Project 1 - FYS-STK4155

Oda Langrekken

October 11, 2022

Abstract

We compare different linear regression methods and use resampling methods to investigate how the model complexity and the data size affect the bias and variance. The methods are used to fit the Franke function. We discover that linear regression methods are efficient when fitting the Franke function, and learn more about the bias-variance trade-off and the effect of regularization on the way.

Github repository: `https://github.com/OdaLangrekken/FYS-STK4155/tree/main/Project1`

1 Introduction

Linear regression models are sometimes overlooked because of the great success of more complicated models such as neural networks. Linear regression models are however easy to implement and understand, and can sometimes be surprisingly effective.
In this project we learn more about linear regresssion by implementing some popular methods and fitting them to the Franke function. We use our results to discuss bias-variance trade-off and the danger of overfitting.

We start out by introducing the theory. First we discuss linear regression models in general, before introducing Ordinary Least Squares. Then we consider some methods to test and assess our models, such as resampling methods. Finally we introduce Ridge and Lasso regression.
After the theory we move directly on to results. Results are presented for Ordinary Least Squares, Ridge and Lasso using both boostrap and cross-validation.
Finally the conclusion includes some final thoughts.

2 Methods

Note on notation
In this report I have adopted the notation of Hastie et al [3]. Input features are typically denoted by X. Observed values are written in lowercase, so x_i is the ith observation of input feature X. Vectors are bold if their dimension is n, where n is the number of observations in our data. So all observations of input feature j is denoted by the vector $\mathbf{x}_j = (x_{1j}, x_{2j}, \dots, x_{nj})^T$. Matrices are bold. To denote the matrix consisting of all input I use $\mathbf{X}$, which is an $n \times p$-matrix where p is the number of input features.

2.1 Linear Regressions Models

Let's say we have a matrix of inputs $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p)$, where $\mathbf{x}_j$ is a vector containing all observations of input feature j. In addition we have a corresponding vector of outputs $\mathbf{y}$. The goal of regression models is to find a functional relationship between $\mathbf{X}$ and $\mathbf{y}$ such that

$$\mathbf{y} = f(\mathbf{X}) + \boldsymbol{\epsilon} \tag{1}$$

where $\boldsymbol{\epsilon}$ is some random error.

A linear regression model is linear in its parameters. In a linear regression model we assume that the function $f(\mathbf{X})$ is linear, that is [3]

$$f(\mathbf{X}) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j \tag{2}$$

where the vector $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_p)$ are the coefficients of the model. The coefficient β_0 is commonly known as the bias.
We can write 2 more succinctly by defining $\mathbf{X} = (\mathbf{1}, \mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_p)$, such that

$$f(\mathbf{X}) = \mathbf{X}\boldsymbol{\beta} \tag{3}$$

Given a matrix of observed input $\mathbf{X}$ and a regression method for finding the optimal coefficients we can then predict the output $\mathbf{y}$

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}} \tag{4}$$

where $\hat{\boldsymbol{\beta}}$ and $\hat{\mathbf{y}}$ are our best estimate of the coefficients $\boldsymbol{\beta}$ and output $\mathbf{y}$ respectively given by our chosen regression method.

So how do we find an estimate of the coefficients? We need a regression method and a matrix of observed inputs $\mathbf{X}$ to find our estimate. Generally in supervised machine learning we have a set of observations $(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_n}, y_n)$ that we can use to fit the coefficients to our data. Then all that remains is to chose the regression method. One such method for fitting is the Ordinary Least Squares.

2.1.1 Ordinary Least Squares

The Ordinary Least Squares (OLS) method is among the most popular methods to estimate the coefficients $\boldsymbol{\beta}$ of a linear regression model. In OLS the coefficients are chosen such as to minimize the residual sum of squares:

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{N}(y_1 - f(x_i))^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{5}$$

The minimum of the above equation can be found by setting its derivative with respect to $\boldsymbol{\beta}$ equal to 0:

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

Assuming the matrix $(\mathbf{X}^T\mathbf{X})$ is non-singular such that its inverse exists, 6 has the unique solution

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{7}$$

where we have put a hat on $\boldsymbol{\beta}$ because it is the best estimate of the coefficients when we have chosen the best coefficients to be the ones that minimize the residual sum of squares.

And we can predict the output by

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

Expectations and variances
It is of interest to know the expectation and the variance of $\mathbf{y}$ and $\hat{\boldsymbol{\beta}}$. If we have a random variable $\hat{\mathbf{x}}$ its expected value, also known as the mean, is denoted by $\mathbb{E}[\hat{\mathbf{x}}]$. The variance is a measure of how far the random variable is expected to deviate from the mean, and is given by

$$\mathrm{Var}(\hat{\mathbf{x}}) = \mathbb{E}[(\hat{\mathbf{x}} - \mathbb{E}[\hat{\mathbf{x}}])^2] \tag{9}$$

which we can rewrite as

$$\mathrm{Var}(\hat{x}) = \mathbb{E}[\hat{\mathbf{x}}^2] - \mathbb{E}[\hat{\mathbf{x}}]^2 \tag{10}$$

The following property of the expectation will be useful

$$\mathbb{E}[aY] = a\mathbb{E}[y] \tag{11}$$

when a is a constant.

Additionally we will use the following property of the matrix transpose

$$(AB)^T = B^T A^T \tag{12}$$

Now we have the necessary background to calculate the expectation and variance of y_i. We have a functional relationship between $\mathbf{y}$ and $\mathbf{X}$

$$\mathbf{y} = f(\mathbf{X}) + \boldsymbol{\epsilon}$$

where we assume that $\boldsymbol{\epsilon}$ is a normally distributed error, $\boldsymbol{\epsilon} \sim N(0, \sigma^2)$ and our estimate of $f(\mathbf{X})$ is $\mathbf{X}\boldsymbol{\beta}$. Then

$$\mathbb{E}[y_i] = \mathbb{E}[f(x_i) + \epsilon_i] = \mathbf{X}_{i,*}\boldsymbol{\beta} \tag{13}$$

where we have used that $\mathbb{E}[\epsilon_i] = 0$.

We can now find the variance

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

where we have used $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i^2] = \mathrm{Var}(\epsilon_i) + \mathbb{E}[\epsilon]^2 = \sigma^2$.

So we see that $y_i \sim N(\mathbf{X}_{i,*}\boldsymbol{\beta}, \sigma^2)$.

Now we can find the expectation and variance of our fitted coefficients $\hat{\boldsymbol{\beta}}$. $\hat{\boldsymbol{\beta}}$ is our estimate of $\boldsymbol{\beta}$ given that the observed input is $\mathbf{X}$. Thus $\mathbf{X}$ is a constant.

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

So the expected values of $\hat{\boldsymbol{\beta}}$ are the true values $\boldsymbol{\beta}$ and $\hat{\boldsymbol{\beta}}$ is an unbiased estimator.

Now we find the variance:

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

Now we can use our previous results for $\mathbf{y}$ and equation 10 to find that

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

Then we find

$$\begin{aligned}\text{Var}(\hat{\boldsymbol{\beta}}) &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\sigma^2 - \mathbf{X}\boldsymbol{\beta}\boldsymbol{\beta}^T\mathbf{X}^T)\mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1} + \boldsymbol{\beta}\boldsymbol{\beta}^T \\ &= \sigma^2(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\boldsymbol{\beta}^T + \boldsymbol{\beta}\boldsymbol{\beta}^T \\ &= \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}\end{aligned}$$

So we see that $\hat{\boldsymbol{\beta}} \sim N(\beta, \sigma^2(\mathbf{X}^T\mathbf{X})^{-1})$.

Implementation

Using the powerful numpy library the OLS method is easily implemented in python.

```
    def OLS(X, y):
        """
        Function that finds the coefficients that minimize the residual sum of
    squares.

        Parameters:
            x (array of shape (n, m)): array containing the m different
    features
            y (array of shape (n, 1)): array containing the output

        Output:
            coeffs (array of shape (m+1,1)): coefficients that minimize the
    residual sum of squares
        """
        # Solve for the coefficents
        coeffs = np.linalg.inv(X.T @ X) @ X.T @ y
        return coeffs
```

To test our ordinary least squares method we use our method to fit coefficients for a one-dimensional example. The data points are generated according to $\mathbf{y} = \sin(\mathbf{x}) + \boldsymbol{\epsilon}$ where ϵ follows a gaussian distribution. The result is shown in figure 2.1

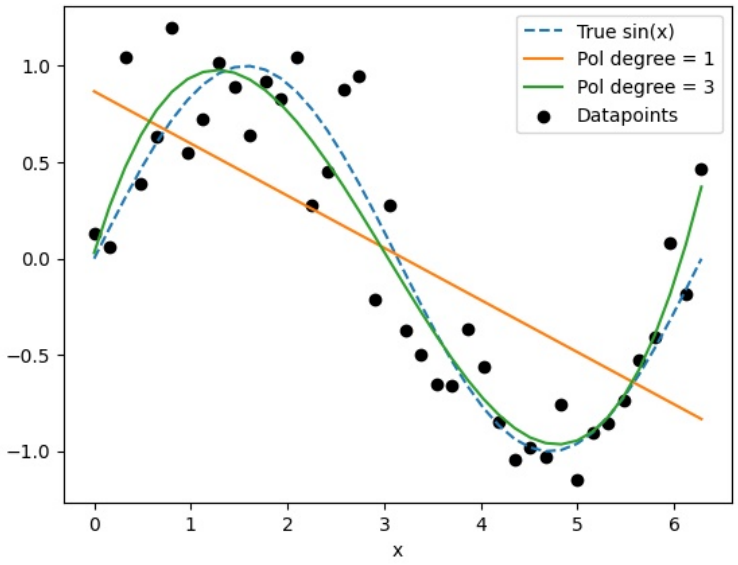


Figure 2.1: Example of fitting to data using ordinary least squares. Here the data points are generated by a sine-function with some gaussian noise added. Curves are predicted for two different degrees of polynomial in the input, 1 (x) and 3 (x^3). The true sine is plotted for comparison.

From 2.1 we see that the predicted curve seems fairly good when we use terms up to a polynomial degree of 3 in the input variable x. For a polynomial degree of 1 the fit fails to capture the periodicity in the data. Based on the curve with polynomial degree 3 this result gives us confidence in our ordinary least squares method. However, this judging my eye approach is not very scientific. We need more sophisticated methods to asses the power of our models and select the best models.

2.2 Model Assesment and Selection

We have seen that the ordinary least squares works well for fitting a sine function. To choose the best model we need more sophisticated regimes for model selection. Let's start with how we measure the peformance of our model through error metrics.

2.2.1 Error metrics

A commonly used metric to evaluate the strength of a linear regression model is the mean squared error (MSE)

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2 \tag{14}$$

where $\hat{y}_i$ is our ith prediction of the output y. The MSE measures the average squared difference between our prediction and the true output value. An MSE of 0 indicates a perfect model.

Another popular error metric for linear regression is the R^2 score

$$R^2(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} \tag{15}$$

where $\bar{y}$ is the mean of $\mathbf{y}$.
The R^2 score is a measure of how well the predictions capture the patterns in the data. An R^2 score of 1 indicates a perfect fit.

2.2.2 Train and test set

Imagine that we train a model, compute the mean squared error and choose the model with the minimal mean squared error. Is this the best approach? If we train and test the model on the same data, we don't know whether the model will generalize well to unseen data. If we train a model that performs extremely well on the training data we risk that the model has been trained to describe every outlier and anomaly in the training data, but fails to capture the general patterns in the data.
To ensure that the prediction power of our model is high when dealing with new and unseen data, we typically split the data in a train set and a test set. The model is trained on the train data, and the best model is chosen by testing on the test data. That way we ensure that the model does not fit the data 'too well', i.e. that it is overfit to the training data.

2.2.3 Bias-variance trade-off

The balance between train and test error can be quantified by the bias variance trade-off. Let's take a closer look at the expected squared difference between our predictions and the true data.

As before we have $\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}$. The approximation to the function f is $\hat{f}(\mathbf{x}) = \mathbf{X}\boldsymbol{\beta}$.

$$\mathbb{E}[(\mathbf{y} - \hat{\mathbf{y}})^2] = \mathbb{E}[\mathbf{y}^2 - 2\mathbf{y}\hat{\mathbf{y}} + \hat{\mathbf{y}}^2]$$

Let's do the first term first. Using 10 and our our result from calculating the expectation and variance of y_i, we have

$$\mathbb{E}[\mathbf{y}^2] = \mathrm{Var}(\mathbf{y}) + \mathbb{E}[\mathbf{y}]^2 = \boldsymbol{\sigma}^2 + \mathbb{E}[f(\mathbf{x}) + \boldsymbol{\epsilon}]^2 = \boldsymbol{\sigma}^2 + f(\mathbf{x})^2$$

where we have used that $\mathbb{E}[\boldsymbol{\epsilon}] = 0$ and that there is no randomness in $f(\mathbf{x})$.
Similarly for the last term:

$$\mathbb{E}[\hat{\mathbf{y}}^2] = \mathrm{Var}(\hat{\mathbf{y}}) + \mathbb{E}[\hat{\mathbf{y}}]^2$$

For the cross-term we use that $\hat{\mathbf{y}}$ and $\boldsymbol{\epsilon}$ are independent:

$$\begin{aligned} -2\mathbb{E}[\mathbf{y}\hat{\mathbf{y}}] &= -2\mathbb{E}[(f(\mathbf{x}) + \boldsymbol{\epsilon})\hat{\mathbf{y}}] \\ &= -2f(\mathbf{x})\mathbb{E}[\hat{\mathbf{y}}] \end{aligned}$$

Now we can put it all together

$$\begin{aligned} \mathbb{E}[\mathbf{y}^2] &= \boldsymbol{\sigma}^2 + f(\mathbf{x})^2 + \mathrm{Var}(\hat{\mathbf{y}}) + \mathbb{E}[\hat{\mathbf{y}}]^2 - 2f(\mathbf{x})\mathbb{E}[\hat{\mathbf{y}}] \\ &= \boldsymbol{\sigma}^2 + (f(\mathbf{x}) - \mathbb{E}[\hat{\mathbf{y}}])^2 + \mathrm{Var}(\hat{\mathbf{y}}) \end{aligned}$$

which is the bias-variance decomposition.

The first term is an irreducible error. This is the variance of the output $\mathbf{y}$ around its mean.
The second term is known as the squared bias. This is a measure of how much the average prediction differs from the actual value. A high squared bias can indicate a model that has not been sufficiently trained.
The last term is the variance. It measures how much spread there is in the estimates. A high variance can be a sign of overfitting, as the model has failed to generalize.

2.2.4 Bootstrap

When splitting the data in a train and test set we risk that the train and test errors do not give a complete picture of the bias-variance trade-off. It might happen that the test set is by chance very similar to the train such that the variance is underestimated, or we might be unlucky and get an outlier in the test data that leads to an overestimated variance.
One method to remedy this is the bootstrap. The idea behind the bootstrap is that we can create pesudo-experiment by sampling with replacement from the dataset. The train and test set is sampled from the original data with replacement, which means that the same data point can be chosen more than once. The model is then trained on the sampled train set and tested on the sampled test set. This is repeated a certain amount of times, and the train and test errors are computed as the averages over all iterations.

Algorithm 1 Boostrap with n iterations

1: **for** $i = 1 : n$ **do**
2: Sample data until train set is same size as input data, reserve remaining data for test set
3: Train model on train set and test on test set
4: Compute test error as average over all n iterations

The function below shows how we can generate one bootstrap sample.

```
def make_bootstrap_sample(X, z, sample_size=1):
    """
    Function that generates one bootstrap sample.

    Input
    -----
    X (dataframe or matrix): design matrix containing all input data
    z (array): array of outputs
    sample_size (float): percentage of input to use for bootstrap sample

    Returns
    -------
    X_sample (dataframe): bootstrap sample of input data
    z_sample (array): output data corresponding to input data in bootstrap
   sample
    X_test (dataframe): input data not sampled, used as test data
    z_test (dataframe): output data corresponding to input data not sampled
    """
    X = X.reset_index(drop=True)
    # Randomly draw n rows from design matrix X with replacement
    X_sample = X.sample(n=sample_size*len(X), replace=True)
    rows_chosen = X_sample.index
    # Choose same rows from z to get output training data
```

```
    z_sample = z[rows_chosen]

    # Use rows not sampled as test set
    #X_test = X[~X.index.isin(rows_chosen)]
    #z_test = np.delete(z, rows_chosen)

    return X_sample, z_sample
```

2.2.5 Cross-validation

When training machine learning models a big obstacle can be lack of data. If we have a limited amount of data for training, it is harder to train models that perform well. In such cases it is not ideal to split the data into a train set and a test set, as we need all the data we can get for training.
A solution to this problem is the k-fold cross validation. The method is as follows:

Algorithm 2 K-fold cross-validation

1: Split the data in K folds of approximately equal size
2: **for** $i = 1 : K$ **do**
3: Train model on all folds except the ith fold
4: Compute test error using ith fold as test data
5: Compute test error as average over all K folds

The method is also illustrated in figure 2.2. Using cross-validation we can use all the data for training, while also splitting out data for testing.

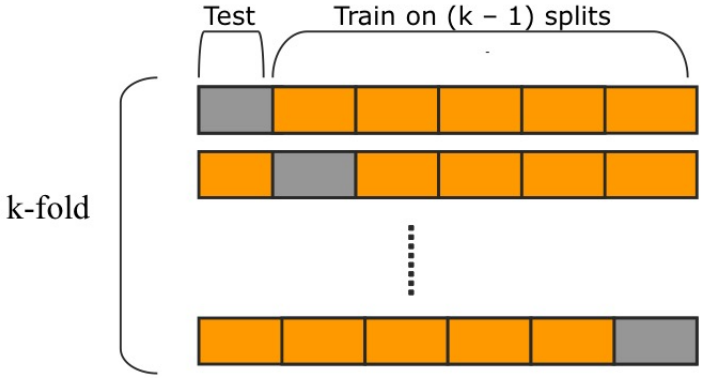


Figure 2.2: An illustration of the k-fold cross-validation, from [1]

We can use cross-validation to calculate test and train mean squared errors by using the following python function

```
def cross_validation(model, X, z, num_folds=5, scale=False):
    """
    Function that uses cross-validation to compute test MSE
    by splitting the data in num_folds folds and computes test error on
    ith fold by training on all except the ith fold. This is repeated for all
   num_folds.
    The test error is computed as the average error over all folds.

```

```
    Input
    -----
        model (LinearModel class instance): model to be trained
        X (dataframe): the design matrix containing all input data
        z (array): an array of outputs
        num_folds (int): number of folds

    Returns
    -----
       mse_test (float): average mean squared error on test set
       mse_train (float): average mean squared error on training set
    """
    # Define empty lists to store mean squared errors
    mse_test = []
    mse_train = []

    # Find size of each fold
    fold_size = int(len(X) / num_folds)

    if ~isinstance(X, pd.DataFrame):
        X = pd.DataFrame(X)


    # Loop thorough all folds
    for i in range(num_folds):
        # Select only the ith fold of the data for the test set
        X_test = X.iloc[i*fold_size:(i+1)*fold_size]
        z_test = z[i*fold_size:(i+1)*fold_size]

        # Make sure all data is included if len_data/num_folds is uneven
        if i == num_folds-1:
            X_test = X.iloc[i*fold_size:]
            z_test = z[i*fold_size:]

        # Select the rest of the data for the training set
        X_train = pd.concat([X.iloc[:i*fold_size], X.iloc[(i+1)*fold_size:]])
        z_train = np.concatenate([z[:i*fold_size], z[(i+1)*fold_size:]])

        if scale:
            scaler = StandardScaler()
            scaler.fit(X_train)

            X_train = scaler.transform(X_train)
            X_test = scaler.transform(X_test)

        # Train the model
        model.fit(X_train, z_train)

        # Make predictions
        z_train_predict = model.predict(X_train)
        z_test_predict = model.predict(X_test)

        # Compute mean squared error for fold
        mse_test.append(MSE(z_test, z_test_predict))
        mse_train.append(MSE(z_train, z_train_predict))

```

```
    # Compute and return average mean squared error
    mse_test = np.mean(mse_test)
    mse_train = np.mean(mse_train)
    return mse_test, mse_train
```

2.3 More Linear Regression Methods

2.3.1 Ridge Regression

With many features as input to our model we run a higher risk of overfitting. One way to produce a model that has lower test error is to either decrease the number of input features or to give each input feature less weight by imposing a penalty on the size of the coefficients. Ridge regression does the latter. In ridge regression the sum of squares is

$$\sum_{i=1}^{N}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \tag{16}$$

where λ is the regularization parameter. By adding a term proportional to β^2 to the error the model penalizes large coefficients. This in turn helps prevent high bias.

As the size of the coefficient affect the error we need scaled data to use ridge regression. All features should have the same scale and be centered at 0. When we have replaced all input $\mathbf{x}_j$ with $\mathbf{x}_j - \bar{\mathbf{x}}$ we can find the coefficients [3]

$$\hat{\beta}^{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{y} \tag{17}$$

So when finding the optimal coefficient using ridge regression a value λ is added to the diagonal of the matrix $\mathbf{X}^T\mathbf{X}$.
In addition to preventing high variance ridge regression solves OLS's potential issue of singluar matrices. Finding the optimal coefficients using ordinary least squares involves inversion of the matrix X^TX. It may happen that the features in the input data are linearly dependent, such that the inverse does not exist. By adding λ to the diagonal we solve the problem with singular matrices.

Implementing ridge regression is quite similar to OLS. One has to be careful to scale and center the data beforehand.

```
def ridge(X, y, lamb):
    """
    Function that finds the coefficients that minimize the residual sum of
   squares using ridge regression.

    Parameters:
        x (array of shape (n, m)): array containing the m different features
        y (array of shape (n, 1)): array containing the output

    Output:
        coeffs (array of shape (m+1,1)): coefficients that minimize the
   residual sum of squares using ridge regression
    """
    # Find number of features
    p = X.shape[1]
```

```
    # Solve for the coefficents
    coeffs = np.linalg.inv(X.T @ X + lamb*np.identity(p)) @ X.T @ y
    return coeffs
```

2.3.2 Lasso Regression

In lasso regression a term proportional with the absolute value of the coefficients is added to the squared error:

$$\sum_{i=1}^{N}\left(y_i-\beta_0-\sum_{j=1}^{p}X_{ij}\beta_j\right)^2+\lambda\sum_{j=1}^{p}|\beta_j| \tag{18}$$

3 Results

3.1 The Franke Function

In the first part of the project we will test the different methods on the Franke function, which is function of two variables given by [2]

$$\begin{aligned} f(x,y)=&\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4}-\frac{(9y-2)^2}{4}\right)+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49}-\frac{(9y+1)}{10}\right)\\ &+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4}-\frac{(9y-3)^2}{4}\right)-\frac{1}{5}\exp\left(-(9x-4)^2-(9y-7)^2\right)\end{aligned} \tag{19}$$

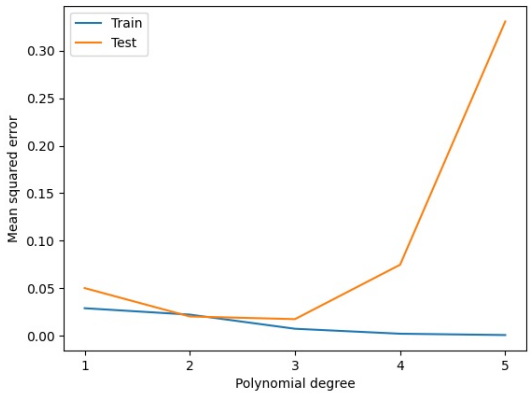
3.2 Franke function with Ordinary Least Squares

First we try fitting coefficients to the Franke function using OLS. In many cases it is important to scale the data before using it to train the model as some methods are sensitive to differences in scale between the input variables. There are many different methods for scaling, bet generally they ensure that the features are distributed equally with the same minimum and maximum values. In our case the data is already scaled, as we have chosen $x, y \in [0, 1]$. Any polynomial of x or y will be distributed between 0 and 1.
Additionally the ordinary least squares gives us an analytical solution for the coefficients. It should then not be sensitive to difference in scale between the features. If we used e.g. gradient descent to find the coefficients it would maybe be another story as gradient descent finds the best coefficients by moving calculating the error and moving the coefficient in the direction of decreasing error. Then difference in scale would mean that error in features with higher scale would have more weight than error in features with smaller scale, which is not ideal. But in our case scaling is of no importance, and I have not scaled the data.

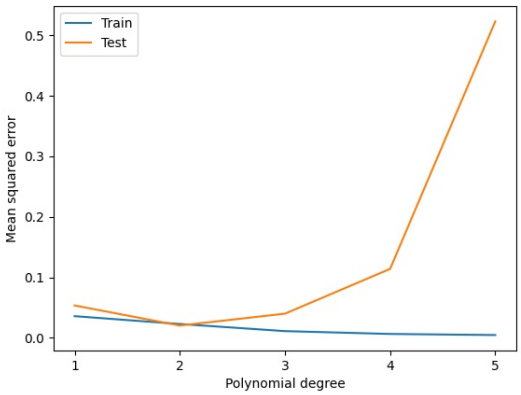
As a sanity check I compared the coefficients from my homemade OLS method to the ones from the python libary sklearn. The results are in table 5.1 in the appendix. The coefficients from the homemade model and the coefficients from sklearn are found to be sufficiently similar.

As the Franke function is exponential we expect the polynomial degree to be important. To asses the model based on polynomial degree the mean squared error and R^2 score are plotted as function of polynomial degree for both the training and the test set in 3.3 and 3.6. In addition the plots

are created both with and without a normal error added to the Franke function, and for different number of data points.

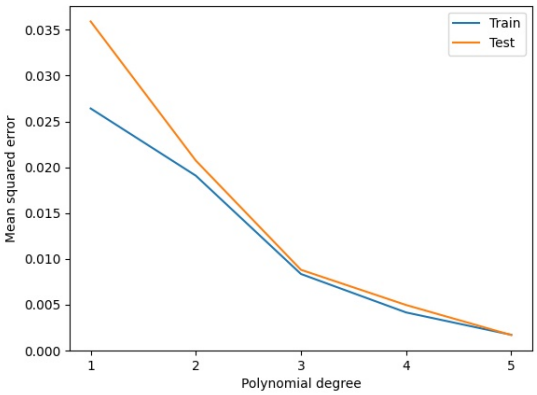


(a) MSE as function of polynomial degree. The data used has 40 data points, and no normal error.

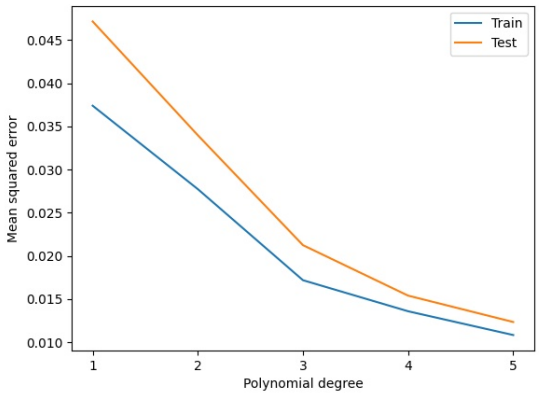


(b) MSE as function of polynomial degree. The data used has 40 data points, and a normal error with $\sigma^2 = 0.1$.

Figure 3.1: Mean squared error as function of model complexity when fitting the Franke function using OSL with 40 data points. The plot to the right has an added normal error term.

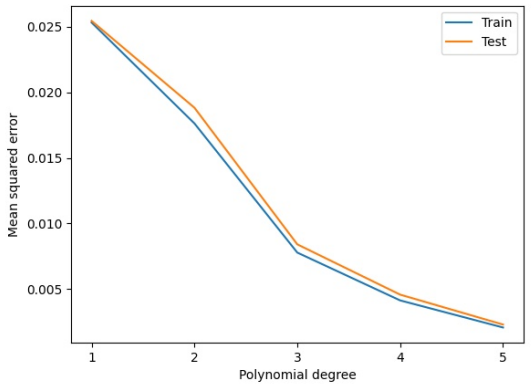


(a) MSE as function of polynomial degree. The data used has 300 data points, and no normal error.

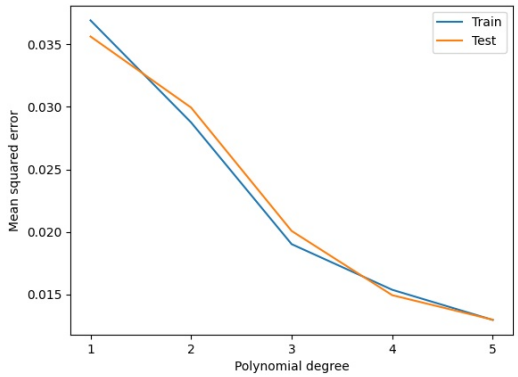


(b) MSE as function of polynomial degree. The data used has 300 data points, and a normal error with $\sigma^2 = 0.1$.

Figure 3.2: Mean squared error as function of model complexity when fitting the Franke function using OSL on 300 data points. The plot to the right has an added normal error term.

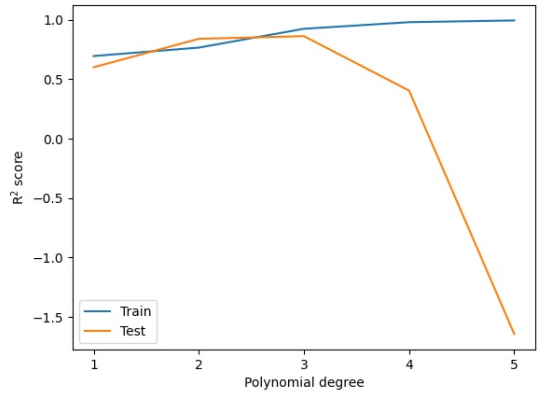


(a) MSE as function of polynomial degree. The data used has 2000 data points and no normal error.

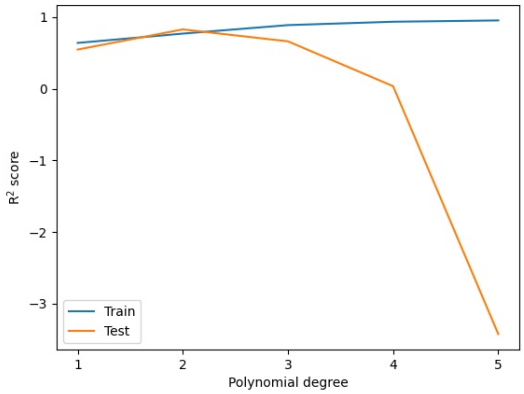


(b) MSE as function of polynomial degree. The data used has 2000 data points, and a normal error with $\sigma^2 = 0.1$.

Figure 3.3: Mean squared error as function of model complexity when fitting the Franke function using OSL on 2000 data points. The plot to the right has an added normal error term.

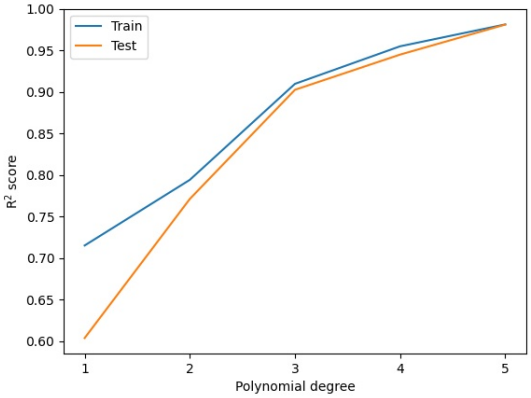


(a) R^2 as function of polynomial degree. The data used has 40 data points, and no normal error.

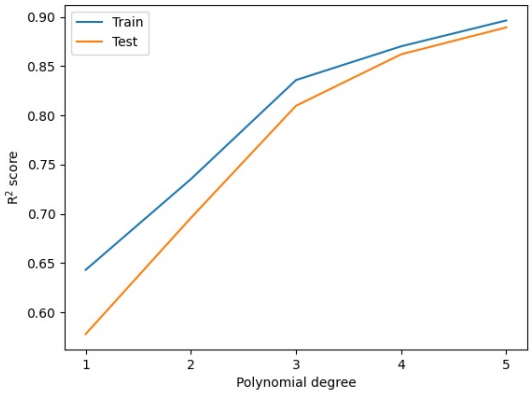


(b) R^2 as function of polynomial degree. The data used has 40 data points, and a normal error with $\sigma^2 = 0.1$.

Figure 3.4: Mean squared error as function of model complexity when fitting the Franke function using OSL on 40 data points. The plot to the right has an added normal error term.

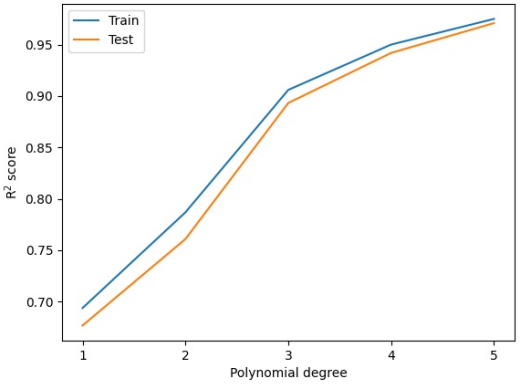


(a) R2 as function of polynomial degree. The data used has 300 data points, and no normal error.

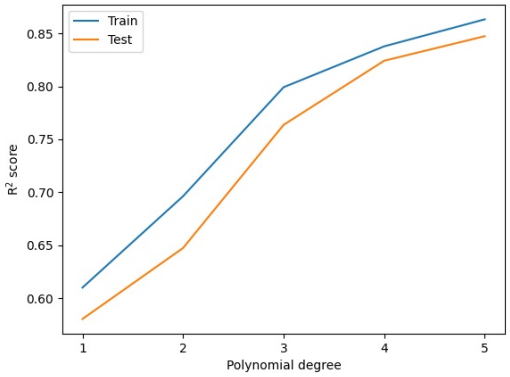


(b) R2 as function of polynomial degree. The function is fitted on 200 datapoints, and a normal error with $\sigma^2 = 0.1$.

Figure 3.5: Mean squared error as function of model complexity when fitting the Franke function using OSL on 300 data points. The plot to the right has an added normal error term.



(a) R2 as function of polynomial degree. The data used has 2000 data points, and no normal error.



(b) R2 as function of polynomial degree. The data used has 2000 data points, and a normal error with $\sigma^2 = 0.1$.

Figure 3.6: R^2 score as function of model complexity when fitting the Franke function using OSL on 2000 data points. The plot to the right has an added normal error term.

Based on the plots 3.1-3.6 the ordinary least squares method seems to perform quite well. Except for the plot where we only have 40 data points the error is decreasing when we increase the complexity of the model.
We can see from the plots that we get much more stable predictions when we have more data. In the plots where we have 2000 data points there is not a lot of difference between train and test error, which means that our model seems to generalize well to unseen data.
For all model the addition of a normal error with $\sigma^2 = 0.1$ increases the error slightly. This is to be expected as adding noise to the data leads to a higher variance in the fitted coefficients, see the calculations of expectations and variances in section 2.1.1.

We can also see that increasing the complexity of the model is beneficial only if we have enough data. For 300 and 2000 data points the mean squared error is decreasing and the R^2 score is approaching 1 for both the train and the test set when we increase the polynomial degree. Figure 3.1a clearly shows that the test error increases a lot at a polynomial degree of 5, while the train error keeps decreasing. Consequently we can conclude that if we have few data points we should try to keep the model complexity low to avoid overfitting.

3.3 Bias-variance trade-off and bootstrap

In the previous section we saw that the mean squared error seems to decrease with the complexity of the model if we have enough data points. Does this mean we should increase the complexity even further? In figure 3.7 we show the mean squared error as a function of polynomial degree, with the polynomial degree going up to ten. We see that while both the train and test errors decrease up to a polynomial degree of 8, the test error starts increasing for polynomial degrees higher than 8. This high-order region is a classical example of a region with high variance and low bias, as discussed in section 2.2.3.

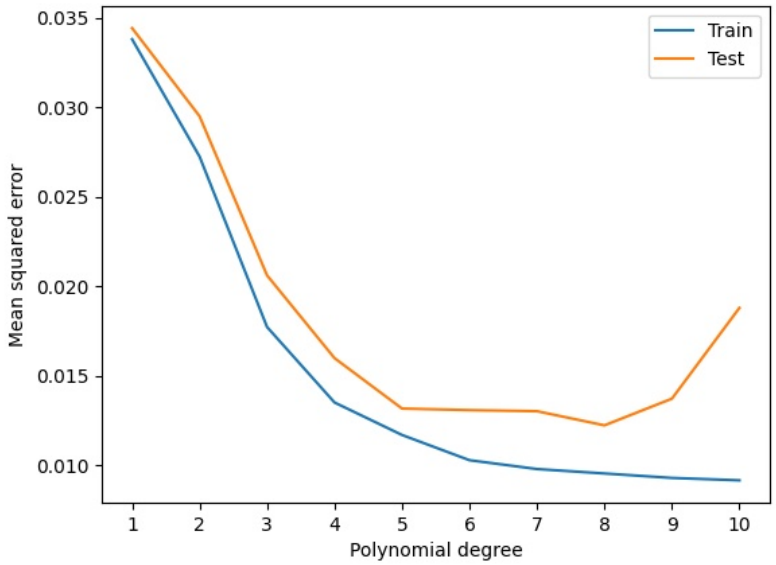


Figure 3.7: Mean squared error as function of the complexity of our model. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

To further analyze the bias and variance of the Franke function we use bootstrap resampling (see section 2.2.4 for more about this method). In figure 3.9 below we can see the MSE calculated using both ordinary least squares with a split in train and test and with bootstrap resampling.
We can see from the figure that the bootstrap errors differ from the errors calculated by splitting the data at high complexity. This suggests that the test data is perhaps too similar to the train data, such that we are too optimistic about the test error.
We also clearly see the same trend as in figure 3.7 with variance increasing with increased complexity.

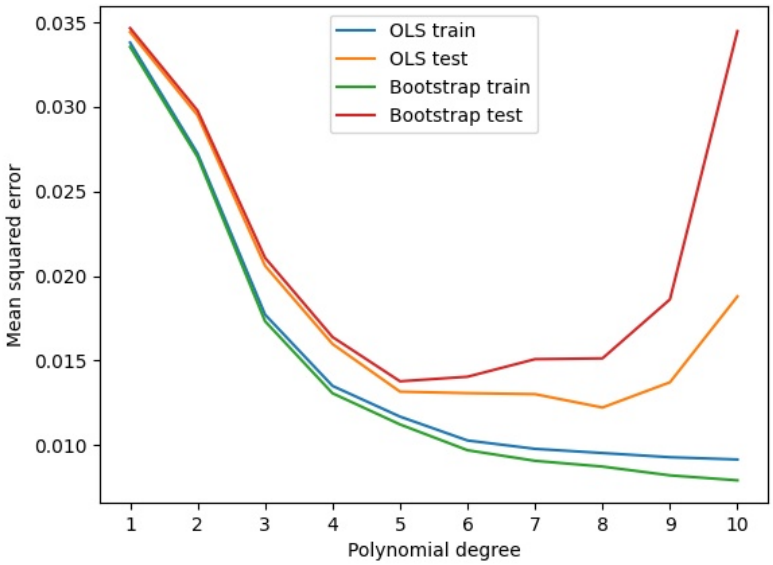


Figure 3.8: Mean squared error as function of the complexity of our model. The bootstrap errors are generated using 100 bootstrap iterations. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

To get a better understanding of the bias-variance trade-off we split the mean squared error into bias^2 and variance. The results are in figure 3.8. We can see that the bias is in general decreasing up to a polynomial degree of 10. At this complexity the model seems to grow unstable. The variance is on the other hand increasing steadily as the complexity is increased, especially at higher polynomial degrees. So we see that in order to ensure low variance we need to sacrifice some bias. The optimal model is the one with the optimal balance of low variance and as low bias as possible without increasing the variance.

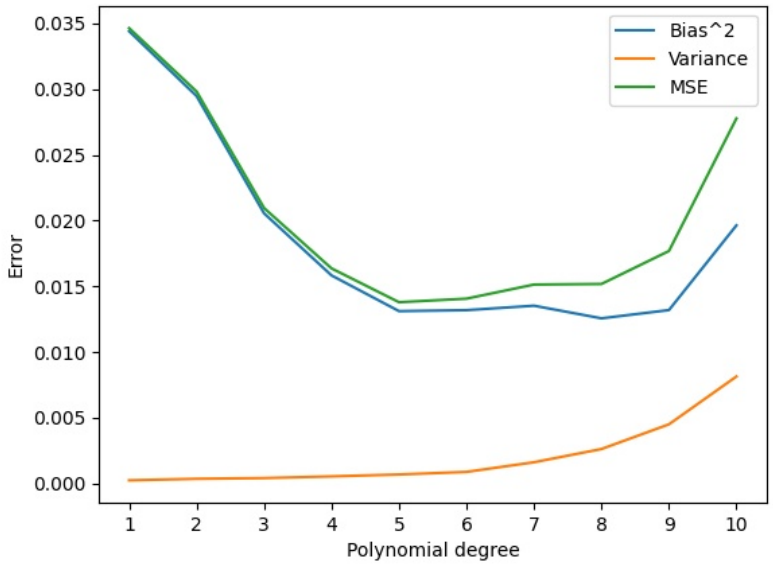
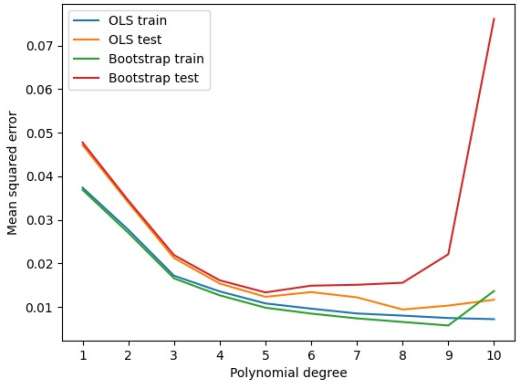
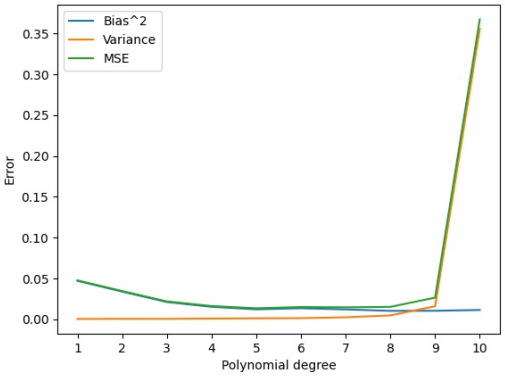


Figure 3.9: Mean squared error as function of the complexity of our model. The bootstrap errors are generated using 100 bootstrap iterations. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

It is interesting to see how the bias and variance is affected by the number of data points used. The figures illustrate how bias and variance change for higher complexities when we have fewer or more data points that in the plot above. As expected the model is more prone to overfitting when we have few datapoints, while more datapoints grants a much more stable model. Thus increasing the complexity does not necessarily increase the variance if we have enough data.

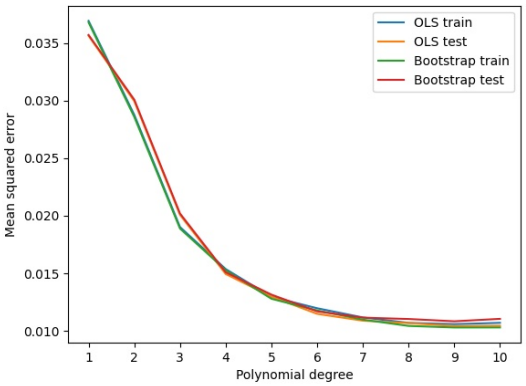


(a) Comparing train and test errors from bootstrap resampling and from regular OLS.

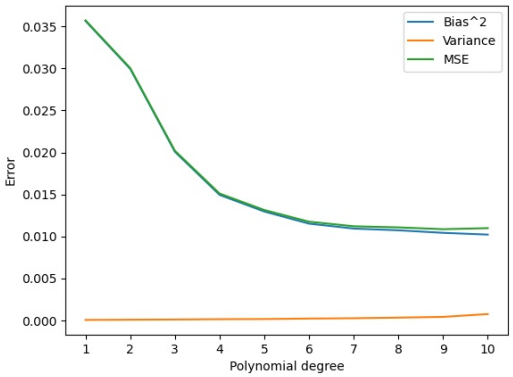


(b) Mean squared error decomposed into bias^2 and variance.

Figure 3.10: Mean squared error as function of model complexity when fitting the Franke function. 100 iterations are used to generate the bootstrap estimates. The data used has 300 data points, and a normal error with $\sigma^2 = 0.1$.



(a) Comparing train and test errors from bootstrap resampling and from regular OLS.



(b) Mean squared error decomposed into bias^2 and variance.

Figure 3.11: Mean squared error as function of model complexity when fitting the Franke function. 100 iterations are used to generate the bootstrap estimates. The data used has 2000 data points, and a normal error with $\sigma^2 = 0.1$.

3.4 Cross-validation

Another popular resampling method is cross-validation (more in section 2.2.5). To test our implementation of cross-validation we performed a comparison with sklearn. The results are in table 5.2 in the appendix. The results are found to be in good accordance with sklearn, except for some slight differences when the ratio data_points/ folds is not an integer.

Folds	Test MSE
5	0.00200938
6	0.00206888
7	0.00208026
8	0.00205038
9	0.00204262
10	0.0020268

Table 3.1: Table displaying test MSE for different number of cross-validation folds when using cross-validation on the Franke function with 300 data points, a polynomial degree of 5 and an error with $\sigma^2 = 0.1$.

In table 3.1 we show the resulting mean squared error from using cross-validation with OLS on 300 data points. The error is calculated for different amounts of folds. The error does not seem to change substantially for folds between 5 and 10.

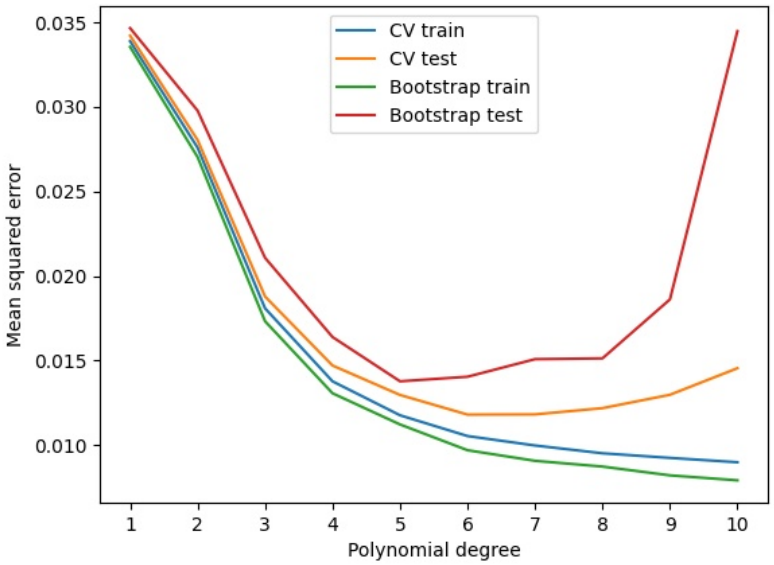
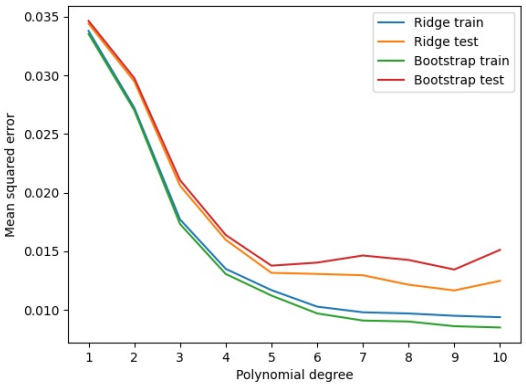


Figure 3.12: Mean squared error as function of the complexity of our model. Errors are estimated using both bootstrap resampling and cross-validation. The bootstrap errors are generated using 100 bootstrap iterations. For cross-validation 5 folds are used. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

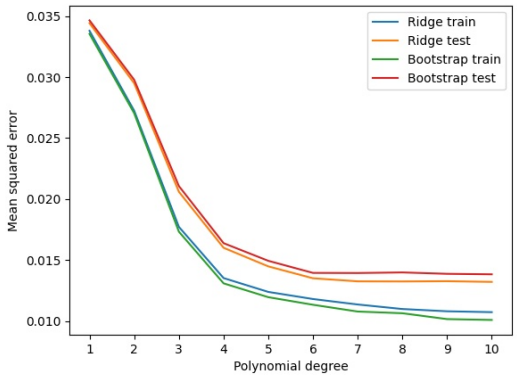
Figure 3.12 compares the errors estimated by bootstrap resampling and the errors estimated by cross-validation. For cross validation the variance is lower.

3.5 Ridge regression

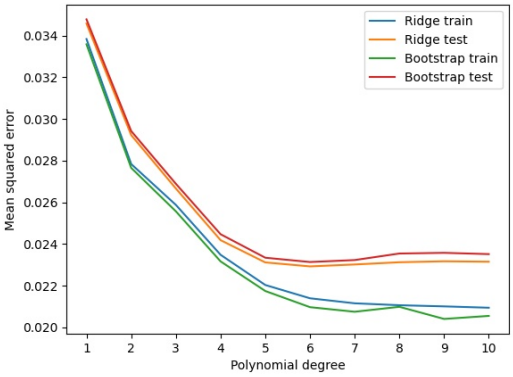
We now use Ridge regression for fitting, in which the size of the coefficients is restricted by adding a term proportional to β^2 to the loss. As the size of the coefficients affect the loss, ridge regression is sensitive to difference in scale between the input features. Our features are already the same scale, but sklearn's `StandardScaler` is used to center the inout data.
The results are below for different values of the regularization parameter, λ. A regularization parameter of 0 is just Ordinary Least Squares. We can clearly see that ridge regression reduces variance compared to OLS, even for very small values of λ. By imposing a penalty on the size of the coefficients, ridge regression makes the model much less likely to overfit. This comes at the expense of higher bias. For the larger regularization parameter $\lambda = 0.5$ below, the bias is greater than for the models with smaller regularization parameters.



(a) $\lambda = 10^{-9}$



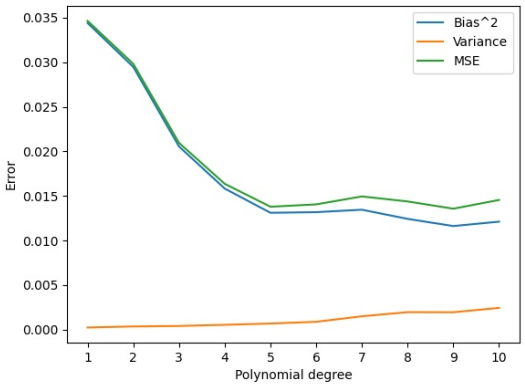
(b) $\lambda = 10^{-5}$



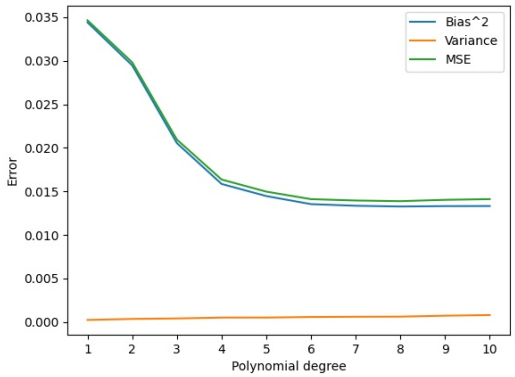
(c) $\lambda = 0.5$

Figure 3.13: Mean squared error, bias and variance as function of model complexity when fitting the Franke function. The functions are fitted using Ridge regression. Several values of the regularization parameter are chosen. 100 iterations are used to generate the bootstrap estimates. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

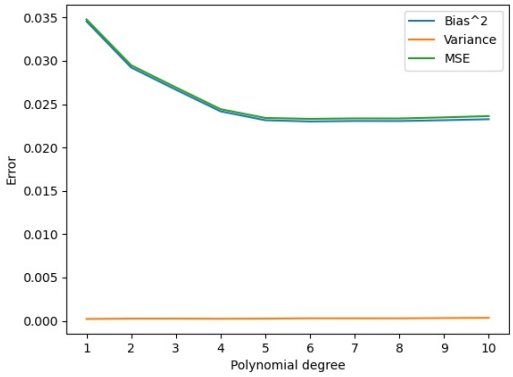
The plots below show the mean squared error decomposed into variance and bias^2. We see the same pattern, where adding a regularization parameter leads to a smaller variance but also a significantly larger bias if the regularization parameter is large enough.



(a) $\lambda = 10^{-9}$



(b) $\lambda = 10^{-5}$



(c) $\lambda = 0.5$

Figure 3.14: Mean squared error as function of model complexity when fitting the Franke function. The functions are fitted using Ridge regression. Several values of the regularization parameter are chosen. 100 iterations are used to generate the bootstrap estimates. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

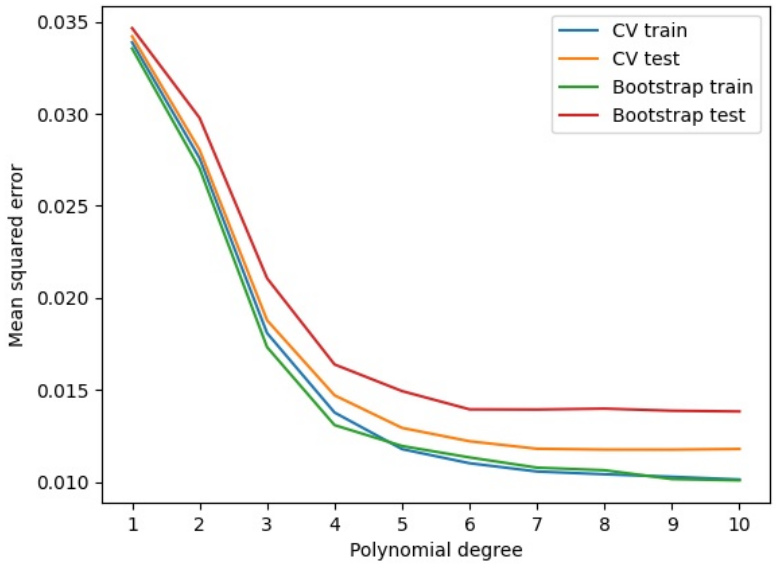


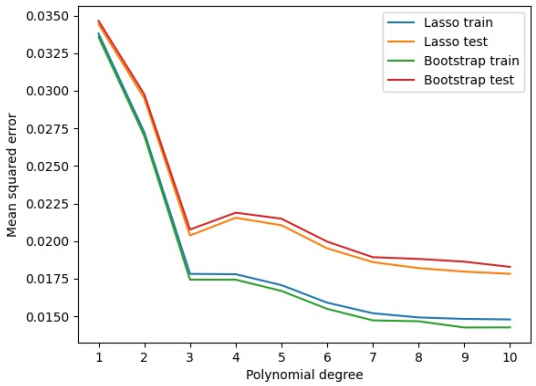
Figure 3.15: Mean squared error as function of the complexity of our model. The model uses Ridge regression with $\lambda = 10^{-5}$. Errors are estimated using both bootstrap resampling and cross-validation. The bootstrap errors are generated using 100 bootstrap iterations. For cross-validation five folds are used. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

3.6 Lasso regression

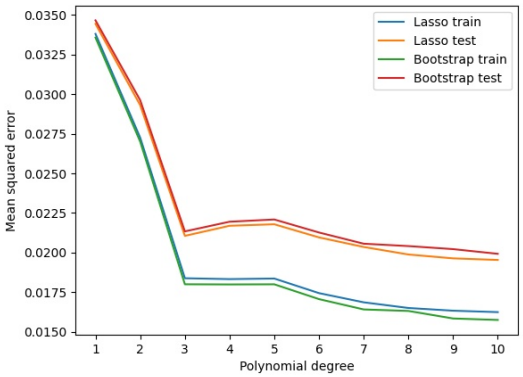
In Lasso regression a term proportional to the absolute value of the coefficients is added to the loss. As the Lasso is more complicated to implement, we have used the model from sklearn.

Below are plots that show the composition of bias and variance. They tell the same story as the plots for ridge regression, so such as not to repeat myself I will let the plots speak for themselves.

We can however see that Ridge seems to perform slightly better than Lasso. Lasso tends to set coefficients to 0, while ridge only makes them small.

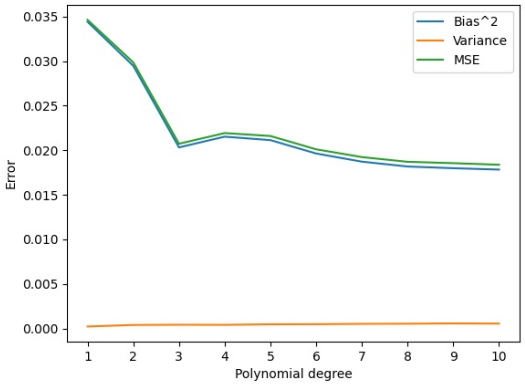


(a) $\lambda = 10^{-9}$

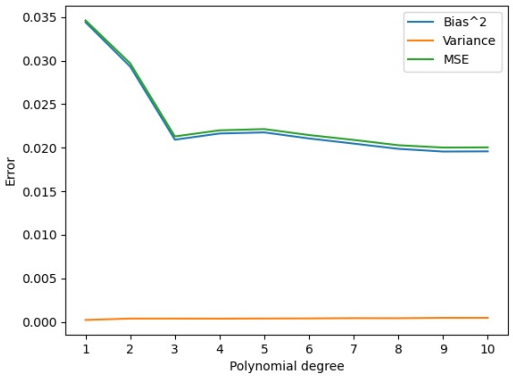


(b) $\lambda = 10^{-5}$

Figure 3.16: Mean squared error, bias and variance as function of model complexity when fitting the Franke function. The functions are fitted using Lasso regression. Several values of the regularization parameter are chosen. 100 iterations are used to generate the bootstrap estimates. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.



(a) $\lambda = 10^{-9}$



(b) $\lambda = 10^{-5}$

Figure 3.17: Mean squared error as function of model complexity when fitting the Franke function. The functions are fitted using Lasso regression. Several values of the regularization parameter are chosen. 100 iterations are used to generate the bootstrap estimates. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

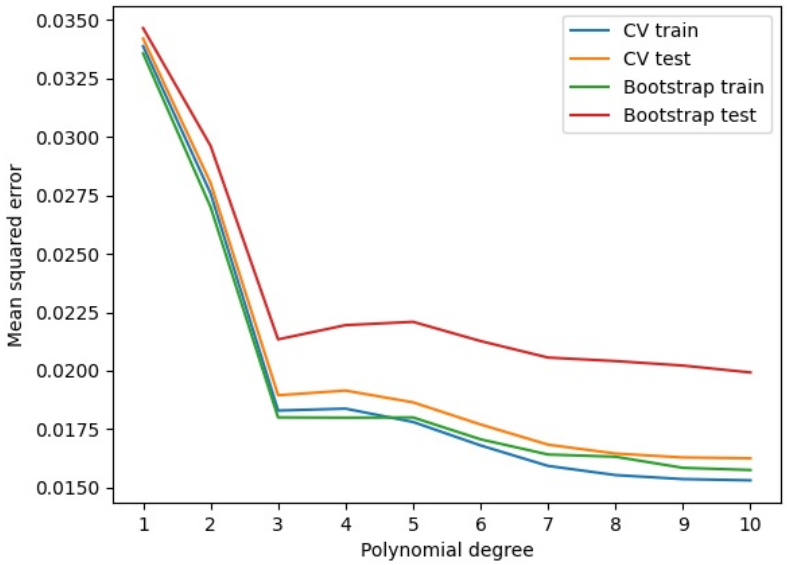


Figure 3.18: Mean squared error as function of the complexity of our model. The model uses Lasso regression with $\lambda = 10^{-5}$. Errors are estimated using both bootstrap resampling and cross-validation. The bootstrap errors are generated using 100 bootstrap iterations. For cross-validation five folds are used. The data used has 600 data points, and a normal error with $\sigma^2 = 0.1$.

4 Conclusions

We have implemented Ordinary Least Squares, Ridge Regression and Lasso Regression and used them to fit the Franke function. Bootstrap resampling and cross-validation were also implemented in order to analyze the bias-variance trade-off for the different models.

We saw that linear regression models perform well when fitting the Franke function. The opacity of the models also enables some investigation of different concepts that are important to understand in order to implement good and robust machine learning models, such as the bias-variance trade-off and the importance of regularization.

It would be very interesting to test the models on real data. Part of this project was testing the models on terrain data. Sadly the deadline came upon me too swiftly, so I did not have time to look into this properly. Linear regression models are often neglected in favor of more complex models such as neural networks, so it would be interesting to test both our homemade linear regression methods and e.g. a convolutional neural network on the terrain data to get a feeling of how much better a neural network would perform and if this better performance is worth the loss of opacity and ease of implementation.

5 Appendix

5.1 Comparisons with sklearn

5.1.1 OSL coefficients

Coefficient	Homemade OLS	Sklearn
Bias	0.0967	0.0967
y^1	6.181	6.181
y^2	-15.47	-15.47
y^3	0.0242	0.0242
y^4	26.1046	26.1046
y^5	-16.8594	-16.8594
x^1	9.8768	9.8768
x^1y^1	-23.0374	-23.0374
x^1y^2	32.6321	32.6321
x^1y^3	-39.7942	-39.7942
x^1y^4	20.9597	20.9597
x^2	-39.7783	-39.7783
x^2y^1	58.0529	58.0529
x^2y^2	-16.0934	-16.0934
x^2y^3	-4.6531	-4.6531
x^3	54.9241	54.9241
x^3y^1	-65.1304	-65.1304
x^3y^2	14.3806	14.3806
x^4	-26.9106	-26.9106
x^4y^1	22.791	22.791
x^5	1.7907	1.7907

Table 5.1: Table comparing fitted coefficients from the homemade OLS and the library sklearn. The coefficients are fitted on the Franke function using a polynomial degree of 5 and 300 data points.

5.1.2 Cross-validation MSEs

Folds	Homemade cross-val	Sklearn
5	0.00200938	0.00200938
6	0.00206888	0.00206888
7	0.00208026	0.00205712
8	0.00205038	0.00205038
9	0.00204262	0.00203867
10	0.0020268	0.0020268

Table 5.2: Table comparing test MSE from the homemade cross validation and the library sklearn for different numbers of cross validation folds. The coefficients are fitted on the Franke function using a polynomial degree of 5 and 300 data points. The discrepancy is caused by difference in how the remaining data points are distributed if the data cannot be distributed evenly between the folds. In the homemade version the remaining data are put in the last fold.

5.1.3 Ridge coefficients

Coefficient	Coefficients (Ridge homemade)	Coefficients (Ridge sklearn)
Bias	0.4121	0.4127
y^1	-0.3484	-0.3484
y^2	-0.8636	-0.8636
y^3	-0.3702	-0.3702
y^4	0.2024	0.2024
y^5	0.6299	0.6299
x^1	-0.5085	-0.5085
x^1y^1	0.339	0.339
x^1y^2	-0.1364	-0.1364
x^1y^3	-0.1263	-0.1263
x^1y^4	0.0165	0.0165
x^2	-0.3731	-0.3731
x^2y^1	0.3731	0.3731
x^2y^2	-0.0329	-0.0329
x^2y^3	-0.1125	-0.1125
x^3	-0.0626	-0.0626
x^3y^1	0.322	0.322
x^3y^2	-0.0463	-0.0463
x^4	0.0238	0.0238
x^4y^1	0.1957	0.1957
x^5	-0.0456	-0.0456

Table 5.3: Table comparing fitted coefficients from homemade ridge regression and sklearn. The coefficients are fitted on the Franke function using a polynomial degree of 5 and 300 data points.

References

[1] Quinkai's blog. Machine learning 9 - more on artificial neural network. `http://qingkaikong.blogspot.com/2017/02/machine-learning-9-more-on-artificial.html`. Accessed: 2022-10-02.

[2] Morten Hjorth-Jensen. Project 1 on machine learning. University Project Decription, 2022.

[3] Jerome Friedman Trevor Hastie, Robert Tibshirani. *The Elements of Statistical Learning*. Springer New York, NY, 2 edition, 2009.