Project 1 - FYS-STK4155

Elias Roland Udnæs, Jacob Lie og Jonas Thoen Faber
(Dated: September 6, 2021)

Ordinary Least Squares (OLS), Ridge regression and Lasso regression are three regression models used in this project. The data used to train these models are self-made by implementing the Franke function before using real data. This a two dimensional function $f(x, y)$ where $x, y \in [0, 1]$ and are uniformly distributed within this interval. Noise is added to the Franke function and assumed to be normally distributed $\mathcal{N}(0, \sigma)$ where $\sigma = 0.1$. We assumed to have small sets of data which lead us to implement resampling methods such as bootstrapping and k-fold cross validation on all models. The OLS models had the lowest $MSE = 1.735 \cdot 10^{-4}$ for 5-fold cross-validation with complexity degree 32. Given good score in terms of error, we obtained overfitting when the model was used on a test set. It was found that Ridge and Lasso was not optimal for this type of data set. The bias-variance trade-off indicated that the OLS model is optimal at a complexity of 10. With the produced data from the Franke function with little noise, we found that the best model in terms of low MSE was OLS without resampling since the noise is low. When greater noise was added ($\sigma = 0.5$), OLS was no longer efficient and so Lasso is preferred. Since the real data contains little noise, OLS without resampling was used and the model showed results with smoothed curves.

I. INTRODUCTION

In recent years the evolution of computers has brought with it the opportunity of analysing large data sets on a basic laptop. Such data sets can contain information about myriads of different parameters. Finding correlations between these parameters is the essence of machine learning.

An example of such a data set which is popular in Machine Learning is the so-called credit card default data from Taiwan [1]. Collecting data on gender, marital status, age, profession, education, etc. on credit card holders, the aim was to predict whether or not a customer would default their debt.

This example highlights how machine learning can be utilized. We have already seen how it is important in risk prediction, but machine learning is not limited to these kinds of applications. In fact, its range of application spans wide. A completely different task one can encounter in machine learning is for instance handwritten digit recognition [2].

Keeping these two examples in mind, our goal is to expand our knowledge on machine learning. Given a noisy data set $\boldsymbol{Y}$ generated from a function, we want to explore different models that predicts events $\tilde{\boldsymbol{Y}}$ outside of the data set. Finding the best model to predict $\tilde{\boldsymbol{Y}}$, we will apply this model to satellite data [3] to try and predict height contours in real terrain.

II. THEORY

1. *Ordinary least squares*

Ordinary least squares is a method of estimating unknown parameters in regression analysis. Say you wish to fit datapoints to a model. You have an input x (e.g. temperature), and want to create the polynomial:

$$\tilde{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i + \hat{\beta}_2 x_i^2 + \cdots \hat{\beta}_n x_i^n.$$

Where $\tilde{y}$ is the model and $\hat{\beta}$ is the optimal parameter. Ordinary least squares aims to find the best parameter β for a chosen polynomial. An example is represented in figure 1.
This line is usually represented in vector notation [4]:

$$\tilde{\boldsymbol{Y}} = \boldsymbol{X}^T \hat{\beta}.$$

Here $\boldsymbol{X}$ is called the design matrix. It has dimension $(n + 1) \times p$ where p is the number of variables and n is number of samples. It contains all the inputs $(x_i^0, x_i^1, x_i^2, \ldots, x_i^n)$. The point of the method is finding the least squares, therefore we use the definition of mean squared error (MSE):

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

Differentiating w.r.t β and setting it equal to 0 we get:

$$\frac{\partial \tilde{y}}{\partial \beta_j} = \frac{\partial}{\partial \beta_j} \frac{1}{n} \sum_{i=0}^{n-1} \left(y_i - (\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n})\right)^2$$

$$= \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \beta_0 - \beta_1 x_{i,1} - \cdots - \beta_n x_{i,n})$$

$$\cdot(-2) \sum_{i=0}^{n-1} x_{i,j} \rightarrow \frac{\partial \tilde{y}}{\partial \beta_j} = \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}).$$

This is the expression we want to minimize, and rewriting it gives us:

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta}$$

which gives:

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{2}$$

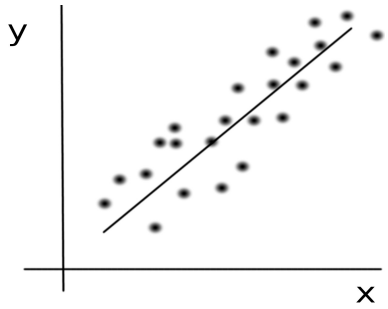

Figure 1. Linear regression analysis on a set of datapoints.

2. Ridge Regression

To solve $\frac{\partial \tilde{y}}{\partial \beta_j} = 0$ in OLS, we assumed that $\boldsymbol{X}^T\boldsymbol{X}$ was invertible. However, that's not always the case. We might encounter a situation where $\det X = 0$, which causes singularity when finding the inverse. The solution is a regularization parameter λ. The expression we now want to optimize is:

$$\frac{\partial}{\partial \beta_j}\Big[(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}^T)\cdot(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\Big] = 0$$

$$-\frac{2}{n}\sum_{i=0}^{n-1} x_{i,j}(y_i - \beta_0 x_{i,0} - \cdots - \beta_n x_{i,n}) + 2\lambda\sum_{j=0}^{p-1}\beta_j = 0$$

Going back to vector notation:

$$\begin{aligned} -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + n\lambda\boldsymbol{\beta} &= 0 \\ \boldsymbol{X}^T\boldsymbol{y} &= (\boldsymbol{X}^T\boldsymbol{X} + n\lambda)\boldsymbol{\beta} \\ \boldsymbol{\beta} &= (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \end{aligned}$$

I is the identity matrix with dimension $p \times p$. Introducing the parameter λ makes sure that there are no 0 elements on the diagonal, and the matrix $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})$ becomes invertible.

λ is called a regularization parameter and reduces the β parameter. This is not intuitive, but a simple example in Hastie [5], shows a case where the design matrix is orthonormal.

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{I}$$

and

$$\boldsymbol{\beta}^{OLS} = \boldsymbol{X}^T\boldsymbol{y}$$

then we get:

$$\beta^{ridge} = (\boldsymbol{I} - \lambda\boldsymbol{I})^{-1}\boldsymbol{\beta}^{OLS} = \frac{\boldsymbol{\beta}^{OLS}}{1+\lambda}.$$

This will help reduce the risk of overfitting our data.

3. Lasso Regression

Lasso regression is *Least Absolute Shrinkage and Selection Parameter*. Where we had $\lambda\boldsymbol{\beta}^T\boldsymbol{\beta}$ in the ridge regression, we now have $\lambda\sqrt{\boldsymbol{\beta}^T\boldsymbol{\beta}}$. This small change has consequences for obtaining $\boldsymbol{\beta}^{lasso}$, because we get a minimization problem without a closed form solution to β^{lasso}. Another consequence is that some parameters might be zeroed. If we were to differentiate the cost function that this new β would produce, we would get:

$$-\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + n\lambda = 0.$$

Solving for β:

$$\beta^{lasso} = \frac{\boldsymbol{X}^T\boldsymbol{y} - n\lambda}{\boldsymbol{X}^T\boldsymbol{X}} = \frac{\beta^{OLS} - n\lambda}{\boldsymbol{X}^T\boldsymbol{X}}.$$

Increasing λ will eventually lead to cancelling of some β, and we'll loose their corresponding features.
Lasso regression's solution is to use an iterative scheme, called cyclical coordinate descent. It iterates and minimizes until it reaches a convergence value, then returns $\hat{\beta}^{lasso}$.

4. Resampling Techniques

Resampling techniques is a method of increasing the statistical significance of your regression analysis. The essence of these methods is to draw smaller samples from the main data set, make a model $\tilde{y}$, find the *Mean Squared Error* and repeat. We often use such techniques if the data set is small. This way we can make the best use of the data we have, and increase the accuracy of our model.

a. Bootstrap In bootstrap we resample with replacement. Say you have a set of data z, which is n long. We then sample n uniformly random numbers from this set. We might sample the same number more than once. After making the sample, we make the prediction model based on your preferred method, compute the statistical scores of your choosing and repeat the process until you have a satisfying answer.
This will reduce the bias of you model. The definition of bias is:

$$Bias = \frac{1}{B}\sum_{i=0}^{B-1}(y_i - E[\tilde{y}])^2, \tag{3}$$

In other words, the difference between the mean of the model of the sampled values and the test data. If $B \to \infty$ you would reach the model that has the best fit, and bias would go towards 0.

b. K Fold Cross Validation There are two different ways one can go about k fold cross validation. Which one you choose, is based mostly on the size of you data set. You can either do the cross validation on your original data set, or on a training set. We've chosen the latter.
First you split your data randomly into unique folds. The most common is choosing 5, 10 or 20 folds. For e.g. 5 folds you split the data into four training sets, and one test set. Then we make one model based on the training data, and find the MSE. This procedure is repeated four more times, where the folds shift on being training and test sets, but the folds can only be the test set once. Therefore, with 5 folds we get five iterations. Finally we take the mean of the MSE's.
This method reduces the variance of your model. We can explain this with an example: Let's say you have ten datapoints. Nine is used for training and one is used for testing. However, this one point is an outlier, a datapoint highly affected by noise. You make the model, and test it. This results in high variance.
We remove this problem when doing the cross validation. Because all folds are used for both training and testing.

III. METHOD

The first thing that we need to do is to get the data that we want to fit to a function using the different methods mentioned above. At first we will create our own data set before using our code on real data. The data set we are to create is based on the Franke function which is defined as:

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

where x and y are equally n-long arrays with uniformly distributed values between zero and one. Using this function alone is not enough to create the data set as it will produce exact values for any x and y. Some noise is added to the Franke function which we will assume to be normal distributed, such that the produced data z is given as:

$$z = f(x,y) + \mathcal{N}(0,\sigma^2)$$

where σ is the standard deviation set to $\sigma = 0.1$ and the mean is equal to zero throughout the project. A seed is used when producing the noise so that we are working with the same data during this project. Note that since we have n datapoints, x and y should consist of the same amount of datapoints. We will then use the notation $\mathbf{x}$ and $\mathbf{y}$ when it is appropriate.

The data we have will then be split into a training set and a test set using the `train_test_split` functionality from the **Sckikit-Learn** module. When the different regression methods is produced, we will use them on the training set and the test set accordingly. As we are trying to fit our methods to the data set by various degree of complexity of the variable $\mathbf{x}$ and $\mathbf{y}$, an important aspect is to make sure not to underfit or overfit the predicted function. To prevent this, we will perform error analysis on our results. Two error functions are used to determine how well the produced function fits the real data. The first function is the mean squared error which we defined from (1) where $\hat{y}$ is the true values and $\hat{\tilde{y}}$ is the predicted values. The goal for this function is to minimize the difference in the predicted values and the true values so that $MSE \to 0$. Another error function is introduced as the R^2 function, also known as the R2 score function:

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

where $\bar{y}$ is the mean of the true values:

$$\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i.$$

here we notice a perfect fit occurs if R^2 is exact equal to one. Then it is no difference between the predicted values and the true values. But having a perfect fit leads very often to overfitting when applying the model on a test set. In this project we will primarily use MSE for the different regression methods using both train data and test data to determine the optimal complexity. This is done by plotting the errors with respect to complexity and observe "by-eye" the optimal fit. To make sure that we do not include extreme values that may affect the training procedure negatively, is to scale our data. This is another built-in function that **Scikit-Learn** offers which we will make use of and is is called `StandardScaler`.

The first method we will study in detail is the ordinary least square, or OLS for short. We remember from (II) that we wanted to find an optimal value for $\boldsymbol{\beta}$. The first thing to do is to set up the design matrix $\mathbf{X}$ which contains the complexity of $\mathbf{x}$ and $\mathbf{y}$. We set complexity of these parameters up to an maximum order of five. When $p = 5$ the dimension of the design matrix is $\mathbf{X} \in \mathbb{R}^{n\times(p+1)(p+2)/2}$. The design matrix is set up by using the binomial theorem so that we have:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & \cdots & x_1^{p-5}y_1^p \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & \cdots & x_2^{p-5}y_2^p \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 1 & x_n & y_n & X_n^2 & x_ny_n & \cdots & x_n^{p-5}y_n^p \end{bmatrix} \tag{4}$$

The creation of the design matrix is coded in a separated Python file as well with other useful function that should be called multiple times (MSE function, R2 function, etc.) during the project. Functions like this can be found in the `functions.py` file. There are in theory four inputs when creating the design matrix. That is $\mathbf{x}$, $\mathbf{y}$, n and p. Since $\mathbf{x}$ and $\mathbf{y}$ will at all time have n data points, only three inputs are needed.

Next is to split the data using Scikit's `train_test_split` function into a train set and a test set. There are indeed no indication of how much of the data that should go to training or testing. We will assume that 0.7/0.3 of the data that goes to training/testing is sufficient enough to produce good models. Each train and test set will call their own design matrix before it is scaled right after. When scaling the design matrix, the `StandardScaler` functionality from **Scikit-Learn** has the drawback of filling the first column to zero but it should in fact be ones which is corrected immediately after scaling. With the trained design matrix, we are ready to calculate $\boldsymbol{\beta}$ using equation (2). This equation is also found in the `functions.py` file as it will be used multiple times. We will use the linear algebra package from the **Numpy** module to calculate $\boldsymbol{\beta}$ in an efficient way. This function requires two inputs which is the design matrix and corresponding data that is needed to predict β. In our case, the calculation will look like:

$$\boldsymbol{\beta} = ({\boldsymbol{X}_{\text{train}}}^T \boldsymbol{X}_{\text{train}})^{-1} {\boldsymbol{X}_{\text{train}}}^T \boldsymbol{z}_{\text{train}} \tag{5}$$

Note that $\mathbf{X}_{\text{train}}$ is scaled and hence $\boldsymbol{\beta}$ is also scaled. The code for the $\boldsymbol{\beta}$ function looks like below:

```
def beta(X, z):
    beta = np.linalg.pinv(X.T @ X) @ X.T @ z
    return beta
```

Note that the @ syntax is a matrix product operation syntax. The **Numpy** module can inverse a matrix by calling the `np.linalg.pinv` function as we have done above and makes the calculations efficient and reliable. The predicted values of the trained dataset is then given as:

$$\tilde{z}_{\text{train}} = \mathbf{X}_{\text{train}} \cdot \boldsymbol{\beta}$$

Similar with the test data set, we get:

$$\tilde{z}_{\text{test}} = \mathbf{X}_{\text{test}} \cdot \boldsymbol{\beta}$$

MSE is then calculated for both the train and test set which it plotted to determine how $\boldsymbol{\beta}$ fits them both.

We will also plot the calculated values of β as they change for various complexity along with its confidence interval. The confidence interval is usually tabulated for different interval levels. We want to obtain a confidence interval of 95 % which leads to the confidence level of $z^* = 1.96$.

We want to visualise how the MSE changes with respect to complexity for both train data and test data. It is obvious that the train data will be a better fit for higher complexity but only for the train data it self. We need to see how well the model derived from the train data works on the test data. This gives a measure of overfitting to the train data.

Before we go deeper into the bias-variance trade-off, we will implement our first resampling technique which is the bootstrap method. As mentioned above, we will perform the OLS method on the data set until we reach a satisfied result. The `bootstrap` function requires initially five inputs when using the OLS method. That is the total times of resampling the data B, the uniformly random distributed values of $\mathbf{x}$ and $\mathbf{y}$, the true data $\mathbf{z}$ and the degree of complexity. Additional two inputs are required for the Lasso and Ridge method which we will get back to. Here we will increase the degree of complexity even further as the Franke function itself is too complex with a predictor of $p = 5$ only. Whenever this function is called upon, it will perform the OLS operations as above 75 times for each degree of complexity. We remember that the data is split and scaled into a train and test data. Only the train data is resampled while the test data is never changed. As the train data is resampled, we may use individual datapoints more than once. A short python script illustrating the procedure is given below:

```
for i in range(degree):
    X_train, z_train, X_test, z_test =
            train_test_split(X_design)
    for j in range(B):
        resample_X_index = resample(X_index)
        X_train_resample =
            X_train[resample_X_index]
        beta = beta(X_train_resample, z_train)
        z_tilde_train[j] = X_train @ beta
        z_tilde_test[j] = X_test @ beta

    z_train_error = np.mean((z_train -
                        z_tilde_train)**2)
    z_test_error = np.mean((z_test -
                        z_tilde_test)**2)
```

Here we have used `train_test_split` functionality that scikit offers. The `resample` function is also a functionality that scikit offers. What it does is to simply rearrange the values in a array with the opportunity that any of the values appears between none or multiple times. β is calculated and used on the train and test data before the

MSE of the models are calculated. The MSE is calculated only 20 times as the first column in the design matrix contains only ones. Note that this is not how the code looks in reality but it should give a good representation on how the algorithm works in practice.

As we go deeper into the analysis of the bias-variance trade-off, we need to calculate these two parameters as a function of complexity on the test data. We remember the definition of bias from (3) where y_i is the true data and $\mathbf{E}[\tilde{y}]$ is the expected value of the predicted data. B is the total number of resampling. The variance is given as:

$$Variance = \frac{1}{B}\sum_{i=0}^{B-1}\left(\tilde{y} - \mathbf{E}[\tilde{y}]\right)^2 \tag{6}$$

where $\tilde{y}$ is the predicted data. See (VII 1) for derivation. The bias and variance of the test set is plotted as a function of complexity. The MSE of the test set is included in the same plot so that we may observe how the total error is distributed between the bias and variance.

The next resampling method we will implement is the k-fold cross validation on the OLS model. As mentioned, this method splits the data set into a number of folds which in our case will be a total of five folds. Four of the folds are used for training individual models while the last fold is used for testing. We remember that individual folds can only be used for testing once so the procedure will be repeated a maximum of five times. The function is called similarly to the `bootstrap` function but instead of the total times of resampling B as input, we use the number of k-folds k as input instead. A small example of the code used is given below:

```
MSE_fold = np.zeros(complexity)
for i in len(complexity):
    X_folds = np.array_split(X_train, k)
    z_folds = np.array_split(z_train, k)
    for j in range(k):
        train_index = np.delete([0,1,2,3,4], j)
        test_index = j
        X_train_fold = X_folds[train_index]
        z_train_fold = z_folds[train_index]
        X_test_fold = X_folds[test_index]
        z_test_fold = z_folds[test_index]
        beta = beta_f(X_train_fold, z_train_fold)
        z_tilde = X_test_fold @ beta
    MSE_fold[i]= np.sum(MSE(z_test_fold, z_tilde))
    MSE_fold[i] \= k
```

This code is also a simplification of the real code for the sake of easier representation. `X_folds` and `z_folds` are both scaled. The data is split into $k-1$ train folds and one test fold. The numpy module has the functionality of splitting any array into smaller folds `np.array_split()` which we have been using. Note that β in this psuedo code will be an array of four different values as the `X_train_fold` is a set of four different train data. The

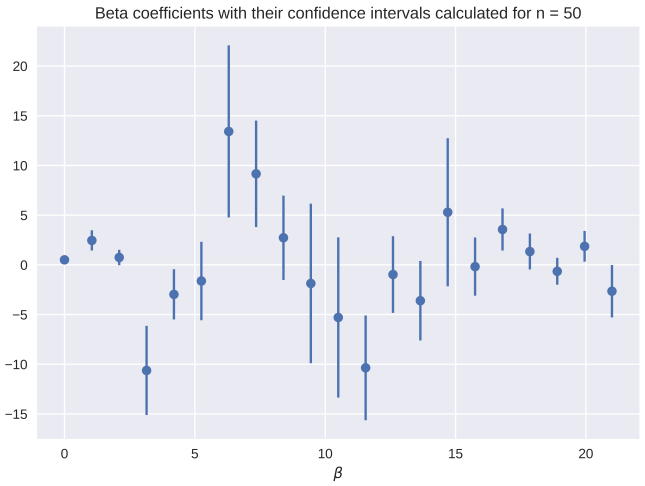

Figure 2. Estimators for a polynomial of degree 5 with a confidence level of 95%.

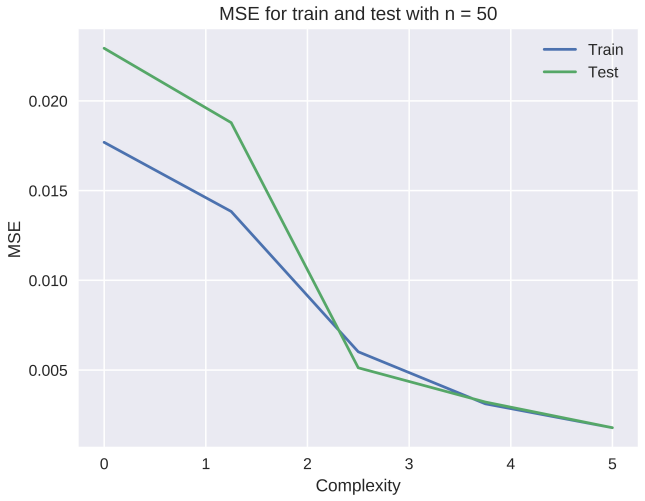

Figure 3. Mean squared error as a measure of accuracy for increasing model complexity.

total MSE is then calculated as a mean of the MSE on each fold model. The total MSE is plotted as a function of complexity alongside with the MSE of the bootstrap model to observe the difference in these two models.

Until now we have only considered the OLS model which then leads us over to the Ridge regression model. The main difference in how we compute this type of model from the OLS, is that we introduce the parameter λ to find β^{Ridge} so that we do not experience any singularities. λ is called a regularization parameter (or hyperparameter) and should be positive and not greater than one. We want to study the dependence of λ and have chosen $\lambda = [10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1]$. We split and scale the data into train and test set as normally and now β^{Ridge} is coded in as:

```
def beta_ridge(X, z, lamb):
```

```
    I = X.shape[1]
    beta_ridge = np.linalg.pinv(X.T @ X +
                 lam*np.identity(I)) @ X.T @ z
    return beta_ridge
```

The functionality `np.identity(I)` creates the identity matrix which has `I` diagonal elements. This function is found in the `functions.py` file. Similar to the OLS method, both bootstrap resampling and cross-validation technique will be implemented on the Ridge models. These two resampling methods can easily be reused as they only need to be defined once. The only difference is how we calculate β. We will compare the results from both models and determine which model fits the data better.

The bias-variance trade-off should be analysed further as we are modelling the data with another regression method. Since we are finding a model with different values of the hyperparameter λ, the bias-variance trade-off analysis should be done on each of the unique value of the λ. This is because we want to investigate the dependency of it. Only the bootstrap resample method will be used here. The implementation of the analysis is the same as above by calculating the bias and variance using the equations (3) and (6), and make various plots as a function of complexity for increasing value of λ.

A final analysis of the Ridge regression is to make a heat map plot of the predictions. An argument for the heat map is to visualise how the MSE of the predicted values evolve as a function of complexity and size of the hyperparameter. We want the MSE to be as small as possible and this kind of plot has the ability to present the optimal value of complexity and λ in a good way. So for each time we train a model, the MSE is stored in a two dimensional array where the two dimensions correspond to complexity and λ. This analysis on the Ridge regression is done on using the bootstrap and cross-validation resample methods. Then we may compare the two resampling methods and see which produce better predictions. We added more noise to the Franke function as well to observe how this would affect the error produced when using Ridge model. This is also done when introducing the Lasso model below.

Lastly, we will introduce the third regression model which is the Lasso regression. Here we will take our liberty of implementing the Lasso functionalty from **Scikit-learn** module and rather compare the results with OLS and Ridge previously calculated. This method finds another β^{Lasso} that we introduced in (II 3) which is baked in the functionality that Scikit offers. Lasso regression is slightly different from Ridge regression in terms of finding β^{Lasso}. That means the input needed to call the function is only λ as we already split the data into train and test. The function call looks like this:

```
import sklearn.linear_model as skl
X_train, z_train, X_test, z_test =
        train_test_split(X_design)
lasso = skl.Lasso(alpha=lambda,
```

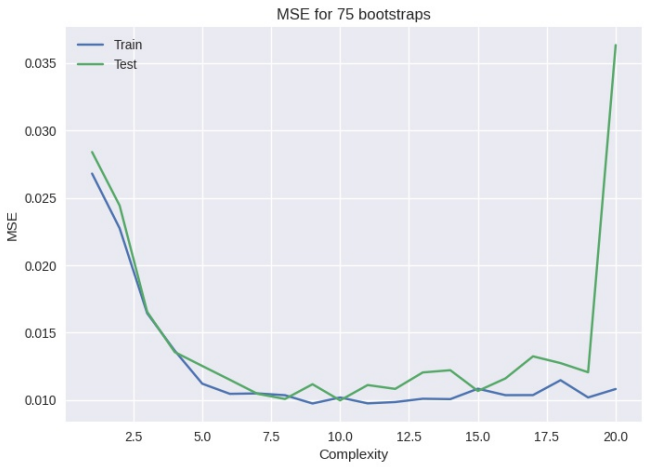

Figure 4. Mean squared error against model complexity on test and train data using the bootstrap resampling method.

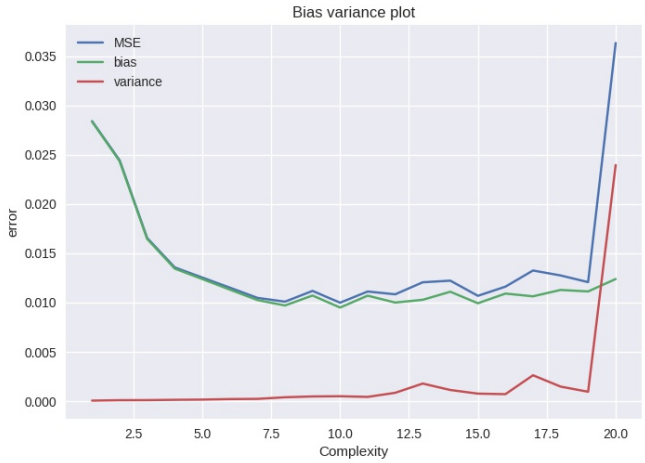

Figure 5. Bias-variance analysis of the Franke function with bootstrap resampling.

```
        fit_intercept=False).(X_train, z_train)
z_tilde = lasso.predict(X_test)
```

which is a compact way of making a model on the data. Notice that we have included `fit_intercept=False` in the Lasso call as we do not want to calculate the intercept for this model.

As previously done in both the OLS regression and Ridge regression, we want to train our model with resampling techniques and consider the performance differences through both. The MSE is stored and then plotted against the complexity for both techniques. Here we use the same approach as above. The bias-variance trade-off should be analysed also. Again, we use equations (3) and (6) to calculate bias and variance respectively and plot them as a function of complexity to observe which of these parameters dominates. Only bootsrap resample method is used for this purpose. A heat map of the Lasso regression should be implemented as well as this model

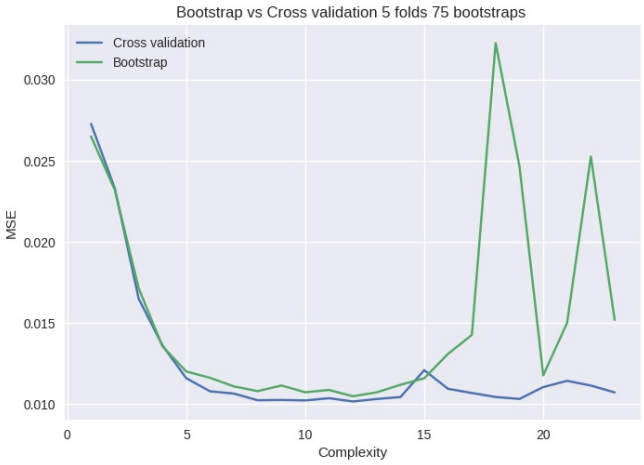

Figure 6. MSE derived from test data for the different resampling techniques Bootstrap and k-fold cross validation.

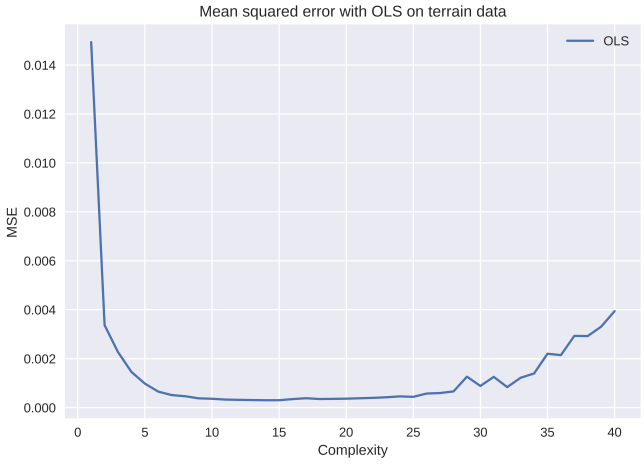

Figure 7. MSE on test terrain data using OLS without resampling.

also has an dependency off the λ parameter. As we did with the Ridge regression, the MSE is stored for each complexity for increasing value of λ. The same values of λ are used to analyse if there are any clear difference in Ridge and Lasso.

We have introduced three different regression methods and will apply those on real data. The data to be used contains information about the terrain at some region taken from satellite photos. The data can be from any regions on Earth and is obtained from U.S. Geological Survey [3]. We have chosen data sets from a region in Kamloops, Canada that we will perform our regression methods on. The file is a `.tif` file and can be read in Python using the `imread` method from the module `imageio`. The terrain data has the same format as the Franke function that we used earlier, a two dimensional data set. The data we have used so far has been quadratic. In theory, the data set can be in any shape as long as the criteria $\tilde{z} = \mathbf{X}\beta$ is fulfilled. That is, the design matrix $\mathbf{X} \in \mathbb{R}^{m\times p}$ and the fitting parameter $\beta \in \mathbb{R}^{p\times n}$ where p is the number of predictors (complexity).

Table I. R2 score from ordinary least square analysis.

Degree	1	2	3	4	5
R2 score	0.75	0.81	0.92	0.96	0.97

Each model is tested on the their respective test sets and the best error in terms of lowest MSE is stored in a separated text file named `g_results.txt`. The MSE values will be tabulated alongside with the degree such that we may observe the differences in an intuitive way based on the model fitted and resampling methods used. Two plots will be made of the real data against the model to visualise how well the model has performed on the data. One plot should be of the terrain and one should be a contour plot so that we may observe presumably smoother curves as the noise has been dealt with in this project. We begin with a small region that is 50x50 pixels in size.

IV. RESULTS

Using OLS analysis, we found the estimators $\boldsymbol{\beta}$ to the 2D polynomial fit of the Franke function. We performed a split on the data with dimensionality $n = 50$. The data were also scaled. With a polynomial of degree 5, we found the estimators from the training set. The estimators are presented in figure 2, with confidence intervals calculated from a 95% confidence level. We see that the values of the estimators span from -15 to 20.

For the training and test data, we studied the mean square error as a measure of the model's accuracy for polynomial degrees up to 5. As we can see in figure 3, the accuracy decreased steadily with model complexity on both the training and test data.

The R2 score function for polynomials up to degree 5 is presented in table I. The R2 score increases for more complex polynomials, and at degree 5 it is close to 1.

We studied regions of model complexity with low/high bias and variance by expanding on the analysis performed in figure 3. Using the bootstrap resampling technique, we fitted polynomials up until degree 20. The mean squared error was computed on training and test data. The general trend was a decrease in MSE for both train and test data with increasing model complexity. However, once the model became too complex, the test MSE increased dramatically. This analysis can be seen in figure 4. From the plot we can see the test MSE diverging from the train MSE at polynomials of degree higher than 15. We recognize a complexity of 8 as a region with low test MSE for a low polynomial degree.

We performed a bias-variance analysis of the Franke function by analysing the test data. In figure 5 we see the bias-variance analysis alongside the test MSE. From the

analysis presented in the figure, we see the variance increasing drastically with the MSE at polynomial degree 19. The bias reaches its minimum at model complexity of 10, after which it starts steadily increasing. The bias minimum resides at a model complexity of 10, but it is also good at 8.

We also performed another resampling technique to analyse the MSE from the test data, namely cross-validation. Implementing the k-fold cross-validation, we studied the MSE derived from the test data. Comparing it to the bootstrap, we present the different MSEs in figure 6. We see from the figure a steadily decreasing MSE for both resampling techniques, until a model complexity of around 15. At this point, the bootstrap MSE becomes large, while the cross validation MSE stays approximately constant.

A different regression method we explored, was ridge regression. Performing bootstrap and k-fold cross validation resampling techniques, we studied MSE from the test data. The analysis presented in figure VIII, shows how MSE varies as a function of both model complexity and the regularization parameter λ. A general trend can be seen where the MSE decreases along the diagonal of decreasing λ and increasing model complexity. However, we also identify other regions with low MSE, e.g $\lambda = 0.01$, Complexity $= 23$, or $\lambda = 0.0001$, Complexity $= 14$.

Repeating the same analysis with k-fold cross validation resampling, yielded the heatmap presented in figure VIII. Using 5 folds, we see an overall decrease in MSE for higher model complexities and smaller λ. In this figure, we do not see indications of over-fitting.

We studied the bias-variance trade-off as function of various values of λ. We present the analysis for the bootstrap resampling technique in figure VIII. By inspection, it seems like higher values for λ don't result in overfitting when model complexity gets higher.

Finally, we studied Lasso Regression. Using this regression on the Franke function, we performed analysis of the MSE through bootstrap and k-fold cross validation resampling. For bootstrap resampling, we present a heatmap for MSE as a function of complexity and λ in figure VIII. The same analysis is presented in figure VIII for k-fold cross validation resampling.

For bootstrap and k-fold cross validation resampling, we see similar trends. The MSE decreases with smaller λ. however it does not have the same dependency on model complexity as OLS and ridge regression. The MSE decreases very little for polynomials of degree 2 and up.

We also experimented with adding more noise to the data, now using $\sigma = 0.5$. We performed lasso and ridge regression on the more noisy data. The results can be seen in figure VIII. From the plots, we see that the MSE is higher. This was expected as the data got noisier. However, there is one feature which is very important. The region where we find lower MSEs is generally shifted towards higher values of λ. This is more pronounced for ridge regression, but we can also observe it in lasso regression with bootstrap resampling.

Table II. Minimum MSE from the different regression methods with various resampling methods. In this analysis, we fitted polynomials up to degree 20.

Method	OLS			Ridge		Lasso	
Resampling	w/o	boot	cross	boot	cross	boot	cross
$MSE \times 10^4$	1.796	2.248	1.993	3.082	3.026	8.168	8.213
Degree	20	14	20	18	20	20	20
λ	–	–	–	0.0001	0.0001	0.0001	0.0001

As outlined in section III, we performed analysis on real terrain data. However, by taking advantage of the results from the analysis of the Franke function, the analysis on the terrain data will not be as rigorous.

Using k-fold cross validation and bootstrap resampling on the terrain data, we analysed terrain data with the different regression methods. First, we tested models up to degree 20.

The results from this analysis can be seen in table II. Here, we focused on the MSE as a benchmark for the regression methods. From the MSE values presented in the table, we can see that OLS regression is generally best. The ridge regression method has about 3/2 times higher MSE than OLS, while lasso yields an MSE which is around four times larger. Inspecting the different resampling methods, we observe that k-fold cross-validation is slightly better than bootstrap for OLS and ridge regression, while bootstrap is marginally best for lasso regression. The best resampling method however, is no resampling method at all for OLS regression.

We also tried running for models up to degree 40. We present the results in III, where we see that OLS is still the best method. However, now we see that k-fold cross validation is slightly better than no resampling at all.

Picking OLS without resampling as our preferred regression method, we create a new model to visualise the terrain data. In figure 7, we present the MSE as function of model complexity. In this figure, we see clear indications of overfitting for Complexity > 25.

The best model had degree 14. We see this as a minimum in figure 8. The R2 score was 0.991, and the MSE was 0.0002995. In figure 8, we can see contour plots of the terrain data and the height yielded by our model. The two contour plots are mostly similar, with the difference that the model has smoother height contours. In figure 9, we can see the height plotted as images.

V. DISCUSSION

The estimators $\boldsymbol{\beta}$ were shown in figure 2. It makes sense that β_0 which is the intercept has a very narrow confidence interval. The other estimators however, were found to have large confidence intervals. This comes from the fact that there is noise in the data. Because the noise

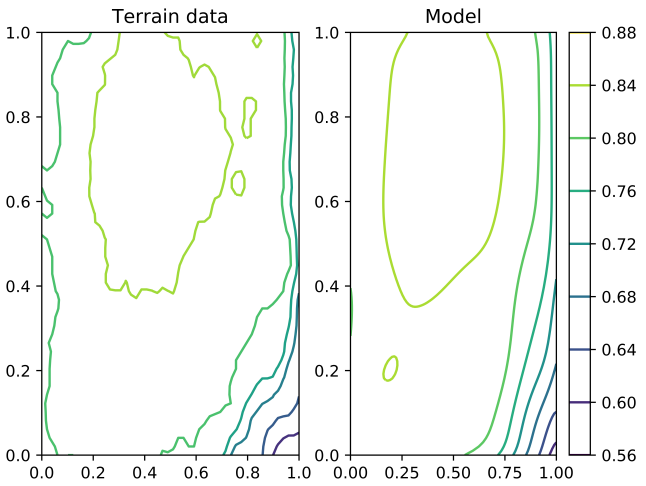

Figure 8. Contour plot of the terrain data and our model.

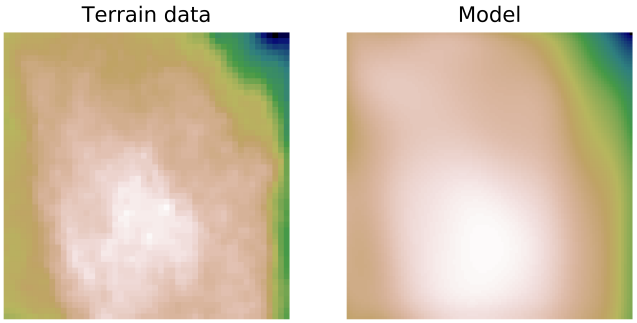

Figure 9. Image of the terrain data and our model.

is random, it gives the estimators some uncertainty. If we include more noise in the data, the estimators would get larger confidence intervals. There is not a large spread in the values of the estimators, and most of the estimators are in the range -5 to 5. The narrow range of the estimators will in general indicate less bias to the model, as the estimators do not vary a lot to try and fit to potential noise in the data.

It was found that OLS regression without resampling performed best in regards to the MSE on the test data. By inspection of figure 3 and 6, we see that this is true for the Franke function. This result was not surprising, as the noise in the data from the Franke function is "white noise".

OLS regression without resampling also performed best on terrain data as presented in table II. This was when we ran the model for polynomial degrees up to 20. However, by expanding the analysis with degrees up to 40, we found a different result. Presented in table III, we see the k-fold cross validation resampling technique performing best. We also observe that the ridge and lasso regression methods perform slightly better for higher polynomial degree.

Because the splitting of the data is random, the MSE will vary from time to time for each model. Therefore, it is hard to determine the impact of small differences in MSE. The MSE for OLS without resampling presented in table III comes from a polynomial of degree 35. This a very complex polynomial, which can indicate overfitting to the data. It also means that it is more difficult to work with this model than e.g. the OLS w/o resampling of degree 15 that has the smallest MSE in figure 7.

Table III. Similar to table II, but testing model complexities up to 40.

Method	OLS			Ridge		Lasso	
Resampling	w/o	boot	cross	boot	cross	boot	cross
$MSE \times 10^4$	2075	2208	1735	2996	2758	7342	7462
Degree	35	21	32	23	36	39	30
λ	–	–	–	0.0001	0.0001	0.0001	0.0001

This brings us in on the subject of the bias-variance trade-off. The complex polynomials are more likely to be biased by noise in the data set. In the case of the resampled Franke function, we know that there is normally distributed noise on each data point. Therefore, complex models on the training data give a large MSE on the test data. This is seen in figure 4 and 6 for OLS regression. We also see the same trend in VIII for ridge regression. To avoid this bias, we generally want to choose models with low complexities.

As seen for the different analyses on the terrain data, the train-test split yielded different results for each model. The randomness of the splitting means that it is unsafe to rely on a model of high complexity. Whereas it might yield the best MSE in one case, it can for other splittings of the data give very large MSEs. Since we in general want our model to be good for different splittings of the data, we wish to pick out a lower complexity model with a good MSE. The model of Complexity 35 in table III should therefore not be trusted. This is supported by the clear indications of overfitting for models with Complexity > 25 in figure 7.

In the previous analyses we introduced a relative small amount of noise to the Franke function. For the terrain data however, we had no knowledge of the noise. As we saw in figure VIII, noisier data benefits from a larger regularization parameter. The satellite data is presumed to be precise and thus have very little noise. Therefore, we do not have to worry about extreme values caused by noise. Because we do not need to worry about correlated noise, the regularization parameter λ should be very small or zero. Thus, OLS regression should be the preferred method of choice. This is supported by the fact that OLS regression was the best regression method on the terrain data.

VI. CONCLUSION

Through the previous analyses we have expanded our knowledge on machine learning. By investigating three different regression methods – OLS, ridge, and lasso – we observed where each method is best.

We have analysed different regression methods on data generated by the Franke function and real terrain data. In the analysis on the Franke function, we split the data in a test and train set. By fitting 2D polynomials up to degree 20, we measured the MSE derived from the model on the test data. We found the MSE generally decreasing until model complexities of 15. After this region, we encountered a divergence in MSE in the test data from the train data. This phenomenon was caused by overfitting of the training set.

Overfitting was observed in OLS and ridge regression, with both bootstrap and k-fold cross validation resampling. Its effect was not noticeable with lasso regression. However, lasso regression yielded consistently larger MSEs than the other regression methods. It was also found that the regularization parameter λ only gave better MSEs when we introduced more noise in the Franke data.

Analysing the terrain data, it was also found that OLS regression gave the best models. k-fold cross validation resampling and no resampling at all performed approximately equally good for different splitting of the data in train and test sets. We chose OLS regression without resampling to make a model of the terrain data. We found the lowest MSEs for polynomial degrees between 10 to 20. For model complexities higher than this, we observed overfitting.

The 2D polynomial from OLS regression without resampling reproduced the terrain fairly well. The trends in terrain contours were similar. The main difference came from the model reproducing smoother terrain contours.

We also discussed a reason as to why OLS regression consistently proved best for our cases. Because the source of noise in the Franke data was normally distributed, OLS was expected to be the best regression method. In the satellite data, noise was presumed to be small. There was also no cause to believe that there was no correlated noise in the data set.

VII. APPENDIX

1. *Rewriting the cost function for bias variance trade-off*

The cost function for *Ordinary Least Squares* (OLS) is:

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

We will show that we can find the $bias^2$ and the $variance^2$, from this expression. Using $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$ where ϵ is stochastic noise. And by adding and subtracting $\mathbf{E}[\tilde{y}]$, we get:

$$\begin{aligned}
&\mathbf{E}\Big[\big((f - \tilde{\mathbf{y}}) - (\tilde{y} - \mathbf{E}[\tilde{y}]) + \epsilon\big)^2\Big] \\
&= \mathbf{E}\Big[\big(f - \mathbf{E}[\tilde{y}]\big)^2 + \epsilon^2 - \big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)^2 \\
&+2\epsilon\big(f - \mathbf{E}[\tilde{y}]\big) - 2\epsilon\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big) - 2\big(\tilde{y} - E[\tilde{y}]\big)\big(f - \mathbf{E}[\tilde{y}]\big)\Big] \\
&= \mathbf{E}\Big[\big(f - \mathbf{E}[\tilde{y}]\big)^2\Big] + \mathbf{E}\big[\epsilon^2\big] - \mathbf{E}\Big[\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)^2\Big] \\
&+\mathbf{E}\Big[2\epsilon\big(f - \mathbf{E}[\tilde{y}]\big)\Big] - \mathbf{E}\Big[2\epsilon\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)\Big] \\
&-\mathbf{E}\Big[2\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)\big(f - \mathbf{E}[\tilde{y}]\big)\Big]
\end{aligned}$$

The stochastic noise is constant, therefore the expected value $\mathbf{E}[\epsilon] = 0$ and $\mathbf{E}\big[\epsilon^2\big] = \sigma^2$. The later is not intuitive, but if you look at the definition of variance, you see that the stochastic noise has mean $= \mu = 0$. It therefore makes sense that we get σ^2. Also remember that $\mathbf{E}\big[\tilde{y} - \mathbf{E}[\tilde{y}]\big] = \mathbf{E}[\tilde{y}] - \mathbf{E}[\tilde{y}] = 0$.
We recognize $f - \mathbf{E}[\tilde{y}]$ as the bias, and bias is independent of ϵ. Finally we assume that bias is constant for all bootstraps. The expression we're left with is:

$$\mathbf{E}\Big[\big(f - \mathbf{E}[\tilde{y}]\big)^2\Big] + \sigma^2 - \mathbf{E}\Big[\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)^2\Big].$$

Rewriting the expression using summation we get:

$$\mathbf{E}[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2] = \frac{1}{n}\sum_{i=0}^{n-1}\big(f_i - \mathbf{E}[\tilde{y}]\big)^2 - \frac{1}{n}\sum_{i=0}^{n-1}\big(\tilde{y} - \mathbf{E}[\tilde{y}]\big)^2 + \sigma^2. \tag{7}$$

Where the first expression is $bias^2$ and the second is $variance^2$.

[1] Yeha, I.-C., Lien, C.-H. (2009). The comparisons of data mining techniques for the predictiveaccuracy of probability of default of credit card clients. *Expert Systems with*

Applications, 36, 2473–2480.
[2] Kulkarni S. R., Rajendran, B. (2018). Spiking neural networks for handwritten digit recognition — Supervised learning and network optimization. *Neural Networks, 103*, 118–127.
[3] U.S. Geological Survey, 2014, Earthexplorer, accessed October 1., 2020 at URL https://earthexplorer.usgs.gov/
[4] Hastie T., Tibshirani R. and Friedman J., 2008. The Elements of Statistical Learning. 2nd ed. Springer, p.11.
[5] Hastie T., Tibshirani R. and Friedman J., 2008. The Elements of Statistical Learning. 2nd ed. Springer, p. 64.

VIII. SOURCE CODE

Source code and additional results can be found in the following github repository.

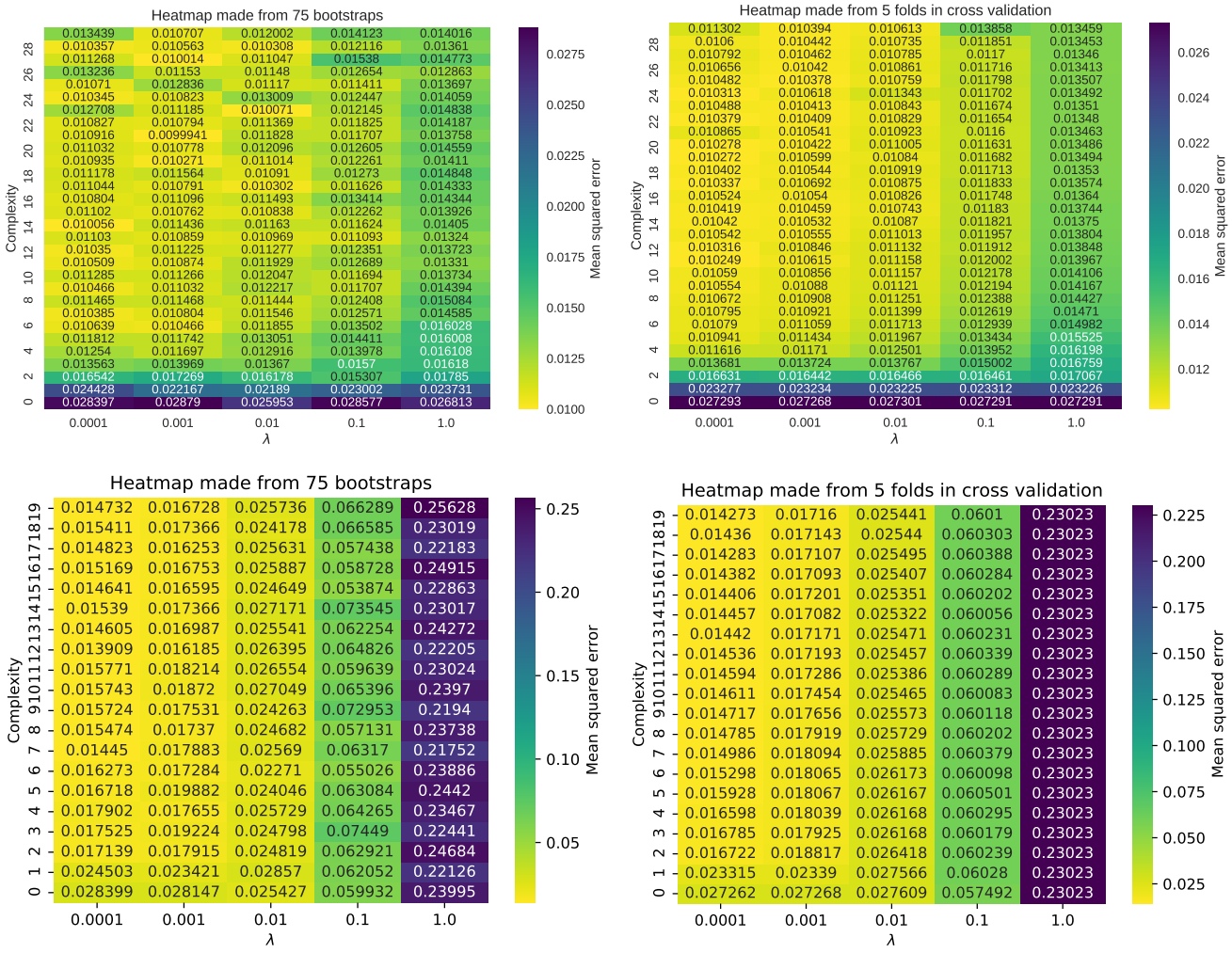

Figure 10. *MSE* for different model complexities and regularisation parameters. Top row is calculated using ridge regression, and bottom row comes from lasso regression. The columns indicate different resampling methods, bootstrap (left) and k-fold cross validation (right). Polynomial degree is Complexity + 1.

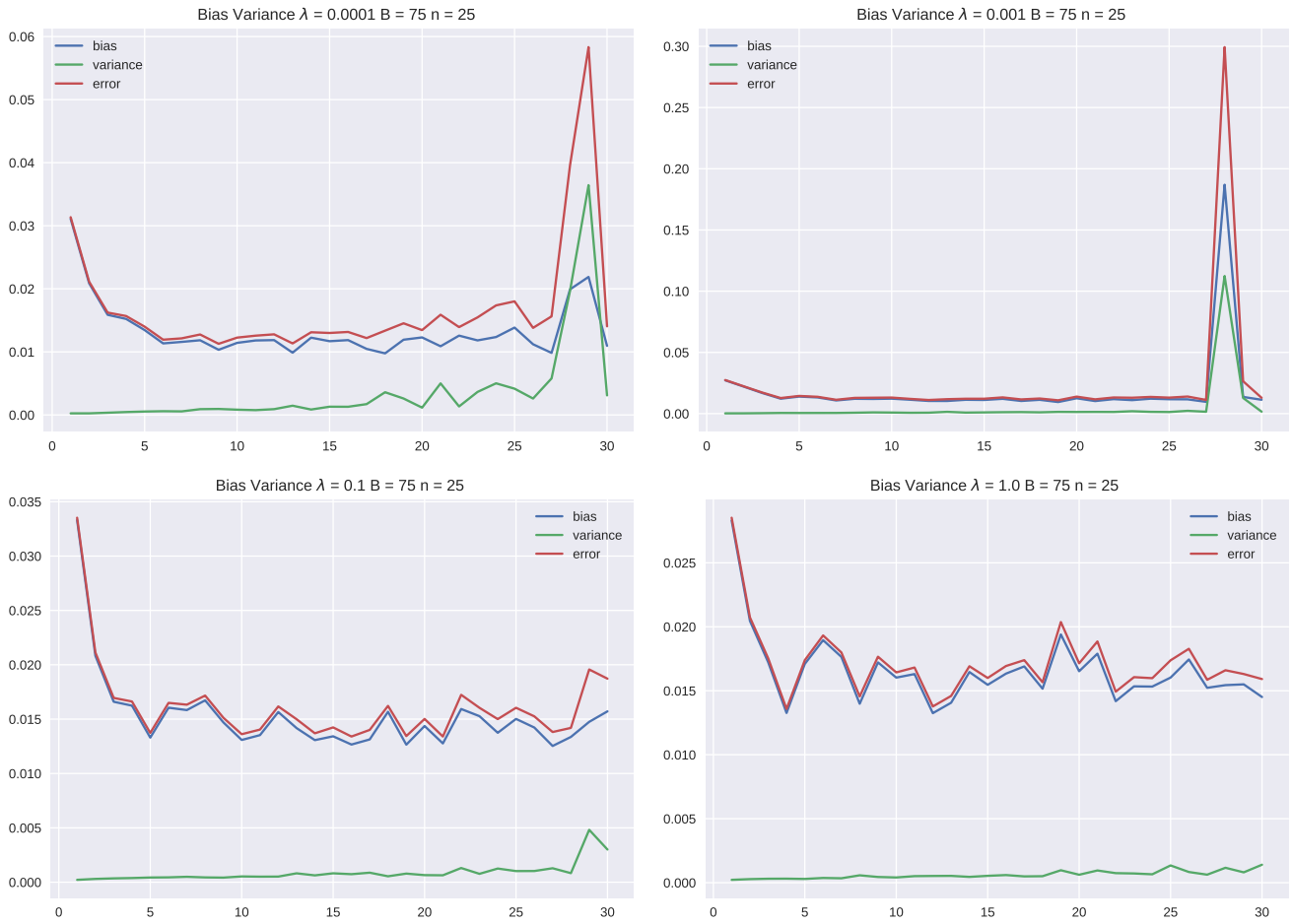

Figure 11. Bias-variance analysis for ridge regression using bootstrap resampling for various parameters λ.

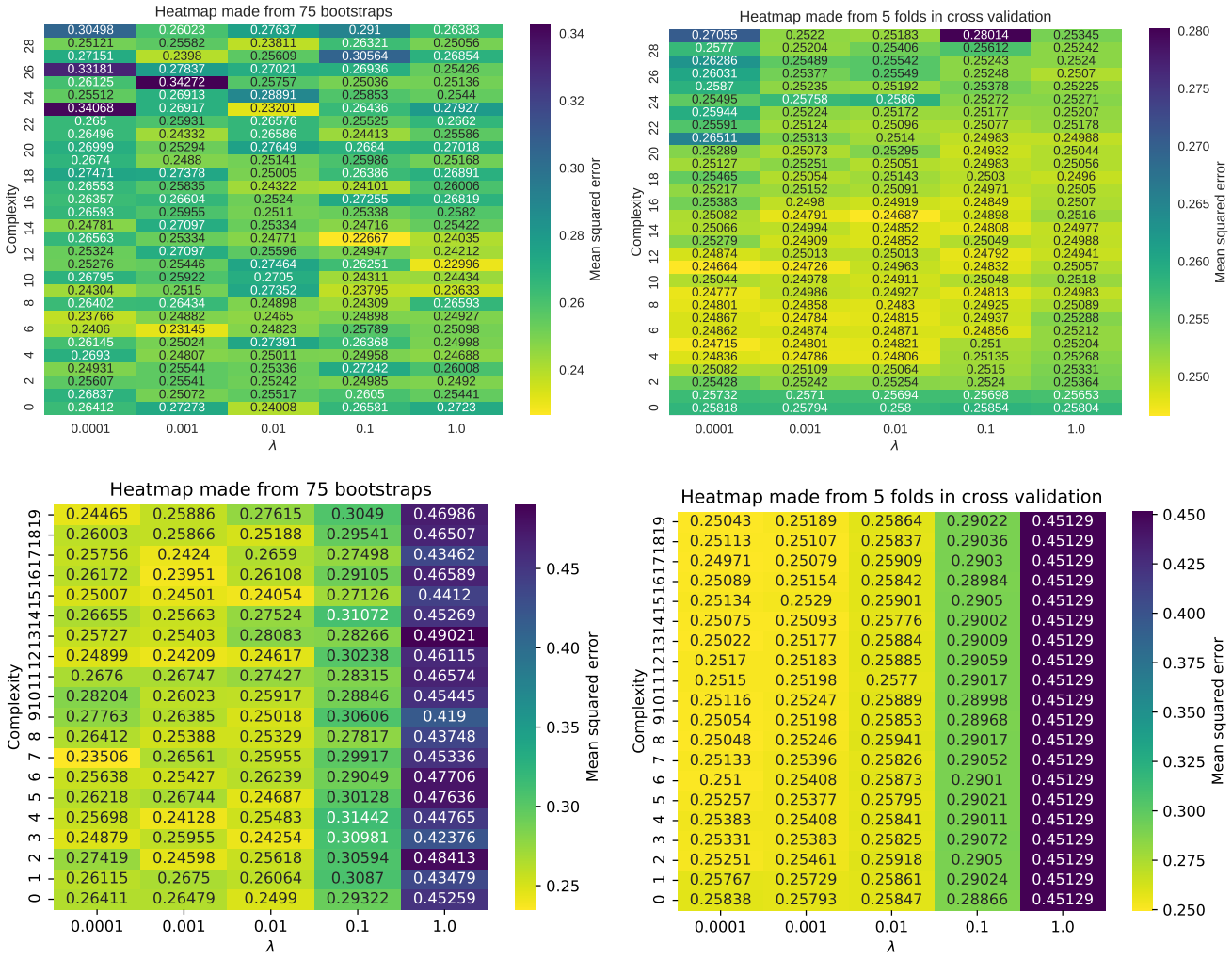

Figure 12. Similar to figure VIII, but with more noise in the data. Here, $\sigma = 0.5$.