Finding Optimal Regression Method on Three-Dimensional Data

Justina Grønbekk,
Sophia Gaupp,
Mina Piasdatter Walderhaug

UNIVERSITETET
I OSLO

Project 1
of FYS-STK4155 – Applied Data Analysis and Machine Learning.
at the University of Oslo
Oslo, Norway

October 2024

Abstract

In this work, we aim to identify the most effective regression technique for modeling three-dimensional data by comparing three widely used methods: Ordinary Least Squares (OLS), Ridge and Lasso regression.

We implemented these regression methods using Franke's function. First using normally distributed data, then using real world data from a Norwegian landscape. We looked at different performance measures in order to evaluate which regression method performed the best. Modeling and finding the best regression method on topographical data could particularly be relevant for applications such as predicting erosion patterns in geological studies, modeling habitat distributions in biology, and optimizing land use in environmental management, where accurate topographical modeling is crucial for informed decision-making.

We found that OLS was most successful when using the Franke's function. However the implementations on real world data ended up being not a trivial task to solve and the data seemed to be too complex to be captured by any of the models we have implemented.

Contents

Chapter 1

Introduction

Machine Learning has garnered significant attention in recent years. Particularly with the rise of widely used large language models such as ChatGPT, the field finds its way into our daily lives more and more. Understanding the inner workings of these sophisticated models is becoming a pressing issue of our time. While these methods all seem so advanced and complex, a surprisingly large part of Machine Learning is the rather underwhelming task of optimizing parameters and fine-tuning the models (Hjorth-Jensen 2024).

When starting a new project, the first and most important step is selecting the method that best fits the project's requirements. This principle aligns with the concept that failing early means failing cheaply. The challenge lies in identifying the right method but also optimizing the relevant parameters for the task at hand.

In this work, we focus on finding the optimal regression technique and refining the model's parameters. We use Franke's function as a reference to establish a baseline, optimizing the parameters accordingly and utilizing it as a foundation for further adjustments. We apply three widely-used regression methods: Ordinary Least Squares (OLS), Ridge and Lasso regression. These methods are then implemented on digital terrain data from a Norwegian landscape. Interestingly, while OLS is a relatively simple approach, it is a successful option when implementing it with Franke's function.

Here we examine the underlying methodologies and implementations of each regression model. Additionally, we explore different performance measures and their implications, allowing us to compare the various regression techniques and ultimately concluding which method performs best.

Chapter 2

Methods

2.1 Franke's function

Franke's function is a two-dimensional function that we use to test our fitting algorithm before applying it to our topographical data. Franke's function is a weighted sum of four exponentials (Hjorth-Jensen 2024) and is given by

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

It is only defined in the range $[0,1]$, so we have a specific domain for the data.

We used Franke's function for testing our algorithms for the different regression models. We also looked at resampling techniques (which we explain later in the report) before testing on the topographical data.

2.2 Regression models

In Machine Learning we use different models for regression analysis. The point of doing a regression analysis is to find a relationship between the input $\mathbf{x}$ and the function $\mathbf{y}$, or to infer dependencies, and many other possible uses (Hjorth-Jensen 2024). As we use Franke's function, the input is then x and y, with the output being z.

There are multiple regression models that can be used, but in this report we use OLS, Ridge and Lasso regression.

2.2.1 Ordinary Least Squares

Ordinary Least Squares, shortened as OLS, is a linear regression model. OLS aims to describe y in terms of $\mathbf{X}$, the design matrix (Hjorth-Jensen 2024). As the form of the function is unknown, it is common to assume a linear relationship between $\mathbf{X}$ and $\mathbf{y}$. This gives us the regression parameters $\boldsymbol{\beta}$.

We then have an analytical expression for the parameters of $\boldsymbol{\beta}$. We can write our approximation as

$$\tilde{\mathbf{y}} = \mathbf{X}\beta$$

And then we solve a so-called Cost-Function to find the optimal parameters $\tilde{\boldsymbol{\beta}}$. We use the Mean Squared Error (MSE) (see section 2.3.1) to define this cost function and it is given by

$$\mathbf{C}(\beta) = \frac{\mathbf{1}}{\mathbf{n}}[(\mathbf{y} - \mathbf{X}\beta)^{\mathbf{T}}(\mathbf{y} - \mathbf{X}\beta)]$$

We then minimize the spread of $\mathbf{C}(\beta)$, and using the derivative we end up with the expression for $\tilde{\boldsymbol{\beta}}$ being

$$\tilde{\beta} = (\mathbf{X^T X})^{-\mathbf{1}}\mathbf{X^T y}.$$

2.2.2 Ridge Regression

Sometimes OLS does not give a good model for the data. We can then add in a regularization parameter λ to the cost-function. This reduces the likeliness of overfitting, and can penalize the different parameters. This then becomes Ridge regression.

The cost-function here is given by

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

We then require the 2-norm vector $||\beta||_2^2 < t$, where t is a finite number larger than zero (Hjorth-Jensen 2024).

We then use the derivative to minimize the spread of the cost function, and again find the optimal parameter of β, which is given by

$$\tilde{\beta}_{\mathbf{Ridge}} = (\mathbf{X^T X} + \lambda\mathbf{I})^{-\mathbf{1}}\mathbf{X^T y},$$

where I is the p-dimensional identity matrix. This will affect the Bias-Variance Tradeoff, which we will discuss in section 2.4, with the variance decreasing with increasing λ.

2.2.3 Lasso Regression

Lasso regression stands for least absolute shrinkage and selection operator (Hjorth-Jensen 2024). Lasso regression also has the regularization parameter λ, but the cost-function is defined as

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

where we use the 1-norm vector, $||\beta||_1$ instead.

When we take the derivative of this function (to minimize the spread), we do not get the same analytical equation as in OLS and Ridge. Instead we get the equation

$$(\mathbf{X^T X}\beta + \lambda\mathbf{sgn}(\beta)) = \mathbf{2X^T y},$$

where $sgn(\beta) = \frac{\partial|\beta|}{\partial\beta}$. In implementation of this we use the Scikit-Learn library (Pedregosa et al. 2011) in Python. It will give us the optimal $\tilde{\beta}$ which we in turn use to plot our polynomial.

Lasso can be favored as it can drive some of the parameters to become zero depending on the regularization strength. It is useful as a supervised feature selection technique (Raschka et al. 2022).

2.3 Performance measures

We evaluate the different regression models based on **MSE** and $\mathbf{R^2}$.

2.3.1 Mean Squared Error

MSE is the mean squared error, and it is used to evaluate the error of the model compared to the actual data. We typically want to have the lowest MSE possible. It is given by

$$MSE = \frac{1}{N}\sum(\tilde{y} - y)^2$$

We also note that the MSE can never be negative as it is squared.

2.3.2 R^2

R^2 is the coefficient of determination. It is used to measure the goodness of fit of the model. We typically want the highest R^2 possible. It is given by

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

2.4 Bias-Variance Trade-off

Bias describes the accuracy of the prediction of a model. A model that is too simple might not represent the training data well. Such as a model has high bias and leads to higher training and testing error (Goel 2020). A model with low bias fits the training data well, while a model with high bias fits the training data poorly.

Variance is a measure of dispersion. It measures how far away from the mean each point is. High variance means that there is great dispersion, while low variance means that there is little dispersion. We want low variance, as this means the sampled data does not diverge much from the mean.

It might seem that the best model is the most complex model, but that is not necessarily the case. If the model is needlessly complex, it can be prone to overfitting. This means that the model has perfected itself on the training data but works poorly on data it has never seen before. An overfitted model has low bias and high variance. A too simple model can be underfitted. This means that the model fails to predict the training data well. An underfitted model has high bias and low variance. A model should therefore neither be too simple, nor too complex, with low bias and low variance.

The best model has low bias and low variance. The process of finding a model that fits both these criteria is called the Bias-Variance Trade-Off. As it is not always easy to find a perfect model, one usually needs to accept a compromise of bias and variance. You can rewrite the error as a sum of the bias, variance and an irreducible error.

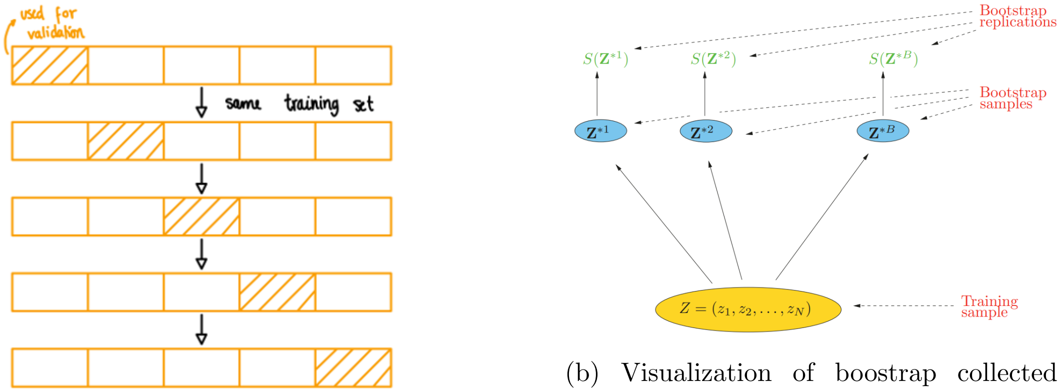
$$\mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right] = \mathrm{Bias}[\tilde{y}] + \mathrm{var}[\tilde{y}] + \sigma^2$$

See Appendix A for the calculation of this expression.

2.5 Resampling

Resampling is an important method for assessing the accuracy of models. The general idea is to divide the data into several smaller datasets, then test the model on each subset individually. The last step is to calculate the mean of each result, and compare it with the model.

Two of the most popular methods of resampling are bootstrap and k-fold cross-validation.



(a) 5-fold cross validation

(b) Visualization of boostrap collected from (Hastie, Tibshirani, and Friedman 2009, 250)

Figure 2.1: A visual representation of k-fold cross validation and bootstrap resampling. For (b), Z is our original dataset. Z*B are the bootstrap datasets with replacements, and S is the model we want to assess.

2.5.1 Bootstrap

The bootstrap model can be used to estimate prediction error. This is done by fitting the model to a set of bootstrap datasets and keeping track of how well the model predicts the training error for each bootstrap set (Hastie, Tibshirani, and Friedman 2009, 249). It can be described through these steps:

(1) Draw random datasets from the training data.

(a) Each sample is the size of the original training set, with replacements.

(2) Repeat this process k times, producing k bootstrap datasets.
(3) Apply the model to each bootstrap dataset.
(4) Calculate the MSE of (3).
(5) See how well the bootstrap resampling predicts the original training set.

2.5.2 Cross-Validation

Cross-Validation can be described as such:

(1) Shuffle the data.
(2) Split the data into k equal data sets.
(3) Choose some of the k sets for training, and some for testing.
(4) Perform the model on each set.
(5) Repeat k times.
(6) Calculate the MSE of each iteration.
(7) Compare with results from model without cross-validation.

Chapter 3

Implementation

Code/Implementations/test: Readability of code, implementation, testing and discussion of benchmarks. Total number of possible points 20

3.1 Franke's function

We implement Franke's function with the following code:

```python
def FrankeFunction(x,y):
    term1 = 0.75*np.exp(-(0.25*(9*x-2)**2) - 0.25*((9*y-2)**2))
    term2 = 0.75*np.exp(-((9*x+1)**2)/49.0 - 0.1*(9*y+1))
    term3 = 0.5*np.exp(-(9*x-7)**2/4.0 - 0.25*((9*y-3)**2))
    term4 = -0.2*np.exp(-(9*x-4)**2 - (9*y-7)**2)
    return term1 + term2 + term3 + term4
```

3.2 Regression models

Here we show our implementation of the different regression models, where we use normally distributed data. In the implementation of the different regression models we first need to define the function for the design matrix, which is the same for all the methods:

```python
# Function to create a design matrix for polynomial terms up to a given
    degree
def create_design_matrix(x, y, degree):
    N = len(x)
    num_terms = (degree + 1) * (degree + 2) // 2  # Number of polynomial
     terms up to the given degree
    X = np.ones((N, num_terms))  # Initialize the design matrix
    index = 1
    for i in range(1, degree+1):
        for j in range(i+1):
            X[:, index] = (x ** (i-j)) * (y ** j)
            index += 1
    return X
```

Furthermore, we generate normally distributed data points, flatten them and compute the Franke function with these data points and add noise.

We then split the data in training and test sets. This is because we want to avoid overfitting and to have a model that will be able to predict new data in a general way. Specifically, we train our model using the training data, and then we test it with our test data to see if it works for data it has not been trained with.

After this, we reshape the data into 2D arrays and scale the data using the Standard Scaler from the scikit-learn library (Pedregosa et al. 2011). This ensures that each feature has a mean of 0 and a standard deviation of 1, standardizing the data. We applied this to improve the performance and stability of the regression models, particularly Ridge regression and Lasso, which are sensitive to the scale of input data. Again, this is the same for all methods of regression analysis.

We can first have a look at the implementation of OLS:

```python
# Lists to store MSE, R2 scores, and beta coefficients
degrees = list(range(1, 6))  # Degrees from 1 to 5
mse_train_scores = []
mse_test_scores = []
r2_train_scores = []
r2_test_scores = []
betas = []  # List to store the beta coefficients for each degree

# Loop over different polynomial degrees
for degree in degrees:
    # Create the design matrix for the current degree for training and testing data
    X_train = create_design_matrix(x_train_scaled.flatten(), y_train_scaled.flatten(), degree)
    X_test = create_design_matrix(x_test_scaled.flatten(), y_test_scaled.flatten(), degree)

    # Perform the OLS regression using the normal equation
    X_train_T = X_train.T
    beta = inv(X_train_T @ X_train) @ X_train_T @ z_train

    # Predict values based on the OLS model
    z_train_pred = X_train @ beta
```

```
21      z_test_pred = X_test @ beta
22
23      # Compute MSE and R2 scores for both training and test data
24      mse_train = mean_squared_error(z_train, z_train_pred)
25      mse_test = mean_squared_error(z_test, z_test_pred)
26      r2_train = r2_score(z_train, z_train_pred)
27      r2_test = r2_score(z_test, z_test_pred)
28
29      # Append the scores to the lists
30      mse_train_scores.append(mse_train)
31      mse_test_scores.append(mse_test)
32      r2_train_scores.append(r2_train)
33      r2_test_scores.append(r2_test)
34
35      # Store the beta coefficients
36      betas.append(beta)
```

Here we see that we first create empty lists to be able to store different values when we loop over different polynomial degrees. We wanted to store the MSE, R^2 and β values to be able to evaluate our models, and look at the polynomial fit.

In line 15-17, we have the actual computation of $\tilde{\beta}$ using OLS. We use the stored values to be able to plot these in the figures presented in the Results Section.

Now, for the Ridge regression, much of the code is the same. But as stated earlier, we add the regularization parameter λ. So, the lines 15-17 become this:

```
    # Perform Ridge regression using the normal equation
    X_train_T = X_train.T
    identity_matrix = np.eye(X_train.shape[1])
    beta = inv(X_train_T @ X_train + lambda_val * identity_matrix) @ X_train_T @ z_train
```

Lastly, we will look at the implementation of Lasso regression. Remember that there was no nice analytical expression for $\tilde{\beta}$, so the implementation looks like this:

```
    # Perform Lasso regression using scikit-learn's Lasso model
    lasso_model = Lasso(alpha=lambda_val, max_iter=10000)  # alpha corresponds to lambda
    lasso_model.fit(X_train, z_train)  # Fit the Lasso model to the training data

    # Get the beta coefficients from the trained model (optional, for inspection)
    beta = lasso_model.coef_
```

Here we imported different functionalities from the Scikit-Learn library (Pedregosa et al. 2011) to be able to find the optimal values of β.

3.3 Resampling

We have implemented both bootstrap and k-fold cross-validation. We have been inspired by code provided by ChatGPT (ChatGPT 2024).

3.3.1 Bootstrap

Before implementing bootstrap, we had to create a design matrix X using the design matrix and Franke's function code. When implementing bootstrap for the topographical data, we used that data to create X instead. See our github repository for the entire code.

Firstly, we defined an empty list that would store the MSE for each bootstrap set.

```python
def bootstrap_resampling(X, z, num_bootstrap_samples, degree, model):
    n, m = X.shape
    mse_bootstrap = []

    for _ in range(num_bootstrap_samples):
        # Generate random indices with replacement
        bootstrap_indices = np.random.choice(n, size=n, replace=True)
        oob_indices = np.setdiff1d(np.arange(n), bootstrap_indices)

        # Bootstrap training data
        X_train = X[bootstrap_indices]
        z_train = z[bootstrap_indices]

        # Out-of-bag test data
        X_test = X[oob_indices]
        z_test = z[oob_indices]

        if model == "Lasso":
            # Fit the Lasso model
            lasso_model = Lasso(0.1, max_iter=10000)
            lasso_model.fit(X_train, z_train)
            z_test_pred = lasso_model.predict(X_test)
        elif model == "Ridge":
            X_train_T = X_train.T
            identity_matrix = np.eye(X_train.shape[1])
            beta = inv(X_train_T @ X_train + degree * identity_matrix) @ X_train_T @ z_train
            z_test_pred = X_test @ beta
        elif model == "OLS":
            X_train_T = X_train.T
            beta = inv(X_train_T @ X_train) @ X_train_T @ z_train
            z_test_pred = X_test @ beta
        else:
            raise Exception("does not recognize model")

        # Calculate MSE for the OOB samples
        mse_test = mean_squared_error(z_test, z_test_pred)
        mse_bootstrap.append(mse_test)

    # Calculate the mean and standard deviation of MSE
    mean_mse = np.mean(mse_bootstrap)
    std_mse = np.std(mse_bootstrap)

    return mean_mse, std_mse
```

Next, we had to iterate through each bootstrap dataset, and generate test and training data for each set. Further, we performed our model on the bootstrap dataset, and stored the MSE of the result in the list mse_bootstrap defined earlier. We return the average MSE across each bootstrap dataset. Lastly, we iterated through degrees one to five, and plotted it.

3.3.2 Cross-Validation

Like with bootstrap, we first created a matrix X using the design matrix and Franke's function. For the topographical data, we used that to create X instead. See the GitHub repository for this code.

Firstly, we split and shuffle the data. Then we define an empty list that will contain the MSE from the different iterations.

```
# Implement k-fold cross-validation
def k_fold_cross_validation(X, z, k, lambda_val, model):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    mse_folds = []
```

We then iterated through each of the k folds, performing the model of choice on each iteration. Lastly, for each iteration, we appended the MSE to the list mse_folds.

```
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        z_train, z_test = z[train_index], z[test_index]

        if model == "Lasso":
            lasso_model = Lasso(alpha=lambda_val, max_iter=10000)
            lasso_model.fit(X_train, z_train)
            z_test_pred = lasso_model.predict(X_test)
        elif model == "Ridge":
            X_train_T = X_train.T
            identity_matrix = np.eye(X_train.shape[1])
            beta = inv(X_train_T @ X_train + lambda_val * identity_matrix) @ X_train_T @ z_train
            z_test_pred = X_test @ beta
        elif model == "OLS":
            X_train_T = X_train.T
            beta = inv(X_train_T @ X_train) @ X_train_T @ z_train
            z_test_pred = X_test @ beta
        else:
            raise Exception("does not recognize model")

        # Predict and calculate MSE for the test fold
        mse_test = mean_squared_error(z_test, z_test_pred)
        mse_folds.append(mse_test)

    return np.mean(mse_folds), np.std(mse_folds)  # Return the average MSE and its standard deviation
```

We then performed cross validation for several different lambda values, and plotted it into a graph to compare with non-resampled data.

3.3.3 Topographical Data

For our implementation of topographical data, we imported the data from a TIFF file:

```
# Load the terrain
terrain1 = imread('SRTM_data_Norway_1.tif')
```

And then we generate x and y data points from the imported data:

```
# Use data points from TIFF file
terrain1_array = np.array(terrain1)
# Get image dimensions
height, width = terrain1_array.shape[:2]
# Create meshgrid for X and Y coordinates
x, y = np.meshgrid(range(width), range(height))
```

The data points are then flattened, and we use them for our regression models instead of normally distributed data points.

We scale the data points to make sure they are inside the domain of Franke's function.

Chapter 4

Results

Analysis: of results and the effectiveness of their selection and presentation. Are the results well understood and discussed? Total number of possible points: 20

4.1 Franke's function

In this analysis, we employed the two performance measures MSE and R^2 to assess the effectiveness of various regression models.

The MSE quantifies the average squared differences between the predicted values and the actual outcomes. It gives us an indication of how far, on average, our predictions deviate from the true values. A lower MSE implies better model performance, with an ideal value of zero indicating a perfect match between predictions and actual observations.

R^2 looks at how well the observed outcomes are depicted by the model. It is based on the proportion of total variation of outcomes explained by the model. So in comparison to MSE, R^2 also looks at the variance of our model. An R^2 of 1 indicates a perfect fit, where the model's predictions fully explain the variability in the data.

4.1.1 Ordinary Least Squares

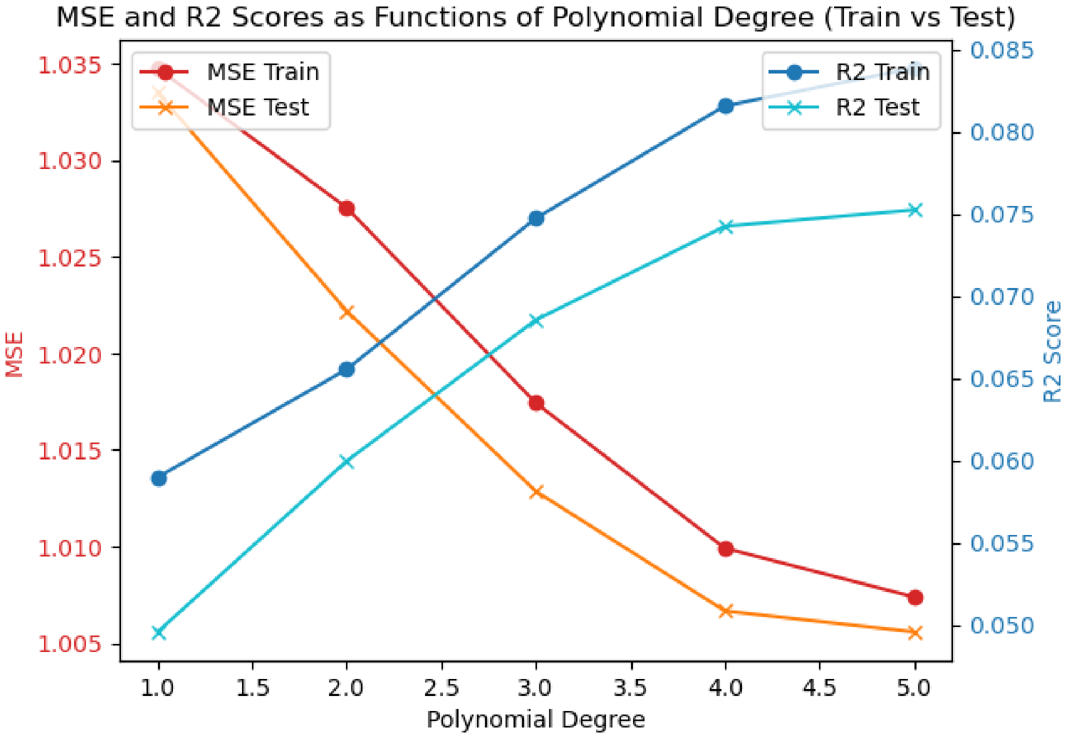


Figure 4.1: Regression method used is OLS. X-axis depicts the different polynomial degrees. Two y-axes show the MSE as well as the R2 values.

As shown in Figure 4.1, the MSE consistently decreases for both the training and test datasets as the polynomial degree increases, indicating an improvement in model fit. In our analysis, we focused on polynomial degrees ranging from one to five. Along with the reduction in MSE, the R^2 score exhibits an increasing trend, reflecting better explanatory power of the model as the polynomial complexity grows. At a polynomial degree of five, the MSE for both training and test datasets falls within a narrow range, between 1.005 and 1.010. The R^2 score at this degree is approximately 0.075 for the test data and 0.085 for the training data, highlighting a slight overfitting trend where the training set is fitted more closely than the test set. This pattern suggests that with OLS higher polynomial degrees improve the fit.

4.1.2 Ridge Regression

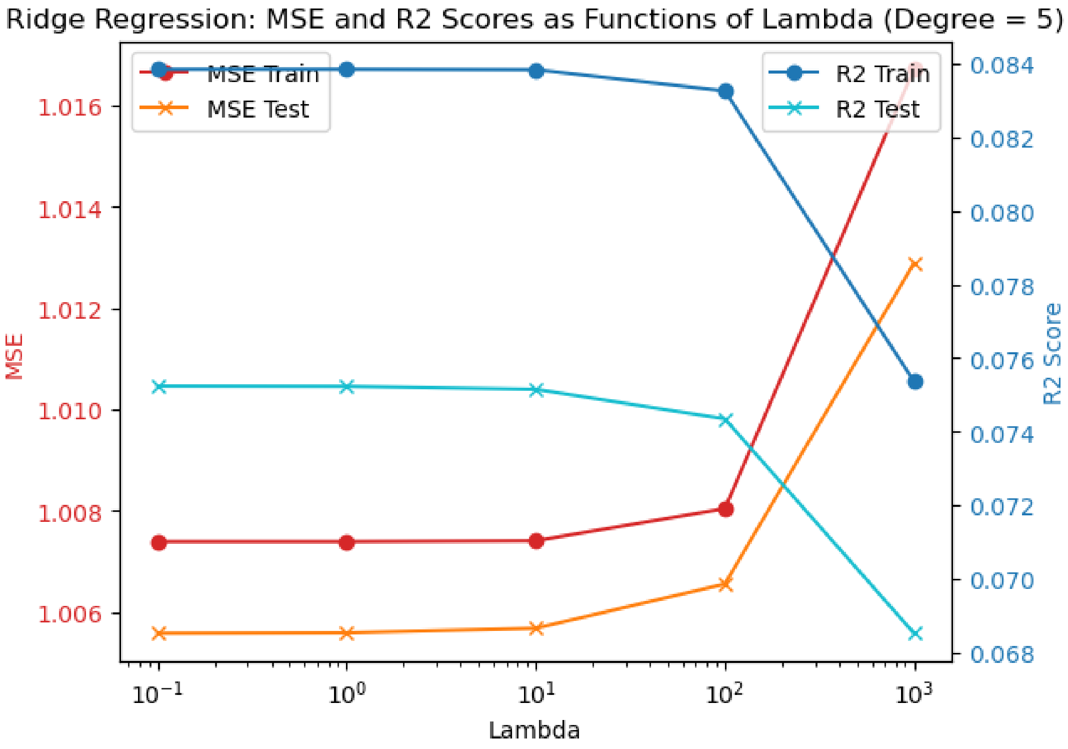
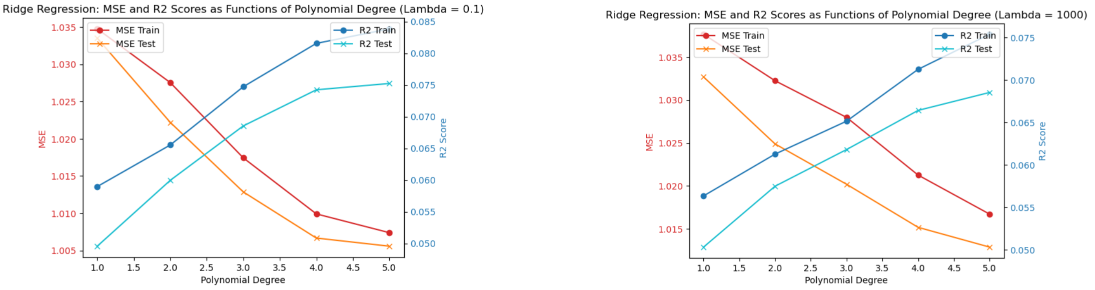


Figure 4.2: Regression method used is Ridge Regression. X-axis depicts different lambda values with a set polynomial degree of five. Two y-axes show the MSE as well as the R2 values.

In Ridge Regression (Figure 4.2), we first evaluated the impact of the regularization parameter, lambda, on model performance. We explored a range of lambda values while fixing the polynomial degree at five. This choice of polynomial degree was based on our previous observations with OLS regression, where higher degrees showed improved model performance. By selecting a degree of five, we aimed for a balance between model complexity and interpretability.

As illustrated in Figure 4.2, lower lambda values result in better performance for both the training and test datasets. Specifically, the MSE for the training data remains around 1.008, while the test data achieves even lower MSE values, approximately 1.006. However, as lambda increases, particularly beyond 10^2, the MSE for both train and test sets rises sharply, and the R^2 scores drop, indicating underfitting due to excessive regularization.



(a) lambda value of 0.1

(b) lambda value of 1000

Figure 4.3: Side-by-side comparison of two different lambda-values. Regression method used is Ridge Regression. X-axis depicts different polynomial degrees. Two y-axes show the Mean Squared Error as well as the R2 values.

In this side-by-side comparison Figure 4.3, we observe that despite the significant difference in lambda values (0.1 and 1000), the overall trends in MSE and R^2 as a function of polynomial degree are quite similar.

For both lambda values, the MSE for both training and test datasets starts at around 1.035 for a polynomial degree of one and decreases as the polynomial complexity increases. With a lambda of 1000, the MSE reaches approximately 1.015 at a polynomial degree of five, while with a lambda of 0.1, a slightly lower MSE of around 1.005 is observed at the same polynomial degree.

The R^2 values, as expected, develops inversely proportional to the MSE. Both lambda values produce similar ranges of R^2 scores, starting at around 0.05 for a polynomial degree of one and increasing as the polynomial degree rises. For the lower lambda (0.1), the R^2 score becomes around 0.08 at a polynomial degree of five, indicating a better fit. With a lambda of 1000, the R^2 score at degree five is similarly low at around 0.07. Despite the large difference in regularization strengths, the overall trends in model performance remain comparable. However, the model with a lower lambda (0.1) outperforms the higher lambda (1000) in both MSE and R^2, particularly at higher polynomial degrees, where the more flexible model fits the data better without the constraints imposed by excessive regularization. This underscores the delicate balance between regularization and polynomial complexity in optimizing model performance.

4.1.3 Lasso Regression

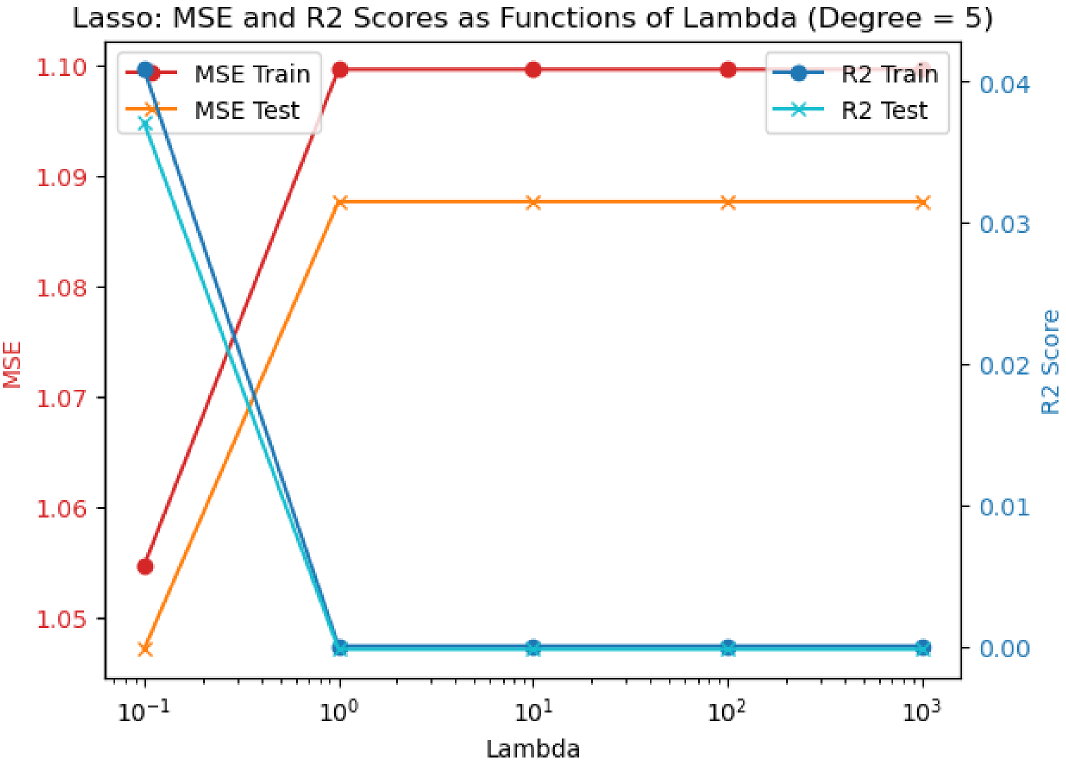


Figure 4.4: Regression method used is Lasso Regression. X-axis depicts different lambda values with a set polynomial degree of five. Two y-axes show the Mean Squared Error as well as the R2 values.

Consistent with how we approached the performance evaluation of Ridge Regression we first looked at different lambda parameters with a fixed polynomial degree of five. We observe that lower lambda values result in the optimal model performance, yielding the smallest MSE and the highest R^2 scores. With a lambda value of 0.1 the test set achieves an MSE as low as 1.05 and an R^2 score of approximately 0.04. However, as lambda increases, the model becomes overly regularized, leading to a significant rise in MSE and a steep drop in R^2. At higher lambda values, the MSE climbs to around 1.10 for the training set, while the R^2 score approaches zero for both the training and test sets. This drastic performance deterioration shows that choosing a low lambda is crucial in this specific implementation of the Lasso regression.

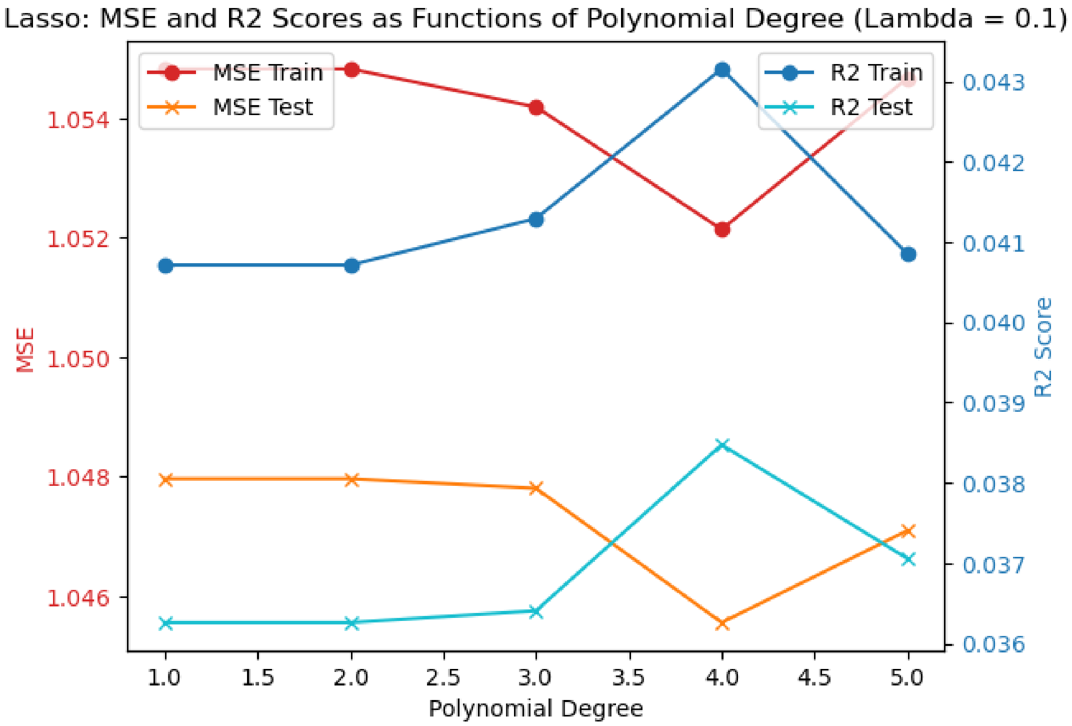


Figure 4.5: Regression method used is Lasso Regression. X-axis depicts different polynomial degrees with a set lambda value of 0.1. Two y-axes show the Mean Squared Error as well as the R2 values.

In Figure 4.5, we examine the performance of Lasso regression as a function of polynomial degree with a fixed lambda value of 0.1.

The MSE remains relatively stable across the range of polynomial degrees, with only minor fluctuations for both the training and test sets. The test MSE hovers around 1.048, while the training MSE remains slightly higher, at approximately 1.054. These small differences between the training and test MSE suggest that the model generalizes reasonably well, although it doesn't significantly improve with higher polynomial degrees.

The R^2 scores exhibit slight variation across polynomial degrees, with the training R^2 peaking at around 0.043 at degree 4 before declining. The test set follows a similar trend, with R^2 scores fluctuating between 0.037 and 0.041.

We observe that the optimal polynomial degree for a low lambda value of 0.1 is degree 4. Interestingly, plotting MSE and R^2 scores for a polynomial degree of four yields similar trends to those seen with a fifth-order polynomial, suggesting minimal performance difference between these degrees at this lambda level.

The fact that both MSE and R^2 remain relatively stable across polynomial degrees suggests that the complexity of the model (as controlled by the polynomial degree) has little influence on performance in this particular Lasso setting with lambda = 0.1. It is important to point out that the scales on our diagrams dynamically fit the range of data points and are different for all our findings. All of these fluctuations we see in this diagram are marginal. We can not really read trends out of these findings since all the values are so close to each other.

4.1.4 Beta values

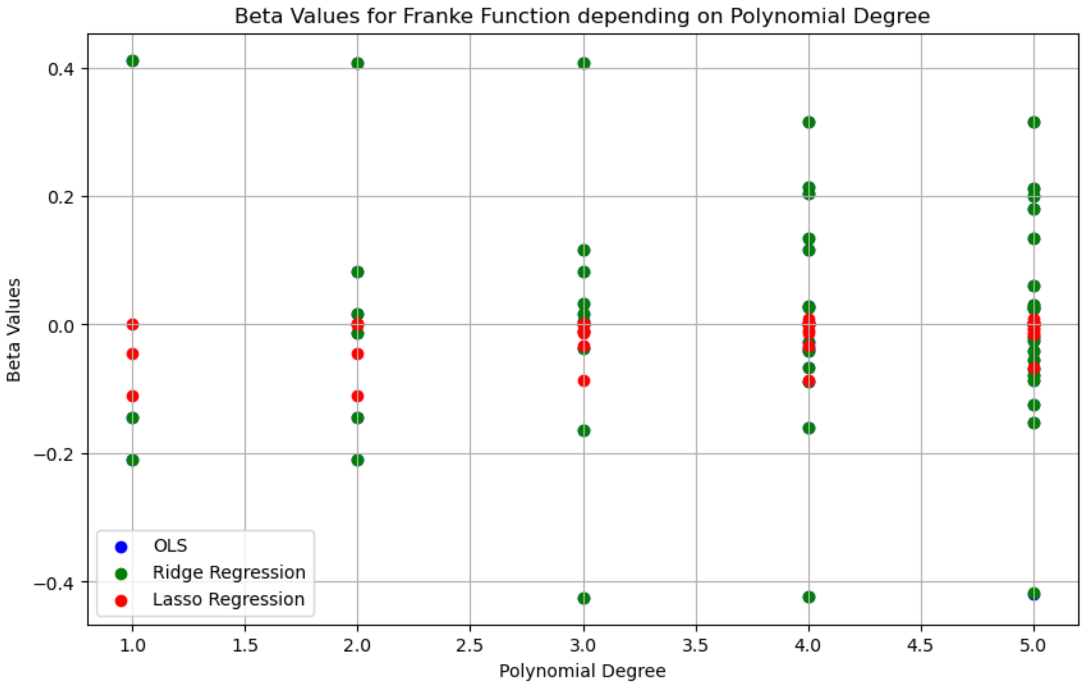


Figure 4.6: The spread of different beta values for increasing polynomial degree. Different regression methods are indicated in different colours: OLS in blue, Ridge regression in green and Lasso regression in red. Betas for a lambda value of 0.1 were plotted for Ridge regression as well as Lasso regression

As we examine the different beta values, we observe that the number of coefficients increases with higher polynomial degrees. Starting with three beta values for a polynomial degree of one, the number expands to 21 beta values at a polynomial degree of five. Across regression methods OLS and Ridge regression, the beta values vary within a range of approximately -0.4 to 0.4. Lasso regression shows a more narrow range of beta values approximately between 0 and -0.1.
The extent of variation in these beta values, differs depending on the regression method. Lasso regression exhibits the smallest variation in beta values, as many coefficients are driven to zero due to its feature selection property. This suggests that Lasso prioritizes simplicity by excluding less important features, especially at higher polynomial degrees.

In contrast, both OLS and Ridge regression demonstrate a larger spread in beta values. OLS as well as Ridge regression show more variability as polynomial degree increases, indicating a higher sensitivity to the added complexity of the model.

4.2 Resampling for Franke's function

4.2.1 Bootstrap

We chose a polynomial degree of five, as it was found to be the best degree from our results from the non-resampled data. We then used bootstrapping with 50 bootstrap samples on the data from Franke's function. We wanted to find both the mean MSE for each sample, as well as the standard deviation for each polynomial degree as that would give us a good insight into how resampling influenced our results. We have plotted the results in a table, and can use this to assess the accuracy of our models.

Degree	**Mean MSE**	**Standard Deviation**
1	1.034819	0.015227
2	1.027157	0.018482
3	1.018910	0.019841
4	1.013072	0.019297
5	1.013253	0.017622

Table 4.1: Mean MSE and Standard Deviation for different Polynomial degrees for the resampled OLS of Franke's function.

Looking at the mean MSE and standard deviation of resampled OLS we can see very similar values across the different polynomial degrees. The mean MSE always lies around 1 while the standard deviation is very low at around 0.012.
As the degree increases, we observe a gradual decrease in the Mean MSE, reaching its lowest value at degree five (1.013). This suggests that higher-degree polynomials provide a slightly better fit for the data. Note that these changes in absolute values are minimal.
The Standard deviation remains very small reflecting a stable model. These findings are consistent with what we saw previously in the results of OLS.

Interestingly when resampling Ridge regression with a lambda value of 0.1 yields the exact same absolute values as seen in Table 4.1. The values differ only slightly when increasing the lambda value.

Degree	Mean MSE	Standard Deviation
1	1.055364	0.016967
2	1.055106	0.021085
3	1.055921	0.020507
4	1.051759	0.020306
5	1.057783	0.020811

Table 4.2: Mean MSE and Standard Deviation for Different Degrees for the resampled Lasso regression of Franke's function

Looking at the absolute values that resampling of Lasso regression results in we see very similar values to the results of OLS and Ridge regression. Slightly higher mean MSEs and a larger standard deviation can be observed. These findings are consistent with our previous results looking at Lasso regression.

Examining the results of the Lasso regression after resampling, we observe values that are very similar to those obtained from both OLS and Ridge regression. However, the mean MSE for Lasso is slightly higher, and the standard deviation is larger, indicating greater variability in model performance. These findings align with our previous analyses of Lasso regression.

4.2.2 Cross-Validation

As with bootstrap, we used lambda= 0.1, as this is what we found to be the best lambda-value when using non-resampled data. We iterated through degrees from one to five, and plotted the MSE and standard deviation for each degree in a table.

Degree	Mean MSE	Standard Deviation
1	1.034856	0.039090
2	1.027354	0.040693
3	1.018236	0.038521
4	1.011781	0.037603
5	1.010592	0.038370

Table 4.3: Mean MSE and Standard Deviation for Different Degrees for the cross-validation resampled OLS and ridge regression of Franke's function

Like with bootstrap, the results for the cross-validation are the same for OLS and ridge when lambda = 0.1. Therefore we only show one plot. The mean MSE for OLS (and thus also ridge) are almost exactly the same when using bootstrap and cross-validation.

This suggests that the model is consistent, as it produces similar results even when using different resampling methods. We also see that the results are very similar to those we get from Figure 4.1, which is the OLS on non-resampled data, and Figure 4.3, which is Ridge regression on non-resampled data. This suggests that when lambda = 0.1, OLS and Ridge regression behaves similarly no matter if the data is resampled or not.

Degree	**Mean MSE**	**Standard Deviation**
1	1.054870	0.037099
2	1.054870	0.037099
3	1.055246	0.039089
4	1.053535	0.039051
5	1.056160	0.039882

Table 4.4: Mean MSE and Standard Deviation for Different Degrees for the cross-validation resampled Lasso regression of Franke's function

For Lasso regression the mean MSE is very similar between cross-validation, bootstrap and non-resampled data. So similar data for resampling and non-resampling can indicate that the models have low variance, which fits with the trend of slight overfitting for all our models.

The standard deviation for all three models, vary between bootstrap and cross-validation, cross validation having a double or almost triple standard deviation compared to bootstrap. This suggests that cross-validation might be better able to capture the variance in the models, than bootstrap.

4.3 Topographical Data

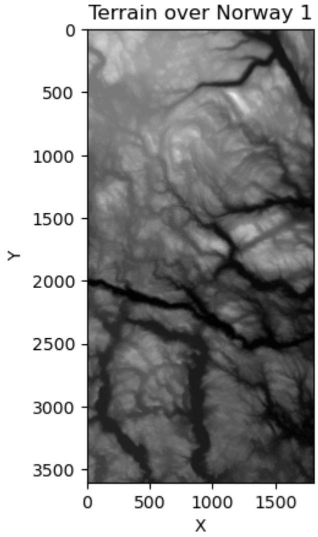


Figure 4.7: A visual representation over our terrain data

For our topographical data (Figure 4.9) we again looked at the performance measures MSE and R^2 to assess the performance of our models.

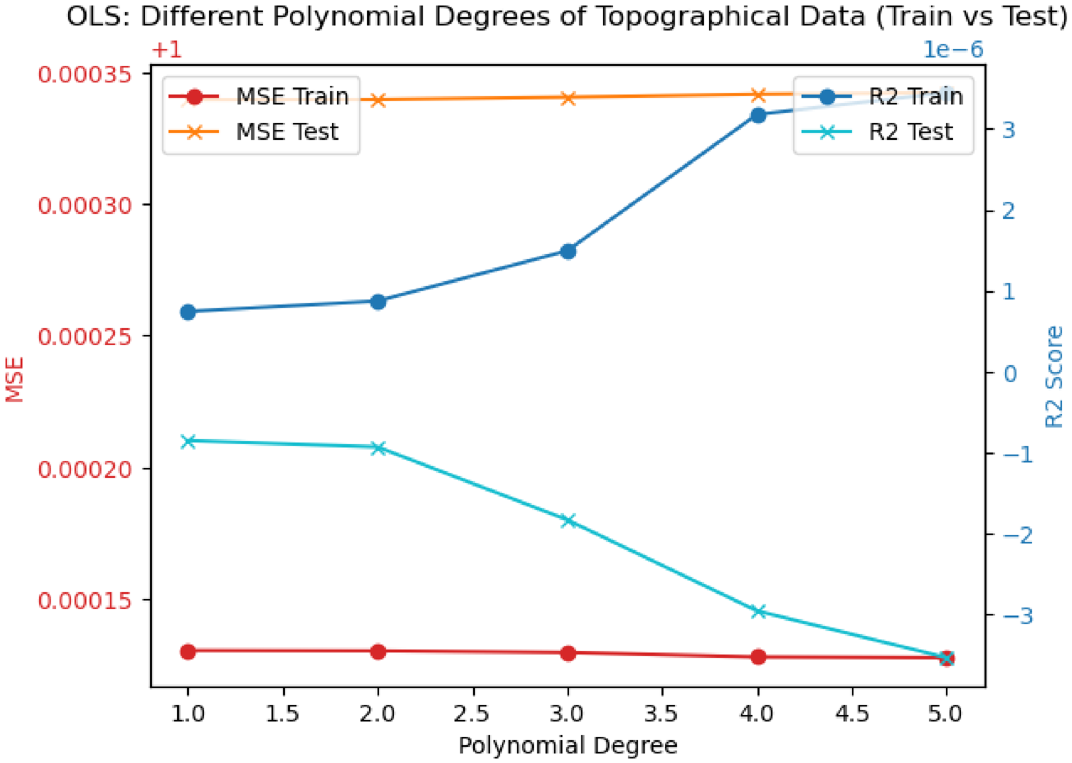
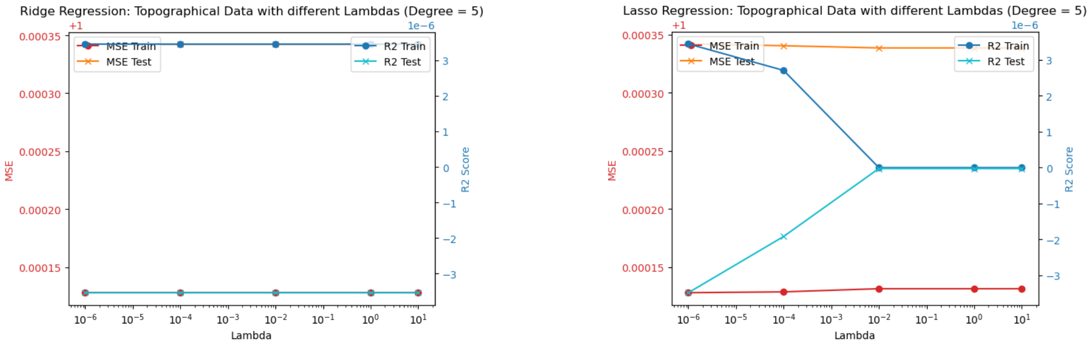


Figure 4.8: Regression method used is Ordinary Least Squares. Here we have used Topographical Data. X-axis depicts the different polynomial degrees. Two y-axes show the Mean Squared Error as well as the R2 values.

From our figure, we note that the MSE is essentially zero for both the training and the test data. We also note that the MSE is constant for different polynomial degrees. Secondly, the R^2 of the training data increases with higher polynomial degree, but the opposite happens for out test data. The R^2 also becomes negative for the test data when the polynomial degree is higher than 1 (a straight line). This suggests that a straight line fits the data points better than a polynomial of a higher degree. We also note that as the range of the R^2 should be in between zero and one.



(a) Ridge regression

(b) Lasso regression

Figure 4.9: Side-by-side comparison of Ridge and Lasso regression with different lambda-values when polynomial degree is 5. X-axis depicts different polynomial degrees. Two y-axes show the Mean Squared Error as well as the R2 values.

This figure shows us the different values of λ for Ridge and lasso regression give no change in the MSE score for our fit of the topographical data when the polynomial degree is five. It shows however that the R^2 is constant for both training and test data, but at different values for Ridge regression. The R^2 for both training and test data converges to zero for Lasso regression with increasing λ.

4.4 Resampling for topographical data

We used the same resampling methods for our topographical data as we used for the franke function. Because it took several minutes to compute the cross-validation, we had to use a low k, with k=5. This was to minimize the amount of computations needed, and reduce the runtime of the program. We assume the slowness came as a result of the large size of the data. For the same reason, we only used 10 bootstrap datasets, even though we had used 100 with the Franke function without an issue.

Chapter 5

Discussion

Conclusions, discussions and critical comments: on what was learned about the method used and on the results obtained. Possible directions and future improvements? Total number of possible points: 10

5.1 Franke's function

We explored the performance of various regression techniques—OLS, Ridge, and Lasso regression—across different polynomial degrees and regularization strengths.

Polynomial Degrees. We focused on polynomial degrees from 1 to 5 because this range provided a balanced exploration of model complexity without overly complicating the interpretation of results or computation. While higher polynomial degrees can provide more flexibility, they also introduce the risk of overfitting, so we aimed to explore the performance trade-offs within a manageable degree range.
It is important to note that most of the differences in performance measures that we found were marginal. We only can see minor trends which we tried to perfect our models for. We want to point out to the reader that the differences we base our optimization on are very small.
Regardless of the regression method the results showed that models performed better at higher polynomial degrees, as indicated by lower MSE and higher R^2 values. This suggests that when working with three-dimensional data, more complex models (in terms of polynomial degree) are generally favored.

Lambda values. Since lambda is a parameter that can not be optimized in an computational approach and is detected by trial and error we chose to examine a large range of lambda values between 0.1 and 1000 for Ridge and Lasso regression. This wide range allowed us to investigate both low regularization (lambda = 0.1) where the model behaves similarly to OLS, and very high regularization (lambda = 1000), which is expected to induce substantial shrinkage of coefficients. One interesting observation was that once lambda reached larger values, the differences in model performance between various lambda values became less pronounced. This could be because, at very high lambda values, further penalization beyond a certain threshold has diminishing effects on the model's performance, as the coefficients are already heavily constrained. For both Ridge and Lasso, we observed that lower lambda values led to better model performance. When lambda was increased too much, both Ridge and Lasso exhibited over-regularization, leading to underfitting, as reflected by rising MSE and falling R^2 values. This underfitting suggests that excessive penalization of model coefficients through high lambda values restricts the model's flexibility, preventing it from capturing the data's underlying patterns.

Beta values. Analyzing the distribution of beta coefficients across regression methods provided additional insights into how each method handles model complexity. We found that OLS and Ridge had the most variation of beta coefficient values. Lasso regression, on the other hand, showed its feature selection ability by driving many coefficients to zero.

Resampling. For the resampling of OLS and Ridge regression our results with a mean MSE close to 1 and a very small standard deviation of around 0.012 underlined the good performance of these two regresssion models. Lasso showed a slightly worse performance with a mean MSE and a standard deviation which were slightly higher than the values of OLS and Ridge regression.

Differences between OLS and Ridge regression. We did see the same absolute values in the mean of the MSEs and Standard deviation. The same happened in It is quite unusual for Ridge regression with a regularization term (lambda value of 0.1) to yield exactly the same results as Ordinary Least Squares (OLS), especially when using a resampling method like bootstrapping. Ridge regression penalizes larger coefficients, which should generally affect the model's predictions and the resulting error metrics. Even with a small regularization term (like 0.1), it typically introduces some differences in the results.

Small Regularization Effect: A lambda value of 0.1 is relatively small, so the regularization might not have been strong enough to significantly alter the coefficients from those of OLS. However, "exactly" the same results would still be uncommon due to the inherent penalty introduced by Ridge regression.

Numerical Precision/Errors: If the computation precision is limited, the differences between Ridge and OLS could be extremely small, especially if the dataset is simple or the regularization is weak. This might result in values that appear identical but differ at higher levels of precision.

Conclusion. In terms of overall performance, the difference between OLS and Ridge regression with a low lambda was marginal. OLS as well as Ridge Regression with a lambda of 0.1 scored an MSE of around 0.05 for both training and test set. R^2 values of around 0.075 for the testing set and 0.085 for the training set were scored by both OLS and Ridge regression with a lambda of 0.1. However lambda increased in Ridge regression, the model's performance deteriorated due to over-regularization.
Lasso regression showed worse performance also with the best lambda value that was found of 0.1 with an MSE of 1.046 for the Training set and 1.052 for the test set at the best found polynomial degree of four. R^2 values with the same parameters lay at around 0.039 for the test set and around 0.043 for the training set.
Based on these performance measure results we would suggest deciding against lasso regression. Given the minimal performance difference between OLS and Ridge at lower lambda values, we suggest opting for the simpler OLS model, in line with the principle of choosing the simplest solution. This aligns with the programming and informatics paradigm of minimizing complexity whenever possible. Additionally, increasing the polynomial degree improved the model's performance across all methods, reinforcing the importance of model complexity in achieving better results for three-dimensional data with the Franke-Function.

5.2 Topographical Data

After testing our algorithms for the different regression models on the random normally distributed data, we then used the different methods on real topographical data.

For OLS, we see that a straight line fits the data points better than a polynomial of a higher degree (as R^2 becomes negative). This is a very surprising result as our topographical data is not completely flat. This could be from the fact that we imported too much data from our file, instead of e.g. every 100th data point. This makes it extremely difficult to only look at a 5th degree polynomial for such varied landscape. A straight line will fit better then as the average of our terrain will then be flat.

Due to this, we tried downsampling our data, and extracted only every 100th data point. We got a slightly different result, but it still did not give a result we would expect (see Figure A.1). The same trend for our performance measures remained.

We then assume our plots are less informative for the topographical data than for Franke's function as the data points differ a lot. The data points are not normally distributed, so when we use the standardscaler of Sckikit-Learn (Pedregosa et al. 2011), the estimators can behave badly. We also tried running the code without scaling, however the outcome stayed the same. We still decided to scale as it ensures we do not get points that are outside our domain.

We also tried downsampling further for every 150th datapoint, and as we looked at all regression models, we saw that there was a general trend for overfitting of the training data compared to the test data. The training data seems to improve with increasing polynomial degree, but the test data does the opposite. Again, showing that a straight line fits our data better than our model.

Furthermore, the results we can infer from Figure 4.10 is that for any λ, the polynomial fit of Ridge and Lasso regression will give the same MSE. This means that the regularization parameter λ has no effect on the fit. In theory, this should not happen for our model. We also saw that the R^2 converges to zero for the training and test data for Lasso regression. This is still very bad, as it says there is no linear relationship between our model and the observed datapoints. Thos suggests that our model is not only bad, but completely wrong.

We tested the code with different values of λ, but it showed us no difference. We would try to change the plots to be improved, but time restrictions did not allow for this. However different plots would not change the values of our results but only their visualization. We also would have tested to see if there could be better polynomial degrees to test for other than a degree of 5, as our data seems to fit better for a degree of 1.

5.3 Conclusion

This study helped us understand that a surprisingly large part of implementing machine learning methods depend on the fine-tuning of parameters. With this study we saw that approaching projects with machine learning methods is about finding the best model that best fits a projects requirements in the first place. We tried to approach this fine-tuning process as systematic as possible. By systematically exploring the effect of parameters such as polynomial degree and regularization strength, we gained a clearer understanding of how to approach the optimization process. Our findings underscore the importance of balancing model complexity and regularization to avoid underfitting or overfitting, and demonstrate that even simple models like OLS can offer competitive performance. However, we also saw that transferring the implemented regression models and their fine-tuned parameters to topographical data is a non-trivial task. Our findings suggest that with topographical data we deal with highly complex data that can not be depicted with the three regression models we have looked at easily.

Bibliography

ChatGPT (2024). "K-fold Cross Validation MSE". In: URL: https://chatgpt.com/share/6703d1a2-5740-8002-bbbc-d1a73b747c1f.

Goel, M. (2020). "The Bias-Variance Trade-Off : A Mathematical View". In: *Medium*. URL: https://medium.com/snu-ai/the-bias-variance-trade-off-a-mathematical-view-14ff9dfe5a3c.

Hastie, T., R. Tibshirani, and J. Friedman (2009). *The Elements of Statistical Learning*. Springer New York. ISBN: 9780387848587. DOI: 10.1007/978-0-387-84858-7. URL: http://dx.doi.org/10.1007/978-0-387-84858-7.

Hjorth-Jensen, M. (2024). *FYS-STK4155 – Applied Data Analysis and Machine Learning*. https://github.com/CompPhysics/MachineLearning/tree/master/doc. Lecture Notes, University of Oslo.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Raschka, S. et al. (2022). *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Packt Publishing. ISBN: 1801819319, 9781801819312.

Appendix A

A.1 GitHub Repository

We have all our source code and test runs in our repository on GitHub at this link:

https://github.com/sphia-g/ML24UiO/tree/main/Project1

A.2 Mathematical formulas

A.2.1 Bias-Variance

$$\begin{aligned}
\mathbb{E}\left[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2\right] &= \mathbb{E}[(f(x)+\epsilon-\tilde{\boldsymbol{y}})^2] \\
&= \mathbb{E}[(f(x)-\tilde{\boldsymbol{y}})^2-2\epsilon(f(x)-\tilde{\boldsymbol{y}})+\epsilon^2] \\
&= \mathbb{E}[(f(x)-\tilde{\boldsymbol{y}})^2]-\mathbb{E}[2\epsilon(f(x)-\tilde{\boldsymbol{y}})]+\mathbb{E}[\epsilon^2] \\
&= \mathbb{E}[(f(x)-\tilde{\boldsymbol{y}})^2]-0+\sigma^2 \\
&= \mathbb{E}[((f(x)-\mathbb{E}[\tilde{\boldsymbol{y}})])-(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}]))^2]+\sigma^2 \\
&= \mathbb{E}[((f(x)-\mathbb{E}[\tilde{\boldsymbol{y}})])^2]-2\mathbb{E}[(f(x)-\mathbb{E}[\tilde{\boldsymbol{y}}])(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}])]+\mathbb{E}[(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}])^2]+\sigma^2 \\
&= \mathrm{Bias}[\tilde{y}]+\mathrm{var}[\tilde{y}]+\sigma^2
\end{aligned}$$

A.2.2 Plots not included in report

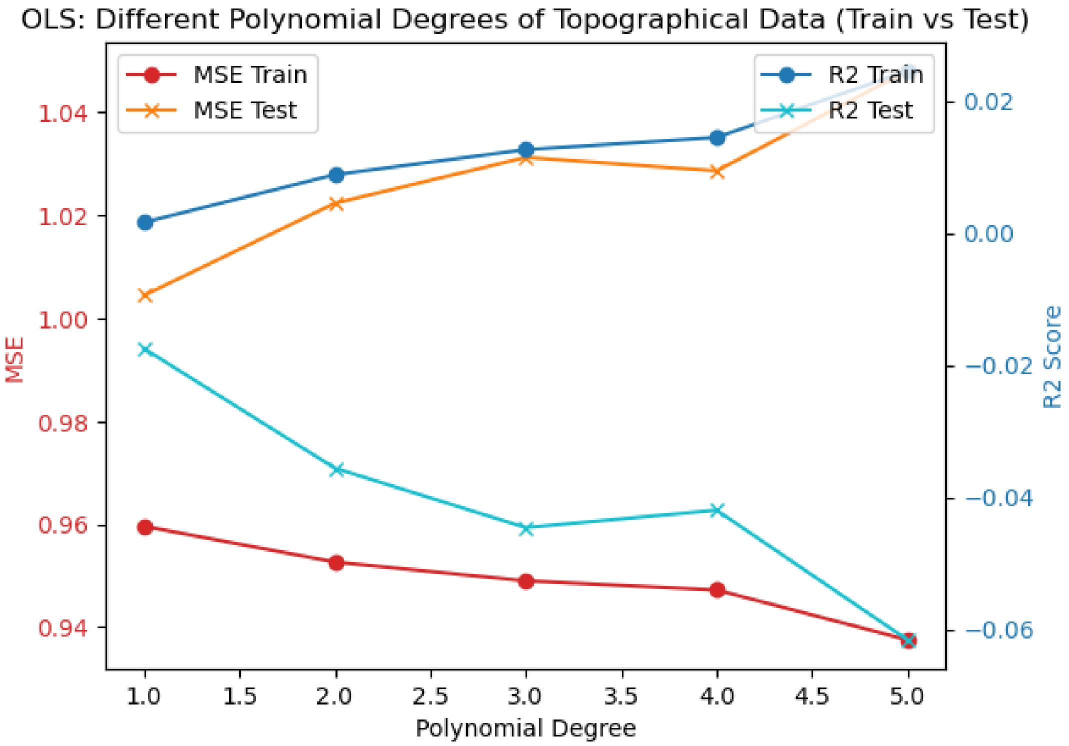


Figure A.1: Regression method used is Ordinary Least Squares. Here we have used downsampled Topographical Data. X-axis depicts the different polynomial degrees. Two y-axis show the Mean Squared Error as well as the R2 values.