Modelling 2D terrain data by Linear regression: An introduction to simple machine learning

Jonny Aarstad Igeh — Fuad Dadvar
Department of Physics, University of Oslo, Norway
Github address: Link here

November 18, 2022

Abstract

In this report we've studied various methods for linear regression of a 2D surface, namely Ordinary least squares (OLS) , Ridge Regression and LASSO. We've built these model using the Franke Function to produce terrain data with added noise, and analyze each method by its error estimates using MSE, Variance and Bias. We ran these tests using $N = 50$ datapoints in the range $[0, 1]$, with the seed = 2001 (for the random module in python) and applied train test splitting of our dataset - and we also scaled our data using *mean normalization.* Furthermore, we also studied how two different resampling methods, Bootstrap and Cross validation, affected our error estimates. Out of the two, Cross validation for $k \geq 10$ produced similar, or better results than bootstrap. We found that, out of the three linear regression methods, LASSO regression produced the smallest MSE when evaluated on the Franke Function.

When fitting a 5th order polynomial to real terrain data of Stavanger, we found the same: Lasso regression produces the smallest MSE - however, the MSE score found for LASSO was 52000, an incredibly large number - indicating that the 5th order polynomial fit was a failure to model our terrain data.

Contents

1 Introduction

The rapid rise of machine learning and AI in modern science has paved the way for numerous new technologies that us humans would not even think possible in the past. This new science has infiltrated every single field in science, be it physics, psychology or agriculture - each and every one of these fields find great use of machine learning and AI.

As this field grows ever faster, the need for new minds to understand, develop and apply these methods is obvious. Which is exactly what we are looking to do in this scientific report. Here we will discuss, and investigate on of the simplest machine learning methods - namely that of linear regression. We will build our program bit by bit, starting simple and extending the complexity of our program as we delve deeper into linear regression.

In this report we will attempt to make a linear polynomial model in order to model terrain data. We will start by building our program with dummy data produced by the Franke-Function [1]. We will make a polynomial fit to this function using different means of linear regression and resampling techniques, and assess the performance of each model using various error estimates.

To build our models we will use the methods of Ordinary Least Squares (OLS), Ridge Regression and LASSO, and resample our data using both bootstrapping and cross-validation.

The necessary theory needed to understand the steps done in this report, as well as how the various method work - and how they've been implemented - will be presented neatly in the method and theory section. Lastly, the results from our analysis will be presented and discussed in the result and discussion section respectively.

[1]See: `https://www.sfu.ca/~ssurjano/franke2d.html`

2 Method and theory

2.1 Linear regression

[1]Linear regression is a method for building a linear model that describes a sampled distribution of a given random variable y and how it varies as function of another variable or set of such variables $\mathbf{x} = [x_0, x_1, ..., x_{n-1}]^T$. It's given that the first variable, y, is called a **dependent** variable, which gives the response of the system, while the set of $\mathbf{x}$ is the **independent** variable, called the predictor. The linear regression model seeks to find a compatible function $p(\mathbf{y}|\mathbf{x})$, where the estimation of the function is made by using a dataset $\{y_i, x_{i1}, ..., x_{ip}\}_{i=1}^{n}$. The linear regression model assumes a linear relationship of the mean of the response variable and the predictor variables. I.e a linear combination of the regression coefficients can be constructed. Furthermore, we assume the predictor variables to be fixed - and thus the linearity only restricts the parameters β to be linear, and not the predictors to be linear (since these are assumed fixed).

In this linear model, a random variable "noise", ϵ, is added as a factor between the relationship of x and y variables. Thus giving us the model:

$$y_i = \beta_0 + \beta_1 x_{i1} +, ..., +\beta_p x_{ip} + \epsilon_i = \mathbf{x}_i^T \beta + \epsilon_i, \quad i = 1, ..., n, \tag{1}$$

and hence we can write in matrix notation:

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

where,

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{bmatrix}$$

and,

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}.$$

2.1.1 Ordinary Least Squares

The method of ordinary least squares goes directly from equation (2), and assumes that the measured data $\mathbf{y}$, is composed of a "true" model $f(x) = \mathbf{X}\beta$ and the noise ϵ, and seeks to find a model

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

that can approximate the true model $f(x)$. By differentiation of the cost function(see [1]), we obtain the following expression for the parameters

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3}$$

As we can see, this is something we can easily compute numerically *if* the $\mathbf{X}^T\mathbf{X}$ matrix is invertible.

```python
import numpy as np
def OLS(DM, Z):
    """Generates the parameters beta for a linear OLS fit.

    args:
        DM      (np.array): Design matrix
        Z       (np.array): Measured datapoints

    returns:
        beta    (np.array): Parameters beta for the OLS model
    """
    beta = np.linalg.pinv(DM.T @ DM) @ DM.T @ Z.ravel()

    return beta
```

Code Listing 1: The following short piece of code illustrates how this can be done using NumPy.

You may notice we've used the function .pinv and not .inv, and this is to combat potential failures in our program due to numerical inaccuracies (i.e uninvertible matrices that in theory should be invertible etc.). Overall this is just a more robust matrix inversion tool. [2]
The punch in this short piece of code is that it does not mind the design matrix to be multidimensional, so this function can easily be applied to our Franke-Function, to obtain a polynomial fit in 2D.

[2]`https://numpy.org/doc/stable/reference/generated/numpy.linalg.pinv.html`

2.1.2 Ridge Regression

We will also implement the linear regression method of Ridge Regression[3]. This method of regression is in fact very similar to the Ordinary Least Squares (OLS) (2.1.1), but provides a solution when the matrix $\mathbf{X^TX}$ is non-invertible (it has 0 somewhere along the diagonal). The genius way to solve this, is to add a very small non-zero number, λ, to the diagonal, and then compute the inversion, as follows

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}. \tag{4}$$

```python
import numpy as np
def Ridge(DM, Y, lmb):
    """Performs the Ridge Regression analysis

    args:
        DM      (np.array): Design Matrix
        Y       (np.array): Measured datapoints

    returns:
        beta    (np.array): Parameters for our polynomial fit

    """
    hessian = DM.T @ DM
    dim1, dim2 = np.shape(hessian)
    I = np.eye(dim1, dim2)
    beta = np.linalg.pinv(hessian + lmb * I) @ DM.T @ Z.ravel()

    return beta
```

Code Listing 2: The following code will perform the Ridge Regression

Naturally, this method will be more robust and can be used without taking into account the correlation in the dependent variables.
However, from this non-zero λ, a new optimization problem arises:
Which values of λ provides the best estimate? We need to make a new estimate, by performing the Ridge Regression for various values of λ, and then find the optimal parameter λ that minimizes the cost function (MSE). The following explains in steps how this procedure can be done

- Define array of λ's
- Evaluate Ridge Regression using the same X,Y datapoints with the i'th value of λ
- Use the i'th model to calculate MSE (or any other estimand desired)
- Find (by plotting or min()/max() function) which λ_i provides the best estimates for your estimands.

```python
import numpy as np
```

[3]https://www.mygreatlearning.com/blog/what-is-ridge-regression/

```
lmbdas = np.logspace(min_lambda,max_lambda,nlambdas)
for lmb in lmbdas:
    model[i] = Ridge_Regression_function(X,Y, lmb)
    MSE[i] = MSE(Y, model[i])
optimal_indx = np.where(MSE == np.min(MSE))
```

Code Listing 3: This short piece of code illustrates how this can be done numerically.

2.1.3 LASSO Regression

Very similar to the Ridge Regression, where the key difference lies in how we define the cost function. Whereas for Ridge Regression we define the cost function with a penalty λ on the parameters beta as follows:

$$C(\beta) = \min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta}||_2^2, \tag{5}$$

using the norm-2 vector

$$||\boldsymbol{x}||_2 = \sqrt{\sum_i x_i^2}.$$

which, by minimizing this function we obtain our expression for β as presented in section (2.1.2). (see chap. 4 in [1]) However, the Lasso regression makes a different penalty on the parameters by using the norm-1 vector

$$||\boldsymbol{x}||_1 = \sum_i |x_i|.$$

Giving us a new cost function on the form

$$C(\beta) = \min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta}||_1,$$

This does not have a pretty analytical expression for defining the parameters β, and such we need some aid when we are to implement this regression method. To perform this regression we will use **Sci-kit learn**, which has LASSO regression[4] implemented in it's linear regression package. The following illustrates how to import and use this method

[4]`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html`

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

def LASSO(X, Y, lmbda):
    """Performs the LASSO regression method

    args:
        X           (np.array) : Design matrix
        Y           (np.array) : Measured datapoints
        lmbda       (float) : Value for the lambda-penalty

    returns:
        beta    (np.array): parameters beta for our Lasso regression fit
    """
    lasso_reg = Lasso(lmbda)
    lasso_reg.fit(X,Y)
    model = lasso_reg.predict(DM)
    beta = lasso_reg.coef_

    return beta
```

Code Listing 4: This code shows the method of LASSO.

And as we can see, we meet the same optimization problem like we did with Ridge Regression. We should iterate over multiple values for λ to find the optimal parameter to fit our model.

2.2 Assessment methods to evaluate linear regression models

In order for us to properly assess whether or not our models are successful in fitting our data, it is not sufficient to merely study the graph as this is tedious and not something that can be done efficient should we produce multiple models (as we will see when do resampling).

What we need are ways to evaluate our models performance by numbers, such that we can use these quantities to compare results from various ways of linear regression - be it different methods or using different parts of the datasets (resampling).

2.2.1 Mean squared error

Perhaps the most well-known way of assessing model performance is the mean squared error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2, \quad i = 0, .., n. \tag{6}$$

Which gives us an estimate of how much our model deviates, on average, from the measured data in each single point. This is a simple, yet efficient (in many cases), way of measuring the performance of our model fit.

```python
def MSE(data, model):
    """Calculates the Mean-Squared-Error given measured data, and model data.

    args:
        data    (np.array): Measured data
        model   (np.array): Model data from the linear model

    returns:
        MSE     (float): Mean squared error of the data and model.
    """
    mse = np.mean( (data - model) **2)

    return mse
```

Code Listing 5: The following piece of code will give the MSE.

If we are working in multiple-dimensions we must make sure the two arrays are of equal dimensions, and be they matrices we can use the $.ravel()$ function in numpy to make 1D arrays of all the datapoints contained in the matrix.

A drawback of the MSE is that it is not scaled, meaning that outliners (points that deviate greatly, corresponding to faulty measurements) contribute the same amount as any other point - which may give us a skewed value for the MSE, fooling us to believe our model fits nicely when it in fact may not fit at all.

2.2.2 R^2 score function (coefficient of determination)

The R^2 score function can be thought of as a more robust version of the MSE. It will give you a scaled estimate of how much the model deviates from the dataset, much like the MSE. However, we scale this quantity by the variance of the datapoints - meaning, outlier that deviate extremely will also be scaled down greatly - and this negates the problem we mentioned when talking about the MSE.
The way to calculate the R^2 score is as follows

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2},$$

where $\bar{y}$ is the mean value of our measured data, y, and $\hat{y}$ is again our modeldata. We recognise the numerator as the MSE, and the denominator as the variance, scaled by a factor n, where n is the number of datapoints.

```python
def R2(data, model):
    """Calculates the R2 score value

    args:
        data    (np.array): Array of measured datapoints
        model   (np.array): Array of modelled datapoints (must be of equal length as data)

    returns:
        r2      (float): The R2 score value
    """
    n = len(data)
    data_mean = np.mean(data)
    r2 = 1 - np.sum((data - model) ** 2) / np.sum((data - data_mean) ** 2)

    return r2
```

Code Listing 6: This method will estimate the R^2 score.

2.2.3 Bias-variance decomposition of the mean-squared-error

We assume that our measured data consist of a true model plus some noise, i.e

$$y = f(x) + \epsilon,$$

where the noise follows from a normal distribution, which further gives us that

$$Var(\epsilon) = \sigma^2.$$

Say that we now approximate a model to fit our data, $\hat{y}$, we can evaluate the fit of our model by calculating the mean squared error:

$$MSE = E[(y - \hat{y})^2].$$

This however, can be rewritten in terms of the bias of our model, the variance in the noise and the variance of our model, as follows

$$MSE = \text{Bias}[\hat{f}]^2 + \sigma^2 + \text{Var}[\hat{f}]. \tag{7}$$

This means, we can evaluate our model by looking at the *bias*: We can interpret this as an error caused by our models inherent characteristics, e.g when approximating a polynomial of degree n with a polynomial of degree m - there must be some error due to the dimensional difference between the two polynomials. On the other hand, the *variance* is a measure of how much the model could possibly change around it's mean value. When fitting a polynomial of degree n, with a polynomial of degree m, when $m > n$ the higher order terms can vary greatly, while still producing a similar fit to our data.
(See appendix (A.4) for full derivation of the decomposition)

2.2.4 Bias-variance tradeoff

An alternative way to evaluate the performance of our model fit is to study the relation between bias and variance of our model. It can be shown that the MSE of a model is decomposed into the variance and the bias of said model (7). In other words, there is a direct connection between low variance $\leftrightarrow$ high bias, and vice versa, for different levels of complexity of our model fit.

As seen from the Bias-Variance decomposition, we minimize the cost function by simultaneously minimizing bias *and* variance. But we need to be careful, if we make our model to minimize the bias $\rightarrow$ consequently the variance will increase, and then we may end up overfitting (i.e we start fitting the noise). Similarly, trying to minimize the variance will increase the bias and we may end up underfitting (i.e our model does not properly link the relation between response and predictor variables). Nonetheless, studying the Bias-Variance tradeoff will give us insight into how well our model fit is for various levels of complexity - while also giving us an estimate on bias and variance - meaning we can ascertain if our low value for MSE is indeed a good fit, or if we've either over- or underfitted our data. The following plot summarizes our statements

`Project template/figs/biasvariancetradeoff.png`

Figure 1: This figure shows a very general illustration of the Bias-Variance tradeoff [5]

[5]Source: `http://scott.fortmann-roe.com/docs/BiasVariance.html`

2.3 Resampling methods

2.3.1 Bootstrap

Bootstrapping is a method to assign best measures of accuracy to sample estimates. The basic idea of the bootstrap method is that we use random sampling with replacement out of a given distribution of data (X, Y)

With this resampling technique, one produces new dataset X_i, Y_i calculates a new model fit, e.g β_{new}, which one can now use to produce new values for different estimands (normally MSE,Variance and Bias). This procedure is repeated over n iterations, where n is your desired number of bootstraps.

Now one uses these n values for the estimands to calculate a mean value for each estimand, which in turn will give you an accurate approximation for the *true* value for each estimand.

```python
def bootstrap(X_train, Y_train, X_test, Y_test, niteration=100):
    """ Performs bootstrap
    args:
        X_train     (np.array): Array of trained datapoints
        Y_train     (np.array): Array of trained datapoints
        X_test      (np.array): Array of measured datapoints
        Y_test      (np.array): Array of measured datapoints

    returns:
        model      (np.array): Matrix with given measured column vectors


    """
    model = np.zeros((np.shape(Y_test)[0], niteration))
        for i in range(niteration):
            new_X, new_Y = resample(X_train, Y_train)
            model[:,i] = X_test @ OLS(new_X, new_Y).ravel()


    return model
```

Code Listing 7: Given method for Bootstrap

2.3.2 Cross-validation

Cross-validation is a method of model validation which splits the data in different ways of order to accomplish the better estimates of so called real world performance, and minimize validation error.
By using K-fold validation which is a very precise method of cross-validation, we shuffle the data and splits it into k number of folds. The basic idea of K-fold validation is to take one group as the test data set, and the other $k-1$ groups as the training data, fitting and evaluating a model, and recording the chosen score. We then repeat this process multiple times for each fold as the test data and all the score averaged to obtain a more comprehensive model validation score.

Project template/figs/K-fold-cross-validation-method.pdf

Figure 2: K-fold cross-validation method

```
def cross_val(X, Y, k = 10):
    """ Performs Cross validation
    args:
        X       (np.array): Design matrix
        Y       (np.array): Measured data
        k       (integer) : Number of folds (default = 10)

    returns:
        model    (float)  : MSE score after cross validation
    """
    kfold = KFold(n_splits=k)
    score_KFold = np.zeros(k)
    model = np.mean(score_KFold)
    for i, (train_inds, test_inds) in enumerate(kfold.split(X)):
            X_train = X[train_inds]
            Y_train = Y[train_inds]

            X_test = X[test_inds]
            Y_test = Y[test_inds]

            beta = OLS(X_train, Y_train)
            model = X_test @ beta
            score_KFold[i] = MSE(Y_test, model)

    model = np.mean(score_KFold)

    return model
```

Code Listing 8: This short piece of code illutrates how Cross validation kfold method is implemented.

2.4 Pre-processing of data

2.4.1 Train-Test splitting of data

When working with machine learning algorithms, and fitting methods in general - especially when we are worried our model might be overfitting, a good practice is to split our datasets into training and testing data. What this means is that we split our data into two separate datasets, and we use the training data to produce our model (to train our algorithm), and then we evaluate our model on the test data.

If we don't do this we cannot guarantee that our model fit is good, it may have a low MSE score because we've overfitted the dataset to fit perfectly to this specific dataset - meaning it may be a rather poor model even if we fit said dataset perfectly. When splitting the data we avoid this issue, by keeping the two datasets separate. This way we can create our model using training data, and check that the model works as intended on a completely "different" dataset. In this numerical report we will implement **scikit learns**[6] train-test-split function to help us split the data.

```
from sklearn.model_selection import train_test_split
def splitdata(X,Y, testsize=0.33):
    """Splits the given datasets into training and testing data
                        using sklearns traintestsplit function

    args:
        X               (np.array): Design matrix
        Y               (np.array): Measured data
        testsize        (float): The splitsize, i.e what percentage will be testdata (default = 0.33)

    returns:
        X_train         (np.array): Training set of design matrix
        X_test          (np.array): Test set of design matrix
        Y_train         (np.array): Training set of measured data
        Y_test          (np.array): Test set of measured data
    """

    X_train, Y_train, X_test, Y_test = train_test_split(X,Y,test_size=testsize)
    return X_train, Y_train, X_test, Y_test
```

[6]

Code Listing 9: The following code shows how this can be imported, and used.

2.4.2 Mean normalization and feature scaling

When working with various datasets, and different design matrices, it is not given that every feature is similar to oneanother. E.g we may produce a design matrix where one feature is the age of a subject, while the next feature corresponds to his or hers yearly salary. Meaning, one feature ranges from 0-99 while the other can range from 0 - 10^6. When we looked at the cost functions for Ridge and Lasso regression (see section (2.1.2), (2.1.3) respectively) we saw that these model penalize each β_i by the same factor, regardless of their size. What this means is that larger values for beta will dominate when we calculate the cost function. The following plot illustrates how this affects gradient descent (which we can compare to our linear fit by thinking that our unscaled datapoints will jump greatly and thus it will be more challenging to fit a model, while when they are all scaled equally this *should* be simpler, or atleast give us a smaller MSE)

`Project template/figs/normalize_gradient_descent.png`

Figure 3: Illustration of the necessity of datascaling[7]

One way to combat this problem is to introduce different penalties, λ_i, for each β_i, but this is tedious. What we can do instead, is to *scale* our design matrix. The method we will implement in this report is the method of **mean normalization**.

Mean normalization scales each column by its mean. When doing mean scaling on a design matrix, this corresponds to removing the first column (since this = 1 on every row, the mean normalized version has 0), and thus removing the intercept. The strength in this is that now all features will have the same meaning of 0, with this we mean that even if the features corresponds to wildly different quantities e.g kilograms and money, the value 0 has the same meaning - and thus the model fit have the same starting point for all features. (This can be taken a step further by dividing by the st.deviation, but since our features are not in different units we will refrain from doing so).
The following illustrates by steps how one should use mean normalization when performing linear regression with a given method

- Aquire dataset X, Y

[7]Source: `https://www.jeremyjordan.me/batch-normalization/`

- Train test split dataset
- Construct new datasets:
  - scaledX = Xtrain - column-mean(Xtrain)
  - scaledY = Ytrain - column-mean(Ytrain)
- Fit data = method.fit(scaledX, scaledY)
- Find model = method.predict(Xtest - column-mean(Xtrain)) + column-mean(Ytrain)

The reason we use the same column mean for both scaling test and training data, is due to the fact that one should leave the test dataset untouched in all forms.
Furthermore, you may be puzzled as to why we need to add the column-mean(Ytrain) to our model - this comes from the fact that mean normalizing removes the intercept, and adding this mean will simply reinstate the intercept.

```python
def mean_scale(data):
    """Mean-scales the dataset by the mean of each column vector in data

    args:
        data        (np.array): meshgrid of datapoints

    returns:
        new_data    (np.array): Meshgrid of scaled datapoints (now also centered)
        mean_data   (np.array): Array of the column mean for data.
    """

    mean_data = np.mean(data, axis=0, keepdims=True)
    new_data = ( data - mean_data )

    return new_data, mean_data
```

Code Listing 10: The following code will mean normalize a given dataset.

2.5 Analysis of real terrain data: Stavanger

We want to implement OLS, Ridge and LASSO onto a real terrain data where we use crossvalidation as a resampling method and calculate the MSE to evaluate a model fit.
We have this following method to

- Obtain data for spesific region
- Specify your dataformat to be: **SRTM Arc-Second Global**
- Download data as **GeoTIF** file.
  - This conducts the files to *tif* format
- Import *tif* file to a python program using *scipy.misc.imread*

```python
import numpy as np
from imageio import imread
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

#Loading terrain data

terrain_ = imread('SRTM_data_Norway_1.tif')
```

Code Listing 11: This code loads terrain data.

3 Results

3.1 Ordinary Least Squares

Using the methods presented in the previous section 2, we make a polynomial fit of the Franke Function. We choose $N = 50$, where N is the number of datapoints in the axises $x, y \in [0, 1]$, and a $\sigma^2 = 0.15$ to produce noise (gaussian distribution), unless otherwise specified. Figure (4)) illustrates how this polynomial fit looks compared to the noisy data from the franke function.

`Project template/figs/Franke_5deg_approx.png`

Figure 4: Comparison of our noisy data, and our 5th order polynomial model using OLS

When dealing with a two-dimensional polynomial fit it is rather difficult to assess a model. We implement our methods (2.2.1) MSE and (2.2.2) R^2 score function to help us assess that model.

3.1.1 MSE and R^2

By following the implementation of the given method in subsection(2.2.1) and subsection (2.2.2), using $N = 50$ data points we produce values for the MSE and R2 score function, which we've plotted against each other for rising degrees of polynomials. This is shown in figure(5)

Project template/figs/task_b/MSE_MSEtrain_R2_plot.pdf

(a)

Project template/figs/task_c/MSE_comp_test_train.pdf

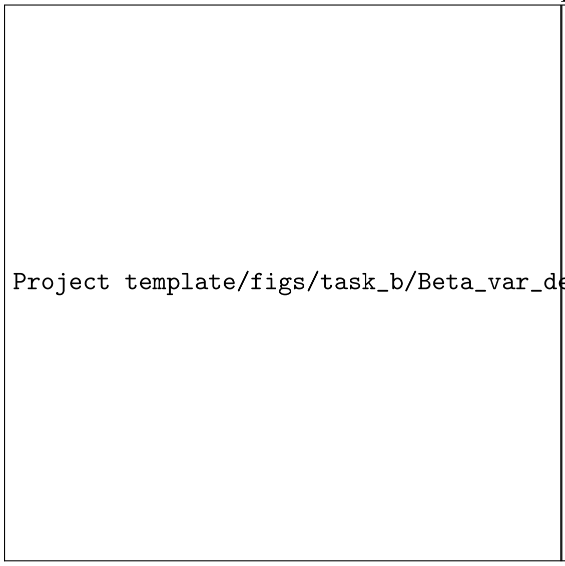
(b)

Figure 5: This figure shows $\mathrm{MSE}_{\mathrm{testdata}}$, $\mathrm{MSE}_{\mathrm{train}}$ and R^2 score as a function of the model complexity, for polynomials up to the degree of 5th order. Figure(b) illustrates a closer look at the comparison between train and test data of mean squared error as funtion of model complexity.

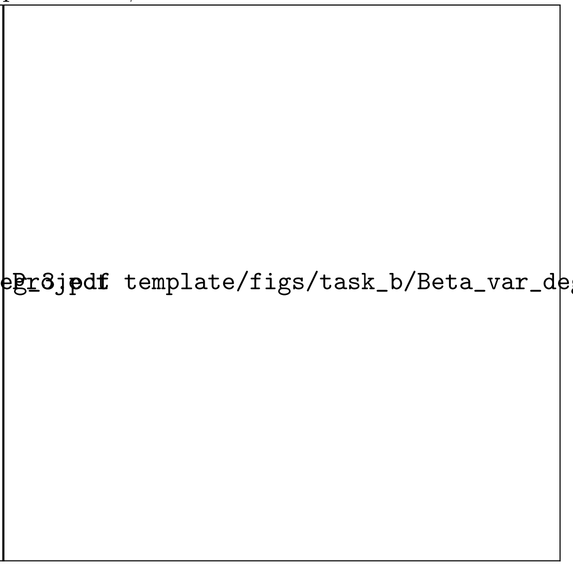
3.1.2 Variance of parameter β

Another interesting quantity we want to study to assess our model is how much variance is in our parameters β. We do this by using NumPys variance function [8], and illustrate this by 2 different order of polynomials as seen in the following plots shown in figure (6).

Variance of parameter β



(a)



(b)

Figure 6: In this figure we present the variance of the parameters β_i. Note we've normalized both the β_i's and the variance. Here the i'th order terms corresponds to the higher order terms in the two-dimensional polynomial (i.e xy, x^2y, xy^2 etc), and we illustrate the for 2 given polynomials of degree 3 and 5 respectively.

[8]See documentation: `https://numpy.org/doc/stable/reference/generated/numpy.var.html`

3.1.3 Bias-variance tradeoff

We use the bootstrap method introduced in section (2.2.4), we obtain values for MSE, Bias and variance for our OLS model. As seen in (9) these 3 values are closely linked, and we plot these together to study the bias-variance tradeoff to our method. We included polynomial degree up to 25th order to illustrate better this tradeoff, and we ran with 15 bootstrap iterations. The resulting graphs are shown in figure(7)

Project template/figs/task_c/bias_var_15boot_n30_seed2001.pdf

Project template/figs/task_c/bias_var_15boot_n50_seed200

(a)

(b)

Figure 7: This figure shows us the bias-variance tradeoff as function of complexity, where complexity represents the order of the polynomial fit. In this figure we consider $N = 30$ and $N = 50$ respectively.

3.1.4 Cross validation vs Bootstrap

By using the two different methods of resampling, cross validation (2.3.2) and bootstrap (2.3.1), we obtain unique estimates for the MSE. In the crossvalidation we run the resampling method for various number of folds (5, 10 and 15) - and we run for a set number of bootstrap iterations (25). This is done for various degrees of complexity in our model. By implementing the code(7) presented in section (2.3.2) we plot the MSE estimates as a function of polynomial degree, as illustrated in figure (8)

`Project template/figs/task_d/MSE_CV_boot25_n50.pdf`

Figure 8: This figure shows MSE as a function of model complexity up to $5th$ polynomial order. We evaluate a comparison between kfold cross validation and bootstrap. Where in this figure we estimate kfold $= [5, 10, 15]$.

3.2 Ridge regression

We are now doing many of the same steps as we did for the ordinary least squares, but on the method of Ridge Regression. We implemented the code presented in section (2.1.2) and choose a λ interval on the logarithmic scale of [-3,1]. We are still using $N = 50$

3.2.1 Bias-variance tradeoff with RIDGE

By implementing Ridge Regression method, we produce the bias-variance tradeoff as we did for OLS. We choose 3 arbitrary λ values within the range $\lambda \in \log([-3, 1])$ to perform the Ridge regression, and we study how the bias-variance change with increasing λ. The results are presented in figure (9)

Project template/figs/task_e/Bias_var_n30_boot15_ridge.pdf

(a)

Project template/figs/task_e/Bias_var_n30_boot15_ridge_lmb0

(b)

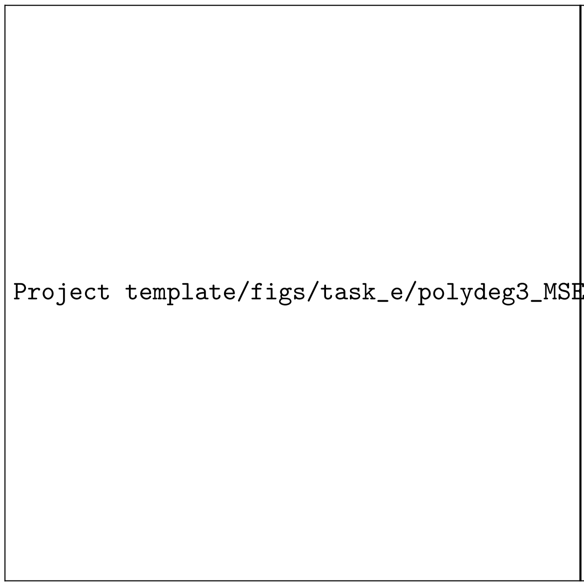
Project template/figs/task_e/Bias_var_n30_boot15_ridge_lmb27.pdf

(c)

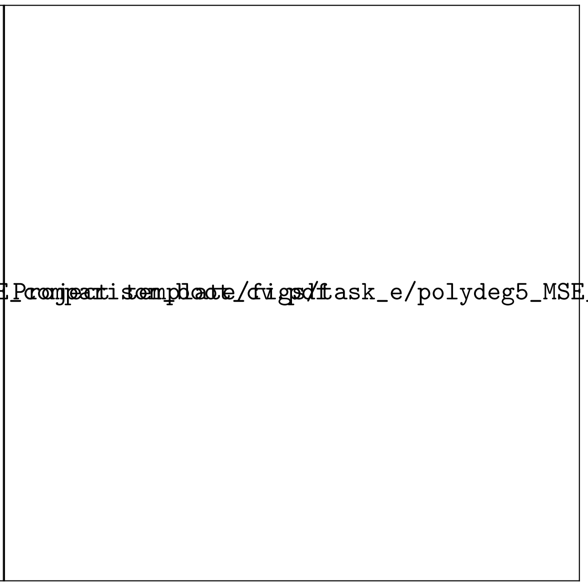
Figure 9: Figure (a), (b) and (c) shows error estimations as function of model complexity, where model complexity is presented as $ith-$ polynomial order. We use three different λ values for each subplot, where we have $\lambda = 0.000359$,$\lambda = 0.016681$ and $\lambda = 2.782559$ for each subplot.

3.2.2 Cross validation vs Bootstrap with RIDGE

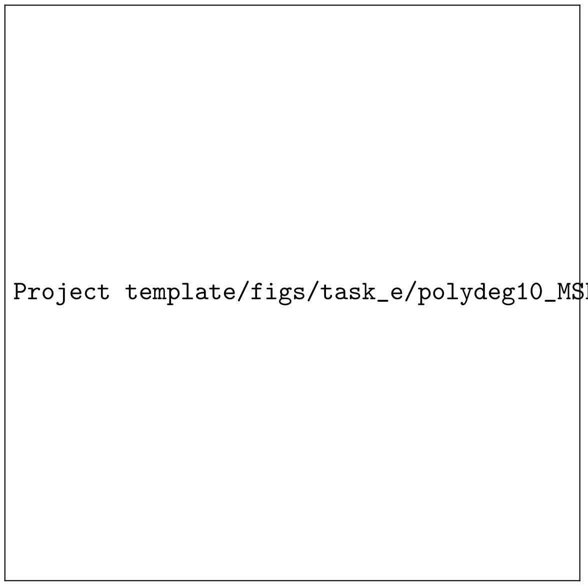
Now we'd like to compare the two different resampling methods as we did for OLS (see section (3.1.4)). We run for the same number of k-folds and the same amount of bootstrap iterations, still on $N = 50$. Doing so produces the following plot


Project template/figs/task_e/polydeg3_MSE_comparison_boot_cv.pdf

(a)



(b)


Project template/figs/task_e/polydeg10_MSE_comparison_boot_cv.pdf

(c)

Figure 10: Figure (a),(b) and (c) shows MSE as a function of $log10(\lambda)$, where we have that the x-axis is a logarithmic range of lambda values. We illustrate a comparison between kfold crossvalidation vs bootstrap, where we implement kfold to range [3, 10, 15] for a 3rd, 5th and 10th order polynomial.

3.3 LASSO: Least absolute shrinkage and selection operator regression

Now we are doing the same analysis as in section (3.2) for the method of Lasso regression that we presented in sectino (2.1.3). Again we are using the same spectrum for the λ values.

3.3.1 Bias-variance tradeoff with LASSO

We do the same bias-variance analysis using bootstrap with 25 iterations, as seen previously, but this time using the linear model produced by LASSO regression. The results (for the same values of λ) are presented in figure (11)

Project template/figs/task_f/Bias_var_n30_boot15_lasso_lmb000359.pdf

(a)

Project template/figs/task_f/Bias_var_n30_boot15_lasso_lmb0

(b)

Project template/figs/task_f/Bias_var_n30_boot15_lasso_lmb27.pdf

(c)

Figure 11: Figure (a), (b) and (c) shows Error estimation as a function of model complexity, where model complexity tells us the i'th order of polynomial degree. There is a variation in λ values for each subplot as shown, where we have $\lambda = 0.000359$,$\lambda = 0.016681$ and $\lambda = 2.782559$ for each subplot.

3.3.2 Cross validation vs bootstrap with LASSO

Again, we are to study the effect of the two resampling methods also for the LASSO regression. We perform the same analysis as in section (3.1.4) and (3.2.2), and the results are presented in figure (12)

Project template/figs/task_f/polydeg3_MSE_comp_boot25_n50_cv_lasso.pdf

Project template/figs/task_f/polydeg5_MSE_comp_boot25_n50_

(a)

(b)

Project template/figs/task_f/polydeg10_MSE_comp_boot25_n50_cv_lasso.pdf

(c)

Figure 12: Figure (a),(b) and (c) shows MSE as a function of $log10(\lambda)$, where we have that the x-axis is a logarithmic range of lambda values. We illustrate a comparison between kfold crossvalidation vs bootstrap, where we implement kfold to range [3, 10, 15] for a 3rd, 5th and 10th order polynomial.

3.4 Analysis of real terrain data: Stavanger

As mentioned in the method section, we are to test our numerical model on a real data set. We will use all three methods, OLS, ridge and lasso - and use the cross validation resampling method with $k = 10$ number of folds. We attempt in this section to fit a 5th order polynomial to the terrain data, and we calculate the MSE to evaluate the model fit. To ease the computational load we run with a smaller subset of lambdas, in the region $\lambda \in \log_10[-2, 1]$. Doing so produces the following graph of the MSE as a function of λ

`Project template/figs/task_g/real_data_MSE.pdf`

Figure 13: MSE for the two models: Ridge and Lasso as functions of λ

To evaluate which model works best for our new terrain data, we find the lowest value for MSE and compare this for the three models, as presented in table (1)

Reg.method	**MSE**	λ
OLS	71300	NaN
Ridge	53000	-0.9286
Lasso	52000	-0.0714

Table 1: Table of smallest MSE values for the various regression models

4 Discussion

4.1 Ordinary Least Squares

From the initial plot in figure (4) we can see that using OLS with a 5th order polynomial reproduces the surface somewhat. However, just studying the plots are not enough to deduce if the model is actually a good fit or not. This is especially true when working in multiple dimensions (if we go even higher, plotting will not even be possible!), and as such we must calculate the various error estimates that we've presented in the result section.

One thing to note before delving into the interpretation of the various results obtained using the Ordinary Least Squares is that we've not scaled any of the data used within this method. [2]This is due to the OLS method being scale equivariant and as such scaling the data will only scale the fit linearly - and the actual model fit will be the same by a scaling factor - meaning we can disregard this, to save on the computational load. This load may very well be quite large when working with real datasets.

4.1.1 MSE and R^2

The simplest way to check if our model fits the data nicely, is to study the error between the model and data. We do this by checking MSE and R2 score - as we've done in figure (5). What we are looking to find, is a point where the R2 score is maximized while the MSE is minimized. This seems to occur at polynomial degree 4. Since we've used the Franke Function[9] to produce our dataset this should make sense that a higher order polynomial produces the best results.

When looking closer at the relationship between the MSE produced from using the train and test data respectively we see that these deviate greatly for lower polynomial degrees, but then they tend similar values for higher orders. One would expect the train MSE to also be quite large at first order - and this could indicate that we've overfitted the data in this particular example. The large deviance between train and test MSE also indicates areas of high variance. As we've mentioned in the theory section (see (2.2.3)) a high variance in our model means that the model varies greatly when we change the input data, i.e when we've split our dataset according to (2.4.1) and evaluate our model on train and test data respectively, a high variance will naturally produce a high discrepancy between the two estimates, while a low variance would not.

And as seen from the Bias-Variance decomposition of the MSE (see (9)), we know that a high variance should correspond to a low bias (unless the MSE is extremely high, but in relative magnitude these two contributions should oppose eachother) - this flows naturally into our argument that the model varies greatly for different datasets with high variance - because a high bias would then

[9]See: `https://www.sfu.ca/~ssurjano/franke2d.html`

correspond to the models attraction to produce the same data for any data set, i.e low variance just as we've proven with the bias-variance decomposition.

4.1.2 Variance of parameter β

In these results shown in figure(6) one can see a clear change in variance of parameter β especially when we increase to a higher order of features. This makes a lot of meaning aswell since when the features increases our degrees of polynomial increases as well and that gives a higher order of spread in our data set. And when we have higher order of spread in data, the larger the variance is in relation to the mean.

But we can also see in figure(6) that the variance of parameter β doesn't always necessary increase when the complexity increases. The complexity increases when the features is set to be more complex. This is given when the set of features have a higher degree of difference. We can see that the complexity increases when we have a greater degree of difference in the exponent of these variables in each set of features. By this I mean that if X and Y have a large degree of inequality in the exponent, i.e $[x^4y^6, x^6y^4, x^5y^5]$ etc, then the complexity will increase effectively much more than if there was not such a large difference in the exponent of X and Y, thus giving higher variance, which is proven to be true in our case and equivalent to figure(6)

4.1.3 Bias-variance tradeoff

In these results we can see a clear pattern recognition between figure(7a) and figure(1), where we see that once there is an increase in variance there is also a decrease in bias, which we can also look at when we examine the (9). This can be interpreted in (1), where we obtained precise data results.

Further we can also observe that the bias and variance changes from 5th to 25th order of polynomial degree. That is also due to the changes of complexity, which is also very good data results, and the reason of this were discussed earlier when we were looking at the variance of parameterβ4.1.2. We can also see that the MSE increases severely due to the changes in bias and variance, which is also set to be proven in (9) and this follows the increasing mean squared error, which is very clear. Once the polynomial degree increases we gain a higher order of complexity within our model, this is very accurate due to the higher order of features and hence the degree of spread in our data set.

4.1.4 Cross validation vs Bootstrap

When performing linear regression and training a model against a data set of our own making we are free to produce as much data as we want, i.e we can increase number of datapoints until our computational load hits the moon. This is not the case when we are studying real life problems, in these problems the data are limited, and thus producing a proper fit may prove difficult. This is why we are to use the resampling methods introduced in the theory-method section (see (2.3)), to produce "new" datasets from our existing sets.

Another feature of resampling is to guarantee that we train our model using as much of the available dataset as possible, exactly for the reason mentioned: We have limited data, so we better use it efficiently. As explained in the methodsection (2.3), these two methods tackle this issue in different ways. While the Bootstrap method keeps the test dataset completely separate and produce "new" training data from the existing training set, the crossvalidation method splits the data into folds, and lets every fold get its turn as a test set, no discrimination whatsoever. As they tackle the same issue in separate ways, we expect them to produce similar results, but not necessarily the same. For the crossvalidation method we can tweak the number of folds, while bootstrap tweaks the number of resampling iterations.

Looking at the illustration in figure (8), ran for a set number of bootstrap iterations (25), the crossvalidation method becomes equally good, and even slightly better for some polynomial fits when we pass the $k = 10$ folds. Exactly why this is, has to do with the size of our dataset. We've produced a grid of data points $50x50$, and if we increase the number of folds we will fit the data extremely well due to each fold being a narrower portion of the dataset. If you've 10 datapoints and 10 fold, then you fit the data 10 times on each data point - naturally this will fit your data extremely well - however, we should be on the lookout for overfitting of the data, if you make too many folds on a very small dataset, the noise will make a greater contribution to the model fit and we run into the problem of overfitting.

4.2 Ridge regression

Now that we move onto Ridge Regression we need to evaluate the pre-processing of our data. As opposed to the necessity of data scaling when working with the OLS, the Ridge Regression method *is* affected substantially when we scale our data - and it is indeed very important that we do so. We will use the mean normalization method as presented in section (2.4.2), and the reason for this can be seen directly from the cost function for the Ridge Regression (see (5)), that the parameters β_i all contribute to the MSE, with the same λ, as explained in (2.4.2)

4.2.1 Bias-Variance tradeoff

As we did with the Ordinary Least Squares we perform the linear regression using Bootstrap resampling to provide more accurate estimates of our desired estimands, namely the MSE, Bias and variance. In figure (8)) we've performed this analysis for 3 selected values for lambda. What we expect to find is that for an increase in λ, we should see a decrease in the models variance. This can be seen mathematically from the analytical expression for the parameters β in eq. (**??**).

In the figure this may not be overwhelmingly clear, but as we rise in value of λ we can see that the bias line edges closer and closer to the MSE line. As we remember from the Bias-Variance decomposition, this must mean that the variance is getting smaller. This is our desired, and expected result - a higher value of lambda means that the $\lambda \mathbf{I}$ term will dominate when determining the parameters β, and thus changing the features inside $\mathbf{X}$ will contribute less - i.e vary the model less (meaning variance is low).

4.2.2 Cross validation vs. bootstrap

Again we will study how the various resampling methods affect our new linear regression model. We would expect to see much similar results in this, like we did for the ordinary least squares (see (8)) - but this time we illustrate this as a function of the new penalty parameter λ, for 3 different polynomials, as seen in figure (10).

Again, we'd expect the two methods to coincide when we increase the number of folds (since we run for a rather high number of bootstrap for such a small dataset). But the interesting part is how the MSE changes for higher values of λ in this analysis. We can see that in the 5th order polynomial bootstrap gives us a minima around $\log_{10}(\lambda) = -1$, while our crossvalidation method indicates this to be a maxima. However, this may come from the bootstrap method being very variant - i.e it jumps up and down quite much, so if we exclude the outliers in the x-axis it seems that the two methods coincide quite well for the higher values of k. What we mean by this, is that the overrall *shape* of the two curves have similar features around this value for λ.

4.3 LASSO

We do the same analysis for LASSO regression as we did for the Ridge Regression. Here we also need to perform mean normalization of our data in order to get a proper model fit, and for our MSE to not be evaluated wrongly due to unscaled terms (as mentioned this is not such a big issue for us, but we still need to perform the scaling to get proper assessment of our model).

4.3.1 Bias-Variance tradeoff

When we study the Bias-Variance of the Lasso regression method we find much of the same that we did for the Ridge Regression analysis. As a matter of fact, these two methods seems to produce almost the same results - which may not come as a surprise given that they are in fact very similar in how they define the cost functions, as we've discussed earlier (see (2.1.3)).

What we see in the results from the LASSO model is that the variance is almost non-existant (zero), regardless of choice of λ. However, one can see a slight difference in MSE and Bias^2 for the smallest value of λ. We may deduce that our model is overfitting the data from the fact that the Bias is so large. Another possibility is that we run the rest using very few datapoints, meaning the LASSO makes an amazing fit (and the fit stays the same even when we resample, since our datapoints lie between [0,1]) regardless of the resampled data set. A third possibility is that we've chosen too large values of λ. Since a high λ equals low variance. Knowing exactly which of these three possibilities are the reason for the shape of our graph is not clear just from reading the plots.

4.3.2 Cross validation vs. bootstrap

What we can see from the plots of crossvalidation versus bootstrap resampling for the LASSO regression case, presented in figure (12) - is that, as opposed to Ridge, bootstrap seems to outperform cross validation regardless of polynomial degree and for most values of λ. We do however, seem to find the same optimal value of λ for both cross validation and bootstrap.

4.4 Analysis of real terrain data: Stavanger

When performing our linear regression methods on a set of real terrain data produced from the beautiful countryside of Stavanger, we see from figure (13) that our polynomial fit is not a great success. From the y-axis you can see the immense values that the MSE takes, in the scale of 10^5. This indicates that a 5th order polynomial is practically useless to approximate such noisy terrain data as the one we have used in this example. We can further back this statement by looking at the values presented in table (1).

What we can deduce however, is that even though our model is quite poor - we can see that Lasso is the best fit for our data, albeit best of the very worst. Another interesting take from our results is that we still recognise similar traits in the cross validation plots, as we did for the Franke function: Ridge and Lasso has minima for different values of lambda, and where ridge has minima - lasso has (local) maxima.

5 Conclusion

When studying the Franke Functions 2D surface data with added noise, it seems to us that the optimal regression method is that of ordinary least squares. As we've seen from the produced plots and graphs, this is the method that produces the smallest MSE both using bootstrap and cross-validation resampling. (≈ 0.15 for the 4th order polynomial fit). Comparing the other two methods to eachother we find that there are rather miniscule differences, and the actual difference in MSE between the two is negligible. Our conclusion is that: For a simple function such as the Franke Function with noise as we've included it, the OLS regression method works rather well. If you do however wish to use a model that are more open for fine-tuning (i.e more parameters), there is only minor differences between LASSO and Ridge - and thus you may use whichever you find easiest to implement.

When we did our analysis of the real data, our 5th order polynomial model failed miserably at fitting the given data. Our MSE score was gargantuan, and we can conclude with GREAT certainty that a 5th order polynomial is not a good fit for 2D noisy terrain data. Despite our poor fit, we did however show that out of the three linear regression methods - LASSO was the best one, that produced the best fit with the lowest MSE score (closely followed by ridge). Improvements to our model could surely be made to make the polynomial fit even better, by either increasing the polynomial order (our educated guess is this to be the most effective measure) or by testing for more values of lambda. The latter might not give any improved results, at least not any major improvements.

References

[1] M. H. Jensen, *Applied data analysis and machine learning*, Copyright 2021, 2021. [Online]. Available: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html#`.

[2] J. F. Trevor Hastie Robert Tibshirani, *The Elements of Statistical Learning*, 2nd ed. Springer Science+Business Media, LLC, part of Springer Nature 2009: Springer New York, NY, 1994, p. 745. [Online]. Available: `https://doi.org/10.1007/978-0-387-84858-7`.

A Appendix

A.1 Source code

All the source code is located in this GitHub repository.

A.2 Derivation of $Var(y_i)$

We have that:

$$\begin{aligned}
\text{Var}(y_i) = \mathbb{E}\{[y_i - \mathbb{E}(y_i)]^2\} \;&=\; \mathbb{E}(y_i^2) - [\mathbb{E}(y_i)]^2 \\
&= \mathbb{E}[(\mathbf{X}_{i,*}\,\beta + \varepsilon_i)^2] - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\
&= \mathbb{E}[(\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\varepsilon_i \mathbf{X}_{i,*}\,\boldsymbol{\beta} + \varepsilon_i^2] - (\mathbf{X}_{i,*}\,\beta)^2 \\
&= (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\mathbb{E}(\varepsilon_i)\mathbf{X}_{i,*}\,\boldsymbol{\beta} + \mathbb{E}(\varepsilon_i^2) - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\
&= \mathbb{E}(\varepsilon_i^2) \;=\; \text{Var}(\varepsilon_i) \;=\; \sigma^2.
\end{aligned}$$

A.3 Derivation of $Var(\beta)$

$$\begin{aligned}
\text{Var}(\boldsymbol{\beta}) &= \mathbb{E}\{[\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})][\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})]^T\} \\
&= \mathbb{E}\{[(\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{Y} - \boldsymbol{\beta}]\,[(\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{Y} - \boldsymbol{\beta}]^T\} \\
&= (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbb{E}\{\mathbf{Y}\,\mathbf{Y}^T\}\,\mathbf{X}\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \\
&= (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\{\mathbf{X}\,\boldsymbol{\beta}\,\boldsymbol{\beta}^T\,\mathbf{X}^T + \sigma^2\}\,\mathbf{X}\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \\
&= \boldsymbol{\beta}\,\boldsymbol{\beta}^T + \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \;=\; \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1}.
\end{aligned}$$

From $\text{Var}(\boldsymbol{\beta}) = \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1}$, we obtain an estimate of the variance of the estimate of the $j-th$ regression coefficient:

$$\boldsymbol{\sigma}^2(\boldsymbol{\beta}_j) = \boldsymbol{\sigma}^2[(\mathbf{X}^T\mathbf{X})^{-1}]_{jj} \tag{8}$$

.

A.4 Derivation of Bias-Variance decomposition of MSE

Start with the expression for our data

$$y = f(x) + \epsilon$$

and the following quantities

$$\begin{aligned}
\text{Var}(\epsilon) &= E[\epsilon^2] - E[\epsilon]^2 = E[\epsilon^2] = \sigma^2 \\
\text{E}[f] &= f \\
\text{Bias}[\hat{f}]^2 &= (f - E[\hat{f}])^2 \\
\text{E}[\epsilon] &= 0 \\
E[A+B] &= E[A] + E[B] \\
E[AB] &= E[A]E[B]
\end{aligned}$$

Lets compute the MSE for our data, assuming we've made a model $\hat{y}$, (here we use $f = f(x)$ to simplify our writing)

$$\begin{aligned}\text{MSE} &= E[(y - \hat{y})^2] = E[(f + \epsilon - \hat{y})^2] \\ &= E[(f - E[\hat{y}] + \epsilon + E[\hat{y}] - \hat{y})^2]\end{aligned}$$

Using the square theorem on this expression, and splitting the expectation value, yields

$$\begin{aligned}&= E[(f - E[\hat{y}])^2] + E[\epsilon^2] + E[(E[\hat{y}] - \hat{y})^2] + 2E[\epsilon]E[(f - E[\hat{y}])] \\ &\qquad +2E[\epsilon]E[E[\hat{y}]] - \hat{y}] + 2E[[E[\hat{y}] - \hat{y}]E[(f - E[\hat{y}])]\end{aligned}$$

now we cancel all the terms that are zero, i.e $E[\epsilon] = 0$, and $E[E[\hat{y}]] = E[\hat{y}]$ (meaning, the final term is zero). Be mindful however, the term $E[(E[\hat{y}] - \hat{y})^2]$ is non-zero, since this has a crossterm that do not cancel. Furthermore, the terms $E[(f - E[\hat{y}])] = (f - E[\hat{y})$, since the argument is just a scaling by $EE[\hat{y}]$ of the initial function f, which we found had $E[f] = f$. Meaning we are left with the following,

$$MSE = (f - E[\hat{y}])^2 + E[\epsilon^2] + E[(E[\hat{y}] - \hat{y})^2]$$

And these three terms we recognise from our initial list of useful quantities, giving us the decomposition we desire

$$\text{MSE} = \text{Bias}[\hat{y}]^2 + \sigma^2 + \text{Var}[\hat{y}] \tag{9}$$