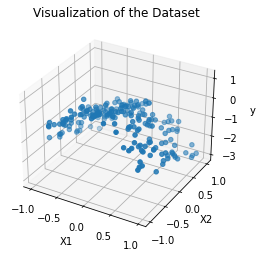
Week 3 homework

(i).

(a). The following 3D scatter plot visualizes the dataset:

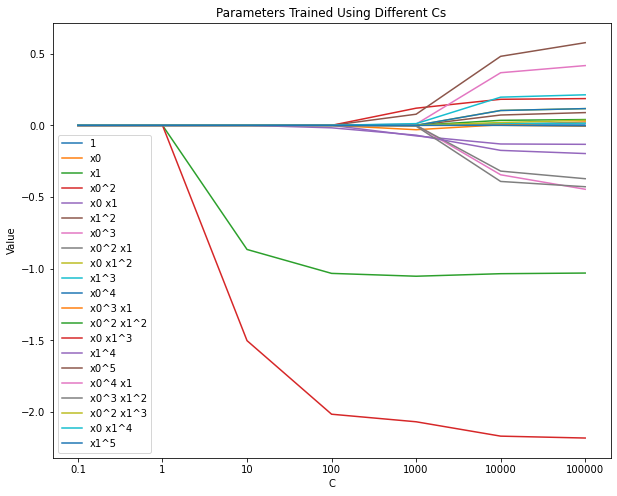


The points look like lying on a curve.

Code:

1. *# Import related packages*
2. import matplotlib.pyplot as plt
3. from mpl\_toolkits.mplot3d import Axes3D
4. import numpy as np
5. from sklearn.preprocessing import PolynomialFeatures
6. from sklearn import linear\_model
7. from sklearn.model\_selection import KFold
8. from sklearn.metrics import mean\_squared\_error
9. *# Data id:14--28--14*
10. *# Question 1*
11. *# (a)*
12. *# Load the data*
13. file = open("week3.txt", "r")
14. data = [line.strip() for line in file.readlines()]
15. data = [line.split(",") for line in data]
16. *# Convert to numpy array*
17. data = np.array(data).astype(float)
18. *# Capture the Xs and ys*
19. X = data[:, :2]
20. y = data[:, 2]
21. *# Plot the graph*
22. fig = plt.figure()
23. ax = fig.add\_subplot(111 , projection ='3d')
24. ax.scatter(X[:, 0], X[:, 1], y)
25. ax.set\_title("Visualization of the Dataset")
26. ax.set\_xlabel("X1")
27. ax.set\_ylabel("X2")
28. ax.set\_zlabel("y")
29. plt.show()

(b). For reproduction, we fixed the random state to 0, and the list of C used for the experiment is [0.1, 1, 10, 100, 1000, 10000, 100000]. For better visualization, we generated a plot to find out the trend of the trained parameters.

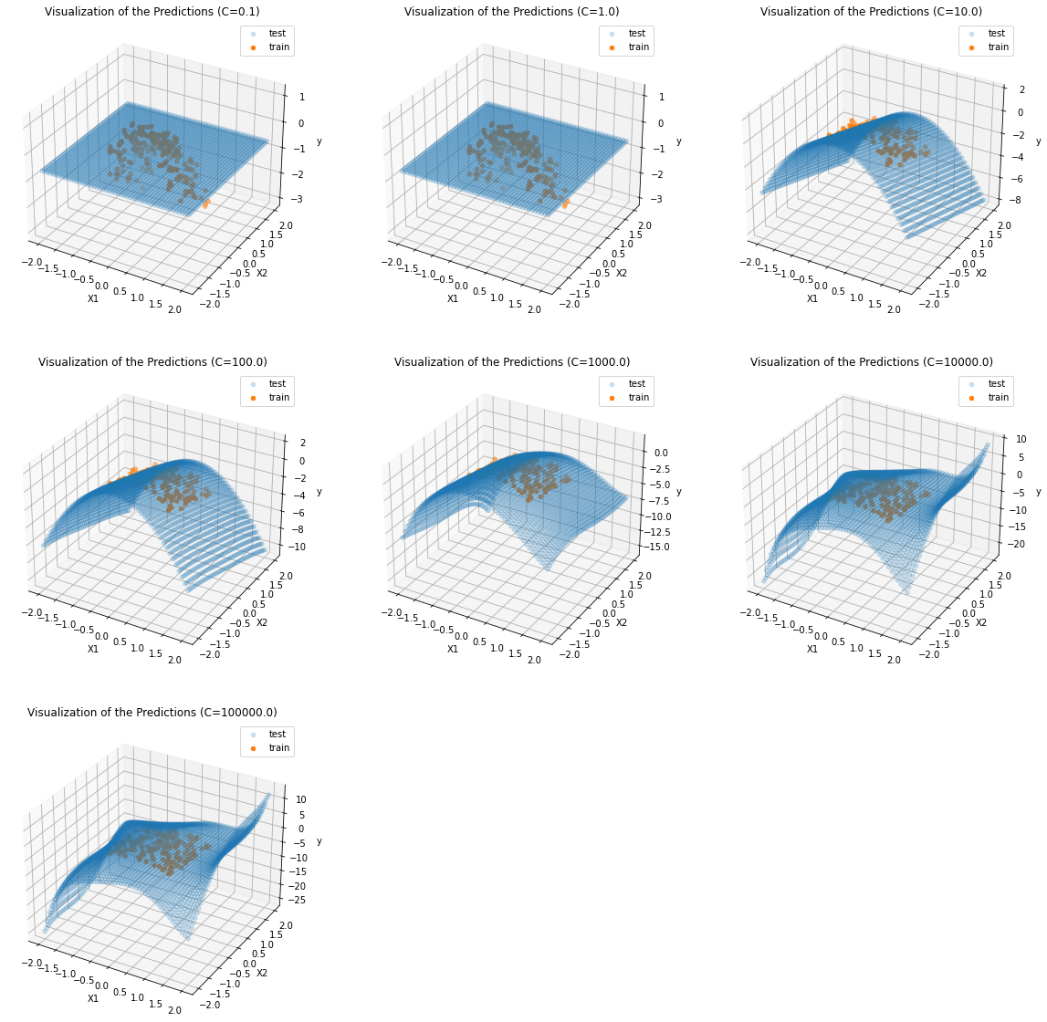


The plot shows that the parameter set is no longer an array of zeros when C is larger than 1. As the value of C increases, the value of alpha decreases, which indicates a decrease in the strength of the penalty. We can notice that there are two parameters varied from 0 when C increased to 1, and the corresponding variables are and . The other parameters remained to be 0 until C is larger than 100. Lasso regression restricts the redundant parameters, therefore the value of the parameters would have larger differences as the value of C increases.

Code:

1. *# (b)*
2. *# Create new features*
3. poly = PolynomialFeatures(5)
4. poly\_X = poly.fit\_transform(X)
5. names = poly.get\_feature\_names()
6. *# Train a list of models*
7. parameters = []
8. c\_list = [0.1, 1, 10, 100, 1000, 10000, 100000]
9. for c in c\_list:
10. *# Transfer c to alpha*
11. alpha = 1/(2\*c)
12. clf = linear\_model.Lasso(random\_state=0, alpha=alpha)
13. clf.fit(poly\_X, y)
14. parameters.append(clf.coef\_)
16. parameters = np.array(parameters).T
17. *# Visualization*
18. plt.figure(figsize=[10, 8])
19. plt.title("Parameters Trained Using Different Cs")
20. for i in range(len(parameters)):
21. plt.plot(np.arange(len(c\_list)), parameters[i], label=names[i])
22. plt.xlabel("C")
23. plt.xticks(np.arange(len(c\_list)), c\_list)
24. plt.ylabel("Value")
25. plt.legend()
26. plt.show()

(c). The 7 plots illustrate the predictions provided by different models.



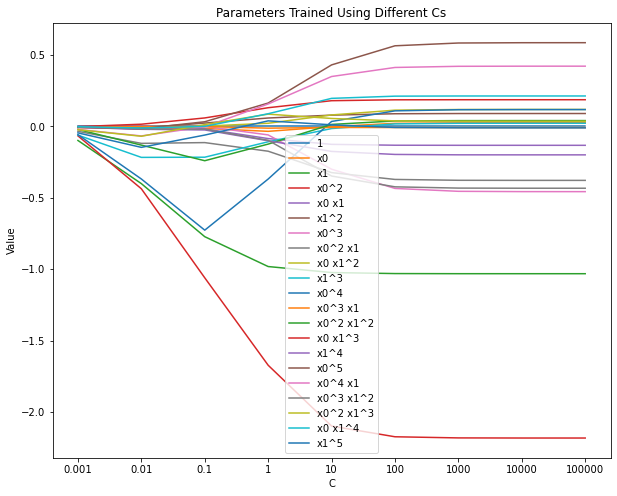
By using the test grid, a prediction surface is generated. The training samples proffer our knowledge about the real labels, therefore a good fit is equal to the training samples lying on the prediction surface. For better visualization, we generated a grid from -2 to 2. We are expecting a curvy surface, and from the graph, we can assert setting C to 1000 gives us a good fit. On the other hand, when C is less than 1, the model failed to fit. Another piece of evidence is that all the parameters are 0 when C is too small. However, when C is too large, the surface becomes too curvy that it may lose generalizability.

Code:

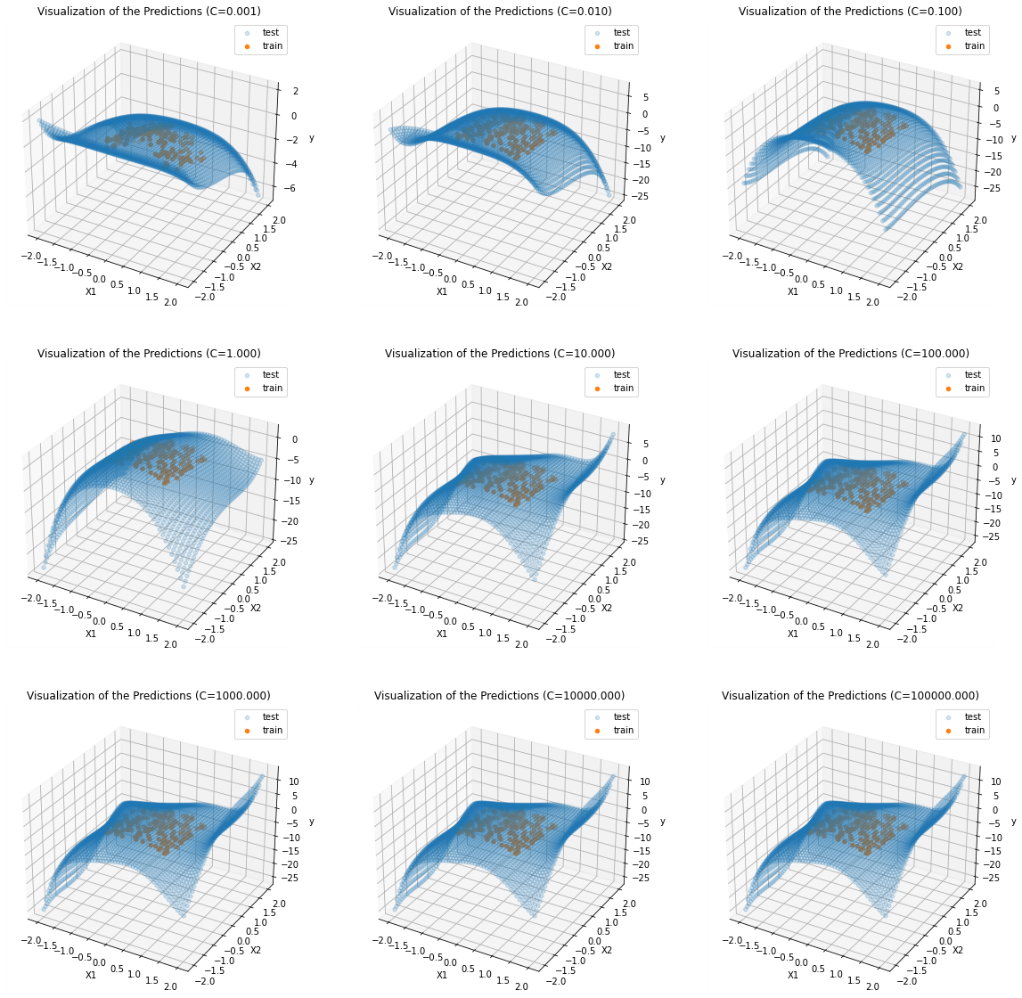
1. *# (c)*
2. *# Generate a grid*
3. Xtest =[ ]
4. grid =np.linspace(-2, 2)
5. for i in grid:
6. for j in grid:
7. Xtest.append([i, j])
8. Xtest = np.array(Xtest)
9. *# Generate the polynomial features*
10. poly\_Xtest = poly.fit\_transform(Xtest)
11. *# Train a list of models and make predictions*
12. c\_list = [0.1, 1, 10, 100, 1000, 10000, 100000]
13. predictions = []
14. for c in c\_list:
15. *# Transfer c to alpha*
16. alpha = 1/(2\*c)
17. clf = linear\_model.Lasso(random\_state=0, alpha=alpha)
18. clf.fit(poly\_X, y)
19. predictions.append(clf.predict(poly\_Xtest))
20. *# Generate the graphs*
21. fig = plt.figure(figsize=[20, 20])
22. for i in range(len(c\_list)):
23. ax = fig.add\_subplot(3, 3, i+1, projection ='3d')
24. *# Predictions*
25. ax.scatter(Xtest[:, 0], Xtest[:, 1], predictions[i], alpha=0.2, label="test")
26. *# Training samples*
27. ax.scatter(X[:, 0], X[:, 1], y, label="train")
28. ax.set\_title("Visualization of the Predictions (C=%.1f)"%c\_list[i])
29. ax.set\_xlabel("X1")
30. ax.set\_ylabel("X2")
31. ax.set\_zlabel("y")
32. plt.legend()
33. plt.show()

(d). Under-fitting means a model failed to fit on the training set, as it obtains little knowledge about the samples. The first two subplots in (c) show cases of under-fitting. Conversely, over-fitting indicates the model learned too much knowledge about the training set, thus it can only predict the specific set and has poor generalizability on the testing sets. For instance, the last two subplots in the last problem are visualizations for over-fitting. In other words, under-fitting has low variance but high bias, and results in a simple model; over-fitting has a low bias but high variance and results in a very complex model. Either model is unsatisfying. To obtain a good fit, the strength of restriction shall be moderate. Setting C to around 1000 could provide us with a good fit for this project.

(e). The first graph shows the value of parameters as C varies. It can be observed that when applying ridge regression, the parameter set will not be an array of zeros even if C is very small, and every parameter are expanding when the value of C increases. Moreover, the value of parameters changes more smoothly. The similarity between the parameter sets trained by the two approaches is that they share a similar overall evolution trend.



The second graph contains subplots showing the prediction surfaces. A good fit is reached when C is larger than 1 but less than 10.

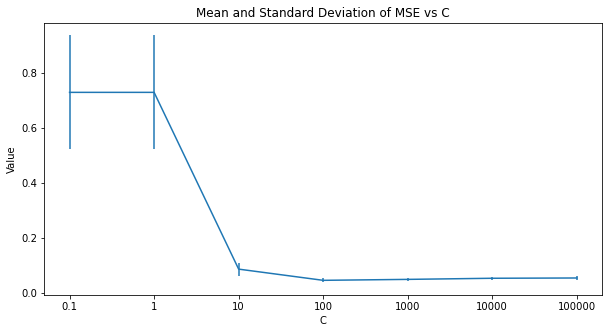


Code:

1. *# (e)*
2. *# Train a list of models, capture the parameters and predictions*
3. parameters = []
4. predictions = []
5. c\_list = [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
6. for c in c\_list:
7. *# Transfer c to alpha*
8. alpha = 1/(2\*c)
9. clf = linear\_model.Ridge(random\_state=0, alpha=alpha)
10. clf.fit(poly\_X, y)
11. parameters.append(clf.coef\_)
12. predictions.append(clf.predict(poly\_Xtest))
14. parameters = np.array(parameters).T
15. *# Visualization (parameters)*
16. plt.figure(figsize=[10, 8])
17. plt.title("Parameters Trained Using Different Cs")
18. for i in range(len(parameters)):
19. plt.plot(np.arange(len(c\_list)), parameters[i], label=names[i])
20. plt.xlabel("C")
21. plt.xticks(np.arange(len(c\_list)), c\_list)
22. plt.ylabel("Value")
23. plt.legend()
24. plt.show()
25. *# Generate the graphs (predictions)*
26. fig = plt.figure(figsize=[20, 20])
27. for i in range(len(c\_list)):
28. ax = fig.add\_subplot(3, 3, i+1, projection ='3d')
29. *# Predictions*
30. ax.scatter(Xtest[:, 0], Xtest[:, 1], predictions[i], alpha=0.2, label="test")
31. *# Training samples*
32. ax.scatter(X[:, 0], X[:, 1], y, label="train")
33. ax.set\_title("Visualization of the Predictions (C=%.3f)"%c\_list[i])
34. ax.set\_xlabel("X1")
35. ax.set\_ylabel("X2")
36. ax.set\_zlabel("y")
37. plt.legend()
38. plt.show()

(ii).

(a). We stuck to the previous C list for better comparison. With the assistant of the generated graph, we can see that the mean error has a descending trend. The standard deviation also shrinks as C increases. However, the mean error value suffers a slight rise when C is larger than 100.



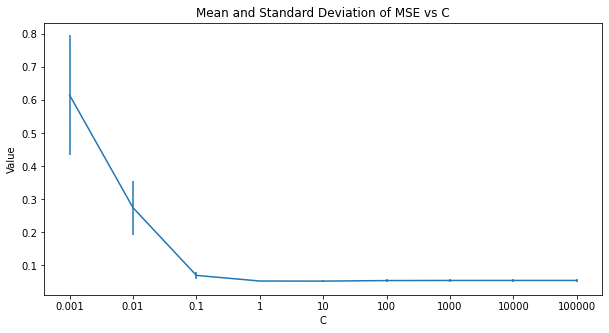
Code:

1. *# Question 2*
2. *# (a)*
3. *# A list of C*
4. c\_list = [0.1, 1, 10, 100, 1000, 10000, 100000]
5. *# Create k folds*
6. kf = KFold(5, shuffle=True, random\_state=0)
7. datasets = [dataset for dataset in kf.split(poly\_X)]
8. *# Test on different Cs*
9. means = []
10. stds = []
11. for c in c\_list:
12. alpha = 1/(2\*c)
13. model = linear\_model.Lasso(random\_state=0, alpha=alpha)
14. errors = []
15. for [train\_indices, test\_indices] in datasets:
16. *# Fit the model*
17. model.fit(poly\_X[train\_indices], y[train\_indices])
18. *# Make predictions*
19. ypred = model.predict(poly\_X[test\_indices])
20. *# Calculate MSE*
21. errors.append(mean\_squared\_error(y[test\_indices], ypred))
22. means.append(np.array(errors).mean())
23. stds.append(np.array(errors).std())
25. *# Plot the graph*
26. plt.figure(figsize=[10, 5])
27. plt.errorbar(np.arange(len(c\_list)), means, stds)
28. plt.xticks(np.arange(len(c\_list)), c\_list)
29. plt.xlabel("C")
30. plt.ylabel("Value")
31. plt.title("Mean and Standard Deviation of MSE vs C")
32. plt.show()

(b). The mean value shows the average level of the error, while the standard deviation indicates the overall correctness in the prediction. If the standard deviation is large, then there may exist several outrageous predictions. On the other hand, a small standard deviation means the model has an average error level on every sample. Thus we should find a model which has a balanced mean and standard deviation of the prediction error on an independent test set to promise good performances.

As what was shown in the last question, the minimum mean error was reached at C=100, and the standard deviation calculated on the test set is also small. In this case, we would recommend 100 as the final value of C.

(c). Similarly, we tested on the previous C list using ridge regression. Using the standard we mentioned in the last problem, 1 shall be chosen as the final value of C, as it shows the least mean and standard deviation of the prediction error.



Code:

1. *# (c)*
2. *# A list of C*
3. c\_list = [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000]
4. *# Create k folds*
5. kf = KFold(5, shuffle=True, random\_state=0)
6. datasets = [dataset for dataset in kf.split(poly\_X)]
7. *# Test on different Cs*
8. means = []
9. stds = []
10. for c in c\_list:
11. alpha = 1/(2\*c)
12. model = linear\_model.Ridge(random\_state=0, alpha=alpha)
13. errors = []
14. for [train\_indices, test\_indices] in datasets:
15. *# Fit the model*
16. model.fit(poly\_X[train\_indices], y[train\_indices])
17. *# Make predictions*
18. ypred = model.predict(poly\_X[test\_indices])
19. *# Calculate MSE*
20. errors.append(mean\_squared\_error(y[test\_indices], ypred))
21. means.append(np.array(errors).mean())
22. stds.append(np.array(errors).std())
24. *# Plot the graph*
25. plt.figure(figsize=[10, 5])
26. plt.errorbar(np.arange(len(c\_list)), means, stds)
27. plt.xticks(np.arange(len(c\_list)), c\_list)
28. plt.xlabel("C")
29. plt.ylabel("Value")
30. plt.title("Mean and Standard Deviation of MSE vs C")
31. plt.show()