Scientific Report for course FYS-STK4155

Project n°1

Linear Regression Methods on Real Topographic data

Sofie Vos, Kristian Bugge, Mathieu Nguyen

Department of Physics
University of Oslo
Oslo, Norway, 2023

Abstract

We studied linear regression methods and resampling techniques in order to solve a linear regression problem on Norwegian geographical data. The linear regression methods, which are Ordinary Least Squares, Ridge and Lasso regression, are implemented first on the 2D Franke Function 3. This Franke Function is a toy function to show and discuss the results and limits of the three linear regression methods.
Afterwards, we have derived and computed the expectation and the variance of the optimal parameter β of a specific method for linear regression, because it is a crucial parameter in order to obtain an accurate model. Then, after implementing and discussing the resampling techniques we concluded that Ridge Regression gave the best results out the three linear regression methods on the Franke Function.

Lastly, we discussed the results of using the same methods and techniques on real topographic data of Norway's landscape. The real data is harder to model properly than the Franke Function, since it gives both more sample points and a randomly generated dataset. It was apparent that Ridge Regression gave the best results. As Norway's topography has quite some noisy datapoints and Ridge regression is less sensitive to noisy data than OLS [7], it is not a surprising result that Ridge regression gives a better fit. Lasso regression gave similar results to Ridge regression, but took more time to find the optimal parameter.

Contents

1 Introduction

Regression Analysis is one of the most widely used techniques for analysing multi-factor data thanks to its broad appeal and usefulness since it uses conceptually logical processes to express the relation between a variable and a set of predictors. [20]
For example, it is used in finance to find how a change in the GDP could affect sales [21] or in medical sciences to explore risk factors when a disease progresses.[25] This is all possible, because regression analysis answers how dependant a variable is on one or multiple predictors to predict future values of a response. So we can discover which predictors are important on a specific data set and topic. [26]
As there are multiple methods for regression analysis, we wonder: Is there one that is better than the others?

Thus the goal of this project is to explore the use of different regression methods as well as their limits and drawbacks. Namely, we will implement the *Ordinary Least Square*, the *Ridge* and the *Lasso* methods to find the most optimal parameters to predict the values of a 2D function called *Franke Function*, a common model used as a **test function in interpolation problems.** [24] The inputs of the created model will represent the surface of a 2D field.
With this, we will discuss which of our three methods is the best suited for our problem by analyzing their respective error between the predicted and real output, with both the *Mean Squared Error* and the R^2 *Error*. Afterwards, we will introduce the bias-variance trade-off of our problem and implement the bootstrap and cross-validation re-sampling techniques in order to further analyze how this affects the MSE and R2 scores of our three methods.
Lastly we will apply all these methods on real topographic data of Norway and analyze if our previous conclusions still maintain themselves on this real situation.

2 Definitions and Notions

2.1 Design Matrix

A design matrix is a matrix of explanatory variables in a set of objects.

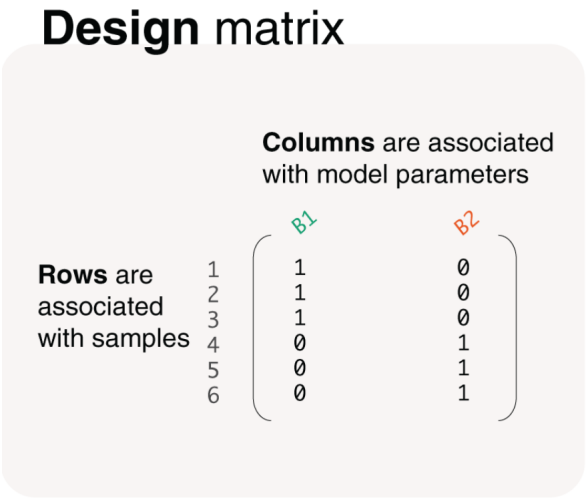

Figure 1: Template of how a Design Matrix is designed.[16]

They are mostly used in linear regression, where we are able to know how the parameters influence a sample's value on a specific parameter, and how impactful each of them are. [28]

2.2 Optimal parameter $\hat{\beta}$

Beta parameters are parameters that are estimated directly in the likelihood function based on the columns of the design matrix and give an estimate of how impactful a column value is compared to all the others [27].
In linear regression, the model is a continuous existing function of the design matrix, containing the inputs/features. This is then multiplied by a vector of beta's and gives us the approximation $\tilde{y}$ such as:

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

The optimal parameter $\hat{\beta}$ is the closest β such as :

$$\tilde{y} = y \tag{2}$$

which can be computed using the **Cost function (or Loss function)**. As we try to fit this function to the real data, we are actually trying to find the optimal vector of beta parameters [13]. We do the same with **Ridge and Lasso regression**, except we introduce a lambda to shrink the contribution of certain inputs in the outcome. [19]

2.3 Cost Function / Loss Function

The Cost Function ($C(\beta)$) is an equation (dependent on β) that allows us to find the **difference between the predicted and the actual value during the training phase**, and is a central element of machine learning since it gives us an **estimate of how accurate our model actually is**. With this, we can compute our optimal parameter $\hat{\beta}$ by minimizing this function. Each model has a different $C(\beta)$ but their use remains the same. [23]

2.4 Mean Squared Error and R^2 Error

The **Mean Squared Error** (or MSE) is a way to **compute the error between predicted and actual value during the test phase**, and show how big the difference is between our model and the actual data points it should fit. We computed it with the following equation.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 \tag{3}$$

The model more accurately predicts the data points as the MSE is small. We use MSE instead of the simpler *relative error*, because **it is differentiable**, and therefore we can simply differentiate this expression to find the $\hat{\beta}$ of our cost function $C(\beta)$. [8]

The R^2 error score function is another statistical metric that will show us how well future samples are likely to be predicted by our model by **indicating how much of the variation of a dependent variable is predictable** from the independent variables in a regression model.
It shows if our model is either overfitting or underfitting by **giving a ratio between 0 and 1**, with 1 being the best [11] [4]. A negative ratio can also be given *when the model cannot fit the data at all*, usually because the model is a poor fit or is a fit for another dataset. [3]

2.5 Random Noise

In real-life modelisation, there is always a small unpredictable variation that we add to simulate randomness in data samples. It is used because variables in real life do not perfectly follow a function, and thus should not be perfectly fitted by a polynomial.
Our model introduces random noise that follows the **Gaussian law**

$$\boxed{\varepsilon \sim N(0, \sigma^2)} \tag{4}$$

Since we assume that the randomness should in average cancel itself out with a large number of samples.
Although it better represents real-life hazardness, it also becomes tougher for our methods to fit an accurate model, and we are gonna study how impactful a small error can be.

2.6 Assumptions for our model

We assumed that there exists a **continuous function** $f(x)$ and a **normal distributed error**, representing the *random noise* 2.5. Thus, both combined can describe our data the following way :

$$\boxed{y = f(x) + \varepsilon} \tag{5}$$

With this, we can now approximate the function $f(x)$ with our model $\tilde{y}$ from the **solution of the linear regression equation**. We approximated $f(x)$ by minimizing $y - \tilde{y}$, thus giving us the following relation:

$$\boxed{\tilde{y} = X\beta} \tag{6}$$

as stated before, with X being the *design matrix*.
This relation describes the model for OLS. For Ridge and Lasso, the relation is similar. But in that case the β parameter also contains the regularization parameter.

2.7 Scaling and Centering Data

In real life scenarios, it is highly likely that some variables will have a much bigger impact than others because of their respective metric units. Indeed without prior data manipulation, a **difference of 400 in caloric intake** will most likely have a bigger impact than a **difference of 5 in age**.
Thus to avoid outliers we **center and scale** our data by subtracting the mean from each input, so that **every variable follows the same metric unit** [9]. We will then see if this helps in the performance of our model.

2.8 Splitting Data

In Machine Learning algorithms, it is common to split our data into both a training and a test set. This way, we can train our model on the training set and then see if it can accurately predict a test set that is most likely following the same function as the training set.
We will be using $\frac{4}{5}$ of the dataset for our training set and $\frac{1}{5}$ for testing. With this we do not need as much data as when $\frac{2}{3}$ is used for the training set, because most of the data will be used to fit our model. [12]

3 Methods implemented

3.1 Ordinary Least Squares

3.1.1 Definition and Use

Ordinary Least Squares (or OLS) is a common technique used to estimate the coefficients of a linear regression equation, by computing the **minimum squared error** between observed and predicted values. We consider for this model that the values are **independents**. The equation of this model is :

$$y = \beta_0 + \sum_{j=1}^{n} \beta_j X_j + \varepsilon \tag{7}$$

where β_0 is the **intercept** of the model, X is the design matrix and ε is the noise error.

If we compute its optimal parameter $\hat{\beta}$ by using the output's function and the cost function, we obtain :

$$\boxed{\hat{\beta} = (X^\top X)^{-1} X^\top Y} \tag{8}$$

It is one of the simplest method for linear regression, however it contains multiple drawbacks. Namely, its $X^\top X$ matrix **may not be invertible** even using Singular Value Decomposition (or SVD), as its **rank must be equal to n+1** [10]. Its result can also be extremely wrong in cases of nonorthogonal problems because of its unbiasedness (resulting in high variance). [17] [18]

3.1.2 Expectation value of y

We will now derive the mean and variance of our linear regression method by **approximating it with our model** $\tilde{y}$ as they give remarkable results that we can use for implementation.

Let $f(x)$ be the **function describing our data** along with its **normal distributed error** ε **such as 4**. If we approximate this function with our model $\tilde{y}$ from the **solution of linear regression equation**, we can then write our *Mean Squared Error (MSE)* :

$$\begin{aligned} y &= f(x) + \varepsilon \\ &\approx \tilde{y} + \varepsilon \\ &\approx X\beta + \varepsilon \qquad \textit{using relation 6} \end{aligned} \tag{9}$$

If we now compute the expectation value, we get :

$$\begin{aligned} E[y] &= E[X\beta + \varepsilon] \\ &= E[X\beta] + E[\varepsilon] \qquad \textit{using expectation properties} \end{aligned} \tag{10}$$

$X\beta$ is a *constant value*, therefore we easily see that $E[X\beta] = X\beta$**.** Moreover, using relation 4, we know that $\mathbf{E}[\varepsilon] = \mathbf{0}$. Thus the relation becomes :

$$\boxed{E[y] = X\beta} \tag{11}$$

If we develop this relation for every single y_i, we get the following result :

$$\boxed{E[y_i] \approx \sum_j X_{ij}\beta_{ij} = X_{i*}\beta} \tag{12}$$

3.1.3 Variance of y

The variance can be computed with the following relation :

$$\begin{aligned} var(y_i) &= E[y_i^2] - E[y_i]^2 \\ &= E[(X_{i*}\beta + \varepsilon_i)^2] - X\beta \qquad \textit{using 9 and 12} \end{aligned} \tag{13}$$

If we focus on the first term, we can expand it with its power property :

$$\begin{aligned} E[(X_{i*}\beta + \varepsilon_i)^2] &= E[(X_{i*}\beta)^2 + 2\varepsilon_i X_{i*}\beta + \varepsilon_i^2] \\ &= E[(X_{i*}\beta)^2] + E[2\varepsilon_i X_{i*}\beta] + E[\varepsilon_i^2] \\ &= (X_{i*}\beta)^2 + 2X_{i*}\beta\cancel{E[\varepsilon_i]} + E[\varepsilon_i^2] \qquad \textit{since } \varepsilon \sim N(0,\sigma^2) \\ &= (X_{i*}\beta)^2 + E[\varepsilon_i^2] \end{aligned} \tag{14}$$

Thus giving us the result :

$$\begin{aligned} var(y_i) &= \cancel{(X_{i*}\beta)^2} + E[\varepsilon_i^2] - \cancel{(X_{i*}\beta)^2} \\ &= E[\varepsilon_i^2] \end{aligned} \tag{15}$$

Using the fact that $var(\varepsilon_i) = E[\varepsilon_i^2) - E[\varepsilon_i]^2$ and that $E[\varepsilon_i]^2 = 0$, we get the final result :

$$\boxed{var(y_i) = var(\varepsilon_i) = \sigma^2} \tag{16}$$

3.1.4 Expectation of $\hat{\beta}$

Since we are studying OLS, we can write $\hat{\beta}$ as the following relation :

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y \tag{17}$$

The expectation value is thus :

$$
\begin{aligned}
E[\hat{\beta}] &= E[(X^\top X)^{-1} X^\top Y] \\
&= (X^\top X)^{-1} X^\top E[Y] \\
&= \cancel{(X^\top X)^{-1} X^\top X}\beta \qquad \textit{given result 11}
\end{aligned} \tag{18}
$$

Which results in :

$$
\boxed{E[\hat{\beta}] = \beta} \tag{19}
$$

3.1.5 Variance of $\hat{\beta}$

We can compute the variance with this time another relation :

$$
\begin{aligned}
var(\hat{\beta}) &= E[(\hat{\beta} - E[\hat{\beta}])^2] \\
&= E[(\hat{\beta} - E[\hat{\beta}])(\hat{\beta} - E[\hat{\beta}])^\top] \\
&= E[((X^\top X)^{-1} X^\top Y - E[\hat{\beta}])((X^\top X)^{-1} X^\top Y - E[\hat{\beta}])^\top] \qquad \textit{using relation 8} \\
&= E[((X^\top X)^{-1} X^\top Y - \beta])((X^\top X)^{-1} X^\top Y - \beta)^\top] \qquad \textit{using relation 19} \\
&= (X^\top X)^{-1} X^\top E[YY^\top] X (X^\top X)^{-1} - E[\beta\beta^\top] \\
&= (X^\top X)^{-1} X^\top E[(X\beta)(X\beta)^\top] X (X^\top X)^{-1} - \beta\beta^\top \qquad \textit{using relation 6} \\
&= (X^\top X)^{-1} X^\top (X\beta\beta^\top X^\top + \sigma^2) X (X^\top X)^{-1} - \beta\beta^\top \\
&= \cancel{(X^\top X)^{-1} X^\top X}\beta\beta^\top \cancel{X^\top X (X^\top X)^{-1}} + (X^\top X)^{-1} X^\top \sigma^2 X (X^\top X)^{-1} - \beta\beta^\top \\
&= \cancel{\beta\beta^\top} + \cancel{X^\top X^{-1} X^\top X}\sigma^2 (X^\top X)^{-1} - \cancel{\beta\beta^\top}
\end{aligned} \tag{20}
$$

The final result becomes :

$$
\boxed{var(\hat{\beta}) = \sigma^2 (X^\top X)^{-1}} \tag{21}
$$

3.2 Ridge Regression Method

One main problem of the OLS as mentioned previously 3.1 is that you cannot always compute $\hat{\beta}$, as the matrix $(X^\top X)$ may not be invertible. The Ridge method overcomes that issue by **introducing a parameter** λ in its cost function, becoming :

$$C(X, \beta) = \frac{1}{n}||y - X\beta||_2^2 + \lambda||\beta||_2^2 \tag{22}$$

This way, we ensure invertibility, as well as introduce a major reduction in variance and allow some small bias compared to OLS. [18]
This lambda is a part of Ridge's optimal parameter $\hat{\beta}$ $\hat{\beta}$ which can now be computed with the following equation :

$$\boxed{\hat{\beta} = (X^\top X - \lambda I)^{-1} X^\top Y} \tag{23}$$

3.3 Lasso Method

The *Least Absolute Shrinkage and Selection Operator* (or Lasso) Method is another machine learning method that try to find a balance between model simplicity and accuracy. It follows the same idea and principle than Ridge's Method 3.2, but its cost function is :

$$C(X, \beta) = \frac{1}{n}||y - X\beta||_2^2 + \lambda||\beta||_1 \tag{24}$$

The main use of this λ in both Ridge and Lasso is to impose a constraint on the model's parameters, shrinking the regression coefficients towards zero. The main difference of Lasso with Ridge, is that the coefficients can become zero. This is useful for **feature selection**, as the variables with coefficients close to zero or equal to zero will be effectively removed from the model. [15]

3.4 Cross Validation

Cross-validation is a statistical method used to **estimate the skill of machine learning models**. It is one of the methods used as a **resampling technique**, a set of methods to either repeat sampling from a given population, or a way to estimate the precision of a statistic [5]. Cross-validation is a way to validate a predictive model, **by dividing our data into unique subsets**. A part of subsets is removed to be used as a validating set, while the remaining data is used to form a training set to predict the validation set. This process is repeated multiple times, typically using different partitions of the data, to assess the model's performance under various conditions and ensure that the results are not overly dependent on a specific data split.[1]

3.5 Bootstrapping

Bootstrapping is a **non-parametric approach** used in statistics and data science in order to obtain asymptotic results while minimizing computation efforts and estimating the accuracy of a model. We sample our original dataset by dividing it in multiple subsets. One or more subset will be used as test data, and the rest as training data, and we proceed as usual. We iterate this process many times, making sure that we use **different test data every iteration** and compute the **mean of each results**, thus obtaining an average on how is the model supposed to behave. [22]

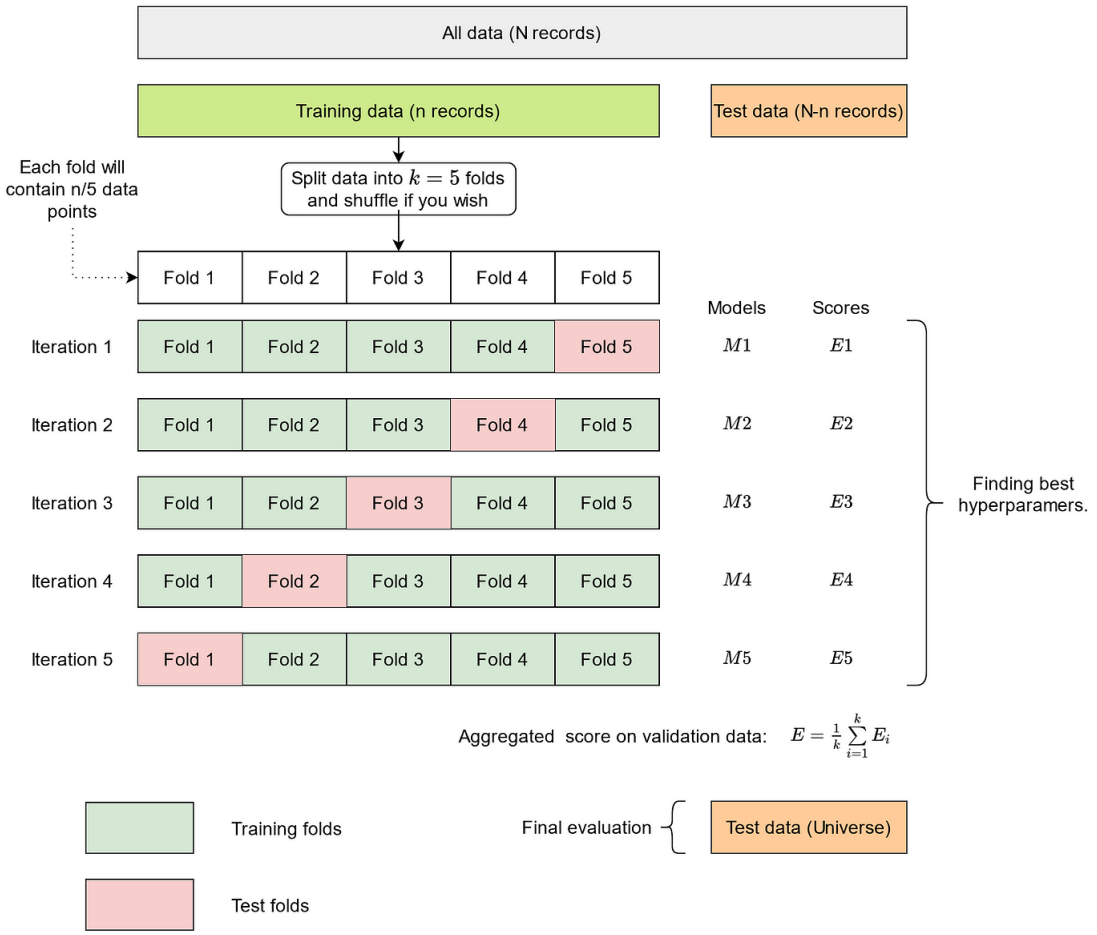

Figure 2: An illustration on how Bootstrapping works. Every iteration the original dataset is splitted into subsets and a portion of them will be used as test data. It is necessary that the test data is different everytime, in order to simulate a different dataset everytime [14]

4 Code Implementation and Results on the Franke Function

Now that we introduced each method and technique and how they work, we will now implement code that uses said methods and compare then on different situations.
We will also discuss the importance of **scaling data** as well as **the effect of random noise** on our input values, as we theoretically concluded that they may have a big impact on our results. We will in the first place study **Franke's Function**, then re-apply those same codes for real topographic data of Norway.
In this section, we will also make sure that our polynomial model goes to a **degree up to 5**, unless mentioned.

4.1 Franke's Function

Franke's function is a simple 2D function that is widely used for interpolation problems. It models two Gaussian peaks of different heights, as well as a smaller dip and is defined by the following equation : [24]

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

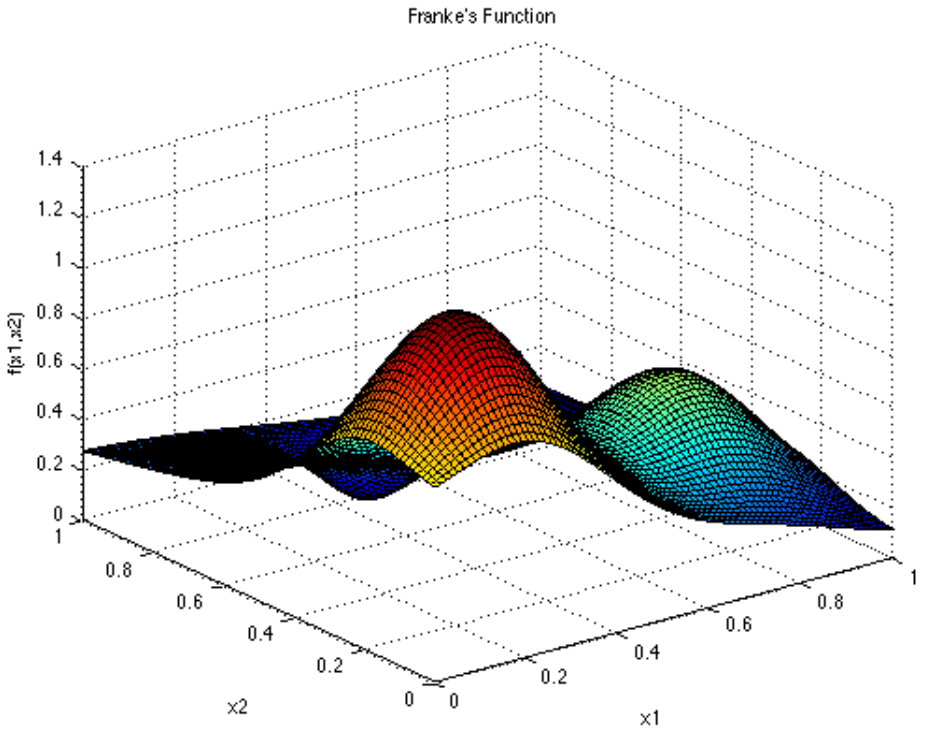

Figure 3: 3D plot of Franke's Function

In this project, we are mostly gonna use Franke's Function as a basic example to code our methods and error computation, as it is in 2D. We will then re-use the same codes to study real topographic data which is also in 2D, but is this time way bigger.

4.2 Random Noise implementation

We stated in 2.7 that scaling data helps us getting better results. Therefore we need to implement noise so that we can clearly observe said theory. We defined the noise to be following a **Gaussian distribution** 2.5, therefore to introduce this notion into Franke's Function, we added a function that modify its output by adding a random value following the corresponding distribution.
But first, we have to decide how much noise to use for the Franke's function, such that we get **more of a landscape instead of smooth hills** :

We implemeted the following code :

```
z = FrankeFunction(x, y)  + np.random.normal(0, 0.15, size=datapoints)
```

To add into our Franke's Function a noise equivalent to the **Gaussian distribution** $\sim N(0, 0.15)$, resulting in this dataset. z is the **array that we wish to add noise to** (here Franke's Function), and the second parameter is how impactful we want the **variance of the noise** to be (with a default value but we can change it anytime).
It then returns our modified output z with noise, giving us the following data :

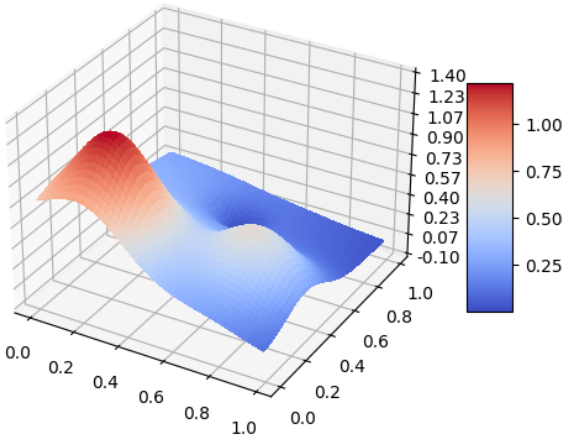

Figure 4: Plot of Franke's Function without noise

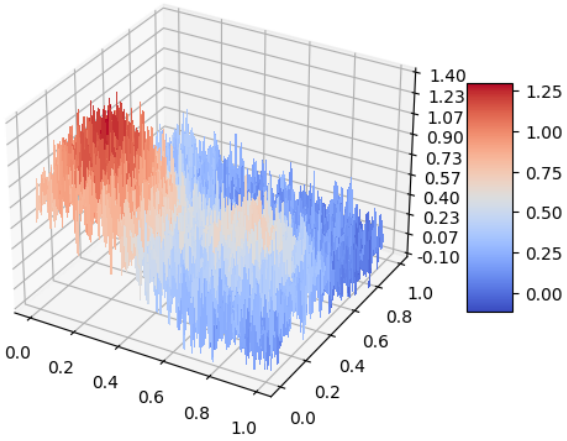

Figure 5: Plot of Franke's Function with noise

This data will thus be the data that we will try to fit with our models when studying noise.

4.3 Ordinary Least Square

We will use ordinary least squares regression (OLS) to find a suitable model for Franke's function, first with the usage of the library *Scikit-Learn*, then we will introduce our own code to compare.

4.3.1 Finding the optimal numbers of datapoints

Our first objective in this part is to find a number of datapoints so that the **test error doesn't become too big**, otherwise it would mean that our model is not able to properly fit our data input (either by underfitting or overfitting). In general, we tend to fit better results when we use a lot of datapoints, hence we will study about 3000 datapoints at first.

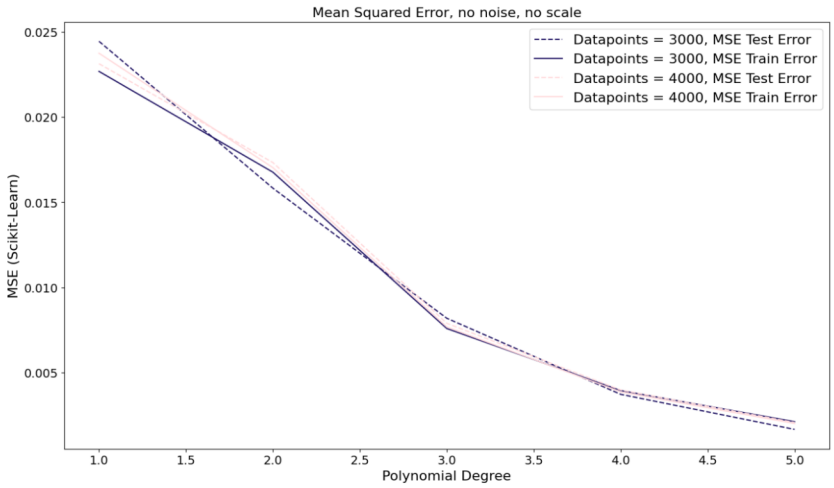

Figure 6: Mean Squared Error for both 3000 and 4000 data points

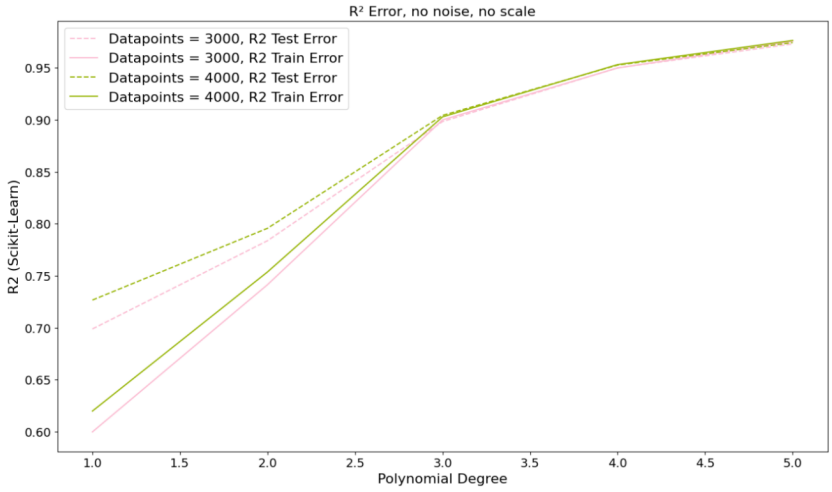

Figure 7: R2 for both 3000 and 4000 data points

Even though there is a 1000 data points difference, the results are very similar (Figures 6 and 7) because these results **overlap well**. because we rather use less than more data points to reduce computation time, we will use **3000 data points for fitting our model**.

The reason why 3000 data points sometimes give better results than 4000 (see degree 2 Figures 6 and 7) is because, even though we have more data, there might be **more outliers**. It depends on the *quality of the data*. We can

also notice that sometimes the *training error is worse than the test error*. This means that the polynomial degree is too low resulting in **underfitting**, which can be expected given our high number of datapoints. We will thus study that same model and look for a more appropriate polynomial degree.

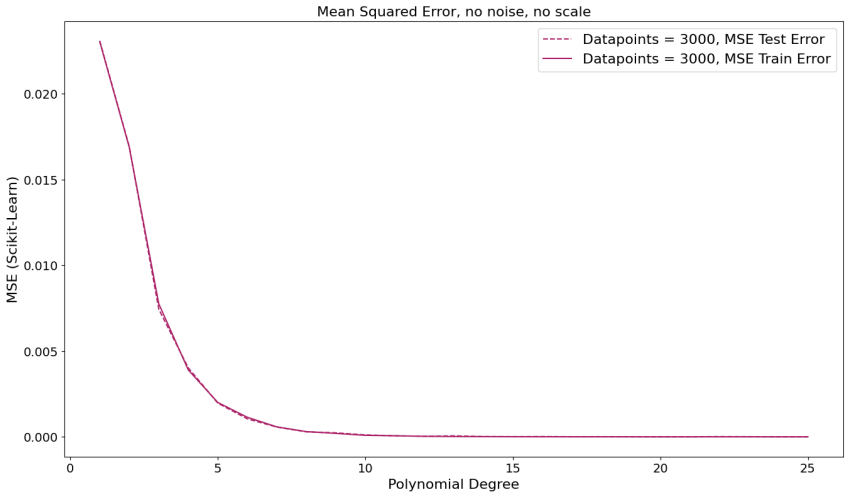

Figure 8: Mean Squared Error for a polynomial degree up to 25, for 3000 datapoints

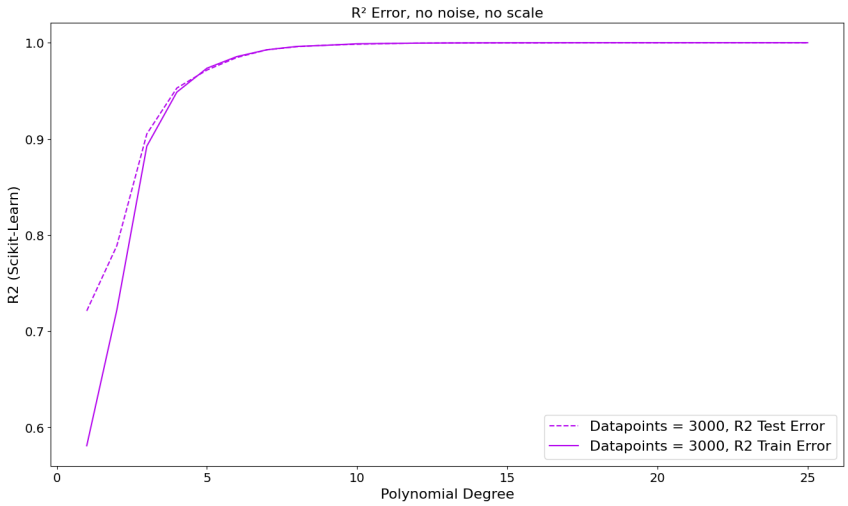

Figure 9: R2 for a polynomial degree up to 25, for 3000 datapoints

The best polynomial degree to fit the data tends to be the highest degree as more data points are used, thus taking a higher complexity to perfectly fit a lot of points (before it starts overfitting). For 3000 data points, even though the error scores slowly converges to a certain value, **a polynomial degree about above 25** seems to be best (Figures 8).

We will show the same thing for 600 data points. Since we have now less datapoints, we should expected the best polynomial degree to be lower than for 3000 data points:

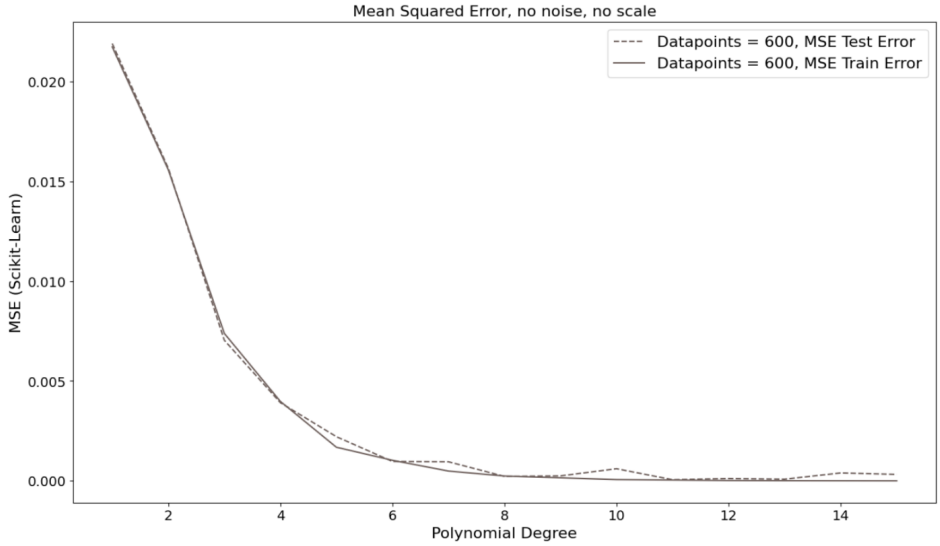

Figure 10: Mean Squared Error for a polynomial degree up to 15, with 600 datapoints

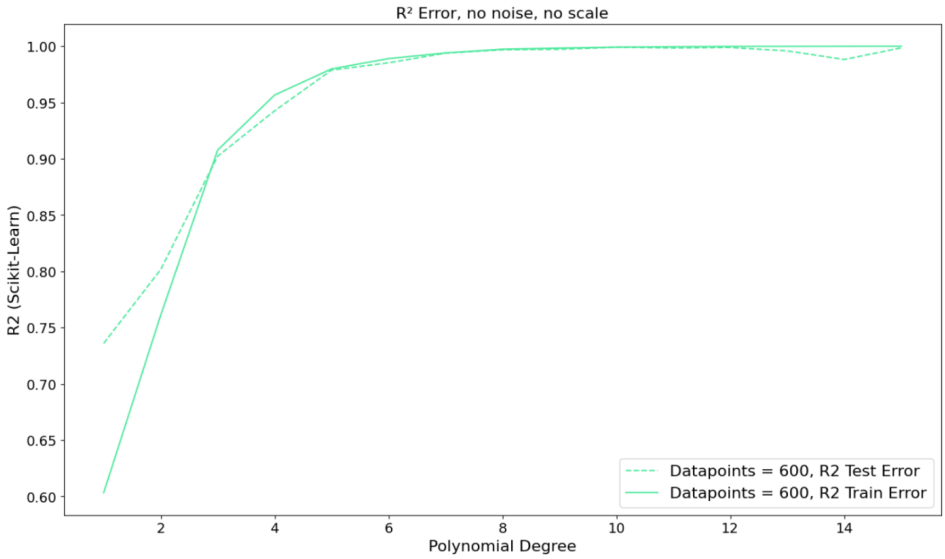

Figure 11: R2 for a polynomial degree up to 15, with 600 datapoints

The best polynomial degree when fitting a model on 600 data points is around 12, due to a **high R2 score** (Figure 11) and a **dipping test MSE** (Figure 10). And as we thought, this optimal degree 12 is a lot lower than the degree for the dipping point for 3000 data points (which was above 25).

4.3.2 Studying the importance of scaling

We are gonna introduce the OLS as defined earlier in 3.1, at first without scaling, and then we will introduce scaling and compare both plots to see if it actually has an impact.

The non-scaled implementation for OLS goes by the following code :

```
model = LinearRegression(fit_intercept=False) # OLS
```

The scaled implementation for OLS follows a *similar code*, only adding a few additional lines to scale data before running the same computation.

```
model = LinearRegression(fit_intercept=False) # OLS
clf = model.fit(X_train, z_train)
z_fit = clf.predict(X_train)
z_pred = clf.predict(X_test)
```

If we now compare both computation on the same problem and input datas (we will use 600 datapoints to avoid too large computation), we obtain the next plots :

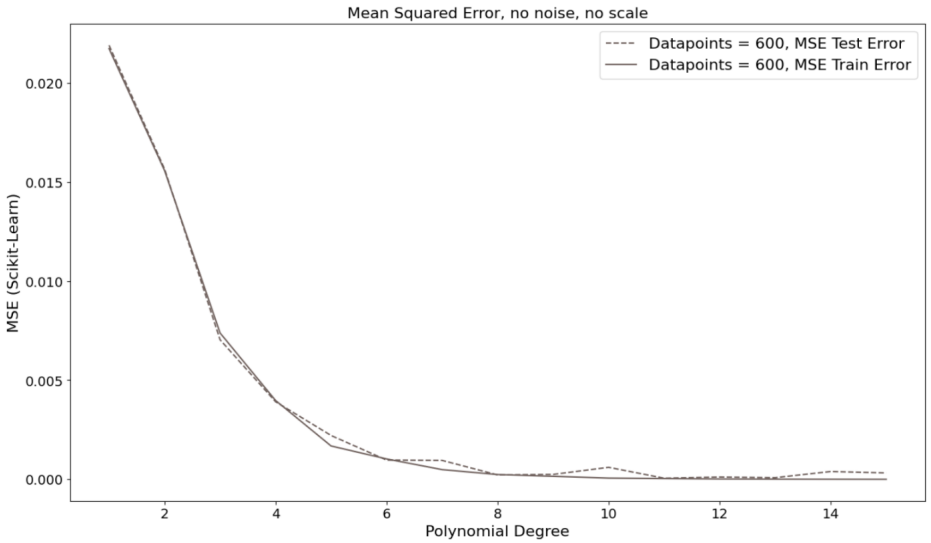

Figure 12: Mean Squared Error on the non-scaled inputs

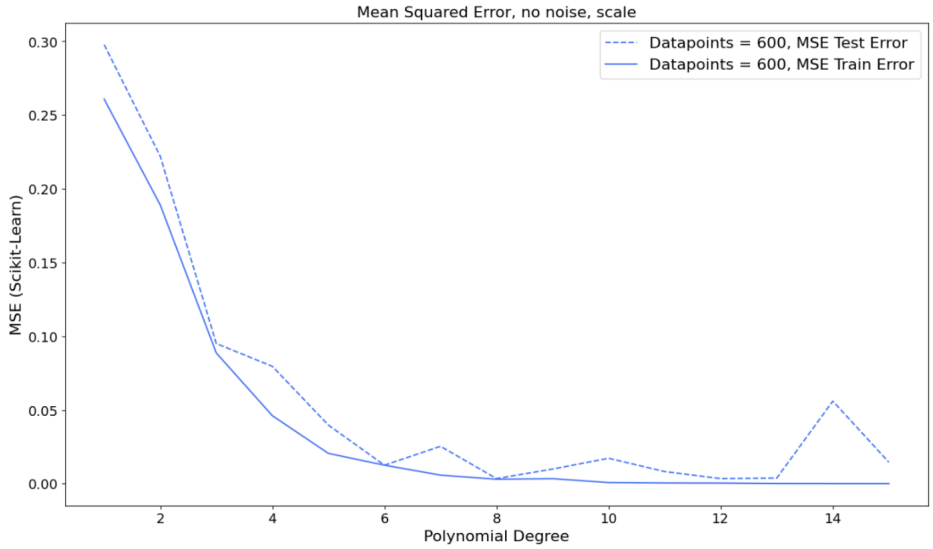

Figure 13: Mean Squared Error on the scaled inputs

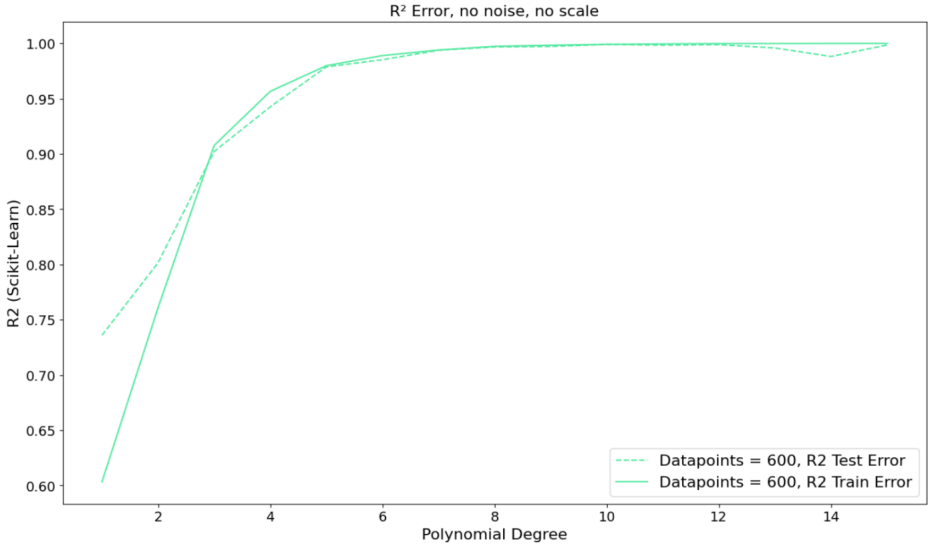

Figure 14: R^2 Error on the non-scaled inputs

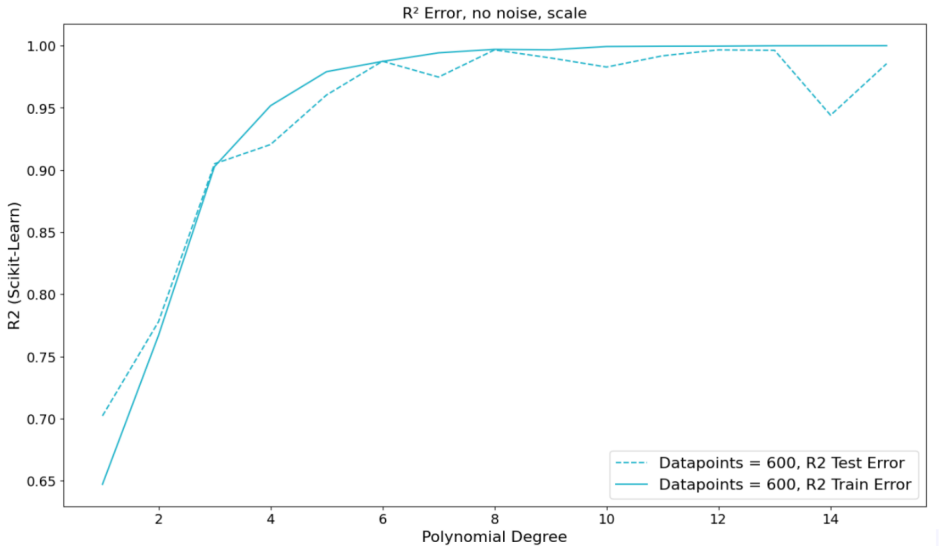

Figure 15: R^2 Error on the scaled inputs

The scaling **does not make the results better**. Indeed, The test MSE's (Figures 12 and 13) and test R2 (Figures 14 and 15) scores got worse. This is probably due to the **scaling avoiding outliers** in the test data. The training scores follow a similar function, possibly because of the values from Franke's function being between 0 and a little bit above 1, as well as x and y following the same probability law, therefore *already following the same scale in the first place.*
Hence, unlike what we expected when defining data scaling 2.7, we conclude here that **the modifications it produce are negligible**.

4.3.3 Studying the importance of noise

We will now introduce the same implementation as the scaled OLS, but this time with the addition of random noise 2.5, and see how impactful it really is.
Since we assumed that scaling does not improve the results (2.7). we will try fitting a model when there is noise *but no scaling*.
We used 3000 data points, as it gave good results for OLS without noise. By introducing the noise using 4.2, we get as a result :

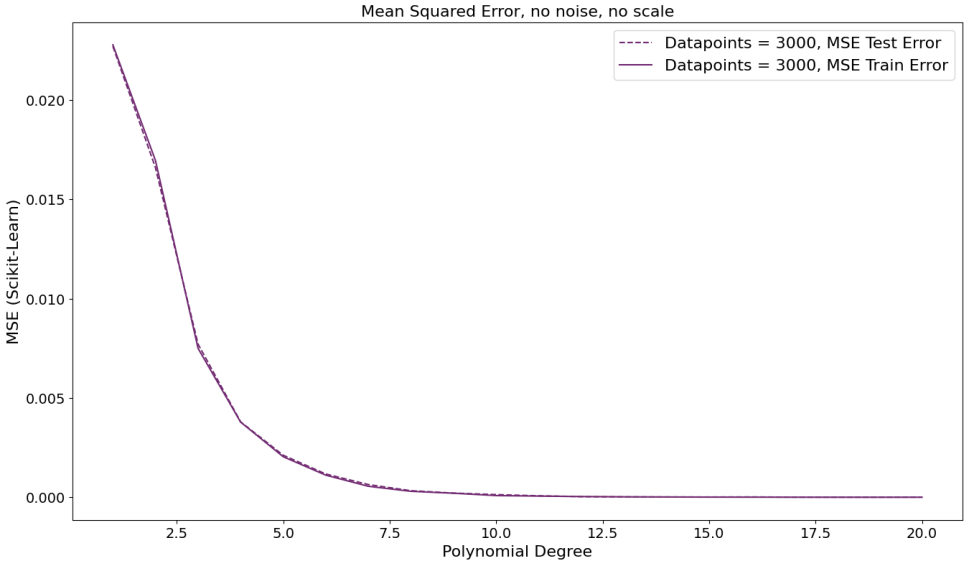

Figure 16: Mean Squared Error on the data without noise

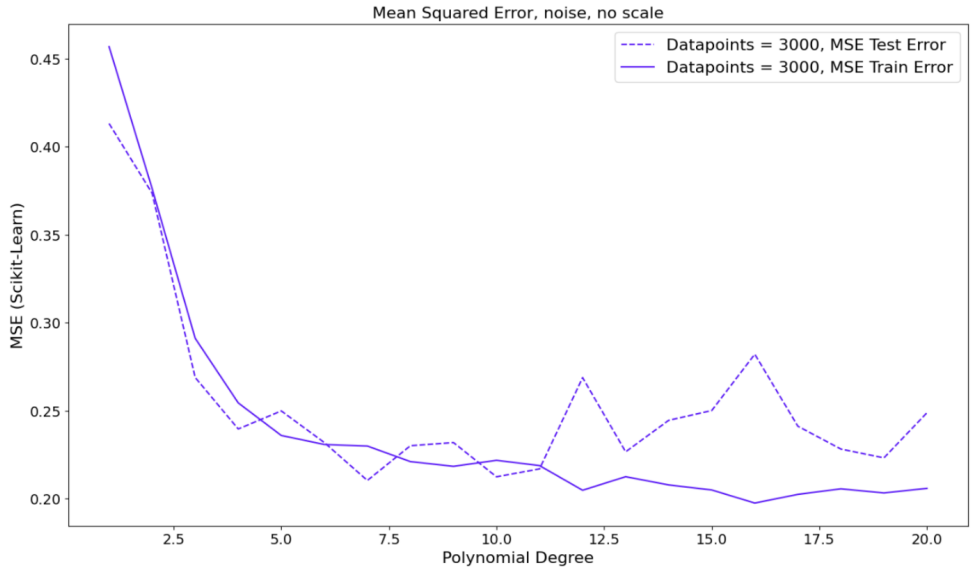

Figure 17: Mean Squared Error on the data with noise

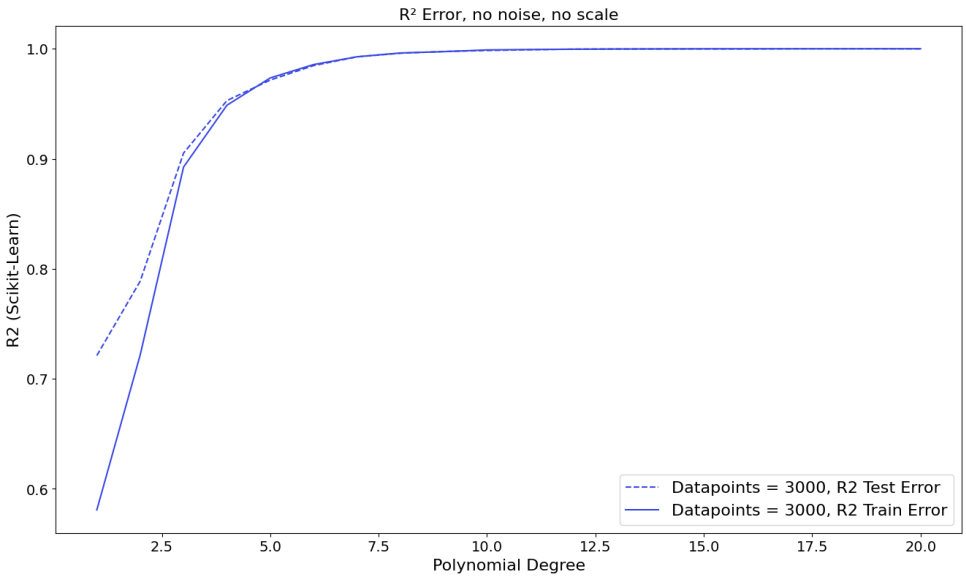

Figure 18: R^2 Error on the data without noise

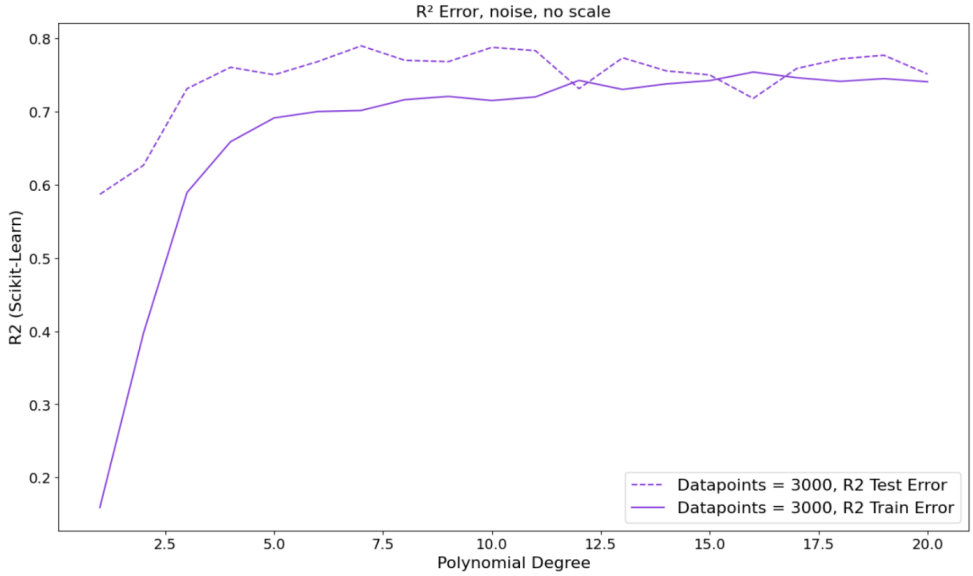

Figure 19: R^2 Error on the data with noise

Because of the added noise, **the MSE is overall higher and R2 is lower** (Figures 16 with 17, 18 with 19). We also find that a model fits **best at a degree of 11**, as the error *starts increasing passing that point.* This is lower than the optimal degree above 25 that we discussed without the noise. This means that when noise is added and the complexity increases, the model overfits quicker.

4.3.4 Implement our own Ordinary Least Squares

Instead of using *Scikit-Learn* library to compute OLS, we can also implement a code doing the same computation by following its definition 3.1. The implemented code is thus :

```
beta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ z_train
z_fit = X_train @ beta
z_predict = X_test @ beta
```

And if we compare both handmade and Scikit-Learn code, we get the following :

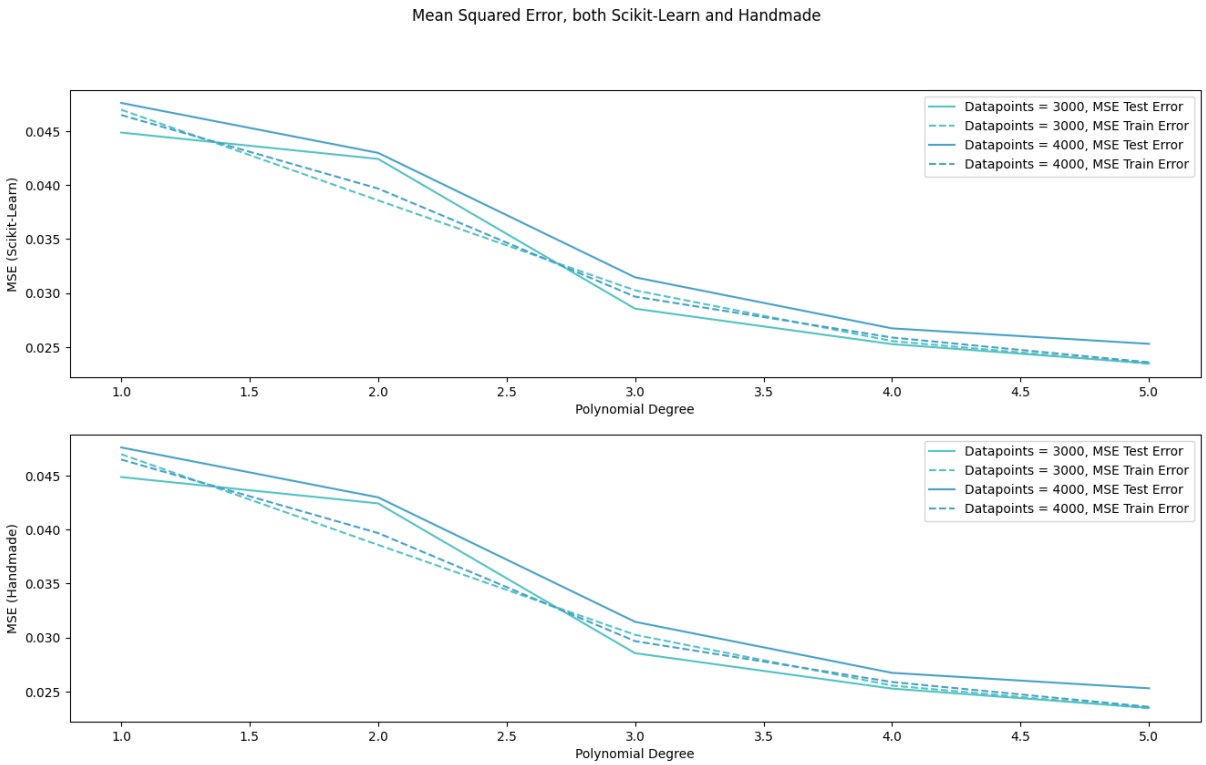

Figure 20: A comparison of both handmade and Scikit-Learn version of OLS's MSE, for 3000 and 4000 datapoints

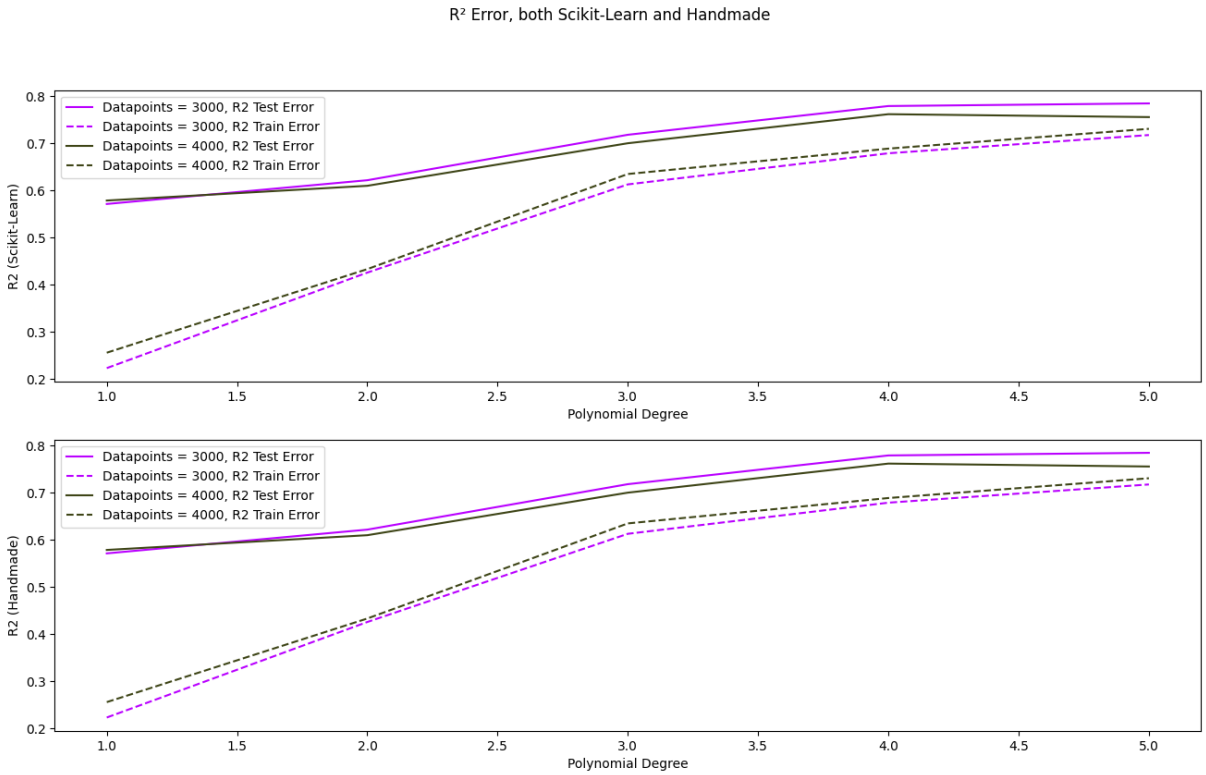

Figure 21: A comparison of both handmade and Scikit-Learn version of OLS's R2, for 3000 and 4000 datapoints

We can then conclude by stating that *Scikit-Learn*'s implementation follows the same computation as we did.

4.4 Ridge Regression Method

We will now implement the Ridge Regression Method 3.2 on Franke's Function, introducing alongside its λ value. Most importantly, because λ can make a big difference to our output compared to OLS, we want to study multiple λ values and discuss on how much impact it has.

4.4.1 Normal Implementation

Our implementation to do so is :

```
[...]
for lamb in lambdas:
    model = Ridge(alpha = lamb, fit_intercept=True)
    clf = model.fit(X_train, z_train)
    z_fit = clf.predict(X_train)
    z_pred = clf.predict(X_test)
[...]
```

Where *lambdas* are all the λ studied during this Ridge implementation. We took 600 datapoints and **100 equally distributed λ values between 10^{-6} and 10** so that we both have very small and quite big values.

We do not include the intercept for this regression, because it might be too large and have an **impact that is too big on the overall equation**, thus resulting in a poor MSE.

We first want to study λ on a dataset *without noise* first, as we mainly want to discuss the impact of our λ parameter, rather than the minimum error. We want to see how impactful the value of λ is on different degrees of our model.
In order to not overfill our plot with too many lines, we are only studying at **polynomial degrees of around 12** as we seen beforehand that **they produced the best error scores**. We thus plot the following graphs :

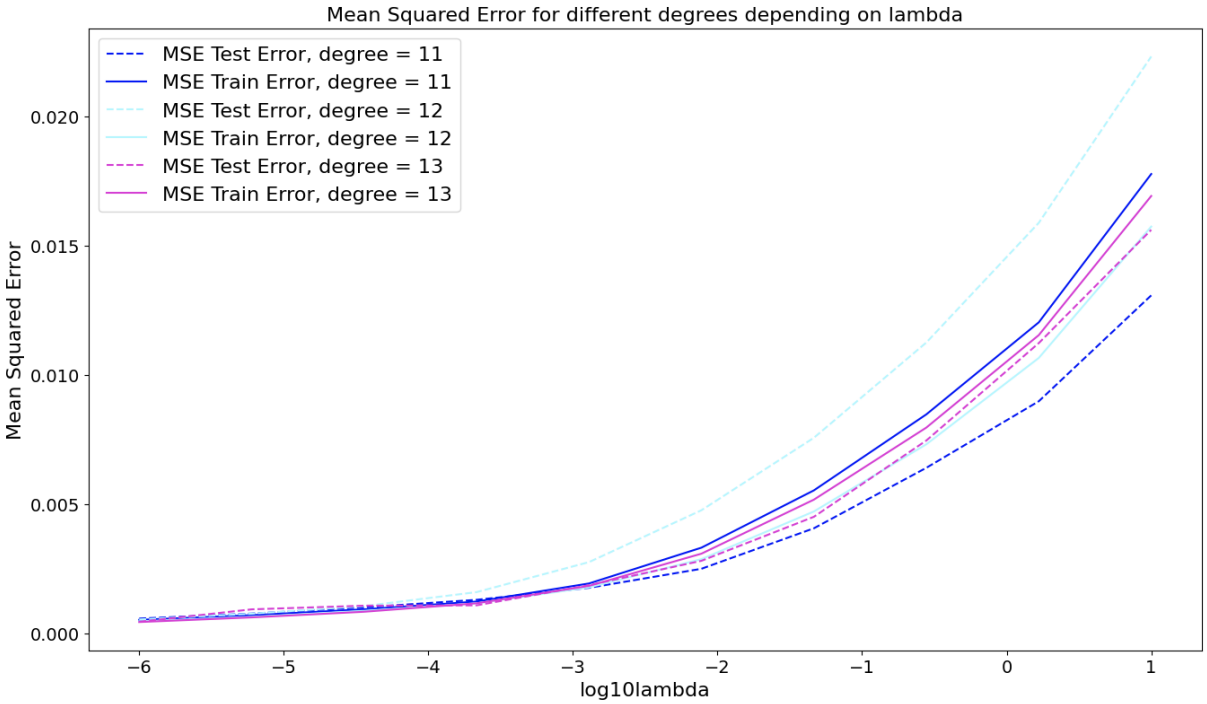

Figure 22: Mean Squared Error of multiple degrees of our model given λ

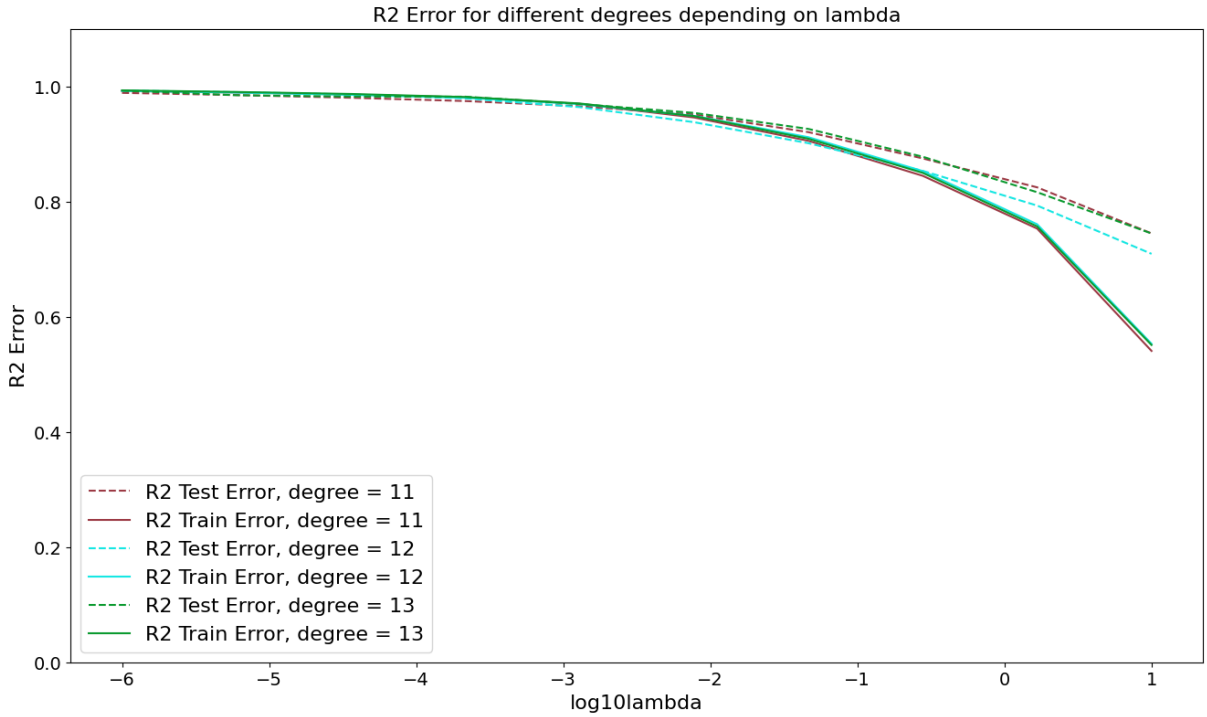

Figure 23: R^2 Error of multiple degrees of our model given λ

From Figures 22 and 23, we notice that the closer λ is to 0 the better are the results. Also, when comparing the degrees, we see that the best degree to use for a model out of 600 data points is around degree 12, as it produces the best test error out of all three.

If we then look at the same method with *noise implemented* with 4.2, we get the following plots :

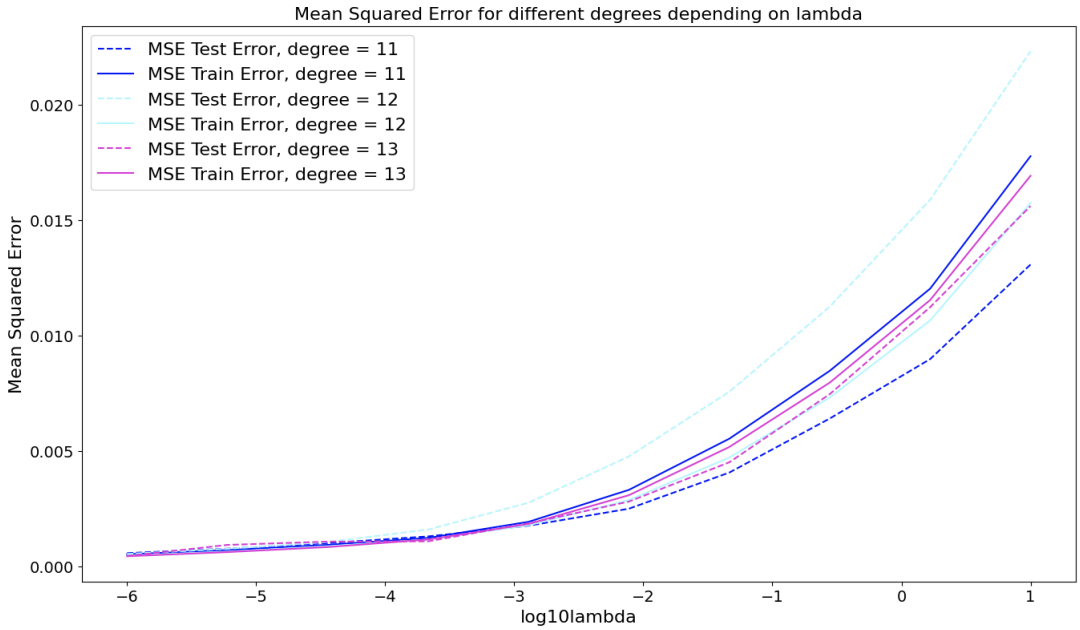

Figure 24: Mean Squared Error on the data without noise

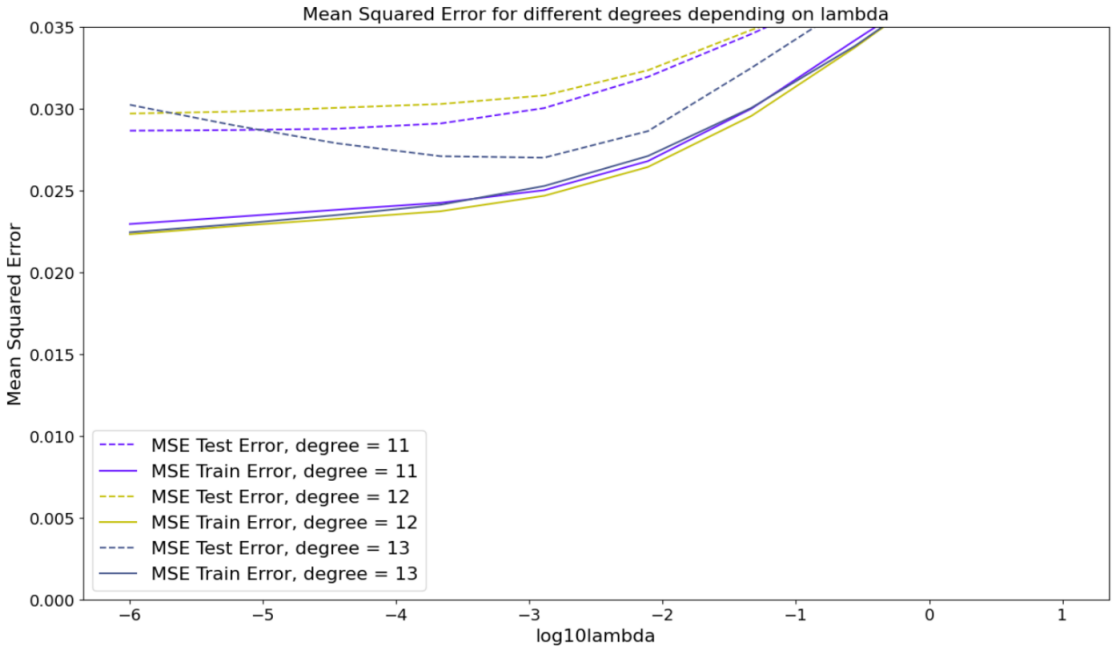

Figure 25: Mean Squared Error on the data with noise

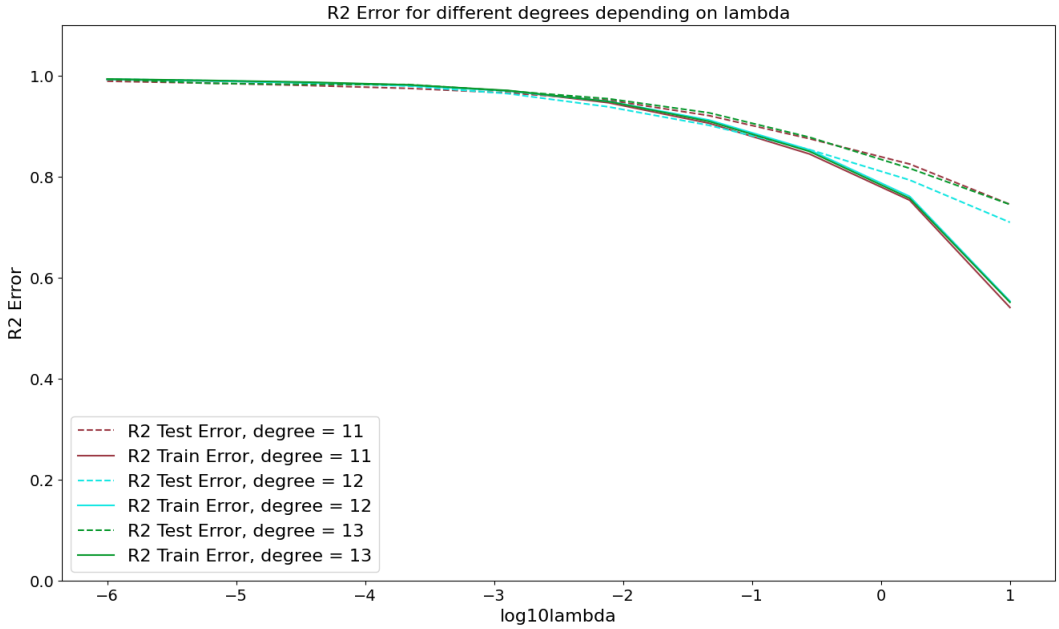

Figure 26: R^2 Error on the data without noise

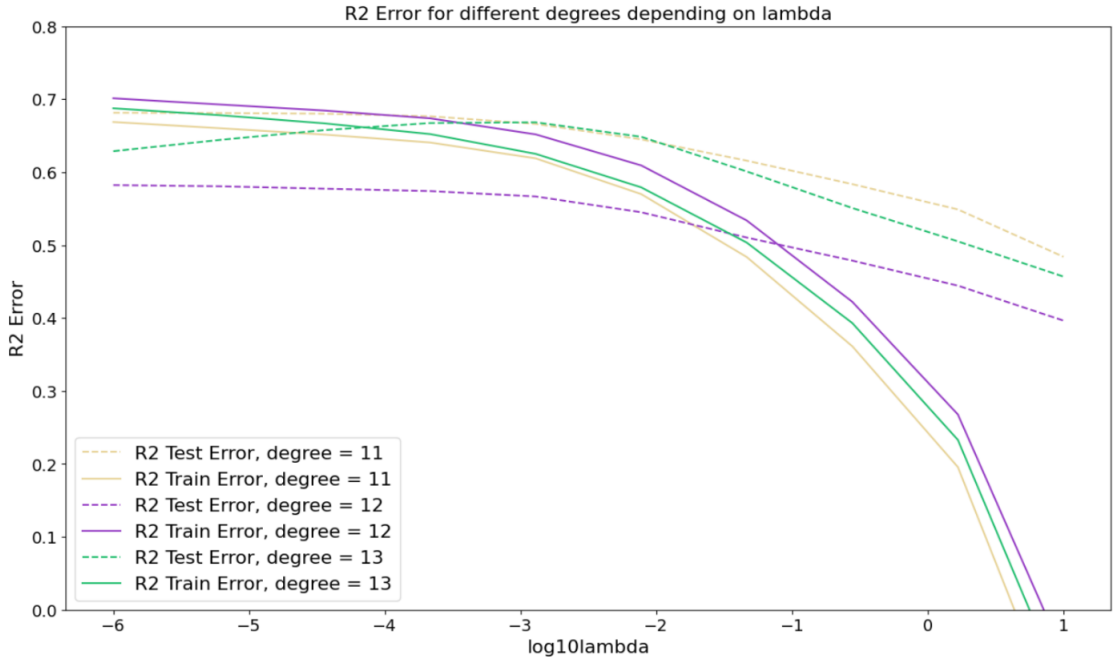

Figure 27: R^2 Error on the data with noise

With noise, a λ close to 0 is still the best to use. The overall MSE and R2 scores in the plots with noise compared to the graphs without noise (Figures 24 with 25, 26 with 27) have however **worsened**. As the noise decreases the quality of the data, these errors hence get worse, which was to be expected. Also, the only degree out of the three, 11, 12 and 13, that has **no R2 test error below its train error when lambda goes to zero is degree 11**. Thus, a model of degree 11 would be best to use, since we observed earlier that a test error lower than train error **indicates underfitting** [2].

4.4.2 Introducing Scaled Data

If we compare the results we get using Ridge Regression on a *scaled dataset*, we obtain the following results :

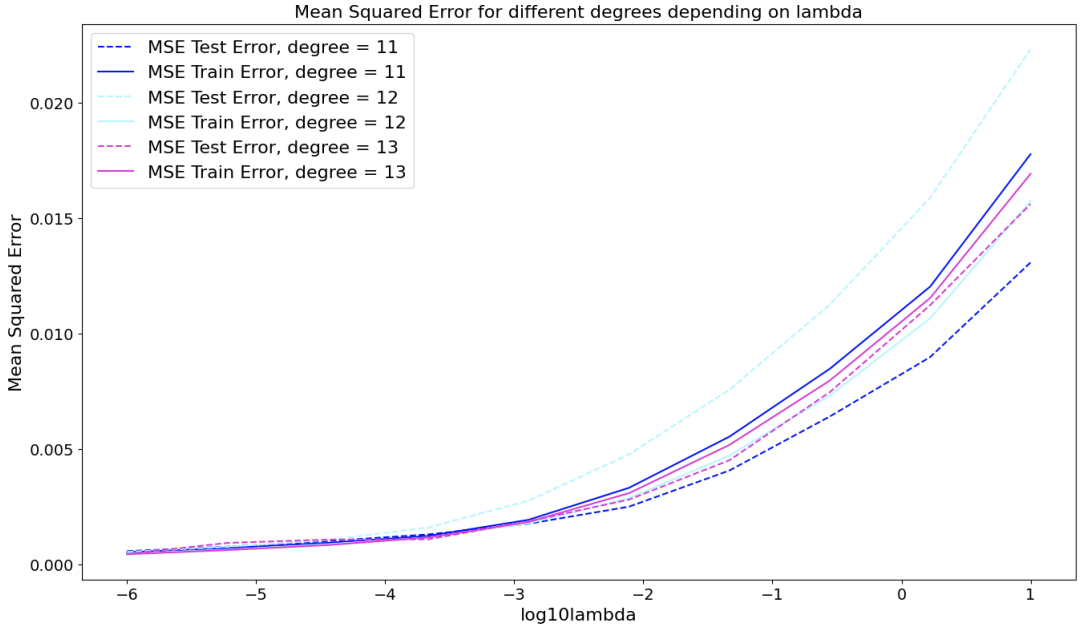

Figure 28: MSE Error on the data without scaling

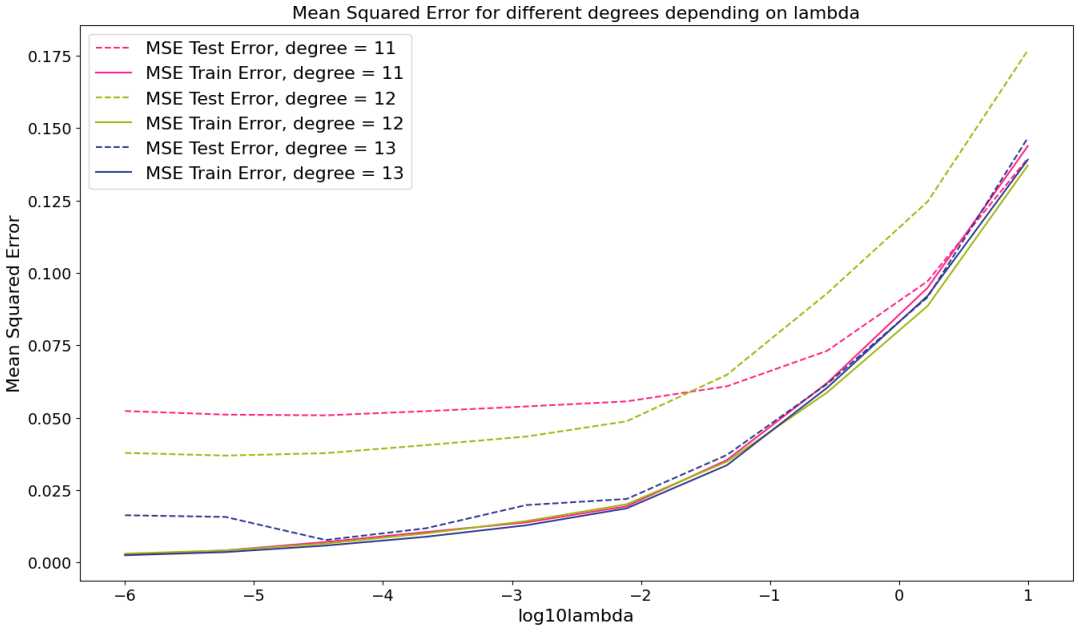

Figure 29: MSE Error on the data with scaling

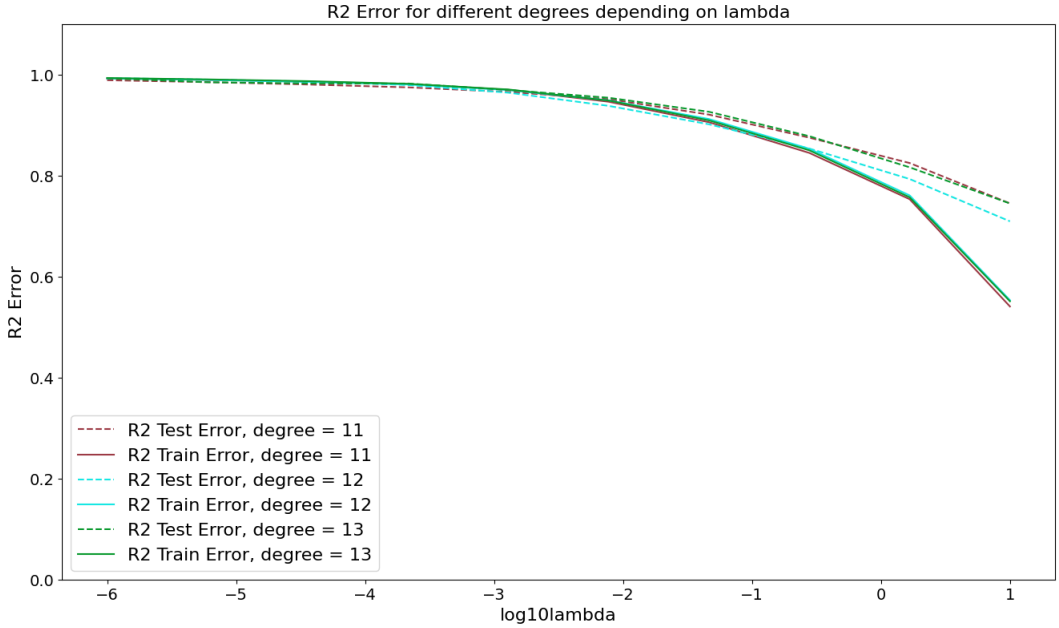

Figure 30: R^2 Error on the data without scaling

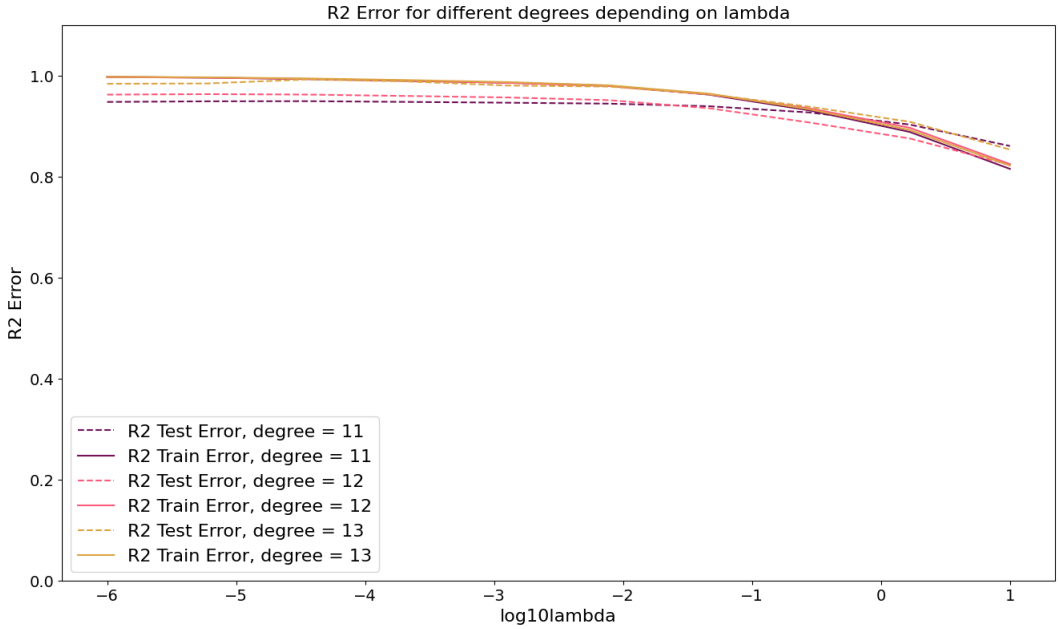

Figure 31: R^2 Error on the data with scaling

It seems that, unlike with OLS, scaling the data does have utility. Indeed, although the MSE score got slightly worse in Figure 29, we can however realise that **the R^2 error got way better** in Figure 30 as it is way closer to 1. Therefore, we can conclude that scaling the data for Ridge Regression gives it a little more error, but **is less tend to overfit**.

4.4.3 Handmade Implementation

The handmade implementation uses the following code :

```
 I = np.eye(X_train.shape[1])
RidgeModel = np.linalg.inv(X_train.T @ X_train+lamb*I) @ X_train.T @ z_train
z_fit = X_train @ RidgeModel
z_pred = X_test @ RidgeModel
```

and gives the same output as *Scikit-Learn*'s implementation :

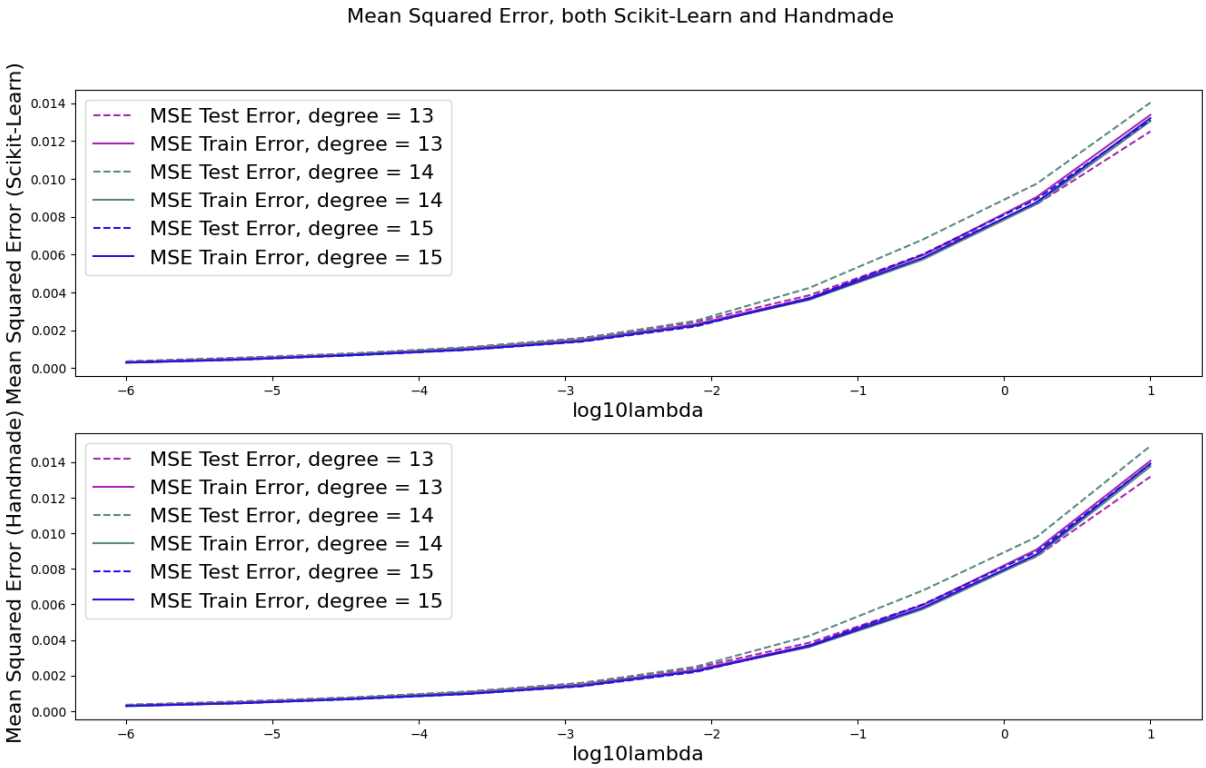

Figure 32: A comparison of both handmade and Scikit-Learn version of Ridge's MSE, for 3000 and 4000 datapoints

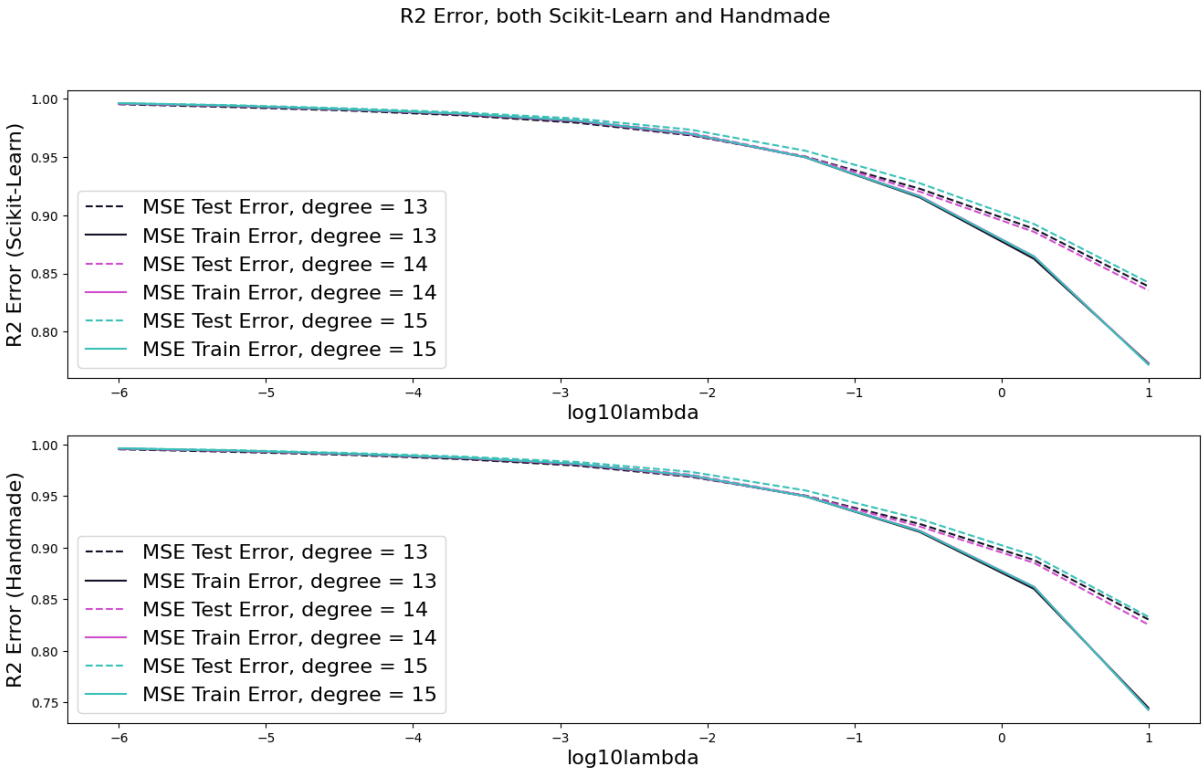

Figure 33: A comparison of both handmade and Scikit-Learn version of Ridge's R2, for 3000 and 4000 datapoints

4.5 Lasso Method

We will now introduce the Lasso Regression Method 3.3 on Franke's Function, with the same λ implementation as for Ridge 3.2.
Our implementation to do so is :

```
[...]
for lamb in lambdas:
    model = Lasso(alpha = lamb, fit_intercept=True)
    clf = model.fit(X_train, z_train)
    z_fit = clf.predict(X_train)
    z_pred = clf.predict(X_test)
[...]
```

Following the same λ distribution and assumption than for Ridge Regression. Again, we study Franke's Function without noise and with 600 datapoints, resulting in the following error scores :

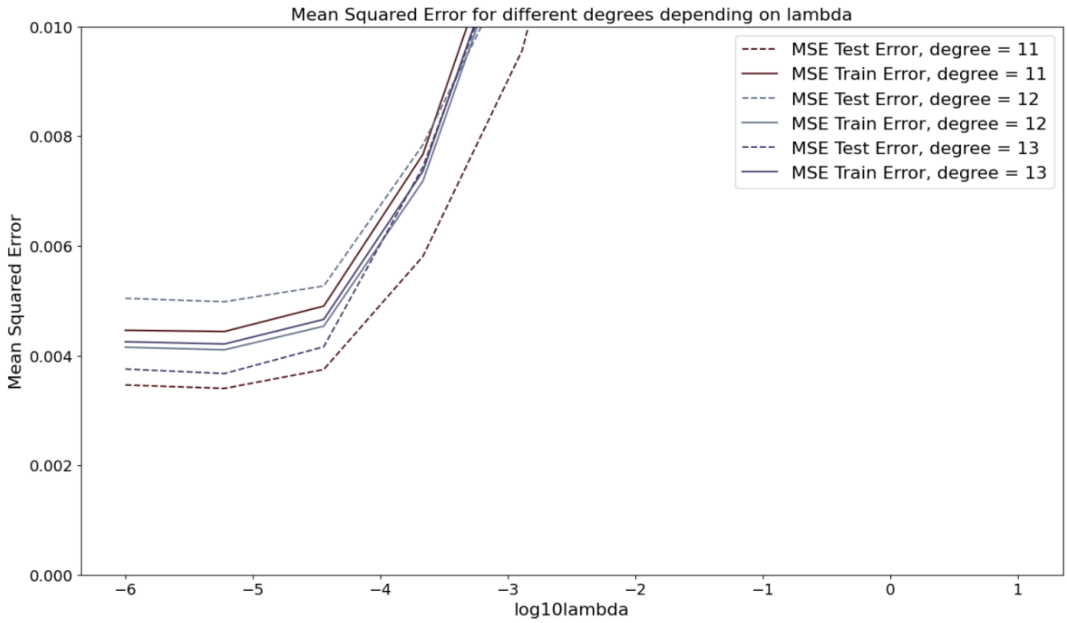

Figure 34: Mean Squared Error of multiple degrees of our model given λ

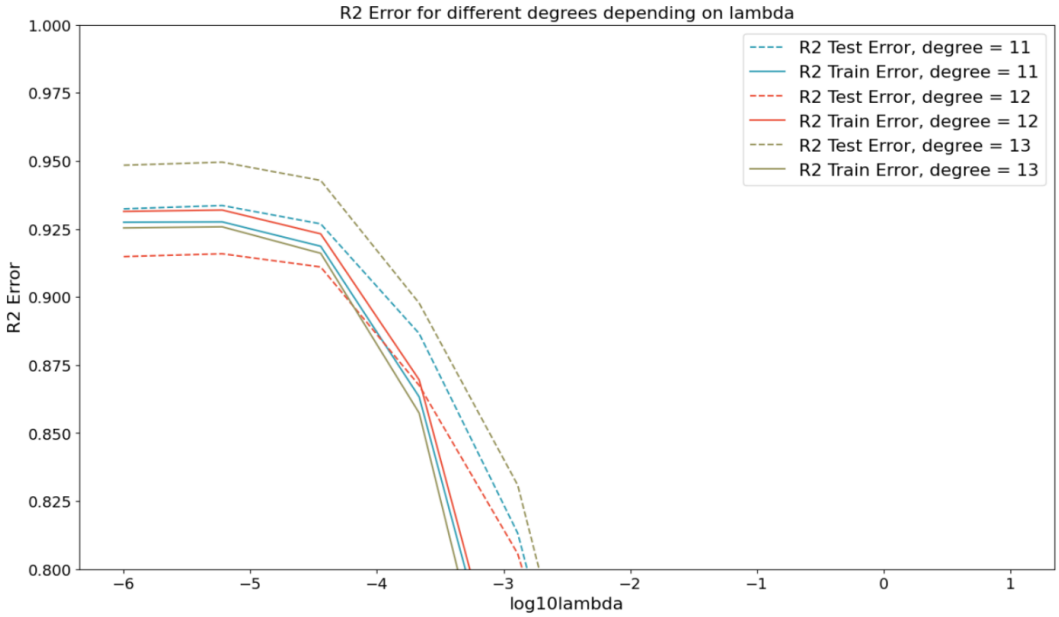

Figure 35: R^2 Error of multiple degrees of our model given λ

Similar to Ridge regression, we see that a lambda close to 0 and a polynomial degree of 12 probably fits best, as it is the only degree of the three that has **the test error worse than the train error**, for the same argument mentioned in Ridge Regression [2].

Now observing Franke's Function with noise, we have the following plots :

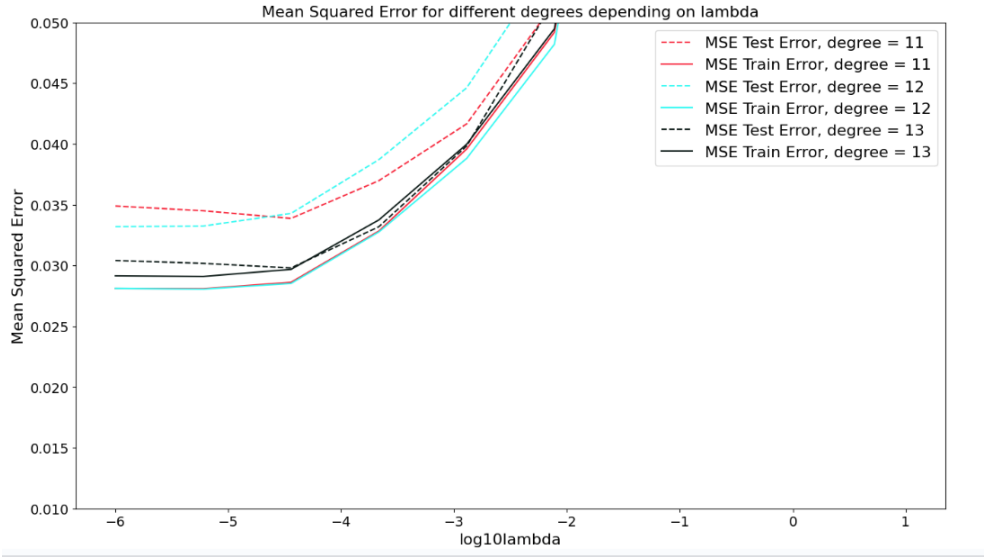

Figure 36: Mean Squared Error of multiple degrees of our model given λ with noise

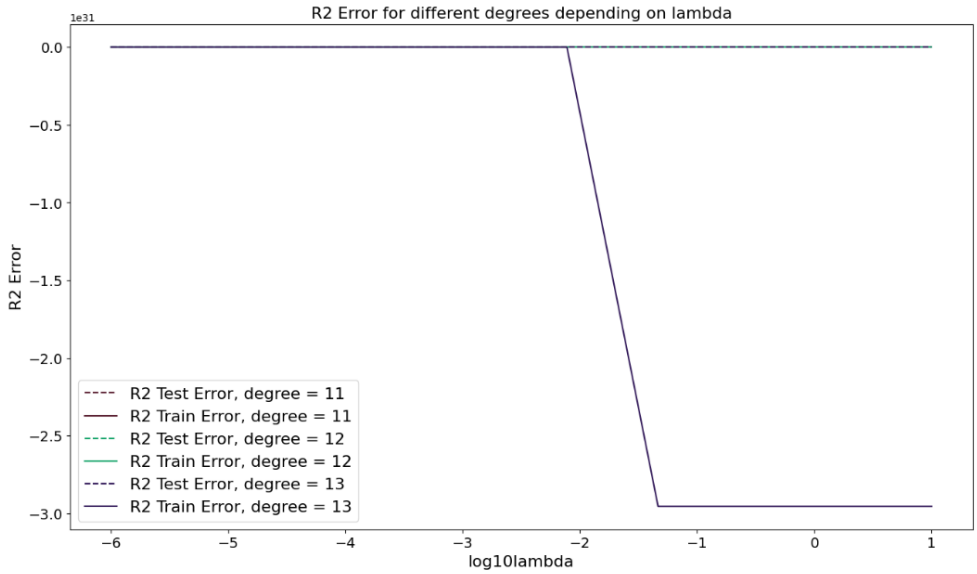

Figure 37: R^2 Error of multiple degrees of our model given λ with noise

Here however, we can clearly see on figure 37 a state of **overfitting**, as the R^2 error is exploding negatively (to an order of 1e31). This is also visible on the MSE figure 36 with all the functions increasing out of border, thus reaching enormous numbers. It becomes clear that **Lasso Regression is not able to converge to proper values on our dataset with noise.**

4.6 Exploiting Bias-Variance trade-off for Resampling Technique

Our goal in this section is to study the **bias-variance trade-off** of the Ordinary Least Square. Most importantly, we will showcase its utility and importance for many Machine Learning algorithms, or classification problems.

4.6.1 Definitions and Interpretation

The bias represents the error caused by the **simplifying assumptions** built into our method. A high bias might indicate that the model lacks **important relations** between the input data and the predicted outputs

The variance represents the error due to the noise caused by the data set that we're studying. A high variance may indicate that the model is trying too hard to model the random noise of the input data rather than the predicted outputs and thus **overfits**.

The bias-variance trade-off is, as its name suggest, a **middle ground to adopt** between the bias and the variance values, as there is no way to minimise both of them. Its goal is to find the most optimal bias and variance values so that **its sum is minimal**.

Thus, we need to find a model that is **complex enough to lower both the bias and the variance**, but it should **not be too complex** so that the variance value does not reach too high values because of overfitting.

The bias-variance trade-off can also be illustrated with the following graph :

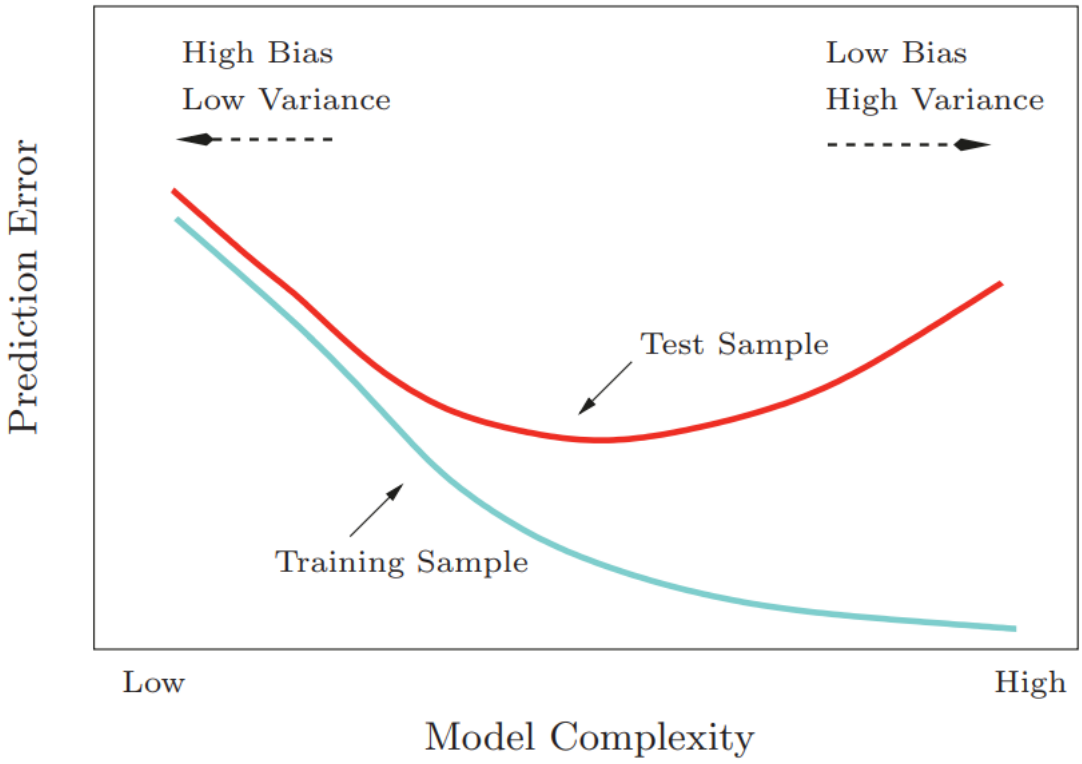

Figure 38: Test and training error as a function of model complexity. [6]

And this is something that we can illustrate with our own dataset as well. Indeed, if we implement the train and test error given the complexity of our model :

```
for samples in range(trials):
    X_train, X_test, z_train, z_tests = train_test_split(X, z, test_size=0.2)
    zpred = model.predict(X_train_scaled)
    ztilde = model.predict(X_test_scaled)
    testerror[polydegree] += mean_squared_error(z_tests, ztilde)
    trainingerror[polydegree] += mean_squared_error(z_train, zpred)
```

We obtain the following plot :

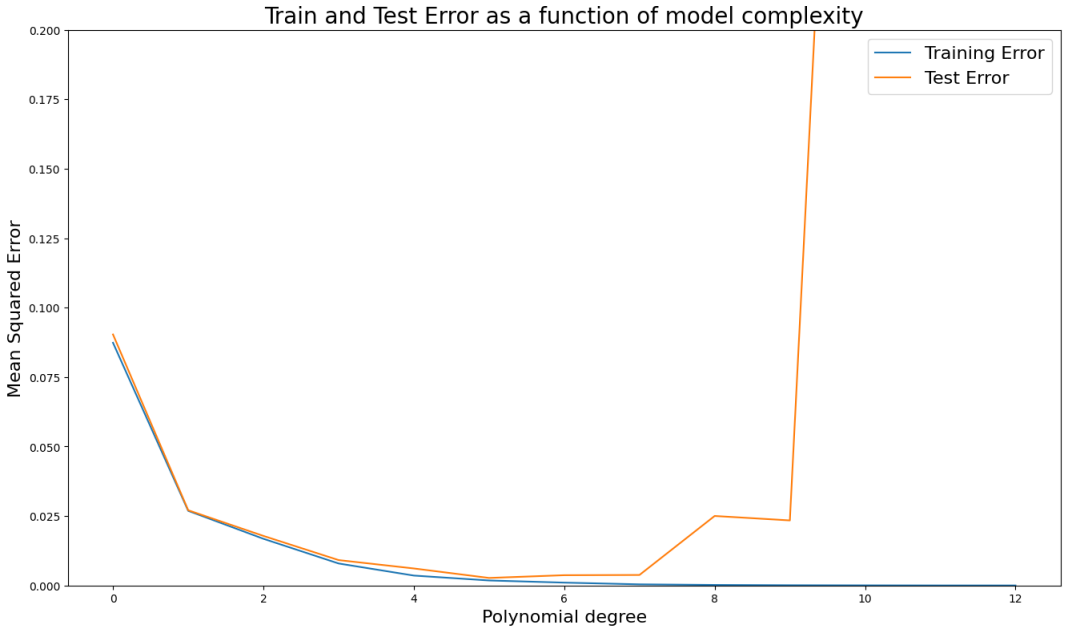

Figure 39: An illustration of the Bias-Variance trade-off on our own input data. It doesn't look perfectly like the theoretical graph 38, but the main idea is here : when the complexity is high enough, the test error explode exponentially

4.6.2 Bias-Variance Trade-Off Analysis

Here, we will derive how can we optimise the **Mean Squared Error** value, using both the bias and the variance.

We consider a dataset $\mathcal{L}$ consisting of the data $X_{\mathcal{L}} = (y_j, x_j), j = 0, ..., n-1$. We also assume that the true data is generated with the same noisy model as 5 and that our model follows the least square approximation defined in 6

We recall that the parameter β is found by **optimizing the cost error of the MSE** :

$$\boxed{C(X, \beta) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = E[(y-)^2]} \tag{25}$$

We will now rewrite $E[(y-)^2]$ with the bias and the variance terms.

If we develop that expectation value, we get :

$$\begin{aligned} E[(y-)^2] &= E[y_i^2 - 2y_i\tilde{y}_i +^2] \\ &= E[y_i^2] - 2E[y_i\tilde{y}_i] + E[^2] \end{aligned} \tag{26}$$

$\rightarrow$ We can rewrite the first term with the variance with the relation :

$$var(\tilde{y}) = E[\tilde{y}^2] - E[\tilde{y}]^2$$
$$\iff E[\tilde{y}^2] = var(\tilde{y}) + E[\tilde{y}]^2 \tag{27}$$

→ Since we assumed that our data is generated following the model 5, the second term is therefore :

$$\begin{aligned} E[y\tilde{y}] &= E[(f(x) + \varepsilon)\tilde{y}] \\ &= E[f\tilde{y}] + E[\varepsilon\tilde{y}] \\ &= fE[\tilde{y}] + E[\varepsilon]E[\tilde{y}] \end{aligned} \tag{28}$$

We assumed the noise to be following the distribution $\varepsilon \sim N(0, \sigma^2)$, therefore

$$E[\varepsilon] = 0 \tag{29}$$

and so does $E[\varepsilon]E[\tilde{y}]$. The second thus results in :

$$E[y\tilde{y}] = fE[\tilde{y}] \tag{30}$$

→ The third term can be computed by using the function of the model 5 :

$$\begin{aligned} E[y^2] &= E[(f(x) + \varepsilon)^2] \\ &= E[f(x)^2 + 2f(x)\varepsilon + \varepsilon^2] \\ &= E[f(x)^2] + 2E[f(x)\varepsilon] + E[\varepsilon^2] \\ &= E[f(x)^2] + \cancel{2f(x)E[\varepsilon]} + E[\varepsilon^2] \qquad \textit{using relation 29} \\ &= E[f(x)^2] + E[\varepsilon^2] \end{aligned} \tag{31}$$

Using the relation between variance and expectation, we know that :

$$\begin{aligned} var(\varepsilon) &= E[\varepsilon^2] - E[\varepsilon]^2 \\ \iff E[\varepsilon^2] &= var(\varepsilon) + E[\varepsilon]^2 \\ &= \sigma^2 \qquad \textit{because } \varepsilon \sim N(0, \sigma^2) \end{aligned} \tag{32}$$

Therefore, the third term becomes :

$$E[y^2] = E[f(x)^2] + \sigma^2$$
$$= f(x)^2 + \sigma^2 \tag{33}$$

Hence, if we group all three terms back together, the cost function becomes :

$$E[(y - \tilde{y})^2] = f(x)^2 + \sigma^2 - 2f(x)E[\tilde{y}] + var(\tilde{y}) + E[\tilde{y}]^2$$
$$= f(x)^2 - 2f(x)E[\tilde{y}] + E[\tilde{y}]^2 + \sigma^2 + var(\tilde{y})$$
$$= (f(x) - E[\tilde{y}])^2 + \sigma^2 + var(\tilde{y}) \tag{34}$$
$$= \frac{1}{n}\sum_{i=0}^{n-1}(f_i - E[\tilde{y}])^2 + \sigma^2 + \frac{1}{n}\sum_i(\tilde{y_i} - E[\tilde{y}])^2$$

With the relation 4, we can assume that, when calculating the mean of f, the mean of the noise error follows this equation :

$$\boxed{\frac{1}{n}\sum_{i=0}^{n-1}\varepsilon = 0} \tag{35}$$

Thus, we can write :

$$\frac{1}{n}\sum_{i=0}^{n-1}(f_i - E[\tilde{y}])^2 = \frac{1}{n}\sum_{i=0}^{n-1}(f_i + \varepsilon - E[\tilde{y}])^2$$
$$= \frac{1}{n}\sum_{i=0}^{n-1}(y_i - E[\tilde{y}])^2 \tag{36}$$

Therefore, it gives us :

$$E[(y - \tilde{y})^2] = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - E[\tilde{y}])^2 + \sigma^2 + \frac{1}{n}\sum_i(\tilde{y_i} - E[\tilde{y}])^2$$
$$= (y - E[\tilde{y}])^2 + \sigma + \frac{1}{n}\sum_i(\tilde{y_i} - E[\tilde{y}])^2 \tag{37}$$

Hence the final result :

$$\boxed{\begin{aligned} E[(y-\tilde{y})^2] &= (Bias[\tilde{y}])^2 + var(\tilde{y}) + \sigma^2 \\ (Bias[\tilde{y}])^2 &= (y - E[\tilde{y}])^2 \\ var(\tilde{y}) &= \frac{1}{n}\sum_i (\tilde{y}_i - E[\tilde{y}])^2 \end{aligned}} \tag{38}$$

Here, σ^2 corresponds to the **error variance** caused by the noise error ε of the model.

We implement this notion with the OLS model with the code :

```
    model = make_pipeline(LinearRegression(fit_intercept=False))
z_pred = np.empty((z_test.shape[0], n_boostraps))
for i in range(n_boostraps):
    x_, y_ = resample(X_train_scaled, z_train)
    z_pred[:, i] = model.fit(x_, y_).predict(X_test_scaled).ravel()
polydegree[degree] = degree
error[degree] = np.mean( np.mean((z_test - z_pred)**2, axis=1, keepdims=True) )
bias[degree] = np.mean( (z_test - np.mean(z_pred, axis=1, keepdims=True))**2 )
variance[degree] = np.mean( np.var(z_pred, axis=1, keepdims=True) )
```

and we get the plot result :

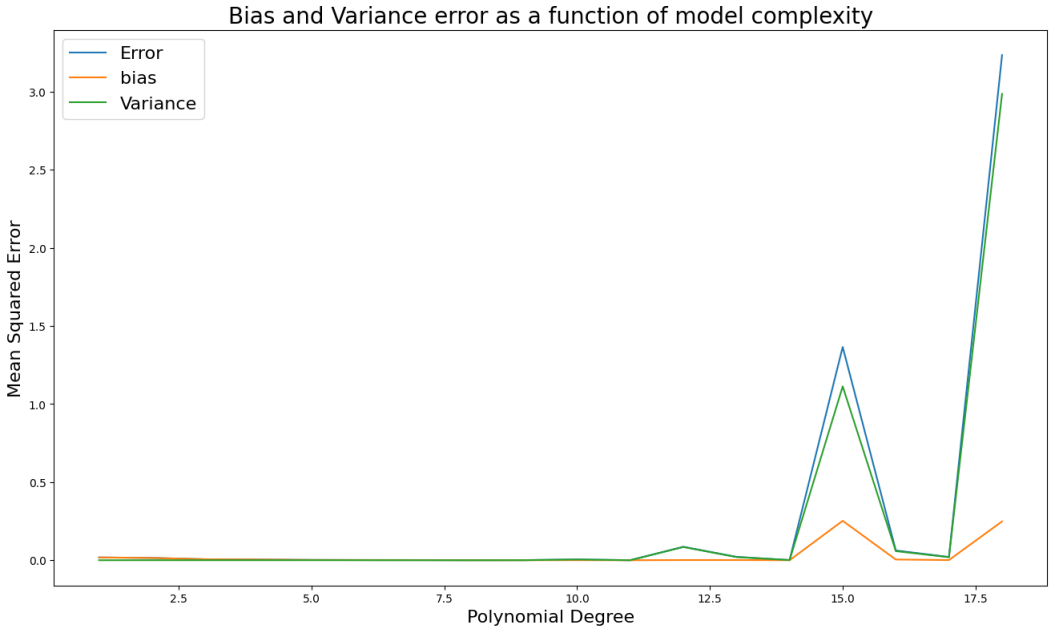

Figure 40: Plot of our Mean Squared Error given the model's complexity.

We can see a huge spike in error at the highest degree, corresponding to the variance error, which correspond to what we have witness theoretically 38.

4.7 Bootstrapping

We introduce bootstrapping (3.5) to our OLS method, by defining a number of trials and **shuffling our original design matrix every trial** in order to simulate a different dataset everytime. To do so, the code is :

```
for samples in range(trials):
    np.random.shuffle(X)
    X_train, X_test, z_train, z_tests = train_test_split(X, z, test_size=0.2)
    model = LinearRegression(fit_intercept=False).fit(X_train, z_train)
```

If we now take a look at our MSE error the same way we did when we studied the **Bias-Variance trade-off** (39), we get :

Figure 41: Mean Square Error given model's complexity **with Bootstrapping implemented**

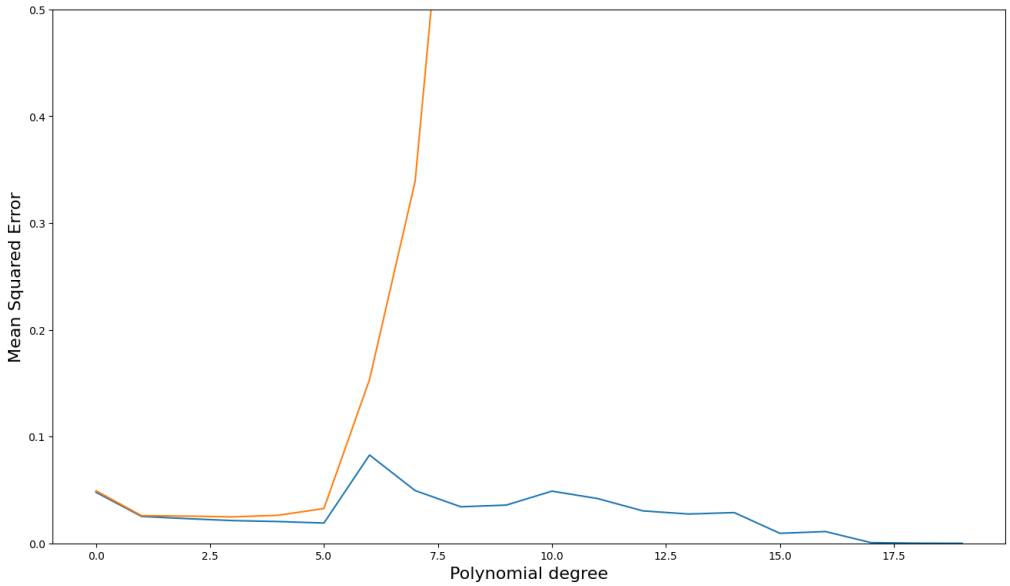

Figure 42: Mean Square Error given model's complexity **without Bootstrapping implemented**

The MSE score is greatly reduced compared to without Bootstrapping (39), which is expected since it allow us to redo the same computation over and over and averaging the results, thus **greatly reducing the impact of outliers.** Even though, we can still see the error spike as the model's complexity increases, we still have an improvement as the spike comes out **later than before.**

4.8 Cross-Validation as Resampling Technique

In this part, we are gonna implement the **K-fold Cross Validation**, which follows the same idea as defined in 3.4. Indeed K-fold means that we are dividing the data into **k subsets**. We are gonna split the data into subsets as use one of the subset as test data, and then redoing the entire process about 100 times in order to compute the **average error**.
We must also take into account that a value is into a specific group during the entire procedure, meaning that a value can only be in the test data once, and in the training data k-1 times.
Here, we are gonna implement K-fold for all three methods used, and for **5, 10 and 20 folds**. Thanks to *Scikit-Learn library*, the code implementation results in :

```
[...]
for k, k_array in k_dict.items():
    kfold = KFold(n_splits = k)
        mse_folds = cross_val_score(model, X_train, z_train[:, np.newaxis], scoring='neg_mean_squared
        k_array[i] = np.mean(-mse_folds)
[...]
```

Since both Ridge and Lasso Regression depends on a λ parameter, we can compare them depending on λ :

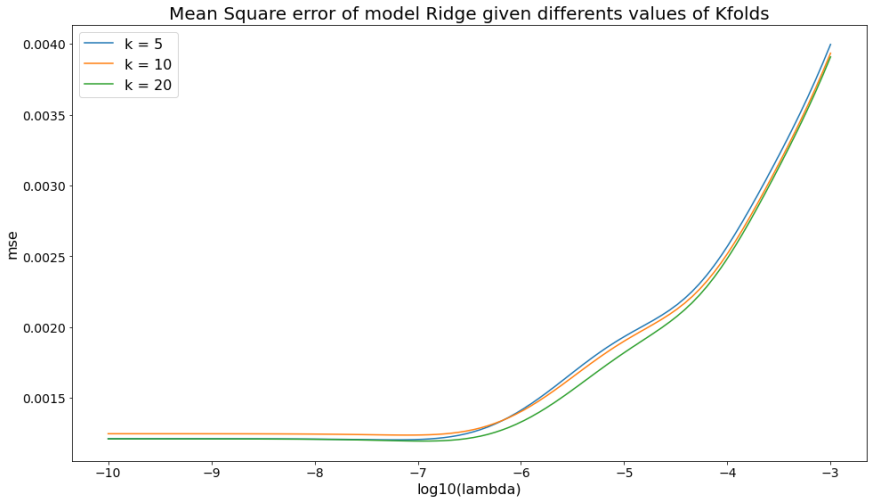

Figure 43: Mean Squared Error of Ridge model given λ with multiple K-fold values

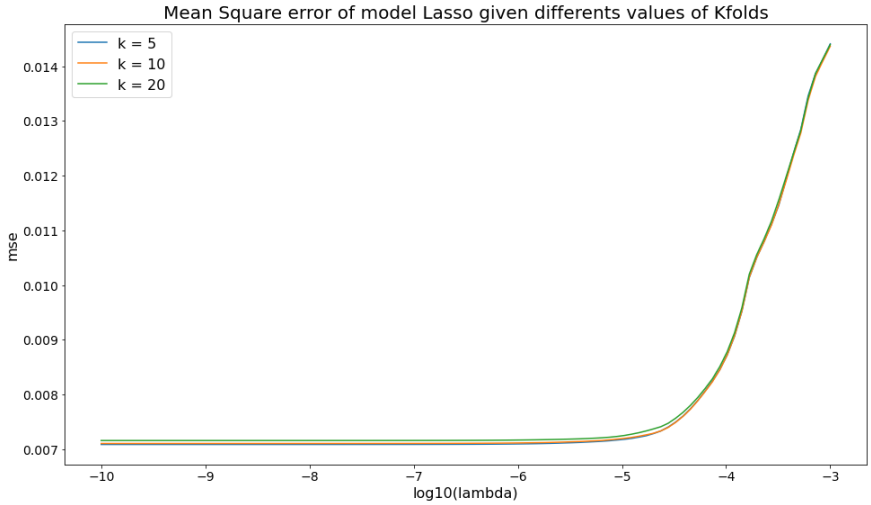

Figure 44: Mean Squared Error of Lasso model given λ with multiple K-fold value

We can see on Figure 43 that **the bigger the K-fold is, the lower is the MSE**. This can be expected because by averaging more train/test subsets, we can avoid uncommon situations and outliers, and thus reduce error.
This deduction however cannot be seen on Lasso Regression, as showed on Figure 44 as this time, the lowest MSE value is for K-fold = 5, which is here the lowest K-fold value.

Ordinary Least Squares cannot be studied in the same way because it does not contain a λ parameter. We can however discuss the K-fold utility on different polynomial degrees and discuss :

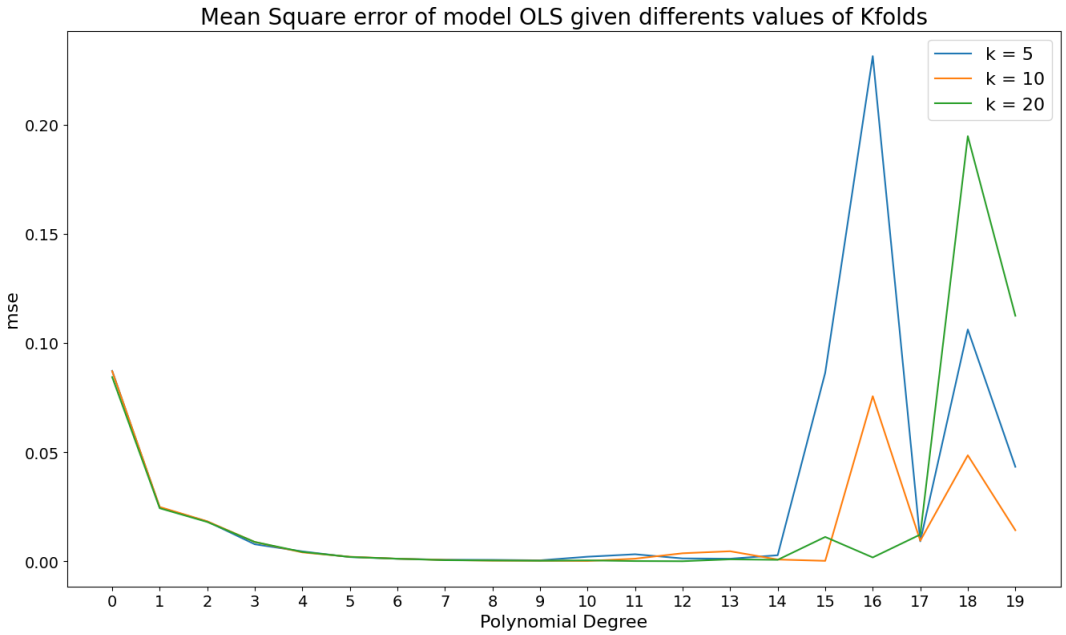

Figure 45: Plot of our Mean Squared Error given the model's complexity for multiple K-folds

Here on figure 45, it is more difficult to find the best K-folds, as there is **no clear winner starting polynomial degree of 14**. We did however discussed that the best MSE score was obtained at polynomial degree 12, and at this degree, it is the **20 K-folds that has the lowest MSE**, thus following the same principle as we saw with Ridge Regression.

5 Results and Conclusions on a function

Overall, if we compare all three methods and discuss which one if the most suited for Franke's Function, we can observe which method was the best one for the same situation (600 datapoints, polynomial degree of 12, with or without noise...).

To sum it up, if we use 600 datapoints and no noise, the best method seems to be **OLS**, as it gave the lowest MSE score of around 10e-4 while also reaching a R2 error very close to 1 at a polynomial degree of 12. Lasso lacks behind as both its MSE and R2 score are worse than its two counterparts.
If we instead study 600 datapoints and noise, the most optimal method now seems to be **Ridge Regression**, as its scores are better than the other two.

Best MSE	OLS	Ridge	Lasso
Without Noise	8.21e-05	0.000496	0.003384
With noise	0.048877	0.028465	0.034749
Best R2	OLS	Ridge	Lasso
Without Noise	0.999001	0.990297	0.933907
With noise	0.537444	0.683533	0.613672

Table 1: Summary of the best value for each method given the values

This can be explained by the λ paramater introduced in both Ridge and Lasso Regression as it makes certain features have a smaller impact on predictions, thus greatly reducing the variance of our method 3.2.
Lasso's lack of performance compared to Ridge Regression may be due to the fact that **Lasso did not manage to converge to a specific value**, as there is a lot of *ConvergenceWarning* stated by *Scikit-Learn* library, thus not reaching an optimal value. It may be possible that Lasso is in fact overperforming if we let it more time to compute, but it also highlights another problem illustrated in the next section.
It is also possible to optimize our results with the aid of **resampling techniques**, and even if these tools does improve our error scores, it does not change the fact that **Ridge Regression is still overperforming the other two methods.**
We thus concluded that OLS and Ridge seems to be the two best methods for prediction out of the three. However this is a conclusion that we made while exploring Franke's Function, a 2D mathematical function defined by the **two parameters x and y**. Even if we add noise, it is still a function that can be approximated. In the next section, we will find out if our observations are still valid if we implement these same methods on data that, at first glance, does not follow any distribution or function.

6 Demonstration on a Real Topographic Dataset

Now that we introduced all of our methods on a fictional dataset (illustrated by Franke's Function), we will now use all of these methods demonstrated before, and show how they work but this time on a real data set : The *Topographic terrain of Norway.*

6.1 Presentation of Dataset and Goals

We implement the same linear regression methods and the bootstrap and cross-validation resampling techniques in this section on a real data set. The real data set is a terrain over Norway with different heights:

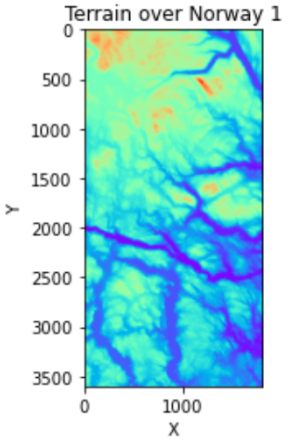

Figure 46: 2D Representation of our Norway topography

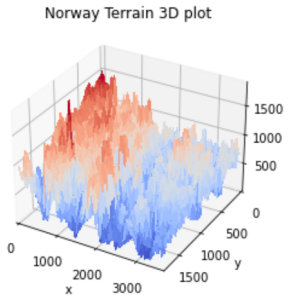

Figure 47: 2D Representation of our Norway topography

6.2 Methods implementation

To start off, we sample random points from the dataset to try to keep the features of the real data, but speed up running time. In our case, we chose to go with 3000 randomly selected points, as shown in the figure below:

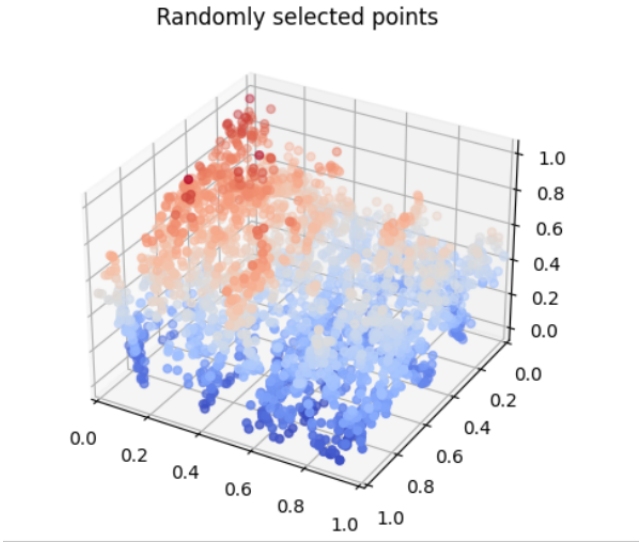

Figure 48: A random sample of 3000 points of our Topographic Dataset

We also use a **min-max scaling function** to rescale all of our values into a [0,1] range.

Let's first start studying our topographic dataset with our **Ordinary Least Square** method.

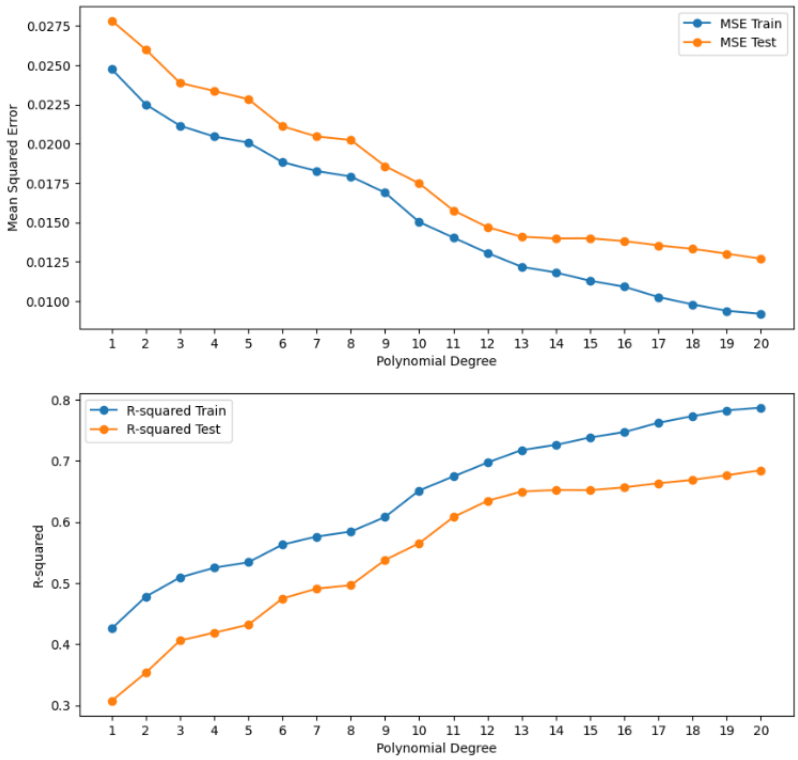

Figure 49: Ordinary Least Squares applied on our Topographic Data

Judging by our results (given figure 49), it is clear that the model improves with its polynomial degree, which is the same trend as when we were experimenting on the Franke Function earlier. Furthermore, **the test error is worse than the train error across the board** for both R^2 score and MSE. This, as seen before with Ridge and Lasso for the Franke Function, means it is overtrained.

Now implementing Ridge Regression, we define our lambda values as:

```
nlambdas = 100 # Number of lambdas studied
lambdas = np.logspace(-8,2,nlambdas) # values from 10^-8 to 10^2
```

This gives us room for trying 100 lambdas in sizes of good variation. After running the python script and printing the lowest test MSE for each degree and its corresponding lambda value, we get the following output:

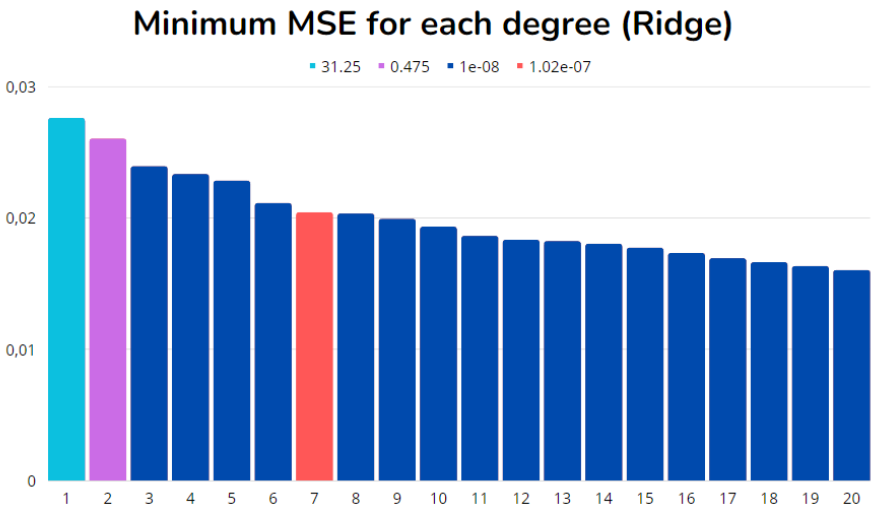

Figure 50: Minimum MSE for each polynomial degree using Ridge Regression. Each plot color correspond to a certain value λ

It is clear in this case that **the absolute lowest lambda value yields the best results**. Therefore we will now discuss a plot of the MSE and R^2 score for λ = **1 e-08**.

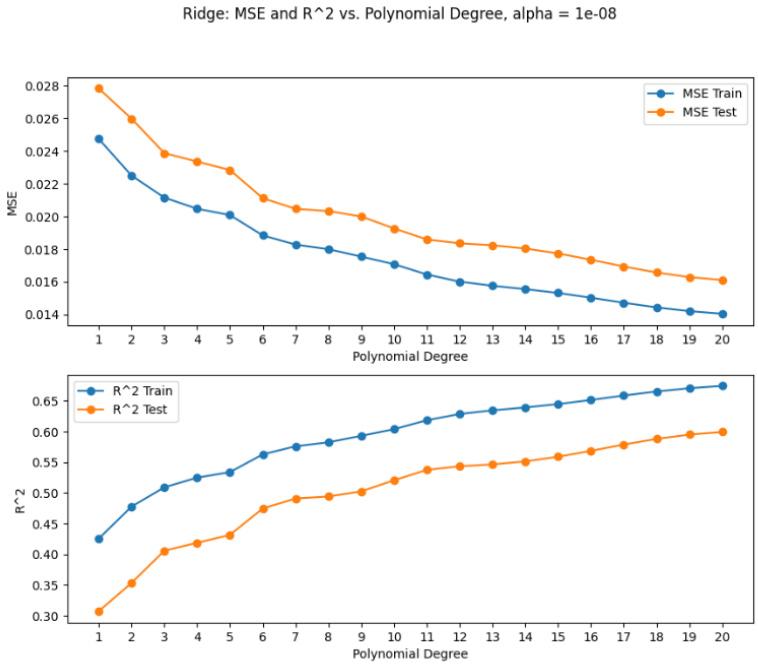

Figure 51: Ridge Regression applied on our Topographic Data for λ = 1e-8.

As with OLS, we can see on figure 51 that both MSE and R^2 score **improve as we increase the polynomial degree**. The relationship between test and train data remains the same; with test data being slightly worse than its training counterpart. One major issue is that it is also evident that **the overall MSE and R^2 score is worse here that OLS**, which begs the question if regularization is needed at all.

We then start off Lasso regression the same way as Ridge Regression by generating the **same lambdas**. By running the model, we calculate these values for every degree:

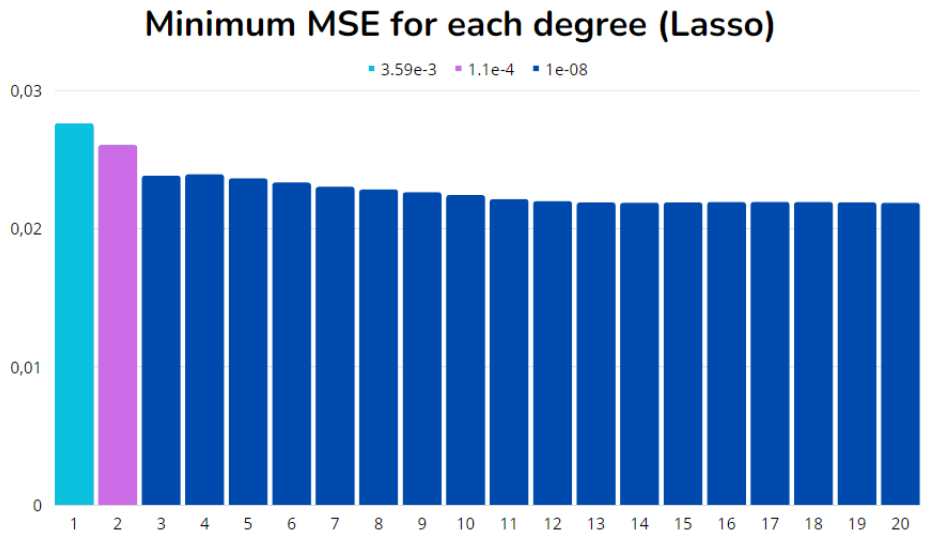

Figure 52: Minimum MSE for each polynomial degree using Lasso Regression. Each plot color correspond to a certain value λ

Once again, the lowest λ is the best for our model at **any polynomial degree greater than 2**, therefore we will study Lasso Regression for this specific λ.
Something worth mentioning is that our minimum MSE, although decreasing as the degree increases, is nonetheless **always higher than his Ridge counterpart**.

The below figure shows the full plot of Lasso regression with $\lambda = \mathbf{1\ e\text{-}08}$:

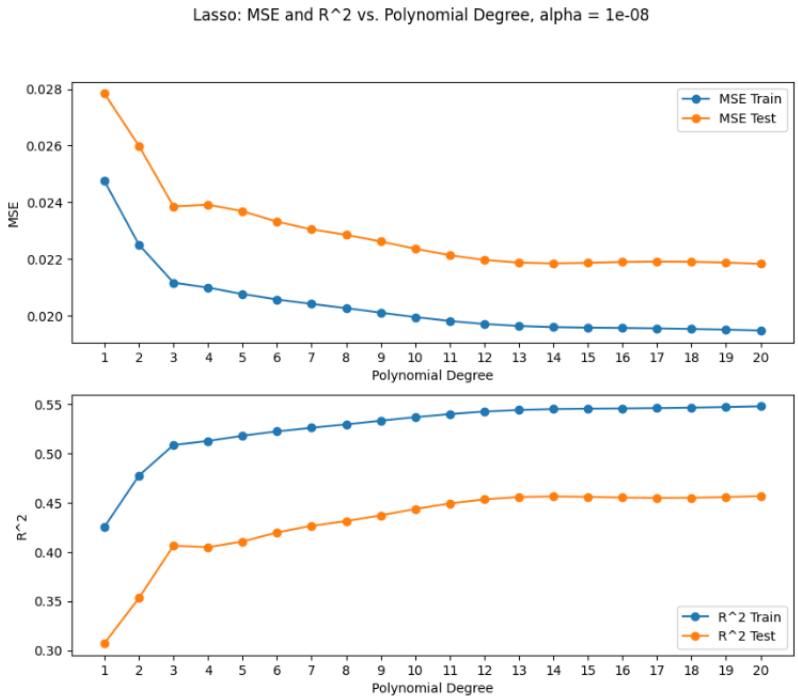

Figure 53: Lasso Regression applied on our Topographic Data for λ = 1e-8.

As with our previous implementation of linear regression, the MSE and R^2 score **both improve with polynomial degree**. Worth noting in this case is that it is our **worst model yet for the real data**, as the MSE and R2 are weaker for higher degrees.

If we want to represent our data's Bias-Variance trade-off, we get :

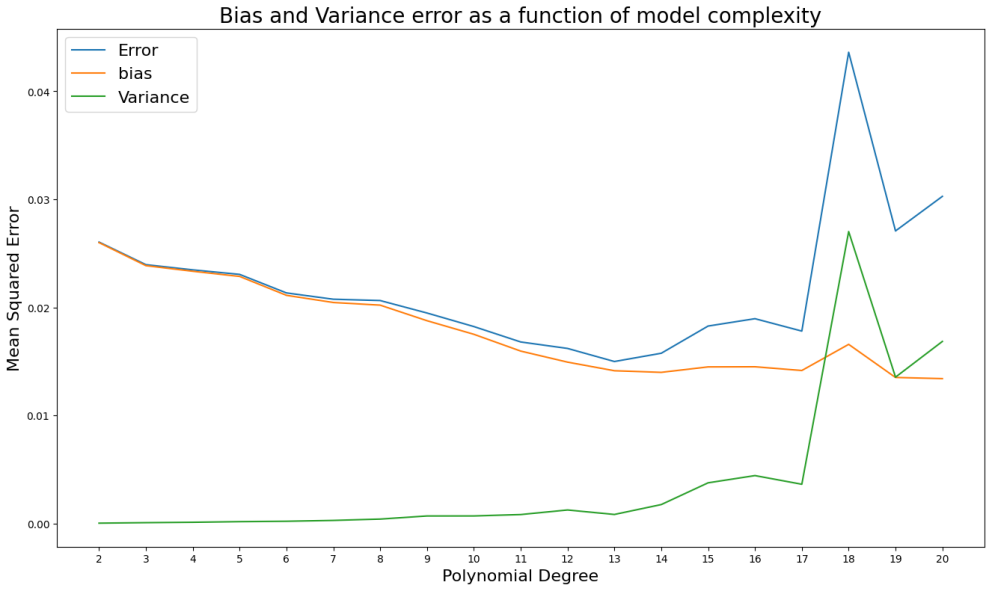

Figure 54: Bias-Variance trade-off of Norway's data

On smaller polynomial degrees, the variance is close to or equal to zero up until around degree 12. In other words, the only contributor to the error in this interval is the bias. This means that our model is underfitted in the interval, and as we could have expected given 38, there is a **big spike error** when the model's complexity is too high. Indeed at higher complexities, we can see that bias is rather stable, whereas the variance takes a huge turn for the worse at degree 18, which is yet again another indication of overfitting. because of an overfit. Therefore it is better to stick with a **polynomial degree of 13**, as it gives the lowest MSE score.

Bootstrapping was also implemented alongside OLS in order to study how impactful an average of multiple trials can be on our topographic data, giving us the two comparison plots :

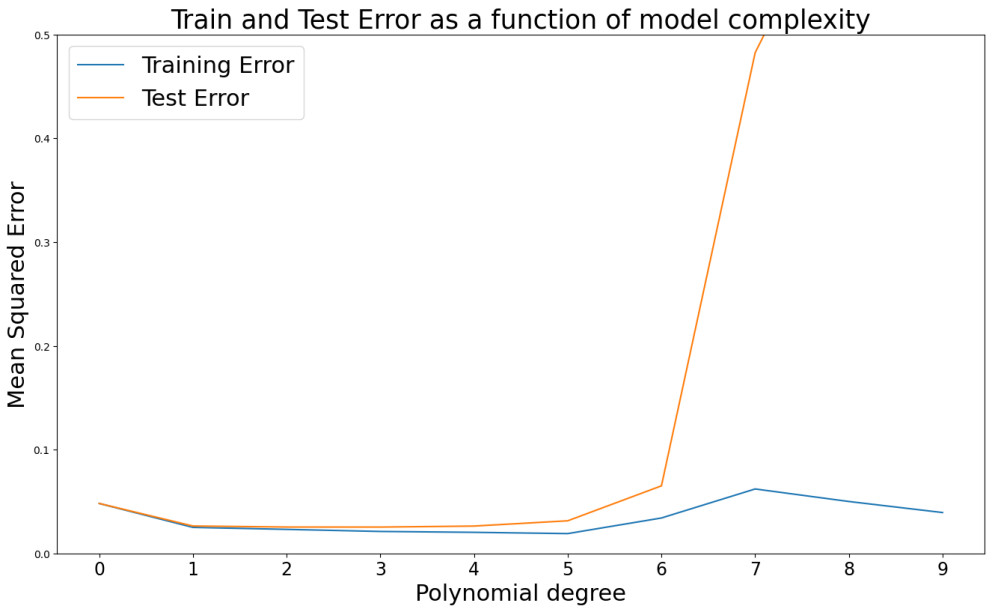

Figure 55: Mean Squared Error given model's complexity **without Boostrapping**

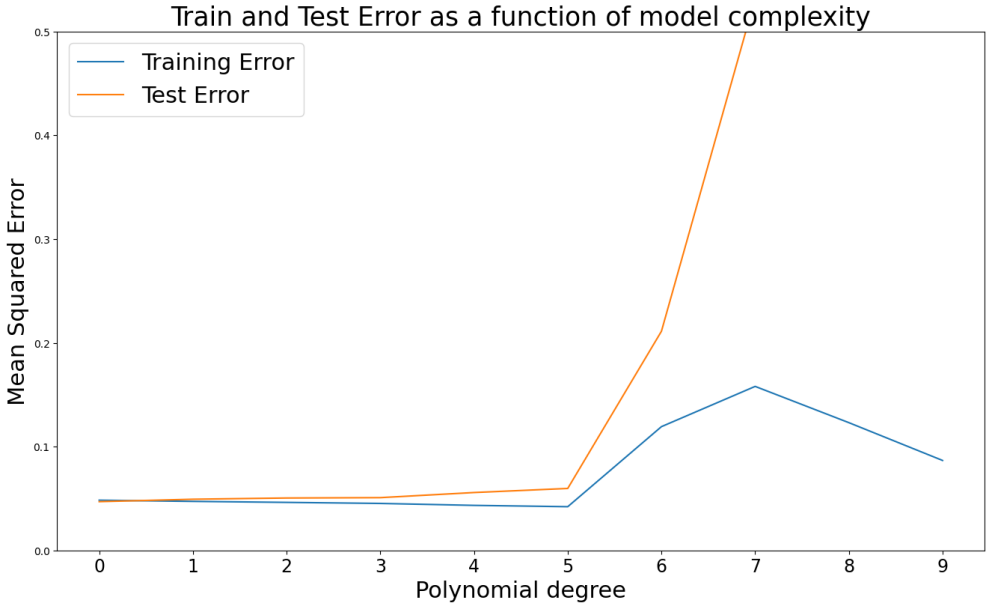

Figure 56: Mean Squared Error given model's complexity **with Boostrapping**

Unlike what we are discussed earlier 3.5, it appears that Bootstrapping **does not help in getting a better error score**. This is indeed due to the fact that the **original data is noisy**, thus the subsets will be composed of very noisy points, leading to the model overfitting by trying to fit these points.

We then implement the K-fold Cross Validation as we defined in 3.4, with 5, 10 and 20 k-folds for all three methods. The Ridge and Lasso Regression gives us the following graph :

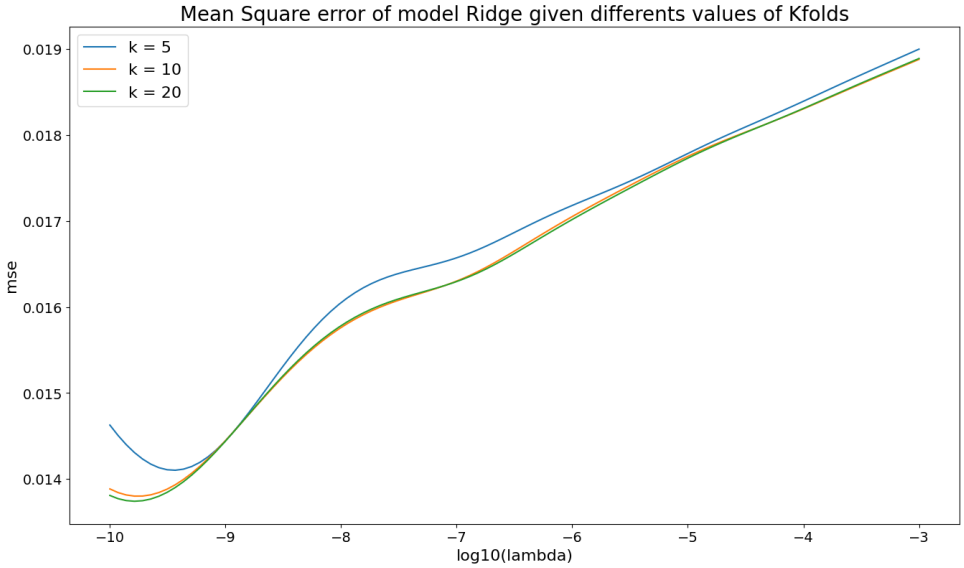

Figure 57: Mean Squared Error of Ridge model given λ with multiple K-fold values

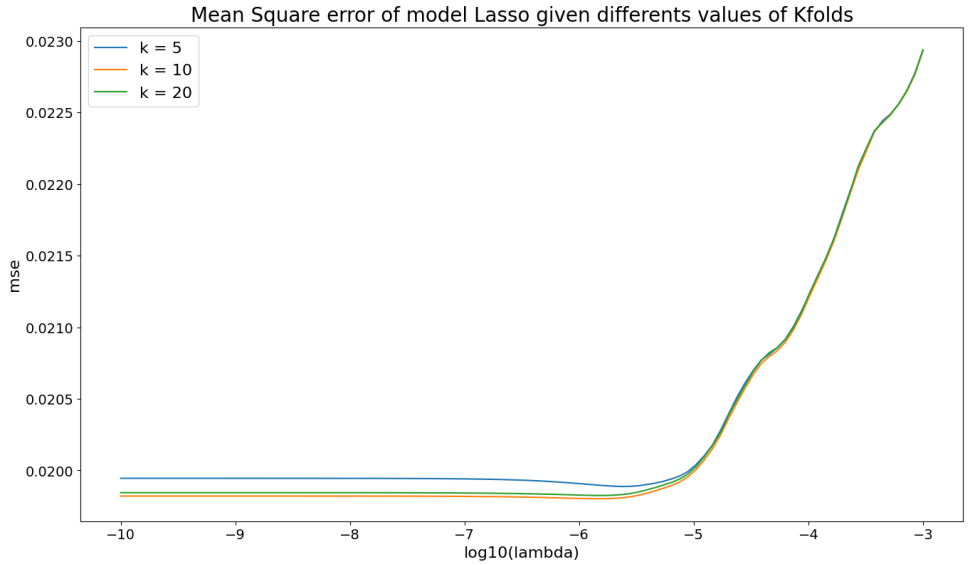

Figure 58: Mean Squared Error of Lasso model given λ with multiple K-fold value

Once again for Ridge Regression, at the lowest λ values, the **model with the most K-folds gives out the best results**, although the difference with the other K-folds is very slim, and the **overall error is about 0.014**. The Lasso Regression this time is a little bit different since it is the 10 folds method that produces the best result. However we can still see that the lowest MSE is obtained at the lowest λ. The MSE score is nonetheless a **little bit higher than Ridge**, at about 0.02.

The OLS K-fold implementation gives the following plot :

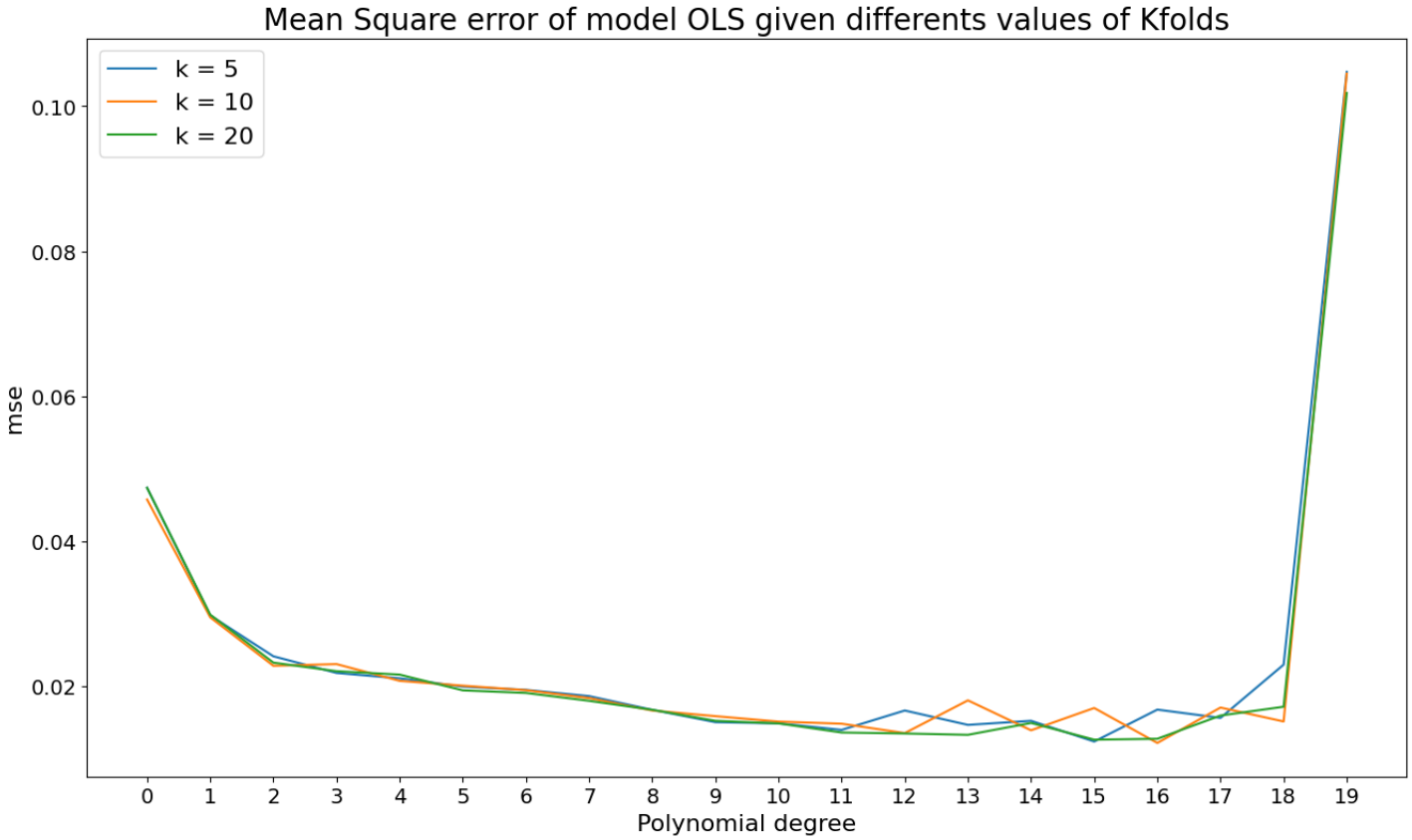

Figure 59: Mean Squared Error of Ridge model given model's complexity with multiple K-fold values.

Once again, there is not really a better K-fold number starting degree 13, but since we have seen in 49 that the best MSE is reached at degree 13, we can thus conclude that the best MSE is reached for 20 K-folds.

6.3 Interpreting Errors and Results

We have seen in our previous subsection 6.2 that all three methods seems to fit our data quite well, since their respective MSE is pretty low. However their R^2 error also indicates that it might overfit our data as well. But our main result is that **both Ridge and Lasso seem to be less optimal than OLS for this particular set of values.**
To make a case for Ridge and Lasso regression, we can see that the generalization gap (test error - train error) is smaller for both compared to OLS, at the highest and best performing degree (20), as we can see on the following table :

Degree = 20	OLS	Ridge	Lasso
Test Error	0.0127	0.0161	0.0218
Train Error	0.0092	0.0140	0.0195
Generalization Gap	0.0035	0.0021	0.0023

Table 2: Generalization gap of every method for our highest polynomial degree.

Even though OLS has the smallest error, it also has the biggest generalization gap, meaning that it is the most likely to overfit our data.
With that taken into account, we can observe that **Ridge Regression** not only has the second best error score, but also the smallest generalization gap, meaning that, on our topographic data, it is the best suited model out of the three methods, both with and without any resampling techniques or such.

7 Conclusion

During this study on Norway's topographic data, we implemented multiple methods to predict the landscape, based on a training dataset of said topography. Every method has its ups and downs, but it appears that for our specific dataset the best method that we implemented was **Ridge Regression**, since it overall gave the best error scores, as well as the minimal generalization gap. It could have been expected since it was our best method for Franke's Function with noise, and since we can assume that Norway's topography does not follow any function, **we can best compare it with a function with noise**, hence its success. Even by using resampling techniques in order to find better results, Ridge Regression overcome the other two methods nonetheless.
Another metric to keep in mind is the **computation time**, ie the time a method takes to converge into a result and compute it. Indeed, although OLS and Ridge took a similar time to compute, **the same can not be said for Lasso Regression**, as it is taking way longer to compute while also giving the worst results out of the three.
All three methods can be optimized into giving better results if we implement resampling techniques such as the Cross-Validation method, reducing their overall error, however is it not achieved without proper studying and more computations.

8 Appendix

All code implementations can be found on Our GitHub, but here will follow either the most important parts, or some other code implementations that need more explanations :

8.1 All libraries used

The main libraries used for this project were **Numpy, Matplotlib and Scikit-Learn**. The complete list can be found here :

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import random as rd
import scipy
import sklearn.linear_model as skl
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler, StandardScaler, Normalizer
from sklearn.metrics import mean_squared_error, r2_score, mean_squared_log_error, mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.utils import resample
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from imageio import imread
import warnings
from sklearn.exceptions import ConvergenceWarning
```

8.2 Min-Max Scaler

We created a function to **min-max our data** and scale them so that the values ranges in [0,1], since we couldn't use the usual *StandardScaler* from *Scikit-Learn* as our data needed more manipulations and computations.

```
    min_i = min(data)
    max_i = max(data)

    scaled = (data-min_i)/(max_i-min_i)
    return scaled
```

8.3 Generate subset of features

Our subset of features from the input data x is computed with the **c_tilde** function in order to further predict MSE and R2 scores at a certain degree **deg**

```
c = int((deg + 2) * (deg + 1) / 2)
tilde = x[:,0:c-1]
return c, tilde
```

8.4 Generating the Design Matrix

The design matrix X is created with the same function than for Franke's Function, however since our datapoints are created differently, we need to manipulate them differently. It is also worth noticing that the first column is **the intercept**, column that we will remove most of the time as it can mess up our output predictions.

```
#create design matrix X
X = create_X(datapoints[:, 0], datapoints[:, 1], PolynomialDegree)

#Remove intercept
X = X[:, 1:]

#Split into train and test sets
x_train, x_test, z_train, z_test = train_test_split(X, z, test_size = 0.2)
```

8.5 Finding Lowest Error Value and Corresponding λ

For Ridge and Lasso Regression, we observed that the best error scores were **not necessarily obtained by the lowest** λ, although it was the case most of the time. This code helped us see which λ was the most optimal for each polynomial degree.

```
for degree in range(1, PolynomialDegree + 1):
# Find the index of the lambda with the minimum MSE Test for the current degree
min_mse_lambda_idx = np.argmin(MSETest[:, degree-1])

# Get the lambda value and the corresponding minimum MSE Test
min_mse_lambda = lambdas[min_mse_lambda_idx]
min_mse_value = MSETest[min_mse_lambda_idx, degree-1]

print(f"Degree {degree}: Minimum MSE Test = {min_mse_value} (Lambda = {min_mse_lambda})")
```

References

[1] Abhishek Sharma 44. *Cross Validation in Machine Learning*. 2023. URL: https://www.geeksforgeeks.org/cross-validation-machine-learning/.

[2] dx2-66. *Is it possible to have a higher train error than a test error in machine learning?* 2022. URL: https://stats.stackexchange.com/questions/582637/is-it-possible-to-have-a-higher-train-error-than-a-test-error-in-machine-learnin#:~:text=Overfitting%20means%20that%20a%20model,error%20are%20about%20equally%20high.

[3] Jason Fernando. *How can R^2 be negative?* 2023. URL: https://www.graphpad.com/support/faq/how-can-rsup2sup-be-negative.

[4] Jason Fernando. *R-Squared: Definition, Calculation Formula, Uses, and Limitations*. 2023. URL: https://www.investopedia.com/terms/r/r-squared.asp.

[5] Stephanie Glen. *Resampling Techniques*. 2023. URL: https://www.statisticshowto.com/resampling-techniques/.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. "The Elements of Statistical Learning (Data Mining, Inference and Predictions)". In: (2009), p. 38. DOI: 10.1007/b94608.

[7] Wouter van Heeswijk. *Regulate Your Regression Model With Ridge, LASSO and ElasticNet*. 2022. URL: https://towardsdatascience.com/regulate-your-regression-model-with-ridge-lasso-and-elasticnet-92735e192e34.

[8] Morten Hjorth-Jensen. *Interpretations and Optimizing our Parameters*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html#interpretations-and-optimizing-our-parameters.

[9] Morten Hjorth-Jensen. *Preprocessing Our Data*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html#preprocessing-our-data.

[10] Morten Hjorth-Jensen. *Ridge and Lasso Regression*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter2.html#id1.

[11] Morten Hjorth-Jensen. *Simple Linear Regression Model Using Scikit-Learn*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week34.html#simple-linear-regression-model-using-scikit-learn.

[12] Morten Hjorth-Jensen. *Splitting our Data in Training and Test Data*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html#splitting-our-data-in-training-and-test-data.

[13] Morten Hjorth-Jensen. *The Equations for Ordinary Least Squares*. 2022. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html#the-equations-for-ordinary-least-squares.

[14] Kiprono Elijah Koech. *Cross Validation in Machine Learning*. 2021. URL: https://towardsdatascience.com/cross-validation-and-bootstrap-sampling-2e041fbec126.

[15] Dinesh Kumar. *A Complete understanding of LASSO Regression*. 2023. URL: https://www.mygreatlearning.com/blog/understanding-of-lasso-regression/#:~:text=LASSO%20regression%2C%20also%20known%20as,Absolute%20Shrinkage%20and%20Selection%20Operator..

[16] Charity W. Law et al. *A guide to creating design matrices for gene expression experiments*. 2020. URL: https://f1000research.com/articles/9-1444.

[17] Lumivero. *ORDINARY LEAST SQUARES REGRESSION (OLS)*. 2023. URL: https://www.xlstat.com/en/solutions/features/ordinary-least-squares-regression-ols.

[18] Donald W. Marquardt and Ronald D. Snee. "Ridge Regression in Practice". In: (1975), p. 5. URL: https://www.jstor.org/stable/2683673.

[19] L.E. Melkumova and S.Ya. Shatskikh. "Comparing Ridge and LASSO estimators for data analysis". In: *Procedia Engineering* 201 (2017). 3rd International Conference "Information Technology and Nanotechnology", ITNT-2017, 25-27 April 2017, Samara, Russia. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2017.09.615. URL: https://www.sciencedirect.com/science/article/pii/S1877705817341474.

[20] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. John Wiley & Sons Inc., 2021. URL: https://lccn.loc.gov/2020034056.

[21] Joseph Nguyen. *Regression Basics for Business Analysis*. 2022. URL: https://www.investopedia.com/articles/financial-theory/09/regression-analysis-basics-business.asp#:~:text=Simple%20linear%20regression%20is%20commonly,could%20affect%20sales%2C%20for%20example.

[22] Nate Rosidi. *Machine Learning: What is Bootstrapping?* 2023. URL: https://www.kdnuggets.com/2023/03/bootstrapping.html.

[23] Saily Shah. *Cost Functions is No Rocket Science!* 2021. URL: https://www.analyticsvidhya.com/blog/2021/02/cost-function-is-no-rocket-science/.

[24] Sonja Surjanovic and Derek Bingham. *Virtual Library of Simulation Experiments : Test functions and datasets*. 2013. URL: https://www.sfu.ca/~ssurjano/franke2d.html.

[25] Liu Wei et al. "Analysis of factors associated with disease outcomes in hospitalized patients with 2019 novel coronavirus disease". In: (2020). URL: https://journals.lww.com/cmj/fulltext/2020/05050/analysis_of_factors_associated_with_disease.5.aspx.

[26] Sanford Weisberg. *Applied Linear Regression*. Wiley-Interscience. John Wiley & Sons Inc., 2005. URL: https://ds.amu.edu.et/xmlui/bitstream/handle/123456789/9424/Book_Applied_Linear_Regression_3rd_ed_By_Weisberg_2005%5B2%5D.pdf?sequence=1&isAllowed=y.

[27] Gary C. White. *Beta Parameters*. 2023. URL: https://sites.warnercnr.colostate.edu/gwhite/beta-parameters/.

[28] Wikipedia. *Design Matrix*. 2023. URL: https://en.wikipedia.org/wiki/Design_matrix.