Samuel Paynter
19695600
CMP3751M

Machine Learning Assignment 1

Task 1

1. To make the polynomial regression, I, first of all, created a helper function called get data matrix which takes in the x train and the degree as parameters. It iterates until it gets to the same number as the degree plus 1 and takes the x train data with the index of the loop as the power as a 1D array. These 1D arrays get stacked into a 2D array which is returned from the function.

```
In [177]: #Creating function
          def getDataMatrix(x, degree):
              #reutrns an array with a new shape
              X = np.ones(x.shape)
              #iterating until the number of the degree + 1
              for i in range(1, degree + 1):
                  #staking 1d arrays into a 2d arrays
                  X  = np.column_stack((X, x ** i))
              return X
```

   Then, in the polynomial regression function, the 2D array is then transposed and multiplied by itself. This is assigned to the variable XX. To get the value of the parameters, NumPy linear algebra solve is used with the XX variable and the transpose of 2D array multiplied by the y train data. This is returned from the function.

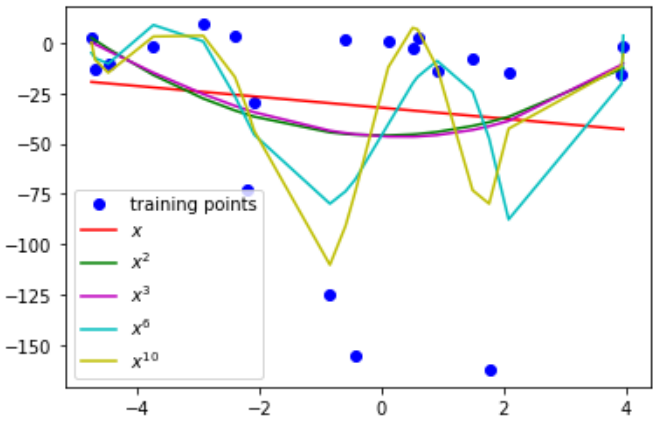
```
In [178]: def pol_regression(features_train, y_train, degree):
              #calling function and assigning it to the variable x
              X = getDataMatrix(features_train, degree)
              #transposing and using the .dot() function with the variable x
              XX = X.transpose().dot(X)
              #calculating the parameters
              parameters = np.linalg.solve(XX, X.transpose().dot(y_train))

              return parameters
```

2. To plot the graph, the polynomial regression function is called and assigned to the variable w. The result of the data matrix function is assigned to the variable xTest. Using these variables, xtest and w are multiplied together to make ytest. Then the X column of the data is plotted on the x-axis and ytest on the y axis. Based on the lines plotted on the graph, the polynomial regression that I would choose is the one with a degree of 10. This is because the line is the closest to the training points from the original data.

```
In [212]: #Splitting the data into x and y based on the two columns in the data
          X = data['x']
          y = data['y']
          #sorting the data
          X = np.sort(X, 0)
```

Samuel Paynter
19695600
CMP3751M

In [213]:
```
plt.figure()
#plottin data points
plt.plot(X, y, 'bo')

#plotting the line with the degree as 1
w1 = pol_regression(X, y, 1)
Xtest1 = getDataMatrix(X, 1)
ytest1 = Xtest1.dot(w1)
plt.plot(X, ytest1, 'r')

#plotting the line with degree as 2
w2 = pol_regression(X, y, 2)
Xtest2 = getDataMatrix(X, 2)
ytest2 = Xtest2.dot(w2)
plt.plot(X, ytest2, 'g')

#plotting the line with the degree as 3
w3 = pol_regression(X, y, 3)
Xtest3 = getDataMatrix(X, 3)
ytest3 = Xtest3.dot(w3)
plt.plot(X, ytest3, 'm')

#plotting the line with the degree as 6
w4 = pol_regression(X, y, 6)
Xtest4 = getDataMatrix(X, 6)
ytest4 = Xtest4.dot(w4)
plt.plot(X, ytest4, 'c')

#plotting the line withh the degree as 10
w5 = pol_regression(X, y, 10)
Xtest5 = getDataMatrix(X, 10)
ytest5 = Xtest5.dot(w5)
plt.plot(X, ytest5,  'y')

plt.legend(('training points', '$x$', '$x^2$', '$x^3$', '$x^6$', '$x^1$$^0$'))
```



3. To calculate the RMSE, first of all, I created a function that takes parameters, x, y and the degree as parameters. The get data matrix function is called and assigned to the variable xTest. This is then used to calculate the prediction by using the dot function with xTest and parameters. Then I calculated the mean square deviation. This is done by subtracting y and

the prediction, then squaring it, then getting the mean of these values. Finally, to get the RMSE I got the square root of the MSE. This is returned in the function.

In [205]:
```
def eval_pol_regression(parameters, x, y, degree):
    #calling the function
    xTest = getDataMatrix(x, degree)
    #calculating the prediction
    yPredict = xTest.dot(parameters)

#calculating the mean squared error
    MSE = np.square(np.subtract(y, yPredict)).mean()
    #calculating root mean squared error
    RMSE = np.sqrt(MSE)

    return RMSE
```

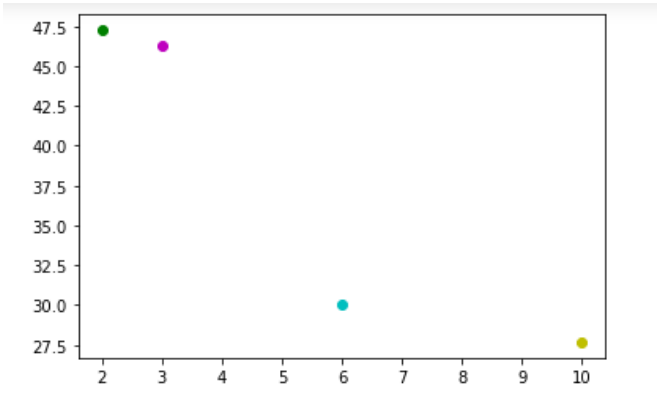
To plot the RMSE against the degree, I, first of all, split the data up into x train and test as well as y test and train with 30% test data and 70% train data. With this, I then call the polynomial regression using the x and y train data. This is then used for evaluation polynomial regression function using either the train data or test data. This is then plotted through matplotlib.

In [192]:
```
from sklearn.model_selection import train_test_split
# creating train and test data with a 70-30 split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

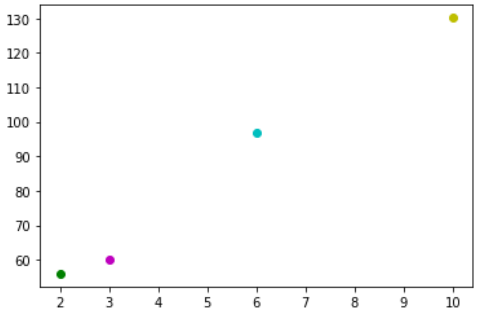
In [210]:
```
wTrain2 = pol_regression(x_train, y_train, 2)
#calculating RMSE for train data with the degrees as 2
train2 = eval_pol_regression(wTrain2, x_train, y_train, 2)
#calculating RMSE for test data with the degrees as 2
test2 = eval_pol_regression(wTrain2, x_test, y_test, 2)
#plotting train data for degree 2
plt.plot(2, train2,  'go')

wTrain3 = pol_regression(x_train, y_train, 3)
#calculating RMSE for train data with the degrees as 3
train3 = eval_pol_regression(wTrain3, x_train, y_train, 3)
#calculating RMSE for test data with the degrees as 3
test3 = eval_pol_regression(wTrain3, x_test, y_test, 3)
#plotting train data for degree 3
plt.plot(3, train3,  'mo')

wTrain4 = pol_regression(x_train, y_train, 6)
#calculating RMSE for train data with the degrees as 6
train4 = eval_pol_regression(wTrain4, x_train, y_train, 6)
#calculating RMSE for test data with the degrees as 6
test4 = eval_pol_regression(wTrain4, x_test, y_test, 6)
#plotting train data for degree 6
plt.plot(6, train4,  'co')

wTrain5 = pol_regression(x_train, y_train, 10)
#calculating RMSE for train data with the degrees as 10
train5 = eval_pol_regression(wTrain5, x_train, y_train, 10)
#calculating RMSE for test data with the degrees as 10
test5 = eval_pol_regression(wTrain5, x_test, y_test, 10)
#plotting train data for degree 10
plt.plot(10, train5,  'yo')
```

The graph for the training data RMSE shows that the greater the degree the lower the RMSE. This is good because it shows that the greater the degree, the better the fit to the model is. However, for the test data, it was the opposite of train data. As the degree increased, the RMSE increased. This is called overfitting. Overfitting is where the model performs well on the training data, however, it does not perform well on the evaluation. This is because the

model learns the detail in the training data that negatively impacts the performance on the data. Underfitting is where a model performs poorly on the training data. This is due to the model not capturing the relationship between the inputs and the target values. Underfitting is shown in these graphs in the lower degrees where the line in the polynomial regression does not fit the data points very well.




```
In [211]: #plotting for test data
          plt.plot(2, test2,  'go')
          plt.plot(3, test3,  'mo')
          plt.plot(6, test4,  'co')
          plt.plot(10, test5,  'yo')
Out[211]: [<matplotlib.lines.Line2D at 0x225a28345e0>]
```



Task 2

1. For the k-means algorithm, I implemented four functions. One to calculate the Euclidean distance, one to initialise the centroids, one to assign the clusters and finally one called k-means which combines all these functions. To calculate Euclidean distance, the sum of vector 1 subtract vector 2 is squared. This result is then raised by the power of 0.5. This value is returned from the function. The assign clusters function uses a nested loop to iterate over the Euclidean distance function, this is then added to a distances array. This array is then used to enumerate to get the closest centroid to a specific cluster.

Samuel Paynter
19695600
CMP3751M

In [142]:
```
def assign_clusters(centroids, dataset):
    #creating an array
    clusters = []

    #Looping the length of the dataset shape
    for i in range(dataset.shape[0]):
        #creating an array
        distances = []
        #looping through centroids
        for centroid in centroids:
            #calling the eclidean distance and the result is appended to the distances array
            distances.append(compute_euclidean_distance(centroid, dataset[i]))
        #adding values to clusters array
        cluster = [z for z, val in enumerate(distances) if val == min(distances)]
        clusters.append(cluster[0])



    return clusters
```

In [88]:
```
def compute_euclidean_distance(vec_1, vec_2):
    #calculating euclidean distance
    distance = (sum(vec_1 - vec_2)**2)**0.5

    return distance
```

To initialise the centroids concatenates the two columns used for the plot, the cluster array, and the cluster that it belongs to. This is then used to get the new centroids by looping through them and calculating the mean of each cluster. The assign clusters and initialise centroids functions are then called in the k-means function in a loop to repeatedly calculate new centroids and clusters.

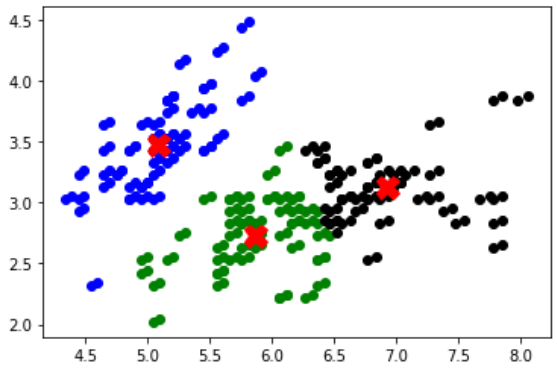
In [146]:
```
def initialise_centroids(dataset, k):
    #creating centroids array
    centroids = []
    #concatenating arrays together
    cluster_df = pd.concat([pd.DataFrame(dataset), pd.DataFrame(k, columns=['cluster'])], axis=1)
    #looping through the concatenated arrays
    for c in set(cluster_df['cluster']):
        #assigning the currentcluster to a variable
        current_cluster = cluster_df[cluster_df.columns[:-1]]
        getting the mean of the clusters
        cluster_mean = current_cluster.mean(axis=0)
        #appending the mean cluster to the centroids
        centroids.append(cluster_mean)


    return centroids
```

In [127]:
```
def kmeans(dataset, k):
    #assigning values to centroids array
    centroids = [dataset[i+2] for i in range(k)]
    #Using clusters array function and assigning it to a variable
    cluster_assigned = assign_clusters(centroids, dataset)
    #looping through to 20
    for i in range(20):
        #calling initialise centroids function
        centroids = initialise_centroids(dataset, cluster_assigned)
        #calling assign cluster function
        cluster_assigned = assign_clusters(centroids, dataset)

    return centroids, cluster_assigned
```

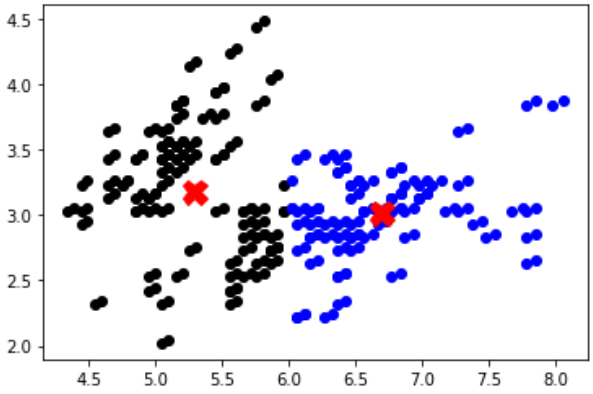
In [171]:
```
#reading csv file
df = pd.read_csv("Task2 - dataset - dog_breeds.csv")
#copying height and leg length to a variable
df2 = df[['height', 'leg length']].copy(deep=True)
df2.dropna(axis=0, inplace=True)
#sorting alues
df2.sort_values(by=['height', 'leg length'], inplace=True)
#creating a numpy array
dataset = np.array(df2)
```

Samuel Paynter
19695600
CMP3751M

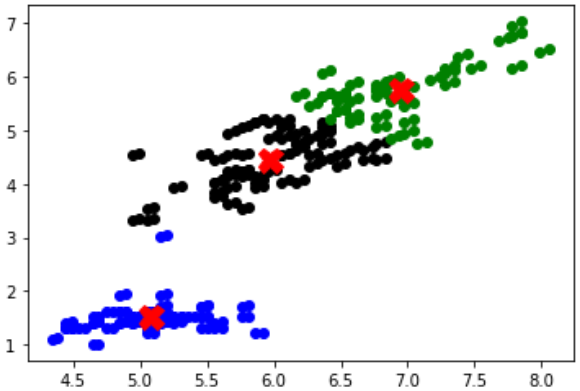
2. The graph below shows the plot of height on the x-axis and tail length on the y-axis for 3 clusters.



The graph below shows height on the x-axis and tail length on the y for 2 clusters.

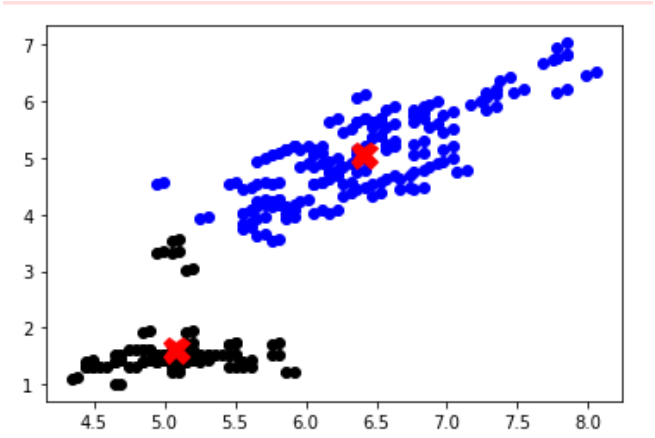


The graph below shows height on the x axis and leg length on the y axis for 3 clusters.



The graph below shows height on the x axis and leg length on the y axis for 2 clusters.

Samuel Paynter
19695600
CMP3751M



Task 3

1. To calculate the mean, minimum and maximum, I first used pandas to import the dataset, then dropped the participant condition column as it is the only column with a string value and therefore cannot have a mean, minimum or maximum value. After that, I used the mean, minimum and maximum functions from pandas on the remaining columns. I then decided to normalise the data because an artificial neural network, which the data is being used for, is an algorithm that does not make assumptions about the distributions of the data. As well as that, it is also good for when you don't know the distribution of the data. To generate the two plots, I used the seaborn boxplot function and for the density plot, I used the 'kdeplot' function.

```
|: #Task 3

   #importing file as df
   df = pd.read_csv("Task3 - dataset - HIV RVG.csv")
   #outputting file
   df
```

```
In [166]: #Dropping the column with string value in order to get mean,min and max values
          df2 = df.drop('Participant Condition', axis = 1)
          df2
```

```
In [16]: #Outputting mean of each column
         df2.mean()
```

```
In [17]: #Outputting minimum values of each column
         df2.min()
```

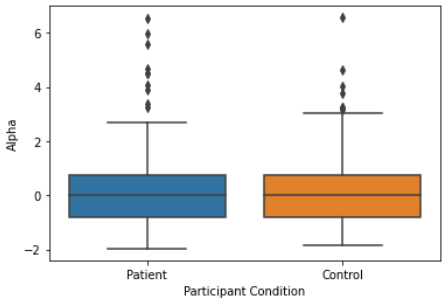
```
In [18]: #Outputting maximum values of each column
         df2.max()
```

```
In [19]: #Using mean normalisation
         df2_norm = (df2 - df2.mean())/df2.std()

         df2 = pd.DataFrame(df2_norm)
         df2
```

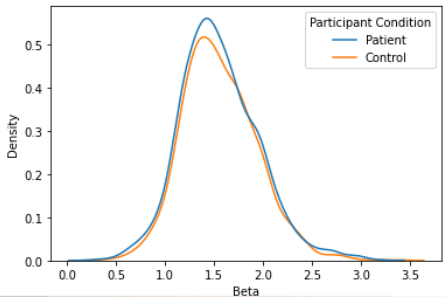
Samuel Paynter
19695600
CMP3751M

```
In [20]: status = df['Participant Condition']
         #Appending the participantcondition back onto the normalised data
         df2 = df2.join(status)
```

```
In [22]: #importing the seaborn library in orderto plot the graphs
         import seaborn as sns
         #plotting the boxplot wit participant condition as the x axis and alpha as the y
         sns.boxplot(x= 'Participant Condition', y= 'Alpha', data= df2)
```




```
In [23]: #Plotting the density plot with beta along the x axis
         sns.kdeplot(data=df, x = 'Beta', hue= 'Participant Condition')
Out[23]: <AxesSubplot:xlabel='Beta', ylabel='Density'>
```



2. For both the neural network and random forest classifier, I used the sklearn machine learning library. I choose to use this due to its robustness, meaning that you can do a lot with it from pre-processing and splitting data to perceptron and regression. Therefore, the use of its built-in functions allows for less and more efficient code compared to the use of other machine learning libraries or coding these two classifiers from scratch. To prepare the data for the classifiers, I, first of all, appended the participant condition column back onto the data. I then assigned the participant condition column to the variable ‘y’ and the rest to the variable ‘x’. Then using shuffle from sklearn to shuffle both x and y. Then using sklearn again to split the data 90% train and 10% test data into the variables x train, x test, y train and y test. I then created a function for the artificial neural network which takes an integer value as an input. This will be used to determine the number of epochs. Inside the function, mlp classifier from sklearn is used to create an artificial neural network that includes two hidden layers, with 500 neurons in each layer, and a logistic activation is used. This is then fitted with the x and y train data. A variable called predicted uses the predict function from sklearn with the x test data to make a prediction. Finally, the accuracy is calculated using metrics accuracy score from sklearn with the y test data and the predicted variable as the parameters. The accuracy is then returned from the function.

Samuel Paynter
19695600
CMP3751M

In [24]:
```
#Assigning all columns apart from particpant condition to the varaible x
x = df2[['Image number', 'Bifurcation number', 'Artery (1)/ Vein (2)', 'Alpha', 'Beta', 'Lambda', 'Lambda1', 'Lambda2']]
```

In [25]:
```
#Importing pre preocessing from sklearn
from sklearn import preprocessing
#Assignning participant condition to the variable y
y = df2["Participant Condition"]
#Outputting the first 5 y values
y[0:5]
```

Out[25]:
```
0    Patient
1    Patient
2    Patient
3    Patient
4    Patient
Name: Participant Condition, dtype: object
```

In [26]:
```
#importing shuffle from sklearn inorder to shuffle the data
from sklearn.utils import shuffle
#Shuffling both x and y
x, y = shuffle(x,y, random_state = 0)
```

In [28]:
```
#impoorting train test split inorder to split the data into training and testing
from sklearn.model_selection import train_test_split
#Splitting the data into 90% training 10% test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.1)
#Printing the shape of x and y train and test
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(2892, 8)
(322, 8)
(2892,)
(322,)
```

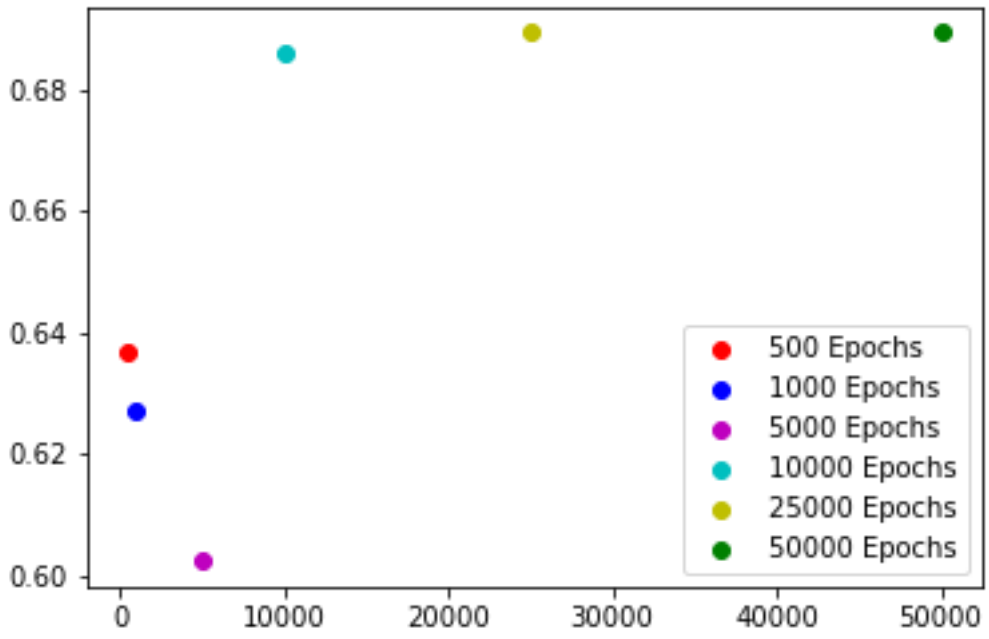
In [29]:
```
#importing both metrics and MLPClassifier from sklearn
from sklearn.neural_network import MLPClassifier
from sklearn import metrics
#creating a function fro the artificial neural networktaking the number of epochs as a parameter
def ann(epochs):

    mlp = MLPClassifier(hidden_layer_sizes=(500,500), activation='logistic', solver= 'adam',  max_iter=epochs)
    mlp.fit(x_train, y_train)

    predicted = mlp.predict(x_test)
    accuracy = metrics.accuracy_score(y_test, predicted)

    return accuracy
```

To plot the graph for the artificial neural network, I used the matplotlib library. I called the function with the number of epochs in the parameters and assigned it to a variable, the used the number of epochs as the x-axis and the accuracy as the y axis. I chose to plot 6 points, with the epochs being 500, 1000, 5000, 10000, 25000 and 50000. The results of the graph are presented below. The average for these is 65.83%.

Samuel Paynter
19695600
CMP3751M

For the random forest classifier, I, first of all, created a function that takes the minimum samples required to be at a leaf node as a parameter. I then used random forest classifier

```
plt.figure()
#outputting a graph with the number of epochs on the x axis and the accuracy on the y
epochs1 = 500
accuracy1 = ann(epochs1)
plt.plot(epochs1, accuracy1, 'ro')

epochs2 = 1000
accuracy2 = ann(epochs2)
plt.plot(epochs2, accuracy2, 'bo')

epochs3 = 5000
accuracy3 = ann(epochs3)
plt.plot(epochs3, accuracy3, 'mo')

epochs4 = 10000
accuracy4 = ann(epochs4)
plt.plot(epochs4, accuracy4, 'co')

epochs5 = 25000
accuracy5 = ann(epochs5)
plt.plot(epochs5, accuracy5, 'yo')

epochs6 = 50000
accuracy6 = ann(epochs6)
plt.plot(epochs6, accuracy6, 'go')

plt.legend(('500 Epochs', '1000 Epochs', '5000 Epochs', '10000 Epochs', '25000 Epochs', '50000 Epochs'))

plt.savefig('ann.png')
```



from sklearn with 1000 trees. Then using the fit function from sklearn to train the classifier with the x and y train data. Then predicting using the x test data. Finally to calculate the accuracy by using metrics accuracy score from sklearn with the y test and predicted as parameters. The accuracy is then returned from the parameters. The accuracy of the minimum number of samples to be a leaf node as 5 was 78.57142857142857%. The accuracy of the minimum number of samples to be a leaf node as 10 was 75.77639751552795%.

Samuel Paynter
19695600
CMP3751M

In [31]:
```
#Importing random forest classifier from sklearn
from sklearn.ensemble import RandomForestClassifier
#creating a function for the random forest classifier taking the minimum number of samples to be a leaf node as the parameter
def rfc(samples):
# Assinging the random forest classifier to the variable clf with the number of tress as 100 and the min sample leaf as the paran
    clf = RandomForestClassifier(n_estimators=1000, min_samples_leaf=samples)
    #Training the classifier using the x and y training data
    clf.fit(x_train, y_train)
    #predicitng using the x tesst data
    predicted = clf.predict(x_test)
    #calculating the accuracy using the y test data and the predicted values
    accuracy = metrics.accuracy_score(y_test, predicted)
    #return the accuracy value
    return accuracy
```

In [42]:
```
# printing the accuracy values by calling the rfc function and using 5 and 10 as the min samples leaf
print("\nAccuracy with 5 as the minimum number of samples to be a leaf node: ", rfc(5))
print("\nAccuracy with 10 as the minimum number of samples to be a leaf node: ", rfc(10))
```

```
Accuracy with 5 as the minimum number of samples to be a leaf node:  0.7857142857142857

Accuracy with 10 as the minimum number of samples to be a leaf node:  0.7577639751552795
```

3. For the cross-validation using 10 folds, I used the KFold function from sklearn. The artificial neural network is set up like before however this time there are 2 hidden layers with either 50, 500 or1000 neurons in each layer. I chose to have 50000 epochs for each artificial neural network. Then looping through the split KFold for the x data, the artificial neural network is trained using the KFold x and y train, then predicted using the KFold x test data. With this, the accuracy is calculated with the y test and the predicted. The random forest classifier is done like the artificial neural network apart from using the random forest classifier function from sklearn with the number of trees varying from 50,500 and 10000 and the minimum number of samples to be a leaf node being 10. The training, predicting and accuracy is done the same as in the artificial neural network.

In [33]:
```
from sklearn.model_selection import KFold
#converting x and y to numpy
x = x.to_numpy()
y = y.to_numpy()

def ann1(epochs):
    #creating 10 folds
    kf = KFold(n_splits=10)
    #creating ANN with 50 neurons in each layer
    mlp = MLPClassifier(hidden_layer_sizes=(50,50), activation='logistic', solver= 'adam',  max_iter=epochs)
    #looping through the kfolds
    for train_indices, test_indices in kf.split(x):
        #training model using the kfolds
        mlp.fit(x[train_indices], y[train_indices])
        #calculating the predicted value using the kfolds
        predicted = mlp.predict(x[test_indices])
        #calculating the acuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

Samuel Paynter
19695600
CMP3751M

In [34]:
```
def ann2(epochs):
    #creating 10 folds
    kf = KFold(n_splits=10)
    #Creating ANN with 500 neurons in each layer
    mlp = MLPClassifier(hidden_layer_sizes=(500,500), activation='logistic', solver= 'adam',  max_iter=epochs)
    #Looping through kfolds
    for train_indices, test_indices in kf.split(x):
        #training ANN
        mlp.fit(x[train_indices], y[train_indices])
        #calculating the predicted value
        predicted = mlp.predict(x[test_indices])
        #Calculating accuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

In [35]:
```
def ann3(epochs):
    #Creating 10 folds
    kf = KFold(n_splits=10)
    #creating ANN with 1000 neurons in each layer
    mlp = MLPClassifier(hidden_layer_sizes=(1000,1000), activation='logistic', solver= 'adam',  max_iter=epochs)
    #looping through kfolds
    for train_indices, test_indices in kf.split(x):
        #training the ANN
        mlp.fit(x[train_indices], y[train_indices])
        #calculating the predicted value
        predicted = mlp.predict(x[test_indices])
        #Calculating the accuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

In [37]:
```
#calling each neural network with 50000 epochs
nn1 = ann1(50000)
nn2 = ann2(50000)
nn3 = ann3(50000)
#outputtingg the accuracy of each neuracl netwrok
print(nn1)
print(nn2)
print(nn3)
#calculating the mean
ann_mean = (nn1 + nn2 + nn3)/3
#outputting the mean
print(ann_mean)
```

```
0.6728971962616822
0.5981308411214953
0.6635514018691588
0.6448598130841121
```

For the artificial neural network with 50 neurons in each layer, the accuracy was 67.28971962616822%. For 500 neurons, the accuracy was 59.81308411214953%. For 1000 neurons, the accuracy was 66.35514018691588%. The mean of the artificial neural network's accuracy was 64.48598130841121%. I believe the best parameter for the artificial neural network is the number of epochs. The average for the number of epochs was just higher at 65.83% compared to the number of neurons per layer which was 64.49%. However, based on the graph, for the number of epochs over 10000, the accuracy increased as the number of epochs did. Compared to the number of neurons in each layer where there seems to be no pattern.

Samuel Paynter
19695600
CMP3751M

In [38]:
```
def rfc1(samples):
   #creating 10 folds
    kf = KFold(n_splits=10)
    #creating a random forest classifier with 50 trees
    clf = RandomForestClassifier(n_estimators=50, min_samples_leaf=samples)
    #looping through the kfolds
    for train_indices, test_indices in kf.split(x):
        #training the RFC
        clf.fit(x[train_indices], y[train_indices])
        #calculating the predicted value
        predicted = clf.predict(x[test_indices])
        #calculating the accuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

In [39]:
```
def rfc2(samples):
   #creating 10 kfolds
    kf = KFold(n_splits=10)
    #creating a RFC with 500 trees
    clf = RandomForestClassifier(n_estimators=500, min_samples_leaf=samples)
    #looping through the kfolds
    for train_indices, test_indices in kf.split(x):
        #training the RFC
        clf.fit(x[train_indices], y[train_indices])
        #calculating the predicted value
        predicted = clf.predict(x[test_indices])
        #calculating the accuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

In [40]:
```
def rfc3(samples):
   #creating 10 kfolds
    kf = KFold(n_splits=10)
    #creating a RFC with 1000 trees
    clf = RandomForestClassifier(n_estimators=10000, min_samples_leaf=samples)
    #Looping through the kfolds
    for train_indices, test_indices in kf.split(x):
        #training the RFC
        clf.fit(x[train_indices], y[train_indices])
        #calculating the predicted value
        predicted = clf.predict(x[test_indices])
        #calculating the accuracy
        accuracy = metrics.accuracy_score(y[test_indices], predicted)

    return accuracy
```

In [41]:
```
#Calling the RFC functions witht the minimum samples to be a leaf node as 10
rf1 = rfc1(10)
rf2 = rfc2(10)
rf3 = rfc3(10)
#outputting the accuracy of these functions
print(rf1)
print(rf2)
print(rf2)
#calculating the mean accuracy
rf_mean = (rf1 + rf2 + rf3)/3
#outputting the mean
print(rf_mean)
```

```
0.8099688473520249
0.8193146417445483
0.8193146417445483
0.8172377985462097
```

For the random forest classifier with 50 trees, the accuracy was 80.99688473520249%. For 500 trees, the accuracy was 81.93146417445483%. Finally, for 10000 trees, the accuracy was 81.93146417445483%. This makes the mean of the random forest classifier to be

Samuel Paynter
19695600
CMP3751M

81.72377985462097%. The best parameter for the random forest classifier seems to be the number of trees. This was because the average accuracy for the number of trees was 81.7% compared to 77.1739130435% for the minimum number of samples to be a leaf node.

Based on the information I have collected; the random forest classifier was better than the artificial neural network. This is because, for both tests for each parameter, the random forest classifier had an accuracy of over 75% compared to the artificial neural network with around 65%. This would not be good enough when monitoring a patient with HIV as there are still 35% of patients who could be identified as a false positive or false negative which may put the patient in danger.