Applying Regression and Resampling Techniques to Norwegian Terrain Data with Franke's Function as Test Function

Knut Magnus Aasrud
Philip Karim Niane
Ida Due-Sørensen

October 10, 2020

Abstract

This report studies three linear regression methods - ordinary least squares, Ridge and the Lasso - with the purpose of modelling existing data, assessing the perfomance of said model and predicting new unknown data. The modelled data are a set generated from the analytical "Franke function" along with a real terrain dataset. Our results are mostly satisfactory and the models succeed in predicting new data, the exception being a faulty prediction of the Franke function by our Lasso implementation.

1 Introduction

Modern science, including physics, is hallmarked by the availability of large datasets, making data analysis an important field of study. Machine learning provides us with methods to learn from and make predictions about datasets. Nevertheless, applying the concepts and tools used in machine learning can be difficult with common pitfalls as under- and overfitting and human bias affecting the models. Thus, it is increasingly relevant for physicists to have a thorough understanding of machine learning.

The purpose of this project is to explore a number of different regression methods which is quite popular within the field of machine learning. The methods that will be explored are the ordinary least square method (OLS), Ridge and Lasso regression which will all be used to find unknown parameters when performing interpolation and fitting tasks. The methods will be tested on a two-dimensional function called Franke's function.

After the methods are settled and ready to go, a few different resampling techniques will be combined with the methods to evaluate the effectiveness. The

resampling techniques to be used are the bootstrap method and the cross-validation method. The bias-variance trade off will also be studied in the article.

To finish of the project, the different methods will be tested on digital terrain data which descends from real life landscape.

2 Theory

2.1 Franke's function

Franke's function is a function which is often used to test different regression and interpolation methods. The function has two Gaussian peaks of differing heights, and a smaller dip. It's expression is

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

and we define it for the interval $x, y \in [0, 1]$. An illustration of Franke's function can be seen in figure 1

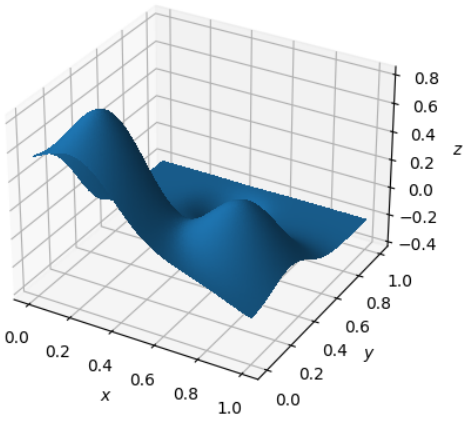


Figure 1: Shape of Franke's function which will be used as a interpolating goal

2.1.1 The Vandermonde matrix applied to Franke's function

When performing the different regression methods, it is important to sort the data into a well designed system for easy access and easier calculations. The data which is to be used during the calculations is sorted into a so called Vandermonde matrix, which is also known as a design matrix. A Vandermonde matrix is a matrix where the rows is built up by geometric progression [1].

Franke's function is a two dimensional function defined by the components x and y. Then it is instinctive to have a design matrix X also built up by x and y components into a $n \times p$ matrix, the following way

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 & y_0 & x_0^2 & x_0y_0 & y_0^2 & \dots & x_0^d y_0^{p-d} \\ 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 & \dots & x_1^d y_1^{p-d} \\ & & & & & & \vdots & \\ 1 & x_{n-1} & y_{n-1} & x_{n-1}^2 & x_{n-1}y_{n-1} & y_{n-1}^2 & \dots & x_{n-1}^d y_{n-1}^{p-d} \end{pmatrix}$$

Where the d is the degree of polynomial. Making the degree higher leads to more complicated equations which naturally gives a more precise regression. Having a design matrix built up by too many terms per polynomial increases the possibility of eventually getting an over fitting model, which will be discussed later in the article.

By having different weights β in front of each term in the design matrix, it is possible to calculate how large each term in the polynomials should be in order to make the best approximation. This is the clue of having a design matrix built up by x and y components, making it possible to approximate the polynomials which will fit the Franke's function best. Then the approximation can be written as the following

$$\hat{\mathbf{y}} = \mathbf{X}\beta + \epsilon \tag{2}$$

Where ϵ is the error. This can be written as

.

$$\hat{\mathbf{y}} = \begin{pmatrix} \epsilon_0 & 1 & \beta_0 x_0 & \beta_1 y_0 & \beta_2 x_0^2 & \beta_3 x_0 y_0 & \beta_4 y_0^2 & \dots & \beta_5 x_0^d y_0^{p-d} \\ \epsilon_1 & 1 & \beta_0 x_1 & \beta_1 y_1 & \beta_2 x_1^2 & \beta_3 x_1 y_1 & \beta_4 y_1^2 & \dots & \beta_5 x_1^d y_1^{p-d} \\ & & & & & & \vdots & & \\ \epsilon_{n-1} & 1 & \beta_0 x_{n-1} & \beta_1 y_{n-1} & \beta_2 x_{n-1}^2 & \beta_3 x_{n-1} y_{n-1} & \beta_4 y_{n-1}^2 & \dots & \beta_5 x_{n-1}^d y_{n-1}^{p-d} \end{pmatrix}.$$

2.2 Linear regression

The goal of a linear regression model is to find these coefficients $\hat{\beta}$, which are best suited for predicting new data via the expression:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta},$$

where $\mathbf{X}$ is the design matrix constructed from the input data (as explained above), and $\hat{\mathbf{y}}$ is the resulting prediction. To achieve this, we define a "cost function" of sorts, that evaluates each coefficients ability to predict the initial training dataset. We then find the $\hat{\beta}$ that minimizes this cost function. More formally put, we want to find

$$\hat{\beta} = \arg\min_{\beta} C(\beta), \tag{3}$$

where $C(\beta)$ is the cost function.

2.2.1 Ordinary least squares regression (L_0)

Ordinary least squares (OLS) regression uses the residual sum of squares (RSS) function as the cost function. Given N datapoints and the predicted output $\mathbf{y}$, it reads

$$\text{RSS}(\beta) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{4}$$

As found in appendix A.2, a cost function like (4) gives the following matrix equation for $\hat{\beta}$

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{5}$$

OLS regression has very low variance, but high bias - as a consequence of the bias-variance tradeoff. This makes OLS regression an "accurate" predictor of its own training data, but susceptible to overfitting, which the following two models are better suited to handle.

2.2.2 Lasso regression (L_1)

Lasso regression expands upon the above cost function by adding a term that penalizes the size of each coefficient. This is done by a factor of λ, as shown here:

$$\hat{\beta} = \arg\min_{\beta} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \tag{6}$$

The first sum is from the RSS function, while the second sum is the imposed penalty. The Lasso decreases bias from the OLS model, by decreasing the size of the coefficients, and thus making the variables more equally weighted.

Lasso regression has no closed form expression for $\hat{\beta}$, which means it must be calculated programatically.

2.2.3 Ridge regression (L_2)

Ridge regression has a penalty corresponding to the coefficient's squared sizes, further decreasing the bias from The Lasso, but obviously also increases variance. It defines $\hat{\beta}$ like this

$$\hat{\beta}^{\text{ridge}} = \arg\min_{\beta} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\},$$

which - by the same procedure as shown in appenix A.2, has this closed form expression

$$\hat{\beta}^{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{7}$$

2.3 Assessment

2.3.1 K-fold cross-validation

When faced with the issue of a small dataset, a good solution to circumvent issues during assessment is to utilize whats called K-fold cross-validation. It involves splitting the dataset in to K equally sized "folds", and using each fold sequentially as the validation set, whilst training on the others. Say, for example, that we set $K = 6$. In this example we would first train our model on the last 5 datasets and evaluate the skill of our model by trying to predict the first. The evaluation is done by calculating the R^2 score, MSE or similar. Thereafter, we use the second fold as the testing set, train on the others, and so forth. The final cross-validation value is the mean of all the calculated statistics.

This is especially a great tool when trying to find the best parameter α present in a model to minimize a certain error/statistic. Cross-validation will then serve as a function we want to minimize to achieve the best result, namely

$$\text{CV}(\hat{f}, \alpha) = \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \hat{f}^{-\kappa(i)}(x_i, \alpha) \right) \tag{8}$$

where L is the error function we want to optimize, y_i are the true values, $\hat{f}^{-k}$ is the fitted function excluding the fold k and $\kappa(i)$ is the function taking in the index of the original dataset and returning the fold containing it.

2.3.2 The Bootstrap

Bootstrapping is another validation method that effectively resamples a smaller dataset. It involves creating B bootstraping samples, each the same size of the original dataset. The data points in each sample are drawn from the original dataset randomly with replacement, which means data points can appear multiple times in the same bootstrapping set. For each bootstrapping set, the desired statistic (in our case, the MSE, bias or variance) is calculated and the mean over all bootstrapping sets is the final bootstrap value of said statistic.

2.4 Bias-variance decomposition

Consider a dataset $\mathcal{L}$ consisting of the data $\mathbf{X}_\mathcal{L} = \{(y_j, \boldsymbol{x}_j) \text{ for } j = 0 \dots n-1\}$. We assume that the true data is generated from the noisy model

$$\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}, \tag{9}$$

where $\boldsymbol{\epsilon}$ is the normal distributed noise with zero mean and a standard deviation σ^2. For the sake of simplicity we consider OLS. In section 2.2.1 defined a approximation to the function $f(\mathbf{x})$ in terms of the parameters $\boldsymbol{\beta}$ and the design matrix $\mathbf{X}$ which embody our model, that is $\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$. The parameters $\boldsymbol{\beta}$ can then be found by optimizing the mean squared error via the so-called cost function

$$C(\mathbf{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\mathbf{y} - \tilde{\mathbf{y}})^2\right]. \tag{10}$$

We want to decompose eq. (10) so that it can be expressed as the sum of bias, variance and the standard deviation. This derivation can be found in appendix A.3, where we obtain the result

$$\mathbb{E}\left[(\mathbf{y} - \tilde{\mathbf{y}})^2\right] = \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\tilde{\mathbf{y}}\right])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}\left[\tilde{\mathbf{y}}\right])^2 + \sigma^2 \tag{11}$$

$$= \text{Bias}[\tilde{\mathbf{y}}]^2 + \sigma^2 + \text{Var}[\tilde{\mathbf{y}}]. \tag{12}$$

The squared bias term is a measure of the best out-of-sample error in the infinite data-limit. The variance, intuitively, measures how much the learning method will fluctuate around its mean. An increasingly complex model, will capture more data points, and the bias will generally decrease as a function of model complexity. However, complexity will generally increase the variance. We say that the model is *underfitted* when a low model complexity gives a high bias and low variance, and that the model is *overfitted* when a high model complexity gives a low bias and high variance. Thus, we need to find the optimal choice of complexity that gives the lowest possible out-of-sample error.

3 Method and implementation

We've placed most of our general code in the root `src` directory. `regression_methods.py` holds all regression classes. The Franke function is defined in `franke.py`. `terraincoding.py` contains most of the parts that has to do with the terrain dataset. The resampling techniques are gathered in a separate script - `assesment.py` - containing functions used for bootstrapping and K-Fold cross-validation. Lastly, `utils.py` is a collection of a variety of different functions that are handy throughout the entire project, including a mean squared error function, R^2 score function, a function for constructing the design matrix, a single value decomposition function and a some other key functions.

Some subdirectories are used for "throwaway" scripts used for producing the actual plots. These are named accordingly.

3.1 Datasets

In this work, we study regression on two different datasets. The first dataset is generated from the known two-dimensional Franke's function [2]. The second dataset is extracted from real terrain data downloaded from EarthExplorer [3].

3.1.1 Franke's function

A dataset, $\mathcal{L}$, is generated with Franke's function (eq. (1)) in accordance with the noisy model (eq. (9)). In other words, the ground-truth is now given by Franke's function with added stochastic noise $\epsilon \sim \mathcal{N}(0, \sigma)$ and n random tuples of $x, y \in [0, 1]$. Our dataset can then be written

$$\mathcal{L} = \{\{(x_1, y_1), (x_2, y_2), \dots (x_n, y_n), \}, \{z(x_1, y_1), z(x_2, y_2), \dots z(x_n, y_n), \}\},$$

for $z(x_i, y_i) = f(x_i, y_i) + \epsilon_i$ for $i \in [1, n]$

3.1.2 Terrain data

The terrain data is saved as a standard GeoTIFF image file in which a grid of pixels compose an image. In our case, each pixel has a grayscale value. To fit the data to the regression models, we interpret each pixel value as a function value of some unknown function $z(x, y) = f(x, y) + \epsilon$, with both $f(x, y)$ and ϵ are unknowns. We parametrize the grid to obtain the (x, y)-tuples needed to construct the design matrix. A a small patch of the terrain data was used in the following analysis due to limited time and RAM-capacities of the computers used.

3.2 The design matrix

To create the design matrix, the x and y arrays are sent into a function together with the desired degree of polynomial. Then the function checks the lengths of the arrays and the degree wanted, and uses this to calculate the dimensions of the design matrix needed to have the matrix on the form like the Vandermonde matrix discussed in the theory section. Then a double for loop is used to fill an identity matrix of the desired dimensions with the preferred x and y combinations to represent the right polynomials.

3.2.1 Input- design matrix

It is natural to start wondering how the x and y data that is sent into the design matrix function is produced. This is done by making arrays of the size $N = 18$ with values between between 0 and 1, and applying frankes function to the data points.

3.3 Regression methods

3.3.1 Ridge

To explain the implementation of the regression methods, it makes more sense to explain the ridge class first. The implemented ridge class starts of by defining different variables like the design matrix X, the dimensions n and p, y, lambdas lmb and the betas.

Then the ridge coefficient vector is calculated the way explained in the theory section (2.2.3), which is easier to understand by seeing the code

```
        # Matrix inversion to find beta
        return np.linalg.inv(self.X.T @ self.X + self.lmb*I) @ self
                                            .X.T @ self.y
```

Where the `linalg` package is being used to invert the matrix. After this the prediction is calculated by taking the product of the design matrix and the calculated betas.

3.3.2 OLS

The OLS method is computed by using the ridge class, which was the reason the ridge class was explained first. The difference between the implementations is that in OLS, $\lambda = 0$. Which makes it quiet straight forward to call the class and accessing the methods.

3.3.3 Lasso

Using the Lasso regression method the implementation was done by using `scikit-learn`, which is a python library with various machine learning algorithms [4]. The method was performed by the following code

```
    def fit_beta(self):
        sklearn_lasso = linear_model.Lasso(alpha=self.lmb)
        sklearn_lasso.fit(self.X, self.y)
        return sklearn_lasso.coef_
```

where the `self.X` is the design matrix and `self.y` is the response.

3.4 Resampling

3.4.1 Cross validation

The implementation of the cross validation is based on the two functions `kfolds` which splits up the data into k number of folds, and `CV` which is performing the cross validation itself.

The design matrix is sent into the CV function as an argument along the output array, the amount of folds wanted and a lambda value if wanted. The arguments is then used to define the folds by sending them into the kfold function. Within the calculations of kfold the data is shuffled, and split into folds of equally lengths by using a loop. After this the excessive data that was not enough to make a fold, is appended the different folds already made by using a loop. The folds is the returned to the main CV function as a list of k tuples where each tuple is being a (X, y) pair.

Each fold is then taken further into the CV function where it chooses an element in the fold and sets the rest of the elements as training data. The current element is set test data. The chosen regression methods is then performed and the predicted and test values is added to the mean squared class to calculate the error.

3.4.2 Bootstrap

The bootstrap function is implemented by having the design matrix, the output array y, N_bootstraps, the method and lambda if needed as arguments. The data is first splitted and scale by using a function `split_and_scale` written in `utils.py` which does exactly that. Splits and scales the data, where $\frac{4}{5}$ of the data is split into training data, and the rest is test data. The bootstrap is performed by using `sklearn`s resampling technique which resamples arrays in a consistent way. Each loop uses the resample function which implements one step of the bootstrapping procedure [4]. The results from the `skrlearn.utils.resample` command is then used in the preferred regression method, which is appended to the bootstrap prediction list. The bootstrap predicted values is returned as an array along the y test values.

4 Results

4.1 Franke's function

4.1.1 Actual prediction of data

We use a dataset of $N = 100$ datapoints generated from the Franke function (including noise from the normal distribution, with $\sigma = 0.1$), and split it into a training and testing set. Figure 2 shows the testing set plotted in 3D, which is what we aim at recreating.

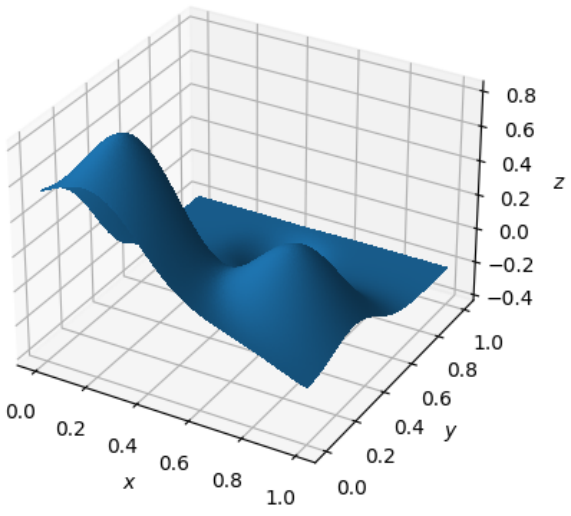
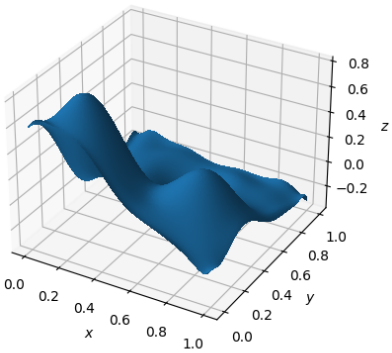


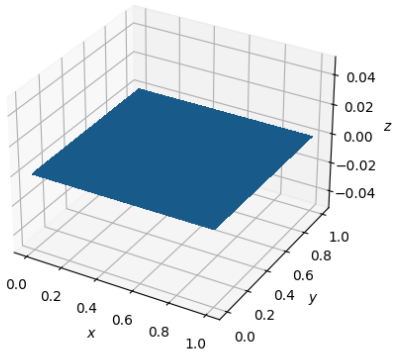
Figure 2: Real valued plot of the testing set of a dataset generated from the Franke function.

Figure 3 shows the predictions of this data, using different regression methods. The design matrix used goes up to degree 12. There's some apparent overfitting in the OLS regressed model. The Ridge model comes quite close, while the Lasso does not give any satisfactory results.

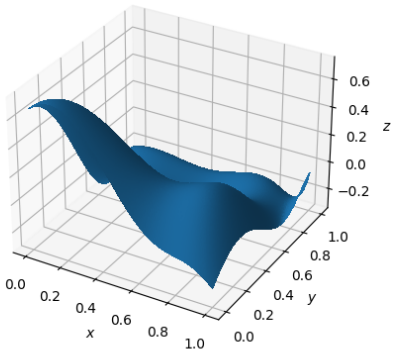
The confidence intervals was also calculated and can be seen in figure 4.



(a) OLS



(b) Lasso



(c) Ridge

Figure 3: Prediction of the data shown in figure 2 using different regression methods.

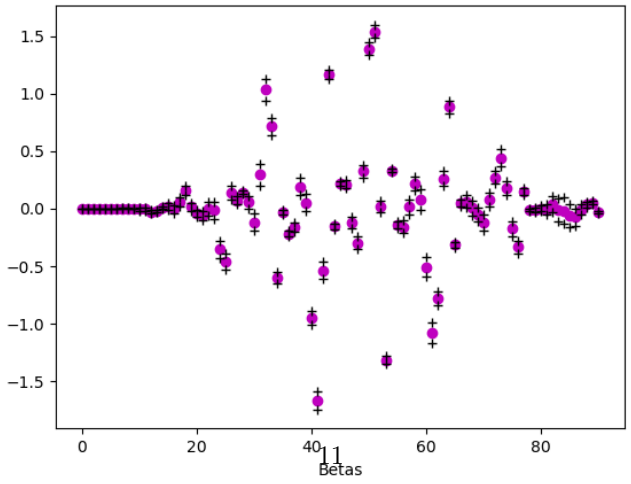


Figure 4: The regression parameters β for the OLS regression method, calculated with a confidence interval of 95%

4.1.2 The bias-variance tradeoff

With a dataset of values produced from Franke's function (number of datapoints $N = 18$, with random normally distributed noise added), we've fitted a model based on the training set and predicted new data based on the testing set. This was done with a design matrix of increasing degrees, from 1 to 9. The following figure 5 is a plot relating the mean squared error of the model to the complexity (or degree).

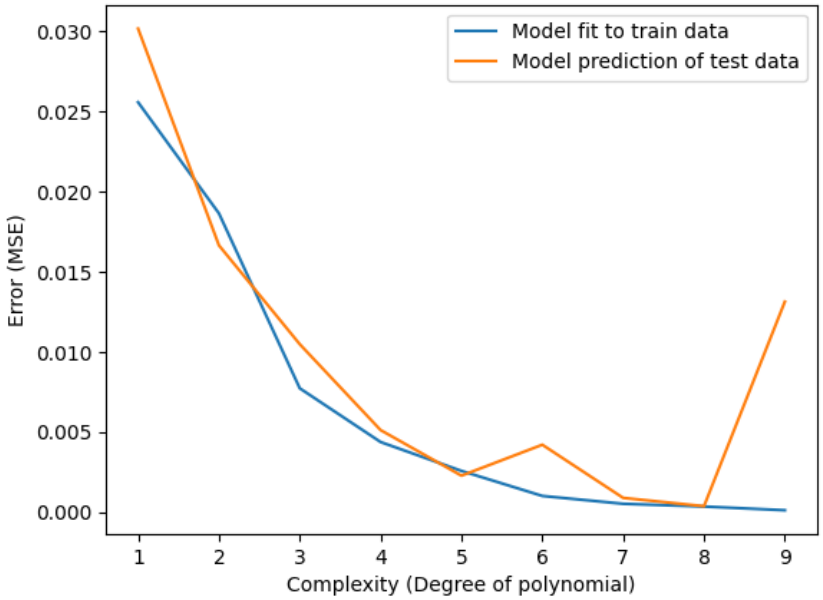


Figure 5: The error of a model based on the train and test set respectively, in relation to the polynomial degree of the model.

Once again, we produce a dataset from the Franke function (number of datapoints $N = 40$, no noise added) and fit the model to design matrices of varying degree, from 1 to 15. Using The Bootstrap, we record the bias and variance (as they are explained in section A.3. Figure 6 shows their relation with increasing complexity/degree of polynomial.

The above results uses bootstrapping as an assessment technique. We have also implemented K-fold cross-validation, and figure 7 shows how their calculated MSE compares

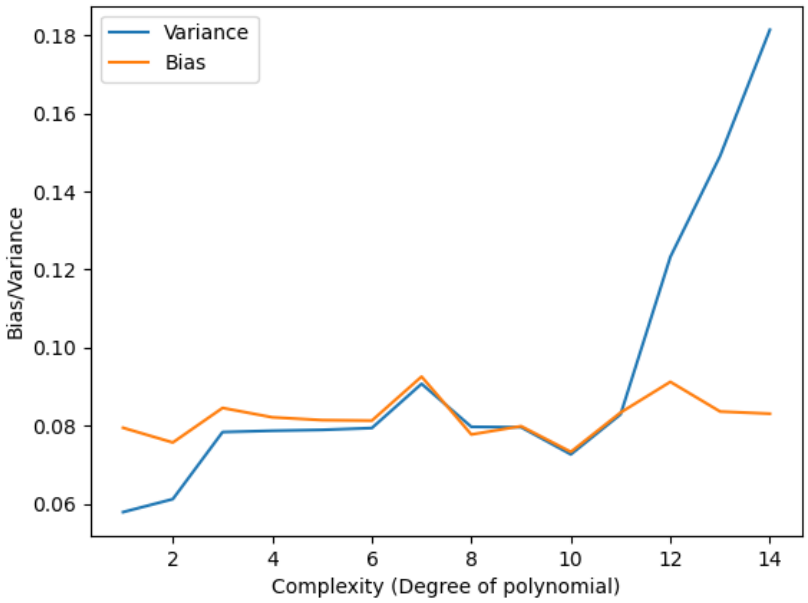


Figure 6: Bias and variance plotted in relation to the polynomial degree of the model.

4.2 Terrain data

The models were tested on terraindata over Norway. The full terrain that the project was based on can be seen in figure 8a. The small patch from the terrain data that was analyzed by using the regression models can be seen in figure 8b. The size of the patch analyzed was 20 datapoints in each direction, making a total of 400 datapoints.

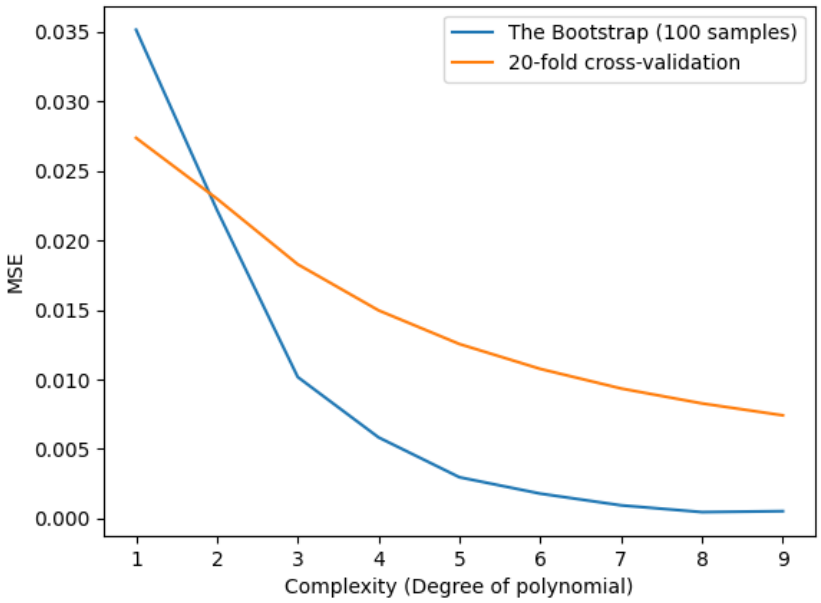
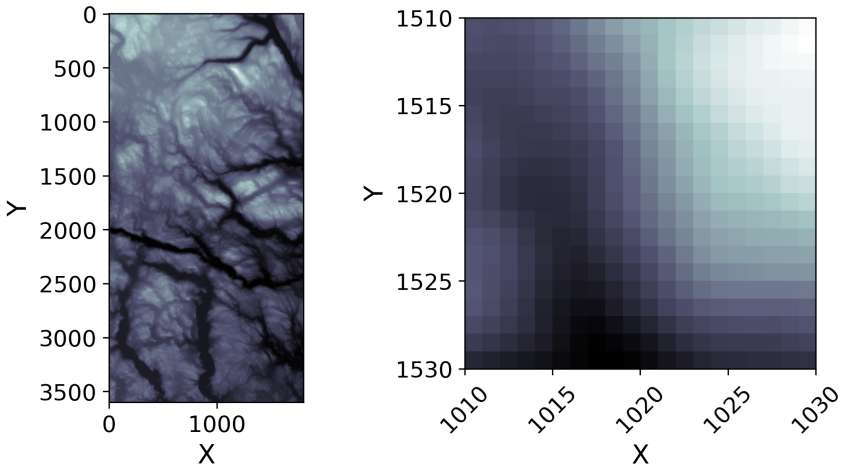


Figure 7: Mean squared error calculated with $K = 20$-fold cross-validation and bootstrapping with $B = 100$.

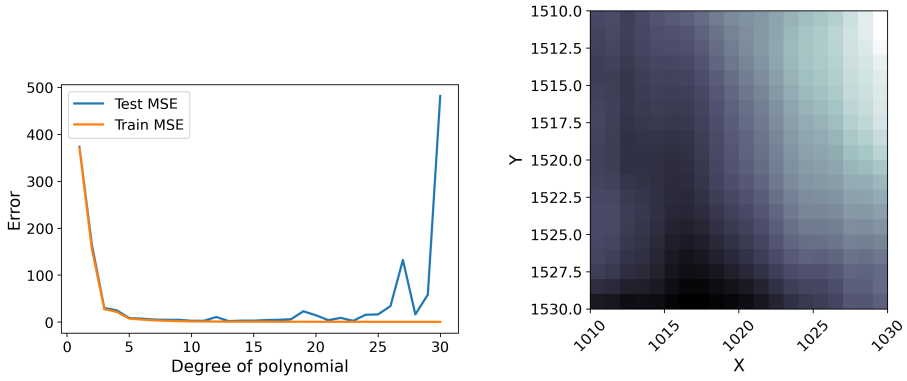


(a) (b)

Figure 8: (a) Terrain data over Norway. (b) Small part of the terrain data consisting of 400 datapoints.

4.2.1 OLS

Ordinary least squares method was performed on the terrain data, the results from Figure 9a shows the test-train error-plot obtained by using cross validation with increasing degree to find the polynomial degree which gives the lowest MSE for linear regression. A minimum is reached at $d = 12$. Figure 9b shows the result of performing OLS with $d = 12$ on the terrain dataset. We obtain MSE and R2-scores of 160.9 and 0.9156, respectively.



(a) (b)

Figure 9: (a) The test and training error as a function of polynomial degree. (b) The result of OLS performed on the terrain data with degree=12

The confidence intervals of the parameters β were also calculated for the OLS regression. The regression parameters β are shown in figure 10.

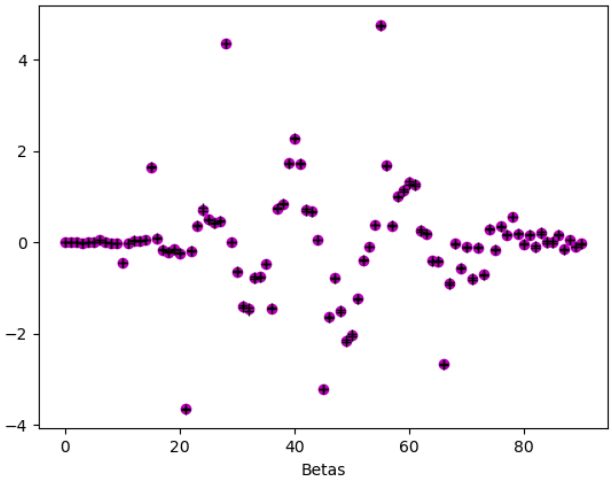
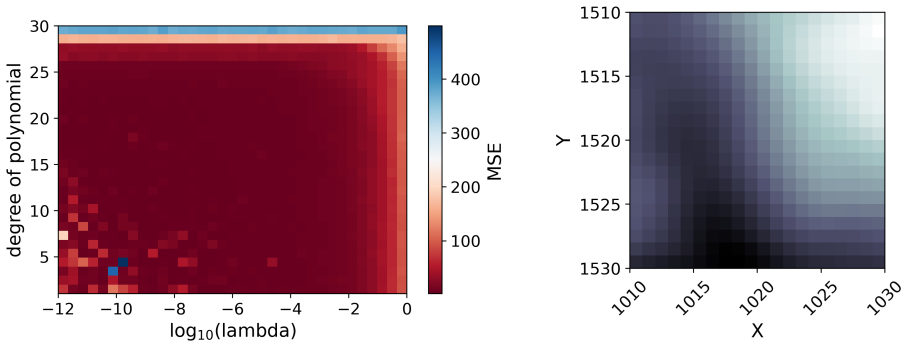


Figure 10: The regression parameters β for the OLS regression method, calculated with a confidence interval of 95%

4.2.2 Ridge regression

Figure 11a depicts the computed regularization path MSE$(d, \log_{10}(\lambda))$ with 5-fold cross validation. The minimum MSE is found to be 2.020 for $d = 19$ and $\lambda = 1.311 \cdot 10^{-10}$. Figure 11b shows the result of performing Ridge regression with $d = 19$ and $\lambda = 1.311 \cdot 10^{-10}$ on the terrain dataset. We obtain MSE and R2-scores of 0.9099 and 0.9995, respectively.



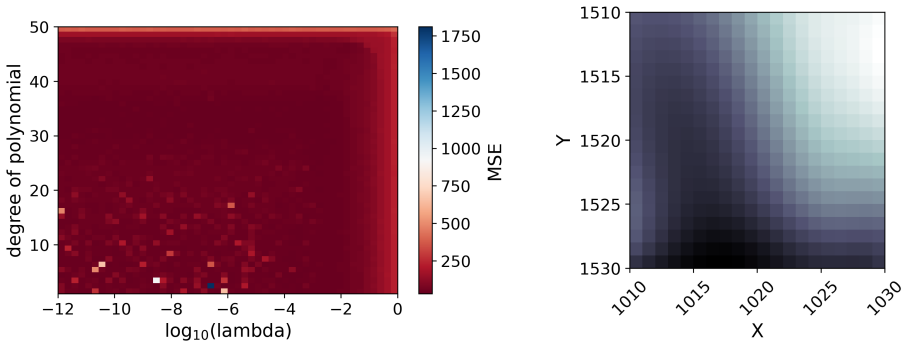
(a) (b)

Figure 11: (a) Color map: Defining the MSE by color. The MSE is calculated for every combination of λ and the degree of polynomial. Degrees up to 30 and λ in the interval $\log_{10}[-12, 0]$ divided into 35 uniformly spaced values, was used to find the best parameters.
(b) The result of Ridge performed on the terrain data with degree=19 and $\lambda = 1.311 \cdot 10^{-10}$.

Ridge method was also performed with smaller spacing of λ in a smaller interval. Letting λ iterate in the interval $\log_{10}[-4, 0]$ with the minimum MSE found to be 5.043 for $d = 22$ and $\lambda = 0.0001 \cdot 10^{-10}$. The obtained MSE and R2-scores are 3.748 and 0.9980, respectively. The colormap and the terrain for this run can be seen in the github folder under `terrain_result/OLS_terrain`.

4.2.3 Lasso regression

Figure 12a depicts the computed regularization path $\text{MSE}(d, \log_{10}(\lambda))$ with 5-fold cross validation. The minimum MSE is found with $d = 21$ and $\lambda = 1.51 \cdot 10^{-3}$. Figure 12b shows the result of performing Ridge regression with $d = 121$ and $\lambda = 1.51 \cdot 10^{-3}$ on the terrain dataset. We obtain MSE and R2-scores of 9.56 and 0.9949, respectively.



(a) (b)

Figure 12: (a) Color map: Defining the MSE by color. The MSE is calculated for every combination of λ and the degree of polynomial. Degrees up to 50 and λ in the interval $\log_{10}[-12, 0]$ divided into 50 uniformly spaced values was used to find the best parameters.
(b) The result of Lasso performed on the terrain data with degree=21 and $\lambda = 1.51 \cdot 10^{-3}$.

5 Discussion

5.1 Methods applied to Franke's function

As shown in figure 5, there's a risk of overfitting by increasing the complexity of the model. An ideal polynomial degree for the design matrix may be 8, judging by this plot. In addition, we experience the bias-variance tradeoff (as shown by figure 6) in our model, albeit not as distinctly as many textbooks showcase it. It's also worth noting that we've found an extreme sensitivity with regards to the number of data points, which clearly brings out the importance of assessing the performance of our regressed model.

Looking at the real world results of our regression modelling, we notice potential in predicting terrain data, as we are clearly able to mimic the actual Franke function values. The resulting plot from our Lasso regression stands out as nonsensical compared to the other two, even though it utilizes `sklearn`'s Lasso implementation. This is probably a fault of ours, and how we've wrapped the module.

5.2 Methods applied to Terrain data

By comparing the statistical values MSE and R^2 for the three models it is possible to determine which regression model has the best prediction to the terrain dataset. Since the MSE is a statistical value of the errors, the less the value of MSE the better. The R^2 score is a measurement of how well the model pre-

dicts, and account for the variance. Closer to 1 the R^2 score is, the better the prediction is.

Having a look at table 1, the MSE for OLS is a bit high having a value of 160.9 with a R^2 score that is doing quite well, 0.9156. The OLS regression parameters in figure 10 also seem to have quite low variance which is supporting the good R^2 score. Looking at the Ridge results it is easy to see that the MSE and R^2 score are extremely good with about the value of 1 for each one. The LASSO regression was also doing quite good with a R^2 score that is about the same as Ridge, but with a bit higher MSE, 9.56. All though the methods are predicting the data quite good, Ridge and LASSO having such low MSE and high R^2 score, makes them belong a couple of levels higher than OLS.

Table 1: The MSE and R^2 values from using different regression methods on the terrain dataset

	OLS	Ridge	$\text{Ridge}_{\lambda \in \log_{10}[-4,0]}$	LASSO
MSE	160.9	0.9099	3.748	9.56
R^2	0.9156	0.9995	0.9980	0.9949

Knowing that OLS is Ridge having λ=0, it is within reason to assume that the results of Ridge and OLS could have been more similar, due to the low $\lambda = 1.311 \cdot 10^{-10}$ used in Ridge. But then remembering the computational expense used to calculate the best d and λ for both Ridge and LASSO it makes sense having Ridge and LASSO that good.

Then computing Ridge, letting the iteration interval for λ be in the space of $\log_{10}[-4, 0]$ with the same amount of iterations, instead of the basic $\log_{10}[-12, 0]$ had quite an inpact on the MSE. Still reading of table 1 the run gave a MSE value of 3.748, making the MSE increase by a factor of 4.12. This shows that it is important to span the values of λ into a large enough space, to be sure that the best λ is within reach of the interval.

LASSO was especially computational expensive. Knowing that there is no analytical solution to LASSO, this opened up for the idea of treating the LASSO method with a bit more iterations than Ridge. LASSO was treated with the privilege of gridding over 50 different values for both λ and d, searching for the lowest MSE, to be sure that the correct parameters were chosen. This ended up with a quite good result. LASSO was a bit more computational heavy than Ridge, and still got a bit worse results, this puts LASSO in a 2nd place.

6 Conclusion

The aim of this work was to gain a further understanding of machine learning concepts and tools by studying three regression methods. After applying the three different regression models to both Franke's function and the real life terrain data, we conclude that they produce satisfactory predictions in both cases, as long as the parameters are chosen carefully. An exception would be the Lasso prediction of the Franke function, which we did not have the time to debug. For the terrain data, we conclude that the best regression method was Ridge regression, closely followed by LASSO regression and OLS being the most inaccurate.

An interesting prospect for the future could be to try these methods on different datasets, to see how they fare in other scenarios. Given more time and computing power, we could also analyze the whole terrain dataset instead of only the small part we've used.

References

[1] Wikipedia contributors, *Vandermonde matrix — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Vandermonde_matrix&oldid=980114781`, [Online; accessed 29-September-2020], 2020.

[2] R. Franke, *A Critical Comparison of Some Methods for Interpolation of Scattered Data*, eng. 1979.

[3] *EarthExplorer*, `https://earthexplorer.usgs.gov/`, Accessed: 2020-09-25.

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

A Appendix

A.1 Source code

All the source code is located in this GitHub repository.

A.2 L_0 regression on matrix form

The cost function we use for OLS regression is the residual sum of squares (RSS) function:

$$\text{RSS}(\beta) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 .$$

Changing into matrix notation, we get

$$\text{RSS}(\beta) = (\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}}) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta),$$

which we can differentiate with respect to β to find the minimum.

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

Assuming full column rank for $\mathbf{X}$, $(\mathbf{X}^T\mathbf{X})$ is thus positive definite (and importantly, invertible). Setting the first derivative to 0, we get

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) = 0$$

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

A.3 Deriving the bias-variance decomposition

The cost fuction is defined as

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

We want to decompose the cost function so that it can be expressed as the sum of bias, variance and the standard deviation.

The first step is to substitute $\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}$ into the cost function, and add and subtract the expectation value of the estimator variable $\mathbb{E}[\tilde{\mathbf{y}}]$. We get

$$\mathbb{E}\left[(\mathbf{f}+\epsilon-\tilde{\mathbf{y}}+\mathbb{E}[\tilde{\mathbf{y}}]-\mathbb{E}[\tilde{\mathbf{y}}])^2\right]=\sum_i \mathbb{E}\left[((\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])+\epsilon+(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}}))^2\right].$$

By writing out this expression and using that $\mathbb{E}[\epsilon]=0$ and $\mathbb{E}[\mathbf{f}]=f$ we get

$$\begin{aligned}\mathbb{E}\left[(\mathbf{f}+\epsilon-\tilde{\mathbf{y}}+\mathbb{E}[\tilde{\mathbf{y}}]-\mathbb{E}[\tilde{\mathbf{y}}])^2\right] &= \mathbb{E}[(\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])]+\mathbb{E}[\boldsymbol{\epsilon}^{\mathbf{2}}]+\mathbb{E}[\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}}]\\ &\quad+\underbrace{2\mathbb{E}[\boldsymbol{\epsilon}(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})]}_{=0}+\underbrace{2\mathbb{E}[\boldsymbol{\epsilon}(\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])]}_{=0}\\ &\quad+\underbrace{\mathbb{E}[(\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})]}_{=0}\end{aligned}$$

Thus, we are left with

$$(\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])^2+\mathbb{E}[\boldsymbol{\epsilon}^{\mathbf{2}}]+\mathbb{E}[(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})^2].$$

The variance of the response variable is given by

$$\begin{aligned}\mathrm{Var}(\mathbf{f}+\boldsymbol{\epsilon}) &= \mathbb{E}[(\mathbf{f}+\boldsymbol{\epsilon})^2]-\mathbb{E}[\mathbf{f}+\boldsymbol{\epsilon}]\\ &= \mathbb{E}[\mathbf{y}]^2+\underbrace{2\mathbb{E}[\mathbf{f}]\mathbb{E}[\boldsymbol{\epsilon}]}_{=0}+\mathbb{E}[\boldsymbol{\epsilon}^2]+(\underbrace{\mathbb{E}[\boldsymbol{\epsilon}]}_{=0}+\mathbb{E}[\mathbf{f}])^2\end{aligned}$$

We see that

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

The bias is defined as $(\mathbf{f}-\mathbb{E}[\tilde{\mathbf{y}}])^2=\mathrm{Bias}(\tilde{\mathbf{y}})$. $\mathbb{E}[(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})^2]$ can be rewritten as

$$\begin{aligned}\mathbb{E}[(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})^2] &= \mathbb{E}[\mathbb{E}[\tilde{\mathbf{y}}]]-2\mathbb{E}[\tilde{\mathbf{y}}]\mathbb{E}[\mathbb{E}[\tilde{\mathbf{y}}]+\mathbb{E}[\tilde{\mathbf{y}}]^2]\\ &= \mathbb{E}[\tilde{\mathbf{y}}^2]-\mathbb{E}[\tilde{\mathbf{y}}]^2\\ &= \mathrm{Var}(\tilde{\mathbf{y}})\end{aligned}$$

By discretizing this equation the expression above can be written as

$$\mathbb{E}[(\mathbb{E}[\tilde{\mathbf{y}}]-\tilde{\mathbf{y}})^2]=\mathrm{Var}(\tilde{\mathbf{y}})=\frac{1}{n}\sum_{i=0}^{n-1}(\tilde{y}_i-\mathbb{E}[\tilde{\mathbf{y}})^2]$$

Finally, we obtain the result

$$\begin{aligned}\mathbb{E}\left[(\mathbf{y}-\tilde{\mathbf{y}})^2\right] &= \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\tilde{\mathbf{y}}\right])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}\left[\tilde{\mathbf{y}}\right])^2 + \sigma^2 \\ &= \mathrm{Bias}[\tilde{\mathbf{y}}]^2 + \sigma^2 + \mathrm{Var}[\tilde{\mathbf{y}}].\end{aligned}$$