FYS-STK4155
Applied data analysis and machine learning
Project 1

Martin Krokan Hovden

October 9, 2019

Abstract

Regression methods are powerful methods for modelling linear and polynomial relationships between data. In this paper we have studied three regression methods; the Ordinary Least Squares regression, Ridge regression and Lasso regression. The models was first compared on the Franke function and later on terrain data taken from a region in Oslo. The differences in performance between the models are illustrated using the bias-variance trade-off, and the performance of each method was measured using k-fold cross validation. OLS turned out to be the best model for both cases. For the Franke function without noise it managed to obtain an test-MSE = 7.5e-3. For the terrain data it obtained an test-MSE = 2.1e-3.

INTRODUCTION

Regression methods have been around for over 200 years [5]. To this day, they are still popular methods in machine learning. They are used for prediction and interference in various applications ranging from biology to finance. Over the years, extensions of the original Ordinary Least Squares Regression model have been introduced. We will look at Lasso Regression and Ridge Regression, which is regularization methods that builds on the ordinary least squares method. In general, regression is used to find a relationship between a dependent variable and some independent variables. Regression can be used for both prediction and interference. For prediction the aim is to train a model that can accurately predict output from new test data. For interference, the prediction accuracy are not the main goal. Instead we want to see how the response variable varies as a function of the dependenent variables.

In this report we will focus on prediction. We will study the prediction-accuracy of the models on two different data sets. The first data set is the Franke function. Adding noise to the data will help illustrating important concepts in machine learning. After that, we will use the same models to study terrain data taken from a region in Oslo. For both data sets we have looked closely at the bias-variance trade-off for the different methods. This is important to understand the differences between the models. The different parts of the report are based on Project 1 in the subject FYS-STK 4155 at the University of Oslo.

The rest of the report is structured in two main parts. First, the theory behind the methods and error estimation will be presented. We will also look at how the methods can be implemented in python. After that we will look at the results from the code implemented in python, and study how the different models behave on different data sets.

THEORY AND METHODS

The theory and methods part are based on the book "Elements of statistical learning" by Hastie et al[1] and "Hands-On Machine Learning with Scikit-Learn" by Aurélien Géron [3].

We will start by applying the methods to the Franke function

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

The function is an exponential function, so it might seem hard to estimate with a polynomial. However, by studying the function in the domain $[0,1]x[0,1]$, we will see that the approximations will be quite good. Most of the report will be based on function values on a 50x50 grid in the given domain. However, we will also take a look at how the models behaves with different grid sizes. After that, we apply them to a real world example where we will test our methods on terrain data. The terrain data are given as a matrix, where each element are the height of the terrain at the given coordinate. The data are collected from `https://earthexplorer.usgs.gov/`. The exact data-file can be found at the GitHub-address: `https://github.com/MartinKHovden/FYS-STK4155-project-1`. Due to the large size of the terrain data, we have to study a small part of the data set. This is so that we are able to see the effects we are looking for.

The first method we will use is the ordinary least squares method. This is one of the simplest methods in machine learning. However, it works surprisingly well in many situations.

Ordinary least square methods

The ordinary least square method is a popular method for solving machine learning problems. It can be used to model both linear and polynomial relationships in the data. With ordinary least square regression the aim is to find a relationship between a response variable and a set of corresponding predictor variables. The goal is to be able to predict the outcome, given some input, and study how the explanatory variables effect the response variable. We try to find a model for the expected value of the response variable, given some covariates.

$$\mathrm{E}[y|x_1, x_2, ..., x_n] \tag{2}$$

We assume that our data follows a function

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{X}) + \boldsymbol{\epsilon}, \tag{3}$$

where $\boldsymbol{f}$ is a non-random function of the predictors and $\boldsymbol{X}$ is a matrix with data points. We call $\boldsymbol{X}$ the design matrix. $\boldsymbol{y}$ is a vector of response variables. Our goal is to find an approximation of $\boldsymbol{f}$. We will start by looking at the linear regression case. In linear regression, we assume a linear relationship between the response variable and the covariates, so a natural choice is to try to find a function $\boldsymbol{f}$ such that

$$\boldsymbol{y} = \beta_0 + \boldsymbol{x_1}\beta_1 + ... + \boldsymbol{x_p}\beta_p + \epsilon \tag{4}$$

where β_i is called the regression coefficients and $\boldsymbol{x_i}$ is a vector containing samples of feature i. β_0 is called the bias term. This expression means that by increasing x_i by one, we get an increase of β_i in $\boldsymbol{y}$. We assume that

$$\boldsymbol{\epsilon} \sim N(0, \sigma^2). \tag{5}$$

Since $\boldsymbol{f}$ is a non-stocastic variable, it then follows that

$$\boldsymbol{y} = \boldsymbol{f} + \boldsymbol{\epsilon} \sim N(0, \sigma^2). \tag{6}$$

Let's now say that our data are placed in a n x (p+1) matrix X:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & ... & ... & x_{1p} \\ 1 & x_{21} & x_{22} & ... & ... & x_{2p} \\ 1 & ... & ... & ... & ... & ... \\ 1 & x_{n1} & x_{n2} & ... & ... & x_{np} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x_1} & \boldsymbol{x_2} & ... & ... & \boldsymbol{x_p} \end{bmatrix} \tag{7}$$

The column of ones are included to add the bias term mentioned earlier. We will define x_{ij} as the i'th sample of the j'th predictor. Since OLS is a supervised method, we also have to have a set of target values for each sample. We define the vector $\boldsymbol{y}$ as the vector of target values.

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ ... \\ y_n \end{bmatrix} \tag{8}$$

where y_i is the target-value for the i'th sample. We then have that

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{9}$$

We can easily extend the linear regression to polynomial regression by expanding the basis. By adding a component wise power of a feature vector, $\boldsymbol{x_i^j}$ for some $i, j \in \mathcal{R}$, to $\boldsymbol{X}$, we can also get a polynomial relationship between the response and of variable i. By adding for example $\boldsymbol{x_j x_i}$ we can add interactions between variable j and i. We can then use the same method for training polynomial regression as for linear regression, only with an extended design matrix $\boldsymbol{X}$. This makes the ordinary least squares regression able to model more complex data. From now on we will write the estimate of $\boldsymbol{\beta}$ as $\boldsymbol{\beta}$. Eq. 6 can then be approximated by

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta} \tag{10}$$

where $\boldsymbol{\beta}$ is a vector that contains the regression coefficients,

$$\boldsymbol{\beta} = (\beta_0, \beta_1, ..., \beta_n)^T \tag{11}$$

The goal is then to find the unknown $\boldsymbol{\beta}$. This is an optimization problem, which is solved by minimizing the residual sum of squares, RSS, as a function of $\boldsymbol{\beta}$. The easiest way to do this is by writing the minimization problem as an minimization of an matrix equation.

$$\mathrm{RSS}(\beta) = \sum_i (y_i - \hat{y}_i)^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}). \tag{12}$$

Taking the derivative with respect to β, and setting equal to zero

$$\frac{\partial \mathrm{RSS}(\beta)}{\partial \beta} = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) = 0. \tag{13}$$

By rearranging the equation, the expression for the coefficients are $\boldsymbol{\beta}$:

$$\boldsymbol{\beta}_{\mathrm{OLS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{14}$$

This can be calculated in python with the following code-snippet:

```
def OLS_beta(design_matrix, target_values):
    """
    Returns the beta for the ordinary least squares.
    """
    return np.linalg.inv(design_matrix.T.dot(design_matrix)).dot(design_matrix.T).dot(target_values)
```

In the code, we have accounted for one of the main problems with fitting ordinary least squares. This occurs when we encounter singular or nearly singular matrices. We then get problems with the inversion. One way to fix this problem is by using regularization techniques that will be discussed later. However, in the code we have fixed the problem by using numpy.pinv instead of numpy.inv. numpy.pinv uses a singualar value decomposition and calculates the so-called pseudo-inverse. For more information about the techniques, see [1].

Since the betas have some uncertainty, we are unable to know the exact value of each coefficient. Therefore it is interesting to look at the confidence intervals for the coefficients. We start by finding the variance for each coefficient with the formula

$$\mathrm{Var}(\boldsymbol{\beta}_{\mathrm{OLS}}) = \mathrm{Var}((\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}) = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\mathrm{Var}(\boldsymbol{y})((\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T)^T = \sigma^2(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}. \tag{15}$$

The last equality comes from the fact that $\mathrm{Var}(y) = \mathrm{Var}(\boldsymbol{f}) + \mathrm{Var}(\boldsymbol{\epsilon}) = \sigma^2$, since $\boldsymbol{f}$ is non-stochastic and the variance of $\boldsymbol{\epsilon}$ is σ^2. The variance can be implemented in python by

```
sigma_2 = (1./(n-p-1))*sum((y - y_hat)**2)
beta_variance = sigma_2*np.diag(np.linalg.inv(np.dot(X.T, X)))
```

We can see that the distribution of the betas are given by

$$\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{\beta}^{\mathrm{exact}}, \sigma^2(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}). \tag{16}$$

Now that we know the variance of each coefficient, we can create confidence intervals for each one of them. The variance of the coefficients are found on the diagonal of the covariance matrix in equation 16. The 95% confidence interval for coefficient i is given by

$$\left[\beta_i - 1.96\sqrt{\mathrm{Var}(\beta_i)}, \beta_i + 1.96\sqrt{\mathrm{Var}(\beta_i)}\right] \tag{17}$$

Ridge regression

Ridge regression is an extension of the simple OLS-model. It adresses one of the main problems with OLS, which is the occurence of non-invertible (X^TX). We will also look at Lasso regression which is another example of methods that builds on the OLS-model. These are regularization techniques, and are also used to reduce the variance of the model. By adding a restriction to the coefficients, we are able to reduce the degrees of freedom, and thereby the variance. By reducing the coefficients in the regression, the goal is to reduce the variance, while still keeping the bias low.[1] This is done by adding a penalty parameter to the minimisation problem. The minimization problem now becomes

$$\hat{\beta}_{\mathrm{ridge}} = \mathrm{argmin}_\beta \left(\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right) \tag{18}$$

[1] One important part of this equation is the exclusion of the intercept β_0 in the last term. This problem can be rewritten to matrix notation, so we see that 18 is the same as minimizing

$$(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}. \tag{19}$$

with respect to $\boldsymbol{\beta}$, where λ is a regularization parameter. By setting λ to zero, we get the normal least squares estimator. By increasing λ we can shrink the coefficients. λ can therefore be used to control the size of the coefficients. To find the optimal β, we take the derivative of equation 19, and find the value of $\boldsymbol{\beta}$ that gives the derivative equal to zero.

$$-2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + 2\lambda\boldsymbol{\beta} = 0 \tag{20}$$

By rearranging the equation, the least-squares estimator becomes

$$\hat{\boldsymbol{\beta}}_{\mathrm{ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{21}$$

where $\mathbf{I}$ is the identity matrix. This can be implemented in python with the following code

```
beta_ridge = np.linalg.inv(design_matrix.T.dot(design_matrix) + \
    lam*np.eye(len(design_matrix[1,:]))).dot(design_matrix.T).dot(target_values)
```

We can find the variance of the coefficients by using a similar technique as for OLS.

$$\begin{aligned}
\mathrm{Var}(\hat{\boldsymbol{\beta}}_{\mathrm{ridge}}) =& \mathrm{Var}((\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}) \\
=& (\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T\mathrm{Var}(\boldsymbol{y})((\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T)^T \\
=& \sigma^2(\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T((\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T)^T \\
=& \sigma^2(\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1}\boldsymbol{X}^T\boldsymbol{X}((\boldsymbol{X}^T\boldsymbol{X} + \lambda\mathbf{I})^{-1})^T
\end{aligned} \tag{22}$$

This can be implemented in python with the following code

```
beta_var = np.diag(sigma_2*(np.linalg.inv(X.T@X +
    self.lam*np.eye(len(X[1,:])))@X.T@X@(np.linalg.inv(X.T@X+self.lam*np.eye(len(X[1,:])))).T))
```

We see from the equation that the variance of the coefficients decrease when we increase lambda. Similar to what was done for OLS, we can find confidence intervals for each coefficient. The 95% confidence interval are given by

$$\left[\beta_i - 1.96\sqrt{\text{Var}(\beta_i)}, \beta_i + 1.96\sqrt{\text{Var}(\beta_i)}\right] \tag{23}$$

Now that we have expressions for the variance of the coefficients for both OLS and ridge, we can look at the difference between the two of them.

$$\text{Var}(\beta_{\text{OLS}}) - \text{Var}(\beta_{\text{ridge}}) = \sigma^2[X^TX + \lambda I]^{-1}[2\lambda I + \lambda^2(X^TX)^{-1}][X^TX + \lambda I]^T \tag{24}$$

This expression is always positive, which means that as long as $\lambda > 0$, the variance of the coefficients for OLS are larger than for Ridge.[1]

When finding the optimal complexity for ridge regression, there are now to parameters that have to be tuned. We have to look at both the value of lambda and the polynomial degree.

Lasso regression

Lasso regression is another regularization method. It is quite similar to the ridge regression, but uses the ℓ_1-norm in the penalty term. The minimisation problem is now given by

$$\hat{\boldsymbol{\beta}}_{\text{lasso}} = \text{argmin}_\beta \left(\frac{1}{2}\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right). \tag{25}$$

[1]
This gives us an challenge when we try to find the optimal $\boldsymbol{\beta}$. We are no longer able to find an analytical solution, so we have to use numerical optimization techniques.[3] One of the easiest methods is the gradient descent method. The gradient descent method minimizes the MSE by repeatedly finding the gradient of the cost function with respect to $\boldsymbol{\beta}$, and then move in the opposite direction of the gradient for some distance. This distance is called the learning rate. This have to be chosen carefully for the algorithm to converge in reasonable time. One way to do this is to test for different fixed values and find out which one is best. Line-search methods are an alternative, where the optimal learning rate are found for each gradient.[4] Scikit-learn have this functionality built in. Since lasso dont have a closed form solution to the beta problem, it is a bit more tricky to implement. For this report we have therefore used Scikit learns functionality to fit the Lasso regression. Predictions for the Lasso regression in python can be implemented with the following code

```
from sklearn.linear_model import Lasso
Lasso_reg = Lasso(alpha = lambda_value)
Lasso_reg.fit(design_matrix, targets)
predictions = Lasso_reg.predict(desing_matrix_2)
```

The main difference when it comes to the regression coefficients we get, are that lasso are able to make some of the coefficients equal to zero. This means that Lasso can actually be used for feature selection, and are able to remove features that are not relevant to the regression.[1]

Comparison of time used for fitting model. Sklearn vs. own code.

It is interesting to see how our "home-made models" compare to sklearn with regards to time. In figure 1 and 2 the difference are shown. For OLS, the home-made version is quite a lot faster than the one in sklearn for fitting the data. However, for ridge regression, sklearn beats our code.

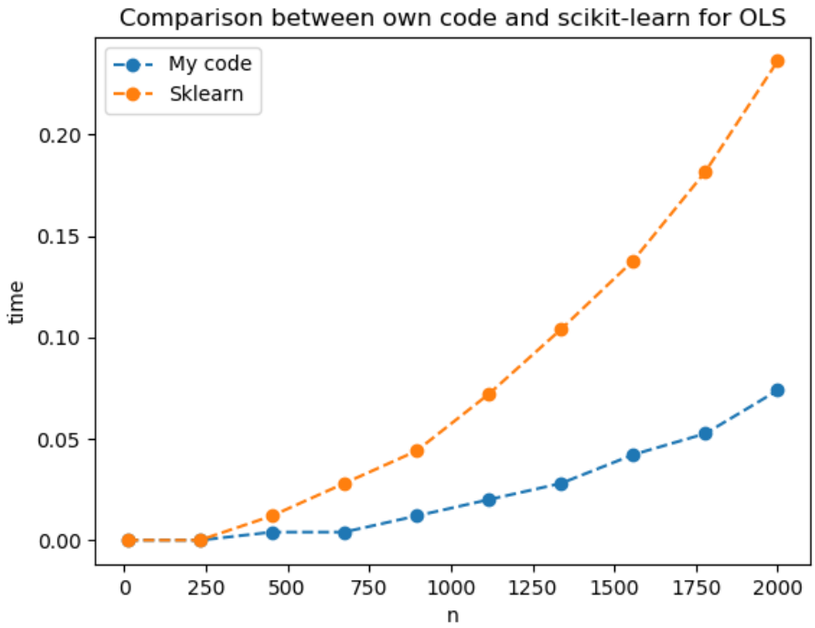

Figure 1: Comparison of time used for fitting the OLS models on the Franke function for different matrix sizes n.

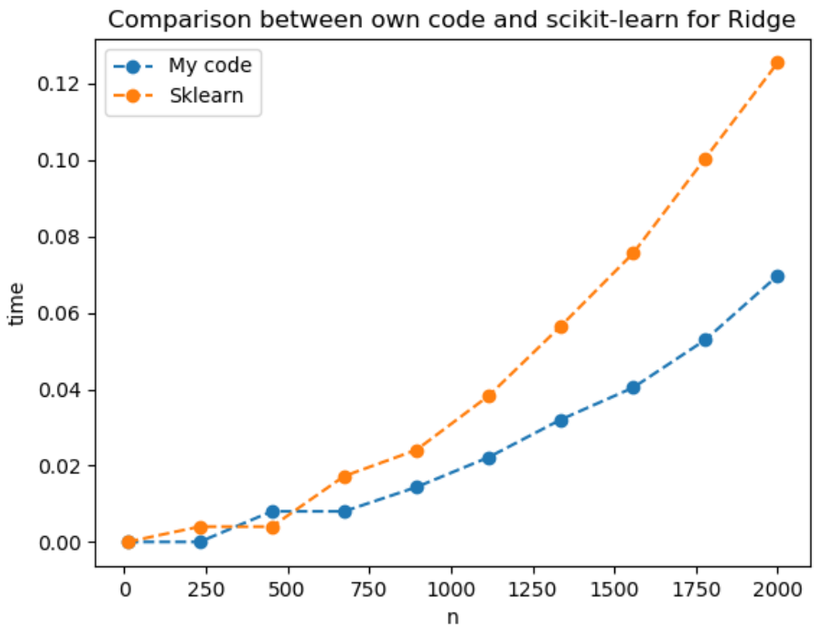

Figure 2: Comparison of time used for fitting the Ridge models on the Franke function for different matrix sizes n.

Comparing the models

To compare the different models, we have to use an error-estimate. For regression type problems, we typically use the mean squared error, denoted by MSE.[1] The mean squared error measures the mean of the squared errors, where the errors is the difference between the predicted value and the actual value. The MSE says something about how good we are able to model the data. The lower the score, the better the model. The MSE can be written as

$$MSE = \mathrm{E}[(y - \hat{y})^2] = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2 \tag{26}$$

where y is a vector of the actual target values, and $\hat{y}$ is a vector of the models predicted values. The MSE-error can be implemented in python with the following code:

```
MSE = sum((y_pred - y)**2)*(1./n)
```

If the model interpolates all the data points, the MSE is equal to zero. This means that zero MSE is equal to perfect fit.

Another measure that can be used to compare the models are the R^2-value, also called the coefficient of determination. The R^2-value is a measure of how much of the variance in the data we are able to explain with our regression.[1] The R^2-score goes from zero to one, and we want the score to be as close to one as possible. If we get an R2-value that is close to one, it is able to explain a lot about the variability in the data. This probably

means that we have chosen the right complexity. If its close to zero, it might be evidence that we use the wrong complexity. For example that we are trying to fit highly polynomial data with a linear model. There might also be that there is a lot of noise in the data, so that the so called irreducible error is high. The function for calculating the R^2-value are given by

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)}{\sum_i (y_i - \bar{y})} \tag{27}$$

and can be implemented in python with the following code:

```
R2 = 1 - (sum((y - y_pred)**2)/sum((y - y_bar)**2))
```

There are different ways to measure those values. We can for example do it on the training data, and find the training MSE and the training R2. This is however not the usual way to do it. We are more interested in the generalization error or the test error, instead of the training error. The test error is measured on a new and independent data set, which the model has not been trained on. We will very often overestimate the models capabilities for prediction if we use the training error as the measurement of goodness of fit. This is because that by increasing the complexity of the model, we can almost get the line to interpolate our data. But this is not necessarily a good thing, if we try to do predictions on a new data set. Therefore, we use the test error to choose our model. One way to measure the test error is to split the data into three parts; a training part, a test part and a validation part. However, we often do not have access to large amounts of data. By splitting it up in many parts, we get less data to train our model on, and the fit is typically worse than if used more data. Instead we can use resampling techniques. These techniques are able to estimate the test error for smaller sets of data.

Resampling methods

Resampling methods is a way for estimating the test-error of a model. One way to do this is by a cross-valdiation. The idea is to randomly split our data into folds, leave one of the folds as a unseen test-set, and then train our model on the remaining folds of data.[1] We then calculate the MSE for the test data, and repeat the procedure, but now choosing another fold as test data. Taking the mean of the k MSE's gives the CV-error. This can be summarized in the following list:

1. Split the data into k-folds. It is important that this split is random.
2. Leave one of the folds out and train the model on the remaining folds.
3. Estimate the test-error on the left-out fold.
4. We repeat step 2 and 3 k times for different fold, and calculate the prediction error for each.
5. Take the mean of the previously measured prediction errors. This is the CV-error.

The cross-validation error can be seen as an estimate of how well we can expect our model to perform on unseen data.
The k-fold cross validation can be implemented in python with the following code

```
k_folds = KFold(n_splits = num_folds,shuffle=True)
fold_score = np.zeros(num_folds)
i = 0
for train_index, test_index in k_folds.split(data):
    x_train = data[train_index]
    x_test = data[test_index]
    y_train = response[train_index]
    y_test = response[test_index]
    model.fit(x_train, y_train)
    y_pred = model.predict(x_test)
    fold_score[i] = MSE(y_pred, y_test)
    i+=1
return np.mean(fold_score)
```

Typical values of k are 5 and 10. In this report we will use k=5. We have to keep in mind that the choice of k leads to different bias and variance of the estimator. By choosing 5 we decrease the variance compared to 10, but we might get some more bias.[1]

Bias-Variance trade-off

One of the problems with fitting machine learning models, is the choice of complexity. The complexity of a model can for example be the degree of polynomial we fit the data with in a polynomial regression. This problem can be summarized in the bias-variance trade-off. Training error will often underestimate the true error, because it uses the same data both for fitting and for validation.[1]

We can use the mean-squared error, MSE, to get an idea about how good our model is. The MSE is given by

$$MSE = \boldsymbol{E}[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2]. \tag{28}$$

This can be used to study the bias-variance trade-off. This is an important effect we have to be aware off when fitting a model. I will start with a rewriting off the MSE, and then explain the effects of the bias-variance trade-off. We see that we can write

$$MSE = \boldsymbol{E}[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2] = \frac{1}{n}\sum_i (f_i - E[\hat{\boldsymbol{y}}])^2 + \frac{1}{n}\sum_i (\hat{y}_i - E[\hat{\boldsymbol{y}}])^2 + \sigma^2. \tag{29}$$

The expression can be derived by using that

$$\boldsymbol{y} = \boldsymbol{f} + \boldsymbol{\epsilon} \tag{30}$$

By adding and subtracting $\mathrm{E}(\hat{y})$ we get that

$$\begin{aligned} \mathrm{E}[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2] =& \mathrm{E}[((\boldsymbol{y} - \mathrm{E}(\hat{\boldsymbol{y}})) - (\hat{\boldsymbol{y}} - \mathrm{E}(\hat{\boldsymbol{y}})))^2] \\ =& \mathrm{E}[(y - \mathrm{E}(\hat{y}))^2] - 2\mathrm{E}[(y - \mathrm{E}(\hat{y}))(\hat{y} - \mathrm{E}(\hat{y}))] + \mathrm{E}(\hat{y} - \mathrm{E}(\hat{y})^2) \\ =& \mathrm{E}[(y - \mathrm{E}(\hat{y}))^2] + \mathrm{E}[(\hat{y} - \mathrm{E}(\hat{y}))^2] \end{aligned} \tag{31}$$

The last step is because

$$2\mathrm{E}[(y - \mathrm{E}(\hat{y}))(\hat{y} - \mathrm{E}(\hat{y}))] = 0 \tag{32}$$

since the two factors of the equation are independent, and

$$\mathrm{E}(\hat{y} - \mathrm{E}(\hat{y})) = 0. \tag{33}$$

Further, we can show that

$$\mathrm{E}[(\hat{y} - \mathrm{E}(\hat{y}))^2] = \mathrm{Var}(\hat{y}) = \frac{1}{n}\sum_i (\hat{y}_i - \mathrm{E}(\hat{y}))^2. \tag{34}$$

By showing that

$$\begin{aligned} \mathrm{E}[(y - \mathrm{E}(\hat{y}))^2] =& \mathrm{E}[(f + \epsilon)^2 - 2(f + \epsilon)\mathrm{E}(\hat{y}) + \mathrm{E}(\hat{y})^2] \\ =& \mathrm{E}[(f^2 - 2f\mathrm{E}(\hat{y}) + \mathrm{E}(\hat{y})^2) + 2f\epsilon + \epsilon^2 - 2\epsilon\mathrm{E}(\hat{y})] \end{aligned} \tag{35}$$

Since

$$\mathrm{E}(f^2 - 2f\mathrm{E}(\hat{y}) + \mathrm{E}(\hat{y})^2) = \mathrm{E}((f - \mathrm{E}(\hat{y}))^2) = \frac{1}{n}\sum_i (f_i - \mathrm{E}(\hat{y}))^2 \tag{36}$$

we can use that

$$\mathrm{E}(2f\epsilon) = \mathrm{E}(\epsilon\mathrm{E}(\hat{y})) = 0 \tag{37}$$

and

$$\mathrm{E}(\epsilon^2) = \mathrm{Var}(\epsilon) + \mathrm{E}(\epsilon)^2 = \mathrm{Var}(\epsilon) = \sigma^2 \tag{38}$$

to get

$$\mathrm{E}[(y - \hat{y})^2] = \sigma^2 + \frac{1}{n}\sum_i (f_i - \mathrm{E}(\hat{y}))^2 + \frac{1}{n}\sum_i (\hat{y}_i - \mathrm{E}(\hat{y}))^2 \tag{39}$$

The equation is a sum of an irreducible part and a reducible part. The irreducible part is the part of the error that we can not remove.[1] This part is "built" in to the data, and can for example be a result of failing equipment for measuring. The irreducible error is represented by σ^2. The reducible part is a result of model choice, and we can reduce it by tweaking the model.[1] The reducible part consist of bias and variance. The bias is the second term in equation 39 and the variance are the third term. The bias comes from wrong assumptions about the model, while variance comes from how sensitive the model is to changes in the data. This is quantities that can be tweaked by choosing the correct model complexity. When fitting a model, we have do a trade-off between bias and variance. When we increase the complexity, the model we get are very data-dependent. By changing the training data-set, our model can change drastically. The model will vary a lot if we change the training data. This means that when we test our model on the test set, it might be to specialized for the training data, and will not give a good prediction. However, when the complexity is this high, the model is free to fit to the shape of the training data, so the bias is low. On the other hand, when we have a low model complexity, we get high bias and low variance. The high bias comes from the fact that we might try to model complex data with for example a linear model.

We can avoid overfitting by using more data when fitting the model. We will see that the test error improves when we add more and more data to the model. Overfitting occurs when we tune our model to well to the training data, so that it does not work well on unseen data.

Preprocessing

There is often useful to scale the data before we do a regression. Especially for the lasso regression, which uses a gradient descent, we can often improve convergence by scaling [4]. Since we want to compare the methods, it will be useful to scale the data for all types of regression, to get a meaningful relationship between the different methods. We could also have rescaled the data back by saving the scaling factors. However, in this report we are not that interested in the exact scale of the target values. Instead we focus on how got our model fit the data. Scaling is also useful to avoid outliers. Scaling can be done by subtracting the means and dividing by the standard deviation of the data.[3] Implementation of scaling in python has been done using the built in Sklearn function scale. It was done before fitting and resampling.

```
from sklearn.preprocessing import scale
X = scale(X)
z = scale(z)
```

However, another, and probably better idea,is to scale both the training part and the test part after the splitting of the data, with the mean and std. of the training part. This way we will get the correctly scaled data fitted. If we scale all the data first and then pick out a training part, this is no longer necessarily scaled. This might be a good idea to try to implement at a later stage. The results seemed fine for the way I did it, but one might think that the MSE's could be improved further by doing the scaling the correct way.

RESULTS AND DISCUSSION

All code used for producing the different plots can be found at the GitHub-address: `https://github.com/MartinKHovden/FYS-STK4155-project-1`. In the project1_lib-file you will be able to find the functions that are used. In the caption of the pictures, it is stated what methods are used, and for what value. The other files contains code that are used to answer the different parts of the project mentioned earlier. For the problems we will study, I find it most appropriate to present the data in plots. This is to make it easier to see how the errors vary with the complexity. The exact values are not the most import, but rather the difference between the values and how they behave when tuning the different parameters. As mentioned earlier, we will scale all our data so that it has mean = 0 and std = 1. This is done as explained in the theory part.

OLS on the Franke function

We will start by looking at the Franke function and the training error for polynomial regression of degree up to ten. The Franke function can be seen in figure 3. We will use 20 data points in each direction, so that the data are represented on a 20x20 grid. This will make it easier to study the effects of the bias-variance trade-off and overfitting.

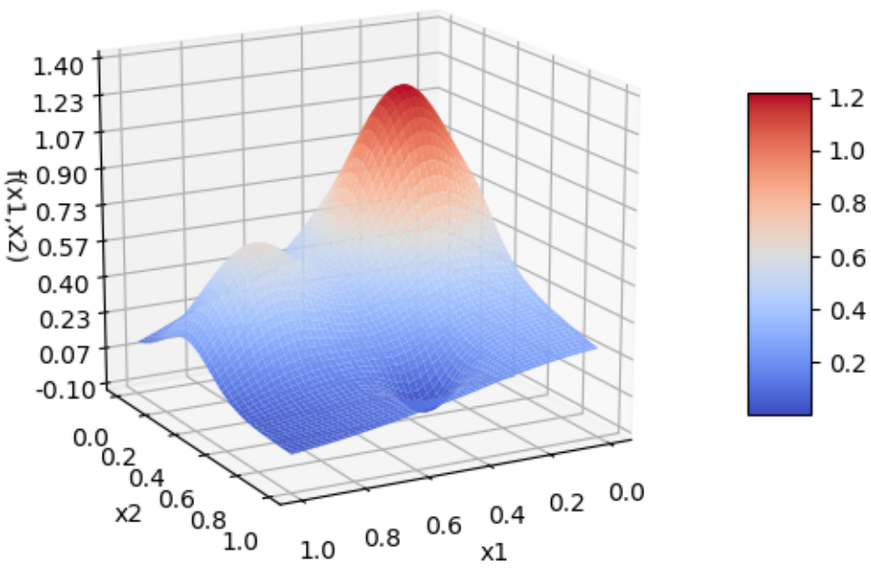

Figure 3: Franke function on the domain [0,1]x[0,1] with 20 points in each direction

From the discussion above we know that the training error should decrease when we increase the polynomial degree. By looking at the Franke function, we would expect that a polynomial would be able to model the function quite well in the given domain, due to its smoothness. By choosing a high polynomial degree, we can actually get the function to approximately interpolate all of the data points, so measuring the prediction error on a point from the training sample would get close to perfect fit. The training MSE for different complexity are shown in figure 4. For polynomial degree 10 we get an training MSE = 1.2e-3 on the terrain data without added noise. By increasing the degree to 20 the MSE can be further reduced to MSE = 1.74e-5. We also see that the R^2 is close to 1, so we are able to explain almost all of the variance in the data when predicting on the training data.

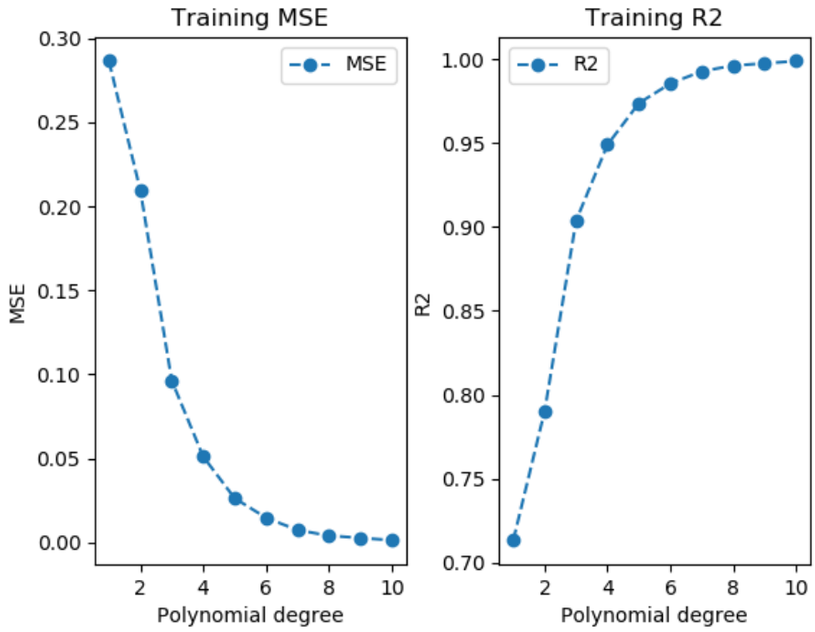

Figure 4: Training MSE and R2 with least squares regression for different polynomial degrees on the Franke function without noise.

It is interesting to see how the training error changes if we add some noise to the data. The noise is introduced by adding kN to each point where N is a normally distributed random variable and k is a constant. In figure 5 we use $k = 0.1$ and in figure 6 we use $k = 0.3$. The MSE increases with the increasing noise. In figure 5 the curve has shifted upwards in y-direction, and the lowest MSE is now MSE $= 0.11$. This is natural since the data no longer lies on a smooth curve and are a lot more noisy. The polynomial are unable to interpolate each point and the MSE increases. We can however decrease the MSE even further by using higher order polynomials. This can be seen from running the part_a python script for higher polynomial degrees. For degree equal to 20, we get an MSE $= 0.046$. After this I see no significant effect of increasing the complexity. We see that with the added noise, the MSE is higher.

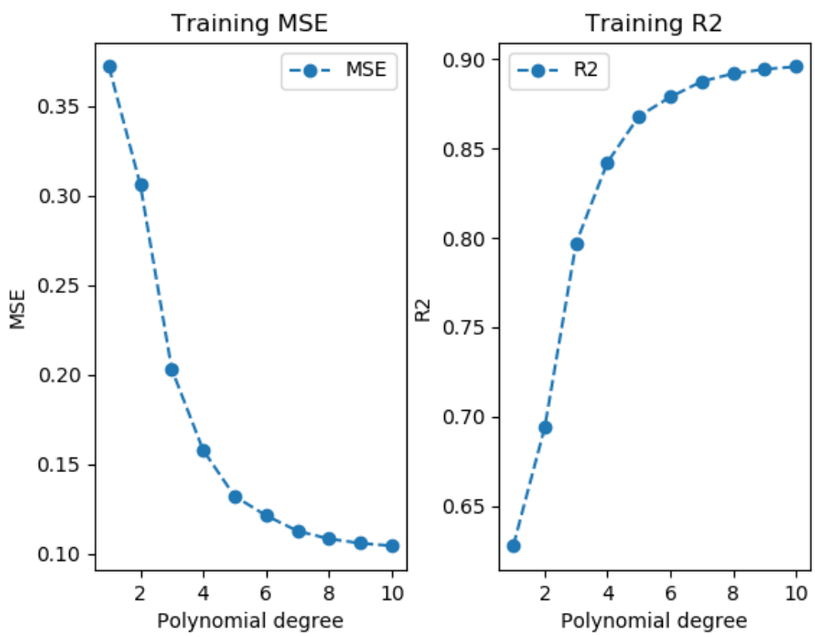

Figure 5: Training MSE and R2 with least squares regression for different polynomial degrees on the Franke function with added noise, k = 0.1.

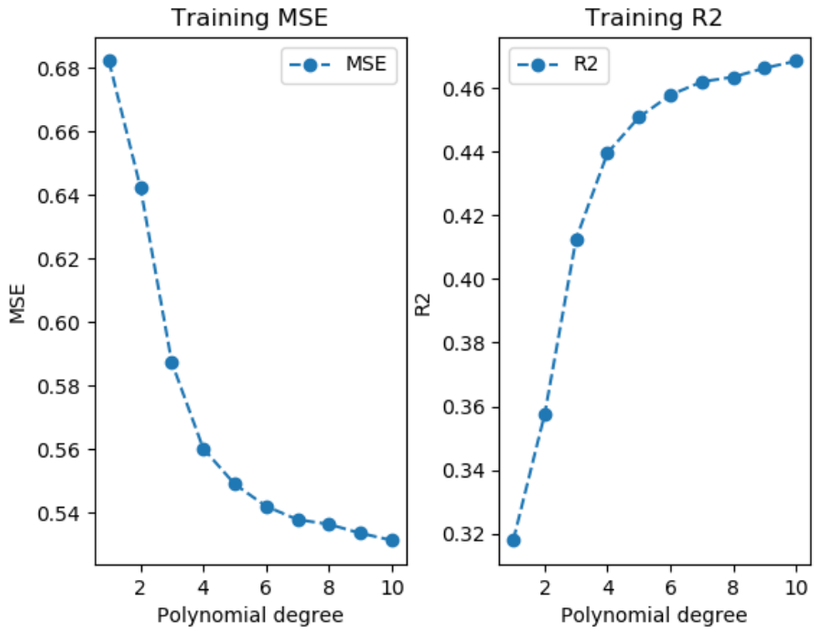

Figure 6: Training MSE and R2 with least squares regression for different polynomial degrees on the Franke function with added noise, k = 0.3.

Coefficient	Value	Confidence interval
β_0	-4.27435864e-15	[-0.02817734 0.02817734]
β_{x_1}	-4.27211836e-01	[-0.45538918 -0.3990345]
β_{x_2}	-5.49207937e-01	[-0.57738528 -0.5210306]

Figure 7: Regression coefficients up to degree 1 on Franke function with OLS.

Coefficient	Value	Confidence interval
β_0	0.08912259	[0.03897049, 0.1392747]
β_{x_1}	-0.42721184	[-0.45400776, -0.40041591]
β_{x_2}	-0.54920794	[-0.57600386, -0.52241201]
$\beta_{x_1^2}$	0.00968737	[-0.02028939 0.03966412]
$\beta_{x_1x_2}$	0.20500088	[0.17820495 0.2317968]
$\beta_{x_2^2}$	-0.09880996	[-0.12878672 -0.0688332]

Figure 8: Regression coefficients up to degree 2 on Franke function with OLS.

The confidence intervals for the regression coefficients for polynomial degree one and two are shown in table 7 and 8. We can look at the coefficients and their confidence interval for a polynomial of degree 5. In the printout bellow, we can see the coefficients and their confidence interval.The confidence interval are found on the Franke function without noise for a polynomial of degree five. To find more confidence intervals, you can run the code from part_a.py for a higher degree of polynomial.

```
beta_1: -0.129 CI: [-0.17607369 -0.08143425]
beta_x0: -0.340 CI: [-0.43253606 -0.24774255]
beta_x1: -1.735 CI: [-1.82751324 -1.64271973]
beta_x0^2: 0.661 CI: [0.58924312 0.73238401]
beta_x0 x1: 0.579 CI: [0.52101916 0.63778192]
beta_x1^2: 0.089 CI: [0.01781975 0.16096064]
beta_x0^3: -0.001 CI: [-0.10639527 0.10426782]
beta_x0^2 x1: -0.194 CI: [-0.27529502 -0.11251327]
beta_x0 x1^2: -0.565 CI: [-0.64683886 -0.48405711]
beta_x1^3: 0.932 CI: [0.8267449 1.037408 ]
beta_x0^4: -0.264 CI: [-0.28988206 -0.23847988]
beta_x0^3 x1: -0.127 CI: [-0.14880465 -0.1047886 ]
beta_x0^2 x1^2: 0.009 CI: [-0.01262737 0.03032178]
beta_x0 x1^3: -0.053 CI: [-0.07463827 -0.03062222]
beta_x1^4: -0.087 CI: [-0.11285919 -0.06145701]
beta_x0^5: 0.006 CI: [-0.02417588 0.03674123]
beta_x0^4 x1: 0.139 CI: [0.11368834 0.16509052]
beta_x0^3 x1^2: 0.076 CI: [0.05131705 0.10071454]
beta_x0^2 x1^3: -0.061 CI: [-0.0852939 -0.03589641]
beta_x0 x1^4: 0.125 CI: [0.09938503 0.15078721]
beta_x1^5: -0.142 CI: [-0.17227114 -0.11135403]
```

The generalization error is as mentioned often more interesting than the training error. When choosing what complexity to use it is a good idea to look at the test-error for different polynomial degrees. To study the test error, the data set can be split into a training part and a test part. We will use 20% of the original data set for testing. There are multiple ways to find an estimate for the test error. We can use the training set to fit the model, and then find the MSE on the test set. Another method is to use cross-validation. There are also measures like the BIC and AIC that can give an estimate of the test error, by adding a penalty term to the training error.
In figure 11, 12 and 13 the test are calculated by the test-train split method. The data was split into two parts. The training part was 80 percent of the original data, and the test part was 20 percent. The model was fitted on the training data. In the figures we see the training error and the test error. The training error was found by calculating the MSE on the test data, and the training error was found by calculating the MSE on the training data. For each plot, different amounts of noise are added. We see that when no noise are added, the test-error seems to stay low, even for polynomials of degree up to 15. It is only marginally higher for some values, but seems to stick to the training error. Since no noise are added, the test points will be so close to the training points, that the MSE will be very small. We see that both the training MSE and the test MSS converges to zero in figure 11.

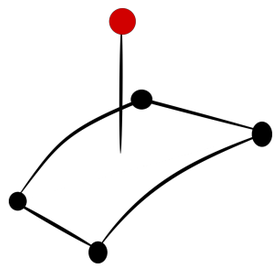

Figure 9: Illustration of test error. Red dot is a new test data. Black points are in the training data and a polynomial are fitted to those. The surface interpolates the training points.

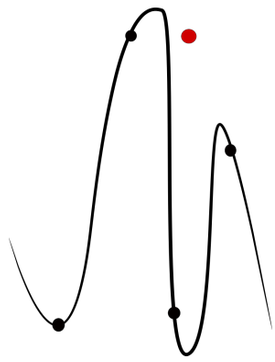

Figure 10: Illustration of overfitting. Black points are in the training data, and red points are is new test data.

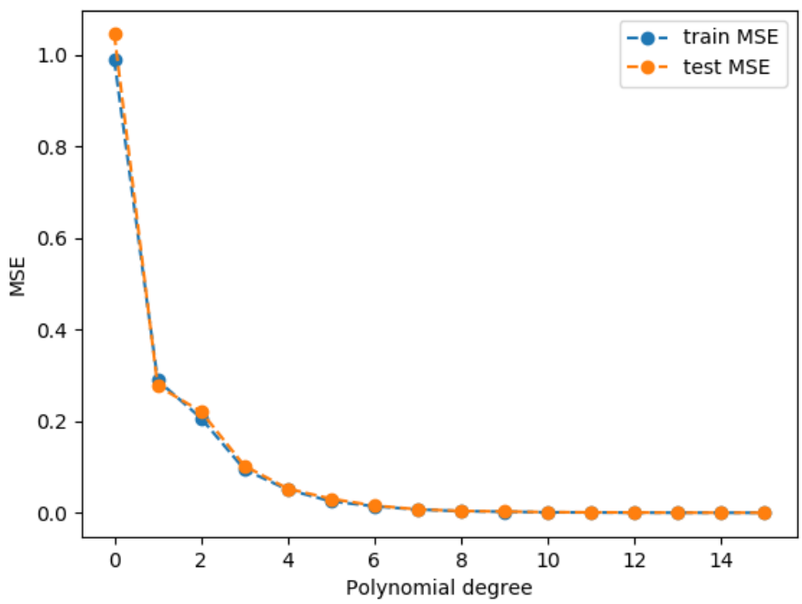

Figure 11: Test and training MSE with least squares regression for different polynomial degrees on the Franke function without added noise, k = 0

In figure 12 we see the training and test error with added noise. The noise added is small, with k = 0.1. The test error starts to deviate slightly from the training error for higher polynomial degrees, but the deviation is small. This is because the noise added are not large enough to make any significant outliers. This problem can be

illustrated in figure 9. If the distance from the test point to the training points are low, the MSE of the new test point will still be low, which we see in figure 9.

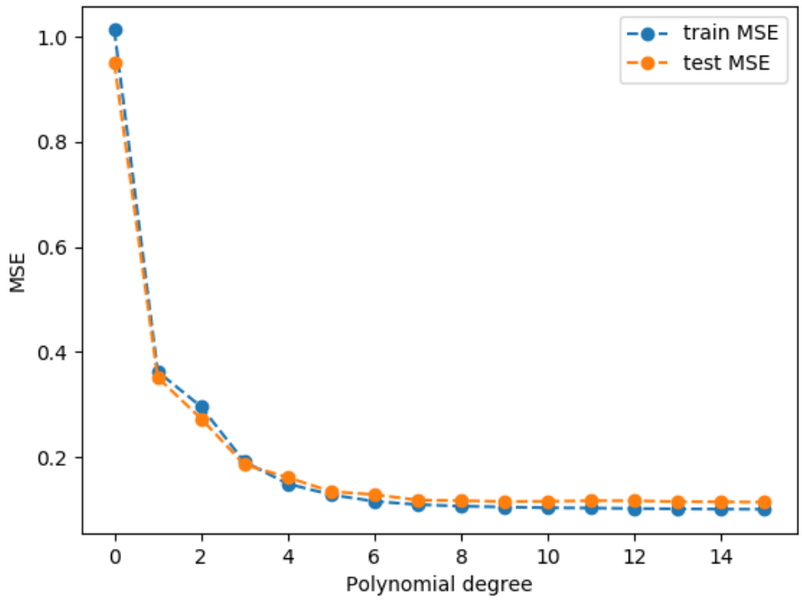

Figure 12: Test and training MSE with least squares regression for different polynomial degrees on the Franke function with added noise, k = 0.1.

We can increase the noise further, as seen in figure 13. Now k = 0.3 and we can start to see more evidence of overfitting. Now the test-error deviates significantly from the training error. Until polynomial degree 8 the test error is following the training error quite good, but after this, we see that we have an overfit. The problem now is that the points start to be further apart. By fitting a complex model to the training data, we might get fluctuations in the curve so that some of the test points is far away from the curve. This problem is illustrated in figure 10. By looking at the illustration, we see that the squared error the new red point are high.

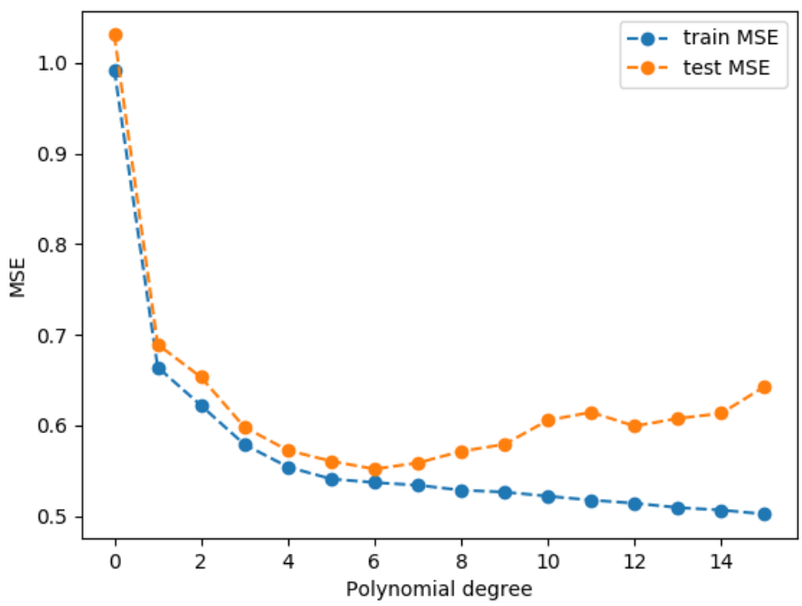

Figure 13: Test and training MSE with least squares regression for different polynomial degrees on the Franke function with added noise, k = 0.3.

The problem with overfitting can be summarized in the bias variance trade-off. We can see what part of the error in equation 29 contributes to the MSE the most for each degree of complexity. In the bias-variance trade-off we see how both bias, variance and MSE changes as a function of model complexity. In the implementation of bias-variance cross-validation is used to compute the different parts. By plotting the different parts of equation 29 we get the plot in figure 14 and 15.

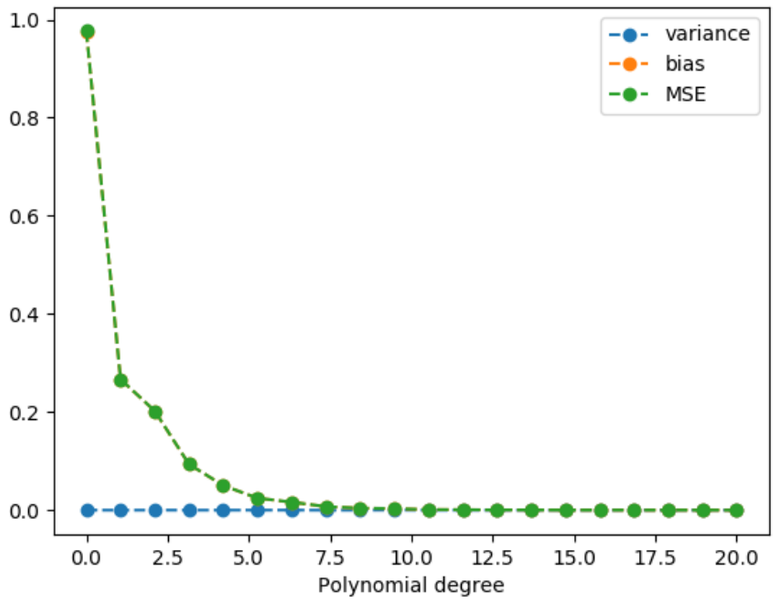

Figure 14: Bias-variance trade-off for Franke function without noise, k = 0

We see that when no noise is added to the model, the variance is very low. The variance is stable as a function of polynomial degree and does not change significantly in the range plotted. However, we see that the bias decreases when we increase the complexity. In the range of polynomial degree between 0 and 8 the bias decreases very fast and the bias makes up approximately all of the MSE.

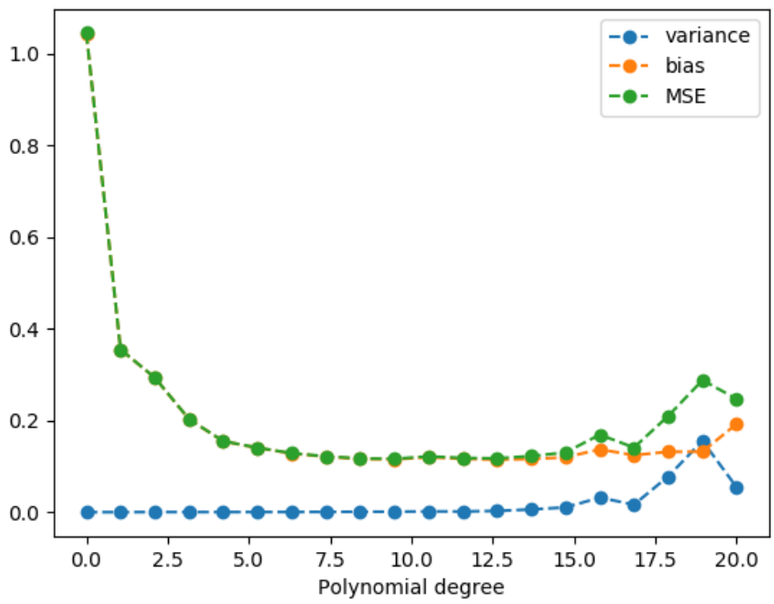

Figure 15: Bias-variance trade-off for OLS on Franke function with noise, k = 0.1

When we add noise to the data, we can again see the evidence of overfitting. Now the data have been fitted with a to complex model, and the variance starts to increase. As before, we see that for low polynomials, all of the MSE is given by the bias. When we increase complexity, variance starts to contribute more and more.

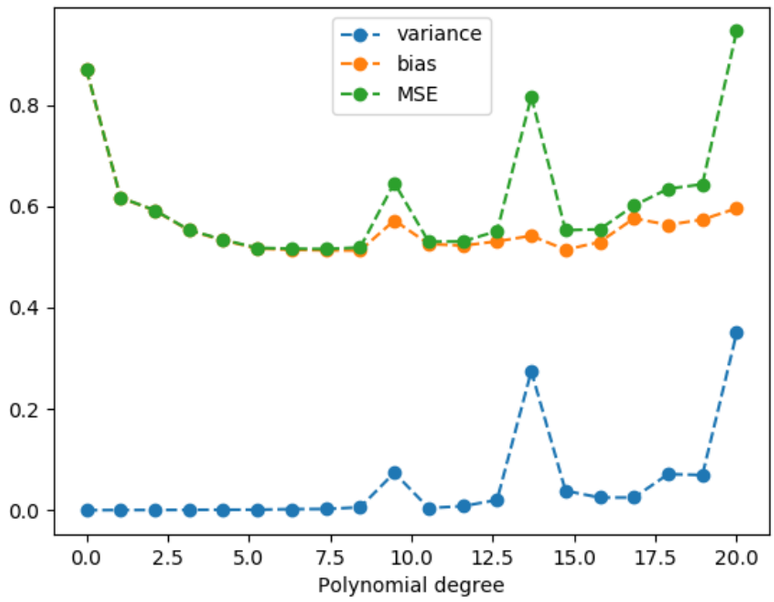

Figure 16: Bias-variance trade-off for OLS on Franke function with noise, k = 0.3

With even more added noise, the models are not able to fit the data very well. The bias and MSE is high even for high degrees of polynomials, so it seems like a polynomial of any degree is a good fit to the data. However, we see evidence of the bias variance trade-off. For low complexity we have high bias and low variance. When we increase the complexity, the bias reduces, and the variance increase. This is what would be expected expected.

Ridge regression on the Franke function

We can now look at ridge regression and see how it performs on the same cases as we have studied for polynomial regression. By looking at the training MSE and R^2, we see in figure 17 and 18 that we have the same trend as before. The training error decreases with increasing complexity, and the R^2-values increases. In the figures the training error are plotted for different values of lambda. Again, we see that adding noise to the data results in higher MSE and lower R^2, by comparing the two plots. In figure 19 the same plots are shown, only zoomed in. Here we can clearly see that the MSE decreases as lambda decreases. From the training error it seems like lambda equal to zero is the optimal solution.

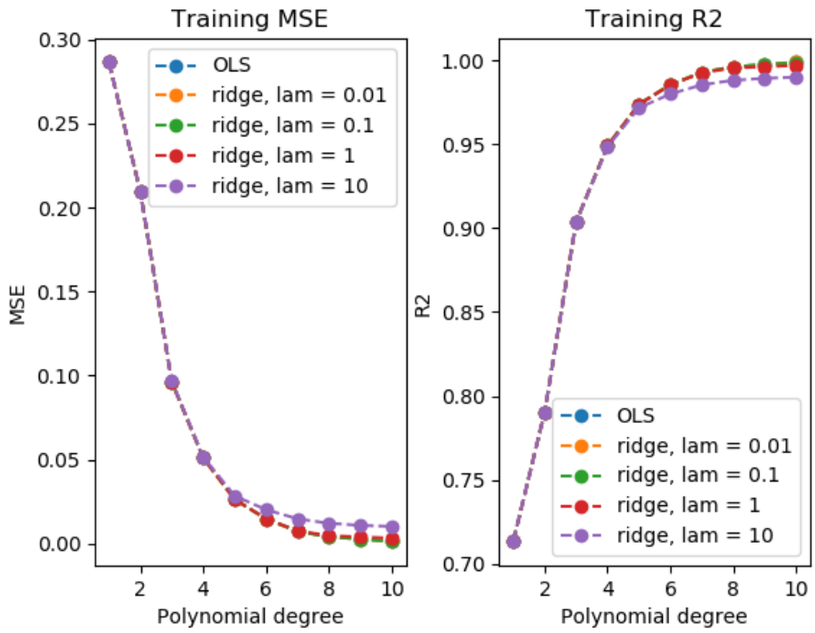

Figure 17: Training MSE and R2 with ridge regression for different polynomial degrees on the Franke function without noise

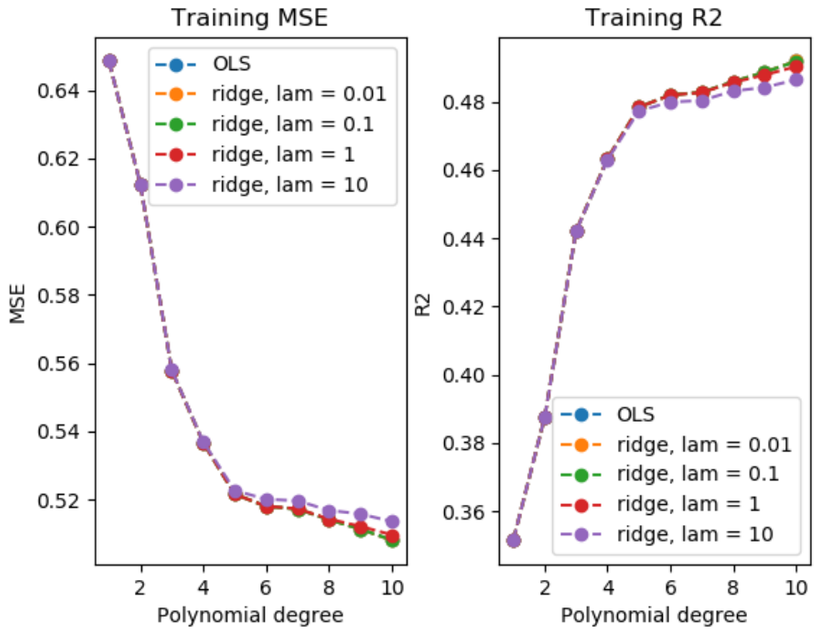

Figure 18: Training MSE and R2 with ridge regression for different polynomial degrees on the Franke function with added noise, k = 0.3

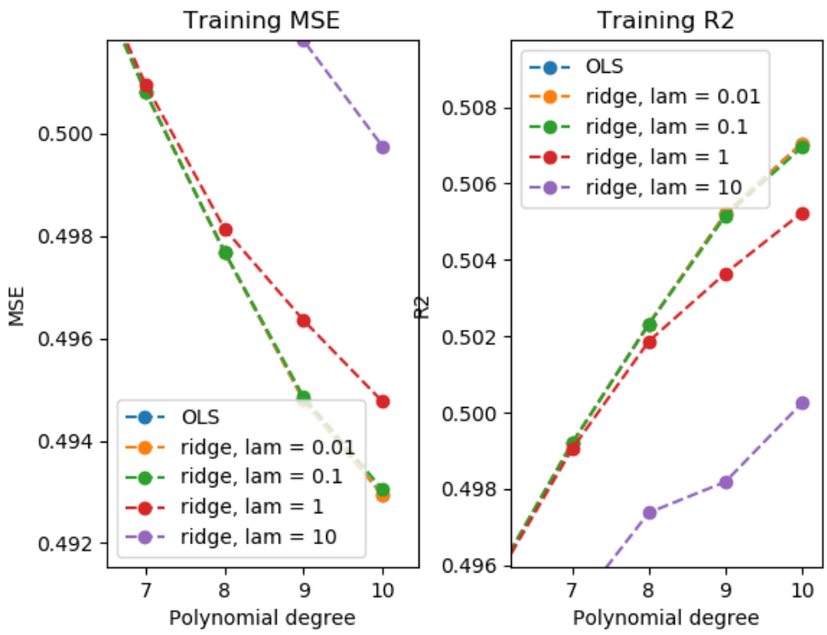

Figure 19: Training MSE and R2 with ridge regression for different polynomial degrees on the Franke function with added noise, k = 0.3. Zoomed in.

However, we have to look at the test error to make a conclusion. We can plot the different parts of equation 29 as a function of lambda for different polynomial degrees. Each part is found using 5-fold cross-validation. We start by looking at figure 20 where no noise is added. The first thing one sees is that the test bias and test MSE increases as lambda increases. In figure 21 the variance zoomed in on. We see that for degree 1 and 4, the variance decreases when lambda increases. For degree 5 it decreases at first and then flattens. For degree 10, the variance seems to fluctuate a lot more. For degree 1, 4 and 5, the variance behaves as one would expect. With increasing lambda, we want to decrease the variance. For degree 10, it seems to get some problem with constraining the coefficients. Maybe due to overfitting.

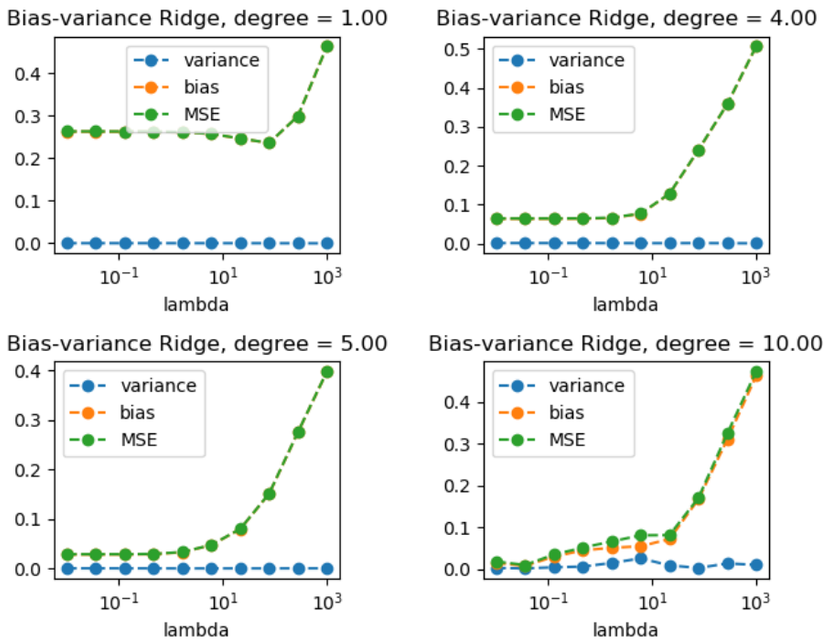

Figure 20: Test bias-variance trade-off with ridge regression on the Franke function without noise.

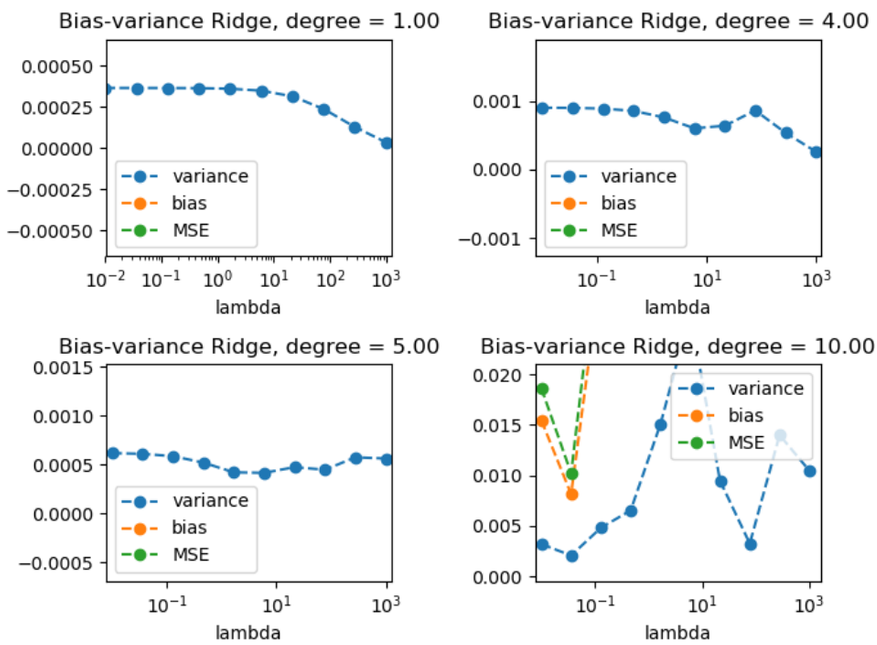

Figure 21: Closer look at the variance for the test bias-variance trade-off with ridge regression on the Franke function without noise.

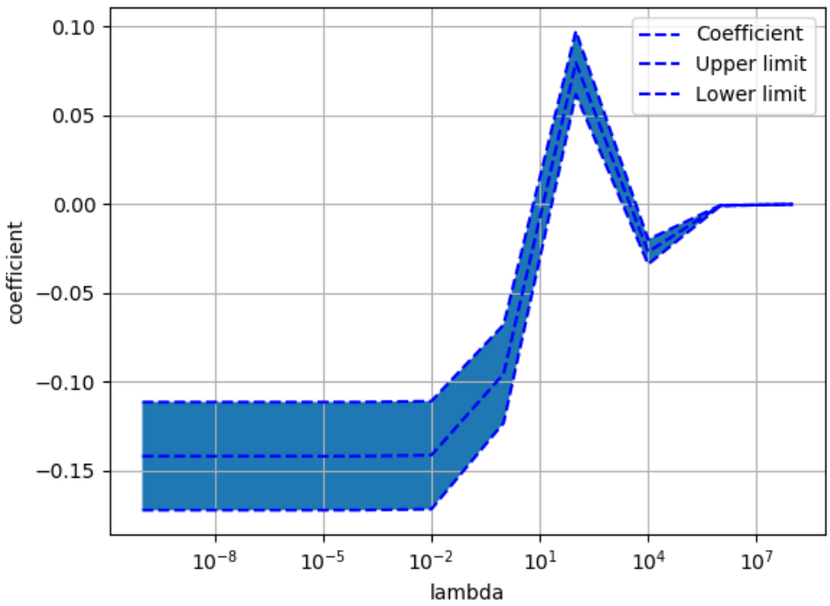

Figure 22: The confidence interval of β_{x^5} as a function of lambda on the Franke function without noise.

With noise added to the model in figure 23, it is easier to see the expected effects of ridge regression. Again, we see that the bias increases as lambda increases. When zooming in and looking at how the variance behaves as a function of lambda in figure 24 we see that increasing lambda decreases the variance. From looking at the problem both with added noise and without added noise, we see that ridge regression does not obtain what we would have hoped for. As we would expect, bias increased and variance decreased. However, the bias increased a lot more than the variance decreased. This led to the MSE in total being larger when we added lambda. This means that OLS seems to be the best model also for the test MSE. One last check is to look at the heatmaps for the cross-validation error in figure 26 and 27. Here we see that the MSE increases as we move closer to the lower left corner. This is where lambda is low and the polynomial degree is high. We can therefore conclude that the OLS is better than Ridge on the Franke function.

Looking at the confidence intervals for the coefficients, it is interesting to see how they change when we increase lambda. In figure 22 we see how to coefficient of x^5 varies. The variance decrease when lambda is increased. In addition, we can see that for high values of lambda, the coefficient goes towards zero. This is what was expected from the formula for the variance of the betas for Ridge regression and the goal of ridge regression. When adding noise to the data, the confidence interval gets bigger. This is plotted in figure 25. This is because of the noise the uncertainty of the betas will vary more.

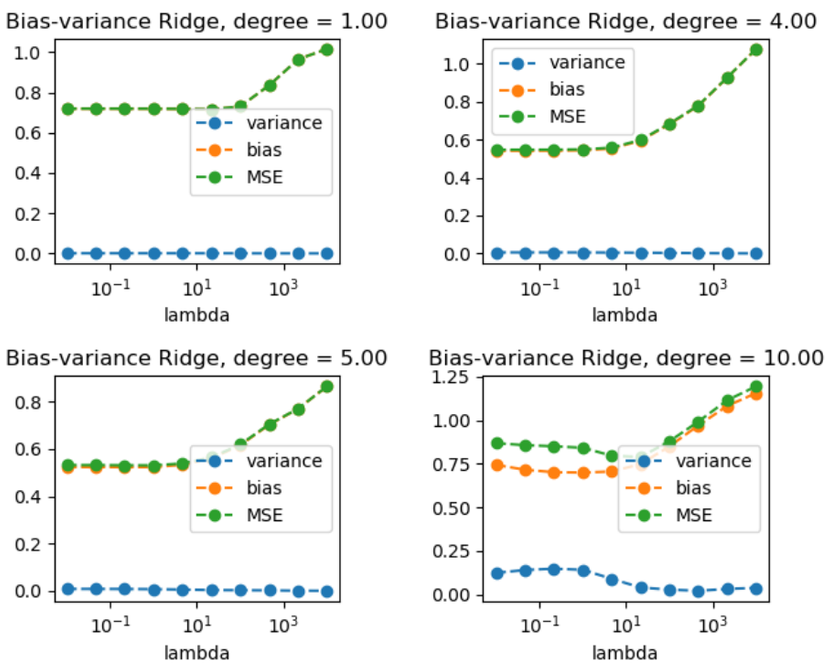

Figure 23: Test bias-variance trade-off with ridge regression on the Franke function with noise, k = 0.3.

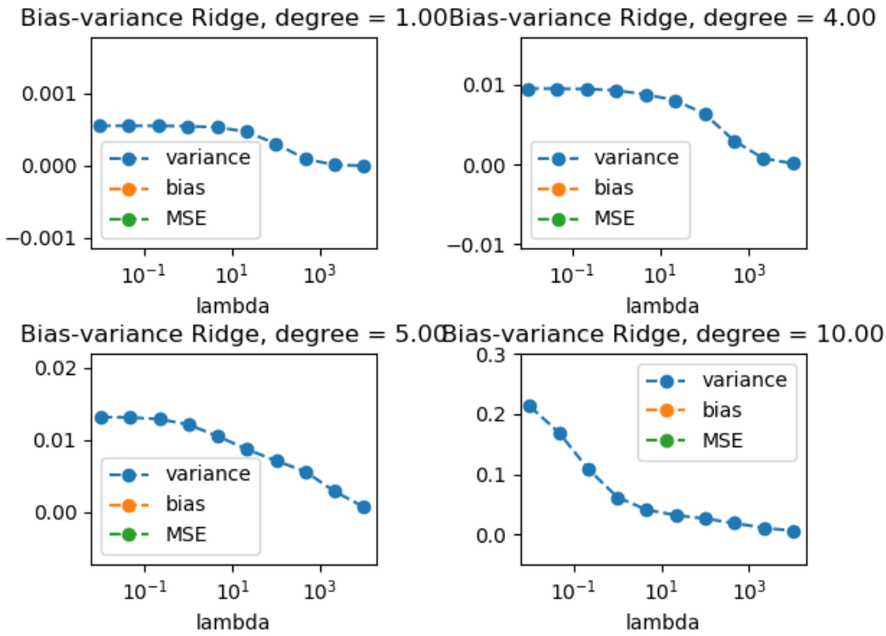

Figure 24: Closer look at variance for test bias-variance trade-off with ridge regression on the Franke function with noise, k = 0.3.

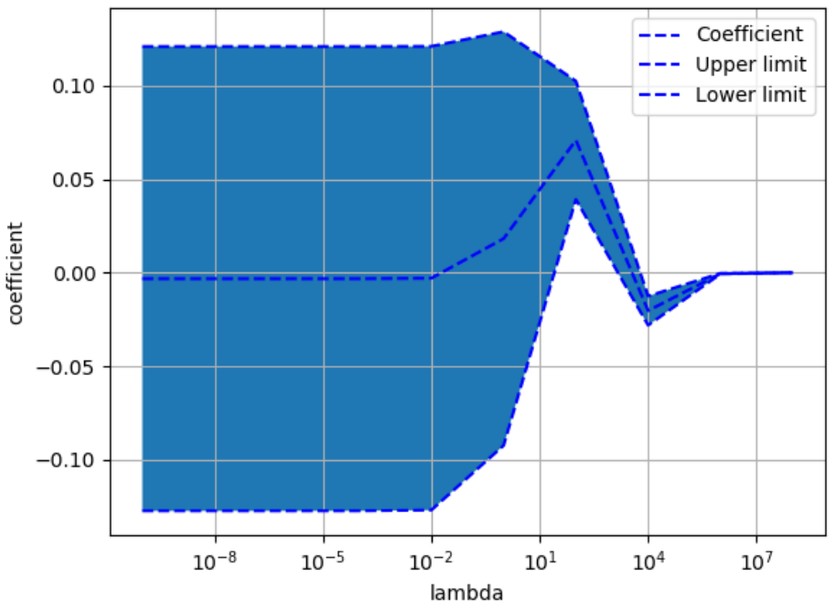

Figure 25: The confidence interval of β_{x^5} as a function of lambda on the Franke function with noise, k = 0.3.

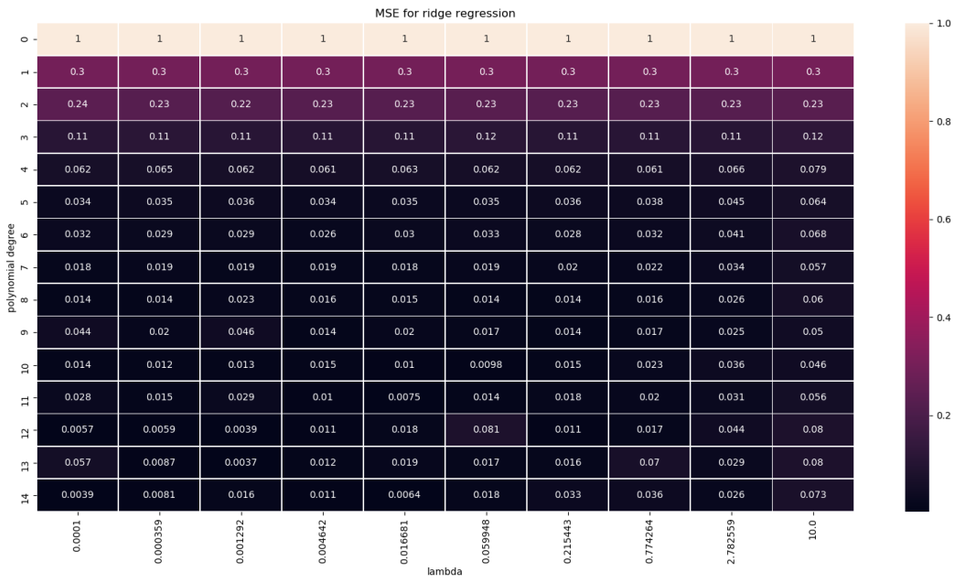

Figure 26: Heatmap for MSE for ridge regression. The values are calculated using ridge regression and 5-fold cross-validation on the Franke function without noise.

and

Figure 27: Heatmap for MSE for ridge regression. The values are calculated using ridge regression and 5-fold cross-validation without noise.

Lasso regression on the Franke function

We can do the same analysis for Lasso regression. The training error is plotted in figure 28. The results are similar to those we obtained for Ridge regression. When decreasing lambda, the MSE curve for the Lasso converges to the OLS curve, and again OLS beats the Lasso on the training error. The added noise in figure we can study how the test-error calculated using 5-fold cross-validation varies as a function of both lambda and the polynomial degree. We see that the optimal test MSE are obtained for lambda = 0.001 and polynomial degree 10. It can be seen by running the script for higher polynomial degress and lower lambda that we can obtain even smaller MSE-values. By using a polynomial of degree 20 and a lambda value of 1e-7, we can get test MSE = 0.01. However, the most important feature of the heat-maps are that the MSE increases towards the left lower corner of the heat-map. This is, as we have also seen before, where we have a high polynomial degree and a low lambda value. 28 results in higher training MSE as we have seen before. Since both Ridge and Lasso are limited below by OLS, we would not expect any different behaviour. By looking at the heatmaps in figure 30 we can again see that the lower left corner is where the lowest MSE values are found. This again shows that OLS is the best method.

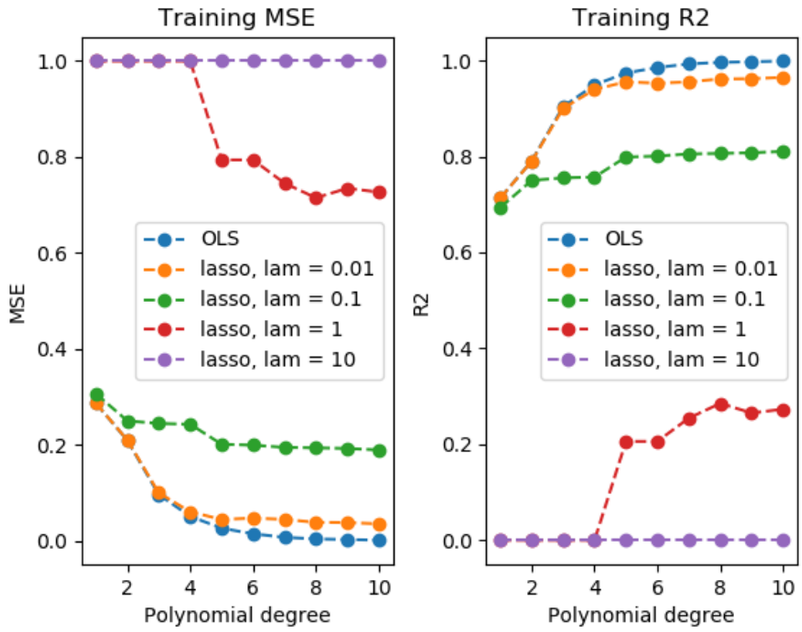

Figure 28: Test and train MSE for lasso regression on the Franke function without noise.

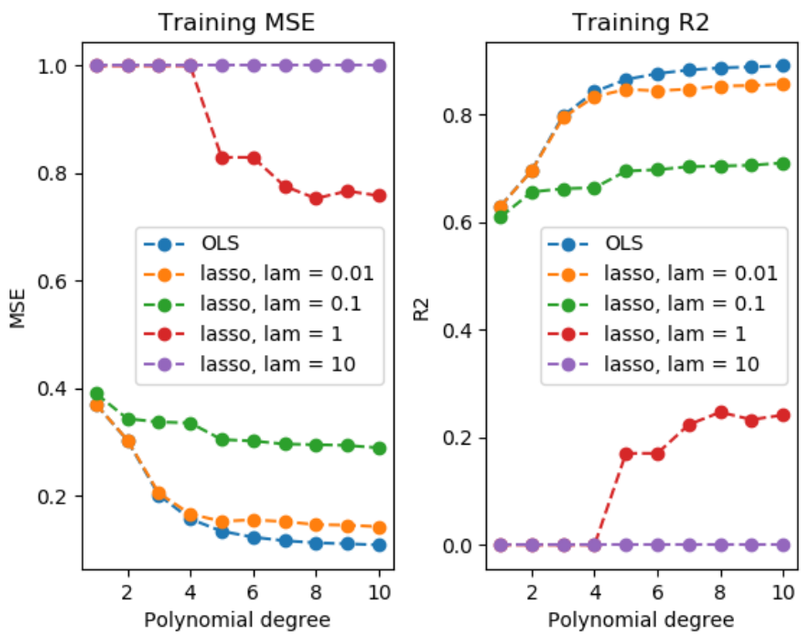

Figure 29: Test and train MSE for lasso regression on the Franke function with added noise, k = 0.1

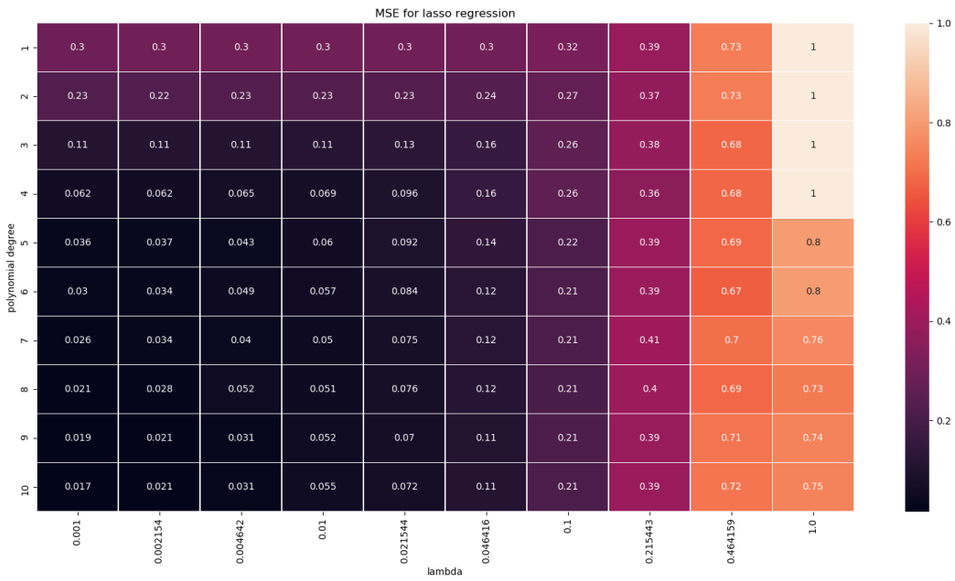

Figure 30: Heatmap for MSE for Lasso regression. The values are calculated using ridge regression and 5-fold cross-validation on the Franke function without noise.

Figure 31: Heatmap for R2 for Lasso regression. The values are calculated using ridge regression and 5-fold cross-validation on the Franke function without noise

Looking at the bias-variance trade-off for lasso regression, we get similar results as we did for ridge regression. In figure 32 and 34 we see that by increasing lambda, the MSE and bias increases. The values are found using cross-validation. The increase in bias is expected. However, we have the same problem as for ridge. The bias increases much more than the decrease in variance, so in total, we get a worse MSE than with OLS. The decrease in variance is seen in figure 33 and 35.

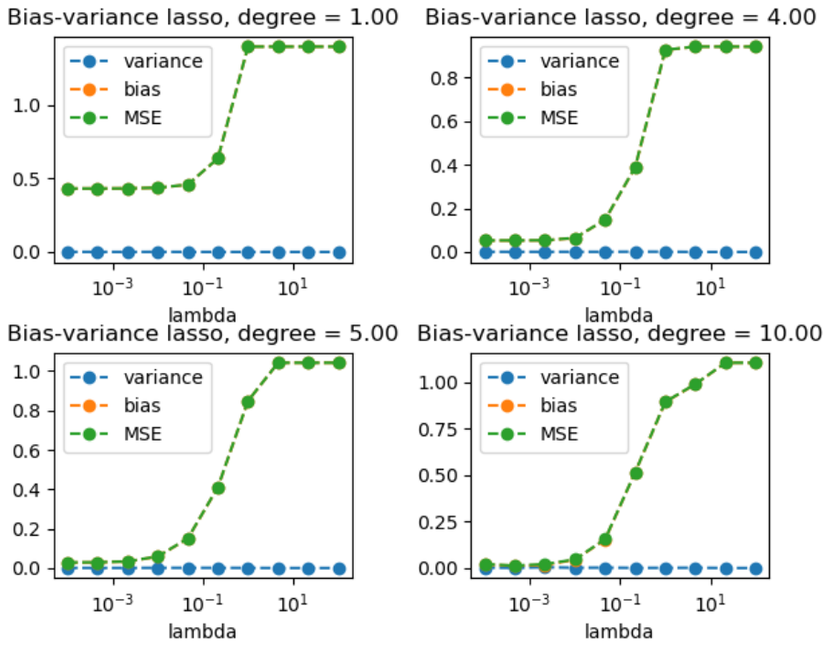

Figure 32: Bias-variance trade-off for lasso regression on the Franke function without noise

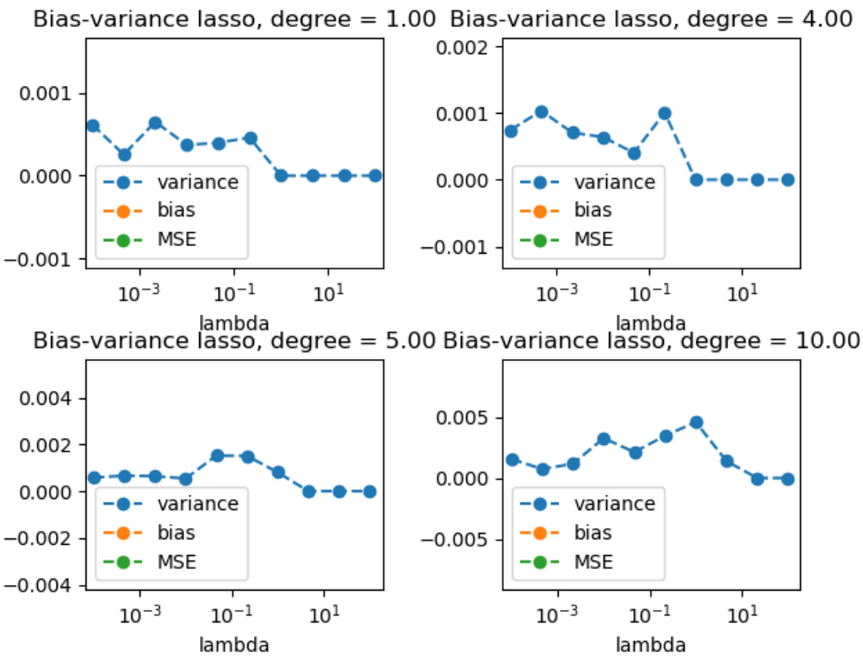

Figure 33: Bias-variance trade-off for lasso regression on the Franke function without noise. Zoomed in to get a better look at the variance.

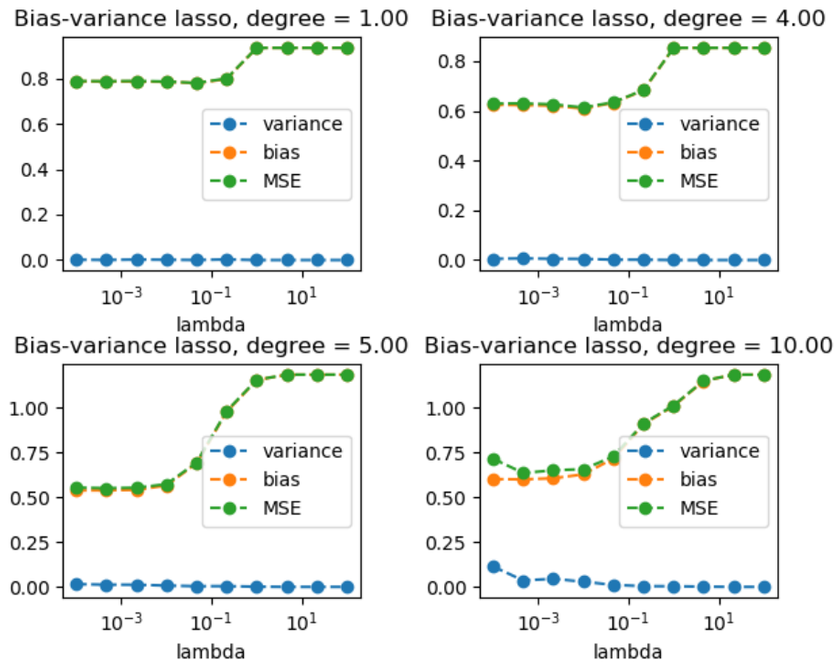

Figure 34: Bias-variance trade-off for lasso regression on the Franke function with added noise, k = 0.3

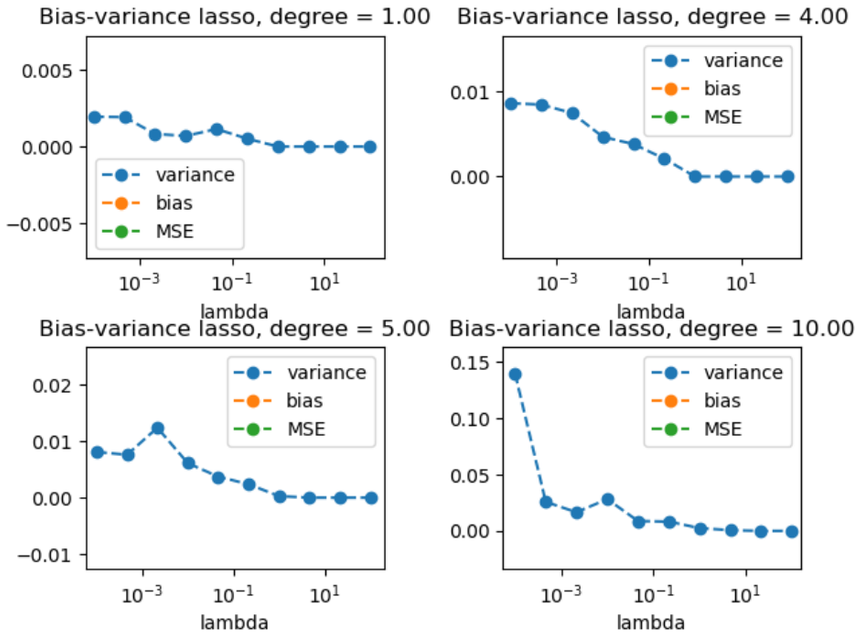

Figure 35: Bias-variance trade-off for lasso regression on the Franke function with added noise, k = 0.3. Zoomed in to get a better look at the variance

From the previous results, we can conclude that OLS is the best method for fitting the Franke function. Without noise, we are able to get an test-MSE as low as 0.0039 for a polynomial regression of degree 14. By letting lambda go to zero for ridge and lasso gets a better and better MSE until it reaches zero, and are equal to the MSE of OLS.

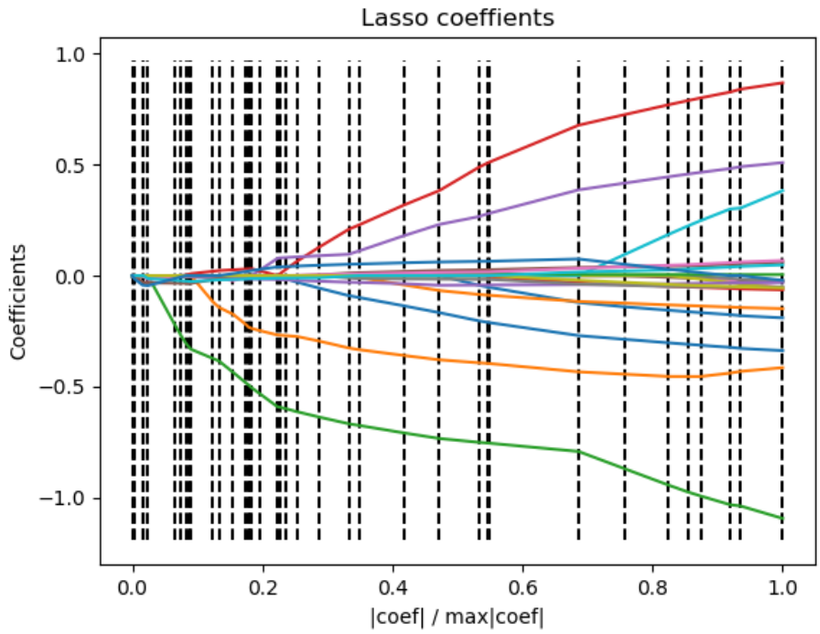

Figure 36: Lasso coefficients as a function of lambda for polynomial degree of 5.

Summary of methods for the Franke function

A table of the best values obtained for MSE with the three methods are presented in 37.

	MSE
OLS	7.3e-3
Ridge	3.2e-2
Lasso	0.21

Figure 37: Best MSE for each method. In Ridge and Lasso, $\lambda = 0.1$. All values calculated using 5-fold cross validation MSEs with no added noise to the data.

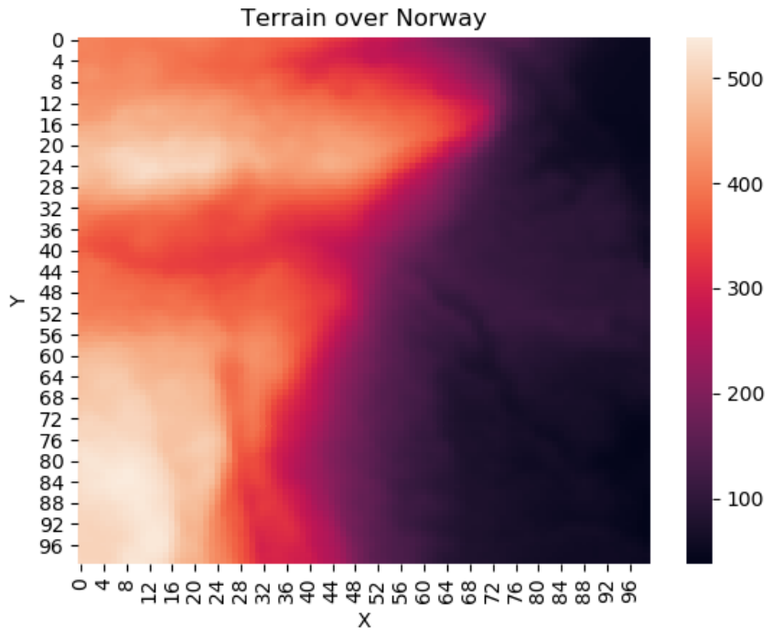

Figure 38: 100x100 of the terrain data

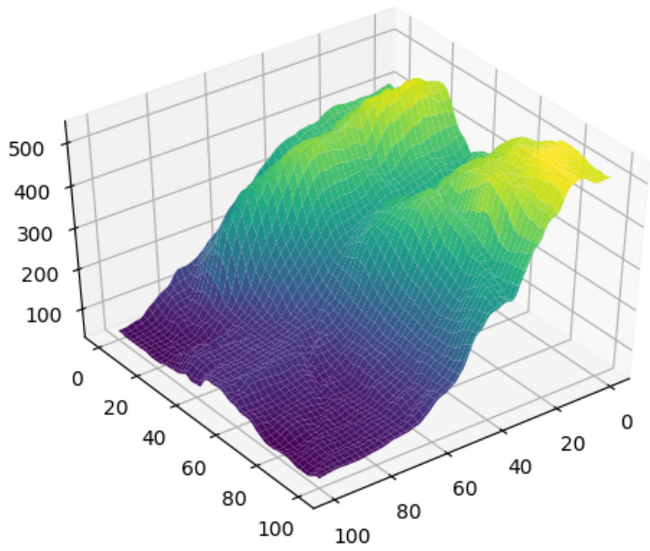

Figure 39: Shape of the terrain

OLS on the terrain data

We can now look at the terrain data and see how the models perform on a more practical case. I will use only a 100x100 grid of the data. This is pictured in figure 38 and 39. By choosing a smaller number of points it will be easier to see the trends in the errors that we are trying to study. We will start by looking at the OLS. In figure 38. we see that as before, the training error decreases when we increase the complexity. When looking at the training and test error in figure 41 we see that the test error and training error are the same for polynomials up to degree 20. This seems to be similar results to what we got for the Franke function without noise. This is due to the amount of noise in the terrain data. Since we typically look at terrain values of magnitude of hundreds, and the error in each data point probably are in centimeters, the noise is extremely small. We have to remember that the data is scaled to a standard deviation of 1, so the MSEs are according to the scaled values and not the actual values. We would however expect the test error and the training error to be very similar, as was discussed for OLS on the Franke function without noise.

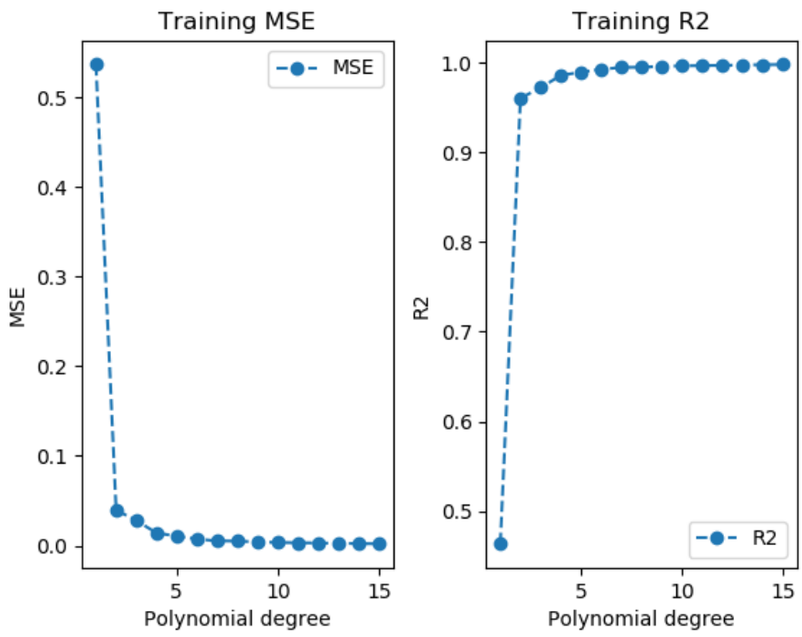

Figure 40: Trianing MSE and R2 on the terrain data using OLS

We can look at the proportion of the MSE that comes from variance and bias. In figure 42 we see that most of the MSE comes from the bias term. As we can see from figure 39 the terrain can not be approximated by a straight line or a second order polynomial. This is why we see an significant amount of bias for polynomials of degree smaller than 3.

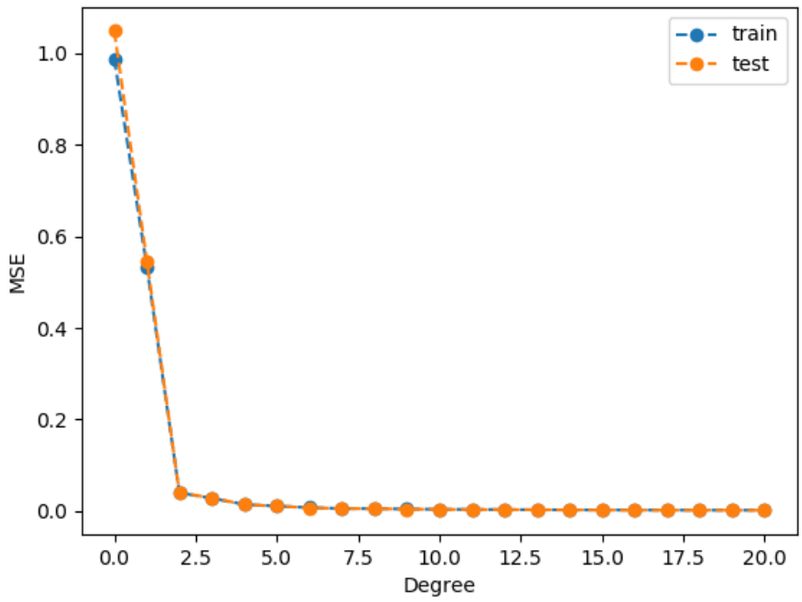

Figure 41: Train and test MSE on the terrain data using OLS.

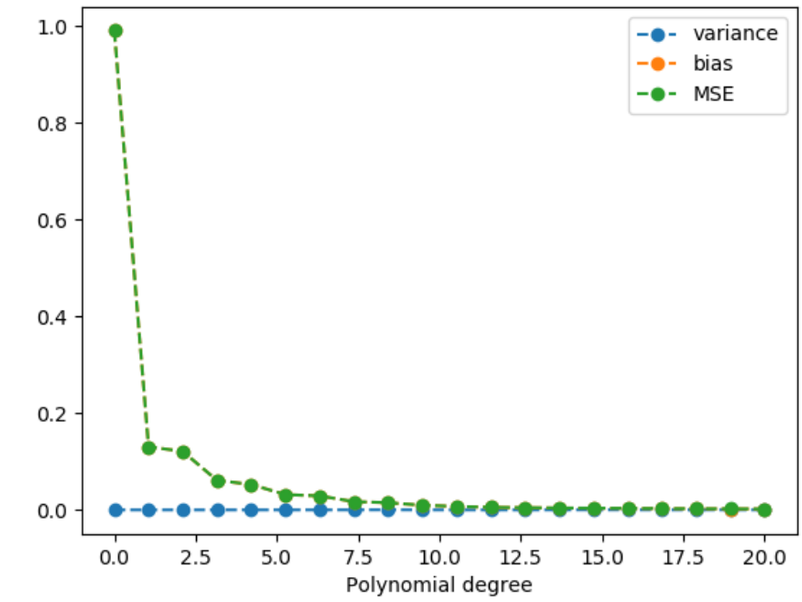

Figure 42: Bias-variance trade-off for the terrain data using OLS.

Ridge regression on the terrain data

Ridge regression on the terrain data gives similar results as before, but the effect of an increase in lambda is not as evident as for the terrain data. This can be seen both in figure 43 for the training error and in figure 47 for the test error. In the heatmap, the MSE does not seem to vary as much for different values of lambda. This is probably because there is so little noise in the data. Overfitting does not seem to be a problem for polynomials up degree of 20. The MSE increases up to this. However, in the bias variance trade off in figure 45 we see that by choosing lambda to high, the bias start to increase again. It is interesting to see that the increase in bias start for lower lambda values as we increase the polynomial degree. Again, we see that the MSE-value are lowest for the highest polynomial degree and the lowest lamda. A closer look at the variance in figure 46 shows that the variance is extremely small. For polynomial degree 1 and 4, it seems to decrease with lambda, but for degree 5 and over, it seems to increase again for higher lambda. This is insignificant on the MSE, due to the small values compared to the bias. All of the MSE is therefore given by the bias. This means that similarly to for the Franke function, OLS is the superior method. Figure 48 shows that we are able to explain all of the variance in the data, by using OLS with a polynomial degree of 13. As for the Franke function, the confidence interval of the coefficients narrows as we increase lambda. This can for example be seen in figure 49. Again, the coefficient shrinks towars zero as lambda is increased.

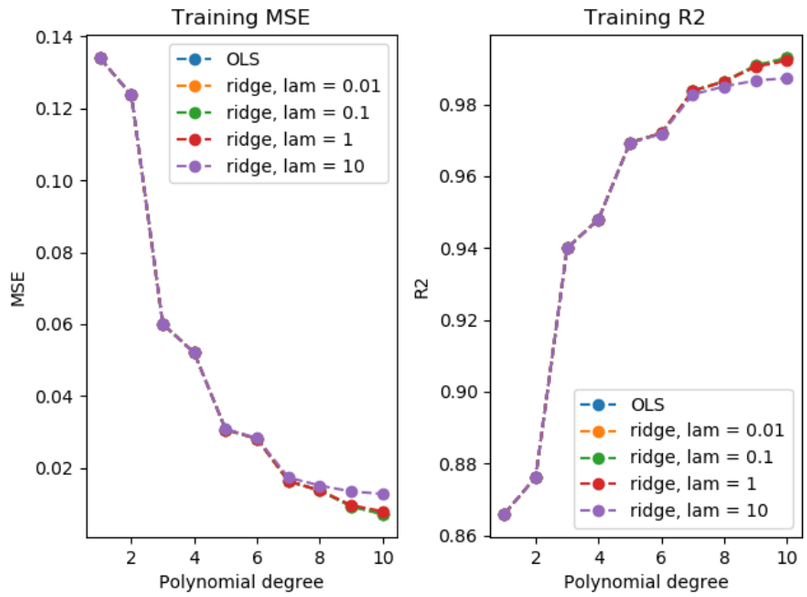

Figure 43: Training MSE and R2 for the terrain data using ridge

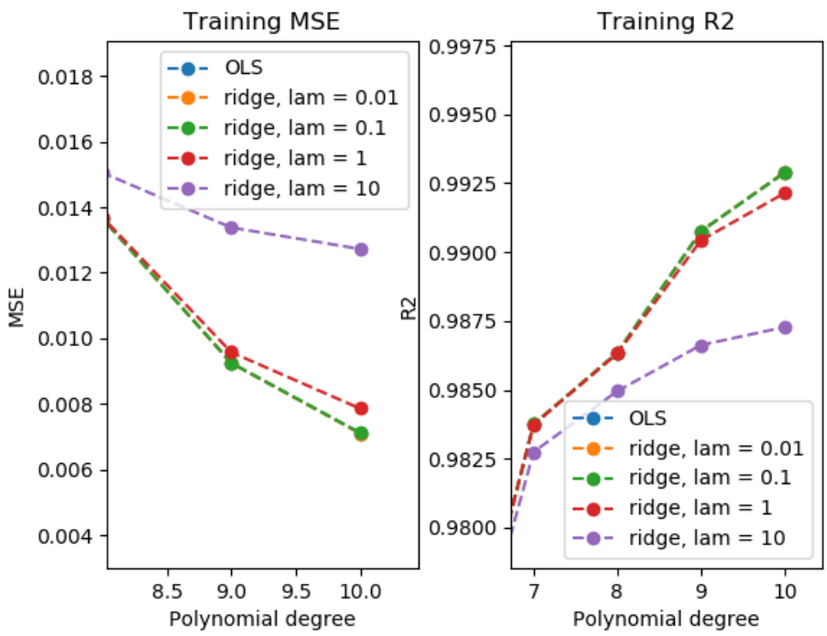

Figure 44: Training MSE and R2 for the terrain data using ridge

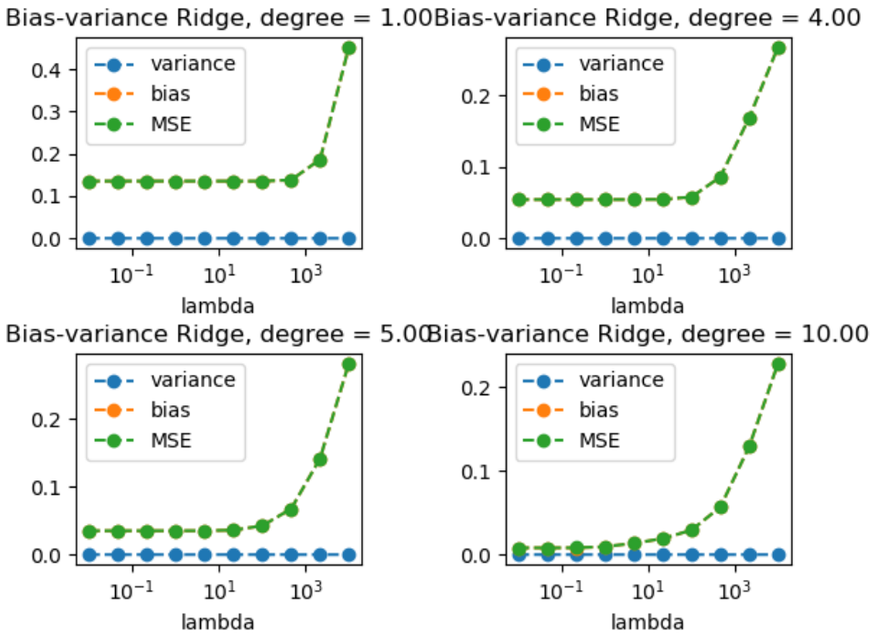

Figure 45: Bias-variance trade-off calculated using 5-fold cross-validation for Ridge regression on the terrain data..

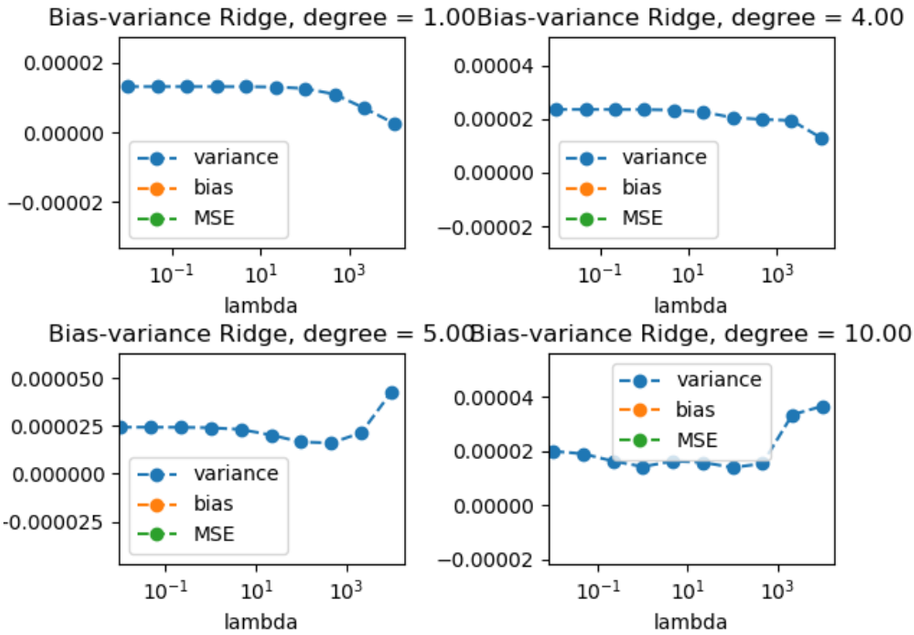

Figure 46: Bias-variance trade-off calculated using 5-fold cross validation for Ridge regression on the terrain data. Zoomed in to study the variance.

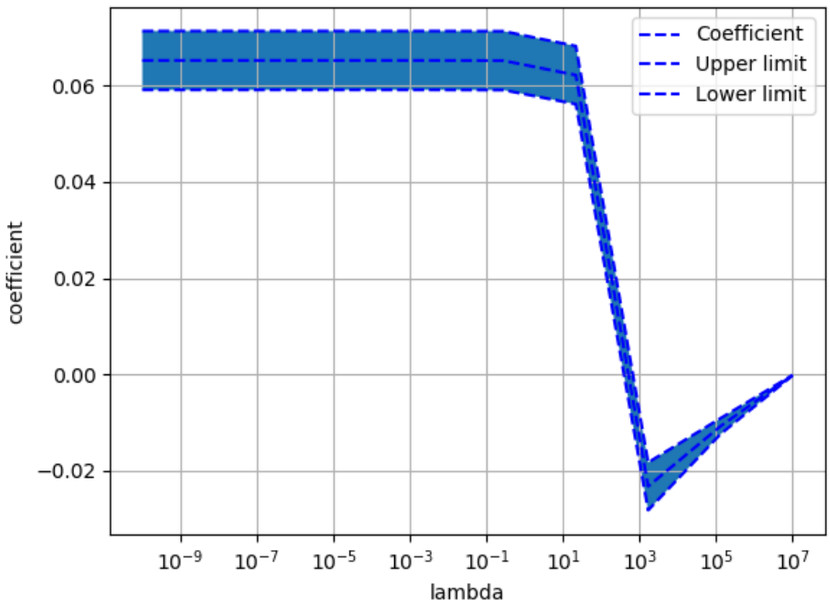

Figure 49: Confidence interval for β_{x^3} in ridge regression with lambda

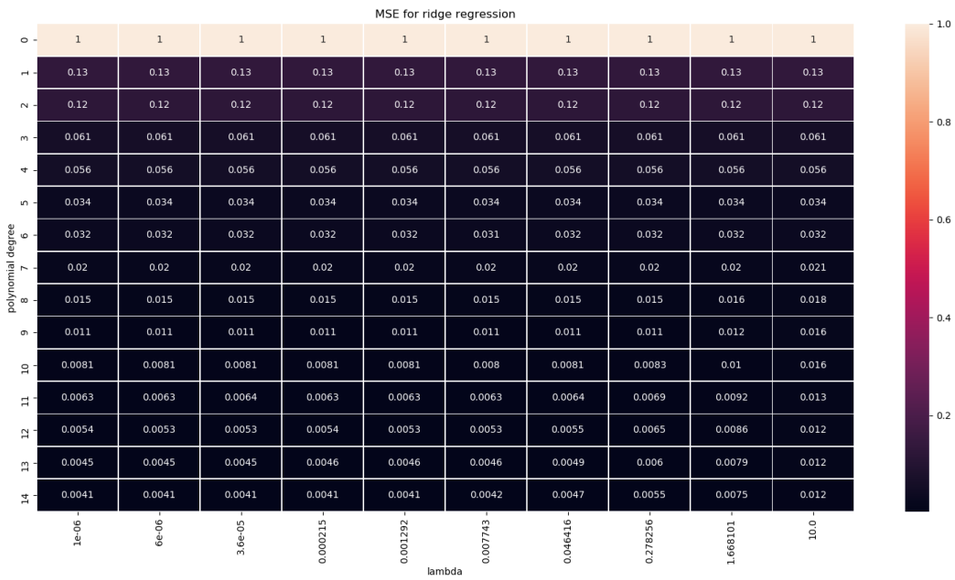

Figure 47: Cross-validation test MSE for Ridge regression on the terrain data

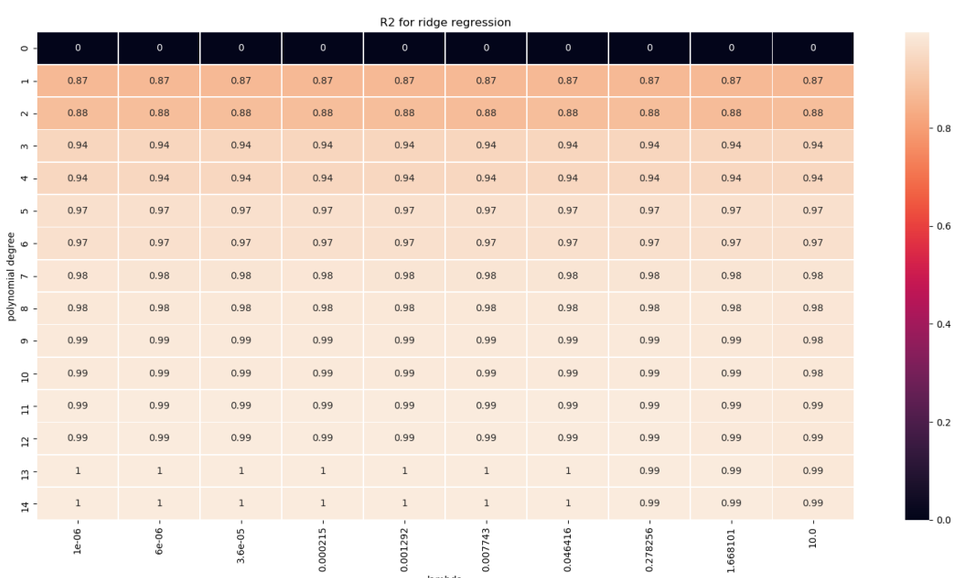

Figure 48: Cross-validation test R2 for Ridge regression with polynomial degree 3 on the terrain data

Lasso regression on the terrain data

Again, the Lasso behaves similarly on the terrain data as it did on the Franke function. There is however quite an significant difference between the lasso and ridge when looking at the dependence on lambda. We see in figure 50 that the difference in training MSE and R2 are larger for different values of lambda, compared to what we saw for ridge regression. The reason for this might be that Lasso actually cancels out some of the coefficients when lambda is high, so that the predictions gets really bad. For smaller values of lambda, we do see that the Lasso MSE converges towards the OLS MSE.

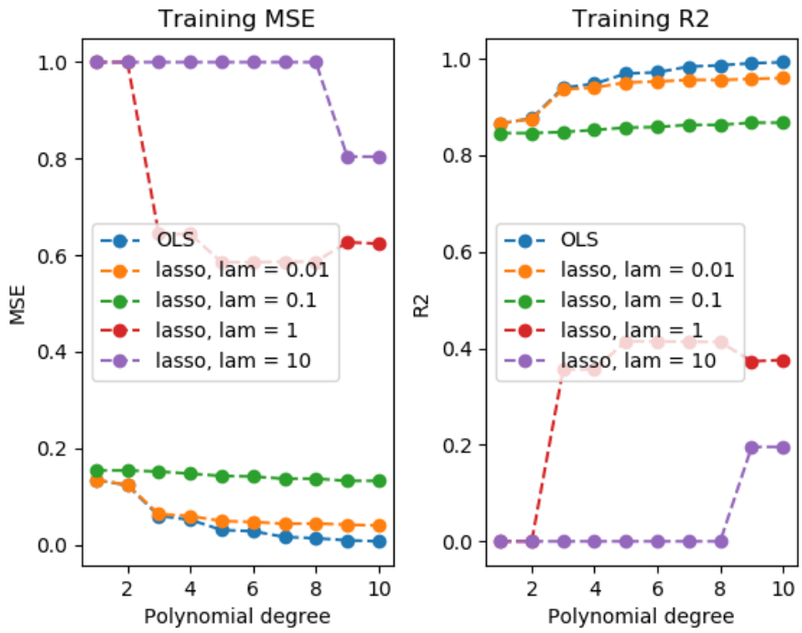

Figure 50: Training MSE and R2 for different polynomial degrees and lambda on the terrain data using lasso.

The bias-variance trade-off for Lasso on the terrain data behaves similar to what it did for the Franke function. We see that by increasing lambda, we get an increase in bias and MSE. In addition, like for Ridge, the variance is very small as seen in figure 52. No significant decrease in variance is obtained by increasing lambda. In figure 55 we see how the coefficients shrinks to zero as a function of lambda.

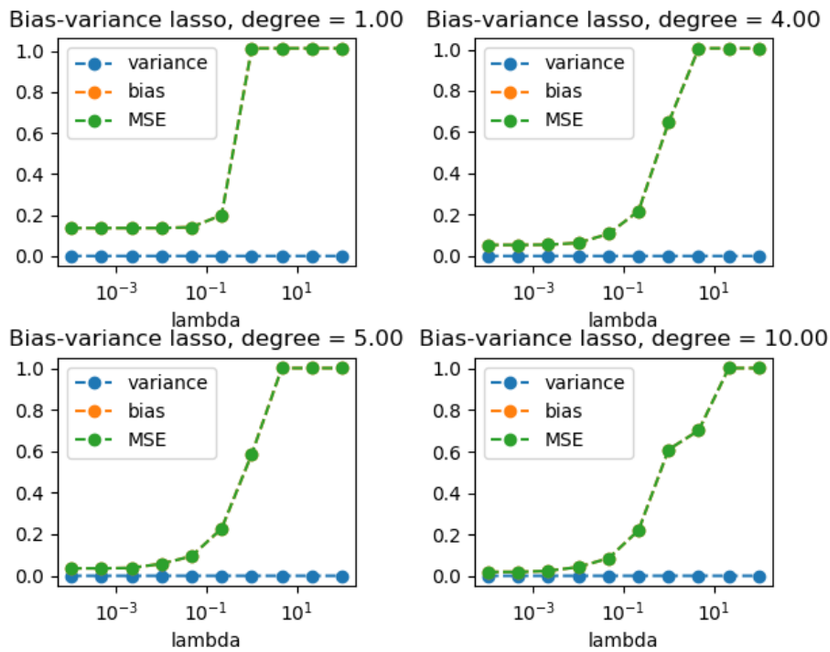

Figure 51: Bias-variance trade-off for Lasso on the terrain data. Calculated using 5-fold cross validation.

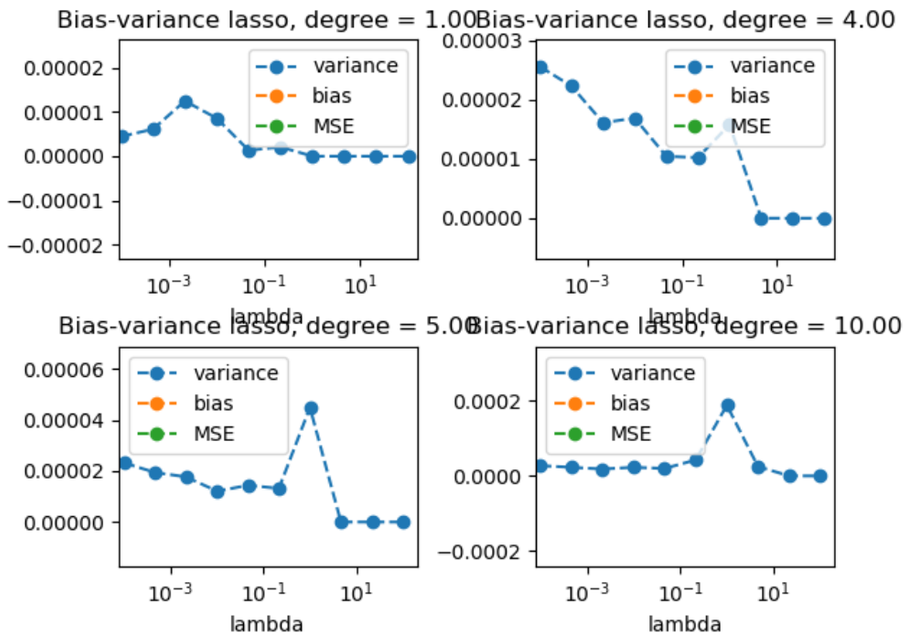

Figure 52: Bias-variance trade-off for Lasso on the terrain data. Zoomed in to study the variance. Calculated using 5-fold cross validation.

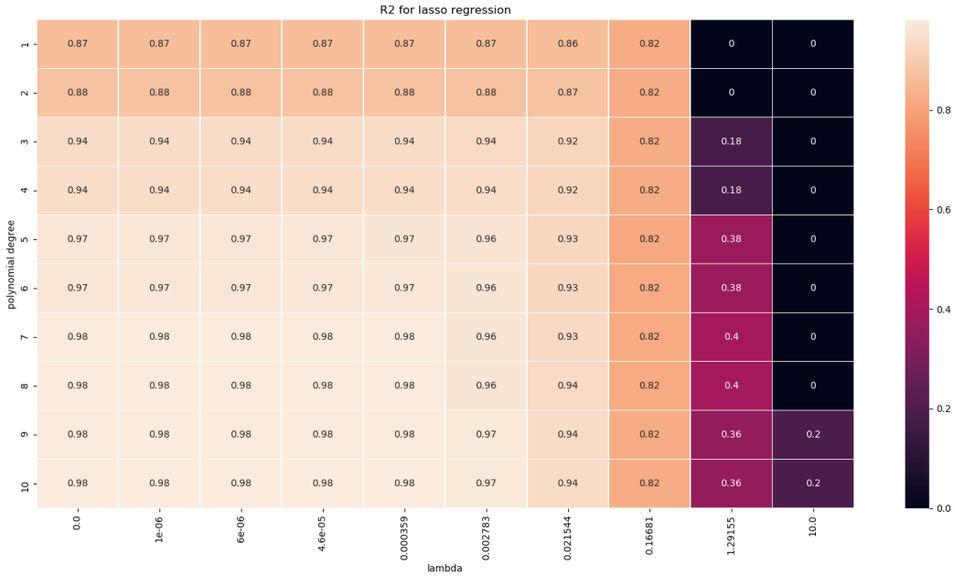

Figure 53: Lasso CV R2 for the terrain data

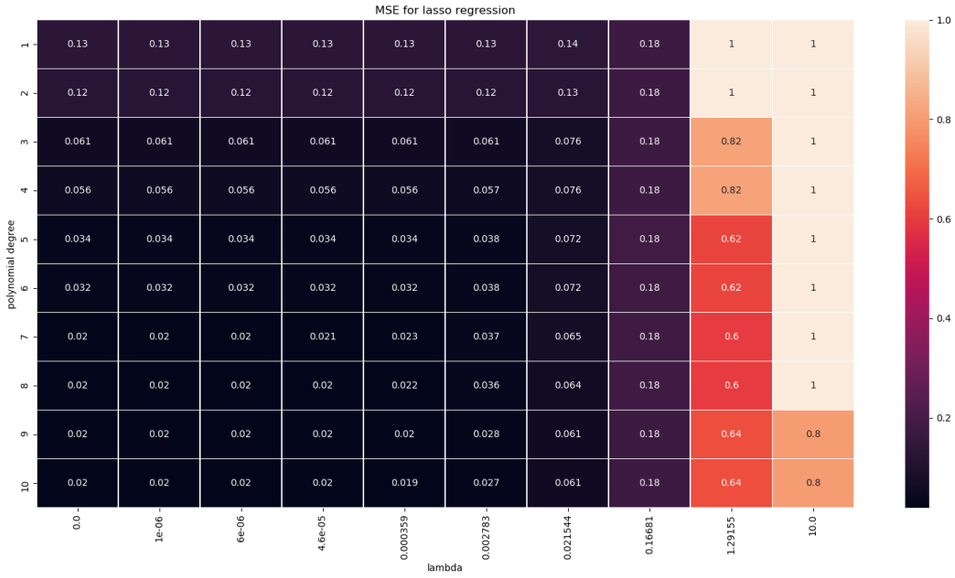

Figure 54: Lasso CV MSE for the terrain data

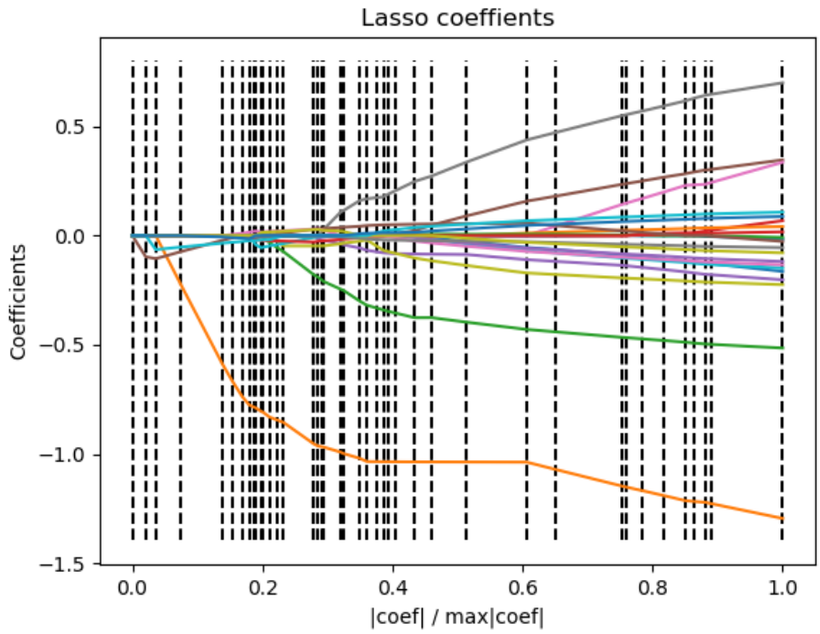

Figure 55: Coefficients of the lasso for polynomial degree 5 for varying lambda

Summary of the methods on terrain data

A table of the best values obtained for MSE for the three methods are given in figure 56.

	MSE
OLS	2.1e-3
Ridge	5.5e-2
Lasso	0.18

Figure 56: Best MSE for each method. In Ridge and Lasso, $\lambda = 0.1$. All values calculated using 5-fold cross validation MSEs with no added noise to the data.

CONCLUSION

After comparing the different regression methods on the two data sets, we see that OLS comes out on top with regards to prediction accuracy. The problem with OLS for high polynomial degree is that we get a lot of coefficients, and interpretation of the model might be hard. For Ridge regression, we have the same problem since it does not actually shrink any coefficients to zero. Lasso on the other hand might give better interpretability by only including the relevant variables as seen in figure 55 and 36. However, since the goal was to get a good prediction, we can conclude that the OLS was the best method for the two data sets. The main reason for not getting better results using Ridge and Lasso are that there was so little variance to begin with, that by increasing lambda, the only significant effect was increasing the bias and thereby the MSE. For further work on the problems, I would suggest implementing the scaling as discussed in the theory part to see if that is of any importance.

References

[1] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics, Second Edition, 2017.

[2] Ludwig Fahrmeier, Thomas Kneib, Stefan Lang and Brian Marx. *Regression.* Springer, 2013.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow.* O'REILLY, 2017.

[4] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Series in Operating Research and Financial Engineering, 2006.

[5] A.M.Legendre. *Noucelles méthodes pour la détermination des ortibes cométes.* Firmin Didot, Paris, 1805.