ALGORITHMS FOR DATA MINING

CP3744M

Brandon Slater 16604148

Table of Contents

Task 1:

Section 1.1: Description of Ridge Regression

1.1.1: Linear Regression Models

1.1.1.1: Simple Linear Regression:

$$Y = \beta_0 + \beta_1 X,$$

The equation to the right represents the linear regression model; where b0 is called the intercept and b1 is the called the slope, which both relate in the cartesian plan to the Y and X axis. This regression type establishes a relationship between X and Y and can be used as a predictor to estimate X or Y future values. For example, the correlation coefficient would be used to establish a relative prediction of Y when the X is the known value. (Johnson, 1971). The slope and the intercept play a crucial role in positioning the regression through the intercept and association of X and Y through quantification. This association can be seen through the absolution value of the slope: negative provides Y decreasing by b1, neutral presents no association and Position shows increases in both the X and Y axis, where Y increases by b1. (Bangiwala, 2018)

1.1.1.2: Multiple Linear Regression:

$$\hat{Y} = b_0 + b_1 X_1 + b_2 X_2$$

Multiple linear regression like simple but using values of dependent variables and set of explanatory variables to predict the Y label. The equation to the right represents the relationship between dependent variables and the explanatory variables. (Mark Tranmer, 2008). Multiple Linear Regression is useful for taking in many characteristics which determine a singular out come to produce a linear regression line for predictions.

1.1.2: Features in Linear Regression:

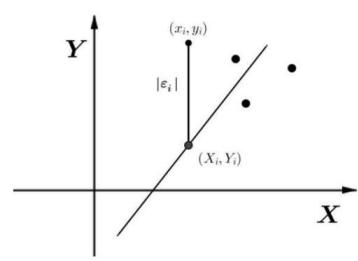
features1	features2	features3	features4	features5	features6	features7	features8	features9	features1(	features11
-1.71482	3.288177	-2.56807	3.604938	-3.15697	3.966326	-3.6162	4.291073	-3.99329	4.576418	-4.31202
-1.68017	3.156664	-2.41555	3.322339	-2.85071	3.509196	-3.13478	3.64467	-3.32323	3.731566	-3.44495
-1.64553	3.027834	-2.2692	3.056691	-2.56869	3.096841	-2.70939	3.085128	-2.75503	3.029771	-2.73939
-1.61089	2.901688	-2.12887	2.8073	-2.30945	2.72568	-2.33446	2.602244	-2.27489	2.449082	-2.16774
-1.57625	2.778226	-1.99446	2.573491	-2.07158	2.392355	-2.00491	2.186832	-1.87062	1.970551	-1.70667

In both linear and multiple regression features are the dependent variables used for predicting the explanatory variables. In simple linear regression we only have a singular value to determine the relationship between the Y intercept and X slope. Whereas, multiple linear regression has multiple dependent variables used in trying to predict the Y Label. These multiple features have an individual weighting of contributing to predicting the explanatory variable.

1.1.3: Least Squares Solution

Equation: "y = mx + b"

This method is using the sum of the squared differences between the explanatory values and the predicted values to minimize the error of estimating the unknown parameters of the linear regression model. For example, this function takes the actual value and minuses predicted value squares and then sums all these errors and when the lowest is found the line of best fit will be places through the graph allowing for estimation.



(a)

Figure 1. Classical Least Squares Regression (a),

1.1.4: MSE (Mean Squared Error) and SSE (Sum Squared Error)

$$\mathrm{MSE} = \frac{1}{N}\sum_{t=1}^{N}\left(q_s^t - q_o^t\right)^2,$$

Both the MSE and SSE are methods in calculating the estimation of variance error within the whole dataset of explanatory and prediction data points. SSE also takes into consideration both the sum of squared from the factors and from the randomness/ error. (Bruce, 2018) The SSE is an approach to expressing the total variation for the Y's, for example; within regression the variation is calculated through sum of squares and is attributed to the tie between X's and Y's. SSE is important in regression models for explaining its proportion of the total variation; when expressed in large terms, the better the relationship is at explain Y of X. (Bruce, 2018)

1.1.5: Ridge Regression

Equation: "W = ((X'X + (LAMBDA * I))-1 * X' * y"

This formula is often used/found when dealing with muilticollinearity in data where the OLS estimation performs badly. In addition to the OLS solution the use of a weighted penalty which acts as a tunning hyperparrameter. This parameter when equal to 0 provides no penalty term which has no effect and the OLS will be produced.

1.1.5.1: Comparison of Overfitting and Underfitting

In contrast to ridge regression, the task is to fit the model to the training data to allow reliable predictions to be made on untrained/testing data. Overfitting is a low bias/high variance prediction of data where the model has too much reliance on training data. The high variance represents that the model significantly changed base on the training data (low bias) making untrained data hard to predict accurately. Underfitting is the opposite; the relationship between all the features cannot be captured accurately. This in turn leads to poor generalization on the test data where estimates may have high inconsistencies.

However, the goal of a model such as ridge regression is to find the sweat spot between overfitting and underfitting. This can be challenging and difficult in practice but implementations which consider the highest error dependent on a bias factor (lambda).

1.1.5.2: Objective Function of Ridge Regression

$$Q(\boldsymbol{\beta}|\mathbf{X},\mathbf{y}) = \frac{1}{2\sigma^2}\sum_i (y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \frac{1}{2\tau^2}\sum_{j=1}^{p}\beta_j^2$$

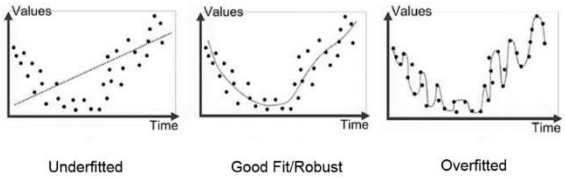
Sum of squared error within weights with addition of the lambda regularization tries to minimize all non-influential features whilst maintaining influential ones.

1.1.5.3: Weight Penalty Term for Ridge Regression

$$\boldsymbol{\beta}_{\text{ridge}} = \min_{\boldsymbol{\beta}} \underbrace{\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2}_{\text{RSS}} + \underbrace{\lambda\sum_{j=1}^{p}\beta_j^2}_{\text{shrinkage penalty}}$$

The weight penalty term also known as Lambda alongside a least squares solution produces ridge regression. This penalty is important for controlling the bias-variance trade off to avoid overfitting. A large penalty applied to the training data makes predictions that don't fit well to the data; whereas a small penalty fits the training data better. In comparison to testing data, large penalty sees high biases within the data set which in turn creates overfitting; whereas small penalty creates high variance producing overfitting. With this being said, using cross validation for the penalty term helps to find the optimal fit for outputting estimates consistently for testing.

1.1.5.4: Comparison of advantages and disadvantages for Ridge Regression



The main advantages of using a ridge regression is the ability to include multicollinearity which allows multiple values to be used in estimating a prediction Y. However this can introduce a higher standard error of the coefficients which has the potential to overinflate errors and make some features "statistically insignificant when they should be significant". (Daoud, 2017)

Ridge regression unlike linear regression penalizes the estimates; estimates which are large will be increased whereas SSE will minimize and vice versa for small estimates. This normalization process takes into consideration the estimates in such a way that less influential features are applied more penalization. This method stops training data being such a big influence on estimating predictions and focuses on getting accurate predictions on all data. Furthermore, having multiple features can provide difficult for a linear regression to compute especially when features are correlated with each other. This is solved through the penalty function mentioned before. On the other hand, a drawback of the ridge regression method is the penalization factor which needs to be chosen through testing or using a ridge test.

Section 1.2: Implementation of Ridge Regression

1.2.1: Function handling the implementation of ridge regression

This function is used as a ridge regression controller, which reads and intialises all variables required for implementing the ridge regression training model.

These files are read in:
regression_train_assignment2019.csv
regression_plotting_assignment2019.csv
The two files are pre-processed for the RR, which extracts the training Y label and features for both data sets.

The ridge regression function is then stored for each of the regularisation factor for training_features, using the Y label and the designated regularisation factor passed. This methods functionality can be seen below under the 'Function for Ridge Regression' heading and these are stored to an array.

```python
# solution for section 1.2
def implementation_ridge_regression(regularisation_factors):
    # read in files training & plotting
    data_train = pd.read_csv('regression_train_assignment2019.csv')
    data_plot = pd.read_csv('regression_plotting_assignment2019.csv')

    # copies the training data x and y values to a separate entity
    x_train = data_train['x']
    y_train = data_train['y']

    # convert y_train to np array
    y = np.array(y_train)

    # copy the data_plot file x values to a separate entity
    x_plot = data_plot['x']

    # drop the x and y columns from the training data
    # take all the training data values minus headings and place them in a np array
    data_train.drop(['x', 'y'], axis=1, inplace=True)
    data_train = data_train.values
    data_train = data_train.transpose()
    features = np.array(data_train)

    # drop the x and y columns from the testing (plotting) data
    # take all the training data values minus headings and place them in a np array
    data_plot.drop(['x'], axis=1, inplace=True)
    data_plot = data_plot.values
    data_plot = data_plot.transpose()
    features_plot = np.array(data_plot)

    # delete the first row of the features (training) and the features_plot (testing)
    features = np.delete(features, (0), axis=0)
    features_train = features.transpose()
    features_plot = np.delete(features_plot, (0), axis=0)
    features_plot = features_plot.transpose()

    # calculates the 12 weighting for each feature with the regularisation
    # output 4x12
    rr_weights = []
    for reg_i in range(4):
        rr_weights.append(ridge_regression(features_train, y, regularisation_factors[reg_i]))

    # calculates the features_trains y values using the weights for each 4 regularisation factors
    rr_prediction__y = []
    for reg_i in range(4):
        rr_prediction__y.append(rr_squared_error(rr_weights[reg_i], features_train, y))

    # calculates the features_plot y values using the weights for each 4 regularisation factors
    rr_prediction_plotting_y = []
    for reg_i in range(4):
        rr_prediction_plotting_y.append(prediction_y_rr_reg(rr_weights[reg_i], features_plot))

    # paces over the x axis for train and plot along with the predicted values for each regularisation factor
    plotting_predicted_y(x_train, x_plot, rr_prediction__y, rr_prediction_plotting_y)
```

The ridge regression square error is then calculated for each of the regularization factors using the weights created for each in the ridge regression function. This methods functionality can be seen below under the 'Function for Calculating the Ridge Regression Squared Error for Training Features' heading and these are stored to an array.

Finally a prediction function calculates the Y label for each of the training sets weights using the 12 features. This methods functionality can be seen below under the 'Function for Calculating the Ridge Regression Squared Error for Training Features' heading and these are stored to an array and these are stored to an array.

The plotting prediction function displays the training and testing for training and plotting data on the x asis of -1000 to 1000 and plots the Y label. This is done 4 times to display each regularisations effect on the training and plotting predictions. More information can be seen below under the heading '1.2.1: Graphs for Regularization factors [0.1, 0.01, 0.0001, 0.000001]', along with a detailed comparison of the data.

1.2.2: Function for Ridge Regression:

The ridge regression function used the following equation: **W = ((X'X + (LAMBDA * I))-1 * X' * y**

The output of this function returns 12 weights for each of the features, which represents how much they effect the Y label. LAMBDA represents the regularisation factor used and the identity matric resolves around a ridge.

```python
# returns 12 feature weights np array based on a penalty regression factor
def ridge_regression(features_train, y_train, regularisation_factor):

    identity_ = np.identity(features_train.shape[1])
    parameters = linalg.solve((features_train.transpose().dot(features_train) +
                               (regularisation_factor * identity_)),
                              features_train.transpose().dot(y_train))

    return np.array(parameters)
```

1.2.3: Function for Calculating the Ridge Regression Squared Error for Training Features

This functions take the weights for each of the regularization factors, the features and the Y labels. The objective of this function is to compare the error between the predicted Y and the actual Y labels for training. Prediction RR stores the ridge regression predictions by adding up all the weights multiplied by its feature, which is applied to all rows of features to create 12 Y labels. This function also displays the error of the ridge regression squared error for each regularization factor along with the RSS (Residual Sum of Squares).

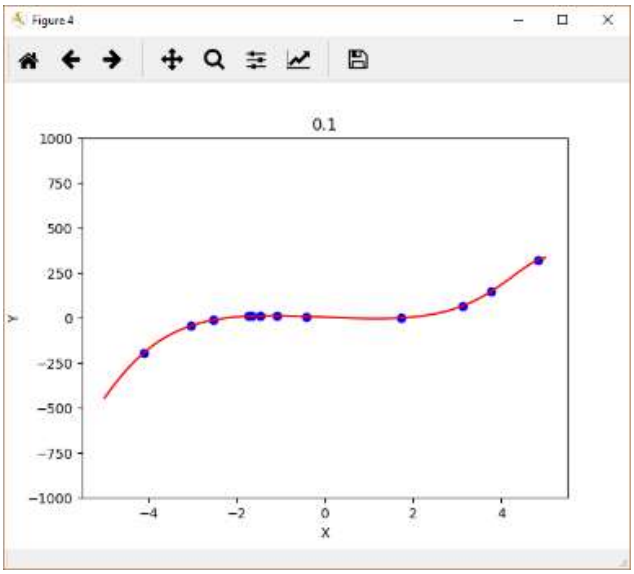
```python
# calculates the error between the predicted y and actual y
# returns 12 predicted y for each 12 features in training
def rr_squared_error(weights, x, y_train):

    prediction_rr = []
    error_rr = 0

    for ys in range(12):
        y_rr = 0  # rr_weights0[0]

        for i in range(12):
            y_rr = y_rr + (weights[i] * x[ys][i])

        prediction_rr.append(y_rr)

        error_rr = error_rr + ((y_train[ys] - y_rr) ** 2)

        print("                    Actual Y: " + str(y_train[ys]))
        print(" Predicted RR; 0.000001 Y: " + str(y_rr))
        print("                    Error RR: " + str(abs((y_train[ys] - y_rr))))
        print("_____________________________________________________________________")

    print("RR, Residual Sum of Squares (RSS): " + str(error_rr))

    return prediction_rr
```

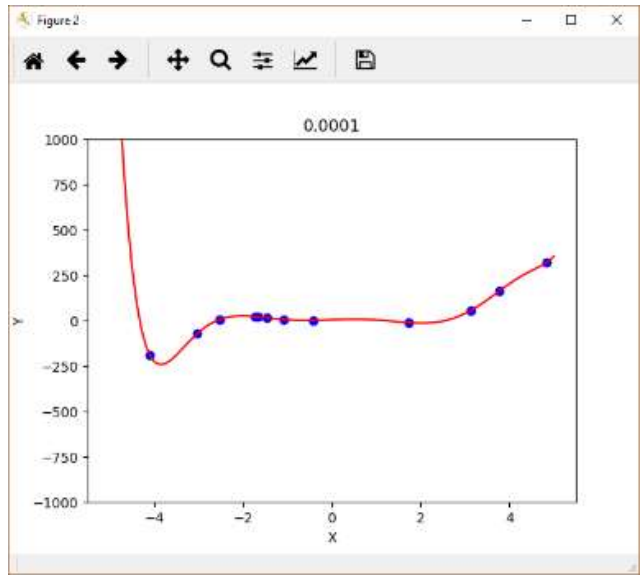
1.2.4: Function for predicting the y labels for the 100 sets of 12 features

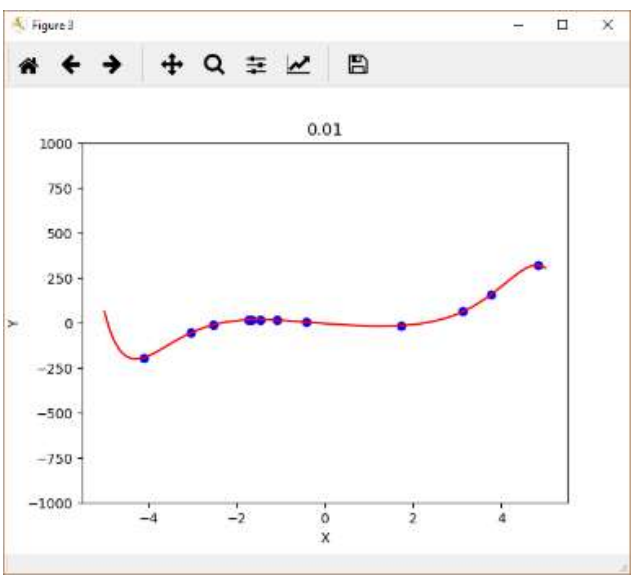
This function similar to the last one takes all the plotting data 100 * 12 pieces and calculates the predicted y values based on the weight created for each of the regularization factors.

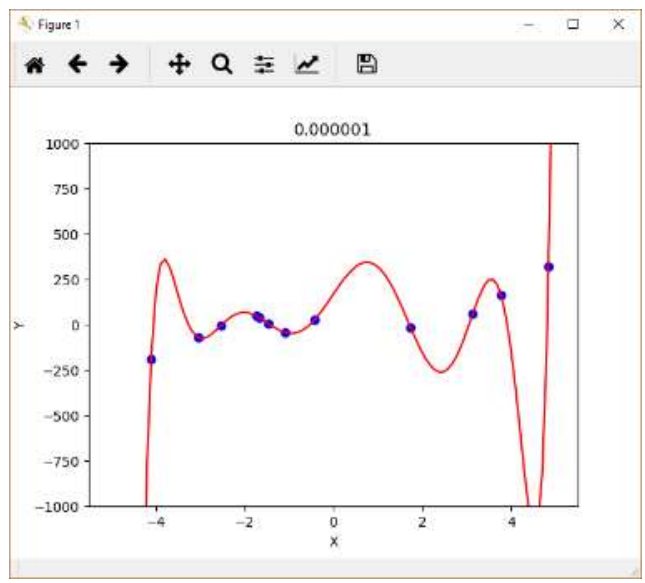
```python
# returns 100 predicted y values for each 12 features in testing data
# using the weights the y values are calculated
def prediction_y_rr_reg(weights, plotting_x):
    # stores all the values for the predicted y
    prediction_y_plotting = []

    # loops through all 100 values
    for ii in range(100):
        y_plot = 0  # rr_weights0[0]

        # loops through all 12 features for predicting y
        for i in range(12):

            # calculates the predicted y using all the weights and the features
            y_plot = y_plot + (weights[i] * plotting_x[ii][i])
        # appends all the values to an array
        prediction_y_plotting.append(y_plot)

    # returns the array of the predicted y values
    return prediction_y_plotting
```

1.2.6: Graphs for Regularization factors [0.1, 0.01, 0.0001, 0.000001]:









The red lines present the X plotting data for the Y predictions and the blue dots show the X training data for the Y predictions. The above represents the 4 regularization factors predictions for the training and plotting Y labels. The regularization factor 0.1 produces the best fit graph out of the four factors tested. The 0.000001 factor produces predictions which have over fit the data and have high variance, which has shown to produce prediction which aren't following the training data. 0.0001 factor produces less variance but over fits the data still, 0.01 factor produces a slightly better fitting, but 0.1 factor (the smallest regularization factor) produces the best fit for predicting values using ridge regression. The smallest regularization graphs (0.01, 0.1) both show levels of over fitting but the 0.01 factor computes a larger over fit where the linear seemed to be linear when passing through the training. Whereas, 0.1 regularization factor has more of a visual flow to the data where the blue data points are seen to be on a similar track to the training predictions.

Section 1.3: Evaluation

1.3.1: Function used for Ridge Regression Evaluation

This function is use to carry out '1.3 evaluation'. This is used on the training data set by splitting it into 8 pieces of data for training and 4 pieces of data for testing, which evaluation the trainings RMSE preformation using the 8 regularization factors. This experiment is then computed 10 times to get an average whilst randomizing the order of the data points.

```python
# solution for section 1.3
def evaluation(regularisation_factors):

    # holds the RSME values for training and testing 70% 30%
    rsme_train_single, rsme_test_single = single_reg_factor_testing(regularisation_factors, False)

    # lists to store 10x randomisation of the training data set
    rsme_train_multiple = []
    rsme_test_multiple = []

    # runs through 10 factor testing with random data set 70% 30%
    for loop in range(10):

        # holds the RSME values for training and testing 70% 30%
        rsme_train_s, rsme_test_s = single_reg_factor_testing(regularisation_factors, True)

        # adds each testing to a list which stores 10
        rsme_train_multiple.append(rsme_train_s)
        rsme_test_multiple.append(rsme_test_s)

    # used for storing all 10x train and test rsme
    avg_reg_plot_train = []
    avg_reg_plot_test = []

    # iterates through all 8 regularisation factors
    for it1 in range(8):
        train = 0
        test = 0

        # iterates through the all the iterates and creates and avg rsme
        for it2 in range(10):
            train = train + rsme_train_multiple[it2][it1]
            test = test + rsme_test_multiple[it2][it1]

        # creates an avg for each reg factors from all the 10 tests for train and tests
        # appends these to a list for storages of all 8 reg factors
        avg_reg_plot_train.append(train / 10)
        avg_reg_plot_test.append(test / 10)

    # sends single train and test figures
    # sends the x10 avg for train and test figures
    plotting_1x_10x_reg_factors(rsme_train_single, rsme_test_single, avg_reg_plot_train, avg_reg_plot_test, regularisation_factors)
```

1.3.2: Function used for a single and random computation for one set of training and testing data

This function calculates the RSME for each regularization in training (70%) and testing (30%) of the training data set. Firstly it does pre-processing of the dataset and then creates all the weights for each regularization factor, and finally uses the eval_regression function for evaluation which factor is optimal. More information about this can be found below.

```python
# train and test RSME calculation
# returns 2 by 8 list of train and test for each 8 regularisation factors
def single_reg_factor_testing(reg_factors, random):

    # reads in the training file
    data_train = pd.read_csv('regression_train_assignment2019.csv')

    # checks if doing a random sample
    if random:

        # randomises the data to select testing and training data
        data_train = data_train.sample(frac=1).reset_index(drop=True)

    # Preprocessing Data
    # takes the y column, convert to np.array, and drope the x and y columns
    # deletes the first row to remove index
    y_plot = data_train['y']
    y_plot = np.array(y_plot)
    data_train.drop(['x', 'y'], axis=1, inplace=True)
    train = data_train.values
    train = train.transpose()
    train = np.array(train)
    train = np.delete(train, 0, axis=0)
    train = train.transpose()

    # splits the data in to 8 train and 4 test
    split = 8
    training, test = train[:split, :], train[split:, :]

    # creates an identity matrix to the size of training first column
    identity_ = np.identity(training.shape[1])

    # stores the weights for all regularisations
    # output 4x12
    holder_rr_weights = []
    for reg in range(8):
        holder_rr_weights.append(
            linalg.solve((training.transpose().dot(training) + reg_factors[reg] * identity_),
                         training.transpose().dot(y_plot[0:8])))

    # stores all the training and testing for RSME values for each reg factors
    # outputs to lists of RSME for each regularisation
    holder_rsme_train = []
    holder_rsme_test = []

    # loops through all the regularisation factors for both the train and test
    for test_RSME in range(8):
        # stores the training plots for all the regularisation and weights for 8 sets of the 12 features for TRAINING
        holder_rsme_train.append(eval_regression(holder_rr_weights[test_RSME], training, y_plot[0:8]))
        # stores the testing plots for all the regularisation and weights for 4 sets of the 12 features for TESTING
        holder_rsme_test.append(eval_regression(holder_rr_weights[test_RSME], test, y_plot[8:12]))

    # returns the training and testing ROOT SQUARED MEAN ERROR
    return holder_rsme_train, holder_rsme_test
```

1.3.3: Function used for evaluation of regression, produces the ROOT MEAN SQUARE ERROR

The root mean square error is created from weights, features and the Ylabel from training data. Firstly calculates the difference between the predicted value and the actual label, which is then squared to remove any negatives, summed, then divided by the length of the Y values and squared. This is used to evaluate a models estimator abilities for predictions.

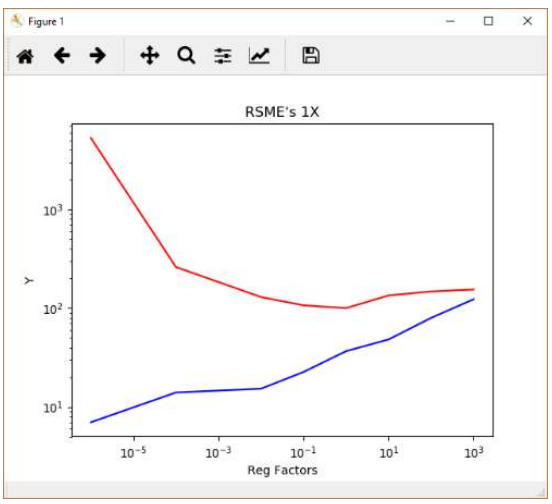
```python
# produces the RMSE (root mean squared error)
def eval_regression(parameters, features, y):
    # produces a 1 d array
    predy = features.dot(parameters)
    # calculates the different between actual and prediction
    difference = y - predy
    # squares the error (creates a positive)
    difference = difference**2
    # sums the differences into var
    totalsum = sum(difference)
    # divides by the length of y -1
    rmse2 = totalsum / (len(y)-1)
    # square root the answer and return single variable
    rmse = np.sqrt(rmse2)

    return rmse
```

1.3.4: RMSE (Root Mean Squared Error) Evaluation for Regularization Factors

$$\mathrm{RMSE}_{fo} = [\sum_{i=1}^{N} (z_{f_i} - z_{o_i})^2 / N]^{1/2}$$

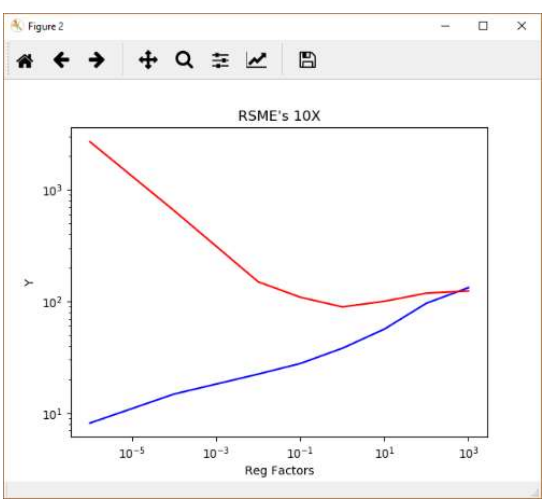
The RMSE is the standard deviation of the vertical distance between the data point and the ridge regression like. The function above describes the equation for calculating the RSME, which is described as squaring the residuals, applying an average and taking the square root of that result.
The first graph represents a single iteration of RSME for 8 regularization factors and the second shows an average of 10 iterations of RSME. The blue line represents the training data at 70% of the training data and the red line shows the test data at 30% of the training data. The plots describe the relationship between the RMSE for each regularization factor using in ridge regression.

1.3.4.1: Graphs for a single ROOT SQUARE MEAN ERROR



The single ridge regression on each regularization factor can been seen that the two lines converge towards each other. The minimum distance between both sets is regularization factor 10^0 (1) as this is the point on both lines where both the training is increase and test is decreasing. The reason that the training error increases is due to the regularization (L2) being increased. The point of regularization is to increase the bias whilst reducing the variance of the model.

1.3.4.2: Graph for 10x random ordered ROOT SQUARE MEAN ERROR Averages



The average of 10 applications on random data for ridge regression using the regularization factors can be seen to the left. This graph whilst similar has differences to the single implementation. The lines are gradually moving towards each other whereas the single RSME presents drastic decreases in variance for testing data. This proves that using an average can provide results which are more consistent throughout the regularizations. With this being said, the optimal regularization is still the same 10^0.

Task 2:

Section 2.1: Description of the K-Means Clustering

2.1.1: Objective function: SSE (Sum Squared Error)

SSE is the calculation of the differences between each group observation and the groups mean. It's used for measuring the variation between each data point. This is used within K means by finding the Euclidean distance between a data point and the centroid assigned. This distance is then squared and worked out for each data point assigned to each centroid.

number of clusters
number of cases
centroid for cluster j
case i

$$\text{objective function} \leftarrow J = \sum_{j=1}^{k} \sum_{i=1}^{n} \underbrace{\left\| x_i^{(j)} - c_j \right\|^2}_{\text{Distance function}}$$

2.1.2: Centroids

The number of K is often calculated from the scale and shape of the data features. Increasing K without penalty can be seen in the graphs below; increasing the value can lead to less error being detected. For example increase K to the number of data points in the dataset will produce 0 error as each point has its own cluster. There are many methods for choosing the number K centroids: "Elbow method, Average silhouette method and Gap statistic method" (Kassambara, 2018). The cluster centroids are used to assign each data point to one of the K number groups. These will iteratively change throughout the K means process as they move to areas which more feature similarity. For each of the centroid clusters there is a collection of features used to define each group; weights are used to interpret what the centroid cluster represents. The centroids new values kept iterating till the values remain the same; calculation is done by mean of all examples in the cluster. However, first randomization of the centroids is problematic due their positioning in relation to the data points themselves, in term this can take longer to converge or get itself stuck in a local optima; resulting in bad clustering but random centroid assignment or distribution of them over the space can be two solutions to initializing the centroids.

2.1.3: Euclidean distance

$$\sqrt{(x1-y1)^2 + (x2-y2)^2}$$

Euclidean distance is the distance between two points squared. In the case for K means clustering, the first point is the data point at x and y minus the centroid at x and y, square rooted.

2.1.4: Assignment step

$$\underset{c_i \in C}{\operatorname{argmin}}\ dist(c_i, x)^2$$

First assignment is the initialization of the centroids to random locations as mentioned above. After the initialization the assignment step is iterated and each of the data points is placed under the correct centroid, based on the squared Euclidean distance. The 'ci' represents the collection of centroids in the set 'C' and each data point is represent by X is assigned to a cluster base on 'dist.' and then minimum is selected to assign the new centroid.

2.1.5: Update step

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

The updating step requires the new means of centroids to be calculated to created new clusters. This update step iterates until K-means is converged to a local minimum, which can be identified when the mean centroids don't change

and the clusters remain the same. However, the algorithm is usually fast but doesn't guarantee optimum convergence be found due to the randomness of the initialization of centroids.

Section 2.2: Implementation of the K-Means Clustering

2.2.1: Initialize Centroids Function

This method handles the initialization of the centroids on the first iteration. All data points get randomized so that data isn't order related, along with a centroids array which holds all 4 features datasets as index and 4 K centroids for the first position. The iteration of 'i' through features is the assigned the top 4 data locations as the random k clusters on the plot.

```
# intialises the first iteration of the centroid assignment
def initialise_centroids(dataset, K):

        # data gets randomised
        np.random.shuffle(dataset)

        # centroids array stores K amount against the number of features (4)
        centroids = np.zeros((K, 4))

        # selects the top k ampount to be the first iterations centroid assignment
        # loops through the amount of K
        for i in range(K):

            # for each feature (4)
            for features in range(4):

                # stores the centroids from top of the data set to the centroid array
                centroids[i,features] = dataset[i,features]

        #Return
        return centroids
```

2.2.2: Computing the Euclidean Distance function

Two vectors for dataset and centroids are used to compute the Euclidean distance between the dataset and the centroid. Iteration through all the points in the dataset are then applied to the Euclidean equation of squaring the distance from the data point to the centroid for all data in the set and then applying a square root to the sum. The squaring of the distance removes negatives and the square root reverts the data to normalized form.

```
# calculates the distance of two vectors (euclidean distance)
def compute_euclidean_distance(dataset, Centroids):

        # size of the centroid
        centroid_size = Centroids.shape[0]

        # new arry to store the distance, creates by the length of the data in rows and centroid size of columsn
        data_centroid_size_distance = np.zeros (((len(dataset)),centroid_size))

        #for the amount of data
        for _data in range(len(dataset)):
            #for the amount of centroids
            for _centroid in range((centroid_size)):

                # distance for plants
                # loops through centrid i which iterates 0- 4 for addding up the below equation for each distance
                # Equation is Sqaure root(datapoint feature - centroid of feature)^2
                for centroid_i in range(4):
                    data_centroid_size_distance[_data, _centroid] = data_centroid_size_distance[_data, _centroid]
                    + (dataset[_data,centroid_i] - Centroids[_centroid,centroid_i])**2

                # square roots the added up distances
                data_centroid_size_distance[_data, _centroid] = math.sqrt(data_centroid_size_distance[_data, _centroid])

        # returns the distance
        return data_centroid_size_distance
```

2.2.3: Centroid Assignment

The centroid assignment accepts the dataset and the distance created by the Euclidean distance. A run through the size of the dataset (300) is carried out to find the minimum distance of that indexed data point to a centroid. The minimum indicates the centroid which is closest to the data point; all minimum distances are added to an array and are used for averaging on the next iteration.
Also during this calculation a sum of the minimum distances are squared and summed together to create sum squared error for that iteration, which is used later to plot iteration against objective function.

```
# assignign the centroids to the data set
def Centroid_Assignment(dataset, Distance):
    datset_size = dataset.shape[0]
    # array for storing centroids to the data set
    Centroid_assign_data = np.zeros((dataset.shape[0]))

    summed_list = 0.0
    # for all data length (how many samples we have)
    # for each peice of data for the length of the data set
    for single_data in range(datset_size):

        # dist stores the centroid distances from the selected single_data
        dist = Distance [single_data,:]

        # applies the minium distance to the index which is the closest centoird
        min_distance = np.argmin(dist)

        summed_list += np.min(dist)**2

        # minium distance for each centoird is applied to each data point
        Centroid_assign_data[single_data] = min_distance

    # returns the centroid assignement for all data
    return Centroid_assign_data, summed_list
```

2.2.4: Average Centroids

```
# calculates the average centroid for the new iteration, used within the k mean while loop
def Avergae_Centroids(K, dataset, Centroid_assign_data):

    # centroids array stores K amount against the number of features (4)
    new_Centroids = np.zeros((K,4))

    # size of the data set
    dataset_size = dataset.shape[0]

    # for each centroid in K
    for centroid_K in range(K):

                # counter used for applying average
                avg_count = 0

                # array for each feature
                features = [0,0,0,0]

                # each single data in the size of the data set
                for single_data in range(dataset_size):

                        # checks if the centroid assignment is == K
                        # used to count the number for each centroid_K
                        if (Centroid_assign_data[single_data] == centroid_K):

                            # avg_count incremenets (used later for average calculation)
                            avg_count+= 1

                            # for each data point == K add to each feature array
                            for i in range(4):
                                features[i]+= dataset[single_data,i]

                # for each centroid calculation of new mean using avg_count
                for i in range(4):

                    # each cendroid for each feature is given a new value
                    new_Centroids[centroid_K,i] = features[i]/avg_count

    # new vavlues for centroids
    return new_Centroids
```

Iteration of the whole data set is then used to check the index of the data point against each K centroid to identify it and add to average count, which is used to divide later. The features are iterated and each data row for the index is summed together for the averaging calculation later. Once the features have been summed for the entirety of the data set the new centroids can be created by calculating the average for each feature and applying that feature to a new centroid. The returned result should include a k by number of features array, storing the k number of centroids for each features.

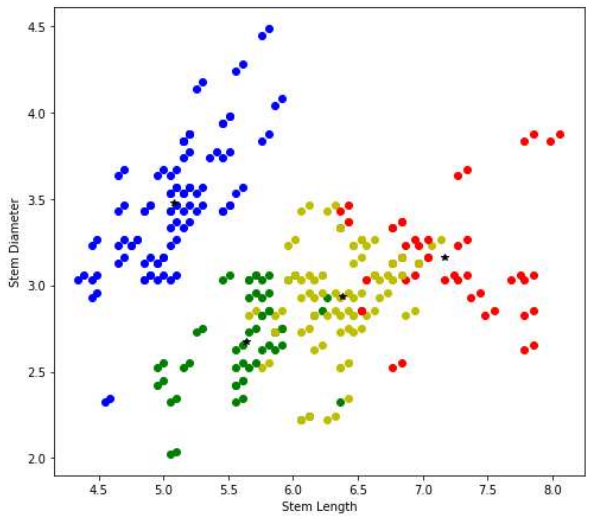
2.2.5: K Means

```python
# Calculates the K menas after intialisation
def kmeans(data_set, K):
        # FIRST ITERATION INTIALISATION
        # requires randomising of the data set + data very unlikely to match on first iteration
        #-----------------------------------------------------------------------------------------------
        #-----------------------------------------------------------------------------------------------

        dist_sums = []

        # Initialise Centroids - First Time select random Data Points
        Centroids = initialise_centroids(data_set, K)

        # Compute distances between data points and centroids
        data_centroid_size_distance = compute_euclidean_distance(data_set, Centroids)

        # Assign Centroids to data
        Centroid_assign_data, dist_sum = Centroid_Assignment(data_set, data_centroid_size_distance)
        dist_sums.append(dist_sum)

        #-----------------------------------------------------------------------------------------------
        #-----------------------------------------------------------------------------------------------

        no_iteration = 1
        loop = 0
        _iterate_K_means = True
        # Start While loop
        while loop == 0:

            # prints the iteration number
            print(no_iteration)

            # new centroids qre creates through averages from the previous data
            Centroids = Avergae_Centroids(K, data_set, Centroid_assign_data)

            # calculates the distance from each centroid to data
            Distance = compute_euclidean_distance(data_set, Centroids)

            # copy saved for comparision to check if completed
            PrevoisCentroidAssignmet = Centroid_assign_data

            # create new assignmetns from the distances and the current data_set
            Centroid_assign_data, summed_dist = Centroid_Assignment(data_set, Distance)

            # checks to see if K means complete, only when previous and new centroid_assign_data is equal is complete
            if (np.array_equal(PrevoisCentroidAssignmet, Centroid_assign_data)):
                print("TRUE")
                break

            # calculates the number of iterations taken
            no_iteration+=1
            dist_sums.append(summed_dist)

        # completed centroid assignment when no changes were found
        print("Completed K means clustering for "+str(K)+" number of centroids!")

        # adds an addition column to the array stating which centroid the each data set belongs to
        Cluster_Assigned = np.column_stack((data_set,Centroid_assign_data))

        #-----------------------------------------------------------------------------------------------
        #-----------------------------------------------------------------------------------------------


        # Return the centroids and Assigned Clustered dataset
        return Centroids, Cluster_Assigned, dist_sums, no_iteration
```

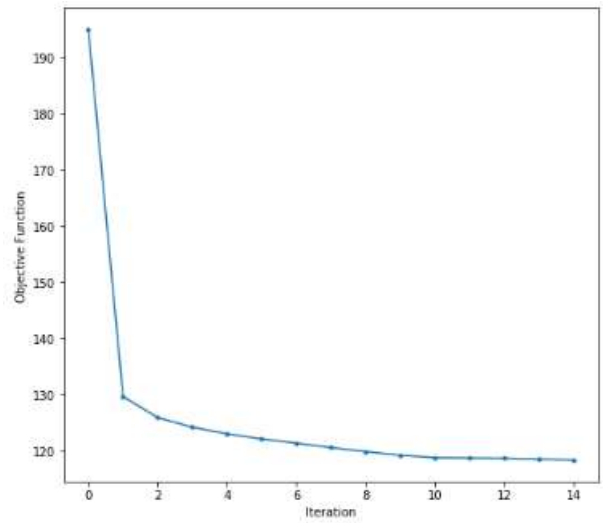
The k means function initializes the first cluster formation, and proceeds to iterate by first creating averages from the centroid assignment, calculating the distance and then applying a new centroid assignment. This new centroid is checked against the previous to check if the optimum convergence has been found; at which the break will kill the program and plot the resulting centroids on a 2d plane of 'stem length and stem diameter'. This 2d plot represents the number of K centroids for each feature.

2.2.6: Graph Plots

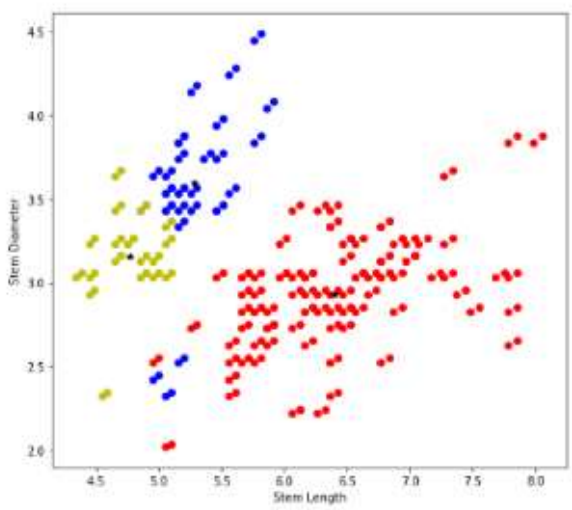
Stem Length to Stem Diameter: K = 4



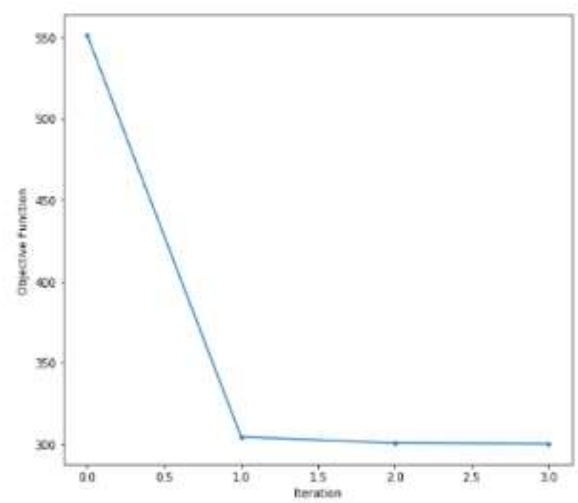
Iteration to Objective Function; K = 4



Stem Length to Stem Diameter; K = 3:



Iteration to Objective Function; K = 3:



The above left graphs are presented in a 2d plane representing the 4d plane of the features clustered to 4 centroids (K). This visual queue allows us to see the visual implementation of the 4 centroids and their clustered data points.

The above Right graphs is in a 2d plane representing the SSE (Sum Squared Error) per iteration of the clustering distances. It shows a decreasing trend as the mean distances becomes smaller to the point of optimal convergence. From K == 4 and K == 3 we can see a drastic change in the amount of error present. With 4 centroids the error at optimal convergence is massively smaller than that on the 3 centroids; previously I mentioned the differences in K number of assignments affecting the optimal convergence and this can be seen outputted in the following graphs.

References

Bangiwala, S. L., 2018. Regresion: Simple Linear. *interations journal of injury control and safety promtotion,* 25(1), pp. 113-1145.

Bruce, M., 2018. *minitab.* [Online]
Available at: https://support.minitab.com/en-us/minitab/18/help-and-how-to/modeling-statistics/anova/supporting-topics/anova-statistics/understanding-sums-of-squares/
[Accessed 15 03 2019].

Daoud, J. I., 2017. Multicollinearity and Regression Analysis. *journal of Physics:.*

Johnson, m. M., 1971. Simple Linear Regression. *Journal of Quality Technology ,* 3(3), pp. 138-143.

Kassambara, A., 2018. *Data Novia.* [Online]
Available at: https://www.datanovia.com/en/lessons/determining-the-optimal-number-of-clusters-3-must-know-methods/
[Accessed 19 03 2019].

Mark Tranmer, M. E., 2008. *hummedia.* [Online]
Available at: http://hummedia.manchester.ac.uk/institutes/cmist/archive-publications/working-papers/2008/2008-19-multiple-linear-regression.pdf
[Accessed 18 04 2019].