CS7CS4 Machine Learning
Week 3 Assignment

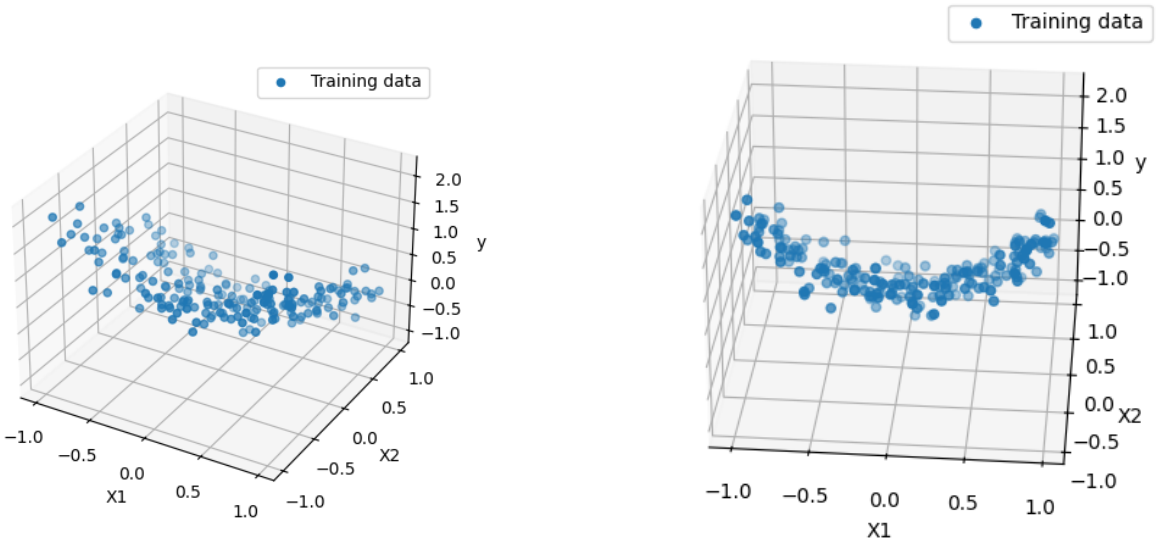
Michael Millard
Student ID: 24364218
Dataset ID: # id:15-15–15

October 12, 2024

Question (i)

(i)(a)

The dataset provided for this assignment contains two input features, $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$, and target values $\boldsymbol{y}$. A useful way to visualize the relationship between the input features and their corresponding target labels is to graph the dataset on a 3D scatter plot, as shown below in Figures 1a and 1b. Figure 1a provides a corner view of the scatter plot and Figure 1b provides a rotated view of the plot that gives the viewer a better sense of the shape of the data.



(a) Corner View

(b) Curve View

Figure 1: 3D scatter plot of input features $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$ against target labels $\boldsymbol{y}$

It is clear in Figure 1b above that this dataset appears to lie on a curve rather than a plane. From the 2D perspective of the view provided in Figure 1b, the dataset appears to lie on a convex parabola.

(i)(b)

Using the two input features provided in the original dataset, an additional 19 input features were created using various combinations of these two input features raised to powers up to degree 5. This was achieved using the sklearn **PolynomialFeatures** function and resulted in a new dataset called $\boldsymbol{X_{poly}}$, which contained 21 input features. This dataset was then used to train Lasso Regression models using the sklearn **Lasso** function with different values for the L1 penalty weight parameter **C** in each iteration. The range of **C** values used was chosen using trial and error in order to find a suitable range for the dataset. It was found that starting with $\mathbf{C} = 1$ was sufficient for the penalty parameter to eliminate all input feature coefficients and simply have only a non-zero intercept value. As such, the chosen range of values for the penalty weight parameter was $\mathbf{C} = [1, 5, 10, 100, 1000]$. For each value, a Lasso Regression

model was trained on the same $\boldsymbol{X_{poly}}$ dataset and the resulting 21 model parameters in each instance are tabulated below in Table 1.

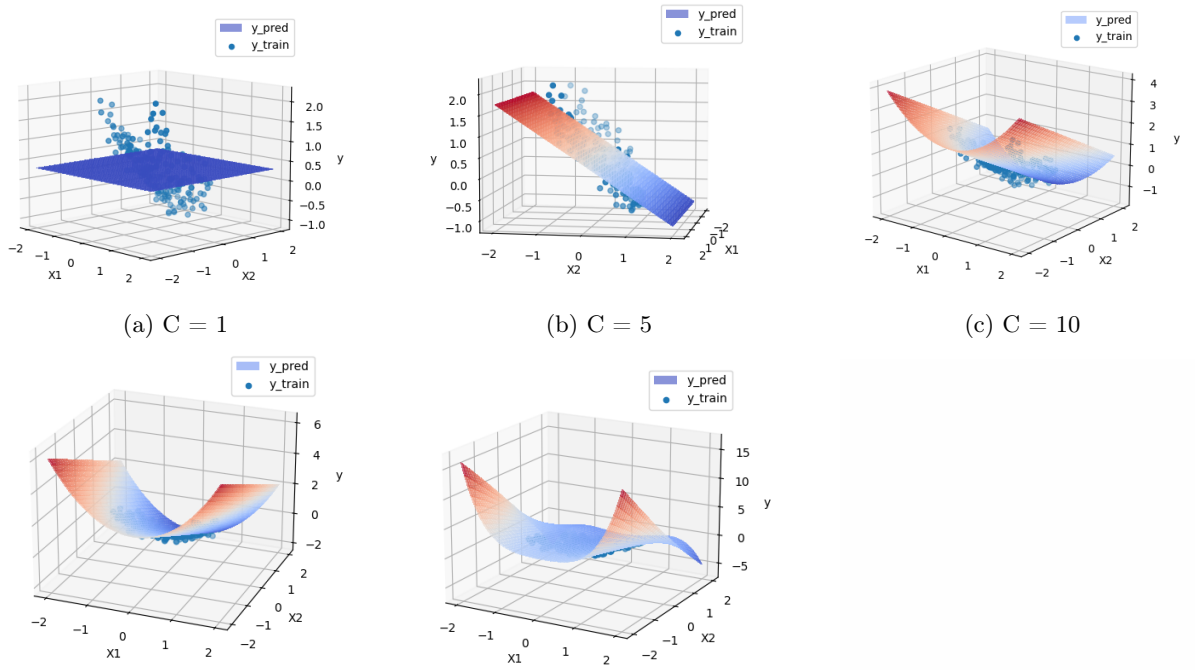
C	1	5	10	100	1000
θ_0	0.2978	0.3431	0.1715	0.0179	-0.0061
θ_1	0.0000	0.0000	0.0000	0.0000	0.0000
θ_2	-0.0000	-0.0000	0.0000	0.0000	0.0017
θ_3	-0.0000	-0.6964	-0.8418	-0.9726	-0.9840
θ_4	0.0000	0.0012	0.5045	0.9399	0.9150
θ_5	-0.0000	-0.0000	0.0000	0.0000	0.0287
θ_6	-0.0000	0.0000	0.0000	0.0031	0.0520
θ_7	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000
θ_8	-0.0000	-0.0000	-0.0000	-0.0000	0.1904
θ_9	0.0000	0.0000	0.0000	0.0000	0.0000
θ_{10}	-0.0000	-0.0000	-0.0000	-0.0000	0.0000
θ_{11}	0.0000	0.0000	0.0000	0.0199	0.0919
θ_{12}	-0.0000	-0.0000	-0.0000	0.0000	0.0000
θ_{13}	0.0000	0.0000	0.0000	0.0000	-0.0000
θ_{14}	-0.0000	-0.0000	-0.0000	0.0000	0.0090
θ_{15}	-0.0000	0.0000	0.0000	0.0000	-0.0084
θ_{16}	-0.0000	-0.0000	-0.0000	-0.0000	-0.0176
θ_{17}	-0.0000	-0.0000	-0.0000	-0.0000	-0.3214
θ_{18}	0.0000	0.0000	0.0000	0.0000	0.0229
θ_{19}	-0.0000	-0.0000	-0.0000	-0.0000	-0.0000
θ_{20}	0.0000	0.0000	0.0000	0.0000	-0.0000
θ_{21}	-0.0000	-0.0000	-0.0000	-0.0000	0.0000

Table 1: Lasso model parameters for various values of L1 penalty weight parameter C

From Table 1 above, it is evident that as **C** is increased, fewer of the Lasso model coefficients are eliminated (set to zero). This is because increasing **C** results in a smaller penalty factor in the Lasso model, resulting in more non-zero model coefficients. However, quite a few of the model parameters, such as θ_1, θ_7, θ_9, θ_{10}, θ_{12}, θ_{13}, θ_{19}, θ_{20} and θ_{21} are zero regardless of the value of **C**. Meaning the choice of the penalty weight parameter has no impact on the value of these coefficients, their corresponding input features will always be eliminated from the model. What is also noteworthy is that the intercept value θ_0, despite slighlty increasing from $\mathbf{C} = 1$ to $\mathbf{C} = 5$, gets closer to zero as **C** is increased beyond 5.

(i)(c)

For each of the Lasso Regression models trained using various values for the L1 penalty weight parameter **C**, predictions were generated on a test dataset comprising a grid of feature vectors. This test dataset was made by identifying the minimum and maximum values of the original two input features $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$, expanding their extremes by one unit to increase the range of the dataset, and creating 100 linearly spaced feature values between the new extremes for both features. These new features were then placed into a grid using nested for loops before using sklearn's **PolynomialFeatures** function to generate the feature vector values for the 19 other features in the dataset (see code in Appendix A). The predictions for each model were then plotted as 3D surfaces superimposed on a 3D scatter plot of the original training data, as shown in the sub-figures below.



(a) C = 1 (b) C = 5 (c) C = 10

(d) C = 100 (e) C = 1000

Figure 2: Predictions made by Lasso Regression models of varying penalty weight $\boldsymbol{C}$ plotted as 3D surfaces against a 3D scatter plot of the training data

It is evident in the sub-figures above that as the value of **C** increases, the complexity of the model fitted to the data increases in that the smaller penalty factor eliminates fewer of the input features as their corresponding model parameters θ_i are non-zero. From the plots and the knowledge that the training data appears lie on a quadratic curve, it appears that $\mathbf{C} = 10$ is the best penalty parameter in this range. Although $\mathbf{C} = 100$ appears almost identical, a smaller **C** value is desired as it is less likely to result in the model over-fitting to the training data. In Figure 2e above, where $\mathbf{C} = 1000$, it is evident that the model is over-fitted to the training data in that for feature values slightly outside the range of the training data it makes poor predictions, suggested by its complex shape.

(i)(d)

Under-fitting is when a model is too simple and as a result fails to adequately capture the behaviour of the training data. An example would be trying to use a linear model to make predictions on quadratic data. An under-fitted model tends to generalize poorly.

Over-fitting is when a model is too complex and fits the the training dataset too well, including the inherent noise in the dataset. An example would be fitting a high-degree polynomial to a linear dataset. An over-fitted model tends to perform poorly on unseen data.

From Table 1 and the sub-figures contained in Figure 2 above, it is clear that as the value of **C** increases, the L1 penalty factor decreases, resulting in fewer of the model coefficients in θ being zero, thereby incorporating more input features into the model and increasing its complexity. It is evident that when $\mathbf{C} = 1$, the model is linear (a plane for two input features) in that only θ_0 and θ_3 have non-zero values (θ_4 is negligibly small). This is not well suited to the quadratic dataset, which the Lasso models for $\mathbf{C} = 5$ and $\mathbf{C} = 10$ are better suited to with three non-zero model coefficients (θ_0, θ_3 and θ_4), which is consistent with their quadratic shapes above in sub-figures 2c and 2d, respectively. For $\mathbf{C} = \mathbf{1000}$, the Lasso model is over-fitted to the training data, resulting in a complex model (see shape in sub-figure 2e above), corresponding to more non-zero coefficients in θ (see final column in Table 1).

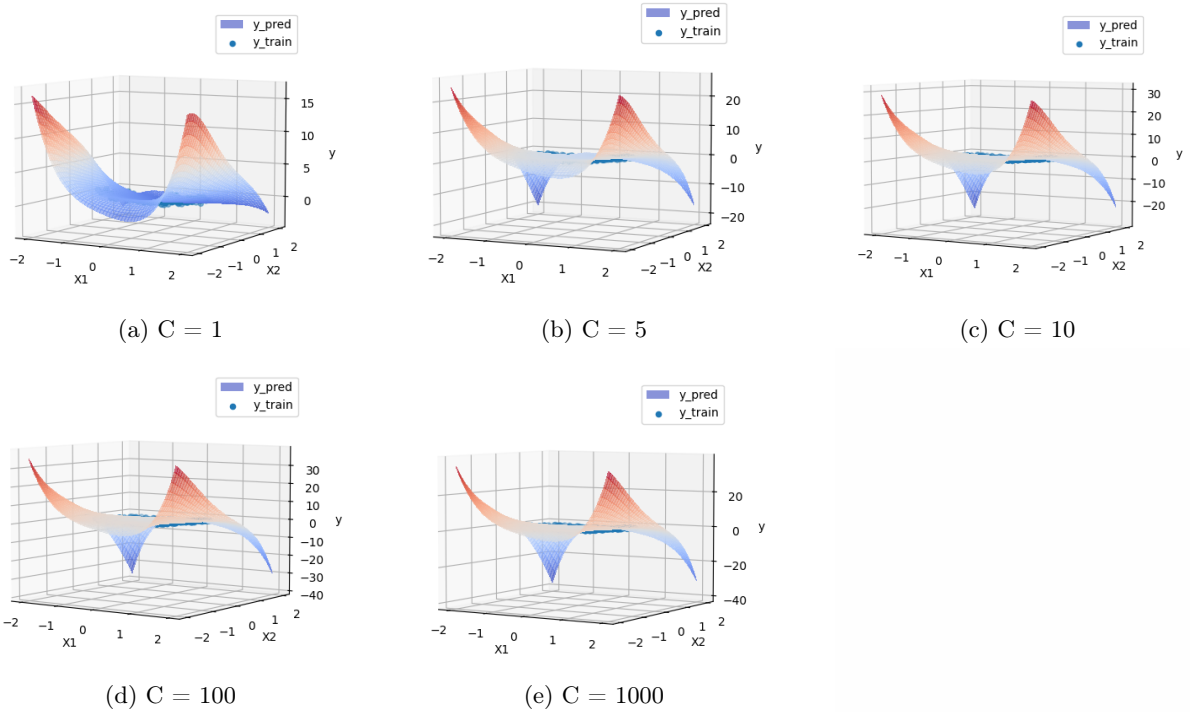
(i)(e)

Using the same training dataset comprising 21 input features created using sklearn's **PolynomialFeatures** function on the two original dataset features, five Ridge Regression models were trained by sweeping through the same range of penalty weight parameter values used for the Lasso Regression models above: $\mathbf{C} = [1, 5, 10, 100, 1000]$. The resulting 22 model parameters (21 input feature weights and one intercept) were recorded in Table 2 below.

C	1	5	10	100	1000
θ_0	0.0067	-0.0240	-0.0306	-0.0376	-0.0383
θ_1	0.0000	0.0000	0.0000	0.0000	0.0000
θ_2	0.0259	0.0437	0.0509	0.0613	0.0626
θ_3	-0.9459	-1.0451	-1.0802	-1.1313	-1.1382
θ_4	0.7455	0.8817	0.9098	0.9385	0.9416
θ_5	0.0298	0.0150	0.0059	-0.0084	-0.0103
θ_6	0.1248	0.2228	0.2475	0.2753	0.2785
θ_7	-0.0589	-0.1318	-0.1616	-0.2046	-0.2102
θ_8	0.1453	0.5489	0.7009	0.9122	0.9395
θ_9	0.0447	0.0700	0.0785	0.0898	0.0912
θ_{10}	-0.0835	0.0216	0.0733	0.1623	0.1756
θ_{11}	0.2639	0.1481	0.1238	0.0996	0.0971
θ_{12}	-0.0147	0.0003	0.0104	0.0259	0.0280
θ_{13}	-0.0012	-0.0798	-0.0997	-0.1237	-0.1267
θ_{14}	0.0396	0.0533	0.0602	0.0711	0.0726
θ_{15}	-0.0886	-0.1703	-0.1922	-0.2175	-0.2204
θ_{16}	-0.0132	0.0244	0.0426	0.0710	0.0749
θ_{17}	-0.2471	-0.5614	-0.6819	-0.8501	-0.8719
θ_{18}	0.1013	0.2045	0.2342	0.2685	0.2724
θ_{19}	-0.0653	-0.2463	-0.3169	-0.4159	-0.4288
θ_{20}	-0.0908	-0.1917	-0.2230	-0.2616	-0.2662
θ_{21}	0.0693	0.0569	0.0366	-0.0078	-0.0151

Table 2: Ridge model parameters for various values of penalty weight parameter C

It is immediately evident that Ridge Regression models (over this range of **C** values) are far more complex than Lasso Regression models with almost all model parameters in θ being non-zero. This can be attributed to the fact that Ridge Regression models use an L2 penalty factor (sum of squared θ elements), which encourages the elements of θ to be small rather than eliminated (equal to zero). The same grid test dataset used to analyse the performance of the Lasso Regression models above was used on the Ridge Regression models to see how well they generalize. The predictions made by each model are plotted as a 3D surface superimposed on a 3D scatter plot of the training data in the corresponding sub-figures below.



(a) C = 1 (b) C = 5 (c) C = 10

(d) C = 100 (e) C = 1000

Figure 3: Predictions made by Ridge Regression models of varying penalty weight ***C*** plotted as surfaces against a 3D scatter plot of the training data

As the value of **C** increases, the model parameters increase slightly, however, the amount by which these values change is far smaller than was seen in the case of the Lasso Regression models (at least for this range of **C** values). The most substantial change in model parameters is noticed in the change from **C** = 1 to **C** = 10 where the amount by which the model parameters increase is greater than from any other successive increments (see columns one and two in Table 2 above). This is consistent with the complexity of the model visualized in sub-figure 3a above, which is the least complex of all the trained Ridge Regression models, which appear almost identically complex in their respective sub-figures. This suggests that for Ridge Regression models, a smaller range of **C** values would yield better performing (less over-fitted) models than the range used for the Lasso Regression models.

Question (ii)

(ii)(a)

In order to identify the optimal value for **C** for a Lasso Regression model trained on the polynomial features dataset described in question (i), k-fold cross-validation was utilized. Again, the value of **C** was swept through the same range of values as before: **C** = [1, 5, 10, 100, 1000]. For each **C** value, the dataset was divided into 5 folds using sklearn's **KFold** function. Each fold was used as a validation set once with the 4 other folds used as the training dataset for that iteration. At each iteration, the mean squared error of the model on the validation fold was computed and subsequently used to compute the mean and standard deviation of the model prediction error at the end of the 5-fold cross-validation. This was performed for each value of **C** and the resulting error bar plot is shown below in Figure 4.

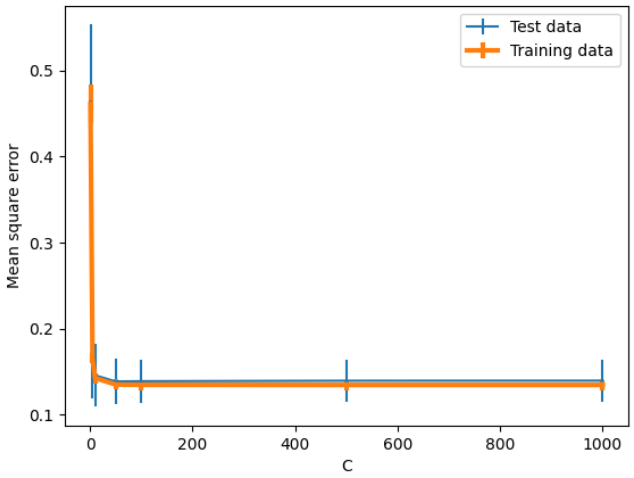


Figure 4: Error bar plot showing the mean and standard deviation of each Lasso model's prediction error

It is immediately evident that the range of **C** values is far too vast in that the mean and standard deviation of the prediction error decrease quickly for small values of **C** and remain approximately the same from **C** = 10 to **C** = 1000. As such, a new range of **C** values was chosen and the process was repeated. The new range was **C** = [1, 2, 4, 6, 8, 10, 16]. The predicted error mean and standard deviation for each of the Lasso models trained on the new range of **C** values are tabulated below in Table 3.

C	1	2	4	6	8	10	16
Mean Error	0.4637	0.3329	0.1862	0.1592	0.1498	0.1455	0.1408
Std Dev Error	0.0900	0.0903	0.0568	0.0456	0.0400	0.0367	0.0317

Table 3: The mean and standard deviation of the Lasso model prediction error for various values of **C**

The error bar plot corresponding to the mean and standard deviation prediction errors in Table 3 is shown below in Figure 5.

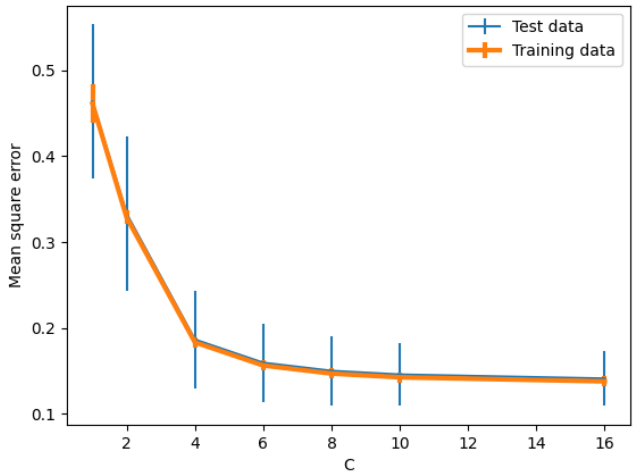


Figure 5: Error bar plot showing the mean and standard deviation of each Lasso model's prediction error

From Figure 5, it is evident that for small values of **C** (**C** < 4), the mean and standard deviation of the prediction error is relatively high in comparison to models trained with a higher penalty weight parameter. Once the value of **C** exceeds 8, the model error remains much the same, as is consistent with the initial error bar plot above in Figure 4 with a poor range of **C** values chosen (up to 1000).

(ii)(b)

Based on the results of 5-fold cross-validation shown in Table 3 and Figure 5 above, I would choose the penalty weight parameter value of **C** $= 8$. This is motivated by the fact that the prediction errors are greater for **C** values smaller than 8, and further motivated by the mean and standard deviation of the model's prediction error beyond this **C** value remaining much the same. Additionally, it is best to choose the smallest possible value of **C** in order to avoid the model over-fitting to the training set.

(ii)(c)

The steps outlined in questions (ii)(a) and (b) above were then repeated on Ridge Regression models. Again, the polynomial features dataset was used and initially the revised range of **C** values used for the Lasso models above was swept to train the various Ridge Regression models: **C** $= [1, 2, 4, 6, 8, 10, 16]$. 5-fold cross-validation was utilized in order to generate an error bar plot showing the mean and standard deviation of the prediction errors for each model. This plot is shown below in Figure 6.

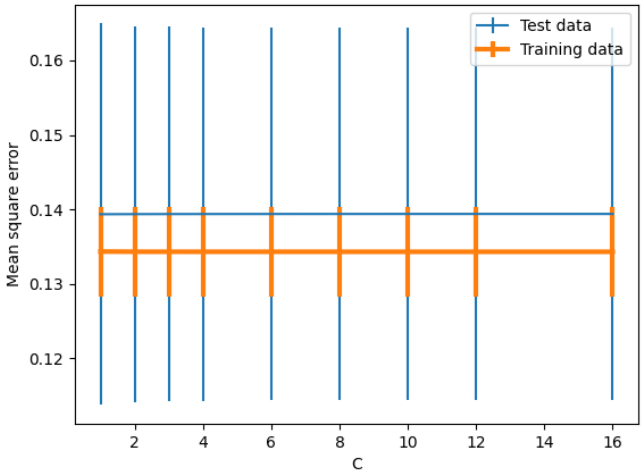


Figure 6: Error bar plot showing the mean and standard deviation of each Ridge model's prediction error

Again, it is immediately evident that this range of **C** values is poorly suited to this Ridge Regression problem on the current dataset (all mean and standard deviation values are the same in the plot). As such, a more suitable smaller range of **C** values was identified and used. The revised set was **C** = [0.01, 0.05, 0.1, 0.2, 0.4, 0.6, 1]. Using 5-fold cross-validation, the mean and standard deviation of the prediction errors for these new Ridge Regression models were computed and are tabulated below in Table 4.

C	0.01	0.05	0.1	0.2	0.4	0.6	1
Mean Error	0.2174	0.1472	0.1414	0.1398	0.1394	0.1394	0.1393
Std Dev Error	0.0614	0.0375	0.0316	0.0283	0.0266	0.0261	0.0256

Table 4: The mean and standard deviation of the Regression model prediction error for various values of **C**

These prediction error results above were also used to plot an error bar graph for these Ridge Regression models, which is shown below in Figure 7.

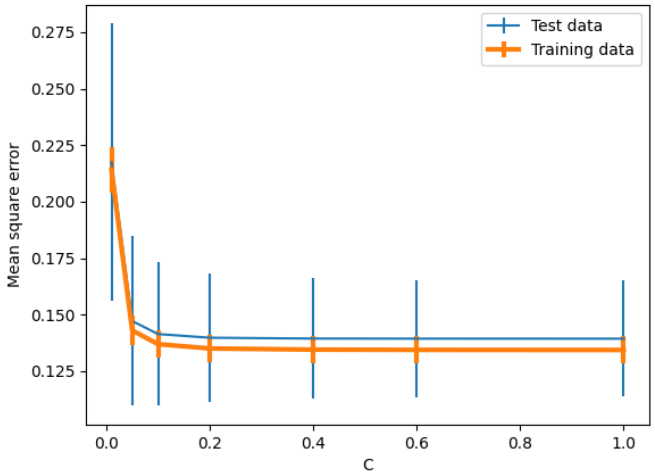


Figure 7: Error bar plot showing the mean and standard deviation of each Ridge model's prediction error

The mean and standard deviation of the Ridge Regression models' prediction errors got smaller very quickly as the value of **C** increased. Based on Figure 7 above, the mean and standard deviation of the error initially start high for very small values of **C** but converged to their final values at approximately **C** = 0.2.

Based on these results, I would choose the penalty weight parameter value **C** = 0.2 to train a Ridge Regression model on this dataset. Any **C** value lower than this results in a model with greater prediction errors and any **C** value greater than this results in a model with approximately the same prediction error mean and standard deviation. As such, since choosing the smallest possible **C** value is desirable to avoid model over-fitting, **C** = 0.2 is an ideal choice.

References

All Python code used in this assignment was based on snippets taken from the first two weeks of lectures provided for this course. This includes the use of the sklearn toolkit, the numpy and pandas libraries, as well as plotting using matplotlib.

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

I consent / do not consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

I agree that this thesis will not be publicly available, but will be available to TCD staff and students in the University's open access institutional repository on the Trinity domain only, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement. **Please consult with your supervisor on this last item before agreeing, and delete if you do not consent**

Signed: Michael Millard Date: 11/10/2024

A Question (i) Appendix

This appendix contains the code used to answer question (a) of the assignment.

```
# ---------------------------------------------------------- #
# Imports
# ---------------------------------------------------------- #

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from mpl_toolkits.mplot3d import Axes3D
from sklearn.metrics import accuracy_score

# ---------------------------------------------------------- #
# Read in data and set labels
# ---------------------------------------------------------- #

# NB: dataset csv file must be in same directory as this solution
labels = ["X1", "X2", "y"]
df = pd.read_csv("millardm_W3_dataset.csv", names=labels)
print("Dataframe head:")
print(df.head())

# Split data frame up into X and y
X1 = df["X1"].to_numpy()
X2 = df["X2"].to_numpy()
X = np.column_stack((X1, X2))
y = df["y"].to_numpy()

# ---------------------------------------------------------- #
# Question (i)(a)
# ---------------------------------------------------------- #

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X1, X2, y)

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('y')
ax.legend(['Training data'])

# Saving done manually
plt.show()

# ---------------------------------------------------------- #
# Question (i)(b)
# ---------------------------------------------------------- #

print("\nLasso:")

# Adding extra polynomial features (combos of all powers up to 5)
X_poly = PolynomialFeatures(5).fit_transform(X)

# Lasso model param arrays
```

```
lasso_models = []

# Range of C values to sweep
theta0s = []
thetas1to21 = []
C_range = [1, 5, 10, 100, 1000]
for C in C_range:
    # Create Lasso model and fit training data
    lasso_model = Lasso(alpha=1/(2*C)).fit(X_poly, y)
    lasso_models.append(lasso_model)

    # Extract params and print them out
    theta0 = lasso_model.intercept_.item()
    thetas = lasso_model.coef_.T
    theta0s.append(theta0)

    if (len(thetas1to21) == 0):
        thetas1to21 = thetas
    else:
        thetas1to21 = np.column_stack((thetas1to21, thetas))
    result_str = "C, theta0"
    for i in range(len(thetas)):
        result_str += ", theta{}".format(i + 1)
    result_str += " = {c} & {theta:.3f}".format(c=C, theta=theta0)
    for i in range(len(thetas)):
        result_str += " & {theta:.3f}".format(theta=thetas[i].item())
    print(result_str)

print(np.shape(theta0s))
print(np.shape(thetas1to21))

# These print statements are used to make Latex table creation easy
print("")
result_str = "C = "
for i in range(len(C_range)):
    result_str += " & {C}".format(C=C_range[i])
print(result_str)

result_str = "theta0 = "
for i in range(len(C_range)):
    result_str += " & {theta:.4f}".format(theta=theta0s[i])
print(result_str)

for i in range(np.shape(thetas1to21)[0]):
    result_str = "theta{} = ".format(i + 1)
    for j in range(np.shape(thetas1to21)[1]):
        result_str += " & {theta:.4f}".format(theta=thetas1to21[i][j])
    print(result_str)

# ---------------------------------------------------------- #
# Question (i)(c)
# ---------------------------------------------------------- #

# Creating grid of features extending beyond current dataset, need to do this for each f

# Find min and max values for X1 and X2
min_X1, max_X1 = np.min(X1), np.max(X1)
min_X2, max_X2 = np.min(X2), np.max(X2)

# Expand their ranges by 1 unit beyond either extreme and create a new column of values
num_pts = 100
```

```
X1_test = np.linspace(min_X1 - 1, max_X1 + 1, num_pts).reshape(-1, 1)
X2_test = np.linspace(min_X2 - 1, max_X2 + 1, num_pts).reshape(-1, 1)

# Make grid from test values
X_test = []
for i in X1_test:
    for j in X2_test:
        X_test.append([i, j])
X_test = np.array(X_test).reshape(-1, 2)
X1_test, X2_test = X_test[:, 0], X_test[:, 1]

# Create polynomial features for X_test
X_test_poly = PolynomialFeatures(5).fit_transform(X_test)

# Try first model
for i in range(len(lasso_models)):
    y_pred = lasso_models[i].predict(X_test_poly)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X1_test.reshape(num_pts, num_pts), X2_test.reshape(num_pts, num_pts)
    ax.scatter(X1, X2, y)

    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('y')
    ax.legend(["y_pred", "y_train"])

    # Saving done manually
    plt.show()

# ---------------------------------------------------- #
# Question (i)(d)
# ---------------------------------------------------- #

# Written answer

# ---------------------------------------------------- #
# Question (i)(e)
# ---------------------------------------------------- #

print("\nRidge:")

# Ridge model param arrays
ridge_models = []

# Sweep through same C range as Lasso
theta0s = []
thetas1to21 = []
for C in C_range:
    # Create Ridge model and fit training data
    ridge_model = Ridge(alpha=1/(2*C)).fit(X_poly, y)
    ridge_models.append(ridge_model)

    # Extract params and print them out
    theta0 = ridge_model.intercept_.item()
    thetas = ridge_model.coef_.T
    theta0s.append(theta0)

    if (len(thetas1to21) == 0):
        thetas1to21 = thetas
```

```
        else:
            thetas1to21 = np.column_stack((thetas1to21, thetas))
        result_str = "C, theta0"
        for i in range(len(thetas)):
            result_str += ", theta{}".format(i + 1)
        result_str += " = {c} & {theta:.3f}".format(c=C, theta=theta0)
        for i in range(len(thetas)):
            result_str += " & {theta:.3f}".format(theta=thetas[i].item())
        print(result_str)

print(np.shape(theta0s))
print(np.shape(thetas1to21))

# For easy Latex table populating
print("")
result_str = "C = "
for i in range(len(C_range)):
    result_str += " & {C}".format(C=C_range[i])
print(result_str)

result_str = "theta0 = "
for i in range(len(C_range)):
    result_str += " & {theta:.4f}".format(theta=theta0s[i])
print(result_str)

for i in range(np.shape(thetas1to21)[0]):
    result_str = "theta{} = ".format(i + 1)
    for j in range(np.shape(thetas1to21)[1]):
        result_str += " & {theta:.4f}".format(theta=thetas1to21[i][j])
    print(result_str)

# Use same grids defined above (independent of model type)

# Try first model
for i in range(len(ridge_models)):
    y_pred = ridge_models[i].predict(X_test_poly)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X1_test.reshape(num_pts, num_pts), X2_test.reshape(num_pts, num_pts)
    ax.scatter(X1, X2, y)

    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('y')
    ax.legend(["y_pred", "y_train"])

    # Saving done manually
    plt.show()
```

B Question (ii) Appendix

This appendix contains the code used to answer question (ii) of the assignment.

```
# ---------------------------------------------------------- #
# Question (ii)(a)
# ---------------------------------------------------------- #

# Test arrays
mean_error_test=[]
```

```
std_error_test=[]

# Train arrays
mean_error_train=[]
std_error_train=[]

# Smaller range for C taken here
C_range = [1, 2, 4, 6, 8, 10, 16]
for C in C_range:
    # Create Lasso model
    model = Lasso(alpha=1/(2*C))

    # Temp arrays for storing MSE in each fold below
    temp_test=[]
    temp_train=[]

    # K-fold cross validation with 5 splits
    k_fold = KFold(n_splits=5)
    for train, test in k_fold.split(X):
        # Fit model to training data for this fold
        model.fit(X[train], y[train])

        # Validate on test data for this fold
        y_pred_test = model.predict(X[test])
        temp_test.append(mean_squared_error(y[test], y_pred_test))

        # Performance on training data for this fold
        y_pred_train = model.predict(X[train])
        temp_train.append(mean_squared_error(y[train], y_pred_train))

    # Append errors to their respective arrays
    mean_error_test.append(np.array(temp_test).mean())
    std_error_test.append(np.array(temp_test).std())
    mean_error_train.append(np.array(temp_train).mean())
    std_error_train.append(np.array(temp_train).std())

# Generate error bar plot
plt.errorbar(C_range, mean_error_test, yerr=std_error_test)
plt.errorbar(C_range, mean_error_train, yerr=std_error_train, linewidth=3)
plt.xlabel('C')
plt.ylabel('Mean square error')
plt.legend(['Test data', 'Training data'])
plt.savefig("error_bar_lasso_ii_a.png")
plt.show()

# These print statements are used to make Latex table creation easy
print("")
result_str = "C = "
for i in range(len(C_range)):
    result_str += " & {C}".format(C=C_range[i])
print(result_str)

result_str = "mean_error = "
for i in range(len(C_range)):
    result_str += " & {err:.4f}".format(err=mean_error_test[i])
print(result_str)

result_str = "std_error = "
for i in range(len(C_range)):
    result_str += " & {std:.4f}".format(std=std_error_test[i])
print(result_str)
```

```
# ---------------------------------------------------------- #
# Question (ii)(b)
# ---------------------------------------------------------- #

# Written answer

# ---------------------------------------------------------- #
# Question (ii)(c)
# ---------------------------------------------------------- #

# Test arrays
mean_error_test=[]
std_error_test=[]

# Train arrays
mean_error_train=[]
std_error_train=[]

# Need to change range of C values again for Ridge
C_range = [0.01, 0.05, 0.1, 0.2, 0.4, 0.6, 1]
for C in C_range:
    # Create Ridge model
    model = Ridge(alpha=1/(2*C))

    # Temp arrays for storing MSE in each fold below
    temp_test=[]
    temp_train=[]

    # K-fold cross validation with 5 splits
    k_fold = KFold(n_splits=5)
    for train, test in k_fold.split(X):
        # Fit model to training data for this fold
        model.fit(X[train], y[train])

        # Validate on test data for this fold
        y_pred_test = model.predict(X[test])
        temp_test.append(mean_squared_error(y[test], y_pred_test))

        # Performance on training data for this fold
        y_pred_train = model.predict(X[train])
        temp_train.append(mean_squared_error(y[train], y_pred_train))

    # Append errors to their respective arrays
    mean_error_test.append(np.array(temp_test).mean())
    std_error_test.append(np.array(temp_test).std())
    mean_error_train.append(np.array(temp_train).mean())
    std_error_train.append(np.array(temp_train).std())

# Generate error bar plot
plt.errorbar(C_range, mean_error_test, yerr=std_error_test)
plt.errorbar(C_range, mean_error_train, yerr=std_error_train, linewidth=3)
plt.xlabel('C')
plt.ylabel('Mean square error')
plt.legend(['Test data', 'Training data'])
plt.savefig("error_bar_ridge_ii_c.png")
plt.show()

# These print statements are used to make Latex table creation easy
print("")
result_str = "C = "
```

```
for i in range(len(C_range)):
    result_str += " & {C}".format(C=C_range[i])
print(result_str)

result_str = "mean_error = "
for i in range(len(C_range)):
    result_str += " & {err:.4f}".format(err=mean_error_test[i])
print(result_str)

result_str = "std_error = "
for i in range(len(C_range)):
    result_str += " & {std:.4f}".format(std=std_error_test[i])
print(result_str)

# ---------------------------------------------------------- #
# END OF ASSIGNMENT
# ---------------------------------------------------------- #
```

—— END OF ASSIGNMENT ——