Machine Learning – Assessment 2

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

There are 127 samples of data, with 9 features available for training (or 10 if including ‘status’). A summary of these features is provided in the table below, generated from the ‘describe()’ function.

```
data.describe()
```

	Age	BMI	Glucose	Insulin	HOMA	Leptin	Adiponectin	Resistin	MCP.1
count	127.000000	127.000000	127.000000	127.000000	127.000000	127.000000	127.000000	127.000000	127.000000
mean	55.850394	27.215728	95.700787	9.557037	2.547656	25.621757	9.831036	14.085466	511.389102
std	16.351826	5.008564	22.696869	9.738491	3.513617	18.705080	6.671655	12.040539	341.194930
min	24.000000	18.370000	60.000000	2.432000	0.467409	4.311000	1.656020	3.210000	45.843000
25%	44.000000	22.876410	84.000000	4.336500	0.898747	12.226100	5.445953	6.780130	263.079267
50%	53.000000	27.100000	92.000000	5.782000	1.341324	19.065300	8.237405	10.344550	426.175000
75%	69.000000	31.066529	101.000000	10.523000	2.628942	34.375600	11.127345	17.181865	662.160500
max	89.000000	38.578759	201.000000	58.460000	25.050342	90.280000	38.040000	82.100000	1698.440000

Missing values have been removed using the following ‘dropna’.

```
data = data.dropna(how='all')
```

X and Y have been obtained by separating the column ‘status’ from the original data – this creates Y, the object labels. This column is then removed from the data, creating X – the training data.

```
label = data['Status']
data.pop('Status')

X = np.array(data)

label = label.replace(['healthy'], 0)
label = label.replace(['cancerous'], 1)

y = np.array([label])
y = y.T
```

The values of ‘healthy’ and ‘cancerous’ are changed to 0 and 1 respectively for ease of computation. Y is also transposed to fit the dimensions of X.

```
print(f"Shape of X: {X.shape}")
print(f"Shape of Y: {y.shape}")
```

```
Shape of X: (127, 9)
Shape of Y: (127, 1)
```

Also, to ensure that data was in a range appropriate for machine learning methods, X was normalized using the following method:

```
X = X / np.linalg.norm(X)
```

Normalizing the data increases the speed and stability of machine learning algorithms as variance is standardised (Cao et Al, 2016), and reduces the computational requirements of each model.

After this stage, a split of training/test data was generated using sklearn’s ‘train_test_split’ function, with the random state set to 1 to ensure results could be replicated.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, random_state = 1)
```

This results in a training set with 101 elements and testing set with 26 (with either 9 or 1 feature), demonstrated as such:

```
print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of y_train: {y_train.shape}")
```

```
Shape of X_train: (101, 9)
Shape of y_train: (101, 1)
```

```
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_test: {y_test.shape}")
```

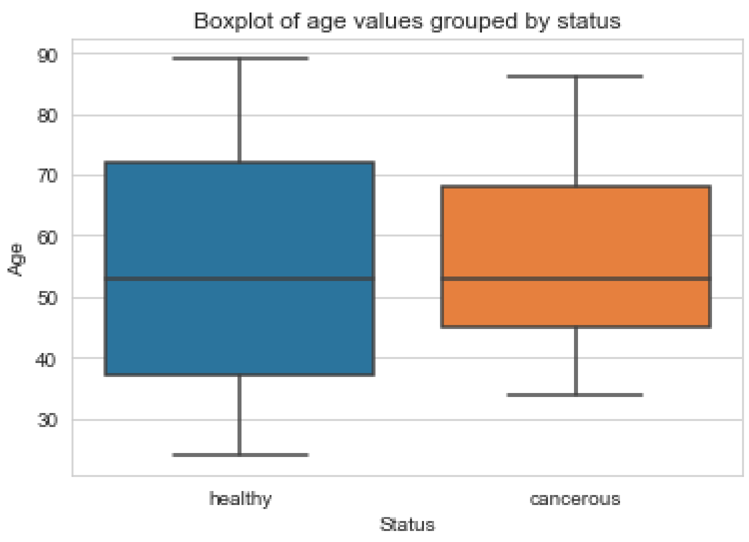
```
Shape of X_test: (26, 9)
Shape of y_test: (26, 1)
```

Plotting the two charts required was done through using Pandas density plots and Seaborn.

This code was used to create the boxplot:

```
sns.set_style("whitegrid")
sns.boxplot(x = 'Status', y = 'Age', data = graph_info).set_title('Boxplot of age values grouped by status')
plt.show()
```

Resulting in this graph:

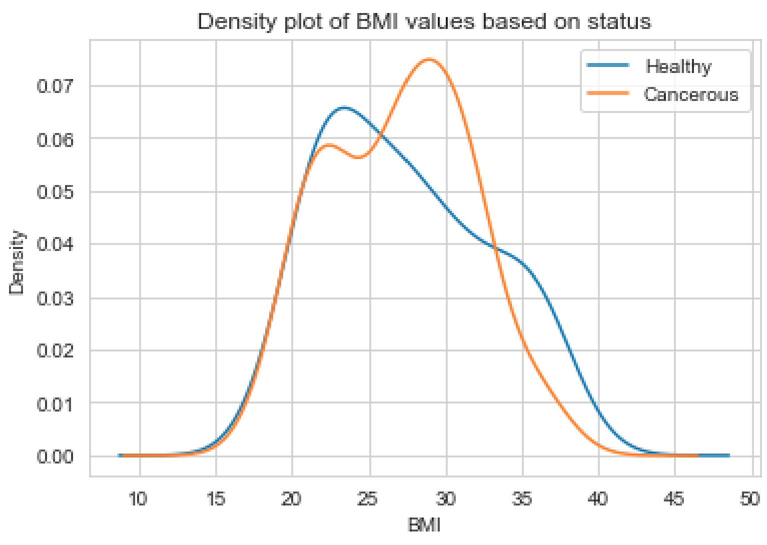


This boxplot visualises the likelihood of one having cancer at certain ages, providing evidence that a person is likely to older than younger to be at risk.

To generate the layered density plot, the status of healthy and cancerous were obtained separately to segment BMI values. The Pandas density plot function 'kde' was then called to create the charts on the same axis.

```
healthy = graph_info[graph_info.Status.isin(['healthy'])]
cancerous = graph_info[graph_info.Status.isin(['cancerous'])]

ax = healthy[['BMI']].plot.kde()
cancerous[['BMI']].plot.kde(ax=ax)
ax.legend(['Healthy', 'Cancerous'])
ax.set_title('Density plot of BMI values based on status')
ax.set_xlabel('BMI')
plt.show()
```

This results in the following graph:



This graph shows that a higher BMI can often lead to an increase in cancer risk. Although, at very high BMI values the healthy status becomes more common.

At first glance, it can be guessed that a higher BMI would lead to a greater risk of cancer, but this belief is not supported at higher values. As BMI is not indicative of a higher body fat (Nutall, 2015), it is possible that the BMI values of 35 and above were generated thanks to a large amount of muscle and healthy tissue, rather than body mass gained from obesity. Reportedly, obesity can be linked to one in seven cancer deaths within the USA (Calle et Al, 2004), thus it can be assumed that a high BMI within this chart does not necessarily support a status of cancerous.

Additionally, the cancerous status appears to mainly effect those within a narrower age range than that of the healthy status. As people age, cancer risk increases, with the median age of patient death from cancer being approximately 66 years old (White et Al, 2014). Interestingly, being older than this seemingly reduces the chances of one obtaining the healthy status. Despite the positivity of this, I would estimate that this is likely a case of survivorship bias, whereby people with a higher resistance to developing cancer are likely to live slightly longer. Another possibility for this could be because of other health risks that are not included within the scope of this data, such as heart attacks or strokes.

Section 2: Discussion on Selecting an Algorithm

Regarding the implications of a 90% accuracy being the most effective for a trained machine learning model; an algorithm can be trained to certain points where it can be considered either *underfitted* or *overfitted*. Underfitting is a concept in machine learning where a trained model is too simple to fully understand the structure and features of input data (Mitchell, 1997). When a model underfits, it allows room for more behaviour than anticipated (Van der Aalst et Al, 2010), as the variability of data is left uncaptured and thus unrecognised (Jabbar et Al, 2015).

Overfitting is quite simply the opposite of underfitting. When a model is trained on an excessive number of epochs or features, it propagates additional levels of complexity with no benefit to performance (Hawkins, 2004). Essentially, the algorithm begins to learn from random irregularity that has no actual given context within the data: increasing performance costs and potentially decreasing prediction accuracy.

Goodhart’s Law is a premise that has basis in machine learning, which states that, essentially, forcing a measure to become a target. Within machine learning, there are a variety of measures that could be considered ‘targets’, such as: accuracy, weighted accuracy, precision, recall, f-measure, and ROC curve (receiver operator characteristic curve). When these targets become metrics, optimization can become ineffective or harmful (Manheim et Al, 2018). Therefore, the question of ‘is 90% accuracy proof of an effective algorithm?’ is difficult to wholly ascertain without knowledge of other metrics.

As mentioned earlier, there are other metrics to measure other than just accuracy, all of which having varying levels of authenticity and applicability within a problem scope. Under the background given for this task and assessment (classification of 'healthy' or 'unhealthy'), accuracy is better broken down into two subsets; precision and recall. Precision and recall are, respectively, the measure of true positives against false positives, and the measure of true positives against false negatives (Shajahaan et Al, 2013).

Depending on background, one of these two metrics may be considered more important than the other (Morstatter et Al, 2016), which I believe is the case for this background. Consider the environment of the problem scope, and how the data is gathered: complex physiological information regarding one's health cannot be easily ascertained from home or with common equipment, thus is it likely that it was generated from equipment and trained specialists at a hospital (or similar health centre).

As said, precision is concerned with false positives – in this case, 'healthy' individuals being labelled as 'unhealthy'. This would lead to further testing, doctor referral, and treatment (MacArthur et Al, 1984). Given the circumstances, there are an incredible number of scenarios where the false positive could be detected, such through MRI scanning (Reference), biomarkers (Reference), even nanotechnology, in recent trends. All of the above indicates that a false positive is, ultimately, of little concern, thanks to the availability of further methods that can be used to detect it.

However, on the other hand therein lies the importance of detecting false negatives: recall. Once again, background is assumed to remain the same, thus a patient who is declared to have a 'healthy status' is likely to return home and delay treatment. This could possibly lead to potential legal action against the health service that released the false negative, or reduce - potentially nullifying - the effectiveness of treatment (Petticrew et Al, 2000). For these reasons alone, a false negative is of far greater consequence than a false positive.

To extend this further, the majority of people will not test positive for cancer – 44% of men, and 38% of women will test positive for cancer in their lifetime (Siegel et Al, 2014) – and the chances of a large group of people developing cancer around the same time is (without external circumstances) incredibly low. Because of this, a machine learning model can receive, for example, 100 data samples in a batch, 90 of them with a status of 'healthy, and 10 with a status of 'unhealthy'. If the model were to classify every sample as 'healthy' then the network would have 90% accuracy – a near perfect result – with 0% effectiveness. Reasons such as this are why metrics are only as important as their background allows. If the intern within this scenario was to have trained a model that classifies every status as 'healthy', receiving an overall accuracy of 90%, the model would be incredibly ineffective.

Despite its subjective nature, another component of machine learning that must be measured is interpretability (Bibal et Al, 2016). Simply trusting the results of a machine learning model is often ill-advised, as single metrics do not denote a full description of real world tasks (Doshi-Velez et Al, 2017). Interpretability can be broken down into two definitions: transparency, and post-hoc (Lipton, 2018).

Transparency indicates a level of understanding given to a machine learning model – what is known about a model -, whereas post-hoc indicates what can be learnt from a model. An example of interpretability common in machine learning is the format of weights: floats or integers formatted in scientific notation makes them far more difficult to understand comparative to plain numeric. Methods exist that can modify these values to transform them into more interpretable ones, but this becomes extensive in the case of larger models. If a model is uninterpretable then the user is less likely to trust the results, and, in a background of medical science, this is vital. Physicians, doctors, and other medical professionals must be able to put full stock in the tools they use, and in many cases the information used in a model is sparse or relatively unknown. The limited knowledge available in such cases makes interpretability vital, as any error can originate from either the incompleteness of data or a badly trained model (Vellido et Al, 2012).

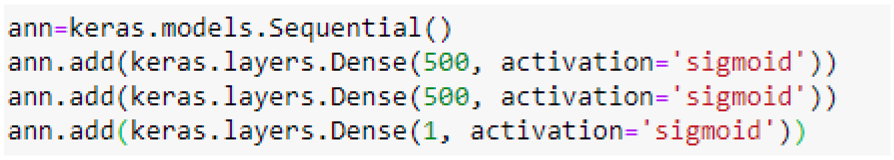
To conclude this segment; answering the question of whether or not I agree with the stance that the best algorithm receives a success rate of 90% is based on the general circumstance surrounding the training of the algorithm, and the reasons I have discussed prior. If the algorithm has a 90% success rate, with no bias towards metrics (such as precision and recall), and is interpretable, then it is likely to be superior to many others that could also be considered applicable in the problem scope.

Section 3: Designing Algorithms

Artificial Neural Network

An artificial neural network is a machine learning method inspired by the basic learning and thinking method of the human brain. An ANN is built using a series of nodes and connecting weights, representing the interconnected neurons and synapses within a biological brain. There are three basic layers to the network: input layers – layers that accept an input of data; output layers – layers that output a prediction for every element of data; and hidden layers – so named for them being placed in-between the input and output layers. General use case scenarios mean that the hidden layers go by unnoticed, whereas the user fully interacts with input and output layers.

To build an ANN capable of classifying data to one of two classes, the Keras package was used to build and configure a network that met the prerequisites.



```
ann=keras.models.Sequential()
ann.add(keras.layers.Dense(500, activation='sigmoid'))
ann.add(keras.layers.Dense(500, activation='sigmoid'))
ann.add(keras.layers.Dense(1, activation='sigmoid'))
```

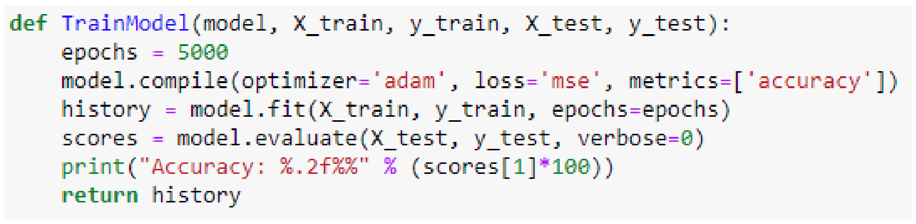
Declaring the model as sequential transforms the network into a standard feed-forward network, where each layer has one forward and backwards connection. There are two hidden layers (‘Dense’ layers) with 500 neurons, and one output layer (Also a ‘Dense’ layer). The activation functions for each layer has been set to sigmoid, which reduces linearity (Karlik et Al, 2011).

The formula for forward propagation is often calculated as the following linear equation:

$$O = (I \cdot W) + b,$$

where O is the output, I is an input, W are the weights connected nodes, and b is the neuron bias term. Bias is added to the linear equation of input multiplied by weights to adjust the output, ensuring that every node outputs a value if it is required. This is repeated for a set number of epochs, where the weights and biases are subtly modified and recalculated to find the most relevant output for n given inputs. This is achieved through a multitude of ways, although one common method is backpropagation, where the error rate of each layer’s weights and biases is calcualted starting from the output layer, propagating error backwards based on past gradients.

To train the network, the following function has been created, using a selection of in-built features from Keras:



```
def TrainModel(model, X_train, y_train, X_test, y_test):
    epochs = 5000
    model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=epochs)
    scores = model.evaluate(X_test, y_test, verbose=0)
    print("Accuracy: %.2f%%" % (scores[1]*100))
    return history
```

This function trains the network on a set amount of epochs (5000 in the image above), and establishes the optimizer, loss method, and metric. The optimizer establishes how the network will learn – in this case, using the ‘adam’ method. Loss identifies how loss, or cost, will be calculated. Metrics distinguishes what the network will attempt to improve above all other methods – accuracy, in this case.

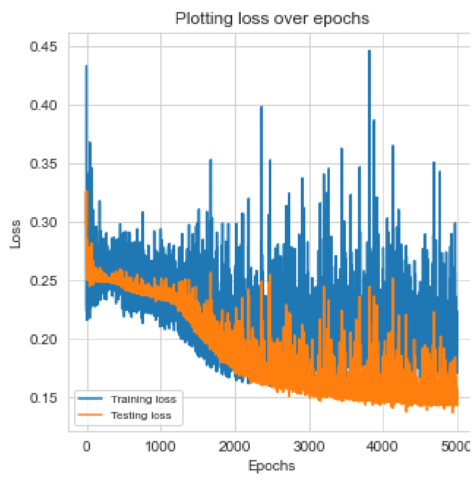
Using the function '.fit' begins to train the network using a set of training data and labels. This is saved to a list compiler 'history', which is used later to generate plot data showing cost and accuracy through each epoch. Finally, the test set is used to evaluate the performance of the network on unseen data. The accuracy of these predictions is displayed, and history is returned.

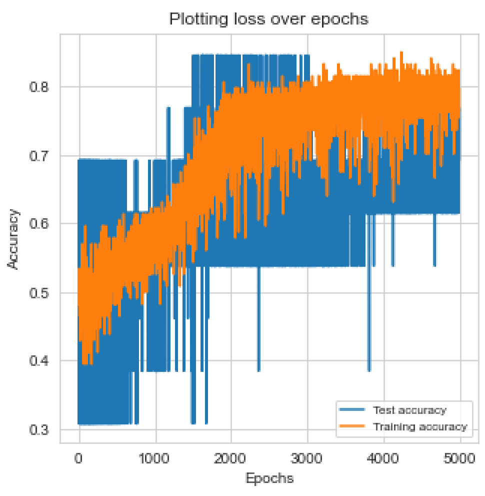
The amount of epochs changes the overall accuracy of the network on the test set, records of which have been saved and are displayed below:

Epochs	Accuracy	Time (Seconds)
10	69.23%	0.67
100	69.23%	1.56
1000	53.85%	8.68
2500	61.54%	20.28
5000	76.92%	42.67
10,000	76.92%	80.52
20,000	76.92%	178.89

Note that weights are randomly generated, thus these results cannot be exactly replicated. Additionally, an attempt at creating an artificial neural network from scratch has been made, although it fails to produce accurate prediction results. It has been attached to the supporting Python code alongside this neural network and the following random forests algorithm.

Two charts have been generated to detail the change of accuracy and loss as epochs are initialised. The epoch of 5000 has been chosen to represent the neural network in this case as it receives a high accuracy, and a greater number of epochs would make the graphs difficult to interpret.





```
              precision    recall  f1-score   support

           0       0.31      1.00      0.47         4
           1       0.00      0.00      0.00         9

    accuracy                           0.31        13
   macro avg       0.15      0.50      0.24        13
weighted avg       0.09      0.31      0.14        13
```

This data shows the summary statistics of the results indicating a high recall which is important towards the greater functionality of the problem scope. The module 'classification_report' from sklearn.metrics was used to generate these values.

Decision Tree & Random Forest

The decision tree was coded from scratch using mostly Numpy operations.

Entropy is a formula that is used within decision trees (and thus random forests) to calculate the levels of disorder within a set of data (Lit et Al, 2006). This formula is used to calculate the overall information gain of particular thresholds and nodes, and it can be calculated as follows:

$$E(S) = \Sigma - p_i \; log_2 \, p_i$$

Where E(S) is entropy and p is a value within a set. This can be converted to Python script as follows:

```python
def entropy(y): # Calculates entropy
    hist = np.bincount(y) # Obtains frequency of elements
    ps = hist / len(y)
    return -np.sum([p * np.log2(p) for p in ps if p > 0]) # Calculates entropy
```

Leaves/decision points are stored as a custom ‘Node’ object that contains the following parameters:

```python
class Node: # Node for use in decision tree
    # Sets parameters on __init__
    def __init__(self, feature=None, threshold=None, left=None, right=None,*,value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value
    def is_leaf_node(self): # Returns a value if self.value has a value
        return self.value is not None
```

Said parameters store values that determine the purpose of each node: is it a decision node with following nodes linking to it’s left and right; which feature does it make decisions on; and so on. The definition of ‘is_leaf_node’ simply returns a positive Boolean as long as the node has a value, which indicates it as being a leaf node.

Firstly, parameters are established using an ‘__init__’ function, which is activated upon class initialisation. This sets generic parameters such as the maximum depth of the tree.

```python
class DecisionTree: # Decision tree
    # Sets paramaters on __init__
    def __init__(self, min_samples_split=5, max_depth=10, n_feats=None):
        self.min_samples_split = min_samples_split # The minimum samples required for a split
        self.max_depth = max_depth # How deep the tree will go
        self.n_feats = n_feats # Number of features
        self.root = None # Path of tree nodes
```

The function ‘fit’ is used to fit the number of features in an input dataset and grow the tree, calling a separate function ‘_grow tree’. Note that all functions with the prefix ‘_’ indicate a class only function that cannot be called outside of the class declaration.

```python
def fit(self, X, y): # Fits features for X
    self.n_feats = X.shape[1] if not self.n_feats else min(self.n_feats, X.shape[1]) # Finds minimum of features in X
    # Or self.n_feats, whichever is smaller
    self.root = self._grow_tree(X, y) # Grows the tree
```

Grow tree is used to first generate a number of samples and features from both X and y. As long as the depth of the tree is not greater than the maximum depth, the tree will continue to grow, splitting the indices of features for the node created by this function and generating the best feature and threshold for the current node. This is generated by the function ‘_best_criteria’.

```python
def _grow_tree(self, X, y, depth=0): # Function to grow tree, starts at depth 0
    n_samples, n_features = X.shape # Gets shape of X to identify number of features and samples
    n_labels = len(np.unique(y)) # Gets labels (classes)

    # Stopping Criteria
    if (depth >= self.max_depth
            or n_labels == 1
            or n_samples < self.min_samples_split):
        leaf_value = most_common_label(y)
        return Node(value=leaf_value)
    # Gets features indices for random selection of data
    feat_idxs = np.random.choice(n_features, self.n_feats, replace=False)

    # Greedily select the best split according to information gain
    best_feat, best_thresh = self._best_criteria(X, y, feat_idxs)

    # Grows the nodes resulting from the split (child nodes)
    left_idxs, right_idxs = self._split(X[:, best_feat], best_thresh)
    left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth+1)
    right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth+1)
    return Node(best_feat, best_thresh, left, right)
```

The best criteria function is used to generate the optimal threshold for a leaf node: it attempts to produce the greatest information gain from a set of thresholds and data. Once again, this gain is generated from another function '_information_gain'.

```python
def _best_criteria(self, X, y, feat_idxs): # Selects best criteria
    best_gain = -1 # Default value is worse than any gain to stop incorrect learning
    split_idx, split_thresh = None, None # Default
    for feat_idx in feat_idxs: # For every feature index
        X_column = X[:, feat_idx] # Gets column
        thresholds = np.unique(X_column) # Gets threshold
        for threshold in thresholds: # For every threshold
            gain = self._information_gain(y, X_column, threshold) # Calculates individual info gain

            if gain > best_gain: #If gain is superior
                best_gain = gain # Sets new values
                split_idx = feat_idx
                split_thresh = threshold

    return split_idx, split_thresh
```

Information_gain generates the greatest gain of information for indices to the left and right of each split. This is done through the calculation of entropy: the function for which demonstrated earlier.

```python
def _information_gain(self, y, X_column, s_thresh): # Calculates greatest info gain
    parent_entropy = entropy(y) # Calculates entropy

    left_idxs, right_idxs = self._split(X_column, s_thresh) # S

    if len(left_idxs) == 0 or len(right_idxs) == 0: # If no valid split
        return 0

    n = len(y) # Length of y
    n_left, n_right = len(left_idxs), len(right_idxs) # Gets length of left and right
    entropy_left, entropy_right = entropy(y[left_idxs]), entropy(y[right_idxs]) # Calculates entropy of both
    child_entropy = (n_left / n) * entropy_left + (n_right / n) * entropy_right # Calculates child entropy

    info_gain = parent_entropy - child_entropy # Establishes overall info gain
    return info_gain
```

The split function simply splits the indices for X based on the split threshold.

```python
def _split(self, X_column, s_thresh): # Split decision for node
    left_idxs = np.argwhere(X_column <= s_thresh).flatten() # Creates index for splits
    right_idxs = np.argwhere(X_column > s_thresh).flatten()
    return left_idxs, right_idxs
```

The predict function sends each value of X through the tree for prediction.

```python
def predict(self, X): # Traverses tree and calculates predictions for every value of X
    return np.array([self._traverse_tree(x, self.root) for x in X])
```

The 'traverse_tree' function uses a sample of data to maneuver said sample to the left or right of each node until it reaches a leaf node, at which point a prediction can be made.

```python
def _traverse_tree(self, x, node): # Traversal of tree
    if node.is_leaf_node(): # If at a leaf node, return it's value
        return node.value
    elif x[node.feature] <= node.threshold: # If not, continue to traverse left to right based on threshold
        return self._traverse_tree(x, node.left)
    else:
        return self._traverse_tree(x, node.right)
```

The following code demonstrates the concatenation of decision trees into a random forest algorithm. As before, parameters are initialised:

```python
class RandomForest: # Random forest for decision tree
    # Sets paramaters on __init__
    def __init__(self, n_trees=5, min_samples_split=50, max_depth=5, n_feats=None):
        self.n_trees = n_trees
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_feats = n_feats
        self.trees = []
```

The 'fit' function of the random forest class creates and appends a decision tree to the forest, using a bootstrap sample (a random sample of data given from X and y) to fit that tree with data.

```
def fit(self, X, y): # Fits data by creating multiple decision trees and fitting data to each one
    self.trees = []
    for _ in range(self.n_trees):
        tree = DecisionTree(min_samples_split=self.min_samples_split, max_depth = self.max_depth,
                            n_feats = self.n_feats) # Makes new tree with parameters
        X_sample, y_sample = bootstrap_sample(X, y) # Gets sample to send through tree
        tree.fit(X_sample, y_sample)
        self.trees.append(tree) # Adds tree to dictionary
```

Finally, the predict function is used to send data throughout the forest and generate a list of possible outcomes for each sample from every tree.

```
def predict(self, X): # Predicts by going through each tree
    tree_preds = np.array([tree.predict(X) for tree in self.trees])
    tree_preds = np.swapaxes(tree_preds, 0, 1)
    # Returns most common label for every tree's prediction
    y_pred = [most_common_label(tree_pred) for tree_pred in tree_preds]
    return np.array(y_pred)
```

The function 'bootstrap_sample' is used to emulate bootstrap sampling; a sampling metric that takes samples from a set of data and replaces them. This has been implemented as follows:

```
def bootstrap_sample(X, y): # Gets smaller sample from data
    n_samples = X.shape[0]
    # Random choice between 0 and n_samples
    idxs = np.random.choice(n_samples, size = n_samples, replace=True)
    return X[idxs], y[idxs]
```

The following table outlines the overall results to train the random forests algorithm using a minimum of 5 or 50 samples of data:

Trees	Accuracy (5 Samples)	Time (seconds)	Accuracy (50 Samples)	Time (seconds)
1	46.15%	0.185	69.23%	0.102
10	69.23%	1.488	69.23%	0.745
50	84.61%	7.845	76.92%	4.695
100	76.92%	14.622	84.61%	7.3
1000	76.92%	163.53	69.23%	78.24
5000	92.3%	1798.3	69.23%	809.23

From this, it can be extrapolated that the most efficient number of trees in terms of accuracy and speed is 50 trees with 5 samples, as it receives 80% accuracy in around 7 seconds. In terms of accuracy, the most effective method is 5000 trees with 5 minimum samples, although this takes an extensive amount of time.

To evaluate other metrics of performance as was done with the artificial neural network, the module 'classification_report' was used once more on the forest with 100 trees and 50 minimum samples, which resulted in this summary:

```
              precision    recall  f1-score   support

           0       0.60      0.75      0.67         4
           1       0.88      0.78      0.82         9

    accuracy                           0.77        13
   macro avg       0.74      0.76      0.75        13
weighted avg       0.79      0.77      0.78        13
```

Compared to the ANN, this value predicts a more balanced prediction of precision and recall.

Section 4: Model Selection

Before implementing k-fold CV, a copy of the unedited data is reindexed using the following functions:

```python
folds_info = graph_info.reindex(np.random.permutation(graph_info.index))
folds_info = folds_info.reset_index(drop=True)
```

This takes the data and shuffles it in, keeping row-wise elements together. Resetting the index is then done using ‘reset_index’. Standard pre-processing is then applied to the shuffled data:

```python
folds_label = folds_info['Status']
folds_info.pop('Status')

folds_label = folds_label.replace(['healthy'], 0)
folds_label = folds_label.replace(['cancerous'], 1)

folds_info = folds_info / np.linalg.norm(folds_info)
```

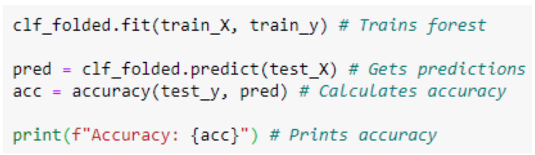
A function was then designed to allow for the generation of k-folds.

```python
def kfold(X, y, n_folds=0):

    fold_size = int(len(X) / n_folds) # Gets size of each fold
    if n_folds > int(len(X)): # If number of folds exceeds the maximum possible
        print(f"Number of folds exceeds maximum number of folds possible ({int(len(X))})") # Error message
        return np.empty(0), np.empty(0) # Returns empty arrays
    fold_data_x = X.copy() # Copies data for modification
    fold_data_y = y.copy()

    folds_X = [] # Declaration of empty arrays
    folds_y = []

    for _ in range(n_folds): # For every fold
        folds_X.append(fold_data_x[0:fold_size]) # Appends the first selection of data to relevant array
        fold_data_x = fold_data_x.drop(fold_data_x.index[0:fold_size]) # Removes this from copy of data

        # Repeats for y
        folds_y.append(fold_data_y[0:fold_size])
        fold_data_y = fold_data_y.drop(fold_data_y.index[0:fold_size])

    return folds_X, folds_y # Returns two folds
```

As a basis, the k-fold function designed allows for any number of folds as long as the number does not exceed the number of elements within the array. Otherwise, this would return k empty arrays. This function simply iterates through the shuffled array, appends the first set of results to separate arrays (for X and y), then removes those values from the array. Any remainder values are not included in the output folds. These folds are then used to train both models, the neural network being trained as follows:

```python
epochs = 50 # Number of epochs
ann.compile(optimizer='adam', loss='mse', metrics=['accuracy']) # Compiles the network

why_X = pd.DataFrame(X[0]) # Creates dataframe from first list in X
y_stretch = np.array(y[0]) # Creates array from first list in y

for i in range(1, n_folds): # For every fold not including those used for dataframe initialisation
    y_stretch = np.append(y_stretch, y[i]) # Adds y[i] to y
    why_X = pd.concat([why_X, X[i]]) # Concatenates the X dataframe and X[i]

why_y = pd.DataFrame(y_stretch) # Creates dataframe from y array

for i in range(n_folds): # For every fold
    cur_X = why_X.copy() # Creates copies for modification
    cur_y = why_y.copy()

    test_X = X[i] # Gets test values for X and Y
    test_y = y[i].astype(int)

    train_X = np.array(cur_X.drop(index=test_X.index)) # Drops test indices from dataframe copies to generate training values.
    train_y = np.array(cur_y.drop(index=test_y.index))
    train_y = train_y.squeeze().astype(int) # Formats y

    test_X = np.array(test_X) # Transforms test_X into array

    history = ann.fit(train_X, train_y, epochs=epochs) # Fits current folds
    scores = ann.evaluate(test_X, test_y, verbose=0) # Evaluates

    print("Accuracy: %.2f%%" % (scores[1]*100)) # Prints accuracy
```

This creates two dataframes: one for X and y each. These dataframes are copied to allow for modification at the start of each loop, which runs until every fold has been processed. Testing data is first sampled from the original k-fold list, the indices of which are used to remove said values form the copied dataframes, in turn creating the training data. These sets of data are then used to train the network.



```
clf_folded.fit(train_X, train_y) # Trains forest

pred = clf_folded.predict(test_X) # Gets predictions
acc = accuracy(test_y, pred) # Calculates accuracy

print(f"Accuracy: {acc}") # Prints accuracy
```

The same code was used to train the random forests algorithm with the following changes:

Number of Trees	**Mean Accuracy**	**Time (seconds)**	**Number of Epochs**	**Mean Accuracy**	**Time (seconds)**
20	76.92%	34	50	92.3%	6
500	84.61%	882	500	84.61%	48
10,000	69.23%	14,701	1000	69.23%	105

From these results I believe that the superior parameters for both methods to be 20 trees for the random forest algorithm, and 50 epochs for the artificial neural network. The minimum number of samples for the random forest algorithm has been set to 5.

Overall, I believe that the most effective machine learning model within this problem scope is the artificial neural network. Although it receives similar results to random forests when using the training and test split method, the k-fold cross validation highlights the strength of the neural network, as it reaches a near perfect accuracy (12 out of 13 correct predictions) within 50 epochs at under 7 seconds. Whilst a random forests algorithm has it's uses, I believe that it takes too long to construct one using the functionality provided. Additionally, the high recall score makes the neural network a good option for this choice, for the reasons discussed within section 2. Although the random forests algorithm does approach quite a high recall score, the neural network approaches a near perfect, sometimes perfect, score for recall, which is of utmost importance for this background.

Interpretability has little variation in both results, but I would consider the random forests algorithm to be more interpretable than the network, as decision nodes can be evaluated much more easily compared to the weights and biases of the neural network. However, I do not think that this is a strong enough point for the random forests to weigh out as the more effective machine learning model.

References

Bibal, A. and Frénay, B., 2016, April. Interpretability of machine learning models and representations: an introduction. In *ESANN*.
Calle, E.E. and Thun, M.J., 2004. Obesity and cancer. *Oncogene*, *23*(38), pp.6365-6378.
Cao, X.H., Stojkovic, I. and Obradovic, Z., 2016. A robust data scaling algorithm to improve classification accuracies in biomedical data. *BMC bioinformatics*, *17*(1), p.359.

Doshi-Velez, F. and Kim, B., 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.
Hawkins, D.M., 2004. The problem of overfitting. *Journal of chemical information and computer sciences*, *44*(1), pp.1-12.
Shajahaan, S.S., Shanthi, S. and ManoChitra, V., 2013. Application of data mining techniques to model breast cancer data. *International Journal of Emerging Technology and Advanced Engineering*, *3*(11), pp.362-369.

Jabbar, H. and Khan, R.Z., 2015. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, pp.163-172.

Karlik, B. and Olgac, A.V., 2011. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, *1*(4), pp.111-122.
Li, X. and Claramunt, C., 2006. A spatial entropy-based decision tree for classification of geographical information. *Transactions in GIS*, *10*(3), pp.451-467.
Lipton, Z.C., 2018. The Mythos of Model Interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, *16*(3), pp.31-57.
MacArthur, C. and Smith, A., 1984. Factors associated with speed of diagnosis, referral, and treatment in colorectal cancer. *Journal of Epidemiology & Community Health*, *38*(2), pp.122-126.

Manheim, D. and Garrabrant, S., 2018. Categorizing variants of Goodhart's Law. *arXiv preprint arXiv:1803.04585*.

Mitchell, T., 1997. Introduction to machine learning. *Machine Learning*, *7*, pp.2-5.

Morstatter, F., Wu, L., Nazer, T.H., Carley, K.M. and Liu, H., 2016, August. A new approach to bot detection: striking the balance between precision and recall. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (pp. 533-540). IEEE.

Nuttall, F.Q., 2015. Body mass index: obesity, BMI, and health: a critical review. *Nutrition today*, *50*(3), p.117.

Petticrew, M.P., Sowden, A.J., Lister-Sharp, D. and Wright, K., 2000. False-negative results in screening programmes: systematic review of impact and implications. *Health technology assessment (Winchester, England)*, *4*(5), pp.1-120.

Siegel, R., Ma, J., Zou, Z. and Jemal, A., 2014. Cancer statistics, 2014. *CA: a cancer journal for clinicians*, *64*(1), pp.9-29.
Van der Aalst, W.M., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E. and Günther, C.W., 2010. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, *9*(1), p.87.

Vellido, A., Romero, E., Julia-Sape, M., Majós, C., Moreno-Torres, A., Pujol, J. and Arús, C., 2012. Robust discrimination of glioblastomas from metastatic brain tumors on the basis of single-voxel 1H MRS. *NMR in Biomedicine*, *25*(6), pp.819-828.

White, M.C., Holman, D.M., Boehm, J.E., Peipins, L.A., Grossman, M. and Henley, S.J., 2014. Age and cancer risk: a potentially modifiable relationship. *American journal of preventive medicine*, *46*(3), pp.S7-S15.