Task 1

Integration of Polynomial Regression

The *pol_regression* function calculates coefficients for the features and y-axis data supplied to it. The structure of the data needs to be prepared, so *getPolynominalDataMatrix* is called. Then the least squares method is used in order to calculate the polynomial weights.

```
def pol_regression(features_train, y_train, degree):
    X = getPolynomialDataMatrix(features_train, degree) # get prepared array

    # perform least squares algorithm to calculate weights
    XX = X.transpose().dot(X)
    w = numpy.linalg.inv(XX).dot(X.transpose().dot(y_train))

    return w

def getPolynomialDataMatrix(x, degree):
    # prepare data
    X = numpy.ones(x.shape) # create array with same size as x
    for i in range(1, degree + 1): # for every degree, starting with 1
        # add new dimension to array, containing given data raised to the degree
        X = numpy.column_stack((X, x ** i ))
    return X
```

Figure 1 *pol_regression* implementation

Regressing Polynomials

Figure 2 shows the polynomial function at a number of different degrees. A single degree doesn't appear to fit the data well, as it fails to replicate the curves of the training data. The line showing 2 degrees is a step closer, as it almost mimics the plateau seen near $x=0$. However, the data seems best represented with 3 degrees. The y increase clearly slows to plateau along with the training data, even reflecting the slight decrease along the y axis close to $x=0$. The line curves upwards again to show the end of the plateau near $x=4$. When the polynomial is calculated with 6 degrees, the estimation is similar to the previous fit, but less pronunciation is placed on the slight decrease in the plateau. However, the polynomial calculated with 10 degrees shows clear inaccuracy. The line plummets with no indication that it should do so, rendering it wholly inaccurate for a large portion of the -5 to 5 range.

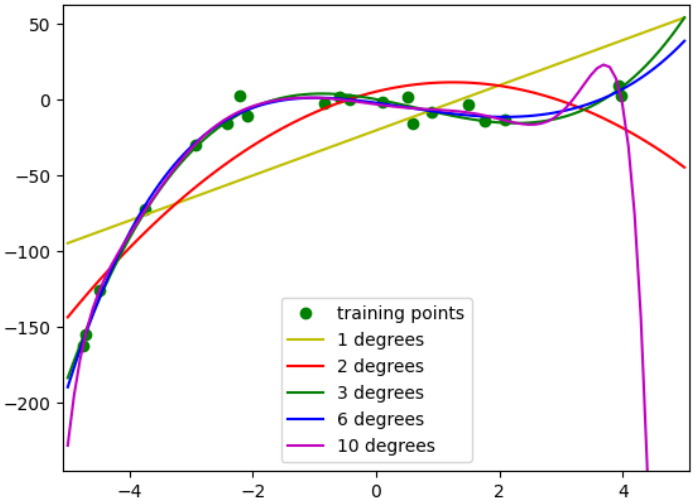


Figure 2 Predicted y values calculated with various degree values

The code to generate the graph seen in *figure 2* can be seen in the appendix of this report, labelled *Polynominal graph generation.* The polynomial result data is stored in an array for each of the training degrees. Each degree array is then plotted in the problem space, which is -5 to 5.

Evaluation

The MSSE was calculated using the *eval_pol_regression* function, shown in *figure 3*, which begins by preparing the *x axis* data to be modified by the calculated weights using *getPolynominalDataMatrix*. The model is applied to the testing data by modifying each *x axis* point of data with the calculated weights and is subtracted by the respective known *y* value. This value is squared to remove any polarity, since only absolute values should be dealt with to avoid errors cancelling each other out. This calculates the absolute error between each predicted and actual value. The MSSE is the mean average of all of these values, and represents the average inaccuracy for the given testing data.

```
def eval_pol_regression(parameters, x, y, degree):
    prepared_x = getPolynomialDataMatrix(x, degree) # get results
    # and measure mean squared sum of errors
    msse = numpy.mean(((prepared_x).dot(parameters) - y) ** 2)
    return msse
```

Figure 3 eval_pol_regression implementation

Figure 4 shows the mean sum of squared errors (MSSE) for each trained degree of the polynomial, plotting errors for both the training and the testing set. Overfitting is when the polynomial is too specialised towards the training set. The polynomial creates artificial curves between the training points but still passes through the training point themselves. This renders evaluations using the training set useless, as it is not a reflection of real-world data and only intended to suggest trends to the model. It is for this reason that more attention should be paid to error intensity on the test set, as this better represents the model's performance for unseen data.

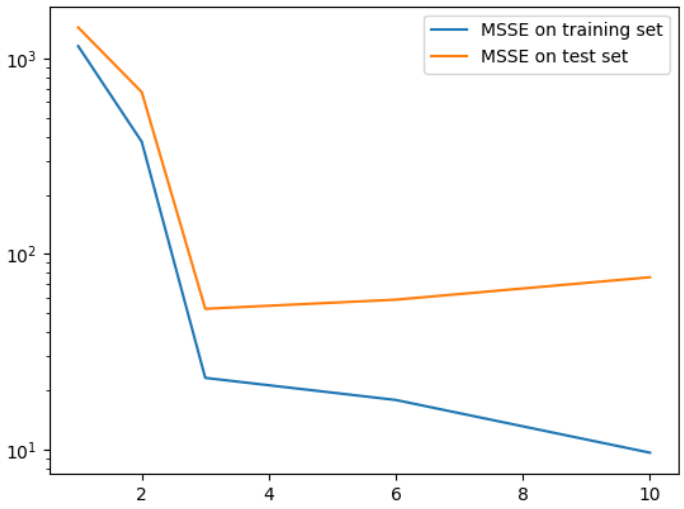


Figure 4 Mean Sum of Squared Errors for each trained degree

Using the test set line, it is possible to figure out where the model begins to overfit. Overfitting is characterised by an upwards trend after the initial decline, the point of which can be seen at $x = 3$. The training set will continue to decline in error intensity as the polynomial further flexes to accommodate noise in the training set. This flexing between the noise in the training points results in inaccuracies

when running the model on test data, seen from $x < 3$ onwards, which will only worsen as the degrees increase.

The code to generate this graph can be seen in the appendix, labelled *Mean sum of squared errors graph generation*. The accuracy arrays are created and then added to. They are then plotted against a list of integers, which are the degrees to which the polynomials were fitted.

The complete code for task one can be seen in the appendix labelled *Task1.py.*

Task 2

Description of K-means Clustering

The K-means algorithm is an unsupervised method of classifying data into distinct categories. It relies on creating a centroid for each cluster of data, which describes the average placement of each data point in the cluster. To clearly show the steps involved in this algorithm, the K-means method is talked through below.

Firstly, the centroids need to be created. The centroids for a simple implementation of K-means can be generated one of two ways. They can be randomly sampled data points, taking on the features of some existing data point, or they can be randomly generated points somewhere within the upper and lower bounds for each feature.

Next, each point will be assigned to a cluster, depending on which centroid it is closest to. This assignment is not permanent and may be changed in later loop iterations. This assignment is calculated using euclidean distance. Euclidean distance measures the disparity between two points by applying them to some dimensional space. For example, if two features are being used to group the data, the disparity between the data point and the centroid will be calculated with two dimensions. Similarly, datasets with five features will have their euclidean distance to the centroid calculated in five dimensions. To calculate euclidean distance, the formula shown in *figure 5* can be used, where x and y are the two points between which the distance is being measured, and n is equal to the number of features of the data point.

$$\textbf{\textit{distance(x, y)}} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$$

Figure 5 A formula to calculate euclidean distance with multiple dimensions

Using this formula, euclidean distance can be trivially implemented for any amount of numeric features a dataset may have. The reason the values are being squared and subsequently square-rooted is to remove any polarity from the measurements to produce an absolute measurement.

After cluster assignment has taken place, the centroids will need to be recalculated, as their random position may not represent their cluster's positioning. The new centroid coordinates should be equal to the average of each feature for every data point assigned to that centroid's cluster. For example, the new y coordinate of centroid A should be equal to the mean average of feature y for all data points assigned to cluster A.

The process of reassigning data points and recentering centroids continues until the location of centroids stabilises. Such stability indicates a solution. However, this solution may not provide

maximum separation between the classes. This is because the data may have a number of local minimums that differ to a global minimum, meaning there may be more than one way in which the classes can be split and a stability can be reached. With randomised starting points for the centroids, which would be considered an "input" to the model, finding the global minimum successfully is largely left to chance. With a more careful selection of centroid starting points, this chance can be increased. More effective centroid initialisation methods include K-means++ and naive sharding (Mayo, 2020), but such methods fall out of the scope of this report.

Implementation of K-means Clustering

The implementation of the K-means algorithm consists of three functions and the main script.

Firstly, all the required functions are defined. *Figure 6* shows the integration of the euclidean distance calculation function.

```
def compute_euclidean_distance(vec_1, vec_2):

    distance = float('inf') # create placeholder value
    if (len(vec_1) == len(vec_2)): # if possible to calculate distance
        sum = 0 # initialise running total
        for feature_index in range(0, len(vec_1) - 1): # loop through each feature for both
vectors
            squared = (vec_1[feature_index] - vec_2[feature_index]) ** 2 # get squared
distance
            sum = sum + squared # add to running total

        distance = math.sqrt(squared) # get square root and return result

    return distance
```

Figure 6 Euclidean distance calculation implementation

This implementation will work for all dimensions of data. Firstly, a placeholder distance value is created so that, if invalid values are passed, an infinitely high distance will be returned. The vectors' validities are then checked, ensuring they are of the same length. A variable is created to hold the sum of the squared distances. Each featureset is looped through and calculated for each vector, using the formula mentioned earlier in this report. The collective squared distance is square rooted, resulting in a distance that is then returned. The values are squared and then square rooted to remove any polarities from the values, resulting in a more simple absolute distance, rather than a distance that could be positive or negative.

```
def initialise_centroids(dataset, k):
    centroids = []
    # randomly initialise centroids using range mechanism
    for i in range(0, k):
        height = random.uniform(min(dataset["height"]), max(dataset["height"]))
        tail = random.uniform(min(dataset["tail"]), max(dataset["tail"]))
        leg = random.uniform(min(dataset["leg"]), max(dataset["leg"]))
        nose = random.uniform(min(dataset["nose"]), max(dataset["nose"]))

        centroids.append([height, tail, leg, nose])

    return centroids
```

Figure 7 Simple centroid initialisation function

If the random sample method was implemented, some mechanism would need to be implemented that tracks which samples have already been selected to make sure they aren't selected a second time. In order to avoid this excess complexity, *figure 7* shows that the random within range approach was taken, where identical centroids are highly improbable, albeit still mathematically possible.

```
def kmeans(dataset, k):

    dataset = dataset.values # remove columns for now

    changed = True  # tracks stability
    old_k = []

    # used for evaluation
    cluster_assignment_history = []
    k_history = [k]

    while(changed): # Assign each data point to a cluster and calculate new centroid
position
        cluster_assignment = [] # This will hold cluster assignments for each record
        # For every record in dataset
        for index in range(len(dataset)):
            distances = [] # create intermediary place to store distances
            for centroid in k:
                # calculate distance from each centroid
                distances.append(compute_euclidean_distance(dataset[index], centroid))

            # get id of shortest distance in array (which is equal to the respective
centroid id)
            closest_centroid_index = numpy.argmin(distances)
            cluster_assignment.append(closest_centroid_index) # save centroid id

        # re-center centroids
        numpy_results = numpy.array(cluster_assignment)
        # Use cluster assignment to group values and produce averages. Assign averages to
centroids
        k = pandas.DataFrame(data.values).groupby(numpy_results).mean().values
        k_history.append(k)

        if numpy.array_equiv(k, old_k): # If stability reached
            changed = False  # stop looping next loop

        cluster_assignment_history.append(cluster_assignment) # save cluster info for eval
        old_k = k # save current centroid values to see if they change

    # Evaluate K-means
    eval_kmeans(dataset, cluster_assignment_history, k_history)

    return k, cluster_assignment
```

Figure 8 K-means algorithm implementation

The K-means function, shown in *figure 8,* begins with removing the headers from the dataset, as they are not needed for k-means. Then, *changed* and *old_k* are created, which will later assist with deciding when a solution has been found. The main loop of assigning centroids and altering centroid positions then begins. The dataset is looped through, and the closest centroid is found and saved in the *cluster_assignment* array. To clarify the structure of *cluster_assignment*, it will contain a list of cluster IDs, ordered with respect to the dataset's ordering. That means, if you had the ID of a record in the dataset, you could use the same ID to retrieve its associated cluster ID from *cluster_assignment.* The *cluster_assignment* array is then converted to a numpy array. This is because the mean averages for each column need to be calculated, which means the dataset needs to be grouped by cluster. Using the

cluster assignments as a map allows the data to be grouped into their clusters and their average values can be easily retrieved with *mean()*. The centroid data is appended to *k_history* so it can later be evaluated. At this point, a solution may have been reached, so the new centroids are measured against the centroids from the previous iteration. If the centroid positions are the same, stability has been reached, meaning the algorithm's performance can be evaluated and K values and the *cluster_assignment* array can be returned. However, if the solution has not yet been found, the loop should continue. The evaluation function, *eval_kmeans* draws assessment graphs from the historic data given to it, producing *figure 9*. The *eval_kmeans* function can be found in the appendix.

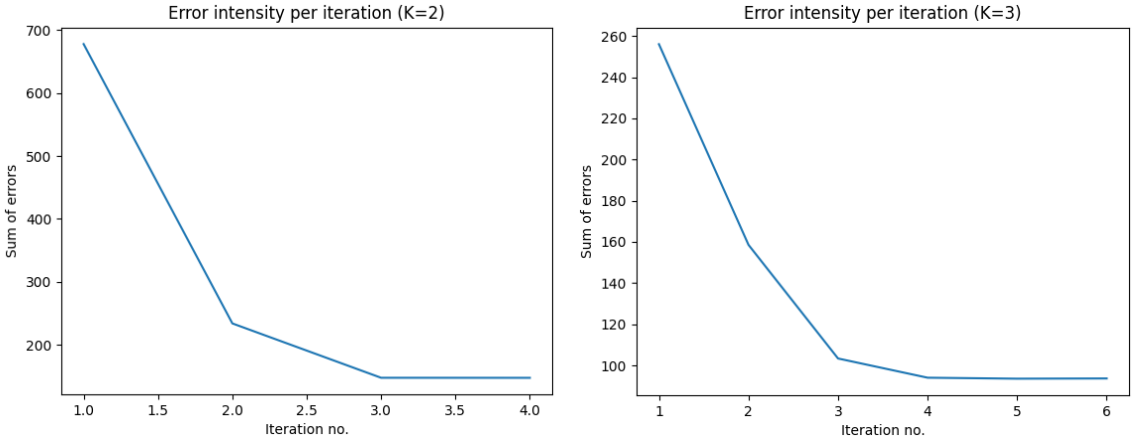


Figure 9 K-means evaluation graphs, showing sum of errors over iteration count

The K-means algorithm produces cluster positions and a cluster assignment array. These are plotted with the dataset and shown in *figure 10.*

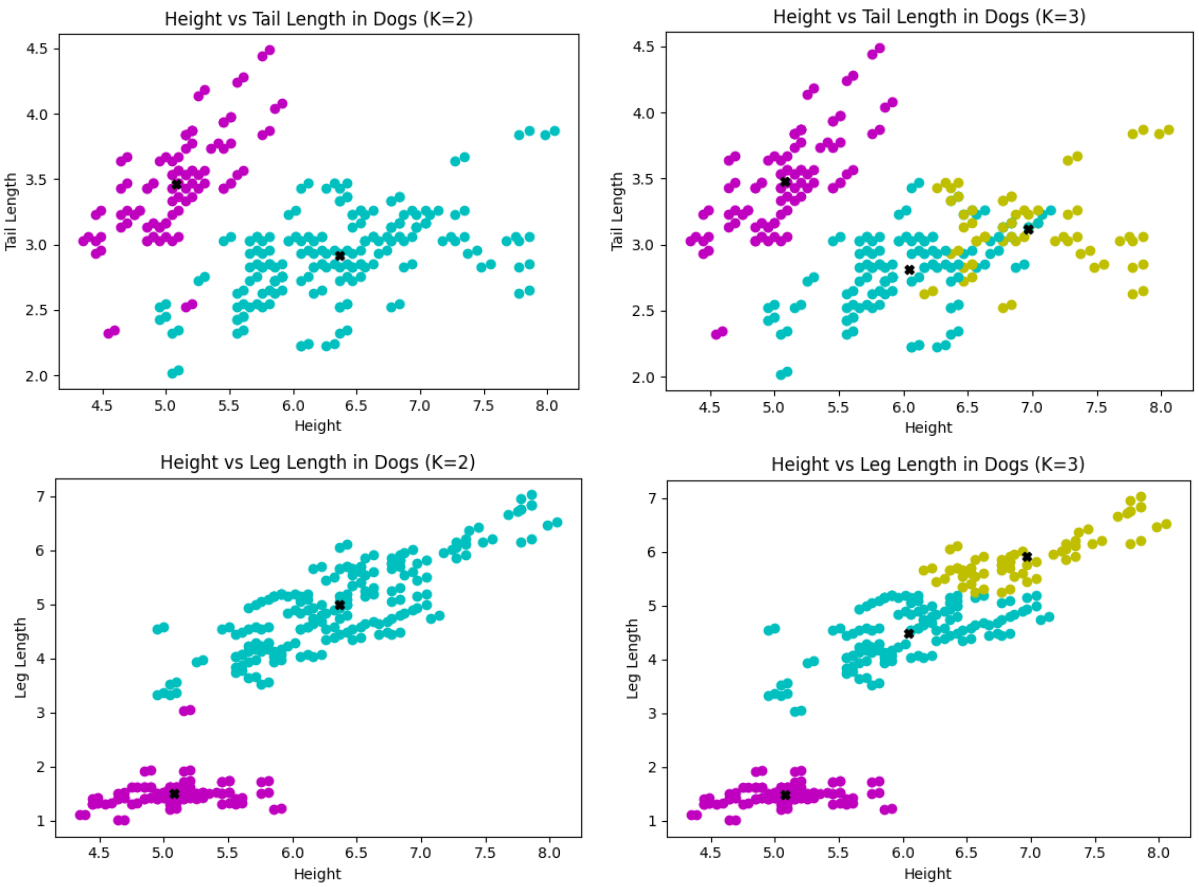


Figure 10 Resultant plots of K-means outcome

After the main functions have been defined, the dataset is read from the CSV file, as seen in *figure 11*. The columns are replaced with shorthand names for two reasons. Firstly, the default column names contain spaces, requiring pandas to switch from its C engine to its Python engine. This can result in much slower processing (Kumar, 2019). The second reason is to produce a more succinct experience viewing and interacting with the project code, as the engine switch warning in the console is avoided altogether and the shorter column names are less visually distracting in the code editor.

```
column_names = ["height", "tail", "leg", "nose"]
data = pandas.read_csv("Task2 - dataset - dog_breeds.csv", names=column_names, header=0)

k_num = 2
k = initialise_centroids(data, k_num)
k, cluster_assignment = kmeans(data, k)
show_plots(data, cluster_assignment, k)

k_num = 3
k = initialise_centroids(data, k_num)
k, cluster_assignment = kmeans(data, k)
show_plots(data, cluster_assignment, k)
```

Figure 11 The main program calling the various functions of the project

The centroids are initialised and the K-means algorithm is run for each value of *k*. The resultant charts are then shown. The function *show_plots* has been included in the appendix with the label *Implementation for show_plots()*. The complete python file for this task has also been included, titled *Task2.py*.

Task 3

Data import, summary, pre-processing and visualisation

The data is read from the CSV into a Pandas DataFrame with custom *dtypes* to ensure correct handling of the data, the process of which can be seen in *figure 12*.

```
dtypes = {"Image number": int, "Bifurcation number": int, "Artery (1)/ Vein (2)": int,
        "Alpha": float, "Beta": float, "Lambda": float, "Lambda1": float, "Lambda2": float,
        "Participant Condition": str}
data = pandas.read_csv("Task3 - dataset - HIV RVG.csv", dtype=dtypes) # read data with
custom
        dtypes
# create column shorthand column names for easier handling
data.columns = ["image", "bifurcation", "arteryvein", "alpha", "beta", "lambda", "lambda1",
        "lambda2", "status"]

data_summary = data.agg(["mean", "std", "min", "max"])
```

Figure 12 Dataframe initialisation

A data summary is also created, comprising of mean, standard deviation, minimum, and maximum values. The summary statistics themselves are shown in *figure 13*, and their calculation can be seen in on the final line of *figure 12*. They have been gathered using the aggregate function, passing a custom list of which functions to run for each column of data.

	image	bifurcation	arteryvein	alpha	beta	lambda	lambda1	lambda2	status
mean	107.355009	6.220597	1.573118	0.615305	1.556093	0.765141	0.981465	0.741929	NaN
std	58.933207	4.103341	0.494702	0.265245	0.387726	0.172840	0.130492	0.159205	NaN
min	1.000000	1.000000	1.000000	0.092770	0.283299	0.304582	0.390920	0.309526	Control
max	203.000000	25.000000	2.000000	2.356406	3.376731	1.535059	1.467637	1.246102	Patient

Figure 13 Summary statistics of the HIV RVG dataset

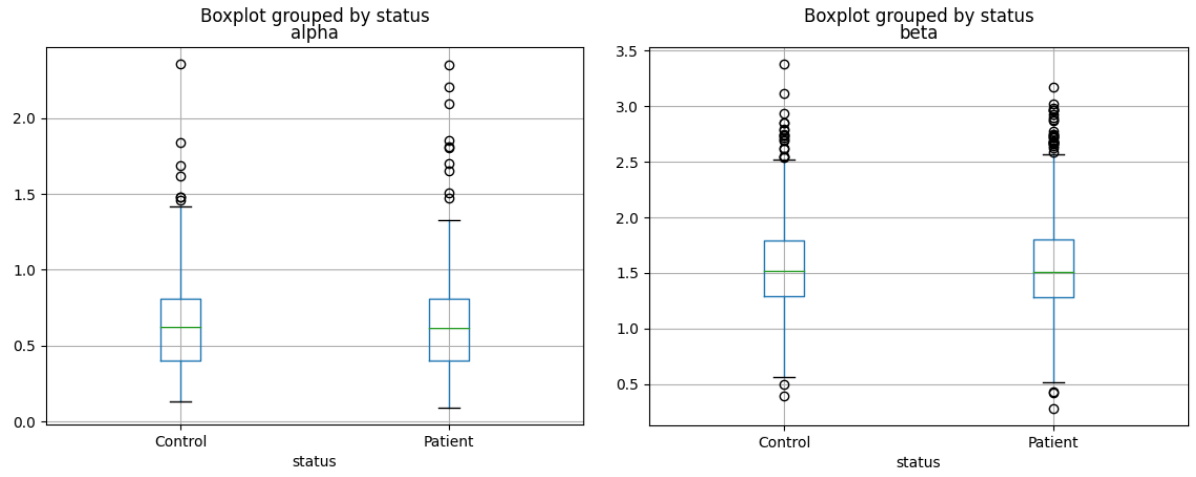


Figure 14 Boxplots of *alpha* and *beta* values

```
alpha_boxplot = data.boxplot(column="alpha", by="status")
beta_boxplot = data.boxplot(column="beta", by="status")
plt.show()
```

Figure 15 Code to draw boxplots from dataset

The code that creates the boxplots of *figure 14* can be seen in *figure 15*. From these boxplots, it is possible to see the outliers in the data. They can be identified by residing beyond the whiskers of the boxplot, notified with a black "O". The data should be standardised, as they have slightly different ranges. To standardise the data is to transform the data so that it has a distribution of zero and a typical deviation of one, so that the value has a universal scale and can be compared with other values of different scales (Frost, undated). Also, if the data wasn't standardised, columns with a commonly higher range of values may have more weight than those with a lower range during the calculation, potentially causing a bias in results. However, standardisation is sensitive to outliers, so they must be removed before performing standardisation. The process of cleaning and standardising the data can be seen in *figure 16*.

```
def clean_normalise(data, column):
    std = 3 # remove row not within x standard deviations
    # calculate z-score, relative to population mean, and remove all rows that fall outside
threshold
    data = data[((data[column] - data[column].mean()) / data[column].std()).abs() < std]

    # use the maximum value to normalise the data[column] data to values between 0 and 1
    data[column] = (data[column] - data[column].min(axis=0)) / (data[column].max(axis=0) -
data[column].min(axis=0))

    return data
```

Figure 16 Outlier removal and normalisation technique

Originally, the outliers were being specified based on their quartile. The top and bottom 1% were being removed without discrimination. Although this was sure to remove outliers, it may also have been removing useful data, and in any case, assuming that outliers existed in the data was a faulty assertion to make. Instead, outliers are now removed based on many standard deviations they are away from zero. To calculate this, the column average is subtracted from the raw value, and this new value is divided by the standard deviation of the data. This creates a z-score for the data point, which can be used to decide if the data point is valuable to the analysis method. The threshold to identify outliers is 3 standard deviations away from the mean (Dienes, 2011), and if a data point is over that threshold, it is removed. This method is far more reliable at removing outliers. *Figure 17* shows the boxplots and summary statistics after the data has been stripped of outliers and normalised. Although the boxplots still identify some outliers, these are far less influential on the remaining data than the outliers already removed.

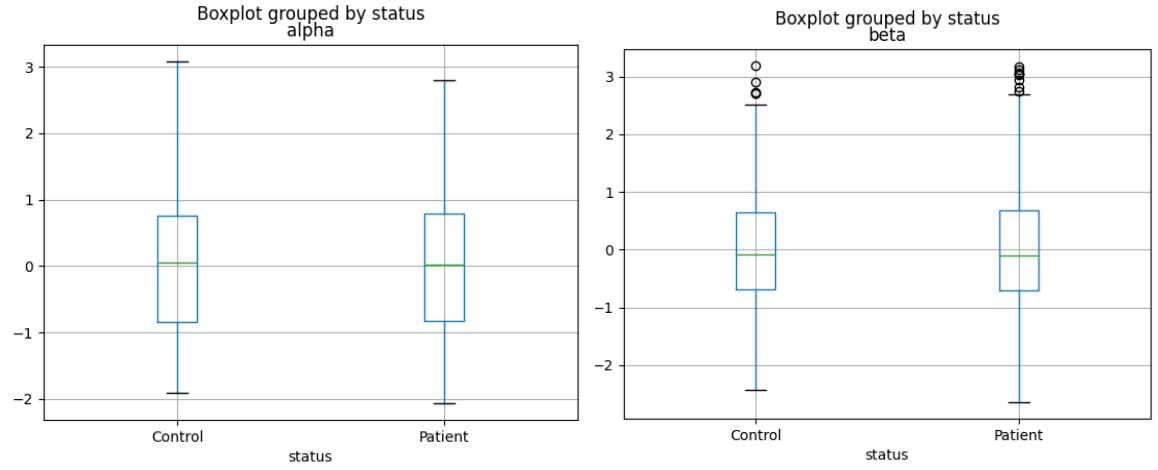


	image	bifurcation	arteryvein	alpha	beta	lambda	lambda1	lambda2	status
mean	107.652840	6.170389	1.574984	-0.004472	0.005627	-0.002910	-0.000239	3.808908e-16	NaN
std	58.901218	4.082496	0.494424	0.998014	0.966372	0.997770	1.000070	1.000000e+00	NaN
min	1.000000	1.000000	1.000000	-2.060026	-2.635359	-2.718192	-3.165102	-2.758776e+00	Control
max	203.000000	25.000000	2.000000	3.076510	3.180213	2.454028	3.134792	2.734546e+00	Patient

Figure 17 Boxplots and summary statistics after data has been cleaned and normalised

Designing Algorithms

The data was split using the *train_test_split* function from *sklearn*. This function takes two arrays and splits each one in two, returning four arrays. The chunk size is determined by an argument called *test_size*, which in this instance, is set to 0.1 to signify that 10% of the data passed should form training data. Originally, the data split was occurring within the loop that creates and fits the classifier, and the *random_state* field was being used to ensure consistent data splits across loop iterations. However, it is more efficient to simply call it once and pass each split to the classifier function.

Since the classifier would be run multiple times, it made sense to put the classifier in a loop. This loop can iterate through a list of epoch lengths, defined in integers, and the epoch length value can be given to SKLearn's *MLPClassifier*.

The data split and function call can be seen in *figure 18,* and the function implementation can be found in *figure 19*.

```
# split dataset into training and testing data, 9:1 ratio, with a replicable split
x_train, x_test, y_train, y_test = train_test_split(x_vector, y_vector, test_size=0.1,
random_state=30)
epochs = [50, 100, 150, 200, 250, 300] # define epoch counts
mlpc_scores = mlpc_train_epochs(x_train, x_test, y_train, y_test, epochs) # train
```

Figure 18 Splitting dataset and calling classifier function

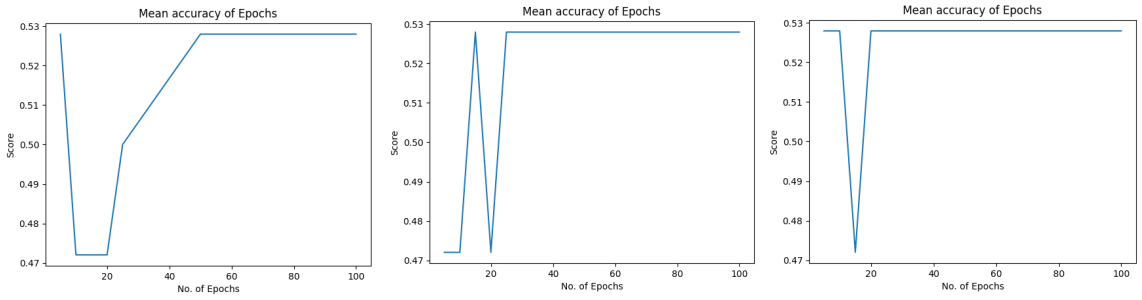
```
def mlpc_train_epochs(x_train, x_test, y_train, y_test, epochs):
    scores = []
    for iterations in epochs:
        # create ANN classifier
        clf = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(500,2),
activation="logistic", max_iter=iterations)
        clf.fit(x_train, y_train) # "train" or fit classifier into model
        scores.append(clf.score(x_test, y_test)) # save score
    return scores
```

Figure 19 Classifier function

A multilayer perceptron classifier was chosen to implement an artificial neural network, as it is well trusted and configurable to the needs of the project. An artificial neural network consists of many nodes, called perceptrons, connected together and arranged into layers. There are input layers, hidden inner layers, and output layers, and data flows into the input layers, through the hidden layers, and the output layers are triggered, showing some result or estimation.

The input layers are mapped to how many features are chosen as input to the classifier, in this case 5, which are then connected to the hidden inner layers. For this project, it was decided that 2 hidden layers of 500 perceptrons in each layer should make up these intermediary perceptrons. There are two output nodes, as the classifier was classifying whether people belonged to the “patient” or “control” group. The “status” column that holds their group membership was changed to a label column, consisting of 0 for a control record, and 1 for a patient record. This is because the classifiers being used are only compatible with numerical output groups. If the problem scenario necessitated three or more output groups, the groups would be named 0, 1, 2, continuing until all groups had an assigned number.

After running the code a number of times and noting the results, the classifier wasn’t running quite as expected. *Figure 20* shows the rate at which correct guesses were made at three different occasions of running the code, along with the score array for each occasion.




```
Run 1:
[0.5279503105590062, 0.4720496894409938, 0.4720496894409938, 0.4720496894409938, 0.5, 0.5279503105590062,
0.5279503105590062]

Run 2:
[0.4720496894409938, 0.4720496894409938, 0.5279503105590062, 0.4720496894409938, 0.5279503105590062,
0.5279503105590062, 0.5279503105590062]

Run 3:
[0.5279503105590062, 0.5279503105590062, 0.4720496894409938, 0.5279503105590062, 0.5279503105590062,
0.5279503105590062, 0.5279503105590062]
```

Figure 20 Classifier score for each code runthrough

These results suggest that the model is simply predicting the most common value in the training set and taking a look at the results from *clf.predict()* in each iteration can confirm this. This could be an issue with data preprocessing, the dataset, or the setup of the classifier itself.

Next, the random forest classifier was trained. A random forest classifier is a method of machine learning that combines many decision trees to make a better decision.

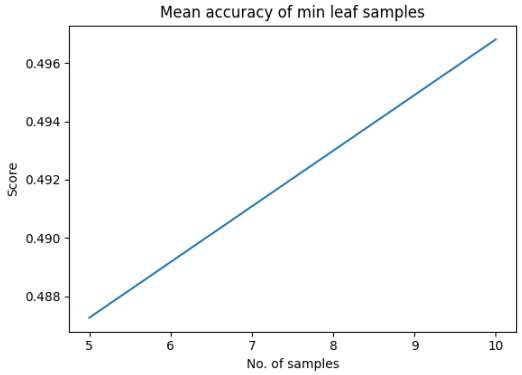


Figure 21 Random Forest score

A decision tree is a collection of nodes that are arranged into a hierarchy. Each node makes some decision about the data, passing it to its appropriate child node. An assignment is made when the data point reaches a leaf node, which is a node without any children nodes. Leaf nodes define any data passed to it as a specific class, in this case 0 or 1, or "control" or "patient".

A random forest, which is what is being trained here, generates many random decision trees, 1000 in this case, and consolidates their results into a single result. Two random forest classifiers were trained, one with a minimum of 5 minimum samples per leaf, and another with 10. This value dictates how little samples in each leaf there can be present before it is ignored and not used in the final model. This has the effect of smoothing the model out, making it more generally applicable (SciKitLearn, undated). The decision tree score can be seen in *figure 21,* and only a marginal improvement can be seen between using 5 and 10 minimum samples per leaf.

Model Selection

The data is split into 10 nearly equal parts using *numpy.array_split(),* seen in *figure 22.* Next, the relevant classifiers are created. Three artificial neural networks are created, with 50, 500, and 1000 nodes in the hidden layers respectively. Three random forests are created in a similar fashion to the neural networks, but with 50, 500, and 1000 decision trees.

```
# split data into 10 (nearly) equal parts
data_arrays = numpy.array_split(data, 10)

# create neural networks
mlpc_50 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(50,2), activation="logistic", max_iter=100)
mlpc_500 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(500,2), activation="logistic", max_iter=100)
mlpc_1000 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(1000,2), activation="logistic", max_iter=100)

# create random forests
forest_50 = RandomForestClassifier(n_estimators=50, min_samples_leaf=10)
forest_500 = RandomForestClassifier(n_estimators=500, min_samples_leaf=10)
forest_1000 = RandomForestClassifier(n_estimators=1000, min_samples_leaf=10)
```

Figure 22 Data splitting and creating new classifiers

When adjusting hyper-parameters to fine-tune a machine learning model, the model's performance is usually measured against the test set. This can result in a sort of overfitting on the test set. It could be a solution to split the data into three sets and assess the fine-tuned model on some final validation set. However, this may split the dataset into too many parts, so another solution is needed.

K-fold cross-validation splits the dataset into k number of groups. Then, the model is trained on k-1 groups and the remaining group is used as a test set. This training and testing is carried out once for every group in the dataset, so that, by the end of the loop, every group would have been used as training data once.

To see the implementation of the k-fold cross-validation method, please refer to the appropriately labelled item in the appendix.To run this method of cross-validation on the models, KFold is instantiated with 10 splits and its *split* function is iterated over. The *split* function returns training and testing indexes for each split, which can be used to extract the appropriate data from the dataset. Each classifier is then created, trained, and scored. These final score values are put into arrays, with one score array for each classifier. The mean values of each score array are plotted in a abr chart in *figure 23*.

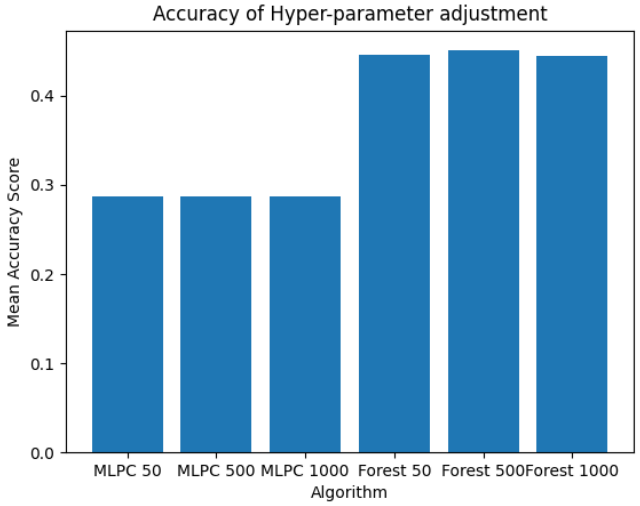


Figure 23 Mean accuracy score of various classifiers

Based on these accuracy scores, there is no difference in iteration count for the Artificial Neural Network, however this is almost certainly due to the faulty training of the classifier. For the Random Forest classifier, the results are extremely close. They show that 500 decision trees is the highest-scoring hyper-parameter, followed by 50 trees and then 1000 trees. However the difference is so marginal that, upon re-running the code, any of the three random forest models could perform better than the others.

If one of these models had to be implemented, the Random Forest model with 500 decision trees would be the best option. This model performs noticeably better than the zero rule, which is a method of estimating outcome by simply guessing the most common label (machine Learning Catalogue, undated). If the zero rule was followed, “patient” would have been estimated every time and the hit rate would be 41.07%. This is significantly better than the multilayer perceptron classifier models. Comparing this to the random forest models also helps give context to their accuracy scores.

The complete python file for the final task can be found in the appendix with the title *Task3.py*.

References

- Mayo, M. (2020) *Centroid Initialization Methods for k-means Clustering.* KDNuggets. Available from https://www.kdnuggets.com/2020/06/centroid-initialization-k-means-clustering.html [accessed 08 February 2022].
- Kumar, V. (2019) *Testing pandas.read_csv performance* [blog]. Available from https://www.kaggle.com/timetraveller98/testing-pandas-read-csv-performance [accessed 09 February 2022].
- Frost, J. (undated) *Standardization.* Available from https://statisticsbyjim.com/glossary/standardization/ [accessed 10 February 2022].
- Dienes, E. (2011) *I Have an Outlier!* CTSpedia. Available from https://www.ctspedia.org/do/view/CTSpedia/OutLier [accessed 10 February 2022].
- Scikit-learn (undated) *sklearn.ensemble.RandomForestClassifier.* Available from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html [accessed 10 February 2022].
- Machine Learning Catalogue (undated) *Zero Rule.* Available from https://machinelearningcatalogue.com/algorithm/alg_zero-rule.html [accessed 13 February 2022].

Appendix

```
plt.clf()
plt.plot(x_train, y_train, 'go')
#plt.plot(x_space, y_predicted0, 'b')
plt.plot(x_space, y_predicted1, 'y')
plt.plot(x_space, y_predicted2, 'r')
plt.plot(x_space, y_predicted3, 'g')
plt.plot(x_space, y_predicted6, 'b')
plt.plot(x_space, y_predicted10, 'm')
plt.legend(['training points', "1 degrees", "2 degrees", "3 degrees", "6 degrees", "10
degrees"])
plt.show()
```

Polynomial graph generation

```
# fill accuracy arrays for training and testing data
MSSEtrain.append(eval_pol_regression(w1, x_train2, y_train2, 1))
MSSEtest.append(eval_pol_regression(w1, x_test2, y_test2, 1))

MSSEtrain.append(eval_pol_regression(w2, x_train2, y_train2, 2))
MSSEtest.append(eval_pol_regression(w2, x_test2, y_test2, 2))

MSSEtrain.append(eval_pol_regression(w3, x_train2, y_train2, 3))
MSSEtest.append(eval_pol_regression(w3, x_test2, y_test2, 3))

MSSEtrain.append(eval_pol_regression(w6, x_train2, y_train2, 6))
MSSEtest.append(eval_pol_regression(w6, x_test2, y_test2, 6))

MSSEtrain.append(eval_pol_regression(w10, x_train2, y_train2, 10))
MSSEtest.append(eval_pol_regression(w10, x_test2, y_test2, 10))

# plot accuracy arrays for each polynomial degree
plt.figure()
plt.semilogy([1,2,3,6,10], MSSEtrain)
plt.semilogy([1,2,3,6,10], MSSEtest)
plt.legend(('MSSE on training set', 'MSSE on test set'))
plt.show()
```

Mean sum of squared errors graph generation

```
import pandas
import numpy
import matplotlib.pyplot as plt

def pol_regression(features_train, y_train, degree):
    X = getPolynomialDataMatrix(features_train, degree) # get prepared array

    # perform least squares algorithm to calculate weights
    XX = X.transpose().dot(X)
    w = numpy.linalg.inv(XX).dot(X.transpose().dot(y_train))

    return w

def getPolynomialDataMatrix(x, degree):
    # prepare data
    X = numpy.ones(x.shape) # create array with same size as x
    for i in range(1, degree + 1): # for every degree, starting with 1
        # add new dimension to array, containing given data raised to the degree
        X = numpy.column_stack((X, x ** i ))
    return X
# read data
data = pandas.read_csv("pol_regression.csv")
x_train = data["x"]
y_train = data["y"]
```

```
X = numpy.column_stack((numpy.ones(x_train.shape), x_train))
# create solution space
x_space = numpy.linspace(-5, 5, 100)

# get weights and use on data
w1 = pol_regression(x_train, y_train, 1)
x_space1 = getPolynomialDataMatrix(x_space, 1)
y_predicted1 = x_space1.dot(w1)

w2 = pol_regression(x_train, y_train, 2)
x_space2 = getPolynomialDataMatrix(x_space, 2)
y_predicted2 = x_space2.dot(w2)

w3 = pol_regression(x_train, y_train, 3)
x_space3 = getPolynomialDataMatrix(x_space, 3)
y_predicted3 = x_space3.dot(w3)

w6 = pol_regression(x_train, y_train, 6)
x_space6 = getPolynomialDataMatrix(x_space, 6)
y_predicted6 = x_space6.dot(w6)

w10 = pol_regression(x_train, y_train, 10)
x_space10 = getPolynomialDataMatrix(x_space, 10)
y_predicted10 = x_space10.dot(w10)

# plot polynominal results
plt.clf()
plt.plot(x_train, y_train, 'go')
plt.plot(x_space, y_predicted1, 'y')
plt.plot(x_space, y_predicted2, 'r')
plt.plot(x_space, y_predicted3, 'g')
plt.plot(x_space, y_predicted6, 'b')
plt.plot(x_space, y_predicted10, 'm')
plt.legend(['training points', "1 degrees", "2 degrees", "3 degrees", "6 degrees", "10
degrees"])
plt.show()

# Split training and testing data
count = len(x_train)
train_split = int(count * 0.7)
x_train2 = x_train[:train_split]
y_train2 = y_train[:train_split]
x_test2 = x_train[train_split:]
y_test2 = y_train[train_split:]

# again, get weights for training data and plly
w1 = pol_regression(x_train2, y_train2, 1)
x_space1 = getPolynomialDataMatrix(x_space, 1)
y_predicted1 = x_space1.dot(w1)

w2 = pol_regression(x_train2, y_train2, 2)
x_space2 = getPolynomialDataMatrix(x_space, 2)
y_predicted2 = x_space2.dot(w2)

w3 = pol_regression(x_train2, y_train2, 3)
x_space3 = getPolynomialDataMatrix(x_space, 3)
y_predicted3 = x_space3.dot(w3)

w6 = pol_regression(x_train2, y_train2, 6)
x_space6 = getPolynomialDataMatrix(x_space, 6)
y_predicted6 = x_space6.dot(w6)

w10 = pol_regression(x_train2, y_train2, 10)
x_space10 = getPolynomialDataMatrix(x_space, 10)
y_predicted10 = x_space10.dot(w10)

# create accuracy arrays
```

```
MSSEtrain = []
MSSEtest = []

def eval_pol_regression(parameters, x, y, degree):
    prepared_x = getPolynomialDataMatrix(x, degree) # get results
    # and measure mean squared sum of errors
    msse = numpy.mean(((prepared_x).dot(parameters) - y) ** 2)
    return msse

# fill accuracy arrays for training and testing data
MSSEtrain.append(eval_pol_regression(w1, x_train2, y_train2, 1))
MSSEtest.append(eval_pol_regression(w1, x_test2, y_test2, 1))

MSSEtrain.append(eval_pol_regression(w2, x_train2, y_train2, 2))
MSSEtest.append(eval_pol_regression(w2, x_test2, y_test2, 2))

MSSEtrain.append(eval_pol_regression(w3, x_train2, y_train2, 3))
MSSEtest.append(eval_pol_regression(w3, x_test2, y_test2, 3))

MSSEtrain.append(eval_pol_regression(w6, x_train2, y_train2, 6))
MSSEtest.append(eval_pol_regression(w6, x_test2, y_test2, 6))

MSSEtrain.append(eval_pol_regression(w10, x_train2, y_train2, 10))
MSSEtest.append(eval_pol_regression(w10, x_test2, y_test2, 10))

# plot accuracy arrays for each polynomial degree
plt.figure()
plt.semilogy([1,2,3,6,10], MSSEtrain)
plt.semilogy([1,2,3,6,10], MSSEtest)
plt.legend(('MSSE on training set', 'MSSE on test set'))
plt.show()
```

Task1.py

```
def show_plots(data, cluster_assignment, k):

    k_num = len(k)
    fig = plt.figure()
    fig1, fig2 = fig.subplots(2)

    ## Plot height vs tail length
    colours = ["c", "m", "y"]
    for index in range(0, len(cluster_assignment)):
        this_colour = colours[cluster_assignment[index]] # get cluster colour
        row = data.iloc[index] # get row via index
        fig1.scatter(row["height"], row["tail"], c=this_colour) # plot this data point

    for cluster in k: # plot each centroid
        fig1.scatter(cluster[0], cluster[1], c="k", marker="X")

    fig1.set_xlabel("Height")
    fig1.set_ylabel("Tail Length")
    fig1.set_title(f"Height vs Tail Length in Dogs (K={k_num})")

    ## Plot Height vs leg length
    for index in range(0, len(cluster_assignment)):
        this_colour = colours[cluster_assignment[index]] # get cluster colour
        row = data.iloc[index] # get row via index
        fig2.scatter(row["height"], row["leg"], c=this_colour) # plot this data point

    for cluster in k: # plot each centroid
        fig2.scatter(cluster[0], cluster[2], c="k", marker="X")
```

```
    fig2.set_xlabel("Height")
    fig2.set_ylabel("Leg Length")
    fig2.set_title(f"Height vs Leg Length in Dogs (K={k_num})")

    ## Show results
    plt.show()
```

Implementation for *show_plots()*

```
from random import random
import numpy
import pandas
import math
import random
import matplotlib.pyplot as plt

def compute_euclidean_distance(vec_1, vec_2):

    distance = float('inf') # create placeholder value
    if (len(vec_1) == len(vec_2)): # if possible to calculate distance
        sum = 0 # initialise running total
        for feature_index in range(0, len(vec_1) - 1): # loop through each feature for both
vectors
            squared = (vec_1[feature_index] - vec_2[feature_index]) ** 2 # get squared
distance
            sum = sum + squared # add to running total

        distance = math.sqrt(squared) # get square root and return result

    return distance

def initialise_centroids(dataset, k):
    centroids = []
    # randomly initialise centroids using range mechanism
    for i in range(0, k):
        height = random.uniform(min(dataset["height"]), max(dataset["height"]))
        tail = random.uniform(min(dataset["tail"]), max(dataset["tail"]))
        leg = random.uniform(min(dataset["leg"]), max(dataset["leg"]))
        nose = random.uniform(min(dataset["nose"]), max(dataset["nose"]))

        centroids.append([height, tail, leg, nose])

    return centroids

def kmeans(dataset, k):

    dataset = dataset.values # remove columns for now

    changed = True  # tracks stability
    old_k = []

    # used for evaluation
    cluster_assignment_history = []
    k_history = [k]

    while(changed): # Assign each data point to a cluster and calculate new centroid
position
        cluster_assignment = [] # This will hold cluster assignments for each record
        # For every record in dataset
        for index in range(len(dataset)):
            distances = [] # create intermediary place to store distances
```

```
            for centroid in k:
                # calculate distance from each centroid
                distances.append(compute_euclidean_distance(dataset[index], centroid))

            # get id of shortest distance in array (which is equal to the respective
centroid id)
            closest_centroid_index = numpy.argmin(distances)
            cluster_assignment.append(closest_centroid_index) # save centroid id

        # re-center centroids
        numpy_results = numpy.array(cluster_assignment)
        # Use cluster assignment to group values and produce averages. Assign averages to
centroids
        k = pandas.DataFrame(data.values).groupby(numpy_results).mean().values
        k_history.append(k)

        if numpy.array_equiv(k, old_k): # If stability reached
            changed = False  # stop looping next loop

        cluster_assignment_history.append(cluster_assignment) # save cluster info for eval
        old_k = k # save current centroid values to see if they change

    # Evaluate K-means
    eval_kmeans(dataset, cluster_assignment_history, k_history)

    return k, cluster_assignment

def eval_kmeans(dataset, cluster_assignment_history, k_history):
    # Calculate the sum of squared distances from each data point to centroid
    iteration_values = []
    error_sum_values = []
    for iteration in range(len(cluster_assignment_history)):
        error_sum = 0 # create running total
        for index in range(len(cluster_assignment_history[iteration])): # for every
iteration of kmeans
            centroid_index = cluster_assignment_history[iteration][index] # get centroid
assignment of record
            row = dataset[index] # and get the record features
            error = compute_euclidean_distance(row, k_history[iteration][centroid_index]) #
calculate absolute difference
            error_sum = error_sum + error # add to running total
        iteration_values.append(iteration + 1) # arrays start at zero, but our iteration
axis shouldn't, so add 1
        error_sum_values.append(error_sum) # These two arrays will be plotted

    # Plot data
    k = len(k_history[0])
    fig = plt.figure()
    fig3 = fig.add_subplot()
    fig3.set_xlabel("Iteration no.")
    fig3.set_ylabel("Sum of errors")
    fig3.set_title(f"Error intensity per iteration (K={k})")
    plt.plot(iteration_values, error_sum_values)

def show_plots(data, cluster_assignment, k):

    k_num = len(k)
    fig = plt.figure()
    fig1, fig2 = fig.subplots(2)

    ## Plot height vs tail length
    colours = ["c", "m", "y"]
    for index in range(0, len(cluster_assignment)):
        this_colour = colours[cluster_assignment[index]] # get cluster colour
        row = data.iloc[index] # get row via index
        fig1.scatter(row["height"], row["tail"], c=this_colour) # plot this data point
```

```
    for cluster in k: # plot each centroid
        fig1.scatter(cluster[0], cluster[1], c="k", marker="X")

    fig1.set_xlabel("Height")
    fig1.set_ylabel("Tail Length")
    fig1.set_title(f"Height vs Tail Length in Dogs (K={k_num})")

    ## Plot Height vs leg length
    for index in range(0, len(cluster_assignment)):
        this_colour = colours[cluster_assignment[index]] # get cluster colour
        row = data.iloc[index] # get row via index
        fig2.scatter(row["height"], row["leg"], c=this_colour) # plot this data point

    for cluster in k: # plot each centroid
        fig2.scatter(cluster[0], cluster[2], c="k", marker="X")

    fig2.set_xlabel("Height")
    fig2.set_ylabel("Leg Length")
    fig2.set_title(f"Height vs Leg Length in Dogs (K={k_num})")

    ## Show results
    plt.show()

column_names = ["height", "tail", "leg", "nose"]
data = pandas.read_csv("Task2 - dataset - dog_breeds.csv", names=column_names, header=0)

k_num = 2
k = initialise_centroids(data, k_num)
k, cluster_assignment = kmeans(data, k)
show_plots(data, cluster_assignment, k)

k_num = 3
k = initialise_centroids(data, k_num)
k, cluster_assignment = kmeans(data, k)
show_plots(data, cluster_assignment, k)
```

Task2.py

```
def eval_kmeans(dataset, cluster_assignment_history, k_history):
    # Calculate the sum of squared distances from each data point to centroid
    iteration_values = []
    error_sum_values = []
    for iteration in range(len(cluster_assignment_history)):
        error_sum = 0
        for index in range(len(cluster_assignment_history[iteration])):
            centroid_index = cluster_assignment_history[iteration][index]
            row = dataset[index]
            error = compute_euclidean_distance(row, k_history[iteration][centroid_index])
            error_sum = error_sum + error
        iteration_values.append(iteration + 1)
        error_sum_values.append(error_sum)

    # Plot data
    k = len(k_history[0])
    fig = plt.figure()
    fig3 = fig.add_subplot()
    fig3.set_xlabel("Iteration no.")
    fig3.set_ylabel("Sum of errors")
    fig3.set_title(f"Error intensity per iteration (K={k})")
    plt.plot(iteration_values, error_sum_values)
```

Implementation for *eval_kmeans()*

```
# Create KFold object with 10 splits/iterations
k = 10
kf = KFold(n_splits=k, random_state=30)

# create score arrays
mlpc_50_accuracy = []
mlpc_500_accuracy = []
mlpc_1000_accuracy = []
forest_50_accuracy = []
forest_500_accuracy = []
forest_1000_accuracy = []

for train_i, test_i in kf.split(data):
    # split data according to this split's indexes
    x_train = data.iloc[train_i, : ]
    x_test = data.iloc[test_i, : ]
    y_train = data.iloc[train_i, : ]
    y_test = data.iloc[test_i, : ]

    # create neural networks
    mlpc_50 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(50,2),
activation="logistic", max_iter=100)
    mlpc_500 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(500,2),
activation="logistic", max_iter=100)
    mlpc_1000 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(1000,2),
activation="logistic", max_iter=100)
    # create random forests
    forest_50 = RandomForestClassifier(n_estimators=50, min_samples_leaf=10)
    forest_500 = RandomForestClassifier(n_estimators=500, min_samples_leaf=10)
    forest_1000 = RandomForestClassifier(n_estimators=1000, min_samples_leaf=10)
    # fit (train) classifiers with this split's training data
    mlpc_50.fit(x_train, y_train)
    mlpc_500.fit(x_train, y_train)
    mlpc_1000.fit(x_train, y_train)
    forest_50.fit(x_train, y_train)
    forest_500.fit(x_train, y_train)
    forest_1000.fit(x_train, y_train)

    # test classifiers with this split's testing data
    mlpc_50_pred = mlpc_50.predict(x_test)
    mlpc_500_pred = mlpc_500.predict(x_test)
    mlpc_1000_pred = mlpc_1000.predict(x_test)
    forest_50_pred = forest_50.predict(x_test)
    forest_500_pred = forest_500.predict(x_test)
    forest_1000_pred = forest_1000.predict(x_test)

    # append accuracy score classifier-specific arrays
    mlpc_50_accuracy.append(accuracy_score(mlpc_50_pred, y_test))
    mlpc_500_accuracy.append(accuracy_score(mlpc_500_pred, y_test))
    mlpc_1000_accuracy.append(accuracy_score(mlpc_1000_pred, y_test))
    forest_50_accuracy.append(accuracy_score(forest_50_pred, y_test))
    forest_500_accuracy.append(accuracy_score(forest_500_pred, y_test))
    forest_1000_accuracy.append(accuracy_score(forest_1000_pred, y_test))
```

Implementation of the k-fold cross-validation method

```
import pandas
import matplotlib.pyplot as plt
import numpy

dtypes = {"Image number": int, "Bifurcation number": int, "Artery (1)/ Vein (2)": int,
"Alpha": float, "Beta": float, "Lambda": float, "Lambda1": float, "Lambda2": float,
"Participant Condition": str}
data = pandas.read_csv("Task3 - dataset - HIV RVG.csv", dtype=dtypes) # read data with
custom dtypes
# create column shorthand column names for easier handling
data.columns = ["image", "bifurcation", "arteryvein", "alpha", "beta", "lambda", "lambda1",
"lambda2", "status"]

# get summary statistics
data_summary = data.agg(["mean", "std", "min", "max"])
print(data_summary)

# show boxplots of raw data
alpha_boxplot = data.boxplot(column="alpha", by="status")
beta_boxplot = data.boxplot(column="beta", by="status")
plt.show()

def clean_normalise(data, column):
    std = 3 # remove row not within x standard deviations
    # calculate z-score, relative to population mean, and remove all rows that fall outside
threshold
    data = data[((data[column] - data[column].mean()) / data[column].std()).abs() < std]

    # use the maximum value to normalise the data[column] data to values between 0 and 1
    data[column] = (data[column] - data[column].min(axis=0)) / (data[column].max(axis=0) -
data[column].min(axis=0))

    return data

data = clean_normalise(data, "alpha")
data = clean_normalise(data, "beta")
data = clean_normalise(data, "lambda")
data = clean_normalise(data, "lambda1")
data = clean_normalise(data, "lambda2")

# show boxplots after outlier removal and standardisation
alpha_boxplot = data.boxplot(column="alpha", by="status")
beta_boxplot = data.boxplot(column="beta", by="status")
plt.show()

data_summary = data.agg(["mean", "std", "min", "max"])
print(data_summary)

## Section 2

from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

# prepare the labels into numerical form for the classifiers
data.loc[(data["status"] == "Control"), "status"] = 0
data.loc[(data["status"] == "Patient"), "status"] = 1
data["status"] = data["status"].astype(int)

#create a dataframe with all rows for our input data that contain a NaN value
mask = numpy.all(numpy.isnan(data), axis=1)
# and remove all rows from our main dataset that exist in that NaN dataframe
data_nonnan = data[~mask] # leaving only values that contain no NaN values
```

```
features = data_nonnan.drop(["image", "bifurcation", "arteryvein", "status"], 1) # remove
label column

x_vector = features.values # prepare input vector
y_vector = data_nonnan["status"].values # prepare output vector

def mlpc_train_epochs(x_train, x_test, y_train, y_test, epochs):
    scores = []
    for iterations in epochs:
        # create ANN classifier
        clf = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(500,2),
activation="logistic", max_iter=iterations)
        clf.fit(x_train, y_train) # "train" or fit classifier into model
        scores.append(clf.score(x_test, y_test)) # save score
    return scores

def forest_train(x_train, x_test, y_train, y_test, min_samples):
    scores = []
    for min_sample in min_samples:
        # create random forest classifier
        clf = RandomForestClassifier(n_estimators=1000, min_samples_leaf=min_sample)
        clf.fit(x_train, y_train) # "train" or fit classifier into model
        scores.append(clf.score(x_test, y_test)) # save score
    return scores

# split dataset into training and testing data, 9:1 ratio, with a replicable split
x_train, x_test, y_train, y_test = train_test_split(x_vector, y_vector, test_size=0.1,
random_state=30)
epochs = [5, 10, 15, 20, 25, 50, 100] # define epoch counts
mlpc_scores = mlpc_train_epochs(x_train, x_test, y_train, y_test, epochs) # train

min_samples = [5, 10]
forest_scores = forest_train(x_train, x_test, y_train, y_test, min_samples)

fig = plt.figure()
fig1, fig2 = fig.subplots(2)
fig1.set_xlabel("No. of Epochs")
fig1.set_ylabel("Score")
fig1.set_title("Mean accuracy of Epochs")
fig1.plot(epochs, mlpc_scores)

fig2.set_xlabel("No. of samples")
fig2.set_ylabel("Score")
fig2.set_title("Mean accuracy of min leaf samples")
fig2.plot(min_samples, forest_scores)
plt.show()

## Section 3

# Create KFold object with 10 splits/iterations
k = 10
kf = KFold(n_splits=k)

# create score arrays
mlpc_50_accuracy = []
mlpc_500_accuracy = []
mlpc_1000_accuracy = []
forest_50_accuracy = []
forest_500_accuracy = []
forest_1000_accuracy = []

for train_i, test_i in kf.split(data):
    # split data according to this split's indexes
    x_train = x_vector[train_i]
    x_test = x_vector[test_i]
    y_train = y_vector[train_i]
    y_test = y_vector[test_i]
```

```
    with numpy.printoptions(threshold=numpy.inf):
        print(y_test)

    # create neural networks
    mlpc_50 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(50,2),
activation="logistic", max_iter=100)
    mlpc_500 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(500,2),
activation="logistic", max_iter=100)
    mlpc_1000 = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(1000,2),
activation="logistic", max_iter=100)

    # create random forests
    forest_50 = RandomForestClassifier(n_estimators=50, min_samples_leaf=10)
    forest_500 = RandomForestClassifier(n_estimators=500, min_samples_leaf=10)
    forest_1000 = RandomForestClassifier(n_estimators=1000, min_samples_leaf=10)

    # fit (train) classifiers with this split's training data
    mlpc_50.fit(x_train, y_train)
    mlpc_500.fit(x_train, y_train)
    mlpc_1000.fit(x_train, y_train)
    forest_50.fit(x_train, y_train)
    forest_500.fit(x_train, y_train)
    forest_1000.fit(x_train, y_train)

    # test classifiers with this split's testing data
    mlpc_50_pred = mlpc_50.predict(x_test)
    mlpc_500_pred = mlpc_500.predict(x_test)
    mlpc_1000_pred = mlpc_1000.predict(x_test)
    forest_50_pred = forest_50.predict(x_test)
    forest_500_pred = forest_500.predict(x_test)
    forest_1000_pred = forest_1000.predict(x_test)

    # append accuracy score classifier-specific arrays
    mlpc_50_accuracy.append(accuracy_score(mlpc_50_pred, y_test))
    mlpc_500_accuracy.append(accuracy_score(mlpc_500_pred, y_test))
    mlpc_1000_accuracy.append(accuracy_score(mlpc_1000_pred, y_test))
    forest_50_accuracy.append(accuracy_score(forest_50_pred, y_test))
    forest_500_accuracy.append(accuracy_score(forest_500_pred, y_test))
    forest_1000_accuracy.append(accuracy_score(forest_1000_pred, y_test))

average_scores = []
average_scores.append(sum(mlpc_50_accuracy) / len(mlpc_50_accuracy))
average_scores.append(sum(mlpc_500_accuracy) / len(mlpc_500_accuracy))
average_scores.append(sum(mlpc_1000_accuracy) / len(mlpc_1000_accuracy))
average_scores.append(sum(forest_50_accuracy) / len(forest_50_accuracy))
average_scores.append(sum(forest_500_accuracy) / len(forest_500_accuracy))
average_scores.append(sum(forest_1000_accuracy) / len(forest_1000_accuracy))

plt.bar(["MLPC 50" , "MLPC 500" , "MLPC 1000" , "Forest 50" , "Forest 500" , "Forest 1000"
], average_scores)
plt.xlabel("Algorithm")
plt.ylabel("Mean Accuracy Score")
plt.title("Accuracy of Hyper-parameter adjustment")
plt.show()
```

Task3.py