Task 1

1:1

For my implementation of polynomial regression, I needed to import multiple libraries such as pandas, NumPy and matplotlib. These are so I can manage the data and do certain calculations needed to calculate the correct degree. I then also imported the CVS document as a pandas' data frame and split that data frame into two others depending on their column. I have also imported a function from sklearn to split the data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

#splits the data depending on their column
data_frame = pd.read_csv('pol_regression.csv')
x_train = data_frame['x']
y_train = data_frame['y']
train_df, test_df = train_test_split(data_frame, test_size=0.3)
x_train = train_df['x']
y_train = train_df['y']
x_test = test_df['x']
y_test = test_df['y']
x_train = x_train.sort_values(ascending=True)
y_train = y_train.sort_values(ascending=True)
x_test = x_test.sort_values(ascending=True)
y_test = y_test.sort_values(ascending=True)
```

My polynomial regression function takes the information from the databases given to it, this includes which degree it will be calculating at. The function then calls another function I have that expands the features. Using the output, it then transposes it which flips the data around and then using dot to matrix the data

```
def pol_regression(x, y, degree):
    X = getPolynomialDataMatrix(x, degree)

    XX = X.transpose().dot(X)
    w = np.linalg.solve(XX, X.transpose().dot(y))

    return w
```

This is the matrix function that is called from the pol_regression function, this matrix's the data so that it can properly put into back into the function for the next steps in regressing

```
def getPolynomialDataMatrix(features, degree):
    X = np.ones(features.shape)
    for i in range(1, degree + 1):
        X = np.column_stack((X, features ** i))

    return X
```

1:2

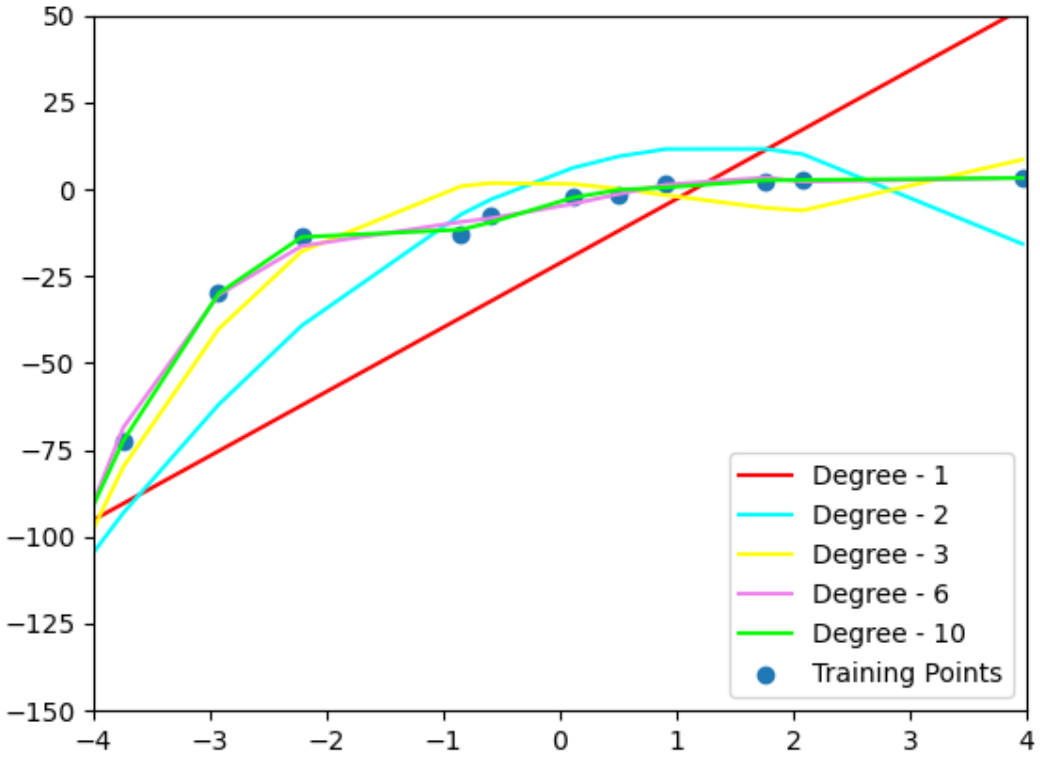
First, I calculate the weights of each of the degree ill be testing at, although commented out I do have code for calculating at the degree of 0, the reasoning of its commenting out is due to not being able to have a degree of 0 without causing an error of some sort. The weight is then used to get test and training data for plotting. This could be fixed with an error detection, but I am not going to include this.

```
#wd0 = pol_regression(x_train,y_train,0)
#Xtest0 = getPolynomialDataMatrix(x_train, 0)
#Xtrain0 = getPolynomialDataMatrix(x_test, 0)
#ytest0 = Xtest0.dot(wd0)
#ytrain0 = Xtrain0.dot(wd0)
#plt.plot(x_train, ytest0, 'green')

# Calculates the weights:
wd1 = pol_regression(x_train,y_train,1)
Xtest1 = getPolynomialDataMatrix(x_train, 1)
Xtrain1 = getPolynomialDataMatrix(x_test, 1)
# Calculates predicted:
ytest1 = Xtest1.dot(wd1)
ytrain1 = Xtrain1.dot(wd1)
# Plots the data on the graph
plt.plot(x_train, ytest1, 'red')

wd2 = pol_regression(x_train,y_train,2)
Xtest2 = getPolynomialDataMatrix(x_train, 2)
Xtrain2 = getPolynomialDataMatrix(x_test, 2)
ytest2 = Xtest2.dot(wd2)
ytrain2 = Xtrain2.dot(wd2)
plt.plot(x_train, ytest2, 'cyan')
```

As you can see from the table below you can see what the results are from the polynomial regression of different degrees are. This can be used to assume that degree 10 is the best for polynomial regression for our data this is because it fits best with out training data, with each degree you can see the relations between the points getting better fitted, this is important to us as we need to make sure we are not overfitting the data and not underfitting the data.



1:3

This is my evaluation of the polynomial regression function, this is used to calculate the root, mean, squared, error of the trained data and test data.

```
def eval_pol_regression(parameters, x, y, degree):
    MSE = np.square(np.subtract(x, y)).mean()
    RMSE = np.sqrt(MSE)
    return RMSE
```

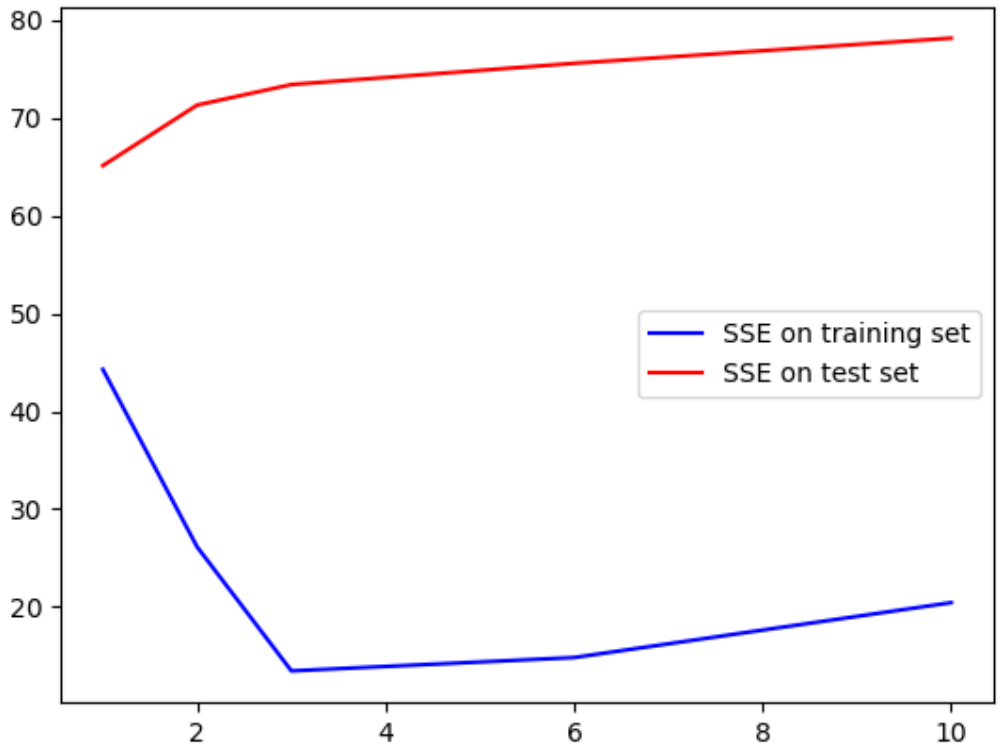
Using the function, I can input the data that I want to be evaluated for each degree, this is then put into a list and plotted on a graph that shows us the effectiveness of the training degree. Although I have the code for the function, I am not certain of its effectiveness and how trustworthy it is. The data used for this is split earlier on in the code.

```
degrees = [1, 2, 3, 6, 10]
#train data evaluation
tRMSE_degree1 = eval_pol_regression(parameters=wd1, x=1, y=ytest1, degree=1)
tRMSE_degree2 = eval_pol_regression(parameters=wd2, x=2, y=ytest2, degree=2)
tRMSE_degree3 = eval_pol_regression(parameters=wd3, x=3, y=ytest3, degree=3)
tRMSE_degree6 = eval_pol_regression(parameters=wd6, x=6, y=ytest6, degree=6)
tRMSE_degree10 = eval_pol_regression(parameters=wd10, x=10, y=ytest10, degree=10)

#test data evaluation
RMSE_degree1 = eval_pol_regression(parameters=wd1, x=1, y=ytrain1, degree=1)
RMSE_degree2 = eval_pol_regression(parameters=wd2, x=2, y=ytrain2, degree=2)
RMSE_degree3 = eval_pol_regression(parameters=wd3, x=3, y=ytrain3, degree=3)
RMSE_degree6 = eval_pol_regression(parameters=wd6, x=6, y=ytrain6, degree=6)
RMSE_degree10 = eval_pol_regression(parameters=wd10, x=10, y=ytrain10, degree=10)

#combinds the veriables to a list
errors_test = [RMSE_degree1, RMSE_degree2, RMSE_degree3, RMSE_degree6, RMSE_degree10]
errors_train = [tRMSE_degree1, tRMSE_degree2, tRMSE_degree3, tRMSE_degree6, tRMSE_degree10]
#plots the data
plt.plot(degrees, errors_test, color='blue')
plt.plot(degrees, errors_train, color='red')
plt.legend(('SSE on training set', 'SSE on test set'))
plt.savefig('polynomial_evaluation.png')
plt.show()
```

As you can see from the graph below the training data has a higher performance rate than the test data, and which is best for the polynomial regression degree, from this data I can see that 4 seems to be the best degree for the equations as it has the higher performance rate. This is different to my assumption on what the best degree would be.



Task 2

2:1

K-means clustering is where you group values based on random values taken and the distance between them and other chosen values in the cluster, this is done multiple times to find the best groups for the values in the cluster. We can set how many groups we want to make in the data, this determines how many values will be selected at random. The distance between values is calculated by square rooting the values which are squared and added together

My kmeans function is designed to determine what group a value belongs to depending on its distance to the randomly selected values that are picked by another function, these values are then grouped. The kmeans function calls the compute_euclidean_distance function, which is responsible for calculating the distance, the initialise_centroids function is also called to select the initial values that the groups are based on

```
def kmeans(dataset, k):
    ds = []
    vs = []
    dis = np.array(ds)
    vis = np.array(vs)
    dm = dataset.to_numpy()
    #stores assigned cluster and distance
    ca = np.zeros((len(dm), 2))
    centroids = initialise_centroids(dataset, k)
    #measures the distance of each point in each row in the dataset
    for vi, v in enumerate(dm):
        centdis = np.zeros(len(centroids))
        for i, centroid in enumerate(centroids):
            d = compute_euclidean_distance(v, centroid)
            centdis[i] = d
            print(d)
            print(i)
            dis = np.append(dis, d)
            vis = np.append(vis, i)
        md = np.min(centdis)
        ac = np.argmin(centdis)
        ca[vi] = np.array([md, ac])
    #creats a new dataset with the chosen clusters as the new column
    cadf = dataset
    cadf['ac'] = ca[:, 1]
    nc = np.zeros([k, len(centroids[0])])
    #calculates the mean of the clusters
    for i, centroid in enumerate(centroids):
        cc = cadf
        cc = cc[cc['ac'] == i]
        cc = cc.drop(['ac'], axis=1)
        cgnp = cc.to_numpy()
        #updates the centroid with mean of each cluster
        for x, val in enumerate(centroid):
            cc = cgnp[:, x]
            mean = np.mean(cc)
            nc[i, x] = mean
    findf = pd.DataFrame(columns=['iteration_step','objective_function_value'])
    findf['iteration_step'] = dis.tolist()
    findf['objective_function_value'] = dis.tolist()
    findf.sort_values('iteration_step', ascending=True,)
    plt.plot(findf['iteration_step'], findf['objective_function_value'], color='black')
    plt.xlabel('iteration_step')
    plt.ylabel('objective_function_value')
    plt.savefig('objective_function_value.png')
    plt.show()
```

This is my function that gets the initial values that are used by the kmeans function to determine which group a value belongs to. Before the initial values are picked, we shuffle the array to make sure its not going to be the same values chosen every time and then returned as an array

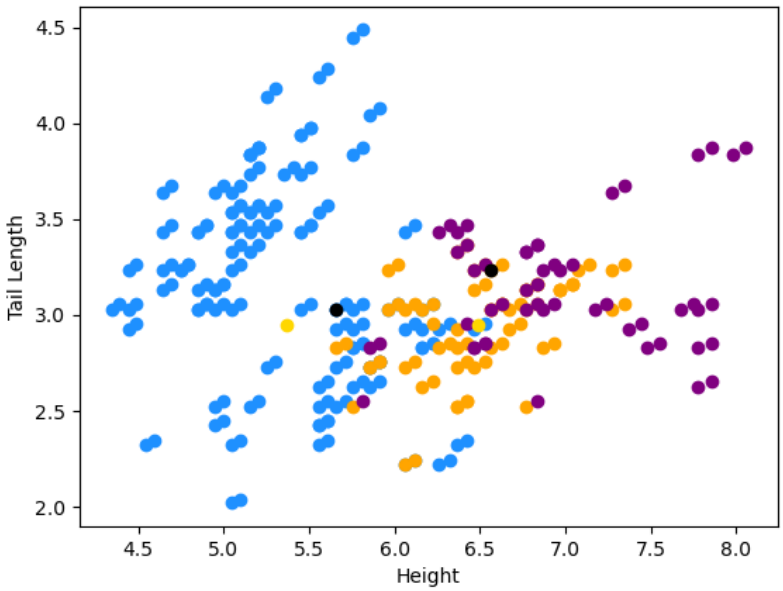
```
def initialise_centroids(dataset, k):
    dt = dataset.to_numpy()
    #shuffels the data
    np.random.shuffle(dt)
    #gets k number of row
    centroids = dt[:k, :]
    return centroids
```

The function calculates the distance between values by looping through the two values dimensions and calculating them, this is then squared so that it can later be used to find the square root, which should be the two values distance from each other, one of the provided values will be the randomly chosen values from the previous function

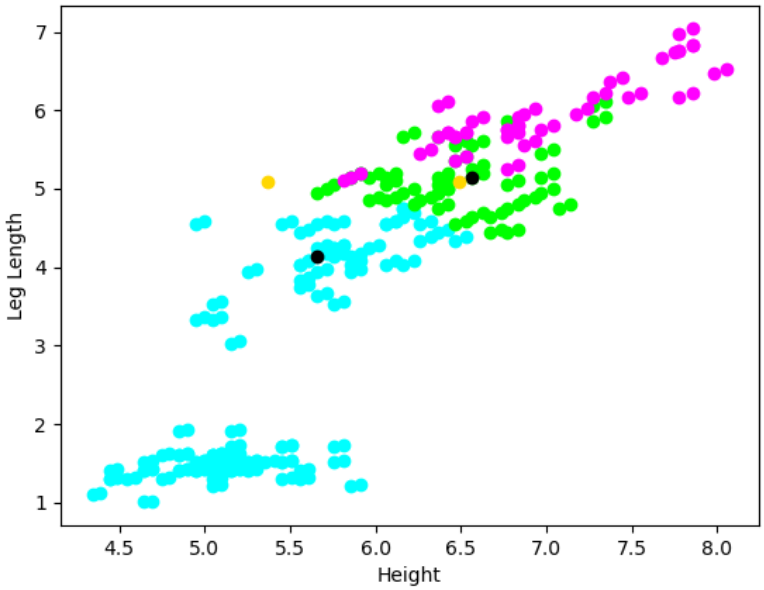
```
def compute_euclidean_distance(vec_1, vec_2):
    t = 0
    for i, value in enumerate(vec_1): t += (vec_2[i] - value) ** 2
    distance = np.sqrt(t)
    return distance
```

2:2

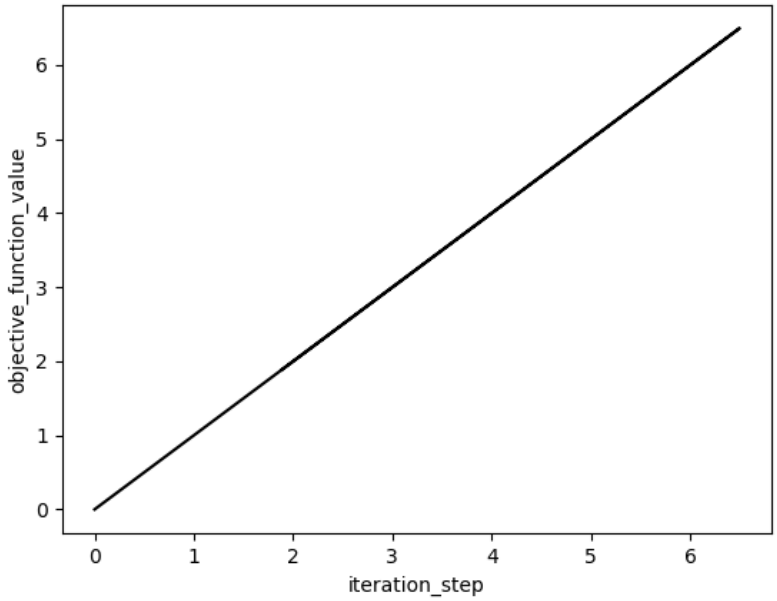
Using the functions, I have created I am able to plot the data in a scatter graph, with putting different colours for each group you can easily see what group each value belongs to, but as you can see there are some values that are closer to other different coloured values than what they apparently belong to. The graph below is the dog's height as the x axis and the tail length as the y axis.



This is the kmeans cluster graph for the dog's leg length, the data is a lot more separated than the one above, this is due to it being different data to the other graph. Like the other graph the hight is the x axis, but unlike the other graph the y axis is leg length of the dogs,



The graph below is supposed to show a line plot of the error rate, however as you can see I was unable to get the graph to correctly plot the data, all the data is available in a data frame and can be printed. The line should be showing a decreasing trend to show that the more training the lower the error rate



3:1

For calculating the summary of the data, I created a function that takes the data, the data then has a column dropped to allow for the calculations to be done as the dropped column is a string. The data is then taken and used to calculate the mean, min, and max of each column in the given data. These

are then taken and put into one panda's data frame and printed. I will be using pandas as it is easy to add multiple data frames together without causing issues

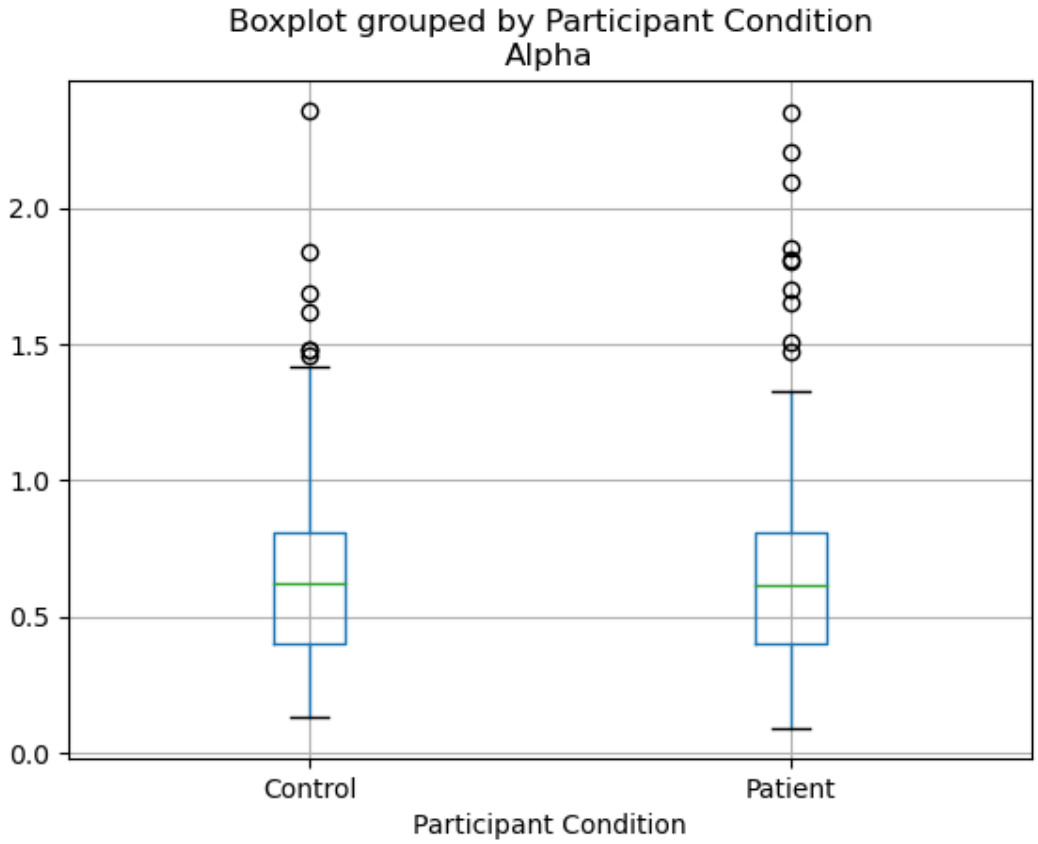
```
def statistical_summary(data):
    #mean values, standard deviations, min/max values
    data = data.drop(columns=['Participant Condition'])
    min = {}
    max = {}
    mean = {}
    min['summary'] = ['min']
    max['summary'] = ['max']
    mean['summary'] = ['mean']
    for i in data.columns:
        min[i] = [data[i].min()]
    for i in data.columns:
        max[i] = [data[i].max()]
    for i in data.columns:
        mean[i] = [data[i].mean()]
    mindf = pd.DataFrame(min)
    maxdf = pd.DataFrame(max)
    meandf = pd.DataFrame(mean)
    sdf = pd.concat([mindf, maxdf, meandf])
    print(sdf)
    return mindf, maxdf, meandf
```

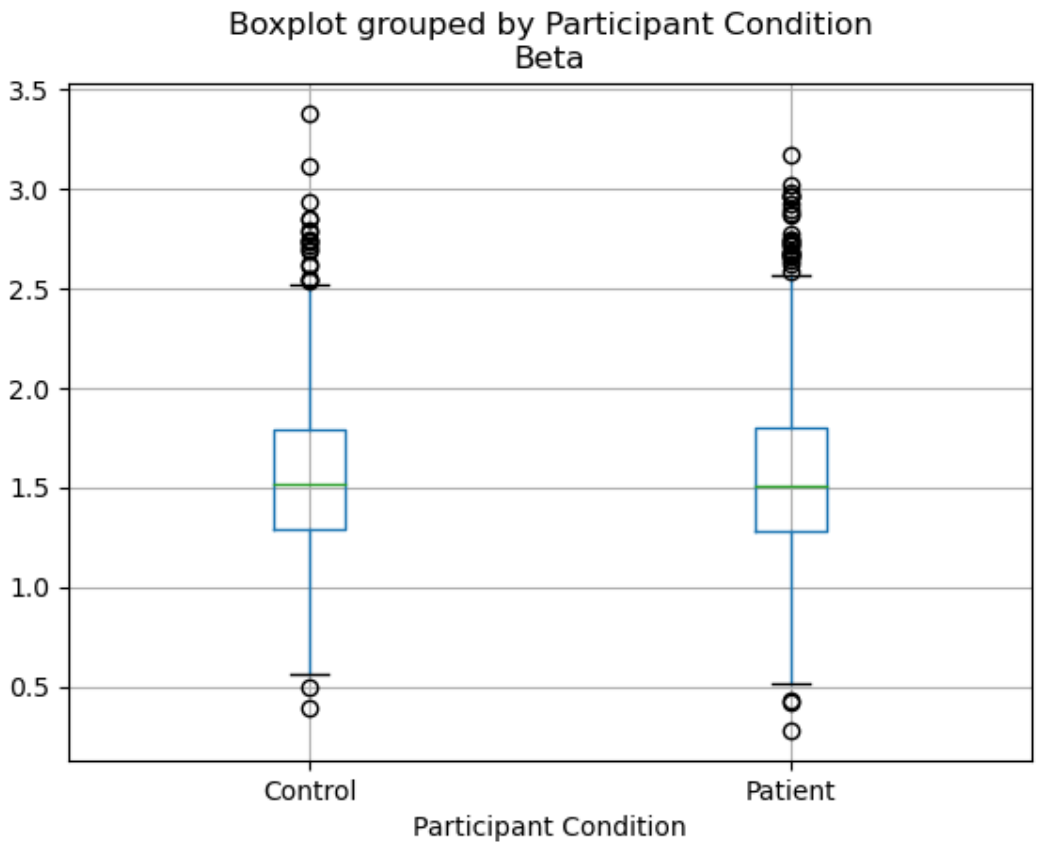
Although this data can be taken and used to box plot, I decided to use an easier method of just plotting a pandas data frame with matplotlib, this is done using a function that splits the data depending on what the Participant Condition column is, this is a reason as to why I just use a pandas data frame as I dropped this column to do the calculations. The function allows for me to chose which column gets plotted, I have done this by just allowing you to assign the chosen in the function's parameters

```
def display_boxplot(data, column):
    data.boxplot(by='Participant Condition', column=column)
    test = f"""boxplot_{column}.png"""
    plt.savefig(test)
    plt.show()
```

These boxplots are then plotted depending on what column is chosen when calling the function, this allows for multiple boxplots to be created without having to run the program multiple times. From

these boxplots you can see the values that I also calculated with another function in the program.





3:2

I created a function for splitting the data in into the test and train data, this also prepares the data to be put into the methods, this is done using a simple train_test_split function that convers the data 90% train and 10% test data, this is to try and yield higher accuracy due to having more data to train from

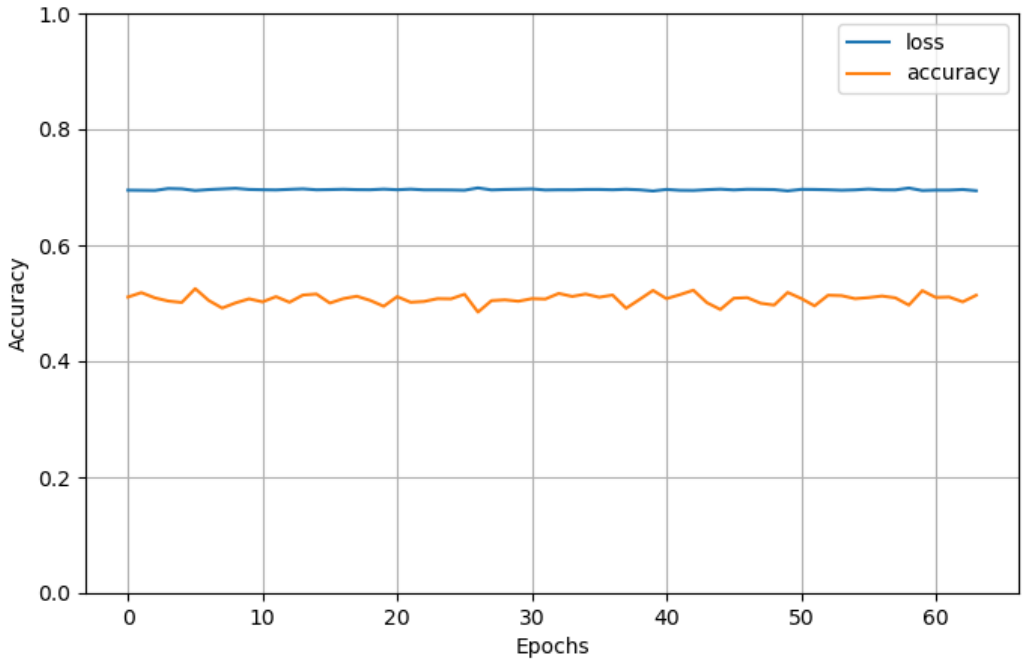
```
def split_data(data):
    print('prep')
    raw = data['Participant Condition'].to_numpy()
    le = preprocessing.LabelEncoder()
    le.fit(raw)
    labels = le.transform(raw)
    data['Labels'] = labels
    traindf, testdf = train_test_split(data, test_size=0.1)
    return traindf, testdf
```

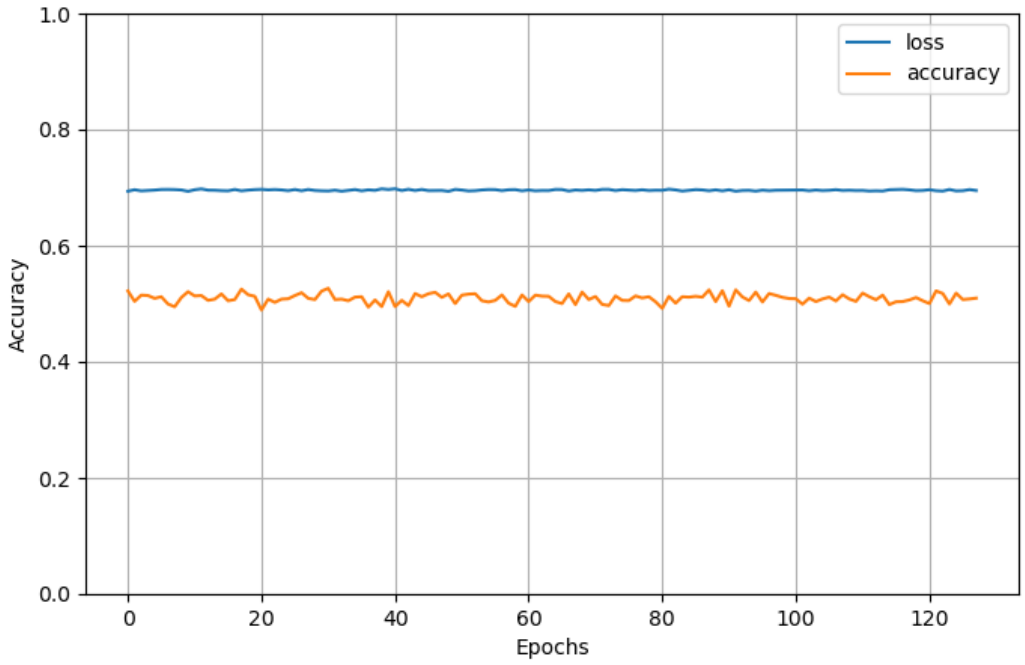
I have created an artificial neural network classifier function that will use the train and test data from the previous function to try and predict if the test data is a Patient or a Control sample, this is done with different epochs, this is to increase the accuracy of the neural network, the data from this is then plotted on a graph to show the accuracy of the training at each level of epoch given

```
def ann(train, test, epochs):
    # Select values to create X and Y:
    tr_y = train[['Labels']].to_numpy()
    te_y = train[['Labels']].to_numpy()
    tr_x = train[['Alpha', 'Beta', 'Lambda','Lambda1', 'Lambda2']].to_numpy()
    te_x = test[['Alpha', 'Beta', 'Lambda','Lambda1', 'Lambda2']].to_numpy()

    # Reshapes the Y values
    tr_y = np.reshape(tr_y, (-1, len(tr_y)))
    te_y = np.reshape(te_y, (-1, len(te_y)))
    tr_y = tr_y[0]
    te_y = te_y[0]
    # Create neural network model
    model = keras.models.Sequential([keras.layers.Flatten(input_shape=[5]),keras.layers
    model.compile(loss='binary_crossentropy',optimizer='sgd',metrics=['accuracy'])
    # Train the model:
    h = model.fit(tr_x, tr_y, epochs=epochs)
    # Present any metrics that it produces:
    hdf = pd.DataFrame(h.history).plot(figsize=(8, 5))
    test = f"""accuracy_graph_epoch_{epochs}.png"""
    plt.grid(True)
    plt.gca().set_ylim(0, 1)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.savefig(test)
    plt.show()
```

As you can see in the graphs below increasing the epochs amount did not seem to increase the accuracy of the training, we can use some code to find the best for us to use.





3:3

I have set up a function for an artificial neural network classifier 10-fold, this will allow me to see what the best number of neurons is to get the highest accuracy. I decided to split the data again in the function to allow the data to not be like the other neural network I have. In the function I create the neural network and then compile the model. I then selected certain values as the x and y of test and train which are then reshaped so that they can then be trained and tested by the rest of the function

```python
def ann10Fold(df, neurons, k, epochs):
    print('prep')
    raw = df['Participant Condition'].to_numpy()
    l = preprocessing.LabelEncoder()
    l.fit(raw)
    labels = l.transform(raw)
    df['Labels'] = labels
    traindf, testdf = train_test_split(df, test_size=0.1)
    # Create the neural network:
    model = keras.models.Sequential([keras.layers.Flatten(input_shape=[5]),keras.layers
    # Complie the model:
    model.compile(loss='binary_crossentropy',optimizer='sgd',metrics=['accuracy'])
    k = KFold(n_splits=k, random_state=None)
    acc_score = []
    f = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    f = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    acc_score = np.zeros(shape=(len(f), 0))
    for train_index, test_index in k.split(df):
        train = df.iloc[train_index]
        test = df.iloc[test_index]
        # Select the values to create X:
        train_x = train[['Alpha', 'Beta', 'Lambda','Lambda1', 'Lambda2']].to_numpy()
        test_x = test[['Alpha', 'Beta', 'Lambda','Lambda1', 'Lambda2']].to_numpy()
        # Select the values to create Y:
        train_y = train[['Labels']].to_numpy()
        test_y = test[['Labels']].to_numpy()
        # Reshape the Y values:
        train_y = np.reshape(train_y, (-1, len(train_y)))
        train_y = train_y[0]
        # Train the model here:
        history = model.fit(train_x, train_y, epochs=epochs)
        y_pred = model.predict(test_x)
        print(y_pred)
        print(test_y)
        ypred1d = y_pred.flatten()
        acc = accuracy_score(ypred1d.round(), test_y)
        acc_score = np.array([])
        acc_score = np.append(acc_score, acc)

    print(acc_score)
    print(f)
    print(k)

    return acc_score
```

From the results below you can see that I have an error somewhere in the calculations that is causing all my accuracy to be set to the same value, this does not affect the results produced by the function, only how I have outputted this data

```
[0.12149533 0.12149533 0.12149533]
```