Data Analysis and Machine Learning, Project 1

Francesco Pogliano and Marianne Bjerke

October 2019

Department of Physics, University of Oslo, Norway

Abstract

To model the world around us is an incredibly important tool, advancing our everyday life and helping out the industry and academia alike, though to make a model that fits you needs and can give you descent predictions, you need to be critical of the chosen algorithms involved. Even a mathematically sound model might be contextually horrible if its parameters are over- or under fitted to an inappropriate degree. To avoid this, solid statistics and different models such as the LASSO and ridge algorithm is explored with ways of sampling and re-sampling. The OLS and the ridge performed quite well, where the OLS of course gave the lowest errors, while the ridge moved away from the local minimum of the OLS cost function yielding worse R2 score, but improved predicting ability (measured in the R2 score of the test set in k-fold cross-validation). The LASSO method was computationally unstable and required way less data points in order to converge, and when converging giving actually worse predicting power than the ridge analysis. All in all the LASSO method was the worst one.
This was reflected on the digital terrain data analysis, where the OLS and the ridge made a good job in modelling the plot (to the degree a max. 13th degree 3D polynomial can model a fjord-rich region of western norway), while the LASSO made poor predictions and required less input data in order to converge.

Contents

1 Introduction

To learn how to make more advanced models, you need an understanding of the simpler ones. In a naïve state of mind, it is simple to think "the more complex model must be a better model", though that might not necessarily be the case. Bluntly speaking, if you got a linear data set, your model should probably be linear!

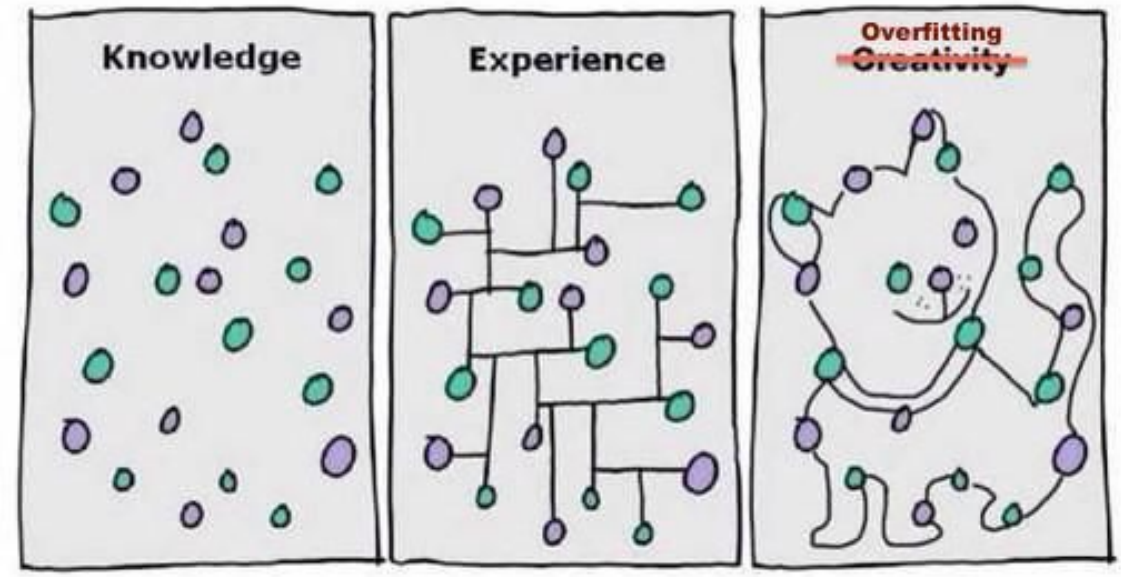

Figure 1: Visual representation of what happens if you over-fit your models.

If a model is not complex enough, it will most certainly not properly represent the data, as most can imagine. Try modelling a sine-wave with a first order polynomial. If the model is too complex and too well fitted to the training data you have, it might pick up every little random variation and end up missing the actual trends, focusing too much on the individual data points. Imagine a squiggly line drawn through almost every single point not predicting a single useful thing.

This is, however, not always so easy to evaluate without some proper statistics and quantitative criteria. By finding a fitting data set, (here, the Franke function is chosen, along with some actual terrain data), and run some different models and fitting algorithms with different complexities and samplings, we will strive to make some compelling arguments concerning these kind of model choices. The strength of this code and project is the variety of models explored and methods used. We will be checking out three different ways of generating the fit, the standard ordinary

least squares (OLS), and its modified counterparts, LASSO and ridge. Sampling is investigated, though with a heavy focus on k-fold batching, dividing the data into a certain number of batches and switching them around, making them take turns on being tested or trained. The variable complexity is directly plotted to see its direct influence on the statistical properties of the model. Lastly, we will use several statistical parameters to evaluate these variables and how the different methods change our results.

We will be introducing the formalism and theory, giving you an overview of the mathematical and theoretical background firstly, and then an introduction to how it has been implemented in the actual code. Here, further details of the structure and logic is given, as well as a practical overview. In chapter four we will show you what has been run and show you results from the software and models, before there are some closing remarks in the conclusion.

2 Formalism

This project will describe the use of different methods in the subject of regressions, these being Ordinary Least Squares (OLS), Ridge and LASSO, together with resampling techniques such as k-fold cross-validation, and discussions around the bias-variance tradeoff. In this section we will introduce these concepts, as they will become central in the rest of the report.

2.1 Linear Regression and OLS

2.1.1 Introduction

Linear regression is a method for analytically deriving a model from a set of data, this model can in turn be used for many purposes, such as fitting, describing a phenomenon or to predict the behaviour of possible, future datapoints [1]. As the name suggests, linear regression is a linear method, and it assumes a linear response between a scalar dependent variable and possibly several inputs. This response is modeled by the parameters $\boldsymbol{\beta}$, and the method allows for finding different interesting statistical properties of the fit, such as the variance, the mean, the mean square error and the R2 score. Introducing the math, the response (our data points) can be written as

$$\boldsymbol{y} = [y_0, y_1, y_2, ..., y_{n-1}]^T, \tag{1}$$

these being n values, associated with the independent variables (our input)

$$\boldsymbol{x} = [x_0, x_1, x_2, ..., x_{n-1}]^T \tag{2}$$

by the parameters $\boldsymbol{\beta}$,

$$\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, ..., \beta_{p-1}]^T, \tag{3}$$

p being the number of parameters and a measure of the complexity of the fit. The fitted model will also take $\boldsymbol{x}$ as input, and give an estimate of $\boldsymbol{y}$, here called $\tilde{\boldsymbol{y}}$,

$$\tilde{\boldsymbol{y}} = [\tilde{y}_0, \tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_{n-1}]^T, \tag{4}$$

the difference between the two being

$$\boldsymbol{\epsilon} = \boldsymbol{y} - \tilde{\boldsymbol{y}} = [\epsilon_0, \epsilon_1, \epsilon_2, ..., \epsilon_{n-1}]^T. \tag{5}$$

The fitted model is a linear combination of some orthogonal functions such as polynomials or terms from a Fourier expansion, with each term modelled by one of our $\boldsymbol{\beta}$ parameters (e.g. if we were to model our dataset with a polynomial up to the 4-th power in the independent variable, we would

need five parameters, $p = 5$, in order to weigh each of the terms). This gives us a system of n equations:

$$\begin{aligned} y_0 &= \beta_0 x_{0,0} + \beta_1 x_{0,1} + \beta_2 x_{0,2} + ... + \beta_p x_{0,p-1} + \epsilon_0 \\ y_1 &= \beta_0 x_{1,0} + \beta_1 x_{1,1} + \beta_2 x_{1,2} + ... + \beta_p x_{1,p-1} + \epsilon_1 \\ ... &= ... \\ y_{n-1} &= \beta_0 x_{n-1,0} + \beta_1 x_{n-1,1} + \beta_2 x_{n-1,2} + ... + \beta_p x_{n-1,p-1} + \epsilon_{n-1} \end{aligned} \tag{6}$$

which can be rewritten as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{7}$$

where

$$\boldsymbol{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & ... & x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & ... & x_{1,p-1} \\ ... & ... & ... & ... & ... \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & ... & x_{n-1,p-1} \end{bmatrix} \tag{8}$$

is called the *design matrix*. From the above equation, we then have also the following relation for the predicted values of $\tilde{\boldsymbol{y}}$:

$$\tilde{\boldsymbol{y}} = \boldsymbol{y} - \boldsymbol{\epsilon} = \boldsymbol{X}\boldsymbol{\beta}. \tag{9}$$

2.1.2 The cost function, and solving for β

We can find an analytical expression for the parameters $\boldsymbol{\beta}$. We introduce the *cost function*:

$$\begin{aligned} C(\boldsymbol{\beta}) &= \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \frac{1}{n}(\boldsymbol{y} - \tilde{\boldsymbol{y}})^T(\boldsymbol{y} - \tilde{\boldsymbol{y}}) \\ &= \frac{1}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}). \end{aligned} \tag{10}$$

For the OLS method, the cost function corresponds to the *Mean Square Error* (MSE), a statistic that will turn useful in evaluating how close or far off a model is to the data set. Later, when we will refer to the MSE, we will be referring to this statistic:

$$MSE = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2. \tag{11}$$

Another useful statistic in this regard is the R2 score, which is nothing more than a normalized version of the MSE:

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2}, \tag{12}$$

where $\bar{y}$ is nothing more than the average of all the y_i. The R^2 score is easier to interpret as it is normalized and is independent of the scale of the model.
The core of the least squares method is then trying to find the parameters $\boldsymbol{\beta}$ which give the lowest value of $C(\boldsymbol{\beta})$. This means finding the zeros of its derivatives in terms of all the β_i terms,

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) = 0, \tag{13}$$

where we can ignore the $2/n$ factor (it cancels out since we are equating the expression to zero). We see that the condition that has to be met is

$$\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) = 0 \tag{14}$$

and by then multiplying it out and rearranging, we arrive to an expression for the β parameters:

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{15}$$

which works as long as the $\boldsymbol{X}^T\boldsymbol{X}$ matrix is invertible.

2.2 Ridge and LASSO

When having many parameters, we encounter the danger of overfitting: the phenomenon for which the fit is very close to all the data points (thus having a very small variance) but becomes so specific that it becomes terrible at predicting future values. At the same time, having too few parameters may make the fit simple, but a too simple fit will also have a big variance meaning, again, loss in predicting power. Ridge and LASSO are two algorithms that may help us tuning between overfitting and underfitting in this regard. This is also related to the discussion on the bias-variance tradeoff. As explained in [1], these are shrinkage methods, and they help us putting a constraint on the size of the parameters, so that our linear regression will prefer fits with smaller - rather than bigger - parameters.

2.2.1 Ridge Regression

The ridge regression simply penalizes the cost function in equation (10) by adding a term proportional to the square of the parameters:

$$C(\boldsymbol{\beta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}, \tag{16}$$

where $\lambda \geq 0$ is a weighing parameter controlling by how much we want to penalize the cost function for having big parameters. This is called a hyperparameter, and although it may seem counterintuitive to introduce a new parameter in order to reduce overfitting, it can be shown that its value actually reduces the degrees of freedom of the fit [1]. The ridge calculation leaves also out the calculation of the intercept, which can be calculated separately by taking the average of the data points:

$$\beta_0 = \frac{1}{n}\sum_{i=0}^{n-1} y_i. \tag{17}$$

The design matrix will then also be modified, now having only $p-1$ columns as we peel of the first one. The expression for the other parameters will then be

$$\boldsymbol{\beta}^{ridge} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{18}$$

We see also how the introduction of the parameter makes the expression inside the parenthesis always invertible, thus solving that possible source of problems from the OLS method.

2.2.2 LASSO Regression

The LASSO (Least Absolute Shrinkage and Selection Operator) regression is very similar to the ridge one, the only difference being the fact that we add the 1-norm (instead of the 2-norm) of the $\boldsymbol{\beta}$ vector to the OLS cost function:

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

This small difference has quite some significance. For the first, the solution becomes nonlinear in $\boldsymbol{y}$ and it makes it suddenly much more difficult to find a closed expression for $\boldsymbol{\beta}^{LASSO}$. For the second, the hyperparameter λ has a different effect on $\boldsymbol{\beta}$. Instead of shrinking the parameters asymptotically to 0, it does so almost linearly, with a cutoff at 0.

2.3 Resampling

When evaluating a fit, we are mostly interested in knowing its predicting power. Naively, we might want to get new data points in order to test how well they match with our model, but these might not be readily available. A solution to this problem is to test the predicting power of the model onto itself, by dividing the data set into two parts: one will be the training set, and the other the testing set. The training set will have the majority of the data points, and will be used to calculate either the statistical properties we might be interested in, or to dreive a fit: this might for example be done with an OLS, ridge or LASSO method. The fit will then be tested by checking the variance or the mean square error of the test set against the predictions made by the training set. This process may be repeated again for different divisions between training and test sets, for then either calculating an average of the statistical value of interest, or choosing the fit with the best match with the test set. Some of the most known resampling methods are the bootstrap, the jackknife and the k-fold cross validation.

2.3.1 k-fold cross-validation

Among the different resampling methods the one we will be focusing on will be the k-fold cross-validation (CV). This method consists in dividing the data set into k subsets, and let each of them play the role of the test, while all the others being part of a common, training set. After each subset has played the role of the test set, we will have k values for our $\boldsymbol{\beta}$ parameters, and as many MSE and R2score values for how well they match with the test set. The one $\boldsymbol{\beta}$ with the best match might be retained. This method has several advantages, such as the fact that the data points from each subset play the role of the test set at least once, and only once. This avoids that some data points might play a more significant role by being randomly picked more times than other data points.

2.4 bias-variance tradeoff

When using a simple OLS method for a linear regression as explained above, we are finding the analytical solution for the fit that gives us the least sum of squares between the predicted value and the data points from the set. This sum can then be again lowered by increasing the complexity of the system, for example increasing the number of parameters. When the goal of the fit is to create a model with high predicting power, we might though encounter the problem of overfitting, meaning that we are starting to model the noise of the data set and thus losing predicting power. When we have a model with too few parameters we say that we have high bias error, meaning that the fit is too simple to manage to catch all the information the data set has to give. We are "biased" in setting all the possible new parameters to zero.
By opening to the possibility for the other parameters to be nonzero we decrease the bias error, but we might overshoot and start modelling the noise. This means we have high variance, i.e. the variance in the test set will be big.
When we modify the OLS method (with e.g. ridge or LASSO), we are moving away from that analytical solution and thus increasing our bias. As stated above, this might be a good idea when we are overfitting and the aim of our fit is predicting future values: a polynomial OLS fit with too many parameters might pass through every data point and thus have zero variance in the training set, but if the points were normally distributed around a straight line, the fit would completely fail to predict future values in the same distribution. This is the core of the bias-variance tradeoff: by adding a bias, we are moving away from the "optimal" OLS solution, but this might actually improve the predicting power of the fit.

A theoretical understanding of this, is to analyze the cost function in equation (10),

$$C(\boldsymbol{X},\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2]. \tag{20}$$

By assuming that the noise is normally distributed around a true function f, we might write

$$\begin{aligned} y_i &= f_i + \epsilon, \\ \mathrm{Var}(\epsilon) &= \sigma^2, \end{aligned}$$

where y_i are our data points.
From this we may then write that

$$\begin{aligned} \mathbb{E}[(\boldsymbol{y}-\tilde{\boldsymbol{y}})^2] =& \mathbb{E}[(\boldsymbol{f}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}})^2] = \mathbb{E}[(\boldsymbol{f}+\boldsymbol{\epsilon}-\tilde{\boldsymbol{y}}+\mathbb{E}[\tilde{\boldsymbol{y}}]-\mathbb{E}[\tilde{\boldsymbol{y}}])^2] \\ =& \mathbb{E}[((\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])-(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}])+\boldsymbol{\epsilon})^2], \end{aligned}$$

which, multiplied out, becomes

$$\begin{aligned} \mathbb{E}[(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])^2 + (\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}])^2 + \boldsymbol{\epsilon}^2 - 2(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}]) \\ + 2(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])\boldsymbol{\epsilon} - 2(\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}])\boldsymbol{\epsilon}]. \end{aligned} \tag{21}$$

By the property of the expectation value that $\mathbb{E}[A+B] = \mathbb{E}[A]+\mathbb{E}[B]$, and $\mathbb{E}[aA] = a\mathbb{E}[A]$ where A, B are stochastic values and a is a constant, we can analyse the sum as if we were taking the expectation value of only the stochastic variables of each term.
First, we consider $\boldsymbol{\epsilon}$. being normally distributed around zero, we have $\mathbb{E}[\boldsymbol{\epsilon}] = 0$, and thus

$$\mathbb{E}[\boldsymbol{\epsilon}^2] = \mathbb{E}[\boldsymbol{\epsilon}^2 - 2\boldsymbol{\epsilon}\mathbb{E}[\boldsymbol{\epsilon}] + \mathbb{E}[\boldsymbol{\epsilon}]^2] = \mathbb{E}[(\boldsymbol{\epsilon}-\mathbb{E}[\boldsymbol{\epsilon}])^2] = \mathrm{Var}[\boldsymbol{\epsilon}] = \sigma^2, \tag{22}$$

where we used the definition of the variance:

$$\mathrm{Var} = \mathbb{E}[(\boldsymbol{X}-\mathbb{E}[\boldsymbol{X}])^2]. \tag{23}$$

Then, we notice that $(\boldsymbol{f}-\mathbb{E}[\tilde{\boldsymbol{y}}])$ is deterministic and not stochastic, meaning we can treat it as a constant and take it out of the evaluation of the expectation value of the respective terms. In addition, we notice that $\mathbb{E}[\tilde{\boldsymbol{y}}-\mathbb{E}[\tilde{\boldsymbol{y}}]]$ must be zero, as the expectation value of a stochastic value and its expectation value is zero. For the last three terms of equation (21) we then have

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

where in the last sum we can factorize the expectation value because $\boldsymbol{\epsilon}$ and $\tilde{\boldsymbol{y}}$ are independent. All the terms multiplied by $\mathbb{E}[\boldsymbol{\epsilon}]$ become zero, and so does the first term for the reasons explained above. All the three last terms of equation (21) disappear, leaving us only with:

$$\begin{aligned} C(\boldsymbol{X},\boldsymbol{\beta}) =& \mathbb{E}[(f_i-\mathbb{E}[\tilde{\boldsymbol{y}}])^2] + \mathbb{E}[(\tilde{y}_i-\mathbb{E}[\tilde{\boldsymbol{y}}])^2] + \mathbb{E}[\epsilon^2] \\ =& \frac{1}{n}\sum_{i=0}^{n-1}(f_i-\mathbb{E}[\tilde{\boldsymbol{y}}])^2 + \frac{1}{n}\sum_{i=0}^{n-1}(\tilde{y}_i-\mathbb{E}[\tilde{\boldsymbol{y}}])^2 + \sigma^2. \end{aligned} \tag{24}$$

The sum above consists of three terms, where each of them can be interpreted in the light of the bias-variance tradeoff.
The first term is the bias:

$$Bias = \frac{1}{n}\sum_{i=0}^{n-1}(f_i-\mathbb{E}[\tilde{\boldsymbol{y}}])^2, \tag{25}$$

meaning the error due to the chosen complexity of the model (or, more correctly, *the lack* of it). The second term,

$$Variance = \frac{1}{n}\sum_{i=0}^{n-1}(\tilde{y}_i - \mathbb{E}[\tilde{\boldsymbol{y}}])^2, \tag{26}$$

is instead the variance of our model, as the sum follows the definition in equation (23).
The last term is then just the (irreducible) variance of the error.

3 Code, implementation and testing.

3.1 Program structure

The software is run from different run*.py-files that imports and calls upon other different instances of the program files. From this heavy dependence on object oriented stucture, different instances of data generation and different methods can easily be used. An intuitive overview of the program logic can be seen in fig. 2

3.2 Example run of a case of franke function with OLS

To run one of the most simple case of a linear regression fit model from a franke-function dataset, the run-file would look like this: (Notice the ".....", these are areas with significant shortening with code which is unimportant for our purpose of understanding implementation here.)

```
import numpy as np
import matplotlib.pyplot as plt
import sys
import os

from data_generation import data_generate
from fit_matrix import fit
from visualization import plot_3d
import statistical_functions as statistics
from sampling_methods import sampling

""" Generating dataset from Franke function with background noise
for standard least square regression w/polynomials up to the
fifth order. Also adding MSE and R^2 score."""

n = 150                 # no. of x and y coordinates
deg = 5                 # degree of polynomial
noise = 0.05            # if zero, no contribution. Otherwise scaling the noise.

# Load dataset and generate Franke function
dataset = data_generate()
dataset.generate_franke(n, noise)

# Normalize the dataset
dataset.normalize_dataset()

# Fit design matrix
fitted_model = fit(dataset)

# Ordinary least square fitting
fitted_model.create_design_matrix(deg)
z_model_norm, beta = fitted_model.fit_design_matrix_numpy()

# Scale back the dataset
rescaled_dataset = dataset.rescale_back(z = z_model_norm)
```

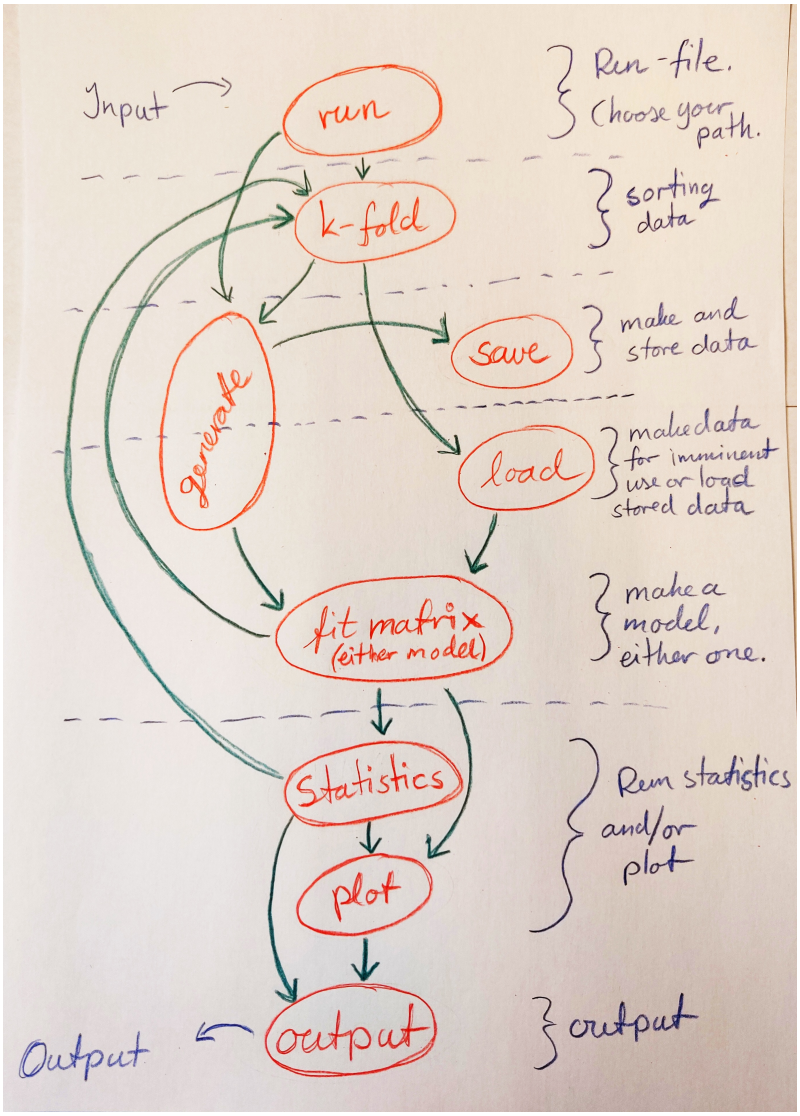

Figure 2: A visualisation of the program logic demonstrated in this report and in the program listed in section 3.2

```
x_model = rescaled_dataset[0]
y_model = rescaled_dataset[1]
z_model = rescaled_dataset[2]

# Generate analytical solution for plotting purposes
analytical = data_generate()
analytical.generate_franke(n, noise=0)

# Statistical evaluation
mse, calc_r2 = statistics.calc_statistics(dataset.z_1d, z_model)
print("Mean square error: ", mse, "\n", "R2 score: ", calc_r2)

# Plot solutions and analytical for comparison
plot_3d(dataset.x_unscaled, dataset.y_unscaled, z_model, analytical.x_mesh,
    analytical.y_mesh, analytical.z_mesh, ["surface", "scatter"])
```

Firstly, some standard python packages are imported, as well as some classes from the other program files (line 6-10). Then, after giving some values for the number of x- and y- coordinates, the dataset it generated and normalized in line 20-25. At this stage, you could also split into training and testing datasets, or instead put everything into the k-fold sampling which will be discussed later. "dataset" is now an instance containing all the data that will be used for fitting the matrix.

On line 28, the instance for generating models are made, and in line 31 a basic polynomial design matrix is generated, and filled in the line below with the newly generated fitted values. After this, z is rescaled and an equivalent analytical data set it generated for plotting purposes, before you can do some statistics to get the MSE, or R2-values is needed.

The results are then plotted with the analytical results in plot-3d(), and the result might look like fig. 3

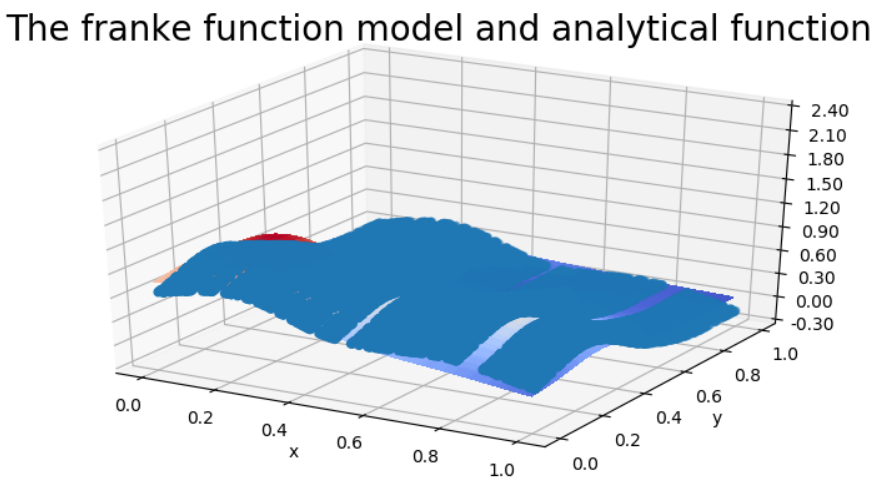

Figure 3: The result of a simple least squares approximation with a polynomial of 5th order.

3.3 Generating the data

The two types of datasets generated in this code is sampling of the analytical franke function and the loading of actual geological mapping data. Both are handled in the file `"data_ generation.py`.

The overview of this data_generate class, has a good amount of the functionality you need for data generating, sorting and normalizing.

There are a significant amount of functions spanning these functionalities as listet below. The headers of the functions of the file look like this:

```
import numpy as np
from sklearn import preprocessing
import sys

from functions import franke_function

class data_generate():
    def __init__(self):

    def generate_franke(self, n, noise ):
        """ Generate franke-data with randomised noise in a n*n random grid. """

    def normalize_dataset(self):
        """ Uses the scikit-learn preprocessing for scaling the data sets for
    computational stability. """

    def rescale_back(self, x=0, y=0, z=0):
        """ After processing, the data must be scaled back to normal by scalers
    inverse_transform for mainly plotting purposes."""
        return rescaled_matrix.T

    def load_terrain_data(self,terrain):
        """ Loads the terrain data for usage. """

    def sort_in_k_batches(self, k, random=True):
        """ Sorts the data into k batches, i.e. prepares the data for k-fold cross
        validation. Recommended numbers are k = 3, 4 or 5. "random" sorts the
        dataset randomly. if random==False, it sorts them statistically"""

    def sort_training_test_kfold(self, i):
        """After sorting the dataset into k batches, pick one of them and this one
        will play the part of the test set, while the rest will end up being
        the training set. the input i should be an integer between 0 and k-1, and
    it
        picks the test set. """

    def fill_array_test_training(self):
        """ Fill the arrays, eg. test_x_1d and x_1d for x, y and z with
        the actual training data according to how the indicies was sorted in
        sort_training_test_kfold."""

    def reload_data(self):
        """ Neat system for automatically make a backup of data sets if you resort.
     """

    def save_data(self):
        """ Saves generated data for later use. """

    def load_data(self):
        """ Loads "pregen_dataset.npz" for previously saved datasets. """
```

This function works as both a generator of the data, but also a manipulator of the data to sort it and scale it to the users wishes, as well as a class to actually hold all the data. As most of the functionalities are described in the descriptions, I will focus on the more technical details.

`Generate_franke` both initialises numpy arrays, finds random x- and y- values as well as add values from the franke function with randomised noise from the `numpy.random.randn` function with an expectation value of 0. The `normalize_dataset` was added to help the computational stability of the fit. It uses the scikit learn `preprocessing.StandardScaler()` for normalising and it also makes it a lot easier to rescale, as is done in `rescale_back`.

`load_terrain_data` is where the real terrain data is handled. This is also very dependent on the normalisation function, of course, but the data is loaded in pretty straight forward from a `.tif` file.

To do some proper sampling and sorting into test and training data, the k-fold-style sorting into batches is implemented here, sorting the data by indexes into k approximately equal sized batches by indices. This only makes lists of indices of data to be used in the different batches, both for flexibility and performance. There are two main statistical and random ways of splitting the data, one in which you loop over all indicies and generate a random number whereas the value of the random number decides which batch the data will belong to. The other one, with a more statistical approach, shuffles all the indices randomly and splits them into the number of batches. Both should perform well within the needs of this style project.

After splitting the indices, the batches can be sorted into training and test data in `sort_training_test_kfold` which sorts all but one batch into training data and the last one into testing. These two sorting functions is then making themselves useful by `fill_array_test_training` which actually fills the arrays with the test and training data given by the indices.

The rest of the file is simply functions for loading and saving data to `.npz`-files.

3.4 Sampling methods

The main and only sampling method we have added for this project is the k-fold, though with this modular of a code you should be able to easily add more complex samplings.

The k-fold was implemented as a class instance of its own that divides data and runs for different training/testing data combinations and saves the different mse, R2 and bias-variance tradeoffs for plotting, printing and evaluating.

The `__init__` of this class has an input of inst which is one of the instances of data from the `generate_data` function discussed above. A shortened version of the code is shown below. This is to be read as an explanation of what is does and kind of psudeocode based on the actual file.

```
.....
import statistical_functions as statistics
from fit_matrix import fit
from functions import franke_function
import copy

class sampling():
    def __init__(self, inst):
        .......
    def kfold_cross_validation(self, k, method, deg=5, lambd=1):
        """Method that implements the k-fold cross-validation algorithm. It takes
    as input the method we want to use. if "least squares" an ordinary OLS will be
    evaulated. If "ridge" then the ridge method will be used, and respectively the
    same for "lasso"."""
        ......

        design_matrix = fit(inst)
        whole_DM = design_matrix.create_design_matrix(deg = deg).copy() #design
    matrix for the whole dataset
```

```
16          whole_z = inst.z_1d.copy() #save the whole output
17
18          for i in range(self.inst.k):
19              #pick the i-th set as test
20              inst.sort_training_test_kfold(i)
21              inst.fill_array_test_training()
22
23              design_matrix.create_design_matrix(deg = deg) #create design matrix for
     the training set, and evaluate
24              .......
25              elif method == "ridge":
26                  z_train, beta_train = design_matrix.fit_design_matrix_ridge(lambd)
27              ......
28              #Find out which values get predicted by the training set
29              X_test = design_matrix.create_design_matrix(x=inst.test_x_1d, y=inst.
    test_y_1d, z=inst.test_z_1d, N=inst.N_testing, deg=deg)
30              z_pred = design_matrix.test_design_matrix(beta_train, X=X_test)
31
32              #Take the real values from the dataset for comparison
33              z_test = inst.test_z_1d
34
35              #Calculate the prediction for the whole dataset
36              whole_z_pred = design_matrix.test_design_matrix(beta_train, X=whole_DM)
37
38              # Statistically evaluate the training set with test and predicted
    solution.
39              mse, calc_r2 = statistics.calc_statistics(z_test, z_pred)
40
41              # Statistically evaluate the training set with itself
42              mse_train, calc_r2_train = statistics.calc_statistics(inst.z_1d,
    z_train)
43
44              # Get the values for the bias and the variance
45              bias, variance = statistics.calc_bias_variance(whole_z, whole_z_pred)
46
47              .....
```

The class instance begins by defining an instance of data generated (or loaded from previously generated and saved data) and then make an instance for the `design_matrix` in line 14. A copy is then made of this instance for the whole dataset and the loop is begun. The number of batches and the sorting into k batches is already done in the `run`-file (see example in github code `run_b.py`). It then chooses the i-th batch as the test set and sorts the rest of the data fr training. It then runs through, generates a model based on training data, tests it on the test batch, does some statistical analysis and returns to choose a different batch for testing, as explained in section 2.3.1. From this, a quite thorough analysis of the performance of the method can be done.

3.5 Design matrices

The different design matrices is within the file `fit_matrix.py` of the class `fit()` which has one initial input of an instance from `generate_data.py` which is the data it will be fitting the model to. To generate any fit, a design matrix must be generated in the function `create_design_matrix` or given as an input in selected functions for testing or evaluating previous models. The design matrix is based on polynomials of degree deg.

The simplest model is the OLS, ordinary least squares, given in function `fit_design_matrix_numpy`. Here, we find the analytical solution to the OLS as described in section 2.1. This is calculated by using several numpy.linalg functions for specific operations from linear algebra. Its output is the model prediction `y_tilde` and the `beta` as described in the theory section mentioned above.

Ridge is also an analytical function, closely related to the OLS, though with an additional term based on an input of lambda. The linear algebra is also very similar, so the numpy.linalg is the machinery calculating this. The LASSO, however, is a bit more of a headache since it happens to turn non-linear and is best solved numerically. Therefore, an already made function from `sklearn.linear_model.Lasso` is used to solve this. Other than this, the function has the same dependencies as the ridge, and other than being more computationally demanding than the ridge, needs no other treatment than the other methods.

Lastly in this file, a simple function to test a design matrix is added, to be able to check a model with whatever design matrix you would like. A shortened version can be seen here as following:

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

import statistical_functions as statistics

class fit():
    def __init__(self, inst):
        self.inst = inst

    def create_design_matrix(self, x=0, y=0, z=0, N=0, deg=17):
        """ Function for creating a design X-matrix with rows [1, x, y, x^2, xy, xy
    ^2 , etc.] Input is x and y mesh or raveled mesh, keyword argument deg is the
    degree of the polynomial you want to fit. """
        ........
        self.l = int((deg + 1)*(deg + 2) / 2)   # Number of elements in beta
        X = np.ones((N, self.l))

        for i in range(1, deg + 1):
            q = int( i * (i + 1) / 2)
            for k in range(i + 1):
                X[:, q + k] = x**(i - k) + y**k

        #Design matrix
        self.X = X
        return X

    def fit_design_matrix_numpy(self):
        """Method that uses the design matrix to find the coefficients beta, and
    thus the prediction y_tilde"""
        X = self.X
        z = self.z

        beta = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(z)
        y_tilde = X @ beta
        return y_tilde, beta

    def fit_design_matrix_ridge(self, lambd):
        """Method that uses the design matrix to find the coefficients beta with
    the ridge method, and thus the prediction y_tilde"""
        X = self.X
        z = self.z

        beta = np.linalg.pinv(X.T.dot(X) + lambd*np.identity(self.l)).dot(X.T).dot(
    z)
        y_tilde = X @ beta
        return y_tilde, beta

    def fit_design_matrix_lasso(self, lambd):
        """The lasso regression algorithm implemented from scikit learn."""
```

```
        lasso = Lasso(alpha = lambd, max_iter = 10e5, tol = 0.01, normalize= (not
    self.inst.normalized), fit_intercept=(not self.inst.normalized))
        lasso.fit(self.X,self.z)
        beta = lasso.coef_
        y_tilde = self.X@beta
        return y_tilde, beta

    def test_design_matrix(self, beta, X = 0):
        """Testing the design matrix with a beta and"""
        ......
        return y_tilde
```

3.6 Statistical evaluations of results

There are three main statistical evaluations of the results that can be given, mainly the MSE (mean square error), the R2 score and the bias-variance tradeoff. All of these are calculated in the file `statistical_functions.py` which also includes nice ways of printing it out to command line. The calculation of the bias-variance-tradeoff is quite straight forward as described in section 2.4, and the MSE in eq. 11, as well as the R2 score in eq. 12.

These calculate the statistical fit of the models and how well the models correspond to the actual data, which is an important tool in evaluating.

4 Analysis and results

In this section we will go through our runs of the code, together with an analysis and discussion on the results of the different parts of the code. This section will loosely follow the structure given in the project description, and will describe first our attempts to analyse the Franke function with added noise using the OLS, ridge and LASSO methods and k-fold cross-validation, and then the application of these methods to real geographical data.

4.1 OLS and introducing the Franke function

The OLS method has been described in the formalism in section 2.1, and it consists in finding the fit that gives the least sum of the distance squared between the real data points and the prediction of the model. This method is then applied to the Franke function, a two dimensional function of exponential on the form

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) + \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \quad (27)$$

The code for this task is implemented using the classes and algorithms introduced in chapter 3. We firstly generate an instance of the `data_generate` class (see chapter 3.3), which we initiate with the Franke function method, add some randomly generated (but still normally distributed) noise and then normalise the data set. For consistency only one data set has been generated for the simulations in this report, in order to give a more clear comparison of the different regression methods. This is generated in `run_generate_dataset.py` and saved to a file, and then loaded in all the different `run` files. This dataset will be used in all the exercises concerning the Franke function, that is all but the last one, where real data will be analysed. In order to reproduce the results of this report, the saved dataset is saved in the repository as `Dataset_report.npz`, and has to be renamed to `pregen_dataset.npz` in order to be fetched by the codes.

	MSE	R^2 score
OLS	0.15227	0.84772
k-fold CV (best)	0.15658	0.84051
k-fold CV (average)	0.16063	0.83933

Table 1: Table showing the values of the MSE and the R^2 score for the OLS method and the k-fold cross validation ($k = 5$) for the same data set. We see how the OLS method gives us the lowest MSE and the highest R^2 score. This agrees with the fact that the OLS solution for the parameters is the analytical minimum of the cost function.

In order to analyse the OLS method we use file `run_a.py`, where we load the data set, we normalize it, we create an instance of the `fit` class (see chapter 3.5) and then use this in order to create a design matrix, its respective prediction, the MSE (equation (11)) and R^2 score (equation (12)). This will then have to be rescaled back in order to be able to plot it against the true Franke function, as shown in fig. 3.
The MSE and R^2 score function are shown in table 1, and are discussed in chapter 4.2.
This is a simple linear regression, and will lay the basis for all the following work.

4.2 Sampling: k-fold cross-validation

Cross-validation (CV), as explained in chapter 2.3.1, is a method to test the ability of a model to predict new data by mocking a set up where some randomly chosen data points from our set play the role of new data (test set) that tests the predictions calculated from the rest of the data points (train set). k-fold CV then simply means dividing the data set into k sets, where each of them plays the role of the test set at least once (while the others play the role of the train test) and then picking the parameters that give the best prediction power.
This method is implemented for the first time in `run_b.py`, where a new class, `sample`, takes care of the sampling and the cross validation. A plot of the result with this method can be seen in fig. **??**, and the MSE and R^2 scores for both the best fitting test set and the whole model are given in table 1.
As we see, the MSE is higher for this model than for the OLS one, and the respective R^2 score is lower. This is to expect, as the OLS is analytically the one with the least variance (it is the analytical minimum of the cost function) and any change from that that will lead to a higher MSE (and, respectively, a lower R^2 score).

4.3 Bias-variance tradeoff

As introduced in the previous section, sometimes we are more interested in a model with other capabilities than the one fitting perfectly the data set (capabilities such as, e.g., predicting power). This motivates us to look more into the relationship between the bias and the variance.
A mathematical definition of the bias and the variance can be found in the theoretical section (equations (25) and (26)), and these can be plotted against the complexity of the model for an OLS method. According to theory, we should expect high bias error and low variance for the test set for lower complexity, and the opposite for higher complexity. In our code, complexity translates to the degree of the polynomial, and in `run_c.py` we run a loop where we fit the same dataset previously analyzed to a polynomial of different degrees, starting from `deg=1` to `deg=20` and cross validate it in five batches. The output is shown in figure 4, where it matches with our prediction, and the figures in 5 recreate figure 2.11 in [1] for the same dataset.

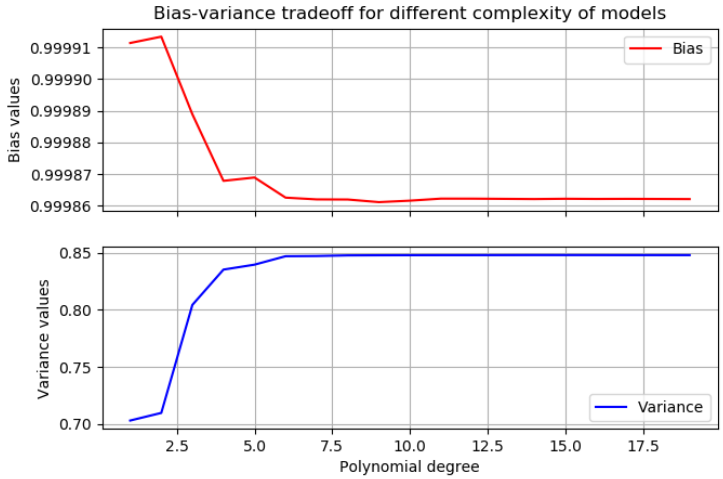

Figure 4: Bias error and variance for the OLS method with cross validation, plotted against the model complexity (in this case the degree of the fitting polynomial).

4.4 Introducing ridge and LASSO for the Franke function

Ridge and LASSO, as explained in the theoretical part, are methods that modify the OLS one by introducing a hyperparamter λ in order to weigh for too large parameters $\boldsymbol{\beta}$.

4.4.1 Ridge

The file `run_d.py` implements the ridge algorithm for different λ's, while for the analysis for different polynomials is done in `run_c.py`. The ridge method, however, is calculated analytically for a given λ. The $\boldsymbol{\beta}$ parameters go asymptotically to 0, as shown in figure 10, as lambda gets bigger. Generally we get a good fit for $\lambda \approx 0.01$, as shown in figure 11. The average MSE error for the test sets in a 5-fold CV for a fifth degree polynomial fit also is slightly better than for normal OLS, giving 0.1572 for ridge, against 0.1608 for OLS run in `run_c.py`, giving thus the method a better predicting power.
The bias and the variance both correlate to λ. The bias increases and the variance decreses for bigger lambdas, as we should expect as the hyperparameter reduces the degrees of freedom of the fit, as shown in figure 7.

4.4.2 LASSO

The file `run_d.py` can also implement the LASSO algorithm as described in **??**, by changing the value of the variable `method` in the header of the file to `method="lasso"`. This cannot be calculated analytically, and we must use the `scikitlearn` package in order to get a numerical evaluation of the model for given λ's. The $\boldsymbol{\beta}$ parameters go almost linearly to 0, as shown in figure 10, as λ gets bigger. Generally we get a satisfying fit for $\lambda \approx 0.01$, as shown in figure 11. The average MSE error for the test sets in a 5-fold CV for fifth degree polynomial is not better than ridge, giving 0.1607 for a lambda of $\lambda = 10^{-5}$ against the previously obtained 0.1572 for ridge, but still slightly better than the 0.1608 for the OLS run in `run_c.py`, although for a way longer computing time. Simulations for even lower lambdas would not converge, and were not considered.

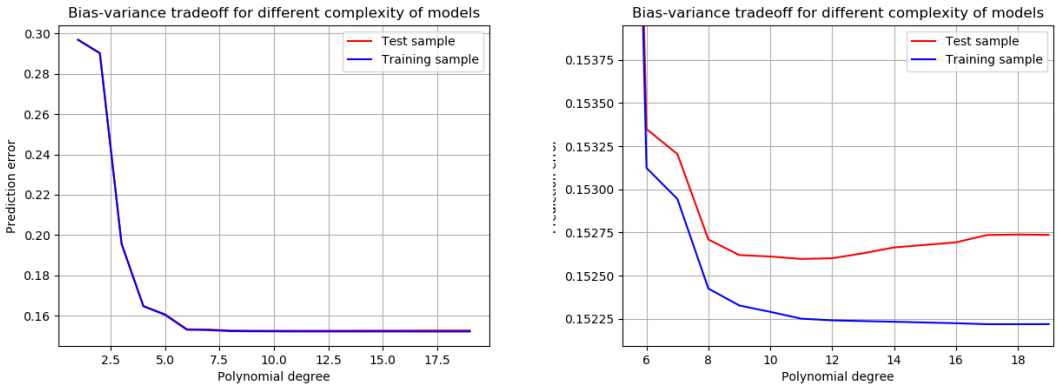

Figure 5: Plot of the MSE for both the training and the test sets: the whole plot on the left, and the zoomed part of the same plot on the right, where we observe how the MSE for the test set increases after a minimum at `deg=10` due to overfitting, while the MSE for the training test keeps diniminishing. Plots drawn using the `run_c.py` code.

4.5 Franke function: wrapping up

for the Franke function we have very good matches for the OLS method when it comes to the MSE and the R2 score. This is in accordance to the theory, as this method gives the analytical best fit to the data set. When it comes to improving the predictability, we tested both the ridge and the LASSO method, which modulate the parameters $\boldsymbol{\beta}$ in different ways: either reducing them asymptotically to zero (ridge) or more directly (LASSO). By testing with a k-fold cross validation method, it turns out that ridge gives better MSE and R2 scores for the test set (which we use in order to mock new data the training set has not been modeled to fit), and the LASSO method, since numerical, has problems with both convergence and running time compared to the other two.

4.6 Introducing real data

Now that we have tested the algorithms on the Franke function, we may try doing the same on real digital terrain data. This is fetched from the website `https://earthexplorer.usgs.gov/` and more specifically from `https://github.com/CompPhysics/MachineLearning/tree/master/doc/Projects/2019/Project1/` where we used the terrain `SRTM_data_Norway_1.tif`.
This is a lot of data, and in order to use it, we had to reduce it by the function `reduce4` in `functions.py`, which takes only the first element of every 2x2 matrix, reducing thus the dimension by 4 (hence the name).
For this task we also generated one map by running `run_generate_terrain_data.py`, and then loaded for every processing and plotting using the `data_generate` class methods `save_data` and `load_data`.
File `_g.py` loads the dataset and then runs the function `rung` from `run_g_evaluate_variable.py`, which takes as input all the regression parameters (`CV` is boolean and indicates whether we want to evaluate a cross validation; `k` is for the k-fold CV; `method` takes as usual `"OLS"`, `"ridge"` or `"lasso"`; `lambda` takes the value of in case we are doing a ridge or LASSO regression; `pol_deg` takes the degree of the polynomial of the fit), and gives as output the quantities of interest such as the model matrix, the MSE, the R2, the bias and the variance.
This can then be looped for the variables of interest, such as different polynomial degrees, or λ's in

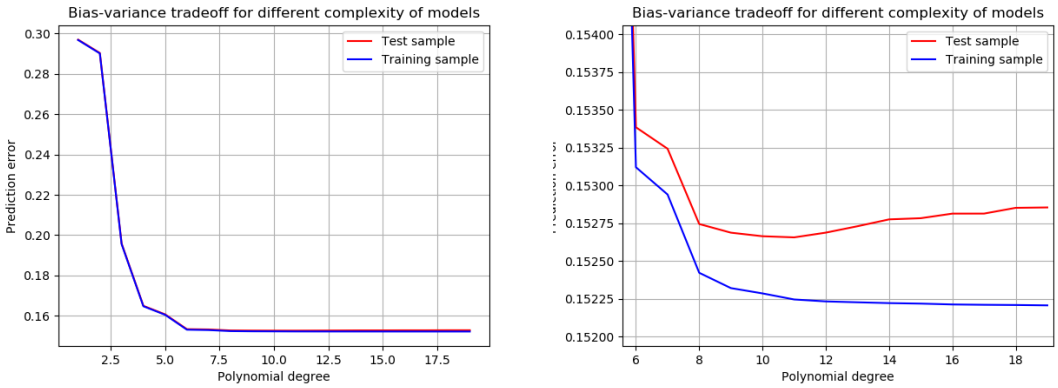

Figure 6: Ridge regression. Plot of the MSE for both the training and the test sets: the whole plot on the left, and the zoomed part of the same plot on the right, where we observe how the MSE for the test set increases after a minimum at around `deg=10` due to overfitting, while the MSE for the training test keeps diniminishing, similarly to the OLS.

case of ridge or LASSO regression. For reference, the map is plotted in fig 12.

4.6.1 Terrain regression analysis, OLS

As a first run, we show an OLS regression for different polynomial degrees, this is shown in fig. 13, and the error analysis in 14. We see how the model becomes more complex and try to match the real data, and how the MSE and the R2 score behave as expected as the complexity of the fit increases.
The next step is to include cross validation, plotted in the same pictures. the plot seem identical, as is maybe to expect for so many datapoints. With CV, we can analyse the bias-variance tradeoff in fig. 15: No surprises here either, as the behavious is exactly what we have seen in with the Franke function.

4.6.2 Terrain regression analysis, Ridge

For ridge, we analyse how the regression changes for different polynomial degrees and hyperparameter values, together with a bias-variance tradeoff analysis.
We start from the ridge, and we plot the cross-validated regression for both varying λ's (deg=5) and varying polynomial degree (lambda=1e-2). This is shown in fig. 16, and the errors in fig. 17. The bias and the variance are plotted in fig. 18. We see how the bias increases for higher lambdas, and decreases for higher complexity, and the opposite for the variance. This is in accordance to the theory, and with our previous simulations with the Franke function.

4.6.3 Terrain regression analysis, Ridge

For LASSO, we do the same analysis as for ridge with varying polynomial degrees and hyperparameter values, together with the bias-variance tradeoff analysis.
LASSO analysis is though very computationally heavy, and for this reason we had to model it on a fourth of the data points, in other words running reduce4 another time.
We start with plotting the cross-validated regression for both varying λ's (deg=5) and varying

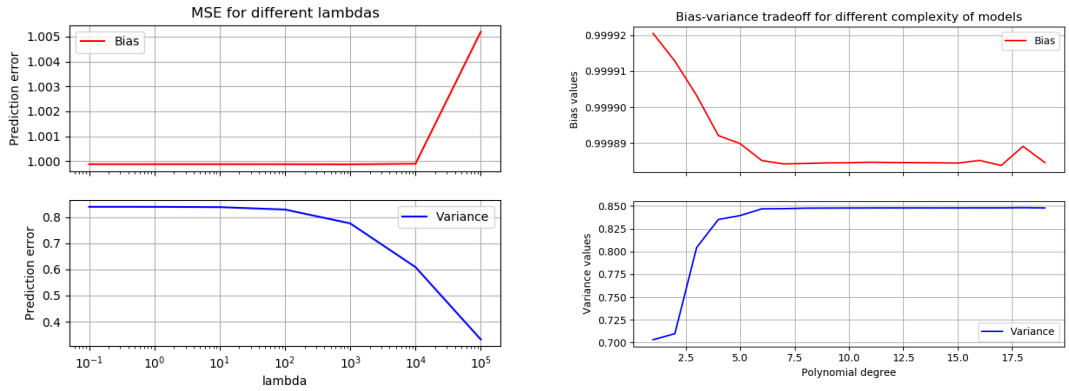

Figure 7: Ridge regression. The bias increases and the variance decreases for bigger lambdas, while as a function of the model complexity it follows the same pattern as for OLS.

polynomial degree (lambda=1e-2). This is shown in fig. 19, and the errors in fig. 20. The bias and the variance are plotted in fig. 21. The most apparent thing to notice is the behaviour of LASSO for varying degrees of lambda. The model seems ok for lower polynomial degrees, but becomes more and more flat as the parameters $\boldsymbol{\beta}$ are drawn to zero, and the last two graphs are completely blank.

5 Conclusion

The different modelling methods and algorithms did perform quite well for the Franke function generated data, but as this data is already based on an analytical original equation, this is not too surprising. Noise was of course included. The OLS and the ridge performed quite well, where the OLS of course gave the lowest errors, while the ridge moved away from the local minimum of the OLS cost function yielding worse R2 score, but improved predicting ability (measured in the R2 score of the test set in k-fold cross-validation). The LASSO method was computationally unstable and required way less data points in order to converge, and when converging giving actually worse predicting power than the ridge analysis. All in all the LASSO method was the worst one.
This was reflected on the digital terrain data analysis, where the OLS and the ridge made a good job in modelling the plot (to the degree a max. 13th degree 3D polynomial can model a fjord-rich region of western norway), while the LASSO made poor predictions and required less input data in order to converge. All in all, we think the ridge was probably the best one of the three. for this task.

References

[1] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. eng. Second Edition. Springer Series in Statistics. New York, NY: Springer New York, 2009. ISBN: 9780387848570.

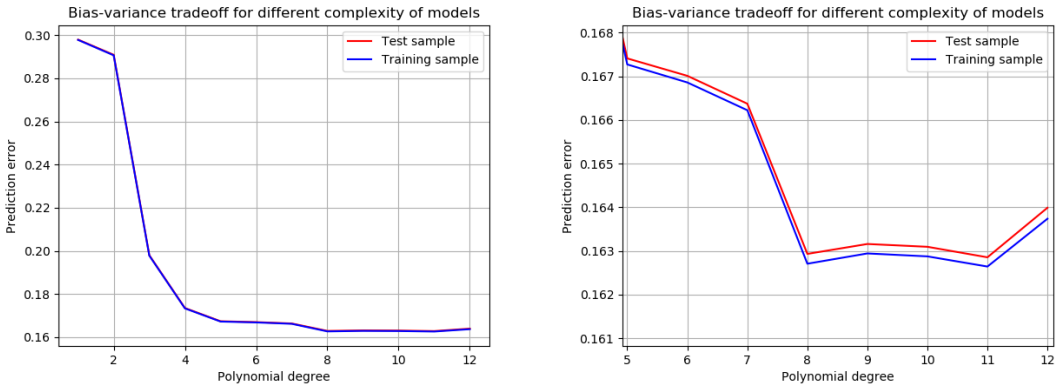

Figure 8: LASSO regression. Plot of the MSE for both the training and the test sets: the whole plot on the left, and the zoomed part of the same plot on the right, where we observe how the MSE for both the training and the test set increases after a minimum at around `deg=8, deg=11` due to overfitting, while the MSE for the training test keeps diniminishing, similarly to the OLS. Lambda used: $\lambda = 0.01$.

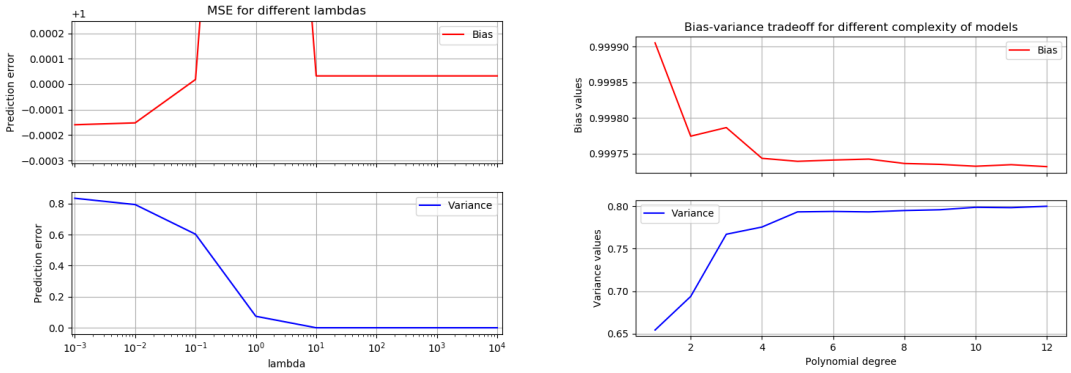

Figure 9: LASSO regression. The bias increases and the variance decreases for bigger lambdas, while as a function of the model complexity it follows the same pattern as for OLS and ridge. The spike at $\lambda = 1$ in the left graph is due to numerical instability around that value.

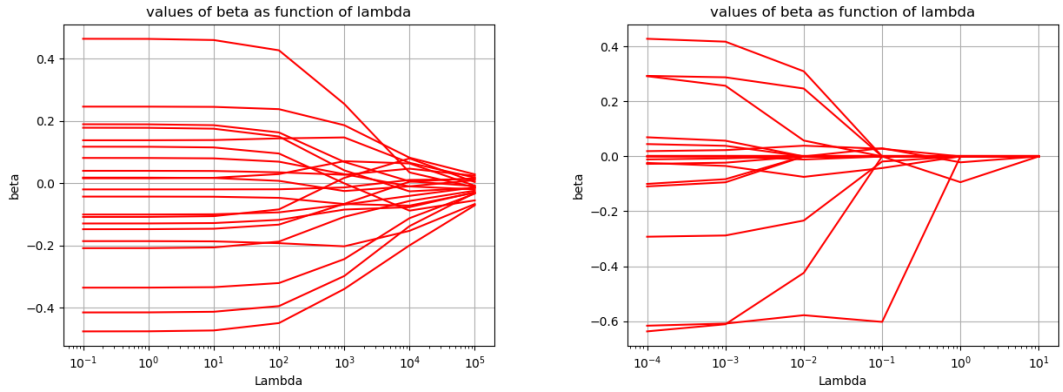

Figure 10: Ridge method (left) VS LASSO method (right). The 21 parameters for a fifth degree polynomial, plotted against various values of λ. We see that for very high values of λ, these go asymptotically towards 0 for ridge while for LASSO they go quickly to towards 0 at orders of magnitude lower than in the ridge algorithm.

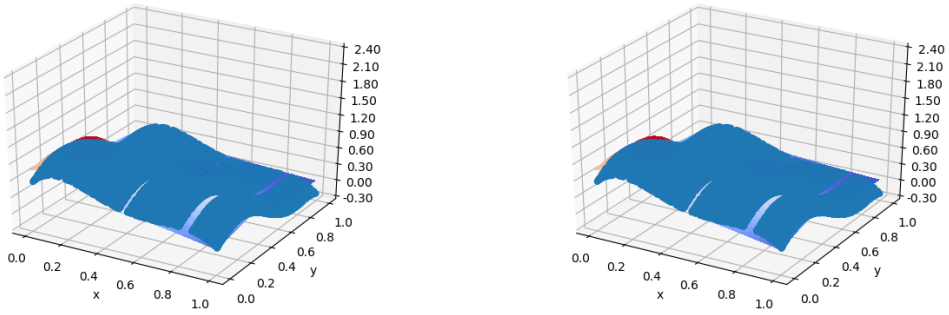

Figure 11: The fit of the ridge (left) and LASSO (right) models against the analytical Franke function, for a $\lambda = 0.01$ and a fifth degree polynomial. The data set is the same as used in the other simulations.

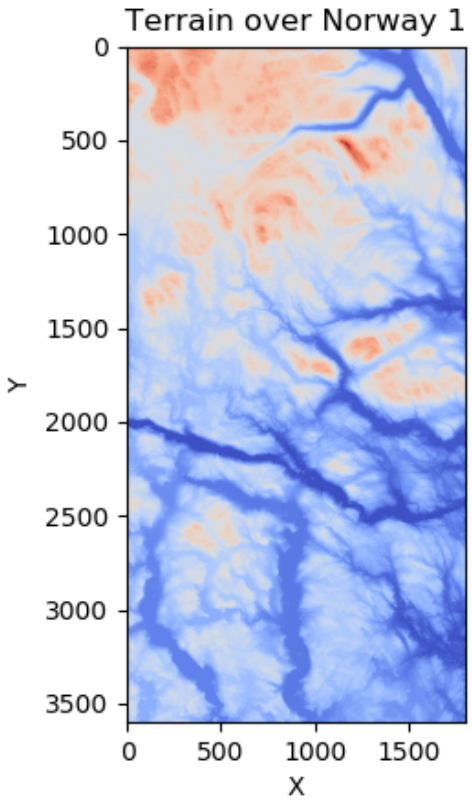

Figure 12: Plot of the digital terrain data.

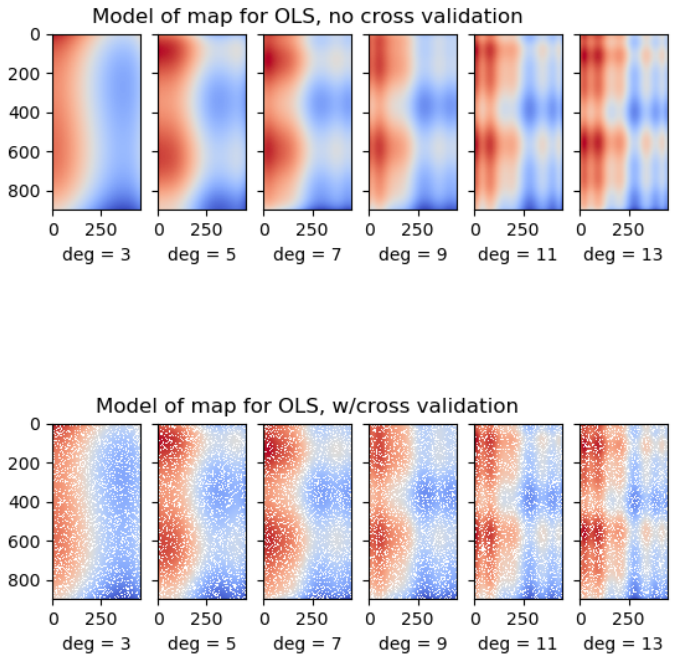

Figure 13: Plot of the fit of the digital terrain data for ordinary least squares, no CV over, with CV under.

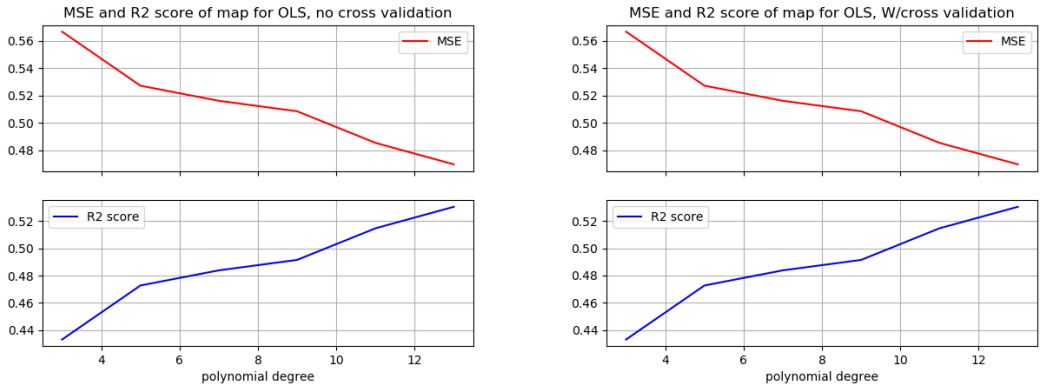

Figure 14: Plot of the fit of the digital terrain data for ordinary least squares, no CV on the left, with CV on the right.

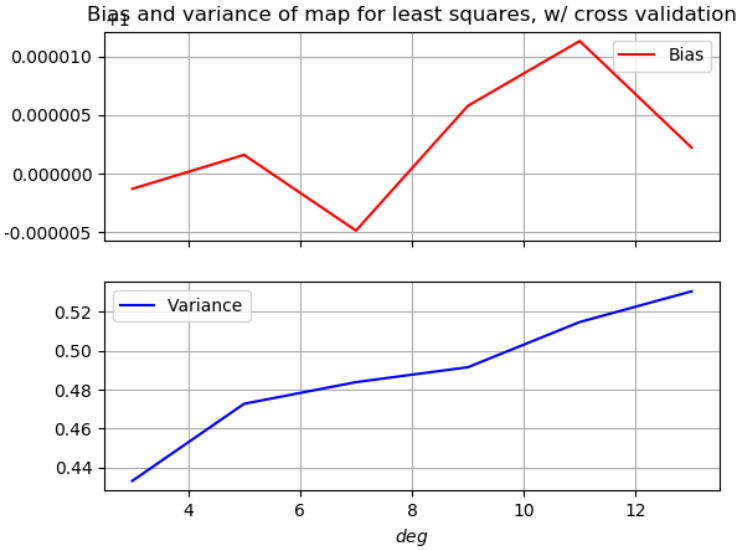

Figure 15: Plot of the bias and variance against the complexity of the model. As expected, the variance increases as the bias decreases for high polynomial fits.

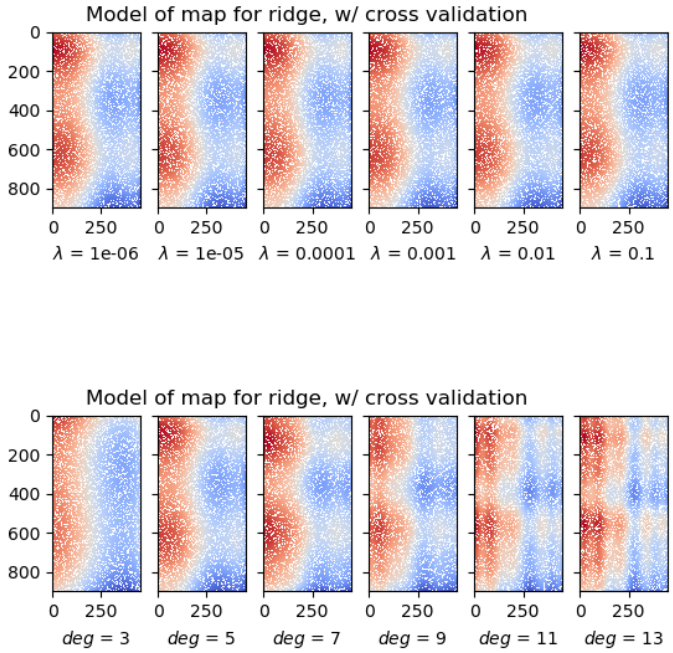

Figure 16: Plot of the fit of the digital terrain data for cross validated ridge regression, with varying lambda and fifth order polynomial over, and with varying polynomial and $\lambda = 0.01$ under.

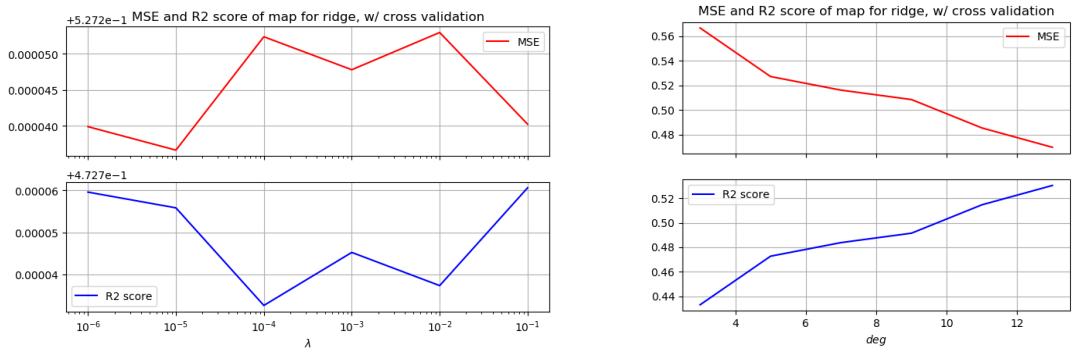

Figure 17: Plot of the errors of the digital terrain data for cross validated ridge regression, with varying lambda and fifth order polynomial (left), and with varying polynomial and $\lambda = 0.01$ (right). We don't see any clear correlation with varying lambdas, but the pattern when varying the polynomial degree follows the usual pattern.

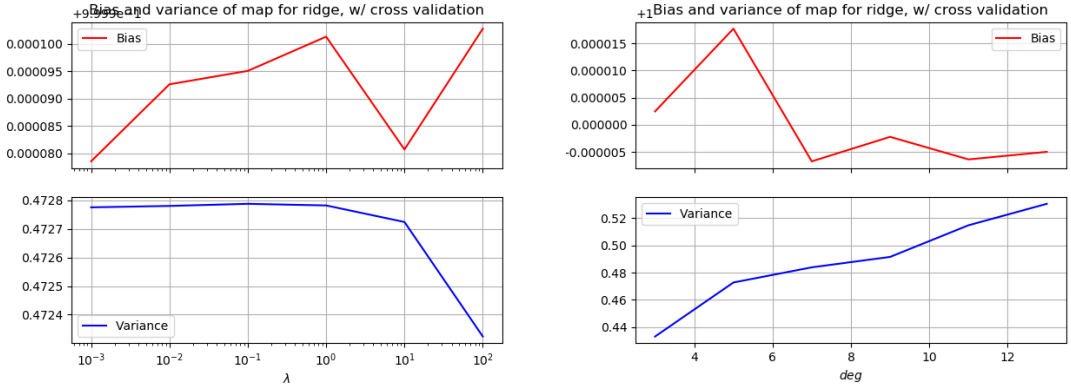

Figure 18: Plot of the bias and variance against different lambdas (left) and the complexity of the model for fixed lambda at 0.01 (right). We see how the bias increases for higher lambdas, and decreases for higher complexity, and the opposite for the variance.

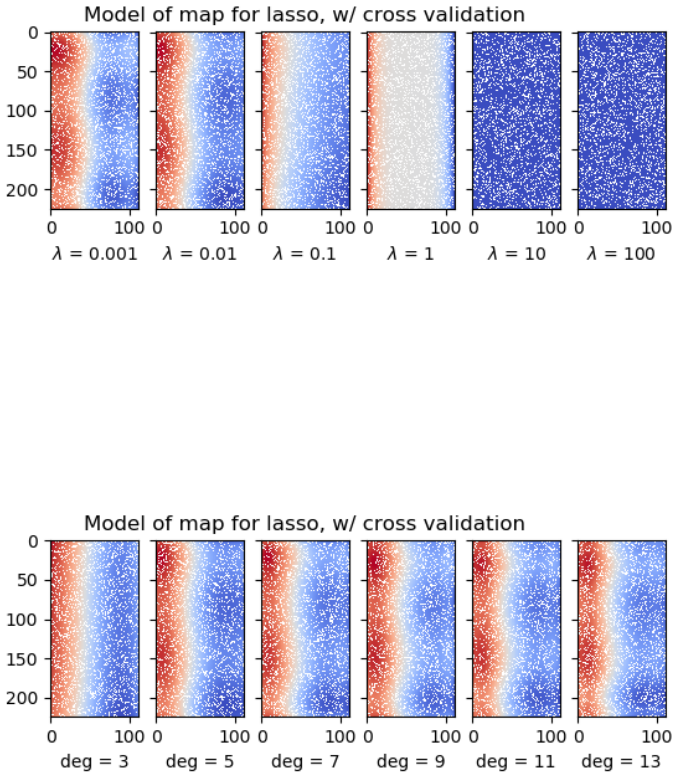

Figure 19: Plot of the fit of the digital terrain data for cross validated LASSO regression, with varying lambda and fifth order polynomial over, and with varying polynomial and $\lambda = 0.01$ under.

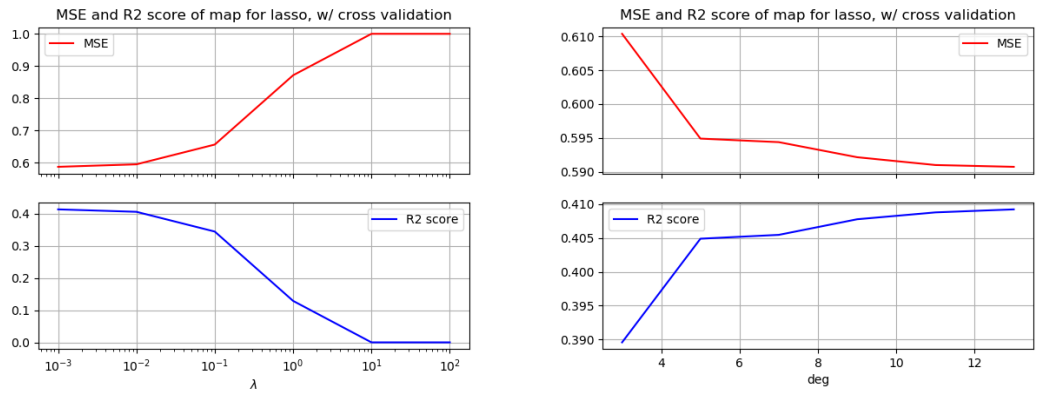

Figure 20: Plot of the errors of the digital terrain data for cross validated LASSO regression, with varying lambda and fifth order polynomial on the left, and with varying polynomial and $\lambda = 0.01$ on the right. Here we see how the model becomes worse and worse for higher lambdas, but becomes better for higher degree fits.

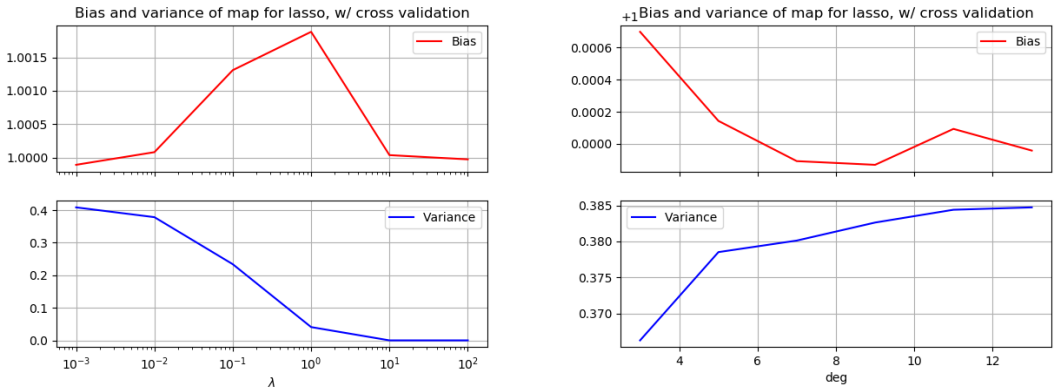

Figure 21: Plot of the bias and variance against different lambdas (left) and the complexity of the model for fixed lambda at 0.01 (right). We see how the bias increases for higher lambdas (if we ignore the region around $\lambda = 10^0$), and decreases for higher complexity, and the opposite for the variance. Although computationally unstable, this is what we should expect from theory.