Linear regression for approximating real and synthetic data

Jonatan H. Hanssen
Bachelor Student, Robotics and Intelligent Systems
The faculty of Mathmatics and Natural Sciences
Department of Informatics
Email: jonatahh@ifi.uio.no

Eric E. Reber
Bachelor Student, Robotics and Intelligent Systems
The faculty of Mathmatics and Natural Sciences
Department of Informatics
Email: ericer@ifi.uio.no

1 Abstract

We used linear regression to create a fit of a synthetic dataset generated by the Franke Function, and for real datasets of elevation data from terrain in Norway. More specifically, we used Ordinary Least Squares (OLS), Ridge and Lasso regression to explore how the different methods perform on different datasets. Comparing the Mean Squared Error (MSE) for the different methods, we found that Ridge and Lasso outperform OLS for small datasets and high polynomial degrees, but that OLS, given the optimal polynomial degree for the dataset, yields the lowest MSE. Overall we found that the different forms of linear regression fit the synthetic data well. However, when looking at the real datasets, we found that all forms of linear regression performed relatively poorly. We concluded that linear regression perhaps is not the best way of predicting terrain data with rapid and irregular changes in elevation.

2 Introduction

One of the core challenges in machine learning is model selection, because we cannot make a priori assessments about how well the model will fit our dataset. A multitude of factors determine the efficacy of our model, the machine learning method we use, the hyperparameters of said method, and the model complexity we select. An uninformed decision in model selection could result in an under- or overfitting to the data, yielding poor results. Thus, it is important that we rigorously test our tuning such that these factors coincide at an intersection, providing the minimal error in unity.

In this paper, we will provide a detailed inspection of the Ordinary Least Squares (OLS), Ridge and Lasso linear regression methods, it's hyperparameters and varying model complexity with the aim of making an informed decision about model selection when fitting to different datasets. The datasets consist of synthetic data created by the Franke function, and real datasets of terrain in Norway. We inform our decisions by plotting the mean squared error (MSE), our cost function, over different polynomial degrees describing model complexity, over different resampling techniques and over different values of lambda, which is the hyperparameter responsible for a suitable regularization. We will perform a bias-variance decomposition for the different linear regression methods to check for under- and overfitting and plot our implementations of the methods up against popular implementations such as scikit learn to ensure that the behaviour of our models is expected.

First, we will go more into depth about the methods and the theory behind them, along with a brief description of our implementation. We will further continue to the results, followed up by a critical discussion of the results. Finally, we will give a conclusion on the optimal model selection for the different datasets.

3 Method

3.1 Theory

This sections covers the theoretic background which underlies the methods used in this paper. We use the following shorthands for our mathematical notation.

3.1.1 Linear regression

To describe how linear regression works, let us first describe the problem it aims to solve. Assume we have a dataset, which contains a series of inputs $\{x_i\}$, and corresponding target values $\{y_i\}$. Our assumption is that there is some relationship $y_i = f(x_i) + \varepsilon_i$ where $\varepsilon_i \sim N(0, \sigma^2)$, and we wish to create a model which approximates this relationship. More specifically, we wish to find a vector β which gives us the "best" approximation $\tilde{\boldsymbol{y}}$ of $\boldsymbol{y}$, by using the parameters in the

following equation:

$$X\boldsymbol{\beta} = \tilde{\boldsymbol{y}} \tag{1}$$

X is called the design matrix, and is an n by p matrix created by our input data and our choice of basis functions, where n is the amount of different input states we have and p is the number of features we use for our fit. For our problem, n is the number of distinct x values times the number of distinct y values, and p is based on the polynomial degree we use. The basis functions are the summands of a two variable polynomial of degree N. The element x_{ij} is calculated by evaluating the basis function corresponding the coloumn j for the input data corresponding to the row i. As we can see from Eq. 1, each row of this matrix is scalar multiplied by the vector β, giving us a linear combination of the values of X_{i*} for each y_i in the vector $\tilde{\boldsymbol{y}}$. The vector $\boldsymbol{\beta}$ is called the parameter vector, and finding the optimal values $\hat{\boldsymbol{\beta}}$ is the problem we wish to solve. But for this, we need to define what an "optimal" prediction $\tilde{\boldsymbol{y}}$ is.

3.1.2 The cost function

To define the optimal prediction, we use a cost function, and the choice of cost function greatly influences what optimal parameters we get.[1] An intuitive cost function is the mean squared error (MSE). This function simply computes the average distance between our prediction and the target value, squared:

$$C_{mse}(\tilde{\boldsymbol{y}}) = \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{2}$$

We may now plug in the definition of $\tilde{\boldsymbol{y}}$.

$$C_{mse}(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} (y_i - X_{i*}\boldsymbol{\beta})^2 \tag{3}$$

We see that we have a function of our parameters $\boldsymbol{\beta}$, which describe how much our prediction deviates from our target values. Our problem now is to optimize this function, that is, find the values of β which gives the MSE the lowest value. Luckily for us, it can be proven that the second derivative of this function is always positive (WHERE?), which means that we can find the global minimum by solving the equation

Hastie 45

$$\frac{\delta}{\delta\beta} \sum_{i=0}^{n-1} (y_i - X_{i*}\boldsymbol{\beta})^2 = 0 \tag{4}$$

$$-\frac{2}{n} X^T (\boldsymbol{y} - X\boldsymbol{\beta}) = 0 \tag{5}$$

Solving this for β gives us the so called normal equation, an analytical solution[2] for the optimal $\hat{\beta}$.

$$-\frac{2}{n} X^T (\boldsymbol{y} - X\boldsymbol{\beta}) = 0 \tag{6}$$

$$X^T (\boldsymbol{y} - X\boldsymbol{\beta}) = 0 \tag{7}$$

$$X^T \boldsymbol{y} - X^T X\boldsymbol{\beta} = 0 \tag{8}$$

$$X^T \boldsymbol{y} = X^T X\boldsymbol{\beta} \tag{9}$$

$$\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \boldsymbol{y} \tag{10}$$

The process we have just described, with MSE as the cost function, is what is known as Ordinary Least Squares (OLS) regression. Using different cost functions when fitting the model leads to different equations for $\hat{\boldsymbol{\beta}}$. This leads us to the two next forms of linear regression we use in this paper. But before this, a...

3.1.3 Brief statistical intermission

We must remember that our target values are defined as $y_i = f + \varepsilon_i$, that is, they are distorted by a stochastic noise $\varepsilon_i \sim N(0, \sigma^2)$. We should therefore take a brief intermission to prove some of the statistical qualities of the method we have just described, so as to reassure ourselves that we are justified in applying linear regression to noisy data. We will start by showing that

$$E[y_i] = \sum_j X_{ij}\beta_j \tag{11}$$

The proof is as follows:

$\boldsymbol{y}$ is defined as the $f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$. We make an assumption that this can be approximated as a matrix multiplication $X\boldsymbol{\beta}$ plus an error vector $\boldsymbol{\varepsilon}$. This means that each element of the vector $\boldsymbol{y}$ can be expressed as follows:

$$y_i = \sum_j x_{ij}\beta_j + \varepsilon_i \tag{12}$$

[1] It is here important to keep in mind that the cost function used when fitting the model (training) rarely is the same as the one used for testing our prediction, as described in ¡somewhere¿

[2] This solution requires that $X^T X$ is non-singular, which may not be the case. This is solved by using the singular value decomposition of the design matrix, which we will not go into here

If we take the expectation value of this expression we get the following:

$$E[y_i] = E\left[\sum_j x_{ij}\beta_j + \varepsilon_i\right] \tag{13}$$

$$E[y_i] = E\left[\sum_j x_{ij}\beta_j\right] + E[\varepsilon_i] \tag{14}$$

However, the elements of X are not stochastic, and neither are the elements of β and therefore the first expectation value is simply the sum itself. Furthermore, ε is explicitly defined as a normal distribution $N(0,\sigma^2)$, and will by definition have the expectation value 0. Therefore, we end up with the final expression:

$$E[y_i] = \sum_j x_{ij}\beta_j = \boldsymbol{X}_{i,*}\boldsymbol{\beta} \tag{15}$$

Next, we show that the variance of y_i is equal to the variance of the noise:

$$var[y_i] = E\left[(y_i - E[y_i])^2\right] \tag{16}$$

$$= E\left[y_i^2 - 2E[y_i]y_i + E[y_i]^2\right] \tag{17}$$

Distributing the outer expectation value function:

$$var[y_i] = E[y_i^2] - 2E\left[E[y_i]\right]E[y_i] + E\left[E[y]^2\right] \tag{18}$$

$$= E[y_i^2] - 2E\left[E[y_i]\right]E[y_i] + E\left[E[y]\right]E\left[E[y]\right] \tag{19}$$

The result of calculating the expectation value is non-stochastic. This means that $E[E[X]] = E[X]$. From this it follows that

$$var[y_i] = E[y_i^2] - 2E[y_i]E[y_i] + E[y_i]E[y_i] \tag{20}$$

$$= E[y_i^2] - E[y_i]E[y_i] \tag{21}$$

$$= E[y_i^2] - E[y_i]^2 \tag{22}$$

The expectation value in the second summand has been proven to be equal to $\boldsymbol{X}_{i,*}\boldsymbol{\beta}$ above. We therefore have

$$var[y_i] = E[y_i^2] - (\boldsymbol{X}_{i,*}\boldsymbol{\beta})^2 \tag{23}$$

$$= E\left[(X_{i,*}\boldsymbol{\beta})^2 + X_{i,*}\boldsymbol{\beta}\varepsilon_i + \varepsilon_i^2\right] - (\boldsymbol{X}_{i,*}\boldsymbol{\beta})^2 \tag{24}$$

$$= E\left[(X_{i,*}\boldsymbol{\beta})^2\right] + E[X_{i,*}\boldsymbol{\beta}\varepsilon_i] + E[\varepsilon_i^2] - (\boldsymbol{X}_{i,*}\boldsymbol{\beta})^2 \tag{25}$$

$X_{i,*}\boldsymbol{\beta}$ and ε_i are both scalars. Therefore the expectation value can be written as $E[X_{i,*}\boldsymbol{\beta}]E[\varepsilon_i]$. However, $E[\varepsilon_i]$ is by definition 0, because ε_i is defined as a normal distribution of mean 0 and variance σ^2. Furthermore, the expectation value of the non-stochastic $(X_{i,*}\boldsymbol{\beta})^2$ is simply the expression itself. We therefore have

$$var[y_i] = (X_{i,*}\boldsymbol{\beta})^2 + E[X_{i,*}\boldsymbol{\beta}]\cdot 0 + E[\varepsilon_i^2] - (\boldsymbol{X}_{i,*}\boldsymbol{\beta})^2 \tag{26}$$

$$= E[\varepsilon_i^2] \tag{27}$$

We can prove that this is equal to the variance of ε_i:

$$E[\varepsilon_i^2] = \frac{1}{n}\sum_i \varepsilon_i^2 \tag{28}$$

$$var[\varepsilon_i] = \frac{1}{n}\sum_i (\varepsilon_i - \bar{\varepsilon}_i)^2 \tag{29}$$

$$var[\varepsilon_i] = \frac{1}{n}\sum_i (\varepsilon_i - 0)^2 \tag{30}$$

$$var[\varepsilon_i] = \frac{1}{n}\sum_i \varepsilon_i^2 = E[\varepsilon_i^2] \tag{31}$$

And of course, we know that the variance of ε_i by definition is σ^2.

$$var[y_i] = E[\varepsilon_i^2] = var[\varepsilon_i] = \sigma^2 \tag{32}$$

We should also prove some qualities about or parameters. We start by proving that $E[\hat{\beta}] = \beta$

Because we are assuming $(X^TX)^{-1}X^T$ to be deterministic, $E[(X^TX)^{-1}X^T] = (X^TX)^{-1}X^T$. Moreover, we know $E[y] = y$, and we can thus rewrite the equation as follows:

$$E[(X^TX)^{-1}X^Ty] = (X^TX)^{-1}X^Ty = \beta \tag{33}$$

Next, we prove that $var(\beta) = \sigma^2(X^TX)^{-1}$.

$$var(\beta) = var((X^T X)^{-1} X^T y)$$

Because we are assuming $(X^T X)^{-1} X^T$ to be deterministic and y to be stochastic, we can rewrite the $var(\beta)$ as following:

$$var(\beta) = (X^T X)^{-1} X^T var(y)((X^T X)^{-1} X^T)^T \quad (34)$$
$$= (X^T X)^{-1} X^T var(y)(X^T)^T ((X^T X)^{-1})^T \quad (35)$$

Since $(X^T)^T = X$ and $(X^T X)^{-1})^T = (X^T X)^T)^{-1} = (X^T X)^{-1}$

$$= (X^T X)^{-1} X^T var(y) X (X^T X)^{-1} \quad (36)$$

We know var(y) = σ^2, and since it is a scalar it is commutative. Thus we may move it freely

$$= \sigma^2 (X^T X)^{-1} X^T X (X^T X)^{-1} \quad (37)$$
$$= \sigma^2 (X^T X)^{-1} \quad (38)$$

3.1.4 Ridge Regression

A common problem with OLS is what is known as overfitting (we will discuss this in greater detail in later sections). A symptom of overfitting is that the values in the parameter vector $\boldsymbol{\beta}$ grow very large, and vary wildly for small changes in our dataset. To counteract this, we can add to the cost function a term which increases with the values in $\boldsymbol{\beta}$. Optimizing the cost function would now not only mean finding the smallest difference between target and predicted values, but also keeping the parameters small. We now define a new cost function

$$C_{lasso}(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} (y_i - X_{i*}\boldsymbol{\beta})^2 + \lambda \sum_{j=0}^{p-1} \beta_j^2 \quad (39)$$

$||\beta||_2^2$ is equal to $\beta^T \beta$ and is simply the norm of the parameter vector, squared. With the new hyperparameter $\lambda \in (0, \inf)$, we can decide how harshly we should punish large values in our parameter vector. Finding an optimal λ is an important step in the process of finding the best parameters for our model.

A similar derivation as done in the previous section gives us the normal equation for Ridge regression:

$$\hat{\boldsymbol{\beta}} = (X^T X + \lambda I)^{-1} X^T \boldsymbol{y} \quad (40)$$

We see that this equation, with a $\lambda > 0$ avoids the problem of an inversion of a non-singular matrix, and gives us an analytical solution for $\hat{\boldsymbol{\beta}}$ yet again. [3] By using ridge regresison, we can control the complexity of our model and effectively hinder overfitting.

3.1.5 Lasso Regression

Another cost function is one where we add the L1-norm (the sum of the absolute values of the parameters), instead of the L2-norm. The new cost function is as follows.

$$C_{lasso}(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} (y_i - X_{i*}\boldsymbol{\beta})^2 + \lambda \sum_{j=0}^{p-1} |\beta_j| \quad (41)$$

However, the derivative of the absolute value is not continous, so there is sadly no analytical solution for the optimal parameters with this form of linear regression. We must then solve the problem numerically, which can be done by using Gradient Descent. Like Ridge regression, this form of regression controls the complexity of the model, with the added benifit that Lasso can drive parameters to exactly zero.

3.1.6 Resampling methods: The bootstrap

Bootstrap is a resampling method divided into simple steps. First, we reshuffle our dataset with replacement, which will provide us with a new dataset. Note that we will already have split our data into train and test data before we begin the bootstrap, so when we reshuffle, we will only reshuffle the training data. The reshuffled datasets act as our samples.

Secondly, we will perform our linear regression on this new dataset and store the predictions of the test data. Thirdly, we repeat for a given large number of iterations. Since we use all the test predictions to calculate the MSE, bias and variance, we will then according to the central limit theorem (REFERENCE!) obtain a normal distribution of these variables. The law of large numbers ¡reference?¿ then tells us that the mean sample values of the MSE, bias and variance approach the true value of MSE, bias and variance. This is dependent on our assumption that our data is independently and identically distributed.

These approximations of the true value of MSE, bias and variance will become useful when performing our bias-variance trade-off analysis.

[3] We should cherish this analytical solution, as it may well be the last time we see a cost function that can be optimized analytically.

3.1.7 Cross validation

Cross validation is another resampling technique, in which we reshuffle our data and then split them into k-folds, where k specifies the number of folds. One fold is used as our test data, and the rest is used as training data. We run our linear regression, then switch up which fold is used for testing until all folds have been used. As in bootstrapping, we will approach the true values of for example the mean squared error for our model. Theoretically, because both two resampling techniques use the central limit theorem and the law of large numbers, they should return the same values for the MSE, bias and variance given the same dataset. Though resampling techniques can be computationally expensive, they make up for it by aiding in model evaluation by supplying us with variables obtained from a larger number of samples.

3.1.8 Bias-variance trade-off

A crucial part of model evaluation is ensuring that the model is neither underfitted nor overfitted, which we can determine by observing the relationship between the bias and the variance. Bias is defined as $(Bias[\tilde{y}])^2 = (y - E[\tilde{y}])^2$, the difference between the expected value of our prediction and the observed value. Models with high bias are usually underfitted, whilst models with low bias are the opposite. Variance on the other hand is defined as $var[\tilde{f}] = \frac{1}{n}\sum(\tilde{y}_i - E[\tilde{y}])^2$, how far away predictions are from the mean prediction. Models with high variance are usually overfitted to the training data, whilst low variance models are the opposite.

A good rule of thumb for finding a good fit is looking at the intercept between bias and variance. This is because the cost function can be decomposed into these two elements. The proof for this is as follows:

Given data $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$ and a OLS model $\tilde{\boldsymbol{y}} = X\boldsymbol{\beta}$, we define the Mean Squared Error as follows

$$C(X,\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{42}$$

This is simply the expectation value of $(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2$

$$C(X,\beta) = E\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right] \tag{43}$$

To derive the bias variance decomposition, we now add $E[\tilde{\boldsymbol{y}}] - E[\tilde{\boldsymbol{y}}]$ as a sneaky trick. We will also replace $\boldsymbol{y}$ with our definition $\boldsymbol{y} = \boldsymbol{f} + \boldsymbol{\varepsilon}$

$$C(X,\beta) = E\left[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}] + E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})^2\right] \tag{44}$$

$$= E[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])^2 + 2[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])(E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})] + (E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})^2] \tag{45}$$

$$= E[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])^2] + 2E[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])(E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})] + E[(E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})^2] \tag{46}$$

Let us focus our attention on the middle term first.

$$2E\left[(\boldsymbol{f} + \boldsymbol{\varepsilon} + E[\tilde{\boldsymbol{y}}])(E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})\right] \tag{47}$$

Expanding the multiplication:

$$2E\left[\boldsymbol{f}E[\tilde{\boldsymbol{y}}] + \boldsymbol{\varepsilon}E[\tilde{\boldsymbol{y}}] - E[\tilde{\boldsymbol{y}}]E[\tilde{\boldsymbol{y}}] - \boldsymbol{f}\tilde{\boldsymbol{y}} - \boldsymbol{\varepsilon}\tilde{\boldsymbol{y}} + E[\tilde{\boldsymbol{y}}]\tilde{\boldsymbol{y}}\right] \tag{48}$$

Distributing the expectation value:

$$2\left[E[\boldsymbol{f}E[\tilde{\boldsymbol{y}}]] + E[\boldsymbol{\varepsilon}E[\tilde{\boldsymbol{y}}]] - E[E[\tilde{\boldsymbol{y}}]E[\tilde{\boldsymbol{y}}]] - E[\boldsymbol{f}\tilde{\boldsymbol{y}}] - E[\boldsymbol{\varepsilon}\tilde{\boldsymbol{y}}] + E[E[\tilde{\boldsymbol{y}}]\tilde{\boldsymbol{y}}]\right] \tag{49}$$

We know that $\tilde{\boldsymbol{y}}$ and $\boldsymbol{f}$ are non-stochastic, and we know that the expectation value of $\boldsymbol{\varepsilon}$ is 0. We also know that $\boldsymbol{\varepsilon}$ and $\tilde{\boldsymbol{y}}$ are independent, which means we can write $E[\boldsymbol{\varepsilon}\tilde{\boldsymbol{y}}]$ as $E[\boldsymbol{\varepsilon}]E[\tilde{\boldsymbol{y}}]$ We therefore have

$$2\left[\boldsymbol{f}\tilde{\boldsymbol{y}} + 0 - \tilde{\boldsymbol{y}}\tilde{\boldsymbol{y}} - \boldsymbol{f}\tilde{\boldsymbol{y}} - 0 + \tilde{\boldsymbol{y}}\tilde{\boldsymbol{y}}\right] = 0 \tag{50}$$

We then have

$$C(X,\beta) = E\left[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])^2\right] + E\left[(E[\tilde{\boldsymbol{y}}] - \tilde{\boldsymbol{y}})^2\right] \tag{51}$$

The second term is by definition the variance of model.

$$C(X,\beta) = E\left[(\boldsymbol{f} + \boldsymbol{\varepsilon} - E[\tilde{\boldsymbol{y}}])^2\right] + var[\tilde{\boldsymbol{f}}] \tag{52}$$

And now the final term

$$E\left[(\boldsymbol{f}+\boldsymbol{\varepsilon}-E[\tilde{\boldsymbol{y}}])^2\right] \tag{53}$$

$$= E[\boldsymbol{f}^2+\boldsymbol{f}\boldsymbol{\varepsilon}-\boldsymbol{f}E[\tilde{\boldsymbol{y}}]+\boldsymbol{f}\boldsymbol{\varepsilon}+\boldsymbol{\varepsilon}^2- \boldsymbol{\varepsilon}E[\tilde{\boldsymbol{y}}]-\boldsymbol{f}E[\tilde{\boldsymbol{y}}]-\boldsymbol{\varepsilon}E[\tilde{\boldsymbol{y}}]+E[\tilde{\boldsymbol{y}}]^2] \tag{54}$$

After distributing the expectation value we get the following, keeping in mind that $E[\boldsymbol{f}] = E[\boldsymbol{y}-\boldsymbol{\varepsilon}] = E[\boldsymbol{y}]-E[\boldsymbol{\varepsilon}] = \boldsymbol{y}$

$$\boldsymbol{y}^2-2\boldsymbol{y}\tilde{\boldsymbol{y}}+E[\tilde{\boldsymbol{y}}]^2+\sigma^2 \tag{55}$$

$$(\boldsymbol{y}+E[\tilde{\boldsymbol{y}}]^2+\sigma^2 \tag{56}$$

We see that the quadratic form is equal to $bias[\tilde{\boldsymbol{y}}]^2$. We can then conclude that the cost function can be rewritten as follows:

$$C(X,\beta) = bias[\tilde{\boldsymbol{y}}]^2+var[\tilde{\boldsymbol{f}}]+\sigma^2 \tag{57}$$

Because the cost function is a sum of the bias and the variance, the lowest error will generally occur when both bias and variance are low. This also fits our understanding of the definitions of bias and variance, as we wish to obtain a model that is neither high in bias (underfitted) or high in variance (overfitted).

With the theory explained, we can go on to describe how we have applied it to our problem.

3.2 Implementation

3.2.1 Preprocessing: Creating the design matrix and normalizing our data

We have two datasets, one synthetic and one real. Regardless of the source of the data, we create the design matrix in a similar manner, as the relation between our input data and our target values are the same for both datasets, that is, for each combination of x and y, we have a height z. For the Franke function this height is no more than the height over the xy-plane, but for our real data, this value represents the elevation in meters above ocean level for a terrain in Norway. As we want to do a polynomial fit of the data, we create our design matrix such that each row i has the following values:

$$X_{i*} = 1 \;\; x_i \;\; y_i \; x_i^2 \;\; x_i y_i \;\; y_i^2 \;\; \ldots \;\; x_i^p y_i^p \tag{58}$$

Below is the python code used to create the design matrix, which can be found in `src/utils.py`.

```
    return 0 + 1*x + 2*y + 3*x**2 + 4*x*y + 5*y**2


def create_X(x, y, n):
    if len(x.shape) > 1:
        x = np.ravel(x)
        y = np.ravel(y)

    N = len(x)
    l = int((n + 1) * (n + 2) / 2)  # Number of
    elements in beta
    X = np.ones((N, l))

    for i in range(1, n + 1):
        q = int((i) * (i + 1) / 2)
        for k in range(i + 1):
            X[:, q + k] = (x ** (i - k)) * (y**k)
```

While the creating of the design matrix is equivalent for the two datasets, the preprocessing of the data afterwards is not. For the Franke function, we claim that no normalization is needed, as we generate the data ourselves to be between 0 and 1. Given that we are doing a polynomial fit, every element in our design matrix will be a multiplication of x^p and y^k for different p's and k's, which, given our choice of range for x and y, guarantees that every element is between 0 and 1. However, we have included a centering of the data, which has no effect on the Ordinary Least Squares regression, but which is used when applying Ridge and Lasso regression, to avoid punishing the intercept. The centering is done as follows: First, the first coloumn of the design matrix (the intercept coloumn) is discarderd. Then we calculate the mean for the target values and the design matrix (the mean for the design matrix is a vector with mean values for each coloumn). We subtract this from the design matrix and target values when performing the fit giving us a vector $\boldsymbol{\beta}$. Afterwards we calculate the intercept with the following equation

$$intercept = mean(\bar{y}-\bar{X}_c^T\boldsymbol{\beta}) \tag{59}$$

where $\bar{y}$ is the mean value of $\boldsymbol{y}$ and $\bar{\boldsymbol{X}}_{\boldsymbol{c}}$ is a vector of size p with $\bar{X}_i = mean(X_i)$ (the mean of each coloumn in X). We finally create our prediction by using the $\boldsymbol{\beta}$, X and intercept we calculated above:

$$\hat{\boldsymbol{y}} = X\boldsymbol{\beta}+intercept \tag{60}$$

The code for this, taken from the `evaluate_model` function in `src/utils.py` is as follows: [4]

[4] Some intermediate parts of our implementation not related to this have been removed for readability

```
X_train = X_train[:, 1:]
z_train_mean = np.mean(z_train, axis=0)
X_train_mean = np.mean(X_train, axis=0)
beta = model((X_train - X_train_mean), (z_train -
    z_train_mean))
intercept = np.mean(z_train_mean - X_train_mean @
    beta)
z_pred_train = X_train @ beta + intercept
```

Where `model` calculates `beta` using the normal equations for OLS.

For the real elevation data, normalization is definitely required, as the values for x, y reach several hundred and z reaches over a thousand. For polynomial degree 10 and an x value which reaches 400, we will have a design matrix with values from 0 to 400^{10}. This large range of values makes our linear regression perform poorly (WHY?). We have therefore chosen to scale our design matrix using Scikit's `MinMaxScaler`, which scales each column to be in the range of $(0,1)$. The calculation of mean and variance is done on the training data, and applied to both the training and test data. We have done the same with the values for z. This gives a much more stable model. The code for doing this on the design matrix is as follows, taken from `src/utils.py`:

```

    return beta, z_pred_train, z_pred_test, z_pred


def minmax_dataset(X, X_train, X_test, z, z_train,
     z_test):
    x_scaler = MinMaxScaler()
    z_scaler = MinMaxScaler()
```

Another form of "preprocessing" which is strictly practical in nature, is that we have reduced the dimensions of the data to 320 by 640. This was done by downscaling the image. [5] This was simply done because the design matrix becomes several gigabytes in size if one uses the whole dataset, and model fitting takes an unreasonable amount of time.

3.2.2 Implementation of linear regression

Linear regression for OLS and Ridge are implemented by using NumPy's `numpy.linalg.pinv` and the normal equations. We have implemented them as functions which take in the target values and the design matrix and return $\hat{\boldsymbol{\beta}}$. They are called from a wrapper function called `evaluate_model` which takes care of the scaling and works with both SciKit models and our own. Below are the functions, taken from `src/utils.py`:

```
def OLS(X_train: np.ndarray, z_train: np.ndarray):
    beta = np.linalg.pinv(X_train.T @ X_train) @
    X_train.T @ z_train
    return beta


def ridge(X_train, z_train, lam):
    L = X_train.shape[1]
    beta = np.linalg.pinv(X_train.T @ X_train +
    lam * np.eye(L)) @ X_train.T @ z_train
    return beta
```

For repeated calls of these functions with different polynomial degrees, we use the function `linreg_to_N`, which passes the correctly sliced design matrix to the regression functions and saves the predictions and various performance metrics for each polynomial degree for easy plotting.

For Lasso regression, we use SciKit's implementation. We also use SciKit's implementation for OLS and Ridge for comparisons with our own implementation.

The correctness of our Ordinary Least Squares regression can be confirmed by running our debug function (the so called SkrankeFunction)[6], which is a second degree polynomial where $b_0 = 0$, $b_1 = 1$, $b_2 = 2$ and so on. The SkrankeFunction (taken from `src/utils.py`) looks like this

```
def SkrankeFunction(x, y):
    return 0 + 1*x + 2*y + 3*x**2 + 4*x*y + 5*y**2
```

and the result of our regression is correctly given as $\boldsymbol{\beta} = (0\,1\,2\,3\,4\,5\,)^T$.

3.2.3 Implementation of resampling methods

We use resampling methods to better estimate our performance metrics. We use both cross validation and the bootstrap method. Cross validation is implemented in the function `crossval` in `src/utils.py`. We split the data into K folds by slicing with NumPy taking consecutive rows of the design matrix and target vector. But before we do this, we resample the data with Scikit's `resample`. This is important because our x's and y's are not normal distributions but consecutive values between 0 and 1, and therefore consecutive rows correspond specific areas of our data, which is not wanted. We then fit on $K-1$ folds, and test on the last. We return the average error.

We implement bootstrap by repeatedly resampling our data with replacement and performing the prediction. We use the predictions returned from hundreds of resamplings to calculate the bias and variance of our model.

3.2.4 Franke Function

We use the Franke Function with an added stochastic noise of to test our models. We explore how higher and lower levels of noise affect the model. Below is the Franke function:

[5] The interpolation was done using "nearest neighbor interpolation", which does not introduce new values into the data. An equivalent programmatic solution would be to for instance only look at every 4 pixels

[6] this can be done by running `$ python3 task_b.py --debug --noscale -n 2`

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) + \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \quad (61)$$

As we can see, it is simply a weighted sum of exponentials. As such it has an infinite Taylor expansion into polynomials.[7]

4 Results

4.1 Franke Function

To gain insight into how the three different regression methods work, we experiment with synthetic data generated by the Franke function.

4.1.1 Ordinary Least Squares

We find that OLS creates a good fit for the Franke Function. With 400 datapoints, we can increase our polynomial degree to around 20 before we see signs of overfitting.[8] This is most likely because the Franke function can be expanded into a sum of polynomials. We would expect to see higher polynomial degrees give us better and better prediction, up to a certain point.[9] When we reach degrees higher than around 20, we see that our test error increases slightly, a sign of overfitting. This is also to be expected, as even if the continual Franke function could be expanded into a series of polynomials, we are working with a limited dataset of only 400 samples (and only 320 of those in our training set). As we reach higher and higher degrees, we are more and more likely to see unlucky train-test-splits where the best fit for the training data leads to large errors for the test data. See Fig. 1 for a comparison between the Franke function and our best prediction, which shows that they are essentially identical. Fig. 2 shows the train and test MSE by polynomial degree. The MSE for test in these plots is the average MSE for 100 bootstraps. For a comprehensive list of the shell command required to reproduce this and every other plot, see Table. 4 in the appendix.

[7]The proof for this is left as an exercise for the reader

[8]Here we also see something a bit surprising: We see that our own implementation of OLS follows Scikit's implementation up to around polynomial degree 10, at which point scikit's MSE increases rapidly. We chalk this up to numerical errors and differing implementation of the normal equations, as we simply use the normal equation exactly, while scikit uses an implementation which is probably computationally more efficient but gives slight differences when comparing to the normal equation. These differences seem to create a large error after a certain size of the design matrix is reached, which we can not adequately explain.

[9]And indeed, this seems to be the case, as if we increase the availiable datapoints considerably, as seen in Fig. 9 in the appendix, we can reach much higher polynomial degrees without overfitting

Table 1. Best MSE and polynomial degree for OLS

Std of noise	Best MSE	Best degree
0	0.0005109	9
0.05	0.0038302	7
0.10	0.0139833	7

However, when we introduce a stochastic noise, we see that overfitting happens rapidly. Without a regularization hyperparameter, parameters fluctuate wildly and our prediction suffers. Fig. 3 shows our MSE up to polynomial degree 25 with an added stochastic noise $\varepsilon \sim N(0, 0.05)$, where we see that we are overfitting severely. See Fig. 7 in the appendix for test MSE for more and more noisy target values.

By using bootstrap resampling, we can study the bias-variance-tradeoff, which shows how the variance of our model between different bootstraps iterations increases as our model overfits our noisy data. See Fig. 4. Clearly, we need to introduce regularization if we want a model that does not overfit. Fig. 10 in the appendix shows the beta progression for selected betas, clearly showing that we are overfitting.

Table. 1 shows the best polynomial degree and MSE for regular and noisy data. Here we see that the best degree without noise is 9, which shows that even without noise, the low amount of data leads to a slight overfit even at very small polynomial degrees. This is not surprising, as at $N = 10$ we have 66 features, which is a substantial amount compared to our measly 320 training datapoints. We could of course arbitrarily increase our dataset to ludicrous levels, but doing linear regression with this dataset gives valuable insight into how our performance is dependent on our availiable data.

4.1.2 Ridge

Using Ridge regression, we see from Fig. 5 that larger and larger values of λ efficiently prevents overfitting and reduces the variance between predictions. Looking at the beta progression with for the same values of λ (Fig. 6), we see that the regularization parameter is successfully shrinking the beta values. Using Scikit's gridsearch, we find the optimal λ to be 10^{-8} and the optimal polynomial degree to be 7, with an average MSE of 0.0037999, which is very slightly lower than OLS for the Franke function with an added stochastic noise $\varepsilon \sim N(0, 0.05)$. We see from this that Ridge regression has not really had much of an effect in decreasing the MSE. We must now remember what Ridge aims to do. With the regularization parameter, we can prevent overfitting if our model is too complex for our data. Looking at Fig. 8 in comparison to Fig. 7 (both in the appendix), we can see that Ridge really has achieved this, reducing overfitting as our data becomes more and more noisy. But we should recall that we decide the complexity of our model, by deciding which polynomial degree we use when creating the design matrix. In fact, by choosing polynomial degree 7 for our OLS method to get the best MSE for noisy data, we are in fact doing a primi-

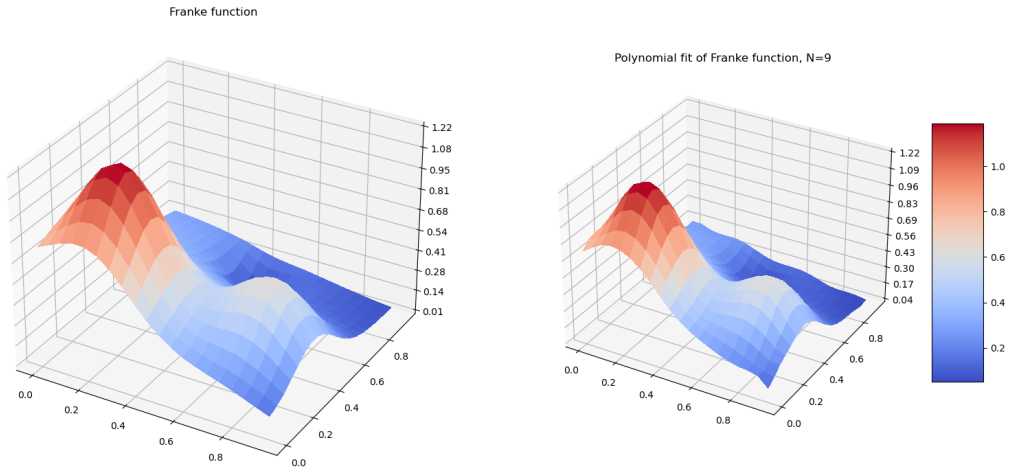

Fig. 1. A comparison of our best OLS prediction and the Franke function with no noise, at polynomial degree 10

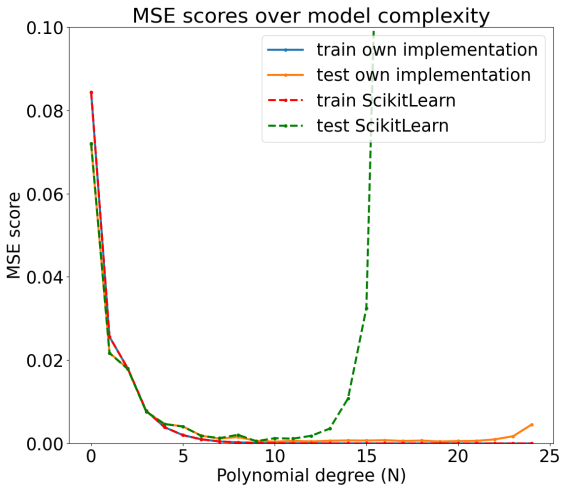

Fig. 2. Test and train MSE for the Franke function with no noise, using OLS

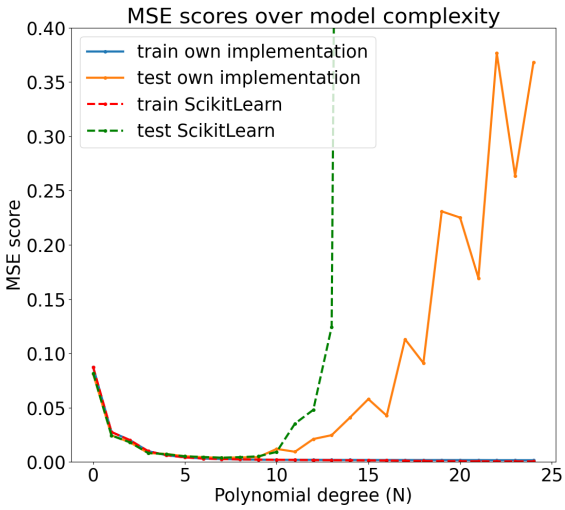

Fig. 3. Test and train MSE for the Franke function with noise, using OLS

tive form of best subset selection, as described in Hastie et Al. [Hastie et al., 2009, 57]. Almost by definition, the polynomial degree with the best MSE is not overfitted, because if it was, it would not be the model with the lowest MSE among all polynomial degrees. Therefore it is not so unexpected that Ridge regression has a relatively low impact on the MSE. We could expect that Ridge would allow us to reach higher polynomial degrees without overfitting, by shrinking features of the higher polynomial degrees that are not supported but allowing us to keep other high degree terms (for example, allowing us to use x^5y^4, but discard x^9, if only x^5y^4 was supported by the data). But we can see from the Table. 2 that we are not able to push into higher degrees. A possible explanation for this is that every feature is useful, due to the polynomial expansion of the Franke function.

4.1.3 Lasso

Looking at Lasso regression, we find that it has the same capability to reduce overfitting as Ridge, as can be seen in Fig. **??**. However, looking at Table. 2, we see that Lasso falls short when finding a good MSE. Intuitively, it does not make sense for Lasso to be worse than OLS, because Lasso with $\lambda = 0$ is equivalent to OLS. But we cannot avoid considering the fact that Lasso does not have an analytical solution, and as such we must use numerical methods to find the optimal

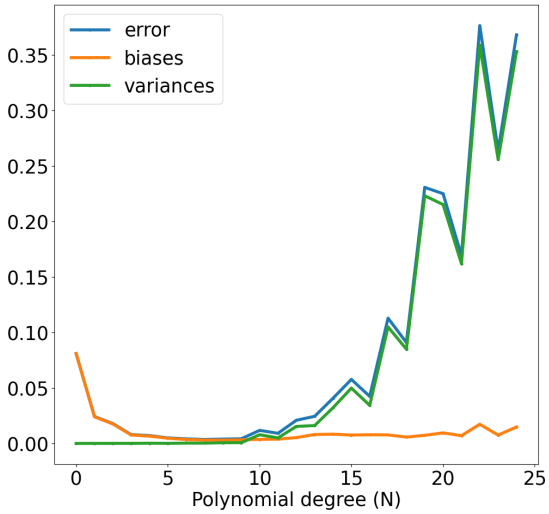

Fig. 4. Bias, variance and test MSE plotted over polynomial degree for our OLS prediction on the Franke function with an added stochastic noise $\varepsilon \sim N(0, 0.05)$

Table 2. Best MSE for OLS, Ridge and Lasso with 400 datapoints

Regression	Std of noise	MSE	Degree	λ
OLS	0.05	0.0038302	7	-
Ridge	0.05	0.0037999	7	10^{-8}
Lasso	0.05	0.0095846	10	$2.5118 \cdot 1$

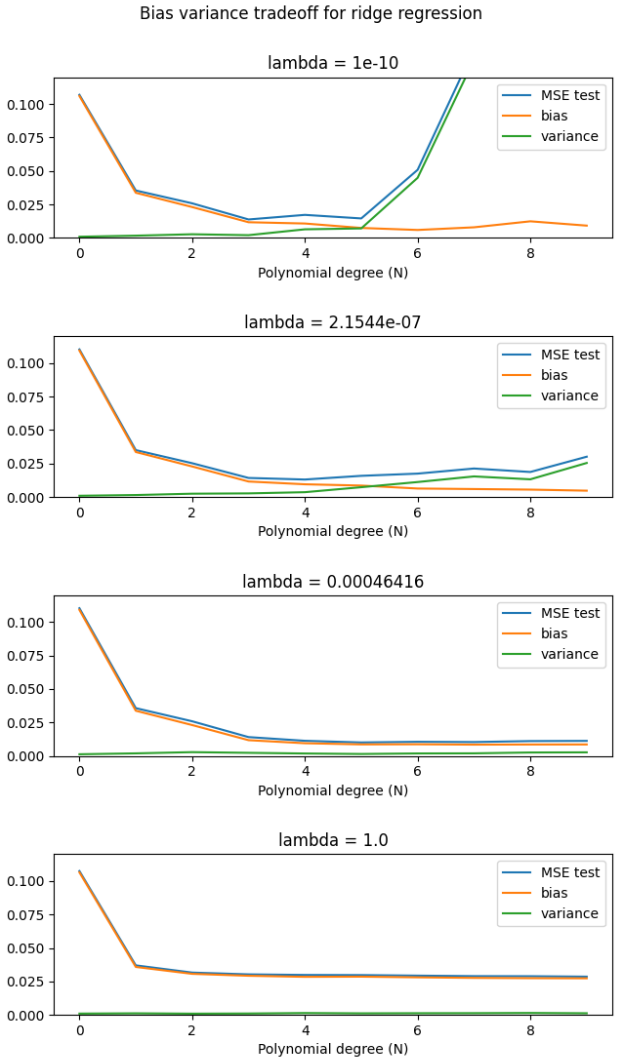

Fig. 5. Bias, variance and test MSE plotted over polynomial degree for our Ridge prediction on the Franke function with an added stochastic noise $\varepsilon \sim N(0, 0.05)$, using lambdas of increasing size. Here we have reduced the number of datapoints to only 100 to force overfitting earlier.

parameters. Using scikit's implementation, we have not been able to get the numerical method to converge, which means that we have not found a global minimum.[10] For this reason, Lasso regression has not given satisfying results.

4.2 Real elevation data

With the confidence that our methods work as expected on synthetic data, we turn our attention to the real elevation data.

4.2.1 Ordinary Least Squares

Here we find some limitations of linear regression. We see from Fig. **??** that the we are able to fit the terrain up to a certain point, gaining an MSE of NUMBER with OLS with polynomial degree DEGREE. However, we see that beyond this, we have little to gain. This can be seen when we increase the polynomial degree to very large amounts, without seeing any decrease in MSE (Fig. NUMBER). Looking at the beta progression for selected betas (Fig. NUMBER), we see that the parameters stabilize, which implies that we are not overfitting, but simply reaching our best possible prediction for polynomial regression. The fact that we are not overfitting is further reinforced by looking at the bias-variance plot in Fig. NUMBER, which shows no increase in variance. It is reasonable to expect that our model will not overfit, due to the large increase in data we now use for the fit. Our synthetic data used $20x20 = 400$ datapoints, while we now have $320x640 = 204800$ datapoints. This means that for a 50th order polynomial, which will have 1326 features, our number of features are less than 1% of our datapoints. If we decrease the number of datapoints used for training considerably, we can see overfitting start to occur (see Appendix 1). So we are not overfitting, but we are not decreasing our MSE either. This is likely because the elevation data has a jagged and complex shape, which cannot be accurately recreated, even with a polynomial of degree 50. We should remember that the Franke function can be expanded into a series of polynomials, which is why we can expect to get a lower and lower MSE as the polynomial degree increases (so long as

[10] Scikit explicitly mentions that using $\lambda = 0$ does not converge well with their implementation

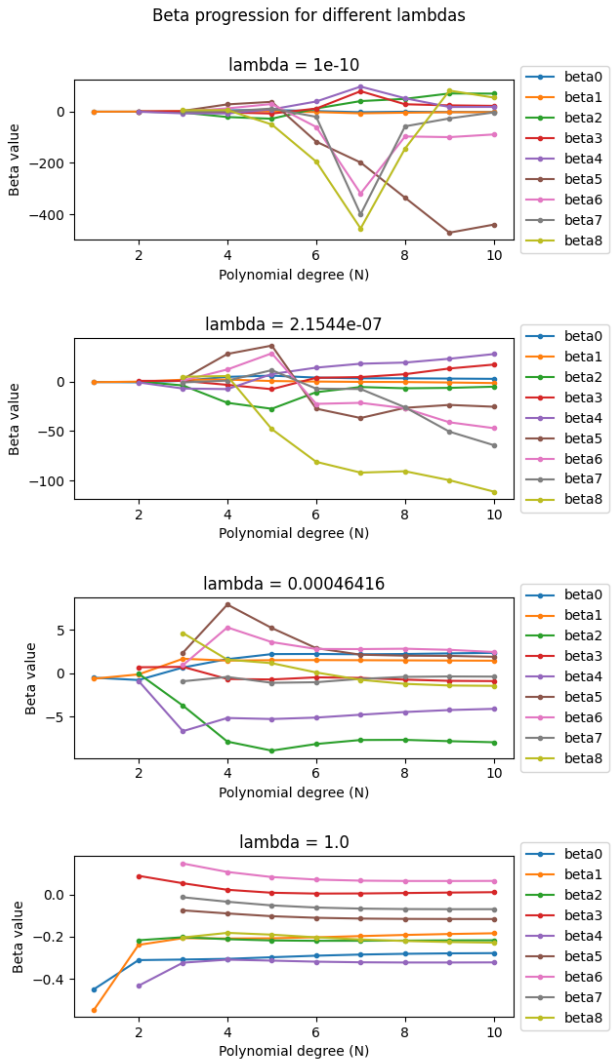

Fig. 6. Beta progression for different values of lambda in Ridge regression. Note the different scales on each of the plots, showing how increasing the regularization parameter shrinks the betas

we have enough data and it is not noisy). The same is not true for our elevation data, and as such we have little to gain from increasing our model complexity after a certain point. Clearly, linear regression does not offer a very good tradeoff between computational cost and performance for this data.

4.2.2 Ridge Regression

As our model is not overfitting, we do not expect Ridge to offer much improvement.

References

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Springer, New Nork, NY.

Appendix A: Plots and tables

The next pages contain plots and tables that are of interest.

Table 3. This table shows every applicable parameter for every figure, so that they may be easily reproduced. Every plot uses the random seed 42069

Figure	Highest N	Std of noise	Step (# of points)	Bootstraps	K-folds	λ	Scaling
4	25	0.05	0.1 (100)	100	-	-	None
5	25	0.05	0.1 (100)	100	-	variable	Centering
6	25	0.05	0.1 (100)	-	-	variable	Centering
10	25	0.05	0.05 (400)	-	-	-	Centering
7	25	variable	0.05 (400)	100	-	-	None
9	30	0	0.01 (10000)	100	-	-	None

Table 4. This table shows the command to execute, as well as the advanced parameters to change, to reproduce every figure in the report. (More info about the scripts and their parameters can be found in the README)

Figure	Shell command (leading `python3` omitted)	Bootstraps	K-folds	λ
1	`task_b.py --noise 0 -n 25`	100	-	-
2	`task_c.py --noise 0 -n 25`	100	-	-
3	`task_c.py -n 25`	100	-	-
7	`noise_plot.py`	100	-	-
9	`task_c.py --noise 0 -n 30 --step 0.01`	100	-	-

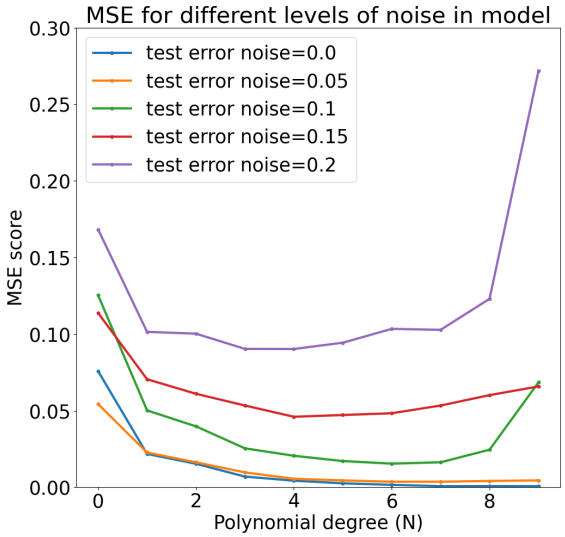

Fig. 7. Test and train MSE for the OLS prediction of Franke function with an increasing amount of noise

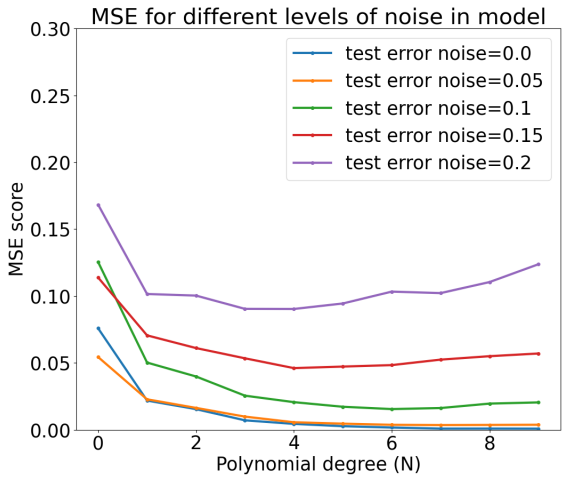

Fig. 8. Test and train MSE for the Ridge prediction with $\lambda = 10^{-8}$ of the Franke function with an increasing amount of noise

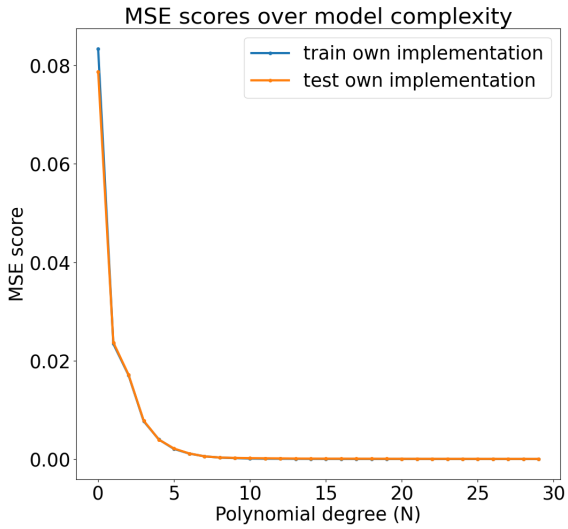

Fig. 9. Franke function fitted up to polynomial degree 30, with 10000 datapoints instead of 400. As we can see, we do not overfit even at this polynomial degree when the availiable data is plentiful. The highest MSE was reached at $N = 29$, and was 0.00003511, one tenth of our best prediction with only 400 datapoints

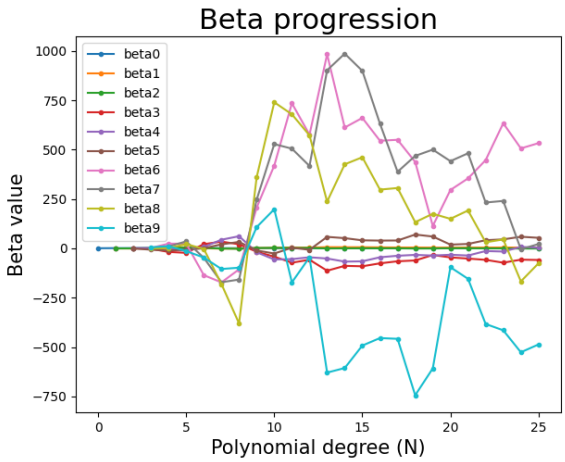

Fig. 10. Beta progression for OLS