CMP3751M Machine Learning

Contents

Task 1 – Polynomial Regression

Description

Polynomial Regression

Polynomial Regression a form of multiple linear regression analysis which estimates a relationship using an nth degree polynomial (Eva Ostertagová, 2012).

A polynomial is an expression 'of one or more terms, each of which consists of a constant multiplied by one or more variables raised to a nonnegative integral power' (Webster, n.d.).

The representation a Polynomial equation is as follows.

$$y_i = \beta_0 + \beta_1 X_i + \beta_2 X_i^2 + \cdots + \beta_m X_i^m + \varepsilon_i \ (i = 1,2 \ldots n),$$

Polynomial's regression can also be defined in a matrix, like so.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

Polynomials are integral, not only in mathematics, but for use in machine learning amongst data science populace.

In polynomial regression, the independent variables (denoted as β) are indefinite, meaning they cannot be changed. The resulting expansion of these variables are known as “higher-degree terms”

Regression

Regression allows us to use relationships between variables to find a line of best fit, that can be used to interpret optimal data or imply preferred parameters.
Polynomial regression allows us to provide the “best approximation of the relationship between the dependent and independent variable” (Pant, 2019).

Error functions

Error functions are used as terms to add bias towards a proposed model of prediction, they are residual variables produced by the model and used to predict a model’s fit to real statistics.

Least squares Method

The least squares method is employed to validate the authenticity of a prediction model. It is plotted using the expression **ŷ= a + b x.** And is calculated by the sum of residuals minimised by the sum squared. (Kenton, 2022)

Root means squared error

The RMSE is a standard deviation of the residual errors. The RMSE can be interpreted as the square root of “the difference between forecast and corresponding observed values, squared and averages over the sample” (eumetrain, n.d.)

Mathematically, the expression for RMSE can be interpreted as follows.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

The RMSE provides a relatively high weight to large errors, this is because the errors are squared before being averaged, making RMSE suitable for minimising respectively large errors.

Implementation

Firstly, appropriated libraries for the task were imported

```
#import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import numpy.linalg as linalg
```

Then the dataset was read using ‘pd.read_csv’ and split into two arrays from their respective x and y variables.

```
#Pandas to read csv
data_train = pd.read_csv('Task1 - dataset - pol_regression.csv')
#Convert to 1D numpy arrays
x_train = data_train['x']
y_train = data_train['y']
```

A helper function was created to perform feature expansion at a specified degree for data set ‘x’.

```
#Implementing Polynomial Regression
#Convert to Matrix
def getPolyMatrix(x, degree):
    X = np.ones(x.shape)
    for i in range(1,degree +1):
        X = np.column_stack((X,x**i))
    return X
```

A function was created to compute optimal beta values given the input x, output y and user defined degree for a polynomial.
This function makes use of the aforementioned ‘getPolyMatrix’ feature expansion function.

```
def pol_regression(x,y,degree):
    if (degree == 0):
        degreemean = 0
        degreelength = len(y_train)
        for j in range(degreelength):
            degreemean = y_train[j]
            w = (degreemean/degreelength)
    #If weight is 0.
    else:
        X = getPolyMatrix(x, degree)
        XX = X.transpose().dot(X)
        w = np.linalg.solve(XX, X.transpose().dot(y))
    return polynomial coefficients
```

Plotting

The data was plot using the matplotlib library function ‘pyplot’, a figure of the baseline data was created using ‘plt.figure()’ and ‘plt.plot’ to provide a baseline of comparison for the best fit lines of the polynomial.

```
#Create figure
plt.figure()

#Plot 'raw' data as blue dots
plt.plot(x_train,y_train, 'bo')

#Generate evenly spaced numbers over a specified interval
line = np.linspace(-5,5,20,0)
```

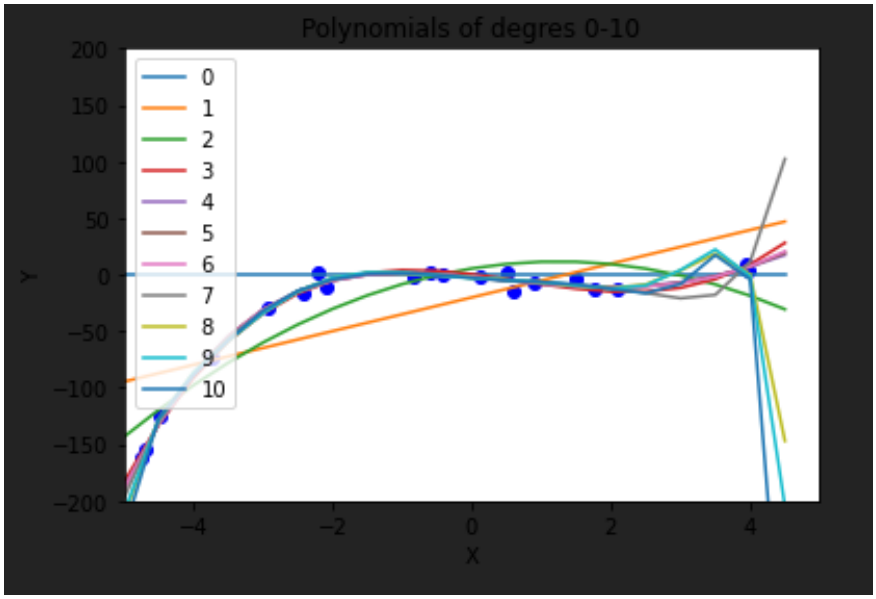
The fitted polynomials were plot using a for loop in range 0-10, which with every pass calculated the optimal Betas using ‘pol_regression’, and performed feature expansion using ‘getPolyMatrix’ (returned in a 1D numpy array) using numpy matrix operations. The line was plot using plt.plot and plt.legend.

```
#Loops through degrees 0-10, calculates and plots
# the line of best fit for each degree, plots on the same graph.
for x in range(11):
    w = pol_regression(x_train,y_train,x)
    xt = getPolyMatrix(line,x)
    yt = xt.dot(w)
    plt.title("Polynomials of degres 0-10")
    plt.plot(line,yt, label=x)
    plt.legend()
```

The scope, x and y axis were created and shown using pyplot operations.

```
#Lims as requested from the brief
#for better viewing of graph
plt.xlim(-5,5)
plt.xlabel('X')
plt.ylabel('Y')
plt.ylim(-200,200)
plt.show
```

Below is a fitted plot of each Polynomial degree 0-10.



From the data shown, the most optimal polynomial beta degree fit is number 6.

Evaluation – Root Mean Squared Error

To calculate the RMSE of the training and testing sets the polynomial weights (betas) for each degree had to be calculated. The coefficients of these could be used to calculated the MSE (Mean Squared Error) and Ultimately RMSE (Root Mean Squared Error)

A function to calculate the RMSE of the Polynomial was created named ‘eval_pol_regression’

```
def eval_pol_regression(parameters,x,y,degrees):
    #Get coefficients
    coeff = getPolyMatrix(x,i)
    #Get weights (betas)
    w = pol_regression(x,y,i)
    #Calculate the RMSE, Square root performed (.sqrt) on the MSE (Mean Squared Error)
    rmse = np.sqrt(np.mean((coeff.dot(w)-y)**2))
    return rmse
```

This calculated the coefficients using getPolyMatrix, and weights using pol_regression to retrieve optimal betas for the polynomial. Then calculated the RMSE of the coefficient using numpy operation ‘.sqrt’ to square root the MSE.

The data was split into X train, Y train, X test and Y test variables using the scikit learn library function ‘train_test_split’. With 70% of the split being the train size, and 30% being the test size, shuffling was disabled to produce reproduceable results.

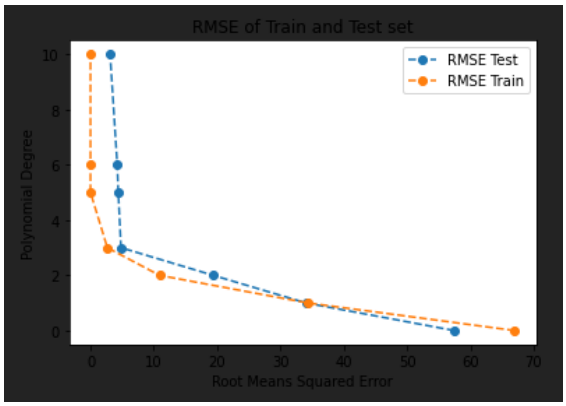
```
#Split data set into 30% test and 70% train
#disabled shuffle for reproducable results
X_train, X_test, Y_train, Y_test = train_test_split(x_train, y_train, train_size=0.7,shuffle=False)
```

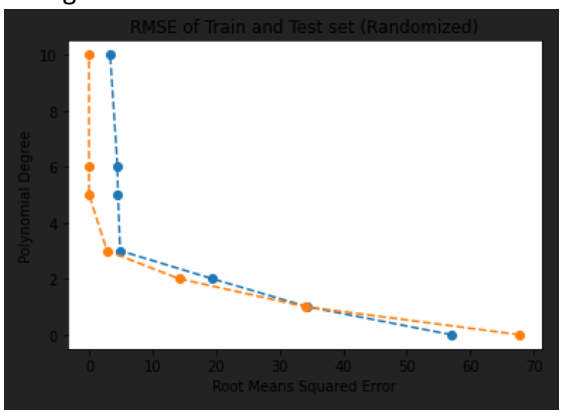
To plot the data, two arrays were initialized called rmseTrain and rmseTest, the given degrees were also created in an array.

A loop was created to loop through the given degrees, calculate the weights, evaluate them using RMSE and append to the train / test set.

```
#Initialize rmseTrain and rmseTest Arrays to store rmse values of both sets
rmseTrain = []
rmseTest = []
#Array of degrees to pass in accordance with brief
degrees = [0,1,2,3,5,6,10]
#For each degree in the array, evaluate the Betas of each degree, and append to the respective array for later usage
for i in degrees:
    rmseTrain.append(eval_pol_regression(pol_regression(X_train,Y_train,i),X_train,Y_train,i))
    rmseTest.append(eval_pol_regression(pol_regression(X_test,Y_test,i),X_test,Y_test,i))
```

The results of the RMSE values were plotted in the figures below.





Evaluation

As the RMSE of these plots is relatively low, its hard to identify clear underfitting. However there is minor evidence of overfitting at higher polynomial degrees, such as degree 6, showing the model has over adapted to the data provided, making it difficult to fit more data.
Regardless, the model is able to analyse test data whilst producing a low RMSE, this outcome is preferable, as it evidences the model's adaptability to the data in general.

Task 2 – K-means Clustering

Description

K-Means Clustering

K-means clustering is a method of clustering analysis that aims to partition sets of data (or observations) into groups of clusters (techopedia, n.d.). Several K clusters are produced in the set of data, and each data point is related to a centroid (The central point of a cluster).
This is typically calculated through the lowest mean distance of each data point's relation to each centroid.

K means is calculated as follows:

- K points are implemented into a data structure to represent initial centroid groups
- Each object in the data structure is assigned to their closest K-value relation.

- Once all objects have been assigned to a K-value, the Centroids are recalculated
- The previous two steps are repeated until a maximum iteration is achieved, or the centroids are no longer able to move.

K-means clustering can be used to sift through huge scales of data and is used to identify and classify which group an object should belong to.

Euclidian Distance

Euclidian distance is defined as the length of a line segment between two points. It can be calculated using the formula.

$$q(\mathbf{b}\cdot\mathbf{d}) = \sqrt{(b^1 - d^1)_2 + (b^2 - d^2)_2 + \cdots + (b^i - d^i)_2 + \cdots + (b^u - d^u)_2}.$$

Centroids

A centroid is defined as “a location representing the centre of a cluster of data” (Garbade, 2018), it can be calculated at random or via consistent iterations of each cluster, such as in the K-means method. In which overall distance is calculated from each object to each centroid, ultimately re-classifying new centroids as the resultant mean changes.

Implementation

Additional libraries were loaded for the implementation of K-Means clustering.

```
#Task 2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import numpy.linalg as linalg
from sklearn.cluster import KMeans
```

Helper functions were implemented to aid the K-means clustering algorithm. These functions included ‘compute_euclidian_distance’. ‘initialise_centroids’ and ‘kmeans’ respectively.

Linalg.norm was used to return the norm of the difference from the two vectors, in order to calculate Euclidian distance

```
def compute_euclidean_distance(vec_1, vec_2):
    #The function compute_euclidean_distance() calculates the distance of two vectors, e.g., E
    #returns the norm of the difference of both vectors to calculate euclidian distance
    distance = np.linalg.norm(vec_1-vec_2,axis=1)
    return distance
```

Centroids were initialised in a random location of the cluster using the numpy randint function

```
def initialise_centroids(dataset, k):
    #Initialises a centroid in a random location of the cluster range using randint
    centroids = dataset[np.random.randint(dataset.shape[0],size=k)]
    return centroids
```

The K-means algorithm was calculated using the following function, it returned the centroids and clusters for use plotting.

```
def kmeans(dataset, k):
    #The function kmeans() clusters the data into k groups
    centroids = initialise_centroids(dataset,k)
    #Max iterations 50
    maxiter = 50

    #Initialises two matrices using the dataset and converts them to float
    #Same oepration for the k value
    classes = np.zeros(dataset.shape[0],dtype=np.float64)
    distances = np.zeros([dataset.shape[0],k], dtype=np.float64)

    #Pass through until reaches maxiter
    for i in range(maxiter):
        #Loop for number of centroids
        for i, c in enumerate(centroids):
            #Get distances with helper function
            distances[:,i] = compute_euclidean_distance(c,data)
            #Find smallest value within distances
            classes = np.argmin(distances,axis=1)
            #get the means of the classes and assign them an array of centroids
            for c in range(k):
                centroids[c] = np.mean(dataset[classes ==c],0)
            clusters = classes

    return centroids, clusters
```

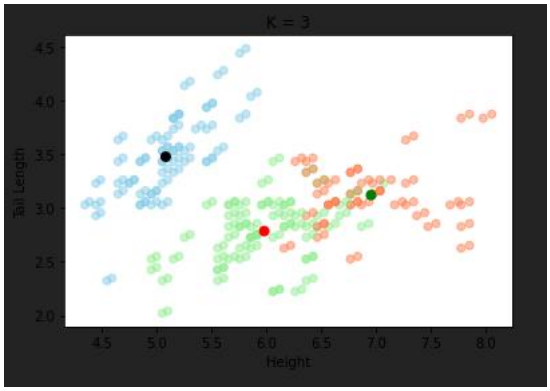
Below is the function used to plot the clusters, the final block is repeated for the second axes also (Leg Length, Height).

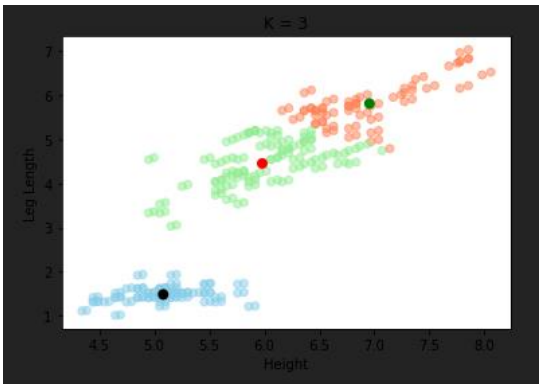
```
#Plots the clusters
def plotclusters(data,k):
    #gets appropriate color quantity
    ctColors = wrapColors(k)

    #Calculates Kmeans of data and k value
    centroids, classes = kmeans(data,k)
    #Gets title for k, used for plotting layer
    title = str(k)

    #Colors for plotting
    gcolors = ['skyblue','coral','lightgreen']
    colors = [gcolors[j] for j in classes]

    #Draws plot for k value on both axis
    #This block is repeated for second axes
    plt.figure()
    plt.scatter(data[:,0],data[:,1], color=colors,alpha=0.5)
    #Uses random colors
    plt.scatter(centroids[:,0], centroids[:,1], color = ctColors.copy(), marker='o', lw=2)
    plt.xlabel('Height')
    plt.ylabel('Tail Length')
    plt.title("K = "+title)
```

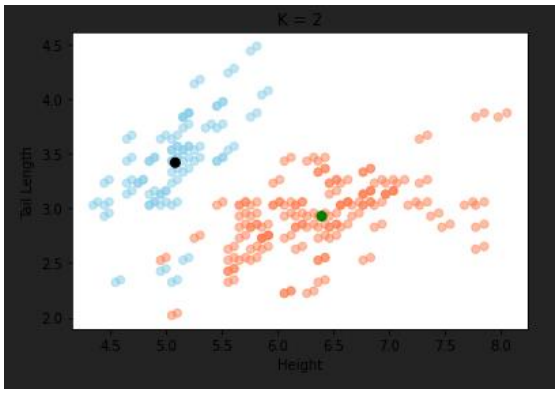
The data is read in and the plotclusters algorithm (which calls k-means, etc.) is ran only twice.

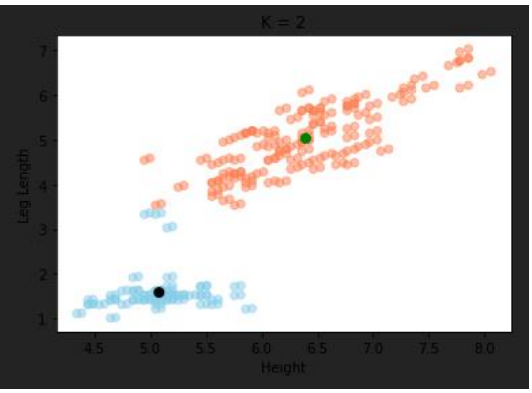
```
#Get data set
data = pd.read_csv('Task2 - dataset - dog_breeds.csv').values

#Run K-Means Clustering with given K values 3,2 using given data set
plotclusters(data,3)
plotclusters(data,2)
```

Below are the results of the K-means clustering algorithm using the K value 2 and 3.









Evaluation

From analysis of the K-means data, we can identify a clear link between Height and leg length, both clusters are closely correlated and overlap in some areas, this is most noticeable when implementing K-means with 3 centroids.

Upon evaluation and visualisation of the K-means cluster data, I believe the optimal number of centroids to be k=3.

Task 3 – Classification and Model Evaluation

Description

Artificial Neural Nets

https://ieeexplore.ieee.org/abstract/document/1165576
https://www.ibm.com/uk-en/cloud/learn/neural-networks

Artificial Neural Networks (ANNs) are implemented to reflect the behaviour of the human brain (Lippmann, 1987). Allowing programs to recognize patterns and calculate defined “success metrics”, allowing the processing, classification and evaluation of large datasets with little human intervention, work, cost or time consumption. Simply put, neural nets are ideal classifiers in the field of machine learning.

Neural networks can be perceived as a series of nodes existing of a linear regression model, composed entirely through input data, weights, thresholds and an output.

The output of a node can be defined as follows:

$$f(w1.X1 + w2.X2 + b)$$

Sigmoid Function

The sigmoid function is a form of activation function commonly used within machine learning and in conjunction with the *logistic function*. Sigmoid functions are an important part of logistic regression and are commonly employed where "a real number needs to be converted into a probability" (Wood, n.d.).

The Sigmoid function maps a real value into another value between 0 and 1, this can be used to map predictions to probabilities.

The logistic Sigmoid function can be interpreted as follows.

$$\begin{aligned} S(x) &= \frac{1}{1+e^{-x}} \\ &= \frac{e^{x}}{e^{x}+1} \end{aligned}$$

Random Forests Classifier

A Random Forests algorithm is a form of classification consisting of decision trees that operate in conjunction with one another. Each tree in the random forest outputs a prediction, the output with the highest value is assigned the model's prediction (Breiman, 2001).

The Random Forest classifier utilizes a method known as *Ensemble learning*. This is the process of using groups of 'processors' (trees) to weigh input data and aggregate the result to identify the most popular result.

What separates a Random Forests algorithm, however? Is the utilisation of ensemble learning and "feature randomness" to create an 'uncorrelated forest' of decision trees (Education, 2020).

Typical decision trees are trained to consider all possibilities in feature splits. Random forests are trained to only select a subset of features; this ensures resulting outputs will maintain a low correlation against other trees.

Accuracy

Accuracy is a popular metric for evaluating classification models. We can calculate the accuracy of a model by dividing the Number of correct predictions by the Total number of predictions to receive a float value or 'ratio' of correct results.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Accuracy is typically visualised using a confusion matrix, which is the sum of True Positives and True Negatives, divided by True and False Positives / Negatives.

Cross Evaluation and K-fold

Cross evaluation methods are used to estimate the efficacy or 'skill' of machine learning models using new or 'unforeseen' data. These methods are typically implemented to compare predictive models and result in a lower comparative bias (Brownlee, 2018).

K-fold cross validation is performed by shuffling a data set randomly, splitting the set into K number of groups. For each group, one is selected as a test set, the others are chosen as the training set. A model is fit using the train set and evaluated using the test set. Finally, the skill of the model is summarized using a score metric, such as the accuracy.

Implementation

Summary and Pre-Processing

Firstly, additional libraries were loaded, and the dataset was read

```
#Task 3.1
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

#Read Dataset
df = pd.read_csv('Task3 - dataset - HIV RVG.csv')
```

To show a summarization list of the desired outputs, an array of operations and column names were created.

```
#Declare Summarization operations
ops = ["mean","median","std","max","min"]
#Get column names
columnNames = list(df.columns)
```

Df.agg() was used to display the desired summarization calculations defined in 'ops'

```
#Summarize using DataFrame.Aggregate with the ops argument to only display desired calculation
summ=df.agg(ops)
display(summ)
```

A boxplot was created using pandas 'groupby' function to assign 'Condition as the x axis, and Alpha as the y axis.

```
#Create a boxplot using 'Condition' as the x value and Alpha as the Y value
s = df.groupby(['Participant Condition'])
s.boxplot(column=['Alpha'],subplots=False)
```

The dataframe was split by Participant condition type using Pandas equals operators.

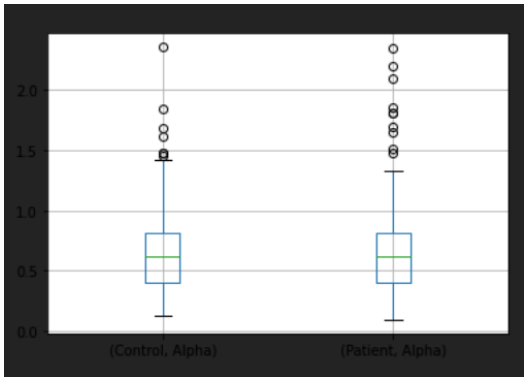
```
#Split DataFrames based on Participant Condition
df1 = df[df['Participant Condition'] == "Patient"]
df2 = df[df['Participant Condition'] == "Control"]
```

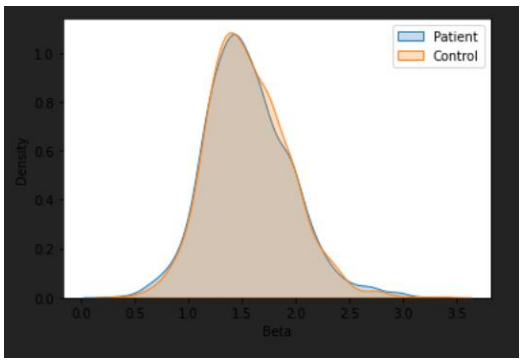
A density plot was created using the betas of Patient and Control Participant conditions using the seaborn library.

```
#Plot Density Plot using the Betas of 'Patient' and 'Control' Participant
#conditions using Seaborn (Shading operator makes it easier to compare)
plt.figure()
sns.kdeplot(df1['Beta'], label='Patient',shade=True)
sns.kdeplot(df2['Beta'], label='Control',shade=True)
plt.legend()
```

Below are the summarized results, boxplot and density plot for the data. Normalised data was tested, but the results did not provide a significant difference to warrant efficacy.

	Image number	Bifurcation number	Artery (1)/ Vein (2)	Alpha	Beta	Lambda	Lambda1	Lambda2	Participant Condition
mean	107.355009	6.220597	1.573118	0.615305	1.556093	0.765141	0.981465	0.741929	NaN
median	109.000000	5.000000	2.000000	0.620978	1.515562	0.788022	0.984015	0.756226	NaN
std	58.933207	4.103341	0.494702	0.265245	0.387726	0.172840	0.130492	0.159205	NaN
max	203.000000	25.000000	2.000000	2.356406	3.376731	1.535059	1.467637	1.246102	Patient
min	1.000000	1.000000	1.000000	0.092770	0.283299	0.304582	0.390920	0.309526	Control





Through analysis of this data, we can identify relationships between both Participant types given alpha and beta values. For example, in the density plots we can see the Density of beta values peak at the very similar degrees for both Patient and Control participant conditions.

Designing Algorithms

ANN

An artificial neural net was created to train the provided data set, a fully connected network architecture was created using 500 neurons for two hidden layers.
The data was randomly selected using train_test_split, with 90% of the data as the training set, and 10% as the test, using ‘shuffle=True’ to randomize.

Extra libraries were loaded

```
#Import libraries
from sklearn import neural_network
```

Next, a function for an ANN (Artificial Neural Net) was implemented, using the sigmoid function as the activation function and logistic function for the output layer.
The data was pre-processed, non-appropriate values were omitted using ‘pandas.drop()’ function. Then, x and y matrices were created using x as the training set and y as the label.

```
#Split data
#sigmoid function as the non-linear activation function for the hidden layers and logistic
#function for the output layer;
def ANN():
    #Load Dataset
    df = pd.read_csv('Task3 - dataset - HIV RVG.csv')
    #Drop unnecessary columns
    df = df.drop(['Image number', 'Bifurcation number', 'Artery (1)/ Vein (2)'],axis=1)
    #Drop Condition for use as label
    x = df.drop('Participant Condition',1)
    #Create label as Participant Condition
    y = df['Participant Condition']
```

The data was split using train_test_split, as mentioned.

```
#Split data, add shuffle to randomize results on each pass
trainx,testx, trainy,testy = train_test_split(x,y, train_size=0.9, shuffle=True)

clf = neural_network.MLPClassifier(learning_rate_init = 0.1, solver = 'sgd',
                                   max_iter = 80000, hidden_layer_sizes = (500, 500),
                                   activation = 'logistic',
                                   momentum = 0, alpha = 0,
                                   verbose = False, tol = 1e-10,
                                   random_state = 11)
#Fit training data against Participant Condition
clf = clf.fit(trainx, trainy)
#Train against the testing set
y_pred=clf.predict(testx)
```

MLPC classifier was used to create a Multi-Layer Perceptron Classifier for the Artificial Neural net, the hidden layer sizes were both set to 500 respectively. The model was trained using .fit() with the training data.

The accuracy was calculated using the sklearn library function ‘metrics.accuracy_score’

```
#Import libraries
from sklearn import metrics

# print("Accuracy:",metrics.accuracy_score(testy, pr))
#Calculate accuracy using sklearn.
accr = metrics.accuracy_score(testy, y_pred)
return accr
```

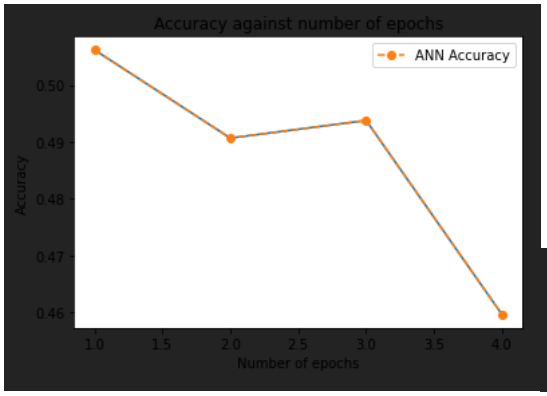
Arrays containing the index and accuracy values for multiple epochs in the neural net, a figure was created using matplotlib.

```
#for different epochs, run in a for loop, append to a list.
plt.figure()
epochs=[]
for i in range(1,5):
    accr = ANN()
    epochs.append(accr)
```

```
epochn=[]
for i in epochs:
#     print(i)
    epochn.append((epochs.index(i)+1))
```

```
#Plot figure
plt.plot(epochn,epochs)
plt.title("Accuracy against number of epochs")
plt.xlabel('Number of epochs')
plt.ylabel('Accuracy')
plt.plot(epochn,epochs, linestyle='--', marker='o',label='ANN Accuracy')
plt.legend()
plt.show()
```

Below is the figure showing the Accuracy of the ANN over several epochs and the accuracy values of the epochs.




```
Accuracy:  0.5093167701863354
Accuracy:  0.5372670807453416
Accuracy:  0.5279503105590062
Accuracy:  0.46273291925465837
```

Random Forests

A random forest classifier was also created using 1000 trees, tested with the minimum number of samples 5, and 10.

```
def RandomTrees(a,e,trainx,trainy):
    #e = estimators, a = minimum leaf sample
    clf=RandomForestClassifier(n_estimators=e, min_samples_leaf = a)
    #Fit the model
    clf.fit(trainx,trainy)
    #Train against the test set
    y_pred=clf.predict(testx)

    from sklearn import metrics
    print("Accuracy \nEstimators:",e,"\nMin samples:",a,"\n=",metrics.accuracy_score(testy, y_pred))
    return
```

Using a similar method, with the exception of using RandomForestClassifier function to classify data, with the number of estimators being ‘e’, and samples being ‘a’, the accuracy using the test set was calculated.

```
samples = [5,10]
for i in samples:
    RandomTrees(i,1000,trainx,trainy)
```

```
Accuracy
Estimators: 1000
Min samples: 5
= 0.4782608695652174
Accuracy
Estimators: 1000
Min samples: 10
= 0.453416149068323
```

Model Selection

To determine which set of parameters are most preferable for the ANN and Random Forests classifiers, a Cross-Validation (CV) process was employed.
A 10-fold K-fold implementation was used to randomly split the data into 10 folds of equal size using scikit’s KFold and cross_val_score library functions. The dataset was prepared in the same fashion as earlier.

```
#Load Libraries for K-Fold, CV Eval
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

```
#Split into 10 equal K-Folds
kf = KFold(n_splits=10)
```

Using created ANN Random Forests ‘eval’ functions, the dataset was classified using the same processes as before, but called within a for loop.

```
#Task 3.3
#Function that uses an ANN, Sigmoid and Logistic functions to classify based on user-inputs (varying estimates 50,500,10000)
def ANN_eval(n,trainx,testx,trainy,testy):
#     trainx,testx, trainy,testy = train_test_split(x,y, train_size=0.9)
    clf = neural_network.MLPClassifier(learning_rate_init = 0.1, solver = 'sgd',
                                       max_iter = 80000, hidden_layer_sizes = (n, n),
                                       activation = 'logistic',
                                       momentum = 0, alpha = 0,
                                       # tol is tolerance -- needs to be this low as improvements are so small
                                       verbose = False, tol = 1e-10,
                                       random_state = 11)
    return clf
```

```
#Function that uses a random forests classifier to train data based on input estimators and minimum samples (50,500,1000)
def RandomForest_eval(e,a,trainx,trainy):
    clf=RandomForestClassifier(n_estimators=e, min_samples_leaf = a)
    return clf
```

An ‘ests’ array denotes the estimates required to train both the ANN and RandomForests_eval classifiers, and on pass, the ANN and Random forests functions are both called and return the value to be cross-evaluated using ‘cross_val_score’

```
#Estimates
ests = [50,500,1000]

#Calculate 50,500,10000 estimates for the ANN
for i in ests:
    clf = ANN_eval(i,trainx,testx,trainy,testy)
    #Print Score
    print(cross_val_score(clf, x, y, cv = kf).mean())
```

```
#Calculate 50,500,10000 estimates for 10 min leaf samples
for i in ests:
    clf = RandomForest_eval(i,10,trainx,trainy)
    #Print Score
    print(cross_val_score(clf, x, y, cv = kf).mean())
```

Below is the output for each classifier, with the appropriate Estimates and minimum leaf samples.

```
Artificial Neural Net:
Estimates:  50
Accuracy:  0.3462684545577679
Estimates:  500
Accuracy:  0.4406319537160658
Estimates:  1000
Accuracy:  0.5027901936882027
Random Forests:
Forests:  50 , Min Leaf Samples: 10
Accuracy:  0.46016234205994466
Forests:  500 , Min Leaf Samples: 10
Accuracy:  0.4505234031849229
Forests:  1000 , Min Leaf Samples: 10
Accuracy:  0.45270021864902
```

Evaluation

From the statistics gathered, the most desirable parameters of the Random Forests Algorithm will be 10000 Forests, with Minimum leaf samples 10.

The most desirable parameters of the ANN are also 10000 Estimates, however it could be argued a similar value of Accuracy is calculated using less trees, which in turn is a more efficient process.

From the data gathered, I believe the ANN to be the preferable method at 10000 Estimates, as it produces the highest accuracy score, this denotes the ratio of the correct predictions to the total number of samples – meaning the ANN is the more accurate classifier.

Bibliography

Breiman, L. (2001). *Random Forests. Machine Learning*. Retrieved from https://doi.org/10.1023/A:1010933404324

Brownlee, J. (2018). *A Gentle Introduction to k-fold Cross-Validation*. Retrieved from MachineLearningMastery.com: https://machinelearningmastery.com/k-fold-cross-validation/

Education, I. C. (2020). *Random Forest*. Retrieved from IBM: https://www.ibm.com/cloud/learn/random-forest

eumetrain. (n.d.). *Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE)*. Retrieved from http://www.eumetrain.org/data/4/451/english/msg/ver_cont_var/uos3/uos3_ko1.htm

Eva Ostertagová. (2012). *Modelling using Polynomial Regression.* ISSN.

Garbade, M. K. (2018). *Understanding K-means Clustering in Machine Learning*. Retrieved from TowardsDataScience.com: https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1

Kenton, W. (2022). *Least Squares Method*. Retrieved from Investopedia: https://www.investopedia.com/terms/l/least-squares-method.asp

Lippmann, R. (1987). *An introduction to computing with neural nets.* IEEE.

Pant, A. (2019). *Introduction to Linear Regression and Polynomial Regression*. Retrieved from TowardsDataScience.com: https://towardsdatascience.com/introduction-to-linear-regression-and-polynomial-regression-f8adc96f31cb

techopedia. (n.d.). *K-Means Clustering*. Retrieved from https://www.techopedia.com/definition/32057/k-means-clustering

Webster, M. (n.d.). Retrieved from https://www.merriam-webster.com/dictionary/polynomial

Wood, T. (n.d.). *What is the Sigmoid Function?* Retrieved from DeepAI: https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function