Linear Regression and Basic Resampling Techniques for Modeling 2D Datasets

Emma Storberg & Frida Lien

October 7, 2024

Abstract

Understanding the intricacies of machine learning is an essential part of navigating our data-driven world. In this project, we explored various methods for solving linear regression on the 2D Franke function [1] and a dataset from the United States Geological Survey [2]. Our analysis focused on Ordinary Least Squares (OLS), ridge regression and LASSO regression, evaluated through statistical metrics such as the mean squared error and R^2 score. Additionally, we used resampling methods like bootstrap resampling and k-fold cross-validation to achieve more reliable results. We investigated the bias-variance trade-off and challenged our models with the goal of first inducing overfitting, and thereafter combating it using our understanding of the theory and techniques we have learned. Initially, our OLS model performed well, but the introduction of regularization terms revealed an error in our procedure for determining optimal hyperparameters, resulting in unreasonable values despite passing tests against the methods of `scikit-learn`. We emphasize the importance of following sound statistical intuition and employing supplementary methods such as visual inspection when interpreting results, rather than solely relying on numerical metrics.

Contents

1 Introduction

In today's highly interconnected and digitally-driven society, data has become the cornerstone of decision-making processes across many sectors. We are increasingly reliant on the vast amounts of data generated daily to advance research, industry, innovation and optimization of operations within various fields. Fundamental to this data-centric world are machine learning and data analysis techniques, which enable the extraction of meaningful patterns and predictive capabilities from raw data. This report will cover some of the foundational methods in machine learning and data analysis, with a particular focus on linear regression and resampling.

There are many scenarios in which we may be interested in knowing some value y based on some other factor x. Data-driven techniques, such as machine learning, leverage the core concept that data from a given system can be utilized to describe the relationship between inputs and outputs through a computational model. We have limited ourselves to polynomial models, meaning we want to find the polynomial of best fit for our data. The assumption of a polynomial model allows us to treat the coefficients of the polynomial as unknowns that we want to identify, giving us a linear relationship with respect to these coefficients. Fitting a linear equation to observed data is a fundamental technique in data analysis known as linear regression, and if we find a model of "good" fit, we hope to use it to make meaningful predictions about probable values of the output y, even for previously unseen input data x from the same system.

There are multiple methods for making these predictions, as well as a variety of metrics to assess their accuracy. One approach is to quantify how well the model fits the data using a function of the outputs we know and the outputs the model predicts. In this way, with some choice of tuned parameters, we can determine a "cost" of using certain coefficients in our model. This cost is what we are trying to minimize, but the exact way the cost is calculated can vary, and is what distinguishes the various regression methods. However, choice of method is just one factor among several that can influence a model's accuracy. Other important considerations include the introduction of additional parameters, the selection of statistical metrics, in addition to the application of pre-processing and post-processing techniques. These decisions depend on the characteristics of the dataset and the objectives of the analysis.

This report considers three regression methods in particular: Ordinary Least Squares (OLS), ridge regression and LASSO regression. We will compare and contrast these methods when used to predict values from a dataset on a noisy Franke function [1], as well as data from the U.S. Geological Survey [2]. First, we will show the results and some relevant statistical metrics for accurately assessing the methods, and use two popular resampling techniques to perform a bias-variance analysis of the model. Later on, we will examine the advantages and disadvantages of each method, with the aim of determining the best method for making predictions about our dataset.

2 Theory

We assume that our dataset provides inputs $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^n$, and corresponding noisy observations $\boldsymbol{y} \in \mathbb{R}^n$, such that

$$\boldsymbol{y} = f(\boldsymbol{x}_1, \boldsymbol{x}_2) + \boldsymbol{\varepsilon},$$

The noise in the observations is described by $\boldsymbol{\varepsilon}$, which is assumed to be a collection of independent and identically distributed (i.i.d.) random variables following a Gaussian distribution $\boldsymbol{\varepsilon} \sim N(0, \sigma^2)$. Observations may be noisy due to for instance technical errors when performing the measurements.

We wish to approximate $f(\boldsymbol{x}_1, \boldsymbol{x}_2)$ to a model $\tilde{\boldsymbol{y}}$. Our model must be linear with respect to its regression coefficients, but may belong to various function classes, such as exponential or trigonometric functions. In this report, we will consider the case where our model is a polynomial. Our goal is then to find an appropriate polynomial degree m and optimal coefficients $\hat{\boldsymbol{\beta}}$, such that $\tilde{\boldsymbol{y}}$ is sufficiently close to $f(\boldsymbol{x}_1, \boldsymbol{x}_2)$.

The expectation value[1] of $\boldsymbol{y}$ for some element y_i is given by

$$\mathbb{E}[y_i] = \sum_j x_{ij}\beta_j = \mathbf{X}_{i,*}\boldsymbol{\beta}.$$

This expression tells us that we can expect our output to reflect the true relationship between the input and output. Furthermore, the variance[2] of a given element of the output is given by

$$\mathrm{Var}(y_i) = \sigma^2,$$

where σ^2 is the variance of the normally-distributed error term $\boldsymbol{\varepsilon}$. We can interpret the variance as a measure of the spread of the dataset. Later on, we will create models and assess them using a variety of statistical methods. One of these is the R^2 *score*, which is a measure of the proportion of the variance of our output data that can be explained by the input data [3].

We can organize our input data in a *design matrix*. The design matrix has dimensionality $n \times p$, where n corresponds to the number of observations, and p corresponds to the number of *features*. Generally, a feature represents a characteristic in the input data that we want our model to consider. For instance, features such as height, age, and gender could be used to predict a person's weight. In our case, we will construct the design matrix $\mathbf{X}$ as follows:

$$\begin{bmatrix} \mathbf{1} & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_1^2 & \boldsymbol{x}_1\boldsymbol{x}_2 & \boldsymbol{x}_2^2 & \dots & \boldsymbol{x}_1\boldsymbol{x}_2^{m-1} & \boldsymbol{x}_2^m \end{bmatrix}$$

Throughout this report, we will denote the coefficients in our model as $\boldsymbol{\beta} \in \mathbb{R}^p$. This way, our model is of the form

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

2.1 Cost Functions

The *cost function* $C(\boldsymbol{\beta})$ quantifies the accuracy of our model by measuring the error it produces relative to the given observations. The optimal coefficients $\hat{\boldsymbol{\beta}}$ are found by minimizing the cost function:

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^p}{\arg\min}\, C(\boldsymbol{\beta}).$$

Once we have an expression for the cost function, we can find $\hat{\boldsymbol{\beta}}$ by solving the equation

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0,$$

where we must also ensure we are finding a minimum and not a maximum before proceeding. The cost function can be constructed in various ways, depending on the characteristics of the dataset and the goals of the analysis. This brings us to the three methods we will discuss and compare in this report.

2.1.1 Ordinary Least Squares

To understand the regression methods we will be using, we must first introduce an important concept: the *Mean Squared Error* (MSE). The MSE is a widely-used statistical metric to measure the performance of a model. It is the average of the squared distance from an observation to the corresponding predicted value.

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

Hence, the large deviations in the predictions have a greater influence on the MSE compared to the small deviations. In vector notation, we can express it as

$$\mathrm{MSE}(\tilde{\boldsymbol{y}}) = \frac{1}{n}(\boldsymbol{y} - \tilde{\boldsymbol{y}})^T(\boldsymbol{y} - \tilde{\boldsymbol{y}}),$$

and in statistical notation, it can be written as the expectation value of the squared error:

$$\mathrm{MSE}(\tilde{\boldsymbol{y}}) = \mathbb{E}[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2].$$

We will use all three forms throughout this report wherever they are most appropriate.

The *Ordinary Least Squares* (OLS) method employs the MSE as its cost function. That is,

$$C(\hat{\boldsymbol{\beta}}) = \frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}).$$

[1] See appendix A.1 for full derivation.
[2] See appendix A.2 for full derivation.

Defining the cost function in this way also ensures that taking its derivative will find a minimum, seeing as it is a quadratic function and thus convex. In doing so, we find that the derivative of the cost function with respect to the coefficients $\boldsymbol{\beta}$ is given by

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\frac{2}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T \mathbf{X}.$$

Finally, this leads us to the expression for optimal coefficients in the OLS method[3]:

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y}.$$

We can also show that the expectation value of the optimal coefficients found by using the OLS method $\hat{\boldsymbol{\beta}}_{\text{OLS}}$ is equal to the true optimal coefficients[4], denoted by $\boldsymbol{\beta}$:

$$\mathbb{E}[\hat{\boldsymbol{\beta}}_{\text{OLS}}] = \boldsymbol{\beta}$$

This means that OLS is a so-called *unbiased* method, since the expectation value of the coefficients we calculate is exactly equal to the theoretical best possible value of what we are approximating. Additionally, the variance of the optimal coefficients[5] is given by

$$\text{Var}[\hat{\boldsymbol{\beta}}] = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}.$$

As we have seen, the OLS method represents the most intuitive cost function possible by determining the coefficients $\boldsymbol{\beta}$ simply by minimizing the distance between the output the model predicts and the output data in our data set. In addition, the *Gauss-Markov theorem* states that for all unbiased models, OLS has the lowest variance[6] [4]. Even with these qualities, there may be instances in which this basic approach is not well-suited. In particular, we may have a design matrix $\mathbf{X}$ that results in the matrix $\mathbf{X}^T\mathbf{X}$ being singular or nearly-singular, especially for high dimensions [5]. Furthermore, data with large outliers are typically not well-analyzed by OLS, because out-lying observations will skew the fit of the regression line at the expense of the rest of the data to minimize the MSE [6]. Hence, if the dataset is full of outliers and inconsistency, it will destabilize the predictions. In cases like these, OLS may not perform well, and some alterations to the regression method may be necessary to find a better fit.

[3]See appendix A.3 for full derivation.

[4]See appendix A.4 for full derivation.

[5]See appendix A.5 for full derivation.

[6]under the assumption of uncorrelated errors with equal variance and an expectation of zero

2.1.2 Ridge Regression

Ridge regression is another type of regression that may be useful. Its cost function is given by:

$$C(\boldsymbol{\beta}) = \underbrace{\frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})}_{\text{MSE}} + \underbrace{\frac{1}{n}\lambda\boldsymbol{\beta}^T\boldsymbol{\beta}}_{\text{Regularization term}}$$

As we can see, it differs from the MSE by the addition of a regularization term scaled by a parameter λ. By the same strategy of finding where the derivative of the cost function is equal to zero, we find the expression for the optimal parameters[7]:

$$\hat{\boldsymbol{\beta}}_{\text{Ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbb{I}_p)^{-1}\mathbf{X}^T\boldsymbol{y}$$

As opposed to OLS, we see from the following expression[8] that ridge regression is not equal to the value we are trying to approximate, and therefore not unbiased:

$$\mathbb{E}[\hat{\boldsymbol{\beta}}_{\text{Ridge}}] = (\mathbf{X}^T\mathbf{X} + \lambda\mathbb{I}_p)^{-1}(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}$$

The introduction of the parameter λ impacts how the parameters $\hat{\boldsymbol{\beta}}_{\text{Ridge}}$ are determined. To understand exactly how, it is useful to consider the singular value decomposition (SVD) [7] of the design matrix $\mathbf{X}$. SVD is a common mathematical technique wherein a matrix is decomposed into three other matrices. Specifically, it expresses the original matrix as the product of an orthogonal matrix, a diagonal matrix of singular values, and the transpose of another orthogonal matrix. This factorization of a matrix has many uses, but in our case it will allow us to see how the hyperparameter λ contributes to scaling our output as a projection of orthogonal vectors.

[7]See appendix A.6 for full derivation.

[8]See appendix A.7 for full derivation.

In the expression below [9], we see that the predicted output values $\tilde{\boldsymbol{y}}_{\text{Ridge}}$ are given by:

$$\begin{aligned}\tilde{\boldsymbol{y}}_{\text{Ridge}} &= \mathbf{X}\boldsymbol{\beta}_{\text{Ridge}} \\ &= \mathbf{U\Sigma V}^T(\mathbf{V\Sigma V}^T + \lambda\mathbf{I})^{-1}(\mathbf{U\Sigma V}^T)^T\boldsymbol{y} \\ &= \sum_{j=0}^{p-1}\mathbf{u}_j\mathbf{u}_j^T\frac{\sigma_j^2}{\sigma_j^2+\lambda}\boldsymbol{y}\end{aligned}$$

where σ_i are the singular values from the SVD. Using our knowledge of linear algebra, we can use this to understand how ridge regression operates. First, we use the column vectors of $\mathbf{U}$ to project our desired output $\boldsymbol{y}$ into the column space of the design matrix $\mathbf{X}$, which represents the output space of our model. These coordinates are then shrunk by a factor of $\frac{\sigma_i^2}{\sigma_i^2+\lambda}$. Intuitively, the eigenvalues of a matrix represent how much the matrix scales a vector in the direction of its corresponding eigenvector. When dealing with small eigenvalues, their contributions to the model become less significant. By shrinking them by a scalar $\lambda > 0$, ridge regression effectively reduces the number of degrees of freedom in the model. Since we are working with a design matrix, this corresponds to allowing scaling in the direction of the eigenvectors with the largest eigenvalues (i.e. the most impactful features) to dominate [5].

Another way to describe the impact of ridge regression on optimal coefficients is through its cost function directly. It has two terms, the first of which is simply the MSE, which is minimized by the smallest possible distance between predicted and observed outputs. The regularization term $\frac{1}{n}\lambda\boldsymbol{\beta}^T\boldsymbol{\beta}$ is minimized for $\lambda > 0$ by letting $\boldsymbol{\beta} \to 0$. This means that in comparison to OLS, ridge regression will aim to make coefficients smaller, the extent of which is determined by the scalar λ. In addition, it can never set any coefficients to 0, and will prioritize minimizing larger coefficients over coefficients that are already small, since the final term is effectively squaring their values. This minimization of coefficients can improve the model in important ways when compared to the straightforward approach of OLS, as we will explore in later on.

[9]See appendix A.8 for full derivation.

2.1.3 LASSO Regression

The third regression method we will consider in this report is LASSO regression, which stands for Least Absolute Shrinkage and Selection Operator [5]. Its cost function is given by

$$C(\boldsymbol{\beta}) = \underbrace{\frac{1}{n}\sum_{i=0}^{n}(y_i - \tilde{y}_i)^2}_{\text{MSE}} + \underbrace{\frac{1}{n}\lambda\sum_{i=0}^{p-1}|\beta_i|}_{\text{Regularization term}}$$

Similar to the cost function of ridge regression, we see there is a compromise between choosing coefficients such that the MSE is minimized, and making the regularization term as small as possible. An important distinction lies in the regularization term however, with ridge regression making use of the *2-norm*, while LASSO uses the *1-norm* (the average absolute value of the coefficients). Once again, the effect of the regularization term $\frac{1}{n}\lambda\sum_{i=0}^{p-1}|\beta_i|$ is determined by a scalar $\lambda > 0$.

The use of different norms is responsible for some key differences between models created using ridge and LASSO regression. For instance, we saw that ridge regression prioritizes limiting larger coefficients, because this has a larger impact on reducing the 2-norm of the coefficients overall. In contrast, LASSO regression limits all coefficients equally, and will allow coefficients to be eliminated entirely [8].

2.2 Scaling

We have seen that ridge and LASSO regression both utilize cost functions that contain the MSE, but importantly, they also penalize large coefficients. A problem with these cost functions quickly becomes apparent when we try to model systems where the data points are of large magnitude. In essence, since the norm of the coefficients plays a role in the cost functions, the cost will reflect the range of the dataset we are modeling, and may be skewed for ridge and LASSO regression.

As a simple example, consider a one-dimensional system with input data $\boldsymbol{x}$, and the two sets of output data we want to model, $\boldsymbol{y}_1 = f_1(\boldsymbol{x}) = 5\boldsymbol{x}$ and

$\boldsymbol{y}_2 = f_2(\boldsymbol{x}) = 5000\boldsymbol{x}$. It is clear that these two possible outputs are linearly dependent, and we might go as far as to say that they represent the "same" system $f_1(\boldsymbol{x})$, up to multiplication by a factor of 1000. For simplicity, let's assume there is no noise here, such that when we use a design matrix with one feature on this system (for a polynomial model of degree 1), we determine the optimal coefficients to be the true coefficients, $\beta_1 = 5$ and $\beta_2 = 5000$ for $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ respectively. As this is the exact solution, it will result in an MSE equal to 0, and the MSE will therefore not contribute anything to any of the cost functions. Thus, $C(\hat{\boldsymbol{\beta}}_{\text{OLS}}) = 0$ for both systems, while ridge and LASSO still have their regularization terms that depend on the chosen coefficients. Their costs will be massively impacted by whether or not we have multiplied the system by 1000, even though both of these noise-less systems have what we understand to be equivalent relationships between their input and output values.

The issue we illustrate in this example can be mitigated by introducing *scaling*. This technique helps us to negate the effects that the range of the data values can have on the cost function, such that it is not artificially high or low solely because of their magnitudes. This way, we can ensure that when we make decisions or evaluate a model based on its cost, the cost reflects the fit of the chosen coefficients alone.

2.3 Bias-Variance Trade-off

We are now arriving at a core concept, namely the *bias-variance trade-off*. Previously, we have seen expressions for the bias of OLS and ridge regression, meaning how the expected value of the coefficients skews away from the "true" parameters $\boldsymbol{\beta}$ for some input data. We found that OLS is unbiased, but ridge is not. To understand the significance of this, let's consider the following expression[10]:

$$\mathbb{E}[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2] = \text{Bias}[\tilde{y}] + \text{Var}[\tilde{y}] + \sigma^2$$

[10]See appendix A.9 for full derivation.

On the left hand side, we have the MSE of the model, and it is given as the sum of its bias and its variance, in addition to the variance σ^2 of the normally-distributed noise $\boldsymbol{\varepsilon}$. The latter is considered an irreducible error, so we can disregard it for the moment and focus on the first two terms. Unbiased methods, such as OLS, must necessarily have variance $\text{Var}[\tilde{y}]$ that accounts for virtually all of the MSE. Compare this with ridge regression for instance, which is a biased method, and in addition to $\text{Var}[\tilde{y}]$, the error must also be made up of the model bias, $\text{Bias}[\tilde{y}]$. This tells us that for equal MSEs of an OLS model and a ridge regression model, their make-up is different: OLS has a low bias and higher variance, while ridge regression results in higher bias, but lower variance. Among others, this phenomenon is what we aim to reproduce experimentally in this report.

As we have seen, the cost functions of ridge and LASSO regression tend to shrink the coefficients of the model or eliminate them altogether, unlike OLS. Additionally, based on the bias-variance trade-off, we expect that ridge and LASSO regression will display lower variances. But we still have not answered an important question: *Why does this occur?* On surface level, there does not seem to be an intuitive reason for why decreasing the magnitudes of the chosen coefficients necessarily makes a model perform better; on the contrary, the regularization term competes with the MSE in the cost functions, meaning the predictions deviate more from our observations if we optimize something other than the MSE. How, then, does the introduction of a regularization term contribute to making models that are still measurably "good fits" for our data?

The answer, in short, is *overfitting*. When the model has high complexity, for instance by having a high polynomial degree and many terms of high magnitude, the training of the model may begin to reflect the noise in the dataset, such that the model "learns" specifics of the data and imitates other behaviors than what we wish to predict. When we create our models based on our datasets, we use some of the data to calibrate the model and find coefficients, while the rest is set aside and not used in calibration, so that we can test and quantify how well the model performs on unseen data. Thus, overfitting is typically characterized by a relatively low MSE of the training data, as the model is trained to fit these data very well, but a significantly higher MSE of the testing data, since these will usually not contain the exact same inconsistencies that the model is overfitted to from its training.

Reducing the ability of the model to learn complex information by limiting the amount and magnitude of certain terms is a way one might combat overfitting, which is exactly what the regularization terms do. This has consequences for the bias-variance trade-off, because if the models can reduce overfitting, they can also improve the fit of the model through its MSE when encountering new data [5].

3 Method

This section will elaborate on the datasets used, practical considerations when applying key expressions from the theory, details about the code, as well as justifications for certain decisions we have made.

We aimed to use expressions from the theory to as large an extent as possible in this project, with a few exceptions where we used external packages. For instance, while we have previously found relatively simple expressions for the coefficients of OLS and Ridge regression, we have not found an equivalent analytical expression[11] for the optimal coefficients $\hat{\boldsymbol{\beta}}_{\text{LASSO}}$, as Hjorth-Jensen [5] also notes. Instead, we have opted for the built-in methods from `scikit-learn` [9], and we will be using these in our experiments.

It is also worth noting that in this experiment, we are exploring the limits of linear regression for predicting datasets of this kind. For this reason, we will be using relatively small datasets, and oftentimes also constructing models of unnecessarily high complexity to better observe the phenomena in question.

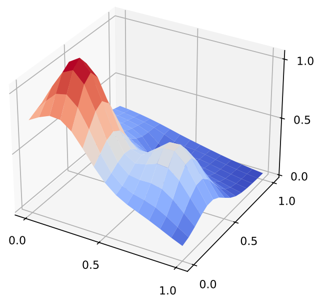
[11]There is a formula for $\hat{\boldsymbol{\beta}}_{\text{LASSO}}$ that applies in the specific case where the design matrix is equal to the identity matrix. This is clearly not the case for us, but we have included it in appendix A.10.

3.1 Datasets

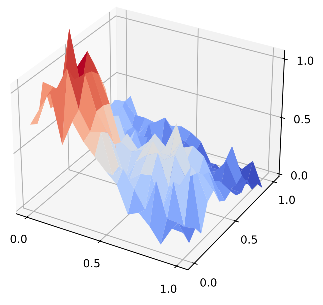
There are two datasets for which we will create polynomial models in this report. The first of these is the two-dimensional Franke function [1], which is widely used to test interpolation and fitting problems. It is defined as

$$\begin{aligned} f(x_1,x_2) = \frac{3}{4}\exp\left(-\frac{(9x_1-2)^2}{4}-\frac{(9x_2-2)^2}{4}\right) \\ +\frac{3}{4}\exp\left(-\frac{(9x_1+1)^2}{49}-\frac{(9x_2-2)}{10}\right) \\ +\frac{1}{2}\exp\left(-\frac{(9x_1-7)^2}{4}-\frac{(9x_2-3)^2}{4}\right) \\ -\frac{1}{5}\exp\left(-(9x_1-4)^2-(9x_2-7)^2\right), \end{aligned}$$

where $x_1, x_2 \in [0,1]$. The Franke function with and without noise is displayed in figure 1 below.



(a) no noise



(b) with noise

Figure 1: Comparison of Franke function with and without noise.

The second dataset we are working with comes from the U.S. Geological Survey (USGS) [2]. Collected through satellite, aerial and ground-based monitoring systems [10], the USGS contains detailed geological data from vast expanses across the Earth's surface. It is used in scientific research, policy-making, and natural resource management [11]. For our purposes, these data are useful because they constitute a valuable 2D dataset, acting as another source of comparison and assessment of the regression methods used in this project. A section of the terrain data from Rogaland in Norway is shown in figure 2 below.

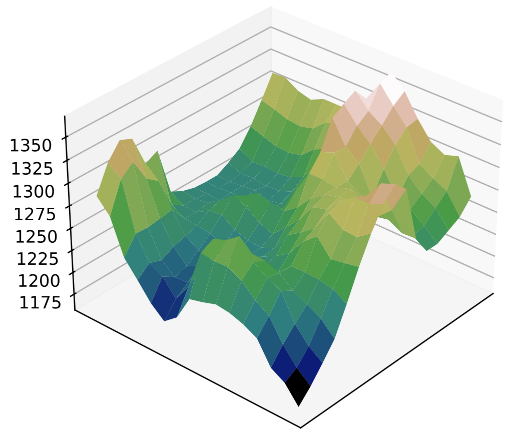


Figure 2: Terrain data from USGS [2] for a 16×16 pixel grid, where the z-axis represents meters above sea level. Note that for visual contrast, the surface of the selected terrain is colored in such a way that the lowest elevations are blue, but they do not represent sea level.

We generated the datasets by using the `meshgrid` method from `numpy` [12], which is a simple way to create every possible combination of points from our input vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. This made sense as a way of generating the data in this case, given that both datasets represented data from a continuous 2D plane where we would need every combination of points, instead of doing this manually adding every pair of points to the design matrix. We chose to use the Franke function with noise, as seen in figure 1b. The code for the generation for the Franke function dataset can be seen in listing 1, and a similar method was used for the terrain data, where output values $\boldsymbol{y}$ were determined not by calculation, but rather by assigning elevations to the generated coordinates based on data from the USGS.

```
import numpy as np

def generate_data_Franke(
    n: int,
    seed: int,
    noise: bool,
    ) -> tuple:
    np.random.seed(seed)

    x1 = np.linspace(0, 1, n)
    x2 = np.linspace(0, 1, n)
    X1, X2 = np.meshgrid(x1, x2)
    Y = Franke_function(X1, X2, noise)

    x1 = X1.flatten().reshape(-1, 1)
    x2 = X2.flatten().reshape(-1, 1)
    y = Y.flatten().reshape(-1, 1)

    return (x1, x2), y
```

Listing 1: Data generation for the Franke function dataset. A version of this function was implemented in the final code.

3.2 General Experimental Procedure

In the code, there is a `BaseModel` class that does most of the processing through numerous methods, which for the most part work simiarly to the analogous methods in `scikit-learn`[12]. It has three subclasses, `OrdinaryLeastSquares`, `RidgeRegression` and `LassoRegression`. They share most of the methods from the superclass, but naturally the training of the model differs between them. Results are stored in a separate class (`Results`) for easier extraction and plotting. Most of the code can be found in appendix F or on the project's GitHub repository[13]. Listing 2 contains sample code demonstrating basic usage.

[12]One key difference is that `scikit-learn` requires specification of a design matrix with polynomial features. Our project exclusively considers polynomial models, so this was not necessary.

[13]`https://github.com/emmastorberg/FYS-STK4155_Project1`

```
x, y = generate_data_Franke(
    n=50,
    seed=3,
    multidim=True,
    )

model = RidgeRegression(
    degree=5,
    lmbda=0.1,
    multidim=True,
    )

X = model.create_design_matrix(x)
X_train, X_test, y_train, y_test = model.
    split_test_train(X, y, test_size=0.2)

# scale
model.fit(X_train, y_train, with_std=True)
X_train = model.transform(X_train)
X_test = model.transform(X_test)

# train and predict
model.train(X_train, y_train)
y_tilde_train = model.predict(X_train)
y_tilde_test = model.predict(X_test)
```

Listing 2: Sample code with classes and methods we defined for this project, which creates a ridge regression model of degree 5 with $\lambda = 0.1$.

3.2.1 Resampling

We use two resampling methods in our experiments, *bootstrap resampling* and *k-fold cross validation*. Bootstrap resampling is used to compensate for having small amounts of data, while cross-validation is a good way to test the performance of our model by creating k disjoint sets of testing data. As these are not the focus of our project, we will not detail them more here. Our implementation closely follows that of Hjorth-Jensen [5]; more information can be found in his online resources.

3.3 Choice of Scaler

We have previously touched upon the concept of scaling and how it may be necessary to avoid punishing datasets of generally large magnitudes. One common approach is to process the data by subtracting the mean of the dataset and dividing by its standard deviation, for a mean of 0 and a standard deviation of 1. This is known as *standard scaling*, and is particularly applicable when the data is normally distributed [13]. In his technical report from 1979, Richard Franke presents his function as "[consisting] of two Gaussian peaks and a sharper Gaussian dip superimposed on a surface sloping toward the first quadrant" [1]. This is a clear way in which the Franke function can be linked to the normal (Gaussian) distribution. As for terrain data, we choose to operate under the assumption that they also follow some distribution that is fairly close to a normal distribution. This is based on the intuition that all elevations are not equally common, with most of the surface being relatively flat, and extreme elevations found in the form of mountains and deep valleys appearing less often the more extreme they are. In this sense, our datasets seem suited for standard scaling.

Another important consideration before implementing standard scaling is whether or not there are large outliers in the dataset. If so, scaling in this way may not be appropriate [14]. As we have no reason to expect this will be an issue, we will use the `StandardScaler` method from `scikit-learn` to scale our models.

One last thing of note with regards to scaling is that we have chosen to exclude the intercept, so to keep the constant term of the polynomial, we add the mean of `y_train` to the predictions `y_tilde`.

3.4 Choice of Hyperparameters

In our experiments, the most important hyperparameters to choose were degree of the polynomial model, as well as λ in the case of ridge and LASSO regression. To do so, we performed a *grid search*. This is done by exhaustively evaluating every combination of hyperparameters. In our case, we needed to determine what combination of polynomial degree and λ gave the lowest MSE.

4 Results

We studied the MSE of using the OLS method on both datasets first, and compared the training data to testing data as we varied the model complexity through its polynomial degree. The results of this are shown in figure 3. All R^2 score plots can be found in appendix B.

We see in this figure that the model overall performs quite well: The MSE seems to be tapering off as we increase model complexity, and without any major spikes in the MSE of the testing data, there does not seem to be massive overfitting. We can also infer this from the fact that the polynomial degrees tested are quite low and the number of data points is relatively high ($n = 50$, which gives $50 \times 50 = 2500$ data points), so it would be hard to learn the patterns in all of them. There is also no major difference in the R^2 score[14], indicating that the model is not substantially better with data it knows than with unknown data.

This model performs so well, in fact, that it is hard to clearly see the phenomena we are exploring in this report. We ran more experiments and pushed the model to its limits by decreasing the grid size to 15 for the Franke function and 16 for the terrain data while also trying higher complexities in an effort to induce overfitting. The results of this experiment can be seen in figure 4.

Just as we saw in figure 3, the more challenging conditions in figure 4 still yield a relatively low MSE overall, and for training data it decreases with increasing complexity. Training and testing data give MSEs that are overall quite close, indicating a good fit of the model. However, we can still see at degree 5 for both datasets that the MSE from testing data begins diverging from the training data MSE. This is what we would expect to see when we have overfitting, as is the high (and growing) R^2 of the training data compared to the testing data with increasing complexity[15]. It warrants further exploration to see if this is truly a case of overfitting.

[14]Appendix B, figure 14

[15]Appendix B, figure 14.

Next, we fixed the degree of highest test MSE for each data set, and considered the other types of regression for various λ to see if we would see any improvements. If so, it might mean that the high test MSE we saw in each case really was due to overfitting, which is what the addition of the regularization term in ridge and LASSO regression aims to combat. The results of this experiment are shown in figure 5, where the Franke function models were fixed at degree 8, and the terrain data models at degree 7.

As we can see in figure 5, even in the OLS case we thought most likely to have some overfitting, the introduction of λ does not lower the MSE below the MSE of the OLS models. The R^2 score is also generally negative for ridge and LASSO regression[16]. These are unexpected findings, and call into question the earlier hypothesis of overfitting, as we would expect ridge or LASSO regression to reduce the MSE and have better R^2 scores of their testing data compared to OLS if that were the case.

At this point, we performed a grid search[17] with 5-fold cross-validation to determine the optimal hyperparameters (degree and λ) for further testing. Using resampling, we saw a lowered MSE of 0.03 of the Franke function model compared to when we fixed a degree earlier without resampling. This led to a choice of $\lambda = 0.001$ for both ridge and LASSO regression for this dataset. For the terrain data however, the MSE consistently had an order of magnitude in the thousands, and was minimized at extremely large values of λ for both regression types, so we chose to set $\lambda = 1000$ to reflect this result. We continued testing with these values for λ (figure 6 and appendix B, figure 17).

Similar to what we saw for a fixed degree, ridge and LASSO regression ostensibly performed worse than the OLS model for all polynomial degrees we tested. The R^2 score is negative[18] for all testing data for both ridge and LASSO, while their training data R^2 score stays close to zero.

[16]Appendix B, figure 16.

[17]Full tables in appendix C.

[18]Appendix B, figure 17.

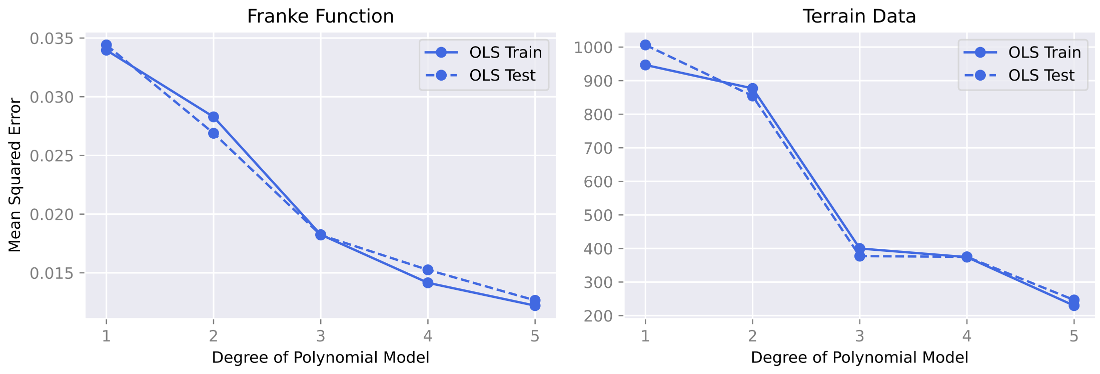


Figure 3: MSE for OLS method as a function of polynomial degree up to degree 5 for each of the two datasets.

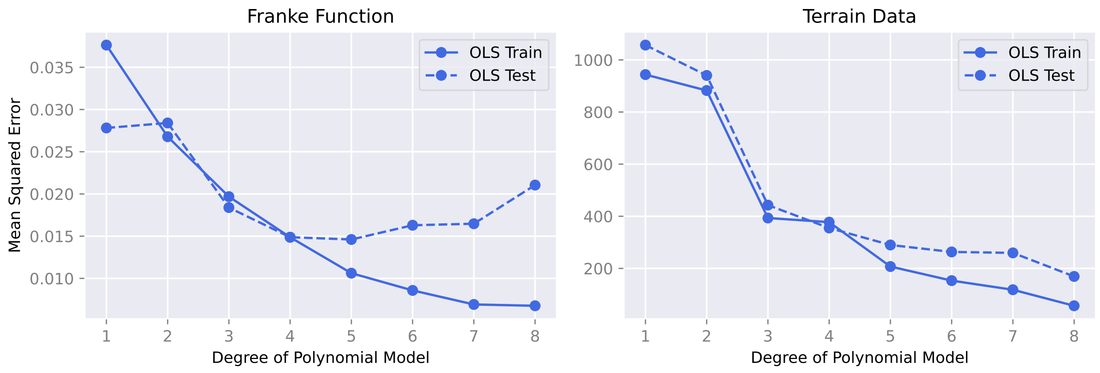


Figure 4: MSE of OLS method as a function of polynomial degree up to degree 8 for each of the two datasets.

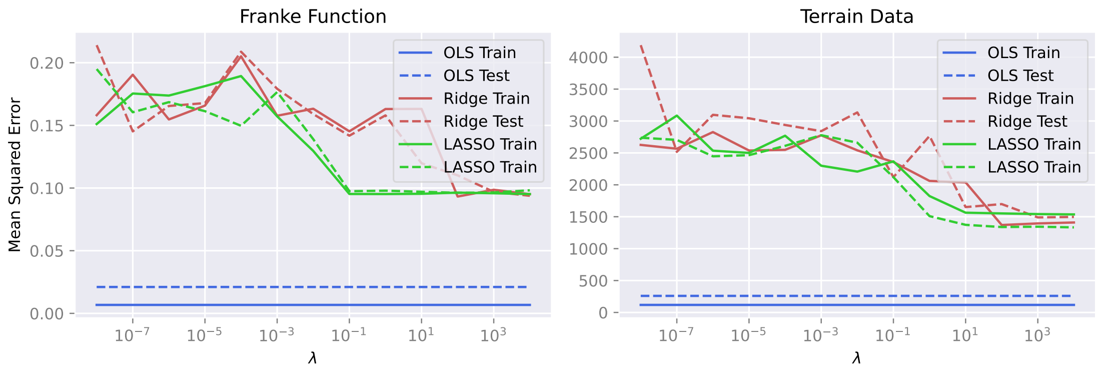


Figure 5: MSE of all models as function of λ for fixed polynomial degrees.

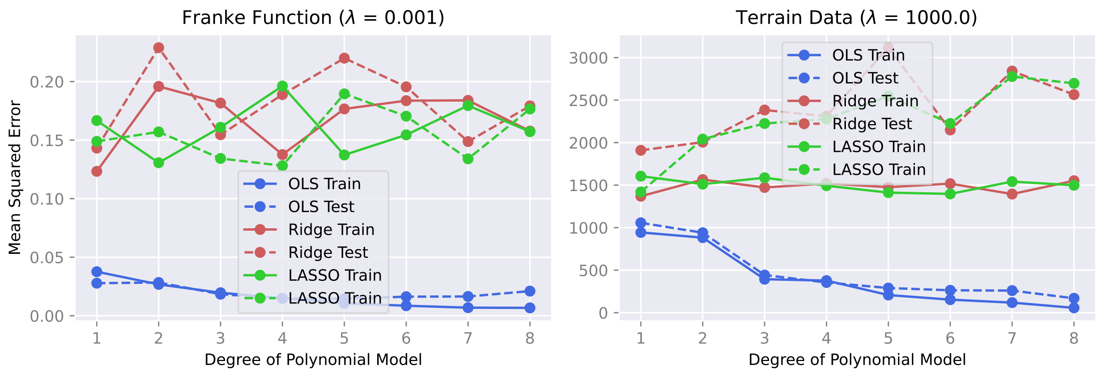


Figure 6: MSE as a function of polynomial degree for all models, with λ fixed at previously determined optimal values.

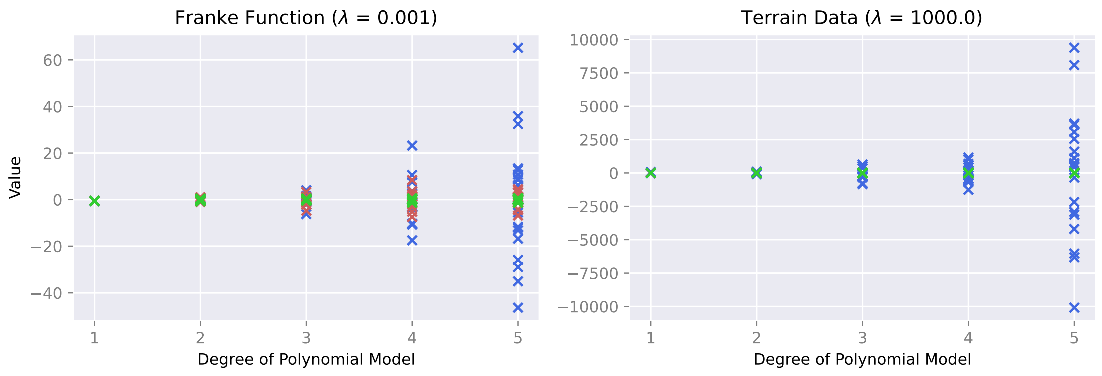


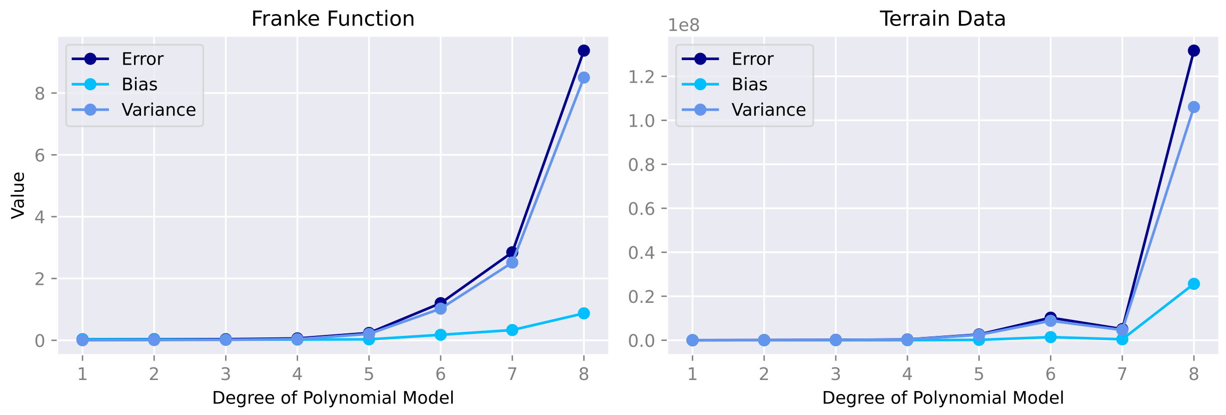
Figure 7: Coefficients of all models by polynomial degree for optimal λ for each dataset up to degree 5.

In addition, we were not able to reproduce the low MSEs found with the grid search, which would be more in line with the observed MSE of the OLS model, as we see in figure 6 above. These statistical findings in combination suggest a very poor fit of the ridge and LASSO regression models.

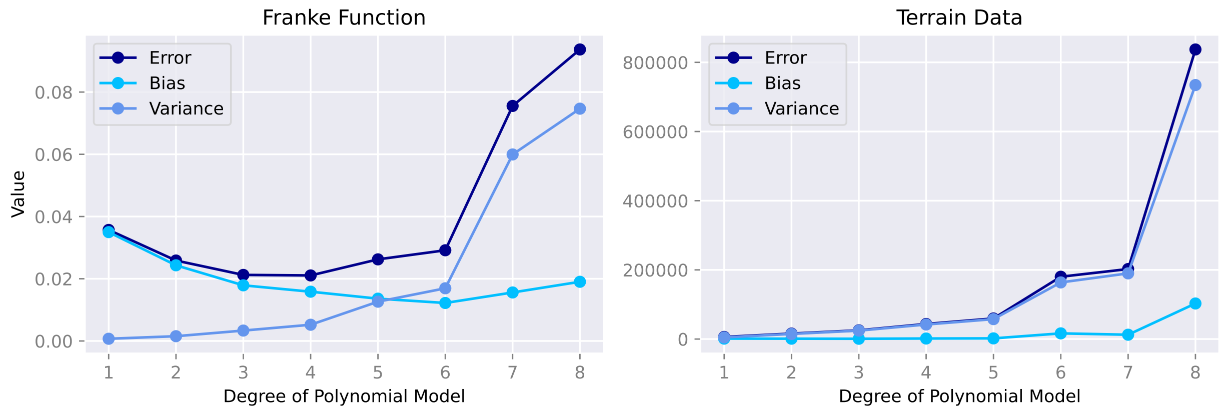
Continuing our tests, we examined the coefficients of the polynomial models to determine whether the ridge and LASSO regression models behaved consistently with the theory in this aspect. The results of this can be seen in figure 7 above. The plot only displays the coefficients of the models up to degree 5 so that the coefficients found by ridge and LASSO regression would remain visible, as the OLS coefficients grew quite large[19]. As we might expect from the theory, the coefficients found for ridge and LASSO regression are smaller than the optimal coefficients of OLS, so our regularization terms are, in fact, influencing what coefficients we find.

The next step in our exploration was to examine the bias-variance trade-off in practice. Using 100 bootstrap resamplings, we compared the bias and variance of OLS models trained on datasets of different sizes. This can be seen in figure 8 on the next page.

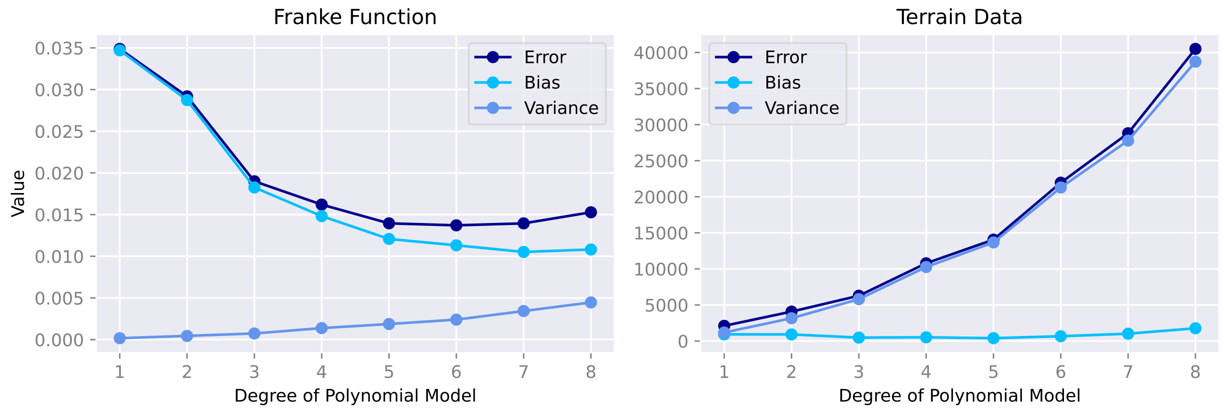
[19]Full plot up to degree 8 can be found in appendix D.



(a) Bias-variance trade-off of OLS models as function of polynomial degree with grid size $n_{\mathrm{Franke}} = 15$ and $n_{\mathrm{Terrain}} = 16$.



(b) Bias-variance trade-off of OLS models as function of polynomial degree with grid size $n_{\mathrm{Franke}} = 30$ and $n_{\mathrm{Terrain}} = 32$.



(c) Bias-variance trade-off of OLS models as function of polynomial degree with grid size $n_{\mathrm{Franke}} = 60$ and $n_{\mathrm{Terrain}} = 64$.

Figure 8: Bias-variance trade-off illustrated for three different dataset sizes for the Franke function and terrain data.

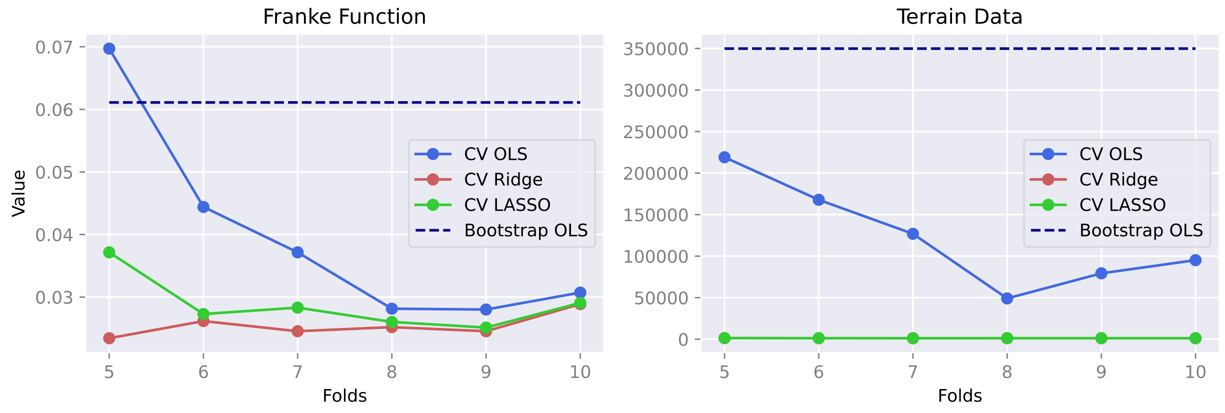


Figure 9: Comparison of MSE in k-fold cross-validation compared to bootstrap resampling.

What we see in figure 8 confirms part of what we expected from the theory: Increasing dataset size leads to better performance (lower MSEs) of all models. In the case of the Franke function, we once again see what looks like overfitting with small dataset sizes and high complexities (MSE grows with polynomial degree), which is exactly as we would expect. This overfitting effect seems to lessens as n grows, since it is harder for the model to learn about all points. For the terrain data, it seems that with the chosen values of n, we did not reach this point, as the error is on an upward trend even at the highest n we tested.

By visual inspection of 3 and the grid search tables, it seems degree 4 has a low MSE for both datasets and all regression types, so we chose a 4th degree polynomial for a final test. We implemented another type of resampling, namely k-fold cross-validation, and examined how the MSE of our 4th degree models varied for different numbers of folds (k). Figure 9 above displays how the MSE is lower than when we used bootstrap resampling for all models when using six or more folds, with $k = 8$ optimal for both datasets.

Now, we wish to compare how well the chosen models can predict the data visually. Table 1 to the right displays the hyperparameters we found through our testing up to this point.

Hyperparameter	**Value**
Degree	4
λ_{Franke}	0.001
λ_{Terrain}	1000
n_{Franke}	15
n_{Terrain}	16

Table 1: Hyperparameters determined through structured experiments (n is grid size).

As we might have suspected based on odd experimental results earlier, visual inspection confirms that our models are not functioning as anticipated, and the predictions from the models besides OLS are poor. In particular, LASSO regression on the terrain dataset performs atrociously, as seen in figure 10 on the next page[20]. This final comparison suggests our models cannot make good predictions and do not fit the data well. However, by trial-and-error adjustment of the parameters, we were able to create models that performed substantially better (judging by the plots). These can be seen in figures 11, 12 and 13.

[20]More examples of poor predictions when using the the hyperparameters in table 1 can be found in appendix E.

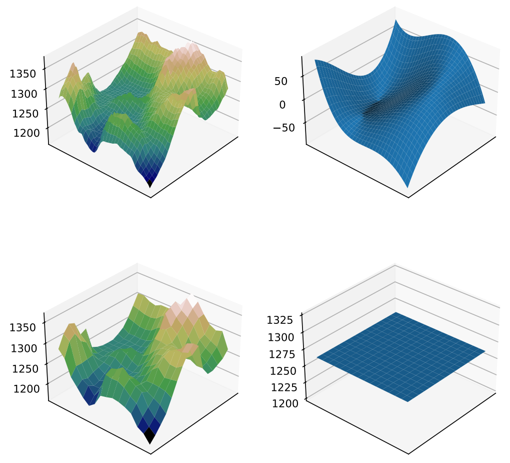


Figure 10: Real terrain data compared to the predictions of our 4th degree LASSO regression model with $\lambda = 1000$ on a 64×64 grid (above) and a 16×16 grid (below).

5 Discussion

It is not immediately clear why our method in determining optimal coefficients and hyperparameters led us astray. However, we saw early on in the grid search that enormous λs were determined to optimize the fit to the terrain dataset ($\lambda_{\text{Ridge}} = 10000$ and $\lambda_{\text{LASSO}} = 1000$). From the theory, we understand that this corresponds to weighting the regularization term heavily, prioritizing minimizing the coefficients themselves over minimizing the error term. It therefore makes sense that large values of λ would set coefficients to zero (or close to it), and in this way, the LASSO regression prediction in figure 10 is consistent with what we expect. What is not apparent is why our structured experiments gave us these values in the first place, especially since all unit testing against `scikit-learn` methods passed without issue[21].

[21] Full code (including tests) can be found at `https://github.com/emmastorberg/FYS-STK4155_Project1`.

Though, as discussed, addition of regularization terms can be a helpful technique for reducing overfitting, we never saw this benefit, even when working with the polynomial degree where training and testing MSE diverged the most (figure 4), where overfitting was most likely. Based on these findings, ridge and LASSO regression exclusively worsened the performance of the models compared to OLS.

We also saw by visual inspection of the predictions compared to the true data that the models performed very poorly with our chosen parameters, as we see in figure 10 to the left and in appendix E. In this sense, we can say that our methods yielded models of poor fit. Still, we saw at the very beginning that our OLS model was able to perform very well with large enough data size (figure 3). It is also clear based on figures 11, 12 and 13 that there exist better choices of hyperparameters than what our search and findings in figure 6 seem to suggest, so it seems unfair to discredit our models entirely. Further discussion of optimal performance will be with reference to these results.

In figures 11, 12 and 13, we have used polynomials of higher degree, set $\lambda = 0.0001$ for both ridge and LASSO regression, and quadrupled the dataset size for comparison. Theoretically, we would expect this to give us better models, and it is evident that it does. Considering first the two different datasets, all models predict Franke function data more accurately than the terrain data, probably owing to the complexities of the input data that are generally removed in the predictions. In the case of the Franke function, this is a benefit, because our dataset was noisy to begin with. For the terrain data, it is suboptimal, as it smooths out the surface too much compared to the real-life terrain.

As for comparison of the different regression types, OLS and ridge give us visually similar predictions that do reasonably well compared to the real data, especially for the Franke function, as mentioned. LASSO stands out as less complex and detailed in its predictions, but still performs much better than with our previously determined hyperparameters.

Nonetheless, OLS and ridge are visually better methods when using the second set of parameters we found. We do not exclude the possibility that there may exist even better parameters for some or all of the regression types, as the purpose of our final test by trial and error was simply to show the error in the previously determined hyperparameters.

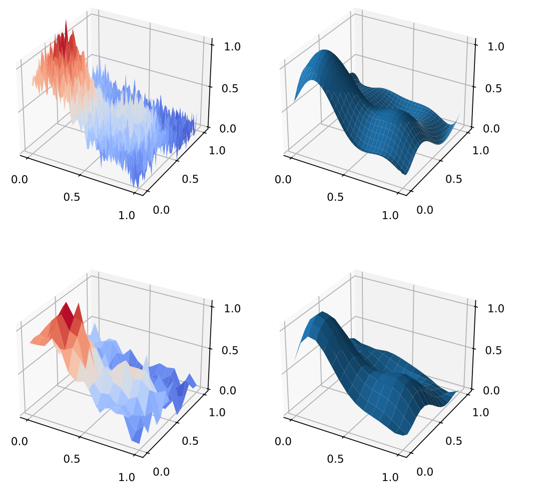
Further research should aim to uncover the error in the method for determining optimal parameters, as we would expect our grid search to yield quite different results than what we determined experimentally in this case. In doing so, it would be useful to clearly establish whether there is overfitting or not, as the statistical metrics shown in figures 4 and 15 suggested this might be the case, yet introduction of regularization terms did not rectify the problem (figures 5 and 16), and was quite frankly not consistent with the theory.

6 Conclusion

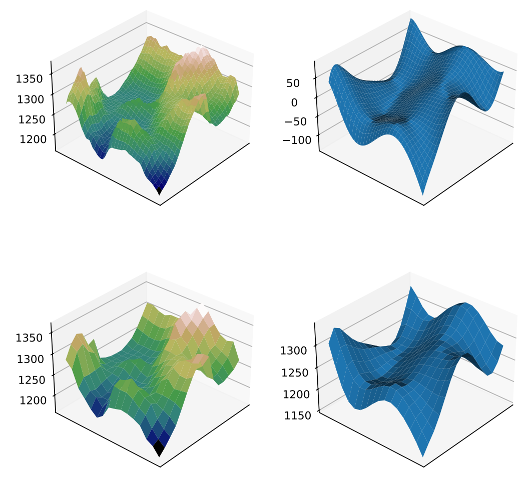
Analytical analysis of the ridge and LASSO regression cost functions in the theory part had us conclude that they will make coefficients smaller (or eliminate them entirely) in comparison with OLS, which we were able to confirm experimentally (figure 7). By the bias-variance trade-off, we also surmised they would aid us in reducing overfitting. Our results did not show this, as a wide variety of values for the hyperparameter λ were tested, without any of them coming close to the MSE of the OLS models (figure 6). The hyperparameters identified in the grid search were nonsensical in terms of the dataset and theory (very large λ and polynomial degree as low as 1 were found to be optimal), and the predictions were unsurprisingly poor.

Altogether, this shows that there is most likely some error in the method for determining hyperparameters, as the implementation of the models themselves passes rigorous testing. We were still able to make models that can predict reasonably well, but without solid methods for finding hyperparameters besides trial and error, we can only rely on theoretical intuition, and are not able to confirm that these are the best possible models.

Our experiments showed the importance of testing and visual inspection, and confirmed our understanding of the theory in the sense that when we found unreasonable values, they produced bad predictions. While we were not able to show or work to reduce overfitting ourselves in the experiments, we are left with heightened awareness of this problem and knowledge of ways to combat it, despite a faulty implementation in this case.

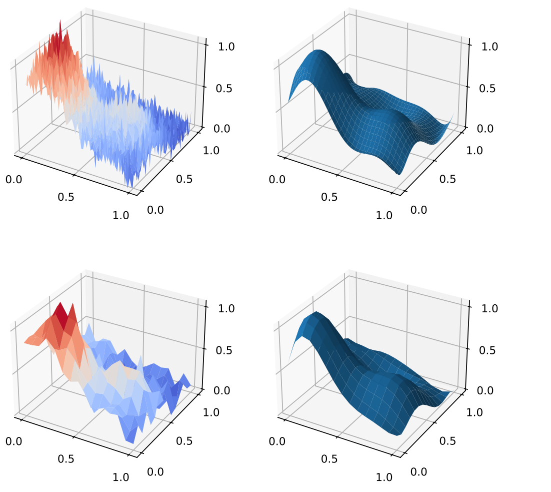


(a) Real Franke function data compared to the predictions of our 6th degree OLS model on a 60×60 grid (above) and a 15×15 grid (below).

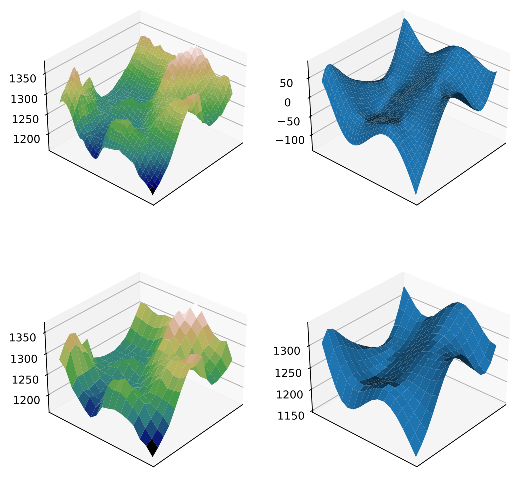


(b) Real terrain data compared to the predictions of our 6th degree OLS model on a 64×64 grid (above) and a 16×16 grid (below).

Figure 11: Predictions from OLS models made using other parameters than the ones determined experimentally in this report.

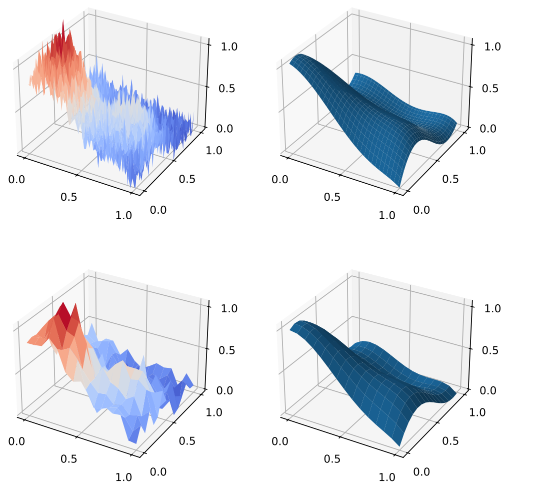


(a) Real Franke function data compared to the predictions of our 6th degree ridge regression model with $\lambda = 0.0001$ on a 60×60 grid (above) and a 15×15 grid (below).

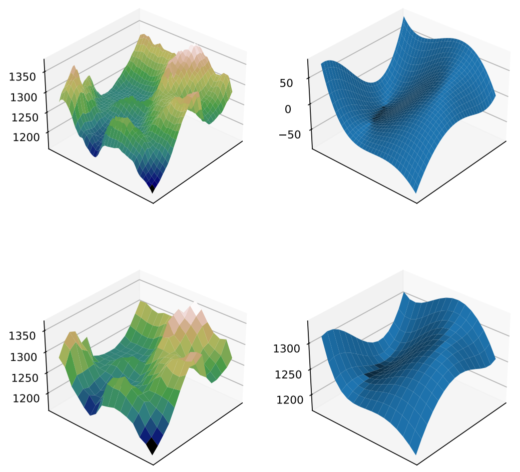


(b) Real terrain data compared to the predictions of our 6th degree ridge regression model with $\lambda = 0.0001$ on a 64×64 grid (above) and a 16×16 grid (below).

Figure 12: Predictions from ridge regression models made using other parameters than the ones determined experimentally in this report.



(a) Real Franke function data compared to the predictions of our 6th degree LASSO regression model with $\lambda = 0.0001$ on a 60×60 grid (above) and a 15×15 grid (below).



(b) Real terrain data compared to the predictions of our 6th degree LASSO regression model with $\lambda = 0.0001$ on a 64×64 grid (above) and a 16×16 grid (below).

Figure 13: Predictions from LASSO regression models made using other parameters than the ones determined experimentally in this report.

7 References

[1] Richard Franke. *A Critical Comparison of Some Methods for Interpolation of Scattered Data*. Tech. rep. NPS-53-79-003. Naval Postgraduate School, 1979, p. 13.

[2] *USGS: Earth Explorer*. U.S. Department of the Interior. URL: https://earthexplorer.usgs.gov.

[3] *Coefficient of determination*. Wikipedia. URL: https://en.wikipedia.org/wiki/Coefficient_of_determination.

[4] *Gauss-Markov theorem*. Wikipedia. URL: https://en.wikipedia.org/wiki/Gauss%E2%80%93Markov_theorem.

[5] Morten Hjorth-Jensen. *From Ordinary Linear Regression to Ridge and Lasso Regression*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week35.html#ridge-and-lasso-regression.

[6] Seung-Whan Choi. "The Effect of Outliers on Regression Analysis: Regime Type and Foreign Direct Investment". In: *Quarterly Journal of Political Science* (2009).

[7] *Singular value decomposition*. Wikipedia. URL: https://en.wikipedia.org/wiki/Singular_value_decomposition.

[8] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, 2022.

[9] *Lasso*. Scikit-Learn Developers. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html.

[10] *USGS Geospatial Data Sources*. U.S. Department of the Interior. URL: https://www.usgs.gov/educational-resources/usgs-geospatial-data-sources.

[11] *What does the USGS (United States Geological Survey) do?* U.S. Department of the Interior. URL: https://www.usgs.gov/faqs/what-does-usgs-united-states-geological-survey-do.

[12] *numpy.meshgrid*. NumPy Developers. URL: https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html.

[13] Dexter Chu. *When to Normalize or Standardize Data*. URL: https://www.secoda.co/learn/when-to-normalize-or-standardize-data.

[14] *Compare the effect of different scalers on data with outliers*. Scikit-Learn Developers. URL: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html.

A Derivation of Key Expressions

A.1 Expectation Value of $\boldsymbol{y}$ for a Given Element y_i

$$\begin{aligned}\mathbb{E}[y_i] &= \mathbb{E}[f(\boldsymbol{x}) + \boldsymbol{\epsilon}] \\ &= \mathbb{E}[f(\boldsymbol{x})] + \mathbb{E}[\boldsymbol{\epsilon}] \\ &= \mathbb{E}[f(\boldsymbol{x})] \\ &= \sum_j x_{ij}\beta_j\end{aligned}$$

A.2 Variance of a Given Element y_i

$$\begin{aligned}\mathrm{Var}(y_i) &= \mathbb{E}[(y_i - \mathbb{E}[y_i]^2] \\ &= \mathbb{E}[y_i^2 - 2y_i\mathbb{E}[y_i] + E[y_i]^2] \\ &= \mathbb{E}[(f(x_i) + \varepsilon_i)^2 - 2(f(x_i) + \varepsilon_i)\mathbb{E}[(f(x_i) + \varepsilon_i)] + E[(f(x_i) + \varepsilon_i)]^2] \\ &= \mathbb{E}[f(x_i)^2 + 2f(x_i)\varepsilon_i + \varepsilon_i^2 - 2(f(x_i) + \varepsilon_i)\mathbb{E}[f(x_i)] + E[f(x_i)]^2] \\ &= \mathbb{E}[f(x_i)^2 + 2f(x_i)\varepsilon_i + \varepsilon_i^2 - 2f(x_i)\epsilon_i - 2f(x_i)^2 + f(x_i)^2] \\ &= \mathbb{E}[\varepsilon^2]\end{aligned}$$

We know that $Var(\varepsilon_i) = \sigma^2 = \mathbb{E}[\varepsilon^2] - \mathbb{E}[\varepsilon]^2$. We therefore see that:

$$\mathrm{Var}(y_i) = \mathbb{E}[\varepsilon^2] = \sigma^2$$

A.3 Optimal $\hat{\boldsymbol{\beta}}$ for the OLS Method

$$\begin{aligned}-\frac{2}{n}(\boldsymbol{y} - X\hat{\boldsymbol{\beta}})^T X &= 0 \\ \hat{\boldsymbol{\beta}}^T X^T X &= \boldsymbol{y}^T X \\ \hat{\boldsymbol{\beta}}^T &= \boldsymbol{y}^T X (X^T X)^{-1} \\ \hat{\boldsymbol{\beta}} &= (X^T X)^{-1} X^T \boldsymbol{y}\end{aligned}$$

A.4 Expectation Value of Optimal $\hat{\boldsymbol{\beta}}$ for the OLS Method

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

A.5 Variance of $\hat{\boldsymbol{\beta}}$ for the OLS Method

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

A.6 Optimal $\hat{\boldsymbol{\beta}}$ for Ridge Regression

We know that the optimal parameters $\hat{\boldsymbol{\beta}}_{\text{Ridge}}$ can be determined from the minimum of the cost function $C(\boldsymbol{\beta})$, given by:

$$C(\boldsymbol{\beta}) = \frac{1}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) + \frac{1}{n}\lambda\boldsymbol{\beta}^T\boldsymbol{\beta}$$

We find the derivative of each term of C:

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

$$\frac{\partial}{\partial\boldsymbol{\beta}}(\boldsymbol{\beta}^T\boldsymbol{I}\boldsymbol{\beta}) = \boldsymbol{\beta}^T(\mathbf{I} + \mathbf{I}^T) = 2\boldsymbol{\beta}^T\mathbf{I}$$

Combining these and setting equal to 0 to find the minimum gives us the following:

$$\begin{aligned}
\frac{\partial C}{\partial \boldsymbol{\beta}} = \frac{-2}{n}(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T\mathbf{X} + \frac{2}{n}\boldsymbol{\beta}^T\mathbf{I} &= 0 \\
-\boldsymbol{y}^T\mathbf{X} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X} + \lambda\boldsymbol{\beta}^T\mathbf{I} &= 0 \\
\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}) &= \boldsymbol{y}^T\mathbf{X} \\
(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^T\boldsymbol{\beta} &= \mathbf{X}^T\boldsymbol{y} \\
(\mathbf{X}^T(\mathbf{X}^T)^T + \lambda\mathbf{I}^T)\boldsymbol{\beta} &= \mathbf{X}^T\boldsymbol{y} \\
(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} &= \mathbf{X}^T\boldsymbol{y} \\
\hat{\boldsymbol{\beta}} &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\boldsymbol{y}
\end{aligned}$$

A.7 Expectation Value of Optimal $\hat{\boldsymbol{\beta}}$ for Ridge Regression

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

A.8 Predicted Output $\tilde{\boldsymbol{y}}_{\textbf{Ridge}}$ in Terms of Desired Output $\boldsymbol{y}$

$$\tilde{\boldsymbol{y}}_{\text{Ridge}} = \mathbf{X}\boldsymbol{\beta}_{\text{Ridge}} = \mathbf{U\Sigma V}^T(\mathbf{V\Sigma V}^T + \lambda\mathbf{I})^{-1}(\mathbf{U\Sigma V}^T)^T\boldsymbol{y}$$

We replace $\mathbf{I}$ with $\mathbf{VV}^T$:

$$\begin{aligned}
\tilde{\boldsymbol{y}}_{\text{Ridge}} = \mathbf{X}\boldsymbol{\beta}_{\text{Ridge}} &= \mathbf{U\Sigma V}^T(\mathbf{V\Sigma V}^T + \lambda\mathbf{VV}^T)^{-1}(\mathbf{U\Sigma V}^T)^T\boldsymbol{y} \\
&= \mathbf{U\Sigma V}^T(\mathbf{V}(\tilde{\mathbf{\Sigma}} + \lambda\mathbf{I})\mathbf{V}^T)^{-1}\mathbf{V\Sigma}^T\mathbf{U}^T\boldsymbol{y} \\
&= \mathbf{U\Sigma V}^T(\mathbf{V}^T)^{-1}(\tilde{\mathbf{\Sigma}} + \lambda\mathbf{I})^{-1}\mathbf{V}^{-1}\mathbf{V\Sigma}^T\mathbf{U}^T\boldsymbol{y} \\
&= \mathbf{U\Sigma}(\tilde{\mathbf{\Sigma}} + \lambda\mathbf{I})^{-1}\mathbf{\Sigma}^T\mathbf{U}^T\boldsymbol{y}
\end{aligned}$$

At this point it is important to note that the matrix $\tilde{\mathbf{\Sigma}} + \lambda\mathbf{I}$ is nonsingular and therefore invertible. We see that it is a diagonal matrix, with elements in the form $\sigma_i^2 + \lambda$. As such, when it is inverted, it remains a diagonal matrix with elements in the form $(\sigma_i^2 + \lambda)^{-1}$. We see that the product $\mathbf{\Sigma}(\tilde{\mathbf{\Sigma}} + \lambda\mathbf{I})^{-1}\mathbf{\Sigma}^T$ has elements in the form $\sigma_i \cdot (\sigma_i^2 + \lambda)^{-1} \cdot \sigma_i = \frac{\sigma_i^2}{\sigma_i^2 + \lambda}$ up to diagonal element $p - 1$, and it has zeros on the diagonals after.

This means that we have:

$$\mathbf{U}\begin{pmatrix} \frac{\sigma_0^2}{\sigma_0^2+\lambda} & & & & & & \\ & \frac{\sigma_1^2}{\sigma_1^2+\lambda} & & & & & \\ & & \ddots & & & & \\ & & & \frac{\sigma_{p-1}^2}{\sigma_{p-1}^2+\lambda} & & & \\ & & & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{pmatrix}\mathbf{U}^T\boldsymbol{y} = \sum_{j=0}^{p-1}\mathbf{u}_j\mathbf{u}_j^T\frac{\sigma_j^2}{\sigma_j^2+\lambda}\boldsymbol{y}$$

A.9 Bias-Variance Trade-off

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

A.10 Optimal $\hat{\boldsymbol{\beta}}$ for Lasso Regression

This formula is only valid in the specific case where the design matrix is equal to the identity matrix.

$$\hat{\beta}_i = \begin{cases} y_i - \frac{\lambda}{2} \text{ if } y_i > \frac{\lambda}{2} \\ y_i + \frac{\lambda}{2} \text{ if } y_i < -\frac{\lambda}{2} \\ 0 \text{ if } |y_i| \leq \frac{\lambda}{2} \end{cases}$$

B Graphs of R^2 Scores

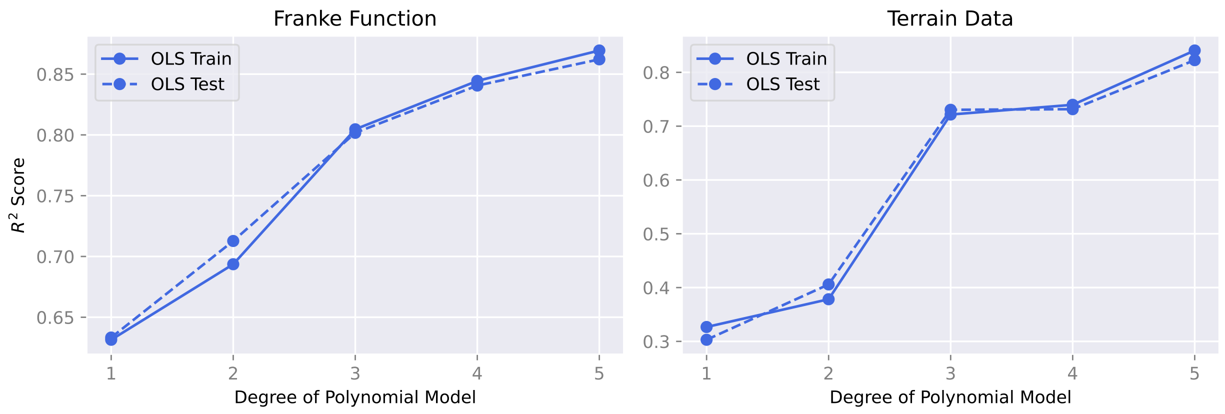


Figure 14: R^2 score of OLS method as a function of polynomial degree up to degree 5 for each of the two datasets, with grid size $n = 50$.

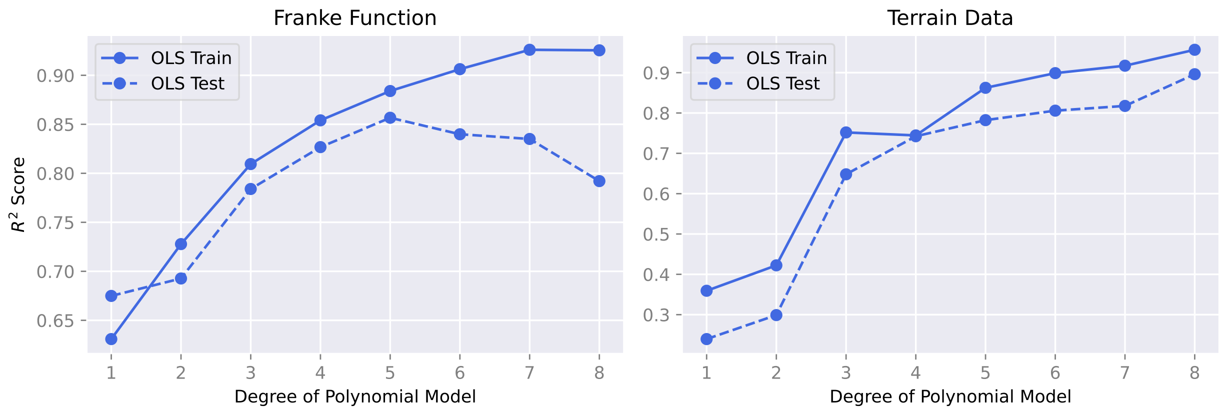


Figure 15: R^2 score of OLS method as a function of polynomial degree up to degree 8 for each of the two datasets, with grid size reduced to $n = 15$ for Franke function and $n = 16$ for terrain data.

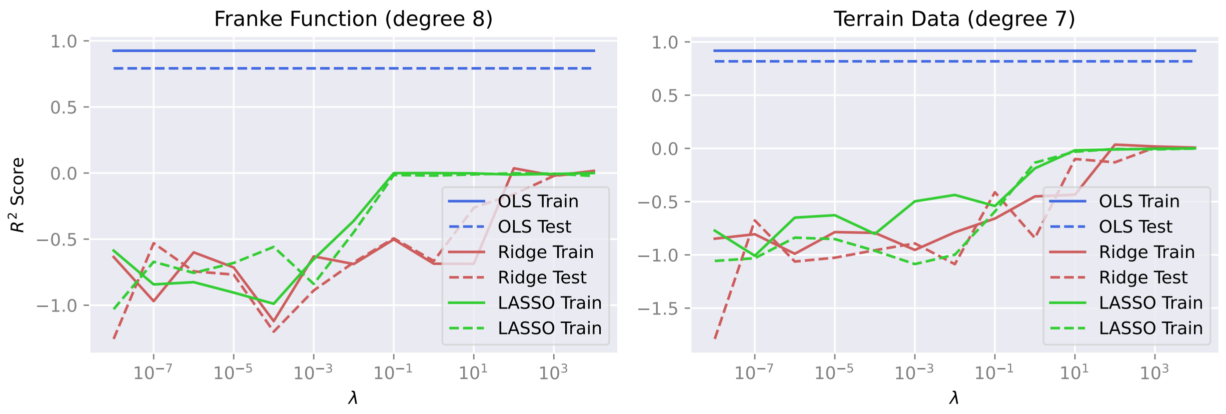


Figure 16: R^2 score of all models as function of λ for fixed polynomial degrees.

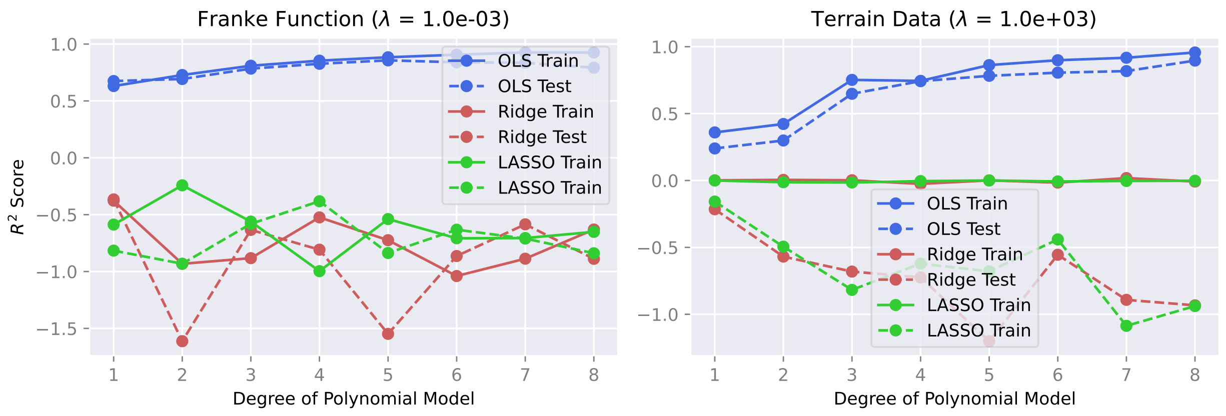
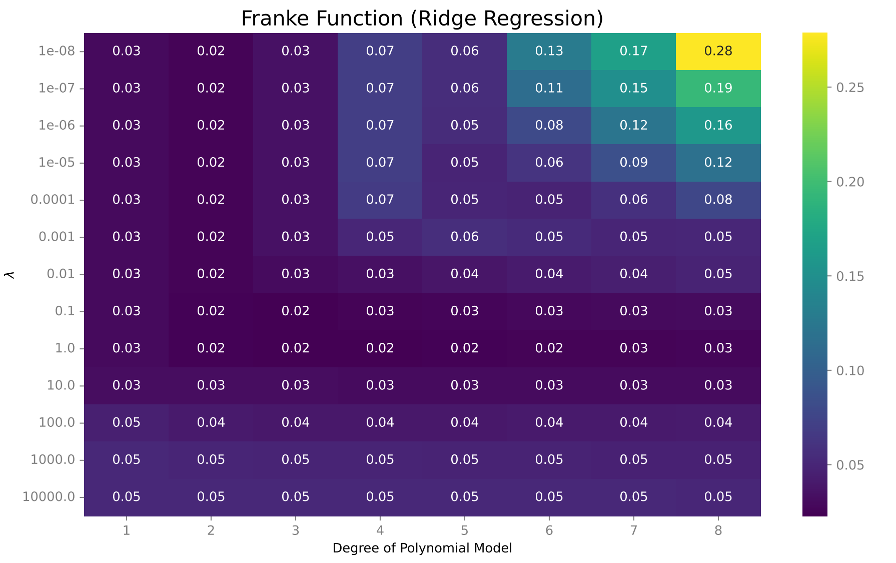


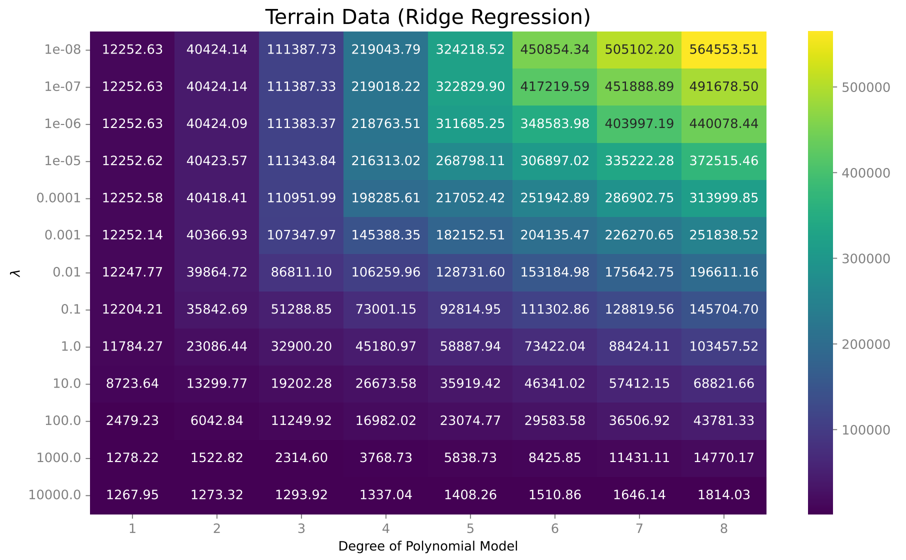
Figure 17: R^2 score as a function of polynomial degree for all models, with λ fixed at previously determined optimal values.

C Grid Search Tables



λ	1	2	3	4	5	6	7	8
1e-08	0.03	0.02	0.03	0.07	0.06	0.13	0.17	0.28
1e-07	0.03	0.02	0.03	0.07	0.06	0.11	0.15	0.19
1e-06	0.03	0.02	0.03	0.07	0.05	0.08	0.12	0.16
1e-05	0.03	0.02	0.03	0.07	0.05	0.06	0.09	0.12
0.0001	0.03	0.02	0.03	0.07	0.05	0.05	0.06	0.08
0.001	0.03	0.02	0.03	0.05	0.06	0.05	0.05	0.05
0.01	0.03	0.02	0.03	0.03	0.04	0.04	0.04	0.05
0.1	0.03	0.02	0.02	0.03	0.03	0.03	0.03	0.03
1.0	0.03	0.02	0.02	0.02	0.02	0.02	0.03	0.03
10.0	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
100.0	0.05	0.04	0.04	0.04	0.04	0.04	0.04	0.04
1000.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
10000.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05

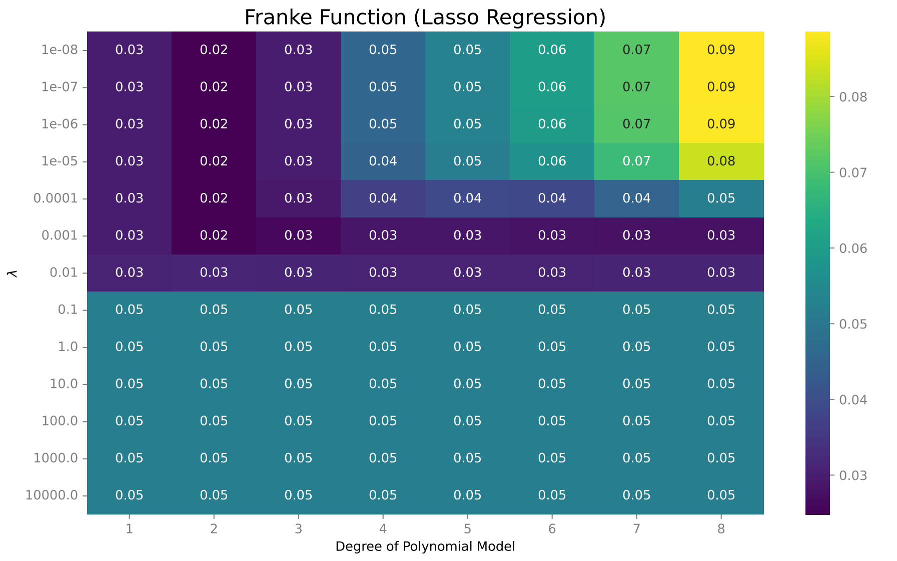
(a) Grid search table of ridge regression for Franke function dataset



λ	1	2	3	4	5	6	7	8
1e-08	12252.63	40424.14	111387.73	219043.79	324218.52	450854.34	505102.20	564553.51
1e-07	12252.63	40424.14	111387.33	219018.22	322829.90	417219.59	451888.89	491678.50
1e-06	12252.63	40424.09	111383.37	218763.51	311685.25	348583.98	403997.19	440078.44
1e-05	12252.62	40423.57	111343.84	216313.02	268798.11	306897.02	335222.28	372515.46
0.0001	12252.58	40418.41	110951.99	198285.61	217052.42	251942.89	286902.75	313999.85
0.001	12252.14	40366.93	107347.97	145388.35	182152.51	204135.47	226270.65	251838.52
0.01	12247.77	39864.72	86811.10	106259.96	128731.60	153184.98	175642.75	196611.16
0.1	12204.21	35842.69	51288.85	73001.15	92814.95	111302.86	128819.56	145704.70
1.0	11784.27	23086.44	32900.20	45180.97	58887.94	73422.04	88424.11	103457.52
10.0	8723.64	13299.77	19202.28	26673.58	35919.42	46341.02	57412.15	68821.66
100.0	2479.23	6042.84	11249.92	16982.02	23074.77	29583.58	36506.92	43781.33
1000.0	1278.22	1522.82	2314.60	3768.73	5838.73	8425.85	11431.11	14770.17
10000.0	1267.95	1273.32	1293.92	1337.04	1408.26	1510.86	1646.14	1814.03

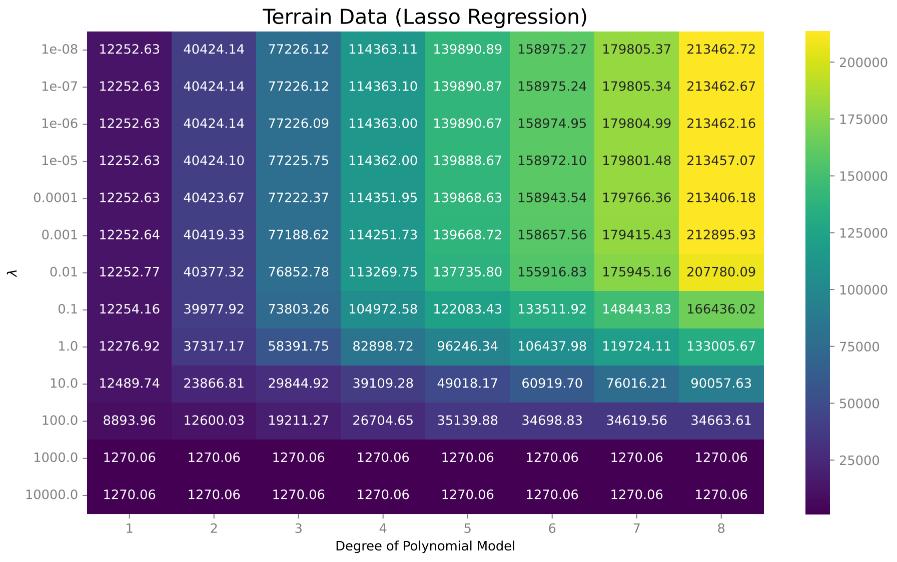
(b) Grid search table of ridge regression for terrain dataset

Figure 18: Grid search tables for finding optimal polynomial degree and λ for ridge regression for each of the data sets.



λ	1	2	3	4	5	6	7	8
1e-08	0.03	0.02	0.03	0.05	0.05	0.06	0.07	0.09
1e-07	0.03	0.02	0.03	0.05	0.05	0.06	0.07	0.09
1e-06	0.03	0.02	0.03	0.05	0.05	0.06	0.07	0.09
1e-05	0.03	0.02	0.03	0.04	0.05	0.06	0.07	0.08
0.0001	0.03	0.02	0.03	0.04	0.04	0.04	0.04	0.05
0.001	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03
0.01	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
0.1	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
1.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
10.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
100.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
1000.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
10000.0	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05

(a) Grid search table of LASSO regression for Franke function dataset



λ	1	2	3	4	5	6	7	8
1e-08	12252.63	40424.14	77226.12	114363.11	139890.89	158975.27	179805.37	213462.72
1e-07	12252.63	40424.14	77226.12	114363.10	139890.87	158975.24	179805.34	213462.67
1e-06	12252.63	40424.14	77226.09	114363.00	139890.67	158974.95	179804.99	213462.16
1e-05	12252.63	40424.10	77225.75	114362.00	139888.67	158972.10	179801.48	213457.07
0.0001	12252.63	40423.67	77222.37	114351.95	139868.63	158943.54	179766.36	213406.18
0.001	12252.64	40419.33	77188.62	114251.73	139668.72	158657.56	179415.43	212895.93
0.01	12252.77	40377.32	76852.78	113269.75	137735.80	155916.83	175945.16	207780.09
0.1	12254.16	39977.92	73803.26	104972.58	122083.43	133511.92	148443.83	166436.02
1.0	12276.92	37317.17	58391.75	82898.72	96246.34	106437.98	119724.11	133005.67
10.0	12489.74	23866.81	29844.92	39109.28	49018.17	60919.70	76016.21	90057.63
100.0	8893.96	12600.03	19211.27	26704.65	35139.88	34698.83	34619.56	34663.61
1000.0	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06
10000.0	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06	1270.06

(b) Grid search table of LASSO regression for terrain dataset

Figure 19: Grid search tables for finding optimal polynomial degree and λ for LASSO regression for each of the data sets.

D Plot of Coefficients up to Degree 8

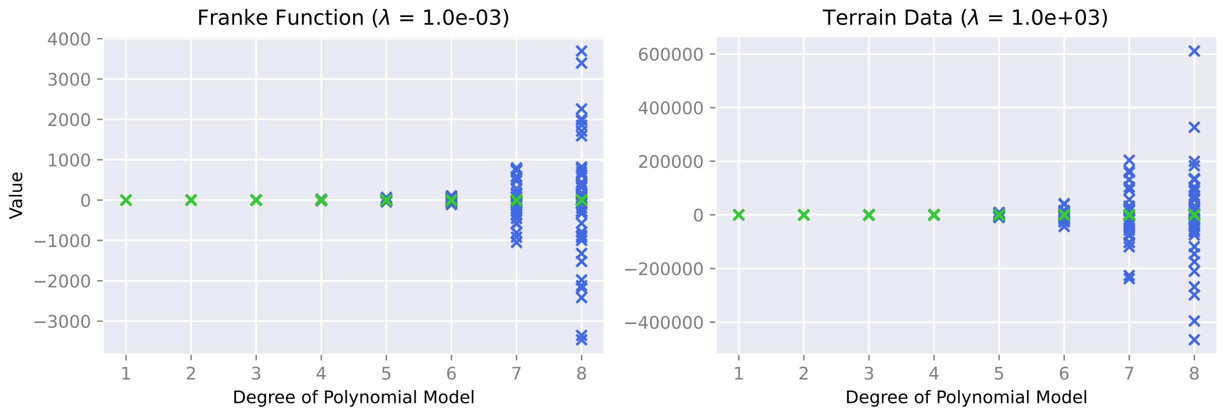
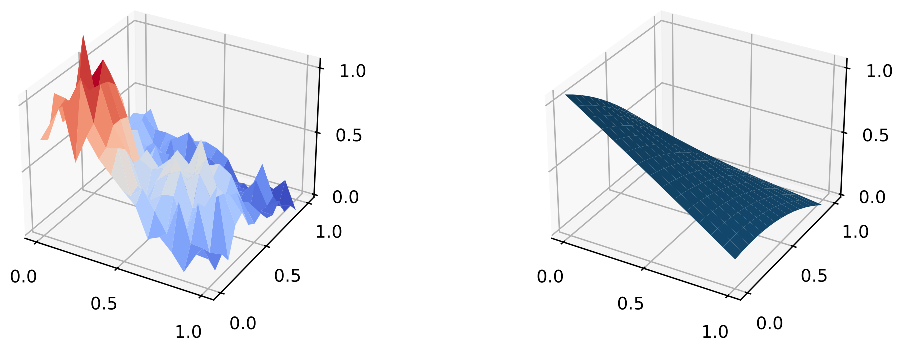
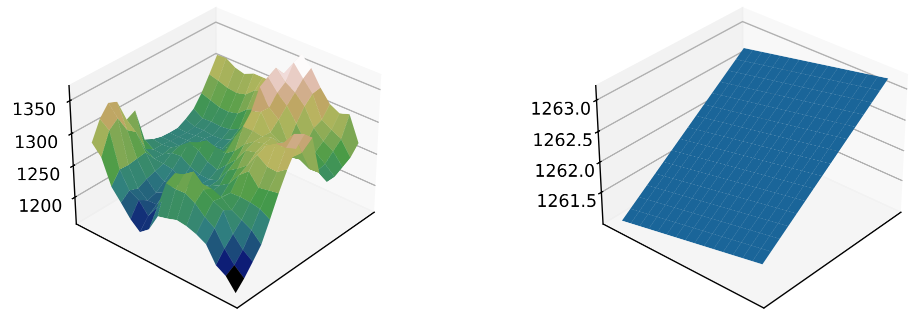


Figure 20: Coefficients of all models by polynomial degree for optimal λ for each dataset up to degree 8.

E Additional Predictions Using Original Parameters



(a) Real Franke function data with noise compared to the predictions of our 2nd degree Ridge model with $\lambda = 1.0$ on a 15×15 grid. These are the hyperparameters indicating the lowest MSE from the grid search (figure 18a).



(b) Real terrain data compared to the predictions of our one degree Ridge model with $\lambda = 10000$ on a 16×16 grid. These are the hyperparameters indicating the lowest MSE from the grid search (figure 18b)

Figure 21: Predictions from ridge models made using parameters determined through the structured experiments in this report.

F Code

These code files include a `BaseModel` class, with subclasses `OrdinaryLeastSquares`, `RidgeRegression` and `LassoRegression`, as well as a `Results` class, and data generating functions. All code for plotting can be found in our GitHub repository at `https://github.com/emmastorberg/FYS-STK4155_Project1`.

`base_model.py`

```python
import numpy as np
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler


class BaseModel:
    def __init__(
            self,
            degree: int,
            multidim: bool = True,
            ) -> None:
        """
        Initialize the BaseModel.

        Args:
            degree (int): The degree of the polynomial for fitting.
            multidim (bool, optional): Whether to use a 2D polynomial. Default is True.
        """
        self.degree = degree
        self.multidim = multidim

        # Number of features
        if multidim:
            self._num_col = int((degree + 1) * (degree + 2) / 2 - 1)
        else:
            self._num_col = degree

        self.scale = False
        self.scaler = None
        self._y_train_mean = None
        self.beta_hat = None

    def create_design_matrix(self, x: np.ndarray | tuple[np.ndarray]) -> np.ndarray:
        """
        Generate a design matrix from the input data for a polynomial fit of the degree
        specified in the constructor.

        Args:
            x (np.ndarray or tuple[np.ndarray, np.ndarray]]):
                - If `self.multidim` is True: A tuple containing two 2D arrays (x1, x2),
                each of shape (n, 1), where n is the number of data points.
                - If `self.multidim` is False: A single 2D array with shape (n, 1).

        Returns:
            np.ndarray: The design matrix corresponding to the polynomial degree specified
    in the constructor.
            The output matrix will be univariate if `self.multidim` is False,
            or multivariate if `self.multidim` is True.
        """
        if self.multidim:
            x1, x2 = x

            n = len(x1)

            X = np.zeros((n, self._num_col))

            for i in range(self.degree):
                # Calculate base index for the current degree
                base_index = int(
                    (i + 1)*(i + 2)/2 - 1
                )

                # Fill the design matrix with x and y raised to appropriate powers
                for j in range(i + 2):
                    X[:, base_index + j] = x1[:,0]**(i + 1 - j) * x2[:,0]**(j)
            return X

        else:
            X = np.zeros((len(x), self.degree))

            for i in range(self.degree):
                X[:, i] = x[:, 0]**(i + 1)
            return X

    def split_test_train(self, X: np.ndarray, y: np.ndarray, test_size: float = 0.2) ->
    tuple:
        """
        Split arrays or matrices into random train and test subsets, with size of test set
        specified in the constructor.

        Args:
            X (np.ndarray): Design matrix.
            y (np.ndarray): Output data.

        Returns:
            tuple[np.ndarray]: X_train, X_test, y_train, y_test
        """
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
        return X_train, X_test, y_train, y_test

    def fit(self, X_train: np.ndarray, y_train: np.ndarray, with_std: bool = True) -> np.
    ndarray:
        """
        Feature scale the ‘X_train‘ by subtracting the mean and dividing by the standard
    deviation (std)
        for each meature.
        Store the feature mean and std in class for use when scaling other matrices.

        Args:
            X_train (np.ndarray): The training matrix for this model.
            y_train (np.ndarray): The training output.

        Returns:
            (np.ndarray): The scaled training matrix, where each feature has been
    standardized to
```

```
            have a mean of 0 and a standard deviation of 1.
        """
        self.scale = True
        self.scaler = StandardScaler(with_std=with_std)
        self.scaler.fit(X_train)
        self._y_train_mean = np.mean(y_train)

    def transform(self, X):
        """
        Scale `X` with same mean (and standard deviation) as `X_train`.

        Args:
            X (np.ndarray): Matrix to be scaled.

        Returns:
            (np.ndarray): Scaled matrix
        """
        return self.scaler.transform(X)

    def train(self, X_train, y_train):
        """Raises error only if a model is incorrectly
        instantiated through the superclass instead of
        the subclasses.

        Raises:
            NotImplementedError: method is not implemented
            because something has been called incorrectly
        """
        raise NotImplementedError

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predict the output for the given input data using the model's learned parameters.
        Adds mean of `y_train` if scaling is used.

        Args:
            X (np.ndarray): Input data for prediction.

        Returns:
            np.ndarray: Predicted output values.
        """
        y_pred = X @ self.beta_hat
        if self.scale:
            y_pred += self._y_train_mean
        return y_pred
```

ordinary_least_squares.py

```
import numpy as np

from .base_model import BaseModel


class OrdinaryLeastSquares(BaseModel):
    def __init__(self, degree, param=None, multidim=True):
        """Initializes an instance of an Ordinary Least Squares model.

        Args:
            degree (int): Degree of polynomial model to create.
            multidim (bool, optional): Whether or not the dataset has multivariate.
    Defaults to True.
            scale (bool, optional): Whether or not to scale the data. Defaults to True.
        """
        super().__init__(degree, multidim)
        self.name = "Ordinary Least Squares"
        self.param_label = None

    def train(self, X_train: np.ndarray, y_train: np.ndarray) -> np.ndarray:
        """Finds optimal coefficients for polynomial model for one degree.

        Args:
            X_train (np.ndarray): Design matrix of training data.
            y_train (np.ndarray): Output values from training data
            param (float, optional): Parameter to give to cost function, but does nothing
    for OLS. Defaults to None.

        Returns:
            np.ndarray: Optimal coefficients of model in an array.
        """
        beta_hat = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
        self.beta_hat = beta_hat
        return beta_hat
```

ridge_regression.py

```python
import numpy as np

from .base_model import BaseModel


class RidgeRegression(BaseModel):
    def __init__(self, degree, param, multidim=True):
        """Initializes an instance of Ridge Regression model.

        Args:
            degree (int): Degree of polynomial model to create.
            lmbda (int | float): Hyper parameter.
            multidim (bool, optional): Whether or not the dataset has multivariate.
    Defaults to True.
            scale (bool, optional): Whether or not to scale the data. Defaults to True.
        """
        super().__init__(degree, multidim)
        self.lmbda = param
        self.name = "Ridge Regression"
        self.param_label = r"$\lambda$"

    def train(self, X_train: np.ndarray, y_train: np.ndarray) -> np.ndarray:
        """Finds optimal coefficients for polynomial model for one degree.

        Args:
            X_train (np.ndarray): Design matrix of training data.
            y_train (np.ndarray): Output values from training data.

        Returns:
            np.ndarray: Optimal coefficients of model in an array.
        """
        beta_hat = (
            np.linalg.inv(X_train.T @ X_train + self.lmbda * np.eye(self._num_col)) @
    X_train.T @ y_train
        )
        self.beta_hat = beta_hat
        return beta_hat
```

lasso_regression.py

```
import numpy as np
from sklearn.linear_model import Lasso

from .base_model import BaseModel


class LassoRegression(BaseModel):
    def __init__(self, degree, param, multidim=True):
        """Initializes an instance of Lasso Regression model.

        Args:
            degree (int): Degree of polynomial model to create.
            lmbda (int | float): Hyper parameter.
            multidim (bool, optional): Whether or not the dataset has multivariate.
    Defaults to True.
            scale (bool, optional): Whether or not to scale the data. Defaults to True.
        """
        super().__init__(degree, multidim)
        self.alpha = param
        self.name = "Lasso Regression"
        self.param_label = r"$\alpha$"

    def train(self, X_train: np.ndarray, y_train: np.ndarray) -> np.ndarray:
        """Finds optimal coefficients for polynomial model for one degree.

        Args:
            X_train (np.ndarray): Design matrix of training data.
            y_train (np.ndarray): Output values from training data.

        Returns:
            np.ndarray: Optimal coefficients of model in an array.
        """
        linreg = Lasso(alpha=self.alpha, fit_intercept=False)
        linreg.fit(X_train, y_train)
        beta_hat = linreg.coef_.reshape(-1, 1)
        self.beta_hat = beta_hat
        return beta_hat
```

results.py

```
from collections.abc import Iterable

import numpy as np
from sklearn.model_selection import KFold
from imageio.v2 import imread
import pandas as pd
from sklearn.utils import resample
from sklearn.metrics import mean_squared_error

from LinearRegression import BaseModel, OrdinaryLeastSquares, RidgeRegression,
    LassoRegression


class Results:
    def __init__(self, model: BaseModel, x, y, maxdegree, params=None, multidim=True, scale
    =True, with_std=True, test_size=0.2):
        self.model = model
        self.x = x
        self.y = y
        self.maxdegree = maxdegree
        if not isinstance(params, Iterable):
            params = [params]
        self.params = params
        self.multidim = multidim
        self.scale = scale
        self.with_std = with_std
        self.test_size = test_size

        self.name = None
        self.param_label = None
        self.degrees = range(1, maxdegree + 1)
        self.X = {}
        self.X_train = {}
        self.X_test = {}
        self.y_train = {}
        self.y_test = {}
        self.beta_hat = {}
        self.mse_train = {}
        self.mse_test = {}
        self.r2_score_train = {}
        self.r2_score_test = {}
        self.y_tilde_train = {}
        self.y_tilde_test = {}

    def train_and_predict_all_models(self) -> None:
        """
        Train, predict and store values across all degrees and hyper parameters.
        """
        for param in self.params:
            self.beta_hat[param] = {}
            self.y_tilde_train[param] = {}
            self.y_tilde_test[param] = {}
            for degree in self.degrees:
                linreg = self.model(degree, param, multidim=self.multidim)
                X = linreg.create_design_matrix(self.x)
                X_train, X_test, y_train, y_test = linreg.split_test_train(
                    X, self.y, test_size=self.test_size
                )
                if self.scale:
                    linreg.fit(X_train, y_train, with_std=self.with_std)
                    X_train = linreg.transform(X_train)
                    X_test = linreg.transform(X_test)
                beta_hat = linreg.train(X_train, y_train)
                y_tilde_train = linreg.predict(X_train)
                y_tilde_test = linreg.predict(X_test)

                self.X[degree] = X
                self.X_train[degree] = X_train
                self.X_test[degree] = X_test
                self.y_train[degree] = y_train
                self.y_test[degree] = y_test
                self.beta_hat[param][degree] = beta_hat
                self.y_tilde_train[param][degree] = y_tilde_train
                self.y_tilde_test[param][degree] = y_tilde_test
        self.param_label = linreg.param_label
        self.name = linreg.name

    @staticmethod
    def mean_squared_error(y: np.ndarray, y_tilde: np.ndarray) -> float:
        """Calculates mean squared error of predicted y_tilde compared to
        some known output values y.

        Args:
            y (np.ndarray): known y values.
            y_tilde (np.ndarray): y values as predicted by model.

        Returns:
            float: calculated mean squared error
        """
        try:
            y = np.array(y)
            y_tilde = np.array(y_tilde)
        except:
            raise TypeError("input must be iterable")
        return np.sum((y - y_tilde)**2) / len(y)

    @staticmethod
    def r2_score(y: np.ndarray, y_tilde: np.ndarray) -> float:
        """Calculates R2 score of predicted y_tilde comapred some known output
        values y.

        Args:
            y (np.ndarray): known y values.
            y_tilde (np.ndarray): y values as predicted by model.

        Returns:
            float: calculated R2 score
        """
        try:
            y = np.array(y)
            y_tilde = np.array(y_tilde)
        except:
            raise TypeError("input must be iterable")
```

```python
        R2_score = 1 - (np.sum((y - y_tilde) ** 2) / np.sum((y - np.mean(y))**2))
        return R2_score

    def calculate_MSE_across_degrees(self) -> None:
        """Fills dictionaries of MSE from training and
        testing data with lists containing of MSE values.
        Keys are the different parameter values, and each
        parameter has MSE calculated for all degrees up
        to self.maxdegree.
        """
        for param in self.params:
            self.mse_train[param] = np.zeros(self.maxdegree)
            self.mse_test[param] = np.zeros(self.maxdegree)
            for degree in self.degrees:
                mse_train = self.mean_squared_error(
                    self.y_train[degree], self.y_tilde_train[param][degree]
                )
                mse_test = self.mean_squared_error(
                    self.y_test[degree], self.y_tilde_test[param][degree]
                )
                self.mse_train[param][degree - 1] = mse_train
                self.mse_test[param][degree - 1] = mse_test

    def calculate_R2_across_degrees(self) -> None:
        """Fills dictionaries of R2 scores from training
        and testing data with lists containing of R2 scores.
        Keys are the different parameter values, and each
        parameter has R2 calculated for all degrees up
        to self.maxdegree.
        """
        for param in self.params:
            self.r2_score_train[param] = np.zeros(self.maxdegree)
            self.r2_score_test[param] = np.zeros(self.maxdegree)
            for degree in range(1, self.maxdegree + 1):
                r2_score_train = self.r2_score(
                    self.y_train[degree], self.y_tilde_train[param][degree]
                )
                r2_score_test = self.r2_score(
                    self.y_test[degree], self.y_tilde_test[param][degree]
                )
                self.r2_score_train[param][degree - 1] = r2_score_train
                self.r2_score_test[param][degree - 1] = r2_score_test

    def kfold_CV(self, k:  int, degree: int, param: float) -> float:
        """Performs K-Fold cross validation resampling.

        Args:
            k (int): number of folds
            degree (int): degree of model to look at
            param (float): parameter to be given to model

        Returns:
            float: mean squared error after cross validation
        """
        X, y = self.X[degree], self.y
        kfold = KFold(n_splits=k)
        scores_kfold = np.zeros(k)
        i = 0

        for train_inds, test_inds in kfold.split(X):
            X_train = X[train_inds]
            y_train = y[train_inds]

            X_test = X[test_inds]
            y_test = y[test_inds]

            linreg = self.model(degree, multidim=self.multidim, param=param)

            if self.scale:
                linreg.fit(X_test, y_test, with_std=self.with_std)
                X_train = linreg.transform(X_train)
                X_test = linreg.transform(X_test)

            linreg.train(X_train, y_train)
            y_pred_test = linreg.predict(X_test)

            MSE_test = self.mean_squared_error(y_test, y_pred_test)
            scores_kfold[i] = MSE_test
            i += 1
        return np.mean(scores_kfold)

    def grid_search(self) -> pd.DataFrame:
        """
        Perform a grid search to evaluate the model's performance
        over a range of hyperparameters and polynomial degrees.

        This method uses k-fold cross-validation to compute the Mean Squared Error (MSE)
        for each combination of hyperparameter and polynomial degree specified in the input
    .

        Args:
        params (list | None): A list of hyperparameter values to be evaluated during the
    grid search.
        If `params` is None, the parameter values provided to the constructor is used.

        Returns:
        (pd.DataFrame): A DataFrame containing the cross-validated MSE for each combination
     of
        hyperparameter and polynomial degree.

        Notes:
        The DataFrame's columns correspond to the hyperparameter values,
        while the index corresponds to the polynomial degrees.
        """
        cv_grid = {}
        for param in self.params:
            cv_grid[param] = {}
            for degree in self.degrees:
                mse_cv = self.kfold_CV(5, degree, param)
                cv_grid[param][degree] = mse_cv
        df = pd.DataFrame(cv_grid)
        return df

    def bootstrap_resampling(self, num_bootstraps: int = 100, param=None) -> tuple[np.
```

```
ndarray]:
    """Performs bias-variance analysis of dataset using bootstrap resampling.

    Args:
        num_bootstraps (int, optional): Number of bootstrap resamplings. Defaults to
100.

    Returns:
        tuple[np.ndarray]: a tuple with arrays for error, bias and variance.
    """
    error = np.zeros(self.maxdegree)
    bias = np.zeros(self.maxdegree)
    variance = np.zeros(self.maxdegree)

    for degree in self.degrees:

        X = self.X[degree]
        linreg = self.model(degree, param, self.multidim)
        X_train, X_test, y_train, y_test = linreg.split_test_train(X, self.y, test_size
=self.test_size)

        if self.scale:
            linreg.fit(X_train, y_train)
            X_train = linreg.transform(X_train)
            X_test = linreg.transform(X_test)

        y_pred = np.zeros((y_test.shape[0], num_bootstraps))

        for i in range(num_bootstraps):
            X_, y_ = resample(X_train, y_train)
            linreg.train(X_, y_)
            y_tilde = linreg.predict(X_test)
            y_pred[:, i] = y_tilde.ravel()

        error[degree - 1] = np.mean(np.mean((y_test - y_pred)**2, axis=1, keepdims=True
))
        bias[degree - 1] = np.mean((y_test - np.mean(y_pred, axis=1, keepdims=True))
**2)
        variance[degree - 1] = np.mean(np.var(y_pred, axis=1, keepdims=True))

    return error, bias, variance

def print_correlation_matrix(self, degree: int | None = None) -> None:
    """
    Print the correlation matrix for the specified polynomial degree.

    Args:
        polynomial_degree (int, optional): The polynomial degree for which to print the
 correlation matrix.

    Returns:
        None
    """
    if degree is None:
        degree = self.maxdegree
    X = self.X[degree]
    Xpd = pd.DataFrame(X)
```

```
        Xpd = Xpd - Xpd.mean()
        correlation_matrix = Xpd.corr()
        formatted_matrix = correlation_matrix.round(2)
        print(formatted_matrix)


def Franke_function(x: np.ndarray, y: np.ndarray, noise: bool = True) -> np.ndarray:
    """Generates output data from Franke function, with optional noise.

    Args:
        x (np.ndarray): input values
        y (np.ndarray): input values
        noise (bool, optional): Boolean deciding whether or not to make noisy data.
    Defaults to True.

    Returns:
        np.ndarray: Output after input data is given to Franke function, and noise is
    possibly applied.
    """
    term1 = 3 / 4 * np.exp(-((9 * x - 2) ** 2) / 4 - ((9 * y - 2) ** 2) / 4)
    term2 = 3 / 4 * np.exp(-((9 * x + 1) ** 2) / 49 - (9 * y + 1) / 10)
    term3 = 1 / 2 * np.exp(-((9 * x - 7) ** 2) / 4 - ((9 * y - 3) ** 2) / 4)
    term4 = -1 / 5 * np.exp(-((9 * x - 4) ** 2) - (9 * y - 7) ** 2)

    Franke = term1 + term2 + term3 + term4

    if noise:
        state = np.random.get_state()
        Franke += np.random.normal(0, 0.1, x.shape)
        np.random.set_state(state)

    return Franke


def generate_data_Franke(n: int, seed: int | None = None, multidim: bool = False, noise:
    bool = True) -> tuple[np.ndarray]:
    """Generates noisy data to be given to our model. Can give multivariate or univariate
    data.

    Args:
        n (int): number of data points
        seed (int or None): set seed for consistency with random noise
        multidim (bool, optional): Whether or not to make the data multivariate. Defaults
    to False.

    Returns:
        tuple[np.ndarray]: Input and output data in a tuple. If multivariate, input data is
     itself
        a tuple of various inputs X1 and X2.
    """
    np.random.seed(seed)
    if multidim:
        x1 = np.linspace(0, 1, n)
        x2 = np.linspace(0, 1, n)
        X1, X2 = np.meshgrid(x1, x2)
        Y = Franke_function(X1, X2, noise)
        x1 = X1.flatten().reshape(-1, 1)
        x2 = X2.flatten().reshape(-1, 1)
        y = Y.flatten().reshape(-1, 1)

        return (x1, x2), y

    else:
        x = np.linspace(-3, 3, n).reshape(-1, 1)
        y = (
            np.exp(-(x**2))
            + 1.5 * np.exp(-((x - 2) ** 2))
            + np.random.normal(0, 0.1, x.shape)
        )
        return x, y


def generate_data_terrain(n: int, start: int, step: int = 1, filename: str = "datasets/
    SRTM_data_Norway_1.tif"):
    """Generates terrain data to be provided to the model.

    Args:
        n (int): The number of data points to generate along each dimension.
        start (int): The starting index in the terrain data from which to extract the
    points.
        step (int): The interval between consecutive data points to collect (i.e., how many
     indices to skip).
        filename (str): The path to the TIFF file containing the terrain data.

    Returns:
        tuple[np.ndarray]: A tuple containing the input data and output data.
                           If multivariate, the input data is itself a tuple of arrays X1
    and X2.
    """
    x1 = np.linspace(0, 1, n)
    x2 = np.linspace(0, 1, n)
    X1, X2 = np.meshgrid(x1, x2)

    terrain = imread(filename)
    Y  = terrain[start : start + (n * step) : step, start : start+ (n * step) : step]

    x1 = X1.flatten().reshape(-1, 1)
    x2 = X2.flatten().reshape(-1, 1)
    y = Y.flatten().reshape(-1, 1)

    return (x1, x2), y
```