FYS-STK4155 - Applied data analysis and machine learning

Project 1

Even M. Nordhagen

October 3, 2018

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

Abstract

The aim of this project is... We found out that... Do not forget to be specific

Contents

1 Introduction

The linear regression was first introduced for more than two hundred years ago, and have been used in a large number of fields after that [**Gauss**][**Legende**]. With this project we examine how good the method is for fitting polynomials to real terrain data, and to see if more complicated regression methods are needed or not. To challenge the method, we chose terrain data from the volcanic island of Lombok, Indonesia, where the contour lines are quite dense.

We developed our own software for ordinary least square (OLS), Ridge and Lasso linear regression, where the latter was based on minimization using gradient descent (GD). To verify the implementation, we tested it on data from the Franke function where we knew what the result should be. Further, the error was analyzed in order to decide which method that gave the best result, and all data was resampled using the bootstrap method to find the correct sampling variance.

For the results, see section *Results* (5), which again is discussed in section *Discussion* (**??**). The background theory can be found in section *Theory* (2), and all methods and techniques are presented in the section *Methods* (3). For code structure and implementation, see section *Code* (4), and finally, the conclusion is found in section (**??**) with the same name.

2 Theory

Regression analysis are widely used in different fields of natural sciences, data science and economics to mention some. The simplest form is the linear regression, which is intuitive and easy to work with.

2.1 Linear regression

In linear regression, the dependent variable y_i is a linear combination of the parameters. [**Hastie**]

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

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

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

$$\vec{y} = \hat{X}^T \vec{\beta} + \vec{\varepsilon}, \tag{1}$$

with

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

as the *design* matrix and

$$\vec{\beta} = (\beta_0, \beta_1, \ldots, \beta_p) \tag{3}$$

as the coefficients.

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

$$\vec{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T \vec{y}, \tag{4}$$

which again corresponds to minimizing the cost function,

$$\vec{\beta} = \text{argmin}, \vec{\beta} \Big\{ \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \Big)^2 \Big\}. \tag{5}$$

where we here have used the standard cost function, given by least squares,

$$Q(\vec{\beta}) = \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2 = (\vec{y} - \hat{X}\vec{\beta})^T (\vec{y} - \hat{X}\vec{\beta}). \tag{6}$$

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,

$$Q^{\text{ridge}}(\vec{\beta}) = \sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 + \lambda||\vec{\beta}||_2^2, \tag{7}$$

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

$$||\vec{v}||_2 = \sqrt{\vec{v}^T\vec{v}} = \Big(\sum_{i=1}^{N} v_i^2\Big)^{1/2}. \tag{8}$$

This will eliminate the singularity problem.

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

$$\vec{\beta}^{\text{ridge}} = \text{argmin}_{\vec{\beta}}\Big\{\sum_{i=1}^{n}\Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\Big\}, \tag{9}$$

or we could simply solve the equation

$$\vec{\beta}^{\text{ridge}} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\vec{\beta}. \tag{10}$$

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

$$Q^{\text{lasso}}(\vec{\beta}) = \sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 + \lambda||\vec{\beta}||_2, \tag{11}$$

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

$$\vec{\beta}^{\text{lasso}} = \text{argmin}_{\vec{\beta}}\Big\{\sum_{i=1}^{n}\Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}|\beta_j|\Big\}. \tag{12}$$

2.1.4 General form

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

$$\vec{\beta}^q = \text{argmin}_{\vec{\beta}} \Big\{ \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \Big)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q \Big\}, \tag{13}$$

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 Multivariate linear 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

$$\begin{aligned}
z_1 &= \beta_0 x_1^0 y_1^0 + \beta_1 x_1^1 y_1^0 + \beta_2 x_1^0 y_1^1 + \ldots + \beta_{p^2} x_1^p y_1^p + \varepsilon_1 \\
z_2 &= \beta_0 x_2^0 y_2^0 + \beta_1 x_2^1 y_2^0 + \beta_2 x_2^0 y_2^1 + \ldots + \beta_{p^2} x_2^p y_2^p + \varepsilon_2 \\
&\vdots \\
z_n &= \beta_0 x_n^0 y_n^0 + \beta_1 x_n^1 y_n^0 + \beta_2 x_n^0 y_n^1 + \ldots + \beta_{p^2} x_n^p y_n^p + \varepsilon_n,
\end{aligned}$$

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 (1), but we will now (typically) have $\vec{z}$ on the left hand side and $\hat{X}$ will contain both x- and y-coordinates:

$$\vec{z} = \hat{X}^T \vec{\beta}. \tag{14}$$

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

$$\hat{X} = \begin{pmatrix}
x_1^0 y_1^0 & x_1^1 y_1^0 & x_1^0 y_1^1 & x_1^1 y_1^1 & \ldots & x_1^p y_1^p \\
x_2^0 y_2^0 & x_2^1 y_2^0 & x_2^0 y_2^1 & x_2^1 y_2^1 & \ldots & x_2^p y_2^p \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
x_n^0 y_n^0 & x_n^1 y_n^0 & x_n^0 y_n^1 & x_n^1 y_n^1 & \ldots & x_n^p y_n^p
\end{pmatrix} \tag{15}$$

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

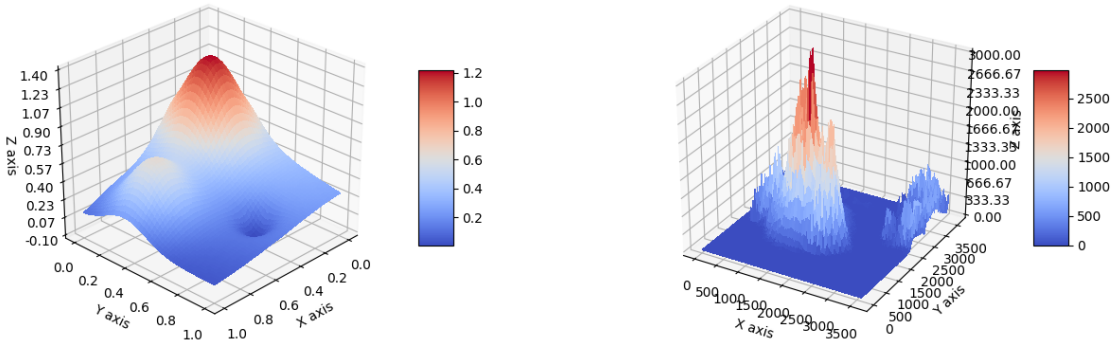
$$\vec{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T \vec{z}. \tag{16}$$

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) [**USGS**], and for investigation we picked the volcanic island of Lombok. 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.

Suppose now that we have a data set of n points in d dimensions, $\{(x_1^1, x_1^2, \ldots, x_1^d), (x_2^1, x_2^2, \ldots, x_2^d), \ldots, (x_n^1, x_n^2, \ldots, x_n^d)\}$, where the superscript indicates in which dimension the coordinate is, and the subscript is the coordinate number. We want to fit the points with a polynomial of degree $p \in \mathbb{R}^d$, where we assume that the fitting polynomial has the same degree in all direction, which makes the notation neater, but is not essential. To be more specific, we want polynomial coefficients $\vec{\beta} = (\beta_0, \beta_1, \ldots, \beta_{p^d})$ that minimizes the error in

$$
\begin{aligned}
x_1^d &= \beta_0 (x_1^1)^0 (x_1^2)^0 \ldots (x_1^d)^0 + \beta_1 (x_1^1)^1 (x_1^2)^0 \ldots (x_1^d)^0 + \ldots + \beta_{p^d} (x_1^1)^p (x_1^2)^p \ldots (x_1^d)^p + \varepsilon_1 \\
x_2^d &= \beta_0 (x_2^1)^0 (x_2^2)^0 \ldots (x_2^d)^0 + \beta_1 (x_2^1)^1 (x_2^2)^0 \ldots (x_2^d)^0 + \ldots + \beta_{p^d} (x_2^1)^p (x_2^2)^p \ldots (x_2^d)^p + \varepsilon_2 \\
&\vdots \\
x_n^d &= \beta_0 (x_n^1)^0 (x_n^2)^0 \ldots (x_n^d)^0 + \beta_1 (x_n^1)^1 (x_n^2)^0 \ldots (x_n^d)^0 + \ldots + \beta_{p^d} (x_n^1)^p (x_n^2)^p \ldots (x_n^d)^p + \varepsilon_n.
\end{aligned}
$$

This gives a design matrix

$$
\hat{X} = \begin{pmatrix}
(x_1^1)^0 (x_1^2)^0 \ldots (x_1^d)^0 & \ldots & (x_1^1)^1 (x_1^2)^0 \ldots (x_1^d)^0 & \ldots & (x_1^1)^p (x_1^2)^p \ldots (x_1^d)^p \\
(x_2^1)^0 (x_2^2)^0 \ldots (x_2^d)^0 & \ldots & (x_2^1)^1 (x_2^2)^0 \ldots (x_2^d)^0 & \ldots & (x_2^1)^p (x_2^2)^p \ldots (x_2^d)^p \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(x_n^1)^0 (x_n^2)^0 \ldots (x_n^d)^0 & \ldots & (x_n^1)^1 (x_n^2)^0 \ldots (x_n^d)^0 & \ldots & (x_n^1)^p (x_n^2)^p \ldots (x_n^d)^p,
\end{pmatrix} \tag{17}
$$

and we can find the optimal $\vec{\beta}$ in the same way as above.

2.3 Some statistics

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. The variance from one data set is given by

$$
\sigma_y^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{18}
$$

where the sample mean is given by

$$
\bar{y} = \sum_{i=1}^{N} y_i. \tag{19}
$$

We will stress that sample mean is not necessarily the same as the distribution mean, but the sample mean will approximate the distribution mean for large data sets, known as the *central limit theorem*. [**StatMek**]

Often we are studying multiple data sets at the same time, and want to estimate the total variance. The sample mean of one data set α is still given by

$$\bar{y}_\alpha = \frac{1}{N}\sum_{i=1}^{N} y_{\alpha,i}, \tag{20}$$

which for M data sets gives a total sample mean of

$$\bar{y}_M = \frac{1}{M}\sum_{\alpha=1}^{M}\bar{y}_\alpha = \frac{1}{MN}\sum_{\alpha=1}^{M}\sum_{i=1}^{N} y_{\alpha,i}. \tag{21}$$

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

$$\sigma_M^2 = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{\sigma_\alpha^2}{N} + \frac{2}{MN^2}\sum_{\alpha=1}^{M}\sum_{i<j}^{N}(y_{\alpha,i}-\bar{y}_M)(y_{\alpha,j}-\bar{y}_M). \tag{22}$$

The last term is called the covariance, and is a measure of 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.

2.3.1 Confidence interval

The confidence interval is the interval that you are within with a probability given by the confidence level. The confidence interval (CI) means

$$\text{CI} = \bar{y} \pm z^* \frac{\sigma}{\sqrt{N}} \tag{23}$$

where $\bar{y}$ is the sample mean, σ is the standard deviation of the PDF, N is the number of points and z^* is directly related to the confidence level and can be found in statistics tables.

As an example, assume that you flip a coin N times and count number of heads. Before you start, you want to know how many heads you will count with a 95% confidence. The procedure for calculating the CI is then

1. Calculate the sample mean of your data set
2. Find z^* from a table. For 95% confidence, the Z-score $z^* = 1.96$

3. Calculate the standard deviation of the PDF. The clue in this example, is that the PDF is approximating a binomial distribution when N increases, with a known σ [https://www.sumproduct.com/thought/simulation-stimulation]

4. Plug into the formula above

So, how can we make us of this in project 1? To predict which coefficients we could get from the regression, it could be useful to calculate their CI. This is possible, since we can find their variance from the analytical formula

$$\text{Var}(\vec{\beta}) = (\hat{X}^T\hat{X})^{-1}\sigma^2 \tag{24}$$

where the σ^2 is the sample variance of the points.

Another approx is to simply calculate N estimates of each beta, and sort them in increasing order. Leaving out the $p\%$ first elements of the data set and the $p\%$ last elements, we are left with the $q = 100\% - 2p\%$ most likely betas, and the confidence interval related to a confidence level of $q\%$ is within the smallest and largest elements of this new set.

2.4 Error analysis

Different methods to estimate error:

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

In this report we will study only the MSE and R^2 score function.

2.4.1 Mean Square Error (MSE)

The most popular error estimation method is the mean square error, also called least squares. [**Hastie**] We study the MSE in order to find out how the cost function is reduced, because it is basically the standard cost function, used in OLS.

$$\text{MSE}(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}(\vec{\beta}))^2 \tag{25}$$

Compared to least absolute value, the points far away from the fitted line are weighted stronger. We will also study a quantity called bias, which is just the sum over differences between the model and all points,

$$\text{bias}(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}(\vec{\beta})). \tag{26}$$

As one can see, the bias can both be positive and negative, where a large positive bias indicates underfitting and a large negative bias indicates overfitting. [**Berkley**] It can be shown that the MSE is a sum of the bias squared and the variance,

$$\text{MSE}(\vec{\beta}) = \text{bias}(\vec{\beta})^2 + \sigma^2, \tag{27}$$

and based on this we can show that the optimal complexity of our model (which is neither underfitted nor overfitted) is when the bias and variance are equal, see figure (2).

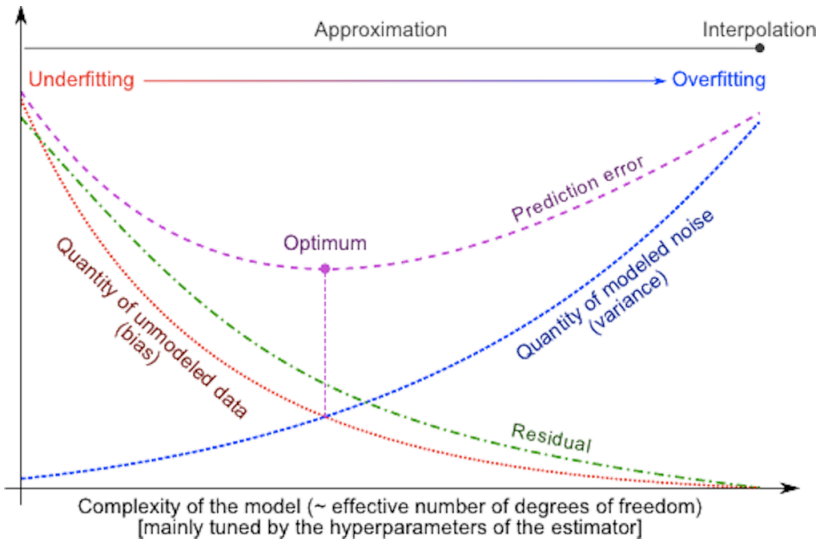


Figure 2: Bias-variance trade-off. Figure showing the bias (red graph). variance (blue graph) and MSE (purple graph) as functions of the complexity of the model. Figure taken from `https://www.reddit.com/r/mlclass/comments/mmlfu/a_nice_alternative_explanation_of_bias_and/`

As we can see from the figure, the bias decreases and the variance increases as the model gets more complex. The MSE is minimized when $\sigma^2 = \text{bias}$.

2.4.2 R^2 score function

The R^2 score function is a measure of how close the data are to the fitted regression line, and is a widely used quantity within statistics. [**R2**]

It is defined by how much of the variation that is not explained by the model, i.e,

$$\mathrm{R}^2 = \frac{\text{Explained variation}}{\text{Total variation}}$$

where the explained variation is the difference between the variance and the MSE and the total variation is given by

$$\text{Total variation} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2.$$

The R^2-score is always between 0 and 1, it is 0 if the model does not explain any of the variations and 1 if it explains all variations. In that manner, we should fight for a high R2-score. It is in entirety given by

$$R^2(\vec{y}, \tilde{\vec{y}}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \tilde{y})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}. \tag{28}$$

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 2.4, 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. Fortunately, 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**]

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
```

which is strongly inspired by the lecture notes [NEED REFERENCE]

It is also worth to mention that we should hold back a small part of the original data set for testing, since we need to test the model on data that it has not seen before.

3.1.2 K-fold validation method

K-fold validation is a method that has more describing name than many of its fellow methods. The idea is to make the most use of our data by splitting it

into k folds and training k times on it. Every time we train, we leave out one of the folds, which gonna be our validation data. This validation data needs to be different every time, and we are therefore restricted to k unique training sessions. A typical overview looks like this:

- Split data set into k equally sized folds
- Use the $k-1$ first folds as training data, and leave the k'th fold for validation
- Use the $k-2$ first folds plus the k-th fold as training data, and leave the $(k-1)$'th fold for validation
- Continue until all folds are used as training data

[**Berkeley**]

3.2 Minimization methods

Suppose we have a very simple model trying to fit a straight line to data points. In that case, we could manually vary the coefficients and find a line that fits the points quite good. However, when the model gets more complicated, this can be a time consuming activity. Would it not be good 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. We will stick to gradient descent in this project, which might be the simplest method for our purposes.

3.2.1 Gradient Descent

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

$$\beta_i^{\text{new}} = \beta_i - \eta \cdot \frac{\partial Q(\beta_i)}{\partial \beta_i} \tag{29}$$

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

$$\frac{\partial Q(\beta_i)}{\partial \beta_i} < \varepsilon \tag{30}$$

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

Before we can implement equation (29), 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

$$Q(\vec{\beta}, \lambda, q) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q, \tag{31}$$

and its derivative with respect to β_k is

$$\frac{\partial Q(\vec{\beta}, \lambda, q)}{\partial \beta_k} = -2 \sum_{i}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right) x_{ik} + \frac{\beta_k}{|\beta_k|} q\lambda |\beta_k|^{q-1}. \tag{32}$$

The vectorized version looks like

$$\frac{\partial Q(\vec{\beta}, \lambda, q)}{\partial \vec{\beta}} = -2\hat{X}^T(\vec{y} - \hat{X}\vec{\beta}) + \frac{\vec{\beta}}{|\vec{\beta}|} q\lambda |\vec{\beta}|^{q-1} \tag{33}$$

where all operations are element wise. The algorithm of this minimization method is thus as follows:

```
while dbeta > epsilon:
        e = z - X.dot(beta)
    debeta = 2*X.T.dot(e) -np.sign(beta)*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
- Scikit Learn - Benchmark for self-produced code
- Scipy - For reading terrain data
- tqdm - For progression bar

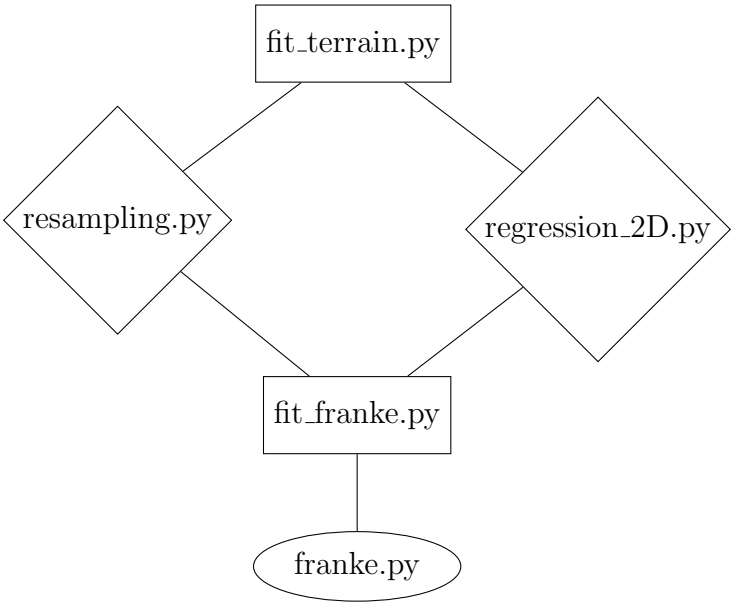


Figure 3: Code structure

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 (3).

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

$$\text{MSE}(\vec{\beta}) = \sum_i \left(y_i - \beta_0 - \sum_j X_{ij}\beta_j \right)^2, \tag{34}$$

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:

$$\text{MSE}(\vec{\beta}) = (\vec{y} - \hat{X}^T\vec{\beta})^T \cdot (\vec{y} - \hat{X}^T\vec{\beta}) \tag{35}$$

which is usually much faster.

5 Results

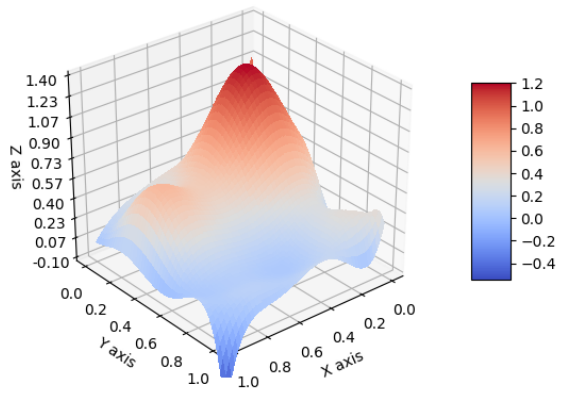
Here the results will be presented. We start looking at how good our linear regression is, by fitting a polynomial to points withdrawn from a known PDF. Thereafter we will see how good the regression is when using real terrain data. To give a intuition on how good the fit is, we will in both cases show 3D plots of the graphs using various methods, and then present the error analysis.

5.1 Franke function

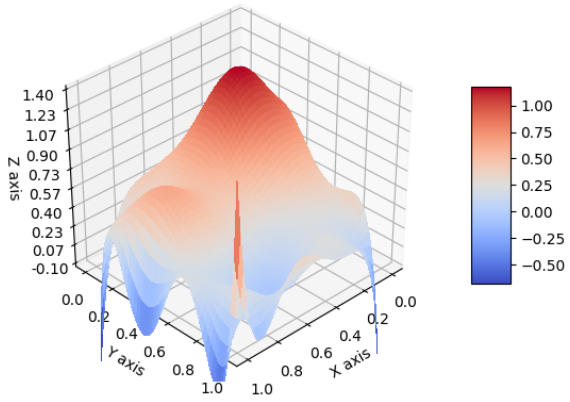
Below one can find the results of the regression on Franke function. All results in this section was obtained from the same data set, with an estimated variance of ...

5.1.1 Visualization of graphes

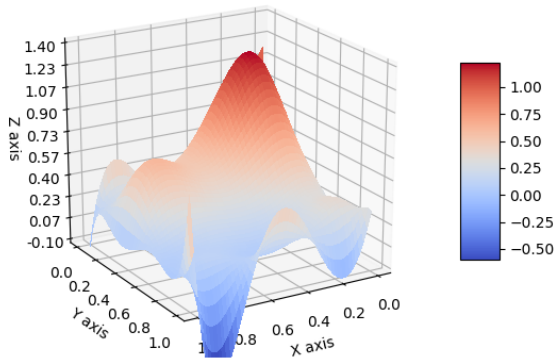
In figure (1) one can find a 3D visualization of the Franke function, for easier comparison with the fitted graphs presented in figure (6). The data sets were composed drawing random x's and y's in the interval $x, y \in [0, 1]$ from a uniform distribution, and the corresponding z-values where obtained from the Franke function. Further, we added noise produced by $\mathcal{N}(0, \sigma^2 = 0.1)$ and compared with the undisturbed data set. For Lasso and Ridge, we used a penalty $\lambda = 1e - 15$, and for Lasso we also used gradient descent for minimization. On that occasion, the learning rate was set to $\eta = 0.001$ with niter= 1000000 as the number of iterations.



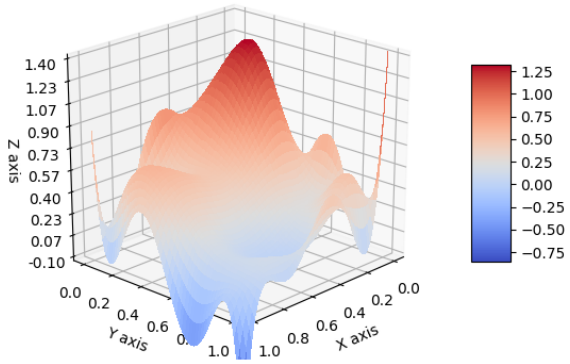
(a) OLS without noise



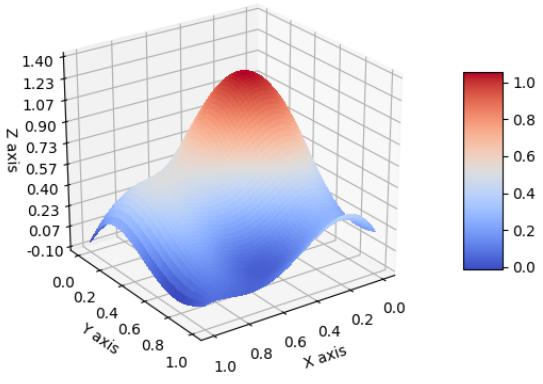
(b) OLS with noise



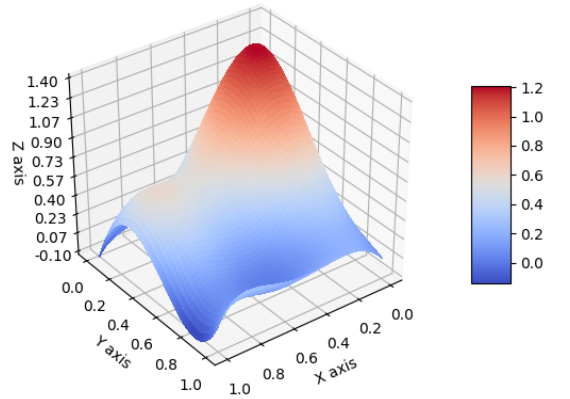
(c) Ridge without noise



(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 Ridge and Lasso, we used a low penalty of $\lambda = 1e - 15$. Lasso was performed with $\eta = 1e - 3$ and 1e6 iterations. The noise was sampled from a normal distribution with $\sigma = 0.1$.

5.1.2 Error

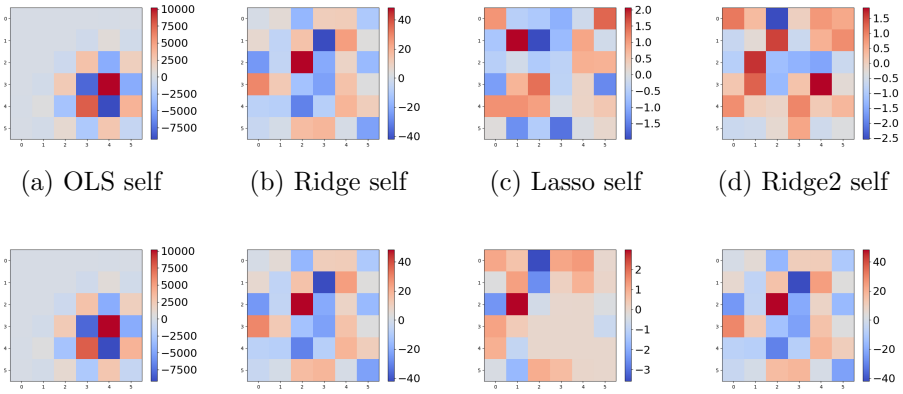
To be able to determine which method is best, we need to analyze the error. In table (3), the MSE and R^2-score are given for OLS, Ridge, Lasso and Ridge with gradient descent.

Table 1: Mean Square Error and R^2-score presented for OLS, Ridge, Lasso and Ridge + gradient descent.

	MSE		**R2**	
	self	scikit	self	scikit
OLS	0.07853	0.86104	0.05386	1.7483
Ridge	0.07852	0.86410	0.05406	1.6904
Lasso	0.07704	0.86231	0.07187	1.6492
RidgeGD	0.07523	0.91850	0.09362	1.3217

5.1.3 Coefficients β

It can also be interesting to see how the coefficients actually differ between the self-built functions and the Scikit Learn function. In figure (7), the beta values are visualized such that large numbers got strong color, negative numbers are blue and positive numbers are red. The methods OLS, Ridge, Lasso and Ridge produced using gradient descent are presented.



(a) OLS self (b) Ridge self (c) Lasso self (d) Ridge2 self

(e) OLS scikit (f) Ridge scikit (g) Lasso scikit (h) Ridge scikit

Figure 5: Beta values visualized for different methods. The upper images are gotten from the self-built functions, where the last one is Ridge found by minimizing beta using gradient descent. The lower images are the benchmarks which are produced by Scikit learn.

Furthermore, the confidence intervals are given in table (4) for all betas and all methods.

Table 2: Write table text here.

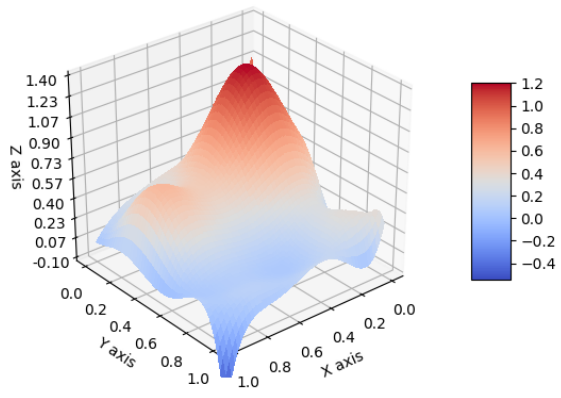
	OLS	**Ridge**	**Lasso**	**Ridge2**
β_1	0.07853	0.86104	0.05386	1.7483
β_2	0.07853	0.86104	0.05386	1.7483
β_3	0.07853	0.86104	0.05386	1.7483
β_4	0.07853	0.86104	0.05386	1.7483
β_5	0.07853	0.86104	0.05386	1.7483
β_6	0.07853	0.86104	0.05386	1.7483
β_7	0.07853	0.86104	0.05386	1.7483
β_8	0.07853	0.86104	0.05386	1.7483
β_9	0.07853	0.86104	0.05386	1.7483
β_{10}	0.07853	0.86104	0.05386	1.7483
β_{11}	0.07853	0.86104	0.05386	1.7483
β_{12}	0.07853	0.86104	0.05386	1.7483
β_{13}	0.07853	0.86104	0.05386	1.7483
β_{14}	0.07853	0.86104	0.05386	1.7483
β_{15}	0.07853	0.86104	0.05386	1.7483
β_{16}	0.07853	0.86104	0.05386	1.7483
β_{17}	0.07853	0.86104	0.05386	1.7483
β_{18}	0.07853	0.86104	0.05386	1.7483
β_{19}	0.07853	0.86104	0.05386	1.7483
β_{20}	0.07853	0.86104	0.05386	1.7483
β_{21}	0.07853	0.86104	0.05386	1.7483
β_{22}	0.07853	0.86104	0.05386	1.7483
β_{23}	0.07853	0.86104	0.05386	1.7483
β_{24}	0.07853	0.86104	0.05386	1.7483
β_{25}	0.07853	0.86104	0.05386	1.7483
β_{26}	0.07853	0.86104	0.05386	1.7483
β_{27}	0.07853	0.86104	0.05386	1.7483
β_{28}	0.07853	0.86104	0.05386	1.7483
β_{29}	0.07853	0.86104	0.05386	1.7483
β_{30}	0.07853	0.86104	0.05386	1.7483
β_{31}	0.07853	0.86104	0.05386	1.7483
β_{32}	0.07853	0.86104	0.05386	1.7483
β_{33}	0.07853	0.86104	0.05386	1.7483
β_{34}	0.07853	0.86104	0.05386	1.7483
β_{35}	0.07853	0.86104	0.05386	1.7483
β_{36}	0.07853	0.86104	0.05386	1.7483

5.2 Real data

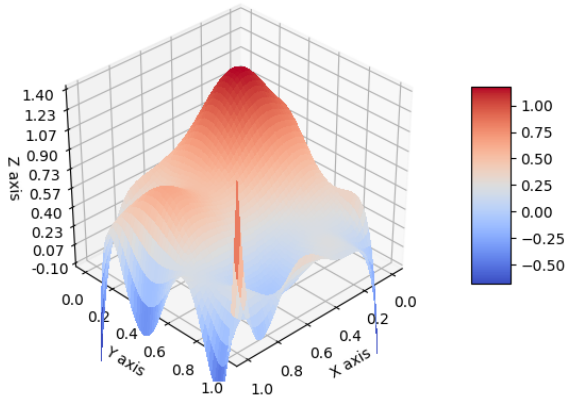
Below one can find the results of the regression on Franke function. All results in this section was obtained from the same data set, with an estimated variance of ...

5.2.1 Visualization of graphes

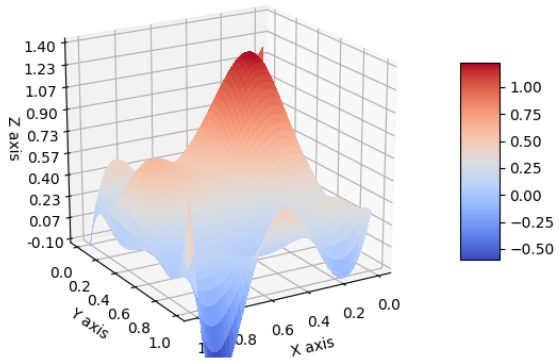
In figure (**??**) one can find a 3D visualization of the Franke function, for easier comparison with the fitted graphs presented in figure (6). The data sets were composed drawing random x's and y's in the interval $x, y \in [0, 1]$ from a uniform distribution, and the corresponding z-values where obtained from the Franke function. Further, we added noise produced by $\mathcal{N}(0, \sigma^2 = 0.1)$ and compared with the undisturbed data set. For Lasso and Ridge, we used a penalty $\lambda = 1e - 15$, and for Lasso we also used gradient descent for minimization. On that occasion, the learning rate was set to $\eta = 0.001$ with niter= 1000000 as the number of iterations.



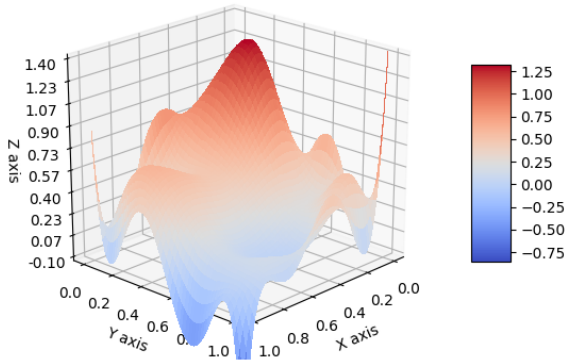
(a) OLS without noise



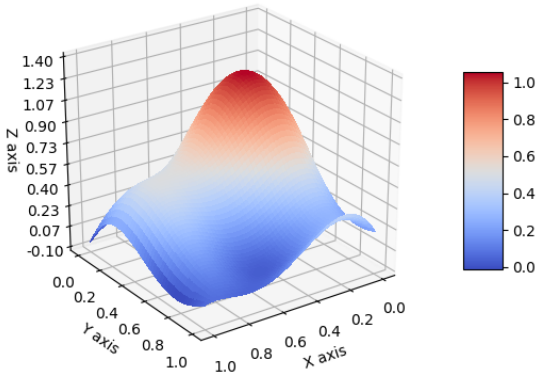
(b) OLS with noise



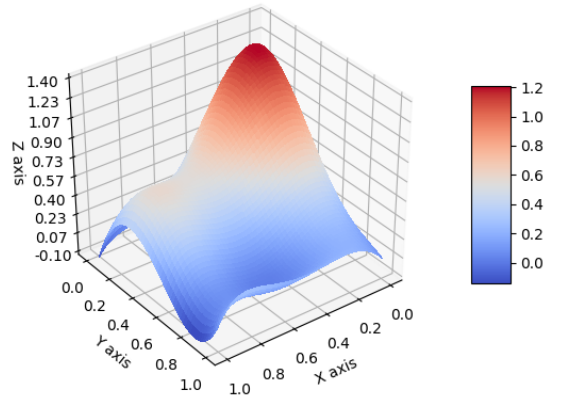
(c) Ridge without noise



(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 Ridge and Lasso, we used a low penalty of $\lambda = 1e - 15$. Lasso was performed with $\eta = 1e - 3$ and 1e6 iterations. The noise was sampled from a normal distribution with $\sigma = 0.1$.

5.2.2 Error

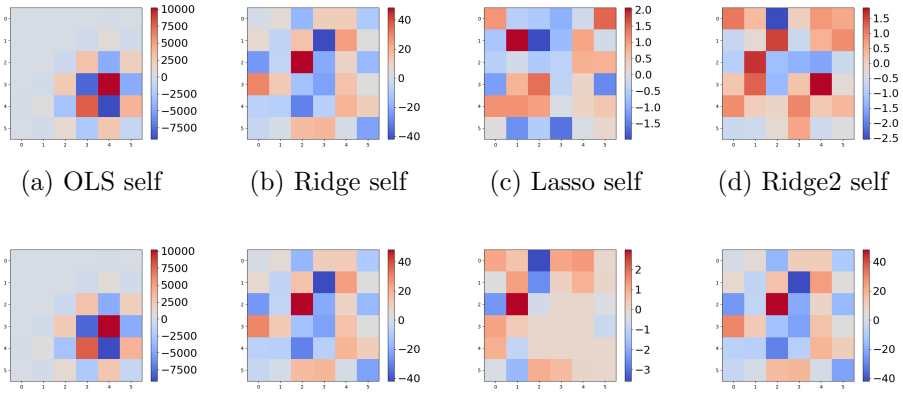
To be able to determine which method is best, we need to analyze the error. In table (3), the MSE and R^2-score are given for OLS, Ridge, Lasso and Ridge with gradient descent.

Table 3: Mean Square Error and R^2-score presented for OLS, Ridge, Lasso and Ridge + gradient descent.

	MSE		**R2**	
	self	scikit	self	scikit
OLS	0.07853	0.86104	0.05386	1.7483
Ridge	0.07852	0.86410	0.05406	1.6904
Lasso	0.07704	0.86231	0.07187	1.6492
RidgeGD	0.07523	0.91850	0.09362	1.3217

5.2.3 Coefficients β

It can also be interesting to see how the coefficients actually differ between the self-built functions and the Scikit Learn function. In figure (7), the beta values are visualized such that large numbers got strong color, negative numbers are blue and positive numbers are red. The methods OLS, Ridge, Lasso and Ridge produced using gradient descent are presented.



(a) OLS self (b) Ridge self (c) Lasso self (d) Ridge2 self

(e) OLS scikit (f) Ridge scikit (g) Lasso scikit (h) Ridge scikit

Figure 7: Beta values visualized for different methods. The upper images are gotten from the self-built functions, where the last one is Ridge found by minimizing beta using gradient descent. The lower images are the benchmarks which are produced by Scikit learn.

Furthermore, the confidence intervals are given in table (4) for all betas and all methods.

Table 4: Write table text here.

	OLS	**Ridge**	**Lasso**	**Ridge2**
β_1	0.07853	0.86104	0.05386	1.7483
β_2	0.07853	0.86104	0.05386	1.7483
β_3	0.07853	0.86104	0.05386	1.7483
β_4	0.07853	0.86104	0.05386	1.7483
β_5	0.07853	0.86104	0.05386	1.7483
β_6	0.07853	0.86104	0.05386	1.7483
β_7	0.07853	0.86104	0.05386	1.7483
β_8	0.07853	0.86104	0.05386	1.7483
β_9	0.07853	0.86104	0.05386	1.7483
β_{10}	0.07853	0.86104	0.05386	1.7483
β_{11}	0.07853	0.86104	0.05386	1.7483
β_{12}	0.07853	0.86104	0.05386	1.7483
β_{13}	0.07853	0.86104	0.05386	1.7483
β_{14}	0.07853	0.86104	0.05386	1.7483
β_{15}	0.07853	0.86104	0.05386	1.7483
β_{16}	0.07853	0.86104	0.05386	1.7483
β_{17}	0.07853	0.86104	0.05386	1.7483
β_{18}	0.07853	0.86104	0.05386	1.7483
β_{19}	0.07853	0.86104	0.05386	1.7483
β_{20}	0.07853	0.86104	0.05386	1.7483
β_{21}	0.07853	0.86104	0.05386	1.7483
β_{22}	0.07853	0.86104	0.05386	1.7483
β_{23}	0.07853	0.86104	0.05386	1.7483
β_{24}	0.07853	0.86104	0.05386	1.7483
β_{25}	0.07853	0.86104	0.05386	1.7483
β_{26}	0.07853	0.86104	0.05386	1.7483
β_{27}	0.07853	0.86104	0.05386	1.7483
β_{28}	0.07853	0.86104	0.05386	1.7483
β_{29}	0.07853	0.86104	0.05386	1.7483
β_{30}	0.07853	0.86104	0.05386	1.7483
β_{31}	0.07853	0.86104	0.05386	1.7483
β_{32}	0.07853	0.86104	0.05386	1.7483
β_{33}	0.07853	0.86104	0.05386	1.7483
β_{34}	0.07853	0.86104	0.05386	1.7483
β_{35}	0.07853	0.86104	0.05386	1.7483
β_{36}	0.07853	0.86104	0.05386	1.7483

6 Discussion

Why do we observe what we observe

7 Conclusion

Which model fits the data best?

Mention that I know I should've studied a polynomial of total order of 5.