CMP3751M - Machine Learning

Assessment 2 - Written Report

Shangyuan Liu - 25344136
University of Lincoln, School of Computer Science
25344136@students.lincoln.ac.uk

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

Load the Data

```
def import_data():
    # Import the dataset by read_excel function
    df = pd.read_excel('clinical_dataset.xlsx', engine='openpyxl')
    print("The original dataset: \n{0}".format(df))  # Formatted output
    data = df.drop(['Status'], axis=1)  # Drop the Status list
    return df, data
```

```
The original dataset:
     Age        BMI  Glucose  ...   Resistin       MCP.1     Status
0     52  26.543128       83  ...  10.084147  416.257761    healthy
1     41  23.527236       74  ...   7.482242  274.366161    healthy
2     40  23.278413       73  ...   7.267574  262.659535    healthy
3     27  19.535749       63  ...   4.038659   86.574756    healthy
4     32  20.839655       66  ...   5.163580  147.920865    healthy
..   ...        ...      ...  ...        ...         ...        ...
122   45  26.850000       92  ...  10.960000  268.230000  cancerous
123   62  26.840000      100  ...   7.320000  330.160000  cancerous
124   65  32.050000       97  ...  10.330000  314.050000  cancerous
125   72  25.590000       82  ...   3.270000  392.460000  cancerous
126   86  27.180000      138  ...   4.350000   90.090000  cancerous

[127 rows x 10 columns]
```

Figure 1: Function import_data() - Load the dataset

Function import_data is used to load the data. The approach of the function is importing Pandas, which is a data analysis package for python providing functions and methods to manipulate data quickly and easily. And use read_excel to an Excel file into a DataFrame, the original dataset size is 127 rows × 10 cols.

Summary of the Dataset

```
def statistics(dataset, name):
    """Calculate a summary of the dataset"""
    mean_value = ("%.2f" % np.mean(dataset))  # Calculate the mean value of array
    std_value = ("%.2f" % np.std(dataset))  # Calculate the standard deviation value
    min_value = ("%.2f" % np.min(dataset))  # Calculate the minimum value of array
    max_value = ("%.2f" % np.max(dataset))  # Calculate the maximum value of array
    # Display the summary of statistics
    print("\n———— The summary statistics of the feature %s ————" % name)
    print("The mean value of %s is: %s" % (name, mean_value))
    print("The standard deviation value of %s is: %s" % (name, std_value))
    print("The minimum value of %s is: %s" % (name, min_value))
    print("The maximum value of %s is: %s" % (name, max_value))
    return
```

Figure 2: Function statistics() - Statistical summary of data set

This function statistics() (figure 2) provides a summary (mean, standard deviations, min/max values) of each feature in the dataset. Using the mean(), max(), and min() methods in the numpy library in the function to provide a summary. And the result of the summary is shown in Figure 3.

```
———— The summary statistics of the feature Age ————
The mean value of Age is: 55.85
The standard deviation value of Age is: 16.29
The minimum value of Age is: 24.00
The maximum value of Age is: 89.00

———— The summary statistics of the feature BMI ————
The mean value of BMI is: 27.22
The standard deviation value of BMI is: 4.99
The minimum value of BMI is: 18.37
The maximum value of BMI is: 38.58
```

```
———— The summary statistics of the feature Leptin ————
The mean value of Leptin is: 25.62
The standard deviation value of Leptin is: 18.63
The minimum value of Leptin is: 4.31
The maximum value of Leptin is: 90.28

———— The summary statistics of the feature Adiponectin ————
The mean value of Adiponectin is: 9.83
The standard deviation value of Adiponectin is: 6.65
The minimum value of Adiponectin is: 1.66
The maximum value of Adiponectin is: 38.04
```

Figure 3: The result of statistical summary

Report the Data Size and Features

```
# Report the size and features
row_data = np.shape(df)[0]  # Rows
col_data = np.shape(df)[1]  # Columns
print("\nThe size of the dataset: %d × %d\nThe number of features: %d\n"
      % (row_data, col_data - 1, col_data - 1))
```

```
The size of the dataset: 127 × 9
The number of features: 9
```

Figure 4: Report the size of the data and number of features

The np.shape method is used in this step, which is to report the dimension of a matrix or array. Shape()[0] is for counting the number of rows, and Shape()[1] is for counting the number of cols. As can be seen from figure 4 that the size of data is 127 rows × 9 cols, there are 9 clinical features that can classify patients as healthy or cancerous, and there is another feature as a prediction label.

Find Missing Values

```
def find_missing_value(dataset):
    """Find missing values"""
    missing_value = dataset.isnull().any()  # Determining "columns"
    print("The feedback of missing values\n", missing_value)
    return
```

Figure 5: Function find_missing_value and feedback

```
The results of missing values
 Age             False
BMI              False
Glucose          False
Insulin          False
HOMA             False
Leptin           False
Adiponectin      False
Resistin         False
MCP.1            False
Status           False
```

In the pandas library, there is an effective function isnull() , which can be used to identify missing values. And isnull().any() will determine which columns contain missing values, and return True if there are missing values in the column, otherwise False. Therefore, as shown in the figure on the right, there is no missing value in this dataset.

Find Categorical Variables

There are any categorical variables in the dataset, the categorical variables are 'cancerous' and 'healthy' respectively in the 'Status' list It is clear from figure 6 that value_counts() is a method to check how many different variables are in a column of the table, and then use the list method to store them in the list.

```
# Find categorical variables in the dataset
list_categories = list(df['Status'].value_counts().index)
print("\nThere are categorical variables in the Status feature:", list_categories)
status_dummy = pd.get_dummies(df['Status'], drop_first=False, prefix='Status')
print(status_dummy)
```

```
There are categorical variables in the Status feature: ['cancerous', 'healthy']
     Status_cancerous  Status_healthy
0                   0               1
1                   0               1
2                   0               1
3                   0               1
4                   0               1
..                ...             ...
122                 1               0
123                 1               0
124                 1               0
125                 1               0
126                 1               0
```

Figure 6: Find categorical variables

Finally, using function get_dummies() for one-hot encoding to convert categorical variables into dummy/indicator variables, and the result of the change in the status column after the one-hot code is applied is as shown in figure 6.

Data Normalisation

```python
def z_score(dataset):
    """Z-score Normalization"""
    # The standardised data is normally distributed with mean 0 and variance 1
    avg = np.mean(dataset)  # Calculate the mean value
    std = np.std(dataset)  # Calculate the standard deviation value
    data_z_score = (dataset - avg) / std  # Z-Score
    return data_z_score
```

```python
def max_min_normalization(dataset):
    """[0,1] Normalization"""
    data_normal = (dataset - np.min(dataset)) / (np.max(dataset) - np.min(dataset))
    return data_normal
```

```python
# Normalise the data before starting training/testing any model
print("\nZ-Score Normalization Method:\n", z_score(data))  # By Z-Score method
print("\nMin-Max Normalization Method:\n", max_min_normalization(data))  # By Min-Max Normalization method
```

Figure 7: Functions z_score() and max_min_normalization() to data normalization

$$z = \frac{x - \mu}{\sigma} \quad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \qquad z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Figure 8: z-score and max_min_normalization formulas

In my opinion, it is really necessary to have a data normalisation processing for machine learning models. Sola and Sevilla (1997) indicated that "An adequate normalization, not only for the network output variables but also for the input ones, previous to the training process is very important to obtain good results and to reduce significantly calculation time". Therefore, it is significant to normalize the data in order to overcome the problem of model learning. Moreover, the model ensures that a similar range of values is used for different features so that gradient descent can converge more quickly.

There are two approaches used for data normalization, which are Z-Score and Min-Max normalisation respectively. The formulas are displayed in figure 8 and implemented in the functions z_score and max_min_normalisation respectively (as shown in figure 7). The Min-Max normalisation is a linear transformation of the original data to map the values to between [0, 1]. However, the Z-Score is based on the mean and standard deviation of the original data to standardise the data, the main purpose is to standardise data of different magnitudes to the same magnitude, using the calculated Z-Score value to ensure comparability between data. The results of the data normalisation are displayed in figure 9.

Z-Score Normalization Method:

	Age	BMI	Glucose	...	Adiponectin	Resistin	MCP.1
0	-0.236404	-0.134822	-0.561799	...	-0.150288	-0.333637	-0.279922
1	-0.911776	-0.739353	-0.959900	...	-0.532929	-0.550588	-0.697436
2	-0.973174	-0.789230	-1.004134	...	-0.564498	-0.568487	-0.731883
3	-1.771341	-1.539442	-1.446468	...	-1.039348	-0.837720	-1.250009
4	-1.464353	-1.278076	-1.313768	...	-0.873916	-0.743922	-1.069499
..	...	...	...	...	...	...	...
122	-0.666187	-0.073310	-0.163699	...	0.341437	-0.260607	-0.715492
123	0.377570	-0.075314	0.190169	...	1.743924	-0.564116	-0.533264
124	0.561763	0.969024	0.057469	...	1.912464	-0.313137	-0.580667
125	0.991545	-0.325875	-0.606033	...	3.599361	-0.901812	-0.349947
126	1.851109	-0.007162	1.871039	...	0.643905	-0.811760	-1.239666

Min-Max Normalization Method:

	Age	BMI	Glucose	...	Adiponectin	Resistin	MCP.1
0	0.430769	0.404435	0.163121	...	0.197238	0.087136	0.224141
1	0.261538	0.255198	0.099291	...	0.127351	0.054154	0.138281
2	0.246154	0.242885	0.092199	...	0.121585	0.051433	0.131197
3	0.046154	0.057685	0.021277	...	0.034856	0.010504	0.024647
4	0.123077	0.122207	0.042553	...	0.065071	0.024763	0.061768
..	...	...	...	...	...	...	...
122	0.323077	0.419620	0.226950	...	0.287049	0.098238	0.134568
123	0.584615	0.419125	0.283688	...	0.543206	0.052098	0.172043
124	0.630769	0.676934	0.262411	...	0.573988	0.090252	0.162294
125	0.738462	0.357271	0.156028	...	0.882091	0.000761	0.209741
126	0.953846	0.435950	0.553191	...	0.342293	0.014451	0.026774

Figure 9: The result of two methods for data normalisation

Data Visualisation

Seaborn is used in this function (shown in figure 10), which is a Python data visualization library based on Matplotlib. The boxplot and density plot are grouped by the categorical variables of 'Status', and these plots can be seen in the figure 11.

The boxplot provides significant information about the location and dispersion of the data, especially

Figure 10: Function box_density_plots when comparing different parent data. As can be seen in figure 11 in the x-axis contains the two classes (Cancerous and Healthy). Each boxplot contains six main data nodes. The Age data are arranged from largest to smallest and its upper edge, upper quartile, mean value, median, lower quartile, and lower edge are calculated.

```python
def box_density_plots(df):
    """Plot the box and density plots"""
    # Plot the box plot between Status and Age
    df.boxplot('Age', 'Status', notch=False, grid=True, meanline=True, showmeans=True,
               boxprops={'color': 'black', 'linewidth': '2.0'},
               capprops={'color': 'red', 'linewidth': '2.0'},
               flierprops={'marker': '*', 'markerfacecolor': 'red', 'color': '654EA3'},
               meanprops={'marker': 'o', 'markerfacecolor': 'blue'},  # Set mean value point
               medianprops={'marker': 'x', 'linestyle': '--', 'color': '#FF6D70'})  # Set median line
    plt.xlabel("Status", fontsize=12)
    plt.ylabel('Age', fontsize=12)
    plt.show()

    # Plot the density plot of BMI
    bmi_healthy = df[df['Status'] == 'healthy']['BMI']  # BMI data when Status is healthy
    bmi_cancerous = df[df['Status'] == 'cancerous']['BMI']  # BMI data when Status is cancerous
    sns.kdeplot(bmi_healthy, label="Healthy status", color="green", alpha=.7)  # Plot health
    sns.kdeplot(bmi_cancerous, label="Cancerous status", color="red", alpha=.7)  # Plot cancerous
    plt.title('Density Plot of BMI', fontsize=18)
    plt.xlabel('BMI', fontsize=12)
    plt.ylabel('Density', fontsize=12)
    plt.legend()
    plt.show()
    return
```

Figure 10: box_density_plots function

The density diagram reflects the proportion of cancerous and healthy patients in the different BMI ranges. As can be seen from the graph below, the green curve represents Healthy Status and the red curve represents Cancerous Status, with the highest proportion of healthy patients in the BMI range of 20 to 25 and the highest proportion of cancerous patients between 25 and 33, which accounts for the highest proportion is over 7%.

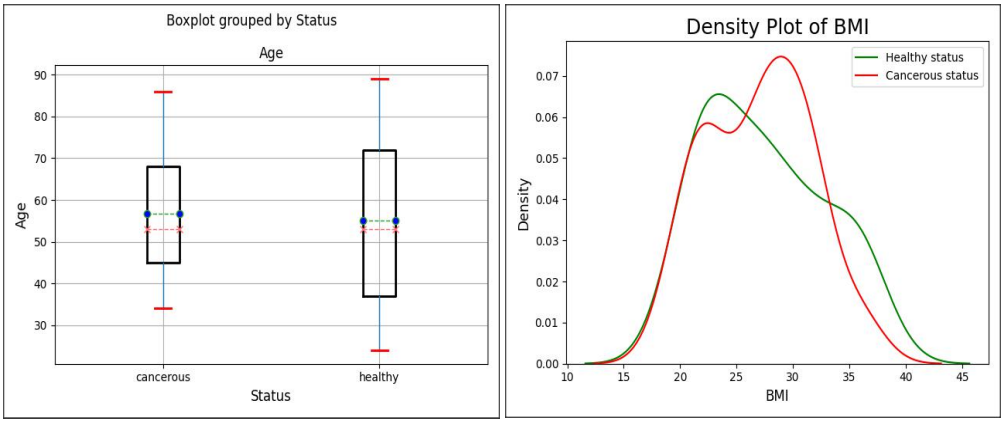

Figure 11: The boxplot and density plot

Box plots can be used to identify outliers, as box plots can be used to look at the overall distribution of data using statistics such as median, 25th/percentile, 75th/percentile, upper bound, and lower bound to describe the overall distribution of data. By calculating these statistics, a box plot is generated, with the box containing most of the normal data, and those outside the upper and lower boundaries of the box, being the abnormal data.

Section II: Discussion on Selecting an Algorithm

I disagree with this view, the classification models are only as good as the scenario in which it is applied. Although in most research scenarios high precision is pursued, in the field of disease diagnosis a test set achieving 90% precision is not a good enough measure of how well a model works in a scenario. I will provide a detailed analysis and explanation of why I disagree with this view in the following points.

1. In terms of the dataset split ratio, 70% of the data as training set, 10% as test set, and there are 20% as a validation set, however, it is not known what the approach to model validation is. If the trainee uses Holdout validation, which is not a cross-validation method, there is an obvious disadvantage for Holdout that is sensitive to the proportion of the training set and test set split, that is, the final evaluation index calculated on the validation set is highly dependent on the original grouping. Therefore, the method of cross-validation can avoid overfitting effectively.

2. In model evaluation, Accuracy (as seen in figure 12) is the proportion of the correct data (TP+TN) to the total data (TP+TN+FP+FN). However, accuracy is not the only best measure for assessing classification models. Ganesan (2014) indicated that "accuracy alone is sometimes quite misleading as you may have a model with relatively 'high' accuracy with the model predicting the 'not so important' class labels fairly accurately but the model may be making all sorts of mistakes on the classes that are actually critical to the application".

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \quad Recall = \frac{TP}{TP+FN} \quad Precision = \frac{TP}{TP+FP}$$

Figure 12: Accuracy, Recall and Precision formula

3. Especially in the disease diagnosis scenario, false positives can be tolerated for classification results, but false negatives must be reduced that means would rather misdiagnose a healthy patient, also don't miss any genuine confirmed cancerous patient. Therefore, sensitivity, accuracy, and other parameters should also be used as criteria for evaluating models. Recall (Sensitivity) represents the percentage of the population that actually has the disease that is tested "positive", also known as the true positive rate. And precision means how many of all predicted positives are correctly classified as positive. The closer these values (especially the sensitivity) to 1, the better the classifier model, As Hakama et al. (2007) said that "sensitivity is the indicator on the ability of screening to find cancer in the detectable preclinical phase (DPCP). The ability is usually specified as to the screening test".

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Figure 13: F1-Score formula

Therefore, for this type of cancer patient prediction model, we need to rely on parameters and criteria to analyse and evaluate the model in order to select the best algorithm. In addition to accuracy, precision, sensitivity, and specificity, the F1-Score can also be used for evaluation (Shown in figure 13). F1-Score is viewed as a superior indicator of the classifier's performance than the conventional accuracy measure. The most desirable value for the F1-score is to close to 1, by having both precision and recall have high values. A value of 1 for both means that the algorithm has the best accuracy.

Moreover, there is another measure that can be used to estimate the classifier's performance that is the receiver operating characteristic curve (ROC) and the area under the curve (AUC). The ROC space is defined by the FPR and TPR as the x and y axes respectively, which describe the relative trade-off between true positives (TP) and false positives (FP). TPR represents the percentage of all samples that are actually positive that is correctly judged to be positive, and FPR is the percentage of all samples that are actually negative that are incorrectly judged to be positive. As a consequence, “the quantitative metrics of accuracy and AUC are used for assessing the overall performance of a classifier. Specifically, accuracy is a measure related to the total number of correct predictions. On the contrary, AUC is a measure of the model's performance which is based on the ROC curve that plots the tradeoffs between sensitivity and 1-specificity (figure 14)” (Kourou et al., 2015). Therefore, when comparing different classification models, the ROC curve for each model can be plotted, and the area under the curve can be calculated and compared, as an indicator to judge the pros and cons of the model.

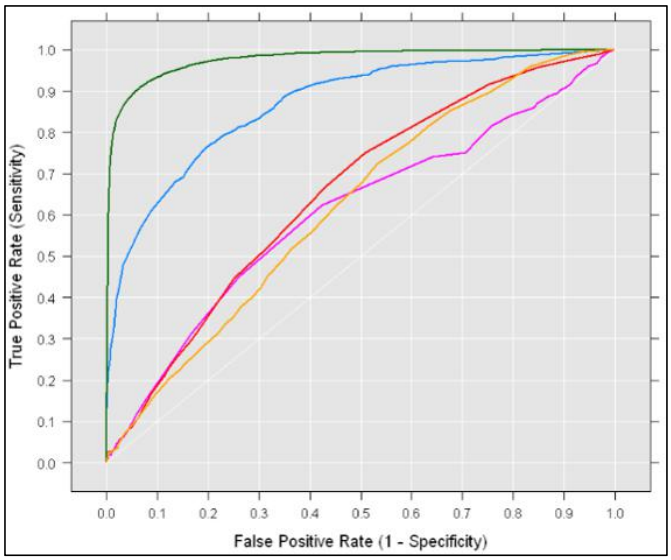

Figure 14: ROC Space and AUC

For selecting the best performing classifier model, the selected model and parameters also need to be evaluated using cross-validation, which can be used not only to assess the predictive performance of the model, especially the performance of the trained model on new data, but also to reduce overfitting to some extent. And the common verification methods are holdout cross validation, k-fold cross validation and Bootstrapping.

Usually use K folds cross-validation method, which means that a given data set is divided into K folds, where each fold is used as a test set at a certain point. Taking the scenario of 10-fold cross-validation, the data set is split into 10 folds randomly. In the first iteration, the first fold is used to test the model, and the rest are used to train the model. In the second iteration, the second fold is used as the test set, and the rest are used as the training set. This process is repeated until each of the 10 folds are used as the test set. This process is repeated until each fold of the 10 folds has been used as the testing set. Finally, report the mean accuracy results are used to evaluate the model.

Section III: Designing Algorithms

Artificial Neural Network (ANN)

```
def data_processing():
    """Data Normalisation & Shuffle, Split Data"""
    [df, data] = import_data()  # Import the origin data
    data_status = df['Status']  # Status list
    data_normal = max_min_normalization(data)  # Data Normalisation
    # data_normal['Status'] = data_status  # Add Status list into data_normal
    # Convert healthy = 0, cancerous = 1
    data_status = data_status.apply(lambda x: 0 if 'healthy' in x else 1)
    # Shuffle data frame and split the training set and testing set
    x_train, x_test, y_train, y_test = train_test_split(data_normal, data_status, test_size=0.1)
    return x_train, x_test, y_train, y_test
```

```
def ann_model(x_train, x_test, y_train, y_test, epochs=200):
    """Artificial Neural Network (ANN) Classifier"""
    global accuracy
    acc_array = []
    for epoch in range(1, epochs, 20):
        # Initialising the model
        ann = MLPClassifier(shuffle=True, hidden_layer_sizes=[500, 500], activation='logistic',
                            solver='lbfgs', alpha=0.1, random_state=1, max_iter=epoch)
        ann.fit(x_train, y_train)  # Fit the model to train_data matrix X and target y
        accuracy = ann.score(x_test, y_test)  # Return the average accuracy
        acc_array.append(accuracy)  # Add the accuracy values into list
    print('ANN Model Accuracy: %.2f%%' % (accuracy * 100))
    plt.plot(range(1, epochs, 20), acc_array, color='r', label='Accuracy')  # Generate the plot
    plt.xlabel('epochs'), plt.ylabel('accuracy'), plt.title("Epochs - Accuracy Plot")
    plt.legend(), plt.show()  # Display plot
    return
```

Figure 15: Functions data_processing() and ann_model()

Before design an artificial neural network (ANN) classifier, need to use function ‘data_processing()’ to normalize the data set, and turn the elements in "Status" to 0 or 1. Then Shuffle the data set and split 90% of the data as the training set and 10 % as the testing set. In this function, I used the customized function ‘max_min_normalization()’ to achieve data normalisation. For data split processing, train_test_split() is a function to randomly divide the training set and test set provided in the model_selection module of the sklearn package, which was used in this function and output the data and labels of the training set and test set separately.

The purpose of function ann_model is building an ANN classify mode and using training data to train this model, in the end, calculate the accuracy of the test set through the trained model. For initialising the model, as shown in figure 15, function MLPClassifier is a multi-layer Perceptron classifier. Setting the number of neurons in two hidden layers to 500, and use the logistic function as the non-linear activation function for the hidden layers. Moreover, this model optimizes the log-loss function using lbfgs (an optimizer in the family of quasi-Newton methods), in terms of the small dataset, it can converge faster and perform better. The maximum number of iterations (max_iter) is equal to epochs. Next step is fitting the model to training matrix data and target label. The accuracy results of the final artificial neural network are 92.31%.

And set the number of epochs is 200 and the step size is 20, in order to calculate the corresponding accuracy according to different epochs by for loop, and store the multiple accuracies in a list. In the end, plot using the number of epochs in the ‘x’ axis and the accuracy in ‘y’ axis (fig. 16). The accuracy results increase as the number of epoch times increases, as observed for the figure.

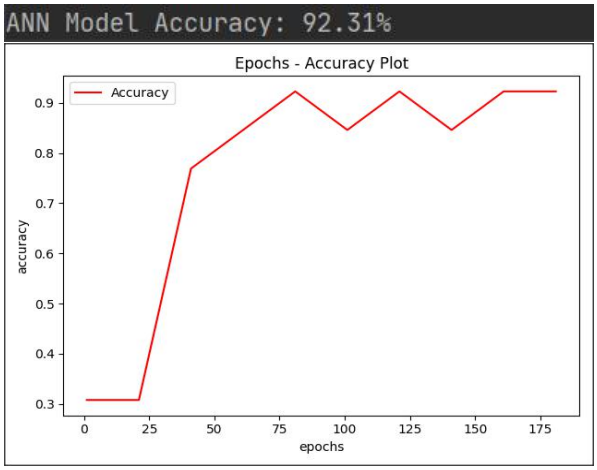

Figure 16: The result accuracy and plot

Random Forests Classifier

```
def forests_model(x_train, x_test, y_train, y_test, tree_num=1000, min_sam=5):
    """Random Forests Classifier"""
    # Initialising the model
    clf = RandomForestClassifier(n_estimators=tree_num, max_depth=None,
                                 min_samples_split=min_sam, bootstrap=True, oob_score=True)
    scores1 = cross_val_score(clf, x_train, y_train)  # Cross validation
    clf.fit(x_train, y_train)  # Fit the classifier model with training data
    y_pred = clf.predict(x_test)  # Using the trained classifier to predict the labels
    acc_test = "%.2f%%" % ((metrics.accuracy_score(y_test, y_pred)) * 100)  # Report the accuracy
    print('Forests Model Accuracy:', acc_test)
    return acc_test
```

Figure 17: Function forests_model() to build random forests and return the accuracy

Random Forest is a classification algorithm that uses a bootstrap resampling technique to repeatedly drawn samples at random from the original training sample set N to generate a new set of training samples for training decision trees, and then follows the above steps to generate m decision trees to form a random forest, with the classification result of the new data determined by the number of votes formed by the classification tree. In essence, it is a modification of the decision tree algorithm, where multiple decision trees are combined together, with the creation of each tree dependent on independently drawn samples.

In the function forests_model (fig. 17), I used the module RandomForestClassifier from the sklearn package, which is a random forest is a meta-estimate that fits many decision tree classifiers on individual subsamples of a dataset and uses averages to improve prediction accuracy and control overfitting. In the module RandomForestClassifier, there are several parameters that need to be set. ‘n_estimators’ stands for the number of trees in the forest, usually n_estimators is too small that is easy to underfitting and too large for trees number, more trees will give the better model performance, therefore, this parameter is set to 1000. And min_samples_split indicates the minimum number of samples needed to split internal nodes, set to {5 & 50}. After model initialization, using the fit method to build a forest of trees from the training set, and then predict the test label by testing class x. Once the algorithm model has been built, it needs to be evaluated to determine its merit. Generally, a training set is used to build the model and a test set is used to evaluate the model. Therefore, return the mean accuracy on the given original test data and predicted labels.

```python
def forests_plot(x_train, x_test, y_train, y_test, tree_number, min_samples):
    """Create a plot and show performance changes as more trees are added"""
    acc_list_5 = []  # When min_samples_split value is 5
    acc_list_50 = []  # When min_samples_split value is 50
    # Foreach to calculation accuracy
    for i in tree_number:
        for j in min_samples:
            acc_values = forests_model(x_train, x_test, y_train, y_test, i, j)
            if j == 5:
                acc_list_5.append(acc_values)  # Store acc_values when min_samples is 5
            elif j == 50:
                acc_list_50.append(acc_values)  # Store acc_values when min_samples is 50
    # Generate the plot
    plt.plot(tree_number, acc_list_5, color='red', label="min_samples_split: 5", linestyle='--')
    plt.plot(tree_number, acc_list_50, color='blue', label="min_samples_split: 50", linestyle='-.')
    plt.legend()
    plt.show()
    return
```

```python
def main():
    """Set the parameters and call the functions"""
    tree_number = [100, 1000, 2000, 3000, 4000, 5000]
    min_samples = [5, 50]
    x_train, x_test, y_train, y_test = data_processing()  # Data Pre-processing
    # ann_model(x_train, x_test, y_train, y_test)  # ANN Classifier
    acc_sam1 = forests_model(x_train, x_test, y_train, y_test, 1000, 50)
    acc_sam2 = forests_model(x_train, x_test, y_train, y_test, 1000, 5)
    print('Samples: 50 Forests Model Accuracy: ', acc_sam1)
    print('Samples: 05 Forests Model Accuracy: ', acc_sam2)
    forests_plot(x_train, x_test, y_train, y_test, tree_number, min_samples)
```

Figure 18: Function forests_plot() to create the plot

Function forests_plot is used for create a plot and show performance changes as more trees are added. First, set up two arrays to hold the accuracy, then iterate through the number of samples and number of trees arrays set up in the main function via a for loop (figure 18). The values inside the arrays are then assigned to the forests_model function called to return the accuracy rates, and the accuracy rates are stored in the arrays. In the end, generate plots from accuracy and number of trees. (Figure 19)

```
Min Samples:50  Forests Model Accuracy: 76.92%
Min Samples:05  Forests Model Accuracy: 84.62%
```

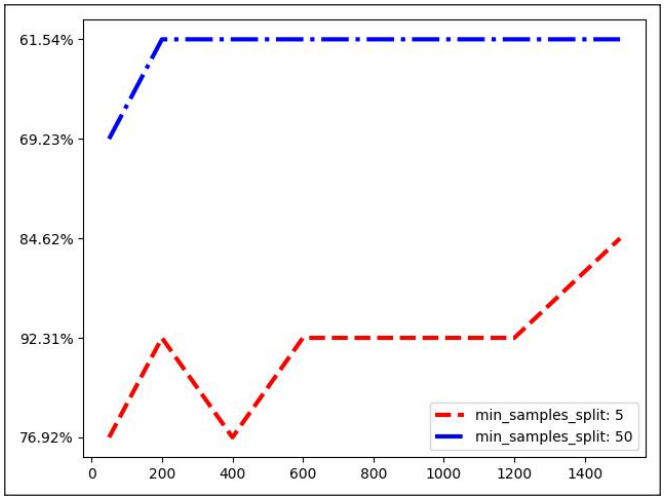

Figure 19: Function forests_plot() to create the plot

When the minimum number of samples required to be at a leaf node is 5, the accuracy results of the random forest is 76.92%, and when the minimum number of samples is 50, the accuracy results will increase to 84.62%. According to figure 19, the more the number of trees in the random forest classifier model, the higher the accuracy of the model when the minimum number of samples is fixed.

Section IV: Model Selection

Cross-Validation is a common method used in machine learning to build models and validate model parameters. It groups the original dataset into a training set and a validation set or test set, first using the training set to train the model, and then using the validation set to test the generalisation error of the model. In addition, data is always limited in reality, so k-fold cross-validation is proposed in order to reuse the data. The k-fold cross-validation reduces the variance by averaging the results of training over k different subgroups, and it can be used to assess the predictive performance of the model, especially the performance of the trained model on new data, and to reduce overfitting to some extent. It can also be used to obtain as much valid information as possible from a limited amount of data.

```
def data_processing():
    """Import data and Pre-Processing"""
    [df, data] = import_data()  # Call function import_data to import data
    data_status = df['Status']  # Status list
    # Convert healthy = 0, cancerous = 1
    data_status = data_status.apply(lambda x: 0 if 'healthy' in x else 1)
    data_normal = max_min_normalization(data)  # Data Normalisation
    return data_normal, data_status
```

Figure 20: Data processing

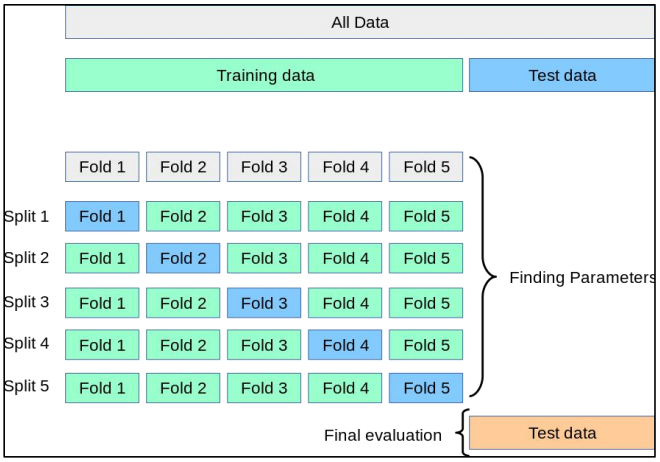

Figure 21: K-folds Cross_Validation

Before evaluating the model parameters, it is necessary to randomly split the data into 10 folds. The original data is imported and pre-processed at first (as shown in fig. 20), and the returned results are the normalised dataset and the corresponding list of labels respectively. And using K-Folds cross-validator from sklearn.model_selection.KFold library. The main approach is to divide the training or test data into n_splits mutually exclusive subsets, each time using one of the subsets as the validation set, and the remaining n_splits - 1 as the training set, and perform n_splits times training and testing to obtain n_splits results (figure 22 & 23), with the value of the n_splits parameter set to 10 as required. Finally, the mean of the k times assessment results is used as the final evaluation indicator.

```
def main():
    """Set the parameters and call the functions"""
    data_normal, data_status = data_processing()  # Data Pre-processing
    hidden_neurons = [50, 500, 1000]  # ANN two hidden layers, [50, 500, 1000] neurons.
    trees_number = [20, 500, 10000]  # Random forest [20, 500, 10,000] trees
    # Set parameters for ANN and Random Forests classifier model to evaluate
    ann_cross_validation(data_normal, data_status, hidden_neurons, folds=10)  # Evaluate ANN
    random_forests_cross_validation(data_normal, data_status, trees_number, folds=10, min_samples=5)
    return
```

Figure 22: Random forests classifier model evaluation

The next step is to set the parameters of the ANN model and the random forest model, there are two hidden layers with [50, 500, 1000] neurons in the ANN model. For random forests, set the number of trees to [20, 500, 10000], set with the minimum number of samples required to be at a leaf node is 5. These parameters are be set in main function (Figure 22).

```
def ann_cross_validation(data_normal, data_status, neurons, folds):
    """Artificial Neural Network (ANN) Classifier"""
    cv_scores = []  # Store the resulting values for each model
    k_folds = KFold(shuffle=True, n_splits=folds, random_state=1)
    for neuron in neurons:  # Apply different number of neuron parameter in ANN model
        ann = MLPClassifier(shuffle=True, hidden_layer_sizes=[neuron, neuron], activation='logistic',
                            solver='lbfgs', alpha=0.1, random_state=1)  # Initialising the model
        score = cross_val_score(ann, data_normal, data_status, cv=k_folds)  # Cross validation
        cv_scores.append(score.mean())
        print("ANN: Neurons:%d Accuracy:%0.2f" % (neuron, score.mean()))  # Report accuracy
    plt.plot(neurons, cv_scores,  color='d')  # Plot the accuracy change tend
    plt.xlabel('Neurons')
    plt.ylabel('Accuracy')
    plt.show()  # Choose the best parameters from the image
    return
```

```
def random_forests_cross_validation(data_normal, data_status, trees, folds, min_samples):
    """Random Forests Classifier"""
    cv_scores = []  # Store the resulting values for each model
    k_folds = KFold(shuffle=True, n_splits=folds, random_state=1)
    for tree in trees:  # Apply different number of trees parameter in random forests model
        forest_model = RandomForestClassifier(n_estimators=tree, min_samples_split=min_samples,
                                              bootstrap=True, oob_score=True)  # Initialising the model
        score = cross_val_score(forest_model, data_normal, data_status, cv=k_folds)  # Cross validation
        cv_scores.append(score.mean())
        print("Random Forests: Trees:%d Accuracy:%0.2f" % (tree, score.mean()))  # Report accuracy
    plt.plot(trees, cv_scores, color='r')  # Plot the accuracy change tend
    plt.xlabel('Trees')
    plt.ylabel('Accuracy')
    plt.show()  # Choose the best parameters from the image
    return
```

Figure 23: ANN classifier model evaluation ANN classifier model evaluation

In functions ann_cross_validation and random_forests_cross_validation (Figure 23), the parameters are continuously changed through a for loop, and cross-validation is used to evaluate the performance of the model with different parameters. Assigning k_folds to ‘cv’ in the cross_val_score, which is the number of cross-validation folds or iterations that can be performed. The cross_val_score function in scikit-learn evaluates the scores by cross-validation, passing the initialised classifier model and the set of parameters into this method, returning the mean accuracy results for each set of parameters, and drawing a plot of the variation in the accuracy change. The final model with the best performance is selected.

```
ANN: Neurons:50 Accuracy:0.83
ANN: Neurons:500 Accuracy:0.77
ANN: Neurons:1000 Accuracy:0.76
Random Forests: Trees:20 Accuracy:0.74
Random Forests: Trees:500 Accuracy:0.80
Random Forests: Trees:10000 Accuracy:0.81
```

Figure 24: Mean accuracy results for different models and Parameters

Finally, the pre-set parameters were applied to the corresponding models, and the mean accuracy results for each set of parameters were calculated (Figure 24). It is clear from figure 24 that the ANN model performs best when the number of neurons is 50, and the random forest model performs best when the number of trees is 10,000. The best mean accuracies of the two classifier models are 0.83 and 0.81 respectively.

Moreover, as seen in figure 25, the mean accuracy of the ANN model decreases as the number of neurons increases, while the mean accuracy of the random forest model improves with the increase of the number of trees. Therefore, for the ANN model, two hidden layers should be used, and the number of neurons in each layer should be 50. In terms of the random forests model, the number of trees in the forest should be 10000, and the minimum number of samples is set for 5.

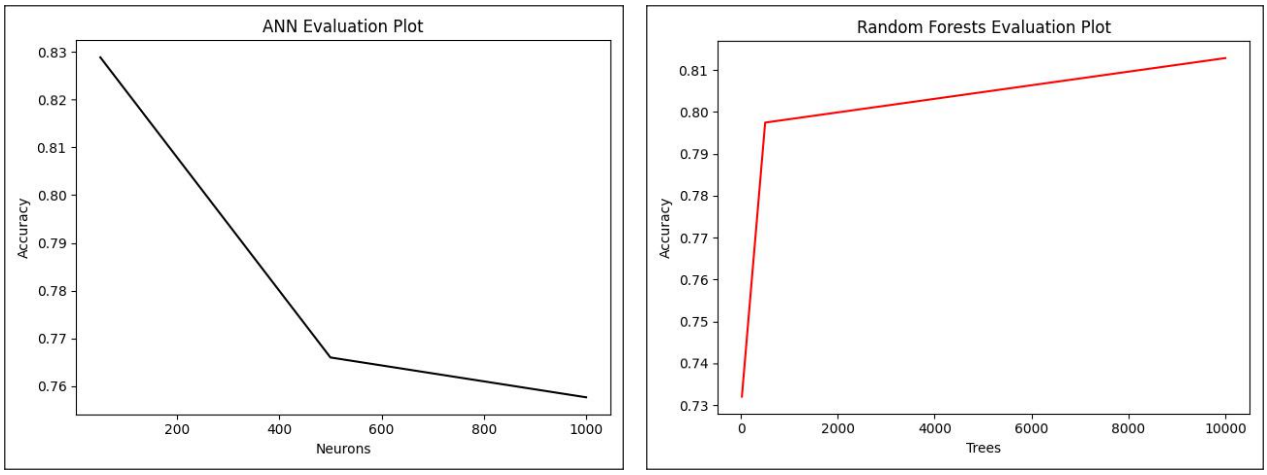

Figure 25: Accuracy changing tend with different parameters

For this dataset, I prefer to use a random forest classifier model. The reason is that "random Forest is less computationally expensive and does not require a GPU to finish training. A random forest can give you a different interpretation of a decision tree but with better performance. Neural Networks will require much more data than an everyday person might have on hand to actually be effective. The neural network will simply decimate the interpretability of your features to the point where it becomes meaningless for the sake of performance" (Montantes, 2020). In addition, the artificial neural networks are more suitable for processing complex natural signals, such as images, sounds, and natural languages.

Reference List

Ganesan, K. (2014). *How to compute precision and recall for a multi-class classification problem*. [online] Kavita Ganesan, Ph.D. Available at: https://kavita-ganesan.com/how-to-compute-precision-and-recall-for-a-multi-class-classification-problem/#.YBa2pNj7SUl [Accessed 31 Jan. 2021].

Hakama, M., Auvinen, A., Day, N.E. and Miller, A.B. (2007). Sensitivity in cancer screening. *Journal of Medical Screening*, 14(4), pp.174–177.

Kourou, K., Exarchos, T.P., Exarchos, K.P., Karamouzis, M.V. and Fotiadis, D.I. (2015). Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, [online] 13, pp.8–17. Available at: https://www.sciencedirect.com/science/article/pii/S2001037014000464 [Accessed 30 Jan. 2021].

Montantes, J. (2020). *3 Reasons to Use Random Forest Over a Neural Network–Comparing Machine Learning versus Deep…*. [online] Medium. Available at: https://towardsdatascience.com/3-reasons-to-use-random-forest-over-a-neural-network-comparing-machine-learning-versus-deep-f9d65a154d89 [Accessed 1 Feb. 2021].

Sola, J. and Sevilla, J. (1997). Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3), pp.1464–1468.