Universitetet i Oslo

FYS-STK 4155

Project 1

Linear Regression and Terrain Data

Author:
Helle Røtterud Gjertsen
September 30, 2024

Abstract

The aim of this project has been to study linear regression models by comparing the performance of Ordinary Least Squares (OLS), Ridge-, and Lasso Regression when applied to terrain data from Møsvatn Austfjell in Norway. These models are implemented in python, making use of libraries such as Numpy, MatPlotLib and Scikit-Learn. Initially, 4/5s of the data was used as a training set while the remaining 1/5 was used as a test set for validation. The terrain data was scaled using scikit-learn's StandardScaler. OLS performed best, with a mean squared error of 0.4 for a polynomial of degree 4. This performance was further improved by implementing splitting using k-folds.

Contents

1 Introduction

When searching for the best way to model a data set, some assumptions must be made. Central to linear regression methods is the assumption that there exists some continuous function $f(x)$ that describes the data (and that we can somehow find an approximation of this function [1, p. 28]. This project compares three linear regression methods: Ordinary Least Squares (OLS) - the simplest - and then Ridge- and Lasso Regression, which both introduce a shortening term λ.

Chpt. 2 outlines the mathematical theory behind these methods, while chpt. 3 goes through the implementation details. Chpt. 4 contains the results from the linear regression analyses done on terrain data from Møsvatn Austfjell, Norway. Chpt. 5 contains a critical discussion of these results.

Calculations of mean squared error (MSE) were first done for an identity matrix with $\lambda = 0$ (for Ridge and Lasso), which results in a perfect fit. This was done to ensure the functions return the expected results. Then, all functions were tested for a surface obtained using Franke's function (eq. 2.2).

All code for this project is written in python, and can be seen in the appendices: Appendix A contains the code used for writing and testing the functions used in the analysis, and appendix B contains changes made for the terrain data analysis. All code can also be found in my github-repo: hellegje/fys-stk4155/Project1.

The following libraries were used:

- Numpy [2] for array handling and certain mathematical functions,
- Matplotlib [3] for plots and visualisation,
- SciKitLearn [4] for regression functions,
- ImageIo [5] for reading image data

Finally, a note on scaling and data handling: No scaling was added for the Franke's function data when testing, as this set was already fairly uniform. The MSE and R2-scores seemed reasonable without any further scaling or pre-processing of the data. The terrain data, on the other hand, has greater height variations which make interpretations of MSE-scores more difficult without first scaling the data. The calculations for higher order polynomials unfortunately required more computational power than was accessible, so in order to obtain results the terrain data had to be simplified somewhat. This limits the analysis, but the results still give insight into the behaviours of the different regression methods.

2 Linear Regression

2.1 Error Functions

As seen in eq. 1 (where $\mathbf{y}$ are the actual values, $\tilde{\mathbf{y}}$ are the predicted values, and n are the number of data points), MSE goes to zero for a perfect fit.

$$MSE(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{1}$$

The R2-sore seen in 2 (where $\overline{y}$ is the mean) weights the difference between real and predicted value by the distance from the mean. Here, a perfect fit will result in an R2-score equal to 1.

$$R^2(\mathbf{y}, \tilde{\mathbf{y}}) = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \overline{y}_i)^2} \tag{2}$$

In this project, both these errors are used to evaluate the model's performance.

2.2 The Franke Function

Franke's function seen in eq. 3 creates surface data and is often used for testing regression models [6].

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

2.3 Ordinary Least Squares

Ordinary Least Squares (OLS) is a simple and often-used method for making predictions in linear regression. The approach seeks to find the coefficients β which minimises the residual sum of squares [1, p. 12].

In linear regression methods, we assume there exists a continuous function $f(x)$ which may describe the data:

$$\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\varepsilon} \tag{4}$$

where $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2)$, i.e. $\boldsymbol{\varepsilon}$ is normally distributed noise with mean 0 and variance σ^2. In addition,

$$\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta} \tag{5}$$

where the function f is approximated by model $\tilde{\mathbf{y}}$, $\mathbf{X}$ is the feature matrix and $\boldsymbol{\beta}$ are the model parameters. The expectation value of $\mathbf{y}$ for element i can be written as:

$$\begin{aligned} \mathbb{E}(y_i) &= \mathbb{E}(\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta}) + \mathbb{E}(\varepsilon_i) \\ &= \mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta}. \end{aligned} \tag{6}$$

The variance can be expressed as:

$$\begin{aligned} Var(y_i) &= \mathbb{E}\{[y_i - \mathbb{E}(y_i)]^2\} = \mathbb{E}(y_i^2) - [\mathbb{E}(y_i)]^2 \\ &= \mathbb{E}[(\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta} + \varepsilon_i)^2] - (\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta})^2 \\ &= \mathbb{E}[(\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta})^2 + 2\varepsilon_i\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta} + \varepsilon_i^2] - (\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta})^2 \\ &= (\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta})^2 + 2\mathbb{E}(\varepsilon_i)\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta} + \mathbb{E}(\varepsilon_i^2) - (\mathbf{X}_{\mathbf{i},*}\boldsymbol{\beta})^2 \\ &= \mathbb{E}(\varepsilon_i^2) = Var(\varepsilon_i) = \sigma^2 \end{aligned} \tag{7}$$

Equations 6 and 7 show that $\mathbf{y}$ follows a normal distribution with mean value $\mathbf{X}\boldsymbol{\beta}$ and variance σ^2.

From OLS the predicted outcome can be expressed as follows [1, p. 46, eq. 3.7]:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{8}$$

where $\hat{\beta}$ are the optimal parameters.

Applying equations 6 and 8 gives us the following expression:

$$\begin{aligned} \mathbb{E}(\hat{\beta}) &= \mathbb{E}[(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})] \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbb{E}(\mathbf{y}) \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \\ &= \boldsymbol{\beta} \end{aligned} \tag{9}$$

The variance for β can be derived as follows:

$$\begin{aligned} Var(\hat{\boldsymbol{\beta}}) &= \mathbb{E}\{[\boldsymbol{\beta} - \mathbb{E}\boldsymbol{\beta}][\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})]^T\} \\ &= \mathbb{E}\{[(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} - \boldsymbol{\beta}][(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} - \boldsymbol{\beta}]^T\} \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbb{E}\{\mathbf{y}\mathbf{y}^T\}\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\boldsymbol{\beta}^T \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\{\mathbf{X}\boldsymbol{\beta}\boldsymbol{\beta}^T\mathbf{X}^T + \sigma^2\}\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\boldsymbol{\beta}^T \\ &= \boldsymbol{\beta}\boldsymbol{\beta}^T + \sigma^2(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\boldsymbol{\beta}^T \\ &= \sigma^2(\mathbf{X}^T\mathbf{X})^{-1} \end{aligned} \tag{10}$$

(Where not otherwise stated, equations are taken from the project description [7]).

2.4 Ridge Regression

Ridge Regression is a shrinkage method: A way of limiting the importance of certain features by introducing a shrinking term λ [1, p. 63, eq. 3.41]:

$$\hat{\beta}^{ridge} = \underset{\beta}{\operatorname{argmin}}\{\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}(x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\} \tag{11}$$

Eq. 11 shows the Ridge equation for optimal coefficients β^{ridge}. $\lambda > 0$ controls the shrinking, penalising large coefficients [1, p. 61].

2.5 Lasso Regression

As in Ridge Regression, Lasso also penalises large coefficients by introducing the shrinking term λ, but here the coefficients are reduced faster.

$$\hat{\beta}^{lasso} = \underset{\beta}{\operatorname{argmin}}\{\frac{1}{2}\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p}(x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j^2|\}[1, p.\ 68, eq.\ 3.52] \tag{12}$$

2.6 Bias-variance Trade-off

Regression models must balance bias, error, and variance, or as Belkin et. al. put it: "[find] the 'sweet spot' between underfitting and overfitting" [8]. The point is to find a model that is complex enough to provide good predictions, without also adding in patterns from random noise in the training data.

From the cost function [7]

$$C(\mathbf{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] \tag{13}$$

with

$$Bias[\tilde{\mathbf{y}}] = \mathbb{E}[(\mathbf{y} - \mathbb{E}[\tilde{\mathbf{y}}])^2] \tag{14}$$

and

$$var[\tilde{\mathbf{y}}] = \mathbb{E}[(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}])^2] = \frac{1}{2}\sum_i(\tilde{\mathbf{y}}_\mathbf{i} - \mathbb{E}[\tilde{\mathbf{y}}])^2 + \sigma^2, \tag{15}$$

the sample value $\mathbb{E}$ can be expressed in terms of variance and bias as follows [7]:

$$\begin{aligned} \mathbb{E}[(\mathbf{y} - \tilde{\mathbf{y}})^2] &= \mathbb{E}[(\mathbf{f} + \boldsymbol{\varepsilon} - \tilde{\mathbf{y}})^2] \\ &= \mathbb{E}[(\mathbf{f} + \boldsymbol{\varepsilon} - \tilde{\mathbf{y}} + \mathbb{E}[\tilde{\mathbf{y}}] - \mathbb{E}[\tilde{\mathbf{y}}])^2] \\ &= \mathbb{E}[(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}])^2] + Var[\tilde{\mathbf{y}}] + \sigma^2 \\ &= Bias[\tilde{\mathbf{y}}] + var[\tilde{\mathbf{y}}] + \sigma^2 \end{aligned} \tag{16}$$

as $var(\mathbf{y}) = var(\varepsilon) = \sigma^2$ and the mean value of ε is 0.

3 Development and Testing

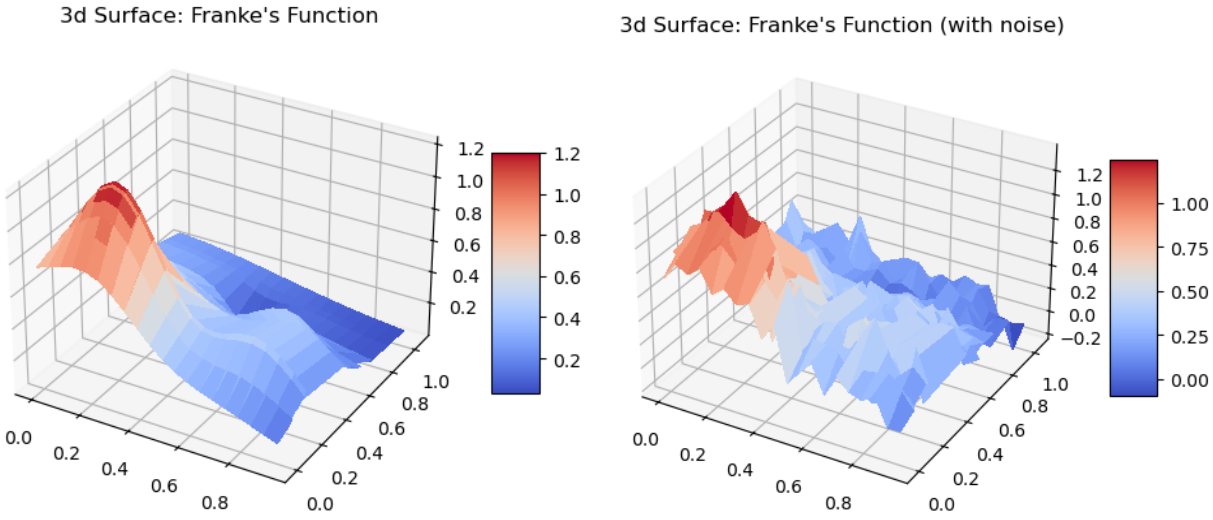
3.1 Initial testing

All functions described in this section were first tested on a 2d dataset. MSE calculations were tested with $\lambda = 0$ and feature matrix set to identity matrix resulting in MSE = 0.

3.2 Ordinary Least Squares and the Franke Function

First test was to perform an OLS analysis on a data set using the Franke Function (see Appendix A for implementation details).

The Franke Function creates a surface as seen in fig. 1.



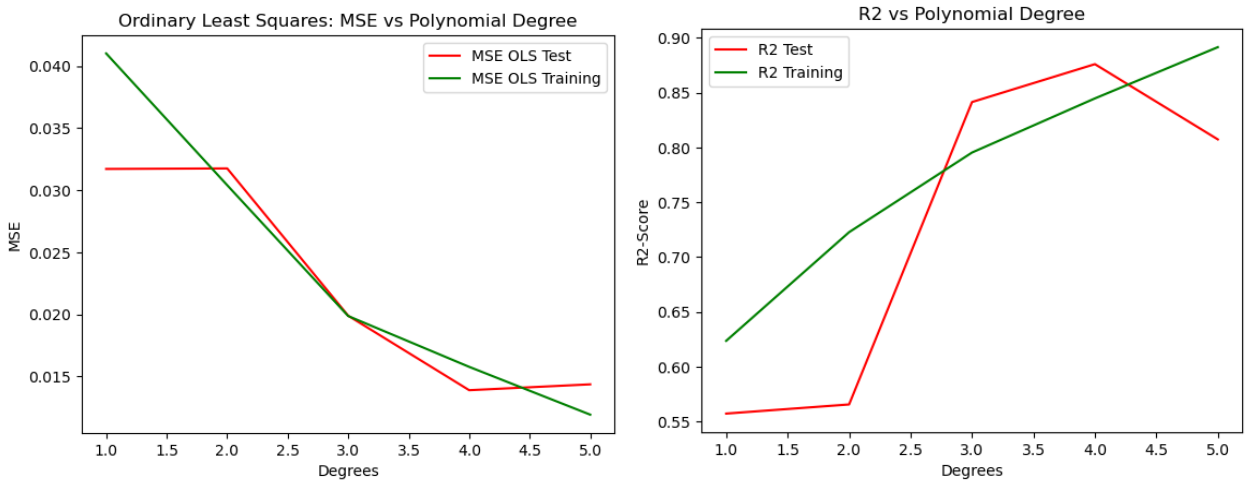
(a) Franke surface

(b) Franke surface with noise

Figure 1: Plotted surfaces for a data set built using Franke's function.

This data set was then used when writing the code for OLS regression analysis (see A, listing 3). For this test the data was not scaled, as the values were low with no extreme outliers. When working with real data, however, scaling becomes necessary, as described in chpt. 5.1.

In order to better decide on an optimal model complexity, the data set was split into training and test data. 4/5th was used as a training set, while the last 1/5 was left as a test set. Fig. 2 shows MSE and R2-scores applied to both training and test sets. As seen in eqs. 1 and 2, for a perfect fit MSE and R2 scores will approach 0 and 1 respectively, and the above figures show the model generally improving for higher order polynomial fits. For the highest order polynomial however, we see a drop in precision. An increasing error can be a sign of over fitting, where the model reflects the training data so closely it fails when applied to new, unseen data (in this case the test set).



(a) MSE vs. Model Complexity.

(b) R2 vs. Model Complexity.

Figure 2: Errors vs. Model Complexity. Fig. a shows how MSE for test and training sets improve with increased complexity. For the very highest degree, however, the test error is overtaken by the training error. Fig. b similarly shows the training error improving for higher degrees of the polynomial, while the test error reaches a peak at the 4th order, and decreases for the highest value.

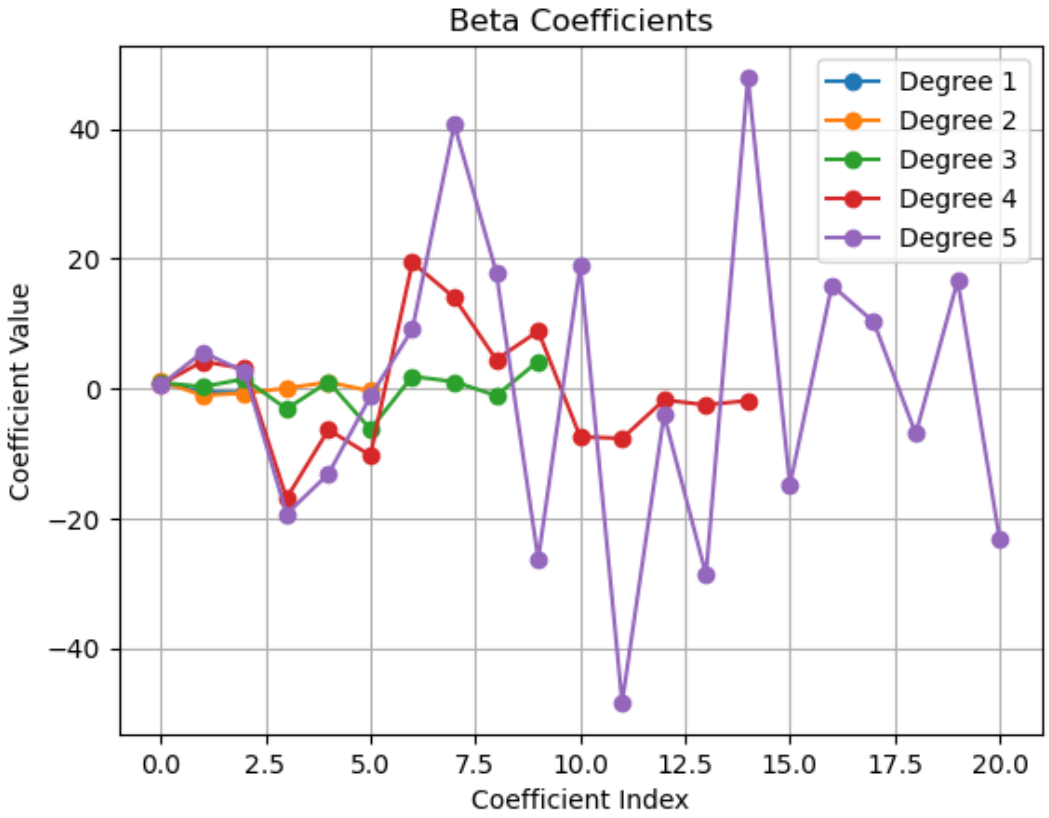
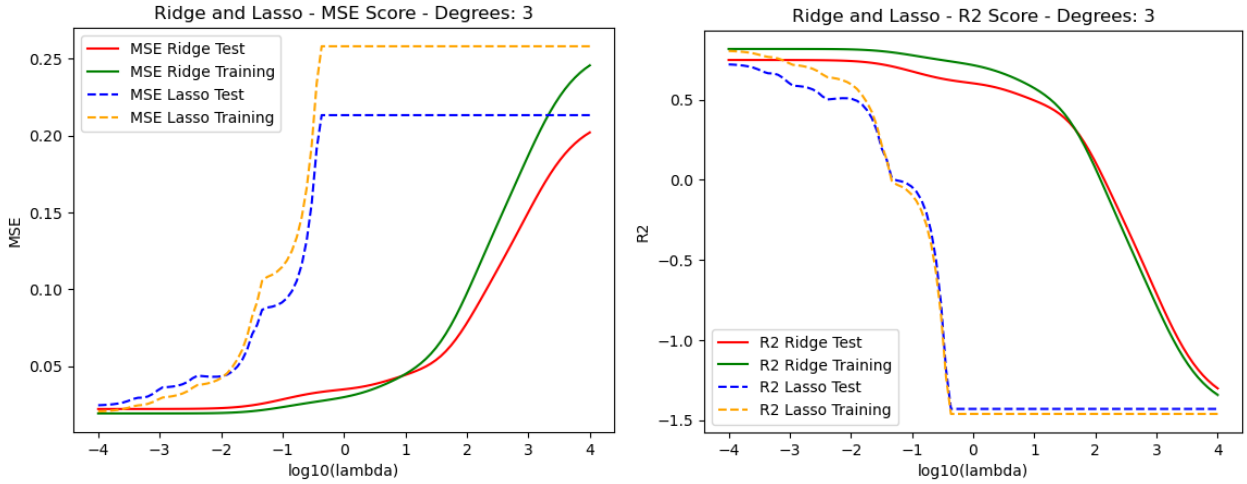


Figure 3: Beta Coefficients as they vary with model complexity.

Fig. 3 shows how the beta coefficients vary with increasing complexity. The larger order polynomials also show a greater variety in coefficient values.

3.3 Adding Ridge and Lasso Regression

In order to compare various regression methods, code for Ridge- and Lasso Regression (as seen in lst. 4) was added to the analysis. Fig. 4 shows how greater values for the dampening lambda term affects the error. Where the value gradually changes in a Ridge regression model, the figure shows a clear, sudden change for the Lasso regression model.



(a) MSE vs. lambda for Ridge and Lasso.

(b) R2 vs. lambda for Ridge and Lasso.

Figure 4: Errors vs. lambda. Fig. a shows how the error increases differently in Ridge (red and green) and Lasso Regression (blue and orange). Fig. b shows the R2-scores. Again with Ridge in red and green and Lasso in blue and orange. For both errors, Ridge changes gradually while Lasso has a sudden change.

3.4 OLS vs. Ridge vs. Lasso

Fig. 4 a shows a low MSE value for low values of lambda, and increasingly poor results as lambda increases. The same effect can be seen in the R2-scores. Both values start out similar to the errors from the OLS analysis. As seen by comparing eq. 11 to OLS, small values of λ approximate the behaviour of OLS, suggesting the OLS-model actually is the best fit to this data.

Fig. 5 shows how both the error and the variance increase above a 3rd order polynomial. The figure was created using the code as seen in lst. 5. We see bias decrease for higher complexity, as more patterns in the data set are captured by the fitted model. We also see that variance shoot up above the 3rd order polynomial. This suggests overfitting, and fits well with the errors from fig. 2.

3.5 Cross-validation

Cross-validation by k-fold involves splitting the data into k-subsets, where each subset constitutes an iteration. For each iteration, one subset is used as the test set while the others constitute the training data. This method is particularly useful if data is scarce, as it allows for all of the data to be used as a test set at some point, greatly increasing the data upon which we can perform the validation [1, p. 241].

Fig. 11 shows the different MSE-scores resulting from varying the number of k-folds. For OLS we see, as from bootstrap in fig. 5, the error increasing above the 3rd order polynomial. Lasso- and Ridge regressions were run for more values of lambda than in previous sections, and we see that both plots now show an initial drop in error before all k-folds stabilise around a similar value. The code used can be seen in lst. 6

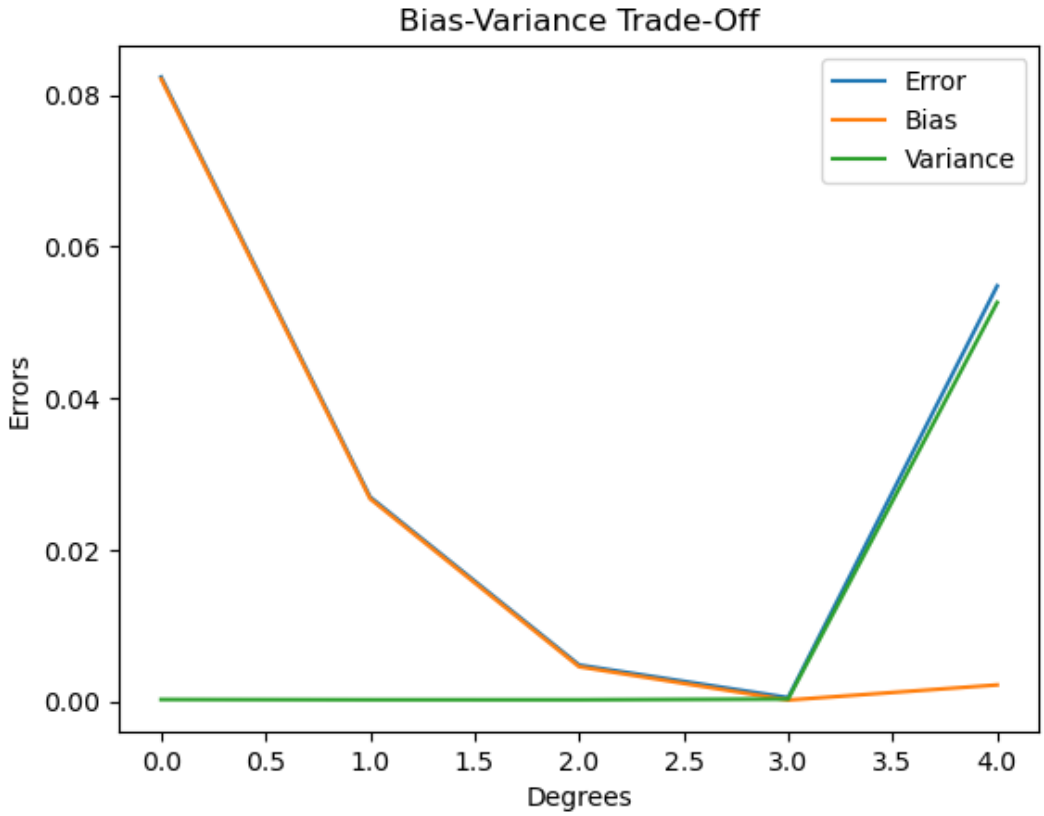
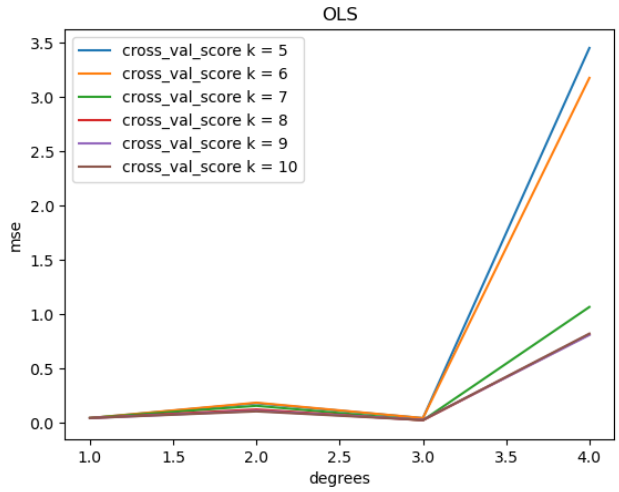


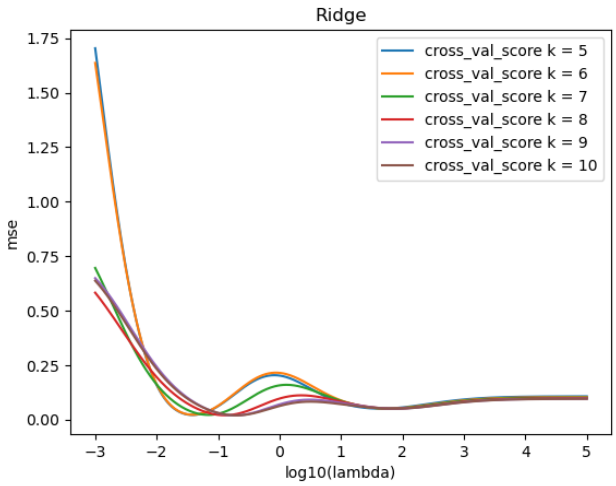
Figure 5: Error (MSE), variance and bias vs. model complexity.

3.6 Real data

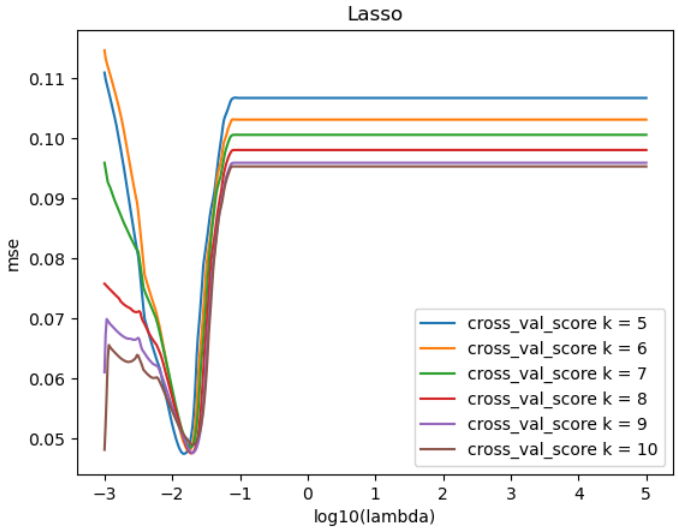
The method used when analysing the real data is the same as the one described above. See section 4 for the results of these analyses.



(a) MSE vs. Model Complexity using OLS



(b) MSE vs. lambda using Ridge Regression



(c) MSE vs. lambda using Lasso Regression

Figure 6: Comparison of MSE values for k-folds 5-10 using different regression models.

4 Results

The same code as described in appendix A was used when analysing the terrain data, with the alterations shown in appendix B, lst. 7. Due to limited computer power for the more costly analyses (mainly k-folds, but bootstrap also took it's toll), the terrain data had to be simplified somewhat more than I would have liked.

4.1 Ordinary Least Squares

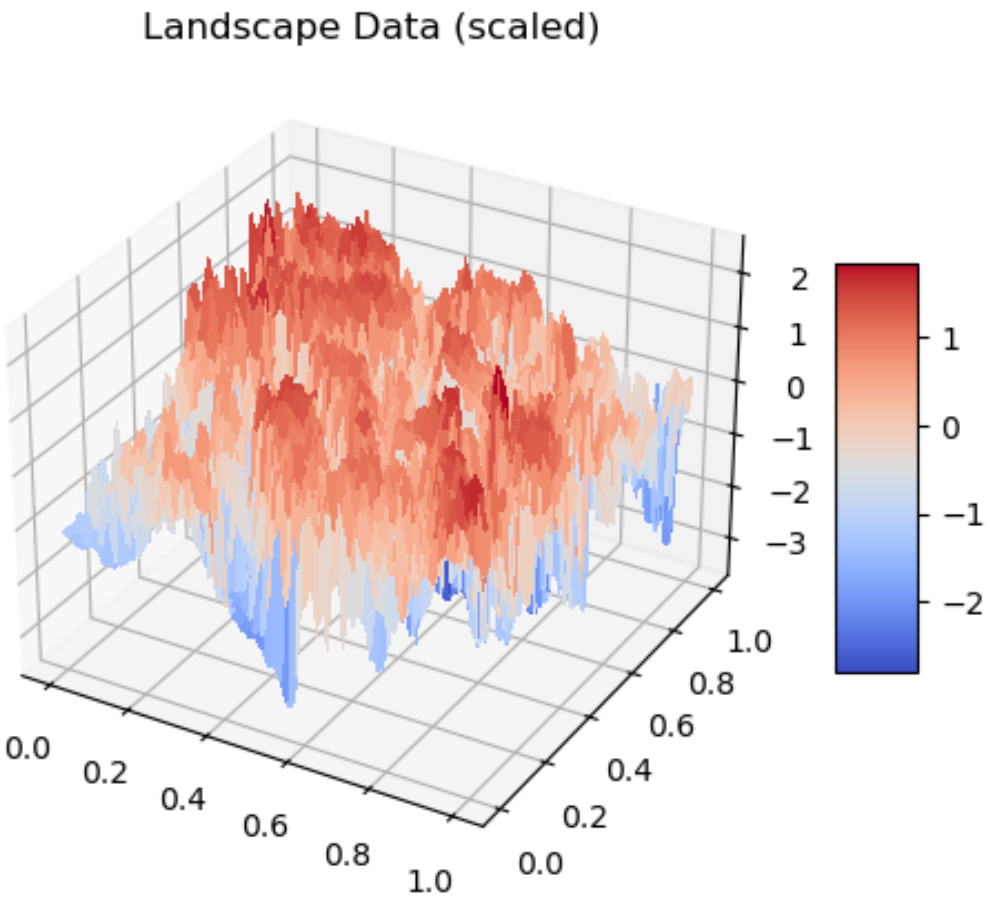
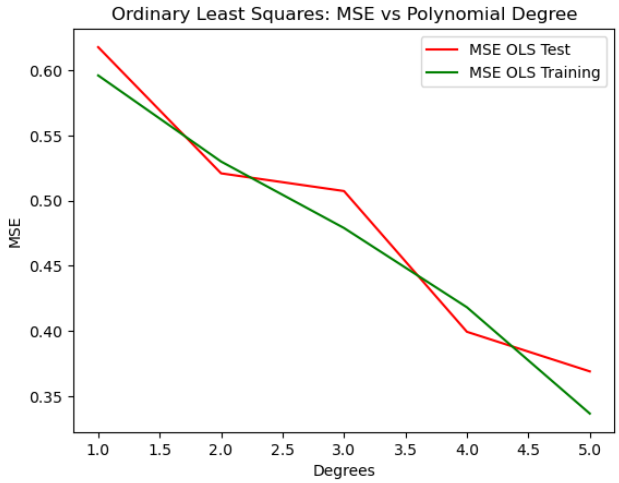
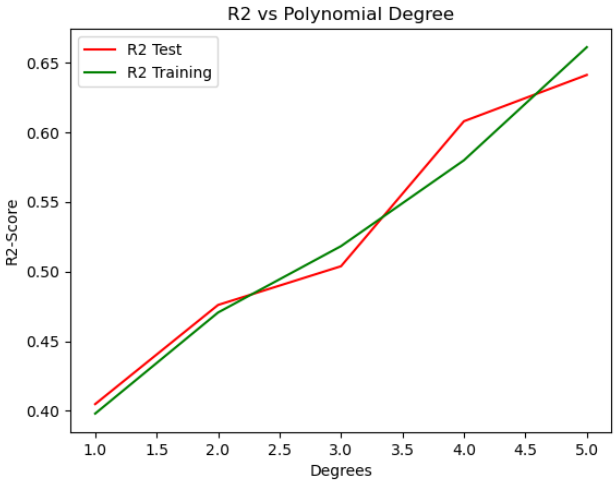


Figure 7: Surface plot of landscape data after scaling.

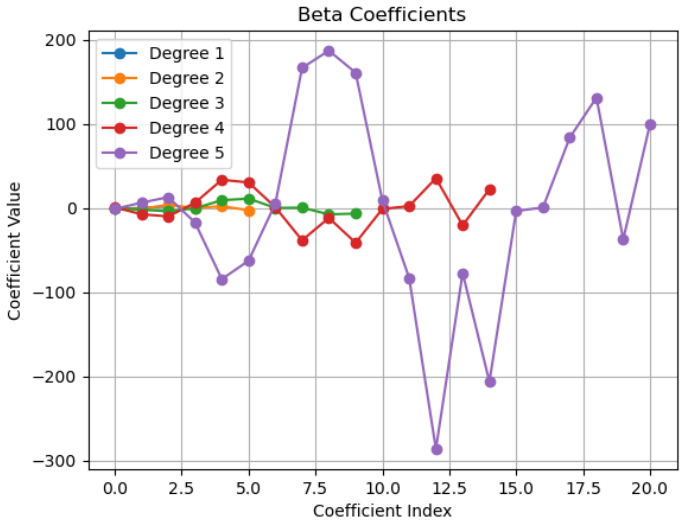
Using the same code as before (see chpt. 3.2) the landscape data was split into training and test sets, and MSE and beta-coefficients were plotted against the complexity of the polynomial, as seen in Fig. 8. Here, MSE and R2 show similar behaviour: MSE is almost 1 for the simplest polynomial, then improves to 0.8 for the 2nd order polynomial. The fit keeps improving for higher complexity, but only slightly.



(a) MSE vs. Model Complexity



(b) R2 vs. Model Complexity

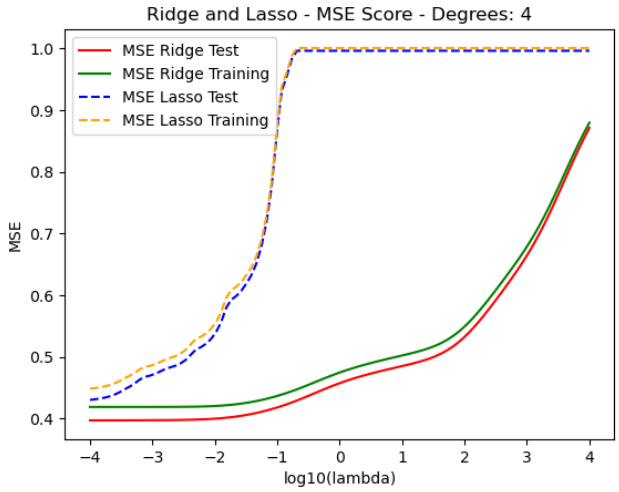


(c) Beta Coefficients for increasing complexity

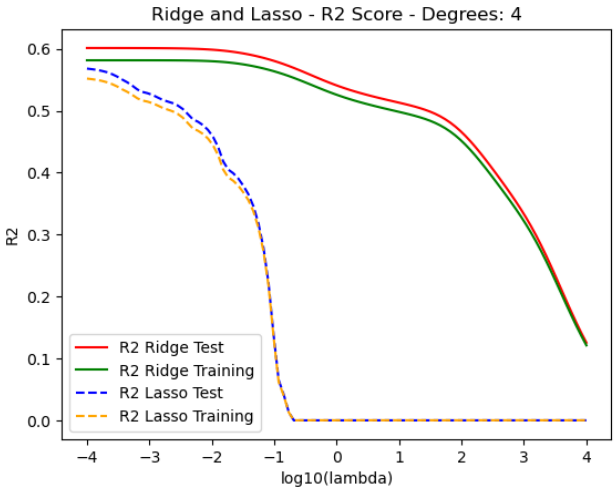
Figure 8: MSE and R2-scores plotted against model complexity. Figs. a and b show an improvement for higher order polynomials. The errors for test- and training sets follow each other. Fig. c shows the beta coefficients for the different polynomials. The values can be seen to vary more for higher degrees.

4.2 Ridge and Lasso Regression

The results from OLS fitting are shown in fig. 9 The error gets worse for higher lambdas. For Lasso, the fit gets suddenly worse as coefficients trend towards zero, while the change is more gradual for the Ridge fit.



(a) MSE vs. lambda for Ridge and Lasso



(b) R2 vs. lambda for Ridge and Lasso

Figure 9: Errors vs. lambda. Test and training errors are similar. The Ridge values show a gradual worsening for higher lambda values, while the Lasso values drop more quickly. The errors are better for lower values of lambda.

4.3 Bias-variance trade-off

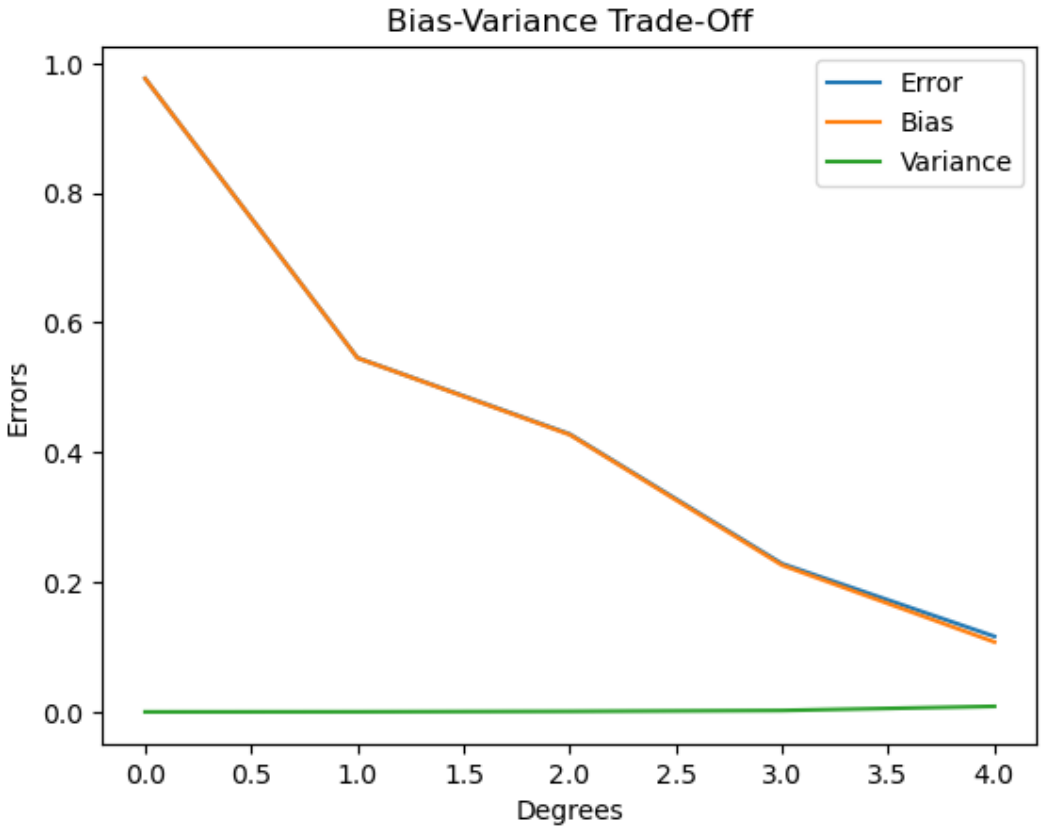
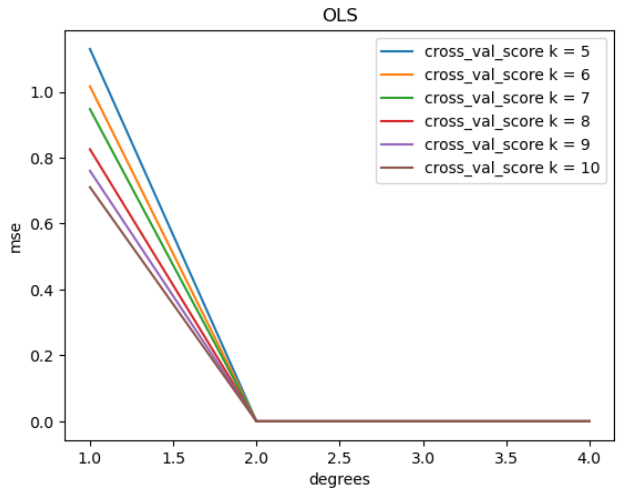


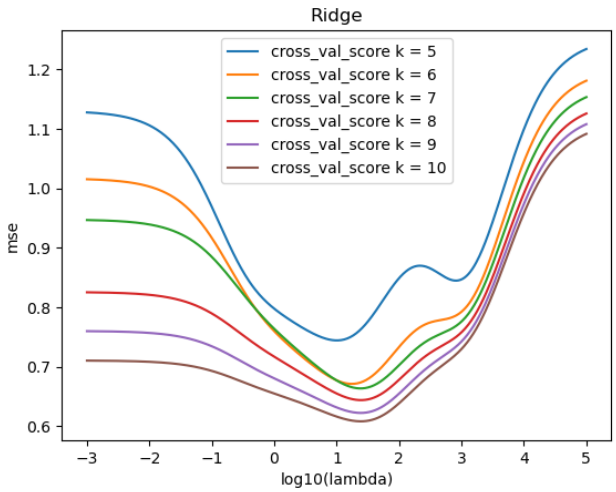
Figure 10: Error, bias and variance plotted against model complexity.

Fig. 10 shows error, bias and variance plotted against the model complexity. Error and bias follow each other closely, while variance shows an extremely slow increase for higher complexity. There is a significant improvement in error from low complexity to high.

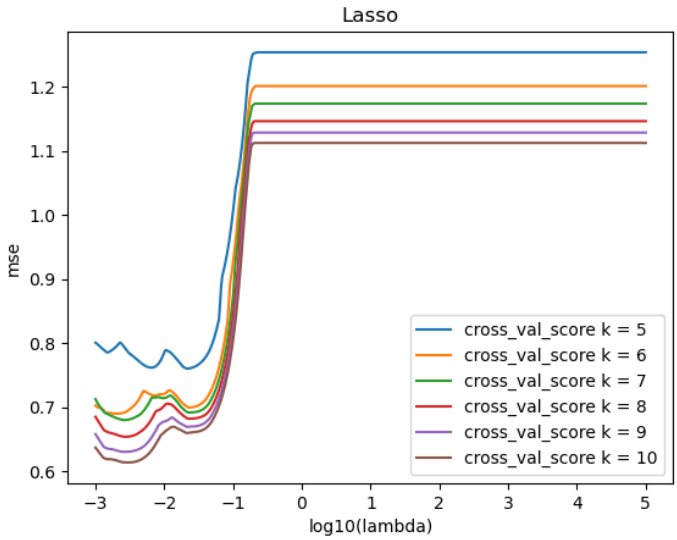
4.4 Cross-validation



(a) MSE vs. Model Complexity using OLS



(b) MSE vs. lambda using Ridge Regression



(c) MSE vs. lambda using Lasso Regression

Figure 11: Comparison of MSE values for k-folds 5-10 using different regression models. Polynomial degree = 3.

Fig. 11 shows mse-scores for different k-folds at degree = 3. There is some spread between the initial error-scores. Ridge shows an initial improvement as λ increases, before performing worse as λ continues to increase. For Lasso, the error gets sharply worse before flattening out, as expected.

5 Discussion

5.1 Scaling

The scaling was added in order to get MSE values of a reasonable size. As seen from eq. 1, the MSE-score depends on the size of the parameters, so large parameters give large MSE even for errors that would normally be considered small when compared to the initial data value. For this reason, the StandardScaler from scikit-learn's preprosessing library [9] was used. This scaler ensures data values centred around zero, as seen in fig. 7. As seen in fig. 8, this ensured MSE-scores between 1 and 0, making it possible to interpret the values.

5.2 Ordinary Least Squares

Unlike what we saw for the Franke data, the errors keep improving for higher complexity for both test and training sets. R2-scores and MSE follow a similar trajectory, suggesting the weighting by mean values seen in eq. 2 had no real impact on the error trend. This suggests there were no large outliers in the data, which is as expected as the changes in a terrain surface should be smooth. This behaviour was also seen in the results from the Franke data.

The error keeps improving for higher complexity, and from these results it is not possible to tell if the test error is becoming worse than the training (at degree = 5), or if this is a random dip similar to the one seen around degree = 3. The graph of beta coefficients show small values for degrees up to 3, before showing a slightly larger variability for the 4th degree and much greater variability for the 5th. Although it could have been interesting to see the behaviour of errors for some larger complexity, based on these results it seems unlikely to keep improving much after the 5th degree without resulting in over fitting.

5.3 Ridge and Lasso

Due to the limited computing power, degree = 4 was chosen for comparing the Lasso and Ridge errors, rather than attempting multiple runs for different polynomial degrees. As expected from the theory discussed in chpt. 2, there is a sharp change in error for Lasso as certain coefficients drop suddenly to zero, while there is a more gradual change in the Ridge errors. Both models perform best for λ close to zero. As coefficients are shortened to a greater degree, the error becomes noticeably worse. Indeed, for higher values of λ the errors are worse than the initial errors were in OLS. This suggests that the features dropped were important to the model, and should be included. As both Ridge and Lasso mirror OLS behaviour for lower λs, and this is where the errors are best, this suggests OLS best models the terrain data, as was also the case for the Franke test.

5.4 Bias-Variance Trade-Off

The results obtained by the bootstrap method provides an initially higher error for lower complexity, but keeps dropping for higher order polynomials, as seen in the previous results. Fig. 10 does not show an increase in bias and error as what was seen for the Franke test. Again, due to lack of computing power it was unfortunately difficult to run the model for higher order polynomials. But based on theory and the Franke test, it seems reasonable to assume that if the trend seen continues, a bottom point would be reached either at the 4th or 5th order, before error and variance increase.

5.5 K-folds Cross Validation Resampling

OLS shows a significant drop for all k-folds after the 2nd order polynomial. This error is better than what was seen in the initial splitting into one set of test data and one of training data. Lasso generally keeps performing worse for increasing λs until the error flattens out. This behaviour is similar to what was seen in the simple split into test- and training data. Ridge on the other hand

shows an improvement for a certain lambda (unlike the results from the simple splitting), but the errors are worse than from OLS. This suggests OLS is still the method that best described the data, but does suggest that splitting and running the data set through more iterations is a better approach than the simple, initial test. Presumably, some features of the data were not evenly divided into the two sets, while the repeated k-folds iterations ensured a more even spread.

6 Conclusion

Ordinary Least Squares was the best method to describe the terrain data, seen by the fact that Ridge and Lasso behaved best for the smaller lambda values. The results from the bias-variance and k-folds validation re-samplings suggest that the initial split into test and training sets may have resulted in an uneven split between significant data points.

For a polynomial order = 4, OLS with the data split simply into a test- and training set provided an MSE = 0.4 and R2 = 0.6, which was improved by running repeated iterations using the k-folds method. While the python functions worked well for the Franke Function test set, it became costly to run them for the terrain data set. The bootstrap- and k-folds analyses were so costly that the number of degrees and the resolution had to be reduced in order for the analysis to run smoothly.

In future, more pre-processing of the data, as well as some parallelisation of the code, may help with this. With better computation powers, I would try re-running the initial MSE-plot for higher degrees than 5 to look for signs of over fitting, as well as re-run the bias-variance bootstrap analysis to look for the expected increase in variance for higher complexity.

References

[1] J. Friedman T. Hastie, R. Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd edition, 2017.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] imageio. imageio.imread. https://pypi.org/project/imageio/, v2.35.1. Accessed: 25/09/2024.

[6] D. Bingham S. Surjanovic. Franke's function. https://www.sfu.ca/~ssurjano/franke2d.html, 2013. Accessed: 28/09/2024.

[7] Project 1 on machine learning. https://github.com/CompPhysics/MachineLearning/blob/master/doc/Projects/2024/Project1/ipynb/Project1.ipynb, 2024. Accessed: 28/09/2024.

[8] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[9] scikit learn. Standard scaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, v1.5.2. Accessed: 25/09/2024.

Appendices

A Testing using Franke's Function

Testing for a 2d surface was done using the Franke Function, implemented as shown in Listing 1.

```
import numpy as np

def frankes_function(x, y):
    z = ((3/4)*np.exp((-1/4)*(9*x-2)**2-(1/4)*(9*y-2)**2)
        + ((3/4)*np.exp((-1/49)*(9*x+1)**2-(1/10)*(9*y+1)))
        + ((1/2)*np.exp((-1/4)*(9*x-7)**2-(1/4)*(9*y-3)**2))
        - ((1/5)*np.exp(-(9*x-4)**2-(9*y-7)**2)))
    return z
```

Listing 1: Franke's function as implemented in the code.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from random import random, seed

x = np.arange(0, 1, 0.05)
y = np.arange(0, 1, 0.05) + np.random.normal(0, 0.1, x.shape)
x, y = np.meshgrid(x, y)
z = frankes_function(x, y)

z_noise = frankes_function(x, y) + np.random.normal(0, 0.1, x.shape)

#Plot (code taken from project description)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm, linewidth=0, antialiased=False)

fig.colorbar(surf, shrink=0.5, aspect=5)

plt.title("3d Surface: Franke's Function")
plt.show()

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

surf = ax.plot_surface(x, y, z_noise, cmap=cm.coolwarm, linewidth=0, antialiased=
    False)

fig.colorbar(surf, shrink=0.5, aspect=5)

plt.title("3d Surface: Franke's Function (with noise)")
plt.show()
```

Listing 2: Plotting a surface using Franke's function.

```
##Polynomials up to 5th order
def create_feature_matrix(x, y, degree):
    # Stack x and y into a feature matrix
    X = np.vstack((x.ravel(), y.ravel())).T

    # Create polynomial features
    poly = PolynomialFeatures(degree, include_bias=True)
    poly_features = poly.fit_transform(X)

    return poly_features
```

```
11
12 def MSE(z_data, z_model):
13     n = np.size(z_model)  # Number of data points
14     return np.sum((z_data - z_model)**2)/n
15
16
17 z = z_noise.ravel()
18 train_MSEs = []
19 test_MSEs = []
20 train_r2s = []
21 test_r2s = []
22 betas = []
23
24 degrees = range(1,6)
25 for order in degrees:
26     feature_matrix = create_feature_matrix(x, y, order)
27
28     X_train, X_test, z_train, z_test = train_test_split(feature_matrix, z, test_size
    =0.2)
29
30     #Matrix inversion
31     beta = np.linalg.pinv(X_train.T.dot(X_train)).dot(X_train.T).dot(z_train) #Pseudo
     inverse to handle singular matrices
32     betas.append(beta)
33
34     #Make the prediction
35     z_tilde = X_train @ beta
36     z_predict = X_test @ beta
37
38     #OLS
39     training_mse = MSE(z_train, z_tilde)
40     test_mse = MSE(z_test, z_predict)
41
42     train_MSEs.append(training_mse)
43     test_MSEs.append(test_mse)
44
45     print(f"Degree: {order}")
46     print(f"MSE (training): {training_mse}")
47     print(f"MSE (test): {test_mse}")
48
49     #R2
50     r2_training = error.r2_score(z_train, z_tilde)
51     r2_test = error.r2_score(z_test, z_predict)
52     print(f"r2 (training): {r2_training}")
53     print(f"r2 (test): {r2_test}")
54     print()
55
56     train_r2s.append(r2_training)
57     test_r2s.append(r2_test)
58
59 #Plot the mse vs. degrees
60 plt.figure()
61 plt.title("Ordinary Least Squares: MSE vs Polynomial Degree")
62 plt.plot(degrees, test_MSEs, 'r', label = 'MSE OLS Test')
63 plt.plot(degrees, train_MSEs, 'g', label = 'MSE OLS Training')
64
65 plt.xlabel('Degrees')
66 plt.ylabel('MSE')
67 plt.legend()
68 plt.show()
69
70 #Plot R2 vs. degrees
71 plt.figure()
```

```
plt.title("R2 vs Polynomial Degree")
plt.plot(degrees, test_r2s, 'r', label = 'R2 Test')
plt.plot(degrees, train_r2s, 'g', label = 'R2 Training')

plt.xlabel('Degrees')
plt.ylabel('R2-Score')
plt.legend()
plt.show()

for i in range(len(betas)):
    plt.plot(betas[i], marker='o', label=f'Degree {degrees[i]}')

# Adding plot details
plt.title('Beta Coefficients')
plt.xlabel('Coefficient Index')
plt.ylabel('Coefficient Value')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 3: Calculating MSE and r2-scores for up to 5th order polynomials.

```
# Ridge and Lasso Regression using matrix inversion
def Beta_ridge_regression(X, z, lmb):
    I = np.identity(X.shape[1])

    return np.linalg.pinv(X.T.dot(X) + lmb*I) @ X.T.dot(z)

def Calculate_y_ridge(X, beta):
    return X.dot(beta)

degree = 3
feature_matrix = create_feature_matrix(x, y, degree)
X_train, X_test, z_train, z_test = train_test_split(feature_matrix, z, test_size=0.2)

training_ridge_mses = []
test_ridge_mses = []
training_ridge_r2s = []
test_ridge_r2s = []
betas_ridge = []

training_lasso_mses = []
test_lasso_mses = []
training_lasso_r2s = []
test_lasso_r2s = []

lambdas = (0.0001, 0.001, 0.01, 0.1, 1.0)
nlambdas = 100
lambdas = np.logspace(-4, 4, nlambdas)
for i in range(nlambdas):
    lmb = lambdas[i]

    #Ridge
    beta_hat_ridge = Beta_ridge_regression(X_train, z_train, lmb)
    betas_ridge.append(beta_hat_ridge)
    z_tilde_ridge = Calculate_y_ridge(X_train, beta_hat_ridge)
    z_predict_ridge = Calculate_y_ridge(X_test, beta_hat_ridge)

    training_ridge_mses.append(error.mean_squared_error(z_train, z_tilde_ridge))
    test_ridge_mses.append(error.mean_squared_error(z_test, z_predict_ridge))

    training_ridge_r2s.append(error.r2_score(z_train, z_tilde_ridge))
    test_ridge_r2s.append(error.r2_score(z_test, z_predict_ridge))

```

```
    #Lasso
    RegLasso = linear_model.Lasso(lmb, fit_intercept=False)
    RegLasso.fit(X_train, z_train)
    y_tilde_lasso = RegLasso.predict(X_train)
    y_predict_lasso = RegLasso.predict(X_test)

    training_lasso_mses.append(error.mean_squared_error(z_train, y_tilde_lasso))
    test_lasso_mses.append(error.mean_squared_error(z_test, y_predict_lasso))

    training_lasso_r2s.append(error.r2_score(z_train, y_tilde_lasso))
    test_lasso_r2s.append(error.r2_score(z_test, y_predict_lasso))

# R2 Ridge/Lasso plots
plt.figure()
plt.title("Ridge and Lasso - R2 Score - Degrees: " + str(degree))

plt.plot(np.log10(lambdas), test_ridge_r2s, 'r', label = 'R2 Ridge Test')
plt.plot(np.log10(lambdas), training_ridge_r2s, 'g', label = 'R2 Ridge Training')

plt.plot(np.log10(lambdas), test_lasso_r2s, 'b--', label = 'R2 Lasso Test')
plt.plot(np.log10(lambdas), training_lasso_r2s, '--', color='orange', label = 'R2
    Lasso Training')

plt.xlabel('log10(lambda)')
plt.ylabel('R2')
plt.legend()
plt.show()

# MSE Ridge/Lasso plots
plt.figure()
plt.title("Ridge and Lasso - MSE Score - Degrees: " + str(degree))

plt.plot(np.log10(lambdas), test_ridge_mses, 'r', label = 'MSE Ridge Test')
plt.plot(np.log10(lambdas), training_ridge_mses, 'g', label = 'MSE Ridge Training')

plt.plot(np.log10(lambdas), test_lasso_mses, 'b--', label = 'MSE Lasso Test')
plt.plot(np.log10(lambdas), training_lasso_mses, '--', color='orange', label = 'MSE
    Lasso Training')

plt.xlabel('log10(lambda)')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

Listing 4: Calculating MSE and r2-scores using Ridge Regression.

```
#This code is mostly from lecture notes
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample

n = 500
n_bootstraps = 100
noise = 0.1
max_degrees = 5

```

```
errors = np.zeros(max_degrees)
biases = np.zeros(max_degrees)
variances = np.zeros(max_degrees)
polydegrees = np.zeros(max_degrees)
for degree in range(max_degrees):
    feature_matrix = create_feature_matrix(x, y, degree)

    z = z.ravel() # Flatten z into a 1D array for regression (from ChatGPT)
    X_train, X_test, z_train, z_test = train_test_split(feature_matrix, z, test_size
    =0.2)

    beta = np.linalg.pinv(X_train.T.dot(X_train)).dot(X_train.T).dot(z_train) #Pseudo
     inverse to handle singular matrices

    #Make the prediction
    z_tilde = X_train @ beta
    z_predict = X_test @ beta

    # Combine x transformation and model into one operation.
    # Not necessary, but convenient.
    model = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression(
    fit_intercept=False))

    # The following (m x n_bootstraps) matrix holds the column vectors z_pred
    # for each bootstrap iteration.
    z_pred = np.empty((z_test.shape[0], n_bootstraps))  # Adjust for 3D case

    # Perform bootstrapping
    for i in range(n_bootstraps):
        # Resample the training data
        X_resampled, z_resampled = resample(X_train, z_train)
        model.fit(X_resampled, z_resampled)
        # Fit the model on resampled data and evaluate on the same test data
        z_pred[:, i] = model.fit(X_resampled, z_resampled).predict(X_test).ravel()  #
     Prediction in 3D

    # Reshape z_test to match the dimensions of z_pred for broadcasting
    z_test_reshaped = z_test.reshape(-1, 1)  # Reshape to (80, 1)

    polydegrees[degree] = degree
    # Error calculation (mean squared error across all bootstrap samples and test
    points)
    error = np.mean(np.mean((z_test_reshaped - z_pred)**2, axis=1, keepdims=True))
    errors[degree] = error
    # Bias^2 calculation (bias as the difference between true values and the mean
    prediction)
    bias = np.mean((z_test_reshaped - np.mean(z_pred, axis=1, keepdims=True))**2)
    biases[degree] = bias
    # Variance calculation (variance of predictions across bootstrap samples)
    variance = np.mean(np.var(z_pred, axis=1,keepdims=True))
    variances[degree] = variance

    # Note: Expectations and variances taken w.r.t. different training
    # data sets, hence the axis=1. Subsequent means are taken across the test data
    # set in order to obtain a total value, but before this we have error/bias/
    variance
    # calculated per data point in the test set.
    # Note 2: The use of keepdims=True is important in the calculation of bias as
    this
    # maintains the column vector form. Dropping this yields very unexpected results.
    # error = np.mean( np.mean((z_test - z_pred)**2, axis=1, keepdims=True) )
    # bias = np.mean( (z_test - np.mean(z_pred, axis=1, keepdims=True))**2 )
    # variance = np.mean( np.var(z_pred, axis=1, keepdims=True) )
```

```
    print('Polynomial degree:', degree)
    print('Error:', error)
    print('Bias^2:', bias)
    print('Var:', variance)
    print('{} >= {} + {} = {}'.format(error, bias, variance, bias+variance))
    print()

plt.plot(polydegrees, errors, label='Error')
plt.plot(polydegrees, biases, label='bias')
plt.plot(polydegrees, variances, label='Variance')
plt.legend()
plt.show()
```

Listing 5: Bias-Variance trade-off with error.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

#Make data
n = 500
x = np.arange(0, 1, 0.05)
y = np.arange(0, 1, 0.05) + np.random.normal(0, 0.1, x.shape)

# x = np.linspace(-1, 3, n)
# y = np.linspace(-1, 3, n)

x, y = np.meshgrid(x, y)
z = frankes_function(x, y) #+ np.random.normal(0, 0.1, x.shape)

# #Test plot
# plot_3d(x, y, z, "Franke's Function Test Plot")

#Real data
#TODO: Smooth switch
# # Load the terrain - again from project description
# terrain = imread('SRTM_data_Norway_2.tif')

# N = 1000
# m = 5 # polynomial order
# terrain = terrain[:N,:N]
# # Creates mesh of image pixels
# x = np.linspace(0,1, np.shape(terrain)[0])
# y = np.linspace(0,1, np.shape(terrain)[1])
# x, y = np.meshgrid(x,y)

# z = terrain

#From lecture notes
k_folds = range(5,11)
degrees = range(1,5)

ols_k_mses = []
ridge_k_mses = []
lasso_k_mses = []
i = 0
for k in k_folds:
    kfold = KFold(n_splits = k)
```

```
50
51      #OLS
52      estimated_mse_ols = np.zeros(len(degrees))
53      j = 0
54      for degree in degrees:
55          # Decide degree on polynomial to fit
56          poly = PolynomialFeatures(degree = degree)
57
58          X = create_feature_matrix(x, y, degree)
59
60          z = z.ravel() # Flatten z into a 1D array for regression (from ChatGPT)
61          X_train, X_test, z_train, z_test = train_test_split(X, z, test_size=0.2)
62
63          ols = LinearRegression()
64          estimated_mse_folds = cross_val_score(ols, X, z[:, np.newaxis], scoring='
    neg_mean_squared_error', cv=kfold)
65          estimated_mse_ols[j] = np.mean(-estimated_mse_folds)
66
67          j += 1
68
69      ols_k_mses.append(estimated_mse_ols)
70
71      #Ridge and Lasso
72      # Decide which values of lambda to use
73      nlambdas = 500
74      lambdas = np.logspace(-3, 5, nlambdas)
75      #From lecture notes:
76      ## Cross-validation using cross_val_score from sklearn along with KFold
77      estimated_mse_ridge = np.zeros(nlambdas)
78      estimated_mse_lasso = np.zeros(nlambdas)
79      j = 0
80      for lmb in lambdas:
81          ridge = Ridge(alpha = lmb)
82          lasso = Lasso(alpha = lmb)
83
84          # cross_val_score return an array containing the estimated negative mse for
    every fold.
85          # we have to the the mean of every array in order to get an estimate of the
    mse of the model
86          estimated_mse_folds = cross_val_score(ridge, X, z[:, np.newaxis], scoring='
    neg_mean_squared_error', cv=kfold)
87          estimated_mse_ridge[j] = np.mean(-estimated_mse_folds)
88
89          estimated_mse_folds = cross_val_score(lasso, X, z[:, np.newaxis], scoring='
    neg_mean_squared_error', cv=kfold)
90          estimated_mse_lasso[j] = np.mean(-estimated_mse_folds)
91
92          j += 1
93
94      i += 1
95      ridge_k_mses.append(estimated_mse_ridge)
96      lasso_k_mses.append(estimated_mse_lasso)
97
98
99  ## Plots
100 #OLS
101 plt.figure()
102 i = 0
103 for estimate in ols_k_mses:
104     plt.plot(degrees, estimate, label = f"cross_val_score k = {k_folds[i]}")
105     i += 1
106 plt.xlabel('degrees')
107 plt.ylabel('mse')
```

```
108 plt.legend()
109 plt.title("OLS")
110
111 plt.show()
112
113 #Ridge
114 plt.figure()
115 i = 0
116 for estimate in ridge_k_mses:
117     plt.plot(np.log10(lambdas), estimate, label = f"cross_val_score k = {k_folds[i]}"
    )
118     i += 1
119 plt.xlabel('log10(lambda)')
120 plt.ylabel('mse')
121 plt.legend()
122 plt.title("Ridge")
123
124 plt.show()
125
126 #Lasso
127 plt.figure()
128 i = 0
129 for estimate in lasso_k_mses:
130     plt.plot(np.log10(lambdas), estimate, label = f"cross_val_score k = {k_folds[i]}"
    )
131     i += 1
132 plt.xlabel('log10(lambda)')
133 plt.ylabel('mse')
134 plt.legend()
135 plt.title("Lasso")
136
137 plt.show()
```

Listing 6: Calculating MSE by OLS, Ridge and Lasso using k-fold cross validation.

B Terrain data

```
terrain = imread('SRTM_data_Norway_2.tif')

N = 100
m = 5 # polynomial order
terrain = terrain[:N,:N]
# Creates mesh of image pixels
x = np.linspace(0,1, np.shape(terrain)[0])
y = np.linspace(0,1, np.shape(terrain)[1])
x, y = np.meshgrid(x,y)

z = terrain
```

Listing 7: New data set created for landscape values.