FYS-STK4155 - Applied data analysis and machine learning

Project 1

Even M. Nordhagen

September 21, 2018

- Github repository containing programs and results: `https://github.com/evenmn/FYS-STK4155`

Abstract

Do not forget to be specific

Contents

1 Introduction

Should write some motivating words about how much regression is used in different fields etc.. Fit polynomial to the volcanic island of Lombok, Indonesia.

2 Theory

2.1 Linear regression

A few general words about regression

2.1.1 Ordinary Least Square (OLS)

Suppose we have a set of points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, and we want to fit a p'th order polynomial to them. The most intuitive way would be to find coefficients $\vec{\beta}$ which minimize the error in

$$\begin{aligned} y_1 &= \beta_0 x_1^0 + \beta_1 x_1^1 + \ldots + \beta_p x_1^p + \varepsilon_1 \\ y_2 &= \beta_0 x_2^0 + \beta_1 x_2^1 + \ldots + \beta_p x_2^p + \varepsilon_2 \\ &\vdots \\ y_n &= \beta_0 x_n^0 + \beta_1 x_n^1 + \ldots + \beta_p x_n^p + \varepsilon_n, \end{aligned}$$

which for OLS is defines as

$$\text{MSE} = \sum_i \varepsilon_i^2 \tag{1}$$

NEED TO REWRITE THIS + COST FUNCTION

Standard cost function

$$Q(\vec{\beta}) = \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = (\vec{y} - \hat{X}\vec{\beta})^T (\vec{y} - \hat{X}\vec{\beta}) \tag{2}$$

Instead of dealing with a set of equations, we can apply linear algebra. One can easily see that the equations above correspond to

$$\vec{y} = \hat{X}^T \vec{\beta} + \vec{\varepsilon}, \tag{3}$$

with

$$\hat{X} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \ldots & x_1^p \\ x_2^0 & x_2^1 & x_2^2 & \ldots & x_2^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & x_n^2 & \ldots & x_n^p \end{pmatrix} \tag{4}$$

as the *design* matrix and

$$\vec{\beta} = (\beta_0, \beta_1, \ldots, \beta_p). \tag{5}$$

For a nonsingular matrix $\hat{X}$ (but not necessary symmetric) we can find the optimal $\vec{\beta}$ by solving

$$\vec{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\vec{y}, \tag{6}$$

which again corresponds to minimizing the cost function,

$$\vec{\beta} = \text{argmin}_{\vec{\beta}}\Big\{ \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 \Big\}. \tag{7}$$

CONFIDENCE INTERVAL of $\hat{\beta}$: $\text{Var}(\hat{\beta})$.

This works perfectly when all rows in $\hat{X}$ are linearly independent, but this will generally not be the case for large data sets. If we are not able to diagonalize the matrix, we will not be able to calculate $(\hat{X}^T\hat{X})^{-1}$, so we need to do something smart.

Fortunately there is a simple trick we can do to make all matrices diagonalizable; we can add a diagonal matrix to the initial matrix.

2.1.2 Ridge regression

Ridge regression is a widely used method that can handle singularities in matrices. The idea is to modify the standard cost function by adding a small term,

$$Q^{\text{ridge}}(\vec{\beta}) = \sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 + \lambda||\vec{\beta}||_2^2, \tag{8}$$

where λ is the so-called *penalty* and $||\vec{v}||_2$ is defined as

$$||\vec{v}||_2 = \sqrt{\vec{v}^T\vec{v}} = \Big(\sum_{i=1}^{N} v_i^2\Big)^{1/2}. \tag{9}$$

This will eliminate the singularity problem.

Further we find the optimal $\vec{\beta}$ values by minimizing the function

$$\vec{\beta}^{\text{ridge}} = \text{argmin}_{\vec{\beta}}\Big\{ \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \Big\}, \tag{10}$$

or we could simply solve the equation

$$\vec{\beta}^{\text{ridge}} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\vec{\beta}. \tag{11}$$

In the latter equation we can easily see why this solves our problem.

2.1.3 Lasso regression

The idea behind Lasso regression is similar to the idea behind Ridge regression, and they differ only by the exponent factor in the last term. The modified cost function now writes

$$Q^{\text{lasso}}(\vec{\beta}) = \sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 + \lambda||\vec{\beta}||_2, \tag{12}$$

and to find the optimal coefficients $\vec{\beta}$, we need to minimize

$$\vec{\beta}^{\text{lasso}} = \text{argmin}_{\vec{\beta}}\left\{ \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j \right\}. \tag{13}$$

2.1.4 General form

We can generalize the models above to a minimization problem where we have a q in the last exponent,

$$\vec{\beta}^{q} = \text{argmin}_{\vec{\beta}}\left\{ \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^q \right\}, \tag{14}$$

such that $q = 2$ corresponds to the Ridge method and $q = 1$ is associated with Lasso regression. It can also be interesting to try other q-values.

2.2 Higher order regression

We can use the same approach as above when dealing with regression of higher order, since the problem is to fit a function to points, no matter how many components they have. We will first take a look at how we can fit a 2D polynomial to some terrain data, before we briefly describe how to fit a function of arbitrary order to points.

2.2.1 Terrain

A set of data points $\{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\}$ gives some coordinates in space, which for instance can describe the terrain. The system of linear equations to solve can then be lined up as

$$\begin{aligned}
z_1 &= \beta_0 x_1^0 y_1^0 + \beta_1 x_1^1 y_1^0 + \beta_2 x_1^0 y_1^1 + \ldots + \beta_p x_1^p y_1^p + \varepsilon_1 \\
z_2 &= \beta_0 x_2^0 y_2^0 + \beta_1 x_2^1 y_2^0 + \beta_2 x_2^0 y_2^1 + \ldots + \beta_p x_2^p y_2^p + \varepsilon_2 \\
&\vdots \\
z_n &= \beta_0 x_n^0 y_n^0 + \beta_1 x_n^1 y_n^0 + \beta_2 x_n^0 y_n^1 + \ldots + \beta_p x_n^p y_n^p + \varepsilon_n,
\end{aligned}$$

when fitting a polynomial of p'th order. In general the order associated with x-direction do not need to be the same as the order associated with y-direction.

We can set up a similar equation as for the first order case, presented in equation (3), but we will now (typically) have $\vec{z}$ on the left hand side and $\hat{X}$ will contain both x- and y-coordinates:

$$\vec{z} = \hat{X}^T \vec{\beta}. \tag{15}$$

The design matrix $\hat{X}$ will now look like

$$\hat{X} = \begin{pmatrix} x_1^0 y_1^0 & x_1^1 y_1^0 & x_1^0 y_1^1 & x_1^1 y_1^1 & \dots & x_1^p y_1^p \\ x_2^0 y_2^0 & x_2^1 y_2^0 & x_2^0 y_2^1 & x_2^1 y_2^1 & \dots & x_2^p y_2^p \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^0 y_n^0 & x_n^1 y_n^0 & x_n^0 y_n^1 & x_n^1 y_n^1 & \dots & x_n^p y_n^p \end{pmatrix} \tag{16}$$

and we can again use the OLS method to find $\vec{\beta}$, i.e, calculating

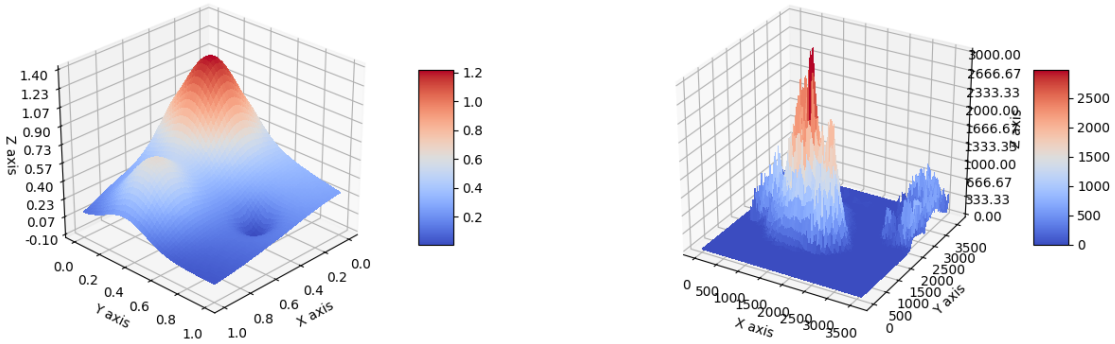
$$\vec{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T \vec{z}. \tag{17}$$

Similarly, we can use Ridge and Lasso regression in the same way as when fitting 1D polynomials.

In this project we will actually deal with terrain data. Firstly, we implement some regression tools which fit a 2D function to points in space. To verify the implementation, we pick points from a known function such that we know the result. The specific verification function used is the Franke Function, which looks like

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right), \end{aligned}$$

and is a smooth 2D function as one can see in figure (1), part (a).



(a) Franke function

(b) Lombok terrain

Figure 1: The two data sets we are going to fit polynomials to.

Secondly, we use the verified regression tools to fit a polynomial to real terrain data. The data is taken from United States Geological Survey (USGS) `https://earthexplorer.usgs.gov/`, and for investigation we picked the volcanic island of Lombok in Indonesia. Despite the island's small extent, it houses the second highest volcano in Indonesia which makes the terrain data interesting. See part (b) of figure (1) for terrain data.

2.2.2 Higher order

There should be no surprise that we can extend the theory above to even higher orders. Although we stick to 2D regression in this project, we add this section for completeness.

2.3 Error analysis

When dealing with data sets, we always want to know how much the data points vary from each other, in other words we want to calculate the variance of the data. Often we are studying multiple data sets at the same time, ...

Sample variance:

$$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{\mu})^2 \tag{18}$$

Sample mean:

$$\bar{\mu} = \sum_{i=1}^{N} x_i \tag{19}$$

Sample mean and distribution mean are not necessarly the same, could point it our by show an example on both (perhaps by using Franke function?). Mean value in one data set α:

$$\bar{\mu}_\alpha = \frac{1}{N}\sum_{k=1}^{N} X_{\alpha,k} \tag{20}$$

Total sample mean of M data sets:

$$\bar{\mu}_M = \frac{1}{M}\sum_{\alpha=1}^{M} \bar{\mu}_\alpha = \frac{1}{MN}\sum_{\alpha=1}^{M}\sum_{k=1}^{N} X_{\alpha,k} \tag{21}$$

NEED TO DECIDE WHETHER I USE MU OR X.

The total variance will then be a sum over the single set variances plus the cross terms:

$$\sigma_M^2 = \frac{\sigma^2}{N} + \frac{2}{MN^2}\sum_{\alpha=1}^{M}\sum_{k<l}^{N}(X_{\alpha,k} - \bar{\mu}_M)(X_{\alpha,l} - \bar{\mu}_M). \tag{22}$$

The last term is called the covariance, and is a measure on how much the data set are related. If they are totally independent, the covariance is zero. For large data sets this term is computational expensive to calculate, and we will rather use resampling techniques to estimate the variance. A few resampling techniques are presented in the Method section.

Cost function (loss function)

Different methods to estimate error:

- Absolute error
- Relative error
- Mean square error (MSE)
- R^2 score function

3 Methods

3.1 Resampling techniques

A resampling technique is a way of estimating the variance of data sets without calculating the covariance. As we saw in section section **??**, the true covariance is given by a double loop which we will avoid calculating if possible. There are different ways of doing this, and we have already went through several of them in this course:

- Jackknife resampling
- K-fold validation
- Bootstrap method
- Blocking method.

For this particular project we have been focusing on the bootstrap and the K-fold validation methods, so only they will be covered here.

3.1.1 Bootstrap method

When we construct a data set, we usually draw samples from a probability density function (PDF) and get a set of samples $\vec{x}$. If we draw a large number of samples, the sample variance will approach the variance of the PDF. The bootstrap method turns this upside down, and tries to estimate the PDF given a set data set, because if we know the PDF, we know in principle everything about the data set.

The assumption we need to make, is that the relative frequency of x_i equals $p(x_i)$, which is reasonable (for instance, think about how the histogram looks like when we draw samples from a normal distribution). In this project the vector that we want to find, $\vec{\beta}(x, y)$, is a function of two set of variables. Fortunetely they are independent of eachother, so we can safely apply the independent bootstrap on them separately. The independent bootstraps goes as

1. Draw n samplings from the data set $\vec{x}$ with replacement and denote the new data set as $\vec{x}^* = \{x_1, x_2, \ldots, x_n\}$
2. Compute $\vec{\beta}(\vec{x}^*) \equiv \vec{\beta}^*$
3. Repeat the procedure above K times
4. The average value of all K $\vec{\beta}^*$'s are stored in a new vector $\vec{\bar{\beta}}^*$
5. Finally, the variance of $\vec{\bar{\beta}}^*$ should correspond to the sample variance

[**BootstrapEfron**] [Efron, B. Bootstrap Methods: Another Look at the Jackknife. Ann. Statist. 7 (1979), no. 1, 1–26. doi:10.1214/aos/1176344552.]

The implementation could look something like this

```
def bootstrap(data, K=1000):
    dataVec = np.zeros(K)
    for k in range(K):
        dataVec[k] = np.average(np.random.choice(data, len(data)))
    Avg = np.average(dataVec)
    Var = np.var(dataVec)
    Std = np.std(dataVec)

    return Avg, Var, Std
```

3.1.2 K-fold validation method

3.2 Minimization methods

When the interaction term is excluded, we know which α that corresponds to the energy minimum, and it is in principle no need to try different α's. However, sometimes we have no idea where to search for the minimum point, and we need to try various α values to determine the lowest energy. If we do not know where to start searching, this can be a time consuming activity. Would it not be nice if the program could do this for us?

In fact there are multiple techniques for doing this, where the most complicated ones obviously also are the best. Anyway, in this project we will have good initial guesses, and are therefore not in need for the most fancy algorithms.

3.2.1 Gradient Descent

Perhaps the simplest and most intuitive method for finding the minimum is the gradient descent method (GD), which reads

$$\beta_i^{\text{new}} = \beta_i - \eta \cdot \frac{\partial Q(\beta_i)}{\partial \beta_i} \tag{23}$$

where β_i^{new} is the updated β and η is a step size, in machine learning often refered to as the learning rate. The idea is to find the gradient of the cost function $Q(\vec{\beta})$ with respect to a certain β_i, and move in the direction which minimizes the cost function. This is repeated until a minimum is found, defined by either

$$\frac{\partial Q(\beta_i)}{\partial \beta_i} < \varepsilon \tag{24}$$

or that the change in β_i for the past x steps is small.

Before we can implement equation (23), we need an expression for the derivative of Q with respect to β_i. The general form of the cost function as discussed in section 2.1.4 reads

$$Q(\vec{\beta}, \lambda, q) = \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \Big)^2 + \lambda \sum_{j=1}^{p} \beta_j^q, \tag{25}$$

and its derivative with respect to β_k is

$$\frac{\partial Q(\vec{\beta}, \lambda, q)}{\partial \beta_k} = -2 \sum_{i}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \Big) x_{ik} + q\lambda \beta_k^{q-1}. \tag{26}$$

The vectorized version looks like

$$\frac{\partial Q(\vec{\beta},\lambda,q)}{\partial \vec{\beta}} = -2\hat{X}^T(\vec{y}-\hat{X}\vec{\beta}) + q\lambda\vec{\beta}^{q-1} \tag{27}$$

The algorithm of this minimization method is thus as follows:

```
while dbeta > epsilon:
        e = y - X.dot(beta)
    debeta = 2*X.T.dot(e) - q*\lambda*np.power(abs(beta), q-1)
    beta += \eta*dbeta
```

4 Code

This project is largely based on numerical calculation, so a few words about the code is necessary. The code is written in Python, which is easy to work with and considered fast enough. The most expensive part is without doubt the minimization using gradient descent, and for really large systems a low-level language might be preferred. Although no performance benchmarks will be provided in this article, we can reveal that the code was more than fast enough for our data sets. To run the code, the following packages are required:

- NumPy - Fundamental package for scientific works in Python
- Matplotlib - For plotting
- Scipy - For reading terrain data
- tqdm - For progression bar

4.1 Code structure

To keep the code neat and clean we decided to write object oriented code, although we do not have that many functions. The code of this project consists of two main functions fit_franke.py and fit_terrain.py, where the former is used to test the tools on a known function and the latter is used to fit a polynomial to terrain data. Both of them are calling the classes for 2D regression and resampling techniques, named regression_2D.py and resampling.py respectively, and fit_franke.py is also communicating with franke.py. For overview of the code structure, see figure (2).

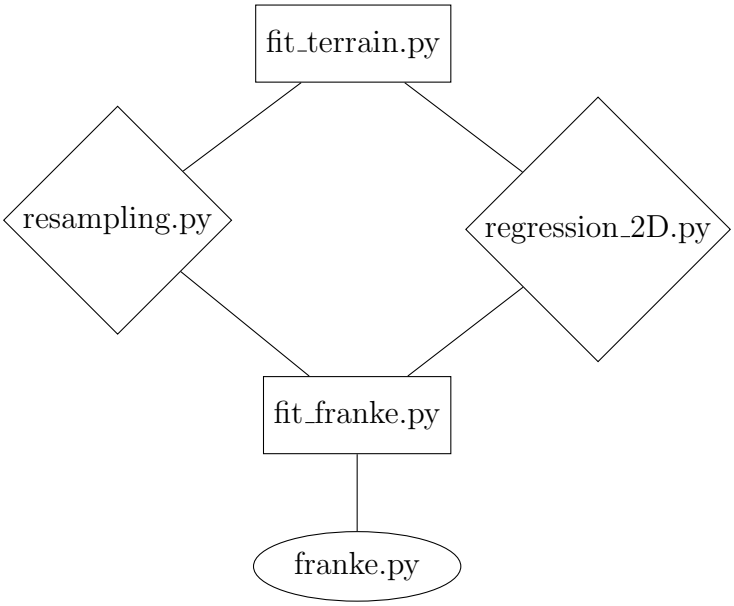


Figure 2: Code structure

4.2 Implementation and optimization

To get a code which provides good performance, we need to be thoughtful when implementing it. The loops are often bottlenecks, so by replacing loops with vector operations we can save a lot of time. An example on this, is when we calculate the mean square error (MSE) given by

$$\text{MSE}(\vec{y}) = \sum_i \left(y_i - \beta_0 - \sum_j X_{ij}\beta_j \right)^2, \tag{28}$$

where $\vec{y}$ and $\vec{\beta}$ are vectors and $\hat{X}$ is a matrix. For implementing this directly, we need a double loops, which will be slow for large systems. A better solution would be to exploit the linear algebra properties of vectors:

$$\text{MSE}(\vec{y}) = (\vec{y} - \hat{X}^T\vec{\beta})^T \cdot (\vec{y} - \hat{X}^T\vec{\beta}) \tag{29}$$

which is usually much faster.

5 Results

5.1 Franke function

First we will take a look at how good our linear regression is...

5.1.1 Visualization of graphs

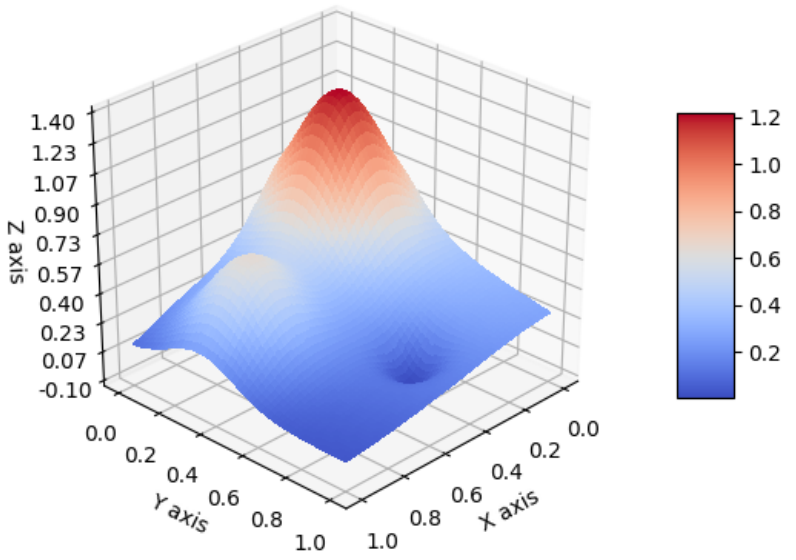
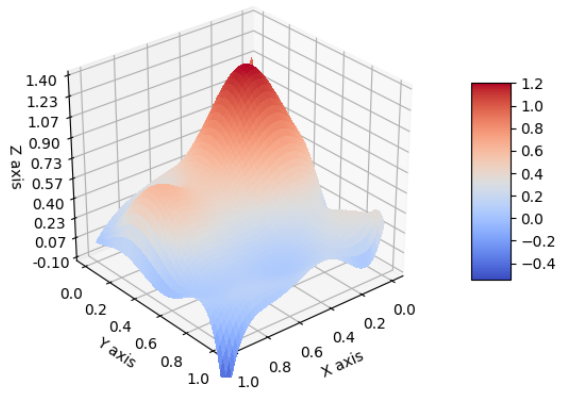
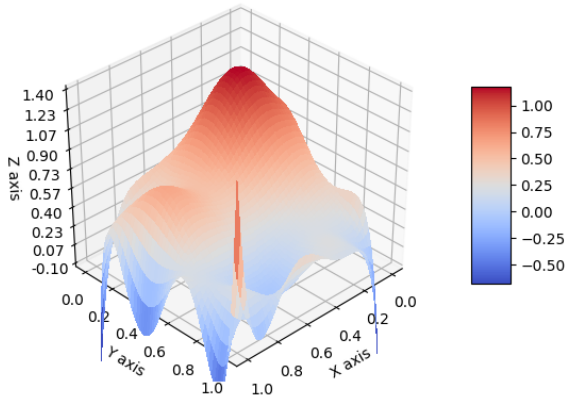


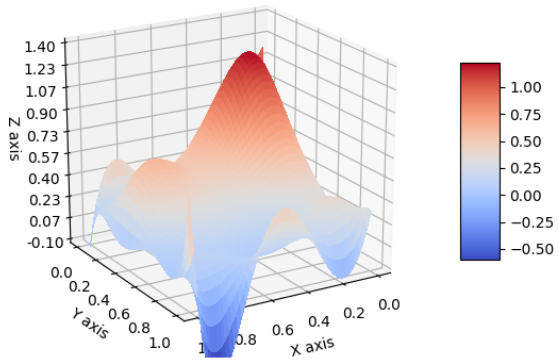
Figure 3: The Franke function in the interval $x \in [0, 1]$, $y \in [0, 1]$, $z \in [0, 1.2]$.



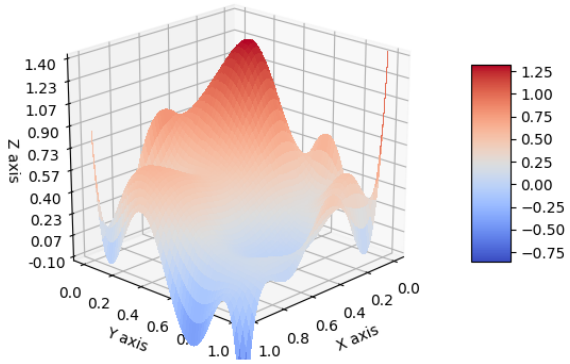
(a) OLS without noise



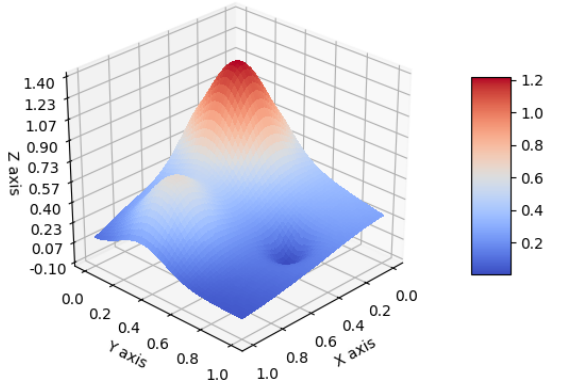
(b) OLS with noise



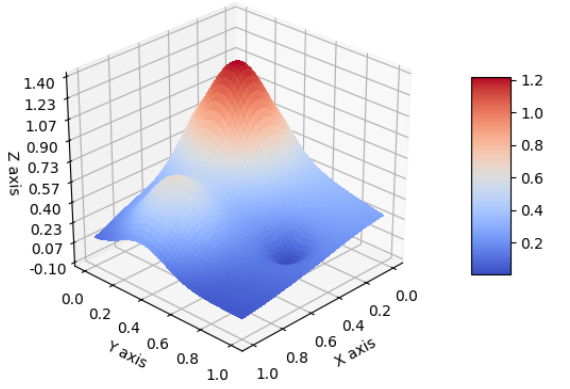
(c) Ridge without noise



(d) Ridge with noise



(e) Lasso without noise



(f) Lasso with noise

Figure 4: Fitted polynomial by OLS, Ridge and Lasso with and without noise. For these plots, we used a low penalty of $\lambda = 1e - 15$.

5.1.2 Error

5.2 Real data

First we will take a look at how good our linear regression is...

5.2.1 Visualization of graphes

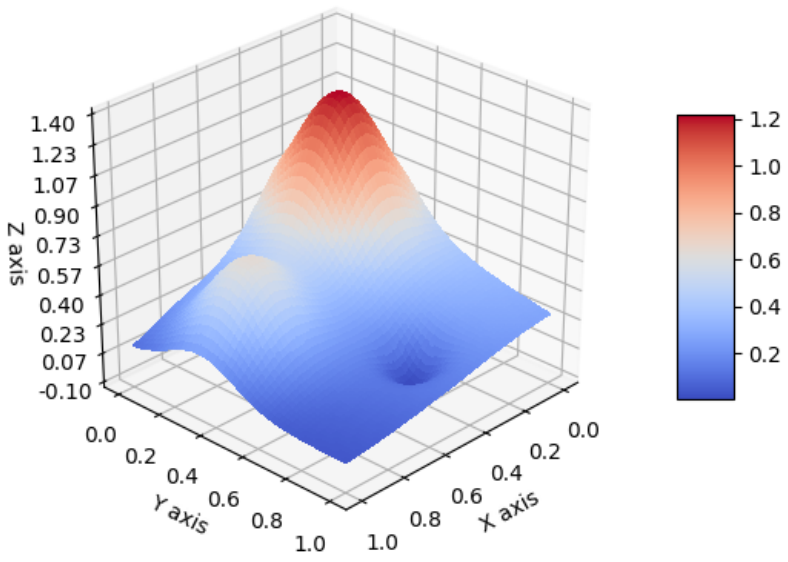
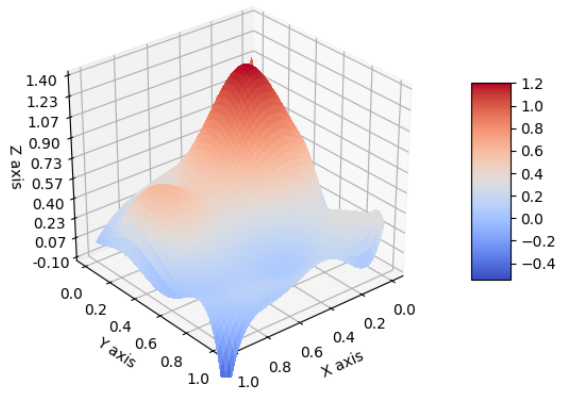
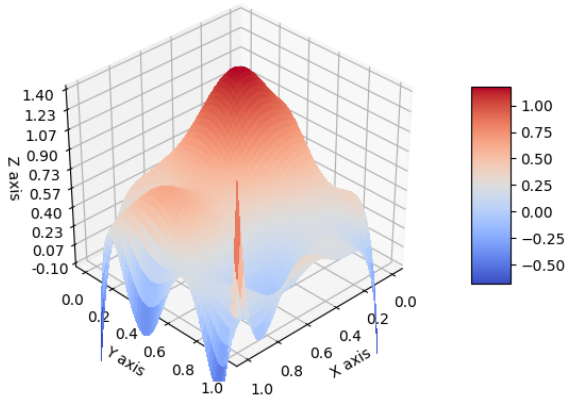


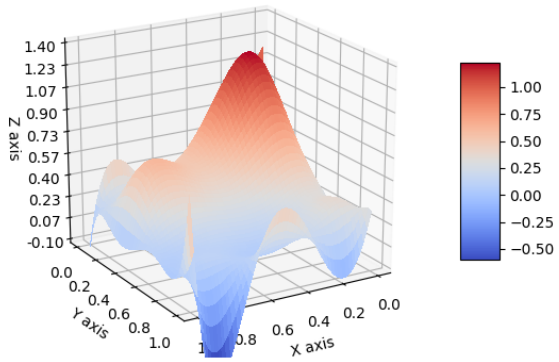
Figure 5: The Franke function in the interval $x \in [0, 1]$, $y \in [0, 1]$, $z \in [0, 1.2]$.



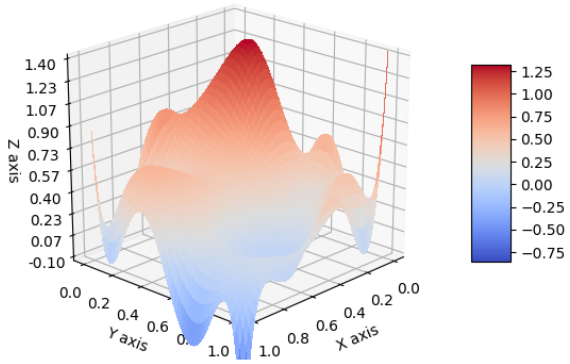
(a) OLS without noise



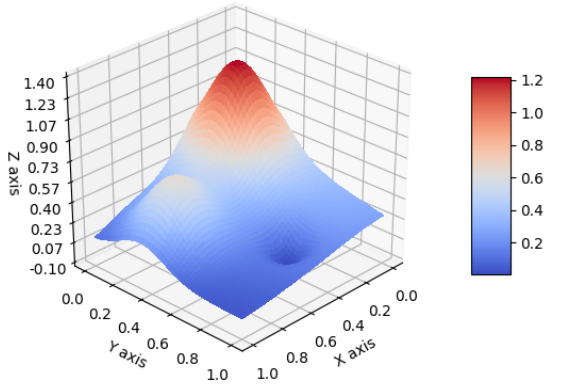
(b) OLS with noise



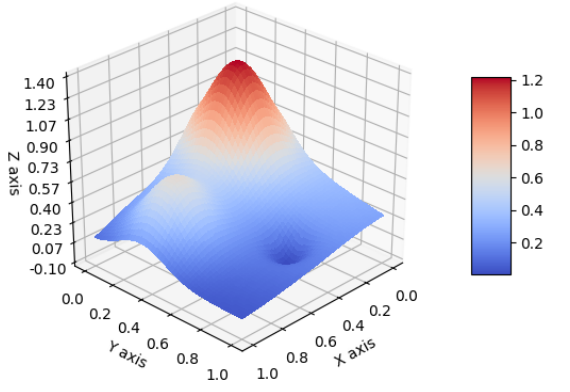
(c) Ridge without noise



(d) Ridge with noise



(e) Lasso without noise



(f) Lasso with noise

Figure 6: Fitted polynomial by OLS, Ridge and Lasso with and without noise. For these plots, we used a low penalty of $\lambda = 1e - 15$.

6 Discussion

7 Conclusion

A Appendix A