Machine Learning | Assignment 3

Thomas Kelly
16323455

Data set identifier: # id:22-44-22

Section (i)

(i) Part A |

Plot the data as a 3D scatter plot - does it look like a curve or plane?

The code to generate a 3D scatter plot is shown below.

```
dataframe = pd.read_csv("week3_data.csv")


X1 = dataframe.iloc[:, 0]
X2 = dataframe.iloc[:, 1]
X = np.column_stack((X1, X2))
y = dataframe.iloc[:, 2]
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X1, X2, y)
```

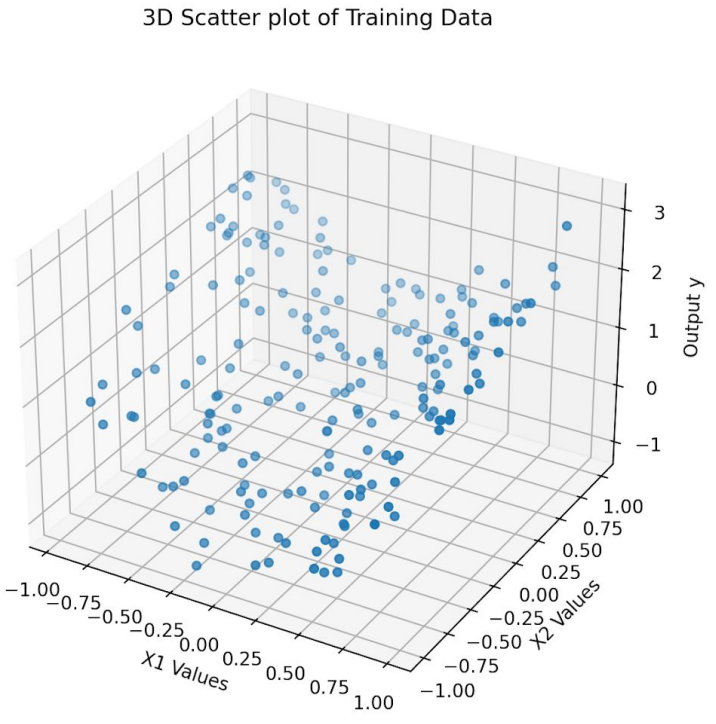

Figure 1 - Scatter plot of the training data (1)

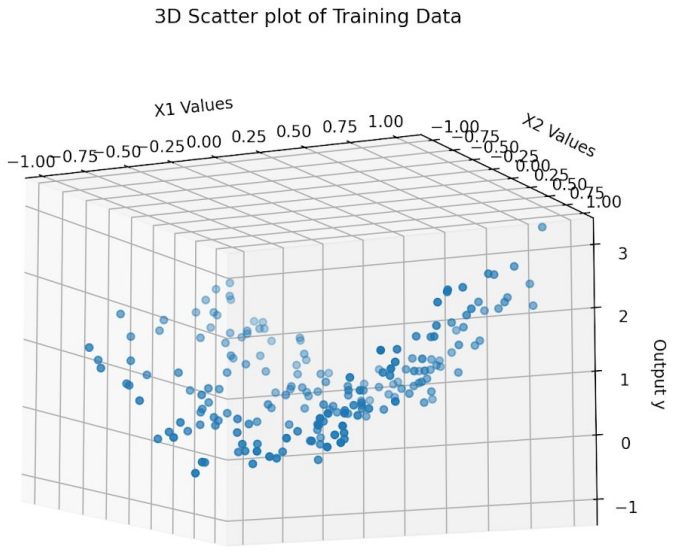

Figure 2 - Scatter plot of training data (2) - an alternative view

The training data appears to lie on a curved plane. Examining the data, I would expect a regression model to take the shape of a curved plane given enough parameters to model the data accurately.

(i) Part B |

Add extra polynomial features, and train a Lasso regression model reporting the parameters for different C values. Discuss how they change as C is varied.

In order to create additional features, the sklearn PolynomialFeatures function is utilised as suggested:

```
polynomial_features = prep.PolynomialFeatures(degree=5)
new_features = polynomial_features.fit_transform(X)
```

This gives a '*new_feature*' array of size 21 to be used to train a Lasso regression model. In order to examine the model parameters for different values of C, the model was trained inside a *for* loop, looping through each '*C*':

```
c_values = [1, 2, 5, 10, 100, 1000, 10000]
thetas = []
for i, C in enumerate(c_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha=1/2C
   lassoModel = linear_model.Lasso(alpha=alpha, random_state=0)
   lassoModel.fit(new_features, y)
   thetas.append(lassoModel.coef_)
```

```
print("--> COEF: ", lassoModel.coef_)
print("-->INTERCEPT: ", lassoModel.intercept_)
```

As mentioned in the assignment document, alpha is defined as; $\alpha = \frac{1}{2C}$;
The value for C = 1 was chosen to start with as anything C = 1 and anything lower than this gave a model where all parameters were 0. The table below shows the model parameters for each Lasso model trained with a different C value.

C	1	2	5	10	100	1000	10,000
θ_1	0	0	0	0	0	0	0
θ_2	0	0	0	0	-0.0112	-0.0112	0.0088
θ_3	0	0.30185	0.73766	0.878	1.003	1.077	0.97633
θ_4	0	0	0.8464	1.455	2.002	2.056	2.067
θ_5	0	0	0	0	0	-0.1078	-0.153
θ_6	0	0	0	0	0	-0.005	-0.178
θ_7	0	0	0	0	0	0	0
θ_8	0	0	0	0	0	0	0.172
θ_9	0	0	0	0	0	-0.1525	-0.35
θ_{10}	0	0	0	0	0	0	-0.3811
θ_{11}	0	0	0	0	0	0	-0.0103
θ_{12}	0	0	0	0	0	-0.1081	-0.109
θ_{13}	0	0	0	0	0	0	0.0104
θ_{14}	0	0	0	0	0	0.3316	0.412
θ_{15}	0	0	0	0	0	0	0.1975
θ_{16}	0	0	0	0	0	0.07633	0.089
θ_{17}	0	0	0	0	0	0	-0.089
θ_{18}	0	0	0	0	0	0	0
θ_{19}	0	0	0	0	0	-0.176	-0.3556
θ_{20}	0	0	0	0	0	0	0.2077
θ_{21}	0	0	0	0	0	-0.0864	-0.389

Figure 3 - Table of C values with reported model parameters

Quickly examining some graphs for some of the more influential parameter values (y-axis) plot vs. values of C (x-axis), a trend away from zero becomes obvious as C increases:

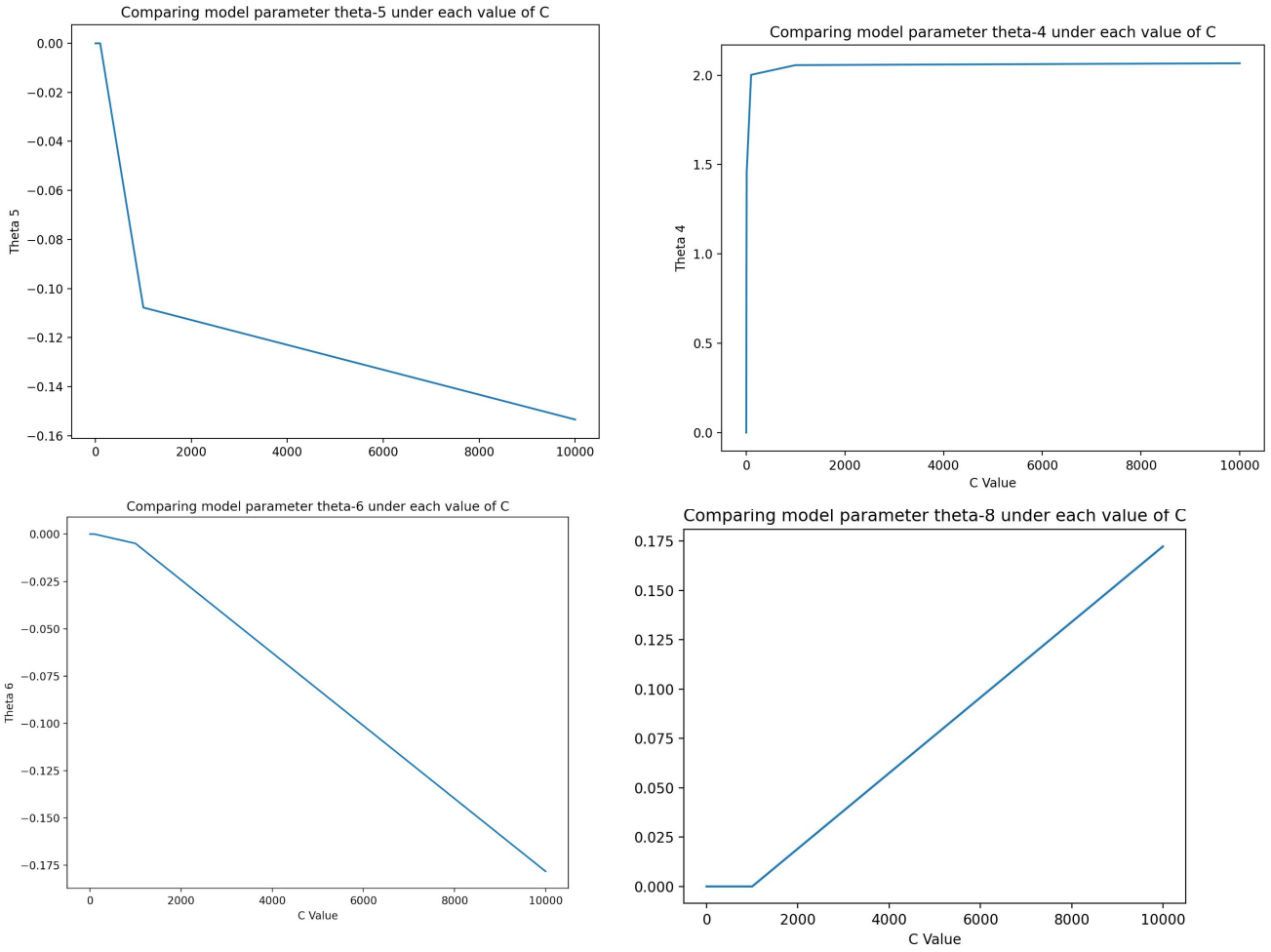

Figure 4 - Plots showing different parameters (theta - θ) vs. potential C values

It is clear that as the model's C value is increased, the parameters $\boldsymbol{\theta}$ tend to move away from zero, whether that be in a negative or positive direction. This is the expected behaviour for the Lasso Regression model, which incorporates a L1 regularisation penalty:

- $J(\boldsymbol{\theta}) = \frac{1}{m} \sum (h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{1}{C} \sum^{n}_{j=1} |\boldsymbol{\theta}_j|$

Focusing on the penalty term, $\frac{1}{C} \sum^{n}_{j=1} |\boldsymbol{\theta}_j|$, when the value for C is lower this penalty term will be larger. The cost function will respond by making as many parameters equal to zero as it can in order to minimise cost. When C is a larger value, the penalty becomes less important and the model moves towards regular linear regression behaviour. More generally, the effect of the L1 regularization penalty is lessened as C increases, and as a result more parameters become non-zero values.

(i) Part C |

After generating ‘Xtest’ data as described in the assignment document, the model is used to predict ‘z’ values for this new training data.

```
predictions = lassoModel.predict(Xtest)
predictions = np.reshape(predictions, (50, 50))
X1_test = np.reshape(X1_test, (50, 50))
X2_test = np.reshape(X2_test, (50, 50))

ax.plot_surface(X1_test, X2_test, predictions,
                color="red", alpha=0.5, vmin=-2, vmax=2)
```

The data is also reformatted using reshape in order to pass it into the ‘*plot_surface*’ function as a series of 2D arrays. This was done for the variety of ‘C’ values previously mentioned, displaying the 3D plots as follows;

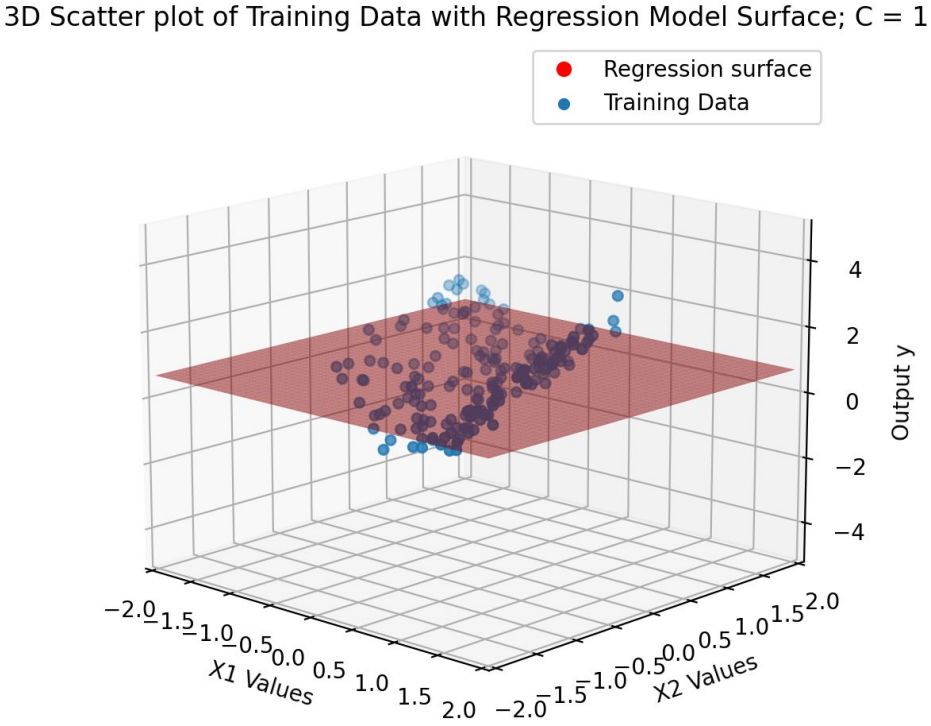

Figure 5 - Training data & regression surface for C = 1

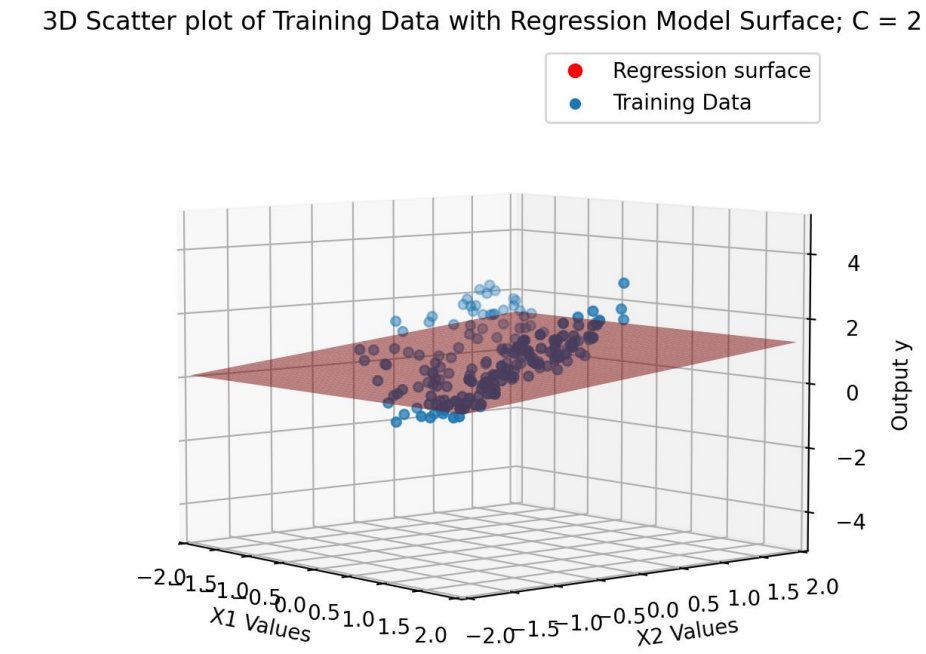

Figure 6 - Training data & regression surface for C = 2

Evidently for low values of C, underfitting occurs. This is expected as we simply do not have enough parameters to accurately predict new data. This can be backed up with reference to the table in figure 3 for C = 1, 2.

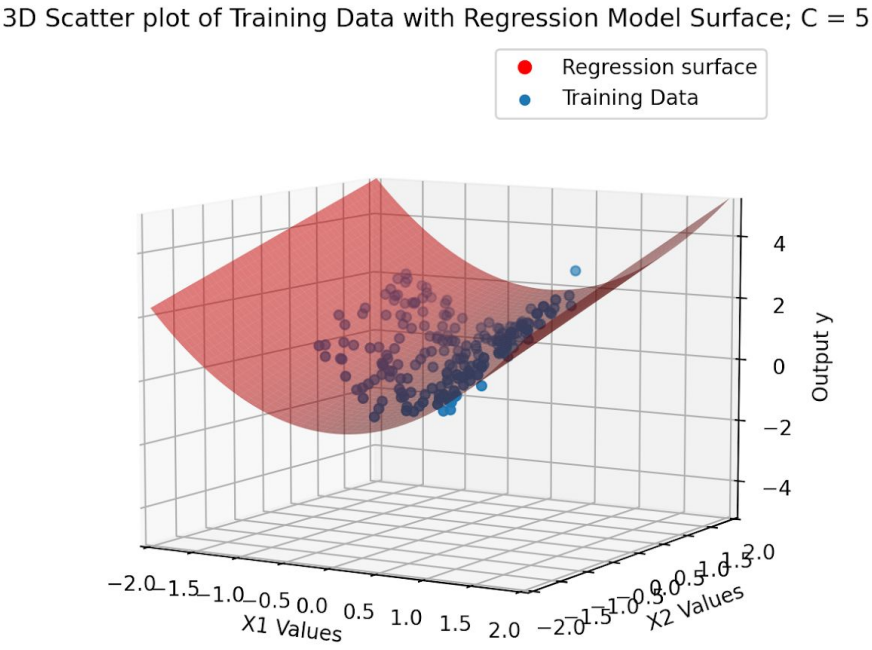

Figure 7 - Training data & regression surface for C = 5

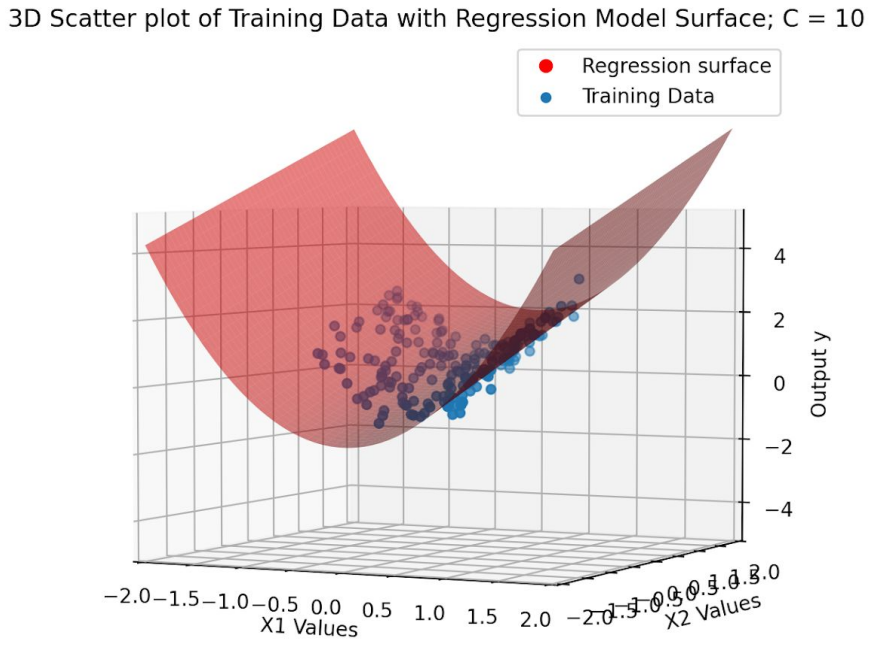

Figure 8 - Training data & regression surface for C = 10

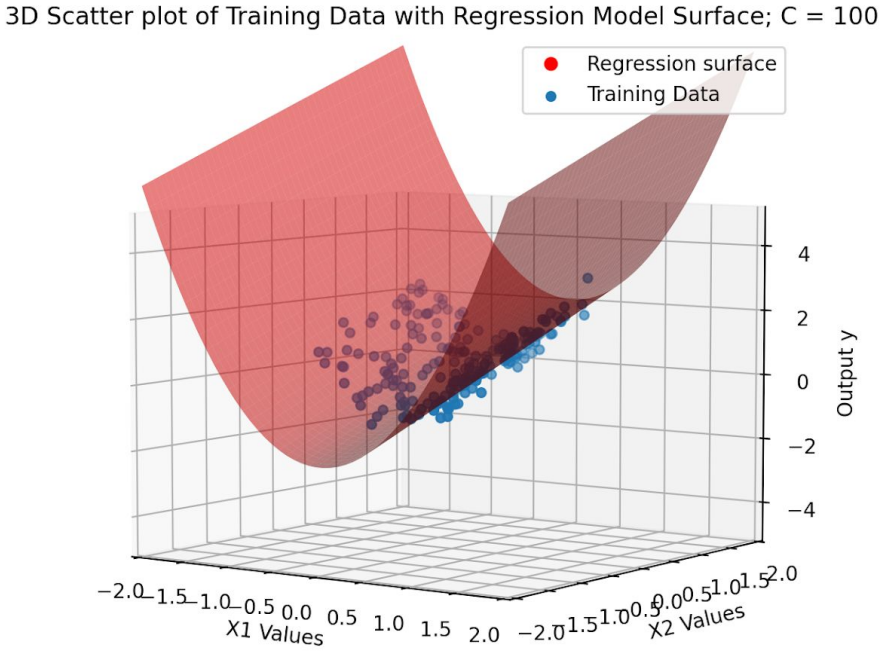

Figure 9 - Training data & regression surface for C = 100

The next three values for C that were used (C = 5, 10, 100) result in a much more visually accurate prediction model. As is evident from these graphs, the regression surface fits the training data accurately in terms of its shape. Referencing the parameter data for these models, they are all common in the way that they utilise parameters Θ_3, Θ_4 with all other Θ set to 0. This appears to fit the model most accurately, with no over or underfitting occurring. As C is increased further, we see overfitting - with the majority of the parameters Θ becoming non-zero.

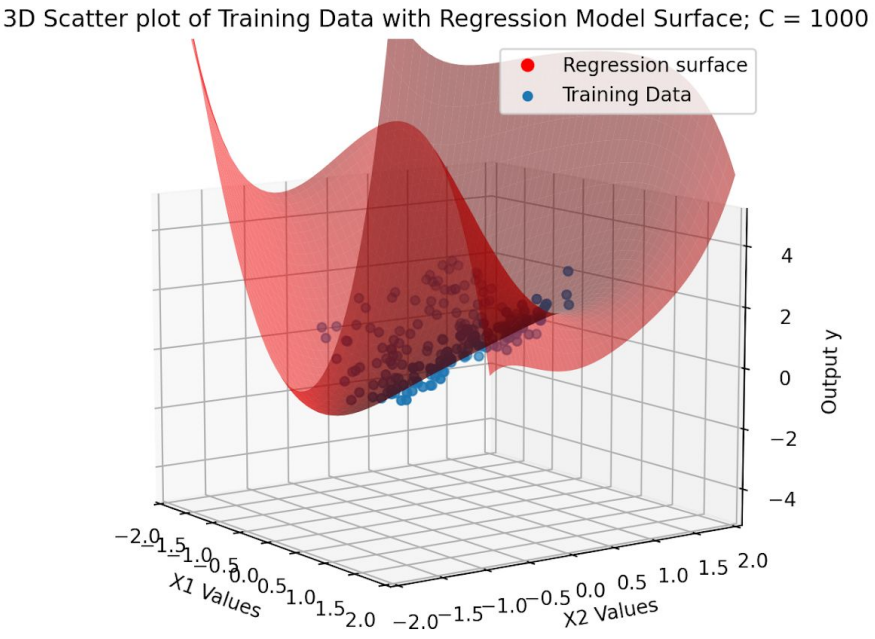

Figure 9 - Training data & regression surface for C = 1000

(i) Part D |

Overfitting

- Overfitting occurs when we over-complicate a model by adding too many parameters, therefore introducing noise to the model. At points where there is no training data, the model generalises very poorly and error is high. However at points where there is training data the error is very low.

Underfitting

- Underfitting occurs in the opposite case in which we do not have sufficient parameters to build an accurate model - the model is too simple and fails to capture the behaviour of the data. The error for both test and training data will be high.

With reference to my answer for (i) (b), the trend in this case can be explained by noting that L1 regularisation attempts to set as many parameters as it can to zero. As we increase C, we reduce the effect that the regularisation penalty has and therefore more parameters will be nonzero. If this penalty is made very insignificant - i.e. C => 1000 - we get overfitting, because we allow more non-zero parameters.

An appropriate value for C here could be chosen through a method for hyperparameter tuning that gives a low error between the training data and trained model, such as k-fold cross validation.

(ii) Part E |

Revisiting parts (b) and (c) with a Ridge Regression model, the same code is utilized to generate ridge regression models for different values of C.

```
c_values = [0.001, 0.01, 0.1, 1, 2, 5, 10, 100]
for i, C in enumerate(c_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha = 1/2C
   ridgeModel = linear_model.Ridge(alpha=alpha, random_state=0)
   ridgeModel.fit(new_features, y)
   thetas.append(ridgeModel.coef_)

   predictions = ridgeModel.predict(Xtest)
```

The parameters for each model are shown (crudely) in a screenshot of my terminal below;

```
------ C:  0.001 ------
--> COEF:  [ 0.         -0.01309615  0.11332384  0.06381489 -0.00697683  0.0021482
 -0.00381475  0.03936502 -0.01230942  0.06349155  0.05394309 -0.00468711
  0.02046789 -0.00348066  0.00076124  0.0010525   0.02277937 -0.00561563
  0.0214073  -0.00914668  0.04350662]
------ C:  0.01 ------
--> COEF:  [ 0.         -0.02929444  0.44795861  0.4175659  -0.01890997 -0.01662057
  0.00362962  0.14978044 -0.04417046  0.20865532  0.3460463  -0.01393699
  0.11711361  0.00372534 -0.02410464  0.02199131  0.08427484 -0.01862219
  0.06750264 -0.02730839  0.12064039]
------ C:  0.1 ------
--> COEF:  [ 0.         -0.02921919  0.81805038  1.03262132 -0.03354569 -0.04404004
  0.01726775  0.1360687  -0.07883092  0.18195356  0.73664215 -0.0491628
  0.22140973  0.10014286 -0.05053051  0.04263367  0.02740716 -0.032349
 -0.04201529 -0.00456149 -0.02489589]
------ C:  1 ------
--> COEF:  [ 0.         -0.01983952  1.00191834  1.57357251 -0.10243738 -0.09351896
  0.037159    0.12397412 -0.18193586  0.17801094  0.47167776 -0.12167372
  0.14208668  0.30616186  0.0568096   0.04717656 -0.0790225  -0.03267573
 -0.24838941  0.0627143  -0.20546297]
------ C:  2 ------
--> COEF:  [ 0.         -0.01326361  1.00270469  1.74273267 -0.12393881 -0.12618174
  0.03529189  0.1537096  -0.23265453  0.2214974   0.31156296 -0.12451441
  0.1044222   0.35591554  0.10840659  0.05290251 -0.09930155 -0.02146742
 -0.29686284  0.1040368  -0.25016016]
------ C:  5 ------
--> COEF:  [ 0.00000000e+00  1.36436730e-04  9.88242467e-01  1.90944691e+00
 -1.42891688e-01 -1.64401166e-01  1.55879087e-02  1.94833082e-01
 -3.04345640e-01  2.98264941e-01  1.44646127e-01 -1.20300314e-01
  6.74982058e-02  3.95205722e-01  1.64808942e-01  7.10703899e-02
 -1.23808212e-01  2.59283958e-04 -3.44113797e-01  1.64223586e-01
 -3.14819318e-01]
------ C:  10 ------
--> COEF:  [ 0.          0.01146418  0.97496156  1.9850729  -0.15010061 -0.18407129
 -0.00860292  0.21889685 -0.34980918  0.354176    0.06674524 -0.11670675
  0.05123505  0.40903379  0.19262963  0.09066661 -0.13817045  0.01806509
 -0.36785167  0.20122036 -0.3599806 ]
------ C:  100 ------
--> COEF:  [ 0.          0.03231695  0.95257576  2.06634602 -0.15571209 -0.20733975
 -0.06346093  0.25069134 -0.41412657  0.44368977 -0.01855778 -0.11219638
  0.03424111  0.41918505  0.22455342  0.13414114 -0.15701868  0.05020982
 -0.39871005  0.24984323 -0.43185762]
```

Figure 10 - Terminal screenshot showing parameter values

Quickling glancing through and comparing these and understanding the L2 ‘Tikhonov Regularisation’ used by Ridge Regression, we can identify a similar pattern in the parameters to that of Lasso Regression. The difference is that L2 regression encourages small parameter values, opposed to attempting to set as many of them to zero as it can. As C increases, we see a **general** movement of the parameter values away from 0.

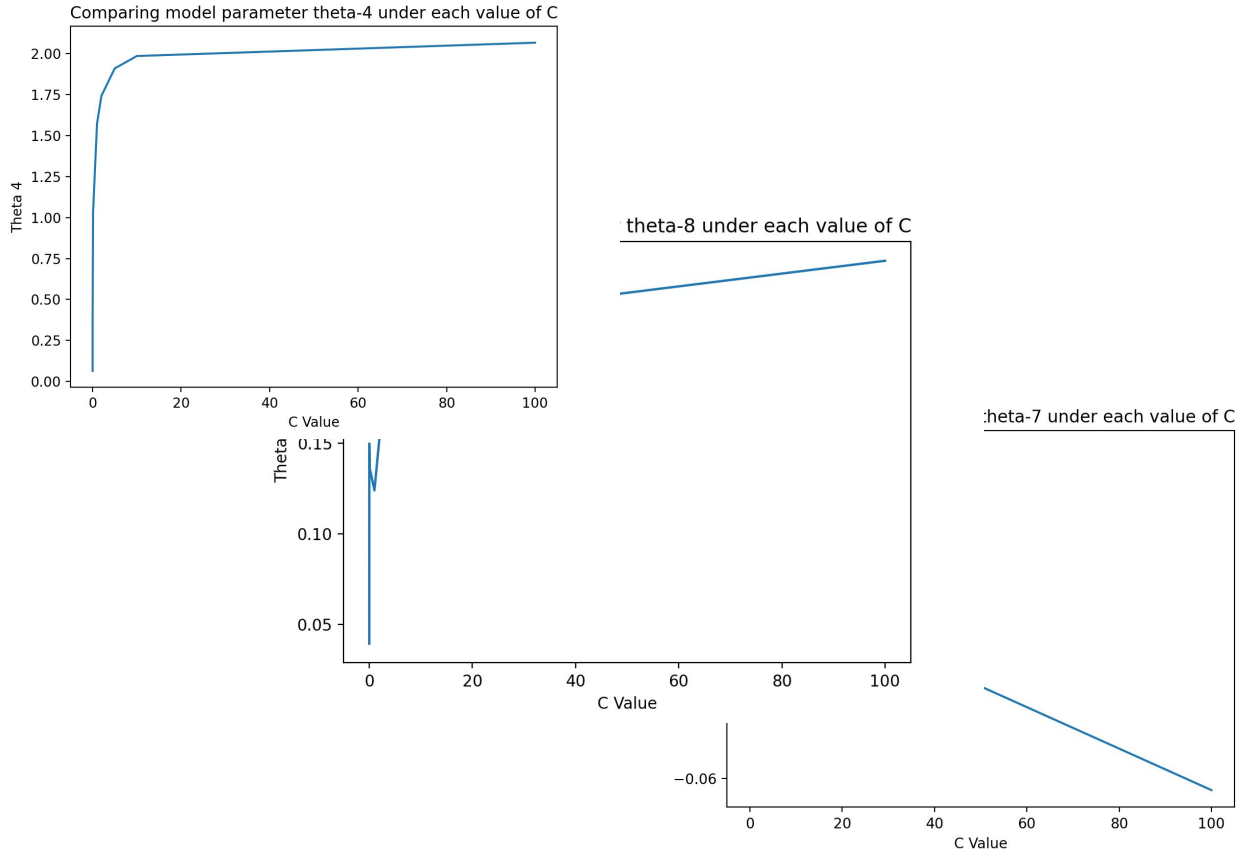

Figure 11 - Showing general trend away from zero for some example parameters as C is increased, for a Ridge Regression model

Due to the fact all of the parameters are non-zero (however mostly very small), I would estimate that this model would be much more vulnerable to overfitting, depending on the value of C chosen. This can be confirmed by examining the regression surfaces reported in graphs for different values of C.
It is also worth noting that smaller values of C were selected to start with in order to demonstrate the most interesting range of C values for this model.

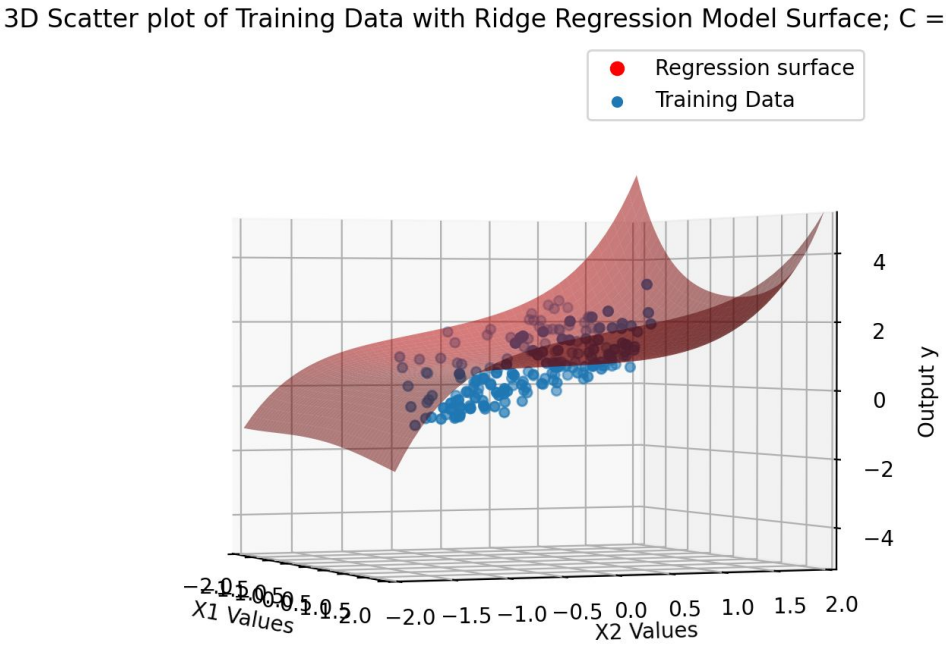

Figure 12 - Training data, ridge regression surface; C = 0.001

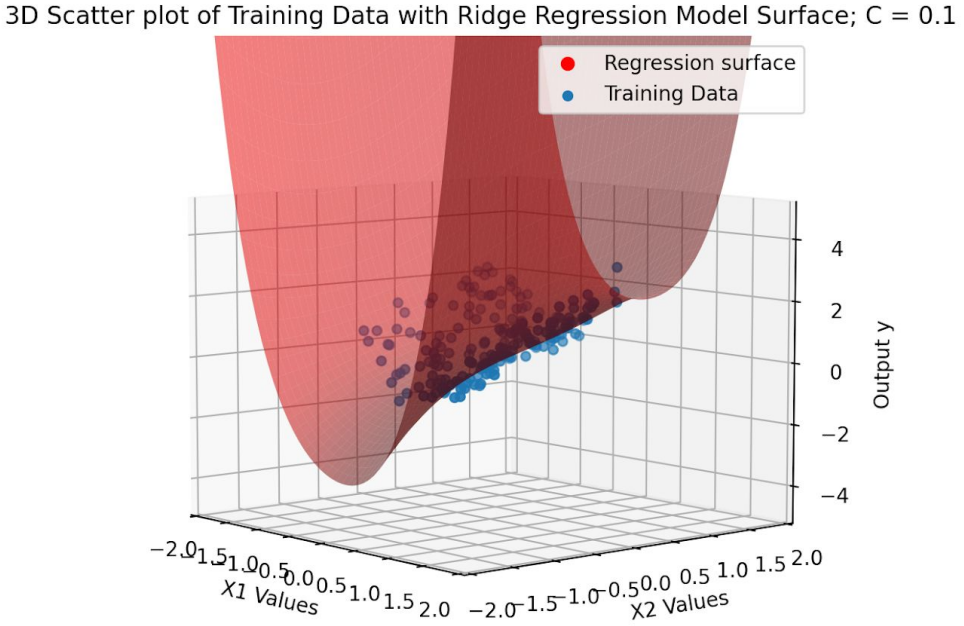

Figure 13 - Training data, ridge regression surface; C = 0.1

In this case, small values of C seem to provide the models with the most potential, based on a visual judgement. This is because these models encourage a high L2 regularisation penalty, keeping most values of $\boldsymbol{\theta}$ close to zero, however still allow enough room for the more influential parameters to be correctly set. After C = 0.1, increasing the order of magnitude to C = 1 overfitting starts to occur as we see an increase in all parameters $\boldsymbol{\theta}$.

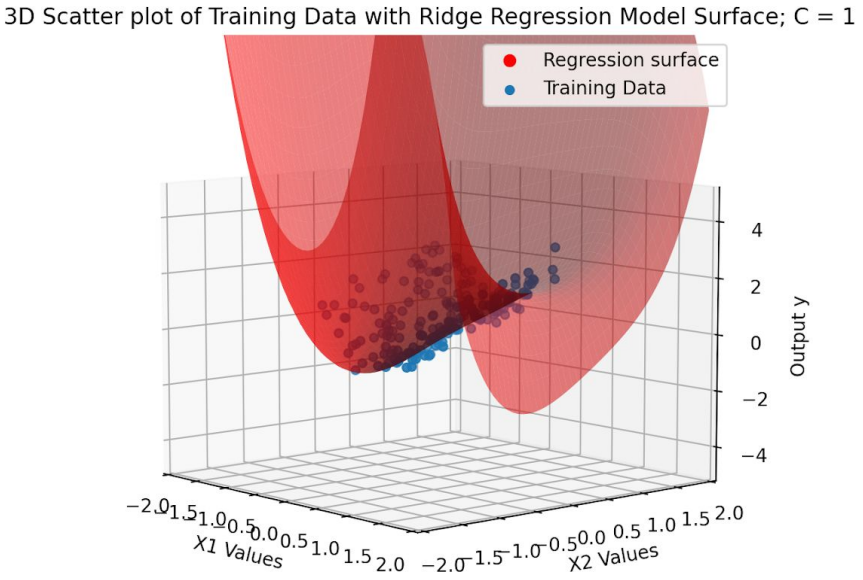

Figure 13 - Training data, ridge regression surface; C = 1

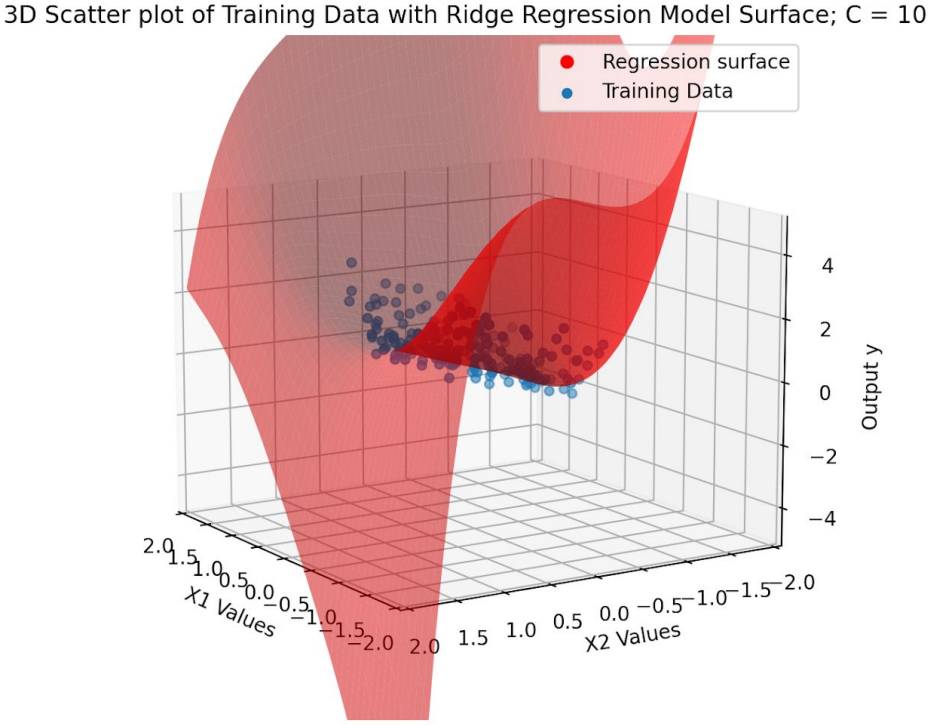

Figure 14 - Training data, ridge regression surface; C = 10

Models using values for C above 10 display graphs that are clearly overfit and are difficult to visualise.

Comparing the impact of modifying C for Ridge & Lasso Regression has on model parameters |
The difference between the behaviour of the models essentially comes down to the regularisation penalty utilized by each model.

- **Lasso Regression**

Using L1 regularisation, as previously mentioned, non-zero parameters are encouraged when the value of C is low. If C is too low, we get underfitting because too many values of are set to 0. As C increases, we see more non-zero parameters and overfitting will occur if C is increased too much.

- **Ridge Regression**

Rather than encouraging parameters to be zero, L2 regularisation simply encourages smaller values of $\boldsymbol{\theta}$. A low value of C will result in smaller values of $\boldsymbol{\theta}$ close to zero, however all parameters will be non-zero. This makes the model much more vulnerable to overfitting when C isn't even that high - in this case C = 1 begins to exhibit overfitting.

There is a different optimal range of C values for both Lasso & Ridge regression, however they are similar in the sense that an increase in C will mean an increase in influence of all parameters and eventual overfitting.

Section (ii)

(ii) Part A |

A value for C = 1 is set and k-fold cross validation is used to set up the lasso regression model.

```
C = 1
split_values = [2, 5, 10, 25, 50, 100]
for i, split in enumerate(split_values):
   alpha = 1/(2*C)
   kf = KFold(n_splits=split)
```

Looping through different values for k, the mean square error for each k ('split') can have its values for mean error and variance examined. The below 'for' loop is nested within the one above.

```
for train, test in kf.split(new_features):
      lassoModel = linear_model.Lasso(alpha=alpha, random_state=0)
      lassoModel.fit(new_features[train], y[train])

      predictions = lassoModel.predict(new_features[test])
      mse = mean_squared_error(y[test], predictions)
      sum_errors = sum_errors + mse
      error_array.append(mse)
      loopCount = loopCount + 1
 mean = sum_errors / split
```

```
print("Mean: ", mean)
variance = np.var(error_array)
print("Variance: ", variance)
mean_array.append(mean)
stan_dev_array.append(np.array(error_array).std())
```

This code results in an array with mean values for mean squared error at each potential k-fold value (2, 5, 10, 25, 50, 100). Plotting the average error against each k-fold value results in the plot below;

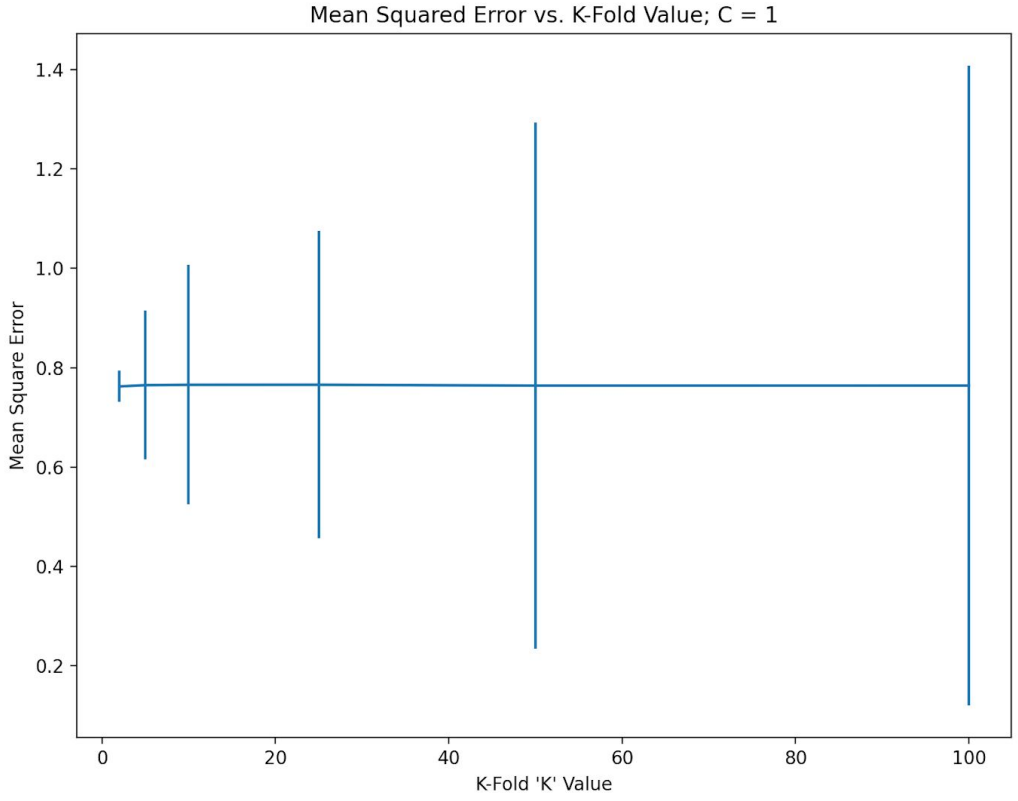

Figure 15 - Errorbar plot for Mean Squared Error vs. KFold values

If a low value of k is selected, we have a larger test set for the model in question. This helps to avoid noise in the test data. However, we also want a large training set of data to ensure the model has sufficient data to be accurately trained. Therefore finding a balance in the amount of testing and training data (dictated by k-fold value) is necessary. With reference to the above plot, it is evident that choosing a low k value of **k = 2 or k = 5** would result in the model with the least prediction error, and lowest standard deviation in this error. It is also worth noting that the higher the value of k, the higher the computational cost is on the model which is also undesirable.

(ii) Part B |

Setting a value for k = 5 for k-fold cross-validation, a plot is generated for mean & standard deviation vs. C.

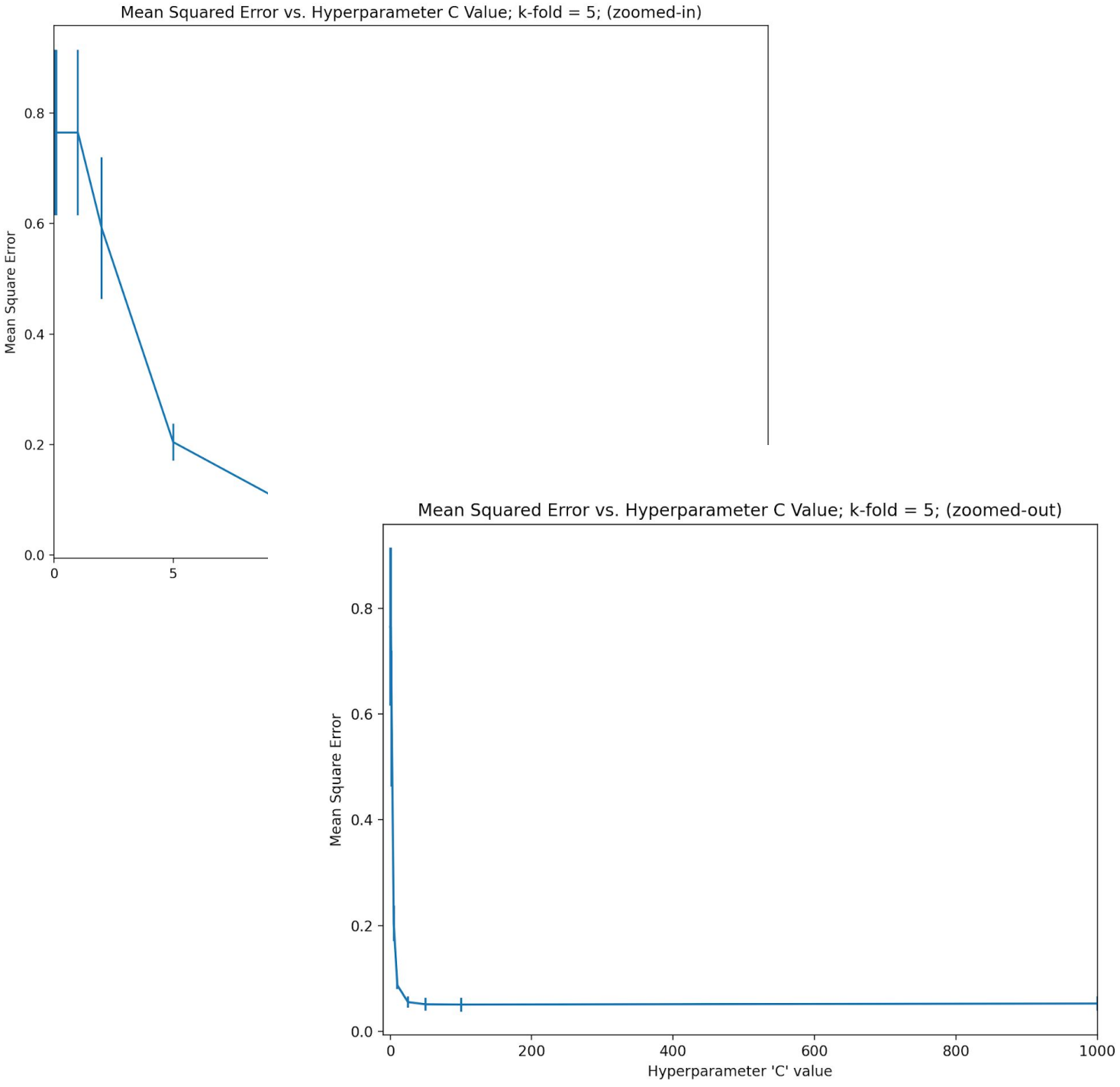

Figure 16 - Mean Squared Error vs. C values

Examining this graph, it is apparent that an optimal value for C occurs after C = 100. With

closer examination of the data, an optimal value for mean error can be verified at C = 100, where the error is 0.05064....

```
--- C =  0.01  ---
Mean:  0.7649924752945837 - Variance:  0.0223714302689082
--- C =  0.1  ---
Mean:  0.7649924752945837 - Variance:  0.0223714302689082
--- C =  1  ---
Mean:  0.7649924752945837 - Variance:  0.0223714302689082
--- C =  2  ---
Mean:  0.5916545329154987 - Variance:  0.016546247982008124
--- C =  5  ---
Mean:  0.20418646250225572 - Variance:  0.0011224933159250754
--- C =  10  ---
Mean:  0.0875017995634746 - Variance:  5.852148778459328e-05
--- C =  25  ---
Mean:  0.05542957969017668 - Variance:  0.00010874851565190607
--- C =  50  ---
Mean:  0.0511148482526043 - Variance:  0.00015191784694476974
--- C =  100  ---
Mean:  0.05064239433601053 - Variance:  0.00017272726544030894
--- C =  1000  ---
Mean:  0.052573061879971475 - Variance:  0.00018987984291686667
--- C =  10000  ---
Mean:  0.05707709990945681 - Variance:  0.00016769674656831788
MINIMUM 0.05064239433601053
```

Figure 17 - Terminal window displaying mean square error values for each value of C

The range of values for C (0.01, 0.1, 1, 2, 5, 10, 25, 50, 1000, 10000) were chosen as such because I think they capture a range of the most interesting behaviour in terms of error. In Part (i), low values of C were shown to produce models where most parameter values were forced to 0, in the case of lasso regression. Low values of 0.01 capture this behaviour also displaying a high error. The values between 1-100 result in an error that drops slowly to a minimum value around C = 100, before increasing very slightly again as it approaches C = 1,000.

(ii) Part C |

Given a k-fold value of 5, choosing a **C = 100** results in the lowest amount of prediction error, while also avoids overfitting. Recalling the graph in Part (i) we can verify that this looks like an accurate selection.

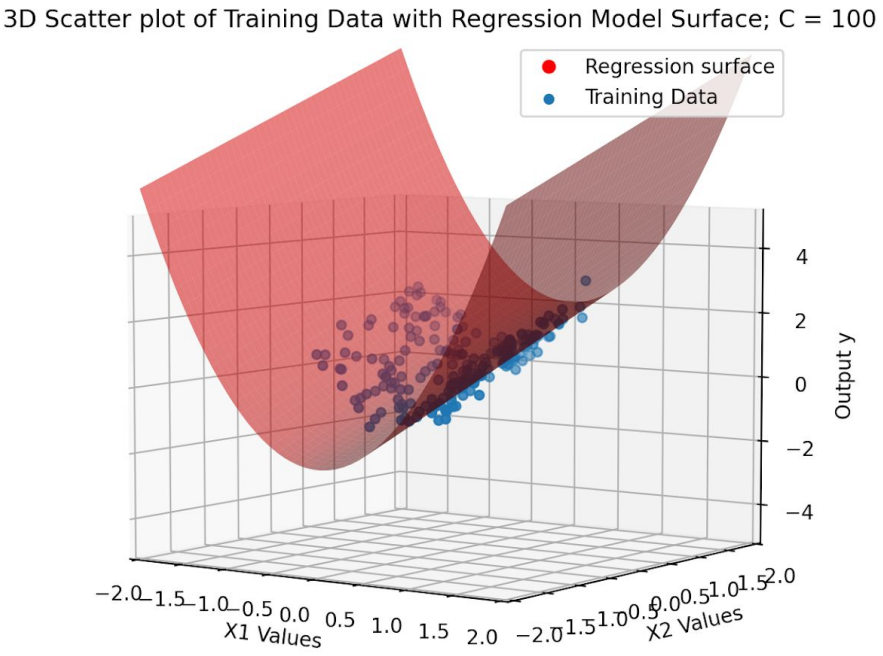

Figure 9 - Training data & regression surface for C = 100

As previously discussed, values for C that are too small make the penalty too strong for the Lasso model, causing underfitting and an inaccurate model. Values of C that are too large will lead to overfitting. Therefore, choosing a value of C that avoids overfitting and results in a low prediction error is the most logical strategy here. It is worth noting that slightly lower value of C = 100 could be used, in order to use a 'simpler' model, once this didn't result in too much of an increase in error. Examining figure 17, it would probably be pretty safe to decrease C slightly towards 50 without too much of an error penalty, however I will settle for C = 100 for the aforementioned reasons.

(ii) Part D | Repeat b-c for a Ridge Regression Model

(b) Plot prediction error vs C, and justify choice of range of C - values

The same code in *'week3_kfold_c.py'* is modified to use a ridge regression model. A different range of values [0.001, 0.01, 0.1, 1, 2, 5, 10, 25, 50, 100] were chosen similar to that of part (i) due to the fact the more accurate models seem to occur where values of C are much lower and with examination of the minimum value for error. This is due to the fact under Ridge Regression and L2 regularisation - low parameter values are encouraged and a low C value is conducive for this. Overfitting occurs at much lower values, after C = 1. A plot for Mean squared error vs. C values is produced;

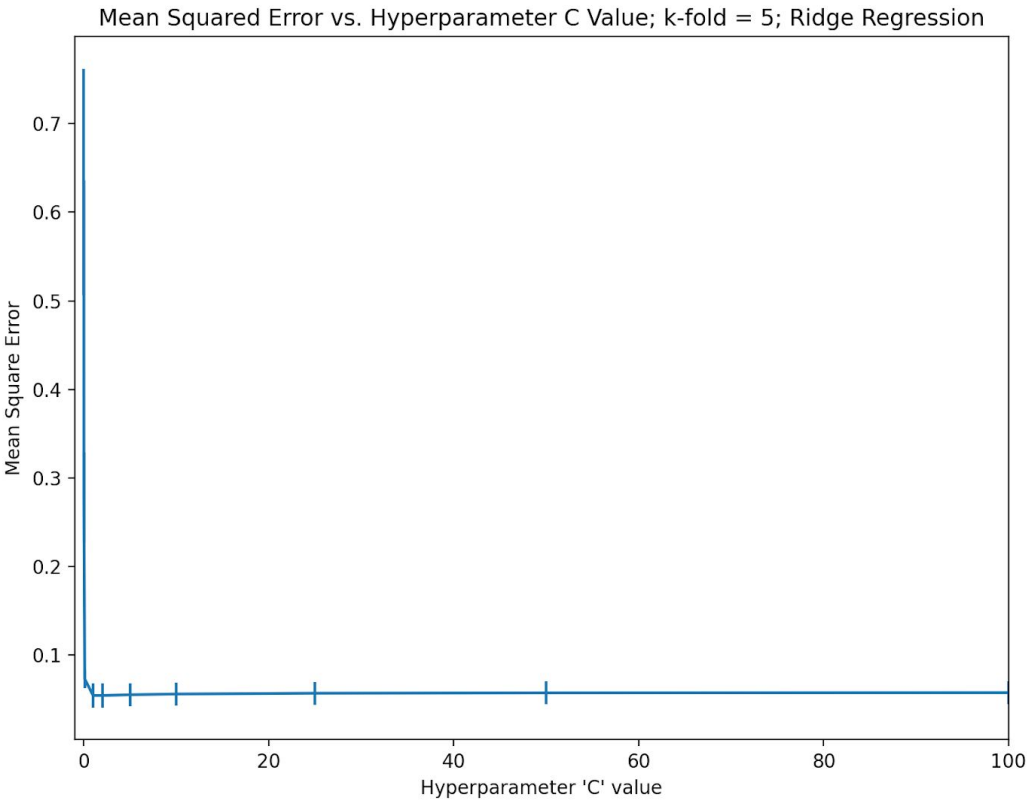

Figure 19 - Ridge Regression; Mean squared error vs C values for 5-fold Cross Validation

(c) What value of C would you recommend and explain your choice?
Although it does not result in the lowest error for the test data, (C = 1 does) I would choose C = 0.1. This is largely due to the fact that a value of C = 1 does not appear to generalise well when its regression surface graph is examined.

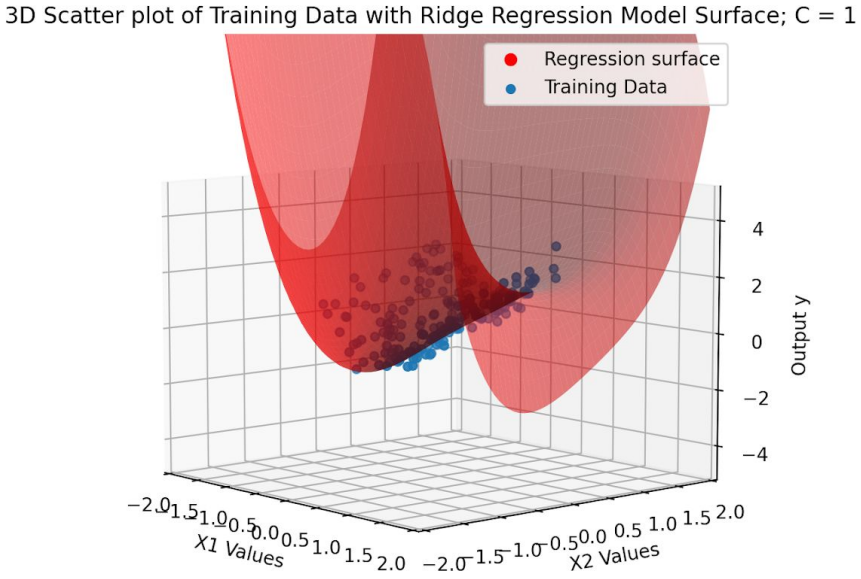

Figure 13 - Training data, ridge regression surface; C = 1

In this case, the test data produced in k-fold validation has failed to encapsulate the behaviour of the model outside training data range. Checking the same graph for C = 0.1, the model

appears to much more accurately model the data, avoiding overfitting.

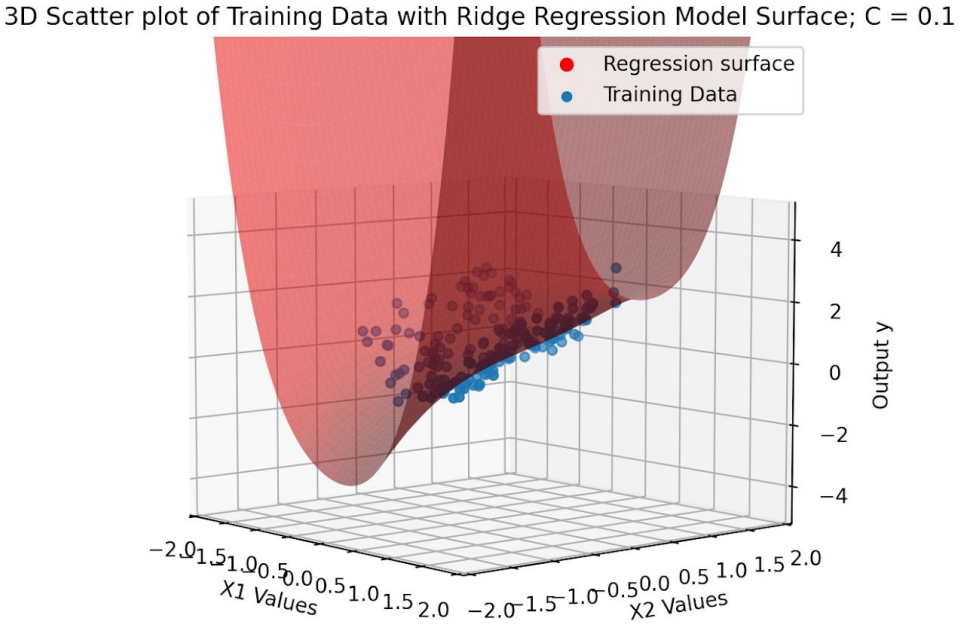

Figure 13 - Training data, ridge regression surface; C = 0.1

Code |

week3_ part1.py | Lasso Regression model questions

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.preprocessing as prep
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl



dataframe = pd.read_csv("week3_data.csv")

X1 = dataframe.iloc[:, 0]
X2 = dataframe.iloc[:, 1]
X = np.column_stack((X1, X2))

y = dataframe.iloc[:, 2]

polynomial_features = prep.PolynomialFeatures(degree=5)
new_features = polynomial_features.fit_transform(X)

# ---- (i)(c) Setup Xtest data
```

```
Xtest = []
grid = np.linspace(-2, 2)
for i in grid:
   for j in grid:
       Xtest.append([i, j])
Xtest = np.array(Xtest)
Xtest = polynomial_features.fit_transform(Xtest)
Xtest_columns = np.column_stack(Xtest)
X1_test = Xtest_columns[1]
X2_test = Xtest_columns[2]

c_values = [1, 2, 5, 10, 100, 1000, 10000]
thetas = []
for i, C in enumerate(c_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha = 1/2C
   lassoModel = linear_model.Lasso(alpha=alpha, random_state=0)

   lassoModel.fit(new_features, y)
   thetas.append(lassoModel.coef_)
   print("XTEST: ", Xtest)
   predictions = lassoModel.predict(Xtest)
   predictions = np.reshape(predictions, (50, 50))
   X1_test = np.reshape(X1_test, (50, 50))
   X2_test = np.reshape(X2_test, (50, 50))
   print("PREDICTIONS: ", predictions)
   print("--> COEF: ", lassoModel.coef_)
   print("-->INTERCEPT: ", lassoModel.intercept_)

   # (i) (a)
   fig = plt.figure(1)
   ax = fig.add_subplot(111, projection='3d')

   sc = ax.scatter(X1, X2, y, label="Training Data")
   ax.plot_surface(X1_test, X2_test, predictions,
                   color="red", alpha=0.5, vmin=-2, vmax=2)

   ax.set_title(
       "3D Scatter plot of Training Data with Regression Model Surface; C = " +
str(C))
   ax.set_xlabel("X1 Values")
   ax.set_ylabel("X2 Values")
   ax.set_zlabel("Output y")
   ax.set_xlim([-2, 2])
   ax.set_ylim([-2, 2])
   ax.set_zlim([-5, 5])
```

```
    fake2Dline = mpl.lines.Line2D(
        [0], [0], linestyle="none", c='r', marker='o')
    ax.legend([fake2Dline, sc], ['Regression surface',
                                 'Training Data'], numpoints=1)
    plt.show()

thetas = np.array(thetas)
theta_for_each_c = np.column_stack(thetas)



# (i)(b)
count = 1
for k in enumerate(theta_for_each_c):
    plt.figure(2)
    plt.plot(c_values, k[1])
    plt.title("Comparing model parameter theta-" +
              str(count) + " under each value of C")
    plt.ylabel("Theta " + str(count))
    plt.xlabel("C Value")
    count = count + 1
    plt.show()
```

week3_ridge.py | Ridge Regression model questions

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.preprocessing as prep
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl



dataframe = pd.read_csv("week3_data.csv")

X1 = dataframe.iloc[:, 0]
X2 = dataframe.iloc[:, 1]
X = np.column_stack((X1, X2))

y = dataframe.iloc[:, 2]

polynomial_features = prep.PolynomialFeatures(degree=5)
new_features = polynomial_features.fit_transform(X)

# ---- (i)(c) Setup Xtest data
```

```
Xtest = []
grid = np.linspace(-2, 2)
for i in grid:
   for j in grid:
       Xtest.append([i, j])
Xtest = np.array(Xtest)
Xtest = polynomial_features.fit_transform(Xtest)
Xtest_columns = np.column_stack(Xtest)
X1_test = Xtest_columns[1]
X2_test = Xtest_columns[2]

thetas = []
c_values = [0.001, 0.01, 0.1, 1, 2, 5, 10, 100]
for i, C in enumerate(c_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha = 1/2C
   ridgeModel = linear_model.Ridge(alpha=alpha, random_state=0)
   ridgeModel.fit(new_features, y)
   thetas.append(ridgeModel.coef_)

   predictions = ridgeModel.predict(Xtest)
   predictions = np.reshape(predictions, (50, 50))
   X1_test = np.reshape(X1_test, (50, 50))
   X2_test = np.reshape(X2_test, (50, 50))
   print("------ C: ", C, "------")
   print("--> COEF: ", ridgeModel.coef_)
   # (i) (a)
   fig = plt.figure(1)
   ax = fig.add_subplot(111, projection='3d')

   sc = ax.scatter(X1, X2, y, label="Training Data")
   ax.plot_surface(X1_test, X2_test, predictions,
                   color="red", alpha=0.5, vmin=-2, vmax=2)

   ax.set_title(
       "3D Scatter plot of Training Data with Ridge Regression Model Surface; C = "
+ str(C))
   ax.set_xlabel("X1 Values")
   ax.set_ylabel("X2 Values")
   ax.set_zlabel("Output y")
   ax.set_xlim([-2, 2])
   ax.set_ylim([-2, 2])
   ax.set_zlim([-5, 5])
   fake2Dline = mpl.lines.Line2D(
       [0], [0], linestyle="none", c='r', marker='o')
   ax.legend([fake2Dline, sc], ['Regression surface',
```

```
                                'Training Data'], numpoints=1)
   plt.show()

thetas = np.array(thetas)
theta_for_each_c = np.column_stack(thetas)



# (i)(b)
count = 1
for k in enumerate(theta_for_each_c):
   plt.figure(2)
   plt.plot(c_values, k[1])
   plt.title("Comparing model parameter theta-" +
             str(count) + " under each value of C")
   plt.ylabel("Theta " + str(count))
   plt.xlabel("C Value")
   count = count + 1
   plt.show()
```

week3_kfold.py | Examining different values for k - KFold cross validation

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.preprocessing as prep
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from matplotlib.pyplot import errorbar
import math



dataframe = pd.read_csv("week3_data.csv")

X1 = dataframe.iloc[:, 0]
X2 = dataframe.iloc[:, 1]
X = np.column_stack((X1, X2))

y = dataframe.iloc[:, 2]

polynomial_features = prep.PolynomialFeatures(degree=5)
```

```
new_features = polynomial_features.fit_transform(X)

C = 1
split_values = [2, 5, 10, 25, 50, 100]
mean_array = []*len(split_values)
stan_dev_array = []*len(split_values)
for i, split in enumerate(split_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha = 1/2C

   print("--- SPLIT = ", split, " ---")
   kf = KFold(n_splits=split)
   loopCount = 0
   sum_errors = 0
   error_array = []*split
   for train, test in kf.split(new_features):

       loopCount = loopCount + 1
       lassoModel = linear_model.Lasso(alpha=alpha, random_state=0)
       lassoModel.fit(new_features[train], y[train])

       predictions = lassoModel.predict(new_features[test])
       mse = mean_squared_error(y[test], predictions)

       sum_errors = sum_errors + mse
       error_array.append(mse)
   mean = sum_errors / split
   variance = np.var(error_array)
   print("Mean: ", mean, "- Variance: ", variance)
   mean_array.append(mean)
   stan_dev_array.append(np.array(error_array).std())

plt.figure(1)
errorbar(split_values, mean_array, yerr=stan_dev_array)
plt.xlabel("K-Fold 'K' Value")
plt.ylabel("Mean Square Error")
plt.title("Mean Squared Error vs. K-Fold Value; C = 1")
plt.show()
```

week3_kfold_c.py | Using k-fold and examining different C values

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.preprocessing as prep
```

```
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from matplotlib.pyplot import errorbar
import math



dataframe = pd.read_csv("week3_data.csv")

X1 = dataframe.iloc[:, 0]
X2 = dataframe.iloc[:, 1]
X = np.column_stack((X1, X2))

y = dataframe.iloc[:, 2]

polynomial_features = prep.PolynomialFeatures(degree=5)
new_features = polynomial_features.fit_transform(X)

split = 5
c_values = [0.001, 0.01, 0.1, 1, 2, 5, 10, 25, 50, 100]
mean_array = []*len(c_values)
stan_dev_array = []*len(c_values)
for i, C in enumerate(c_values):
   alpha = 1/(2*C)  #  As specified in assignment notes, sklearn alpha = 1/2C

   print("--- C = ", C, " ---")
   kf = KFold(n_splits=split)
   loopCount = 0
   sum_errors = 0
   error_array = []*split
   for train, test in kf.split(new_features):

       loopCount = loopCount + 1
       ridgeModel = linear_model.Ridge(alpha=alpha, random_state=0)
       ridgeModel.fit(new_features[train], y[train])

       predictions = ridgeModel.predict(new_features[test])

       mse = mean_squared_error(y[test], predictions)

       sum_errors = sum_errors + mse
       error_array.append(mse)
```

```
    mean = sum_errors / split
    variance = np.var(error_array)
    print("Mean: ", mean, "- Variance: ", variance)
    mean_array.append(mean)
    stan_dev_array.append(np.array(error_array).std())

print("MINIMUM", np.min(mean_array))
plt.figure(1)
errorbar(c_values, mean_array, yerr=stan_dev_array)
plt.xlabel("Hyperparameter 'C' value")
plt.ylabel("Mean Square Error")
plt.xlim(-1, 100)
plt.title(
    "Mean Squared Error vs. Hyperparameter C Value; k-fold = 5; Ridge Regression")
plt.show()
```