Charlie Elliott
19694799
Machine Learning: CMP3751M

Task 1:

Section 1.1 Polynomial Regression implementation:

getPolynomialDataMatrix:

```
def getPolynomialDataMatrix(x, degree):
    # x: training x values (Numpy 1D array)
    # degree: polynomial degree integer (integer)
    # x = x[:-12]
    #Creates new array degExponent containing only 1s
    degExponent =  np.ones(x.shape)

    #creates an degree number of columns each containing the exponent of x uptil degree
    for i in range(1,degree + 1):
        degExponent= np.column_stack((degExponent, x ** i))

    # return exponent array and delete first column of ones
    return np.delete(degExponent,0,1)
```

this function takes in the x input and the and the degree. This function performs the necessary feature expansion to the nth degree by first generating a numpy
using this it creates and NumPy matrix with the same number of columns as the degree argument that contain the exponential of x (see Figure 2)

```
[[  -4.70123789   22.10163771 -103.90505666]
 [  -4.48532797   20.11816697  -90.23657696]]
```

FIGURE 1: SAMPLE OUTPUT OF DEGEXPONENT VARIABLE USING THE 3RD DEGREE

pol_regression Function:

```
# so to get training data to plot needed need to add an arguement that makes it plot accurately
def pol_regression(features_train, y_train,degree):
    # features_train: x values (Numpy 1D array)
    # y_train: y values (Numpy 1D array)
    # degree: polynomial degree(integer)

    # a 0 degree polynomial is a constant value
    # this calculates the mean of the data as that is an approximation of all the data points
    if(degree == 0):
        # Calculates the mean of the whole training set and repeats it for the length y
        parameters = np.repeat(np.mean(y_train),features_train.size)

        return parameters


    else:
        X = getPolynomialDataMatrix(features_train, degree)
        # transpose and dot product against original matrix
        XX = X.transpose().dot(X)
        Y= X.transpose().dot(y_train)
        # solves the matrix
        #inverts XX then dots it with Y producing the alpha coeffeicients
        return linalg.solve(XX,Y)
```

The pol_regression function is the first called, it calculates the alpha coefficients of the features presented. First it checks if the degree is 0, if so it calculates the mean of the data and repeats it for the length of the features (Polynomial Regression unknown) and return's this as parameters. If it's not 0 it first calculates the exponent array by calling getPolynomialDataMatrix storing it as X. It then calculates the XX by dot producting X with X^T (see figure 2). This result in a matrix that follows that follows the below structure:

N	x	…	x^k
x	x^2	…	x^{k+1}
…	…	…	…
x^k	x^{k+1}	…	x^{2k}

```
[[  20.          -13.86328553  146.05154296]
 [ -13.86328553  146.05154296 -272.63897143]
 [ 146.05154296 -272.63897143 2273.40387611]]
```

FIGURE 2 EXAMPLE OUTPUT FROM THE 2ND DEGREE

It then calculates the Y matrix by dot producting X^T and y_train, this result in a matrix that follows this structure:

y	xy	…	$x^k y$

Then finally Y and XX are solved using numpy.linalg.solve() to produce a list of alpha coefficients (See figure 3)

Charlie Elliott
19694799
Machine Learning: CMP3751M

```
[-4.96408881e+00  8.96348689e+00 -2.40578299e+00 -2.76623260e+00
  1.24780599e+00  5.59420746e-01 -2.00071166e-01 -4.96180474e-02
  7.89761149e-03  1.65643310e-03 -9.47507952e-06]
```

FIGURE 3: SAMPLE OUTPUT OF ALPHA COEFFICIENTS FOR THE 10TH DEGREE

Section 1.2 Regress the polynomial to 0,1,2,3,6,10 degrees

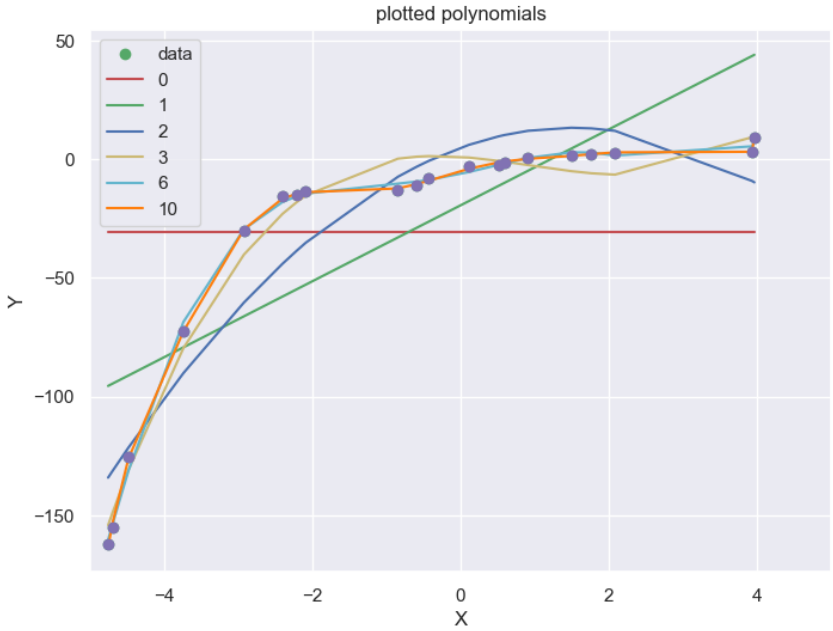

FIGURE 4: PLOTTED POLYNOMIALS ON GRAPH

```
degs = [0,1,2, 3, 6, 10]

pol_return = []

for i in degs:
    output = pol_regression(x_train,y_train,i)
    l = getPolynomialDataMatrix(x_train,i)
    if i == 0:
        l = output
    else:
        l = l.dot(output)
    pol_return.append(l)
```

To display the final regression lines the alpha coefficients must be dotted to the output of getPolynomialDataMatrix, this gives the necessary y values for plotting.

The zero degree is a constant value it is drastically underfits the data, every data is point bar one is a significant outlier; it in no way follows the trend of the data.
First degree still underfits the data as its incapable of capturing the curve of the data and leaves the bottom three and right most five data points are significant outliers; it's still not capable of accurately capturing the trend of the data.
The second degree is beginning to capture the curve of the data however it still under fits the data very slightly with data points in the middle and at the bottom are considerable outliers; however, its starting to capture the trend of the data.
The third degree has now managed to capture the curves of the data and has no significant outliers, however it still slightly underfits the last few data points
After 6 degrees the data has begun to overfit the data there are two small outliers
The 10th degree the data drastically overfits the data with no outliers at all

Section 1.3 Evaluation:

Before any evaluation can be completed, the data is split into testing and training data this is done by the sckitlearn function Train_Test_split, with shuffle enabled and test_size set to 0.3.

Charlie Elliott
19694799
Machine Learning: CMP3751M

Eval_pol_regression function:

```
def eval_pol_regression(parameters, x, y, degree):
    # Parameters: return of pol_regression function (Numpy 1D array)
    # x: x values (Numpy 1D array)
    # y: y values (Numpy 1D array)
    # degree: polynomial degree (Integer)
    testMatrix = getPolynomialDataMatrix(x,degree)

    if degree == 0:
        predictedy= testMatrix
    else:
        predictedy = testMatrix.dot(parameters)

    mse = np.mean((predictedy-y)**2)
    rmse = math.sqrt(mse)
    return rmse
```

The eval_pol_regression function computes the root mean squared error for a given set of parameters compared to a given set of x and y values. It works by calculating how far the given points are from the regression line (Statistics how to, 2022). It does this by getting an exponential matrix of degree and x, then dotting it with the parameters to produce predicted Y points. The distance between these predicted Ys and the actual Y points is squared, then a mean taken. This mean is then square rooted to produce a final error value, this is then collated and shown as a graph (see figure 5)

RMSE Graph

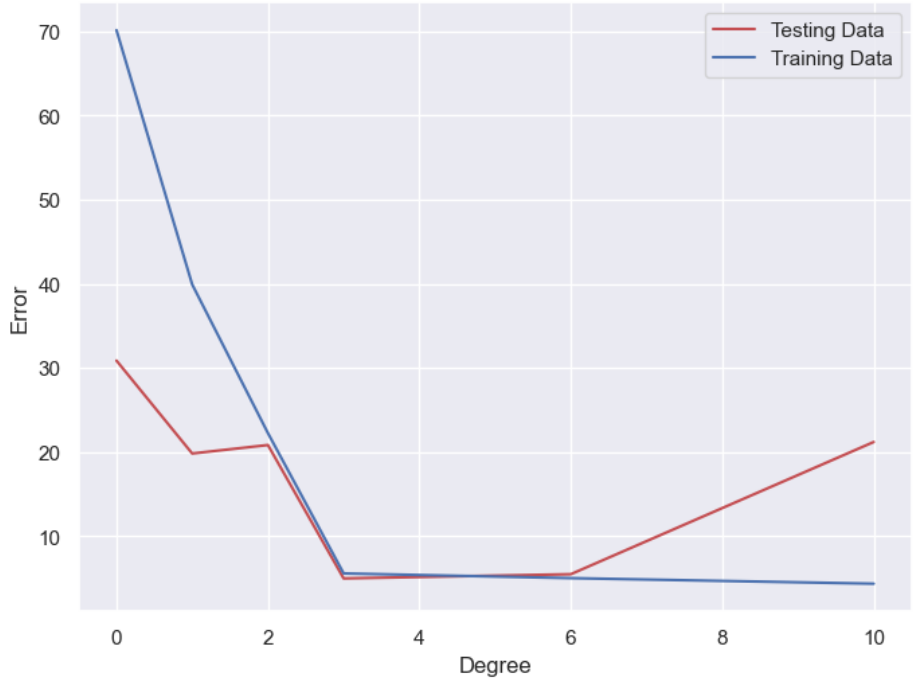

FIGURE 5: RMSE AS DEGREE INCREASES

zero degrees performs overall the worst on the both the training and test set with the largest distances between the two sets and the highest overall errors. The first degree is the second worst with a smaller gap between the two sets, but the overall error is still too high. The third degree has a very low error for both sets and the distance between them is relatively small. The sixth degree contains a slightly lower error and different sets are very close. By the 10^{th} degree as the data is too over fitted the error for the testing set jumps considerably while the training sets error is smaller.

In conclusion, from the RMSE calculations degrees 3 and 6 would potentially be the best option due to the near equilibrium of error rates between the data sets and the overall error being very low.

Secondly as the plots suggest that the six has begun to noticeably overfits the data while three is only slightly underfitting but still captures the trend of the data accurately. Overall degree 3 appears to be the best due to its low accuracy and that it does not considerably over or under fit the data but still following the trend tightly.

Charlie Elliott
19694799
Machine Learning: CMP3751M

Task 2:

Section 2.1: K Means Implementation

K means was implemented using 5 methods:

compute_euclidean_distance:

```
def compute_euclidean_distance(vec_1, vec_2):
    # vec_1 , vec_2 two vectors and it calculates the distance
    distance = np.sum(np.square(vec_1 - vec_2))
    return np.sqrt(distance)
```

This function accepts two vectors as an argument and calculates the Euclidean distance of all their features and returns the result.

initialise_centroids:

```
def initialise_centroids(dataset, k): #k = 2,3
    # dataset: the data set unsplit
    # k: number of classifiers
    # initialise a numpy array of just ones the shape of the data
    centroids = np.ones(dataset.shape[1])

    # generate random centroid points with the data range
    for i in range(0,k):
        # empty array for random values
        new = []
        # cycle through all features
        for j in range(0,dataset.shape[1]):
            # create a random value within the range of the data and appened it to the new array
            new.append(random.uniform(min(dataset[:, j]),max(dataset[:,j])))
        # conver the new array to an np array and add it to the end of the new centroid values
        centroids= np.column_stack((centroids, np.array(new)))
    # slice of the ones created at the start and transpose the array so that it matches the axis of the features
    return (np.delete(centroids,0,1)).transpose()
```

Initialise_centroids is responsible for creating the first set of centroids when Kmeans be ins It starts by initialising an empty numpy array to hold all the data, then loops through a randomiser (k times) that computes random centroid values; using the min and max of each feature to ensure the centroids start within the scope of the dataset making it more accurate, reducing the number of iterations and reduces overall within cluster scatter.

Get_class:

```
    # initialize empty array for holding classification data
    classification = []
    # loop through the dataset

    for i in dataset:
        # initialize empty array to temporarily hold the distances
        distances = []
        # loop through k
        for dist in range(0,k):
            # calculate euclidean distance between point and cluster centre
            distances.append(compute_euclidean_distance(centroids[dist],i))

        # append the shortest the classified distance to the classification array
        classification.append(distances.index(min(distances)))
```

This function classifies the data by finding the closest possible centroids to each point. It starts by wiping any existing classifications from the dataset, then it loops through the dataset checking the euclidean distance between each point and the centroids adding these to an array that will always be in order of what centroid it checked first so an index of the smallest distance will always provide a correct and accurate classification, this output is appended to dataset (see figure 5). With this implementation if there is a tie in data classification, the last centroid to be run will be given the point, for example if the first and second centroids tie, the point will be given to the second centroid (see figure 6).

```
[[5.151 3.535 1.414 0.202 1.   ]
 [4.949 3.03  1.414 0.202 1.   ]
 [4.747 3.232 1.313 0.202 1.   ]
 ...
 [6.63  3.06  5.304 2.04  0.   ]
 [6.324 3.468 5.508 2.346 0.   ]
 [6.018 3.06  5.202 1.836 0.   ]]
```

FIGURE 5: THIS RESULTS IN THE DATASET LOOKING LIKE (EXAMPLE USES K =2) WITH THE RIGHT MOST COLUMN BEING THE CLASSIFICATION.

```
[3.7598556954316664, 5.1883491958014085]
```

FIGURE 6: SAMPLE OUTPUT OF THE DISTANCES ARRAY WHERE K =2

Charlie Elliott
19694799
Machine Learning: CMP3751M

new_cent:

```
def new_cent(classified_set,k):
    # initialise an a numpy array of just ones
    centroids = np.ones((classified_set.shape[1]-1))
    # loop through k
    for fil in range(0,k):
        # use the i value to filter all the different k classifications and add them to a temporary array
        filtered = classified_set[classified_set[:,4] == fil]
        # empty array for holding the new centroid locations
        new = []
        # calculate and save the mean of these columns
        for i in range(classified_set.shape[1]-1):
            new.append(np.mean(filtered[:,i]))
        # append to the centroids array
        centroids= np.column_stack((centroids,np.array(new)))
    # return the centroid array with the 1s removed
    return(np.delete(centroids,0,1)).transpose()
```

This function moves the centroids to the mean of all data points assigned to it. It starts by creating an empty numpy array to hold the new centroid positions. It then loops through fil in range 0 to k, this is used to provide an index for filtering out any entries in the dataset classified as fil, it then calculates the mean of these entries to create a new centroid, after this has been done for all k it returns a numpy array containing the new centroid points.

Kmeans:

```
def kmeans(dataset, k):
    # get the random initial centroids
    centroids = initialise_centroids(dataset, k)
    # set the number of iterations
    iterations = 1000000
    tempcentroids = np.ones(centroids.shape)
    # iterate through the iterations
    cluscatmap = []
    for i in range(iterations):
        # run classificiation
        cluster_assigned= get_class(dataset,k,centroids)
        # calculate new centroids
        centroids = new_cent(cluster_assigned,k)
        if((centroids == tempcentroids).all()):
            break
        else:
            tempcentroids = centroids

        cluscatmap.append([i, within_cluster_scatter(cluster_assigned,centroids,k)])
    # return the final centroid location and the data with attached classification and the withiin cluster
    return centroids, cluster_assigned, np.array(cluscatmap)
```

This is the first function ran, it is responsible for running all other functions and regulates the iterations. It starts by running the initialise_centroids method to create random centroid location. It then defines the iterations value designed to break the loop if it accidentally loops infinitely and an array tempcentroids to hold the previous iteration's centroid locations; an empty array for holding the within cluster scatter is also initialised (more in 2.2). It then starts a loop until the centroids are settled. it then runs get_class to classify the data followed by new_cent to move the centroids. Finally, it checks to see if the centroid locations are the same as the last iteration, if so, it breaks from the loop returning the final centroid locations and the classified data and data used to plot the within cluster scatter (more in 2.2).

Charlie Elliott
19694799
Machine Learning: CMP3751M
Section 2.2 Dog Breed evaluation:
A) Height and Tail length
K= Two

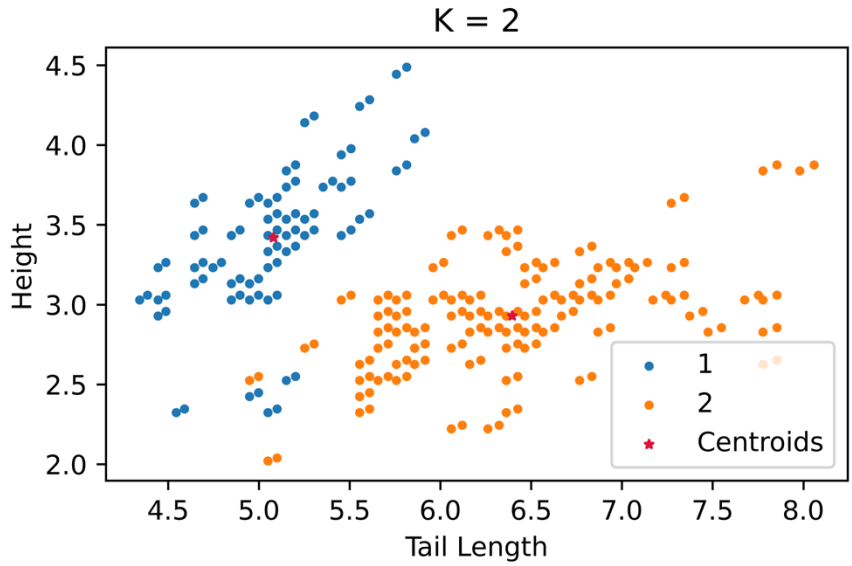

FIGURE 7: HEIGHT AND TAIL LENGTH WITH TWO CENTROIDS

Two centroids quite accurately classifies the data. However, between x 4.5-5.5 y 2.0-.5 there is an outlier of eight blue points where it should have classified it orange, this is possibly caused by skew from the x 7.5-8.0 y 3.5-4.0 or outliers in the other features that have pulled the centroid slightly too far. Overall, two centroids is a very accurate way of classifying this feature pairing.

K= Three

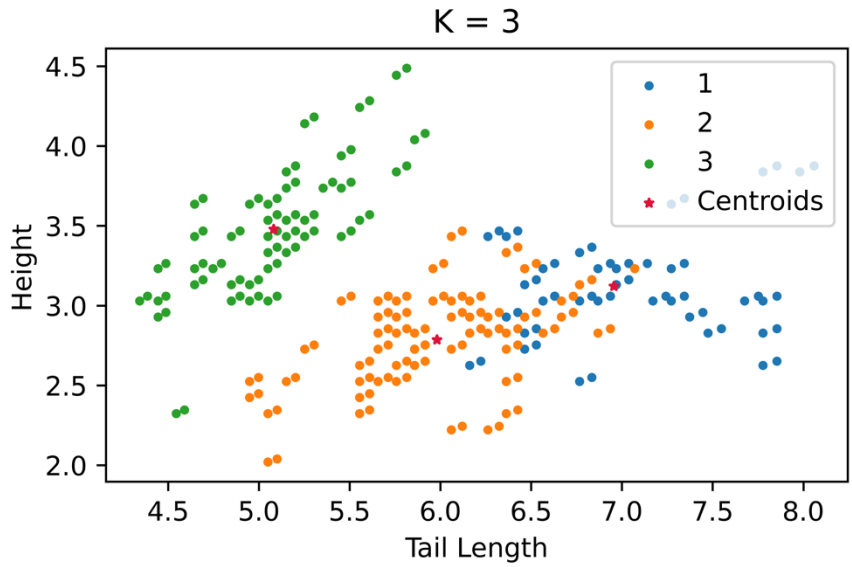

FIGURE 8: HEIGHT AND TAIL LENGTH WITH THREE CENTROIDS

Overall, three centroids do well in the x 4-6 and y 2.0-3.5 range accurately classifying bar two outlier green points at x 4.5 and y 2.2. The first centroid at x 3.2 and y 6.9 introduces a lot of outliers and does not accurately capture the separation of the data, with significant outliers and many points right next to centroid not being classified as part of it, this suggests that three centroids is to many of for this feature pairing.

B) Height and Leg Length

Charlie Elliott
19694799
Machine Learning: CMP3751M

K= Two

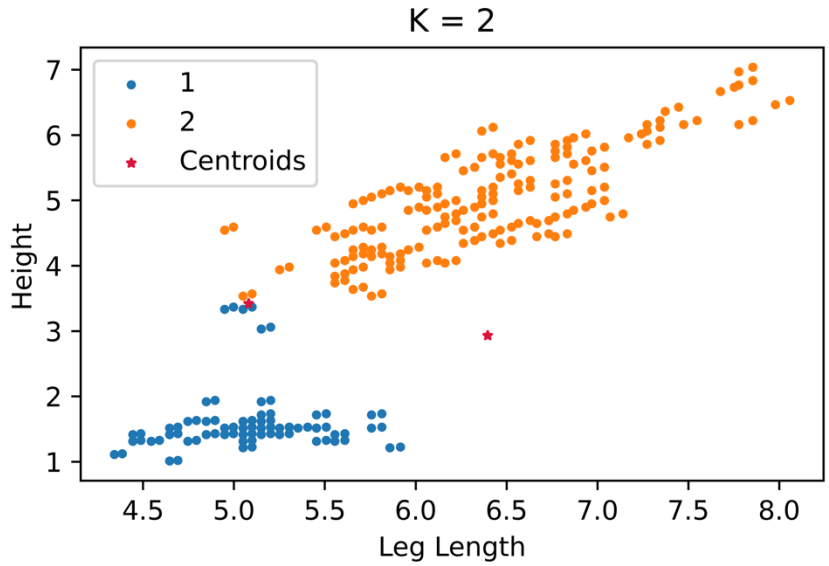

FIGURE 9: HEIGHT AND LEG LENGTH WITH TWO CENTROIDS

Two centroids very accurately classify this feature paring with only six outliers in the 3-3.5 and 5-5.5. the centroids in this pairing are noticeably not close to the data this is likely caused by the compression of the 4D data used in the algorithm down to 2D for the purposes of displaying. Overall, two centroids are incredibly accurate for capturing the data's logical divisions.

K= Three

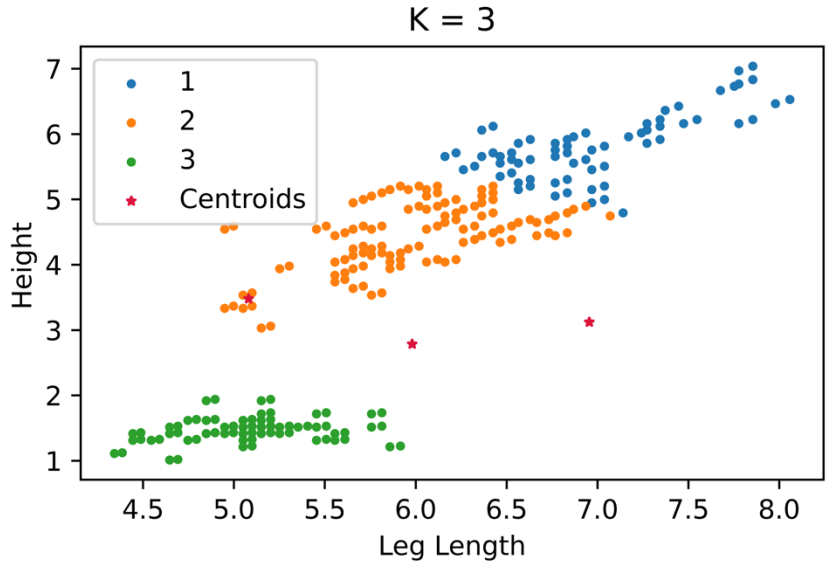

FIGURE 10: HEIGHT AND LEG LENGTH WITH THREE CENTROIDS

Three centroids over comes the outliers present in the two centroids. However, once again the first centroid struggles to accurately classify the data at logical divisions, but, unlike height and tail length there are no other centroid classifications inside the classification. Overall, three centroids is too many for capturing this feature pairings natural divisions.

Charlie Elliott
19694799
Machine Learning: CMP3751M

C) within cluster scatter (Objective function)

K= Two

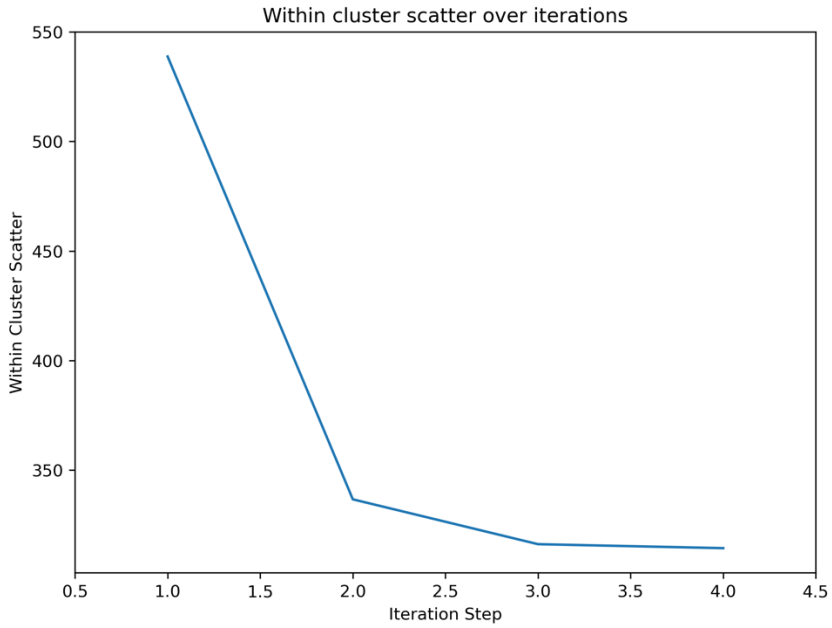

FIGURE 10: WITHIN CLUSTER SCATTER FOR TWO CENTROIDS

Took four iterations to settle with a considerable decrease in scatter after the first iteration ending in a finer adjustment by the final step.

K = Three

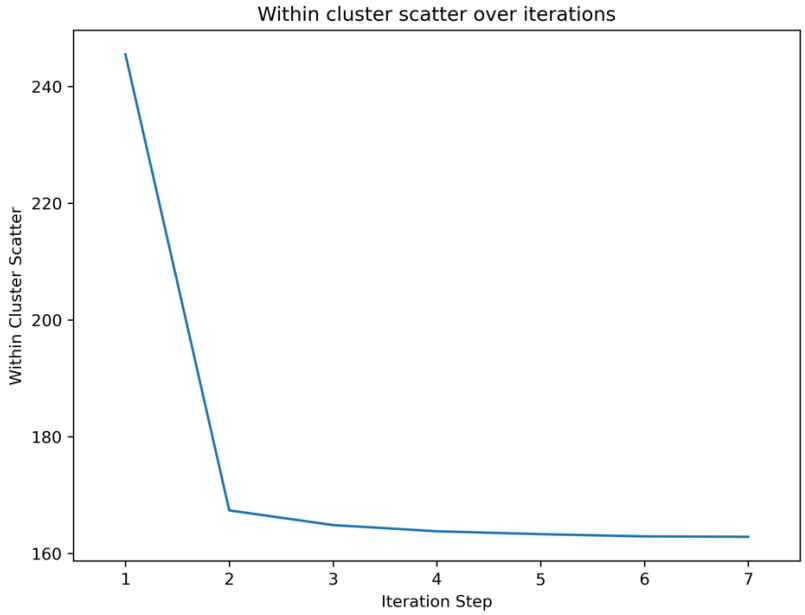

FIGURE 11: WITHIN CLUSTER SCATTER FOR TWO CENTROIDS

Three centroids settled much quicker but required more iterations with a considerable drop between the first and second iteration but required more fine adjustments as the iterations takes place.

In conclusion, while three centroids have a considerably reduced within cluster scatter compared to two. But when looking at the divisions it draws in the features examined, they don't follow the logical grouping the data follows, this could potentially result in a large amount of false negatives/positives, reducing overall accuracy.

Task 3

Section 3.1 Data import, summary, pre-processing and visualisation:

The Data is imported using the pd.read_csv function then is processed using the Stats function

Stats Function:

```
def Stats(frame,statusCategory,columns):
    #get all rows with the assigned status
    fullData= []
    df2 = frame.loc[frame['Participant Condition'] == statusCategory]
    df2 = df2.iloc[:,3:8]


    #Find maximum, minimum, mode, median, mean, and variance of the data
    fullData.append(df2.max().tolist())
    fullData.append(df2.min().tolist())
    fullData.append(df2.mean().tolist())
    fullData.append(df2.var().tolist())
    fullData.append(df2.mode().values[0].tolist())
    fullData.append(df2.median().tolist())
    fullData = np.array(fullData).transpose()

    # Stack this into a pandas dataframe for ease of reading
    StatFrame = pd.DataFrame(fullData,columns=['max', 'min','median','mean','mode','var'],index = columns[3:8])

    return StatFrame
```

This function filters the dataset, so it only contains either Control of Patient participants, then calculates the min, max, mean, variance, mode and median, then packs it into a panda's data frame (see figures 12 and 13).

	max	min	median	mean	mode	var
Alpha	2.350897	0.092770	0.615520	0.071599	0.092770	0.618396
Beta	3.169293	0.283299	1.554242	0.157595	0.283299	1.511789
Lambda	1.533264	0.304582	0.764979	0.030345	0.304582	0.786381
Lambda1	1.467637	0.390920	0.980486	0.017616	0.390920	0.984126
Lambda2	1.232208	0.309526	0.741053	0.026140	0.309526	0.759125

FIGURE 12: PATIENTS STATISTICS

	max	min	median	mean	mode	var
Alpha	2.356406	0.131221	0.615071	0.069044	0.131221	0.623514
Beta	3.376731	0.392472	1.558109	0.142506	0.392472	1.518908
Lambda	1.535059	0.362244	0.765318	0.029379	0.362244	0.789629
Lambda1	1.456809	0.489722	0.982531	0.016396	0.489722	0.983633
Lambda2	1.246102	0.368800	0.742884	0.024496	0.368800	0.753050

FIGURE 13: CONTROL STATISTICS

Genplots function:

```
def GenPlot(data,columnNames):
    # get patient and control frames
    Patient = Stats(data,"Patient",columnNames)
    Control = Stats(data,"Control",columnNames)
    # Show them to the user
    print("Patient Statistics")
    print(Patient)
    print("Control Statistics")
    print(Control)

    # Plot Box Plot Alphas
    Box = [Patient.values.tolist()[0][0:3],Control.values.tolist()[0][0:3]]

    fig1, ax1 = plt.subplots(figsize = (10,8),dpi = 1000)
    ax1.set_title('Alpha for both Categories')
    ax1.set_xticklabels(["Patient", "Control"], ha="center")
    ax1.boxplot(Box)
    plt.show()

    # Plot Density of Betas
    sns.set(style="darkgrid")

    sns.kdeplot(data.loc[data['Participant Condition'] == "Patient"]['Beta'])
    sns.kdeplot(data.loc[data['Participant Condition'] == "Control"]['Beta'])
    plt.title('Density Plot of Betas')
    plt.legend(['Patient','Control'])
    plt.show()
```

Gen plots takes the outputted data frames and plots a boxplot of the alphas and a density plot of the Betas (See figures 14 and 15).

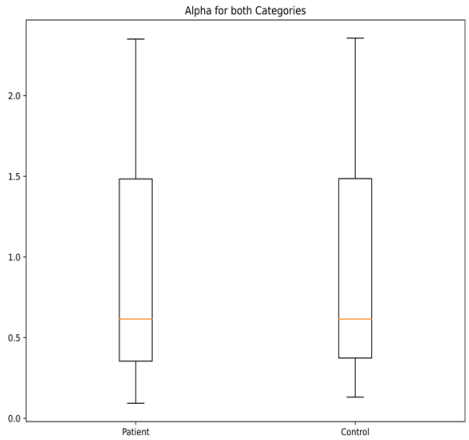

FIGURE 14: ALPHAS BOXPLOT

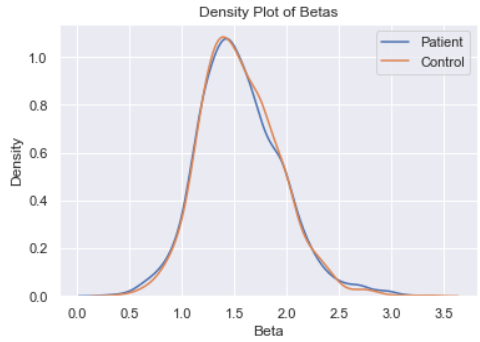

FIGURE 15: DENSITY PLOT OF BETAS

Normalisation:

From the preliminary analyses of the data normalisation will likely need to be performed as the beta values has a considerably wider range than other features, which could potentially lead to inaccuracy when training models. The feature range was set from 0 to 2, this reduces the range of Betas and Alphas

```
def DataNormalizer(data):
    # Normalise the data so that it falls between 0 and 2
    scaler = MinMaxScaler(feature_range = (0,2))
    scaler = scaler.fit(data)
    return scaler.transform(data)
```

Overall,there is very little difference between the two categories. The data frames show that patients and controls have relatively close min and max's and examining the relationship between the mean median and modes yields very similar skews. The difference in the alphas is so minute on the box plots they appear almost very similar with only the smaller minimum for patients being noticeable. The density plot shows that the skew of the Beta values is very similar, with it only being noticeably different between x 0.5-1.0 and y 0.0-0.2, x 1.5-2.0 and y0.6-0.8 and finally between x 2.5-3.0 and y 0.0-0.2. In conclusion, from this preliminary analysis it will likely be very hard to train any classifier to accurately classify this data due to the skew and distribution being so close.

Section 3.2 Designing algorithms:

Setting up an Artificial Neural Network (ANN) and Random Forest Classifier (RFC):

Choosing a library:

As an RFC and ANN are difficult to manually implement efficiently effectively given the time frame a premade library will be used. SckitLearn was a chosen, due to its widespread adoption, making debugging and learning resources easy to find, and it's scalability allowing it to be applied to a variety of data.

```
def ANN(epoch, neurons):
    # setup  an ANN using logistic activation function, adamn solver, with user set epochs and neurons
    print("Number of epochs:  " , str(epoch))
    print("Number of neurons: ", str(neurons) )
    clf = MLPClassifier(max_iter=epoch, activation = 'logistic'
                        ,hidden_layer_sizes=(neurons[0], neurons[1]), solver = "adam")
    return clf



def RFC(samples, estimators):
    # setup and RFC with user se samples and estimators
    print("Number of Samples: " + str(samples))
    print("Number of Trees: "+ str(estimators))
    clf = RandomForestClassifier(min_samples_leaf = samples
                                 ,n_estimators = estimators)

    return clf
```

Two Functions were created to construct them, necessary parameters such as epochs, neuron count, leaf samples and estimators are passed in and a model ready to be trained is returned.

Splitting the dataset

Test_train_split from Task 1 was reused with test_size configured at 0.1 to provide a large amount of training data. Having much larger training data will help in this case as the data is so similar.

Testing apparatus:

```
# See how the iterations effects accuracy
iterationTest = [1,100,250,500,750,1000,2500,5000,10000]
leafs = [5,10]
scores = []
for i in iterationTest:
    NN = ANN(i,[500,500])
    scores.append(ClassEval(NN,xTrain,xTest,yTrain,yTest))
plt.figure(figsize = (10,8),dpi = 1000)
plt.title("Accuracy as iterations increase")
plt.ylabel("Accuracy")
plt.xlabel("Iterations")
plt.plot(iterationTest,scores,c= 'b')

# See how leaf nodes effect accuracy
scores = []
for i in leafs:
    FC = RFC(i, 100)
    scores.append(ClassEval(FC,xTrain,xTest,yTrain,yTest))
plt.figure(figsize = (10,8),dpi = 1000)
plt.title("Accuracy as leaf nodes increase")
plt.ylabel("Accuracy")
plt.xlabel("Nodes")
plt.plot(leafs,scores,c= 'b')
```

Two arrays where setup to hold the iterations and leaf's, they were then looped through to train the models on each of them and then the results where plotted.

Examining the effect of iterations and leaf samples on accuracy:

The Artificial Neural Network:

The ANN performed poorly, result where inconsistent and often it struggled to ever classify anything as Control. The result of this is that Accuracy sees no clear increase as iterations do and more iterations often result in worse results (see figure 15). Overall accuracy never exceeded 51.7%, this could potentially be due to getting bad seeds, but considering previous analysis its likely caused by the lack of clear division in the data.

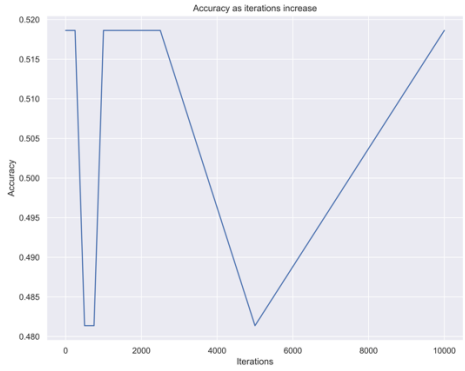

FIGURE 15: THE ANN ACCURACY AS ITERATIONS INCREASE

The Random Forest Classifier:

The RFC Performed as well as the ANNs highest results, but as leaf nodes increased so too did accuracy (See figure 16). Some sample outputs show that it was capable of classifying points as control (See figure 17) which the ANN could never produce; however overall accuracy is still disappointingly low and is likely caused by the same issues.

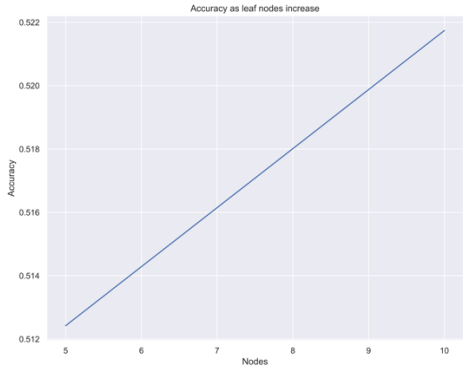

FIGURE 16: THE RFC ACCURACY AS LEAF NODES INCREASE

```
Number of Samples: 10
Number of Trees: 100
Testing Set:
{'Control': 127, 'Patient': 195}
Training Set:
{'Control': 1283, 'Patient': 1609}
Accuracy Score: 0.5217391304347826
```

FIGURE 17: SAMPLE OUTPUT OF THE RFC

In conclusion both performed poorly so ten-fold cross validation will be used to assess If there is any chance of getting an accurate classification.

Section 3.3: Model selection:

Implementing ten-fold cross validation:

Cross_val_score from Sckitlearn was used to perform the cross validation, this function is simple to implement and as it comes from the same library as the ANN and RFC it guarantees great compatibility and efficiency. Validation was performed on using 50,500 and 1000 neurons for the ANN and 50,500,10000 estimators for the RFC. The mean of the validation was taken to assess the overall accuracy. The output for both shows they were incredibly inaccurate (see figures 18,19 and 20). With the RFC fairing the worst and the ANN performing the best. The RFC performed the best when configured with 10 leaf nodes and 50 estimators, while the ANN performed best with 100 iterations and 50 neurons on each layer.

	ANN	RFC
50	0.618206	0.385984
500	0.541918	0.384431
1000/ 10000	0.499829	0.381938

FIGURE 18: SAMPLE OUTPUT OF CROSS VALIDATION AS A PANDA DATA FRAME.

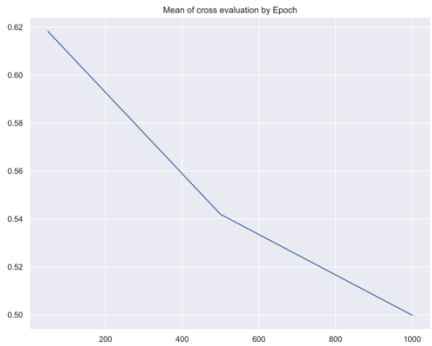

FIGURE 19: SAMPLE OUT OF ANN PLOTTED AS A GRAPH

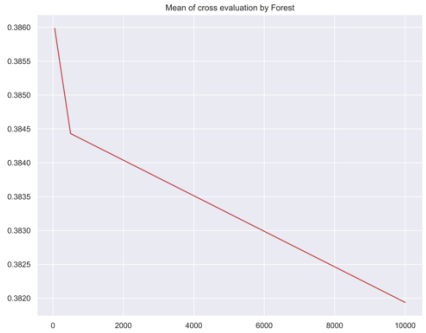

FIGURE 20: SAMPLE OUT OF RFC PLOTTED AS A GRAPH

Overall, neither classifier can provide truly accurate and consistent results, this is likely due to the difference between Control and Patients being so small. This problem could be reduced if there was simply more data as it would afford the models more training data, increasing chances of catching outlier data and more finely tuning classification boundaries, and more testing data leading to better and more consistent accuracy results. Even then the data would likely follow the same trends, so it's probably a better option to find different features to train against; especially considering the effect misclassification can have if implemented in a real world environment.

Charlie Elliott
19694799
Machine Learning: CMP3751M
References:
Polynomial Regression (unknown). What is Polynomial Regression? Unknown: Polynomial Regression. Available from: http://polynomialregression.drque.net/math.html#:~:text=Most%20people%20have%20done%20polynomial,mean%20average%20of%20that%20data. [Accessed 7th February]

Statistics how to. (2022) RMSE: Root Mean Square Error. Unknown: Statistics how to. Available from: https://www.statisticshowto.com/probability-and-statistics/regression-analysis/rmse-root-mean-square-error/ [Accessed 7th February]