Report for FYS-STK4155: Project 1

Maksym Brilenkov

October 2019

I present the code I have developed for fitting the external data set with several different regression algorithms - *Linear*, *Ridge* and *Lasso* regressions. For first two regression techniques, I have written my own methods for computing regression coefficients β, together with its confidence intervals, through *Singular Value Decomposition* (Linear Regression) and via usual matrix inversion (Ridge regression). I also compare obtained values with the ones from implementing Scikit Learn functionalities. Lasso regression was implemented strictly through Scikit Learn. In addition, I also consider kFold cross validation - both my and Scikit Learn methods - for Linear and Ridge. Later on, I discuss all of them and also compare it to the Scikit Learn analog, while k Fold Lasso regression on the train and test data sets is done via Scikit Learn. Finally, I discuss *bias-variance trade-off* in terms of aforementioned regression models.

1 Introduction

Machine Learning (ML) is a rapidly developing field of Science, which, in some cases (e.g. *Optical character recognition*), has been around for decades already. However, it is impact on a various fields, such as: Technology, Humanities, Medicine, Law, Social sciences etc. cannot be overrated. Its applications ranges from development of self-driving cars to solving complicated numerical problems and thus are perceived by many as one of main tools in modern world.

But what is Machine learning? It can have a variety of definitions. O'Reilly *et al* [1] defines it as "*the science (and art) of programming computers so they can learn from data*". Based on the data we have and the outcome we want to have, ML systems can be classified into several broad categories [1]:

- *Supervised*, *unsupervised* or *semi-supervised* ML algorithms. This approach

describes whether or not the pipeline is trained with human interference/supervision;

- *Online/ batch* learning. This approach describes whether or not they can learn incrementally on the fly;
- *Instance-based/ Model-based* learning. This approach describes whether they work by simply comparing new data points to known data points, or instead detect patterns in the training data and build a predictive model.

This entire project was based on the *supervised* learning algorithms, where you have an input variable(s) (also known as *features/ predictors*), $\boldsymbol{x} = [x_0, x_1, x_2, ...x_{n-1}]^T$, and an output variable(s) (*response/ outcome*), $\boldsymbol{y} = [y_0, y_1, y_2, ..., y_{n-1}]^T$, and you use an algorithm to learn the mapping function from the input to the output $Y = f(X)$. Such a task is called *regression* and in this report I have studied three different *Linear Regression* algorithms - Polynomial, Ridge and LASSO regression.

I have written the code, which implements all of the three above mentioned fitting procedures. First, I manually generate the data set on a grid and then tried to fit the system's response, which is represented in terms of Franke function [2]. Later, I calculate the regression coefficients and their respective confidence intervals, together with associated its respective errors. Afterwards I talk about k-Fold cross validation as well as *bias-variance trade-off*. In addition, I talk about their advantages and disadvantages, while comparing my results with Scikit Learn functionality.

The report organized as follows. In section 2, I briefly state the formalism of the problem. Section 3 explains the idea behind the code. In section 4, I analyze results I have obtained. The conclusion and discussion are given in 5. The code is listed in the end of this report.

2 FORMALISM

In this section, I briefly remind the reader about the theoretical background of the problem. For more thorough discussion please refer to [2] and/or [3] (lecture notes and elements of statistical learning).

2.1 POLYNOMIAL REGRESSION

To do the regression analysis means to find a functional relationship between data set and a reference set [2]. Therefore, the aim of the regression analysis is to *construct the function*, which will, in principle, map any input data of a similar format to some continuous output values. To put it simply, we want to describe given data set in terms of some variables and not only that, but also to have a model which will be able to predict similar values. Thus, we need to construct the function in such a way, which will allow us to predict the values y not present in the current set. The most basic idea then is to parametrise the function in terms of polynomial. If we have n points, we get $n-1$ (first is zero degree, i.e. constant), which results in [2]:

$$y(x_i) = \tilde{y}_i + \epsilon_i = \sum_{j=0}^{n-1} \beta_j x_i^j + \epsilon_i \,, \tag{2.1}$$

Using matrix notation, this expression can be written as [2] :

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \,, \tag{2.2}$$

where

$$\boldsymbol{y} = [y_0, y_1, ..., y_{n-1}]^T, \quad \boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_{n-1}]^T, \quad \boldsymbol{\epsilon} = [\epsilon_0, \epsilon_1, ..., \epsilon_{n-1}]^T \,, \tag{2.3}$$

$$\boldsymbol{X} = \begin{bmatrix} x_{00} & x_{01} & x_{02} & ... & x_{0,n-1} \\ x_{10} & x_{11} & x_{12} & ... & x_{1,n-1} \\ ... & ... & ... & ... & ... \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & ... & x_{n-1,n-1} \end{bmatrix}$$

The idea is to obtain an optimal set of β_i values, which can fit best our current and future datasets. We do this by defining an approximation

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta} \,, \tag{2.4}$$

and minimising the so-called *cost function*

$$\boldsymbol{C}(\boldsymbol{\beta}) = \frac{1}{n} \sum_{i=0}^{n-1} \left(y_i - \tilde{y}_i\right) = \frac{1}{n} \left\{ \left(\boldsymbol{y} - \tilde{\boldsymbol{y}}\right)^{\boldsymbol{T}} \left(\boldsymbol{y} - \tilde{\boldsymbol{y}}\right) \right\} , \tag{2.5}$$

i.e. by minimizing the square distance between the estimated function and the observed value, also known as *residual sums of squares*

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta} \left\{ \left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\right)^{\boldsymbol{T}} \left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\right) \right\}, \tag{2.6}$$

Such optimization problem has the solution in the form [2]

$$\hat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\right)^{-1} \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}. \tag{2.7}$$

$\hat{\boldsymbol{\beta}}$ is known as *ordinary least squares estimator*, and due to the common assumption that the error term, ε, has mean zero, the OLS estimator supposed to be *unbiased*. Such property allows OLS to evaluate the *true* values of the regression coefficients.

Sometimes matrix $\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$ cannot be inverted and thus the standard OLS, based on inversion algorithm, will lead to singularities. One of approaches to avoid this is called **Singular Value Decomposition**, which allows us to rewrite X in terms of an orthogonal/unitary transformation U [2]

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}, \tag{2.8}$$

With this in mind, it can be shown that [2]

$$\boldsymbol{X}\boldsymbol{\beta} = \boldsymbol{X}\left(\boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\boldsymbol{T}}\right)^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}\left(\boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\boldsymbol{T}}\right)^{-1}\left(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}\right)^{T}\boldsymbol{y} = \boldsymbol{U}\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{y}. \tag{2.9}$$

2.2 The Bias-variance trade-off

In principle, we want to estimate the regression coefficients with the method, which involves least possible errors. The prediction error is frequently introduced by the metric called *Mean Squared Error* (MSE) and it can be defined as [2]

$$\mathrm{MSE}(\boldsymbol{X}\boldsymbol{\beta}) = E\left[\left(\boldsymbol{y} - \tilde{\boldsymbol{y}}\right)^2\right], \quad \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{\varepsilon}, \tag{2.10}$$

Let us look into this expression in a more detail

$$\begin{aligned}
E\left[\left(\boldsymbol{y} - \tilde{\boldsymbol{y}}\right)^2\right] &= E\left[\left(\boldsymbol{f} + \boldsymbol{\varepsilon} - \tilde{\boldsymbol{y}} + E[\tilde{\boldsymbol{y}}] - E[\tilde{\boldsymbol{y}}]\right)^2\right] && (2.11)\\
&= \left[\left(\boldsymbol{f} - E[[y]]\right) - \left(\tilde{\boldsymbol{y}} - E[\tilde{y}]\right)\right]^2 + \sigma^2 && (2.12)\\
&= \left(\boldsymbol{f} - E[\boldsymbol{y}]\right)^2 - 2\left(\boldsymbol{f} - E[\boldsymbol{y}]\right)\left(\tilde{\boldsymbol{y}} - E[\boldsymbol{y}]\right) + \left(\tilde{\boldsymbol{y}} - E[\tilde{\boldsymbol{y}}]\right)^2 + \varepsilon^2 && (2.13)\\
&= E\left[\left(\boldsymbol{f} - E[\boldsymbol{y}]\right)^2 + \left(\tilde{\boldsymbol{y}} - E[\tilde{\boldsymbol{y}}]\right)^2 + \boldsymbol{\varepsilon}^2\right] - 2E\left[\left(\boldsymbol{f} - E[\boldsymbol{y}]\right)\left(\tilde{\boldsymbol{y}} - E[\boldsymbol{y}]\right)\right] && (2.14)
\end{aligned}$$

where the last part equals to zero because:

$$E[\boldsymbol{f}E[\tilde{\boldsymbol{y}}]] = E[\boldsymbol{f}]E[\tilde{\boldsymbol{y}}], \tag{2.15}$$

Thus, we get

$$\text{MSE}(\boldsymbol{X\beta}) = E\left[\left(\boldsymbol{y}-\tilde{\boldsymbol{y}}\right)^2\right] = E\left[\left(\boldsymbol{f}-E[\boldsymbol{y}]\right)^2 + \left(\tilde{\boldsymbol{y}}-E[\tilde{\boldsymbol{y}}]\right)^2 + \boldsymbol{\varepsilon}^2\right], \tag{2.16}$$

or

$$E\left[\left(\boldsymbol{y}-\tilde{\boldsymbol{y}}\right)^2\right] = \frac{1}{n}\sum_i \left(f_i - E[\tilde{\boldsymbol{y}}]\right)^2 + \frac{1}{n}\sum_i \left(\tilde{y}_i - E[\tilde{\boldsymbol{y}}]\right)^2 + \sigma^2. \tag{2.17}$$

The last term, σ^2 here irreducible error and it is beyond our control. The second term is the *variance* of our model, which is simply the variance of the an average. It decreases with inverse of n (the degrees of freedom/model complexity). The first term is *bias* and it is the squared difference between the true mean and the expected value of the estimate. As n grows, it will increase; hence the bias-variance trade-off.

Because of the existence of bias-variance trade-off, it could be better if we would be able to utilize this property somehow. We can do this by introducing bias to the estimates, while achieving less variability. Unfortunately, OLS are unbiased by default and hence cannot be exploited.

But what if we introduce this bias? For instance, adding the penalty, λ:

$$\boldsymbol{X^T X} \rightarrow \boldsymbol{X^T X} + \boldsymbol{\lambda I}, \tag{2.18}$$

It can be shown (see e.g. [3], that in this way we will get a slightly different optimization problem:

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} = \arg\min_\beta \left\{ \left(\boldsymbol{y}-\boldsymbol{X\beta}\right)^{\boldsymbol{T}} \left(\boldsymbol{y}-\boldsymbol{X\beta}\right) + \lambda \sum_{i=1}^{p} \beta_i^2 \right\}, \tag{2.19}$$

or

$$\hat{\boldsymbol{\beta}}^{\text{ridge}} = \left(\boldsymbol{X^T X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}, \tag{2.20}$$

Such optimization problem is called **Ridge** regression and it is highly dependable on the *tuning* or *penalty* (*hyper*) parameter, λ.

Ridge regression is the part of *shrinkage methods* [3] and it imposes the penalty on the size of regression coefficients and thus shrinks them. The amount of shrinkage is controlled by $\lambda \geq 0$ - larger λ leads to larger amount of shrinkage.

In addition, λ is not affected by the learning algorithm itself and must be set before start training the algorithm. Moreover, its value should remain constant during the entire training process.

Very large values of λ will result in almost flat model (zero slope); thus, it will not achieve *overfitting* of the training data, but there is smaller chance to find a good solution.

2.3 LASSO REGRESSION

The LASSO (Least Absolute Shrinkage and Selection Operator) is a regression method that involves penalizing the absolute size of the regression coefficients.

By penalizing or constraining the sum of the absolute values of the estimates you make some of your coefficients zero. The larger the penalty applied, the further estimates are shrunk towards zero. This is convenient when we want some automatic feature/variable selection, or when dealing with highly correlated predictors, where standard regression will usually have regression coefficients that are too large.

Lasso is quite different in comparison to Ridge, i.e. the estimate now defined as [3]:

$$\hat{\beta}^{lasso} = \text{argmin}_{\beta} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 , \tag{2.21}$$

$$\text{subject to} \sum_{j=1}^{p} |\beta_j| \leq t , \tag{2.22}$$

In this case the solution for $\hat{\beta}^0 =< y >$ which means we need to fit our model without an intercept, and thereafter we fit a model without an intercept.

To summarize, while Ridge regression does a proportional shrinkage, Lasso translates each coefficient by a constant factor λ truncating at zero [3].

2.4 RESAMPLING METHODS - CROSS VALIDATION

Splitting your data set into test and training one is the common practice in ML algorithms. It is used to compare and select a model for a given predictive modeling problem because it is easy to understand, easy to implement, and results in skill estimates that generally have a lower bias than other methods [1] (hands on machine learning).

There are a lot of ways to split your data set into several parts. One of them is called **k-Fold Cross validation** (CV). It is a statistical method used to estimate the "learning" skill of ML models. It is very useful resampling if you have limited data sample and you want to test your model. It is called k-Fold because the parameter k refers to the number of groups that a given data sample is splitted into. When a specific value for e.g. k=10 is chosen, k-Fold becomes 10-fold CV.

After you splitted your entire data set on k subsets, you train each model against different combination of these subsets and validate it against the remaining parts. Once the model type and λ have been selected, a final model is trained using these hyperparameters on the full training set, and the generalized error is measured on the test set.

It is primarily used to estimate the skill of ML model on "unseen" data, i.e. you have a limited sample and you use it estimate how well your model will work in general on the data, which were not used during the training of the model.

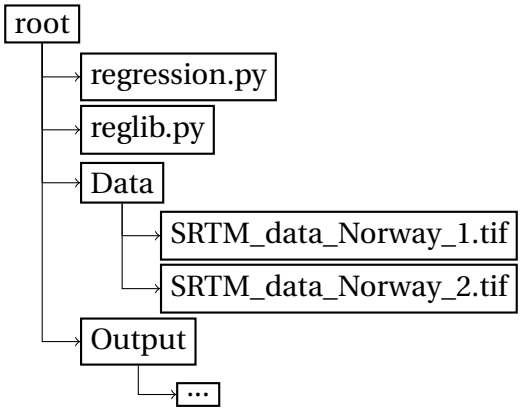
Overall, the algorithm is fairly straightforward:

- Shuffle the data set randomly;
- Split the data set into k groups;
- For each unique group:
    1. Take the group as a hold out or test data set;
    2. Take the remaining groups as a training data set;
    3. Fit a model on the training set and evaluate it on the test set;
    4. Retain the evaluation score and discard the model;
    5. Summarize the skill of the model using the sample of model evaluation scores.

3 CODE IMPLEMENTATION AND TESTING

In this section I describe the various parts of the code I've written. It equally works for "fake" (manually generated) and "real" (Norwegian terrain) data sets. The whole folder structure can be viewed as follows:

- reglib.py - small library, which contains all necessary methods to calculate β's, MSE, CrossValidation and Bias-Variance trade-off;
- regression.py - main module, which contains the entry point of the program together with calling procedures for the reglib.py library;



- **reglib.py** - small library, which contains all necessary methods to calculate β's, MSE, CrossValidation and Bias-Variance trade-off;
- **regression.py** - main module, which contains the entry point of the program together with calling procedures for the reglib.py library;
- **Data** - folder which contains the real data. Note: you do not necessarily need to copy the data files inside, but can use the full path in your OS;
- **Output** - contains all the output files produced during the program run.

To obtain the program please navigate to the directory you want the program to be stored at with:

```
cd /path/to/your/directory/
```

and simply clone the repository by typing:

```
git clone https://github.com/maksymbr/machine-learning-projects.
    ↪ git
```

The script was written and tested with python 3.7. I was running it either through JupyterNotebook, Spider or PyCharm. But, it is possible to run it through terminal. Just locate to the directory with the script and type:

```
python regression.py
```

Before you run it make sure you have installed such libraries as: **numpy**, **sympy**, **scipy**, **joblib**, **itertools**, **matplotlib**.

After execution, you will be asked some questions about the data set and other parameters you want to use. In my opinion, it is quite simple to understand, so I will omit the tedious explanations here and will go straight to the code explanation.

3.1 Fake Data

The original idea was to write a program which, in principle, will be able to work with any amount of independent variables. For this I am using a sympy library to generate the *symbolic* array of independent variables as:

```
x_symb = sp.symarray('x', n_vars, real = True)
```

which will result in a list of variables x_0, x_1, x_2, ... x_n, where n = n_vars.

For a given set of points, I am generating a set of values for the symbolic variables as follows:

```
x_vals = x_symb.copy()
for i in range(n_vars):
    x_vals[i] = np.arange(0, 1, 1./N_points)
```

Because we are using Franke function [2]:

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

I am generating the response as:

```
import reglib as rl
# library object instantiation
lib = rl.RegressionLibrary(x_symb, x_vals)
# setting up the grid
x, y = np.meshgrid(x_vals[0], x_vals[1])
# and getting output based on the Franke Function
z = lib.FrankeFunction(x, y) + 0.1 * np.random.randn(N_points,
    ↪ N_points)
```

and, for each polynomial (up to specified max value), I am instantiating the pipeline, which calculates OLS, Ridge and LASSO regressions with several different hyper parameters

```
for poly_degree in range(1, max_poly_degree+1):
    print('\n')
    print('Starting␣analysis␣for␣polynomial␣of␣degree:␣' + str(
        ↪ poly_degree))
    pipeline = MainPipeline(x_symb, x_vals, x, y, z, confidence,
        ↪ sigma, kfold, lambda_par, output_dir, prefix,
        ↪ poly_degree)
    pipeline.doRegression()
```

3.1.1 Design Matrix

To construct a proper design matrix, I first need to generate the polynomial for appropriate degree. I am doing this by using **sympy** and **itertools** library to account for all possible combinations of multiplications between our variables, x_0, x_1, and 1 (i.e. $x_0 * x_1 * 1$, $x_0 * x_0 * x_1 * 1$, ...) as:

```
variables = list(self.x_symb.copy())
variables.append(1)
terms = [sp.Mul(*i) for i in it.combinations_with_replacement(
    ↪ variables, poly_degree)]
```

After that, I create the matrix and feed it with values, using **lambdify** method from **sympy** library:

```
points = len(x_vals[0]) * len(x_vals[1])
X1 = np.ones((points, len(terms)))
for k in range(len(terms)):
    f = sp.lambdify([self.x_symb[0], self.x_symb[1]], terms[k], "
        ↪ numpy")
    X1[:, k] = [f(i, j) for i in self.x_vals[1] for j in self.
        ↪ x_vals[0]]
```

The whole method,which returns design matrix for a given polynomial degree is then:

```
def constructDesignMatrix(self, *args):
    # the degree of polynomial to be generated
    poly_degree = args[0]
    # getting inputs
    x_vals = self.x_vals
    # using itertools for generating all possible combinations
    # of multiplications between our variables and 1, i.e.:
    # x_0*x_1*1, x_0*x_0*x_1*1 etc. => will get polynomial
    # coefficients
    variables = list(self.x_symb.copy())
```

```
    variables.append(1)
    terms = [sp.Mul(*i) for i in it.combinations_with_replacement(
        ↪ variables, poly_degree)]
    # creating desing matrix
    points = len(x_vals[0]) * len(x_vals[1])
    # creating desing matrix composed of ones
    X1 = np.ones((points, len(terms)))
    # populating design matrix with values
    for k in range(len(terms)):
        f = sp.lambdify([self.x_symb[0], self.x_symb[1]], terms[k
            ↪ ], "numpy")
        X1[:, k] = [f(i, j) for i in self.x_vals[1] for j in self.
            ↪ x_vals[0]]
    # returning constructed design matrix (for 2 approaches if
        ↪ needed)
    return X1
```

With this in hand, I proceed for regression analysis.

3.1.2 Linear Regression

The main method here is dolinearRegression, which returns predicted values, regression coefficients and confidence intervals for β coefficients. It is based on the **Singular Value Decomposition** (code is taken from [2]):

```
def doSVD(self, *args):
    # getting matrix
    X = args[0]
    # Applying SVD
    A = np.transpose(X) @ X
    U, s, VT = np.linalg.svd(A)
    D = np.zeros((len(U), len(VT)))
    for i in range(0, len(VT)):
        D[i, i] = s[i]
    UT = np.transpose(U)
    V = np.transpose(VT)
    invD = np.linalg.inv(D)
    invA = np.matmul(V, np.matmul(invD, UT))

    return invA
```

and takes the generated design matrix and the responses (generated by Franke function) as its inputs:

```
def doLinearRegression(self, *args):
    # getting design matrix
    X = args[0]
    # getting z values and making them 1d
    z = np.ravel(args[1])
```

```
    # calculating variance of data

    # and then make the prediction
    invA = self.doSVD(X)
    beta = invA.dot(X.T).dot(z)
    ztilde = X @ beta
    # calculating beta confidence
    confidence = args[2]   # 1.96
    sigma = args[3] #np.var(z)  # args[3] #1
    SE = sigma * np.sqrt(np.diag(invA)) * confidence
    beta_min = beta - SE
    beta_max = beta + SE

    return ztilde, beta, beta_min, beta_max
```

For a given polynomial degree I get a set of β's, which I plot and save inside "Output" folder.

3.1.3 RIDGE REGRESSION

The main method here is doRidgeRegression. It works similarly to the doLinearRegression method, but it takes one more input - hyper parameter:

```
lambda_par = args[2]
```

and constructs the identity matrix as:

```
XTX = X.T.dot(X)
I = np.identity(len(XTX), dtype=float)
```

which allows calculation of β's as:

```
invA = np.linalg.inv(XTX + lambda_par * I)
beta = invA.dot(X.T).dot(z)
```

Thus the entire method is

```
def doRidgeRegression(self, *args):
    # getting design matrix
    X = args[0]
    # getting z values
    z = np.ravel(args[1])
    # hyper parameter
    lambda_par = args[2]
    # constructing the identity matrix
    XTX = X.T.dot(X)
    I = np.identity(len(XTX), dtype=float)
    # calculating parameters
    invA = np.linalg.inv(XTX + lambda_par * I)
    beta = invA.dot(X.T).dot(z)
```

```
    # and making predictions
    ztilde = X @ beta

    # calculating beta confidence
    confidence = args[3]   # 1.96
    # calculating variance
    sigma = args[4]#np.var(z)   # args[4] #1
    SE = sigma * np.sqrt(np.diag(invA)) * confidence
    beta_min = beta - SE
    beta_max = beta + SE

    return ztilde, beta, beta_min, beta_max
```

3.1.4 Lasso

For Lasso Regression I have used only the Scikit learn functionalities:

```
lasso_reg = Lasso(alpha=self.lambda_par).fit(X_poly, z_rav)
ztilde_sk = lasso_reg.predict(X_poly).reshape(zshape)
```

3.1.5 Cross Validation

Because the original task for Linear Regression analysis was split into several parts, I wrote code for k-Fold cross validation as a separate method. The idea for a division is as follows:

- Instead of two groups, we must return k-folds or k groups of data.
- If the dataset does not cleanly divide by the number of folds, there may be some remainder rows and they will not be used in the split. This means that we need first to account for this division. I have written method which accounts for that, and also shuffling the data set randomly:

```
def shuffleDataSimultaneously(self, a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]
```

and then divides it into equal kfolds:

```
def splitDataset(self, *args):
    # getting inputs
    X = args[0]
    z = args[1]
    kfold = args[2]
    iterator = args[3]
```

```
    # If the dataset does not cleanly divide by the number of
      ↪ folds,
    # there may be some remainder rows and they will not be used
      ↪ in the split.
    length = len(X) % kfold
    if length == 0:
        condition = True
    else:
        condition = False
    while condition is False:
        # removing the element <= they were shuffled randomly,
        # so it doesn't matter which one to remove
        X = np.delete(X, -1, axis = 0)
        z = np.delete(z, -1, axis = 0)
        # checking whether it is divided cleanly
        length = len(X) % kfold
        if length == 0:
            condition = True
    # 2. Split the dataset into k groups:
    X_split = np.array_split(X, kfold, axis=0)
    z_split = np.array_split(z, kfold, axis=0)
    # train data set - making a copy of the shuffled and splitted
      ↪ arrays
    X_train = X_split.copy()
    z_train = z_split.copy()
    # test data set - each time new element
    X_test = X_split[iterator]
    z_test = z_split[iterator]
    # deleting current element
    X_train = np.delete(X_train, iterator, 0)
    z_train = np.delete(z_train, iterator, 0)
    # and adjusting arrays dimensions (e.g. X: [4, 500, 21] =>
      ↪ [2000, 21])
    X_train = np.concatenate(X_train, axis=0)
    z_train = z_train.ravel()

    return X_train, X_test, z_train, z_test
```

The Cross validation itself is done via

```
def doCrossVal(self, *args):
    # getting design matrix
    X = args[0]
    # getting z values and making them 1d
    z = np.ravel(args[1])
    kfold = args[2]
    MSEtest_lintot = []
    MSEtrain_lintot = []
    z_tested = []
    z_trained = []
```

```
    z_t = []
    # bias
    bias = []
    # shuffling dataset randomly
    # 1. Shuffling datasets randomly:
    X, z = self.shuffleDataSimultaneously(X, z)
    # splitting data sets into the kfold and iterate over each of
       ↪ them
    for i in range(kfold):
        # Splitting and shuffling data randomly
        X_train, X_test, z_train, z_test = self.splitDataset(X, z,
           ↪ kfold, i)
        z_t.append(z_test)
        # Train The Pipeline
        invA = self.doSVD(X_train)
        beta_train = invA.dot(X_train.T).dot(z_train)
        # Testing the pipeline
        z_trained.append(X_train @ beta_train)
        z_tested.append(X_test @ beta_train)
        # Calculating MSE for each iteration
        MSEtest_lintot.append(self.getMSE(z_test, z_tested[i]))
        MSEtrain_lintot.append(self.getMSE(z_train, z_trained[i]))
    # linear MSE
    MSEtest_lin = np.mean(MSEtest_lintot)
    MSEtrain_lin = np.mean(MSEtrain_lintot)
    # bias-variance trade off
    z_tested_mean = np.mean(z_tested, axis=1, keepdims=True)
    for i in range(kfold):
        bias.append((z_t[i] - z_tested_mean)**2)
    bias_mean = np.mean( bias )
    variance_mean = np.mean( np.var(z_tested, axis=1, keepdims=
       ↪ True) )

    return MSEtest_lin, MSEtrain_lin, bias_mean, variance_mean
```

which uses singular value decomposition to calculate β. The method returns the Mean Squared Error for test and training data sets, as well as bias and variance for a given polynomial. The code to calculate MSEs is taken from [2]:

```
def getMSE(self, z_data, z_model):
    n = np.size(z_model)
    return np.sum((z_data - z_model) ** 2) / n
```

Cross validation for Ridge regression is done in a similar manner, which includes hyper parameter λ. This method is called "doCrossValRidge".

For Lasso regression I am using Scikit Learn methods and, because, I am also comparing my results with the ones from Scikit Learn (only for Linear and Ridge), I wrote a single method, which handles every regression type for CV. It is based on Scikit

Learn functionality.

The important parameter is a string variable which decides what regression algorithm to use:

```
reg_type = args[5]
if reg_type == 'linear':
    model = LinearRegression(fit_intercept = False)
elif reg_type == 'ridge':
    model = Ridge(alpha = lambda_par, fit_intercept = False)
elif reg_type == 'lasso':
    model = Lasso(alpha = lambda_par)
else:
    print("Houston, we've got a problem!")
```

With this, I go directly to Scikit functionalities for splitting data set into k parts. The entire method is then:

```
def doCrossValScikit(self, *args):
        # getting inputs
        X = args[0]
        z = args[1]
        kfold = args[2]
        poly_degree = args[3]
        lambda_par = args[4]
        # understanding the regression type to use
        reg_type = args[5]
        if reg_type == 'linear':
            model = LinearRegression(fit_intercept = False)
        elif reg_type == 'ridge':
            model = Ridge(alpha = lambda_par, fit_intercept =
               ↪ False)
        elif reg_type == 'lasso':
            model = Lasso(alpha = lambda_par)
        else:
            print("Houston, we've got a problem!")

        MSEtest = []
        MSEtrain = []
        # bias
        bias = []
        z_t = []
        z_tested = []
        # If the dataset does not cleanly divide by the number of
           ↪ folds,
        # there may be some remainder rows and they will not be
           ↪ used in the split.
        length = len(X) % kfold
        if length == 0:
            condition = True
```

```
        else:
            condition = False
        while condition is False:
            # removing the element <= they were shuffled randomly,
            # so it doesn't matter which one to remove
            X = np.delete(X, -1, axis = 0)
            z = np.delete(z, -1, axis = 0)
            # checking whether it is divided cleanly
            length = len(X) % kfold
            if length == 0:
                condition = True
        # making splits - shuffling it
        cv = KFold(n_splits = kfold, shuffle = True, random_state
            ↪ = 1)
        # enumerate splits - splitting the data set to train and
            ↪ test splits
        for train, test in cv.split(X):
            X_train, X_test = X[train], X[test]
            z_train, z_test = z[train], z[test]
            z_t.append(z_test)
            # making the prediction - comparing outputs for
                ↪ current and "future" datasets
            z_tilde = model.fit(X_train, z_train).predict(X_train)
                ↪ .ravel() # z_trained
            z_pred = model.fit(X_train, z_train).predict(X_test).
                ↪ ravel() # z_tested
            z_tested.append(z_pred)

            MSEtest.append(mean_squared_error(z_test, z_pred))
            MSEtrain.append(mean_squared_error(z_train, z_tilde))

        # getting the mean values for errors (to plot them later)
        MSEtest_mean = np.mean(MSEtest)
        MSEtrain_mean = np.mean(MSEtrain)
        # bias-variance trade off
        z_tested_mean = np.mean(z_tested, axis=1, keepdims=True)
        for i in range(kfold):
            bias.append((z_t[i] - z_tested_mean)**2)
        bias_mean = np.mean( bias )
        variance_mean = np.mean( np.var(z_tested, axis=1, keepdims
            ↪ =True) )

        # returning MSE, bias and variance for  a given polynomial
            ↪  degree
        return MSEtest_mean, MSEtrain_mean, bias_mean,
            ↪ variance_mean
```

The method which are located in the regression.py is mostly for plotting, saving and printing various values.

3.2 Real Data - Bug fixes

The real data uses exactly the same method described above. The only difference is that instead of generating function using Franke function, I retrieve it from the tif file.

The problem with the real data here is that the data set is pretty big. For the first run, I didn't account for it, and that is why it took me more than 7 hours to finish calculating things for 5 consecutive polynomials. To be precise, the console output was:

```
-- Program finished at 26326.446545124054 sec --
```

But, as we will see later, there is no actual need to run the entire data set. In fact, some of our fellow students mentioned that 10% is more than enough to run as the amount of points to train and test the model. I was running even lesser amount. All in all, I changed script a little bit in order to run only portion of the data set

```
sys.stdout.write("Please, specify the percentage of the data to
    ↪ use (default = 0.1): ")
cut_dat = input()
terrain.sort()
if cut_dat == '':
    dataset = terrain[int(len(terrain) * 0.9):]#terrain[int(len(
        ↪ terrain) * .05) : int(len(terrain) * .95)]
else:
    dataset = terrain[int(len(terrain) * (1.-float(cut_dat))):]
print(np.shape(dataset))
```

This part is located inside **regression.py** after:

```
if __name__ == '__main__':
```

statement.

After running the script you will be specifically asked what part of the data set you want to use (from 0 to 1, with 1 being the largest value). There is, of course, a default value of 10%.

Basically, after running the script with the command above, you will be asked several questions about the data set you want to use and other small details such as the minimal value of λ to use, the max degree of polynomial, the number of points to generate

The script **regression.py** is quite long, but it mostly contains calling for the methods for calculation and ploting fitted surfaces, MSEs, β, confidence intervals etc. It also saves values, such as regression coefficients and confidence intervals inside **txt** files to be able to access them later on.

4 RESULTS

In this section I am presenting the main results of the current project. I have to run code both for generating artificial data set as described in previous sections, as well as on the real one.

4.1 GENERATED/"FAKE" DATA SET

4.1.1 OLS, RIDGE AND LASSO REGRESSIONS ON FRANKE FUNCTION

For artificial data set, I have used polynomial up to 8th degree with the generated grids of 10×10, 21×21, 50×50 and 100×100 points and hyper parameter, λ, ranging from 0.0001 to 0.1. Please note, that such value for the polynomial is not the limited one - the only limitation is the amount of memory on your machine. The results of the runs, for *MSE*s and R^2 for two "extreme" cases - 10×10 and 100×100 are summarized in the tables 4.1-4.3. **Note**: I didn't save these values inside "txt" files, instead they appear directly as a print statements in the console.

Regression Type				
Polynomial Degree	Linear MSE, R^2	Ridge MSE, R^2	Lasso MSE, R^2	Type
1	0.0304, 0.659 0.0304, 0.659	0.0304, 0.659 0.0304, 0.659	0.0304, 0.659	Scikit Learn Manual
2	0.0259, 0.709 0.0259, 0.709	0.0259, 0.709 0.0259, 0.709	0.0259, 0.709	Scikit Learn Manual
3	0.0153, 0.828 0.0153, 0.828	0.0153, 0.828 0.0153, 0.828	0.0153, 0.828	Scikit Learn Manual
4	0.0153, 0.828 .0153, 0.828	0.0153, 0.828 0.0153, 0.828	0.0159, 0.821	Scikit Learn Manual
5	0.0132, 0.851 0.0132, 0.851	0.0134, 0.849 0.0134, 0.849	0.0156, 0.824	Scikit Learn Manual
6	0.0103, 0.884 0.0103, 0.884	0.0118, 0.8676 0.0118, 0.8676	0.01091, 0.877	Scikit Learn Manual
7	0.0075, 0.9158 0.00998, 0.888	0.01060, 0.8813 0.01060, 0.8786	0.01740, 0.8126	Scikit Learn Manual
8	0.01022, 0.8899 0.01022, 0.8899	0.01118, 0.87967 0.01118, 0.87967	0.01678, 0.81944	Scikit Learn Manual

Table 4.1: Comparison of results for the fitting of several different regression models to the generated data set. Grid is 10×10 points and $\lambda = 0.0001$.

Regression Type				
Polynomial Degree	Linear MSE, R^2	Ridge MSE, R^2	Lasso MSE, R^2	Type
1	0.0336, 0.637 0.0336, 0.637	0.0336, 0.637 0.0336, 0.637	0.0336, 0.637	Scikit Learn Manual
2	0.0271, 0.707 0.0271, 0.707	0.0271, 0.707 0.0271, 0.707	0.0271, 0.707	Scikit Learn Manual
3	0.018, 0.805 0.018, 0.805	0.018, 0.805 0.018, 0.805	0.0186, 0.799	Scikit Learn Manual
4	0.0142, 0.846 0.0142, 0.846	0.0142, 0.846 0.0142, 0.846	0.0188, 0.7975	Scikit Learn Manual
5	0.0122, 0.868 0.0122, 0.868	0.0122, 0.868 0.0122, 0.868	0.019, 0.7952	Scikit Learn Manual
6	0.0111, 0.8800 0.0111, 0.8800	0.01159, 0.8752 0.01159, 0.8752	0.01817, 0.8044	Scikit Learn Manual
7	0.01050, 0.8870 0.01050, 0.8870	0.01127, 0.8813 0.01127, 0.8786	0.01740, 0.8126	Scikit Learn Manual
8	0.01119, 0.894 0.01120, 0.8894	0.01125, 0.893 0.0154, 0.87967	0.0154, 0.8266	Scikit Learn Manual

Table 4.2: Comparison of results for the fitting of several different regression models to the generated data set. Grid is 100×100 points and $\lambda = 0.0001$.

The first set of values obtained through manual algorithm implementation, while the second row is the comparison to the Scikit Learn. As I've already mentioned before, I have not implemented manual algorithm for Lasso regression, but instead was using only scikit learn functionalities. As we can see, they are almost identical, with the Lasso algorithm being slightly behind. The value $\lambda = 0.0001$ is empirical.

Such high values of R^2 squared (and low MSE values) are generally a good thing. However, the danger here is that it can be also the sign of the model overfitting. Indeed, in figures (4.1)-(4.6), you can see the corresponding surfaces of real data, predicted values via manual algorithm and also Scikit Learn analog. The top row shows underfitting, instead of a complex surface, we have a straight one. It improves later on, the more complex our model becomes. However, at some point the model starts overfit the data. This feature is quite evident on both data sets, i.e. on the bottom plot - on the endges our models started to fit noise.

The corresponding beta parameters for Linear and Ridge regressions are plotted on figures 4.7 and 4.8. The left column is calculated for the 10×10 grid, whereas the right one has 100×100 points. In my opinion, it is clear from these plots, that the more points we have in the data set, the less uncertain we be about our model parameters.

Unfortunately, all these values cannot say for sure whether we will fit future data sets correctly (our main goal is to create such model) or not. That is why, as was discussed in the previous sections, we need to consider a resampling techniques.

4.1.2 Resampling techniques: Bias-variance tradeoff

In section 2 I have described implementation of kFold Cross Validation Algorithm (with $k = 5$). The part of the code rsponsible for it is the only part so far which is using Parallelization with joblib to speed-up the process.

Figures 4.9-4.11 shows the MSEs for the training and test data sets respectively for several different grids as a function of polynomial degree (model complexity). The bias-variance trade-off is evident on the top plots (for each of the model). With lesser amount of points it becomes harder to fit the data, that is why the test curve diverges from the train one starting from polynomial of degree 3.

Figures 4.10-4.11 also shows the dependence on the hyper parameter for each regression type. As we can see, the smaller λ becomes, the slopes are to the ones computed with OLS. Conversely, the increase in lambda parameter, will make the slope more flat, and that is because if we set $\lambda \to \infty$, $\beta = 0$.

Figures 4.12-4.14 are plotting bias and varianse of the testing data set, resulted with kFold CV. The reason why bias is the straight line is unclear to me. Also, although variance is increasing with the model complexity, as it should, I don't really understand why it reaches some constant slope and stays there. Unfortunately, I couldn't resolve the issue no matter how hard I tried, and so, eventually, I decided to let it be.

4.2 Real Data

For the real data set we need to repeat the same things we did for the fake one. However, here it is useful to take only a part of the terrain data to train and test your model on.

The procedures for calculating values for real data set is the same as for the manually generated one. The only problem here is that to run the entire data set for 5 different consecutive polynomial degrees, took me about 7 hours. To be more precise, the console output showed:

```
-- Program finished at 26326.446545124054 sec --
```

Below I discuss this in a more detail.

4.2.1 Entire Data set

For this run, I am plotting the resulting fitted surfaces in figures 4.22-4.24. The entire file consists of $3601 \times 1801 = 6485401$ points in total.

Lets take a look for MSE resulting from Cross Validation - figures 4.20-4.25. As we can see, both test and training data sets in ideal sync, which tells us that the amount of points in the system is so large, that the slopes will not reach white noise threshold in the "near future". Unfortunately, the Bias-Variance plots 4.17 also shows the same weird behavior as with the pre-generated data set.

Now, I was not able to test the program for the larger polynomials, but I am confident enough to say it will not show any sigh of overfitting up to polynomial of degree 20.

The beta values with their confidence intervals are plotted in figures 4.15-4.16. Unfortunately, it is hard for me to tell whether these values are correct or not, or why is there this bump for the last coefficient looking on the plots themselves. My idea is that it is because I didn't normalize my data, before actually fitting it, but it is still needs to be tested.

Anyway, I think it is pretty safe to say that for the entire data set, the *best optimal model will be OLS*.

4.2.2 Portion of Dataset

As, I've already stated, the last version of the program allows you to decide on which % of the data set to use. Following some of the students suggestions, I decided to use the 5% of the entire data set, i.e. $181 \times 1801 = 325981$ points. Although, the total amount of points is significantly lesser, it is still pretty huge system to look into.

As we saw, for the system of 10000 (100×100 grid), we start overfitting the model only at polynomial of 8 degree and higher. Now, we have 30 times more data points, which means we need to significantly increase the degree of polynomial to fit in.

I tried to run the code with max polynomial of 20. However, with the increase of polynomial degree, the model started to give completely incorrect results. For instance, figure 4.18 shows the 3d surface visualization for several different polynomials. Until polynomial of 7th degree, the Linear regression model was fitting perfectly, but as it clearly seen, we get completely incorrect results. Another indication of the wrong model - the negative R^2 values in the table 4.3 below.

The Cross Validation MSEs for test and train data sets for a given polynomial degree can be found in the plot 4.19. Once again, we see that the two slopes are perfectly aligned which tells us we won't reach over fitting with this amount of points. So, my understanding is that I need to decrease the amount of points even further down. The bias and variance executes the same weird behavior as in the previous cases, so I do not show them here. The same goes for betas (although the plots and the values are available inside "/Output/" folder under appropriate name).

It is still hard to tell, which of the models will be better to use. However, based on the R^2 and overall fitting performance, I would *prefer Ridge regression over Linear* one. For the Lasso regression my understanding is that I should tune the model with the different the λ parameter.

Regression Type				
Polynomial Degree	Linear MSE, R^2	Ridge MSE, R^2	Lasso MSE, R^2	Type
1	761.9179, 0.9720 761.9179, 0.9720	761.917, 0.972 761.917, 0.972	761.9179, 0.9720	Scikit Learn Manual
2	659.9133, 0.9757 659.9133, 0.9757	659.9133, 0.9757 659.9133, 0.9757	659.9133, 0.975	Scikit Learn Manual
3	278.997, 0.989 278.997, 0.989	278.997, 0.989 278.997, 0.989	287.389, 0.989	Scikit Learn Manual
4	234.27, 0.991 234.27, 0.991	234.278, 0.9913 234.278, 0.9913	263.1075, 0.990	Scikit Learn Manual
5	206.354, 0.992 206.354, 0.992	206.354, 0.992 206.354, 0.992	251.1531, 0.990	Scikit Learn Manual
7	67931.1, -1.496 78386, -1.880	133.822, 0.995 133.82, 0.995	234.29, 0.991	Scikit Learn Manual

Table 4.3: Comparison of results for the fitting of several different regression models to the real data set. The amount of points is 5% from the total one and $\lambda = 0.0001$.

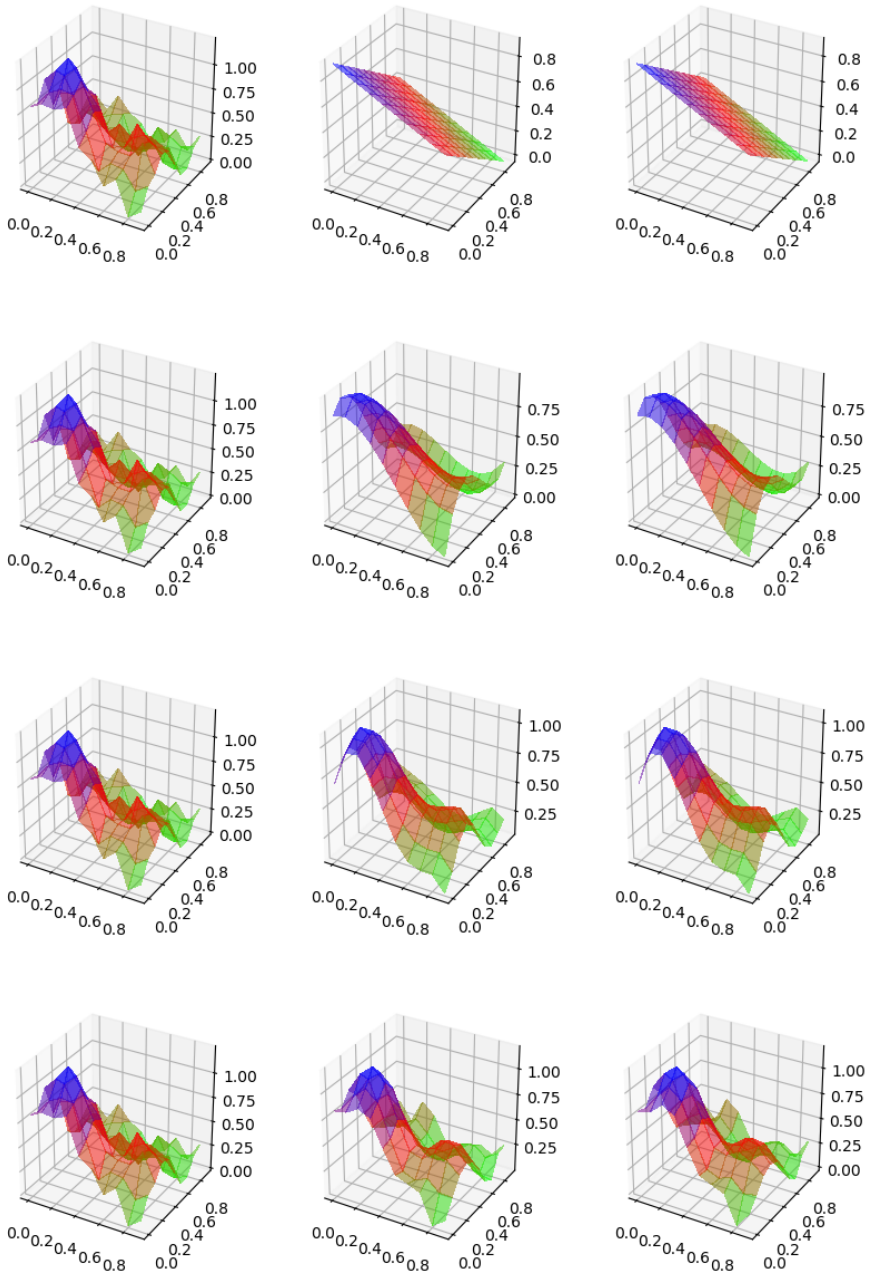


Figure 4.1: The 3D visualisation of the noisy data set (left column) and the predicted surface via Lasso regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 10×10 points.

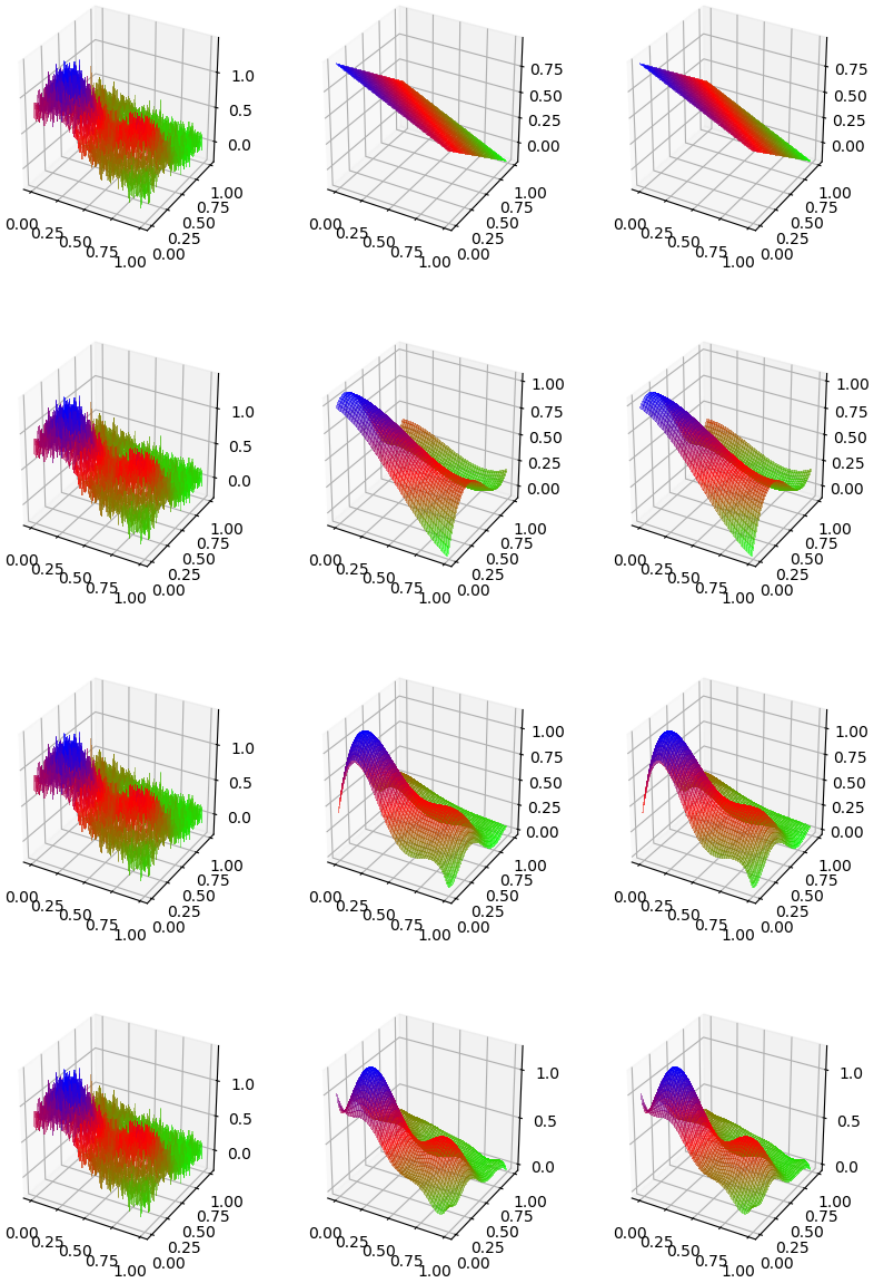


Figure 4.2: The 3D visualisation of the noisy data set (left column) and the predicted surface via Lasso regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 100 × 100 points.

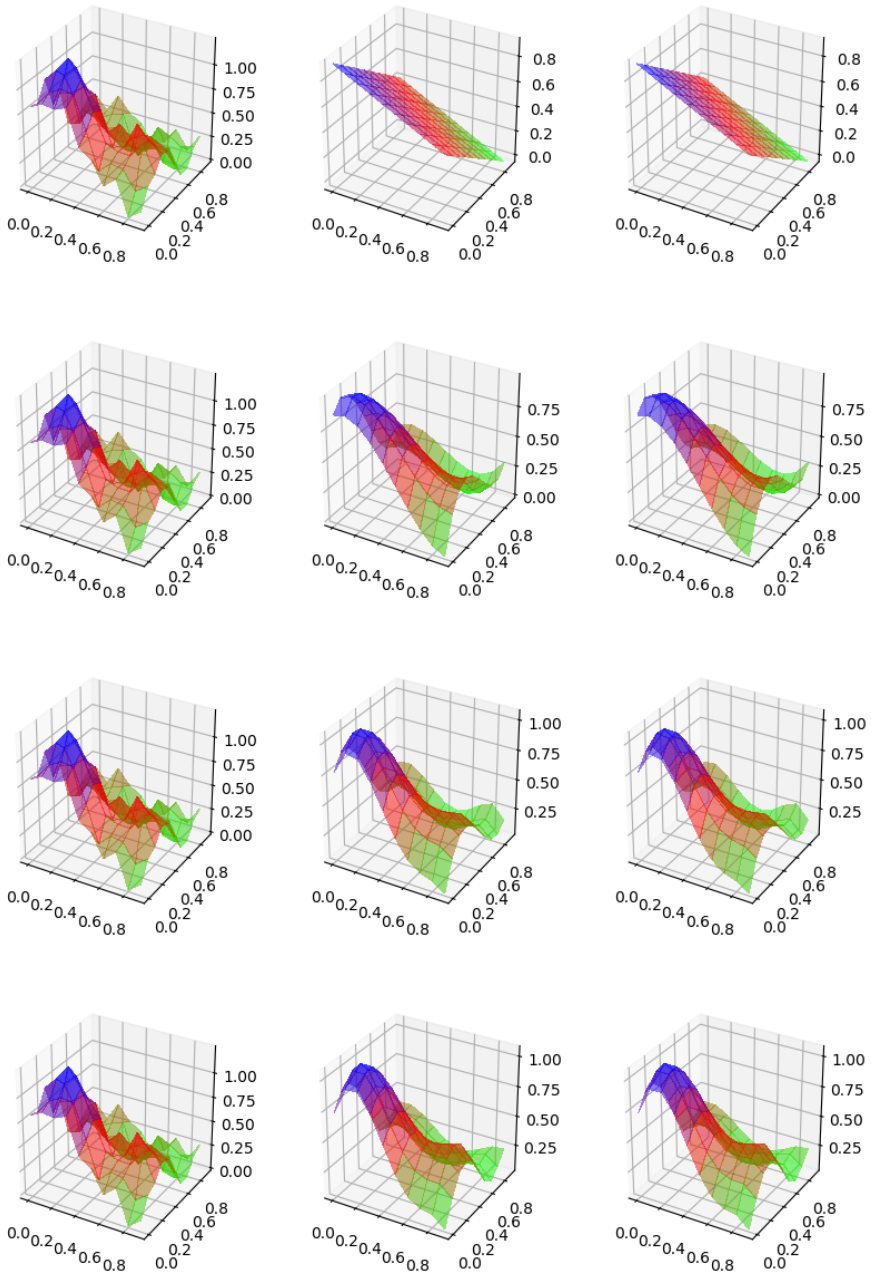


Figure 4.3: The 3D visualisation of the noisy data set (left column) and the predicted surface via Ridge regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 10×10 points and $\lambda = 0.0001$.

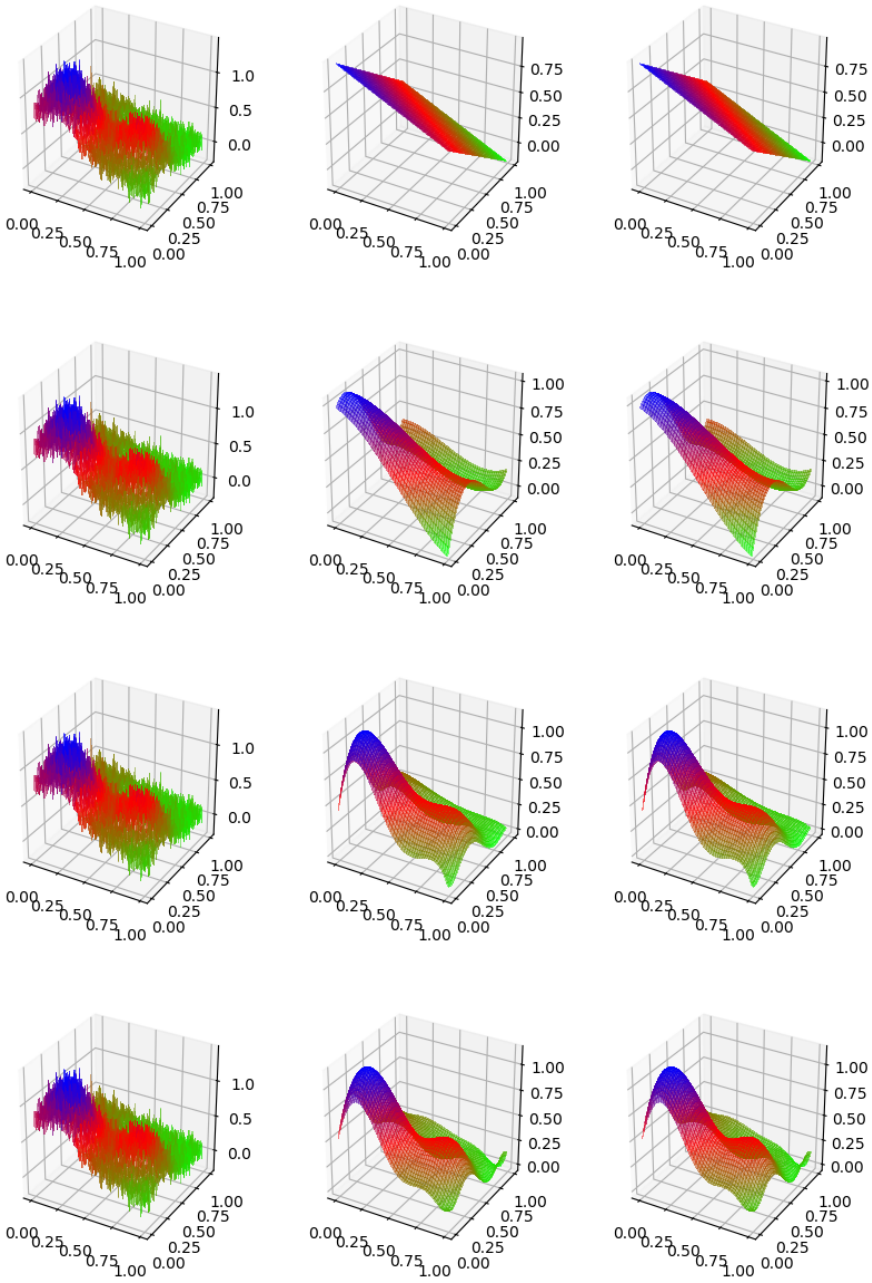


Figure 4.4: The 3D visualisation of the noisy data set (left column) and the predicted surface via Ridge regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 100×100 points and $\lambda = 0.0001$.

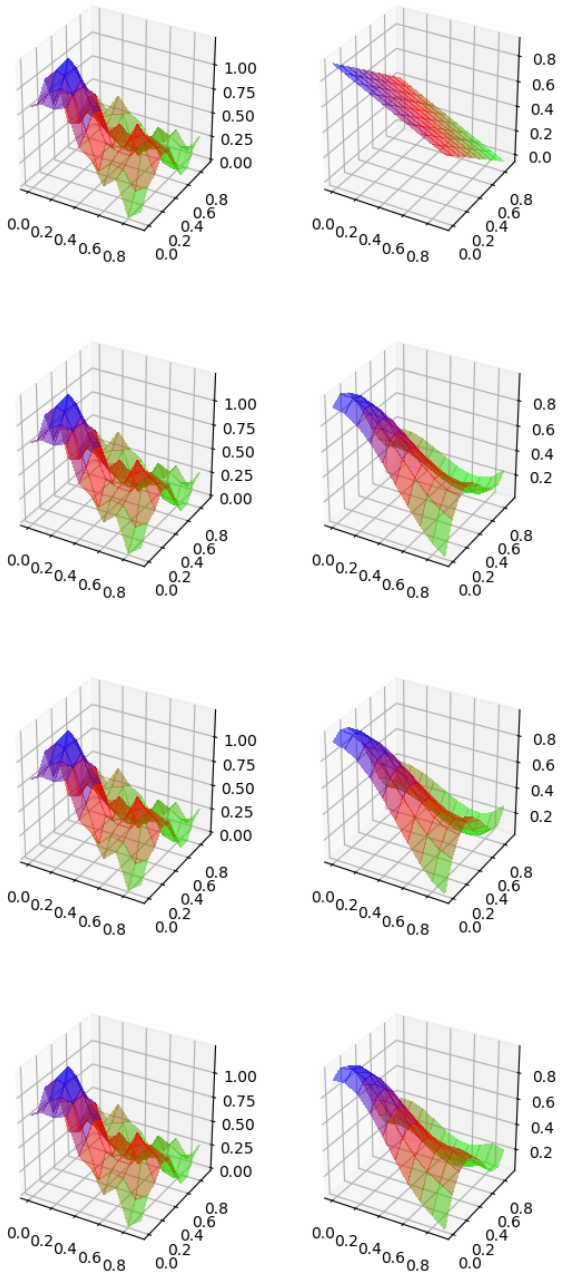


Figure 4.5: The 3D visualisation of the noisy data set (left column) and the predicted surface via Lasso regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 10×10 points and $\lambda = 0.0001$.

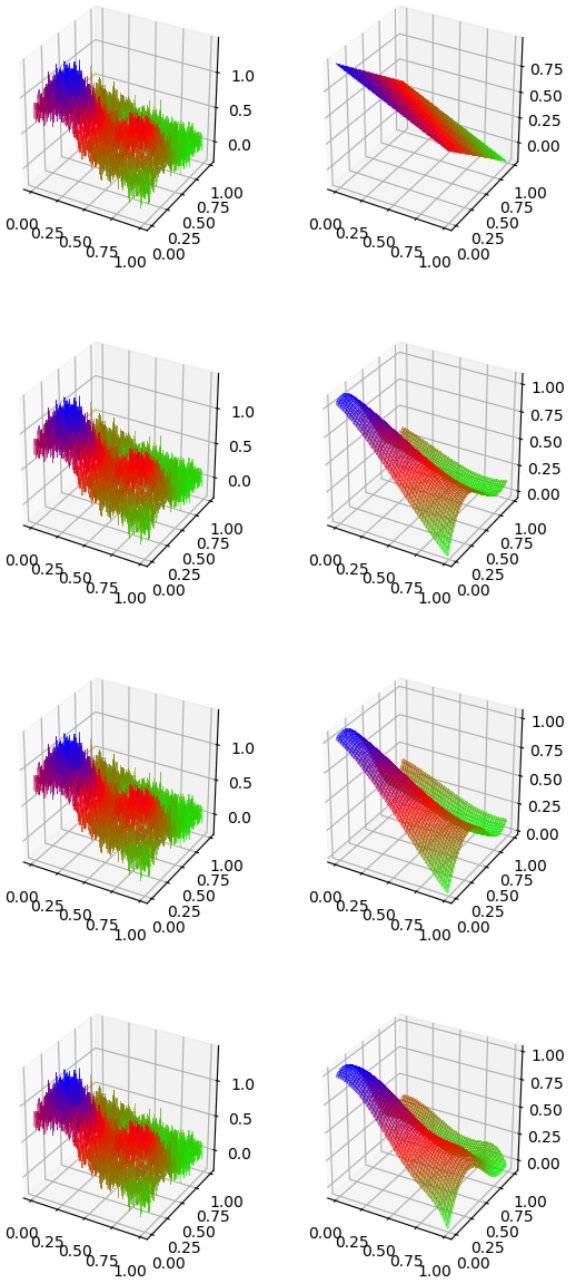


Figure 4.6: The 3D visualisation of the noisy data set (left column) and the predicted surface via Lasso regression (right column) for polynomials 1, 3, 5 and 8 (from top to bottom). Grid is 100×100 points and $\lambda = 0.0001$.

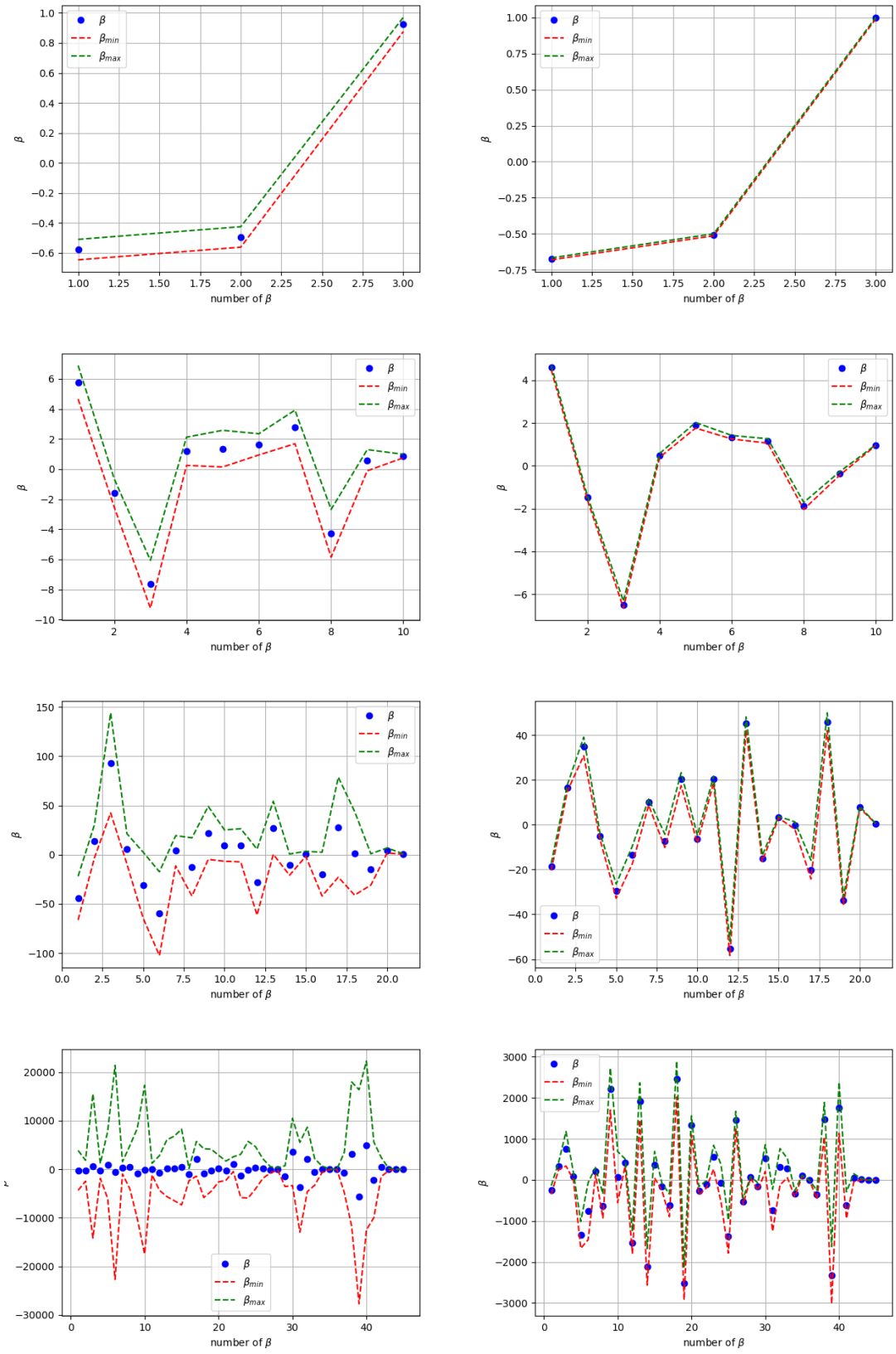


Figure 4.7: Manually calculated parameters β of Linear Regression model for polynomials 1, 3, 5 and 8 (from top to bottom) on the grids 10×10 (left column) and 100×100 (right column) points.

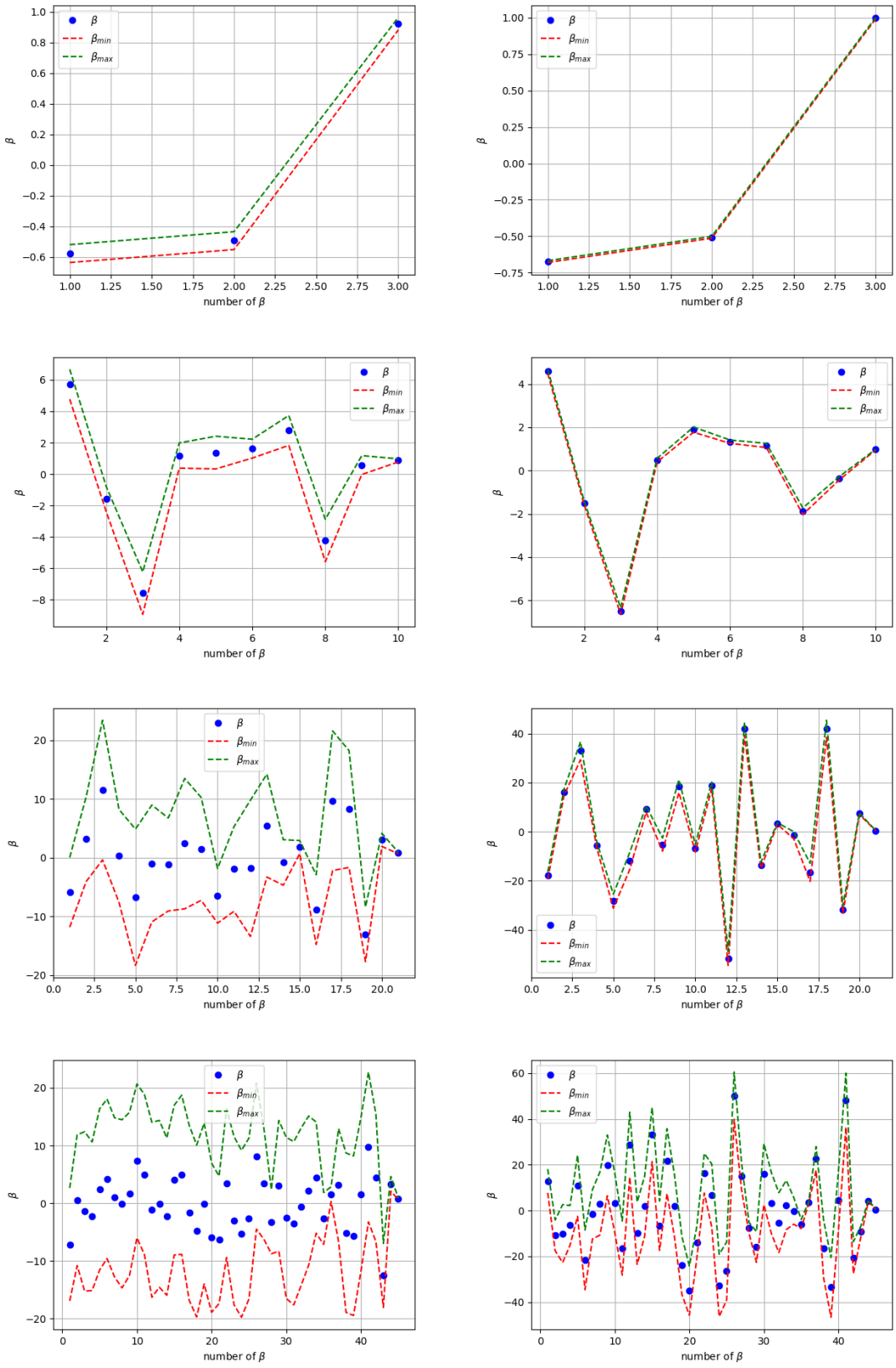


Figure 4.8: Manually calculated parameters β of Ridge Regression model for polynomials 1, 3, 5 and 8 (from top to bottom) on the grids 10×10 (left column) and 100×100 (right column) points. $\lambda = 0.0001$.

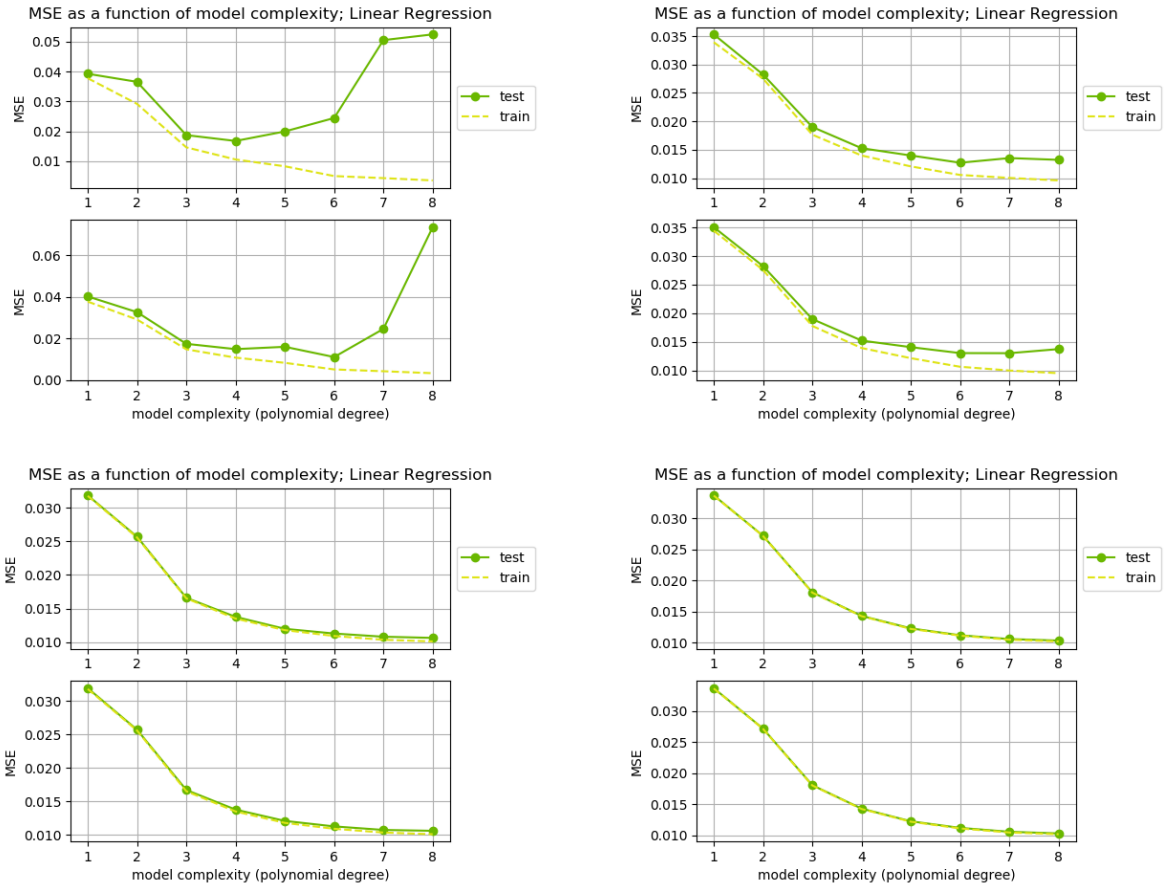


Figure 4.9: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Linear Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right).

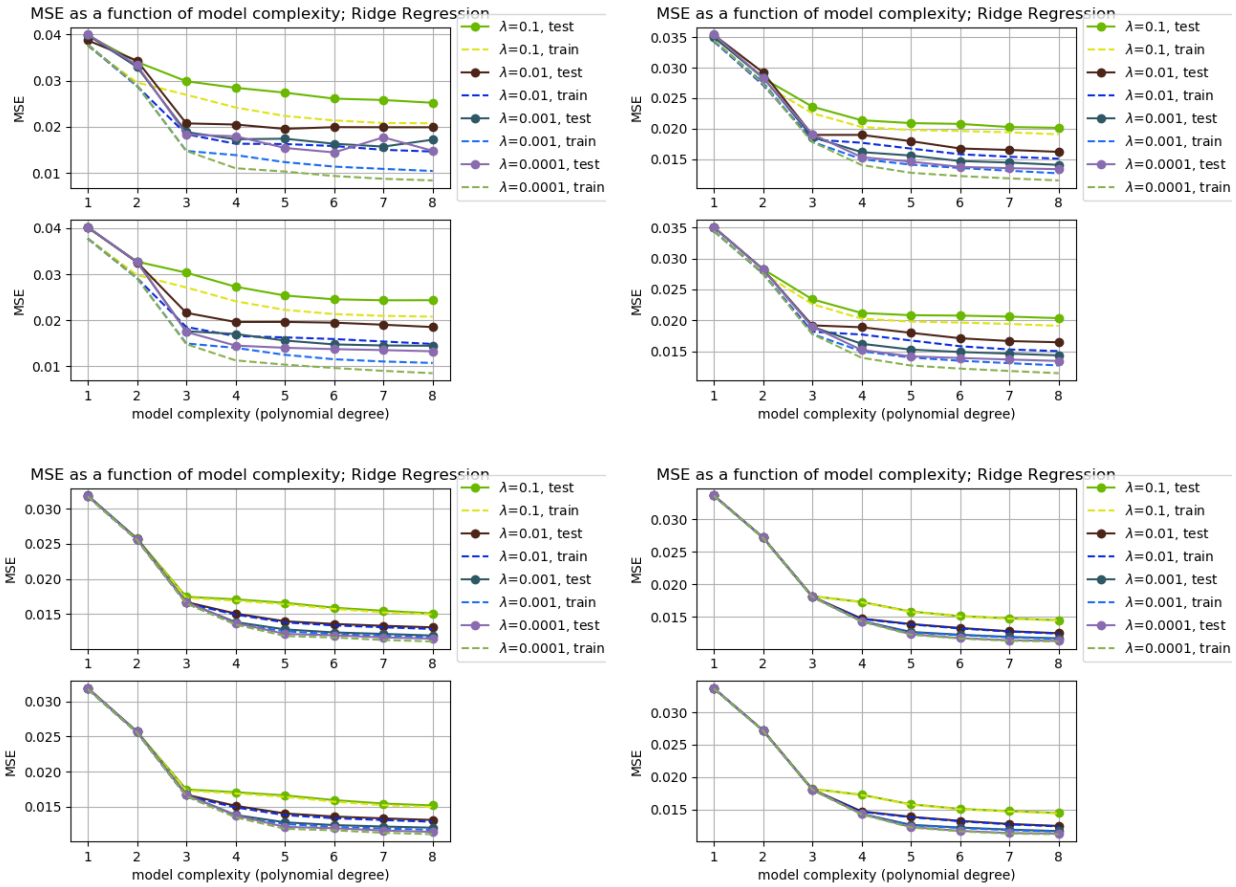


Figure 4.10: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Ridge Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

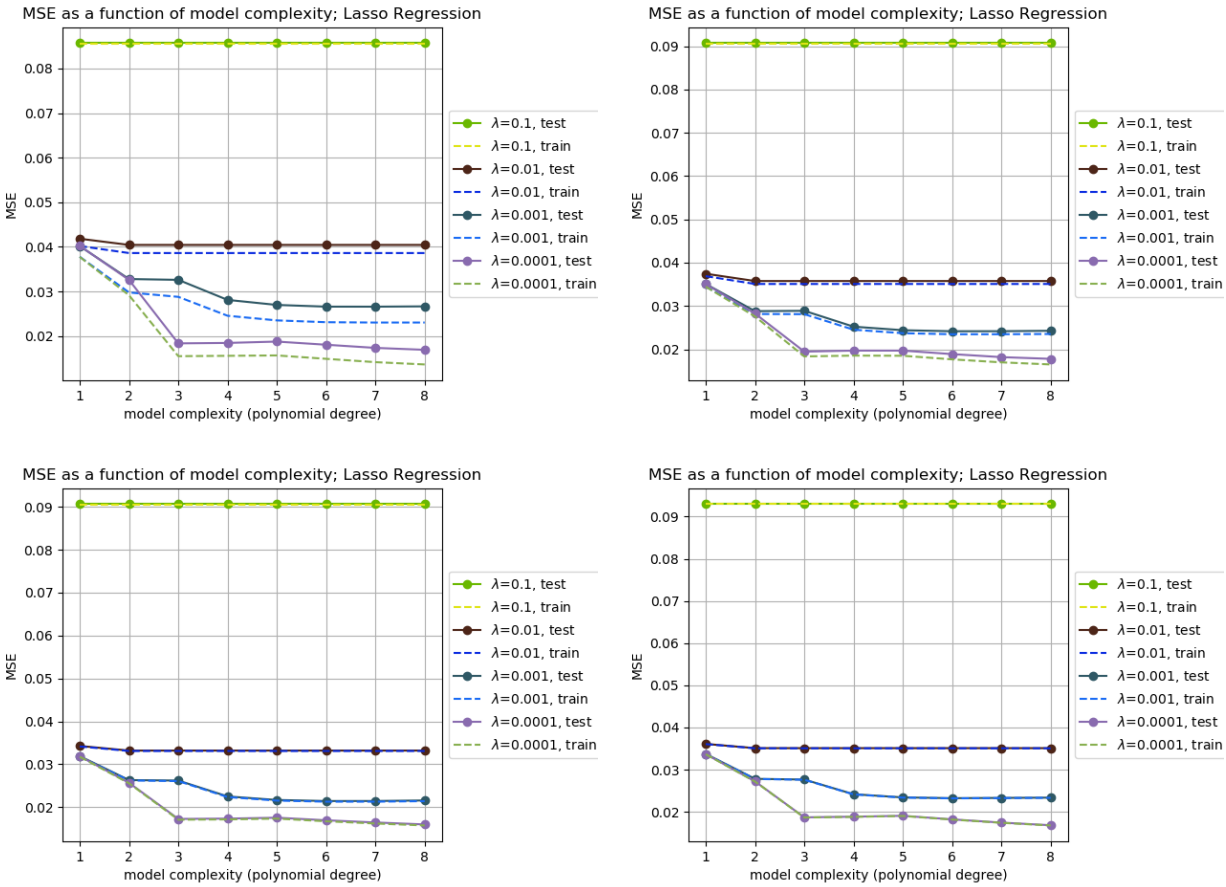


Figure 4.11: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Lasso Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

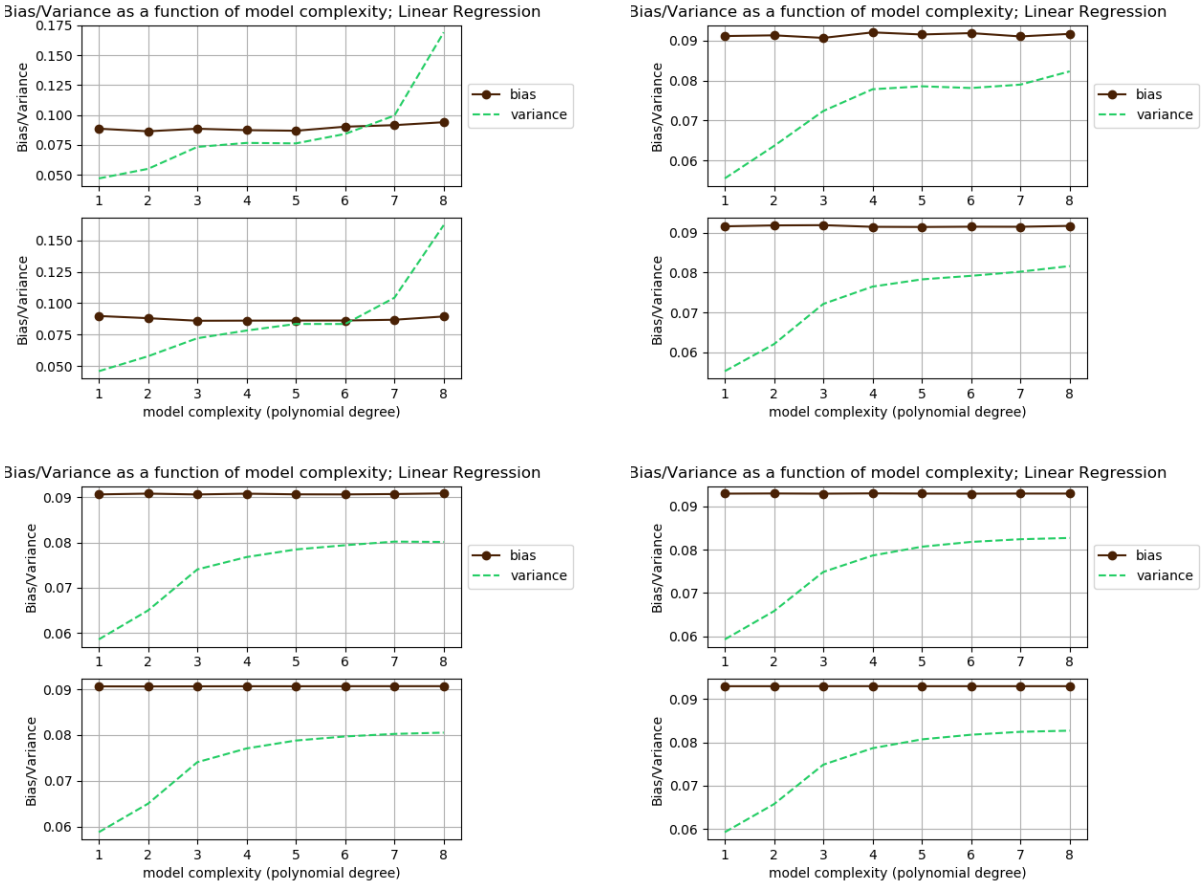


Figure 4.12: Plot of Bias and Variance (to illustrate bias-variance trade-off) as a function of model complexity (polynomial degree) calculated via Linear Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

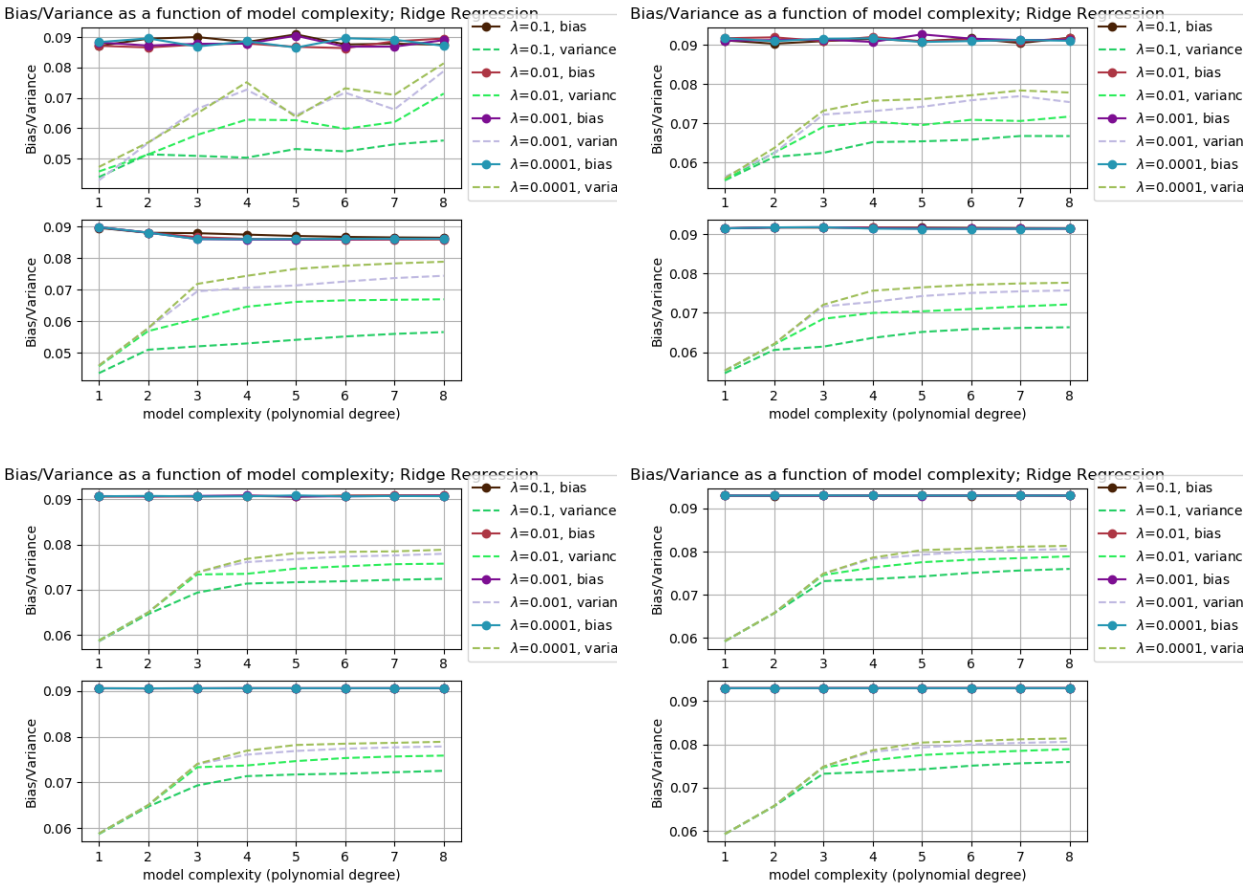


Figure 4.13: Plot of Bias and Variance (to illustrate bias-variance trade-off) as a function of model complexity (polynomial degree) calculated via Ridge Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

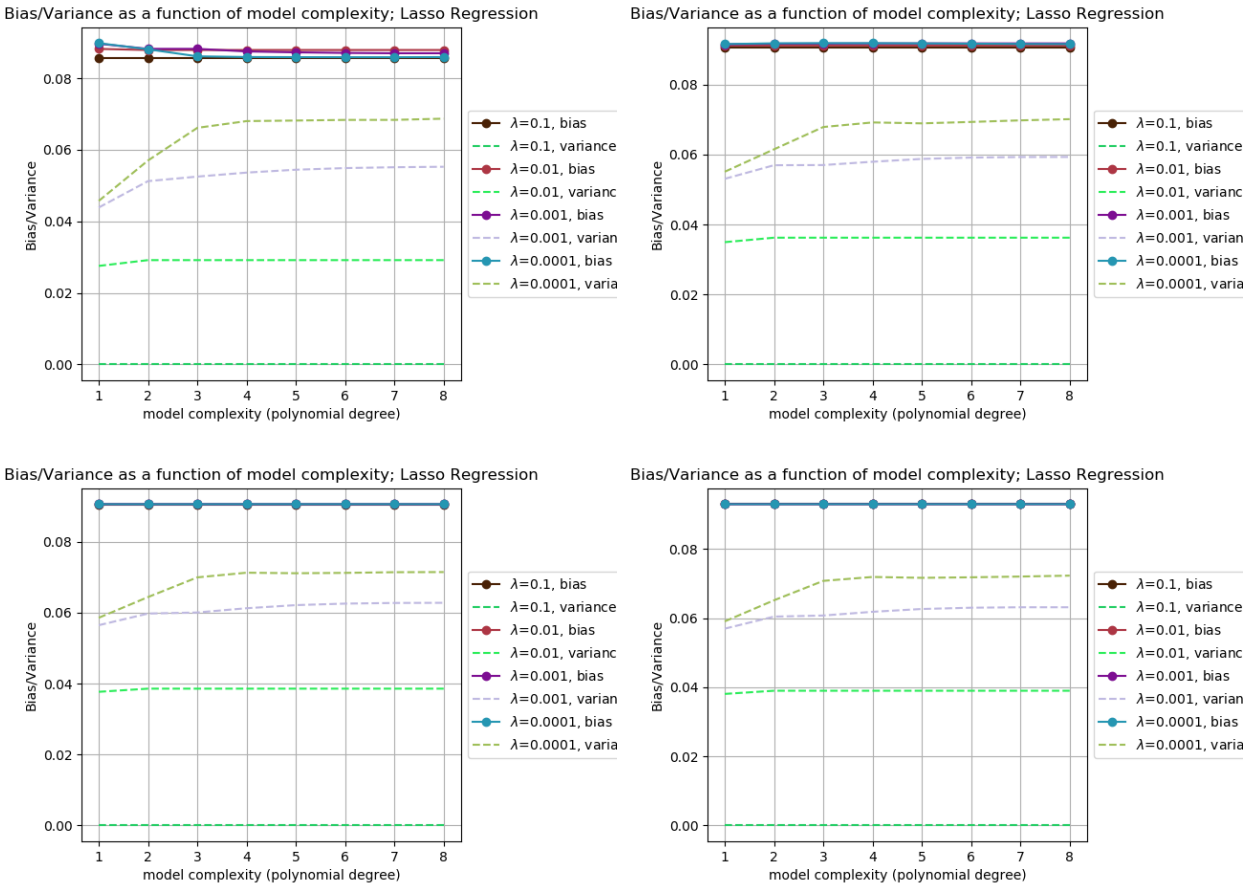


Figure 4.14: Plot of Bias and Variance (to illustrate bias-variance trade-off) as a function of model complexity (polynomial degree) calculated via Lasso Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

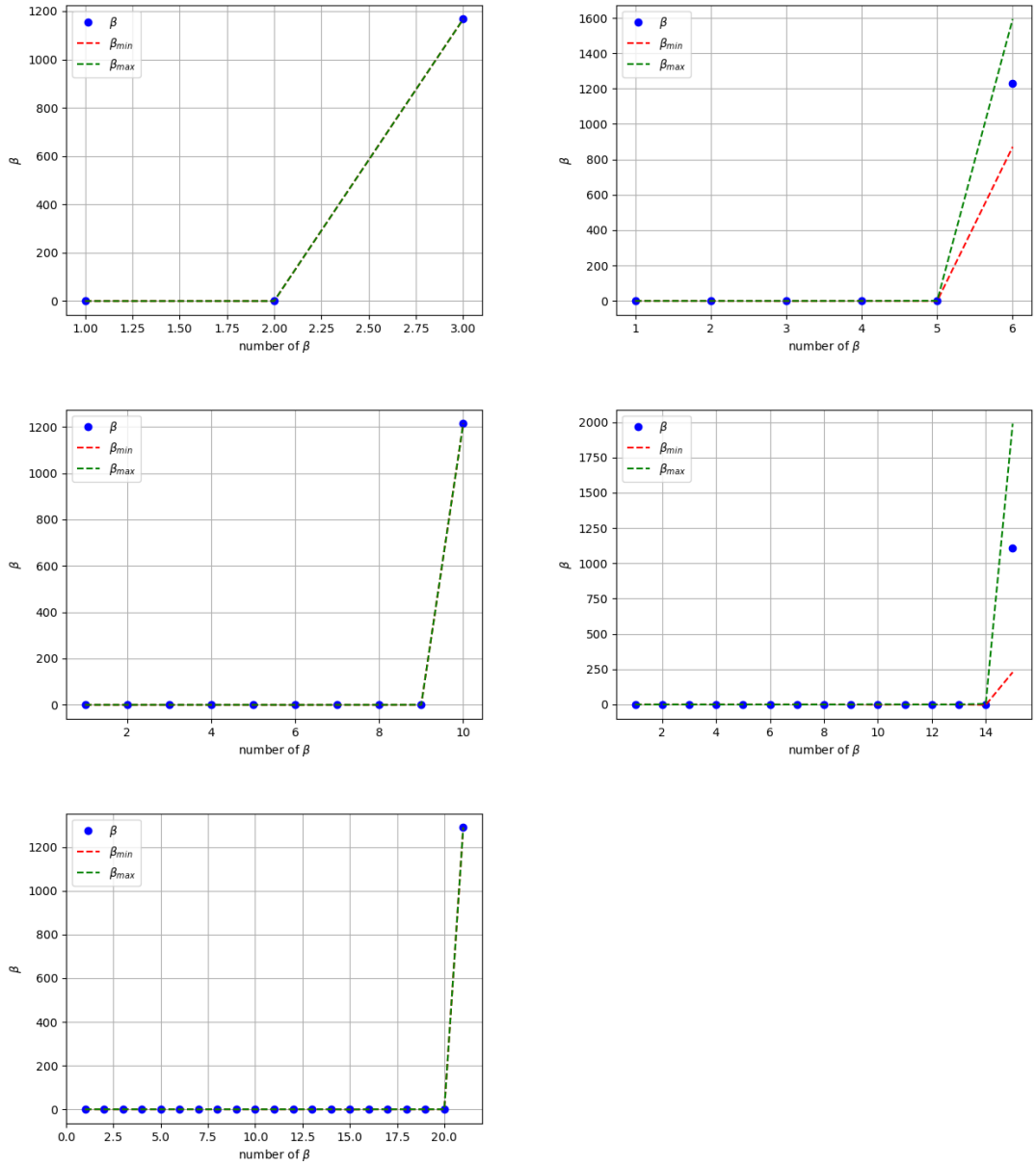


Figure 4.15: Manually calculated parameters β of Linear Regression model for polynomials 1, 2, 3, 4 and 5 (from top to bottom) on the entire data set ($3601 \times 1801 = 6485401$ points).

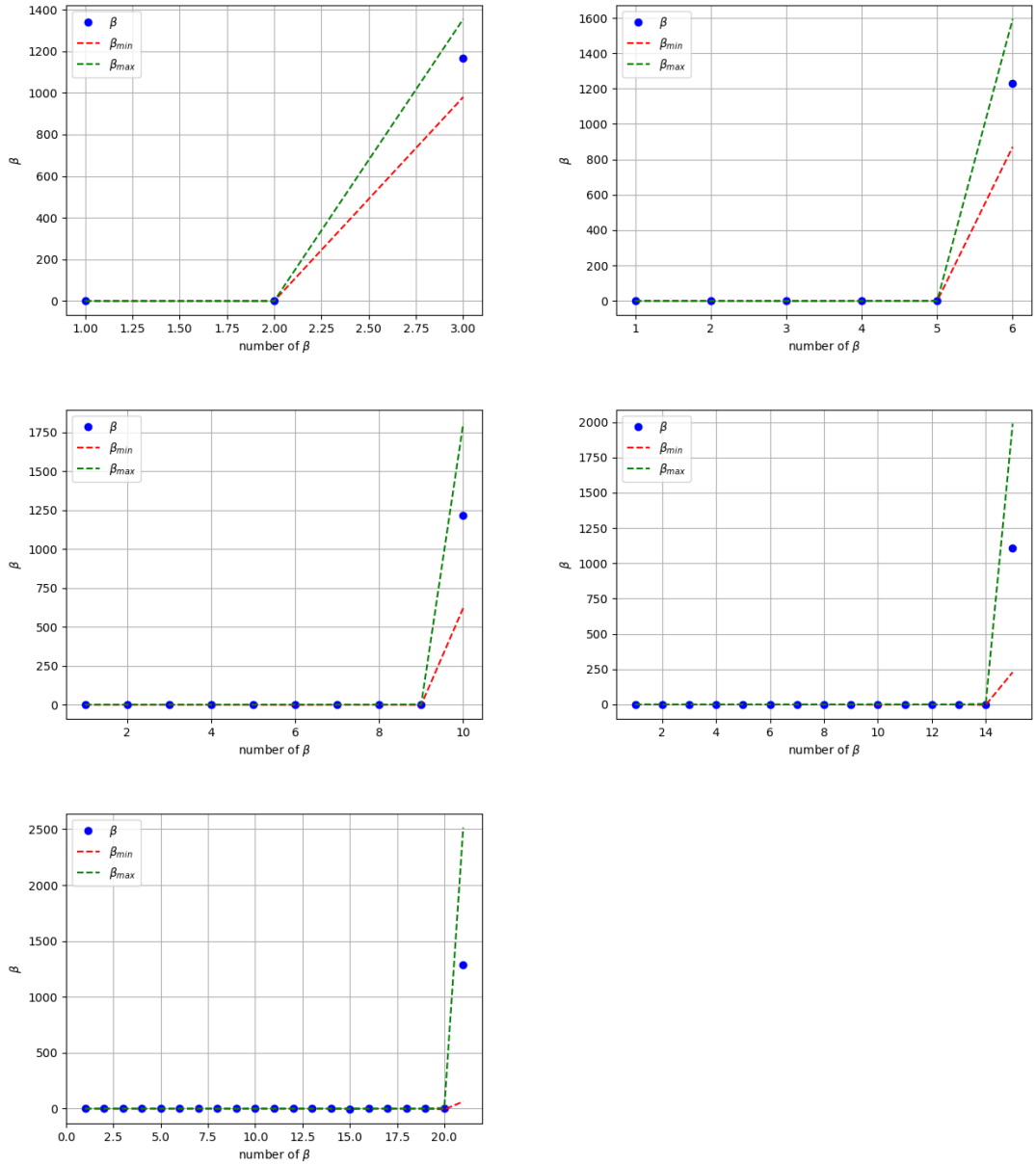


Figure 4.16: Manually calculated parameters β of Linear Regression model for polynomials 1, 2, 3, 4 and 5 (from top to bottom) on the entire data set ($3601 \times 1801 = 6485401$ points). $\lambda = 0.0001$

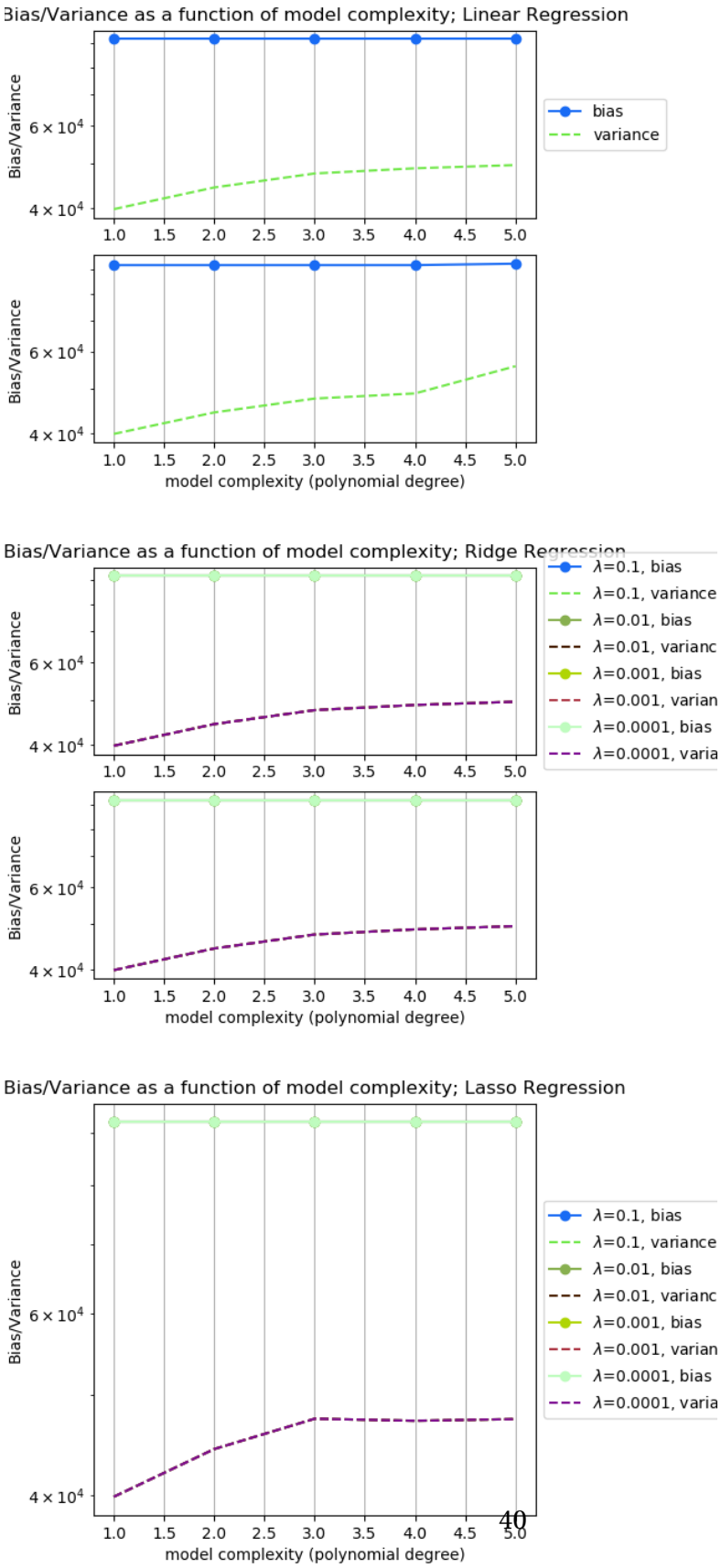


Figure 4.17: Plot of Bias and Variance (to illustrate bias-variance trade-off) as a function of model complexity (polynomial degree) calculated via Linear, Ridge and Lasso Regressions (from top to bottom) for entire data set ($3601 \times 1801 = 6485401$ points). $\lambda_i = 0.1$, 0.01 and 0.0001.

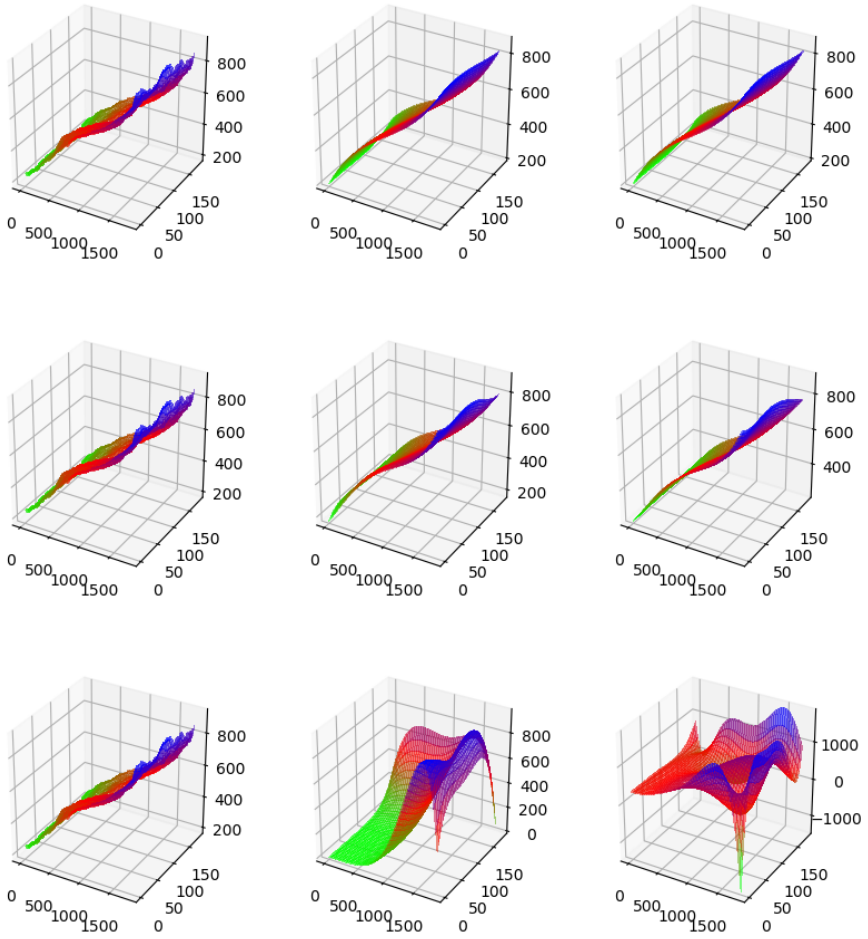


Figure 4.18: The 3D visualisation of the 5% of the real noisy data set (left column) and the predicted surface via Linear regression (right column) for polynomials 3, 5, 7 (from top to bottom).

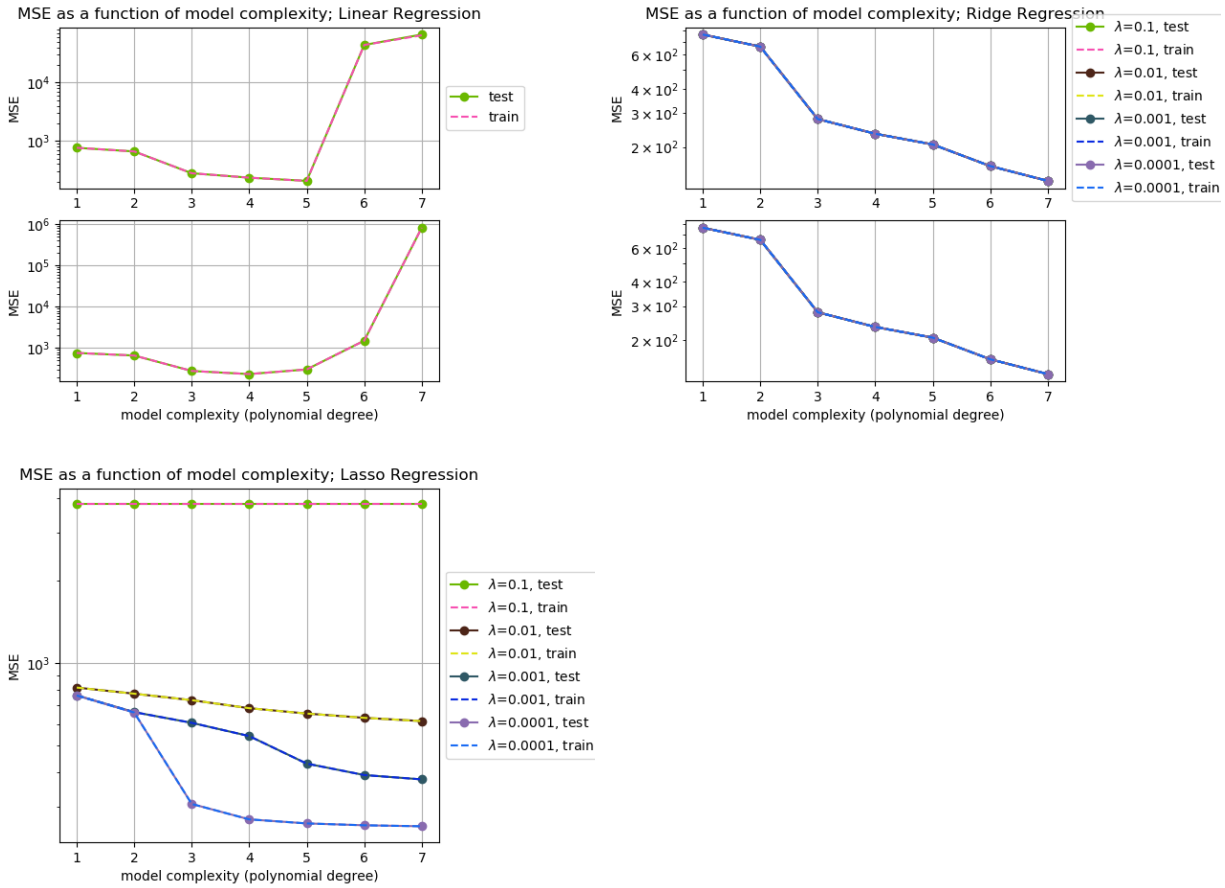


Figure 4.19: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Linear Ridge and Lasso Regression on the 5% of th real data set (bottom right).

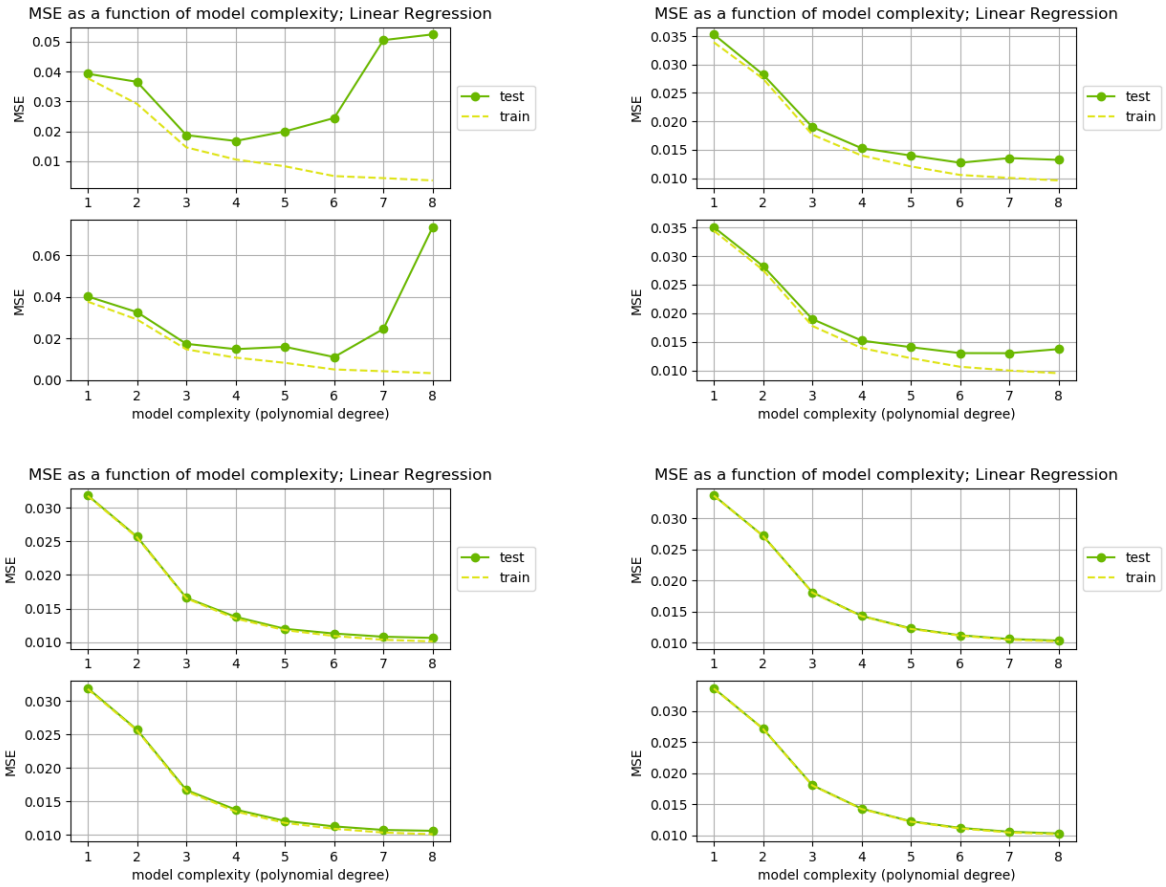


Figure 4.20: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Linear Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right).

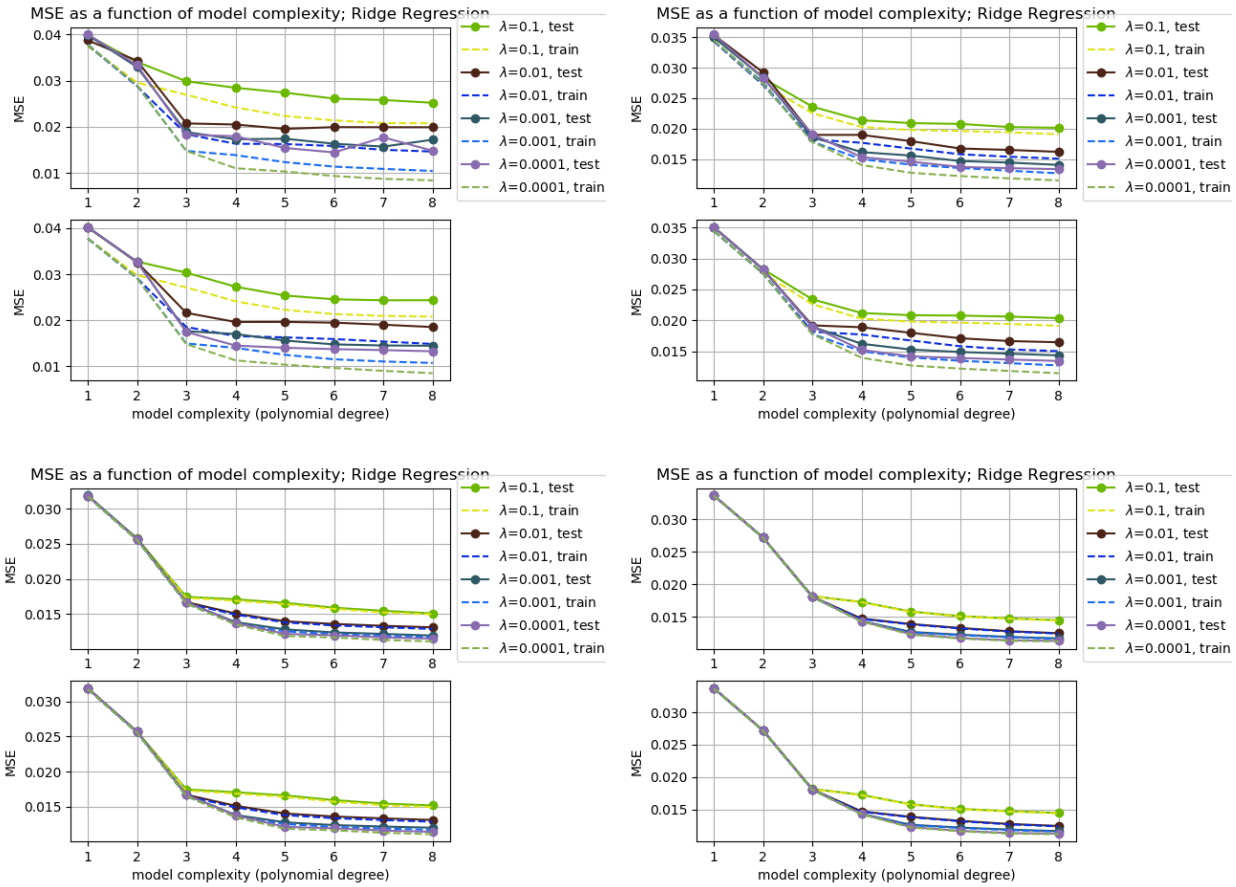


Figure 4.21: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Ridge Regression for several different grids: 10×10 (top left), 21×21 (top right), 50×50 (bottom left), 100×100 (bottom right). $\lambda_i = 0.1$, 0.01 and 0.0001.

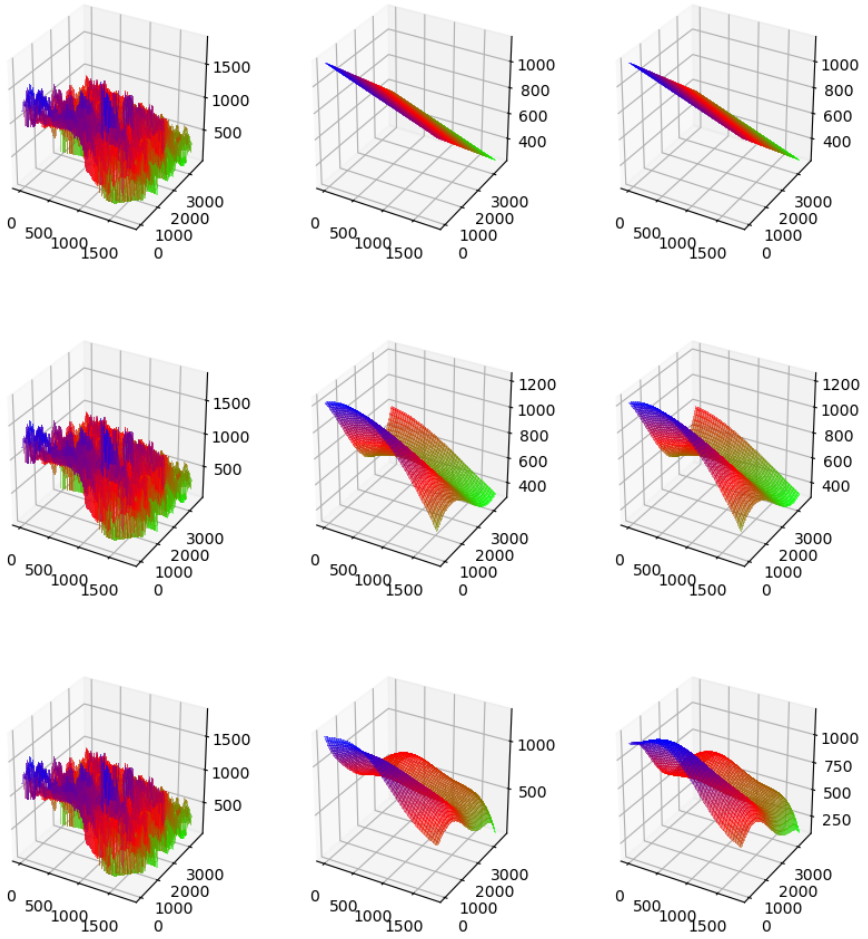


Figure 4.22: The 3D visualisation of the real noisy data set (left column) and the predicted surface via Linear regression (right column) for polynomials 1, 3, 5 (from top to bottom).

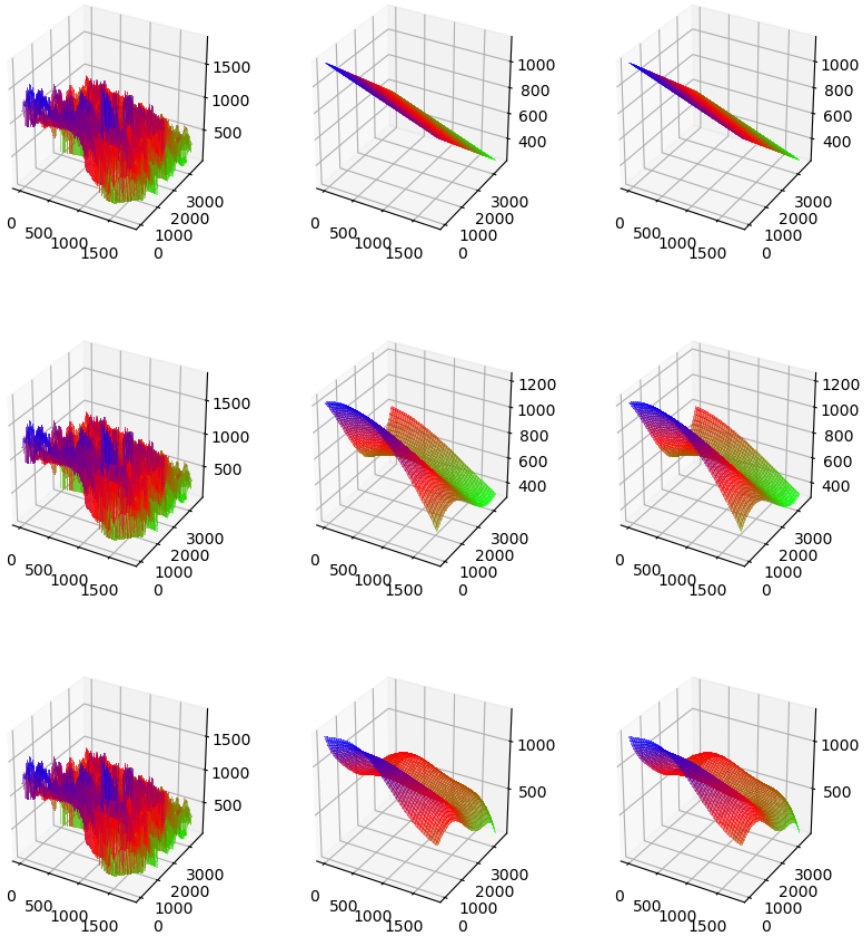


Figure 4.23: The 3D visualisation of the real noisy data set (left column) and the predicted surface via Ridge regression (right column) for polynomials 1, 3, 5 (from top to bottom). $\lambda = 0.0001$.

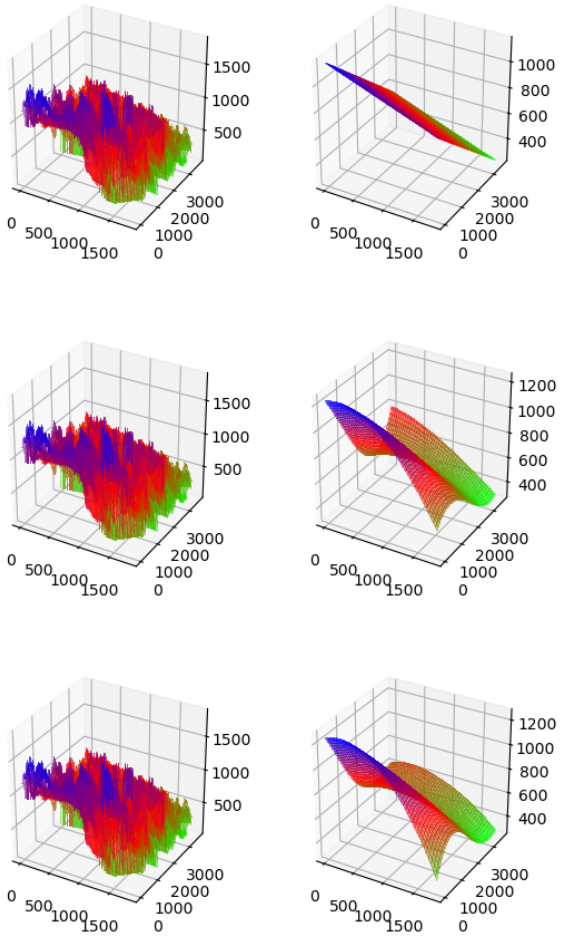


Figure 4.24: The 3D visualisation of the noisy real data set (left column) and the predicted surface via Lasso regression (right column) for polynomials 1, 3, 5 (from top to bottom). $\lambda = 0.0001$.

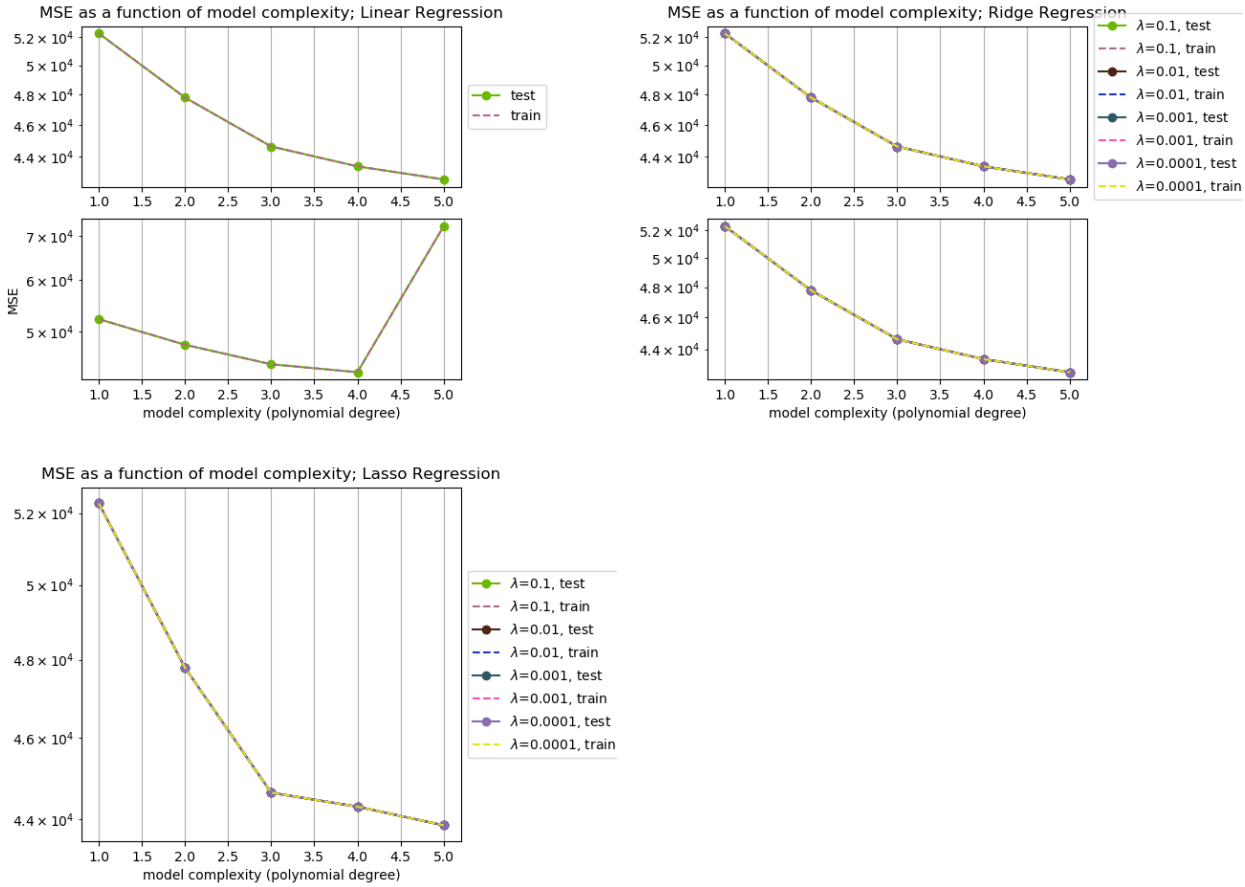


Figure 4.25: MSE plot of train and test data sets as a function of model complexity (polynomial degree) calculated via Linear (top left), Ridge (top right) and Lasso (bottom left) Regressions for the entire dataset. $\lambda_i = 0.1$, 0.01 and 0.0001.

5 CONCLUSIONS

During this project, I have written a self-consistent code, which implements several regression algorithms on both pre-generated data and real data. Moreover, it also computes kFold cross validation (with k=5) with manual code on both Linear and ridge Regressions and also computes the code with the use of Scikit learn standard methods for every regression method discussed in this report.

As outputs, program creates a bunch of "png" and "txt" files which are stored inside Output folder (in the directory where script is located).

Because I haven't accounted for the amount of points present in the real data set, I wasted a lot of time waiting for the script to finish its work. Therefore, I enabled the possibility to use only part of the data set (e.g. 10%). However, based on what I saw so far, for the system with very *large amount of points*, *OLS* would be more than enough to fit the model, whereas for *lesser amount of points* (when bias-variance trade of is quite prominent), the better model is *Ridge* Regression. the problem with Lasso regression is that you need to be quite careful to tun your model with correct λ first.

Although the program runs without crashing, there are several possible improvements, which would be good to implement in observable future:

- Reduce the size of the data set without losing information. During one of the discussions on Piazza, one of the students mentioned possibility to reduce the size of the data set. He/she even mentioned the code snippet which can be used for this and, although I am not sure whether his/her solution is good, the idea itself is very good. If this code doesn't work, maybe I will try to implement something like Huffman Coding or similar thing;

- Better Parallelization. Although I have already implemented this feature, it only works now for calculation of kFold cross validation for various parameters of λ. I'd like it to be expanded on the entire script;

- One more possible solution I thought of to reduce the time, I can simply split the calculating of KFolds and Regression analysis into separate runs. For instance, right now I am running the entire pipeline, i.e. Polynomial, Ridge, Lasso regressions with its respective Scikit learn analogs. So the idea is to be able to run the code for only one or two methods at once, and if, the user really wants to, run the entire code with all methods implemented;

REFERENCES

[1] Aurelien Geron, Hands-On Machine Learning with Scikit-Learn and TensorFlow, O'Reilly

[2] Morten Hjorth-Jensen, "Lectures Notes in FYS-STK4155. Data Analysis and Machine Learning: Linear Regression and more Advanced Regression Analysis", (2019) Unpublished

[3] Trevor Hastie, Robert Tibshirani, Jerome H. Friedman, The Elements of Statistical Learning, Springer

CODE

Listing 1: "Small library"

```
import numpy as np
# for polynimial manipulation
import sympy as sp
# from sympy import *
import itertools as it
# for plotting stuff
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Scikit learn utilities
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score


class RegressionLibrary:
    '''
    class constructor
    '''

    def __init__(self, *args):
        # getting input values in place
        self.x_symb = args[0]
        self.x_vals = args[1]

    '''
    Franke function, used to generate outputs (z values)
    '''
    def FrankeFunction(self, x, y):
        term1 = 0.75 * np.exp(-(0.25 * (9 * x - 2) ** 2) - 0.25 * ((9 * y - 2) **
            ↪ 2))
        term2 = 0.75 * np.exp(-((9 * x + 1) ** 2) / 49.0 - 0.1 * (9 * y + 1))
        term3 = 0.5 * np.exp(-(9 * x - 7) ** 2 / 4.0 - 0.25 * ((9 * y - 3) ** 2))
        term4 = -0.2 * np.exp(-(9 * x - 4) ** 2 - (9 * y - 7) ** 2)
        return term1 + term2 + term3 + term4

    '''
    Generating polynomials for given number of variables for a given degree
    using Newton's Binomial formula, and when returning the design matrix,
    computed from the list of all variables
    '''

    def constructDesignMatrix(self, *args):
        # the degree of polynomial to be generated
        poly_degree = args[0]
        # getting inputs
        x_vals = self.x_vals
        # using itertools for generating all possible combinations
        # of multiplications between our variables and 1, i.e.:
        # x_0*x_1*1, x_0*x_0*x_1*1 etc. => will get polynomial
        # coefficients
        variables = list(self.x_symb.copy())
```

```
        variables.append(1)
        terms = [sp.Mul(*i) for i in it.combinations_with_replacement(variables,
            ↪ poly_degree)]
        # creating desing matrix
        points = len(x_vals[0]) * len(x_vals[1])
        # creating desing matrix composed of ones
        X1 = np.ones((points, len(terms)))
        # populating design matrix with values
        for k in range(len(terms)):
            f = sp.lambdify([self.x_symb[0], self.x_symb[1]], terms[k], "numpy")
            X1[:, k] = [f(i, j) for i in self.x_vals[1] for j in self.x_vals[0]]
        # returning constructed design matrix (for 2 approaches if needed)
        return X1

    '''
    Singular Value Decomposition for Linear Regression
    '''

    def doSVD(self, *args):
        # getting matrix
        X = args[0]
        # Applying SVD
        A = np.transpose(X) @ X
        U, s, VT = np.linalg.svd(A)
        D = np.zeros((len(U), len(VT)))
        for i in range(0, len(VT)):
            D[i, i] = s[i]
        UT = np.transpose(U)
        V = np.transpose(VT)
        invD = np.linalg.inv(D)
        invA = np.matmul(V, np.matmul(invD, UT))

        return invA

    '''
    k-Fold Cross Validation
    '''
    # method for shuffling arrays randomly (but simultaneously <= we still have
        ↪ ordered pairs)
    def shuffleDataSimultaneously(self, a, b):
        assert len(a) == len(b)
        p = np.random.permutation(len(a))
        return a[p], b[p]

    # function to split data set manually
    def splitDataset(self, *args):
        # getting inputs
        X = args[0]
        z = args[1]
        kfold = args[2]
        iterator = args[3]
        # If the dataset does not cleanly divide by the number of folds,
        # there may be some remainder rows and they will not be used in the split.
        length = len(X) % kfold
        if length == 0:
            condition = True
        else:
            condition = False
        while condition is False:
            # removing the element <= they were shuffled randomly,
            # so it doesn't matter which one to remove
            X = np.delete(X, -1, axis = 0)
```

```
            z = np.delete(z, -1, axis = 0)
            # checking whether it is divided cleanly
            length = len(X) % kfold
            if length == 0:
                condition = True
        # 2. Split the dataset into k groups:
        X_split = np.array_split(X, kfold, axis=0)
        z_split = np.array_split(z, kfold, axis=0)
        # train data set - making a copy of the shuffled and splitted arrays
        X_train = X_split.copy()
        z_train = z_split.copy()
        # test data set - each time new element
        X_test = X_split[iterator]
        z_test = z_split[iterator]
        # deleting current element
        X_train = np.delete(X_train, iterator, 0)
        z_train = np.delete(z_train, iterator, 0)
        # and adjusting arrays dimensions (e.g. X: [4, 500, 21] => [2000, 21])
        X_train = np.concatenate(X_train, axis=0)
        z_train = z_train.ravel()

        return X_train, X_test, z_train, z_test
    '''
    Linear Cross validation (manual algorithm)
    '''
    def doCrossVal(self, *args):
        # getting design matrix
        X = args[0]
        # getting z values and making them 1d
        z = np.ravel(args[1])
        kfold = args[2]
        # Splitting and shuffling data randomly
        #X_train, X_test, z_train, z_test = train_test_split(X, z, test_size=1. /
            ↪ kfold, shuffle=True)
        MSEtest_lintot = []
        MSEtrain_lintot = []
        z_tested = []
        z_trained = []
        z_t = []
        # bias
        bias = []
        # shuffling dataset randomly
        # 1. Shuffling datasets randomly:
        X, z = self.shuffleDataSimultaneously(X, z)
        # splitting data sets into the kfold and iterate over each of them
        for i in range(kfold):
            # Splitting and shuffling data randomly
            #X_train, X_test, z_train, z_test = train_test_split(X, z, test_size
                ↪ =1./kfold, shuffle=True)
            X_train, X_test, z_train, z_test = self.splitDataset(X, z, kfold, i)
            z_t.append(z_test)
            # Train The Pipeline
            invA = self.doSVD(X_train)
            beta_train = invA.dot(X_train.T).dot(z_train)
            # Testing the pipeline
            z_trained.append(X_train @ beta_train)
            z_tested.append(X_test @ beta_train)
            # Calculating MSE for each iteration
            MSEtest_lintot.append(self.getMSE(z_test, z_tested[i]))
            MSEtrain_lintot.append(self.getMSE(z_train, z_trained[i]))
        # linear MSE
        MSEtest_lin = np.mean(MSEtest_lintot)
```

```
        MSEtrain_lin = np.mean(MSEtrain_lintot)
        # bias-variance trade off
        z_tested_mean = np.mean(z_tested, axis=1, keepdims=True)
        for i in range(kfold):
            bias.append((z_t[i] - z_tested_mean)**2)
        bias_mean = np.mean( bias )
        variance_mean = np.mean( np.var(z_tested, axis=1, keepdims=True) )

        return MSEtest_lin, MSEtrain_lin, bias_mean, variance_mean

    '''
    Ridge Cross validation - manual algorithm
    '''
    def doCrossValRidge(self, *args):
        # getting design matrix
        X = args[0]
        # getting z values and making them 1d
        z = np.ravel(args[1])
        kfold = args[2]
        lambda_par = args[3]
        MSEtest_ridgetot = []
        MSEtrain_ridgetot = []
        z_tested = []
        z_trained = []
        # saving test data set to calculate bias-variance trade off
        z_t = []
        # bias
        bias = []
        # shuffling dataset randomly
        # 1. Shuffling datasets randomly:
        X, z = self.shuffleDataSimultaneously(X, z)
        for i in range(kfold):
            # Splitting and shuffling data randomly
            #X_train, X_test, z_train, z_test = train_test_split(X, z, test_size
                ↪ =1./kfold, shuffle=True)
            X_train, X_test, z_train, z_test = self.splitDataset(X, z, kfold, i)
            z_t.append(z_test)
            # constructing the identity matrix
            I = np.identity(len(X_train.T.dot(X_train)), dtype=float)
            # Train The Pipeline
            # calculating parameters
            invA = np.linalg.inv(X_train.T.dot(X_train) + lambda_par * I)
            beta_train = invA.dot(X_train.T).dot(z_train)
            # Testing the pipeline
            z_trained.append(X_train @ beta_train)
            z_tested.append(X_test @ beta_train)
            # Calculating MSE for each iteration
            MSEtest_ridgetot.append(self.getMSE(z_test, z_tested[i]))
            MSEtrain_ridgetot.append(self.getMSE(z_train, z_trained[i]))
        # Ridge MSE
        MSEtest_ridge = np.mean(MSEtest_ridgetot)
        MSEtrain_ridge = np.mean(MSEtrain_ridgetot)
        # bias-variance trade off
        z_tested_mean = np.mean(z_tested, axis=1, keepdims=True)
        for i in range(kfold):
            bias.append((z_t[i] - z_tested_mean)**2)
        bias_mean = np.mean( bias )
        variance_mean = np.mean( np.var(z_tested, axis=1, keepdims=True) )

        return MSEtest_ridge, MSEtrain_ridge, bias_mean, variance_mean
    '''
    Cross Validation using Scikit Learn functionalities (all at once)
```

```
    '''
    def doCrossValScikit(self, *args):
        # getting inputs
        X = args[0]
        z = args[1]
        kfold = args[2]
        poly_degree = args[3]
        lambda_par = args[4]
        # understanding the regression type to use
        reg_type = args[5]
        if reg_type == 'linear':
            model = LinearRegression(fit_intercept = False)
        elif reg_type == 'ridge':
            model = Ridge(alpha = lambda_par, fit_intercept = False)
        elif reg_type == 'lasso':
            model = Lasso(alpha = lambda_par, normalize=True)
        else:
            print("Houston, we've got a problem!")

        MSEtest = []
        MSEtrain = []
        # bias
        bias = []
        z_t = []
        z_tested = []
        # If the dataset does not cleanly divide by the number of folds,
        # there may be some remainder rows and they will not be used in the split.
        length = len(X) % kfold
        if length == 0:
            condition = True
        else:
            condition = False
        while condition is False:
            # removing the element <= they were shuffled randomly,
            # so it doesn't matter which one to remove
            X = np.delete(X, -1, axis = 0)
            z = np.delete(z, -1, axis = 0)
            # checking whether it is divided cleanly
            length = len(X) % kfold
            if length == 0:
                condition = True
        # making splits - shuffling it
        cv = KFold(n_splits = kfold, shuffle = True, random_state = 1)
        # enumerate splits - splitting the data set to train and test splits
        for train, test in cv.split(X):
            X_train, X_test = X[train], X[test]
            z_train, z_test = z[train], z[test]
            z_t.append(z_test)
            # making the prediction - comparing outputs for current and "future"
                ↪ datasets
            z_tilde = model.fit(X_train, z_train).predict(X_train).ravel() #
                ↪ z_trained
            z_pred = model.fit(X_train, z_train).predict(X_test).ravel() #
                ↪ z_tested
            z_tested.append(z_pred)

            MSEtest.append(mean_squared_error(z_test, z_pred))
            MSEtrain.append(mean_squared_error(z_train, z_tilde))

        # getting the mean values for errors (to plot them later)
        MSEtest_mean = np.mean(MSEtest)
        MSEtrain_mean = np.mean(MSEtrain)
```

```
        # bias-variance trade off
        z_tested_mean = np.mean(z_tested, axis=1, keepdims=True)
        for i in range(kfold):
            bias.append((z_t[i] - z_tested_mean)**2)
        bias_mean = np.mean( bias )
        variance_mean = np.mean( np.var(z_tested, axis=1, keepdims=True) )

        # returning MSE, bias and variance for  a given polynomial degree
        return MSEtest_mean, MSEtrain_mean, bias_mean, variance_mean

    '''
    MSE - the smaller the better (0 is the best?)
    '''

    def getMSE(self, z_data, z_model):
        n = np.size(z_model)
        return np.sum((z_data - z_model) ** 2) / n

    '''
    R^2 - values should be between 0 and 1 (with 1 being the best)
    '''

    def getR2(self, z_data, z_model):
        return 1 - np.sum((z_data - z_model) ** 2) / np.sum((z_data - np.mean(
            ↪ z_data)) ** 2)

    '''
    #============================#
    # Regression Methods
    #============================#
    '''
    '''
    Polynomial Regression - does linear regression analysis with our generated
    polynomial and returns the predicted values (our model) <= k-fold cross
    validation has been implemented
    '''

    def doLinearRegression(self, *args):
        # getting design matrix
        X = args[0]
        # getting z values and making them 1d
        z = np.ravel(args[1])
        # calculating variance of data

        # and then make the prediction
        invA = self.doSVD(X)
        beta = invA.dot(X.T).dot(z)
        ztilde = X @ beta
        # calculating beta confidence
        confidence = args[2]   # 1.96
        sigma = args[3]#np.var(z)  # args[3] #1
        SE = sigma * np.sqrt(np.diag(invA)) * confidence
        beta_min = beta - SE
        beta_max = beta + SE

        return ztilde, beta, beta_min, beta_max  # z_trained#ztilde#, beta, SE

    '''
    Ridge Regression
    '''

    def doRidgeRegression(self, *args):
```

```
        # getting design matrix
        X = args[0]
        # getting z values
        z = np.ravel(args[1])
        # hyper parameter
        lambda_par = args[2]
        # constructing the identity matrix
        XTX = X.T.dot(X)
        I = np.identity(len(XTX), dtype=float)
        # calculating parameters
        invA = np.linalg.inv(XTX + lambda_par * I)
        beta = invA.dot(X.T).dot(z)
        # and making predictions
        ztilde = X @ beta

        # calculating beta confidence
        confidence = args[3]  # 1.96
        # calculating variance
        sigma = args[4]#np.var(z)  # args[4] #1
        SE = sigma * np.sqrt(np.diag(invA)) * confidence
        beta_min = beta - SE
        beta_max = beta + SE

        return ztilde, beta, beta_min, beta_max  # , beta, SE

    '''
    LASSO Regression
    '''

    def doLASSORegression(self, *args):
        pass

    '''
    Methods to plot data
    '''
    def plotBeta(self, *args):
        x = args[0]
        y = args[1]
        y_min = args[2]
        y_max = args[3]
        output_dir = args[4]
        filename = args[5]
        # Turning interactive mode on
        #plt.ion()
        fig = plt.figure()#figsize = (10, 3))
        axe = fig.add_subplot(1, 1, 1)
        axe.plot(x, y, 'bo', label=r'$\beta$')
        axe.plot(x, y_min, 'r--', label=r'$\beta_{min}$')
        axe.plot(x, y_max, 'g--', label=r'$\beta_{max}$')
        axe.legend()
        # setting axes to log scale (to account for very high beta?)
        #axe.set_yscale('log')
        plt.grid(True)
        plt.xlabel('number␣of␣' + r'$\beta$')
        plt.ylabel(r'$\beta$')
        fig.savefig(output_dir + '/' + filename)
        # close the figure window
        plt.close(fig)
        # turning the interactive mode off
        #plt.ioff()

    def plotSurface(self, *args):
```

```
        # passing coordinates
        x = args[0]
        y = args[1]
        # takes an array of z values
        zarray = args[2]
        # output dir
        output_dir = args[3]
        # filename
        filename = args[4]
        # Turning interactive mode on
        #plt.ion()
        fig = plt.figure(figsize=(10, 3))
        axes = [fig.add_subplot(1, 3, i, projection='3d') for i in range(1, len(
            ↪ zarray) + 1)]
        surf = [axes[i].plot_surface(x, y, zarray[i], alpha = 0.5,
                                     cmap = 'brg_r', linewidth = 0, antialiased =
                                         ↪ False) for i in range(len(zarray))]
        # saving figure with corresponding filename
        fig.savefig(output_dir + '/' + filename)
        # close the figure window
        plt.close(fig)
        # turning the interactive mode off
        #plt.ioff()
```

Listing 2: "Entry point"

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 26 16:13:27 2019

@author: maksymb
"""

# standard imports
import os, sys
import numpy as np
# for polynomial manipulation
import sympy as sp
# from sympy import *
import itertools as it
# importing my library
import reglib as rl
# allowing multiprocessing (because we can? :))
import multiprocessing as mp
from joblib import Parallel, delayed

from mpl_toolkits.mplot3d import Axes3D
import matplotlib
import matplotlib.pyplot as plt
#matplotlib.use('Qt5Agg')
from matplotlib import cm
# to use latex symbols
# from matplotlib import rc

# to read tif files
from imageio import imread

# Machine Learning libraries
# to generate polynomial (for regression)
from sklearn.preprocessing import PolynomialFeatures
# regression libraries
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

from sklearn.metrics import mean_squared_error
# to split data for testing and training - KFold cross validation implementation
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

import time

# rc('text', usetex=True)
# rc('text.latex', preamble=r'\usepackage{amssymb}')

class MainPipeline(object):
    ''' class constructor '''
    def __init__(self, *args):
        #
            ↪ ============================================================================================
            ↪
        # symbolic variables
        self.x_symb = args[0]
        # array of values for each variable/feature
```

```
        self.x_vals = args[1]
        # grid values
        self.x = args[2]
        self.y = args[3]
        self.z = args[4]
        # 1.96 to calculate stuff with 95% confidence
        self.confidence = args[5]
        # noise variance - for confidence intervals estimation
        self.sigma = args[6]
        # k-value for Cross validation
        self.kfold = args[7]
        # hyper parameter
        self.lambda_par = args[8]
        # directory where to store plots
        self.output_dir = args[9]
        self.prefix = args[10]
        # degree of polynomial to fit
        self.poly_degree = args[11]
        #
            ↪ ##########################################################################################
            ↪

    '''
    Method to return calculated values (surface plots, MSEs, betas etc.), based on
        ↪  the user input choice.
    '''
    def doRegression(self, *args):
        # amount of processors to use
        nproc = args[0]
        # for plotting betas (this valu will appear in the file name <= doesn't
            ↪ affect calculations)
        npoints_name = args[1]
        curr_lambda = args[2]
        # library object instantiation
        lib = rl.RegressionLibrary(self.x_symb, self.x_vals)
        # raveling variables (making them 1d
        x_rav, y_rav, z_rav = np.ravel(self.x), np.ravel(self.y), np.ravel(self.z)
        # shape of z
        zshape = np.shape(self.z)
        #
            ↪ ##########################################################################################
            ↪
        ''' Linear Regression '''
        #
            ↪ ##########################################################################################
            ↪
        ''' MANUAL '''
        # getting design matrix
        X = lib.constructDesignMatrix(self.poly_degree)
        # getting predictions
        ztilde_lin, beta_lin, beta_min, beta_max = lib.doLinearRegression(X, z_rav
            ↪ , self.confidence, self.sigma)
        ztilde_lin = ztilde_lin.reshape(zshape)
        ''' Scikit Learn '''
        # generate polynomial
        poly_features = PolynomialFeatures(degree = self.poly_degree)
        # [[x[0], y[0]],[x[1],y[1]], [x[2],y[2]], ...]
        # works mush better than the transpose and reshape
        # (so far reshape, without "F" order was giving crap)
        X_scikit = np.swapaxes(np.array([x_rav, y_rav]), 0, 1)
        X_poly = poly_features.fit_transform(X_scikit)
        lin_reg = LinearRegression().fit(X_poly, z_rav)
```

```
        ztilde_sk = lin_reg.predict(X_poly).reshape(zshape)
        zarray_lin = [self.z, ztilde_lin, ztilde_sk]
        # Errors
        print('\n')
        print("Linear MSE (no CV) - " + str(lib.getMSE(zarray_lin[0], zarray_lin
            ↪ [1])) + "; sklearn - " + str(mean_squared_error(zarray_lin[0],
            ↪ zarray_lin[2])))
        print("Linear R^2 (no CV) - " + str(lib.getR2(zarray_lin[0], zarray_lin
            ↪ [1])) + "; sklearn - " + str(lin_reg.score(X_poly, z_rav)))
        print('\n')
        ''' Plotting Surfaces '''
        filename = self.prefix + '_linear_p' + str(self.poly_degree).zfill(2) + '
            ↪ _n' + npoints_name +'.png'
        # calling method from library to do this for us
        lib.plotSurface(self.x, self.y, zarray_lin, self.output_dir, filename)
        # betas
        filename = self.prefix + '_linear_beta_p' + str(self.poly_degree).zfill(2)
            ↪  + '_n' + npoints_name + '.png'
        t = []
        [t.append(i) for i in range(1, len(beta_lin) + 1)]
        print("betas are %s" %beta_lin)
        print("beta_min are %s" %beta_min)
        print("beta_max are %s" %beta_max)
        lib.plotBeta(t, beta_lin, beta_min, beta_max, output_dir, filename)
        # write betas to a txt file
        filename = self.prefix + '_linear_beta_p' + str(self.poly_degree).zfill(2)
            ↪  + '_n' + npoints_name + '.txt'
        with open(output_dir + '/' + filename, 'w') as file_handler:
            file_handler.write("beta_lin: {}\n".format(beta_lin))
            print('\n')
            file_handler.write("beta_min: {}\n".format(beta_min))
            print('\n')
            file_handler.write("beta_max: {}\n".format(beta_max))
        ''' kFold Cross Validation '''
        ''' MANUAL '''
        print('kFold CV for Linear Regression - Manual \n')
        self.kFoldMSEtest_lin  = lib.doCrossVal(X, self.z, self.kfold)[0]
        self.kFoldMSEtrain_lin = lib.doCrossVal(X, self.z, self.kfold)[1]
        self.kFoldBias_lin     = lib.doCrossVal(X, self.z, self.kfold)[2]
        self.kFoldVariance_lin = lib.doCrossVal(X, self.z, self.kfold)[3]
        ''' Scikit Learn '''
        reg_type = 'linear'
        self.kFoldMSEtestSK_lin  = lib.doCrossValScikit(X_poly, z_rav, self.kfold,
            ↪  self.poly_degree, self.lambda_par, reg_type)[0]
        self.kFoldMSEtrainSK_lin = lib.doCrossValScikit(X_poly, z_rav, self.kfold,
            ↪  self.poly_degree, self.lambda_par, reg_type)[1]
        self.kFoldBiasSK_lin     = lib.doCrossValScikit(X_poly, z_rav, self.kfold,
            ↪  self.poly_degree, self.lambda_par, reg_type)[2]
        self.kFoldVarianceSK_lin = lib.doCrossValScikit(X_poly, z_rav, self.kfold,
            ↪  self.poly_degree, self.lambda_par, reg_type)[3]

        #
            ↪ ================================================================================
            ↪
        ''' Ridge Regression '''
        #
            ↪ ================================================================================
            ↪
        ''' MANUAL '''
        ztilde_ridge, beta_ridge, beta_min, beta_max = lib.doRidgeRegression(X,
            ↪ z_rav, self.lambda_par, self.confidence, self.sigma)
        ztilde_ridge = ztilde_ridge.reshape(zshape)
```

```
        ''' Scikit Learn '''
        ridge_reg = Ridge(alpha = self.lambda_par, fit_intercept=True).fit(X_poly,
            ↪  z_rav)
        ztilde_sk = ridge_reg.predict(X_poly).reshape(zshape)
        zarray_ridge = [self.z, ztilde_ridge, ztilde_sk]
        print('\n')
        print("Ridge MSE (no CV) - " + str(lib.getMSE(zarray_ridge[0],
            ↪ zarray_ridge[1])) + "; sklearn - " + str(mean_squared_error(
            ↪ zarray_ridge[0], zarray_ridge[2])))
        print("Ridge R^2 (no CV) - " + str(lib.getR2(zarray_ridge[0], zarray_ridge
            ↪ [1])) + "; sklearn - " + str(ridge_reg.score(X_poly, z_rav)))
        print('\n')
        ''' Plotting Surfaces '''
        filename = self.prefix + '_ridge_p' + str(self.poly_degree).zfill(2) + '_n
            ↪ ' + npoints_name + '.png'
        lib.plotSurface(self.x, self.y, zarray_ridge, self.output_dir, filename)
        # betas
        filename = self.prefix + '_ridge_beta_p' + str(self.poly_degree).zfill(2)
            ↪ + '_n' + npoints_name + '.png'
        t = []
        [t.append(i) for i in range(1, len(beta_lin) + 1)]
        print("betas are %s" %beta_ridge)
        print("beta_min are %s" %beta_min)
        print("beta_max are %s" %beta_max)
        lib.plotBeta(t, beta_ridge, beta_min, beta_max, output_dir, filename)
        # write betas to a txt file
        filename = self.prefix + '_ridge_beta_p' + str(self.poly_degree).zfill(2)
            ↪ + '_n' + npoints_name + '.txt'
        with open(output_dir + '/' + filename, 'w') as file_handler:
            file_handler.write("beta_ridge: {}\n".format(beta_ridge))
            print('\n')
            file_handler.write("beta_min: {}\n".format(beta_min))
            print('\n')
            file_handler.write("beta_max: {}\n".format(beta_max))
        # Calculating k-Fold Cross Validation
        print('kFold CV for Ridge Regression - Manual \n')
        #curr_lambda = 0.1
        # parallel processing
        manager = mp.Manager()
        # making a dictionary of values which will save "mse mean value"
        self.kFoldMSEtest_ridge  = manager.dict()
        self.kFoldMSEtrain_ridge = manager.dict()
        self.kFoldBias_ridge     = manager.dict()
        self.kFoldVariance_ridge = manager.dict()
        lambdas = []
        # decreasing the value of lambda till certain minimal value
        # and updating the list
        while curr_lambda >= self.lambda_par:
            lambdas.append(curr_lambda)
            curr_lambda = curr_lambda/10
        # creating a method to work with dictionaries (in parallel)
        def doMultiproc1(kFoldMSEtest_ridge, kFoldMSEtrain_ridge, kFoldBias_ridge,
            ↪  kFoldVariance_ridge, lib, X, z, kfold, curr_lambda):
            print('Ridge Manual \n')
            print("Starting to calculate for $\lambda$=%s" %curr_lambda + '\n')
            kFoldMSEtest_ridge[curr_lambda]  = lib.doCrossValRidge(X, z, kfold,
                ↪ curr_lambda)[0]
            kFoldMSEtrain_ridge[curr_lambda] = lib.doCrossValRidge(X, z, kfold,
                ↪ curr_lambda)[1]
            kFoldBias_ridge[curr_lambda]     = lib.doCrossValRidge(X, z, kfold,
                ↪ curr_lambda)[2]
            kFoldVariance_ridge[curr_lambda] = lib.doCrossValRidge(X, z, kfold,
```

```
            ↪ curr_lambda)[3]
        # and exit!
        return
    # using library for parallel processing to loop through all lambda
        ↪ parameters
    Parallel(n_jobs = nproc, verbose = 2)(delayed(doMultiproc1)(self.
        ↪ kFoldMSEtest_ridge, self.kFoldMSEtrain_ridge,
                                                                self.
                                                                    ↪ kFoldBias_ridge
                                                                    ↪ , self.
                                                                    ↪ kFoldVariance_ridge
                                                                    ↪ ,
                                                    lib, X, self.z, self.kfold,
                                                        ↪  curr_lambda) for
                                                        ↪ curr_lambda in
                                                        ↪ lambdas)
    ''' Scikit Learn '''
    print('kFold␣CV␣for␣Ridge␣Regression␣-␣Scikit␣Learn␣\n')
    # chosing the type of regression
    reg_type = 'ridge'
    # making a dictionary of values which will save "mse mean value"
    # parallel processing
    self.kFoldMSEtestSK_ridge  = manager.dict()
    self.kFoldMSEtrainSK_ridge = manager.dict()
    self.kFoldBiasSK_ridge     = manager.dict()
    self.kFoldVarianceSK_ridge = manager.dict()
    # creating a method to work with dictionaries (in parallel)
    def doMultiproc2(kFoldMSEtestSK_ridge, kFoldMSEtrainSK_ridge,
        ↪ kFoldBiasSK_ridge, kFoldVarianceSK_ridge,
                     lib, X_poly, z_rav, kfold, poly_degree, curr_lambda,
                         ↪ reg_type):
        print('Ridge␣Scikit␣Learn␣\n')
        print("Starting␣to␣calculate␣for␣$\lambda$=%s" %curr_lambda + '\n')
        kFoldMSEtestSK_ridge[curr_lambda]  = lib.doCrossValScikit(X_poly,
            ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[0]
        kFoldMSEtrainSK_ridge[curr_lambda] = lib.doCrossValScikit(X_poly,
            ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[1]
        kFoldBiasSK_ridge[curr_lambda]     = lib.doCrossValScikit(X_poly,
            ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[2]
        kFoldVarianceSK_ridge[curr_lambda] = lib.doCrossValScikit(X_poly,
            ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[3]
        # and exit!
        return
    Parallel(n_jobs = nproc, verbose = 2)(delayed(doMultiproc2)(self.
        ↪ kFoldMSEtestSK_ridge, self.kFoldMSEtrainSK_ridge,
                                                                self.
                                                                    ↪ kFoldBiasSK_ridge
                                                                    ↪ , self.
                                                                    ↪ kFoldVarianceSK_ridge
                                                                    ↪ , lib,
                                                                    ↪ X_poly,
                                                    z_rav, self.kfold, self.
                                                        ↪ poly_degree,
                                                        ↪ curr_lambda,
                                                        ↪ reg_type)
                                                    for curr_lambda in lambdas)

    #
        ↪ ========================================================================
        ↪
    ''' LASSO Regression '''
    #
```

```
            ↪ ================================================================================
            ↪
        ''' Scikit Learn '''
        lasso_reg = Lasso(alpha=self.lambda_par).fit(X_poly, z_rav)
        ztilde_sk = lasso_reg.predict(X_poly).reshape(zshape)
        zarray_lasso = [self.z, ztilde_sk]
        print('\n')
        print("SL Lasso MSE (no CV) - " + str(mean_squared_error(zarray_lasso[0],
            ↪ zarray_lasso[1])))
        print("SL Lasso R^2 (no CV) - " + str(lasso_reg.score(X_poly, z_rav)))
        print('\n')
        ''' Plotting Surfaces '''
        filename = self.prefix + '_lasso_p' + str(self.poly_degree).zfill(2) + '_n
            ↪ '+ npoints + '.png'
        lib.plotSurface(self.x, self.y, zarray_lasso, self.output_dir, filename)
        # k-fold - studying the dependence on lambda
        print('kFold CV for LASSO Regression - Scikit Learn \n')
        reg_type = 'lasso'
        # parallel processing
        # making a dictionary of values which will save "mse mean value"
        self.kFoldMSEtestSK_lasso  = manager.dict()
        self.kFoldMSEtrainSK_lasso = manager.dict()
        self.kFoldBiasSK_lasso     = manager.dict()
        self.kFoldVarianceSK_lasso = manager.dict()
        # creating a method to work with dictionaries
        def doMultiproc3(kFoldMSEtestSK_lasso, kFoldMSEtrainSK_lasso,
            ↪ kFoldBiasSK_lasso, kFoldVarianceSK_lasso,
                         lib, X_poly, z_rav, kfold, poly_degree, curr_lambda,
                             ↪ reg_type):
            print('Lasso Scikit Learn \n')
            print("Starting to calculate for $\lambda$=%s" %curr_lambda + '\n')
            kFoldMSEtestSK_lasso[curr_lambda]  = lib.doCrossValScikit(X_poly,
                ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[0]
            kFoldMSEtrainSK_lasso[curr_lambda] = lib.doCrossValScikit(X_poly,
                ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[1]
            kFoldBiasSK_lasso[curr_lambda]     = lib.doCrossValScikit(X_poly,
                ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[2]
            kFoldVarianceSK_lasso[curr_lambda] = lib.doCrossValScikit(X_poly,
                ↪ z_rav, kfold, poly_degree, curr_lambda, reg_type)[3]
            # and exit!
            return
        Parallel(n_jobs = nproc, verbose = 2)(delayed(doMultiproc3)(self.
            ↪ kFoldMSEtestSK_lasso, self.kFoldMSEtrainSK_lasso,
                                                                    self.
                                                                        ↪ kFoldBiasSK_lasso
                                                                        ↪ , self.
                                                                        ↪ kFoldVarianceSK_lasso
                                                                        ↪ , lib,
                                                                        ↪ X_poly,
                                                    z_rav, self.kfold, self.
                                                        ↪ poly_degree,
                                                        ↪ curr_lambda, reg_type)
                                                    for curr_lambda in lambdas)


if __name__ == '__main__':
    # Start time of the program
    start_time = time.time()
    # Creating output directory to save plots (in  png format)
    output_dir = 'Output'
    # checking whether the output directory already exists (if not, create one)
    if not os.path.exists(output_dir):
```

```
    os.makedirs(output_dir)
'''
Yes/no question - based on the user input will return. Basically, the user is
    ↪ asked whether
he/she wants to run script on real data. Is the answer is no, then the data
    ↪ set will be simulated
Is the based on uniform distribution/linear grid.

The answer returns True for 'yes' and False for 'no'
'''
sys.stdout.write('Do you want to use real data? (' + '\033[91m' + '[y]'+'
    ↪ \033[0m' +'/[n]) ')
value = None
while value == None:
    # User input - making it lower case
    choice = input().lower()
    # possible answers
    yes = {'yes': True, 'ye': True, 'yeah': True, 'y': True, '': True}
    no = {'no': False, 'n': False}
    if choice in yes:
        value = True
    elif choice in no:
        value = False
    else:
        sys.stdout.write("Please respond with 'yes' or 'no' (or keep the field
            ↪ empty): ")

# Working with Real data
if value == True:
    print('''
    #=========================#
    # Working with Real Data #
    #=========================#
    ''')
    sys.stdout.write("Please, provide path to data file (default = Data/
        ↪ SRTM_data_Norway_1.tif): ")
    # Load the terrain
    terrain = input()
    if terrain == '':
        terrain = imread('Data/SRTM_data_Norway_1.tif')
    else:
        terrain = imread(terrain)
    '''
    Decide on how much of data you want to use:
    It doesn't make sense to use the entire data set,
    because there are a lot of points => CV will give
    identical results for test and train data. => makes
    sense to use only like 10% of the data.
    '''
    sys.stdout.write("Please, specify the percentage of the data to use (
        ↪ default = 0.1): ")
    cut_dat = input()
    terrain.sort()
    if cut_dat == '':
        dataset = terrain[int(len(terrain) * 0.9):]#terrain[int(len(terrain) *
            ↪  .05) : int(len(terrain) * .95)]
    else:
        dataset = terrain[int(len(terrain) * (1.-float(cut_dat))):]
    print(np.shape(dataset))
    #terrain.sort()
    #dataset = terrain[int(len(terrain) * .05) : int(len(terrain) * .95)]
    #print(np.shape(dataset))
```

```
        # number of independent variables (features)
        n_vars = 2
        # max polynomial degree
        sys.stdout.write("Please, choose the max polynomial degree (default = 5): "
            ↪ ")
        max_poly_degree = input()
        if max_poly_degree == '':
            max_poly_degree = 5
        else:
            max_poly_degree = int(max_poly_degree)
        # number of processors to use <= better use this version
        sys.stdout.write("Please, choose the amount of processors to use (default "
            ↪ =" +str(mp.cpu_count()-1) + "): ")
        nproc = input()
        if nproc == '':
            nproc = mp.cpu_count()-1
        else:
            nproc = int(nproc)
        # just to save your value (e.g. png(s)) under correct prefix
        prefix = 'real'
        # the amount of folds to get from your data
        kfold = 5
        # to calculate confidence intervals
        confidence = 1.96
        sigma = 1 # because I am generating my noise from standard normal
            ↪ distribution
        # lasso very sensitive to this lambda parameter
        sys.stdout.write("Please, choose the min value of hyperparameter (lambda) "
            ↪ (default = 0.0001): ")
        lambda_par = input()
        if lambda_par == '':
            lambda_par = 0.0001
        else:
            lambda_par = float(lambda_par)
        # max lambda value (starting one)
        max_lambda = 0.1
        #lambda_par = 0.000001
        # just to save your value (e.g. png(s)) under correct prefix
        prefix = 'real'
        ''' Generating Data Set '''
        # generating an array of symbolic variables
        # based on the desired amount of variables
        x_symb = sp.symarray('x', n_vars, real = True)
        # making a copy of this array
        x_vals = x_symb.copy()
        x_vals[0] = np.linspace(0, len(dataset[0]), len(dataset[0]))
        x_vals[1] = np.linspace(0, len(dataset), len(dataset))
        # library object instantiation
        lib = rl.RegressionLibrary(x_symb, x_vals)
        # generating output data - first setting-up the proper grid
        x, y = np.meshgrid(x_vals[0], x_vals[1])
        z = dataset#terrain

    # Working with generated (Fake) data
    elif value == False:
        print('''
        #========================#
        # Working with Fake Data #
        #========================#
        ''')
        ''' Input Parameters'''
        sys.stdout.write("Please, choose an amount of points to simulate data (
```

```
        ↪ default␣=␣50):␣")
    # Data points to simulate grid (N_points x N_points)
    N_points = input()
    if N_points == '':
        N_points = 50
    else:
        N_points = int(N_points)
    # number of independent variables (features)
    n_vars = 2
    # max polynomial degree
    sys.stdout.write("Please,␣choose␣the␣max␣polynomial␣degree␣(default␣=␣5):␣
        ↪ ")
    max_poly_degree = input()
    if max_poly_degree == '':
        max_poly_degree = 5
    else:
        max_poly_degree = int(max_poly_degree)
    # number of processors to use <= better use this version
    sys.stdout.write("Please,␣choose␣the␣amount␣of␣processors␣to␣use␣(default␣
        ↪ =␣" +str(mp.cpu_count()-1) + "):␣")
    nproc = input()
    if nproc == '':
        nproc = mp.cpu_count()-1
    else:
        nproc = int(nproc)
    # the amount of folds to get from your data
    kfold = 5
    # to calculate confidence intervals
    confidence = 1.96
    sigma = 0.1
    # lasso very sensitive to this lambda parameter
    sys.stdout.write("Please,␣choose␣the␣mean␣value␣of␣hyperparameter␣(lambda)
        ↪ ␣(default␣=␣0.0001):␣")
    lambda_par = input()
    if lambda_par == '':
        lambda_par = 0.0001
    else:
        lambda_par = float(lambda_par)
    # max lambda value (starting one)
    max_lambda = 0.1
    # just to save your value (e.g. png(s)) under correct prefix
    prefix = 'fake'
    ''' Generating Data Set '''
    # generating an array of symbolic variables
    # based on the desired amount of variables
    x_symb = sp.symarray('x', n_vars, real = True)
    # making a copy of this array
    x_vals = x_symb.copy()
    # and fill it with values
    for i in range(n_vars):
        x_vals[i] = np.arange(0, 1, 1./N_points)#np.sort(np.random.uniform(0,
            ↪ 1, N_points))
    # library object instantiation
    lib = rl.RegressionLibrary(x_symb, x_vals)
    # setting up the grid
    x, y = np.meshgrid(x_vals[0], x_vals[1])
    # and getting output based on the Franke Function
    z = lib.FrankeFunction(x, y) + sigma * np.random.randn(N_points, N_points)

# To plot MSE from kFold Cross Validation
kFoldMSEtest_lin      = []#[None] * max_poly_degree
kFoldMSEtrain_lin     = []#[None] * max_poly_degree
```

```
    kFoldMSEtestSK_lin    = [] #[None] * max_poly_degree
    kFoldMSEtrainSK_lin   = [] #[None] * max_poly_degree

    kFoldMSEtest_ridge    = [] #[None] * max_poly_degree
    kFoldMSEtrain_ridge   = [] #[None] * max_poly_degree
    kFoldMSEtestSK_ridge  = [] #[None] * max_poly_degree
    kFoldMSEtrainSK_ridge = [] #[None] * max_poly_degree

    kFoldMSEtestSK_lasso  = [] #[None] * max_poly_degree
    kFoldMSEtrainSK_lasso = [] #[None] * max_poly_degree

    # for bias-variance trade off
    kFoldBias_lin         = []
    kFoldVariance_lin     = []
    kFoldBiasSK_lin       = []
    kFoldVarianceSK_lin   = []

    kFoldBias_ridge       = []
    kFoldVariance_ridge   = []
    kFoldBiasSK_ridge     = []
    kFoldVarianceSK_ridge = []

    kFoldBiasSK_lasso     = []
    kFoldVarianceSK_lasso = []

    # to better classify output plots, I am adding
    # the amount of points, the png was generated for
    # (if it is a real data => then we get 'real' in the name)
    if prefix == 'real':
        npoints = 'real'
    elif prefix == 'fake':
        npoints = str(N_points)

    # looping through all polynomial degrees
    for poly_degree in range(1, max_poly_degree+1):
        print('\n')
        print('Starting␣analysis␣for␣polynomial␣of␣degree:␣' + str(poly_degree))
        pipeline = MainPipeline(x_symb, x_vals, x, y, z, confidence, sigma, kfold,
        lambda_par, output_dir, prefix, poly_degree)
        pipeline.doRegression(nproc, npoints, max_lambda)

        # getting the list of dictionaries
        # linear regression kfold
        kFoldMSEtest_lin.append(pipeline.kFoldMSEtest_lin)
        kFoldMSEtrain_lin.append(pipeline.kFoldMSEtrain_lin)
        kFoldMSEtestSK_lin.append(pipeline.kFoldMSEtestSK_lin)
        kFoldMSEtrainSK_lin.append(pipeline.kFoldMSEtrainSK_lin)
        # ridge regression kfold
        kFoldMSEtest_ridge.append(dict(pipeline.kFoldMSEtest_ridge))
        kFoldMSEtrain_ridge.append(dict(pipeline.kFoldMSEtrain_ridge))
        kFoldMSEtestSK_ridge.append(dict(pipeline.kFoldMSEtestSK_ridge))
        kFoldMSEtrainSK_ridge.append(dict(pipeline.kFoldMSEtrainSK_ridge))
        # lasso regression kfold
        kFoldMSEtestSK_lasso.append(dict(pipeline.kFoldMSEtestSK_lasso))
        kFoldMSEtrainSK_lasso.append(dict(pipeline.kFoldMSEtrainSK_lasso))

        '''
        getting values for bias and variance. In case of Ridge and Lasso
        regression these values are represented by dictionaries (for each
        values of lambda)
        '''
        kFoldBias_lin.append(pipeline.kFoldBias_lin)
```

```
        kFoldVariance_lin.append(pipeline.kFoldVariance_lin)
        kFoldBiasSK_lin.append(pipeline.kFoldBiasSK_lin)
        kFoldVarianceSK_lin.append(pipeline.kFoldVarianceSK_lin)

        kFoldBias_ridge.append(dict(pipeline.kFoldBias_ridge))
        kFoldVariance_ridge.append(dict(pipeline.kFoldVariance_ridge))
        kFoldBiasSK_ridge.append(dict(pipeline.kFoldBiasSK_ridge))
        kFoldVarianceSK_ridge.append(dict(pipeline.kFoldVarianceSK_ridge))

        kFoldBiasSK_lasso.append(dict(pipeline.kFoldBiasSK_lasso))
        kFoldVarianceSK_lasso.append(dict(pipeline.kFoldVarianceSK_lasso))

    '''
    Makins plots of MSEs
    '''
    # Colors - randomly generating colors
    np.random.seed(1)
    test_colors = [np.random.rand(3,) for i in range(max_poly_degree)] # <= will
        ↪ generate random colors
    train_colors = [np.random.rand(3,) for i in range(max_poly_degree)]
    # Turning interactive mode on
    #plt.ion()
    ''' MSE as a function of model complexity '''
    # plotting MSE from test data
    # Linear Regression
    filename = prefix + '_linear_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
        ↪ npoints + '.png'
    fig = plt.figure()
    ax1 = fig.add_subplot(2, 1, 1)
    ax2 = fig.add_subplot(2, 1, 2)
    t = []
    [t.append(i) for i in range(1, max_poly_degree + 1)]
    # manual
    ax1.plot(t, kFoldMSEtest_lin, color = test_colors[0], marker='o', label='test'
        ↪ )
    ax1.plot(t, kFoldMSEtrain_lin, color = train_colors[0], linestyle='dashed',
        ↪ label='train')
    # scikit learn
    ax2.plot(t, kFoldMSEtestSK_lin, color = test_colors[0], marker='o', label='
        ↪ test')
    ax2.plot(t, kFoldMSEtrainSK_lin, color = train_colors[0], linestyle='dashed',
        ↪ label='train')
    if prefix == 'real':
        ax1.set_yscale('log')
        ax2.set_yscale('log')
    # Shrink current axis by 20% - making legends appear to the right of the plot
    box = ax1.get_position()
    ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    box = ax2.get_position()
    ax2.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put a legend to the right of the current axis
    ax1.legend(loc='center␣left', bbox_to_anchor=(1, 0.5))
    ax1.grid(True)
    ax2.grid(True)
    ax1.set_title('MSE␣as␣a␣function␣of␣model␣complexity;␣Linear␣Regression')
    plt.xlabel('model␣complexity␣(polynomial␣degree)')
    ax1.set_ylabel('MSE')
    ax2.set_ylabel('MSE')
    fig.savefig(output_dir + '/' + filename)
    plt.close(fig)
    # write mses to a txt file
    filename = prefix + '_linear_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
```

```
    ↪ npoints + '.txt'
with open(output_dir + '/' + filename, 'w') as file_handler:
    file_handler.write("MSE test (Me): {}\n".format(kFoldMSEtest_lin))
    print('\n')
    file_handler.write("MSE train (Me): {}\n".format(kFoldMSEtrain_lin))
    print('\n')
    file_handler.write("MSE test (SK): {}\n".format(kFoldMSEtestSK_lin))
    print('\n')
    file_handler.write("MSE train (SK): {}\n".format(kFoldMSEtrainSK_lin))

# Ridge Regression
filename = prefix + '_ridge_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
    ↪ npoints +'.png'
fig = plt.figure()
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)
t = []
[t.append(i) for i in range(1, max_poly_degree + 1)]
keylist = []
# creating a keylist to be able to convert
# a list of dictionaries to a list of lists
curr_lambda = max_lambda
while curr_lambda >= lambda_par:
    # the list of all lambdas
    keylist.append(curr_lambda)
    curr_lambda = curr_lambda/10
# Manual
# converting a list of dictionaries to a list of lists
# (index is the lambda value, i.e. we have a list: [[],[],[],...[]]
# where first sublist corresponds to a maximum lambda value - 1 -
# and the last sublist corresponds to the smallest lambda value - 0.001 (if
    ↪ default))
test_list = [[row[key] for row in kFoldMSEtest_ridge] for key in keylist]
train_list = [[row[key] for row in kFoldMSEtrain_ridge] for key in keylist]
# plotting different mse values for different lambda
for i in range(len(test_list)):
    ax1.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
        ↪ lambda$='+str(keylist[i]) + ', test')
    ax1.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
        ↪ label='$\lambda$='+str(keylist[i]) + ', train')
# Scikit learn
test_list = [[row[key] for row in kFoldMSEtestSK_ridge] for key in keylist]
train_list = [[row[key] for row in kFoldMSEtrainSK_ridge] for key in keylist]
# plotting different mse values for different lambda
for i in range(len(test_list)):
    ax2.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
        ↪ lambda$='+str(keylist[i]) + ', test')
    ax2.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
        ↪ label='$\lambda$='+str(keylist[i]) + ', train')

if prefix == 'real':
    ax1.set_yscale('log')
    ax2.set_yscale('log')
# Shrink current axis by 20%
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
box = ax2.get_position()
ax2.set_position([box.x0, box.y0, box.width * 0.8, box.height])
# Put a legend to the right of the current axis
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax1.grid(True)
ax2.grid(True)
```

```
ax1.set_title('MSE as a function of model complexity; Ridge Regression')
plt.xlabel('model complexity (polynomial degree)')
ax1.set_ylabel('MSE')
ax2.set_ylabel('MSE')
fig.savefig(output_dir + '/' + filename)
plt.close(fig)
# write mses to a txt file
filename = prefix + '_ridge_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
    ↪ npoints +'.txt'
with open(output_dir + '/' + filename, 'w') as file_handler:
    file_handler.write("MSE test (Me): {}\n".format(kFoldMSEtest_ridge))
    print('\n')
    file_handler.write("MSE train (Me): {}\n".format(kFoldMSEtrain_ridge))
    print('\n')
    file_handler.write("MSE test (SK): {}\n".format(kFoldMSEtestSK_ridge))
    print('\n')
    file_handler.write("MSE train (SK): {}\n".format(kFoldMSEtrainSK_ridge))

# LASSO regression
filename = prefix + '_lasso_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
    ↪ npoints +'.png'
fig = plt.figure()
ax1 = fig.add_subplot(1, 1, 1)
t = []
[t.append(i) for i in range(1, max_poly_degree + 1)]
# scikit learn
keylist = []
# creating a keylist to be able to convert
# a list of dictionaries to a list of lists
curr_lambda = max_lambda
while curr_lambda >= lambda_par:
    keylist.append(curr_lambda)
    curr_lambda = curr_lambda/10
# converting a list of dictionaries to a list of lists
# (index is the lambda value, i.e. we have a list: [[],[],[],...[]]
# where first sublist corresponds to a maximum lambda value - 1 -
# and the last sublist corresponds to the smallest lambda value - 0.001 (if
    ↪ default))
test_list = [[row[key] for row in kFoldMSEtestSK_lasso] for key in keylist]
train_list = [[row[key] for row in kFoldMSEtrainSK_lasso] for key in keylist]
# plotting different mse values for different lambda
for i in range(len(test_list)):
    ax1.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
        ↪ lambda$='+str(keylist[i]) + ', test')
    ax1.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
        ↪ label='$\lambda$='+str(keylist[i]) + ', train')
if prefix == 'real':
    ax1.set_yscale('log')
# Shrink current axis by 20%
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
# Put a legend to the right of the current axis
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax1.grid(True)
ax1.set_title('MSE as a function of model complexity; Lasso Regression')
plt.xlabel('model complexity (polynomial degree)')
ax1.set_ylabel('MSE')
fig.savefig(output_dir + '/' + filename)
plt.close(fig)
# write mses to a txt file
filename = prefix + '_lasso_mse_p' + str(max_poly_degree).zfill(2) + '_n'+
    ↪ npoints +'.txt'
```

```
with open(output_dir + '/' + filename, 'w') as file_handler:
    file_handler.write("MSE test (Me): {}\n".format(kFoldMSEtestSK_lasso))
    print('\n')
    file_handler.write("MSE train (Me): {}\n".format(kFoldMSEtrainSK_lasso))

'''
Bias/Variance as a function of model complexity
'''
# Bias-variance trade off
# Colors - randomly generating colors
test_colors = [np.random.rand(3,) for i in range(max_poly_degree)] # <= will
    ↪ generate random colors
train_colors = [np.random.rand(3,) for i in range(max_poly_degree)]
# Turning interactive mode on
#plt.ion()
''' Linear Regression '''
filename = prefix + '_linear_bv_p' + str(max_poly_degree).zfill(2) + '_n'+
    ↪ npoints + '.png'
fig = plt.figure()
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)
t = []
[t.append(i) for i in range(1, max_poly_degree + 1)]
# manual
ax1.plot(t, kFoldBias_lin, color = test_colors[0], marker='o', label='bias')
ax1.plot(t, kFoldVariance_lin, color = train_colors[0], linestyle='dashed',
    ↪ label='variance')
# scikit learn
ax2.plot(t, kFoldBiasSK_lin, color = test_colors[0], marker='o', label='bias')
ax2.plot(t, kFoldVarianceSK_lin, color = train_colors[0], linestyle='dashed',
    ↪ label='variance')
if prefix == 'real':
    ax1.set_yscale('log')
    ax2.set_yscale('log')
# Shrink current axis by 20% - making legends appear to the right of the plot
box = ax1.get_position()
ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
box = ax2.get_position()
ax2.set_position([box.x0, box.y0, box.width * 0.8, box.height])
# Put a legend to the right of the current axis
ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax1.grid(True)
ax2.grid(True)
ax1.set_title('Bias/Variance as a function of model complexity; Linear
    ↪ Regression')
plt.xlabel('model complexity (polynomial degree)')
ax1.set_ylabel('Bias/Variance')
ax2.set_ylabel('Bias/Variance')
fig.savefig(output_dir + '/' + filename)
plt.close(fig)
# write bias/variance to a txt file
filename = prefix + '_linear_bv_p' + str(max_poly_degree).zfill(2) + '_n' +
    ↪ npoints + '.txt'
with open(output_dir + '/' + filename, 'w') as file_handler:
    file_handler.write("Bias (Me): {}\n".format(kFoldBias_lin))
    print('\n')
    file_handler.write("Variance (Me): {}\n".format(kFoldVariance_lin))
    print('\n')
    file_handler.write("Bias (Me): {}\n".format(kFoldBiasSK_lin))
    print('\n')
    file_handler.write("Variance (Me): {}\n".format(kFoldVarianceSK_lin))
```

```
    ''' Ridge Regression '''
    filename = prefix + '_ridge_bv_p' + str(max_poly_degree).zfill(2) + '_n'+
        ↪ npoints +'.png'
    fig = plt.figure()
    ax1 = fig.add_subplot(2, 1, 1)
    ax2 = fig.add_subplot(2, 1, 2)
    t = []
    [t.append(i) for i in range(1, max_poly_degree + 1)]
    keylist = []
    # creating a keylist to be able to convert
    # a list of dictionaries to a list of lists
    curr_lambda = max_lambda
    while curr_lambda >= lambda_par:
        # the list of all lambdas
        keylist.append(curr_lambda)
        curr_lambda = curr_lambda/10
    # Manual
    # converting a list of dictionaries to a list of lists
    # (index is the lambda value, i.e. we have a list: [[],[],[],...[]]
    # where first sublist corresponds to a maximum lambda value - 1 -
    # and the last sublist corresponds to the smallest lambda value - 0.001 (if
        ↪ default))
    test_list = [[row[key] for row in kFoldBias_ridge] for key in keylist]
    train_list = [[row[key] for row in kFoldVariance_ridge] for key in keylist]
    # plotting different mse values for different lambda
    for i in range(len(test_list)):
        ax1.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
            ↪ lambda$='+str(keylist[i]) + ', bias')
        ax1.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
            ↪ label='$\lambda$='+str(keylist[i]) + ', variance')
    # Scikit learn
    test_list = [[row[key] for row in kFoldBiasSK_ridge] for key in keylist]
    train_list = [[row[key] for row in kFoldVarianceSK_ridge] for key in keylist]
    # plotting different mse values for different lambda
    for i in range(len(test_list)):
        ax2.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
            ↪ lambda$='+str(keylist[i]) + ', bias')
        ax2.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
            ↪ label='$\lambda$='+str(keylist[i]) + ', variance')

    if prefix == 'real':
        ax1.set_yscale('log')
        ax2.set_yscale('log')
    # Shrink current axis by 20%
    box = ax1.get_position()
    ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    box = ax2.get_position()
    ax2.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put a legend to the right of the current axis
    ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    ax1.grid(True)
    ax2.grid(True)
    ax1.set_title('Bias/Variance as a function of model complexity; Ridge
        ↪ Regression')
    plt.xlabel('model complexity (polynomial degree)')
    ax1.set_ylabel('Bias/Variance')
    ax2.set_ylabel('Bias/Variance')
    fig.savefig(output_dir + '/' + filename)
    plt.close(fig)
    # write bias/variance to a txt file
    filename = prefix + '_ridge_bv_p' + str(max_poly_degree).zfill(2) + '_n'+
        ↪ npoints +'.txt'
```

```
    with open(output_dir + '/' + filename, 'w') as file_handler:
        file_handler.write("Bias (Me): {}\n".format(kFoldBias_ridge))
        print('\n')
        file_handler.write("Variance (Me): {}\n".format(kFoldVariance_ridge))
        print('\n')
        file_handler.write("Bias (Me): {}\n".format(kFoldBiasSK_ridge))
        print('\n')
        file_handler.write("Variance (Me): {}\n".format(kFoldVarianceSK_ridge))

    '''Lasso'''
    filename = prefix + '_lasso_bv_p' + str(max_poly_degree).zfill(2) + '_n'+
        ↪ npoints +'.png'
    fig = plt.figure()
    ax1 = fig.add_subplot(1, 1, 1)
    t = []
    [t.append(i) for i in range(1, max_poly_degree + 1)]
    # scikit learn
    keylist = []
    # creating a keylist to be able to convert
    # a list of dictionaries to a list of lists
    curr_lambda = max_lambda
    while curr_lambda >= lambda_par:
        keylist.append(curr_lambda)
        curr_lambda = curr_lambda/10
    # converting a list of dictionaries to a list of lists
    # (index is the lambda value, i.e. we have a list: [[],[],[],...[]]
    # where first sublist corresponds to a maximum lambda value - 1 -
    # and the last sublist corresponds to the smallest lambda value - 0.001 (if
        ↪ default))
    test_list = [[row[key] for row in kFoldBiasSK_lasso] for key in keylist]
    train_list = [[row[key] for row in kFoldVarianceSK_lasso] for key in keylist]
    # plotting different mse values for different lambda
    for i in range(len(test_list)):
        ax1.plot(t, test_list[i], color = test_colors[i], marker='o', label='$\
            ↪ lambda$='+str(keylist[i]) + ', bias')
        ax1.plot(t, train_list[i], color = train_colors[i], linestyle='dashed',
            ↪ label='$\lambda$='+str(keylist[i]) + ', variance')

    if prefix == 'real':
        ax1.set_yscale('log')
    # Shrink current axis by 20%
    box = ax1.get_position()
    ax1.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put a legend to the right of the current axis
    ax1.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    ax1.grid(True)
    ax1.set_title('Bias/Variance as a function of model complexity; Lasso
        ↪ Regression')
    plt.xlabel('model complexity (polynomial degree)')
    ax1.set_ylabel('Bias/Variance')
    fig.savefig(output_dir + '/' + filename)
    plt.close(fig)
    # write bias/variance to a txt file
    filename = prefix + '_lasso_bv_p' + str(max_poly_degree).zfill(2) + '_n'+
        ↪ npoints +'.txt'
    with open(output_dir + '/' + filename, 'w') as file_handler:
        file_handler.write("Bias (SK): {}\n".format(kFoldBiasSK_lasso))
        print('\n')
        file_handler.write("Variance (SK): {}\n".format(kFoldVarianceSK_lasso))


    # End time of the program
```

```
    end_time = time.time()
    print("--␣Program␣finished␣at␣%s␣sec␣--" % (end_time - start_time))
```