Modeling Topographic Data Using Several Linear Regression Methods

Evan Matel & Martin Jakhelln

October 2023

Introduction

In this report, we attempt several regression methods, starting with ordinary least squares (OLS), Ridge regression, and Lasso regression. To start with we create our design matrix and expected results using the Franke function. We will also analyze the Bias-Variance trade-off while simultaneously testing sampling techniques, such as bootstrapping and cross-validation. We hope to obtain a deeper understanding of these techniques by analyzing how slightly changing certain parameters can drastically alter the result, and ultimately, to create a model that can accurately recreate topographic data by predicting the height values.

1 Method

1.1 Linear Regression Methods

We start by creating our data using two arrays, x and y, which are N elements long and are uniformly distributed between 0 and 1. We then set these arrays into the function *create_X* which was taken from lecture notes [1] and has our design matrix as the output. The expected values z come from the Franke Function, which is a function that gives a close approximation to a smooth surface with some added noise, which makes it commonly used for testing various fitting algorithms, and is as follows:

$$\begin{aligned} \text{f(x, y)} = {} & 0.75 \exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ & + 0.75 \exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}\right) \\ & + 0.5 \exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \end{aligned}$$

$$- 0.2 \exp\left(-\frac{(9x-4)^2}{4} - \frac{(9y-7)^2}{4}\right)$$

(1)

Since the inputs we give the Frank function are already uniformly distributed between 0 and 1, and the franke functions output is dependant on the input, there is no need to scale the data. We will introduce the concept of scaling later in the text when we start with our topographical data. We first use the Franke function in order to make sure our model is working correctly before applying it to a more complicated set of data like the topographical data we wish to analyze. We then split the data into training and testing sets using Sklearn's *train_test_split* function, and we do this in order to assess the performance of our model and with that, check for over-fitting. We then start fitting the data using different linear regression algorithms. Linear regression models are used when one does not have a function for the data, so one assumes a linear relationship, hence the name. They use the form $y = mx + b$, where y is the output, m is the slope, x is the input, and b is the intercept. The goal is to calculate the m and b values that give the wanted output.

Ordinary Least Squares (OLS) is a linear regression model that aims to estimate the unknown parameters of the model. OLS involves finding the values of β's that minimizes the sum of the squared differences between over test data set and the predicted data set

In the case of OLS, we have our data, or design matrix X, and we multiply that with β in order to get an output $\tilde{y}$(see formula 2).

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

Here $\tilde{y}$ refers to our predicted data and epsilon refers to the error or noise of our data where lastly, we find β by using this formula

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

We have the true labels of our data y and the goal is to find the β's that give us the output $\tilde{y}$ closest to the true labels. In other words, the whole point of these methods is to minimize the error between $\tilde{y}$ and y. This error is known as the Mean Squared Error (MSE) (see equation 4) and is one of the tools we use to asses our model's performance. While knowing the magnitude of error is very informative, it doesn't necessarily tell us exactly how well our model fits the data, that's where the R2 score comes in. The R2 score (see equation 5) explains the variability of the model, with a score of 1 meaning that the model is perfect and explains all variability, and a score of 0 meaning that all variability is unexplained.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y - \tilde{y})^2 \tag{4}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y - \tilde{y})^2}{\sum_{i=1}^{n}(y - \bar{y})^2} \tag{5}$$

The next method we use is Ridge regression, which is related to OLS. The difference between the two is another hyper-parameter λ, sometimes called a penalty term, whose purpose is to alter how much of an effect the β coefficients have on the data (see equation 6). Ridge is generally used for more complex models where there is a greater danger of overfitting if using OLS.

$$\beta_{Ridge} = (1+\lambda)^{-1}\beta_{OLS} \tag{6}$$

Another method that is similar to ridge is called Lasso Regression. Lasso is also used for more complicated models, and it can simplify the models by setting some features equal to zero when λ is large enough, effectively eliminating them. This can be useful for reducing dimensionality which can make it easier to interpret.

1.2 Re-sampling Techniques

A problem that permeates all of machine learning is a lack of data. When one does a long and difficult study, one gets a certain data set out of this study. If that data you get isn't enough, too bad. Therefore, a very important concept in machine learning is re-sampling. There are several re-sampling techniques, but we will only cover two, bootstrapping and k-folds cross-validation.

Bootstrapping works by first splitting the data into a training and test set as normal, and then altering the training set by randomly shuffling N data points with replacement. Using this method requires us to make equal amounts of new data sets as there are data points. This is the main reason that this re-sampling technique is not commonly used in science anymore, since when one uses this technique on a vast data set, which most real-world data is, it can quickly become very computationally heavy.[1]

K-folds cross-validation is another re-sampling technique where one splits the data set into $k+1$ different subsets, using k subsets together as the training set and the remaining subset as the test set. One then alternates which subset is used as the training set until all possibilities are exhausted.

1.3 Bias-Variance Tradeoff

Bias and variance are two key concepts in data science. The bias can be thought of as precision in results, meaning that all the results are centered around a point, and that point is a certain distance from the origin, that distance is the bias. Variance can be thought of as accuracy, where the more spread out the results are, the greater the variance (for a visual aid, see Figure 1). We will apply this technique by using formulas 7, 8, and 9, first on our simple model and then later for our topographical data model.

$$Error[\tilde{y}] = Bias[\tilde{y}] + var[\tilde{y}] + \sigma^2 \tag{7}$$

$$Bias[\tilde{y}] = E[(y - E(\tilde{y})^2] \tag{8}$$

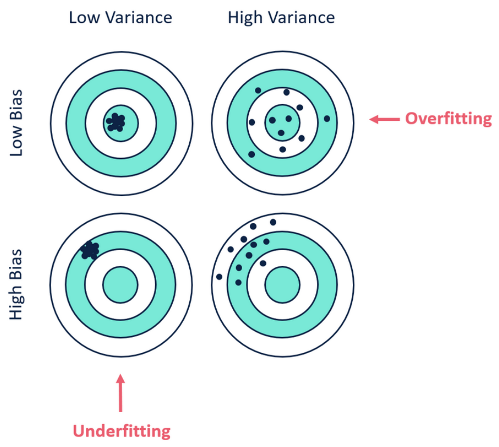

Figure 1: Visual aid for understanding bias and variance.[2]

$$var[\tilde{y}] = E[(y - E(\tilde{y})^2] = \frac{1}{n}\sum_i (y_i - E(\tilde{y}))^2 \tag{9}$$

1.4 Real Data Application

Once we had tested all of the previously mentioned methods and techniques on our created data set, we then took a real data set of 3D topographical data from Møsvatn Austfjell, Norway. This data set was larger and more complicated than the Franke function set, thus it required some manipulation, also known as scaling. There are many ways to scale data, but since the z-level, or elevation, differs so much (the data has a high variance) in the Norwegian mountains, we chose to use SKlearns MinMaxScaler function which scales everything between 0 and 1. After scaling the data we choose a sample size of topographical data, a 500 x 500 data set, and then run through all the previously mentioned methods, now for the topographical data set.

2 Results

2.1 Synthetic Data

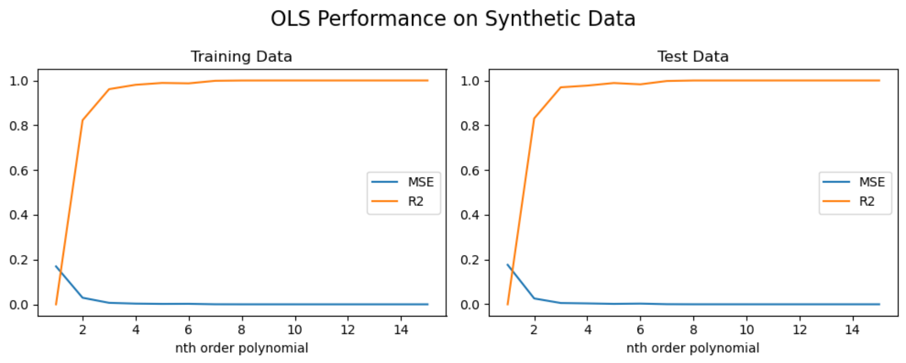

Figure 2

In Figure 2 we can see both the MSE and the R2-score for both the training data and the test data. We observe that both the training data and test data have a very low MSE and a high R2-score which is ideal for the accuracy of our model. Further, we observe a slight difference in training and testing which leads us to believe that there is no overfitting, at least not this far, we will discuss this later.

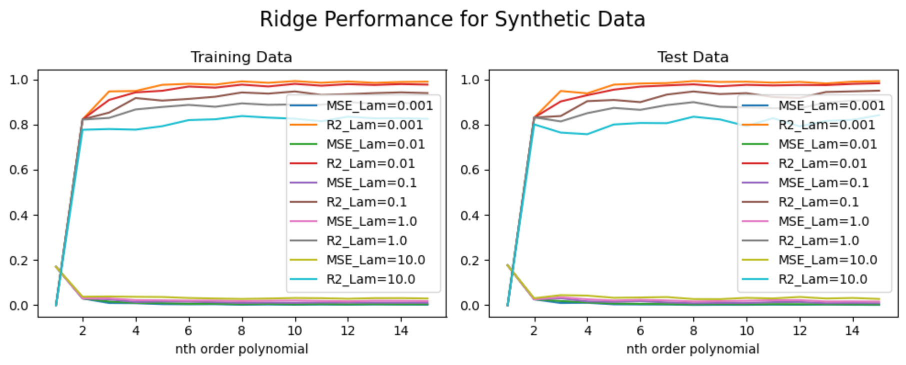

Figure 3

Figure 3 contains the MSE and R2-score for ridge regression for each different lambda, we chose a small array of lambdas containing a wide spread of values, and we will discuss further why we chose this. Generally, both the MSE and the R2-score assume desirable values but we also observe that some lambdas make them perform better, which reassures us that everything is functioning properly.

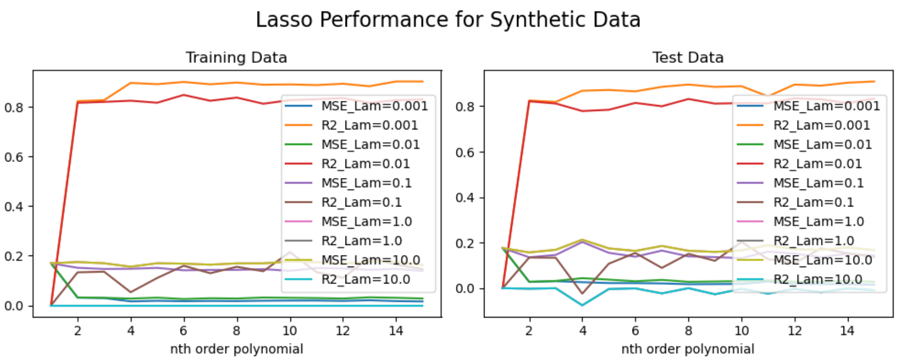

Figure 4

In Figure 4 we can see the MSE and R2-score for lasso regression for each different lambda, with the same choice of lambdas as for ridge regression. We observe that here there is a big difference in performance depending on what value lambda takes, which makes sense considering how lasso regression works. For some lambdas, the performance is good, while others perform horribly.

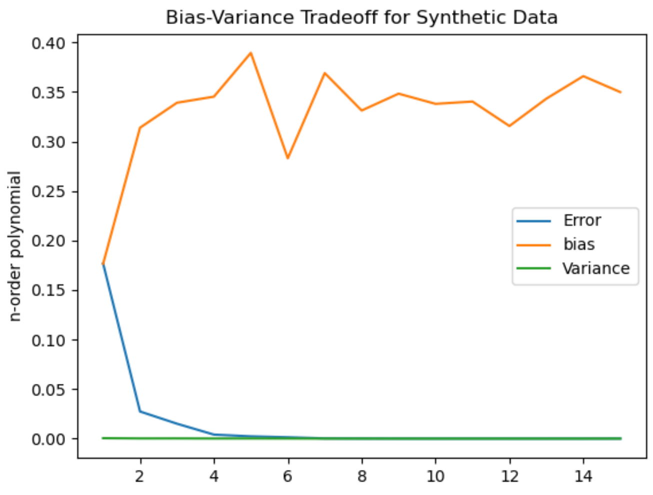

Figure 5

Observing Figure 5 we see that our program is not performing correctly, we would want this plot to show us bias and variance equal to error when adding the two together. However, we observe that bias is high and both error and variance are low. This leads us to believe that there might be a mistake in our program when calculating these values.

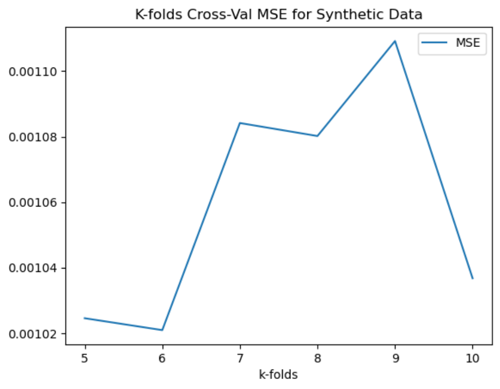

Figure 6

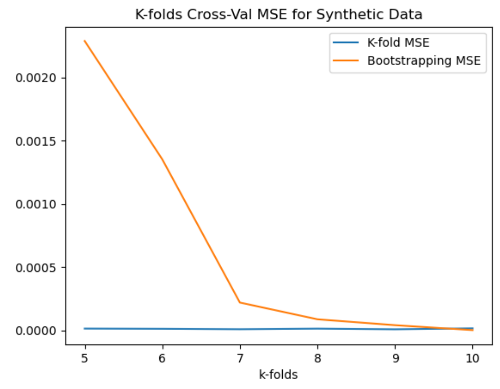

Figure 7: Enter Caption

Figure 6 shows our MSE for k-folds cross-validation and Figure 7 shows it compared to the MSE of our bootsrapping model. We can see that k-folds preformed much better even though it is less computationally heavy.

2.2 Topographic Data

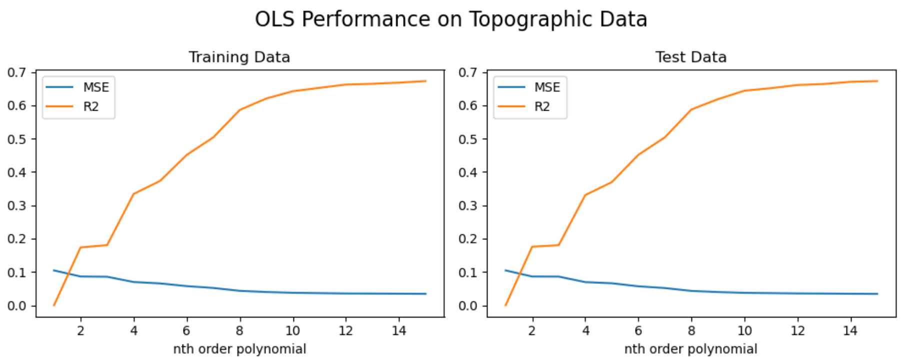

Figure 8

Figure 8 shows us the MSE and R2-score for our model for the topographical data set. Comparing this with Figure 2 we observe that both MSE and R2-score are performing worse, but these are still acceptable results since we now are working with a more complicated data set. Again there seems to be no overfitting based on this plot.

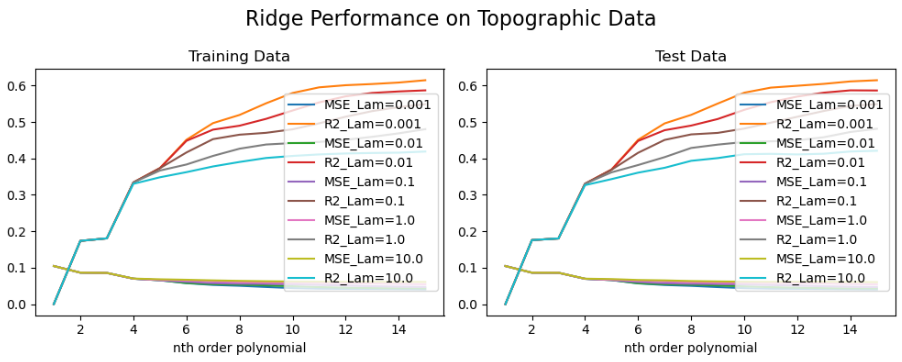

Figure 9

When we observe Figure 9 we see that both MSE and R2-score perform almost as well as our OLS model. Although the results are acceptable we would have wanted Ridge to perform better than OLS, which we will discuss further later. Again we chose the same lambdas and we observed that, as expected, some lambdas performed better.

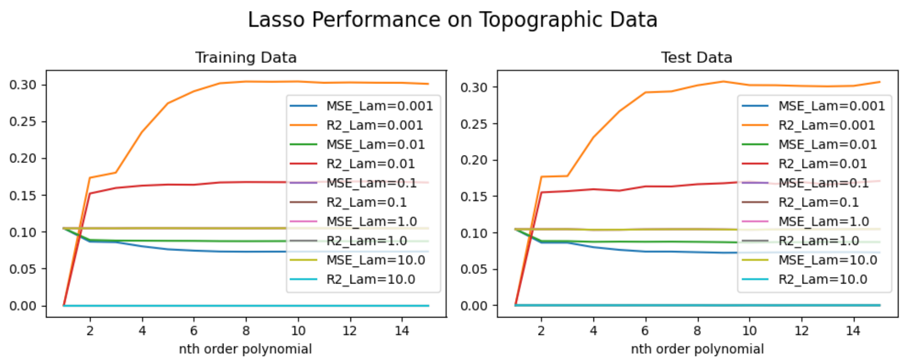

Figure 10

Figure 10 shows us the MSE and R2-score for different lambda values when performing lasso regression. Here we observe that the performance is not what we want it to be, although the MSE is good for some lambdas the R2-score is not good for any.

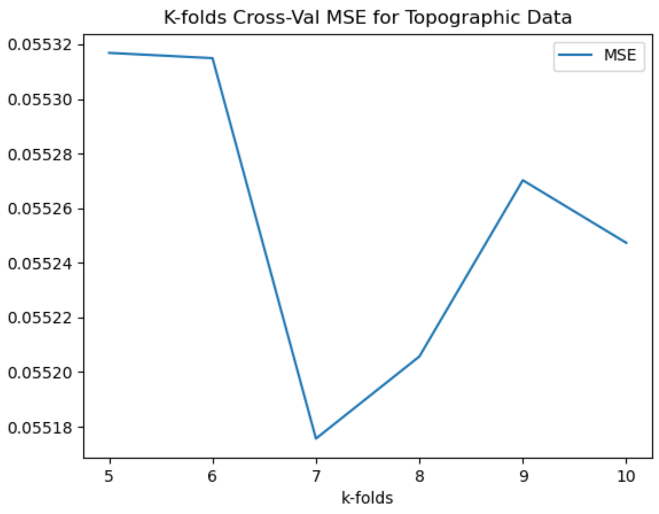

Figure 11

When observing Figure 11 we see k-folds doesn't work quite as well for our topographic data, which is to be expected, the MSE is still quite good but around $5x$ worse than for our synthetic data.

3 Discussion

Considering Figure 2 we think that so far there is no overfitting, this is because both training and test data perform close to equally well. This is also why we want a training and test set because if our model performs well on the training set and then performs badly on the test set, this is a clear sign of overfitting.

Moving to Figure 3 we observe that generally ridge regression is performing similarly well as OLS, this makes a lot of sense because our model is still very simple and there is no need to implement a more complicated method for better performance yet. As for our choice of lambdas, we chose a small sample with a big spread as we didn't want to make this unnecessarily computationally heavy, but still achieve a close to optimal fit through the big spread in lambda values. We have come to the conclusion that for this case, this was a sound decision as we observe that some lambdas perform equally well as our OLS model.

When observing Figure 4 we saw that performance changed drastically as lambda assumed different values. Our reasoning for saying that this seems plausible is when the penalty term of the lasso model (lambda) gets too big, it will oversimplify by canceling too many features. This also ties in with the difference between lasso and ridge, whereas lambda in ridge scales our betas changing it slightly, in lasso, lambda will affect our model greatly as observed in our plots. This is also where increasing the values of lambdas we try could be useful for optimizing our model.

As shown in Figure 5, we were not able to achieve the results we wanted when comparing bias and variance. We had a lot of trouble calculating the error and we think the reason our program is not performing correctly boils down to us not being able to calculate this correctly or potentially something wrong in our model. We were surprised at how well k-folds work compared to the bootstrapping method, as one can see in Figure 7, as k-folds is much less computationally heavy while still performing better than bootstrapping. Since our synthetic data did not need to be scaled, we did not have scaling the data in mind when plotting the MSE of our topographical data, which then led to a high MSE. It is of course a more complex data set and therefore, in hindsight it is obvious that the data would have to be scaled.

When it came to using the Ridge method on our topographical data, the MSE dropped a bit lower than is optimal. The MSE and R2 start to deviate after the 4th order polynomial, which is 2 orders higher than where they start to deviate with the synthetic data. This is as expected since the real-world topographical data is a more complex data set, and should therefore require a higher-order polynomial to approximate.

4 Conclusion

We have introduced several linear regression methods, and some key concepts in data science, like bias, variance, and resampling techniques, such as bootstrapping and kfolds cross-validation. We applied these methods and techniques to a set of synthetic data made using the Franke function, a commonly used function for fitting algorithms, and thereafter used them on a real set of topographical data and compared our results. We came to the conclusion that k-folds is a better resampling technique than bootstrapping. OLS, while a simple method, performed quite robustly with both the synthetic and topographical data. Both ridge and lasso, on the other hand, weren't implemented perfectly and their performance reflects that. In conclusion, this analysis of regression methods and basic machine learning has not only deepened our understanding but also bolstered our curiosity about the topics and hand.

References

[1] Lecture notes for FYS-STK3155 by Morten Hjorth-Jensen

[2] "Bias versus Variance." Maveryx Community, 4 Mar. 2021, community.alteryx.com/t5/Data-Science/Bias-Versus-Variance/ba-p/351862.

Appendix

Pencil and paper part for task e)

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

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

where ϵ can be assumed to be 0 and we are then left with

$$= E[(y-E[\tilde{y})^2] + E[(y+E[\tilde{y})^2] + \sigma^2$$

pencil and paper part for task d)

1

$$E(y_i) = E(X_{ij}, \beta) + E(\epsilon)$$

Where $E(\epsilon) = 0$

$$E(y_i) = E(X_{ij}, \beta)$$

$$E(y_i) = X_{i,*}\beta$$

2

$$Var(y_i) = E(y_i^2) - [E(y_i)]^2$$

$$= E[(X_{i*}\beta + \epsilon_i)^2] - (X_{i*}\beta)^2$$

$$= E[(X_{i*})^2 + 2\epsilon_i X_{i*}\beta + \epsilon_i^2] - (X_{i*}\beta)^2$$

$$= (X_{i*}\beta)^2 + 2E(\epsilon_i)X_{i*}\beta + E(\epsilon_i^2) - (X_{i*}\beta)^2$$

$$= E(\epsilon_i^2) = Var(\epsilon_i)$$

$$Var(y_i) = \sigma^2$$

3

$$E(\hat{\beta}) = E[(X^TX)^{-1}X^TY]$$

$$= (X^TX)^{-1}X^TE(y)$$

$$= (X^TX)^{-1}X^TX\beta = \beta$$

4

$$var(\hat{\beta}) = E[\beta - E(\beta)][\beta - E(\beta)^T]$$

$$= E[(X^TX)^{-1}Xy - \beta][(X^TX)^-1Xy\beta]^T$$
$$= (X^TX)^{-1}X^TEyy^TX(X^TX)^{-1} - \beta\beta^T$$

where $y = X\beta + \epsilon$ so $yy^T = X\beta\beta^TX^T + \sigma^2$

$$= (X^TX)^{-1}X^TEX\beta\beta^TX^T + \sigma^2$$
$$\beta\beta^T + \sigma^2(X^TX)^{-1} - \beta\beta^T$$
$$var(\hat{\beta}) = \sigma^2(X^TX)^{-1}$$

Source Code

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import *
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn import linear_model
from imageio import imread


def R2(y_data, y_model):
    return 1 - (np.sum((y_data - y_model) ** 2) / np.sum((y_data - np.mean(y_data


def MSE(y_data, y_model):
    n = np.size(y_model)
    return np.sum((y_data - y_model) ** 2) / n


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


# taken from week 37
def Bias_Variance(z_test, z_pred):
    bias = np.mean((z_test - np.mean(z_pred, keepdims=True)) ** 2)
    variance = np.mean(np.var(z_pred, keepdims=True))

    return bias, variance
```

```
# taken from lecture notes
def create_X(x, y, n):
    if len(x.shape) > 1:
        x = np.ravel(x)
        y = np.ravel(y)

    N = len(x)
    l = int((n + 1) * (n + 2) / 2)  # Number of elements in beta
    X = np.ones((N, l))

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

    return X



def OLS(N, n, data=None, bootstrap_iter=None):

    if data:

        terrain = read_data()
        z_data = terrain[:N, :N]

        # scaling the z-data
        scaler = MinMaxScaler()
        scaler.fit(z_data)
        z_data = scaler.transform(z_data)

        # both x and y are innately scaled
        x = np.sort(np.linspace(0, 1, np.shape(z_data)[0]))
        y = np.sort(np.linspace(0, 1, np.shape(z_data)[1]))
        x_mesh, y_mesh = np.meshgrid(x, y)
        X_data = create_X(x_mesh, y_mesh, n)


        z_data = z_data.reshape(-1)

        X_train, X_test, z_train, z_test = train_test_split(X_data, z_data, test_

    else:
        # Make data set
        x = np.sort(np.random.uniform(0, 1, N))
        y = np.sort(np.random.uniform(0, 1, N))

        X = create_X(x, y, n)

        # we have to explain our choice to not scale the data

        z = FrankeFunction(x, y)

        X_train, X_test, z_train, z_test = train_test_split(X, z, test_size=0.2)


    if bootstrap_iter:

        # taken from week 37
        def bootstrap(data, z):
            datapoints = len(data)
            rand_idx = np.random.randint(0, datapoints, datapoints)
            new_data = data[rand_idx]
            new_z = z[rand_idx]
            return new_data, new_z

        z_pred = np.empty((z_test.shape[0], bootstrap_iter))
        for i in range(bootstrap_iter):
            new_X_train, new_z_train = bootstrap(X_train, z_train)
            OLS_beta = np.linalg.pinv(new_X_train.T @ new_X_train) @ new_X_train.
            z_pred[:, i] = X_test @ OLS_beta

        return z_pred, z_test  # returns a 2D array of predictions for every boot

    else:

        OLS_beta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ z_train
        OLS_ztilde = X_train @ OLS_beta
        OLS_zpredict = X_test @ OLS_beta

        return z_train, OLS_ztilde, z_test, OLS_zpredict, OLS_beta


def OLS_analysis(N, n, data=None):
    MSE_train_list = []
    R2_train_list = []
    MSE_test_list = []
    R2_test_list = []
    OLS_beta_list = []
    for n in range(n):
            if data:
                z_train, OLS_ztilde, z_test, OLS_zpredict, OLS_beta = OLS(N, n, data=
            else:
                z_train, OLS_ztilde, z_test, OLS_zpredict, OLS_beta = OLS(N, n)
            MSE_train_list.append(MSE(z_train, OLS_ztilde))
            R2_train_list.append(R2(z_train, OLS_ztilde))
            MSE_test_list.append(MSE(z_test, OLS_zpredict))
            R2_test_list.append(R2(z_test, OLS_zpredict))
            OLS_beta_list.append(OLS_beta)

        fig, axs = plt.subplots(1, 2, figsize=(10, 4))

        if data:
            fig.suptitle('OLS Performance on Topographic Data', fontsize=16)
        else:
            fig.suptitle('OLS Performance on Synthetic Data', fontsize=16)

        axs[0].plot(np.arange(1, n+2), MSE_train_list, label='MSE')
        axs[0].plot(np.arange(1, n+2), R2_train_list, label='R2')
        axs[0].set_xlabel('nth order polynomial')
        axs[0].set_title('Training Data')
        axs[0].legend()

        axs[1].plot(np.arange(1, n + 2), MSE_test_list, label='MSE')
        axs[1].plot(np.arange(1, n + 2), R2_test_list, label='R2')
        axs[1].set_xlabel('nth order polynomial')
        axs[1].set_title('Test Data')
        axs[1].legend()

        plt.tight_layout()
        plt.show()

        return OLS_zpredict


def Ridge(N, n, data=None):
    if data:
        terrain = read_data()
        z_data = terrain[:N, :N]

        # scaling the z-data
        scaler = MinMaxScaler()
        scaler.fit(z_data)
        z_data = scaler.transform(z_data)

        x = np.linspace(0, 1, np.shape(z_data)[0])
        y = np.linspace(0, 1, np.shape(z_data)[1])
        x_mesh, y_mesh = np.meshgrid(x, y)
        X_data = create_X(x_mesh, y_mesh, n)
        z_data = z_data.flatten()

        X_train, X_test, z_train, z_test = train_test_split(X_data, z_data, test

    else:
        x = np.sort(np.random.uniform(0, 1, N))
        y = np.sort(np.random.uniform(0, 1, N))

        X = create_X(x, y, n)

        z = FrankeFunction(x, y)

        X_train, X_test, z_train, z_test = train_test_split(X, z, test_size=0.2)
    c, v = np.shape(X_train.T @ X_train)
    I = np.eye(c, v)
    nlambdas = 5
    Ridge_MSEPredict = np.zeros(nlambdas)
    Ridge_MSETrain = np.zeros(nlambdas)
    Ridge_R2Predict = np.zeros(nlambdas)
    Ridge_R2train = np.zeros(nlambdas)
    lambdas = [0.001, 0.01, 0.1, 1, 10]

    for i in range(nlambdas):
        lmb = lambdas[i]
        Ridge_beta = np.linalg.pinv(X_train.T @ X_train + lmb * I) @ X_train.T @
        Ridge_ztilde = X_train @ Ridge_beta
        Ridge_zpredict = X_test @ Ridge_beta
        Ridge_MSETrain[i] = MSE(z_train, Ridge_ztilde)
        Ridge_MSEPredict[i] = MSE(z_test, Ridge_zpredict)
        Ridge_R2train[i] = R2(z_train, Ridge_ztilde)
        Ridge_R2Predict[i] = R2(z_test, Ridge_zpredict)

    return Ridge_MSETrain, Ridge_MSEPredict, Ridge_R2train, Ridge_R2Predict


def Ridge_analysis(N, n, data=None):
    Ridge_MSETrain_list = []
    Ridge_MSEPredict_list = []
    Ridge_R2Train_list = []
    Ridge_R2Predict_list = []

    for n in range(n):
        if data:
            MSETrain, MSEPredict, R2Train, R2Predict = Ridge(N, n, 1)
        else:
            MSETrain, MSEPredict, R2Train, R2Predict = Ridge(N, n)
        Ridge_MSETrain_list.append(MSETrain)
        Ridge_MSEPredict_list.append(MSEPredict)
        Ridge_R2Train_list.append(R2Train)
        Ridge_R2Predict_list.append(R2Predict)

    fig, axs = plt.subplots(1, 2, figsize=(10, 4))

    if data:
        fig.suptitle('Ridge Performance on Topographic Data', fontsize=16)
    else:
        fig.suptitle('Ridge Performance on Synthetic Data', fontsize=16)

    lambdas = np.asarray([0.001, 0.01, 0.1, 1, 10])

    for i in range(len(lambdas)):
        axs[0].plot(np.arange(1, n + 2), np.asarray(Ridge_MSETrain_list)[:, i],
        axs[0].plot(np.arange(1, n + 2), np.asarray(Ridge_R2Train_list)[:, i], l
    axs[0].set_xlabel('nth order polynomial')
    axs[0].set_title('Training Data')
    axs[0].legend(loc='right')

    for i in range(len(lambdas)):
        axs[1].plot(np.arange(1, n + 2), np.asarray(Ridge_MSEPredict_list)[:, i]
        axs[1].plot(np.arange(1, n + 2), np.asarray(Ridge_R2Predict_list)[:, i],
    axs[1].set_xlabel('nth order polynomial')
    axs[1].set_title('Test Data')
    axs[1].legend(loc='right')

    plt.tight_layout()
    plt.show()


def Lasso(N, n, data=None):
    if data:
        terrain = read_data()
        z_data = terrain[:N, :N]

        # scaling the z-data
        scaler = MinMaxScaler()
        scaler.fit(z_data)
        z_data = scaler.transform(z_data)

        x = np.linspace(0, 1, np.shape(z_data)[0])
```

```
            y = np.linspace(0, 1, np.shape(z_data)[1])
            x_mesh, y_mesh = np.meshgrid(x, y)
            X_data = create_X(x_mesh, y_mesh, n)
            z_data = z_data.flatten()

            X_train, X_test, z_train, z_test = train_test_split(X_data, z_data, test

        else:
            x = np.sort(np.random.uniform(0, 1, N))
            y = np.sort(np.random.uniform(0, 1, N))

            X = create_X(x, y, n)

            z = FrankeFunction(x, y)

            X_train, X_test, z_train, z_test = train_test_split(X, z, test_size=0.2)

        lambdas = [0.001, 0.01, 0.1, 1, 10]
        nlambdas = len(lambdas)
        Lasso_MSEPredict = np.zeros(nlambdas)
        Lasso_MSEtrain = np.zeros(nlambdas)
        Lasso_R2Predict = np.zeros(nlambdas)
        Lasso_R2train = np.zeros(nlambdas)

        for i in range(nlambdas):
            lmb = lambdas[i]
            RegLasso = linear_model.Lasso(lmb)
            RegLasso.fit(X_train, z_train)
            ytildeLasso = RegLasso.predict(X_train)
            ypredictLasso = RegLasso.predict(X_test)
            Lasso_MSEtrain[i] = MSE(z_train, ytildeLasso)
            Lasso_MSEPredict[i] = MSE(z_test, ypredictLasso)
            Lasso_R2train[i] = R2(z_train, ytildeLasso)
            Lasso_R2Predict[i] = R2(z_test, ypredictLasso)

        return Lasso_MSEtrain, Lasso_MSEPredict, Lasso_R2train, Lasso_R2Predict


    def Lasso_analysis(N, n, data=None):
        Lasso_MSETrain_list = []
        Lasso_MSEPredict_list = []
        Lasso_R2Train_list = []
        Lasso_R2Predict_list = []

        for deg in range(n):
            if data:
                MSETrain, MSEPredict, R2Train, R2Predict = Lasso(N, deg, data=1)

            else:
                MSETrain, MSEPredict, R2Train, R2Predict = Lasso(N, deg)

            Lasso_MSETrain_list.append(MSETrain)
            Lasso_MSEPredict_list.append(MSEPredict)
            Lasso_R2Train_list.append(R2Train)
            Lasso_R2Predict_list.append(R2Predict)


        fig, axs = plt.subplots(1, 2, figsize=(10, 4))

        if data:
            fig.suptitle('Lasso Performance on Topographic Data', fontsize=16)
        else:
            fig.suptitle('Lasso Performance on Synthetic Data', fontsize=16)

        lambdas = np.asarray([0.001, 0.01, 0.1, 1, 10])

        for i in range(len(lambdas)):
            axs[0].plot(np.arange(1, n + 1), np.asarray(Lasso_MSETrain_list)[:, i],
            axs[0].plot(np.arange(1, n + 1), np.asarray(Lasso_R2Train_list)[:, i], l
        axs[0].set_xlabel('nth order polynomial')
        axs[0].set_title('Training Data')
        axs[0].legend(loc='right')

        for i in range(len(lambdas)):
            axs[1].plot(np.arange(1, n + 1), np.asarray(Lasso_MSEPredict_list)[:, i]
            axs[1].plot(np.arange(1, n + 1), np.asarray(Lasso_R2Predict_list)[:, i],
        axs[1].set_xlabel('nth order polynomial')
        axs[1].set_title('Test Data')
        axs[1].legend(loc='right')

        plt.tight_layout()
        plt.show()


def Bias_variance_figure(N, n, data=None):
    error = np.zeros(n)
    bias = np.zeros(n)
    variance = np.zeros(n)

    for degree in range(n):
        if data:
            z_pred, z_test = OLS(N, degree, bootstrap_iter=N, data=1)
        else:
            z_pred, z_test = OLS(N, degree, bootstrap_iter=N)

        error[degree] = np.mean(((z_test.reshape((N//5, 1)) - z_pred)**2).mean(ax
# np.mean((z_test - z_pred.mean(axis=1)) ** 2)
        bias[degree] = np.mean((z_test - np.mean(z_pred, axis=1, keepdims=True))
        variance[degree] = np.mean(np.var(z_pred, axis=1, keepdims=True))

    polydegree = np.arange(1, n + 1)
    plt.plot(polydegree, error, label='Error')
    plt.plot(polydegree, bias, label='bias')
    plt.plot(polydegree, variance, label='Variance')
    plt.ylabel('n-order polynomial')
    if data:
        plt.title('Bias-Variance Tradeoff for Topographic Data')
    else:
        plt.title('Bias-Variance Tradeoff for Synthetic Data')

    plt.legend()
    plt.show()

    return error


def k_folds_cross_val(X, z, k):
    N = len(X[:, 0])
    # shuffling data and z
    rand_idx = np.random.randint(0, N, N)
    data = X[rand_idx]
    z = z[rand_idx]

    # finding size of subgroups
    subset_size = N // k
    new_datasets = []
    new_z = []

    # splitting into subgroups
    for i in range(k):
        new_datasets.append(data[(i * subset_size):(i + 1) * subset_size][:])
        new_z.append(z[(i * subset_size):(i + 1) * subset_size])

    MSE_list = []
    for i in range(k):
        X_train = np.asarray(list(new_datasets[:i][:]) + list(new_datasets[i + 1:
        # X_train = np.reshape(X_train, -1)
        X_train = X_train.reshape(-1, X_train.shape[-1])
```

```
            X_test = new_datasets[i]
            z_train = np.asarray(list(new_z[:i]) + list(new_z[i + 1:]))
            z_train = np.reshape(z_train, -1)
            z_test = new_z[i]

            OLS_beta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ z_train
            OLS_zpredict = X_test @ OLS_beta

            MSE_list.append(MSE(z_test, OLS_zpredict))

        # print(MSE_list)
        Final_MSE = np.mean(MSE_list)
        # print(Final_MSE)

        return Final_MSE


    def k_folds_analysis(N, n, mse=None, skl=None, data=None):

        if data:
            terrain = read_data()
            z_data = terrain[:N, :N]

            # scaling the z-data
            scaler = MinMaxScaler()
            scaler.fit(z_data)
            z_data = scaler.transform(z_data)

            # both x and y are innately scaled
            x = np.sort(np.linspace(0, 1, np.shape(z_data)[0]))
            y = np.sort(np.linspace(0, 1, np.shape(z_data)[1]))
            x_mesh, y_mesh = np.meshgrid(x, y)
            X = create_X(x_mesh, y_mesh, n)

            z = z_data.reshape(-1)
        else:
            x = np.sort(np.random.uniform(0, 1, N))
            y = np.sort(np.random.uniform(0, 1, N))

            X = create_X(x, y, n)
            z = FrankeFunction(x, y)

        # finding the different MSE's of k using own code and scikit's function
        OLS = LinearRegression()
        SK_MSE_vals = []
        k_fold_MSE_list = []
    k_values = np.arange(5, 11)
    for k in k_values:
        k_fold_MSE_list.append(k_folds_cross_val(X, z, k))
        if skl:
            kf = KFold(n_splits=k)
            SK_MSE_vals.append(np.mean(cross_val_score(OLS, X=X, y=z, cv=kf)))

    plt.plot(k_values, k_fold_MSE_list, label='K-fold MSE')
    plt.xlabel('k-folds')
    if data:
        plt.title('K-folds Cross-Val MSE for Topographic Data')
    else:
        plt.title('K-folds Cross-Val MSE for Synthetic Data')

    if skl:
        plt.plot(k_values, SK_MSE_vals, label='cross_val_score')

    if mse.any():
        plt.plot(k_values, mse[4:], label='Bootstrapping MSE')

    plt.legend()
    plt.show()


def read_data(plot=None):

    terrain2 = imread('SRTM_data_Norway_2.tif')

    if plot == '2D':
        plt.figure()
        plt.title('Terrain over Norway 1')
        plt.imshow(terrain2, cmap='gray')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.show()

    elif plot == '3D':
        x = np.linspace(0, 1, terrain2.shape[1])
        y = np.linspace(0, 1, terrain2.shape[0])
        X, Y = np.meshgrid(x, y)
        Z = terrain2

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(X, Y, Z, cmap='viridis')
        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_zlabel('Z Label')
        ax.set_title('3D Surface Plot')
        plt.show()

    return np.asarray(terrain2)



if __name__ == "__main__":
    np.random.seed(2022)

    #read_data(plot='3D')


    #OLS_analysis(N=500, n=15, data=1)
    #Ridge_analysis(N=500, n=15, data=1)
    #Lasso_analysis(N=500, n=15, data=1)
    #mse = Bias_variance_figure(N=500, n=10)
    #k_folds_analysis(N=500, n=10, mse=mse)
    #read_data(plot=1)
```