Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

CS7CS4 Machine Learning Week 3 Assignment

Boris Flesch
20300025

October 30, 2020

MSc Computer Science, Intelligent Systems

1 Downloaded Dataset

id:22-22-22

2 Questions

(i) (a) The training data from the CSV file has been plotted on Figures 1 and 2 below with features $X1$, $X2$ and output y respectively on x, y and z axis. These two figures only differ in angle of view in order to better visualise the behaviour of the data. Basic Python code with scatter function and elev/azim view parameters has been used to plot this data on a 3D graph:

```
figViews = [[10,10], [40,80]]
for figView in figViews:
    fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], y)
    ax.view_init(elev=figView[0], azim=figView[1])
    ax.set(title='Training data from CSV', xlabel='X1',
    ↪  ylabel='X2', zlabel='Y')
    plt.show()
```

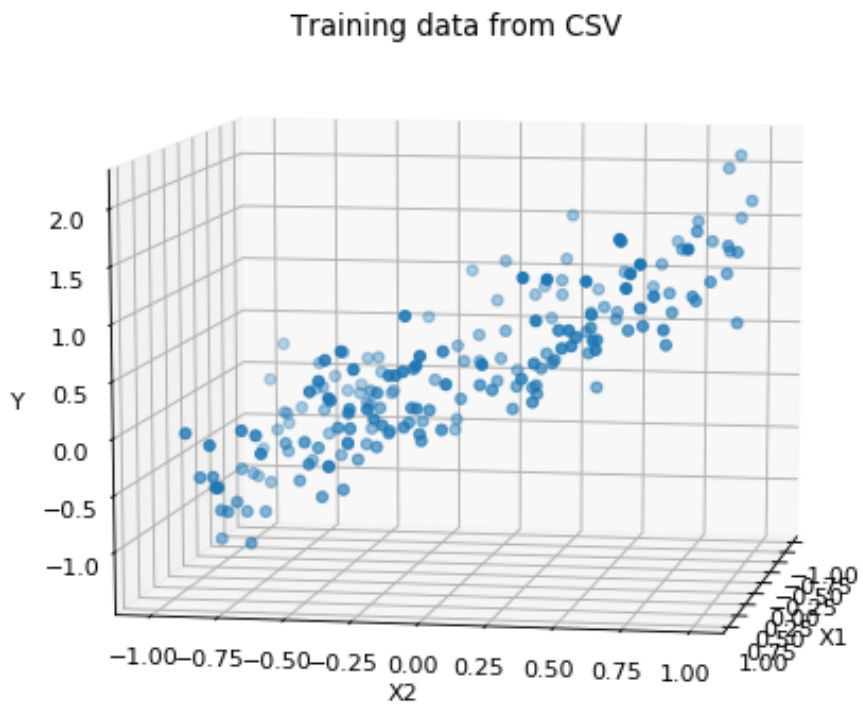

Figure 1

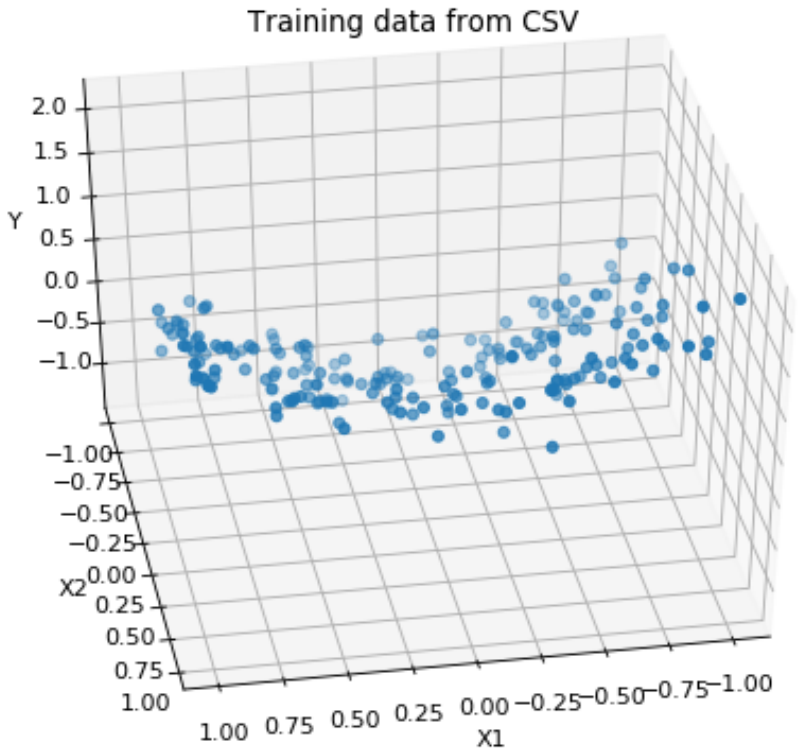

Figure 2

These graphs (especially Figure 2) show us that the data lies on a curve. More precisely, it seems to have a quadratic behaviour as it forms the shape of a parabolic cylinder.

(b) PolynomialFeatures function from sklearn allow us to easily add combinations of powers of the two features up to power 5. Looking at the shape of *Xpoly*, we now have 21 features based on the two original ones.

```
Xpoly = PolynomialFeatures(5).fit_transform(X)
print(Xpoly.shape)
```

Training a Lasso regression with $C = 1$ is small enough to make all the trained model parameter values zero. We can then choose a wide range of values for C (i.e. 1, 10, 100, 1000) to evaluate the parameters of these different models. The following code also calculates the mean squared error (MSE) $J(\theta)$ for each trained model as well as for a baseline model for comparison. This baseline model uses a mean strategy that will be sufficient to follow the inclination of the training data without adopting its behaviour (i.e. the baseline model will generate predictions on a plan rather than on a curved surface).

```
baseline = DummyRegressor(strategy="mean").fit(Xpoly, y)
print("J(theta_baseline) = %f\n"%mean_squared_error(y,
    ↪  baseline.predict(Xpoly)))

C_range = [1, 10, 100, 1000]
for Ci in C_range:
```

```
    model = Lasso(alpha=1/(2*Ci)).fit(Xpoly, y)
    theta = np.insert(model.coef_, 0, model.intercept_)

    print("C = %.1f"%Ci)
    print("theta =", theta)
    print("J(theta) = %f\n"%mean_squared_error(y,
     ↪  model.predict(Xpoly)))
```

The following values have been obtained for the trained models. Note: θ parameters are displayed inline for convenience.

- $J(\theta_{baseline}) = 0.484777$
- For $C = 1$:

$$\theta = \begin{bmatrix} 0.30588102 & 0. & 0. & 0. & 0. & -0. & -0. & 0. & 0. & 0. \\ 0. & 0. & -0. & 0. & -0. & -0. & 0. & 0. & 0. & 0. \\ -0. & 0. & & & & & & & & \end{bmatrix} \tag{1}$$

$$J(\theta) = 0.484777 \tag{2}$$

- For $C = 10$:

$$\theta = \begin{bmatrix} 0.21615007 & 0 & -0 & 0.89527586 & 0.40870961 & -0 & 0 & \\ -0 & 0 & -0 & 0 & 0 & 0 & 0 & -0 \\ 0 & -0 & 0 & -0 & 0 & -0 & 0 & \end{bmatrix} \tag{3}$$

$$J(\theta) = 0.072529 \tag{4}$$

- For $C = 100$:

$$\theta = \begin{bmatrix} 0.05029264 & 0 & -0.02725094 & 0.98245705 & 0.88667204 & \\ -0 & 0 & -0 & -0 & -0 & \\ 0.06677662 & 0 & . - 0 & 0 & -0 & 0 \\ -0 & -0 & -0 & -0 & -0 & 0 \end{bmatrix} \tag{5}$$

$$J(\theta) = 0.038530 \tag{6}$$

- For $C = 1000$:

$$\theta = \begin{bmatrix} 0.03958891 & 0. & -0.00053159 & 0.98549638 \\ 0.84334171 & -0. & 0.00554992 & -0. \\ -0.09025114 & -0.04253571 & 0.14056466 & 0.11699791 \\ 0.05131235 & -0. & -0.08900364 & 0. \\ -0.06474921 & 0. & -0. & -0.03461868 \\ 0. - 0. & & & \end{bmatrix} \tag{7}$$

$$J(\theta) = 0.036759 \tag{8}$$

Lasso regression model uses a L1 penalty which encourages sparsity of solution (i.e. as many elements as possible of theta are 0): bigger is C, less elements of theta parameter are 0.

We can see this behaviour in the trained models. Indeed, when C is small (i.e. $C = 1$), L1 penalty is bigger. Therefore all values of parameter theta are penalised, leading to only zeros (except for the intercept θ_0). Such a model will therefore produce predictions on a plan, which is not appropriate for our curved training data. This also explains why the same MSE value is obtained for both trained and baseline models (i.e. $J(\theta_{C=1}) = J(\theta_{baseline})$).

Bigger is C, smaller is the penalty and greater are the values of parameter theta. Example: for $C = 10$, only θ_0, θ_3 and θ_4 have non-zero values. Therefore, the model will only rely on features $X2$ and $X3 = X1^2$. Thus predictions will follow a quadratic shape (i.e. a curved surface), which is more appropriate than a plan regarding our training data. Values of $C > 10$ that may imply more features and the model could lead to cubic or even quartic (i.e. fifth degree polynomial) behaviour. In other terms: over-fitting.

The comparison with the baseline model also shows some possibility of over-fitting for $C \geq 100$ regarding the very small value of the MSE obtained ($J(\theta_{C \geq 100}) << J(\theta_{baseline})$).

It will be interesting to evaluate these models using cross-evaluation (see (ii)(b)) in order to find the most appropriate value for this hyperparameter C (i.e. the best trade-off between under- and over-fitting).

(c) Features of the dataset have values from -1 to 1. Therefore a range from -5 to 5 seems appropriate for our prediction grid to observe the behaviour of the model outside of the training data scope while still being able to identify training data on the graph. The code given in the assignment sheet has been used to generate this grid. Then, as it has been done for training data, we need to apply PolynomialFeatures to this grid:

```
Xtest = PolynomialFeatures(5).fit_transform(Xtest)
```

We can then generate predictions with Lasso models for different values of C and plot them along with the training data using a surface. The matplotlib function plot_trisurf is quite straightforward to do that. Adding some transparency to the surface also allows the training data to be more clearly visible.

```
C_range = [1, 10, 100, 1000]
for Ci in C_range:
    model = Lasso(alpha=1/(2*Ci))
    model.fit(Xpoly, y)
    fig = plt.figure(num=None, figsize=(10, 9), dpi=120)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], y, c='g', label="Training data")
    surf = ax.plot_trisurf(Xtest[:,1], Xtest[:,2],
     ↪  model.predict(Xtest), cmap=cm.coolwarm, alpha=0.8,
     ↪  linewidth=0, antialiased=True)
    # ...
```

The following Figures 3 to 6 have been obtained for $C \in \{1, 10, 100, 1000\}$ (i.e. same range of values as for (i)(b)):

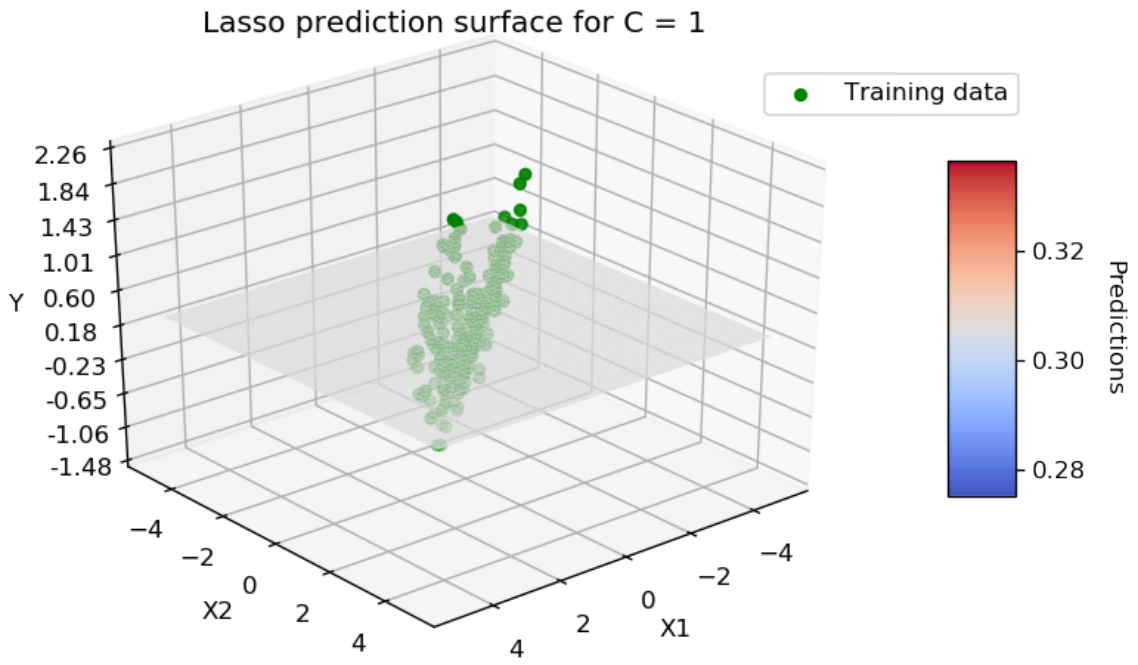

Figure 3

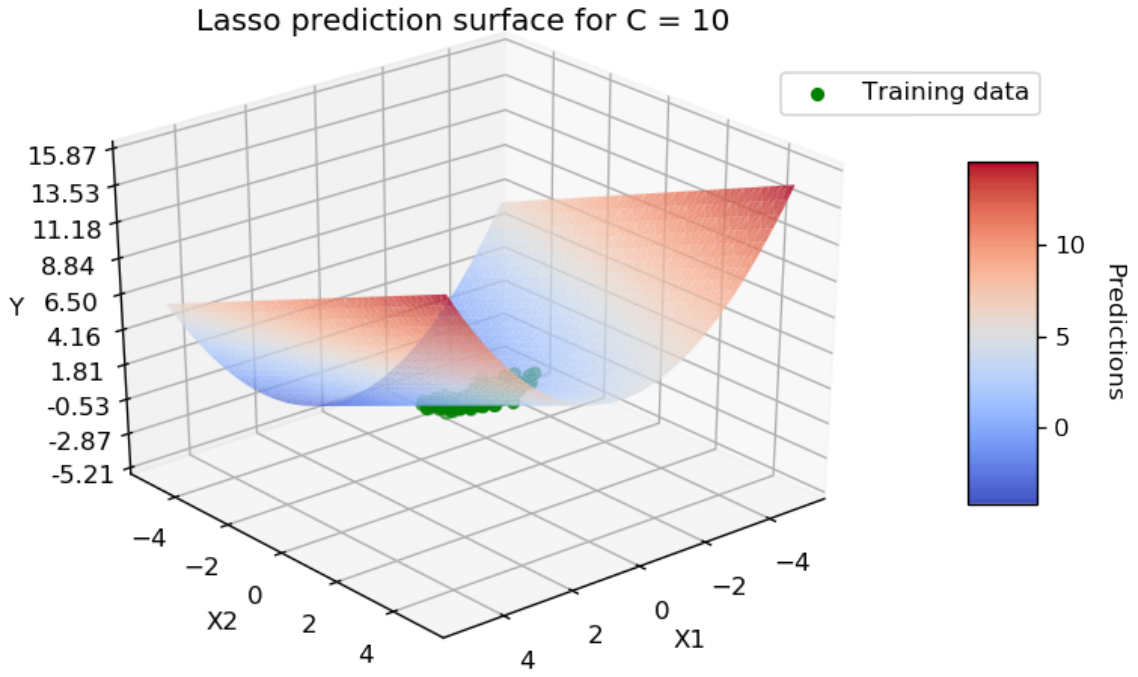

Figure 4

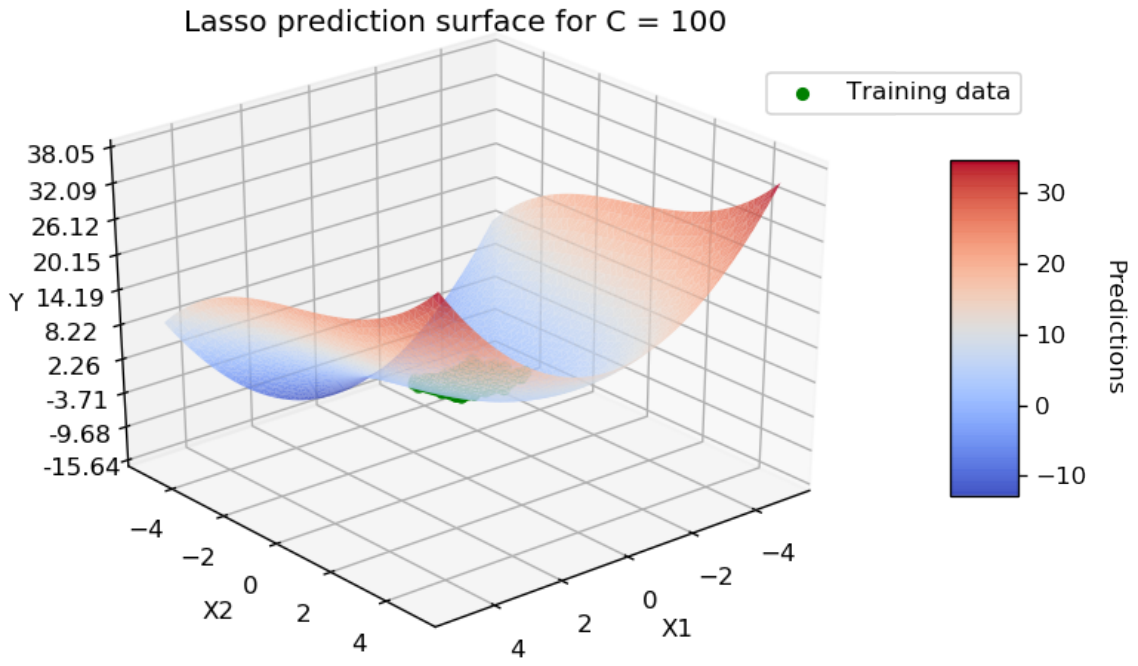

Figure 5

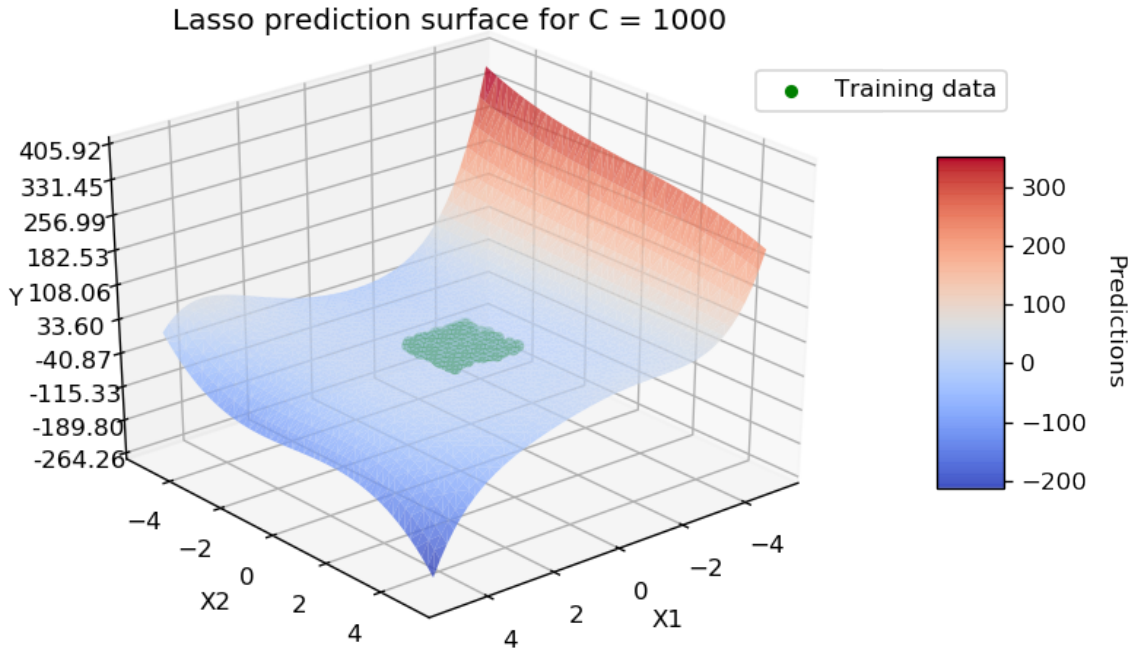

Figure 6

We can see from these models that bigger is C, more complex becomes the prediction surface; starting from a simple plan when $C = 1$ (Figure 3).

These graphs actually give us a clear visualisation of discussion made about the models obtained in question (i)(b). The surface prediction that follows the best the quadratic curved shape of our training data is for $C = 10$ (Figure 4) whereas $C = 1$

(Figure 3) and $C \geq 100$ (Figures 5 and 6) look respectively under- and over-fitted (i.e. not following the behaviour of our data).

(d) We can speak of under-fitting when the model is too simple to follow the behaviour of our data, thus leading to poor predictions. In this case, the training data has a quadratic (i.e. curved) shape: using a flat prediction surface for this data (see Figure 3) is an example of under-fitting.

On the contrary, over-fitting means that the model is taking too much into account small variations (i.e. noise) of our training data; also leading to poor predictions. An example of over-fitting can be seen on Figure 6 as our model is adopting a behaviour that really differs from the shape of our training data.

The hyperparameter C is definitely a great option to manage to trade-off between under- and over-fitting as it allows us to neglect some high-degree features of our model (i.e. X^5), thus adopting a quadratic behaviour from an initially over-fitted model with features up to degree 5.

(e) We can repeat the same steps as for questions (i)(b) and (i)(c) just by changing the model to Ridge Regression and using a slightly different range of C values. In this case, we could use $C \in \{1e{-}7, 1e{-}5, 1e{-}3, 1e{-}1, 1\}$. This range of values is extremely wide and focuses on very small values of C, but it allows us to clearly observe the evolution of trained parameters values regarding C for this training data.

As for Lasso models previously trained, the same baseline model has been kept and MSE has been calculated for each Ridge trained model:

- $J(\theta_{baseline}) = 0.484777$
- For $C = 1e{-}7$:

$$\theta = \begin{bmatrix} 0.30588038 & 0. & 0.00000033 & 0.00001363 \\ 0.00000377 & -0.00000101 & -0.00000047 & 0.00000068 \\ 0.0000049 & 0. & 0.00000836 & 0.00000363 \\ -0.00000019 & 0.00000124 & -0.00000093 & -0.00000043 \\ 0.00000068 & 0.00000319 & 0.00000026 & 0.000003 \\ -0.00000023 & 0.00000603 & & \end{bmatrix} \tag{9}$$

$$J(\theta) = 0.484759 \tag{10}$$

- For $C = 1e{-}5$:

$$\theta = \begin{bmatrix} 0.30581636 & 0. & 0.0000331 & 0.00135965 \\ 0.00037688 & -0.00010065 & -0.00004674 & 0.00006804 \\ 0.00048856 & 0.0000003 & 0.0008337 & 0.00036258 \\ -0.00001891 & 0.00012426 & -0.00009279 & -0.00004302 \\ 0.00006727 & 0.00031862 & 0.0000254 & 0.00029869 \\ -0.00002344 & 0.00060134 & & \end{bmatrix} \tag{11}$$

$$J(\theta) = 0.482928 \tag{12}$$

- For $C = 1e{-}3$:

$$\theta = \begin{bmatrix} 0.29793073 & 0. & 0.00093547 & 0.11049274 \\ 0.03447571 & -0.00714102 & -0.0033799 & 0.00442536 \\ 0.03847374 & -0.0009021 & 0.0668203 & 0.03273728 \\ -0.00083506 & 0.01149555 & -0.00669593 & -0.00308427 \\ 0.00463128 & 0.02487681 & 0.00134991 & 0.02313784 \\ -0.00263697 & 0.04777963 & & \end{bmatrix} \tag{13}$$

$$J(\theta) = 0.348615 \tag{14}$$

- For $C = 1e{-}1$:

$$\theta = \begin{bmatrix} 0.11174315 & 0. & -0.01646859 & 0.75864952 \\ 0.46698809 & -0.02054737 & -0.01464167 & -0.00474842 \\ 0.05806601 & -0.02107957 & 0.26852808 & 0.36041744 \\ 0.03017832 & 0.09457987 & -0.02361904 & -0.00841336 \\ -0.02567778 & 0.00467599 & -0.02190878 & -0.04241076 \\ -0.01583203 & 0.09000128 & & \end{bmatrix} \tag{15}$$

$$J(\theta) = 0.042156 \tag{16}$$

- For $C = 1$:

$$\theta = \begin{bmatrix} 0.05722872 & 0. & -0.01884234 & 0.91883503 \\ 0.68630335 & -0.02087815 & 0.02316895 & 0.0660602 \\ -0.07795242 & -0.09550289 & 0.32178673 & 0.28106702 \\ 0.08913239 & -0.00062012 & -0.10691076 & -0.02246509 \\ -0.11273962 & 0.06226974 & -0.0172614 & -0.14191772 \\ 0.0663384 & -0.11038582 & & \end{bmatrix} \tag{17}$$

$$J(\theta) = 0.036521 \tag{18}$$

Ridge regression model uses a L2 penalty (i.e. quadratic penalty) that encourages small parameters values. The hyperparameter C allows to alter the impact of this penalty: bigger is C, smaller is the penalty and bigger are the parameters values (thus leading to a more complex model).

As an example, for $C = 1e{-}7$, many parameter values are very close to zero. Thus, such a model will lead to a prediction surface that is almost flat, as is the baseline model. The MSE of this model confirms its proximity with the baseline model: $J(\theta_{C=1e-7}) \approx J(\theta_{baseline})$.

On the contrary, when C has a greater value (e.g. $C = 1$), all parameter values are bigger. That leads to models with very small MSE. Such models capture slight variations of our training data and are likely to be over-fitted. That is the case for $C \geq 1e{-}1$ in the above trained models.

The following Figures 7 to 11 show predictions surfaces of Ridge Regression models on a grid from range -5 to 5 in order to have an overview of the behaviour of our models further than the scope of the training data:

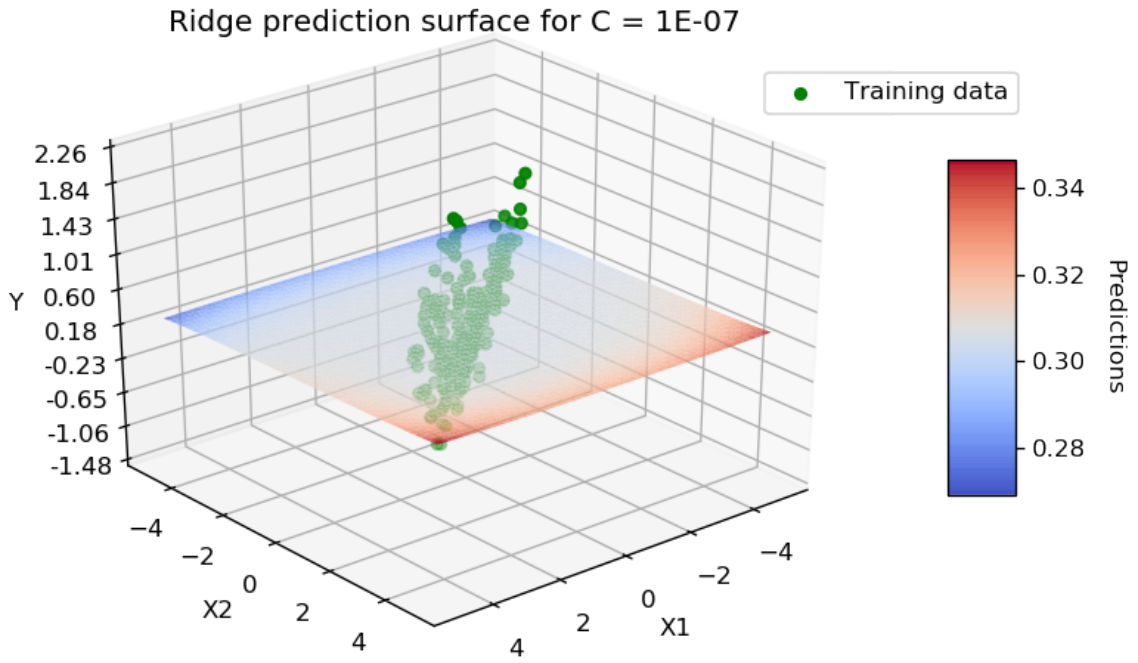

Figure 7

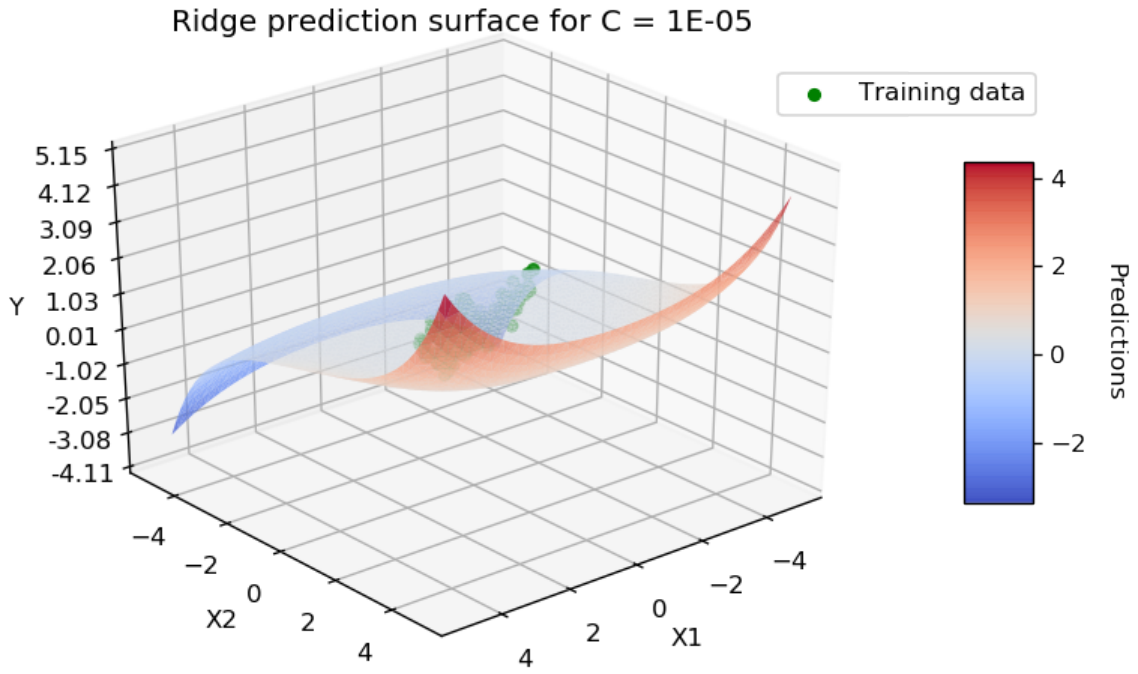

Figure 8

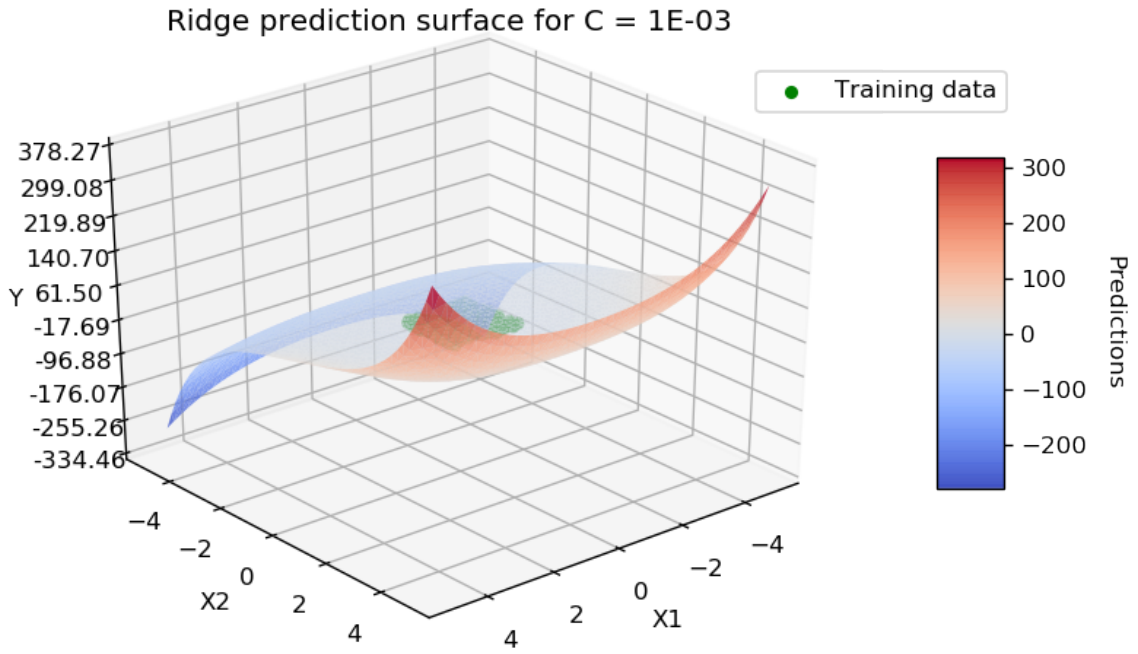

Figure 9

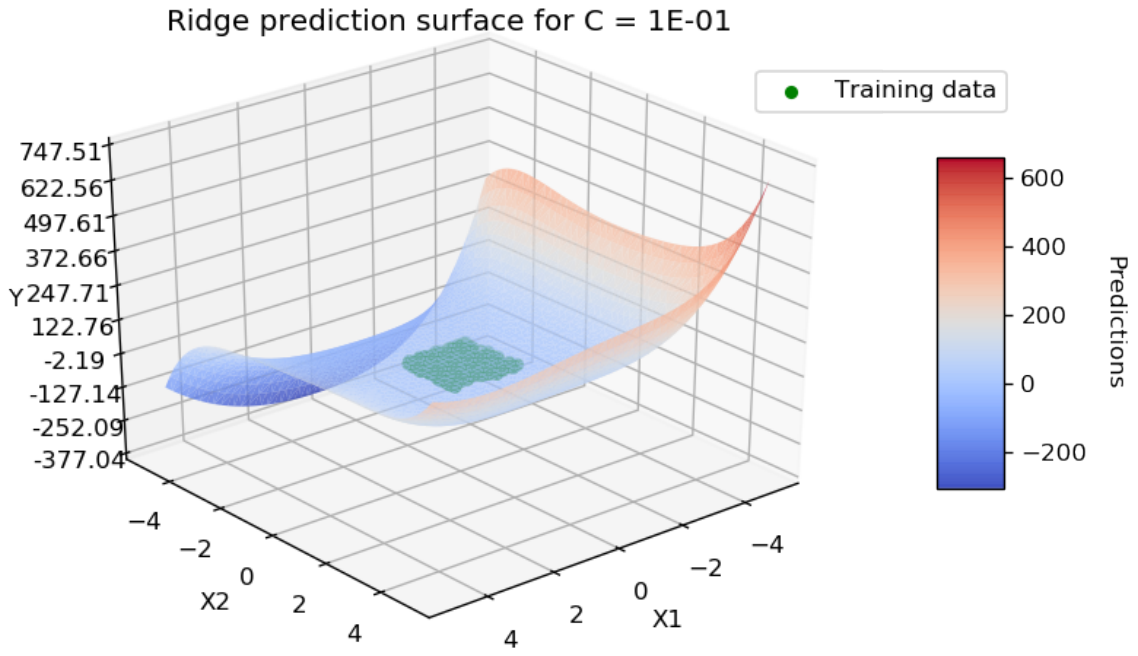

Figure 10

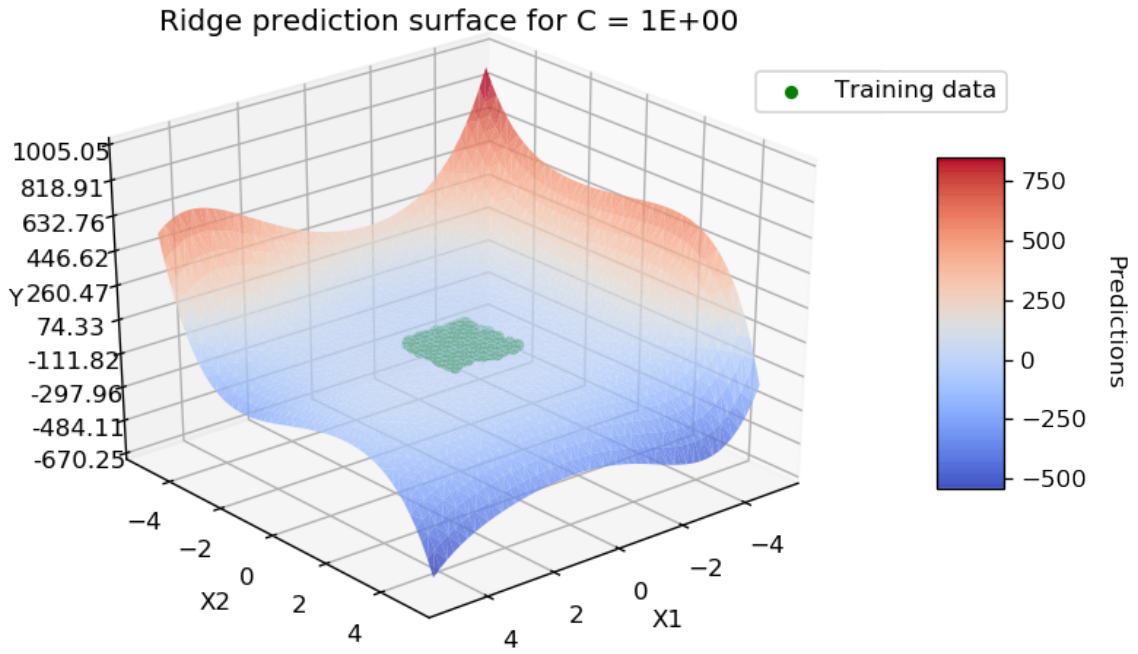

Figure 11

These plots allow us to visualise the trained models parameters previously discussed. As expected, the surface obtained for $C = 1e{-}7$ is nearly flat, whereas other surfaces are curved. Bigger is C, more complex is the surface obtained. We can also note that the range of predictions increases along with C value: for $C = 1e{-}5$, predictions roughly go from -3 to 4 whereas they go from -500 to 750 for $C = 1$.

Generally, the fact that all parameters are systematically implied in these models (even with small values) makes them complex (especially when compared to Lasso Regression models). Hence they are hardly following the behaviour of our training data, particularly when C increases.

Finally, we can see that the impact on the model parameters of changing C with Lasso Regression and with Ridge Regression highly differs:

- In Lasso model, changing C allows to radically change the model's behaviour by cancelling (i.e. putting to zero) some of its parameter values. This allows to easily switch from a quintic behaviour to a quadratic or even linear one for example.
- In Ridge model, on the contrary, changing hyperparameter C only allows to decrease or increase the parameter values of the model without cancelling any of them. Hence in this case it seems difficult to find an appropriate tune for C so that the Ridge model fits our data very well.

We can therefore argue that Lasso Regression offers a more flexible approach than Ridge Regression as tuning C helps fitting training data more easily by "filtering" parameter values. Thus, even when the initial input features would lead to an over-fitted model as it seems to be the case with our input features up to degree 5, finding an appropriate C value can lead to a well-fitted model.

(ii) (a) Using Lasso model from (i) with $C = 1$, the following code allows us to perform 5-fold cross-validation and get both mean square prediction error and standard deviation:

```
model = Lasso(alpha=1/2) # <=> C = 1
kf = KFold(n_splits=5)
temp = []
for train, test in kf.split(Xpoly):
    model.fit(Xpoly[train], y[train])
    ypred = model.predict(Xpoly[test])
    temp.append(mean_squared_error(y[test], ypred))

print("Mean error = %f; Standard deviation =
↪  %f"%(np.array(temp).mean(), np.array(temp).std()))
```

The following results are obtained:

Mean error = 0.488389; Standard deviation = 0.146137

The MSE calculated here is close to the one obtained in (i)(b) without k-fold cross-validation for the same model (0.484777). The small value for standard deviation also indicates that the error did not much vary through the different folds that have been tested.

To choose the most appropriate number of folds to use, we can use the following code to calculate the mean error and standard deviation of a range of values for k:

```
model = Lasso(alpha=1/2) # <=> C = 1
k = [2, 5, 10, 25, 50, 100]
std_error = []
mean_error = []
for ki in k:
    kf = KFold(n_splits=ki)
    temp = []
    for train, test in kf.split(Xpoly):
        model.fit(Xpoly[train], y[train])
        ypred = model.predict(Xpoly[test])
        temp.append(mean_squared_error(y[test], ypred))

    mean_error.append(np.array(temp).mean())
    std_error.append(np.array(temp).std())

plt.errorbar(k, mean_error, yerr=std_error, linewidth=3)
# ...
```

This code gives us the following graph:

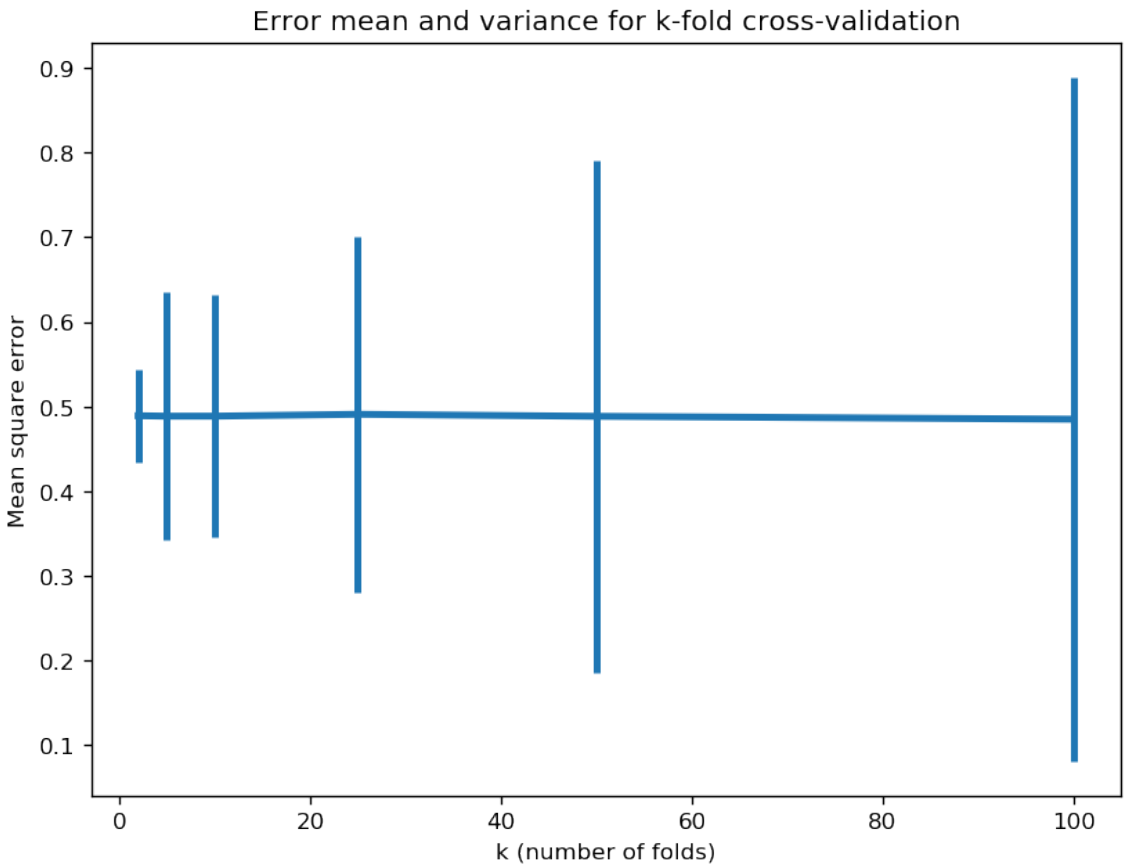

Figure 12

When calculating the accuracy of predictions using test data, the main reason why the accuracy may fluctuate is either because the training data is noisy or the test data is noisy. The idea of taking the mean of the squared error and the standard deviation through k-fold cross-validation is to smooth out this noise.

In one hand, if the test data is noisy, we want to use a small value for k so that each test dataset is large enough (i.e. n/k is large enough) to smooth out the prediction noise. In the other hand, if the training data is noisy, we want to have a big value for k so that fluctuations in the model are not due to noise in the training data. Also bigger is k, more important is the computation time (as our model will be trained k times).

Given that our original data does not seem very noisy (see Figure 1 and 2), we do not have to seek for an extreme value for k, either very big or small. We could therefore go for a common value like 5 or 10 (i.e. respectively 80/20 or 90/10 split). Considering Figure 12, we want to choose a value for k that minimises the MSE and/or the standard deviation. As $k = 10$ seems to provide a slightly smaller standard deviation than $k = 5$ and a similar MSE, I would recommend using $k = 10$ for this data.

(b) Choosing the right value for C can be done by testing a range of values with cross-validation. In this case, we can use 10-fold as it seems to be the best number of folds for this data (see previous question). Regarding the range of values for C, we

know from (i)(b) and (i)(c) that $C = 1$ gives an under-fitted model and $C \geq 100$ seems to lead to over-fitting. We could therefore focus on values between 1 and 100 by choosing $C \in \{1, 5, 10, 50, 100\}$.

The following algorithm will allow us to use 10-fold cross-validation and to plot MSE and standard deviation for every value of C, for both training and test data:

```
stdErrorTest, meanErrorTest, stdErrorTrain, meanErrorTrain = [], [], [], []
plt.figure(num=None, figsize=(8, 6), dpi=120)

C_range = [1, 5, 10, 50, 100]
kf = KFold(n_splits=10)

for Ci in C_range:
    model = Lasso(alpha=1/(2*Ci))
    tempTest, tempTrain = [], []

    for train, test in kf.split(Xpoly):
        model.fit(Xpoly[train], y[train])
        ypred = model.predict(Xpoly[test])
        ypred_train = model.predict(Xpoly[train])

        tempTest.append(mean_squared_error(y[test], ypred))
        tempTrain.append(mean_squared_error(y[train], ypred_train))

    meanErrorTest.append(np.array(tempTest).mean())
    stdErrorTest.append(np.array(tempTest).std())
    meanErrorTrain.append(np.array(tempTrain).mean())
    stdErrorTrain.append(np.array(tempTrain).std())

plt.errorbar(C_range, meanErrorTest, yerr=stdErrorTest, linewidth=3, c="blue", label="Test data")
plt.errorbar(C_range, meanErrorTrain, yerr=stdErrorTrain, linewidth=3, c="orange", label="Training data")
# ...
```

(c) This cross-validation for different values of C gives the following results with C values on x-axis, MSE on y-axis and error bars indicating the standard deviation for each C value (for both test and training data):

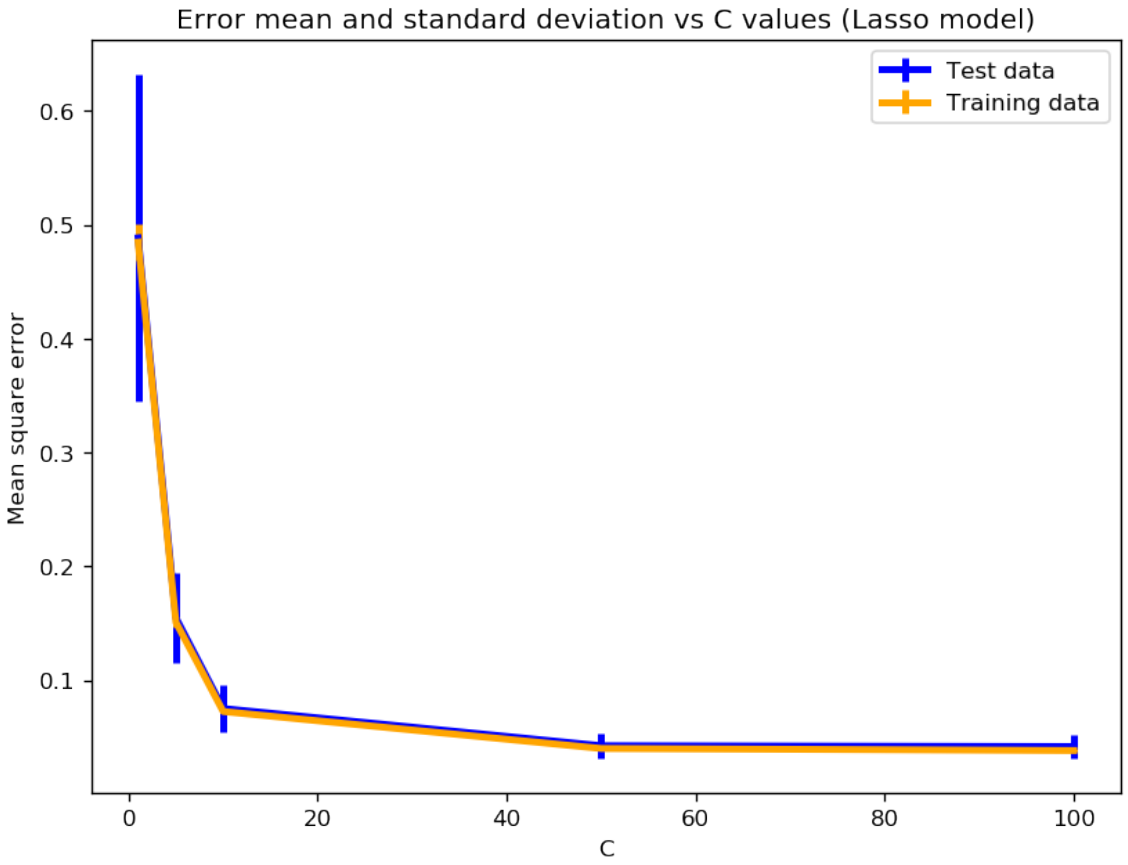

Figure 13

Based on this graph (Figure 13), we can see that the best results (i.e. smallest MSE and standard deviation) are obtained for $C \geq 50$. To avoid over-fitting when choosing either $C = 50$ or $C = 100$, the best solution is to go for the simplest model possible (i.e. the smallest value of C). Therefore, I would recommend using $C = 50$ for this model.

(d) As for Lasso Regression model, we can choose an appropriate value for C in Ridge Regression model by cross-validating a range of different values. In question (i)(e), we have seen that $C = 1e{-}5$ has a MSE close to the baseline model and $C = 1e{-}1$ already leads to over-fitting (we can deduce that from the very low MSE of the model). Hence we could explore values in this range: $C \in \{1e{-}5, 5e{-}5, 1e{-}4, 5e{-}4, 1e{-}3, 5e{-}3, 1e{-}2, 5e{-}2, 1e{-}1\}$

A similar Python code as for (ii)(b) can be used, using Ridge model instead of Lasso and with the appropriate range of values for C. This gives us the following results:

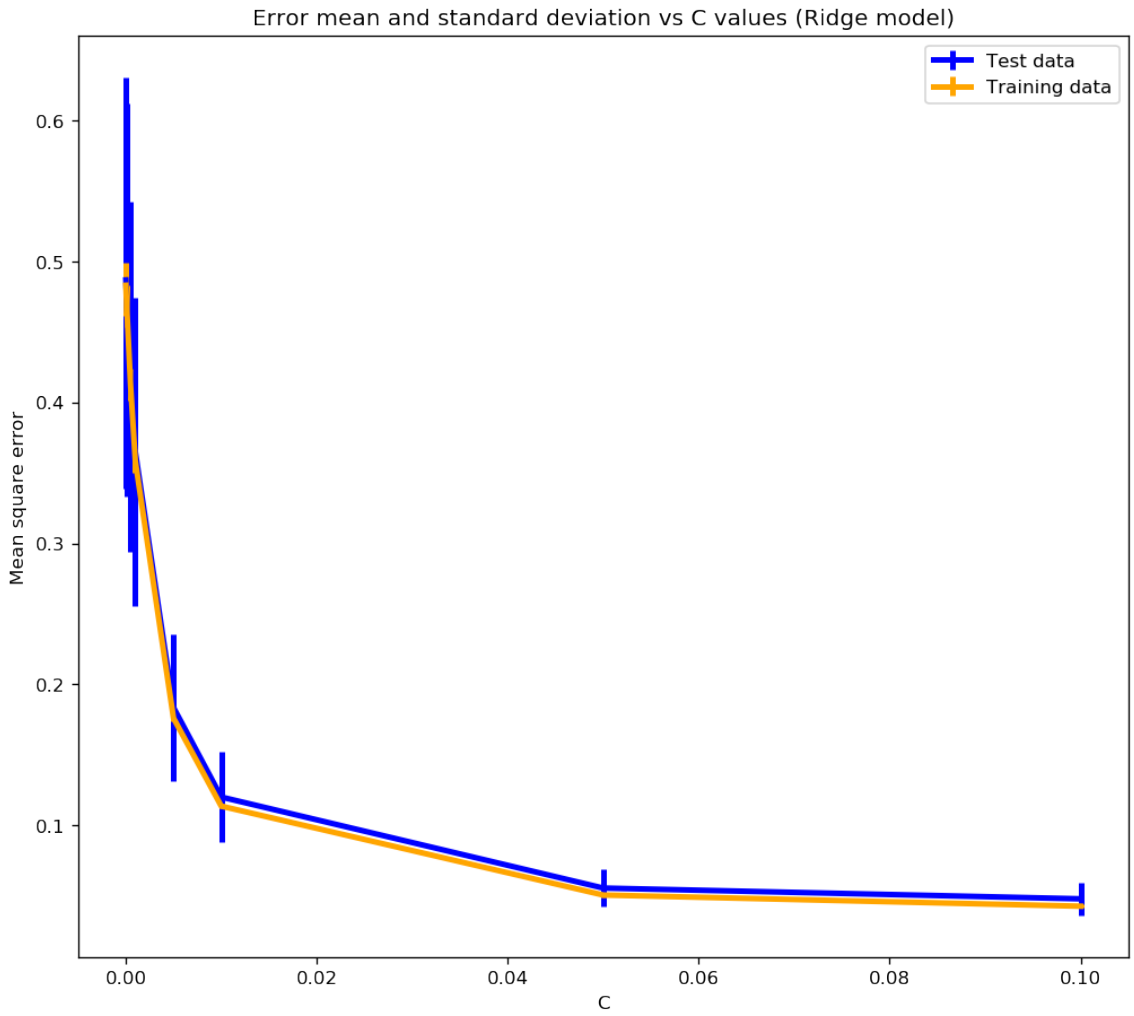

Figure 14

Based on this graph (Figure 14), we can argue that the most appropriate value is $C = 5e{-}2$: it is the smallest value (to minimise over-fitting) that has the smallest MSE and standard deviation (i.e. spread of results). It is also interesting to note the slight gap between training data and test data mean squared errors that intensifies when C increases (largely visible when plotting bigger values of C): this behaviour is characteristic of over-fitting.

Finally, more generally, I would largely recommend using Lasso Regression model rather than Ridge Regression model for this data as its behaviour is more appropriate. Otherwise a possibility to use Ridge Regression model on this data would be to choose, for example by cross-validation, an appropriate value for q (i.e. the degree of polynomial features to use).

A Appendix

A.1 Python Code

```python
#!/usr/bin/env python
# coding: utf-8

# Dataset: id:22-22-22

# Read data
import numpy as np
import pandas as pd
df = pd.read_csv("week3.csv", comment="#")
X1 = df.iloc[:,0]
X2 = df.iloc[:,1]
X = np.column_stack((X1,X2))
y = df.iloc[:,2]

# (i)(a)
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# [[elev, azim], [...]] -> multiple angle of views
figViews = [[10,10], [40,80]]
for figView in figViews:
    fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], y)
    ax.view_init(elev=figView[0], azim=figView[1])
    ax.set(title='Training data from CSV', xlabel='X1', ylabel='X2',
     ↪  zlabel='Y')
    plt.show()

# (i)(b)
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Lasso
from sklearn.dummy import DummyRegressor
from sklearn.metrics import mean_squared_error

Xpoly = PolynomialFeatures(5).fit_transform(X)

baseline = DummyRegressor(strategy="mean").fit(Xpoly, y)
print("J(theta_baseline) = %f\n"%mean_squared_error(y,
 ↪  baseline.predict(Xpoly)))

# Prevents scientific notation for theta values
np.set_printoptions(suppress=True)

```

```python
C_range = [1, 10, 100, 1000]
for Ci in C_range:
    model = Lasso(alpha=1/(2*Ci)).fit(Xpoly, y)
    theta = np.insert(model.coef_, 0, model.intercept_)
    print("C = %.1f"%Ci)
    print("theta =", theta)
    print("J(theta) = %f\n"%mean_squared_error(y, model.predict(Xpoly)))

# (i)(c)
Xtest = []
grid = np.linspace(-5,5)
for i in grid:
    for j in grid:
        Xtest.append([i,j])

Xtest = np.array(Xtest)
Xtest = PolynomialFeatures(5).fit_transform(Xtest)
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

C_range = [1, 10, 100, 1000]
for Ci in C_range:
    model = Lasso(alpha=1/(2*Ci))
    model.fit(Xpoly, y)

    fig = plt.figure(num=None, figsize=(8, 5), dpi=120)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], y, c='g', label="Training data")
    surf = ax.plot_trisurf(Xtest[:,1], Xtest[:,2], model.predict(Xtest),
    ↪  cmap=cm.coolwarm, alpha=0.8, linewidth=0, antialiased=True)
    ax.view_init(elev=30, azim=50)
    ax.set_title('Lasso prediction surface for C = %.0f'%Ci)
    ax.set(xlabel='X1', ylabel='X2', zlabel='Y')
    ax.legend(bbox_to_anchor=(0.84, 0.9), loc='upper left')

    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
    cbar = fig.colorbar(surf, shrink=0.5, aspect=5)
    cbar.ax.get_yaxis().labelpad = 15
    cbar.ax.set_ylabel('Predictions', rotation=270)

    plt.show()

# (i)(e)
from sklearn.linear_model import Ridge

baseline = DummyRegressor(strategy="mean").fit(Xpoly, y)
```

```
print("J(theta_baseline) = %f\n"%mean_squared_error(y, baseline.predict(Xpoly)))

C_range = [1e-7, 1e-5, 1e-3, 1e-2, 1e-1, 1]
for Ci in C_range:
    model = Ridge(alpha=1/(2*Ci)).fit(Xpoly, y)
    theta = np.insert(model.coef_, 0, model.intercept_)
    print("C = %.0E"%Ci)
    print("theta =", theta)
    print("J(theta) = %f\n"%mean_squared_error(y, model.predict(Xpoly)))

    # continue; # Uncomment to display only parameters values (i.e. hide graphs)

    fig = plt.figure(num=None, figsize=(8, 5), dpi=120)
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], y, c='g', label="Training data")
    surf = ax.plot_trisurf(Xtest[:,1], Xtest[:,2], model.predict(Xtest), cmap=cm.coolwarm, alpha=0.8, linewidth=0, antialiased=True)

    ax.view_init(elev=30, azim=50)
    ax.set_title('Ridge prediction surface for C = %.0E'%Ci)
    ax.set(xlabel='X1', ylabel='X2', zlabel='Y')
    ax.legend(bbox_to_anchor=(0.84, 0.9), loc='upper left')

    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
    cbar = fig.colorbar(surf, shrink=0.5, aspect=5)
    cbar.ax.get_yaxis().labelpad = 15
    cbar.ax.set_ylabel('Predictions', rotation=270)

    plt.show()

# (ii)(a)
from sklearn.model_selection import KFold
from sklearn import linear_model
from sklearn.metrics import mean_squared_error

# k = 5 only
model = Lasso(alpha=1/2) # <=> C = 1
kf = KFold(n_splits=5)
temp = []
for train, test in kf.split(Xpoly):
    model.fit(Xpoly[train], y[train])
    ypred = model.predict(Xpoly[test])
    temp.append(mean_squared_error(y[test], ypred))

print("5-fold cross validation results:")
```

```
134 print("Mean error = %f; Standard deviation = %f"%(np.array(temp).mean(),
    ↪  np.array(temp).std()))
135
136 # k-fold cross-validation
137 plt.figure(num=None, figsize=(8, 6), dpi=120)
138 model = Lasso(alpha=1/2) # <=> C = 1
139 k = [2, 5, 10, 25, 50, 100]
140 std_error = []
141 mean_error = []
142
143 for ki in k:
144     kf = KFold(n_splits=ki)
145     temp = []
146     for train, test in kf.split(Xpoly):
147         model.fit(Xpoly[train], y[train])
148         ypred = model.predict(Xpoly[test])
149         temp.append(mean_squared_error(y[test], ypred))
150
151     mean_error.append(np.array(temp).mean())
152     std_error.append(np.array(temp).std())
153
154 plt.errorbar(k, mean_error, yerr=std_error, linewidth=3)
155 plt.title("Error mean and variance for k-fold cross-validation")
156 plt.gca().set(xlabel='k (number of folds)', ylabel='Mean square error')
157 plt.show()
158
159 # (ii)(b) & (ii)(c)
160 stdErrorTest, meanErrorTest, stdErrorTrain, meanErrorTrain = [], [], [],
    ↪  []
161 plt.figure(num=None, figsize=(8, 6), dpi=120)
162 C_range = [1, 5, 10, 50, 100]
163 kf = KFold(n_splits=10)
164
165 for Ci in C_range:
166     model = Lasso(alpha=1/(2*Ci))
167     tempTest, tempTrain = [], []
168
169     for train, test in kf.split(Xpoly):
170         model.fit(Xpoly[train], y[train])
171         ypred = model.predict(Xpoly[test])
172         ypred_train = model.predict(Xpoly[train])
173
174         tempTest.append(mean_squared_error(y[test], ypred))
175         tempTrain.append(mean_squared_error(y[train], ypred_train))
176
177     meanErrorTest.append(np.array(tempTest).mean())
178     stdErrorTest.append(np.array(tempTest).std())
179     meanErrorTrain.append(np.array(tempTrain).mean())
```

```
180     stdErrorTrain.append(np.array(tempTrain).std())
181 
182 plt.errorbar(C_range, meanErrorTest, yerr=stdErrorTest, linewidth=3,
    ↪  c="blue", label="Test data")
183 plt.errorbar(C_range, meanErrorTrain, yerr=stdErrorTrain, linewidth=3,
    ↪  c="orange", label="Training data")
184 plt.title("Error mean and standard deviation vs C values (Lasso model)")
185 plt.gca().set(xlabel='C', ylabel="Mean square error")
186 plt.legend()
187 plt.show()
188 
189 #(ii)(d)
190 C_range = [1e-5, 5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1]
191 stdErrorTest, meanErrorTest, stdErrorTrain, meanErrorTrain = [], [], [],
    ↪  []
192 kf = KFold(n_splits=10) # Justify usage of 5 or 10-fold
193 plt.figure(num=None, figsize=(10, 9), dpi=120)
194 
195 for Ci in C_range:
196     model = Ridge(alpha=1/(2*Ci))
197     tempTest, tempTrain = [], []
198 
199     for train, test in kf.split(Xpoly):
200         model.fit(Xpoly[train], y[train])
201         ypred = model.predict(Xpoly[test])
202         ypred_train = model.predict(Xpoly[train])
203 
204         tempTest.append(mean_squared_error(y[test], ypred))
205         tempTrain.append(mean_squared_error(y[train], ypred_train))
206 
207     meanErrorTest.append(np.array(tempTest).mean())
208     stdErrorTest.append(np.array(tempTest).std())
209     meanErrorTrain.append(np.array(tempTrain).mean())
210     stdErrorTrain.append(np.array(tempTrain).std())
211 
212 plt.errorbar(C_range, meanErrorTest, yerr=stdErrorTest, linewidth=3,
    ↪  c="blue", label="Test data")
213 plt.errorbar(C_range, meanErrorTrain, yerr=stdErrorTrain, linewidth=3,
    ↪  c="orange", label="Training data")
214 plt.title("Error mean and standard deviation vs C values (Ridge model)")
215 plt.gca().set(xlabel='C', ylabel="Mean square error")
216 plt.legend()
217 plt.show()
```