DataFrame(test.iloc[:,1:13])
        forest.fit(train_samp, train_labels)
        importance = forest.feature_importances_
        train_prediction =  forest.predict(train_samp)
        test_prediction  = forest.predict(test_samp)
        #print(prediction)
        #forest.score(prediction, test_labels)
        train_acc = accuracy_score(train_labels, train_prediction)
        test_acc = accuracy_score(test_labels, test_prediction)
        conf_mat = confusion_matrix(test_labels, test_prediction)
        #tn, fp, fn, tp = confusion_matrix(test_labels, test_prediction)
        if (disp == True):
            print("Min number of samples for leaf node: {}".format(minsamp))
            print("Training Accuracy: {}".format(train_acc))
            print("Testing Accuracy: {}".format(test_acc))
            print("Confusion Matrix: {} , {}".format(conf_mat[0], conf_mat[1]))
            #print("Importance of each feature: {}".format(importance))
            #print(tn, fp, fn, tp)
            return totacc
        else:
            #self.meanacc.append(test_acc)
            totacc[0] = totacc[0] + train_acc
            totacc[1] = totacc[1] + test_acc
            return totacc
```

Results for initial tests are shown below:

```
-- Initialising ANN --
Artificial Neural network:
Number of hidden nodes: 100
Training Accuracy: 0.943080357143
Testing Accuracy: 0.83
Confusion Matrix: [44 13] , [ 4 39]
-- Initialising Random Forest --
Random Forest Classifier:
Min number of samples for leaf node: 1
Training Accuracy: 1.0
Testing Accuracy: 0.97
Confusion Matrix: [55  2] , [ 1 42]
---------------------
Random Forest Classifier:
Min number of samples for leaf node: 10
Training Accuracy: 0.946428571429
Testing Accuracy: 0.93
Confusion Matrix: [54  3] , [ 4 39]
---------------------
```

Both models rely heavily on the randomising being in an order that is highly split between the two status classes for best results, if a portion of the data is heavily of type normal or abnormal then issues can occur. Attempts to rectify this will be completed below with cross validation.

Section 4: Model selection

The use of cross validation allows verification that the training set has completed its task effectively and overfitting has not occurred. K-fold splits randomised data into several closely sized chunks which are then used sequentially for with both the ANN and the random forest classifiers.

```
kf =  KFold(n_splits=KFolds, shuffle = True)
for i in range (cvhl.size):
    for traini, testi in kf.split(nuclear_inp_df):
        #print(traini)
        train = pd.DataFrame(nuclear_inp_df, traini)
        test = pd.DataFrame(nuclear_inp_df, testi)

        ANN_error_tot = ann1.process(train, test, cvhl[i], False, ANN_error_tot)
        #print("----------------------")
        RF_error_tot = rfc1.classify(1, train, test,cvnt[i], False, RF_error_tot) #Default is 1
        #print(ann1.meanacc)

    print("Artificial Neural network: ")
    print("Number of hidden nodes: {}".format(cvhl[i]))
    print("Average Training error: {}".format(ANN_error_tot[0]/10))
    print("Average Testing error: {}".format(ANN_error_tot[1]/10))
    print("----------------------")
    print("Random Forest Classifier: ")
    print("Min number of samples for leaf node: {}".format(cvnt[i]))
    print("Average Training error: {}".format(RF_error_tot[0]/10))
    print("Average Testing error: {}".format(RF_error_tot[1]/10))
    print("----------------------")
    ANN_error_tot[0] = 0
    ANN_error_tot[1] = 0
    RF_error_tot[0] = 0
    RF_error_tot[1] = 0
```

The total error is passed back from both functions and then displayed as shown below:

```
--------------------
Cross Validation
--------------------
Artificial Neural network:
Number of hidden nodes: 25
Average Training Accuracy: 0.920798868251
Average Testing Accuracy: 0.959434343434
--------------------
Random Forest Classifier:
Number of Trees: 10
Average Training Accuracy: 0.995648814501
Average Testing Accuracy: 0.908646464646
--------------------
Artificial Neural network:
Number of hidden nodes: 100
Average Training Accuracy: 0.927714778428
Average Testing Accuracy: 0.865515151515
--------------------
Random Forest Classifier:
Number of Trees: 50
Average Training Accuracy: 1.0
Average Testing Accuracy: 0.921686868687
--------------------
Artificial Neural network:
Number of hidden nodes: 500
Average Training Accuracy: 0.751780986224
Average Testing Accuracy: 0.734797979798
--------------------
Random Forest Classifier:
Number of Trees: 100
Average Training Accuracy: 1.0
Average Testing Accuracy: 0.919636363636
--------------------
```

Given the results shown, the best model given from this battery of tests is to use the multi-layer classifier with 25 hidden nodes, this achieved the best accuracy in testing. The initial learning rate for all the ANN test data was set to 0.001 but could have been manipulated to get even more accurate results, this is similar to the minimum number of samples needed to be at a leaf node as the default for the library used was 1 but better results may have been higher.

References:

Bishop, C.M. (2006) *Pattern recognition and machine learning*. Oxford: Springer

Bronshtein, A. (2017) *Train/Test Split and Cross Validation in Python*. Available from https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6 [accessed 26 April 2019]

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. Cambridge, Massachusetts: The MIT Press

Han, J., Kamber, M. and Pei, J. (2012) *Data Mining: Concepts and Techniques*. Amsterdam: Morgan Kaufmann

Leontidis, G. (2019) *Week 27: Model Selection & Artificial Neural Networks Principles* [lecture]. Algorithms for Data Mining CMP3744M-1819, University of Lincoln, 28 March. Available from https://blackboard.lincoln.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_133100_1&content_id=_2201593_1 [accessed 30 April 2019]