University of Oslo

FYS-STK4155

Applied Data Analysis and Machine Learning

Project 1: Regression Analysis and Resampling Methods

Author:
Håkon Fossheim
October 24, 2021

In this project, we have assessed the performance of three different linear regression methods, namely Ordinary Least Squares (OLS), Ridge- and Lasso regression. These models were tasked with fitting a bivariate polynomial to three dimensional data points origininating from either the Franke function or real terrain.

The Bootstrap and Cross-Validation resampling techniques were used to aid in our model assessment, the results of which were used to select the optimal regression method and its parameters.

For the generated Franke function data, the optimal OLS and Ridge models were found to perform very similarly in the parameter space of our search. The optimal OLS model was found to be at $p = 8$, while Ridge gave very similar results in a range of values. Both of these methods outperformed Lasso regression for our chosen level of noise.

For the terrain data, we found that Ridge regression using a polynomial of degree $p \geq 35$ and $\lambda \leq -10$ typically slightly outperforms the optimal OLS model given by $p = 18$, but at approximately four times the computational cost. For this reason, this OLS model was chosen as the most suited to fitting the terrain data.

I. Introduction

Our goal in this report is to apply three different linear regression models, namely OLS, Ridge and Lasso, to two different data sets and assess their performance. The first data set is generated by adding noise to the Franke function. The second uses real terrain data[1] in Norway. To perform this assessment, we calculate their bias, variance, MSPE and R2test score. These measures will also allow us to gain deeper insights of the regression three regression models.

Due to the noise in our data, these estimators will have some variance. We therefore use the bootstrap resampling technique to get more precise measurements as well as cross validation, which is a resampling technique better suited for small data sets. .

[1] terrain 1, found at [3]

II. Theory

A. Linear Regression models

We are interested in approximating the relationship between a set of explanatory variables[2] $[x_0, x_1, ..., x_{n-1}$ and an unknown target function f. We are in possession of a noisy dataset given by $y(x_i) = f(x_i)+_i$ where ε is the noise in our data.

Linear regression tries to model the target function as a linear combination of the explanatory variables raised to various powers. For a given data point y_i it creates a model from the corresponding x_i in the following manner

$$\tilde{y}(x_i) = \beta_0 + 0x_i^1\beta_1 + x_i^2\beta_2 + ... + x_i^n\beta_n \tag{1}$$

Which leads to the following set of equations:

$$\begin{bmatrix} \tilde{y}_0 \\ \tilde{y}_1 \\ \vdots \\ \tilde{y}_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} \tag{2}$$

Which we can write in matrix form at

$$\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta} \tag{3}$$

Where $\mathbf{X}$ is called our design matrix.

To find the optimal values of regression coefficients, $\boldsymbol{\beta}$. To do this, a cost function $C(\mathbf{y}, \tilde{\mathbf{y}})$ is used. This cost function measures in one way or another how far off a hypothetical model $\tilde{\mathbf{y}}(\boldsymbol{\beta})$ is from the target function, and so the linear regression picks the model whos regression coefficients minimize this costfunction, i.e.:

$$\tilde{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}} \tag{4}$$

Where

$$\hat{\boldsymbol{\beta}} = \text{argmin}_\beta C\left(\mathbf{y}, \tilde{\mathbf{y}}(\boldsymbol{\beta})\right) \tag{5}$$

What defines a specific linear regression model is the way its cost function is implemented.

Without knowing better, one might suggest a cost function which measures the absolute distance to from the model to the target function, i.e. the mean absolute error(MSA), given by

$$\text{MSA}(\mathbf{X}, \boldsymbol{\beta}) \equiv \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \tilde{y}_i| = \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \mathbf{X}_{i,*}\boldsymbol{\beta}| \tag{6}$$

[2] Also called independent variables

However, this function suffers from the drawback that it is not differentiable with respect to $\boldsymbol{\beta}$ and so we cannot find a closed form expression for $\hat{\boldsymbol{\beta}}$. A better choice is the mean squared error (MSE), defined as

OLS :

$$C(\boldsymbol{\beta}) = \text{MSE} \equiv \frac{1}{n}\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = \frac{1}{n}\sum_{i=0}^{n-1} (y_i - \mathbf{X}_{i,*}\boldsymbol{\beta})^2 \tag{7}$$

This is the cost function used by the OLS. Since we are taking the measuring the square distance, it will see models with outliers as particularly unfavorable, which is good because these outliers are usually spikes in the noise of our data.

Furthermore, unlike the MSA, the MSE is differentiable w.r.t. $\boldsymbol{\beta}$, meaning we can get a closed form expression for the optimal regression parameters, $\hat{\boldsymbol{\beta}}$.

Differentiating Eq.(7) with respect to $\boldsymbol{\beta}$ and setting the expression equal to zero gives the optimal regression parameters:

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \tag{8}$$

Another useful measure to assess a model's performance is the R^2 score, also called the coefficient of determination. It is defined as

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} \tag{9}$$

Where

$$\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i \tag{10}$$

We see from (9) that a perfect fit of our model to the data points yields an R^2 score of 1. An R^2 score of 0 indicates that our model performed no better than the average value of the data points, and a negative score means it did worse and simply taking the average of the data is a better model.

1. Ridge regression Ridge regression is defined by the following cost function:

$$\begin{aligned} C(\boldsymbol{\beta}) &= \frac{1}{n}\sum_{i=0}^{n-1} (y_i - \mathbf{X}_{i,*}\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p-1}\beta_j^2 \\ &= \frac{1}{n}\left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right)^T\left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \\ &= \frac{1}{n}\left(\mathbf{y}^T - \boldsymbol{\beta}^T\mathbf{X}^T\right)\left(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \end{aligned} \tag{11}$$

The optimal β-parameters are given by $\text{argmin}_{\boldsymbol{\beta}} C(\boldsymbol{\beta})$, i.e. solving for $\partial C(\boldsymbol{\beta})/\partial\boldsymbol{\beta} = 0$

Since $\mathbf{y}$ is not a function of $\boldsymbol{\beta}$, we get

$$-\frac{\partial}{\partial\boldsymbol{\beta}}\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} - \frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} + \frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + n\lambda\frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\boldsymbol{\beta} = 0 \tag{12}$$

By using the following identity[2],

$$\frac{\partial\mathbf{a}^T\mathbf{A}\mathbf{a}}{\partial\mathbf{a}} = \left(\mathbf{A} + \mathbf{A}^T\right)\mathbf{a} \tag{13}$$

the third term becomes

$$\frac{\partial}{\partial\boldsymbol{\beta}}\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} \tag{14}$$

Furthermore, we know that[2]

$$\frac{\partial\mathbf{b}^T\mathbf{a}}{\partial\mathbf{a}} = \mathbf{b} \tag{15}$$

$$\frac{\partial}{\partial\boldsymbol{\beta}}\mathbf{y}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y} \tag{16}$$

Since every term in (11) is a scalar, we have

$$\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}\right) = \left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}\right)^T = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} \tag{17}$$

And so we get

$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + 2n\lambda\boldsymbol{\beta} = 0 \tag{18}$$

$$\left(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I}\right)\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y} \tag{19}$$

If we 'absorb' n into λ, we get

$$\hat{\boldsymbol{\beta}}_{Ridge} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y} \tag{20}$$

B. Lasso regression

Lasso regression, is defined by the following cost function:

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - \mathbf{X}_{i,*}\boldsymbol{\beta}\right)^2 + \lambda\sum_{j=1}^{p-1}|\beta_j| \tag{21}$$

Lassso is an acronym for Least Absolute Shrinkage and Selection Operator and it performs best for data with many features. This is because, like Ridge, it is designed to prevent overfitting by supressing the regression coefficients $\hat{\beta}$. However, unlike Ridge, one cannot find a closed form solution for the optimal regression coefficients. For this, one needs to use computational methods.

It also has the ability to force the β values to zero, which Ridge regression cannot do. We saw this happen for $\log\lambda > -1.21$ with $n = 20$ for the Franke function, where Lasso regression returns only a flat surface at 0. Varying n also varies at which λ this happens.

Lasso regression is defined by the following cost function:

C. Confidence interval of $\hat{\boldsymbol{\beta}}$

Assuming the estimators $\hat{\beta}_i$ are normally distributed with mean $\mathbb{E}[\hat{\beta}_i]$ and variance $\sigma^2_{\hat{\beta}_i}$, i.e. $\hat{\beta}_i \sim \mathcal{N}\left(\mathbb{E}[\beta_i], \sigma^2_{\hat{\beta}_i}\right)$, then we can say that there is a $(1-\alpha)100\%$ [likelihood?probability?] that $E[\hat{\beta}_i]$ lies in the interval $\left[\hat{\beta}_i - Z\sigma_{\beta_i}, \hat{\beta}_i + Z\sigma_{\beta_i}\right]$. In other words

$$P(\hat{\beta}_i - Z\sigma_{\beta_i} \leq \mathbb{E}[\hat{\beta}_i] \leq \hat{\beta}_i + Z\sigma_{\beta_i}) = 1 - \alpha \tag{22}$$

Where Z is how many standard deviations you have to go to the right or left of the mean of the standard normal to get an area of $(1-\alpha/2)$. It is therefore given by

$$Z = \phi^{-1}\left(1 - \frac{\alpha}{2}\right) \tag{23}$$

Where φ^{-1} is the inverse cumulative distribution function of the standard normal distribution. For a 95% CI, i.e. $\alpha = 0.05$, we get $Z \approx 1.96$.

It can be showed[1] that the variance of the estimator $\hat{\boldsymbol{\beta}}$ is given by

$$\text{var}\left(\hat{\boldsymbol{\beta}}\right) = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\sigma^2 \tag{24}$$

Where $\mathbf{X}$ is the design matrix and $\sigma^2 = \text{var}(\epsilon_i)$. Furthermore, $\text{var}\left(\hat{\boldsymbol{\beta}}\right)$ is defined as

$$\text{var}\left(\hat{\boldsymbol{\beta}}\right) = \begin{bmatrix} \text{var}(\beta_0) & \text{cov}(\beta_0,\beta_1) & \dots & \text{cov}(\beta_0,\beta_{p-1}) \\ \text{cov}(\beta_1,\beta_0) & \text{var}(\beta_1) & \dots & \text{cov}(\beta_1,\beta_{p-1}) \\ \vdots & \vdots & \ddots & \vdots \\ \text{cov}(\beta_{p-1},\beta_0) & \text{cov}(\beta_{p-1},\beta_1) & \dots & \text{var}(\beta_{p-1},\beta_{p-1}) \end{bmatrix} \tag{25}$$

And so $\text{var}\left(\hat{\beta}_i\right) = \sigma^2_{\hat{\beta}_i}$ is given by the i-th diagonal entry of $\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\sigma^2$.

D. Bias-variance trade-off

Assumption: We are calculating the MSE on the test set with a model trained on the training data, so that $\tilde{\mathbf{y}}$ and $\boldsymbol{\epsilon}$ are independent.

Our data is distributed according to

$$y_i = f(x_i) + \varepsilon_i \tag{26}$$

With

$$\varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2) \tag{27}$$

Where y_i, x_i and ϵ_i are instances of the stochastic variables $\mathbf{y}$, $\mathbf{x}$ and $\boldsymbol{\epsilon}$. [3]

$$C(\mathbf{X}, \boldsymbol{\beta}) = \mathbf{MSPE} \equiv \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{28}$$

Since we are in fact dealing with continuous variables, the **MSPE** will be an approximation of $\mathbb{E}\big[\,(\mathbf{y}-\tilde{\mathbf{y}})^2\,\big]$. This approximation should improve as we increase n.

$$\mathbf{MSPE} \equiv \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \approx \mathbb{E}\left[\big(\mathbf{y}-\tilde{\mathbf{y}}\big)^2\right] \tag{29}$$

$$= \mathbb{E}\left[\big(f(\mathbf{x}) + \boldsymbol{\epsilon} - \tilde{\mathbf{y}}\big)^2\right] \tag{30}$$

Adding and subtracting $\mathbb{E}[\tilde{\mathbf{y}}]$ yields

$$\begin{aligned}
\mathbb{E}\left[\big(\mathbf{y}-\tilde{\mathbf{y}}\big)^2\right] &= \mathbb{E}\left[\Big(\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big) - \big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big) + \boldsymbol{\epsilon}\Big)^2\right] \\
&= \mathbb{E}\Big[\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)^2\Big] - \mathbb{E}\Big[\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)\Big] \\
&+ \mathbb{E}\Big[\boldsymbol{\epsilon}\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)\Big] - \mathbb{E}\Big[\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)\big(f(\mathbf{x}) - \mathbb{E}\left[\tilde{\mathbf{y}}\right]\big)\Big] \\
&+ \mathbb{E}\Big[\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)^2\Big] - \mathbb{E}\Big[\boldsymbol{\epsilon}\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)\Big] \\
&+ \mathbb{E}\Big[\boldsymbol{\epsilon}\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)\Big] - \mathbb{E}\Big[\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)^2\Big] + \mathbb{E}\big[\boldsymbol{\epsilon}^2\big]
\end{aligned} \tag{31}$$

If two stochastic variables $\mathbf{x}$ and $\mathbf{y}$ are independent, then the expectation value of their product is equal to the product of their expectation values, i.e.

$$\mathbb{E}[\mathbf{xy}] = \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{y}] \tag{32}$$

In general, $\tilde{\mathbf{y}}$ and $\boldsymbol{\epsilon}$ won't be independent and so (32) would not hold for these variables. This is because $\tilde{\mathbf{y}}$ depends on $\mathbf{y}$ which again depends on the stochastic noise, $\boldsymbol{\epsilon}$.

However, if we train the model $\tilde{\mathbf{y}}$ on a training set and then calculate the cost function $C(\mathbf{X}, \boldsymbol{\beta})$ on the test set only, then the noise in this set will not have affected our model, in which case we have

$$\mathbb{E}[\boldsymbol{\epsilon}\tilde{\mathbf{y}}] = \underbrace{\mathbb{E}[\boldsymbol{\epsilon}]}_{=0}\mathbb{E}[\tilde{\mathbf{y}}] = 0 \tag{33}$$

Since $f(\mathbf{x})$ and ϵ also are independent, the third, sixth and seventh terms in (31) are zero.

[Furthermore, the underlying function from which our data is generated, viz. $f(\mathbf{x})$ is deterministic and so]

Expanding the third term yields

$$\begin{aligned}
&\mathbb{E}\Big[\,\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)\,\Big] = \mathbb{E}\big[f(\mathbf{x})\tilde{\mathbf{y}}\big] - \mathbb{E}\big[f(\mathbf{x})\big]\mathbb{E}\big[\tilde{\mathbf{y}}\big] \\
&- \mathbb{E}\big[\tilde{\mathbf{y}}\big]\mathbb{E}\big[\tilde{\mathbf{y}}\big] + \mathbb{E}\big[\tilde{\mathbf{y}}\big]^2 = \mathbb{E}\big[f(\mathbf{x})\tilde{\mathbf{y}}\big] - \mathbb{E}\big[f(\mathbf{x})\big]\mathbb{E}\big[\tilde{\mathbf{y}}\big] \\
&= \mathrm{cov}\left(f(\mathbf{x}), \tilde{\mathbf{y}}\right)
\end{aligned} \tag{34}$$

However, since $\mathbf{x}$ is not a stochastic variable[4], we can set it outside the expectation value operator:

$$\begin{aligned}
\mathrm{cov}\left(f(\mathbf{x}), \tilde{\mathbf{y}}\right) &= \mathbb{E}\big[f(\mathbf{x})\tilde{\mathbf{y}}\big] - \mathbb{E}\big[f(\mathbf{x})\big]\mathbb{E}\big[\tilde{\mathbf{y}}\big] \\
&= f(\mathbf{x})\mathbb{E}\left[\tilde{\mathbf{y}}\right] - f(\mathbf{x})\mathbb{E}\left[\tilde{\mathbf{y}}\right] = 0
\end{aligned} \tag{35}$$

in the expression above, giving . The final expression for the MSPE then becomes

$$\begin{aligned}
\mathbf{MSPE} \approx \mathbb{E}\left[\big(\mathbf{y}-\tilde{\mathbf{y}}\big)^2\right] &= \underbrace{\mathbb{E}\left[\big(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}]\big)^2\right]}_{\mathbf{bias}^2} \\
&+ \underbrace{\mathbb{E}\left[\big(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}]\big)^2\right]}_{\mathbf{variance}} + \underbrace{\mathbb{E}[\boldsymbol{\epsilon}^2]}_{\mathbf{irr.err.}}
\end{aligned} \tag{36}$$

The first term is the square of the bias, which is a measure of how well the interpolated model fits the target function. Low complexity models often have high bias, as they lack the flexibility to accurately capture the features of the target function. We then say that the model suffers from *underfitting*. Note that the target function $f(\mathbf{x})$ is usually not available, since this is what we are trying to interpolate. In its place, we may use the response variable $\mathbf{y}$ to get an estimate of the bias.

The second term measures the variance of the model, i.e. how much it varies around its mean. very high complexity models can fit the training data much better, and thus give low bias. However, it will also fit the stochastic noise in the training data, and thus pick up features that aren't really there. When the model is then tested on test data, it performs poorly because it failed to capture the relationship between $\mathbf{f}$ and $\mathbf{x}$ during training. We say that the model suffers from *overfitting*. In this case, the variance will typically be high, since the model needs to fluctuate a lot to fit the noisy training data.

The last term is the irreducible error, and is the variance of the stochastic noise in the data. No matter how good of a model we generate, it can never predict the noise of yet-unseen data samples, and so this forms a lower bound on the prediction error.

[3] x is not a stochastic variable? Too late to fix this mistake

[4] i.e. x is deterministic

E. Resampling methods

1. Bootstrap Each time a specific model is fitted to the data, we evaluate its performance through various estimators, $\hat{\boldsymbol{\theta}}$. However, the data will likely contain some noise, which means these estimate will vary around their true values each time the model is fitted with data containing new noise.

The central limit theorem tells us that in the case that the noise is i.i.d., we can improve the precision of of an estimate by replacing the estimator $\hat{\theta}$ with

$$\tilde{\hat{\theta}} = \frac{\hat{\theta}_0 + \hat{\theta}_1 + ... + \hat{\theta}_B}{B} \tag{37}$$

Where $\tilde{\hat{\theta}}$ is the mean of $\hat{\theta}$ calculated on B datasamples containing different noise.

The variance of this new estimator is then given by[1]

$$\text{Var}\left[\tilde{\hat{\theta}}\right] = \frac{\text{Var}\left[\hat{\theta}\right]}{B} \tag{38}$$

One time consuming and expensive way to do calculate our new estimates would be to repeat the experiment B times with completely new data sets each time.

A much more practical way of generating "different" data set is to randomly sample with replacement N data points from the original data set, from which we can calculate $\hat{\theta}_i$.

2. K-fold Cross-Validation In order to assess the performance of our model on unseen data, one splits the original data set into a training set and a test set. The training data is then used to produce the model and its performance is gauged using the test data. This way we can assess how well it learned the features of the data rather than how good it is at fitting noisy data, i.e. how "flexible" it is.

For small datas samples, there is a higher probability that the test set given to the model is not representative of our full data set. E.g. the model might be assigned a test set which is a lot less or much more noisy than the training data, resulting in us either over or underestimating the model performance.

To remedy this, Cross Validation we splits the original data set into K samples, where 1 of these is to be used for testing and the remaining $K-1$ samples are to be used as training data. The relevant estimators are then calculated and the result returned.

This is done K times, but each time we pick one of the previously unchosen K samples as our test set. In this way, all the original data will have been included in both the test and training sets. The expected value and variance of the estimators are calculated for each fold, and the mean of these quantities is returned. This averages out any anomalous estimates that any one of the splits might produce due to being assigned an unrepresentative test set.

F. Feauture Scaling

Some machine learning algorithms may have reduced performance unless the features are distributed according to the standard normal distribution. An example of this is the K-nearest neighbours algorithm, which counts the K nearest neighbours in the feature space and performs classification according to which of the categories is most represented. To do this requires it to calculate the euclidean norm to its neighbours, and Unless the features are scaled, this yields suboptimal results.

Furthermore, using gradient descent to find the minimum of the cost function[5], it is important to scale the data. This is because if one of the features has a large magnitude compared with the others, the surface (in the case of two features) will be elongated along that feaure's axis, making gradient descent zigzag its way to the minimum rather than taking a straighter path. This decreases the performance of gradient descent.

In our case the features have the same mean and standard deviation and so scaling them is somewhat redundant, unless performing Lasso regression. We still scale our data[6] when using OLS and Ridge regression, since it is good machine learning practice.

G. The Franke function

The Franke's bivariate test function is commonly used as a test function in interpolation problems, and is given by

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left[-\frac{1}{4}(9x-2)^2 - \frac{1}{4}(9y-2)^2\right] \\ &+ \frac{3}{4}\exp\left[-\frac{1}{49}(9x+1)^2 - \frac{1}{10}(9y+1)\right] \\ &+ \frac{1}{2}\exp\left[-\frac{1}{4}(9x-7)^2 - \frac{1}{4}(9y-3)^2\right] \\ &- \frac{1}{5}\exp\left[-(9x-4)^2 - (9y-7)^2\right] \end{aligned} \tag{39}$$

How to set up the this interpolation problem given this two dimensional target function is described in the appendix.

---

[5] As is done by when performing Lasso regression in our program

[6] by subtracting the mean of each column in the designmatrix from every element in that column.

III. Methods

A. Program description

Our current program stems from an initial desire to be able to control a messy code and produce the desired plots with more ease. To do this, we represented all the variables needed in the form of bools and ints in the main() function, which we will call our control panel (can be found in the appendix). Doing so makes it easy to adjust them and re-run main.py in order to produce a different plot. It also makes it necessary to "reformat" them before usage in the rest of the program for the sake of readability and sanity. For example, the regression method is set as an int in the control panel;

```
regInt     = 0 #0=OLS,          1=ridge,      2=lasso
```

Which gets used by all the resampling methods in resampling_methods.py to determine the regression method to use

```
if(regMeth=='OLS'):
        beta_hat=OLS(X_train,z_train,skOLS)
if(regMeth=='ridge'):
    beta_hat=ridge(X_train,z_train,lmd)
if(regMeth=='lasso'):
    beta_hat=lasso(X_train,z_train,lmd)
```

It would be much more tedious and error prone to remember which ints correspond to which methods etc, in addition to the code being unreadable and containing a lot of functions with tons of arguments.

And so, blow the control panel, we have included the function reformatVariables(original variables), which lies in the scope of the control panel variables, and automatically takes them to be reformatted as lists and dictionaries etc. in reformat$_v$*ariables.py*.

It also generates the data to be interpolated based on the terrainBool and the n, the number of points along the x- and y axis, which are specified in the control panel.

Each resamplingmethod has its own list, which contains the name of the method as its first element along with the variables it needs to perform its task.

This is seen in reformat$_v$*ariables.py* :

Which takes resampInt from the control panel,

```
resampInt = 1
#0=no_resamp., 1=bootstrap, 2=crossval
```

as an argument and determines which list to pass as an argument to perform_regression() based on the specified integer.

This list then determines which resampling method to use, as well as providing the necessary variables to do so.

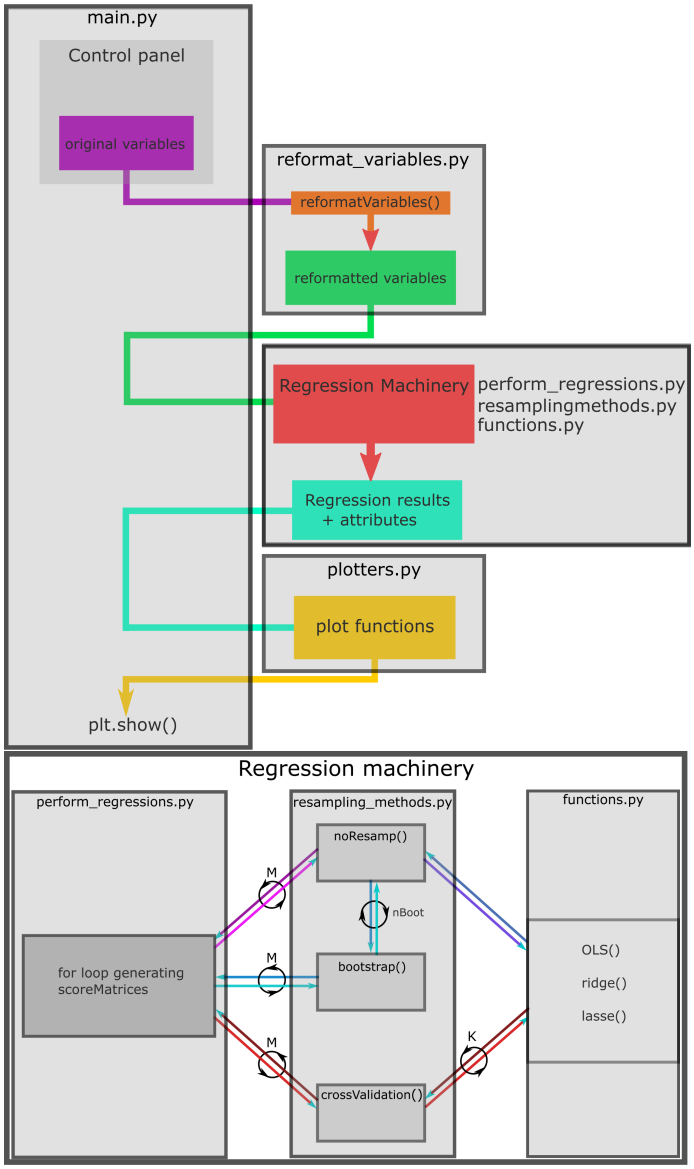


FIG. 1. Here we see the program flow of our code as well as the "regression machinery", which is tasked with returning the requested regression results. The outgoing values are represented by a lighter color than the returning results. The circles specify how many times the the function call occur after we run main.py. Here M is the total number of elements of the matrix or matrices contained in the scoreMatrices dict. K is the number folds specified for Cross-Validaiton and nBoot the number bootstrap samples to be used. After each time call to a regression function, the result is used to again call on the getScores() function in functions.py in order to calculate the errors/scores of that given regression iteration.

We can also specify which ranges of the hyperparameters λ, p and σ to iterate over. This determines what plot gets produced. To store these results, we initialize an "empty" dict containing only the scores[7] specified

[7] We use the term score for any calculation done on the regres-

by the user by passing in the integers corresponding to the scores seen in the dummyList below. This slightly reduces computational cost compared to calculating all scores each time.

```
def getScores(*args):
    dummyList = ['bias','variance','MSEtest',...,]
    emptyScoreScalars = {}
    for i in range(0,len(args)):
        scoreName = dummyList[args[i]]
        emptyScoreScalars[scoreName]=0
    print("The following scores are being
    calculated:\n", emptyScoreScalars,'\n' )
    return emptyScoreScalars
```

As seen, the values belonging to each sting is initially set to zero, but will be filled by the requested score calculation in getScores() in functions.py before storing this result as an element in its corresponding score matrix in the dictionary scoreMatrices. This dictionary is created in perform_regression.py and will create entries based on the titles of emptyScoreScalars.

```
    scoreMatrices = {}
    for scoreName in emptyScoreScalars:
        scoreMatrices[scoreName] = np.zeros((nLambdas,nOrders,nSigmas))
```

The result of regression using different lambdas are stored along the "y-axis" of this matrix, while the result from different orders and sigmas are stored along the x and z axis respectively. We call store these three lists as elements of the list "hyperPars", which again is used as an entry in the dictionary calcAtts along with other things needed for plotting:

```
calcAtts = {'hyperPars': hyperPars,\
'plotTitleAtts':[regMeth,resampMeth,nBoot,K,n]}
```

As the for-loop over all hyperparameters is finished, and all 3D matrices are filled, the calculation results in the form of the dict scoreMatrices along with the calcAtts are returned to main() and lastly sent to the plotting functions in plotters.py, which determines the dimensions of our score-result matrices and generates figures with the relevant variables included in the title, so as to be reproducible.

Note that we also included the script run_exercises.py which allows one to more easily reproduce our results. It is simply a series of if-statements with the variables set according to the exercises and "task", with the reformatter, regression machinery and plotter in their scope.

---

sion data. Due to the use of dicts, adding further scores to our program shouldn't be an issue.

IV. Results

A. Exercise 1: OLS on the Franke function

Throughout the project, our bias calculation were clearly off. They refused to go to zero, which in retrospect might make sense, since we were calculating it on the test data motivated by our "results" in the bias-variance section. Using the following expression

```
scoreVal =  (np.mean((z_tilde-z_train)**2))
```

where scoreVal = bias^2 yields much better results, which also makes sense. As the model fits the training data better and better, this expression should go to zero. There is however no reason for why it should decrease with complexity if evaluated on the test data, since it has not seen the noise in that data, and won't do any better job of predicting the true test data plus noise than the true test data, which it does a horrible job of when overfitted.

We choose our to divide the x- and y-axis into $n = 40$, yielding 1600 gridpoints. A larger n could be chosen to yield better fits, at the cost of computational time. A fixed σ will be used throughout evaluating regression and resampling methods on the Franke function. We will be evaluating these methods by use of two dimensional polynomials of degrees of orders $p = 0$ up to $p = 20$ to find which one is best suited to fit the Franke function. To find an appropriate noise level, we can study how the fit varies with σ for a polynomial fit of e.g. degree $p = 10$, as seen in Fig.2.

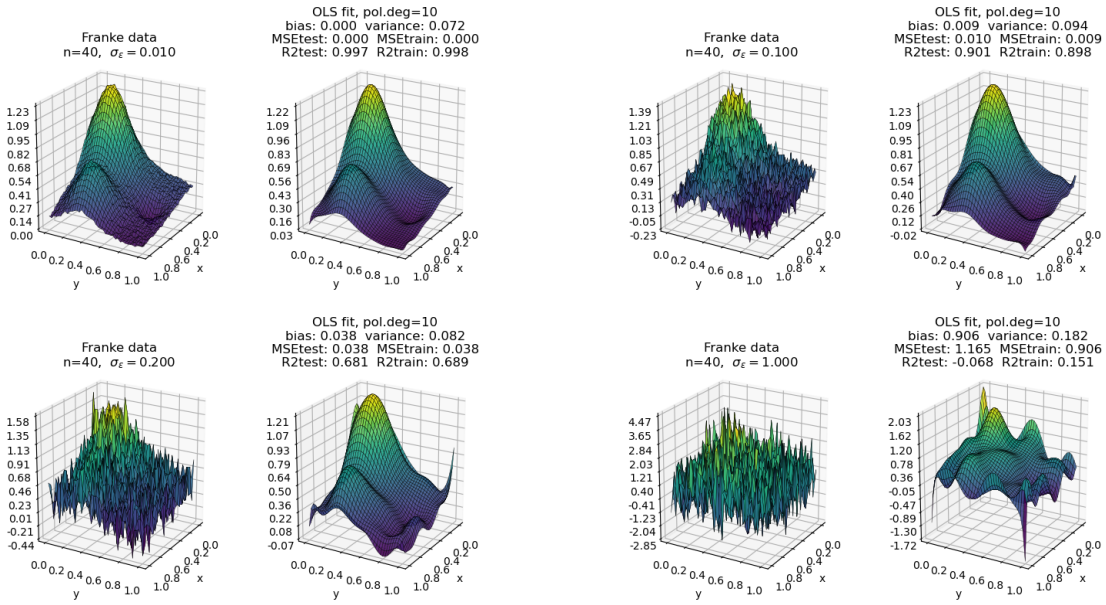


FIG. 2.

We see that for $\sigma = 0.001$, the noise amplitude is so low that it yields an almost perfect fit to the target function. This isn't of interest to us, since we wish to stress test our regression methods to see how they might break down. At around σ_ε, the model starts to overfit, as indicated by a higher bias and lower R2test score. For $\sigma = 1$ and above, the fit barely resembles the target function. The negative R2test-score indicates that even just using the mean of the noisy data is a better model. We therefore settle $\sigma = 0.2$ for our Franke function interpolation.

We start by evaluating the 95% confidence intervals for $\hat{\boldsymbol{\beta}}$ using OLS for $p = 1$ to $p = 5$. The fitted function along with the regression coefficients β_i, are shown in Fig.3 and 4.

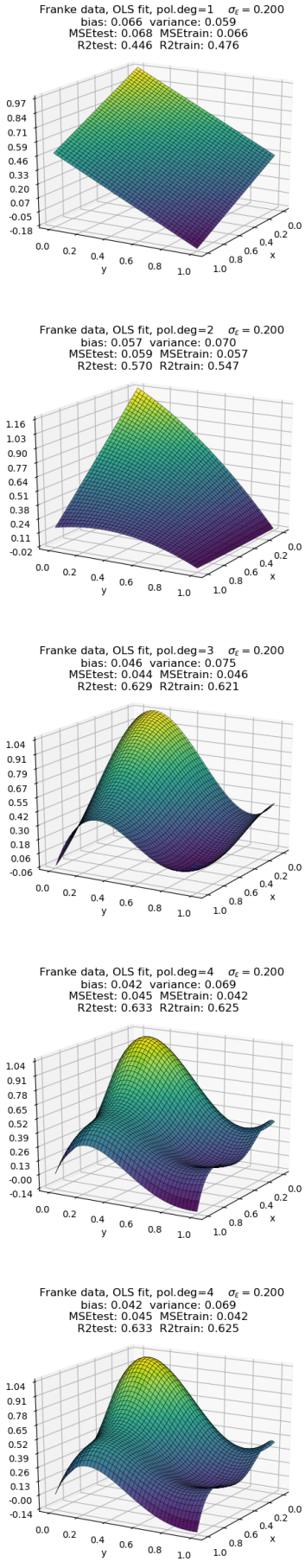


FIG. 3.

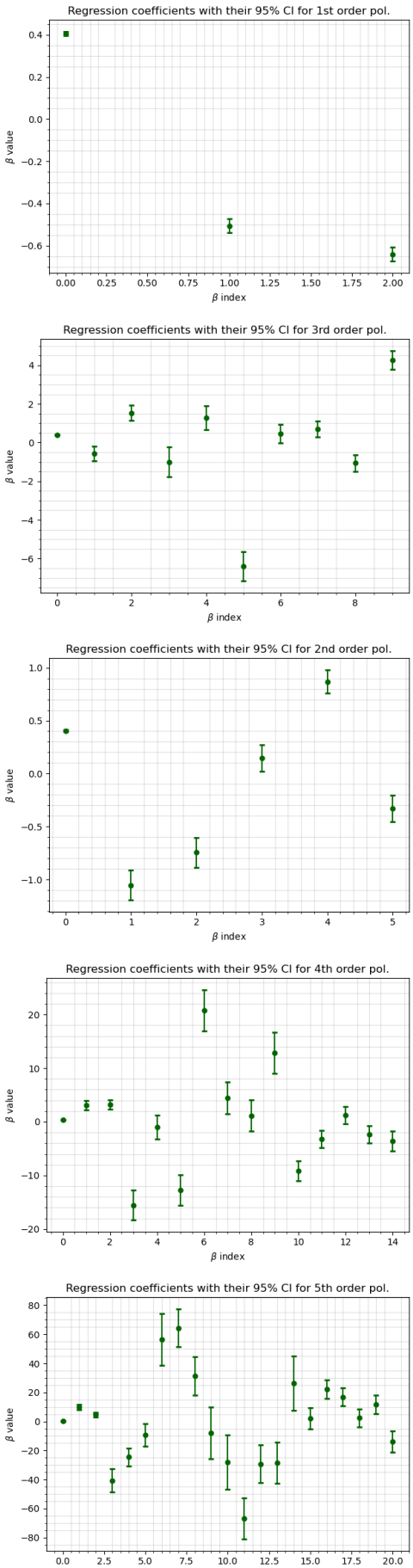


FIG. 4.

We start by evaluating the 95% confidence intervals for $\hat{\beta}$ using OLS for $p = 1$ to $p = 5$. The fitted function along with the regression coefficients β_i,[8] are shown in Fig.3 and 4. As can be seen, the first order polynomial is not complex enough to model the features in the data. It is underfitted. As we increase the complexity, we see the bias go down and that it resembles the Franke function more and more. However, as indicated by the R2test score, $p = 5$ is still not an amazing fit. We can study plot the MSEtest and MSEtrain errors against p to determine which OLS model yields the best fit. As seen in the first panel in Fig.5, for very low amplitude noise, these two values are almost the same, regardless of complexity. This is because even though the interpolating polynomial starts to fit the errors/noise, the amplitude of this noise is sufficiently small that the polynomial closely resembles the original function, and is therefore able to extrapolate well to training data. Thus, the MSPE(MSEtest) goes to zero at high complexity, while at low complexity, the model is not sufficiently complex to give us a good fit of the original function, i.e. it suffers from high bias. As the amplitude of the noise increases, MSEtrain keeps its shape, viz. it decreases to 0 with complexity, while MSEtest diverges from MSEtrain and starts increasing. This is because the model is sufficiently complex to start fitting the noise. Since the amplitude of this noise is not negligible, the fitted function will differ significantly from the target function. So even though it fits the training data well, it extrapolates poorly to test data. It is then in the high variance region.

Studying panel 3, which corresponds to our noise amplitude, we see that the minimum of MSEtest is given by $p = 8$. However, we will reproduce this plot with the use of resampling later in order to say with more certainty that this is indeed the optimal OLS model.

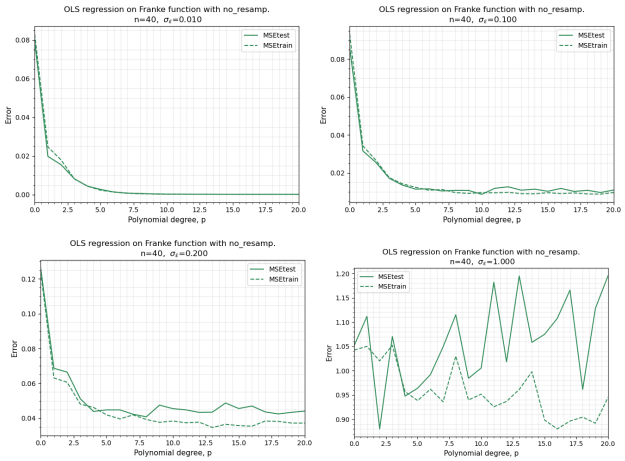


FIG. 5. Here we see the mean squared prediction error for both the training- and test data plotted against the complexity of our OLS model, i.e. the degree of the interpolating polynomial.

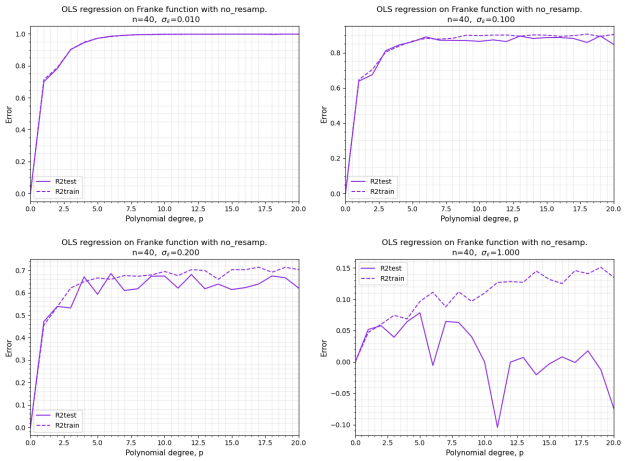


FIG. 6. R2test vs R2train scores plotted against model complexity for various noise amplitudes.

[8] of which there are $(p+2)(p+1)/2$ as discussed in the appendix

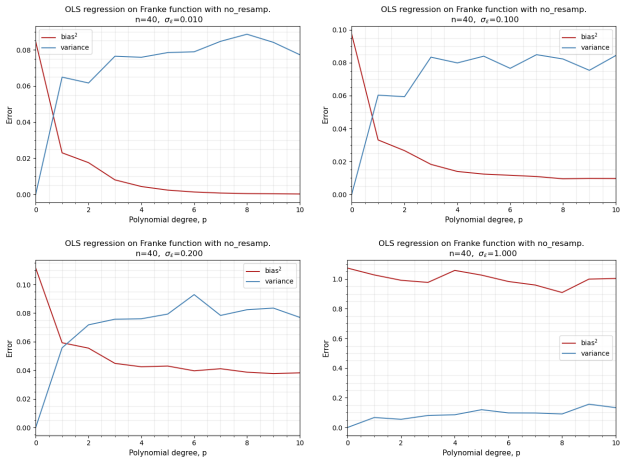


FIG. 7. Bias and variance plotted against model complexity for various noise amplitudes. Here we only plotted up to $p = 10$ to see the intersect better.

As seen in Fig.6, for low noise amplitudes, both R2test and R2train reach a plateau at around $p = 5$, barely inching closer to 1 as model complexity increases. In panels 2 and 3 we see that this plateau is shifted down from 1.0 to 0.9 and 0.7, respectively. This is rather peculiar since the R2 score is given by $R^2 = 1 - \frac{\sum_{i=0}^{n-1}(z_i - \tilde{z}_i)^2}{\sum_{i=0}^{n-1}(z_i - \bar{z})^2}$ and as the model fits the data better and better, $z_i - \tilde{z}_i$ should go to zero for the training data, making R2train approach 1. The plateau was found to stay at 0.7 for $\sigma = 0.2$ even up to $p = 50$, so it is not simply that it is having trouble fitting the noise in the data. The reason for this behavior may be due to some mistake in our R2-score function:

```
def getR2(z,z_tilde):
    n = len(z)
    num = np.sum((z-z_tilde)**2)
    denom = np.sum((z-np.mean(z_tilde))**2)
    return 1-num/denom
#Calculate R2train for a given fit:
scoreVal = getR2(z_test, z_predict)
```

If so, we have not been able to find it.

Regardless, Figure 6 shows that R2test starts to deviate downwards from the plateau as the complexity increases. This is due to overfitting, which explains why this downwards deviation is more pronounced for larger noise amplitudes, as the training data it is fitting gets differs more from the target function.

R2train stays does not exhibit this downwards deviation from the plateau, since this is the data the model was trained to fit (too well).

Now looking at the bias-variance plots of Fig.7., we see in the first panel that the bias rapidly goes to zero. Since the amplitude of the noise is so small, our model shapes itself to fit the training data with minimal ease. We therefore see the bias, which measures how much the model deviates from the target function[9], or in our case the training data, goes to zero quite fast. As we increase the noise, we see that this decline is not as rapid. This is because the model isn't complex enough yet to fit the noise and which leaves us with a greater bias. In addition, it reaches a plateau with a higher value. We plotted the same graph up to $p = 70$ and it never seems to each zero.

The variance also seems to reach a plateau at around $p = 3$. Plotting for larger ranges of p reveals that they are not just simply approaching the same values at slower rates.

We can believe this behavior might be explained by the model not yet having the complexity necessary to fit the noise any better. However, once it does, it more than happily does so, as illustrated in [fig in appendix]

B. Exercise 2: bias-variance tradeoff and Bootstrap

Fig.8 shows the bias squared, variance and MSPE(MSEtest) again plotted vs. complexity, but this time we have used bootstrap resampling with 30 samples to improve the precision of our measurements. From Eq.(31) one would expect that neither the bias nor the variance should ever exceed the MSPE since all terms on the LHS are postive. One might think that this is because we decided on using a different expression for the bias in Fig. 8; [10]

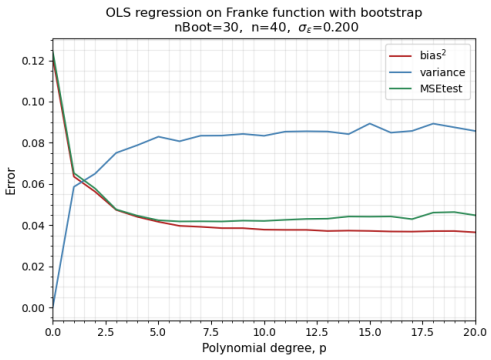


FIG. 8.

But using the original expression for bias leaves us no happier. Now the bias never approaches zero, and lies above the variance even for higher complexities where we would expect it to approach zero. We also see that both the variance and bias now have a higher value than the

[9] Since we usually don't have the target function, the training data is used instead.

[10] We now realized that the expression we used for bias in the above figures is the MSEtrain, but it's too late to change.

MSPE, which doesn't make sense. We have either made some error in our code [11] or some/all of our assumptions in the derivation of the expression for MSPE were wrong.

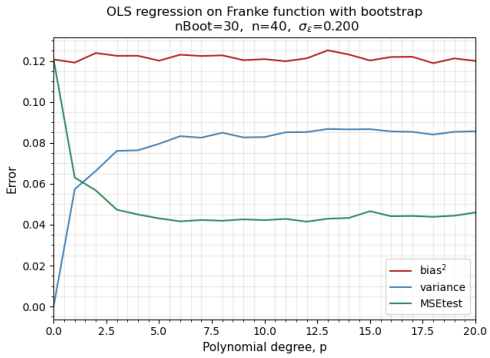


FIG. 9.

What we should expect is to see a decreasing bias and an increasing variance with complexity. The MSPE should initially be high at low complexity, since the bias is high, and reach a minimum around the point where the bias (squared) is equal to the variance. It is at this minimum that we find the optimal model complexity. Our suspicion is that the expressions we use for bias and variance are wrong, since the MSE appears to behave as expected, as can be seen in panel 3 in Fig.5, which corresponds to our chosen noise level. Here one sees that

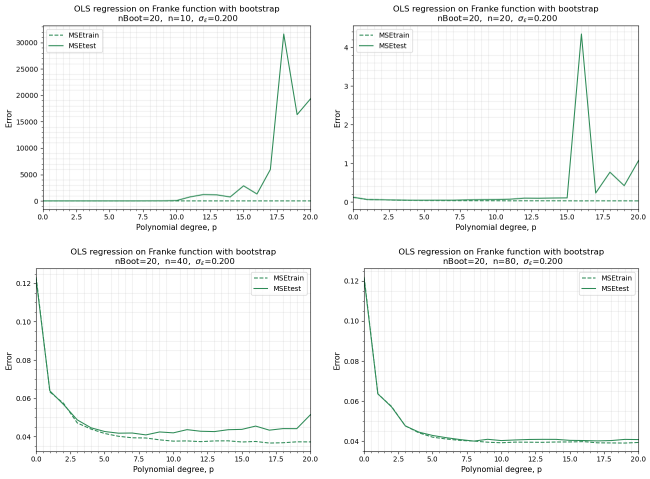


FIG. 10.

Each panel here plots MSEtest vs MSEtrain against model complexity for a specific number of gridpoints, given by n^2 where n is specified in the title. Panels 1 and 2 might init

In contrast to panel 3 in Fig.5, this is done using bootstrap resampling with $B = 30$ bootstrap samples. As can be seen, the result is a much smoother curve. This is because the variance of our estimators MSEtrain and MSEtest goes as $\sim \frac{1}{B}\sigma^2_{\text{MSE}}$ as discussed in the Theory section. To determine the optimal polynomial degree for OLS we therefore reproduce the MSE and R2 score plots, but now with a higher number of bootstrap samples. The results of this are shown in Fig.11., where we used 500 bootstrap samples. We see that the optimal polynomial degree given at $p = 8$ in both cases, with MSEtest = 0.04179 and R2=0.6591 at these extrema. These results will come in handy when comparing with Ridge and Lasso regression.

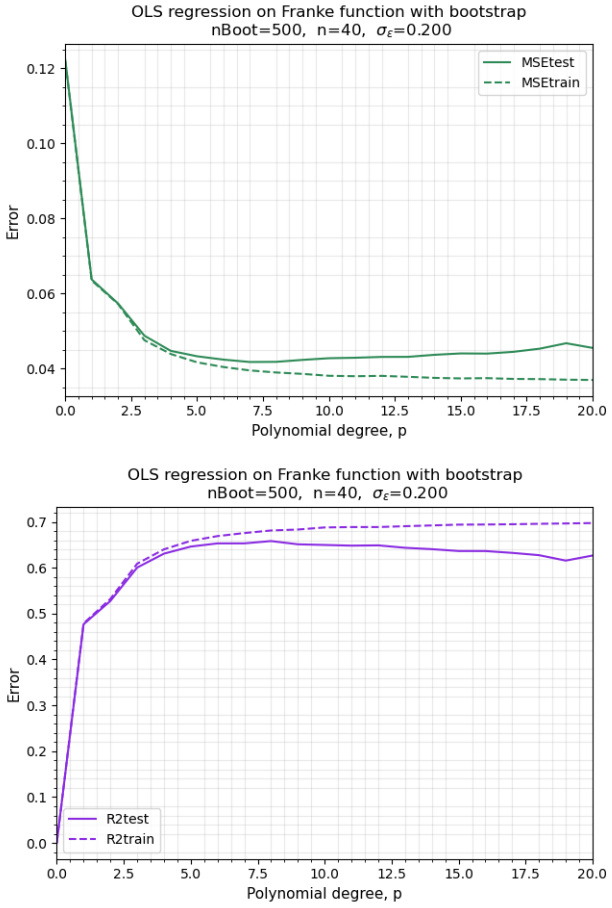


FIG. 11. MSEtest minimum: 0.04179, at p=8. R2test maximum: 0.6591, at p=8. run times: 400.01 and 432.64sec, respectively

[11] all errors are calculated using the same six expressions given in "getTheScore()" in functions.py, which should make it easier to correct for someone less confused

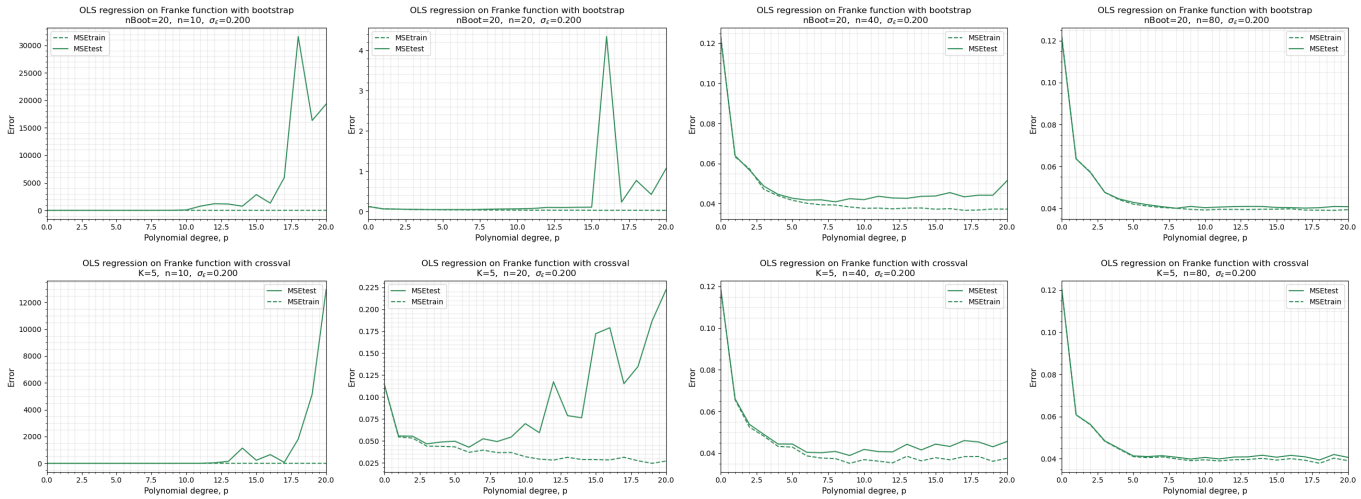


FIG. 12. In the top row, we see MSEtest vs train results using bootstrap for various number of gridpoints, while in the bottom row we used cross validation. For n=10 and n=20, we see that cross validation gives much a much lower error for high complexity. This is to be expected, as discussed in the theory section. The reason we see such high errors at low number of gridpoints is that because it is harder for the model to pick up the underlying shape of the function. To see this, consider for example $n = 1$. Here it is impossible for the model to deduce the target value, as there is no way to differentiate the noise and the true value. However, as the number gridpoints increases the errors will fluctuate around zero and not add any specific shape to the "surface". This allows the model to deduce the shape of the true function and fit to it, rather than the noise. This is demonstrated in Fig.13 Now, for n=40 and n=80, corresponding to 1600 and 6400 gridpoints, we can evaluate the shape of the curves since it is not drowned out by overfitting. We see that even at 20 cycles, the bootstrap method outperforms cross validation in that it produces a smoother curve and thus less variance in the measurements. We will therefore opt to use bootstrap as the resampling method of choice in determining the optimal parameter values.

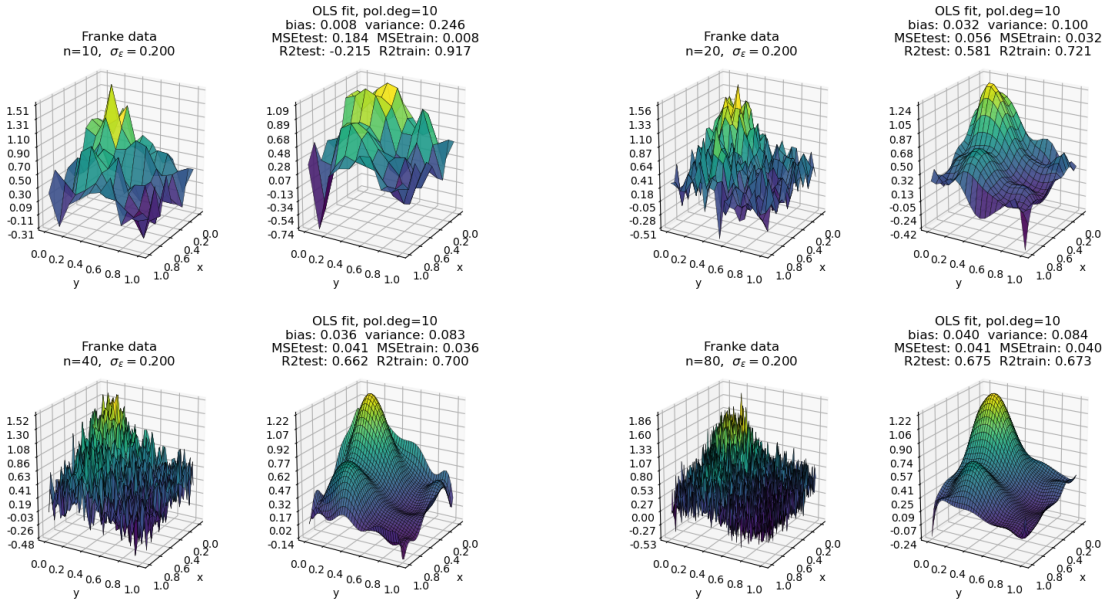


FIG. 13. To reproduce these figures, one can run "run_exercises.py" in the same directory as our other scripts and choose exercise1,task1. Note that in this particular instance, the figures produced don't correspond to the exercise.

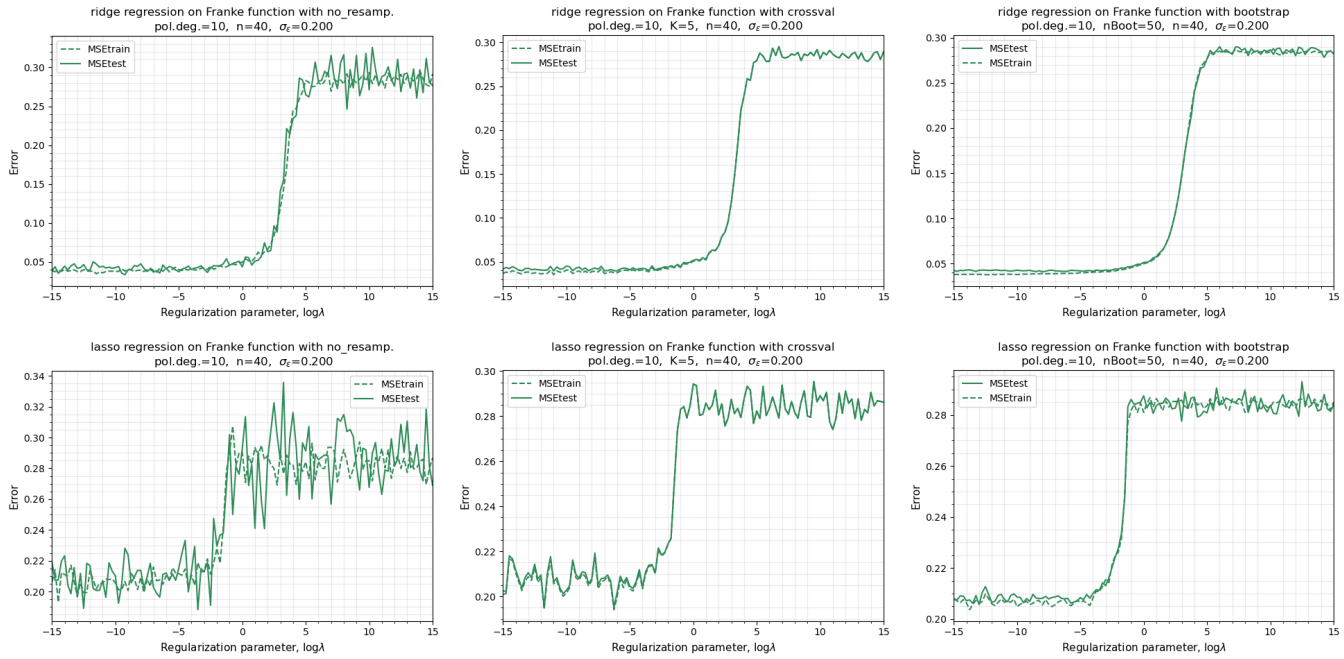


FIG. 14.

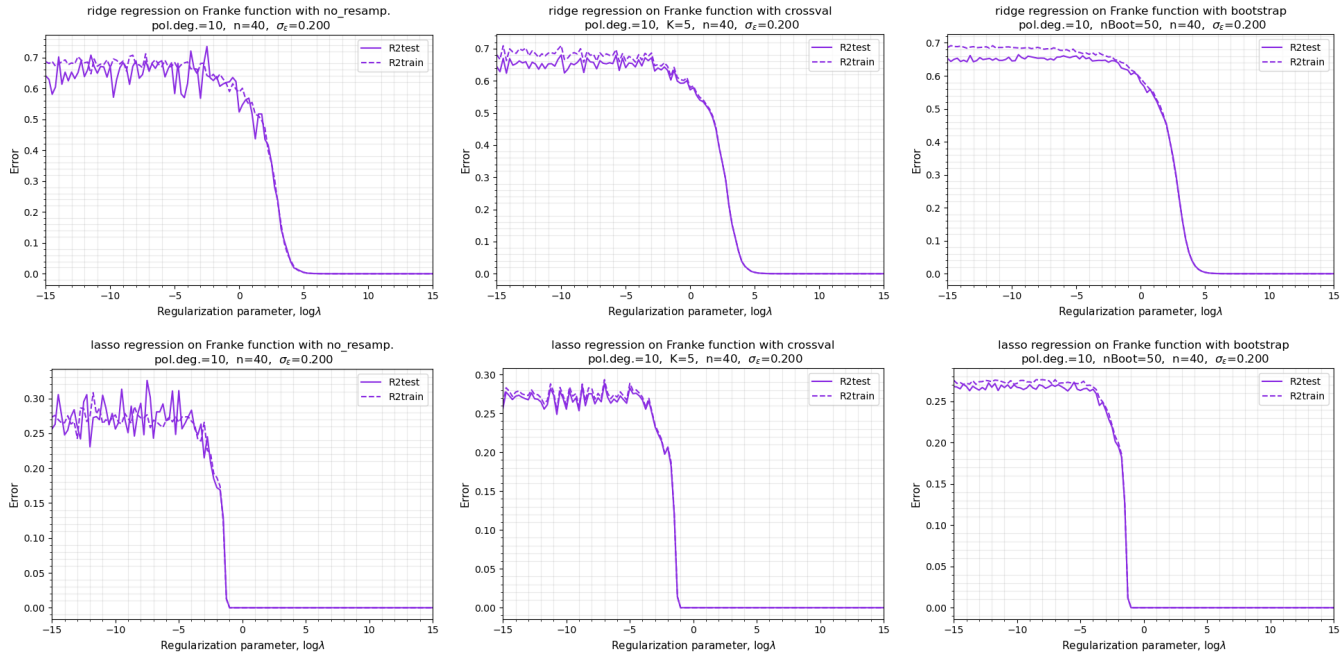


FIG. 15.

Figures 14 and 15 show the MSPE and the R2-score, respectively. In each figure we compare the results Ridge(1st row) and Lasso(2nd row) for no resampling, cross validation(k=5) and lastly bootstrap(B=50). We see that bootstrap produces the smoothest curve in all cases, meaning it gives the lowest variance in our MSPE and R2-score estimates. We also see that Lasso regresion has a higher variance for the MSPE than ridge, while the variance in the R2-score doesn't appear very different for the two regression models.

Furthermore, we see both models performing well up to some threshold value of lambda, which seems to be reached

around at $\log_{10}(\lambda) = 4$ for Ridge and $\log_{10}(\lambda) = -1$ for Ridge. Beyond this point, both methods have a quickly decreasing performance with lambda, (which happens more rapidly for Lasso) before reaching a plateau. At this point, the shrinkage parameter lambda has probably shrunk the regression coefficients close to zero. The fact that Lasso regression can make them go to zero instead of approaching it might explain why we observe a steeper curve o the plateau.

From these figures we can also compare their performance in their "functional" region of lambda. We see that the MSPE of Lasso regression is about four times that of Ridge in this region. In Fig.15 we see that the R2-score for Lasso plateaus at a value of somewhere between 0.25 and 0.3 while Ridge does so at 0.6-0.7.

For this polynomial degree, viz. $p = 10$, Ridge certainly outperforms Lasso on our Franke data. In order to evaluate their performance as a function of both lambda and polynomial degree, we produce heatmaps in their functional range, as seen in Fig.16 and Fig.17. [1]

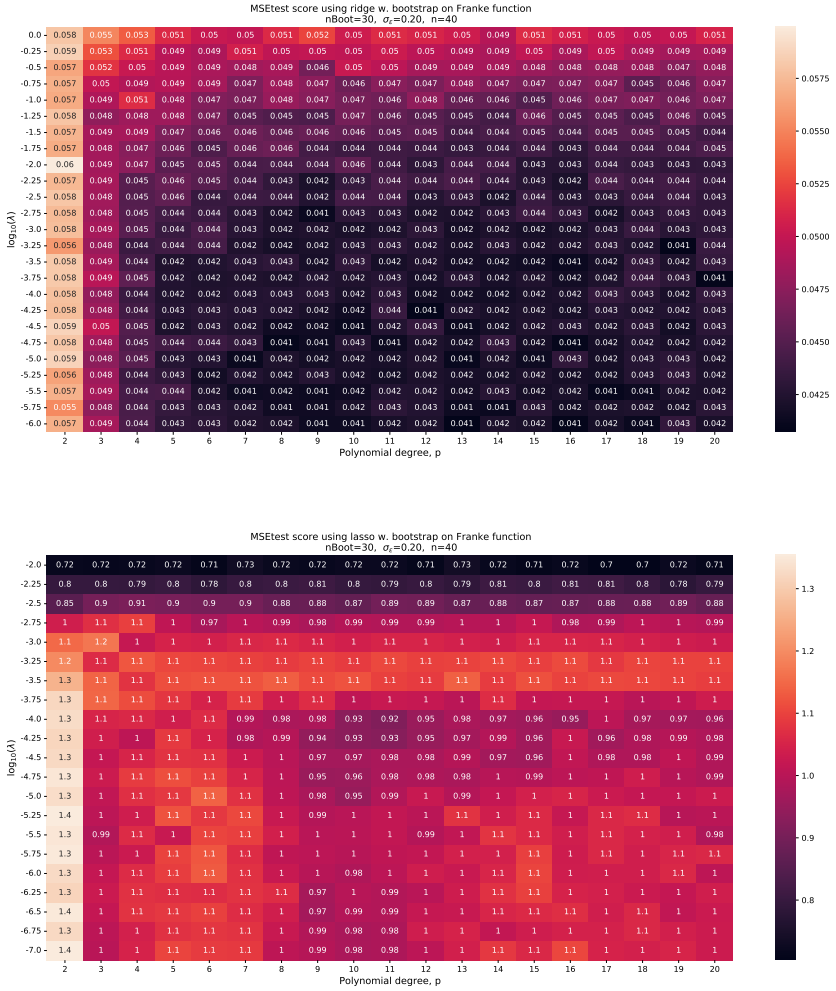


FIG. 16.

As suspected Lasso performs worse for a wide range of values of lambda and p. We also realize that our Ridge MSEtest measurement lacks the number of decimals required to determine whether its minimum is greater or smaller than that of OLS. However, since (unlike OLS) Ridge is inherently a biased estimation method, it might be preferable to simply use OLS of with a polynomial of degree 8.

Further improvements would include determining the minimum of MSEtest for Ridge more accurately.

TABLE I. MSEtest maxima and R2test minima of the three different regression methods, obtained using bootstrap with B=500 for OLS and 30 for Ridge and Lasso.

	OLS	Ridge		Lasso	
MSEtest(OLS) minimum	R2test(OLS) maximum	MSEtest(Ridge) minimum	R2test(Ridge) maximum	MSEtest(Lasso) minimum	R2test(Lasso) maximum
0.04179	0.6591	0.041	0.67	0.7	0.079

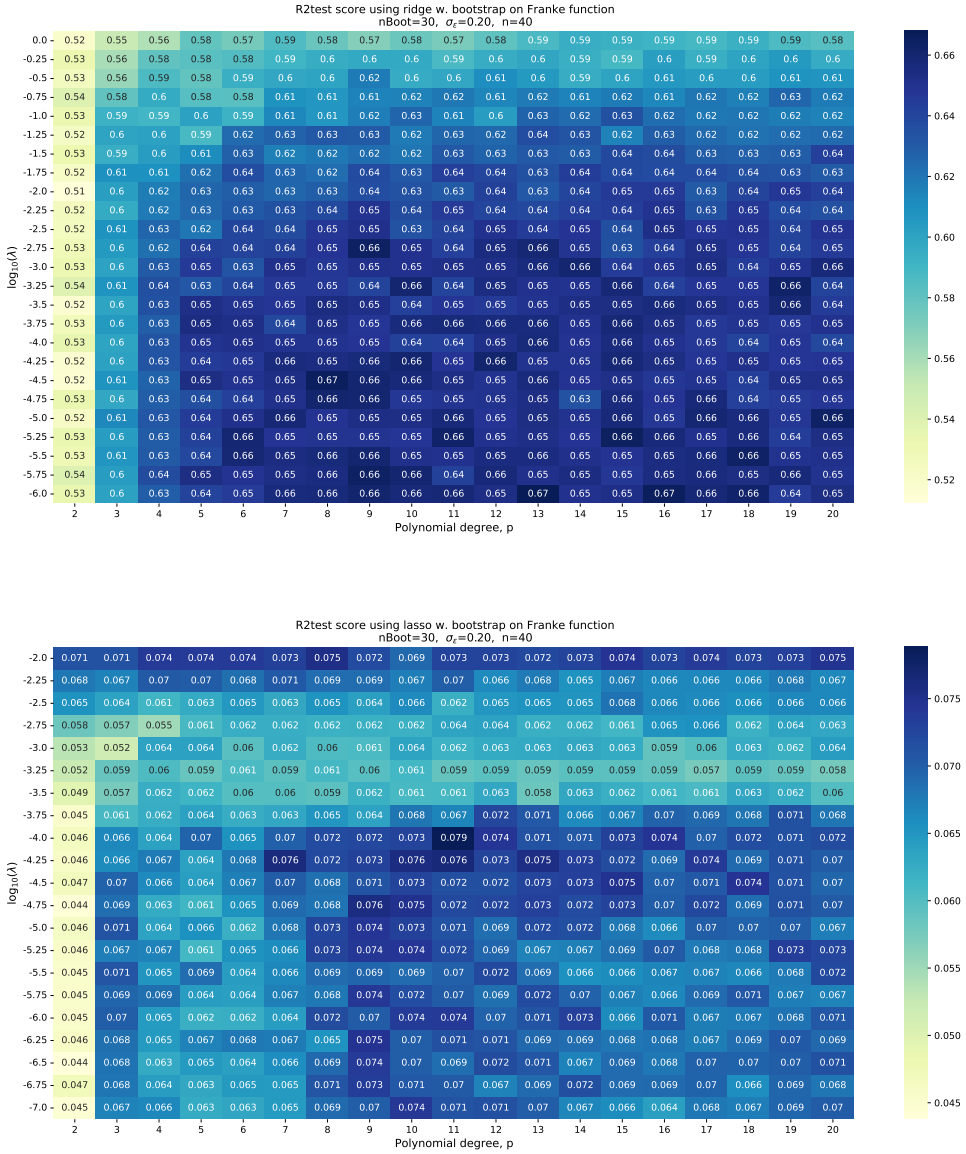
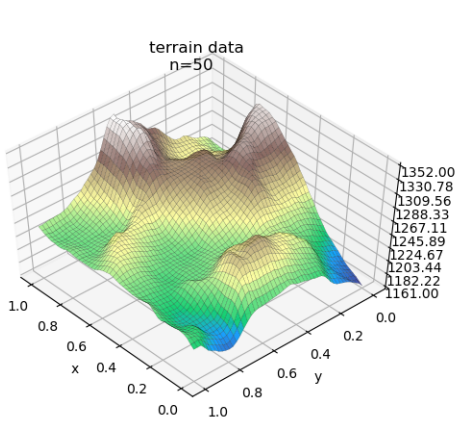
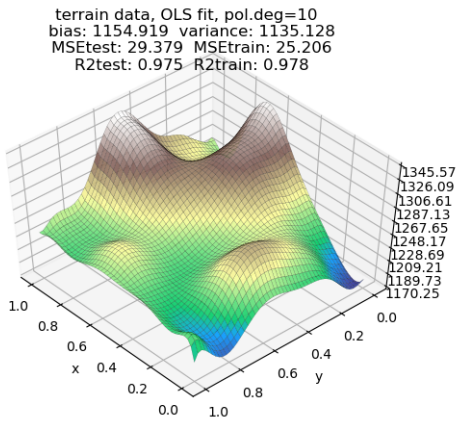


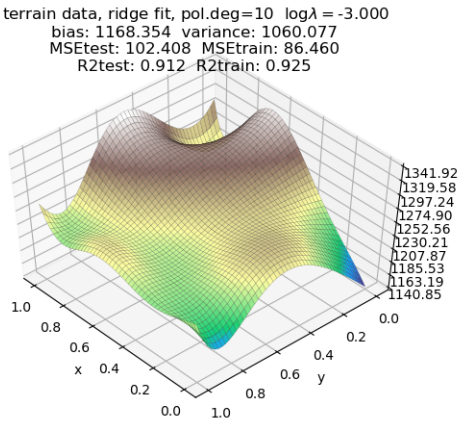
FIG. 17.

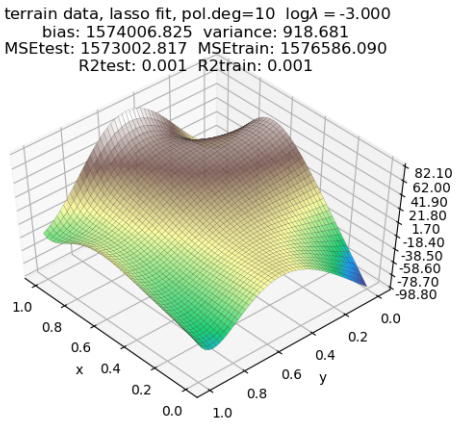
C. Terrain data

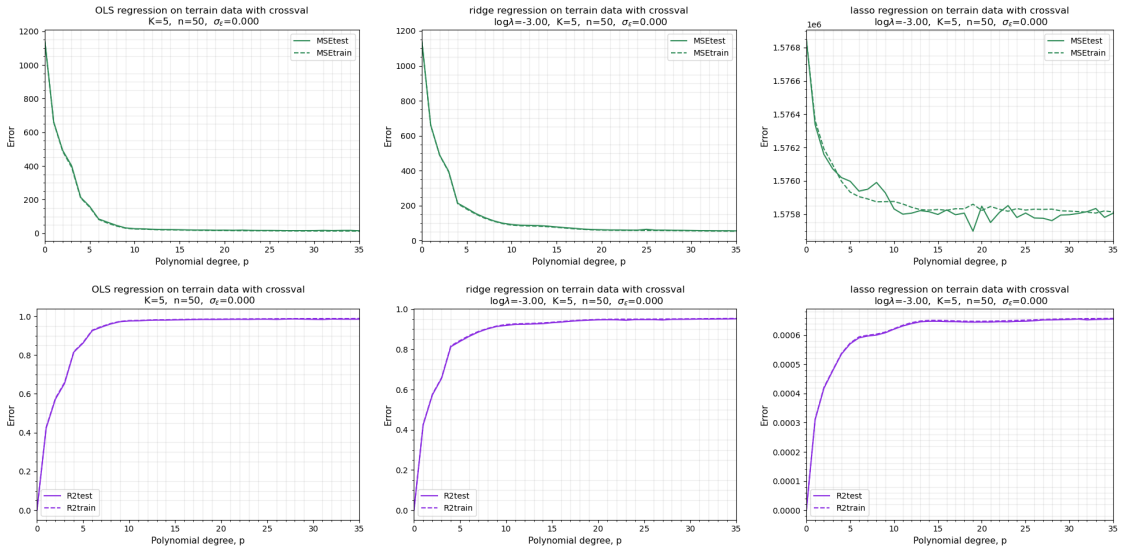
In this section, we use assess our models on the terrain 1 data provided in [3]. While performing model assessment, we limit ourselves to using a data sample of 2500 points, corresponding to $n = 50$.











In IV C, we see the original terrain data along with an OLS, Ridge and Lasso fit on the same data. For this single sample, we see that OLS obtained the best fit, followed by Ridge and Lasso. We have also plotted MSEtest and R2test for these three regression methods as a function of model complexity, which shows Lasso to perform the worst,

while Ridge comes in at at a close second to OLS, which gives a lower error at low complexities than Ridge does. For higher complexity it is not so obvious which of the two performs best.

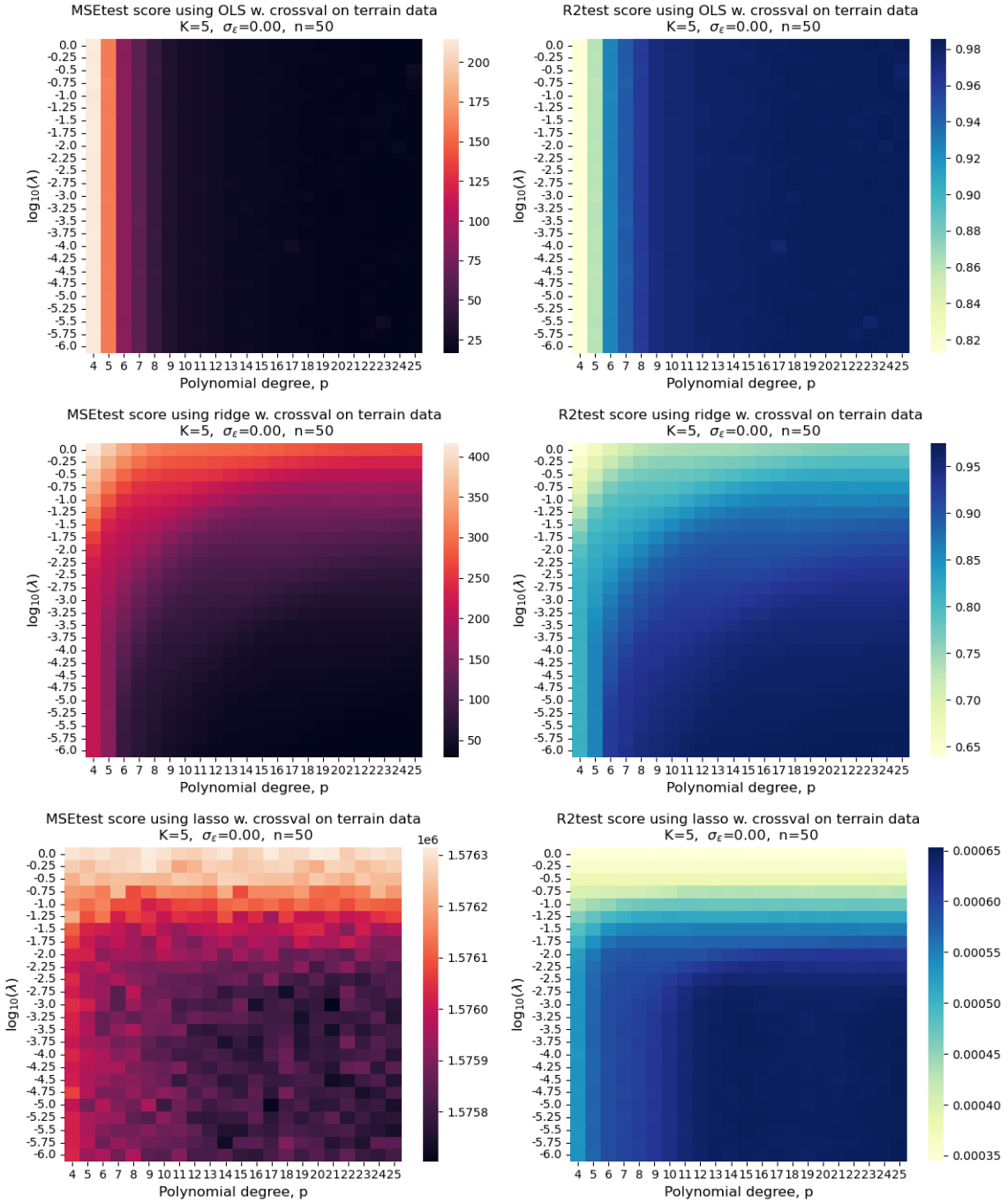


FIG. 18. From these heatmaps we see that that both Ridge and OLS outperforms Lasso regression in the selected parameter space. OLS also seems to outperform Ridge regression, although it is somewhat hard to tell for values of $\lambda < -2.5$ and $p > 10$. We therefore produce new annotated heatmaps with those values as cut-offs

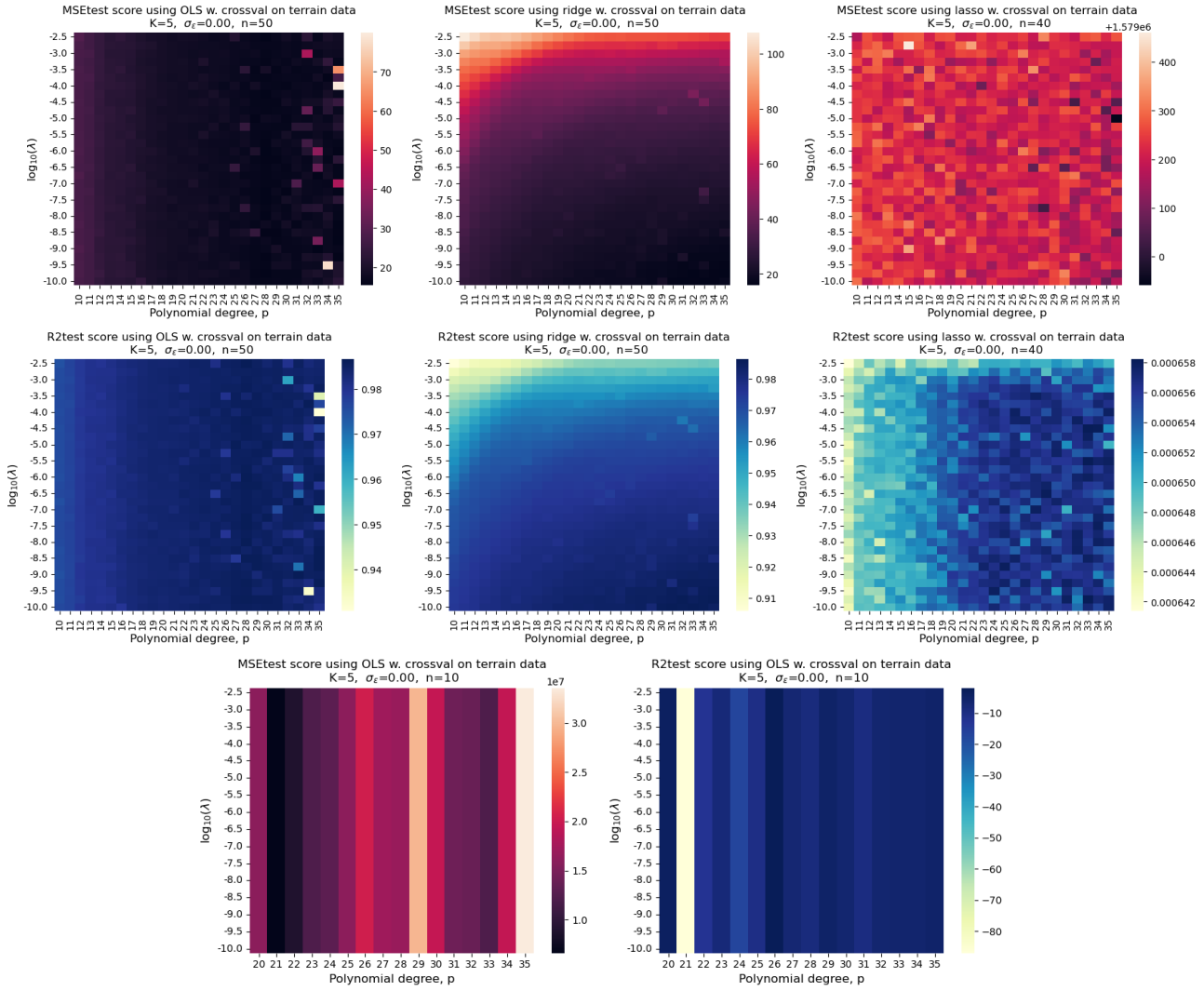


FIG. 19. We see here that for very high degree polynomials, we see start seeing variations in the MSPE and R2-score along the λ-axis of the OLS heatmap. The reason for this behavior is overfitting. As our model becomes increasingly complex, it starts also fitting the noise in the data, inheriting features that aren't present in the target function, i.e. we are overfitting. If the OLS regression algorithm saw the same data set each time, the variations with λ wouldn't make sense, since it would be fitting the same noise each time, and although the fit might be terribly off, it should yield the same errors each time for a given polynomial order. However, since the cross validation algorithm randomly shuffles our data each iteration (but not for each fold), the model fits a new data set for each pixel in the heatmap. Turning shuffling off, should therefore result in our errors being invariant along the λ axis, as is demonstrated by the two bottom heatmaps. We see that for polynomials of degree 25 or higher, Ridge regression doesn't possess this behavior, which is to be expected since it is designed to prevent overfitting through feature suppression.

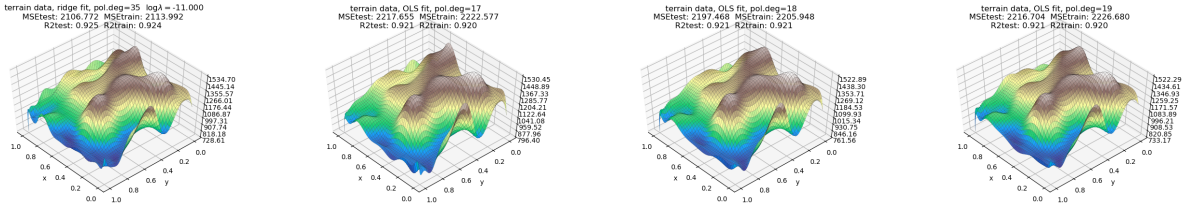


FIG. 20.

The fits in Fig.20 were done using n=400. We see a Ridge fit of order 35 as well as OLS fits of orders 17, 18 and 19. Their run times were 16.74, 3.75, 3.94 and 4.44 seconds respectively. So even though it seems we can achieve a slightly better fit with Ridge regression, it comes at about 4 times the computational cost. Furthermore, we see that among the OLS models, the 18th degree polynomial yields best the lowest MSPE for $n = 500$. Fig.19. shows this (with more precision) to be the case for $n = 50$ as well, so we can suspect that $p = 18$ will remain optimal for larger data sets still. We therefore choose the OLS with polynomial degree 18 as our optimal model for the terrain data. In Fig.21, we can see our chosen model fitten on a million datapoints.

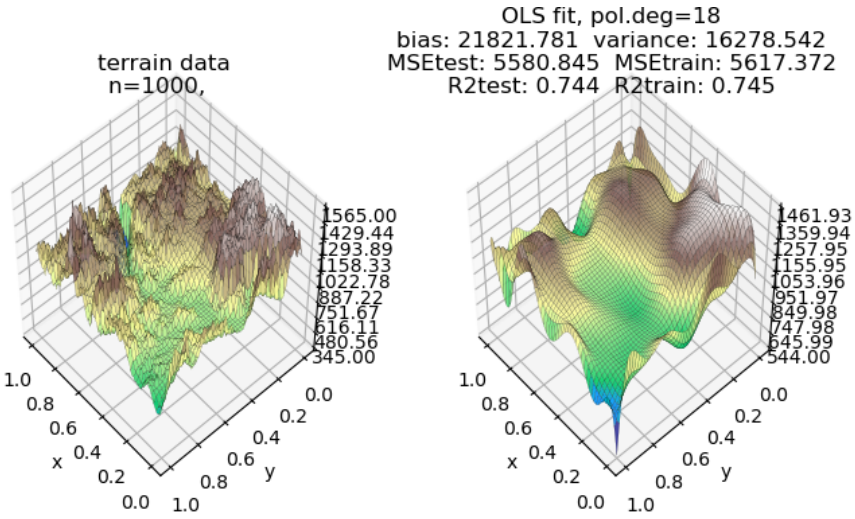


FIG. 21.

References

[1] M. Hjorth-Jensen. *Applied Data Analysis and Machine Learning, FYS-STK3155/4155 at the University of Oslo, Norway.* URL: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`.

[2] M. Hjorth-Jensen. *Overview of course material: Data Analysis and Machine Learning.* URL: `https://compphysics.github.io/MachineLearning/doc/pub/week34/html/week34.html`.

[3] $SRTM_data_Norway_1.tif$. Sept. 2021. URL: `https://github.com/CompPhysics/MachineLearning/tree/master/doc/Projects/2021/Project1/DataFiles`.

Appendices

A. Franke's function

The Franke function is a bivariate function, and so outputs a matrix $\mathbf{Z} = \mathcal{F}(xx, yy)$ given by

$$\mathbf{Z} = \begin{bmatrix} \mathcal{F}(x_0, y_0) & \mathcal{F}(x_1, y_0) & \dots & \mathcal{F}(x_{n-1}, y_0) \\ \mathcal{F}(x_0, y_1) & \mathcal{F}(x_1, y_1) & \dots & \mathcal{F}(x_{n-1}, y_1) \\ \vdots & \vdots & \vdots & \vdots \\ \mathcal{F}(x_0, y_{n-1}) & \mathcal{F}(x_1, y_{n-1}) & \dots & \mathcal{F}(x_{n-1}, y_{n-1}) \end{bmatrix} \tag{A.1}$$

And we would like to use the OLS algorithm from [reference theory] to produce a model

$$\tilde{\mathbf{Z}} = \begin{bmatrix} \tilde{Z}(x_0, y_0) & \tilde{Z}(x_1, y_0) & \dots & \tilde{Z}(x_{n-1}, y_0) \\ \tilde{Z}(x_0, y_1) & \tilde{Z}(x_1, y_1) & \dots & \tilde{Z}(x_{n-1}, y_1) \\ \vdots & \vdots & \vdots & \vdots \\ \tilde{Z}(x_0, y_{n-1}) & \tilde{Z}(x_1, y_{n-1}) & \dots & \tilde{Z}(x_{n-1}, y_{n-1}) \end{bmatrix} \tag{A.2}$$

Where we denote $\tilde{\mathbf{Z}}_{i,j}$ by $\tilde{Z}(x_i, y_j)$. However, $\mathbf{Z}$ and $\tilde{\mathbf{Z}}$ are matrices and the OLS algorithm assumes $\mathbf{y}$ and $\tilde{\mathbf{y}}$ to be column vectors. We can solve this problem by using the NumPy function *ravel* on $\mathbf{Z}$ and $\tilde{\mathbf{Z}}$ to produce $\mathbf{z}$ and $\tilde{\mathbf{z}}$, respectively:

$$\mathbf{z} = \begin{bmatrix} \mathcal{F}(x_0, y_0) \\ \mathcal{F}(x_1, y_0) \\ \vdots \\ \mathcal{F}(x_{n-1}, y_0) \\ \mathcal{F}(x_0, y_1) \\ \mathcal{F}(x_1, y_1) \\ \vdots \\ \mathcal{F}(x_{n-1}, y_1) \\ \mathcal{F}(x_0, y_{n-1}) \\ \mathcal{F}(x_1, y_{n-1}) \\ \vdots \\ \mathcal{F}(x_{n-1}, y_{n-1}) \end{bmatrix} \tag{A.3}$$

$$\tilde{\mathbf{z}} = \begin{bmatrix} \tilde{Z}(x_0, y_0) \\ \tilde{Z}(x_1, y_0) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_0) \\ \tilde{Z}(x_0, y_1) \\ \tilde{Z}(x_1, y_1) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_1) \\ \tilde{Z}(x_0, y_{n-1}) \\ \tilde{Z}(x_1, y_{n-1}) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_{n-1}) \end{bmatrix} \tag{A.4}$$

Since we are using polynomials of x and y to interpolate $\mathbf{z}$, we have

$$\begin{aligned} \tilde{Z}(x_i, y_j) = \beta_0 + \beta_1 x_i + \beta_2 y_i + \beta_3 x_i^2 + \beta_4 x_i y_j + \beta_5 y_j^2 \\ + \dots + \beta_{\frac{(p+2)(p+1)}{2}} y_j^p \end{aligned} \tag{A.5}$$

and so we get

$$\begin{bmatrix} \tilde{Z}(x_0, y_0) \\ \tilde{Z}(x_1, y_0) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_0) \\ \tilde{Z}(x_1, y_1) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_1) \\ \vdots \\ \vdots \\ \tilde{Z}(x_0, y_{n-1}) \\ \tilde{Z}(x_1, y_{n-1}) \\ \vdots \\ \tilde{Z}(x_{n-1}, y_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & y_0 & x_0^2 & x_0 y_0 & y_0^2 & \dots & y_0^p \\ 1 & x_1 & y_0 & x_1^2 & x_1 y_0 & y_0^2 & \dots & y_0^p \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & y_0 & x_{n-1}^2 & x_{n-1} y_0 & y_0^2 & \dots & y_0^p \\ 1 & x_0 & y_1 & x_0^2 & x_0 y_1 & y_1^2 & \dots & y_1^p \\ 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 & \dots & y_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & y_1 & x_{n-1}^2 & x_{n-1} y_1 & y_1^2 & \dots & y_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_0 & y_{n-1} & x_0^2 & x_0 y_{n-1} & y_{n-1}^2 & \dots & y_{n-1}^p \\ 1 & x_1 & y_{n-1} & x_1^2 & x_1 y_{n-1} & y_1^2 & \dots & y_{n-1}^p \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & y_{n-1} & x_{n-1}^2 & x_{n-1} y_{n-1} & y_{n-1}^2 & \dots & y_{n-1}^p \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \\ \vdots \\ \beta_{(p+2)(p+1)/2} \end{bmatrix} \tag{A.6}$$

We write this as $\tilde{\mathbf{z}} = \mathbf{X}\boldsymbol{\beta}$. We see that we can write the design matrix as

$$\mathbf{X} = [\mathbf{1}, x_r, y_r, x_r^2, x_r y_r, y_r^2, \ldots, y_r^p] \tag{A.7}$$

Where the k-th column of $\mathbf{X}$ corresponds to the same combination of x_r and y_r as the combination of x and y in the k-th term of the polynomial given in (A.5). Ignoring the regression coefficients, and dropping the indices, this polynomial is given by

$$P(x, y) = \sum_{l=0}^{p} \sum_{k=0}^{l} x^{l-k} y^k \tag{A.8}$$

The total number of terms in the polynomial is given by $N = \frac{(p+2)(p+1)}{2}$, which can be found by constructing a Pascal's triangle with each row containing only the terms of a given power, and realizing that the number of terms in the rows form a geometric series. We can now construct our design matrix from x_r and y_r (shown below) for a given power of the polynomial, p:

```
#Create the design matrix
def desMat(xr,yr,p):
    N = len(xr)
    #Number of elements in beta:
    numEl = int((p+2)*(p+1)/2)
    X = np.ones((N,numEl))
    colInd=0#Column index
    for l in range(1,p+1):
        for k in range(0,l+1):
            X[:,colInd+1] = (xr**(l-k))*yr**k
            colInd = colInd+1
    return X
```

We can use the unravel function of numpy on xx and yy to yield $\mathbf{x}_r$ and $\mathbf{y}_r$:

$$\mathbf{x}_r = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ \vdots \\ \vdots \\ x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{A.9}$$

$$\mathbf{y}_r = \begin{bmatrix} y_0 \\ y_0 \\ \vdots \\ y_0 \\ y_1 \\ y_1 \\ \vdots \\ y_1 \\ \vdots \\ \vdots \\ y_{n-1} \\ y_{n-1} \\ \vdots \\ y_{n-1} \end{bmatrix} \tag{A.10}$$

We see that the columns of $\mathbf{X}$ are simply products of various powers of $\mathbf{x_r}$ and $\mathbf{y_r}$

B. Matrix calculus identities

In this section we will prove the following useful identities[2]

$$\frac{\partial \mathbf{b}^T \mathbf{a}}{\partial \mathbf{a}} = \mathbf{b} \tag{B.1}$$

$$\frac{\partial \mathbf{a}^T \mathbf{A} \mathbf{a}}{\partial \mathbf{a}} = \left(\mathbf{A} + \mathbf{A}^T\right) \mathbf{a} \tag{B.2}$$

$$\frac{\partial \mathbf{b}^T \mathbf{a}}{\partial \mathbf{a}} = \mathbf{b} \tag{B.3}$$

$$\frac{\partial \mathbf{a}^T \mathbf{A} \mathbf{a}}{\partial \mathbf{a}} = \left(\mathbf{A} + \mathbf{A}^T\right) \mathbf{a} \tag{B.4}$$

Here column vectors are denoted by boldface while matrices are denoted by capital boldface letters. The derivative with respect to a vector is defined as

$$\frac{\partial}{\partial \mathbf{a}} \equiv \begin{bmatrix} \partial/\partial a_0 \\ \partial/\partial a_1 \\ \vdots \\ \partial/\partial a_{n-1} \end{bmatrix} \tag{B.5}$$

```
#=======================================================================================================
#================================ MAIN FUNCTION ========================================================
#=======================================================================================================
def main():
    np.random.seed(420)
#=======================================================================================================
#========================================== CONTROL PANEL ==============================================
    ONbutton   = True    #Set to False to run main.py without performing regression with the parameters
    tinkerBool = False    #If True, a folder named 'tinkerings' will be created if savePlot is True, in
    exercise = 4          #If tinkerBool=False, a folder named 'exercise_results' if savePlot is True. 1

    terrainBool = False  #Use terrain data or Franke function?
    n_t = 100            #How many points on the x and y axes if using terrain data
    n_f = 20             #How many points on the x and y axes if using FrankeFunction
    n = pick_n(n_f,n_t,terrainBool)
    scaling = False      #Scale the design matrix, X?
    skOLS = False        #Use sklearn in OLS (rather than pseudoinverse)?
    skCV = False         #Use sklearn's cross validation contra own code?

    nBoot = 100          #Number of bootstrap samples
    K = 5                #Number of folds in cross validation alg.
    shuffle = True       #Shuffle the data before performing crossval folds?

    #---------------------------------------
    #S=0.1 #Use this for fixed standard deviation as a function of n
    S = 40/(n**2)#Is equal to 0.1 when n=20. Ensures sigma scales correctly with n, discussed in report.
    sigma_v  =          [0.5*S,S,2*S,5*S]  #Make a separate plot for each of these sigmas
    sigma_s  =               [10*S]        #Default standard deviation of epsilon
    #---------------------------------------
    minOrder,maxOrder =      1, 20         #Will make order vector from minOrder to maxOrder
    order_s =                 [10]          #Default pol.degree if we don't plot vs. degrees
    #---------------------------------------
    minLoglmd, maxLoglmd =   -15, 15       #Will make log(lambda) vector from minLoglmd to maxLoglmd
    lambda_s  =               [1e-4]       #Default log(lambda) value. Must be set
    #---------------------------------------
    sigmasBool = False  #If true, will produce a plot for each std. in sigma_v

                # [ordersBool lamdasBool] = plotTypeInt
    plotTypeInt =               0
                # [  False   0   False  ]  Generate surf plots and print results
                # [  True    1   False  ]  Plots error vs. pol.deg.
                # [  False   2   True   ]  Plots error vs. lambda
                # [  True    3   True   ]  Produces heatmap
    savePlot = True #Save plots?
    plotBool = True #Make error/score plots?

    resampInt = 0   #0=no_resamp., 1=bootstrap, 2=crossval
    regInt    = 1   #0=OLS,        1=ridge,     2=lasso

    allScores = [['bias'],['variance'],['MSEtest'],['MSEtrain'],['R2test'],['R2train']] #Which regressio
    #              0          1           2           3          4          5
    scoresNames = getScores(0,1)
    #Sets which scores to calculate by passing in the corresponding index from allScores.
    #Function def before main().
#=======================================================================================================
#=======================================================================================================
#=======================================================================================================
```