FYS-STK4155 - Applied data analysis and machine learning

Project 1

Even M. Nordhagen

October 8, 2018

- Github repository containing programs and results:

  `https://github.com/evenmn/FYS-STK4155`

Abstract

The aim of this project is to study the performance of linear regression in order to fit a two dimensional polynomial to terrain data. Both Ordinary Least Square (OLS), Ridge and Lasso regression methods were implemented, and for minimizing Lasso's cost function Gradient Descent (GD) was used. A fourth method was to minimize the cost function of Ridge using GD. The fitted polynomial was visualized and compared with the data, the Mean Square Error (MSE) and R^2-score were analyzed, and finally the polynomial coefficients were studied applying visualization tools and Confidence Intervals (CI). To benchmark the results, we used Scikit Learn.

We found the self-implemented OLS and Ridge regression functions to reproduce the benchmarks, and Lasso was close to reproducing the benchmark as well. However, the difference between results produced by standard Ridge regression and when minimizing its cost function is large. The OLS regression method is considered as the most successful due to its small MSE and high R^2-score.

Contents

1 Introduction

The linear regression methods were first introduced for more than two centuries ago, and have been used in a large number of fields throughout the years [1][2]. In this project we will investigate whether the methods are sufficient for fitting polynomials to real terrain data, or we need more complicated methods. To challenge the methods, 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 K-fold validation method to estimate the actual error.

For the results, see section *Results* (5), which again is discussed in section *Discussion* (6). 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 (7) with the same name.

2 Theory

Regression analysis are widely used in different fields of natural sciences, data science and economics to mention some. In fact, as fast as we a data set that needs to be analyzed, regression is unavoidable. 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, and for a dependent variable this can be written as

$$y_i = \sum_j X_{ij}\beta_j. \tag{1}$$

In principle, x_ij can be an arbitrary function of the arguments x_i, but in this project we will only fit the data with a polynomial function. [3]

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

$$\vec{y} = \hat{X}^T \vec{\beta} + \vec{\varepsilon}, \tag{2}$$

with

$$\hat{X} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \ldots & x_1^p \\ x_2^0 & x_2^1 & x_2^2 & \ldots & x_2^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & x_n^2 & \ldots & x_n^p \end{pmatrix} \tag{3}$$

as the *design* matrix and

$$\vec{\beta} = (\beta_0, \beta_1, \ldots, \beta_p) \tag{4}$$

as the coefficients.

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

$$\vec{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T \vec{y}, \tag{5}$$

which again corresponds to minimizing the cost function,

$$\vec{\beta} = \text{argmin}, \vec{\beta} \Big\{ \sum_{i=1}^{n} \Big(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \Big)^2 \Big\}. \tag{6}$$

where the standard cost function is used, given by least squares,

$$Q(\vec{\beta}) = \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2 = (\vec{y} - \hat{X}\vec{\beta})^T (\vec{y} - \hat{X}\vec{\beta}). \tag{7}$$

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. For this case, we expect the coefficients to be smaller compared with OLS, since the cost function is directly dependent on $\vec{\beta}^2$.

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

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

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

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

We expect the coefficients to be even smaller for Lasso regression than for Ridge regression, since the cost function is directly dependent on $\vec{\beta}$.

2.1.4 General form

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

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

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 Multivariate linear regression

The same approach as above can be used 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 an 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 (2), 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

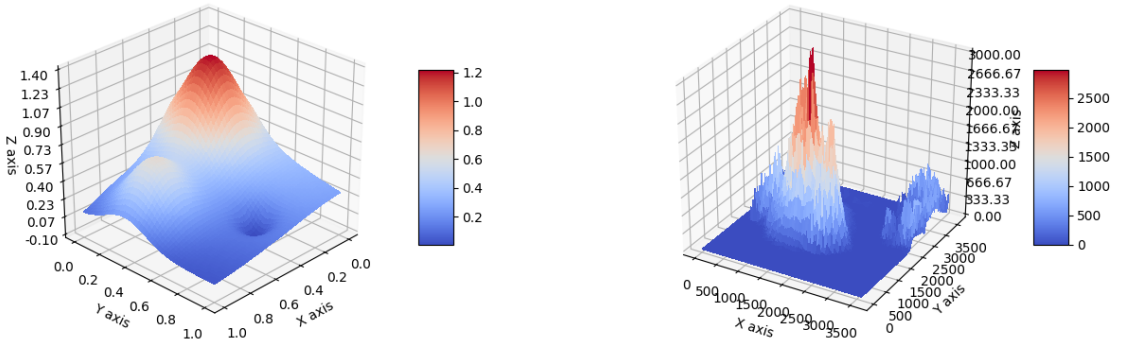
$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right), \end{aligned}$$

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

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) [4], 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.



(a) Franke function

(b) Lombok terrain

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

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 a polynomial of degree $p \in \mathbb{R}^d$ to the points, where we assume that the fitting polynomial has the same degree in all directions, 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

$$
\hat{X} = \begin{pmatrix}
(x_1^1)^0 (x_1^2)^0 \ldots (x_1^d)^0 & \ldots & (x_1^1)^1 (x_1^2)^0 \ldots (x_1^d)^0 & \ldots & (x_1^1)^p (x_1^2)^p \ldots (x_1^d)^p \\
(x_2^1)^0 (x_2^2)^0 \ldots (x_2^d)^0 & \ldots & (x_2^1)^1 (x_2^2)^0 \ldots (x_2^d)^0 & \ldots & (x_2^1)^p (x_2^2)^p \ldots (x_2^d)^p \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
(x_n^1)^0 (x_n^2)^0 \ldots (x_n^d)^0 & \ldots & (x_n^1)^1 (x_n^2)^0 \ldots (x_n^d)^0 & \ldots & (x_n^1)^p (x_n^2)^p \ldots (x_n^d)^p,
\end{pmatrix} \tag{18}
$$

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

$$\sigma_y^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2 \tag{19}$$

where the sample mean is given by

$$\bar{y} = \sum_{i=1}^{N} y_i. \tag{20}$$

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*. [5]

Often we study 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

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

which for M data sets gives a total sample mean of

$$\bar{y}_M = \frac{1}{M}\sum_{\alpha=1}^{M}\bar{y}_\alpha = \frac{1}{MN}\sum_{\alpha=1}^{M}\sum_{i=1}^{N} y_{\alpha,i}. \tag{22}$$

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

$$\sigma_M^2 = \frac{1}{M}\sum_{\alpha=1}^{M}\frac{\sigma_\alpha^2}{N} + \frac{2}{MN^2}\sum_{\alpha=1}^{M}\sum_{i<j}^{N}(y_{\alpha,i} - \bar{y}_M)(y_{\alpha,j} - \bar{y}_M). \tag{23}$$

The last term is called the covariance, and is a measure of how much the data sets are related. If they are totally uncorrelated, 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

$$\text{CI} = \bar{y} \pm z^* \frac{\sigma}{\sqrt{N}} \tag{24}$$

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, suppose 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, which for this case is $N/2$.
2. Find z^* fromhttps://www.sumproduct.com/thought/simulation-stimulation 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 approximates a binomial distribution when N increases, with a known σ. [6]
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 get their variance from the analytical formula

$$\text{Var}(\vec{\beta}) = (\hat{X}^T \hat{X})^{-1} \sigma^2 \tag{25}$$

where the σ^2 is the sample variance of the points. The variance matrix will have the variance of each β_j on the diagonal, while the off-diagonal elements are associated with the covariance.

Another approach 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

To find out how good the fit is, we could potentially compare the polynomial plot to the data plot and decide whether the fit is good or not. However, that approach is risky in the sense that it is hard to determine how good a fit really is, and we therefore rely on error estimates. There are many ways to calculate the error, where some frequently used methods are

- the absolute error
- the relative error
- the mean square error (MSE)
- the 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. [3] We study the MSE in order to compute the reduction of the cost function, because it is basically the standard cost function, used in OLS.

$$\text{MSE}(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}(\vec{\beta}))^2 \tag{26}$$

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,

$$\text{bias}(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}(\vec{\beta})). \tag{27}$$

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. [7] It can be shown that the MSE is a sum of the bias squared and the variance,

$$\text{MSE}(\vec{\beta}) = \text{bias}(\vec{\beta})^2 + \sigma^2, \tag{28}$$

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). From the figure, we observe that the bias decreases and the variance increases as the model gets more complex. The MSE is minimized when $\sigma^2 = \text{bias}$.

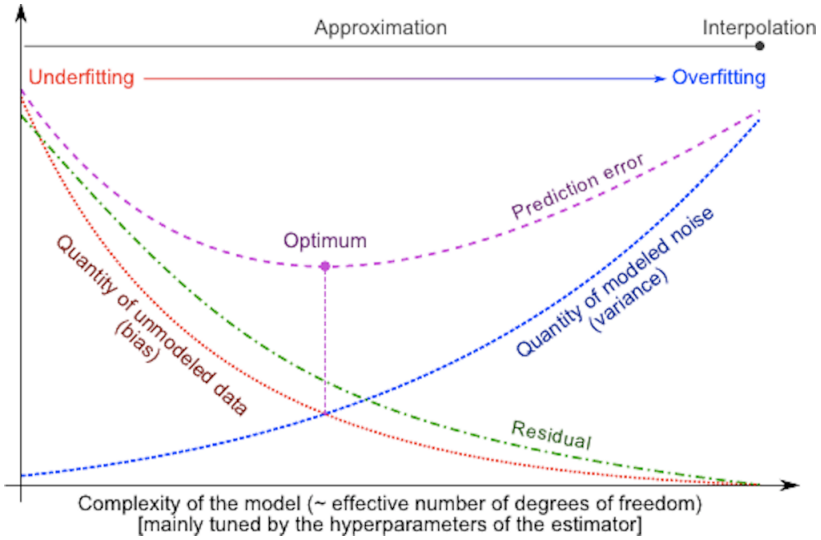


Figure 2: Bias-variance trade-off. Figure showing the bias (red graph). variance (blue graph) and MSE (purple graph) as functions of the complexity of the model. Figure taken from `https://www.reddit.com/r/mlclass/comments/mmlfu/a_nice_alternative_explanation_of_bias_and/`

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. [8]

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

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

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 usually between 0 and 1. If the model does not explain any of the variations it is 0 and if it explains all variations the score is 1. In that manner, we should fight for a high R2-score. In entirety it reads

$$R^2(\vec{y}, \tilde{\vec{y}}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \tilde{y})^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}. \tag{29}$$

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 sum 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 bootstrap 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 $\vec{\beta}^*$ should correspond to the sample variance

[9]

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. [10]

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

[7]. An example implementation of K-fold validation, and in fact the function used in this project, can be seen below.

```
def k_fold(x, y, z, K=10):
  xMat = np.reshape(x, (K, int(len(x)/K)))      # Reshape x-coords
  yMat = np.reshape(y, (K, int(len(y)/K)))      # Reshape y-coords
  zMat = np.reshape(z, (K, int(len(z)/K)))      # Reshape z-coords

  MSE_train = 0                                 # Declare MSE_train variable
  MSE_test  = 0                                 # Declare MSE test variable
  R2_train  = 0                                 # Declare R2 train variable
  R2_test   = 0                                 # Declare R2 test variable
```

```
    for i in range(K):
        xMatNew = np.delete(xMat, i, 0)             # Ignore test sets
        yMatNew = np.delete(yMat, i, 0)             # Ignore test sets
        zMatNew = np.delete(zMat, i, 0)             # Ignore test sets

        xVecNew = xMatNew.flatten()                 # Reshape training sets
        yVecNew = yMatNew.flatten()                 # into vectors
        zVecNew = zMatNew.flatten()

        order5 = Reg_2D(xVecNew, yVecNew, zVecNew, Px=5, Py=5)
        beta_train = order5.ols()

        MSE_train += MSE(xVecNew, yVecNew, zVecNew, beta_train)
        MSE_test += MSE(xMat[i], yMat[i], zMat[i], beta_train)

        R2_train += R2(xVecNew, yVecNew, zVecNew, beta_train)
        R2_test += R2(xMat[i], yMat[i], zMat[i], beta_train)

    return MSE_train/K, MSE_test/K, R2_train/K, R2_test/K
```

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

$$\beta_i^{\text{new}} = \beta_i - \eta \cdot \frac{\partial Q(\beta_i)}{\partial \beta_i} \tag{30}$$

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

$$Q(\beta_i) < \varepsilon \tag{31}$$

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

Before we can implement equation (30), 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

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

and its derivative with respect to β_k is

$$\frac{\partial Q(\vec{\beta}, \lambda, q)}{\partial \beta_k} = -2 \sum_{i}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right) x_{ik} + \frac{\beta_k}{|\beta_k|} q\lambda |\beta_k|^{q-1}. \tag{33}$$

The vectorized version looks like

$$\frac{\partial Q(\vec{\beta}, \lambda, q)}{\partial \vec{\beta}} = -2\hat{X}^T(\vec{y} - \hat{X}\vec{\beta}) + \frac{\vec{\beta}}{|\vec{\beta}|} q\lambda |\vec{\beta}|^{q-1} \tag{34}$$

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
```

3.3 Benchmark

To benchmark the results in this project, we use Scikit Learn, which is a package for machine learning in Python. [11] The built-in regression tools are made to give high performance and be easy to implement, and is therefore the ideal benchmark in this project. The procedure of calling the OLS, Ridge and Lasso functions looks like this:

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso

reg = LinearRegression()
reg.fit (X, z)
beta_OLS = reg.coef_

reg = Ridge(alpha=lambda)
reg.fit(X, z)
beta_Ridge = reg.coef_

reg = Lasso(alpha=lambda)
reg.fit(X, z)
beta_Lasso = reg.coef_
```

where X is the design matrix and z is the dependent variables. See *regression_scikit.py* for actual implementation.

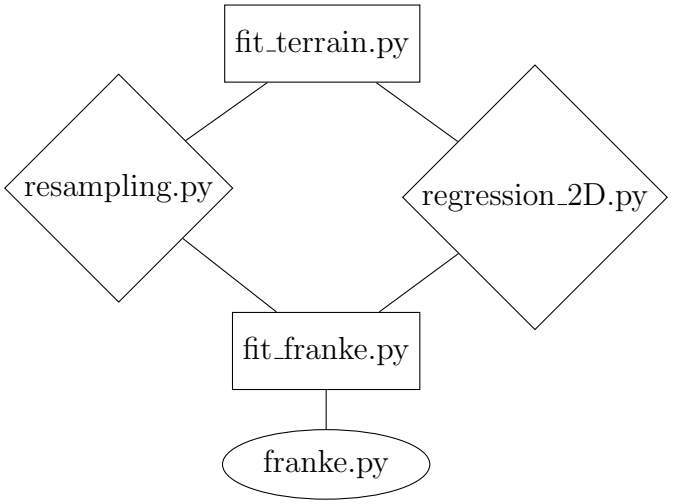


Figure 3: Code structure

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

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 the bottlenecks, so by replacing loops with vector operations we can save a lot of time. An example on this, is when we calculate the MSE

$$\text{MSE}(\vec{\beta}) = \sum_i \left(y_i - \beta_0 - \sum_j X_{ij}\beta_j \right)^2, \tag{35}$$

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

$$\text{MSE}(\vec{\beta}) = (\vec{y} - \hat{X}^T\vec{\beta})^T \cdot (\vec{y} - \hat{X}^T\vec{\beta}) \tag{36}$$

which is usually much faster.

5 Results

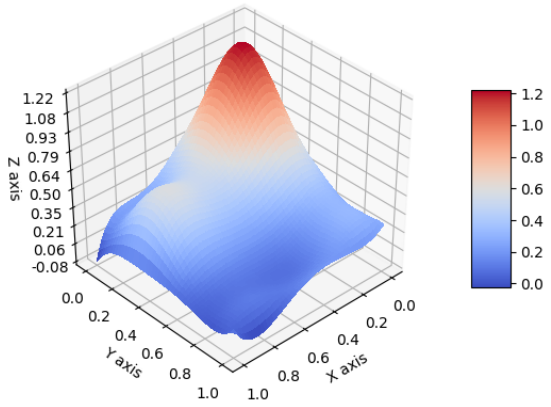
Here the results will be presented. We start looking at how good our linear regression is, by fitting a polynomial to points withdrawn from a known PDF. Thereafter we will see how good the regression is when using real terrain data. To give an intuition on how good the fit is, we will in both cases show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

5.1 Franke function

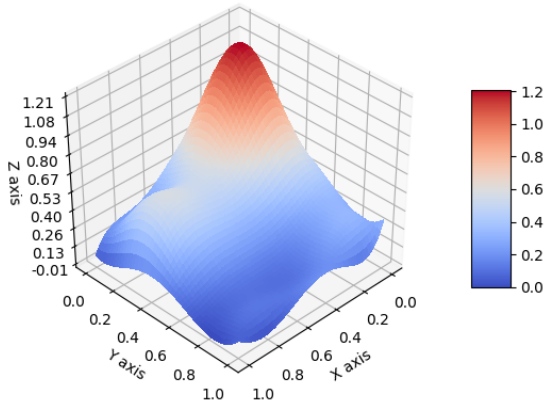
Below one can find the results when fitting the Franke function. All results in this section were obtained from the same data set, consisting of 1000 points, withdrawn from a uniform distribution ($x, y \in [0, 1]$). Further, the polynomial degrees in x- and y-directions were set to $P_x = P_y = 5$. For the methods that require minimization, we used gradient descent with the parameters $\lambda = 1e-5$ (penalty), $\eta = 1e-4$ (learning rate) and niter = $1e5$ (number of iterations) when nothing else is specified. To benchmark the results (mainly to verify the code), Scikit Learn was used. In the first place, the studies were done without noise, and then everything was repeated with adding a noise of $\mathcal{N}(0, \sigma^2 = 0.1)$.

5.1.1 Visualization of graphes

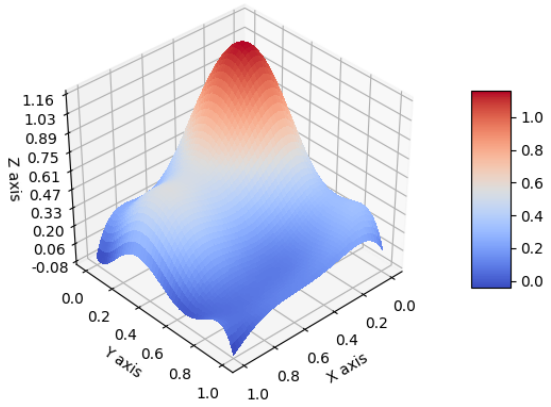
In figure (4) the polynomial produced by regression on noisy data is compared with the polynomial produced by regression on smooth data.



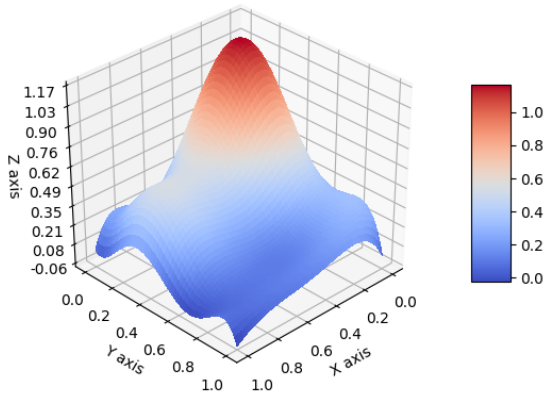
(a) OLS without noise



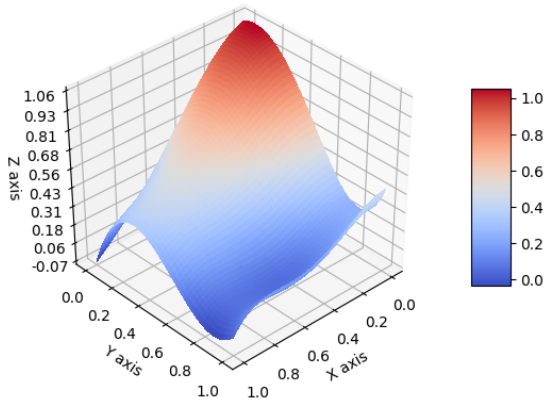
(b) OLS with noise



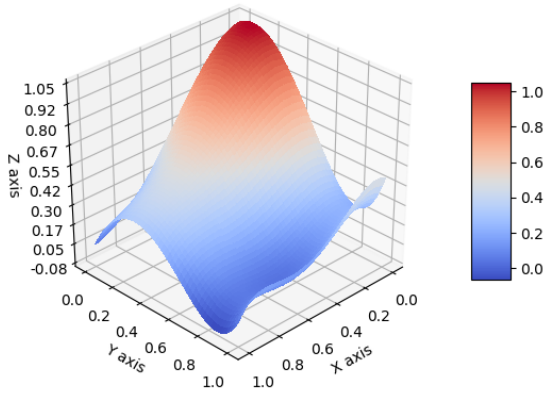
(c) Ridge without noise



(d) Ridge with noise



(e) Lasso without noise



(f) Lasso with noise

Figure 4: Fitted polynomial produced by OLS, Ridge and Lasso with (right hand side (RHS)) and without noise (left hand side (LHS)). For Ridge and Lasso, we used a low penalty of $\lambda = 1e-5$. Lasso was performed with $\eta = 1e-4$ and niter $= 1e5$ iterations. The noise was sampled from a normal distribution with $\sigma^2 = 0.1$.

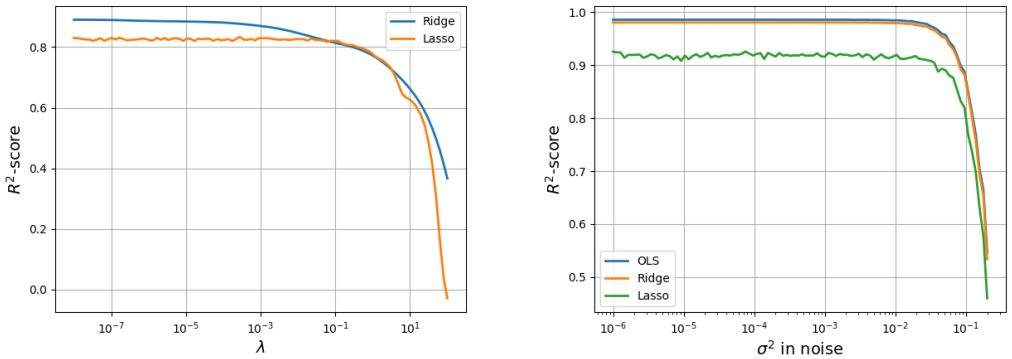
5.1.2 Error

To determine which method is best, we need to analyze the error. In table (1), the MSE and R^2-score are given for OLS, Ridge, Lasso and Ridge with gradient descent (RidgeGD). The errors were evaluated by the self-implemented functions directly (Self), by K-fold resampling test data (K-fold) and by Scikit Learn (Scikit).

Table 1: Mean Square Error and R^2-score presented for OLS, Ridge, Lasso and Ridge + gradient descent (RidgeGD), where noise was added to the data. The parameters used were $\lambda = 1e-5$ (penalty), $\eta = 1e-4$ (learning rate), niter $= 1e5$ (number of iterations) and $\mathcal{N}(0, \sigma^2 = 0.1)$ (noise). See text for more information.

	MSE			**R2**		
	Self	K-fold	Scikit	Self	K-fold	Scikit
OLS	0.008494	0.009119	0.008494	0.9048	0.8956	0.9048
Ridge	0.009128	0.009651	0.009128	0.8977	0.8895	0.8977
Lasso	0.01439	0.01489	0.01555	0.8387	0.8296	0.8257
RidgeGD	0.01451	0.01504	0.009128	0.8373	0.8280	0.8977

We have also studied how the R^2 score is dependent on various parameters. In figure (5), the R^2-score is plotted as a function of the penalty for Ridge and Lasso (a), and as a function of the noise for OLS, Ridge and Lasso (b).



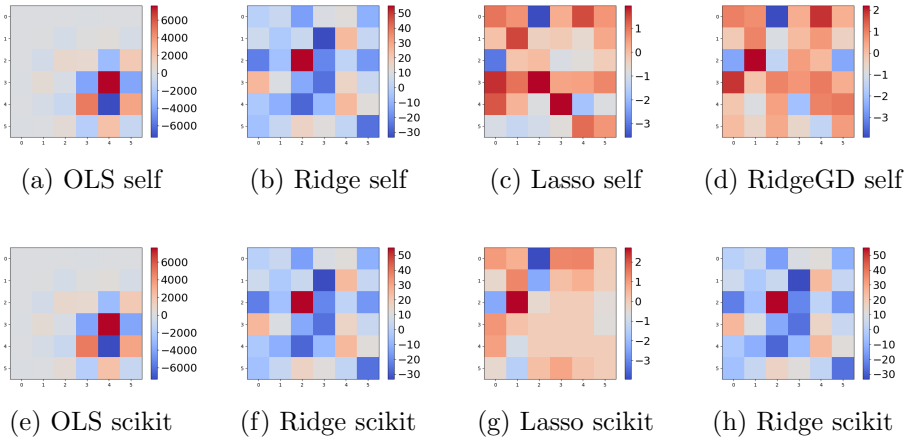
(a) Lambda vs. R2-score

(b) Variance vs. R2-score

Figure 5: R^2-score plotted as a function of the penalty λ (a) and as a function of the noise (b). $\lambda \in [10^{-8}, 10^2]$ in (a) and $\sigma^2 \in [10^{-6}, 10^{-0.7}]$ in (b). The other parameters used were $\lambda = 1e{-}5$ (penalty, was held constant for (b) only), $\eta = 1e{-}4$ (learning rate), $niter = 1e5$ (number of iterations) and $\mathcal{N}(0, \sigma^2 = 0.1)$ (noise, was held constant for (a) only).

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 functions. In figure (6), 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 (RidgeGD) are presented.



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

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

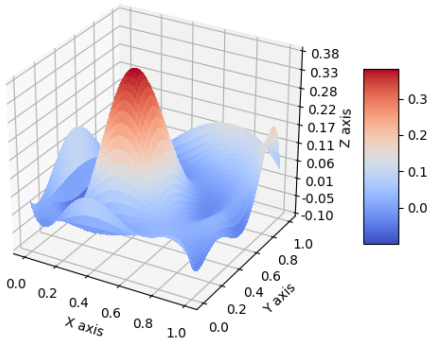
Figure 6: Beta values visualized for various methods. The upper images were produced by the self-built functions, while the lower images are the benchmarks which were produced by Scikit learn. The parameters used were $\lambda = 1e-5$ (penalty), $\eta = 1e-4$ (learning rate), niter $= 1e5$ (number of iterations) and $\mathcal{N}(0, \sigma^2 = 0.1)$ (noise).

5.2 Real data

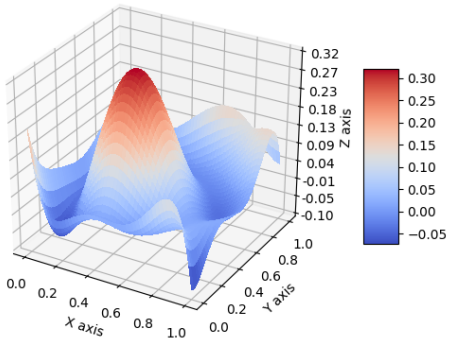
Below one can find the results when fitting the terrain data with a polynomial of a maximum degree of 5 in x- and y-directions. For the methods that require minimization, we used gradient descent with the parameters $\lambda = 1e-5$ (penalty), $\eta = 1e-4$ (learning rate) and niter $= 1e5$ (number of iterations) when nothing else is specified. For benchmarking the results (mainly to verify the code), Scikit Learn was used.

5.2.1 Visualization of graphes

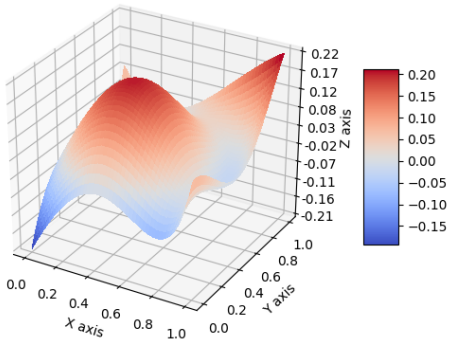
In figure (7), we can see how good we were able to fit the polynomials using OLS, Ridge and Lasso methods.



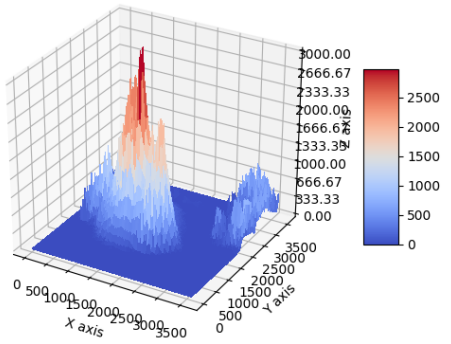
(a) OLS



(b) Ridge



(c) Lasso



(d) Terrain data

Figure 7: Fitted polynomial by OLS (upper left), Ridge (upper right) and Lasso (lower left). In addition the real terrain data was added (lower right). For Ridge and Lasso, we used a low penalty of $\lambda = 1e-5$. Lasso was performed with $\eta = 1e-4$ and niter $= 1e5$ iterations. The terrain data used in regression was normalized, but the data used in the terrain plot was not.

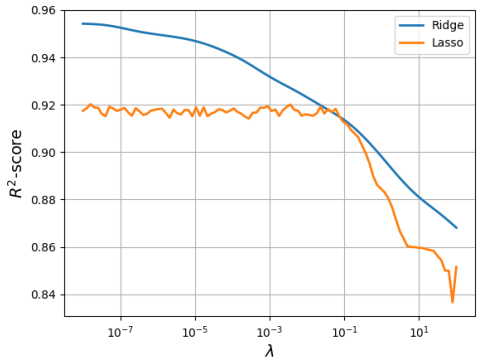


Figure 8: R^2-score plotted as a function of the penalty λ. The penalty was varied in the interval $\lambda \in [10^{-8}, 10^2]$, and the other parameters used were $\eta = 1e-4$ (learning rate) and niter=1e5 (number of iterations).

5.2.2 Error

To determine which method is best, we need to analyze the error. In table (1), the MSE and R^2-score are given for OLS, Ridge, Lasso and Ridge with gradient descent (RidgeGD). The errors were evaluated by the self-implemented functions directly (Self), by K-fold resampling test data (K-fold) and by Scikit Learn (Scikit).

Table 2: Mean Square Error and R^2-score presented for OLS, Ridge, Lasso and Ridge + gradient descent (RidgeGD), where noise was added to the data. The parameters used were $\lambda = 1e-5$ (penalty), $\eta = 1e-4$ (learning rate) and niter = $1e5$ (number of iterations). See text for more information.

	MSE			**R2**		
	Self	K-fold	Scikit	Self	K-fold	Scikit
OLS	0.003665	0.003834	0.003665	0.9550	0.9520	0.9550
Ridge	0.004322	0.004512	0.004322	0.9469	0.9436	0.9469
Lasso	0.006960	0.006893	0.006758	0.9145	0.9138	0.9170
RidgeGD	0.006764	0.006884	0.004322	0.9170	0.9140	0.9469

We have also studied how the R^2 score is dependent on the penalty λ. In figure (8), the R^2-score is plotted as a function of the penalty for both Ridge and Lasso.

6 Discussion

The first thing we can see from the results, is that the fitted functions in figure (4) look very similar with and without noise. This is probably caused by too small noise. However, we observe that both the OLS regression and Ridge are really good fits compared to the actual function in figure (1). Lasso is also similar, but with the naked eye we can see that it is obviously not as good as OLS and Ridge.

This is also reflected in the error analysis, where the MSE is way larger for Lasso compared to OLS and Ridge. OLS seems to have a slightly lower MSE than Ridge again, but for Ridge with gradient descent the MSE is even higher than for Lasso. From this, it may seems like the the gradient descent method does not work properly, because the cost function is obviously not fully minimized. When it comes to the R^2-score, we observe that also here OLS got the best score barely in front of Ridge, with RidgeGD and Lasso far behind. An interesting aspect is that the self-implemented Lasso function gives better results than the Scikit Learn Lasso function.

We expect the MSE produced by K-fold validation method on test data to be larger than the true MSE, and the R^2 score to be lower. We can see that this is

the case here, and it is a good thing that it does not differ too much. A large difference indicated a poor training session.

Furthermore, from the R^2 vs. λ plot in figure (5), we can see that Ridge mostly got a better R^2-score than Lasso, only beaten around $\lambda = 1$. What surprises me, is that the methods do not get more similar when the penalty decreases, since they should both approach OLS when the penalty gets smaller. In the second plot in the same figure, we again find Lasso to give the lowest R^2-score.

In figure (6), we can see that the beta values produced by Scikit's OLS function and the self-built OLS-function are more or less identical. The same applies to Ridge, so we can conclude that those two methods are implemented correctly (also based on the error analysis). The situation is unlike for the gradient descent methods, where the coefficients differ. Especially for RidgeGD, the difference is significantly both in sign and magnitude. For Lasso, we can reveal some similarities, and at least the magnitudes are similar. Again it is tempting to conclude that the minimization is wrong. Something I find strange, is that every second coefficient is negative and every second is positive for OLS, and obtain a cool pattern which I cannot explain.

We have earlier stated that we expect the coefficients in Ridge regression to be smaller than the coefficients in OLS regression, and the coefficients in Lasso regression should be even smaller than Ridge again. This is exactly what we see in figure (6).

Now over to the real terrain data. In figure (7), we can see that OLS reproduces the sharp peak of the volcano best among the methods, followed by Ridge and finally Lasso. This is expected from regression on the Franke function. Another thing we observe, is that none of the models is able to reproduce the flat ocean around the volcano. This is because we have limited the models to 5 degrees in each direction, and that is not enough degrees to fit those areas.

Again we observe that the self-built OLS function and Scikit Learn's OLS function give identical MSE and R^2-scores, and exactly the same applies for Ridge. When it comes to Lasso, the self-built function again gives a lower MSE and higher R^2-score than the function of Scikit learn, which is a bit surprising. Ridge produced from gradient descent is still weak compared to all the other methods. The K-fold validation MSE is in fact lower than the true MSE, but the difference is small, so the training session seems to have been performed correctly.

When it comes to the how the R^2-score is varying as a function of the penalty, we can see that also here the R^2-score from Lasso crosses the R^2-score from Ridge. This happens around $\lambda = 0.1$, and that is the only point where Lasso can be considered as better than Ridge. For small penalties, Ridge is way better than Lasso.

7 Conclusion

After what we have seen, we can conclude that OLS gives the best results for our particular data set, but Ridge is not a bad choice either. Lasso gives both higher mean square error (MSE) and lower R^2-score than the other methods, and since the implemented Lasso gives similar results to the Lasso function of Scikit Learn, the Lasso is considered as the least good method among the methods that we have tested. We have also seen that the lower penalty the higher R^2-score, which is expected since OLS is the best method.

As an afterthought, the choice of terrain data was maybe too ambitious, and I do not find the results sufficient. To obtain a better fit, the complexity could be increased by increasing the polynomial degree, or we could turn to other regression methods such as logistic regression. Apropos the polynomial degree, I misunderstood what polynomial of degree 5 was, and used maximum degree of 5 in each spatial directions (should of course have fixed this). Despite this misunderstanding, the mathematical operations I have used are still as valid as for a polynomial of maximum total degree of 5.

8 References

[1] A.M.Legendre. Nouvelles méthodes pour la détermination des orbites des comètes. Libraure pour les Mathématiques. (1805).

[2] C.F. Gauss. Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientum. Hamburg: Frid. Perthes and I.H. Besser. (1809).

[3] T. Hastie, R. Tibshirani, J. Friedman. The Elements of Statistical Learning. Springer-Verlag, New York. (2009).

[4] United States Geological Survey (USGS). https://earthexplorer.usgs.gov/

[5] Lecture notes in Statistical Physics: Statistical Physics - a second course. F. Ravndal, E. G. Flekkœy. (2014).

[6] Simulation stimulation. L. Bastick. https://www.sumproduct.com/thought/simulation-stimulation

[7] Machine Learning Crash Course: Part 4 - The Bias-Variance Dilemma. D. Geng, S. Shih. https://ml.berkeley.edu/blog/2017/07/13/tutorial-4/ (2017).

[8] Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit? http://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit (2013).

[9] Bootstrap Methods: Another Look at the Jackknife. B. Efron. Ann. Statist., Volume 7, Number 1, 1-26. (1979).

[10] Lecture notes FYS-STK4155: Regression Methods. https://compphysics.github.io/MachineLearning/doc/pub/Regression/pdf/Regression-minted.pdf M. Hjorth-Jensen. (2018).

[11] Scikit-learn: Machine Learning in Python. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thiron, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. Journal of Machine Learning Research, Volume 12. (2011).