

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[0:-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 upto 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 argument 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 coefficients
        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

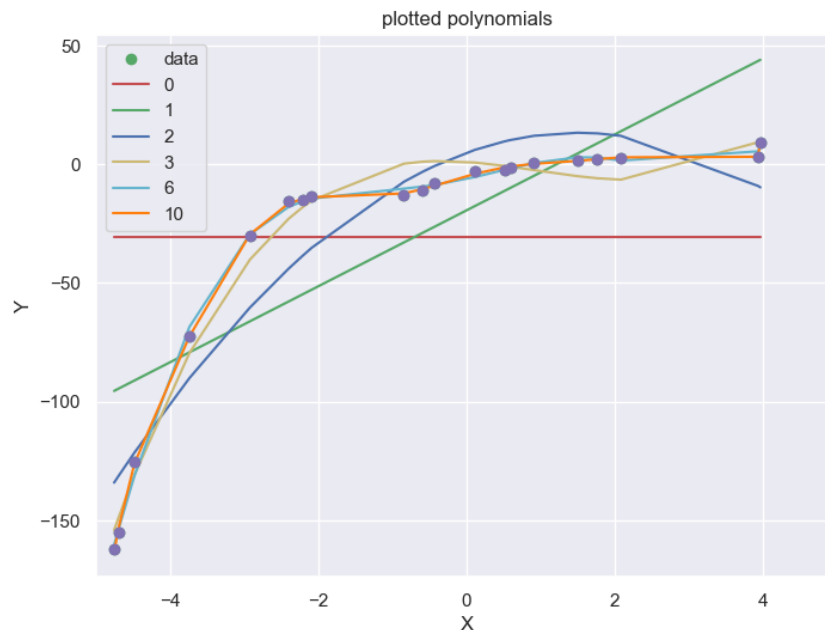


FIGURE 4: PLOTTED POLYNOMIALS ON GRAPH

```
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.

Zero degrees performed the worst as it is just a straight line on the mean of the data so drastically underfits and doesn't follow the trend of the data, all points bar one is an outlier it is almost entirely without merit.

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 sklearn function `Train_Test_split`, with shuffle enabled and test_size set to 0.3.

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:  
        predicted_y = testMatrix  
    else:  
        predicted_y = testMatrix.dot(parameters)  
  
    mse = np.mean((predicted_y - 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

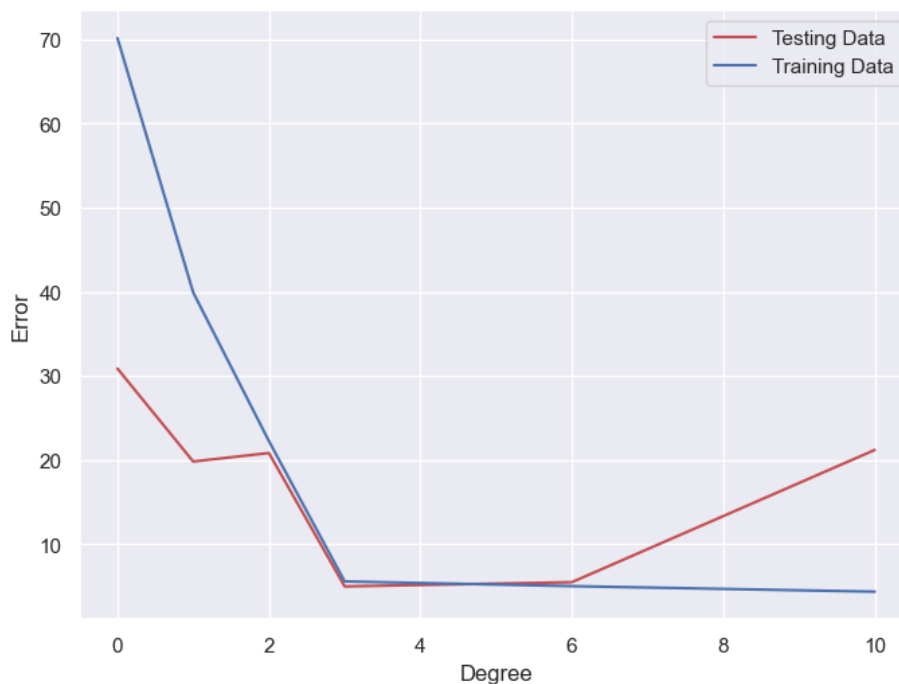


FIGURE 5: RMSE AS DEGREE INCREASES

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 10th 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.

Charlie Elliott

19694799

Machine Learning: CMP3751M

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.

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 append it to the new array  
            new.append(random.uniform(min(dataset[:,j]),max(dataset[:,j])))  
        # convert 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 begins. 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 classification
        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 within 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).

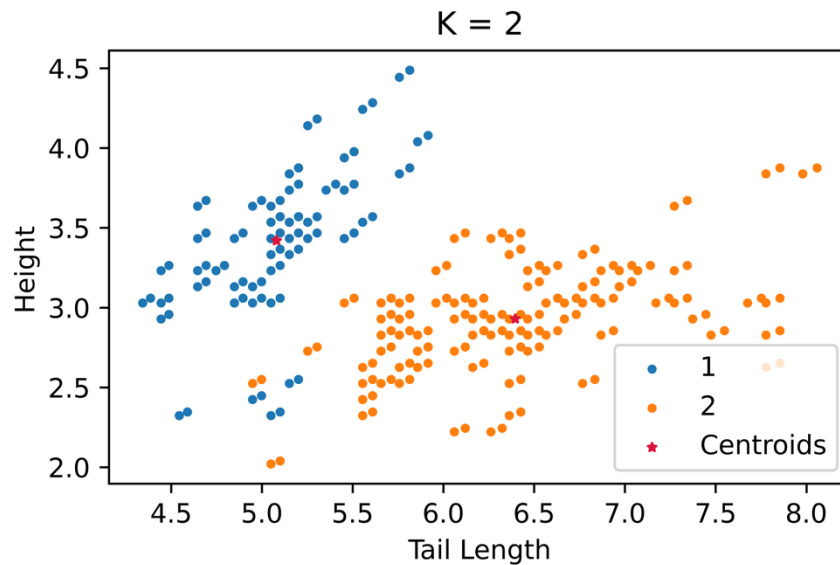


FIGURE 7: HEIGHT AND TAIL LENGTH WITH TWO CENTROIDS

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

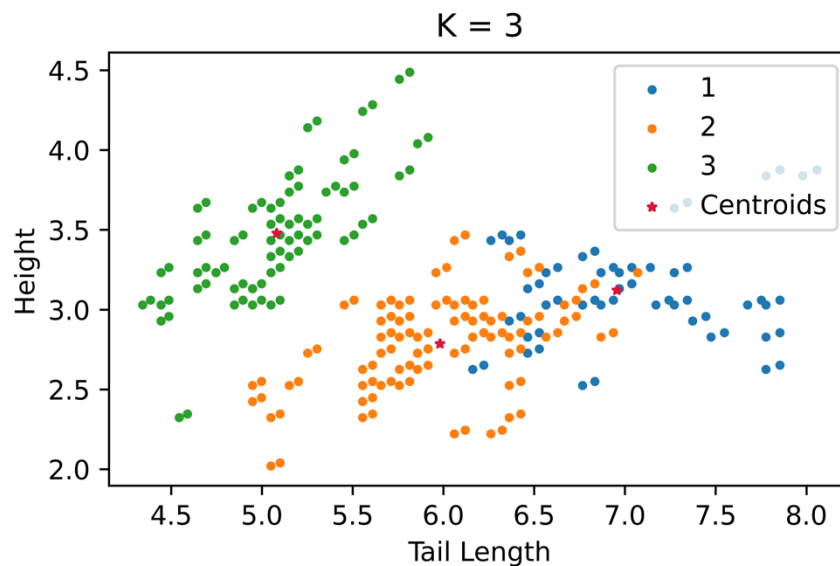


FIGURE 8: HEIGHT AND TAIL LENGTH WITH THREE CENTROIDS

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

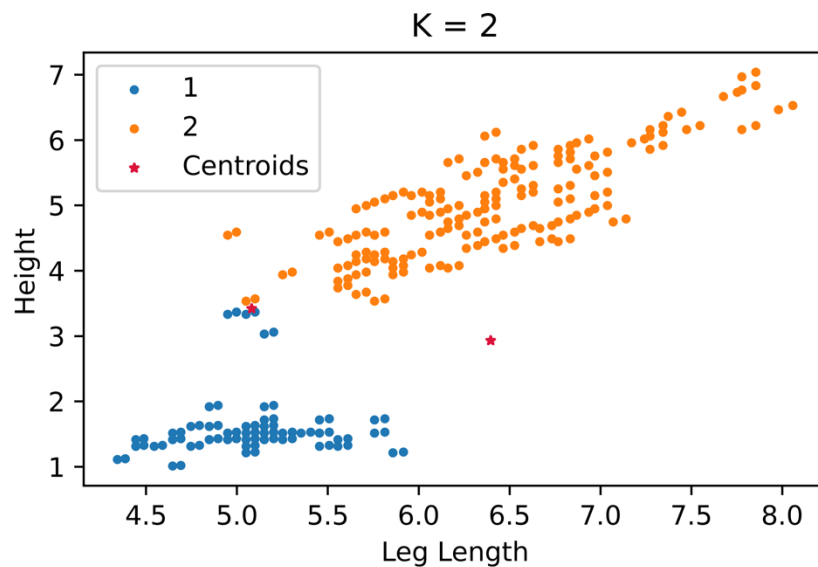


FIGURE 9: HEIGHT AND LEG LENGTH WITH TWO CENTROIDS

Two centroids very accurately classify this feature pairing 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

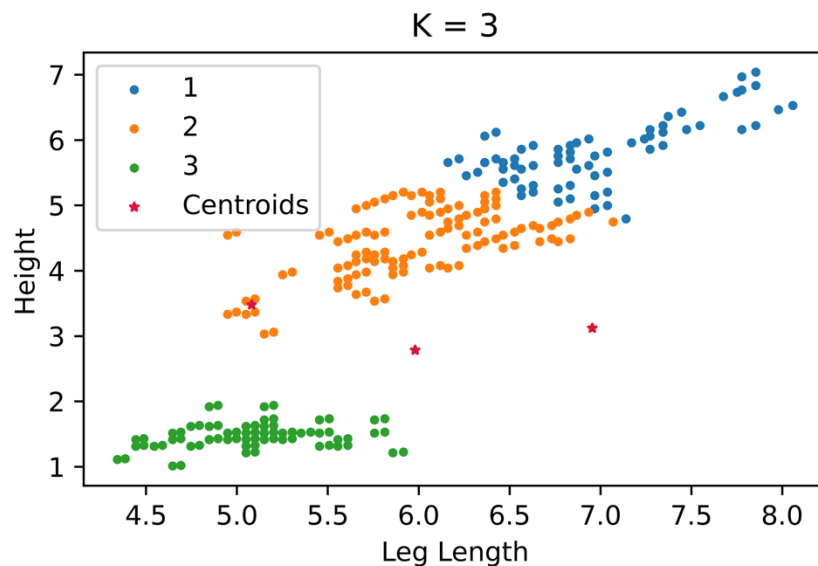


FIGURE 10: HEIGHT AND LEG LENGTH WITH THREE CENTROIDS

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.

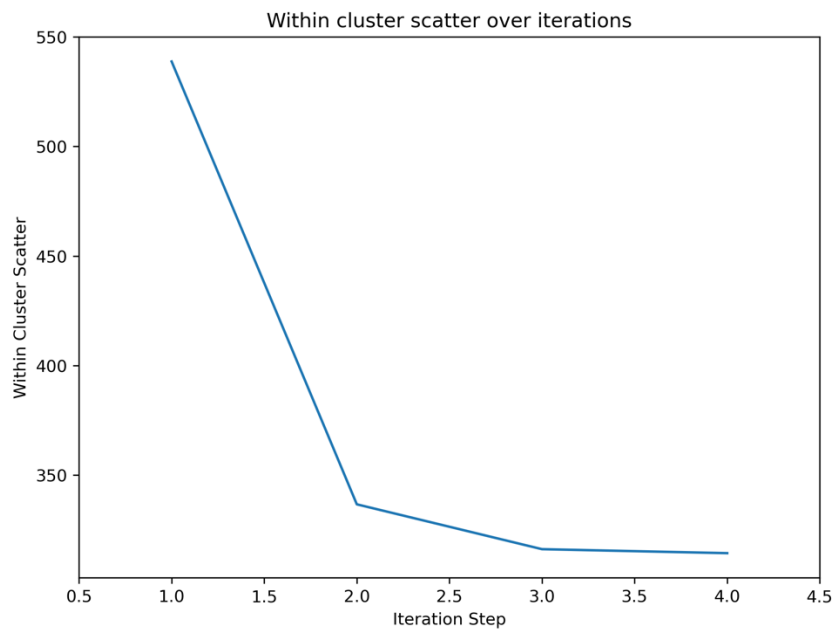


FIGURE 10: WITHIN CLUSTER SCATTER FOR TWO CENTROIDS

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

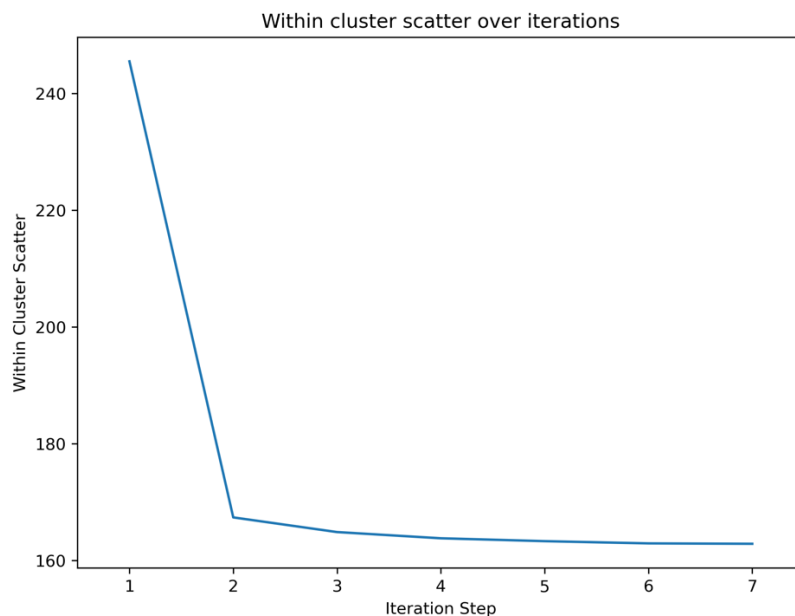


FIGURE 11: WITHIN CLUSTER SCATTER FOR TWO CENTROIDS

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.

Charlie Elliott

19694799

Machine Learning: CMP3751M

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 this imports it as a pandas data frame which is a quick and effective way of storing and analysing the data.

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 or 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	stand dev
Alpha	2.350897	0.092770	0.618396	0.615520	0.092770	0.071599	0.267580
Beta	3.169293	0.283299	1.511789	1.554242	0.283299	0.157595	0.396982
Lambda	1.533264	0.304582	0.786381	0.764979	0.304582	0.030345	0.174199
Lambda1	1.467637	0.390920	0.984126	0.980486	0.390920	0.017616	0.132727
Lambda2	1.232208	0.309526	0.759125	0.741053	0.309526	0.026140	0.161678

FIGURE 12: PATIENTS STATISTICS

	max	min	median	mean	mode	var	stand dev
Alpha	2.350897	0.092770	0.618396	0.615520	0.092770	0.071599	0.267580
Beta	3.169293	0.283299	1.511789	1.554242	0.283299	0.157595	0.396982
Lambda	1.533264	0.304582	0.786381	0.764979	0.304582	0.030345	0.174199
Lambda1	1.467637	0.390920	0.984126	0.980486	0.390920	0.017616	0.132727
Lambda2	1.232208	0.309526	0.759125	0.741053	0.309526	0.026140	0.161678

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).

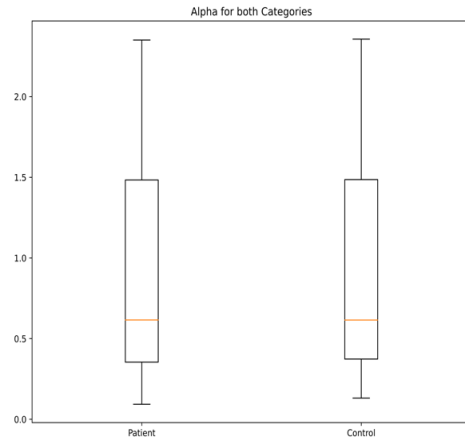


FIGURE 14: ALPHAS BOXPLOT

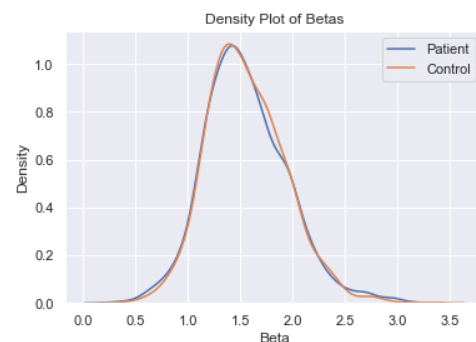


FIGURE 15: DENSITY PLOT OF BETAS

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 y 0.6-0.8 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 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.

Charlie Elliott

19694799

Machine Learning: CMP3751M

```
def ANN(epoch, neurons):  
    # setup an ANN using logistic activation function, adam 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

From examination of a similar study (Al-Diri, Hunter, 2009) an attempt was made to group the data by image number, take a mean for each then split that data; this could have potentially mitigated the similarity issues. However, this shrank the dataset too much to be usable. So 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:

```
xTrain,xTest,yTrain,yTest = DataSplit(data,columnNames,3)  
xTrain,xTest,yTrain,yTest = DataSplit(data,columnNames,0)  
# See how the iterations effects accuracy  
iterationTest = [1,100,250,500,750,1000,2500,5000,10000]  
leafs = [5,10]  
scoresIpa = []  
for i in iterationTest:  
    NN = ANN(i,100,500)  
    scoresIpa.append(ClassEval(NN,xTrain,xTest,yTrain,yTest))  
    scoresIpa.append(ClassEval(NN,xTrain,xTest,yTrain,yTest))  
plt.figure(figsize = (10,8),dpi = 1000)  
plt.title("Accuracy at iterations increase")  
plt.xlabel("Accuracy")  
plt.ylabel("Iterations")  
plt.plot(iterationTest,scoresIpa,c= 'b')  
plt.plot(iterationTest,scoresIpa,c= 'r')  
plt.legend(["Alpha...Lambda2", "Image Number... Lambda2"])  
# See how leaf nodes effect accuracy  
scoresIpa = []  
for i in leafs:  
    FC = RFC(i,100)  
    scoresIpa.append(ClassEval(FC,xTrain,xTest,yTrain,yTest))  
    scoresIpa.append(ClassEval(FC,xTrain,xTest,yTrain,yTest))  
plt.figure(figsize = (10,8),dpi = 1000)  
plt.title("Accuracy at leaf nodes increase")  
plt.xlabel("Accuracy")  
plt.ylabel("Nodes")  
plt.plot(leafs,scoresIpa,c= 'b')  
plt.plot(leafs,scoresIpa,c= 'r')  
plt.legend(["Alpha...Lambda2", "Image Number... Lambda2"])
```

Two arrays were 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 were plotted.

Examining the effect of iterations and leaf samples on accuracy:

Choosing features to use:

Initial training was done using just the alpha, beta and three lambda columns as these columns contained the most varied data, however training was inconsistent and very inaccurate with both models occasionally never being able to classify any point as control, so instead all the data was used including the image number this increased overall accuracy. Both results were plotted for ease of comparison.

The Artificial Neural Network:

On just the Alpha, beta and lambda features 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 52.6%. On all the data, accuracy was much higher overall reaching a max of 65.3% but was more inconsistent, with two drops in accuracy at fifty iterations and one thousands iterations. With the extra features it was now capable of classifying points as both Patient and Control (See figure 16). Overall, accuracy is as expected 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.

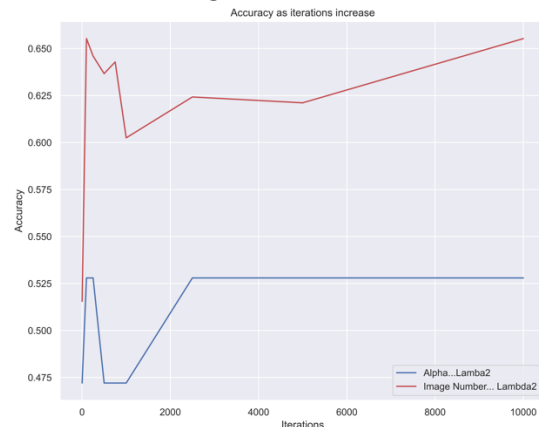


FIGURE 15: THE ANN ACCURACY AS ITERATIONS INCREASE

```
Testing Set:
{'Control': 149, 'Patient': 173}
Training Set:
{'Control': 1340, 'Patient': 1552}
Accuracy Score: 0.6242236024844721
```

FIGURE 16: SAMPLE OUTPUT FROM ANN ON ALL FEATURES

The Random Forest Classifier:

Both feature choices showed a downward trend as leaf nodes increased, this was more pronounced when all features were used. Using just alpha, beta and lambdas overall accuracy was low topping out at 50.1%, but unlike the ANN it actually classified points as both control and patient (see figure 18, top result). On all features it performed exemplary managing 81.9% at highest on five leaf nodes with and at worst performed a 79.8%, much higher than what ANN managed. Overall accuracy is now at an acceptable level when all features are used.

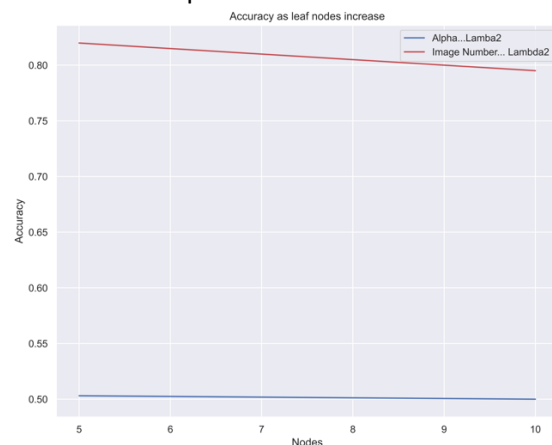


FIGURE 17: THE RFC ACCURACY AS LEAF NODES INCREASE

```
Testing Set:
{'Control': 156, 'Patient': 166}
Training Set:
{'Control': 1334, 'Patient': 1558}
Accuracy Score: 0.5031055900621118

Testing Set:
{'Control': 158, 'Patient': 164}
Training Set:
{'Control': 1325, 'Patient': 1567}
Accuracy Score: 0.8198757763975155
```

FIGURE 18: SAMPLE OUTPUT OF THE RFC

In conclusion, any future evaluation should utilise all features as this leads to better accuracy, optimum leaf nodes appear to be five, while the best number of iterations is one hundred. But before any definitive conclusions can be made its worth cross validating the models so that its clear these results aren't outliers and to hopefully find other optimum values to further improve accuracy.

Section 3.3: Model selection:

Implementing ten-fold cross validation:

Charlie Elliott

19694799

Machine Learning: CMP3751M

```
def CrossVal(xTest,yTest,NeuNet,DecFor,cross):
    # cross eval on the neural network and the random forest classifier
    Neurscores = cross_val_score(NeuNet, xTest, yTest, cv=cross)
    Treescorers = cross_val_score(DecFor, xTest, yTest, cv=cross)

    # Calculate the mean and return it as an array
    print("ANN Mean Scores: " + str(Neurscores.mean()))
    print("RFC Mean Scores: " + str(Treescorers.mean()))
    return [Neurscores.mean(),Treescorers.mean()]
```

Cross_val_score from Skitlearn 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 ANN fared reasonably well on cross validation achieving a highest mean score of 52.4% while the RFC faired horribly it only managed a maximum of 38.6% (see figures 19,20,21). The ANN had the largest range in its results while the RFC had a very small range. But the raw output from the Cross_val_score for the RFC had a considerably wide range which has harmed its overall score, while the ANN has a much smaller range (See figure 22). Overall from cross validation has suggested that the ANN performs best at one hundred iterations with five hundred neurons, while the RFC does best with fifty trees and 10 leaf nodes.

	ANN	RFC
50	0.577113	0.386604
500	0.624814	0.382559
1000/ 10000	0.471323	0.382872

FIGURE 19: SAMPLE OUTPUT OF CROSS VALIDATION AS A PANDA DATA FRAME.

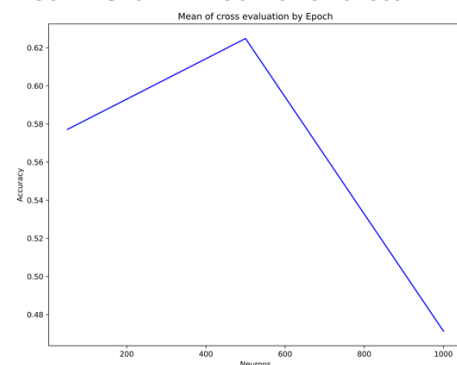


FIGURE 20: SAMPLE OUTPUT OF ANN PLOTTED AS A GRAPH

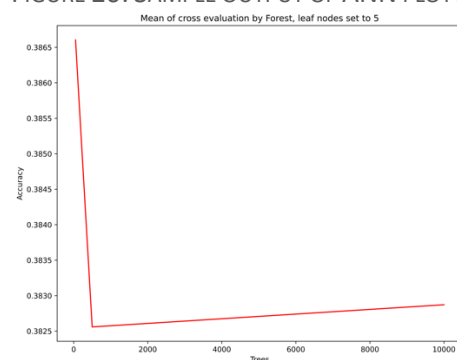


FIGURE 21: SAMPLE OUTPUT OF RFC PLOTTED AS A GRAPH

```
Trees Scores
[0.52173913 0.53416149 0.52173913 0.40372671 0.55140187 0.10903427
 0.00934579 0.17133956 0.51090343 0.4953271 ]
ANN Scores
[0.52173913 0.52173913 0.52173913 0.48757764 0.26479751 0.47663551
 0.47975078 0.47975078 0.47975078 0.47975078]
ANN Mean Scores: 0.4713231168127551
RFC Mean Scores: 0.38287184845494476
```

FIGURE 22: SAMPLE RAW OUTPUT OF CROSS_VAL_SCORE

Overall, while the RFC faired best on the iteration tests it wasn't able to transfer this over to cross validation, which suggests that the randomness is what led to such high results or it just got very

Charlie Elliott
19694799

Machine Learning: CMP3751M

lucky with the training splits. The ANN however performed consistently between both the iteration tests and cross validation. As a result, the ANN is better option of the two given the provided data. However, overall accuracy is still too low given the real-world risk of misclassification, 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.

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]

Al-Diri, B. and Hunter, A. (2009) Automated measurements of retinal bifurcations. In World Congress on Medical Physics and Biomedical Engineering, September 7-12, 2009, Munich, Germany (pp. 205-208). Springer, Berlin, Heidelberg. [Accessed 8th February]