Project 1 - FYS-STK3155

October 9, 2020

1 Abstract

We study linear regression by implementing and trying out three regression methods, ordinary least squares (OLS), ridge, and lasso. We find that OLS perfoms well both on our synthetoc test data and real terrain data, but is vulnerable to overfitting. Ridge and lasso both minimise this issue by restricting the individual coefficient values. Lasso however seems to overdue this effect for our datasets, and is outperformed by the other two models. It is suggested the lasso is reserved for very high dimensionality problems.

2 Introduction

In this first project of the course we are looking at linear regression and resampling methods. The goal is to implement three methods for linear regression; ordinary least squares, ridge, and lasso regression, and study their performance and behavior. Two types of resampling methods, namely the bootstrap and k-fold cross validation is used to better evaluate the method and to determine the optimal value for the relevent parameters.

We study two types of data. First we create synthetic data using the Franke function. This data is then used to explore and verify the models. After this we move on to real data, and in this project we will look at map data from UCSG's EarthExplorer, more specifically elevation. We explore how the models perform on this data,

We will look at the performance of the models and study their bias-variance trade-off. We also study the coefficients $\boldsymbol{\beta}$ and how they vary between the models.

All code as well as the terrain data used is available at my github repository for the class.

3 Data

We create synthetic data using the Franke function, before looking real digital terrain data to explore our regression models. ## The Franke function The Franke function is a two dimensional weighted sum of four exponentials. It has two Gaussian peaks of different heights, and a smaller dip and is often used as a test function in interpolation problems.

The function is defined as

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) + \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) -$$

and will be defined for $x, y \in [0, 1]$. See figure 1 for a plot.

I have added some noise to this function, following a normal distribution, namely $\mathcal{N}(0, 0.1)$.

3.1 Terrain Data

We will use topological map data as real data for trying out our regression methods. I used EarthExplorer[1] to find a suitable map of elevation and chose an area over the Teton mountain range in Wyoming, USA. The map section had the enitityId SRTM1N43W111V3. I downloaded it as a GeoTIFF file with resolution of 1 arc second. A plot of this map is shown in figure 18.

4 Methods

I explore three different methods for linear regression, as well as two methods for resampling. The model itself will be fit using a polynomial of variable degree, built from two input variables. ## Regression Methods

4.0.1 OLS

Ordinary Least Squares Regression (OLS) fits a linear model with coefficients β_i to minimize the residual sum of squares between the output value (aka dependent or target variable) in the dataset, and the output as predicted by the linear approximation. With $\boldsymbol{X}$ as a matrix of the input variables, and $\boldsymbol{y}$ as the output or target, we approximate the target as

$$\boldsymbol{\hat{y}} = \boldsymbol{X}\boldsymbol{\beta},$$

So the goal of OLS is to find the optimal $\boldsymbol{\hat{\beta}}$ that minimizes the difference between the values $\boldsymbol{\hat{y}}$ and $\boldsymbol{y}$.

Defining the loss function to quantify this difference, or spread, as:

$$L(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{\hat{y}})^T(\boldsymbol{y} - \boldsymbol{\hat{y}})\right\},$$

We want to minimize this function, and by taking the derivative of L with respect to the individual $\boldsymbol{\beta_j}$ and solving for $\boldsymbol{\beta}$ we find the solution

$$\boldsymbol{\hat{\beta}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

that can be used to calculate $\boldsymbol{\hat{\beta}}$.

From an new input $\boldsymbol{X_a}$ we can use the found $\boldsymbol{\hat{\beta}}$ to calculate an estimate or prediction for the target $\boldsymbol{y_a}$, $\boldsymbol{\hat{y}_a}$.

4.0.2 Ridge regression

Both ridge regression and the lasso are so called shrinkage methods. Named due to how they shring the contribution from selected coefficients $\boldsymbol{\beta}$.

Ridge regression modifies OLS by putting a restriction on the size of the individual coefficients $\boldsymbol{\beta}$. This is particularly useful in models with many (partly correlated) input values. The coefficients are then likely to become poorly determined, with high variance.

To combat this behavior ridge regression adds a penalty term to the loss function from the OLS model penalizing large beta values. The penalty is equivalent to the square of the magnitude of the coefficients. More succintly the ridge model adds L2 regularization to the OLS model.

starting with the expression from the above section,

$$L_{OLS}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \sum_{j=1}^{p} x_{ij}\beta_j^2)^2,$$

a penalty term is added

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \sum_{j=1}^{p} \beta_j^2.$$

subject to

$$\sum_{j=1}^{p} \beta_j^2 \leq t,$$

where t is a positive number.

In matrix notation

$$L(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n} \left\{ (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \right\} + \lambda \boldsymbol{\beta}^T \boldsymbol{\beta},$$

From which we get an expression for the coefficients, $\boldsymbol{\beta}^{\texttt{ridge}}$

$$\boldsymbol{\beta}^{\texttt{ridge}} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \boldsymbol{I} \right)^{-1} \boldsymbol{X}^T \boldsymbol{y}$$

where $\boldsymbol{I}$ is the `p×p` identity matrix. The parameter λ is often called a regularization parameter or hyperparameter.

We can see from this that for $\boldsymbol{\lambda} = 0$ this model reduces to OLS. The bigger the value of $\boldsymbol{\lambda}$, the stricter the restriction on the size of the $\boldsymbol{\beta}$ values. In this way the ridge regression model can reduce model complexity, and thereby hopefully reduce overfitting. Large values of λ can conversely lead to underfitting.

4.0.3 Lasso regression

Like ridge regression, lasso (least absolute shrinkage and selection operator) regression adds a penalty to the loss function. We say lasso performs L1 regularization by adding a penalty equivalent to the absolute value of the magnitude of the coefficients.

The loss function becomes

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \sum_{j=1}^{p}|\beta_j|.$$

subject to

$$\sum_{j=1}^{p}|\beta_j| \leq t,$$

this constraint makes the solutions nonlinear in $\boldsymbol{y}$ meaning that there is no closed form expression for the lasso, unlike in ridge regression.

The coefficients $\boldsymbol{\beta}^{\texttt{lasso}}$ are given by

$$\boldsymbol{\beta}^{\texttt{lasso}} = \min_{\boldsymbol{\beta}}\Bigg(\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|\Bigg)$$

4.1 Resampling

Resampling methods are in essence methods for efficiently using the (often limited) data available to gain insight about the data. They are among other things used to estimate the precision of sample statistics on the data and for model validation. We will be focusing on this latter application. In essence, resampling methods work by using the (training) data available, and repeatedly drawing samples from this set and refitting the model of interest on each sample in order to obtain additional information about this model.

We have looked at two resampling methods: * the bootstrap method, and

- k-fold cross-validation

4.1.1 The Bootstrap

Bootstrapping is a method which uses resampling with replacement on the available dataset, in essence using the available data as a pdf for drawing datasets. For each sample the model is fitted using this sample, and relevant measures are calculated, for instance the mean square error. This process is repeated B times. For the standard bootstrap, a sample of N data points are drawn for a dataset (training set) of size N.

The bootstrap does not make assumptions about the underlying distribution of the data and is a quite general method. ### k-Fold Cross Validation Cross validation methods are a subset of resampling methods used mainly for model validation. They work by dividing up the available data, i.e. the training set, into a number of sections or *folds*, and then fitting the model on some of the data, and using the remaining data as a test set for validation.

In k-fold cross-validation the (shuffled) dataset is split into k so-called folds, and $k-1$ folds are used for training or fitting the model, and the final fold is used to test the resulting model. This is repeated k times, until all of the folds have been used as test fold exactly once.

4.2 Error measures

The r^2 score, also known as the coefficient of determination, is a common measure of how well a model is able to predict outcomes. It is defined as one minus the residual sum of squares,

$$\text{RSS} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

divided by the total sum of squares,

$$\text{TSS} = \sum_{i=1}^{n} (y_i - y_{mean})^2$$

giving;

$$r^2 = 1 - \frac{RSS}{TSS}$$

Here values closer to 1 are better, with $r^2 = 1.0$ being the optimal model. It is worth noting that r^2 can take negative values.

The MSE is the mean of the square of the errors, or residual sum of squares:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2$$

and naturally values closer to zero are better.

4.2.1 Bias-Variance Trade-off

We will be exploring the bias-variance trade-off for our regrsession models. First now let us look at the error, and how it is divided into a bias term (squared) and a variance term.

We have from the subsection on regression methods our loss, or cost, function

$$L(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2\right].$$

Here the expected value $\mathbb{E}$ is the sample value. This can be manipulated to the form

$$\mathbb{E}\left[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2\right] = \frac{1}{n}\sum_{i}(f_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \frac{1}{n}\sum_{i}(\hat{y}_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \sigma^2.$$

Which shows how the error is divided into a squared bias term, a variance term, and finally σ^2 is the variance of the noise meaning the variance of our target around its true mean. This last term cannot be reduced no matter how good our estimate $\hat{y}$ is.

The bias term corresponds to the difference between the average or expected value of our estimate, and the true mean, while the variance term is the expected squared deviation of $\hat{y}$ around its mean. Generally the variance will increase with model complexity, while the bias^2 reduces.

Now to derive the above expression.

We have $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{\epsilon}$ with $\epsilon \sim \mathcal{N}(\prime, \sigma^{\in})$. Our approximation to $\boldsymbol{y}$ is $\hat{\boldsymbol{y}} = \boldsymbol{X\beta}$. For simplicity I will write $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}$.

I begin by the trick of adding and subtracting $\boldsymbol{f}$ inside the square.

$$
\begin{aligned}
\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] &= \mathbb{E}\left[(\boldsymbol{y}-\boldsymbol{f}) + (\boldsymbol{f}-\hat{\boldsymbol{y}}))^2\right] \\
&= \mathbb{E}\left[(\boldsymbol{y}-\boldsymbol{f})^2 + (\boldsymbol{f}-\hat{\boldsymbol{y}})^2 - 2(\boldsymbol{f}-\hat{\boldsymbol{y}})(\boldsymbol{y}-\boldsymbol{f})\right] \\
&= \mathbb{E}[(\boldsymbol{y}-\boldsymbol{f})^2] + \mathbb{E}[(\boldsymbol{f}-\hat{\boldsymbol{y}})^2] - 2\mathbb{E}[(\boldsymbol{f}-\hat{\boldsymbol{y}})(\boldsymbol{y}-\boldsymbol{f})]
\end{aligned}
$$

Inserting $\boldsymbol{y} = \boldsymbol{f} + \boldsymbol{\epsilon}$ in the first part of this expression gives

$$
\begin{aligned}
\mathbb{E}\left[(\boldsymbol{y}-\boldsymbol{f})^2\right] &= \mathbb{E}\left[\boldsymbol{y}^2 - 2\boldsymbol{y}\boldsymbol{f} + \boldsymbol{f}^2\right] \\
&= \mathbb{E}\left[(\boldsymbol{f}+\epsilon)^2 - 2(\boldsymbol{f}+\epsilon)\boldsymbol{f} + \boldsymbol{f}^2\right] \\
&= \mathbb{E}\left[\epsilon^2\right] \\
&= Var(\epsilon) + (\mathbb{E}\left[\epsilon\right])^2 \\
&= \sigma^2
\end{aligned}
$$

For the third part we can use that $\boldsymbol{f}$ is deterministic and that $\mathbb{E}(\epsilon) = 0$, giving

$$
\begin{aligned}
\mathbb{E}(\boldsymbol{y}\boldsymbol{f}) &= \mathbb{E}((\boldsymbol{f}+\sigma)\boldsymbol{f}) \\
&= \mathbb{E}(\boldsymbol{f}^2) \\
&= \boldsymbol{f}^2
\end{aligned}
$$

and

$$
\begin{aligned}
\mathbb{E}(\boldsymbol{y}\hat{\boldsymbol{y}}) &= \mathbb{E}(\hat{\boldsymbol{y}}(\boldsymbol{f}+\sigma)) \\
&= \mathbb{E}(\hat{\boldsymbol{y}}\boldsymbol{f})
\end{aligned}
$$

Using this we get

$$
\begin{aligned}
2\mathbb{E}[(\boldsymbol{f}-\hat{\boldsymbol{y}})(\boldsymbol{y}-\boldsymbol{f})] &= \mathbb{E}(\boldsymbol{f}\boldsymbol{y}) - \mathbb{E}(\boldsymbol{f}^2) - \mathbb{E}(\boldsymbol{y}\hat{\boldsymbol{y}}) + \mathbb{E}(\hat{\boldsymbol{y}}\boldsymbol{f})) \\
&= 0
\end{aligned}
$$

And we are left with

$$
\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] = \sigma^2 + \mathbb{E}[(\boldsymbol{f}-\hat{\boldsymbol{y}})^2]
$$

Where I now add an subtract $\mathbb{E}[\hat{\boldsymbol{y}}]$ inside the square.

$$\begin{aligned}
\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] &= \sigma^2 + \mathbb{E}[(\boldsymbol{f}-\hat{\boldsymbol{y}})^2] \\
&= \sigma^2 + \mathbb{E}\left[((\boldsymbol{f}-\mathbb{E}[\hat{\boldsymbol{y}}]) + (\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}}))^2\right] \\
&= \sigma^2 + \mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\hat{\boldsymbol{y}}])^2 + (\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}})^2 + 2(\boldsymbol{f}-\mathbb{E}[\hat{\boldsymbol{y}}])(\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}})\right] \\
&= \sigma^2 + \mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\hat{\boldsymbol{y}}])^2 + (\mathbb{E}[\hat{\boldsymbol{y}}]-\hat{\boldsymbol{y}})^2\right]
\end{aligned}$$

Where the last term canceled out using again that $\boldsymbol{f}$ is deterministic, and that $\mathbb{E}[\mathbb{E}(a)] = \mathbb{E}(a)$.

Switching the sign in the last square and using sums for the outer expectations gives us

$$\begin{aligned}
\mathbb{E}\left[(\boldsymbol{y}-\hat{\boldsymbol{y}})^2\right] &= \mathbb{E}\left[(\boldsymbol{f}-\mathbb{E}[\hat{\boldsymbol{y}}])^2 + (\hat{\boldsymbol{y}}-\mathbb{E}[\hat{\boldsymbol{y}}])^2\right] + \sigma^2 \\
&= \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \frac{1}{n}\sum_i (\hat{y}_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \sigma^2.
\end{aligned}$$

Which is what we wanted to show.

4.3 Preprocessing

Before a data analysis can begin is important to preprosess the data we are working on. This includes removing inconsistent and corrupted values, confirming completeness of the dataset and possibly removing highly corrolated features as well as outliers. It may also include selecting certain features from the dataset to focus on to reduce dimensionality.

In this project our data is either synthetic and as such well defined, or we use map data where the issues mentioned are not relevant concerns. What is most relevant in this assignment is scaling. ### Scaling Many models are sensitive to the effective value range of the features or input data. There are several ways to employ scaling. One popular option is to adjust the data so each predictor has mean value equal to zero and a variance of one. Another option is to scale all the data points so each feature vector has the same euclidian lentgh. Yet another option is to scale the values to all lie between a given minimum and maximum value, typically zero and one.

We will be using scikit-learn's `StandardScaler` which standardizes the data by subtracting the mean and scaling to unit variance. ### Dividing the data set A crucial step when trying to use regression to create a model based on a dataset is to divide up the dataset in at least two sets. This being a training set to train the model, and a test set to test it. In addition, if the size of the data set allows, one may also add a validation set for validating and fine tuning the model before tesing it on the test set.

I will be dividing the data into training and test sets, and use cross validation in place of a separate validation test. I will be using a 75%/25% split between training and test data.

4.4 Packages and Tools

While I have written my own code for the OLS and ridge regression models, as well as the bootstrap and kFold CV, I have used functionality from the library scikit-learn[3] for the lasso regression as well as for scaling and splitting the data set. This python library is based on numpy and and scipy, and contains a wide array of machine learning algorithms, including regression methods.

Other packages I've used is numpy[4] for array handling, matplotlib.pyplot[5] for plots and visualizations, and python's random module[6] for generating (pseudo) random numbers. ## Code I have developed a class `CreateData` for creating data, building the design matrix, scaling, plotting, and splitting into training and test sets. For the regression methods I have created a class `OrdinaryLeastSquares` with methods for model fit and prediction. In addition I have included in this class methods for bootstrap and f-fold cross-validation, some erre measures, and methods for finding the confidence intervals of $\boldsymbol{\beta}$. I have then created a class `RidgeRegression` which inherits `OrdinaryLeastSquares` and is only modified to allow for setting and using the λ parameter. I have also chosen to wrap scikit-learns `Lasso` in a class `LassoRegression` which again inherits the class `RidgeRegression`. This is again to allow calling the methods explained for this regression model as well.

For testing I have created unit tests in the file `Project1UnitTests.py`. Unfortunately I did not have time to add tests for all the methods. The tests have been implemented using python's unittesting framework `unittest`[8]. The tests can be run module or class wise as shown in [8], or simply by file as

```
> python -m unittest Project1UnitTests.py
```

In addition I have tested my code by comparing with the implementations in scikit-learn.

In retrospect I see that I should have simple implemented a class for ridge regression, and let $\lambda = 0$ for OLS. I also see that the methods for error measures would probably be better suited outside the regression class itself.

Code available from the following github repository: https://github.com/emiliefj/FYS-STK3155.

5 Results and Discussion

5.1 Ordinary Least Squares

First out is the simpler of the regression methods, ordinary least squares, or OLS. I begin by creating some synthetic data using the Franke function. Note that the n in this code is the number of unique data points in the x and y direction, but as I use numpy's meshgrid when creating data the actual number of datapoints is $n \times n$.

[106]:
```
#
# Make data and preprocess
#

import Code.CreateData as cd

n = 100 # number of datapoints
data = cd.CreateData(n,seed=8)
data.plot_data()
print("Figure 1: 3D plot of the Franke function")
```

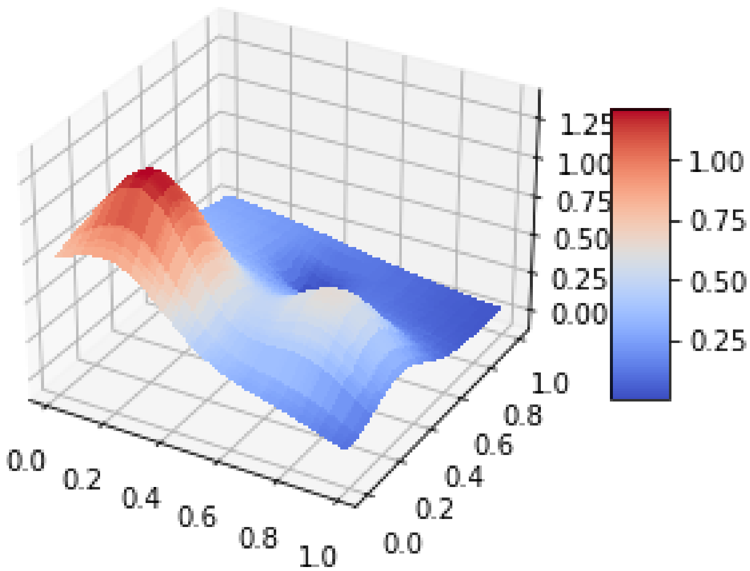


Figure 1: 3D plot of the Franke function

Now I add some noise to this and see how it effects the plot.

[107]:
```
variance = 0.1
data.add_normal_noise(0,variance)
data.plot_data()
print("Figure 2: 3D plot of the Franke function with added noise.")
```

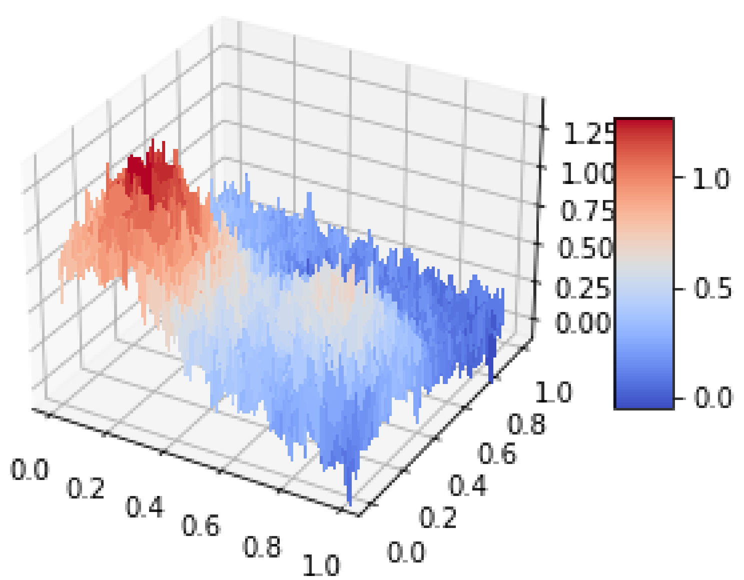


Figure 2: 3D plot of the Franke function with added noise.

With this data I create a design matrix for a polynomial of degree d=5 to start. I split the data into training and test, scale it, and then I train my OLS model on the training data.

[108]:
```python
# Preprocessing data
degree = 5
test_fraction = 0.25
data.create_design_matrix(degree)
data.split_dataset(test_fraction)
data.scale_dataset(type='standard') # Using SciKit's StandardScaler

# Model
import Code.OrdinaryLeastSquares as ols
ols_model = ols.OrdinaryLeastSquares(seed=251)
ols_model.fit(data.X_train,data.z_train)
```

[108]:
```
array([0.52318392, 0.38455936, 0.1889064 , …, 0.11585749, 0.43133426,
       0.25477458])
```

I can try out my new model on the test set, and as a first look at accuracy I plot the prediction against the true values. Ideally this should produce a straight line.

[109]:
```python
import matplotlib.pyplot as plt
import numpy as np

z_hat = ols_model.predict(data.X_test)
mse_ols = ols_model.mean_square_error(z_hat,data.z_test)
print("The mse on the test set is: %.4f" %(mse_ols))

plt.scatter(data.z_test, z_hat)
plt.title('Prediction of z vs test values: OLS')
plt.xlabel('$z_{test}$')
plt.ylabel('$z_{predict}$')
plt.show()
print("Figure 3: predicted values for z on the test set plotted against actual␣
 ↪values.")

data.plot_data(X=data.X_test, z=data.z_test)
data.plot_data(X=data.X_test, z=z_hat)
print("Figure 4: 3D plot of the test data (above) and 3D plot of predicted␣
 ↪values (below).")
```

```
The mse on the test set is: 0.0125
```

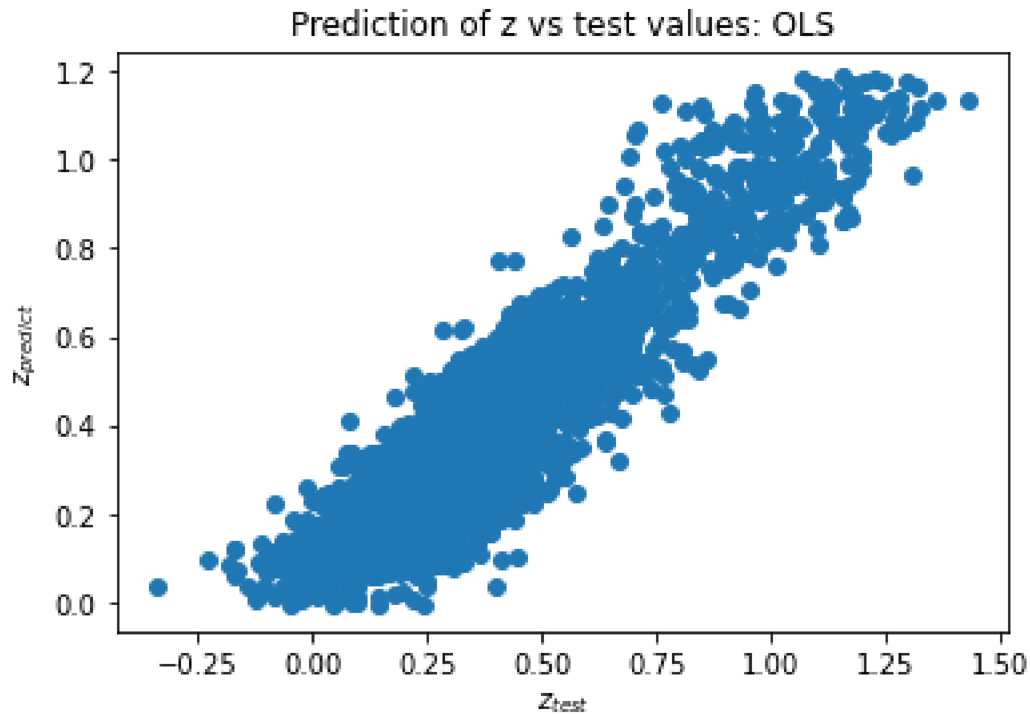
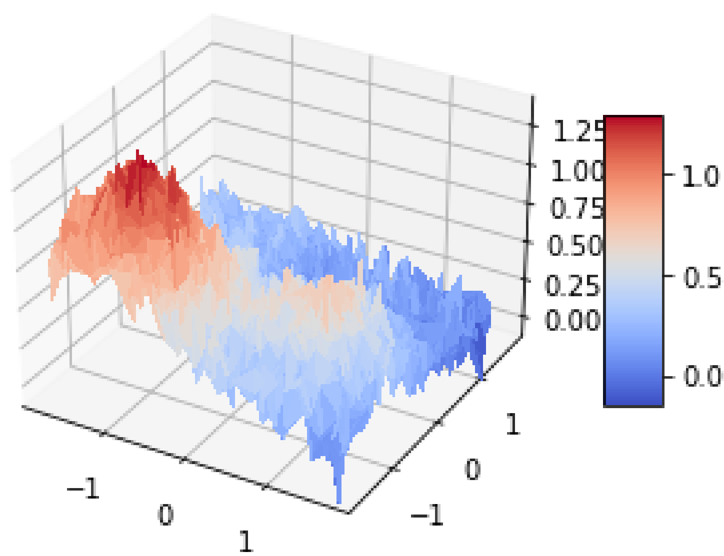


Figure 3: predicted values for z on the test set plotted against actual values.



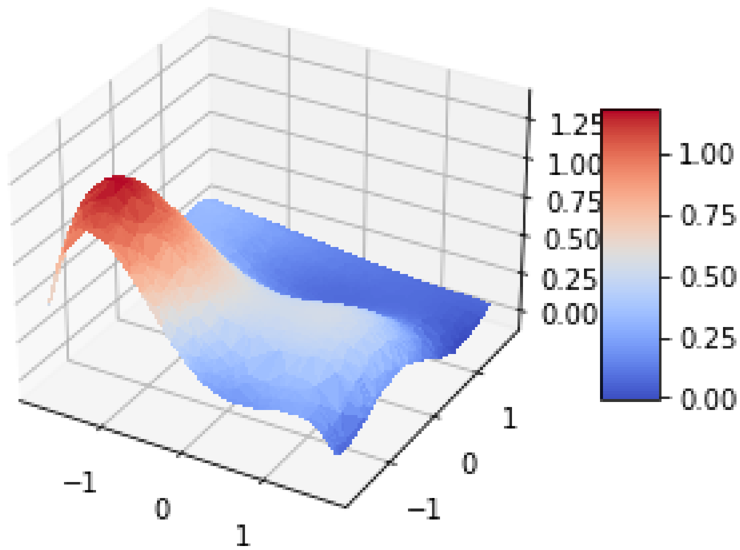


Figure 4: 3D plot of the test data (above) and 3D plot of predicted values (below).

I see we get close to a straight line, and there is quite a bit of spread. To verify my implementtion I can compare the result from my method with that from SciKit-Learn.

[169]:
```
from sklearn.linear_model import LinearRegression

skl_ols = LinearRegression(fit_intercept=False) # OLS
skl_ols.fit(data.X_train, data.z_train)

skl_z_hat = skl_ols.predict(data.X_test)

print("The arrays are about the same: ", np.allclose(z_hat,skl_z_hat))
print("The mean squared error between my result and that of scikit-learn is: ",␣
↪ols_model.mean_square_error(z_hat,skl_z_hat))
```

```
The arrays are about the same:  True
The mean squared error between my result and that of scikit-learn is:
2.0601779122756744e-22
```

I see that my implementation matches that of scikitLearn well. Now let us explore the model more rigorously. I will now compare the r^2 score and the mean square error for increasing model complexity, as well as by increasing the number of data points in my dataset.

[124]:
```
import pandas as pd

def evaluate_model(model,N_array,dmin=2,dmax=15):
```

```
    degree = np.arange(dmin,dmax+1)
    N_label = []
    d_label = ['d=%d'%i for i in range(dmin,dmax+1)]
    r2_scores = np.zeros((len(N_array),len(degree)))
    mse_scores = np.zeros((len(N_array),len(degree)))
    for n in range(len(N_array)):
        data = cd.CreateData(N_array[n])
        data.add_normal_noise(0,variance)
        N_label.append("N="+str(N_array[n])+"x"+str(N_array[n]))
        for d in range(len(degree)):
            data.create_design_matrix(degree[d])
            data.split_dataset(test_fraction)
            data.scale_dataset(type='standard')
            model.fit(data.X_train,data.z_train)
            z_hat = model.predict(data.X_test)
            r2_scores[n,d] = model.r2(z_hat,data.z_test)
            mse_scores[n,d] = model.mean_square_error(z_hat,data.z_test)
    pd.options.display.float_format = '{:,.3f}'.format
    r2_df = pd.DataFrame(r2_scores,index=N_label,columns=d_label)
    mse_df = pd.DataFrame(mse_scores,index=N_label,columns=d_label)
    return r2_df, mse_df


N_array = np.array([5,10,15,25,50,100,250,500])
r2_df, mse_df = evaluate_model(ols_model,N_array,2,20)
print("Table 1: R^2 score for increasing polynomial degree and number of␣
 ↪datapoints")
display(r2_df)
print("Table 2: Mean squared error for increasing polynomial degree and number␣
 ↪of datapoints")
display(mse_df)
```

Table 1: R^2 score for increasing polynomial degree and number of datapoints

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11	\
N=5x5	0.956	0.396	0.338	0.166	0.169	0.141	0.091	0.022	-0.062	-0.157	
N=10x10	0.478	0.666	0.730	0.770	0.478	0.042	-0.055	-0.074	-0.072	-0.069	
N=15x15	0.632	0.671	0.673	0.738	0.743	0.717	0.311	0.008	-0.001	-0.005	
N=25x25	0.338	0.666	0.791	0.843	0.856	0.860	0.874	0.873	0.869	0.871	
N=50x50	0.596	0.740	0.823	0.855	0.870	0.875	0.876	0.875	0.873	0.875	
N=100x100	0.628	0.767	0.820	0.847	0.857	0.866	0.870	0.871	0.873	0.874	
N=250x250	0.577	0.761	0.823	0.850	0.862	0.870	0.874	0.875	0.877	0.877	
N=500x500	0.561	0.752	0.812	0.842	0.854	0.862	0.866	0.867	0.868	0.869	

	d=12	d=13	d=14	d=15	d=16	d=17	d=18	d=19	d=20
N=5x5	-0.258	-0.359	-0.456	-0.545	-0.625	-0.694	-0.753	-0.803	-0.846
N=10x10	-0.085	-0.082	-0.087	-0.090	-0.101	-0.107	-0.110	-0.118	-0.128
N=15x15	-0.012	-0.011	0.007	0.102	0.046	-0.018	-0.016	-0.017	-0.017

N=25x25	0.869	0.870	0.874	0.872	0.872	0.872	0.872	0.869	0.856
N=50x50	0.874	0.874	0.874	0.873	0.875	0.874	0.864	0.870	0.872
N=100x100	0.874	0.873	0.874	0.874	0.873	0.873	0.873	0.873	0.873
N=250x250	0.877	0.877	0.877	0.877	0.877	0.877	0.877	0.877	0.877
N=500x500	0.869	0.869	0.869	0.869	0.869	0.869	0.869	0.869	0.869

Table 2: Mean squared error for increasing polynomial degree and number of datapoints

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11	\
N=5x5	0.003	0.060	0.126	0.163	0.187	0.215	0.245	0.280	0.320	0.366	
N=10x10	0.033	0.020	0.020	0.017	0.062	0.487	10.019	780.331	274.612	160.489	
N=15x15	0.023	0.019	0.020	0.017	0.018	0.019	0.106	4.189	9.095	15.840	
N=25x25	0.033	0.021	0.015	0.013	0.012	0.012	0.012	0.012	0.012	0.013	
N=50x50	0.025	0.017	0.013	0.011	0.010	0.010	0.010	0.010	0.010	0.010	
N=100x100	0.024	0.017	0.013	0.012	0.011	0.010	0.010	0.010	0.010	0.010	
N=250x250	0.027	0.018	0.014	0.012	0.011	0.010	0.010	0.010	0.010	0.010	
N=500x500	0.026	0.017	0.014	0.012	0.011	0.011	0.010	0.010	0.010	0.010	

	d=12	d=13	d=14	d=15	d=16	d=17	\
N=5x5	0.420	0.480	0.546	0.618	0.694	0.773	
N=10x10	496.325	1,591.393	97,565.463	34,684.557	14,135.385	11,310.452	
N=15x15	48.025	52.570	6.247	0.680	1.711	27.143	
N=25x25	0.013	0.013	0.012	0.012	0.013	0.012	
N=50x50	0.010	0.010	0.010	0.010	0.010	0.010	
N=100x100	0.010	0.010	0.010	0.010	0.010	0.010	
N=250x250	0.010	0.010	0.010	0.010	0.010	0.010	
N=500x500	0.010	0.010	0.010	0.010	0.010	0.010	

	d=18	d=19	d=20
N=5x5	0.854	0.935	1.016
N=10x10	10,817.849	9,766.765	8,269.567
N=15x15	2,439.368	228.882	80.002
N=25x25	0.012	0.013	0.014
N=50x50	0.011	0.010	0.010
N=100x100	0.010	0.010	0.010
N=250x250	0.010	0.010	0.010
N=500x500	0.010	0.010	0.010

As expected the performance of the model increases as the number of datapoints increase, but after about $n = 25 \times 25$ the gain is negligible, especially for the higher degree polynomials. For very few datapoints the performance seems to be fairly random, which makes sense as the performance would then depend strongly on the similarity between the data in the training and test set.

While the performance also improves with increasing polynomial degree, we see that this improvement stalls at about the 7th degree. For smaller datasets we see some overfitting for larger polynomial degree.

Let us have a look at the weights or coefficiens $\boldsymbol{\beta}$

5.1.1 The Coefficients β

Let us begin by exploring the values of the coefficients as the model complexity increases.

[64]:
```
def evaluate_betas(model,data,dmin=2,dmax=15):
    degree = np.arange(dmin,dmax+1)
    p_max = int((dmax+1)*(dmax+2)/2)
    d_label = ['d=%d'%i for i in range(dmin,dmax+1)]
    #col_label = ['r^2','mse'] + ['beta_%d'%i for i in range(dmin,dmax+1)]
    row_label =['beta_%d'%i for i in range(p_max)]
    beta_df = pd.DataFrame(index=row_label, columns=d_label)
    for i in range(len(degree)):
        d = degree[i]
        data.create_design_matrix(d)
        data.split_dataset(test_fraction)
        data.scale_dataset(type='standard')
        model.fit(data.X_train,data.z_train)
        p = int((d+1)*(d+2)/2)
        beta_df.iloc[0:p,i] = model.beta
        z_hat = model.predict(data.X_test)
    pd.options.display.float_format = '{:,.2f}'.format
    pd.options.display.max_rows = p_max
    beta_df = beta_df.fillna("-")
    display(beta_df)

print("Table 3: The coefficients beta_i for fitting a polynomial of degree=d␣
↪using OLS.")
evaluate_betas(ols_model,data,2,10)
```

Table 3: The coefficients beta_i for fitting a polynomial of degree=d using OLS.

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10
beta_0	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41
beta_1	-0.30	-0.15	1.22	2.45	0.74	-1.37	-0.99	0.58	0.28
beta_2	-0.21	0.34	0.84	1.10	1.10	-0.36	-1.88	-0.25	1.30
beta_3	0.04	-0.45	-5.94	-11.13	3.04	19.86	9.71	-9.98	8.80
beta_4	0.19	0.42	-0.46	-3.63	-1.36	11.79	21.36	1.06	-19.21
beta_5	-0.11	-1.86	-3.63	-2.92	-3.01	6.67	20.83	2.86	-16.46
beta_6	-	0.29	7.64	14.97	-31.60	-92.06	-26.63	100.93	-116.43
beta_7	-	0.06	1.43	9.79	-2.71	-51.40	-60.09	39.78	107.58
beta_8	-	-0.27	0.33	4.15	7.02	-34.11	-111.41	15.50	225.25
beta_9	-	1.27	3.57	-1.22	-5.89	-36.79	-81.93	13.39	93.35
beta_10	-	-	-3.19	-6.68	68.96	186.31	-22.54	-519.76	713.92
beta_11	-	-	-0.81	-10.80	7.70	96.55	53.47	-212.00	-249.95
beta_12	-	-	-0.03	-1.50	9.25	106.67	240.79	-178.75	-853.88
beta_13	-	-	-0.35	-5.36	-19.85	41.79	271.72	-106.86	-1,035.90
beta_14	-	-	-0.97	6.71	23.29	79.95	127.63	-192.75	-239.50

beta_15	-	-	-	0.19	-59.88	-184.35	200.32	1,407.60	-2,739.68
beta_16	-	-	-	3.67	-7.67	-104.36	-0.22	429.38	239.71
beta_17	-	-	-	1.67	-10.81	-111.94	-172.09	558.01	1,492.69
beta_18	-	-	-	-0.82	-1.49	-95.70	-416.34	541.96	2,961.11
beta_19	-	-	-	2.85	17.58	-35.30	-401.32	186.42	2,537.05
beta_20	-	-	-	-3.79	-22.99	-82.11	-31.28	703.05	-14.25
beta_21	-	-	-	-	18.56	85.16	-329.57	-2,153.35	6,617.67
beta_22	-	-	-	-	2.57	67.63	-10.33	-430.87	-131.70
beta_23	-	-	-	-	3.56	40.27	-51.30	-884.99	-1,111.24
beta_24	-	-	-	-	2.74	80.62	332.37	-718.87	-4,054.99
beta_25	-	-	-	-	-1.90	28.80	314.38	-973.99	-5,276.56
beta_26	-	-	-	-	-4.63	25.97	384.07	-57.85	-3,992.56
beta_27	-	-	-	-	7.35	39.11	-134.75	-1,258.15	1,575.35
beta_28	-	-	-	-	-	-13.75	227.93	1,883.44	-9,884.64
beta_29	-	-	-	-	-	-20.21	-2.83	254.90	371.71
beta_30	-	-	-	-	-	-1.01	87.37	666.61	-262.32
beta_31	-	-	-	-	-	-20.06	-61.24	624.72	2,838.12
beta_32	-	-	-	-	-	-15.40	-206.51	772.38	5,816.64
beta_33	-	-	-	-	-	-1.45	-104.88	844.35	4,924.85
beta_34	-	-	-	-	-	-9.09	-214.27	-125.51	4,564.44
beta_35	-	-	-	-	-	-6.67	149.52	1,223.41	-3,929.78
beta_36	-	-	-	-	-	-	-58.46	-880.56	8,820.64
beta_37	-	-	-	-	-	-	0.62	-116.55	-696.53
beta_38	-	-	-	-	-	-	-15.30	-166.09	1,056.63
beta_39	-	-	-	-	-	-	-23.48	-400.28	-1,058.27
beta_40	-	-	-	-	-	-	48.25	-195.74	-2,693.05
beta_41	-	-	-	-	-	-	30.77	-509.06	-4,725.53
beta_42	-	-	-	-	-	-	15.60	-320.34	-2,116.00
beta_43	-	-	-	-	-	-	50.88	119.12	-3,791.57
beta_44	-	-	-	-	-	-	-48.34	-620.95	4,530.61
beta_45	-	-	-	-	-	-	-	170.87	-4,302.73
beta_46	-	-	-	-	-	-	-	35.23	485.82
beta_47	-	-	-	-	-	-	-	-22.87	-565.38
beta_48	-	-	-	-	-	-	-	137.41	-60.39
beta_49	-	-	-	-	-	-	-	-28.40	779.26
beta_50	-	-	-	-	-	-	-	118.09	1,346.36
beta_51	-	-	-	-	-	-	-	85.97	1,942.49
beta_52	-	-	-	-	-	-	-	50.57	176.28
beta_53	-	-	-	-	-	-	-	-32.36	1,961.69
beta_54	-	-	-	-	-	-	-	129.21	-2,600.80
beta_55	-	-	-	-	-	-	-	-	881.94
beta_56	-	-	-	-	-	-	-	-	-108.78
beta_57	-	-	-	-	-	-	-	-	44.44
beta_58	-	-	-	-	-	-	-	-	207.36
beta_59	-	-	-	-	-	-	-	-	-259.90
beta_60	-	-	-	-	-	-	-	-	8.53
beta_61	-	-	-	-	-	-	-	-	-390.28
beta_62	-	-	-	-	-	-	-	-	-266.82

beta_63	-	-	-	-	-	-	-	-	82.93
beta_64	-	-	-	-	-	-	-	-	-450.06
beta_65	-	-	-	-	-	-	-	-	599.99

We can see from this that as the model complexity increases, the coefficients increase wildly.

We will see below that d=5 seems to give a particularly good fit to our data. In the table above we can see that this fit has reasonably sized coefficients $\boldsymbol{\beta}$. We will now look at the confidence intervals for these. I will look at the 95 % confidence interval.

[111]:
```
def plot_beta_ci(model,data,confidence=0.90):
    CIs = model.get_beta_CIs(confidence)
    fig = plt.figure(figsize=(8, 6))

    beta_label = []
    p = CIs.shape[0]
    for i in range(p):
        beta_label.append(fr"$\beta_{{{i}}}$")
        plt.plot(CIs[i,:],(i,i), 'ro-')
    plt.yticks(np.arange(p), beta_label)
    plt.xlabel("Value")
    plt.title(fr" The {confidence*100} % confidence intervals for beta")
    plt.show()

data.create_design_matrix(5)
data.split_dataset(test_fraction)
data.scale_dataset()
ols_model.fit(data.X_train,data.z_train)
confidence = 0.95
plot_beta_ci(ols_model,data,confidence=0.95)
print("Figure 5: A plot showing the confidence intervals for the coefficients␣
 ↪beta when the fit is made with a polynomial of the fifth degree and OLS.")
```

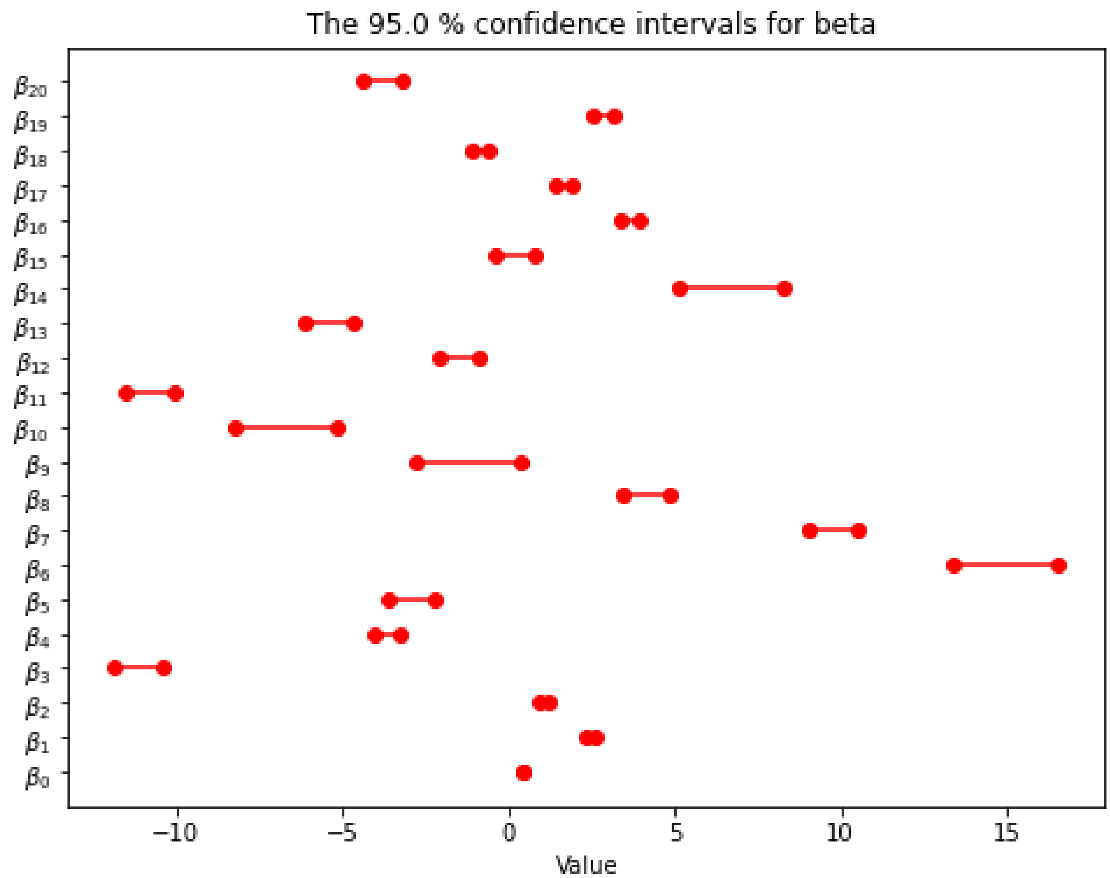


Figure 5: A plot showing the confidence intervals for the coefficients beta when the fit is made with a polynomial of the fifth degree and OLS.

Let us look move on to look at the bias-variance trade-off usinfg the bootstrap method.

5.1.2 Adding Bootstrapping

To explore the bias-variance trade-off we can use the bootstrap method. I now create my model for increasing polynomial degree, and perform the bootstrap method for each polynomial in hope of finding an optimal polynomial degree. I explore how the error on the training as well as the test set relates to the complexity of the model, i.e. the degree of the polynomial of the model.

[112]:
```
from sklearn.preprocessing import StandardScaler

def bias_var_with_bootstrap(model,B=100,n=100,min_degree=1,max_degree=10):
    test_data = cd.CreateData(n,seed=13)# Create separate test data so I can␣
 ↪test on same data every time
    train_data = cd.CreateData(n,seed=9)
    degrees = np.array(range(min_degree,max_degree+1))
    test_data.add_normal_noise(0,variance)
```

```
    train_data.add_normal_noise(0,variance)

    scaler = StandardScaler()

    mse_bs = np.zeros((len(degrees),2))
    print("Mean square error on static test set for B=%d bootstraps." %(B))
    for i in range(len(degrees)):
        train_data.create_design_matrix(degrees[i])
        test_data.create_design_matrix(degrees[i])
        scaler.fit(train_data.X[:,1:])
        X_train_scaled = scaler.transform(train_data.X[:,1:])
        X_test_scaled = scaler.transform(test_data.X[:,1:])
        train_data.X = np.hstack((np.ones((train_data.X.
↪shape[0],1)),X_train_scaled))
        test_data.X = np.hstack((np.ones((test_data.X.
↪shape[0],1)),X_test_scaled))
        mse_bs[i,:] = model.bootstrap_fit(train_data.X,np.ravel(train_data.
↪z_mesh),test_data.X,np.ravel(test_data.z_mesh),B)
        print(f"d=%d: MSE(test set): %f " %(degrees[i],mse_bs[i,0]))

    fig = plt.figure(figsize=(8, 6))
    plt.plot(degrees,mse_bs[:,1], color='#117733', label='$MSE_{train}$')
    plt.plot(degrees,mse_bs[:,0], color='#CC6677', label='$MSE_{test}$')
    plt.xlabel('complexity (d)')
    plt.ylabel('MSE')
    plt.title('MSE for increasing polynimial degree in model')
    plt.legend()
    print('The minimum MSE is: {} found for polynomial degree d = {}'.
↪format(min(mse_bs[:,0]),degrees[np.argmin(mse_bs[:,0])]))
    plt.show()
```

[113]:
```
ols_model = ols.OrdinaryLeastSquares()
bias_var_with_bootstrap(ols_model,B=1000,n=25,max_degree=10)
print("Figure 6: Plot showing the bias-variance trade-off for the OLS model as␣
↪the complexity of the model increases.")
```

```
Mean square error on static test set for B=1000 bootstraps.
d=1: MSE(test set): 0.038591
d=2: MSE(test set): 0.036162
d=3: MSE(test set): 0.020310
d=4: MSE(test set): 0.017509
d=5: MSE(test set): 0.015825
d=6: MSE(test set): 0.016213
d=7: MSE(test set): 0.016748
d=8: MSE(test set): 0.022785
d=9: MSE(test set): 0.066367
d=10: MSE(test set): 0.072707
```

```
The minimum MSE is: 0.01582527389486775 found for polynomial degree d = 5
```

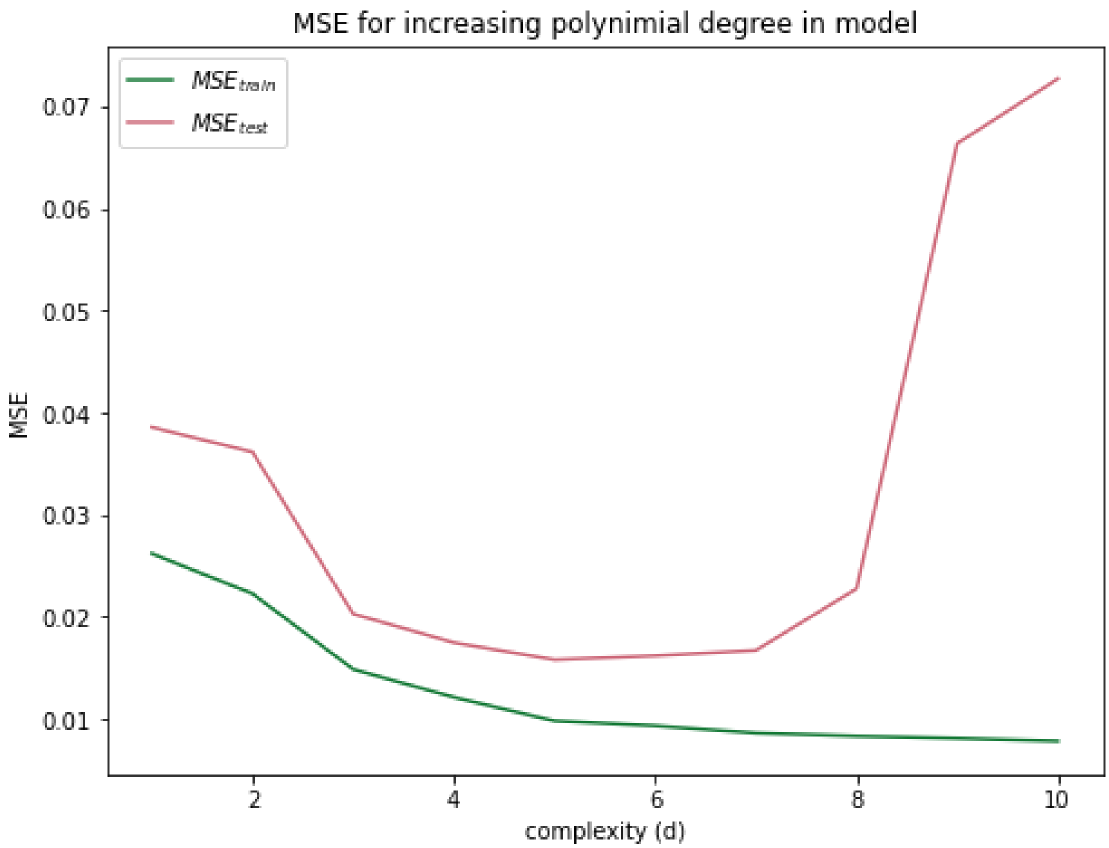


Figure 6: Plot showing the bias-variance trade-off for the OLS model as the complexity of the model increases.

As expected the error on the training set can be reduced indefinitely by increasing the model complexity, but the error on the test set shows the overfitting that is happening. This matches what we saw for the bias-variance trade-off. The bias is reduced with model complexity, but the variance will increase.

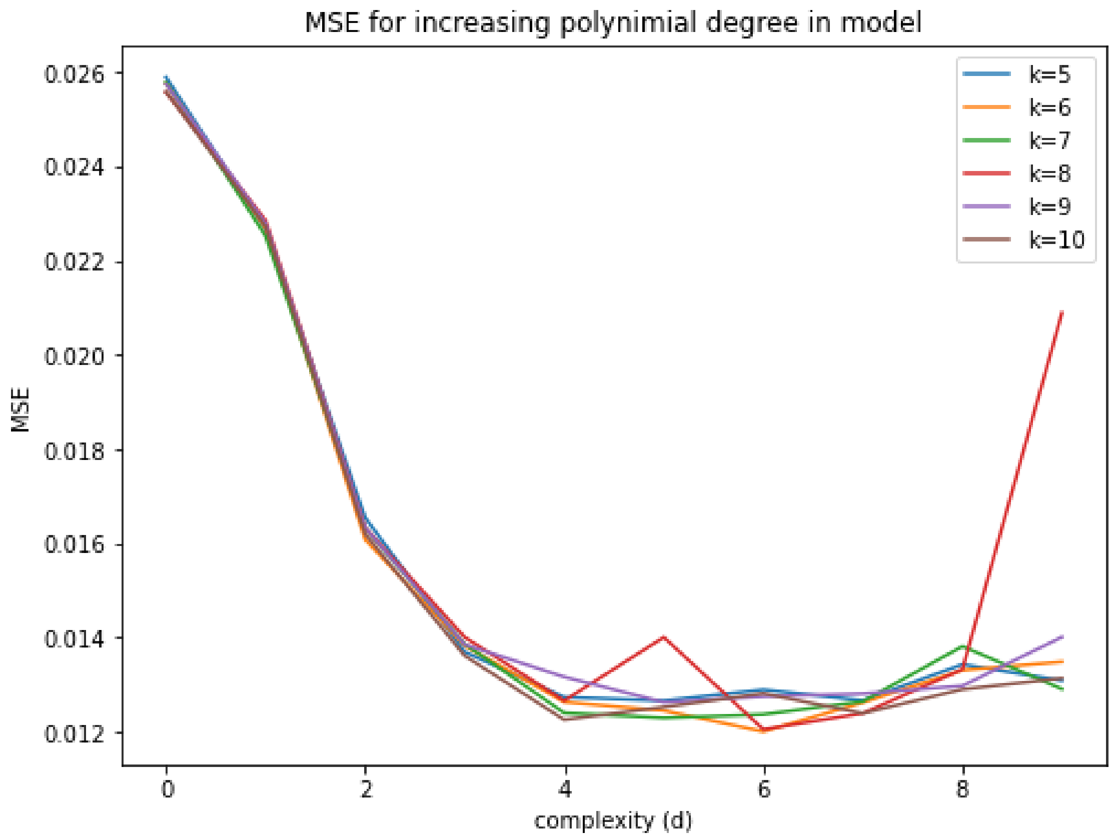
5.1.3 k-Fold Cross-validation

Now over to another resampling technique, k-fold cross-validation (kFold).

[115]:
```python
def mse_with_k_fold(model,data,kmin=5,kmax=10,n=100,max_degree=10):
    mse_cv = np.zeros(max_degree)
    mse_min = np.zeros((kmax+1-kmin,2))
    d_array = np.array(range(max_degree))

    plt.figure(figsize=(8, 6))
    for k in range(kmin,kmax+1):
        for d in range(max_degree):
```

```
            data.create_design_matrix(d+1)
            data.split_dataset(test_fraction)
            data.scale_dataset(type='standard')
            mse_cv[d] = model.k_fold_cv(data.X_train,data.
 ↪z_train,k,shuffle=True)
        mse_min[k-kmin,:] = (min(mse_cv),np.argmin(mse_cv)+1)
        print("k=%d - Minimum MSE=%f found for d=%d" %(k,min(mse_cv),np.
 ↪argmin(mse_cv)+1))
        plt.plot(d_array,mse_cv, label=f'k={k}')

    plt.xlabel('complexity (d)')
    plt.ylabel('MSE')
    plt.title('MSE for increasing polynimial degree in model')
    plt.legend()

ols_cv = ols.OrdinaryLeastSquares()
cv_data = cd.CreateData(25)
cv_data.add_normal_noise(0,variance)
mse_with_k_fold(ols_cv,cv_data,kmin=5,kmax=10,n=100,max_degree=10)
print("Figure 7: Using c-fold cross-validation with varying number of folds k␣
 ↪to explore the mean square error as a function of model complexity.")
```

```
k=5 - Minimum MSE=0.012663 found for d=6
k=6 - Minimum MSE=0.012009 found for d=7
k=7 - Minimum MSE=0.012299 found for d=6
k=8 - Minimum MSE=0.012054 found for d=7
k=9 - Minimum MSE=0.012631 found for d=6
k=10 - Minimum MSE=0.012259 found for d=5
Figure 7: Using c-fold cross-validation with varying number of folds k to
explore the mean square error as a function of model complexity.
```



We see that all the tested values for `k` give similar behavior, with a polynomial of degree between five and seven performing best. It seems the specific value of k is not crucial in this range and for this size dataset. The value for the mean square error is very comparable to what we got for bootstrapping.

5.2 Ridge Regression

Let us move on to the next model, ridge regression. I use the same synthetic data from the Franke function. First I compare to scikit-learn's `Ridge`.

[313]:
```
from sklearn.linear_model import Ridge
import Code.RidgeRegression as rr

lmbd = 0.01
rr_model = rr.RidgeRegression()
rr_model.fit(data.X_train,data.z_train,alpha=lmbd)
z_hat = rr_model.predict(data.X_test)

sk_rr = Ridge(alpha=lmbd,fit_intercept=False)
sk_rr.fit(data.X_train,data.z_train)
skl_z_hat = sk_rr.predict(data.X_test)
```

```
print("The arrays are about the same: ", np.allclose(z_hat,skl_z_hat))
print("The mean squared error between my result and that of scikit-learn is: ",␣
↪rr_model.mean_square_error(z_hat,skl_z_hat))
```

```
The arrays are about the same:  True
The mean squared error between my result and that of scikit-learn is:
3.4439231813169366e-24
```

My implementation appears to match that of scikit-learn as we would hope. Let us explore this model further. ### The Coefficients $\boldsymbol{\beta}$ Like for OLS we begin by exploring the coefficients. As seen below, the coefficients do not grow wildly as the complexity of the model grows, as the ridge model puts a restraint on the absolute size of the coefficients according to the parameter λ.

[116]:
```
print("Table 4: The coefficients beta_i for fitting a polynomial of degree=d␣
↪using ridge regression.")
evaluate_betas(rr_model,data,2,10)
```

Table 4: The coefficients beta_i for fitting a polynomial of degree=d using ridge regression.

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10
beta_0	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41	0.41
beta_1	-0.30	-0.15	0.84	0.88	1.07	1.14	1.11	1.09	1.10
beta_2	-0.21	0.33	0.64	0.64	0.68	0.68	0.68	0.70	0.73
beta_3	0.04	-0.45	-4.49	-4.03	-4.36	-4.39	-4.09	-3.87	-3.78
beta_4	0.19	0.42	-0.20	-0.28	-0.78	-0.64	-0.48	-0.51	-0.59
beta_5	-0.11	-1.84	-2.92	-2.63	-2.30	-2.22	-2.28	-2.34	-2.34
beta_6	-	0.28	5.74	2.87	2.65	2.21	1.52	1.06	0.85
beta_7	-	0.06	1.09	2.10	2.08	1.03	0.60	0.65	0.69
beta_8	-	-0.27	0.11	0.07	0.40	0.26	0.29	0.38	0.36
beta_9	-	1.26	2.64	0.99	0.21	0.21	0.26	0.19	0.08
beta_10	-	-	-2.37	2.10	2.12	2.59	2.75	2.64	2.48
beta_11	-	-	-0.64	-2.63	-0.28	0.81	0.56	0.27	0.27
beta_12	-	-	0.02	1.24	1.69	1.87	1.79	1.89	2.09
beta_13	-	-	-0.26	-1.70	-1.19	-1.08	-1.12	-1.05	-0.97
beta_14	-	-	-0.57	2.48	1.80	1.43	1.48	1.55	1.51
beta_15	-	-	-	-2.07	-0.90	-0.35	0.53	1.08	1.29
beta_16	-	-	-	0.66	-2.31	-0.96	-0.12	-0.08	-0.16
beta_17	-	-	-	0.47	-1.51	-0.35	0.08	0.02	0.11
beta_18	-	-	-	-1.22	-0.83	-0.34	-0.30	-0.33	-0.21
beta_19	-	-	-	1.39	-0.05	-0.17	-0.34	-0.45	-0.46
beta_20	-	-	-	-1.69	0.87	0.75	0.76	0.96	1.11
beta_21	-	-	-	-	-0.82	-1.87	-1.60	-0.93	-0.45
beta_22	-	-	-	-	0.97	-1.53	-0.87	-0.41	-0.40
beta_23	-	-	-	-	1.36	-1.16	-0.62	-0.50	-0.67
beta_24	-	-	-	-	-0.26	-1.06	-0.89	-0.98	-1.03
beta_25	-	-	-	-	0.05	-0.05	0.06	-0.13	-0.28
beta_26	-	-	-	-	0.73	0.49	0.36	0.21	0.14

beta_27	-	-	-	-	-1.44	-0.34	-0.38	-0.22	0.04
beta_28	-	-	-	-	-	0.42	-1.60	-1.77	-1.43
beta_29	-	-	-	-	-	0.98	-0.74	-0.48	-0.32
beta_30	-	-	-	-	-	1.20	-0.28	-0.15	-0.35
beta_31	-	-	-	-	-	0.45	-0.49	-0.48	-0.69
beta_32	-	-	-	-	-	0.20	0.03	-0.08	-0.34
beta_33	-	-	-	-	-	-0.16	0.33	0.36	0.14
beta_34	-	-	-	-	-	0.27	0.41	0.36	0.32
beta_35	-	-	-	-	-	-0.69	-0.77	-0.91	-0.76
beta_36	-	-	-	-	-	-	1.14	-0.94	-1.32
beta_37	-	-	-	-	-	-	0.69	-0.20	-0.06
beta_38	-	-	-	-	-	-	0.86	0.34	0.30
beta_39	-	-	-	-	-	-	0.26	0.10	0.09
beta_40	-	-	-	-	-	-	0.40	0.42	0.29
beta_41	-	-	-	-	-	-	-0.01	0.37	0.28
beta_42	-	-	-	-	-	-	-0.46	0.22	0.25
beta_43	-	-	-	-	-	-	0.08	0.21	0.23
beta_44	-	-	-	-	-	-	0.07	-0.67	-0.91
beta_45	-	-	-	-	-	-	-	1.42	-0.29
beta_46	-	-	-	-	-	-	-	0.32	0.11
beta_47	-	-	-	-	-	-	-	0.47	0.60
beta_48	-	-	-	-	-	-	-	-0.11	0.40
beta_49	-	-	-	-	-	-	-	0.05	0.54
beta_50	-	-	-	-	-	-	-	0.13	0.57
beta_51	-	-	-	-	-	-	-	-0.32	0.22
beta_52	-	-	-	-	-	-	-	-0.49	0.02
beta_53	-	-	-	-	-	-	-	0.04	0.08
beta_54	-	-	-	-	-	-	-	0.55	-0.36
beta_55	-	-	-	-	-	-	-	-	1.35
beta_56	-	-	-	-	-	-	-	-	-0.04
beta_57	-	-	-	-	-	-	-	-	0.13
beta_58	-	-	-	-	-	-	-	-	-0.31
beta_59	-	-	-	-	-	-	-	-	-0.36
beta_60	-	-	-	-	-	-	-	-	-0.16
beta_61	-	-	-	-	-	-	-	-	-0.10
beta_62	-	-	-	-	-	-	-	-	-0.36
beta_63	-	-	-	-	-	-	-	-	-0.33
beta_64	-	-	-	-	-	-	-	-	0.02
beta_65	-	-	-	-	-	-	-	-	0.74

Now let us look at the confidence intervals for these β values. Compared with the CIs for $\boldsymbol{\beta}^{\texttt{OLS}}$ in figure number these are more narrow (note the different x axis).

[117]:
```
data.create_design_matrix(5)
data.split_dataset(test_fraction)
data.scale_dataset()
rr_model.fit(data.X_train,data.z_train)
```

```
confidence = 0.95
plot_beta_ci(rr_model,data,confidence=0.95)
print("Figure 8: A plot showing the confidence intervals for the coefficients␣
↪beta when the fit is made with a polynomial of the fifth degree and ridge␣
↪regression.")
```

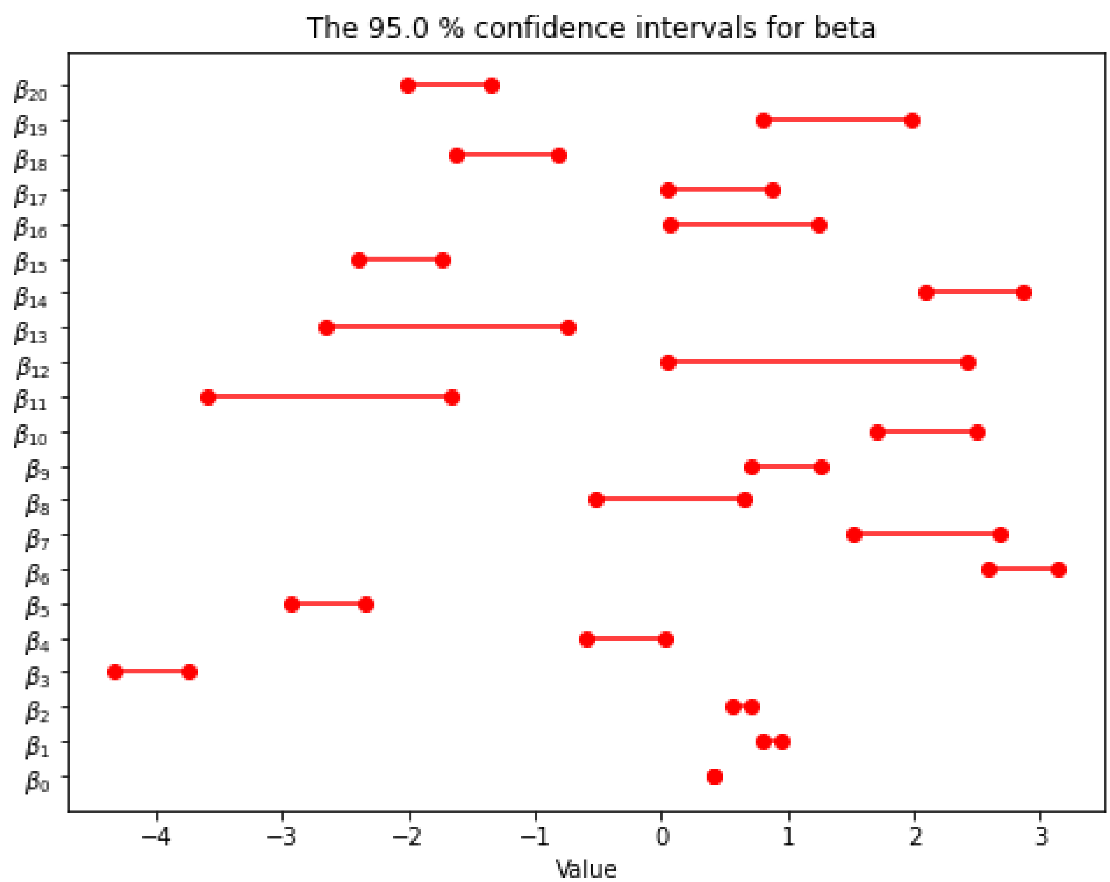


Figure 8: A plot showing the confidence intervals for the coefficients beta when the fit is made with a polynomial of the fifth degree and ridge regression.

5.2.1 Bias-Variance Trade-off and Choice of λ

Repeating the bootstrapping we did on the OLS model on the new model with ridge regression we see that for the same data and model complexity we do not appear to get overfitting issues (see figure 9). I have gone all the way to a polynomial of degree 20 to show this lack of overfitting issues.

What we are seeing is the ridge model keeping the β values from increasing wildly, and effectively reducing model complexity when we increase the polynomial degree.

[118]:
```
bias_var_with_bootstrap(rr_model,B=1000,n=25,max_degree=20)
```

```python
print("Figure 9: Plot showing the bias-variance trade-off for the ridge as the␣
↪complexity of the model increases.")
```

```
Mean square error on static test set for B=1000 bootstraps.
d=1: MSE(test set): 0.038587
d=2: MSE(test set): 0.036084
d=3: MSE(test set): 0.020693
d=4: MSE(test set): 0.020068
d=5: MSE(test set): 0.018653
d=6: MSE(test set): 0.017267
d=7: MSE(test set): 0.016312
d=8: MSE(test set): 0.015718
d=9: MSE(test set): 0.015466
d=10: MSE(test set): 0.015275
d=11: MSE(test set): 0.015246
d=12: MSE(test set): 0.015248
d=13: MSE(test set): 0.015192
d=14: MSE(test set): 0.015181
d=15: MSE(test set): 0.015313
d=16: MSE(test set): 0.015397
d=17: MSE(test set): 0.015542
d=18: MSE(test set): 0.015680
d=19: MSE(test set): 0.015817
d=20: MSE(test set): 0.015901
The minimum MSE is: 0.015180754189300875 found for polynomial degree d = 14
```

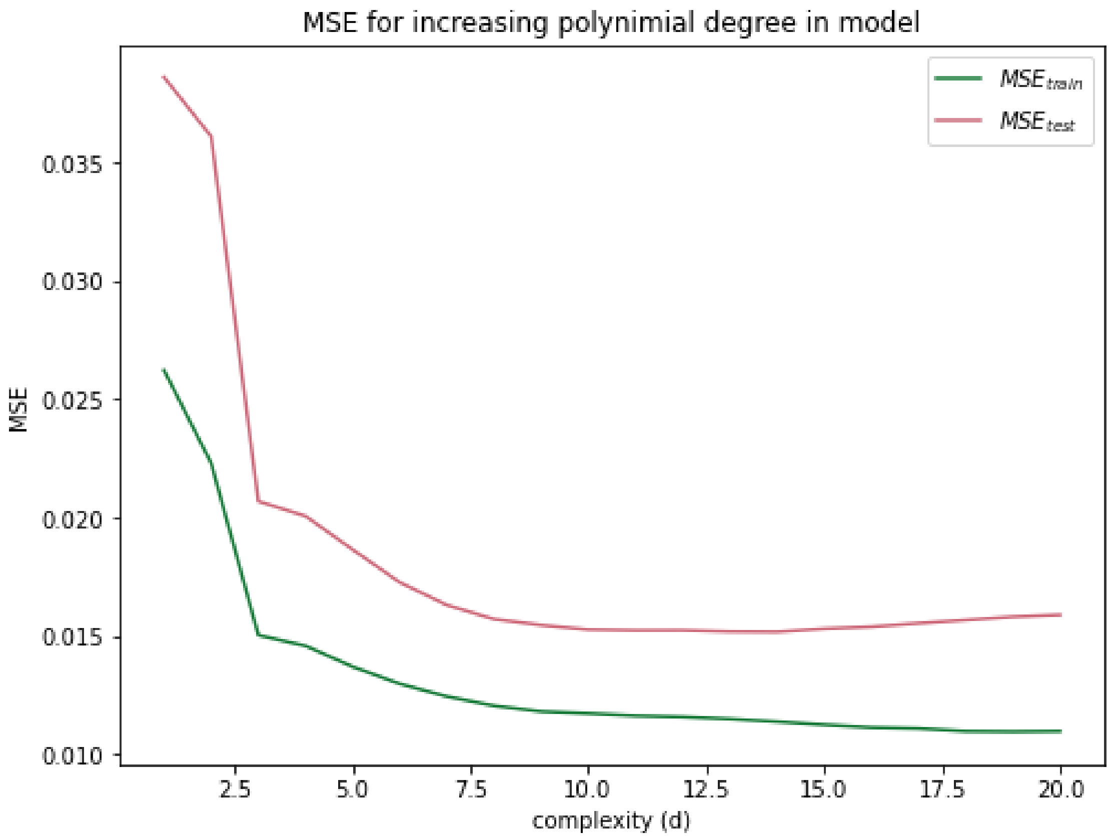


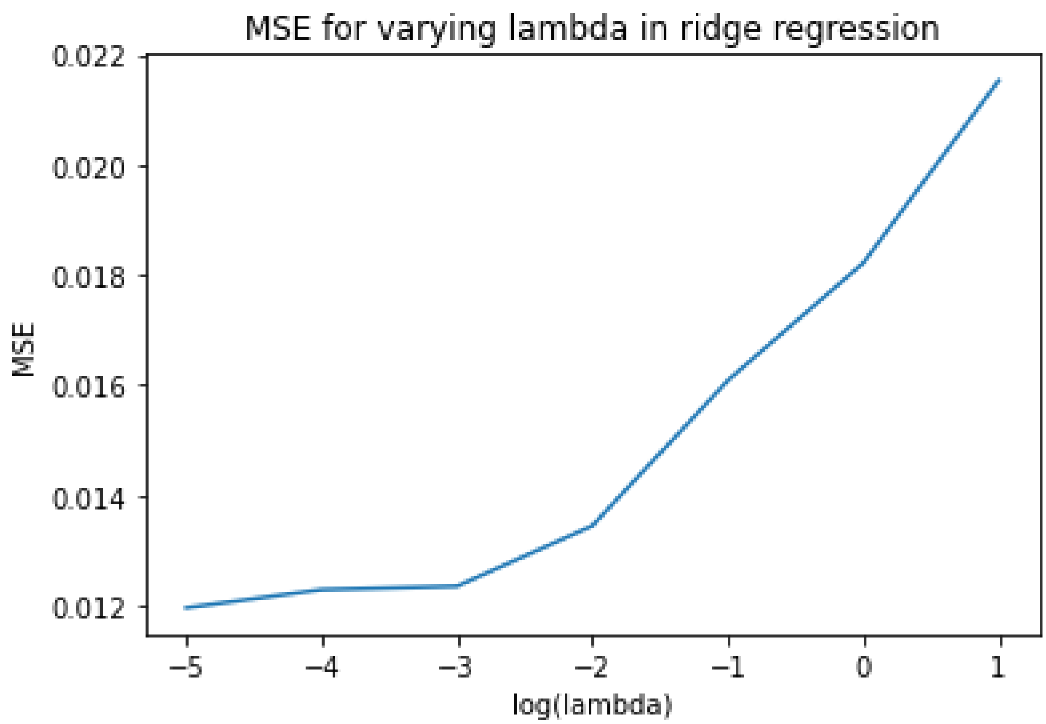
Figure 9: Plot showing the bias-variance trade-off for the ridge as the complexity of the model increases.

Using k=10 folds, let us now use k-fold cross-validation to try and find an optimal value for the ridge parameter λ.

[119]:
```
def lambda_kfold(model,data,lmbd_values,k=10):
    mse_r2_lmbd = np.zeros((len(lmbd_values),2))

    for a in range(len(lmbd_values)):
        mse_r2_lmbd[a,:] = model.k_fold_cv(data.X_train,data.
↪z_train,k,alpha=lmbd_values[a],shuffle=True)
        print(f"MSE,r^2={mse_r2_lmbd[a]} found for lambda={lmbd_values[a]}")

    plt.plot(np.log10(lmbd_values),mse_r2_lmbd[:,0])
    plt.xlabel('log(lambda)')
    plt.ylabel('MSE')
    plt.title('MSE for varying lambda in ridge regression')
    plt.show()

powers = np.array(range(-5,2))
```

```
ones = np.ones_like(powers)*10.0
alpha_values = np.power(ones,powers)

ridge_data = cd.CreateData(n=20)
ridge_data.add_normal_noise(0,variance)
ridge_data.create_design_matrix(d=5)
ridge_data.split_dataset(test_fraction)
ridge_data.scale_dataset(type='standard')
rr_model = rr.RidgeRegression()
lambda_kfold(rr_model,ridge_data,alpha_values,k=5)
print("Figure 10: MSE for ridge regression when varying the parameter lambda.")
```

```
MSE,r^2=[0.01195876 0.83831175] found for lambda=1e-05
MSE,r^2=[0.01228813 0.83151661] found for lambda=0.0001
MSE,r^2=[0.0123494  0.82102821] found for lambda=0.001
MSE,r^2=[0.01344776 0.80612825] found for lambda=0.01
MSE,r^2=[0.01608924 0.75381219] found for lambda=0.1
MSE,r^2=[0.01821817 0.71085962] found for lambda=1.0
MSE,r^2=[0.0215347  0.62599892] found for lambda=10.0
```




```
Figure 10: MSE for ridge regression when varying the parameter lambda.
```

We see that the error is smallest for small λ, indicating that $\lambda = 0$ is optimal, which is equivalent to OLS. Let us explore the behaviour for a more complex model. Here the ridge models tendency to limit overfitting should show itself.

[120]:
```python
def lambda_with_bootstrap(model,data,lambda_values,B=100):
    mse_bs = np.zeros((len(lambda_values),2))
    print("Mean square error on static test set for B=%d bootstraps." %(B))
    for a in range(len(lambda_values)):

        mse_bs[a,:] = model.bootstrap_fit(data.X_train,data.z_train,data.
 ↪X_test,data.z_test,B,lambda_values[a])
        print(f"lambda={lambda_values[a]}: MSE(test set): {mse_bs[a,0]}")

    fig = plt.figure(figsize=(8, 6))
    plt.plot(np.log10(lambda_values),mse_bs[:,1], color='#117733',␣
 ↪label='$MSE_{train}$')
    plt.plot(np.log10(lambda_values),mse_bs[:,0], color='#CC6677',␣
 ↪label='$MSE_{test}$')
    plt.xlabel('log(lambda)')
    plt.ylabel('MSE')
    plt.title('MSE for varying parameter lambda for ridge model')
    plt.legend()
    print('The minimum MSE is: {} found for lambda value = {}'.
 ↪format(min(mse_bs[:,0]),lambda_values[np.argmin(mse_bs[:,0])]))
    plt.show()

powers = np.array(range(-5,2))
ones = np.ones_like(powers)*10.0
alpha_values = np.power(ones,powers)
ridge_data = cd.CreateData(n=25)
ridge_data.add_normal_noise(0,variance)

degree = 25
ridge_data.create_design_matrix(d=degree)
ridge_data.split_dataset(test_fraction)
ridge_data.scale_dataset(type='standard')
rr_model = rr.RidgeRegression()
lambda_with_bootstrap(rr_model,ridge_data,alpha_values,B=100)
print(f"Figure 11: A plot showing the mean square error on training and test␣
 ↪sets for varying lambda values for a high complexity model (d={degree}).")
```

```
Mean square error on static test set for B=100 bootstraps.
lambda=1e-05: MSE(test set): 0.018507699374045304
lambda=0.0001: MSE(test set): 0.014287970247964394
lambda=0.001: MSE(test set): 0.013835353398854982
lambda=0.01: MSE(test set): 0.014761231251062124
lambda=0.1: MSE(test set): 0.017442618062011143
lambda=1.0: MSE(test set): 0.02165589772911971
lambda=10.0: MSE(test set): 0.02823785078741676
The minimum MSE is: 0.013835353398854982 found for lambda value = 0.001
```

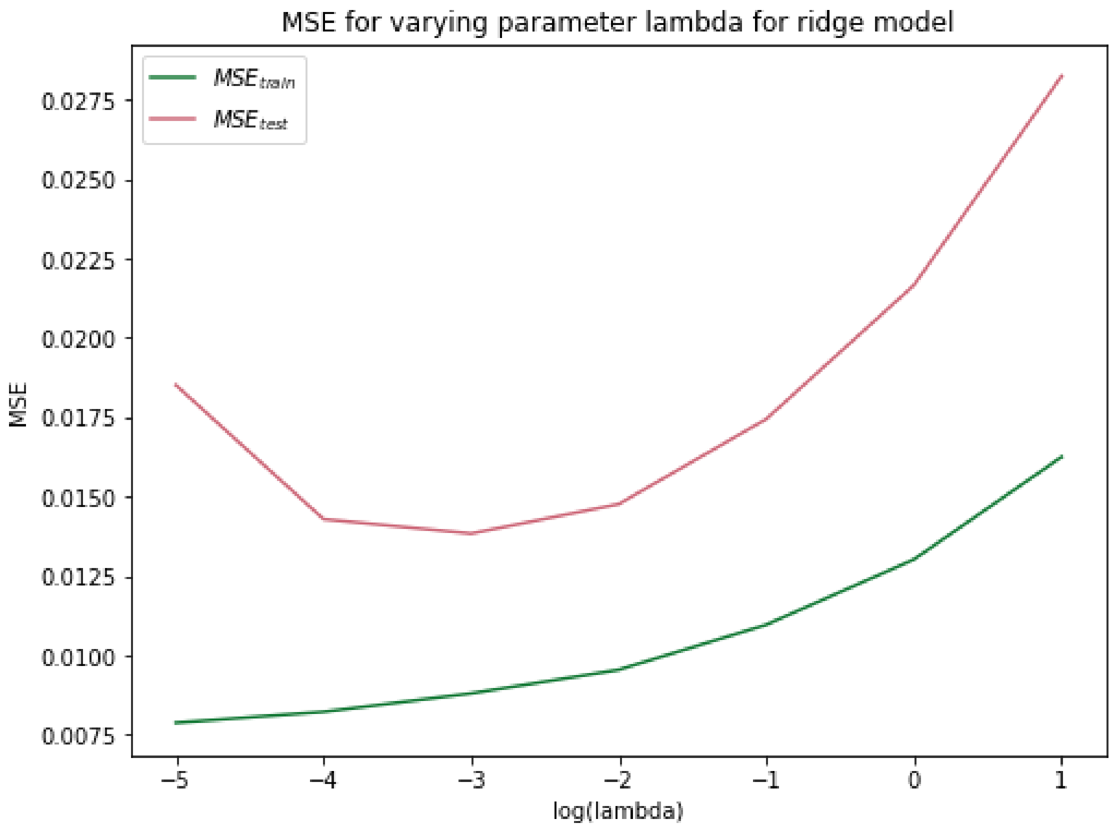


Figure 11: A plot showing the mean square error on training and test sets for varying lambda values for a high complexity model (d=25).

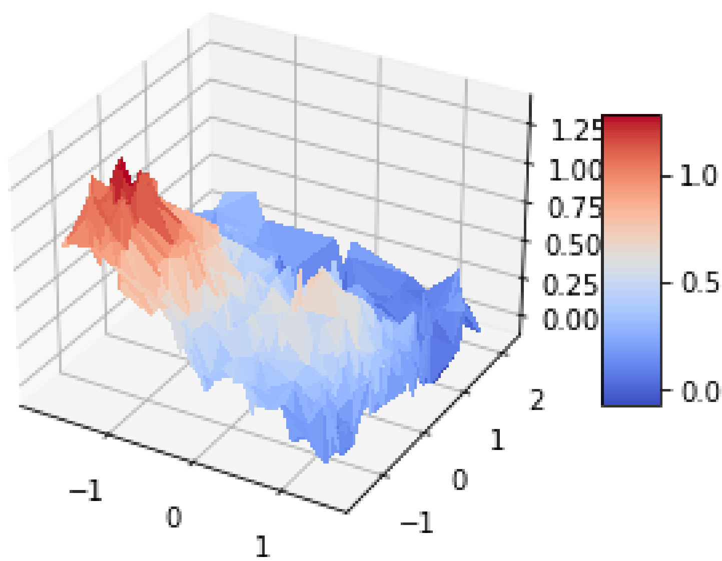
The optimal parameter for this model created with a polynomial of degree d=25 is found as $\lambda = 0.001$. For a model with lower complexity OLS seems to be a better option. For our Franke data a lower complexity model does the job, so no need to use ridge over OLS for this.

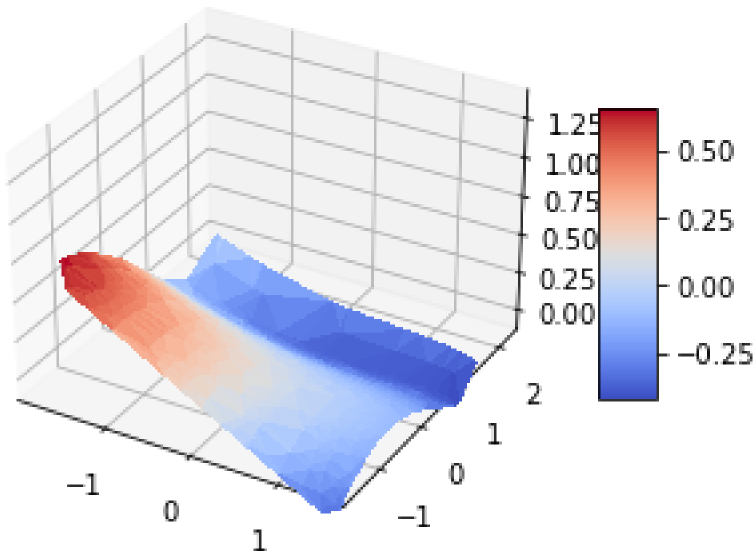
5.3 Lasso Regression

For Lasso regression I will be using scikit-learn's `Lasso`. I have wrapped it in a class `LassoRegression` so I can use the same methods as for OLS and ridge.

[123]:
```
import Code.LassoRegression as lr

# data
n = 50
degree = 7
lasso_data = cd.CreateData(n=n)
lasso_data.add_normal_noise(0,variance)
lasso_data.create_design_matrix(d=degree)
lasso_data.split_dataset(test_fraction)
lasso_data.scale_dataset(type='standard')
```

```
# model
lmb = 0.001
lr_model = lr.LassoRegression(alpha=lmb)
lr_model.fit(lasso_data.X_train,lasso_data.z_train)
z_hat = lr_model.predict(lasso_data.X_test)

# explore
data.plot_data(X=lasso_data.X_test,z=lasso_data.z_test)
data.plot_data(X=lasso_data.X_test,z=z_hat)
print(f"Mean square error for fit: {lr_model.mean_square_error(z_hat,lasso_data.
 ↪z_test)}")
print(f"r^2 score for fit: {lr_model.r2(z_hat,lasso_data.z_test)}")
print(f"Figure 12: The data for test set plotted above, and the result from␣
 ↪estimation with lasso regression with lambda={lmb} and polynomial␣
 ↪degree={degree} plotted below.")
```






```
Mean square error for fit: 0.21004592218452145
r^2 score for fit: -2.3637489731351966
Figure 12: The data for test set plotted above, and the result from estimation
with lasso regression with lambda=0.001 and polynomial degree=7 plotted below.
```

Lasso regression, like ridge, places a restraint on the possible values for β. We see in figure 12 that even with a small value for λ the complexity of the model is noticebly reduced.

[125]:

```python
N_array = np.array([5,10,15,25,50,100,250,500])
r2_df,mse_df = evaluate_model(lr_model,N_array,2,20)
print("Table 5: R^2 score for increasing polynomial degree and number of datapoints.")
display(r2_df)
print("Table 6: Mean squared error for increasing polynomial degree and number of datapoints.")
display(mse_df)
```

Table 5: R^2 score for increasing polynomial degree and number of datapoints.

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	\
N=5x5	-8.297	-7.912	-8.007	-7.963	-7.886	-7.786	-7.613	-7.300	-7.050	
N=10x10	-1.977	-1.975	-2.063	-2.060	-2.020	-1.968	-1.950	-1.932	-1.929	
N=15x15	-1.266	-1.233	-1.249	-1.280	-1.306	-1.316	-1.325	-1.332	-1.338	
N=25x25	-2.559	-2.178	-1.939	-1.908	-1.916	-1.918	-1.916	-1.915	-1.912	
N=50x50	-2.649	-2.468	-2.383	-2.364	-2.366	-2.364	-2.344	-2.331	-2.328	
N=100x100	-1.644	-1.452	-1.374	-1.357	-1.360	-1.358	-1.355	-1.354	-1.351	
N=250x250	-2.057	-1.794	-1.708	-1.694	-1.698	-1.692	-1.687	-1.682	-1.677	
N=500x500	-2.243	-1.952	-1.858	-1.839	-1.843	-1.840	-1.835	-1.829	-1.823	

	d=11	d=12	d=13	d=14	d=15	d=16	d=17	d=18	d=19	d=20
N=5x5	-6.774	-6.588	-6.440	-6.227	-6.092	-5.933	-5.825	-5.717	-5.643	-5.566
N=10x10	-1.926	-1.927	-1.919	-1.921	-1.920	-1.917	-1.915	-1.917	-1.918	-1.921
N=15x15	-1.339	-1.340	-1.340	-1.340	-1.340	-1.340	-1.339	-1.339	-1.339	-1.339
N=25x25	-1.897	-1.883	-1.872	-1.865	-1.861	-1.858	-1.852	-1.848	-1.846	-1.843
N=50x50	-2.321	-2.318	-2.316	-2.315	-2.315	-2.314	-2.313	-2.312	-2.311	-2.308
N=100x100	-1.348	-1.344	-1.343	-1.341	-1.341	-1.339	-1.339	-1.338	-1.338	-1.338
N=250x250	-1.667	-1.660	-1.656	-1.655	-1.656	-1.655	-1.654	-1.653	-1.652	-1.652
N=500x500	-1.812	-1.806	-1.802	-1.800	-1.800	-1.799	-1.798	-1.797	-1.796	-1.795

Table 6: Mean squared error for increasing polynomial degree and number of datapoints.

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11	d=12	d=13	d=14	d=15	d=16	d=17	d=18	d=19	d=20
N=5x5	0.648	0.629	0.626	0.621	0.621	0.624	0.623	0.619	0.616	0.612	0.608	0.609	0.618	0.624	0.629	0.633	0.637	0.640	0.643
N=10x10	0.189	0.184	0.182	0.182	0.183	0.183	0.183	0.183	0.183	0.183	0.183	0.183	0.182	0.182	0.182	0.182	0.182	0.182	0.181
N=15x15	0.135	0.135	0.136	0.137	0.137	0.136	0.135	0.134	0.134	0.134	0.134	0.134	0.134	0.134	0.134	0.134	0.134	0.134	0.134
N=25x25	0.172	0.164	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.160	0.160	0.160	0.160	0.160	0.160	0.160	0.160	0.160	0.160
N=50x50	0.218	0.212	0.211	0.210	0.210	0.210	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209	0.209
N=100x100	0.164	0.159	0.158	0.158	0.158	0.158	0.158	0.158	0.158	0.158	0.157	0.157	0.157	0.157	0.157	0.157	0.157	0.157	0.157
N=250x250	0.191	0.184	0.183	0.183	0.183	0.183	0.183	0.183	0.183	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182
N=500x500	0.190	0.183	0.183	0.183	0.183	0.183	0.183	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182	0.182

From this we see that the performance of the lasso model is very stable for increasing model complexity. The r^2 score is very poor, while the mse is better. Both are markedly worse than for the OLS model.

5.3.1 The Coefficients β

[126]:
```python
print("Table 7: The coefficients beta_i for fitting a polynomial of degree=d using lasso regression.")
evaluate_betas(lr_model,data,2,10)
```

Table 7: The coefficients beta_i for fitting a polynomial of degree=d using lasso regression.

	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10
beta_0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
beta_1	-0.26	-0.30	-0.27	-0.25	-0.24	-0.23	-0.20	-0.17	-0.15
beta_2	-0.20	0.00	0.06	0.03	0.03	0.03	0.02	0.02	0.03
beta_3	0.01	-0.00	-0.03	-0.04	-0.05	-0.09	-0.21	-0.27	-0.30
beta_4	0.17	0.33	0.21	0.16	0.14	0.15	0.17	0.17	0.16
beta_5	-0.11	-0.90	-0.76	-0.60	-0.59	-0.59	-0.59	-0.59	-0.61
beta_6	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
beta_7	-	0.06	0.09	0.10	0.09	0.08	0.09	0.09	0.09
beta_8	-	-0.19	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00
beta_9	-	0.62	0.00	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00
beta_10	-	-	0.00	0.00	0.00	0.06	0.19	0.19	0.16
beta_11	-	-	0.02	0.04	0.06	0.06	0.04	0.04	0.05
beta_12	-	-	-0.04	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00
beta_13	-	-	-0.09	-0.02	-0.02	-0.02	-0.04	-0.04	-0.04
beta_14	-	-	0.44	0.00	0.00	0.00	0.00	0.00	0.00
beta_15	-	-	-	0.00	0.00	0.00	0.00	0.05	0.10
beta_16	-	-	-	0.00	0.00	0.00	0.00	0.00	0.00
beta_17	-	-	-	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00
beta_18	-	-	-	-0.08	-0.03	-0.04	-0.04	-0.05	-0.06
beta_19	-	-	-	-0.00	-0.00	-0.00	-0.00	-0.00	-0.00
beta_20	-	-	-	0.31	0.27	0.28	0.29	0.30	0.30
beta_21	-	-	-	-	-0.00	-0.00	0.00	0.00	0.00
beta_22	-	-	-	-	-0.00	-0.00	0.00	0.00	0.00
beta_23	-	-	-	-	-0.00	-0.00	-0.00	-0.00	-0.00
beta_24	-	-	-	-	-0.03	-0.01	-0.00	-0.00	-0.00
beta_25	-	-	-	-	-0.01	-0.02	-0.02	-0.01	-0.00
beta_26	-	-	-	-	-0.00	-0.00	-0.00	-0.00	-0.00
beta_27	-	-	-	-	0.03	0.02	0.01	0.01	0.02
beta_28	-	-	-	-	-	-0.03	-0.00	0.00	0.00
beta_29	-	-	-	-	-	-0.00	-0.00	-0.00	-0.00
beta_30	-	-	-	-	-	-0.01	-0.00	-0.00	-0.00
beta_31	-	-	-	-	-	-0.00	-0.00	-0.00	-0.00
beta_32	-	-	-	-	-	-0.00	-0.00	-0.00	-0.00
beta_33	-	-	-	-	-	-0.00	-0.00	-0.00	-0.00
beta_34	-	-	-	-	-	-0.00	-0.00	-0.00	-0.00
beta_35	-	-	-	-	-	0.00	0.00	0.00	0.00
beta_36	-	-	-	-	-	-	-0.08	-0.00	-0.00
beta_37	-	-	-	-	-	-	-0.00	-0.00	-0.00

beta_38	-	-	-	-	-	-	-0.01	-0.00	-0.00
beta_39	-	-	-	-	-	-	-0.00	-0.00	-0.00
beta_40	-	-	-	-	-	-	-0.00	-0.00	-0.00
beta_41	-	-	-	-	-	-	-0.00	-0.00	-0.00
beta_42	-	-	-	-	-	-	-0.00	-0.00	-0.00
beta_43	-	-	-	-	-	-	-0.00	-0.00	-0.00
beta_44	-	-	-	-	-	-	0.00	0.00	0.00
beta_45	-	-	-	-	-	-	-	-0.10	-0.00
beta_46	-	-	-	-	-	-	-	-0.01	-0.00
beta_47	-	-	-	-	-	-	-	-0.00	-0.00
beta_48	-	-	-	-	-	-	-	-0.00	-0.00
beta_49	-	-	-	-	-	-	-	-0.00	-0.00
beta_50	-	-	-	-	-	-	-	-0.00	-0.00
beta_51	-	-	-	-	-	-	-	-0.00	-0.00
beta_52	-	-	-	-	-	-	-	-0.00	-0.00
beta_53	-	-	-	-	-	-	-	-0.00	-0.00
beta_54	-	-	-	-	-	-	-	-0.00	-0.00
beta_55	-	-	-	-	-	-	-	-	-0.10
beta_56	-	-	-	-	-	-	-	-	-0.01
beta_57	-	-	-	-	-	-	-	-	-0.00
beta_58	-	-	-	-	-	-	-	-	-0.00
beta_59	-	-	-	-	-	-	-	-	-0.00
beta_60	-	-	-	-	-	-	-	-	-0.00
beta_61	-	-	-	-	-	-	-	-	-0.00
beta_62	-	-	-	-	-	-	-	-	-0.00
beta_63	-	-	-	-	-	-	-	-	-0.01
beta_64	-	-	-	-	-	-	-	-	-0.00
beta_65	-	-	-	-	-	-	-	-	-0.00

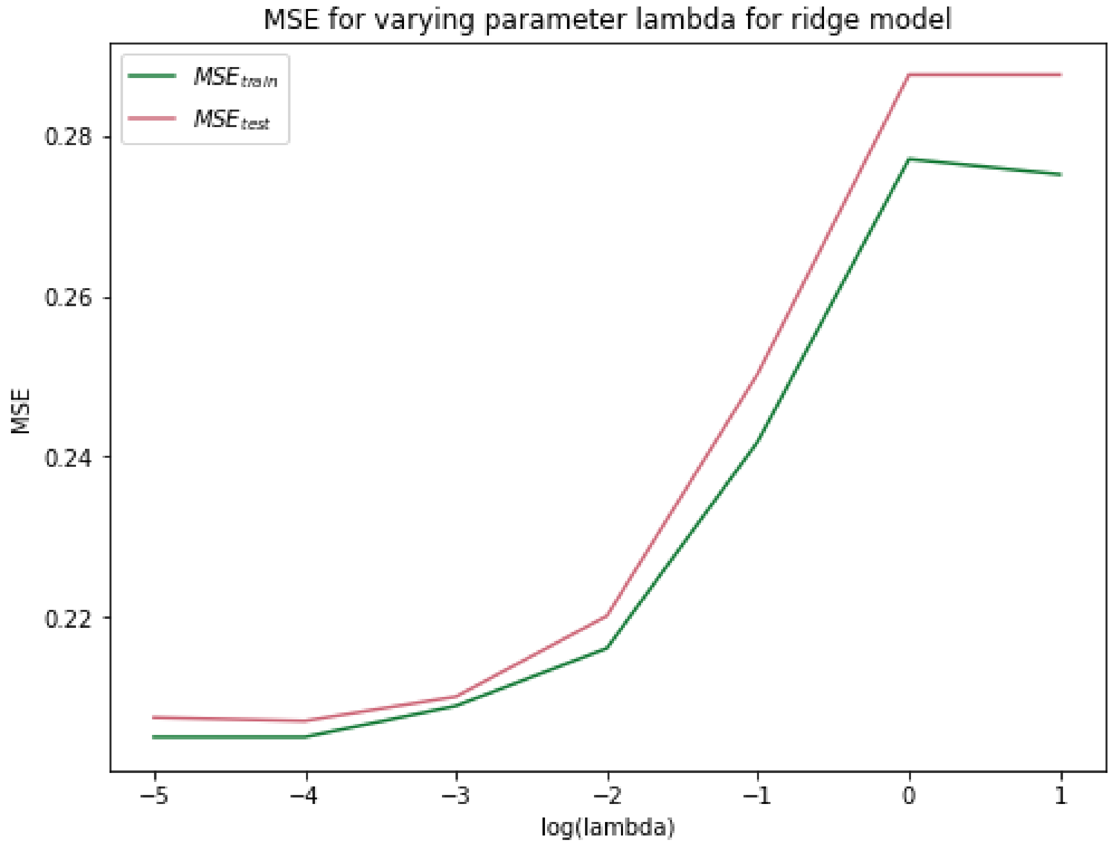
We see that the lasso also restrain the size of the coefficients β, but more strongly than for the ridge model. While the ridge model only reduces the size of the β, but never eliminates them completely, lasso actually sets some of them equal to zero. The larger the value for λ the stronger the reduction of the coefficients, and thereby the reduction in model complexity.

5.3.2 Bias-Variance Trade-off and Choice of λ

[127]:
```
lambda_with_bootstrap(lr_model,lasso_data,alpha_values,B=100)
print(f"Figure 13: A plot showing the mean square error on training and test sets for varying lambda values for a medium complexity model (d={degree}).")
```

```
Mean square error on static test set for B=100 bootstraps.
lambda=1e-05: MSE(test set): 0.20745657230475664
lambda=0.0001: MSE(test set): 0.2070341626320766
lambda=0.001: MSE(test set): 0.21005355774569306
lambda=0.01: MSE(test set): 0.22011981973265615
lambda=0.1: MSE(test set): 0.2503686273266299
lambda=1.0: MSE(test set): 0.28763023822866457
```

```
lambda=10.0: MSE(test set): 0.28763023822866457
The minimum MSE is: 0.2070341626320766 found for lambda value = 0.0001
```



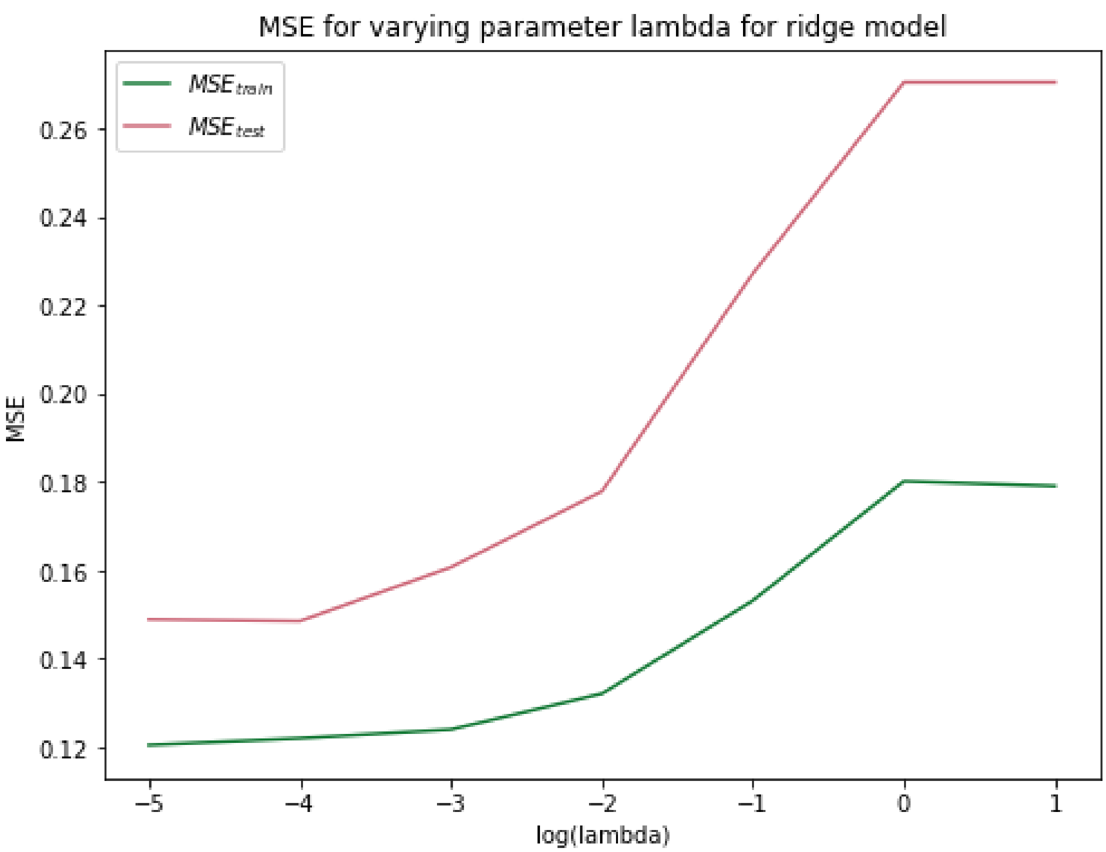

```
Figure 13: A plot showing the mean square error on training and test sets for
varying lambda values for a medium complexity model (d=7).
```

Like we saw for the ridge model, this plot indicates that $\lambda = 0$, which is equvalent to the OLS model is optimal. Let us again look at the result for a high complexity model, where OLS fails.

[128]:
```python
# Using the same data as I used to in ridge regression figure number to␣
↪illustrate a similar point
lambda_with_bootstrap(lr_model,ridge_data,alpha_values,B=100)
print(f"Figure 14: A plot showing the mean square error on training and test␣
↪sets for varying lambda values for a high complexity model (d={25}).")
```

```
Mean square error on static test set for B=100 bootstraps.
lambda=1e-05: MSE(test set): 0.14895862317494918
lambda=0.0001: MSE(test set): 0.14857650873995343
lambda=0.001: MSE(test set): 0.16074630005821589
lambda=0.01: MSE(test set): 0.17791938665222645
lambda=0.1: MSE(test set): 0.22725632564036993
```

```
lambda=1.0: MSE(test set): 0.27053214857497915
lambda=10.0: MSE(test set): 0.27053214857497915
The minimum MSE is: 0.14857650873995343 found for lambda value = 0.0001
```




```
Figure 14: A plot showing the mean square error on training and test sets for
varying lambda values for a high complexity model (d=25).
```

What we see here is that even for a high complexity model λ should be kept fairly small for the best result.

Now we look at the mean square error as the complexity of the model varies.

[129]:
```
warnings.filterwarnings("ignore", message=".*Objective did not converge.*")
lr_model = lr.LassoRegression(alpha=0.001,max_iter=10000)
bias_var_with_bootstrap(lr_model,B=100,n=20,min_degree=20,max_degree=35)
print("Figure 15: Plot showing the bias-variance trade-off for the ridge as the␣
 ↪complexity of the model increases.")
```

```
Mean square error on static test set for B=100 bootstraps.
d=20: MSE(test set): 0.158960
d=21: MSE(test set): 0.158897
d=22: MSE(test set): 0.159214
```

```
d=23: MSE(test set): 0.158311
d=24: MSE(test set): 0.159331
d=25: MSE(test set): 0.157918
d=26: MSE(test set): 0.158931
d=27: MSE(test set): 0.159206
d=28: MSE(test set): 0.158288
d=29: MSE(test set): 0.158781
d=30: MSE(test set): 0.158466
d=31: MSE(test set): 0.158782
d=32: MSE(test set): 0.158633
d=33: MSE(test set): 0.158103
d=34: MSE(test set): 0.159142
d=35: MSE(test set): 0.158120
The minimum MSE is: 0.15791837002056874 found for polynomial degree d = 25
```

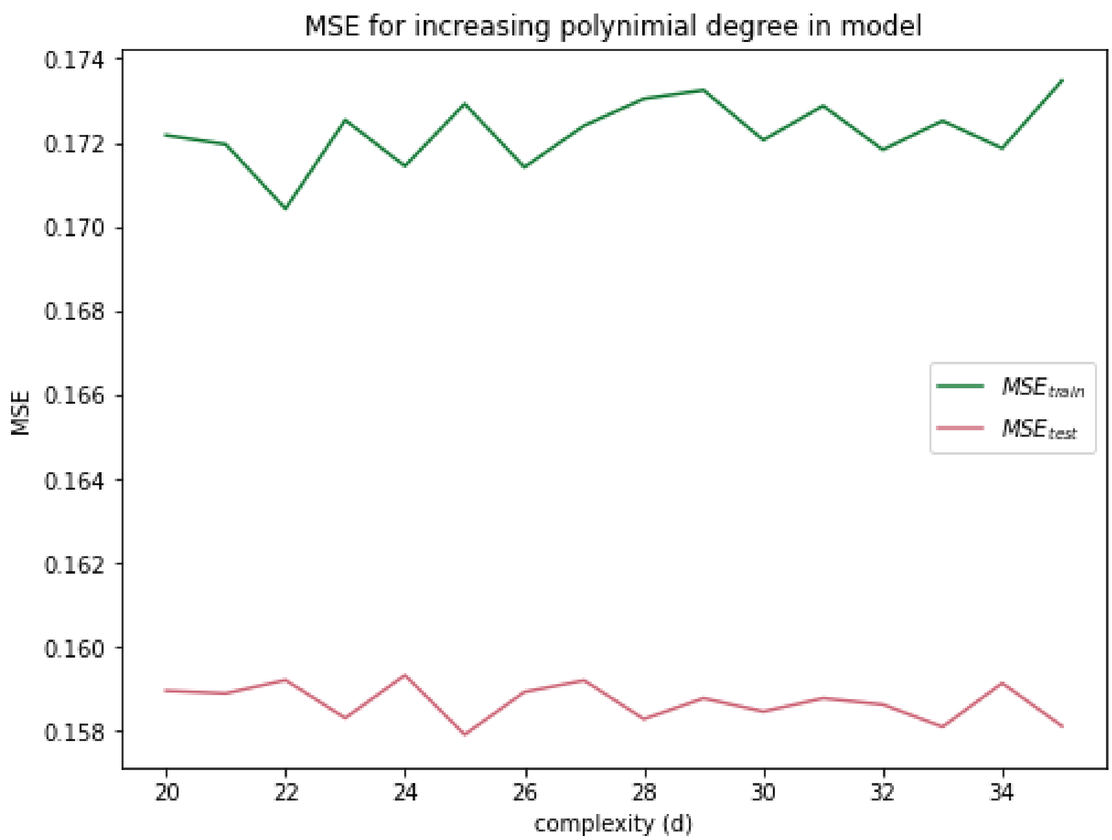


Figure 15: Plot showing the bias-variance trade-off for the ridge as the complexity of the model increases.

I have only plotted for higher polynomial degrees here as the lower degrees showed no sign of overfitting. I see from the plot that the error is not affected by the model complecity. It is odd that the test error here is lower than the training error. I don't know why that could be.

Now we use k-fold cross-validation to try and find the optimal value for the parameter λ.

[130]:
```
import warnings
warnings.filterwarnings("ignore", message=".*Objective did not converge.*")
nlambdas = 20
lambdas = np.logspace(-10, -2, nlambdas)
lr_model = lr.LassoRegression(max_iter=100000)
lambda_kfold(lr_model,ridge_data,lambdas,k=10)
print("Figure 16: Mean square error as a function of lambda for lasso␣
 ↪regression with a complex model (d=25).")
```

```
MSE,r^2=[ 0.12201427 -1.19662744] found for lambda=1e-10
MSE,r^2=[ 0.12244054 -1.19311288] found for lambda=2.6366508987303556e-10
MSE,r^2=[ 0.12078448 -1.21508404] found for lambda=6.951927961775591e-10
MSE,r^2=[ 0.12159225 -1.09800186] found for lambda=1.8329807108324374e-09
MSE,r^2=[ 0.12221851 -1.13419125] found for lambda=4.832930238571752e-09
MSE,r^2=[ 0.12214346 -1.11566355] found for lambda=1.274274985703132e-08
MSE,r^2=[ 0.12076257 -1.15229548] found for lambda=3.3598182862837814e-08
MSE,r^2=[ 0.1216293  -1.29886671] found for lambda=8.858667904100832e-08
MSE,r^2=[ 0.12200524 -1.13363729] found for lambda=2.3357214690901212e-07
MSE,r^2=[ 0.12134773 -1.16570067] found for lambda=6.158482110660254e-07
MSE,r^2=[ 0.12146487 -1.20691421] found for lambda=1.6237767391887209e-06
MSE,r^2=[ 0.12145869 -1.25743522] found for lambda=4.281332398719396e-06
MSE,r^2=[ 0.12249219 -1.16065481] found for lambda=1.1288378916846883e-05
MSE,r^2=[ 0.12181837 -1.20528073] found for lambda=2.976351441631313e-05
MSE,r^2=[ 0.12173323 -1.26580671] found for lambda=7.847599703514606e-05
MSE,r^2=[ 0.12230855 -1.35526887] found for lambda=0.00020691380811147902
MSE,r^2=[ 0.12423487 -1.58709487] found for lambda=0.0005455594781168515
MSE,r^2=[ 0.12629869 -1.69794339] found for lambda=0.0014384498882876
MSE,r^2=[ 0.12956419 -1.93418497] found for lambda=0.003792690190732246
MSE,r^2=[ 0.13346469 -2.43801233] found for lambda=0.01
```

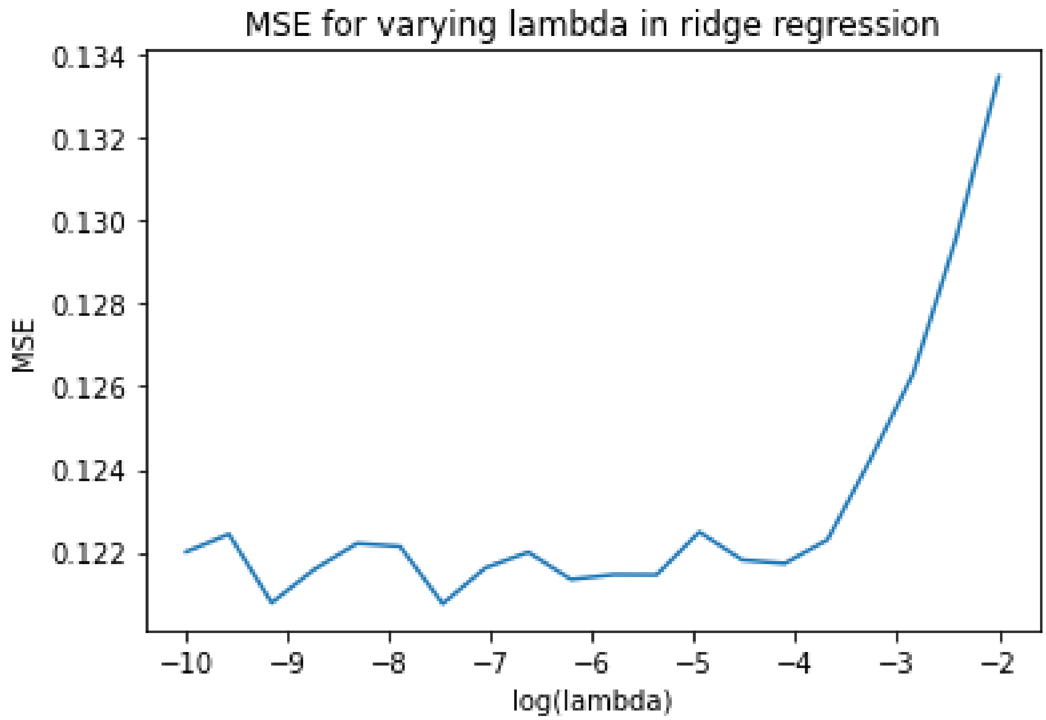


Figure 16: Mean square error as a function of lambda for lasso regression with a complex model (d=25).

We see that this result again shows that smaller λ (moving towards OLS) is better. The optimal value is found for $\lambda \approx 2.64 \cdot 10^{-06}$, but the behavor seems to be fairly equal for any $\lambda < 0.01$. I have made this plot for a complex model as for low complexity the mse simply increased with increasing λ

5.4 Terrain Data

Now that we have explored the models on the synthetic Franke data, we move on to look at real data, namely digital terrain data. I begin by loading in the data, and displaying it.

[137]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from imageio import imread

# Load the terrain
terrain = imread('./Data/wyoming_grand_teton.tif')
# Show the terrain
plt.figure(figsize=(10, 8))
#plt.figure()
plt.title('Terrain Wyoming')
plt.imshow(terrain, cmap='gray')
plt.xlabel('X')
```

```
plt.ylabel('Y')
plt.show()
print("Figure 17: The terrain data I have chosen to look at, showing the Teton␣
 ↪mountain range in Wyoming, USA.")
```

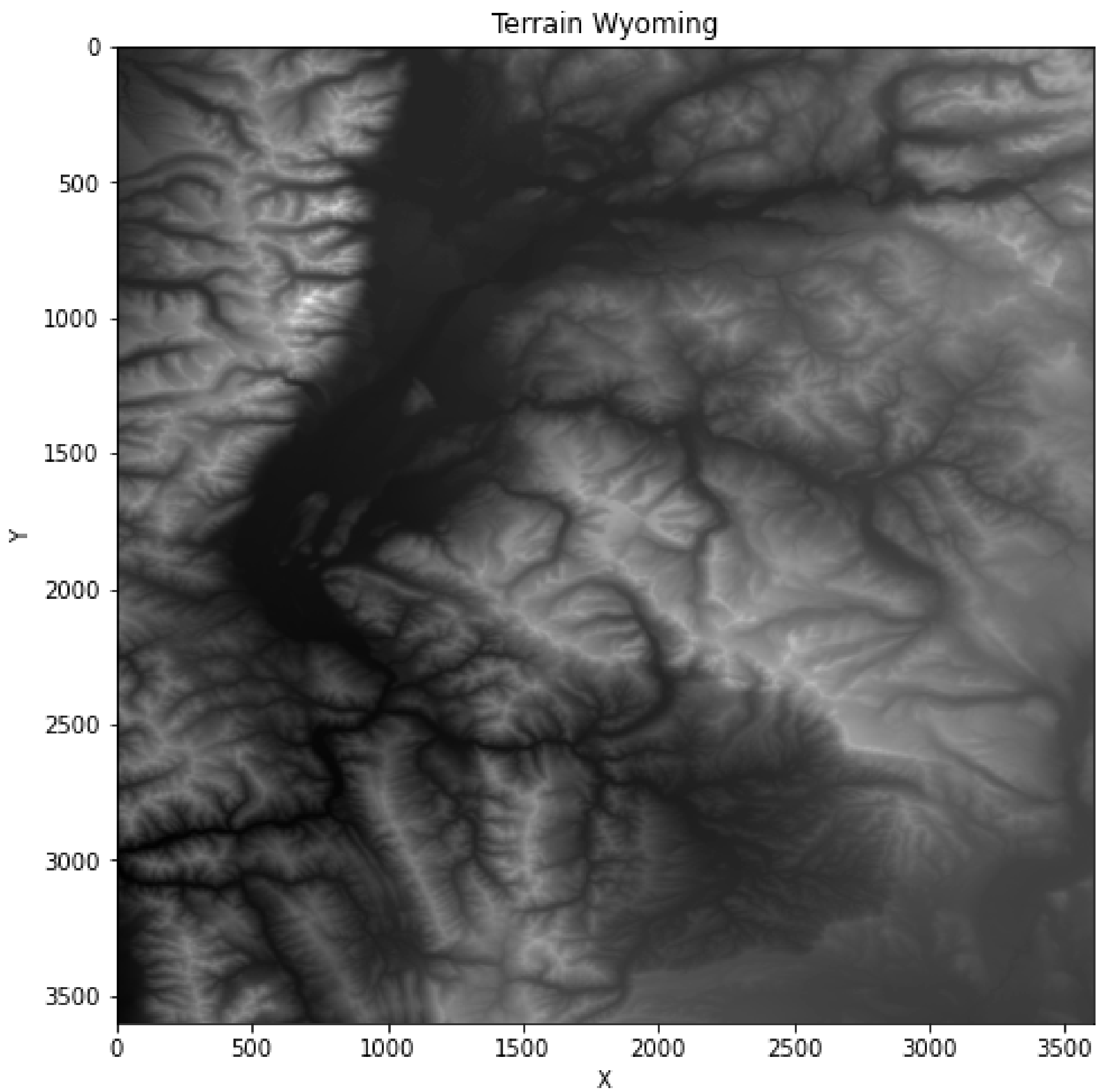
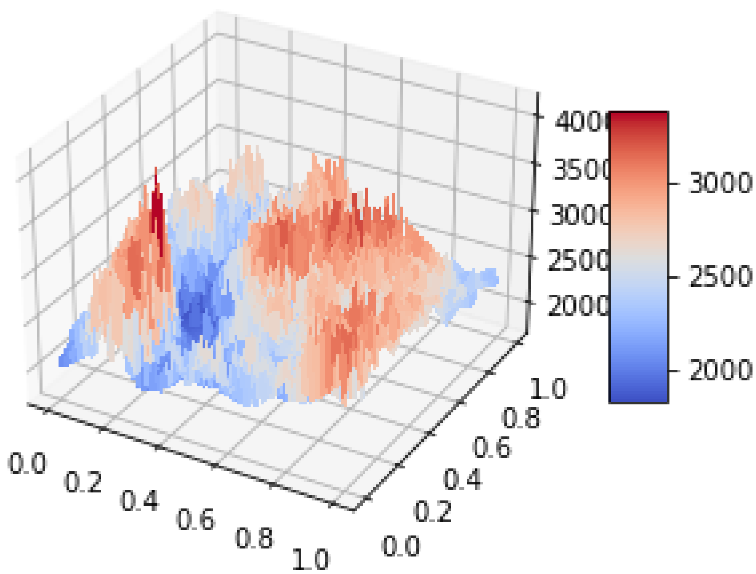


Figure 17: The terrain data I have chosen to look at, showing the Teton mountain range in Wyoming, USA.

[138]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

x = np.arange(terrain.shape[0])
y = np.arange(terrain.shape[1])
```

```
x = x/len(x)
y = y/len(y)
x,y = np.meshgrid(x,y)

# Plotting in the same way as the Franke data, for comparison
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, terrain, cmap=cm.coolwarm, linewidth=0,␣
 ↪antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
print("Figure 18: A 3D projection of the terrain data")
```




```
Figure 18: A 3D projection of the terrain data
```

This data has a much bigger range than the previous data, and it is noisier, but other than that we can see a similarity with the Franke data with noise.

5.4.1 OLS

[139]:
```
import Code.CreateData as cd

degree = 8
terrain_data = cd.CreateData(n=10)
terrain_data.x_mesh, terrain_data.y_mesh = x, y
terrain_data.create_design_matrix(d=degree)
test_fraction = 0.25
```

```
terrain_data.z_mesh=terrain
terrain_data.scale_dataset("standard")
terrain_data.split_dataset(test=test_fraction)
```

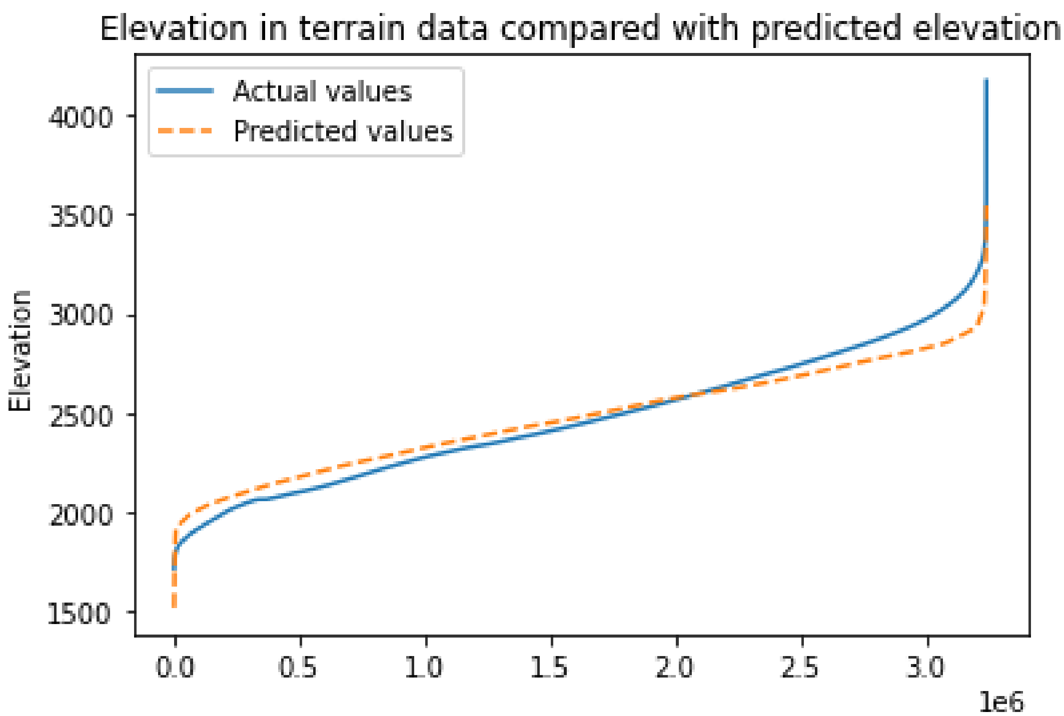
[139]:
```
(array([[ 1.        ,  0.51658502,  0.2125983 , …, -0.30064612,
         -0.37752616, -0.46963136],
        [ 1.        , -1.57861454,  1.28713361, …, -0.40824807,
         -0.33708251,  1.02891663],
        [ 1.        , -1.01008243, -0.66184449, …, -0.41538271,
         -0.4613215 , -0.51495905],
        …,
        [ 1.        ,  0.87540477, -0.45117015, …, -0.4030864 ,
         -0.45648667, -0.51372527],
        [ 1.        ,  1.22075677, -1.2592361 , …, -0.41567075,
         -0.46172857, -0.5153425 ],
        [ 1.        ,  1.26116006, -1.7132922 , …, -0.41571155,
         -0.46173409, -0.51534305]]),
 array([[ 1.        , -0.18662475, -1.42950714, …, -0.41571079,
         -0.46173397, -0.51534304],
        [ 1.        , -0.09619833,  0.38286934, …, -0.31495031,
         -0.35148555, -0.42586463],
        [ 1.        ,  0.38286934,  1.49973192, …,  1.72986959,
          2.31304118,  2.14646571],
        …,
        [ 1.        ,  0.60893541,  1.26981791, …,  1.2725562 ,
          1.37038782,  0.95945845],
        [ 1.        , -0.58392384,  1.31406914, …,  0.0274641 ,
          0.53354967,  1.14265533],
        [ 1.        ,  0.38864124, -0.14044956, …, -0.38496009,
         -0.44220463, -0.5061432 ]]),
 Array([3080, 2235, 2974, …, 2601, 2886, 3091], dtype=int16),
 Array([2172, 3076, 2291, …, 2181, 2799, 2336], dtype=int16))
```

[142]:
```
ols_model = ols.OrdinaryLeastSquares()
ols_model.fit(terrain_data.X_train,terrain_data.z_train)
z_hat = ols_model.predict(terrain_data.X_test)
```

[143]:
```
plt.plot(np.sort(terrain_data.z_test),label="Actual values")
plt.plot(np.sort(z_hat),label="Predicted values",linestyle='dashed')
plt.title("Elevation in terrain data compared with predicted elevation")
plt.ylabel("Elevation")
plt.legend()
print("Mean square error for OLS on terrain data: ", ols_model.
 ↪mean_square_error(z_hat,terrain_data.z_test))
print("r2 score for OLS on terrain data: ", ols_model.r2(z_hat,terrain_data.
 ↪z_test))
plt.show()
```

```
print(f"Figure 19: A plot showing the elevation values in the terrain data␣
↪(orange) compared with that values predicted by a OLS model with polynomial␣
↪of degree={degree} (blue).")
```

```
Mean square error for OLS on terrain data:  44943.70253297553
r2 score for OLS on terrain data:  0.31251054607708484
```




```
Figure 19: A plot showing the elevation values in the terrain data (orange)
compared with that values predicted by a OLS model with polynomial of degree=8
(blue).
```

We now explore the model performance for varying complexity, or polynomial degree. As my map is very big, I will use only a fraction of the data in the following discussion.

[144]:
```
section = terrain[0:500,0:500] # Selecting a secion of suitable size

x = np.arange(section.shape[0])
y = np.arange(section.shape[1])
x = x/len(x)
y = y/len(y)
x,y = np.meshgrid(x,y)

# Plotting in the same way as the previous data
fig = plt.figure()
```

```
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, section, cmap=cm.coolwarm, linewidth=0,␣
 ↪antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
print("Figure 20: A 3D projection of a section of the terrain data")
```

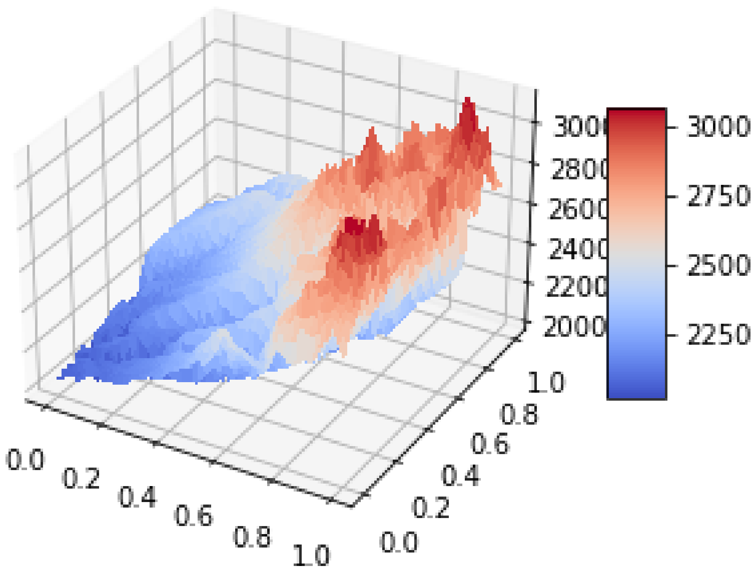


Figure 20: A 3D projection of a section of the terrain data

[146]:
```
def bias_var_terrain_bs(model,data,B=100,min_degree=1,max_degree=10):

    degrees = np.array(range(min_degree,max_degree+1))
    mse_bs = np.zeros((len(degrees),2))
    print("Mean square error on static test set for B=%d bootstraps." %(B))
    for i in range(len(degrees)):
        data.create_design_matrix(degrees[i])
        data.scale_dataset("standard")
        data.split_dataset(test=0.25)
        mse_bs[i,:] = model.bootstrap_fit(data.X_train,data.z_train,data.
 ↪X_test,data.z_test,B)
#         z_hat_train = model.fit(data.X_train,data.z_train)
#         z_hat_test = model.fit(data.X_test,data.z_test)
        mse_bs[i,1] = model.mean_square_error(z_hat_train,data.z_train)
        mse_bs[i,0] = model.mean_square_error(z_hat_test,data.z_test)

        print(f"d=%d: MSE(test set): %f " %(degrees[i],mse_bs[i,0]))
```

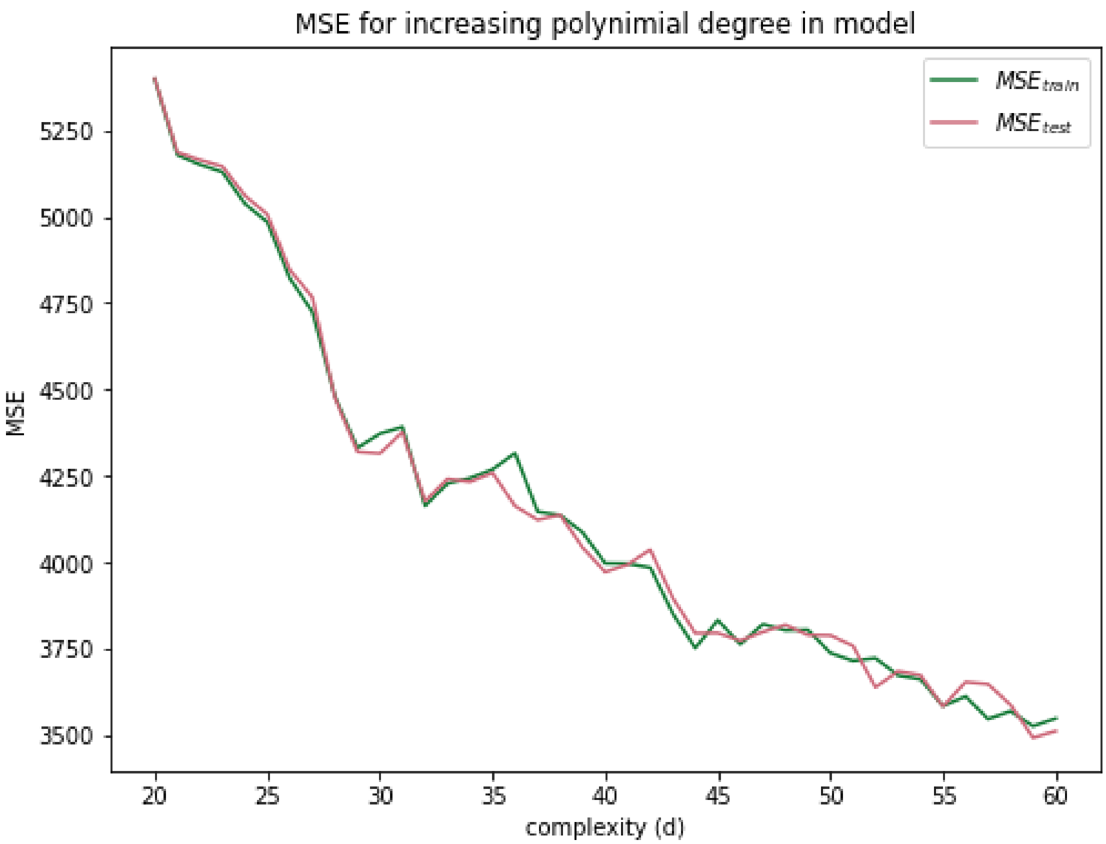
```
    fig = plt.figure(figsize=(8, 6))
    plt.plot(degrees,mse_bs[:,1], color='#117733', label='$MSE_{train}$')
    plt.plot(degrees,mse_bs[:,0], color='#CC6677', label='$MSE_{test}$')
    plt.xlabel('complexity (d)')
    plt.ylabel('MSE')
    plt.title('MSE for increasing polynimial degree in model')
    plt.legend()
    print('The minimum MSE is: {} found for polynomial degree d = {}'.
 ↪format(min(mse_bs[:,0]),degrees[np.argmin(mse_bs[:,0])]))
    plt.show()
```

[147]:
```
terrain_data = cd.CreateData(n=10)
terrain_data.x_mesh, terrain_data.y_mesh,terrain_data.z_mesh = x, y, section
```

[52]:
```
bias_var_terrain_bs(ols_model,terrain_data,B=10,min_degree=20,max_degree=60)
print("Figure 21: The mean square error as a function of model complexity for␣
 ↪OLS on a section of the terrain data.")
```

```
Mean square error on static test set for B=10 bootstraps.
d=20: MSE(test set): 5399.145556
d=21: MSE(test set): 5186.359277
d=22: MSE(test set): 5164.379504
d=23: MSE(test set): 5145.669677
d=24: MSE(test set): 5061.305092
d=25: MSE(test set): 5006.389758
d=26: MSE(test set): 4845.913531
d=27: MSE(test set): 4766.948209
d=28: MSE(test set): 4475.980992
d=29: MSE(test set): 4319.354796
d=30: MSE(test set): 4314.463450
d=31: MSE(test set): 4376.373758
d=32: MSE(test set): 4176.679314
d=33: MSE(test set): 4240.916377
d=34: MSE(test set): 4232.582047
d=35: MSE(test set): 4258.162356
d=36: MSE(test set): 4162.446797
d=37: MSE(test set): 4122.256895
d=38: MSE(test set): 4136.919433
d=39: MSE(test set): 4041.883209
d=40: MSE(test set): 3971.258376
d=41: MSE(test set): 3993.164190
d=42: MSE(test set): 4035.728255
d=43: MSE(test set): 3896.874778
d=44: MSE(test set): 3794.055034
d=45: MSE(test set): 3794.540369
d=46: MSE(test set): 3773.099445
```

```
d=47: MSE(test set): 3797.864801
d=48: MSE(test set): 3818.098581
d=49: MSE(test set): 3789.175575
d=50: MSE(test set): 3788.213769
d=51: MSE(test set): 3757.631480
d=52: MSE(test set): 3637.641724
d=53: MSE(test set): 3684.018407
d=54: MSE(test set): 3672.807227
d=55: MSE(test set): 3582.191150
d=56: MSE(test set): 3652.921127
d=57: MSE(test set): 3646.947553
d=58: MSE(test set): 3586.810829
d=59: MSE(test set): 3491.785999
d=60: MSE(test set): 3510.969162
The minimum MSE is: 3491.78599887064 found for polynomial degree d = 59
```



Overfitting does not seem to be an issue, but computational power limits our ability to explore even larger polynomials, and we see the performance gained is reducing as d increases.

We can have a look at the β values. Based on the above I expect them to be reasonably sized. I only use a couple of polynomials since they need to be so big.

[148]:
```python
print("Table 8: The coefficients beta_i for fitting a polynomial of degree=d␣
↪using OLS regression on a section of the terrain data.")
evaluate_betas(ols_model,data,40,41)
```

Table 8: The coefficients beta_i for fitting a polynomial of degree=d using OLS regression on a section of the terrain data.

	d=40	d=41
beta_0	0.41	0.41
beta_1	0.12	0.47
beta_2	0.38	0.25
beta_3	4.98	-1.89
beta_4	-3.35	-4.67
beta_5	-3.52	0.50
beta_6	-42.53	7.55
beta_7	28.84	51.59
beta_8	59.24	53.08
beta_9	25.45	-8.35
beta_10	156.19	-14.08
beta_11	-159.30	-274.24
beta_12	-63.50	-100.65
beta_13	-313.05	-243.12
beta_14	-102.36	25.74
beta_15	-318.90	-61.70
beta_16	459.47	729.21
beta_17	-88.50	36.06
beta_18	252.67	256.96
beta_19	671.45	458.11
beta_20	169.23	-58.97
beta_21	183.69	125.10
beta_22	-587.06	-822.88
beta_23	61.89	-31.77
beta_24	498.22	355.65
beta_25	-402.62	-273.05
beta_26	-604.43	-395.78
beta_27	-95.37	35.24
beta_28	377.95	161.68
beta_29	104.05	-8.49
beta_30	-102.03	-98.93
beta_31	-357.49	-516.83
beta_32	-206.58	2.43
beta_33	-179.61	-308.16
beta_34	11.30	75.78
beta_35	69.17	163.77
beta_36	-329.20	-277.96
beta_37	320.96	451.25
beta_38	382.73	442.49
beta_39	-326.56	-114.64

beta_40	-164.10	-251.35
beta_41	43.91	96.57
beta_42	368.68	154.66
beta_43	333.98	262.04
beta_44	-163.77	-193.26
beta_45	-246.47	-122.93
beta_46	37.54	231.58
beta_47	53.44	-22.21
beta_48	124.86	410.11
beta_49	50.86	-4.57
beta_50	299.37	318.57
beta_51	-56.59	-199.12
beta_52	320.15	417.31
beta_53	52.31	-56.35
beta_54	-29.01	-153.50
beta_55	-225.00	-100.59
beta_56	-198.42	-117.35
beta_57	-253.86	-367.17
beta_58	-55.93	-103.18
beta_59	204.96	244.81
beta_60	421.19	428.17
beta_61	79.64	-26.34
beta_62	-254.98	-196.09
beta_63	11.14	235.64
beta_64	-181.96	-264.44
beta_65	202.79	114.11
beta_66	327.94	260.11
beta_67	-106.91	-216.87
beta_68	-264.29	-255.71
beta_69	-137.25	-353.63
beta_70	-1.34	-231.38
beta_71	261.92	576.24
beta_72	98.52	-211.44
beta_73	-153.81	-11.54
beta_74	-194.39	-81.40
beta_75	-157.39	-64.71
beta_76	-248.46	-231.61
beta_77	-155.65	-28.66
beta_78	419.24	282.17
beta_79	-42.96	-233.81
beta_80	-19.68	92.74
beta_81	48.43	-48.65
beta_82	-92.34	-454.53
beta_83	-111.58	144.90
beta_84	-57.12	-48.13
beta_85	-121.77	-367.37
beta_86	-167.71	159.53
beta_87	-20.23	-72.05

beta_88	-224.31	-281.31
beta_89	-56.47	33.50
beta_90	298.51	350.17
beta_91	23.48	-20.09
beta_92	106.10	-81.01
beta_93	145.16	278.21
beta_94	225.60	330.17
beta_95	61.91	-183.60
beta_96	-284.26	-117.75
beta_97	-362.27	-305.52
beta_98	-22.06	-72.84
beta_99	-87.33	-205.60
beta_100	64.52	318.50
beta_101	225.59	-21.38
beta_102	-100.60	-230.39
beta_103	95.38	221.30
beta_104	-119.11	-12.27
beta_105	3.09	-15.10
beta_106	52.00	-4.01
beta_107	175.91	241.84
beta_108	232.39	439.90
beta_109	259.07	190.11
beta_110	-123.56	40.18
beta_111	-416.33	-410.15
beta_112	-139.12	-149.50
beta_113	155.99	161.23
beta_114	-47.53	-138.91
beta_115	111.24	245.11
beta_116	322.23	28.25
beta_117	-34.66	-114.84
beta_118	181.44	272.92
beta_119	87.98	45.45
beta_120	-291.72	-207.99
beta_121	43.49	116.20
beta_122	81.77	55.43
beta_123	92.71	269.32
beta_124	160.52	233.05
beta_125	98.28	268.64
beta_126	-226.60	-241.35
beta_127	-125.44	-175.17
beta_128	125.17	155.35
beta_129	211.15	238.66
beta_130	-122.98	-228.04
beta_131	124.14	174.75
beta_132	260.49	67.83
beta_133	68.41	60.86
beta_134	201.16	206.64
beta_135	-192.31	-244.14

beta_136	-112.53	-58.03
beta_137	-36.57	145.87
beta_138	27.13	-88.45
beta_139	-110.04	-34.18
beta_140	34.04	118.40
beta_141	177.63	314.27
beta_142	-7.69	-41.43
beta_143	5.92	-57.73
beta_144	143.31	133.37
beta_145	239.99	283.27
beta_146	90.68	123.92
beta_147	-209.70	-319.44
beta_148	79.11	116.11
beta_149	127.84	100.60
beta_150	-37.78	51.77
beta_151	121.81	45.68
beta_152	-199.36	-277.91
beta_153	-225.93	-146.52
beta_154	-21.61	181.93
beta_155	-122.02	-245.98
beta_156	-247.90	-285.10
beta_157	-153.36	-103.15
beta_158	88.98	159.16
beta_159	34.34	-26.23
beta_160	74.70	12.86
beta_161	161.02	142.80
beta_162	171.98	195.18
beta_163	115.21	165.46
beta_164	-30.25	3.56
beta_165	-295.89	-371.65
beta_166	6.29	63.88
beta_167	18.81	121.71
beta_168	-8.85	93.67
beta_169	12.20	-111.75
beta_170	-182.98	-235.51
beta_171	-2.91	-0.71
beta_172	-16.78	147.54
beta_173	-69.49	-197.07
beta_174	-221.61	-348.09
beta_175	-268.84	-255.74
beta_176	12.56	2.83
beta_177	51.45	-55.42
beta_178	84.65	16.16
beta_179	187.56	154.95
beta_180	155.86	166.13
beta_181	57.17	92.56
beta_182	15.83	51.37
beta_183	-92.97	-50.74

beta_184	-272.64	-320.40
beta_185	-16.81	46.20
beta_186	-128.86	62.09
beta_187	-43.45	41.03
beta_188	-119.67	-239.76
beta_189	162.77	91.98
beta_190	-28.78	-41.58
beta_191	29.42	97.00
beta_192	-91.93	-158.00
beta_193	-135.69	-290.66
beta_194	-268.31	-273.76
beta_195	-37.25	-97.81
beta_196	-9.31	-123.92
beta_197	59.28	-8.82
beta_198	132.79	87.23
beta_199	111.91	105.18
beta_200	7.22	46.96
beta_201	-44.75	-13.39
beta_202	-36.17	-22.07
beta_203	-61.06	-21.89
beta_204	-182.30	-213.03
beta_205	3.22	34.78
beta_206	-241.68	-32.00
beta_207	-14.06	12.77
beta_208	-115.89	-213.02
beta_209	150.12	161.34
beta_210	160.43	82.75
beta_211	31.75	1.72
beta_212	-31.93	-38.70
beta_213	-10.40	-145.91
beta_214	-224.80	-212.20
beta_215	-10.58	-93.41
beta_216	5.90	-93.44
beta_217	-3.61	-28.90
beta_218	70.43	26.83
beta_219	17.81	-14.93
beta_220	-77.89	-45.36
beta_221	-141.41	-74.42
beta_222	-100.43	-80.74
beta_223	24.03	-0.97
beta_224	28.29	59.83
beta_225	-97.06	-120.00
beta_226	61.59	32.31
beta_227	-257.03	-90.32
beta_228	39.81	5.53
beta_229	-109.53	-146.70
beta_230	173.64	239.87
beta_231	282.64	171.81

beta_232	38.15	-83.98
beta_233	27.17	79.74
beta_234	153.69	54.33
beta_235	-88.17	-52.62
beta_236	46.90	-32.66
beta_237	38.81	-22.63
beta_238	4.61	41.51
beta_239	12.69	10.22
beta_240	-37.54	-91.94
beta_241	-135.42	-146.16
beta_242	-178.44	-117.68
beta_243	-142.51	-78.03
beta_244	-17.08	-43.57
beta_245	132.42	59.18
beta_246	131.17	149.31
beta_247	-38.28	-54.03
beta_248	49.76	-29.76
beta_249	-219.70	-123.02
beta_250	119.02	29.09
beta_251	-137.24	-92.52
beta_252	119.53	224.44
beta_253	13.32	-15.46
beta_254	33.76	-142.91
beta_255	84.92	174.34
beta_256	243.90	200.10
beta_257	-38.24	42.32
beta_258	101.57	31.57
beta_259	66.54	48.01
beta_260	-14.77	103.45
beta_261	4.40	65.69
beta_262	-49.62	-92.81
beta_263	-154.65	-208.32
beta_264	-201.51	-184.14
beta_265	-166.67	-89.08
beta_266	-54.31	-18.87
beta_267	75.58	-0.25
beta_268	168.68	70.85
beta_269	162.34	189.55
beta_270	21.36	12.88
beta_271	113.82	-23.32
beta_272	-139.19	-115.75
beta_273	131.72	24.44
beta_274	-127.74	-16.63
beta_275	93.85	189.38
beta_276	179.76	135.01
beta_277	-39.88	-215.06
beta_278	48.55	176.16
beta_279	280.14	286.41

beta_280	51.83	154.33
beta_281	145.47	77.43
beta_282	104.42	107.81
beta_283	5.72	176.50
beta_284	-2.82	128.93
beta_285	-30.46	-29.28
beta_286	-118.32	-183.36
beta_287	-206.84	-232.62
beta_288	-177.07	-133.46
beta_289	-88.37	-14.81
beta_290	57.78	54.54
beta_291	146.72	30.52
beta_292	178.99	73.08
beta_293	107.59	170.22
beta_294	14.77	30.60
beta_295	142.26	-28.69
beta_296	-76.29	-103.63
beta_297	145.04	39.80
beta_298	-40.35	89.68
beta_299	88.79	134.46
beta_300	-22.13	11.64
beta_301	-41.43	-192.76
beta_302	59.90	184.97
beta_303	259.46	299.67
beta_304	63.31	180.45
beta_305	116.14	47.96
beta_306	57.49	71.07
beta_307	-8.65	184.37
beta_308	2.28	170.43
beta_309	17.42	55.42
beta_310	-48.56	-99.00
beta_311	-149.21	-195.01
beta_312	-138.73	-145.89
beta_313	-80.77	-35.26
beta_314	47.36	88.98
beta_315	119.22	88.62
beta_316	135.07	13.25
beta_317	109.27	23.93
beta_318	82.76	173.75
beta_319	21.66	63.23
beta_320	179.59	-7.48
beta_321	-74.90	-114.81
beta_322	141.38	56.22
beta_323	-8.99	125.62
beta_324	-135.17	-100.21
beta_325	-56.42	11.13
beta_326	-91.73	-175.86
beta_327	-32.55	91.11

beta_328	163.92	229.24
beta_329	37.75	158.64
beta_330	104.47	18.82
beta_331	52.40	37.92
beta_332	-22.28	147.87
beta_333	-1.78	162.30
beta_334	41.87	93.28
beta_335	13.57	-12.12
beta_336	-41.02	-85.11
beta_337	-73.27	-102.72
beta_338	-43.44	-47.30
beta_339	52.97	72.71
beta_340	104.53	129.07
beta_341	100.35	70.15
beta_342	37.84	-59.96
beta_343	2.41	-47.22
beta_344	-11.09	123.88
beta_345	1.69	73.78
beta_346	157.09	-10.07
beta_347	26.22	-32.87
beta_348	126.22	74.68
beta_349	78.34	170.30
beta_350	-205.77	-222.59
beta_351	-73.78	8.60
beta_352	-72.56	-89.83
beta_353	-85.14	8.53
beta_354	77.90	139.24
beta_355	-17.10	93.54
beta_356	41.48	-57.01
beta_357	3.09	-39.82
beta_358	-68.00	55.26
beta_359	-30.90	87.71
beta_360	53.31	83.94
beta_361	70.20	49.86
beta_362	20.89	6.72
beta_363	-3.35	-17.80
beta_364	29.55	-2.71
beta_365	85.54	55.85
beta_366	109.09	113.85
beta_367	105.76	131.39
beta_368	52.97	41.91
beta_369	-54.08	-118.47
beta_370	-94.60	-108.79
beta_371	-75.44	86.38
beta_372	-36.75	65.20
beta_373	150.09	8.35
beta_374	17.15	-18.89
beta_375	66.33	59.36

beta_376	111.66	151.54
beta_377	-279.65	-330.70
beta_378	-239.20	-119.33
beta_379	-15.60	28.82
beta_380	-84.80	-48.09
beta_381	-48.18	3.29
beta_382	-58.49	31.20
beta_383	-18.68	-121.16
beta_384	-38.22	-110.20
beta_385	-115.71	-45.69
beta_386	-78.00	-22.33
beta_387	40.52	16.72
beta_388	85.14	47.57
beta_389	72.35	69.60
beta_390	43.55	64.40
beta_391	56.39	48.87
beta_392	94.45	48.20
beta_393	118.23	72.50
beta_394	131.12	126.96
beta_395	48.19	96.68
beta_396	-72.52	-37.20
beta_397	-166.57	-193.39
beta_398	-203.55	-190.99
beta_399	-128.13	46.91
beta_400	-32.56	77.81
beta_401	117.68	19.25
beta_402	59.34	39.87
beta_403	-55.38	-6.58
beta_404	100.14	86.92
beta_405	-115.47	-232.97
beta_406	-157.11	-79.41
beta_407	25.77	124.59
beta_408	-126.10	-135.71
beta_409	-109.95	-92.12
beta_410	-81.93	-19.20
beta_411	-60.28	-157.06
beta_412	-71.22	-153.40
beta_413	-159.17	-130.94
beta_414	-122.24	-130.26
beta_415	-5.55	-90.71
beta_416	103.90	15.90
beta_417	92.46	82.86
beta_418	59.85	111.15
beta_419	60.52	100.75
beta_420	77.48	62.04
beta_421	119.75	49.35
beta_422	146.09	80.68
beta_423	75.92	87.41

beta_424	-30.70	50.67
beta_425	-159.29	-86.14
beta_426	-234.28	-239.52
beta_427	-197.42	-195.91
beta_428	-117.22	34.06
beta_429	-41.24	68.44
beta_430	80.63	23.07
beta_431	55.69	63.88
beta_432	-96.46	-18.38
beta_433	139.50	61.65
beta_434	23.73	-119.30
beta_435	27.57	23.83
beta_436	17.75	157.82
beta_437	-57.57	-137.86
beta_438	-141.15	-160.90
beta_439	-105.43	-66.63
beta_440	-44.46	-134.52
beta_441	-56.01	-133.01
beta_442	-165.50	-157.38
beta_443	-131.66	-188.18
beta_444	4.40	-145.67
beta_445	68.69	-65.72
beta_446	86.22	47.61
beta_447	51.34	111.82
beta_448	27.24	116.04
beta_449	54.43	89.76
beta_450	95.75	45.41
beta_451	106.90	20.46
beta_452	104.54	43.49
beta_453	17.57	57.81
beta_454	-67.96	40.78
beta_455	-160.99	-78.79
beta_456	-223.17	-237.78
beta_457	-193.83	-221.26
beta_458	-79.18	27.89
beta_459	-13.62	72.10
beta_460	34.24	18.00
beta_461	38.83	71.84
beta_462	-181.52	-75.11
beta_463	81.73	-25.17
beta_464	66.30	-45.75
beta_465	44.10	-0.43
beta_466	45.73	194.14
beta_467	-11.97	-142.80
beta_468	-174.71	-224.24
beta_469	-57.50	-52.07
beta_470	-25.96	-92.65
beta_471	-32.60	-80.46

beta_472	-132.41	-122.74
beta_473	-135.60	-206.61
beta_474	-1.08	-182.94
beta_475	81.29	-98.79
beta_476	90.63	17.20
beta_477	38.51	87.72
beta_478	-15.69	97.77
beta_479	-35.08	61.01
beta_480	17.36	28.34
beta_481	51.17	-16.83
beta_482	80.83	-15.36
beta_483	49.97	9.29
beta_484	-32.13	33.88
beta_485	-103.16	28.31
beta_486	-138.29	-63.22
beta_487	-144.43	-194.44
beta_488	-98.28	-177.88
beta_489	-8.84	39.82
beta_490	6.76	64.05
beta_491	-20.47	-1.40
beta_492	-14.15	43.45
beta_493	-191.04	-88.35
beta_494	10.35	-100.58
beta_495	29.86	-8.68
beta_496	63.75	-20.79
beta_497	71.50	199.96
beta_498	38.78	-122.06
beta_499	-152.38	-228.82
beta_500	-18.26	-35.30
beta_501	10.51	-27.11
beta_502	15.94	6.27
beta_503	-74.13	-37.67
beta_504	-109.30	-158.39
beta_505	9.58	-163.36
beta_506	91.10	-97.37
beta_507	81.26	-14.94
beta_508	14.34	43.94
beta_509	-73.70	45.43
beta_510	-98.59	28.50
beta_511	-71.34	-2.56
beta_512	-2.69	-24.65
beta_513	40.04	-47.74
beta_514	25.51	-58.52
beta_515	-12.82	-26.16
beta_516	-41.82	36.23
beta_517	-84.30	43.40
beta_518	-61.89	-17.22
beta_519	-23.46	-125.43

beta_520	3.35	-135.62
beta_521	85.40	70.46
beta_522	24.22	49.15
beta_523	-62.71	-16.21
beta_524	1.73	52.99
beta_525	-231.94	-132.72
beta_526	25.01	-90.07
beta_527	75.18	104.73
beta_528	96.47	-16.40
beta_529	103.05	185.17
beta_530	98.07	-70.65
beta_531	-122.78	-202.75
beta_532	71.22	27.88
beta_533	65.23	53.83
beta_534	87.66	114.69
beta_535	-39.37	43.22
beta_536	-51.44	-58.90
beta_537	22.03	-103.39
beta_538	128.66	-35.33
beta_539	103.86	8.86
beta_540	18.52	27.73
beta_541	-99.72	-0.61
beta_542	-152.43	-23.53
beta_543	-135.73	-39.35
beta_544	-100.89	-66.76
beta_545	-30.38	-69.83
beta_546	16.82	-71.04
beta_547	7.34	-64.77
beta_548	-18.34	-18.39
beta_549	-37.84	44.07
beta_550	-19.70	84.01
beta_551	37.75	44.00
beta_552	85.05	-58.88
beta_553	111.67	-70.91
beta_554	117.72	62.09
beta_555	13.34	12.01
beta_556	-119.55	-50.39
beta_557	29.52	60.55
beta_558	-162.55	-106.98
beta_559	-2.01	-86.15
beta_560	288.55	331.87
beta_561	2.17	-85.45
beta_562	99.09	126.71
beta_563	115.61	-20.51
beta_564	-95.56	-159.13
beta_565	116.04	60.60
beta_566	66.90	89.39
beta_567	121.51	184.81

beta_568	30.66	145.12
beta_569	-24.76	23.89
beta_570	32.06	-19.75
beta_571	112.70	18.72
beta_572	134.52	67.38
beta_573	40.13	42.75
beta_574	-83.89	-15.59
beta_575	-180.74	-77.59
beta_576	-186.36	-93.53
beta_577	-153.91	-96.88
beta_578	-71.15	-69.85
beta_579	-14.33	-64.49
beta_580	22.43	-55.19
beta_581	-3.30	-57.24
beta_582	-29.19	-22.29
beta_583	1.27	67.81
beta_584	38.38	114.05
beta_585	142.54	113.93
beta_586	177.99	11.26
beta_587	175.15	-23.99
beta_588	146.73	71.89
beta_589	11.71	-10.66
beta_590	-141.25	-64.55
beta_591	22.44	32.86
beta_592	-122.77	-102.95
beta_593	-51.07	-83.52
beta_594	80.26	223.03
beta_595	-3.79	-64.76
beta_596	5.64	-3.81
beta_597	149.28	66.34
beta_598	-68.84	-97.39
beta_599	148.58	91.49
beta_600	72.24	116.74
beta_601	172.24	243.49
beta_602	45.06	182.71
beta_603	-3.68	92.14
beta_604	41.30	63.04
beta_605	120.40	104.23
beta_606	137.02	128.10
beta_607	79.93	94.67
beta_608	-41.61	-0.40
beta_609	-158.43	-99.51
beta_610	-210.21	-146.46
beta_611	-162.49	-118.88
beta_612	-84.00	-68.84
beta_613	-16.65	-36.24
beta_614	38.42	-15.67
beta_615	24.19	-38.80

beta_616	2.03	-44.99
beta_617	-14.76	-10.39
beta_618	-1.26	58.87
beta_619	89.54	142.28
beta_620	190.62	153.71
beta_621	227.95	70.91
beta_622	198.24	25.62
beta_623	141.19	77.57
beta_624	2.82	-34.82
beta_625	-190.22	-101.70
beta_626	73.78	40.30
beta_627	6.43	-37.83
beta_628	-85.32	-59.18
beta_629	31.19	197.34
beta_630	104.26	63.37
beta_631	-32.28	-87.51
beta_632	78.28	93.19
beta_633	-49.87	-27.47
beta_634	181.76	125.61
beta_635	15.24	83.05
beta_636	143.39	211.26
beta_637	83.40	197.19
beta_638	-33.02	88.52
beta_639	6.64	92.00
beta_640	98.45	159.14
beta_641	127.87	185.30
beta_642	101.53	143.08
beta_643	8.54	34.98
beta_644	-106.77	-90.61
beta_645	-163.19	-154.87
beta_646	-151.19	-147.77
beta_647	-91.93	-90.88
beta_648	7.82	-1.32
beta_649	70.23	42.33
beta_650	79.85	36.91
beta_651	35.85	-11.48
beta_652	-11.33	-41.97
beta_653	-47.06	-36.40
beta_654	-13.70	36.28
beta_655	87.14	133.63
beta_656	178.46	162.64
beta_657	206.72	105.73
beta_658	147.07	45.72
beta_659	76.03	59.43
beta_660	-23.63	-60.98
beta_661	-178.54	-97.27
beta_662	106.06	36.73
beta_663	114.78	14.31

beta_664	-63.53	4.33
beta_665	-17.11	126.35
beta_666	70.16	97.56
beta_667	-91.56	-177.19
beta_668	58.42	160.37
beta_669	-49.04	32.93
beta_670	151.19	110.31
beta_671	-72.43	12.43
beta_672	88.06	123.91
beta_673	40.93	106.57
beta_674	-63.27	32.63
beta_675	-71.03	45.01
beta_676	14.10	138.05
beta_677	83.43	195.06
beta_678	87.67	161.42
beta_679	36.19	59.05
beta_680	-51.21	-72.77
beta_681	-92.74	-140.67
beta_682	-102.25	-148.56
beta_683	-51.03	-87.38
beta_684	28.99	7.27
beta_685	89.77	77.49
beta_686	143.89	122.63
beta_687	108.17	81.15
beta_688	23.63	-2.22
beta_689	-55.23	-69.18
beta_690	-102.25	-85.77
beta_691	-65.11	-15.74
beta_692	35.77	98.66
beta_693	129.91	159.74
beta_694	109.27	101.86
beta_695	29.36	37.40
beta_696	-2.64	43.02
beta_697	-63.10	-99.35
beta_698	-159.70	-90.11
beta_699	157.79	54.52
beta_700	225.63	85.66
beta_701	-76.72	28.55
beta_702	-69.09	8.11
beta_703	-57.91	59.67
beta_704	-145.34	-241.36
beta_705	33.07	210.43
beta_706	-19.86	98.43
beta_707	107.39	78.58
beta_708	-186.39	-94.39
beta_709	37.54	16.96
beta_710	13.54	-13.74
beta_711	-116.77	-87.24

beta_712	-144.14	-48.91
beta_713	-93.12	50.48
beta_714	16.02	151.76
beta_715	70.16	151.47
beta_716	54.89	63.17
beta_717	35.36	-27.75
beta_718	-23.46	-115.76
beta_719	-26.70	-121.17
beta_720	-7.19	-77.48
beta_721	51.77	9.49
beta_722	121.48	102.72
beta_723	168.33	162.35
beta_724	148.57	151.74
beta_725	68.85	75.09
beta_726	-34.85	-33.09
beta_727	-131.29	-124.92
beta_728	-181.47	-156.07
beta_729	-130.90	-80.41
beta_730	-30.06	44.72
beta_731	34.83	121.35
beta_732	-8.96	85.72
beta_733	-113.30	14.03
beta_734	-92.88	17.87
beta_735	-55.88	-110.69
beta_736	-141.85	-84.44
beta_737	137.37	33.86
beta_738	273.76	133.01
beta_739	-56.66	55.68
beta_740	-109.88	-135.97
beta_741	-148.00	18.05
beta_742	-145.02	-246.66
beta_743	-64.12	165.70
beta_744	-16.38	111.88
beta_745	75.66	44.44
beta_746	-297.28	-193.84
beta_747	-21.99	-108.67
beta_748	22.82	-130.25
beta_749	-127.47	-214.18
beta_750	-202.00	-180.48
beta_751	-162.94	-58.70
beta_752	-73.43	48.32
beta_753	8.47	72.31
beta_754	60.75	36.43
beta_755	59.28	-32.20
beta_756	55.83	-67.56
beta_757	41.35	-68.75
beta_758	37.45	-37.81
beta_759	84.03	41.43

beta_760	135.70	118.25
beta_761	167.61	171.79
beta_762	164.56	187.72
beta_763	119.79	146.54
beta_764	29.86	57.51
beta_765	-88.69	-65.51
beta_766	-192.53	-178.21
beta_767	-215.65	-204.64
beta_768	-165.92	-145.47
beta_769	-85.64	-25.24
beta_770	-45.14	61.62
beta_771	-122.93	44.60
beta_772	-220.99	-9.34
beta_773	-161.90	-17.23
beta_774	6.66	-95.22
beta_775	-57.63	-32.82
beta_776	111.22	40.82
beta_777	226.31	147.70
beta_778	-39.26	47.82
beta_779	-274.01	-381.35
beta_780	-47.57	46.25
beta_781	-65.78	-181.01
beta_782	-97.57	88.52
beta_783	49.35	113.37
beta_784	46.62	-2.48
beta_785	-388.01	-274.59
beta_786	11.85	-148.80
beta_787	123.35	-170.49
beta_788	-20.43	-261.82
beta_789	-168.32	-262.89
beta_790	-181.23	-160.24
beta_791	-119.63	-66.19
beta_792	-38.98	-30.04
beta_793	51.45	-22.61
beta_794	73.97	-47.12
beta_795	85.04	-40.48
beta_796	49.13	-27.27
beta_797	34.83	14.23
beta_798	30.30	57.74
beta_799	38.80	91.76
beta_800	63.73	121.91
beta_801	95.05	147.19
beta_802	100.88	145.87
beta_803	67.43	109.71
beta_804	-15.84	24.69
beta_805	-85.14	-75.33
beta_806	-152.39	-180.77
beta_807	-140.95	-215.37

beta_808	-73.63	-159.58
beta_809	-19.96	-66.46
beta_810	-36.40	4.84
beta_811	-170.24	-18.50
beta_812	-273.96	-54.33
beta_813	-119.98	-16.65
beta_814	166.43	-45.50
beta_815	44.68	35.72
beta_816	-34.56	5.90
beta_817	12.59	100.14
beta_818	47.86	56.68
beta_819	16.22	-261.54
beta_820	88.35	-12.63
beta_821	259.57	73.16
beta_822	-101.06	-91.16
beta_823	178.90	62.35
beta_824	63.12	-41.85
beta_825	-329.17	-217.86
beta_826	158.01	-60.11
beta_827	388.42	-41.86
beta_828	254.11	-140.03
beta_829	57.40	-180.42
beta_830	-62.41	-153.46
beta_831	-46.83	-98.88
beta_832	-6.32	-93.87
beta_833	62.79	-77.65
beta_834	90.03	-59.95
beta_835	68.06	-16.85
beta_836	1.21	27.36
beta_837	-66.63	73.37
beta_838	-133.87	77.09
beta_839	-169.13	64.33
beta_840	-196.43	15.89
beta_841	-161.06	-6.37
beta_842	-116.91	-8.02
beta_843	-56.18	15.04
beta_844	0.40	42.51
beta_845	23.43	29.52
beta_846	50.25	-13.41
beta_847	87.28	-81.83
beta_848	161.74	-117.65
beta_849	251.44	-98.75
beta_850	279.40	-50.36
beta_851	200.46	-12.94
beta_852	-9.90	-47.29
beta_853	-156.17	-80.86
beta_854	60.28	-2.63
beta_855	407.56	18.52

beta_856	145.65	103.29
beta_857	-308.28	-57.99
beta_858	-354.11	20.24
beta_859	105.66	12.97
beta_860	195.94	-143.53
beta_861	-	-83.15
beta_862	-	356.76
beta_863	-	-322.01
beta_864	-	-27.08
beta_865	-	19.31
beta_866	-	-59.97
beta_867	-	280.97
beta_868	-	411.71
beta_869	-	314.70
beta_870	-	205.34
beta_871	-	127.10
beta_872	-	58.15
beta_873	-	-8.62
beta_874	-	-24.66
beta_875	-	-30.55
beta_876	-	46.54
beta_877	-	120.83
beta_878	-	155.43
beta_879	-	119.80
beta_880	-	9.38
beta_881	-	-140.34
beta_882	-	-265.53
beta_883	-	-309.99
beta_884	-	-261.73
beta_885	-	-137.55
beta_886	-	33.16
beta_887	-	167.70
beta_888	-	231.74
beta_889	-	215.46
beta_890	-	157.41
beta_891	-	118.30
beta_892	-	75.02
beta_893	-	20.29
beta_894	-	-65.22
beta_895	-	-87.18
beta_896	-	26.91
beta_897	-	109.59
beta_898	-	200.52
beta_899	-	-169.34
beta_900	-	-195.55
beta_901	-	-61.52
beta_902	-	426.89

We see that we get some fairly large β values, although none that have "exploded".

5.4.2 Ridge Regression

[153]:
```
degree = 40
terrain_data.create_design_matrix(degree)
terrain_data.scale_dataset("standard")
terrain_data.split_dataset(test=test_fraction)

lmb_rr = 4.6e-8
rr_model = rr.RidgeRegression()
rr_model.fit(terrain_data.X_train,terrain_data.z_train,alpha=lmb_rr)
z_hat_rr = rr_model.predict(terrain_data.X_test)
print(f"MSE for ridge regression model with polynomial of degree={degree} and␣
↪lamda={lmb_rr}: ",rr_model.mean_square_error(z_hat_rr,terrain_data.z_test))
print(f"r^2 for ridge regression model with polynomial of degree={degree} and␣
↪lamda={lmb_rr}: ",rr_model.r2(z_hat_rr,terrain_data.z_test))
```

```
MSE for ridge regression model with polynomial of degree=40 and lamda=4.6e-08:
4866.8232176161855
r^2 for ridge regression model with polynomial of degree=40 and lamda=4.6e-08:
0.9025134166389549
```

[151]:
```
import warnings
import Code.LassoRegression as lr
warnings.filterwarnings("ignore", message=".*Objective did not converge.*")
nlambdas = 10
lambdas = np.logspace(-12, -5, nlambdas)
rr_model = rr.RidgeRegression()
lambda_kfold(rr_model,terrain_data,lambdas,k=5)
print(f"Figure 22: Mean square error as a function of lambda for ridge␣
↪regression on terrain data (d={degree}).")
```

```
MSE,r^2=[3.79209259e+03 9.26402757e-01] found for lambda=1e-12
MSE,r^2=[3.82591914e+03 9.25768204e-01] found for lambda=5.994842503189421e-12
MSE,r^2=[3.82394175e+03 9.25803358e-01] found for lambda=3.5938136638046254e-11
MSE,r^2=[3.82161267e+03 9.25795161e-01] found for lambda=2.1544346900318867e-10
MSE,r^2=[3.83107282e+03 9.25637906e-01] found for lambda=1.2915496650148826e-09
MSE,r^2=[3.80014389e+03 9.26143822e-01] found for lambda=7.742636826811278e-09
MSE,r^2=[3.50795491e+03 9.31782399e-01] found for lambda=4.641588833612782e-08
MSE,r^2=[3.80409620e+03 9.25154162e-01] found for lambda=2.782559402207126e-07
MSE,r^2=[4.36641809e+03 9.13093481e-01] found for lambda=1.6681005372000591e-06
MSE,r^2=[4.88415046e+03 9.01857594e-01] found for lambda=1e-05
```

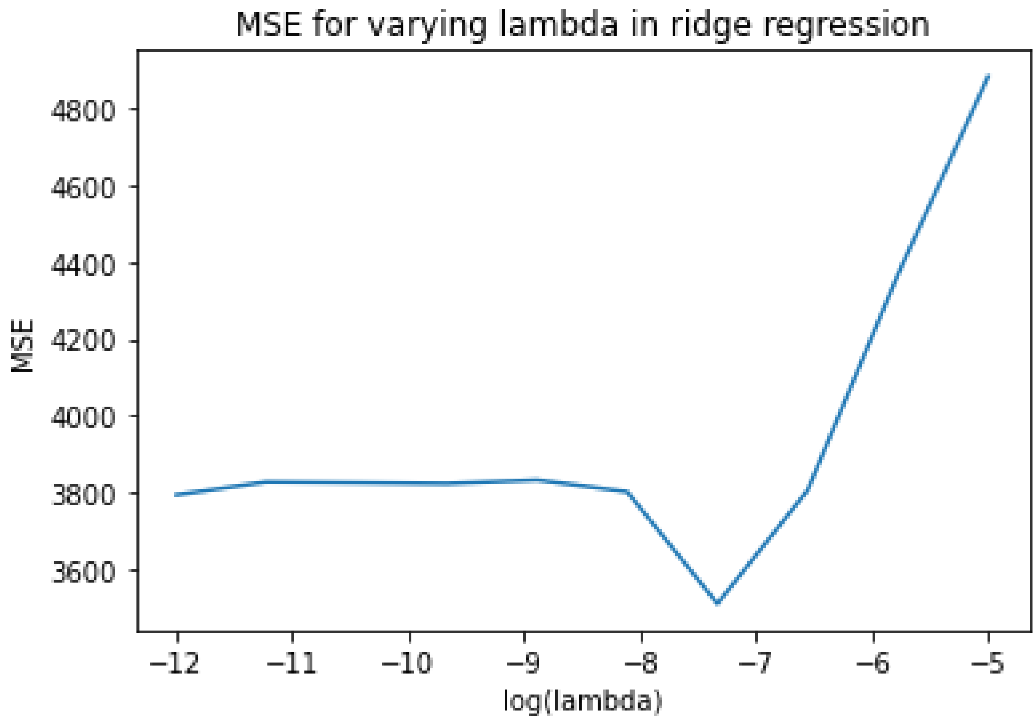


Figure 22: Mean square error as a function of lambda for ridge regression on terrain data (d=40).

We see that the minimum is found for $\lambda \approx 4.64 \cdot 10^{-8}$, so a very small value and the mean square error found matches approximately that for the highest polynomial degree we looked at for the OLS. We move on to look at lasso regression.

5.4.3 Lasso Regression

[156]:
```
lmb_lr = 2.15e-10
lr_model = lr.LassoRegression(alpha=lmb_lr)
lr_model.fit(terrain_data.X_train,terrain_data.z_train)
z_hat_lr = lr_model.predict(terrain_data.X_test)
print(f"MSE for ridge regression model with polynomial of degree={degree} and␣
 ↪lamda={lmb_lr}: ",lr_model.mean_square_error(z_hat_lr,terrain_data.z_test))
print(f"r^2 for ridge regression model with polynomial of degree={degree} and␣
 ↪lamda={lmb_lr}: ",lr_model.r2(z_hat_lr,terrain_data.z_test))
```

```
MSE for ridge regression model with polynomial of degree=40 and lamda=2.15e-10:
4698946.35775391
r^2 for ridge regression model with polynomial of degree=40 and lamda=2.15e-10:
-107.80657643617366
```

[155]:
```
import warnings
warnings.filterwarnings("ignore", message=".*Objective did not converge.*")
nlambdas = 10
lambdas = np.logspace(-12, -5, nlambdas)
lr_model = lr.LassoRegression(max_iter=10000)
lambda_kfold(rr_model,terrain_data,lambdas,k=10)
print(f"Figure 23: Mean square error as a function of lambda for lasso␣
 ↪regression on terrain data (d={degree}).")
```

```
MSE,r^2=[3.99264496e+03 9.22211884e-01] found for lambda=1e-12
MSE,r^2=[3.98748542e+03 9.22383931e-01] found for lambda=5.994842503189421e-12
MSE,r^2=[3.95660136e+03 9.22975923e-01] found for lambda=3.5938136638046254e-11
MSE,r^2=[3.89970775e+03 9.23888018e-01] found for lambda=2.1544346900318867e-10
MSE,r^2=[3.91768070e+03 9.23158451e-01] found for lambda=1.2915496650148826e-09
MSE,r^2=[4.41033102e+03 9.12095259e-01] found for lambda=7.742636826811278e-09
MSE,r^2=[4.86076043e+03 9.02374971e-01] found for lambda=4.641588833612782e-08
MSE,r^2=[5.36420229e+03 8.91155193e-01] found for lambda=2.782559402207126e-07
MSE,r^2=[5.85489672e+03 8.79991398e-01] found for lambda=1.6681005372000591e-06
MSE,r^2=[6.35089280e+03 8.68521853e-01] found for lambda=1e-05
```

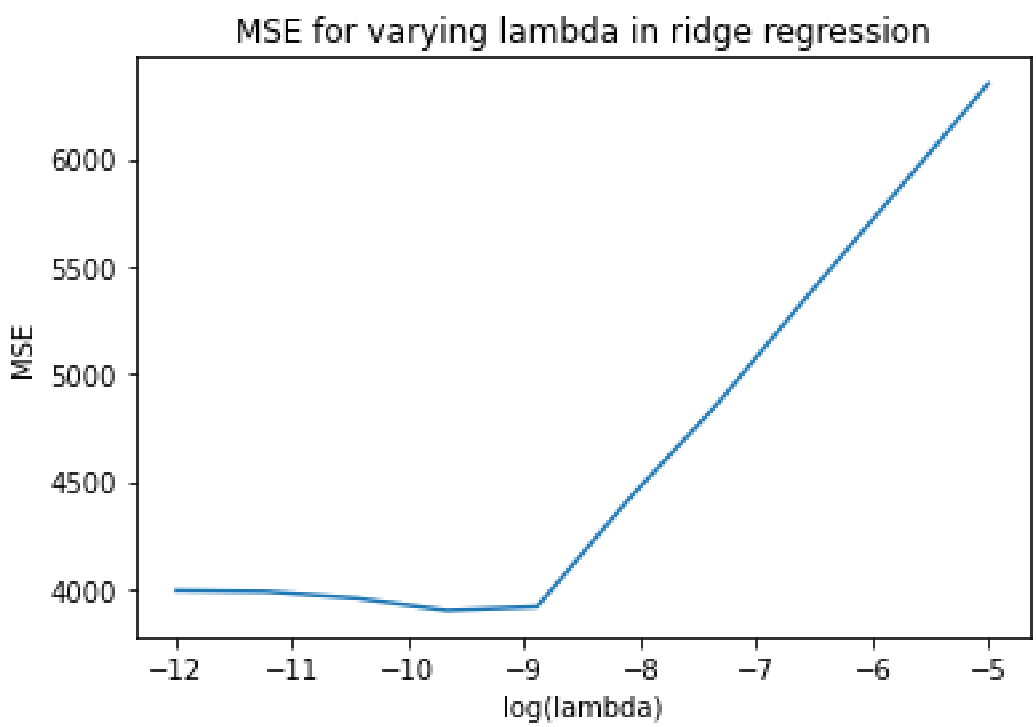


Figure 23: Mean square error as a function of lambda for lasso regression on terrain data (d=40).

Here I am getting some `ConvergenceWarning` warnings meaning we don't reach the convergence threshold within the maximum number of iterations, meaning that the result is a bit uncertain. However I am unable to increase the maximum number of iterations due to the large runtime required.

As expected, the lasso model behaves similarly to the ridge model we saw above in figure 22, but with the optimal λ being even smaller. However, the value of the mean square error is larger than for ridge.

5.4.4 Comparing Models with Cross-Validation

Now that we have explored the choice of parameters for our three models, let us compare their performance.

[157]:
```
k = 5
mse_r2_ols = ols_model.k_fold_cv(data.X_train,data.z_train,k,shuffle=True)
mse_r2_rr = rr_model.k_fold_cv(data.X_train,data.
 ↪z_train,k,alpha=lmb_rr,shuffle=True)
mse_r2_lr = lr_model.k_fold_cv(data.X_train,data.
 ↪z_train,k,alpha=lmb_lr,shuffle=True)
```

[161]:
```
print(f"mean(MSE)={mse_r2_ols[0]}, mean(r^2)={mse_r2_ols[1]} found for OLS with␣
 ↪d={degree}.")
print(f"mean(MSE)={mse_r2_rr[0]}, mean(r^2)={mse_r2_rr[1]} found for ridge with␣
 ↪d={degree} and lambda={lmb_rr}.")
print(f"mean(MSE)={mse_r2_lr[0]},  mean(r^2)={mse_r2_lr[1]} found for OLS with␣
 ↪d={degree} and lambda={lmb_lr}.")
```

```
mean(MSE)=0.010264029613844443, mean(r^2)=0.8702244823461693 found for OLS with
d=40.
mean(MSE)=0.010232850161120336, mean(r^2)=0.8706122958101249 found for ridge
with d=40 and lambda=4.6e-08.
mean(MSE)=0.18661862915304822,  mean(r^2)=-1.596097143872812 found for OLS with
d=40 and lambda=2.15e-10.
```

We see that OLS and ridge has an almost identical performance, while lasso is far worse. This matches what we saw for the Franke data, where the lasso model was more aggressive in reducing effective model complexity. Our terrain data is very complex, and as such we need a high complexity model to accurately represent it.

6 Conclusion

To conclude we that all our regression models performed better on our synthetic Franke data, than on our terrain data. This is not surprising as the terrain data was markedly more noisy and complex. We saw that for both data, OLS performed best out of the three models although ridge matched it's performance on the terrian data.

Ridge regression is mostly useful for reducing overfitting, which was not an issue we saw for either dataset, and as such it did not outperform the standard OLS model. Lasso on the other hand is particularly useful for high dimensionality models where it by eliminating some coefficients it performs feature selections. We did not see an improved fit from this, and it will likely work better on problems of very high dimensionality. Our terrain data had much higher dimensionality than our Franke data, and this is also where the lasso performs best, supporting this explanation.

One of the most useful results from a learning perspective was studying the $\boldsymbol{\beta}$ values for the three models which show quite clearly how they affect the final fits and give insight to their strengths and weaknesses.

7 Bibliography

[1] USGSs (the United States Geological Survey) EarthExplorer tool - https://earthexplorer.usgs.gov/

[2] Franke, R. (1979). A critical comparison of some methods for interpolation of scattered data

[3] SciKit-learn: https://scikit-learn.org/stable/index.html

[4] Numpy: https://numpy.org/

[5] MatPlotLib.PyPlot: https://matplotlib.org/api/pyplot_api.html

[6] Python.random: https://docs.python.org/3/library/random.html

[7] https://www.analyticsvidhya.com/blog/2016/01/ridge-lasso-regression-python-complete-tutorial/

[8] Python framework for unit testing: https://docs.python.org/3/library/unittest.html

[ ]: