FYS-STK 3155 project 1

Theodor Midtbø Alstad

10th October 2020

Abstract

The project detailed in this report is unfinished due to time constraints, and has many ways it could be improved.

This project aims to explore linear regression, with the final goal of learning and predicting the geography of an area. The project is split into 7 sections, labelled a) through g), culminating in application to geographical data in section g), and will be referred to as such in the report.

The final goal of fitting models to geographical data is not met, and the best fit gotten is a model with an R^2 score of -125.

1 Introduction

This project implements the regression methods Ordinary Least Squares, LASSO, and Ridge, with the resampling methods bootstrap and k-fold cross-validation.

The project starts by modelling the Franke function in the for the domain $x, y \in [0, 1]$, before moving on to geographical data at the end.

Throughout the project, multiple bias-variance and test-train MSE analyses are done, combining the regression- and resampling methods to do so.

The goal of this project is two-fold; to model geographical data and to learn more about, and evaluate, the regression methods implemented.

1.1 The structure of the source code

The code is written in python3, formatted as a cascading inheriting class structure. That is, section b) inherits from the class of section a), section c) inherits from the class of section b), and so on.

All outputs are found in the outputs folder, as detailed in the .README file.

1.2 Franke function

The Franke function is modelled in sections a) through e), as a placeholder for geographical data in section g). It is here defined for the domain $x, y \in [0, 1]$

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

1.3 Ordinary Least Squares (OLS)

The OLS method attempts to model a dataset by minimizing the square error between the dataset, y, and a model in the form of $\hat{y} = \mathbf{X}\beta$, where $\mathbf{X}$ is a design matrix and β is a vector of parameters.

OLS successfully and analytically minimizes the cost function $C = \mathbb{E}[(y-\hat{y})^2]$ by using parameters

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

1.4 Bias & Variance

Using a cost function C based on the Mean Square Error between our prediction, $\hat{y}$, and the recorded value, $y = f(x) + \epsilon$, the bias and variance can be found as

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

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

$$\begin{aligned}
\mathbb{E}[2(y - \mathbb{E}[\hat{y}])(\mathbb{E}[\hat{y}] - \hat{y})] &= 2(y - \mathbb{E}[\hat{y}])\mathbb{E}[(\mathbb{E}[\hat{y}] - \hat{y})] \\
&= 2(y - \mathbb{E}[\hat{y}]) \times 0 \\
&= 0
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}[(y - \mathbb{E}[\hat{y}])^2] &= \mathbb{E}[((f(x)+\epsilon) - \mathbb{E}[\hat{y}])^2] \\
&= \mathbb{E}[((f(x) - \mathbb{E}[\hat{y}]) + \epsilon)^2] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2 + \epsilon(f(x) - \mathbb{E}[\hat{y}]) + \epsilon^2] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2 + 0 + \epsilon^2] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2] + \mathbb{E}[\epsilon^2] \\
&= \mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2] + \sigma^2
\end{aligned}$$

$$C = \mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2] + \mathbb{E}[(\mathbb{E}[\hat{y}] - \hat{y})] + \sigma^2$$

where $\mathbb{E}[(f(x) - \mathbb{E}[\hat{y}])^2]$ is the bias, $\mathbb{E}[(\mathbb{E}[\hat{y}] - \hat{y})]$ is the variance, and σ^2 is the irreducible error.
Bias can be interpreted as the expected mean square error between a prediction and their true values, and is often used to determine how well fitted a model is.

It usually goes down with the complexity of a model, as the model is more able to bend to the training points.
Variance can be interpreted as how much a model varies over multiple iterations, and is often used to determine whether a model is overfitted to its test data. It usually goes up with the complexity of a model, as the model will twist, bend, and turn more to fit the training points.
The irreducible error is an error that cannot be overcome by improving the model, and can be attributed to things such as noise.

1.5 Bootstrap Resampling

The bootstrap resampling method aims to evaluate a regression method by generating multiple training sets to a single test set. It accomplishes this by initially splitting a complete data set into training and test sets, then repeatedly creating *sample* sets by picking out random points from the training set (duplicates are allowed in the sample set) until the sample set has a set size (in this report sample sets are given 80% size of the initial training set). Then the regression model is applied to the sample set, evaluated against the testing set, another sample set is drawn, and this is repeated as many times as is specified. An average calculated per evaluation method, and used as an estimate for the evaluation of the regression method.

1.6 K-fold cross-validation resampling

Another resampling method, and the last that will be used here, k-fold cross-validation splits a dataset into k equally sized sets, and treats one of them at a time as a testing set while treating the collection of the remaining $k-1$ sets collectively as a training set. Any evaluations using k-fold cross-validation are evaluated as an average over the k test sets.

1.7 Ridge

The ridge method is an adjustment to the OLS method, and involves introducing a term that is added to the matrix about to be inverted, in the form of a scalar constant λ multiplied into an identity matrix. This helps makes uninvertible matrices invertible, and assists the models accuracy.

$$\beta = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T y$$

They key to using Ridge is finding an optimal λ.

1.8 Lasso

The lasso method differs slightly from ridge in that it attempts to minimize $\sum_{n}^{i=1}\left(y_i - \beta_0 - x_i^T\beta\right)^2$ in terms of β. In this report, scikit learn's method is used in place of creating one;

```
# Python example of lasso implementation using the sklearn module
```

```
# Assuming dataset has been split into X_train, X_test, y_train, and y_test

import sklearn.linear_model as linmod
lassoAlpha = 0.1
clf = linmod.Lasso(alpha=lassoAlpha)
clf.fit(X_train, y_train)
ypredict = clf.predict(X_test)
```

2 Method

2.1 Data generation

The class is given 3 initial arguments; n, p, and noisefactor. n refers to the square of the number of data points used, p refers to the polynomial degree of the design matrix, and noisefactor refers to the amount of noise that should be present in the data.

Raw data generation:
First, $\frac{n}{2}$ points, rounded up, are distributed with linear distribution within the domain of Franke; between 0 and 1.
Second, $\frac{n}{2}$ points, rounded down, are distributed with uniform random distribution within the same domain.
Third, these two sets are combined to form a 1D dataset.
Fourth, this is repeated and the two datasets, x and y, are made to form a meshgrid which is then flattened, creating an even and reliable full, 2D dataset. It is noted that this is a better dataset than one could expect normally, but a similar method is used to extract data from the geographical dataset when the required space or processing power for the full dataset is not available.
Fifth, expected data is generated from these data with the Franke function.
Sixth, normally distributed noise of mean 0 and variance 1, multiplied by the noise factor and the mean of the Franke data, are added to the Franke data.

```
def __init__(self, n, p, noisefactor = 0.1):
  x1 = y1 = np.linspace(0, 1, mt.ceil(n/2))
  x2 = np.random.uniform(0, 1, mt.floor(n/2))
  y2 = np.random.uniform(0, 1, mt.floor(n/2))

  x = np.concatenate((x1, x2))
  y = np.concatenate((y1, y2))

  x, y = np.meshgrid(x,y)
  self.x = x.flatten()
  self.y = y.flatten()

  self.ytrue = self.FrankeFunction(self.x, self.y)
  self.yData = self.ytrue + noisefactor*np.random.randn(n**2)*self.ytrue.mean()
```

Design Matrix generation: The design matrix is made as a polynomial set of x and y, where each column is a polynomial product of the two, and the rows are points of data. The non-constant columns (i.e. all columns but the first) are scaled by subtracting the average value of each respective column from itself.

```
def craftX(self, scaling=True):
  self.nfeatures = int(((self.p+1)*(self.p+2))/2)
  self.X = np.zeros((len(self.x), self.nfeatures))
```

```python

ind = 0
for i in range(self.p+1):
  for j in range(self.p+1-i):
    self.X[:,ind] = self.x**i * self.y**j
    ind += 1

if scaling:
  self.X[:,1:] -= np.mean(self.X[:,1:], axis=0)
```

2.2 a) Ordinary Least Square (OLS) on the Franke function

Implemented in this section:

- Data generation
- R^2 score function
- Mean Squared error (MSE) evaluation function
- Design Matrix & Franke values
- OLS
- Method to find the 95% confidence intervals of the beta parameters of the OLS-method

```python
# Main OLS method
beta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ y_train
```

```python
def MSE(self, y_data, y_model):
    # Optimal value is 0, with higher values beign worse
    return np.sum((y_data-y_model)**2) / np.size(y_model)

def R2(self, y_data, y_model):
    # Optimal value is 1, with 0 implying that model performs
    # exactly as well as predicting using the data average would.
    # Lower values imply that predicting using the average would be ↩
        better
    top = np.sum((y_data - y_model)**2)
    bot = np.sum((y_data - np.mean(y_data))**2)
    return 1 - top/bot
```

This section generated data as described in Section 2.1, attempted to model it with an OLS approximation, and evaluated the MSE, R^2 score, and the confidence interval of the β-parameters.

The confidence interval of the β-parameters was found by running an OLS model 300 times, finding 300 examples of the β-parameters, calculating the standard deviation, σ, for the parameters, then adding and subtracting 1.645σ to and from the mean of the parameters.

2.3 b)Bias-variance trade-off and resampling techniques

Implemented in this section:

- OLS plotting training and test MSE's over complexity

- OLS with bootstrap resampling technique
- Bias-variance analysis

This section aimed to evaluate OLS by plotting error in testing and training data together against a varying complexity in the model, evaluate OLS by using bootstrap resampling, and perform a bias-variance trade-off analysis using bootstrap.

Noisefactor is set to 0 in this section to find an error in the calculations for bias and variance.

```
def sample(self, sourceX, sourceY, Nsamples = 0.6):
    if isinstance(Nsamples, float):
        Nsamples = int(Nsamples*sourceX.shape[0])

    sampleArrayX = np.zeros(sourceX[:Nsamples,:].shape)
    sampleArrayY = np.zeros(Nsamples)
    for i in range(Nsamples):
        ind = np.random.randint(Nsamples)
        sampleArrayX[i] = sourceX[ind]
        sampleArrayY[i] = sourceY[ind]

    return sampleArrayX, sampleArrayY
```

```
def bootstrap(self, sampleSize=None, sampleN=None):
    if sampleSize is None: sampleSize = 0.8
    if sampleN is None: sampleN = 5
    self.craftX()
    if isinstance(sampleSize, float): sampleSize = int(sampleSize*len(↩
        self.x))
    r2list, SElist = np.zeros((2, sampleN))
    X_train, X_test, y_train, y_test = train_test_split(self.X, self.↩
        yData)
    for i in range(sampleN):
        X_sample, y_sample = self.sample(X_train, y_train)
        r2list[i], SElist[i] = self.OLS([X_sample, X_test, y_sample, ↩
            y_test])[:2]
    return r2list, SElist
```

```
def bias(self, y_data, y_model):
    return np.mean((y_data - np.mean(y_model, axis=1))**2)

def variance(self, y_model):
    return np.mean(np.var(y_model, axis=1))
```

The bias-variance analysis was implemented by using bootstrap OLS over a range of polynomial complexities and plotting the bias, variance, and the MSE. It similarly plotted the training MSE and test MSE over complexity.

2.4 c) Cross-validation as resampling techniques, adding more complexity

Implemented in this section:

- k-fold cross-validation

This section implements k-fold cross-validation resampling and compares it to bootstrap resampling, using the same evaluation methods of R^2 and MSE.

k-fold cross-validation and bootstrap are compared for equivalent number of folds and number of samples, 5 and 10.

The k-fold crossvalidation was implemented in three functions; kfold_splitter(), kfold_yielder(), and kfold(). kfold_splitter() shuffles and splits the data into several versions of training sets and testing sets, as a 3D-array for the inputs and a 2D-array for the outputs, functioning as collections of 2D-arrays and 1D-arrays. kfold_yielder() then functions as a generator, supplying these sets as appropriate 2- and 1D-arrays to kfold(), which performs OLS on the sets generated in turn.

2.5 d) Ridge Regression on the Franke function with resampling

Implemented in this section:

- Ridge
- Ridge with bootstrap
- Ridge with k-fold cross-validation

This section implements Ridge and applies bootstrap and k-fold cross-validation resampling techniques to it. It was also supposed to implement a bias-variance analysis, but as of this moment, it is only a copy of the function defined in partB.py, and does not do the job it's supposed to.

2.6 e) Lasso Regression on the Franke function with resampling

Implemented in this section:

- Lasso method
- Bias-variance analysis using lasso and bootstrap

This section implements scikit learn's lasso method with no resampling and inserts it into a bias-variance analysis function with bootstrap resampling, similar to what was done in subsection 2.3.

```
def lasso(self, alpha=None):
    if alpha is None: alpha = 5
    clf = linmod.Lasso(alpha=alpha)
    self.craftX()
    X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)
    clf.fit(X_train, y_train)
    ypredict = clf.predict(X_test)
    return self.R2(y_test, ypredict), self.MSE(y_test, ypredict)
```

2.7 f) Introducing real data and preparing the data analysis

Implemented in this section:

- Geographical data

This section imports geographical data from [1]

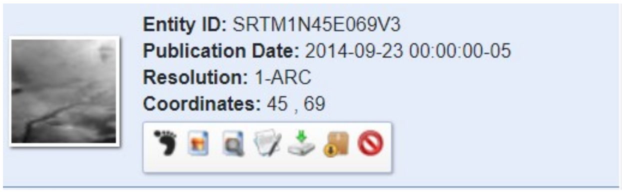

Figure 1: Metadata for the data downloaded.

2.8 g) OLS, Ridge and Lasso regression with resampling

Implemented in this section:

- New data generation

This section was intended to use all methods developed previously on the geographical data, compared and evaluated them against each other as viable methods. The ridge method was to be prepared as a 2D heatmap of polynomial complexity and λ-value. The methods were to be compared to scikit learn's methods to evaluate them as relatively simple implementations against more complex ones. Given the time constraints, the majority of this has not been implemented.

This section, instead, implements geographical data extraction and the train-test MSE over polynomial complexity from section b) over this dataset.

There also exists in the code an unfinished k-fold cross-validation version of the this train-test plot, but this was not completed in time.

3 Results/Discussion

Results (printouts) are described in more detail for each section in the folder outputs/.

3.1 a) Ordinary Least Square (OLS) on the Franke function

With 10,000 datapoints and a polynomial degree of 5, OLS reaches an R^2 value of 0.95 with an MSE of 0.004. If the data generation method is changed to being n^2 points linearly distributed between (0,0), (0,1), (1,0) and (1,1), the R^2 rises to 0.96, indicating that the extra steps taken to vary the dataset actually hinders the quality of the model.

3.2 b) Bias-variance trade-off and resampling techniques

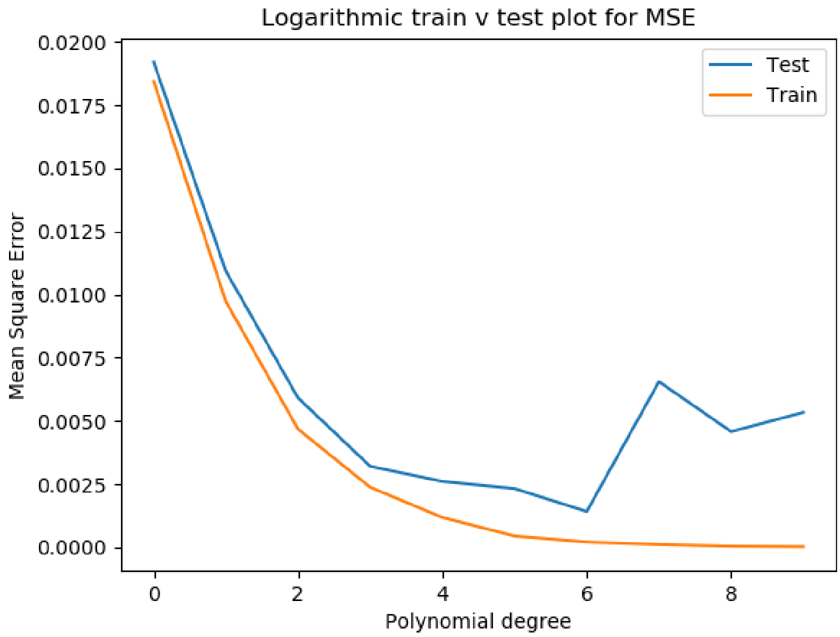

Figure 2: Testing and training errors proceeding as expected over model complexity; training error is continually falling whilst testing data falls to a point and then rises again due to overfitting.

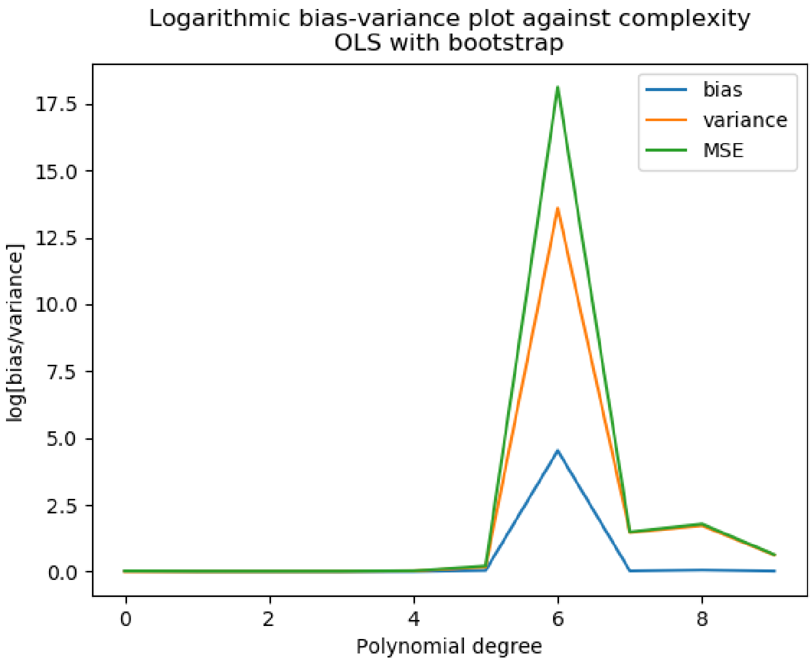

Figure 3: Bias and variance behaving unextpectedly, having a massive spike 17 orders of magnitude above the rest in both bias and variance.

Figure 3 suggests that something has gone wrong during the calculations of bias and variance. Possible reason include:

- A poor model: In addition to the well-behaved Figure 2, my model has been compared to sklearns model (see lines 101-104 of partB.py) and both models exhibit this behaviour.
- Incorrect equations for bias, variance or MSE: All three values are calculated independently, and the relationship MSE = Bias + Variance is found consistently (see last line of printout in outputs/partB.txt or line 114 in partB.py), implying that the calculations are correct.

I have been unable to find any other source of error in the time I have for this project.

3.3 c) Cross-validation as resampling techniques, adding more complexity

```
Folds/number of samples:  5
k-fold          [R2, MSE] : [0.95563823 0.00368468]
Bootstrap       [R2, MSE] : [0.9545114  0.00377724]
kfold/bootstrap [R2, MSE] : [1.00118054 0.9754959 ]

Folds/number of samples:  10
k-fold          [R2, MSE] : [0.95312732 0.00396122]
Bootstrap       [R2, MSE] : [0.9526719  0.00386939]
kfold/bootstrap [R2, MSE] : [1.00047805 1.02373248]
```

Comparing 5-fold cross-validation against 5-sample bootstrap and 10-fold cross-validation against 10-sample bootstrap shows that the R^2 and MSE values are very similar, suggesting that the reasmpling methods have no impact on the actual accuracy of the model.

3.4 d) Ridge Regression on the Franke function with resampling

```
Normal OLS; average values:
OLS, multiple times [R2, MSE] : [0.9533851 0.0038293]
Bootstrap           [R2, MSE] : [0.9545558 0.0038718]
kfold               [R2, MSE] : [0.9540727 0.0037702]

Ridge application; optimal values:
Bootstrap       [R2, MSE] : [0.9524293 0.0039346]
k-fold          [R2, MSE] : [0.9540655 0.0037696]
```

The optimal value for λ is $\lambda = 0$, meaning that for this application, ridge provides no benefit, but it also does not detract from the accuracy in any substantial way.

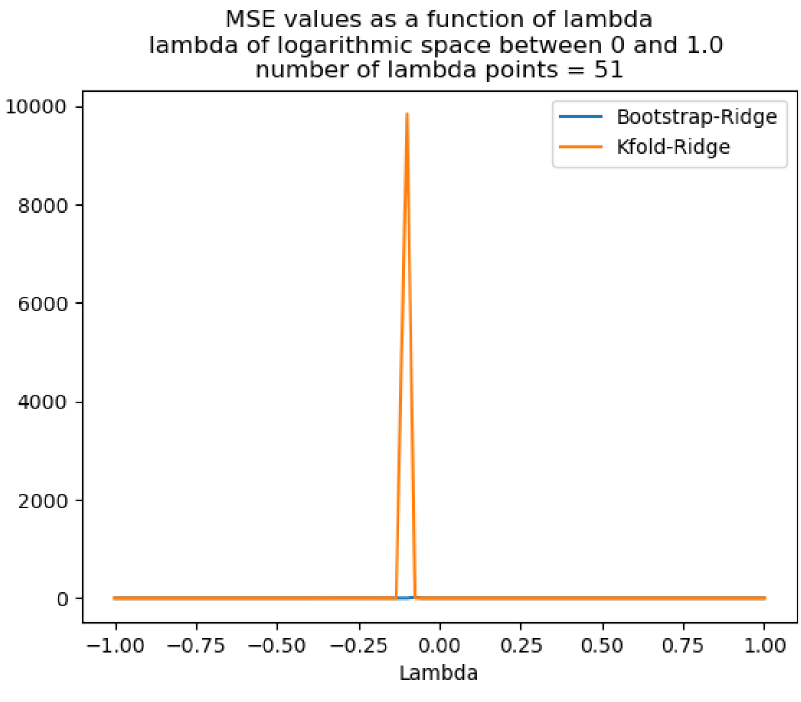

Figure 4: Ridge implemented a variable for MSE, using both bootstrap and k-fold cross-validation resampling methods. There is a big spike on the negative side of 0, possibly implying that the matrix is not invertible in that area, or that the issue seen in Figure 3 may extend to this.

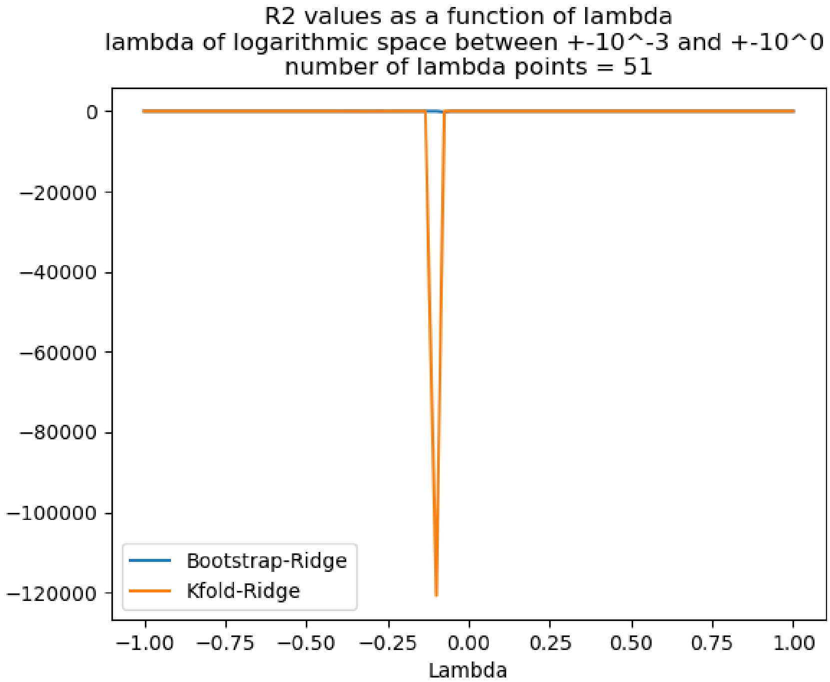

Figure 5: Similar to Figure 4, this figure presents the R^2-evaluation, seeing a big dip on the negative side of 0.

3.5 e) Lasso Regression on the Franke function with resampling

```
Lasso evaluation [R2, MSE] at alpha = 0.01:
  (-0.00040418848759649073, 0.09415771032493848)
Lasso evaluation [R2, MSE] at alpha =  0.1:
  (-0.0008014043714434926, 0.0990173601768499 5)
Lasso evaluation [R2, MSE] at alpha =  0.3:
  (-6.013665053505868e-05, 0.09889216375005189)
Lasso evaluation [R2, MSE] at alpha =    1:
  (-0.00011187832184433866, 0.09252839549202323)
Lasso evaluation [R2, MSE] at alpha =    3:
  (-0.00013499053518417625, 0.09445379258455494)
MSE/(bias+variance) = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

The lasso implementation reaches an R^2 value of -0.001, which is a very poor value, but it also reaches an MSE value down to 0.08, implying that the model might not be bad, but that something else could have gone wrong. This may be a similar issue to what occurs in Figure 3

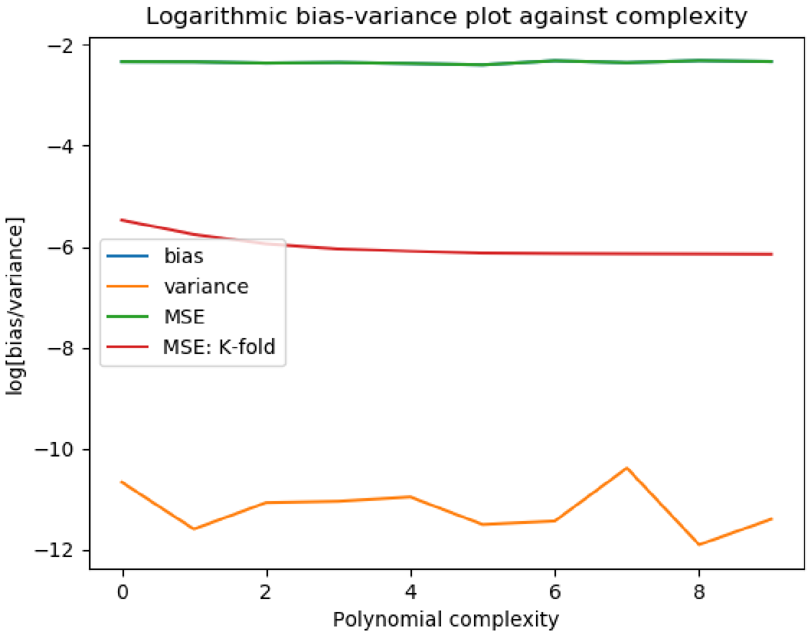

Figure 6: Bias is hidden by the MSE. MSE:K-fold is the MSE calculated during the caluculations, rather than after-the-fact, and the two MSE's disagree, implying that something has gone wrong.

3.6 f) Introducing real data and preparing the data analysis

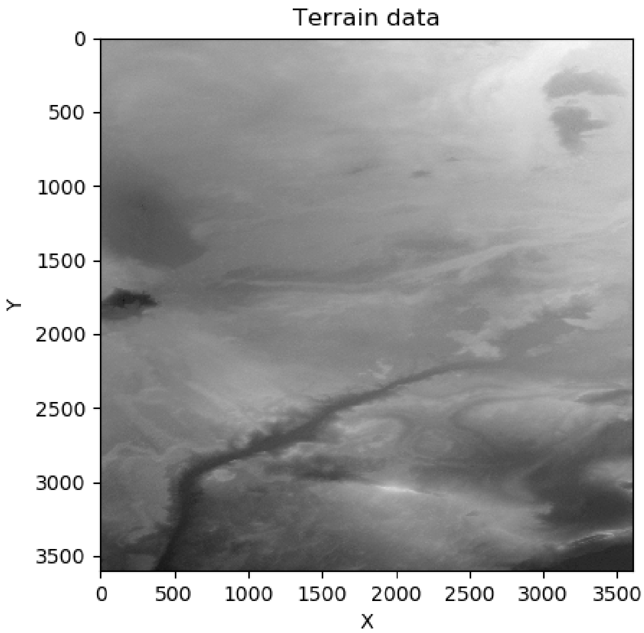

Figure 7: Geographical data, represented on a gray scale.

3.7 g) OLS, Ridge and Lasso regression with resampling

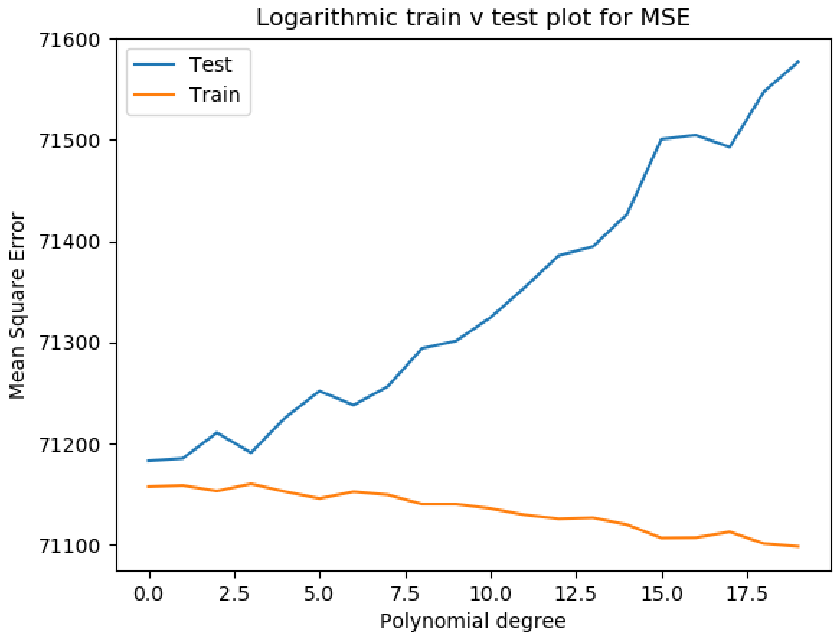

Figure 8

```
R2: ( avg = -126.007750214505 )
    ( max = -125.28996659328048 )
    ( min = -126.63064184509612 )
 [-125.28996659  -126.00075666  -125.99061752
  -126.63064185  -126.33000825  -125.80451042]

MSE: ( avg = 71177.81149110844 )
     ( max =  71218.32821713865 )
     ( min =  71113.55185997034 )
 [71175.41435112  71218.32821714  71167.27618466
  71178.49262646  71213.80570729  71113.55185997]
```

Given the lack of different evaluations, it is difficult to see anything in the data; the poor values could be due to OLS being a poor fit for this data, or it could be a poor implementation. Without additional approaches or different real datasets to compare to, it is difficult to say.

4 Conclusion

This project is riddled with symptoms of being unfinished, and there are few conclusions to be drawn. There are, however, many things to improve on. The initial train-test plot in section b) seems promising, indicating that the base Franke data generation, the basic OLS implementation, and the base evaluation methods are all implemented properly, and fit well with the data. I don't currently have enough information to be able to tell what is wrong with the bias-variance plot in the same section, but given how early it is implemented (although chronologically it was implemented near the end of the project) it should be one of the first places that are looked to for errors and mistakes.

The large spike in the ridge plots in section d) implies that there might be an issue regarding the invertibility of $\mathbf{X}^T\mathbf{X}$, which might lead to very unstable solutions near $\lambda = 0$, which includes the general OLS-solution. This can serve as part of a one-of-many-problems explanation, but is unlikely to be the entire issue, given that the train-test plot and bias-variance plots in section b) behave so differently.

Section c) shows that the resampling methods, bootstrap and k-fold cross-validation, seem to behave well, indicating that they are well implemented and function well for the Franke dataset. Although their aptness for the dataset is derived purely from the base model they use in this instance, OLS, and from the resampling methods themselves, so their fitting the Franke dataset is no surprise given that OLS already fit.

The Lasso implementation, as mentioned in subsection 3.5, has a curious case of a consistently poor R^2 and a consistently good MSE. This is made even more curious by the fact that this is scikit learn's own method implemented, and that it predicts with similar accuracy to a straight average (R^2 score of near 0). Unfortunately, there has not been enough time to look into this curiousity.

The only conclusion that could be drawn about the final section, section g), is that the model used to model it is a poor fit at all complexities used, unfortunately. This section could benefit most from more time, even though I believe that time would have to start at the beginning and root out errors that may propagate throughout the entire project.

Finally, the entire source code needs to have bugs rooted out, functions commented, a greater generalization, a refactoring, and to be finished. Unfortunately, I cannot do this given that I don't have the time to.

5 References

1 https://earthexplorer.usgs.gov/

2 Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer, 2017

6 Appendix

6.1 Source code

6.1.1 a) Ordinary Least Square (OLS) on the Franke function

```python
import numpy as np
from random import random, seed
from sklearn.model_selection import train_test_split
from matplotlib.pyplot import plot, show, legend
import math as mt


# Uses a two-dimensional polynomial of degree p to model Frankes Function
class learningA:
    def __init__(self, n, p, noisefactor=None):
        if noisefactor is None: noisefactor = 0.1
        x1 = y1 = np.linspace(0, 1, mt.ceil(n/2))
        x2 = np.random.uniform(0, 1, mt.floor(n/2))
        y2 = np.random.uniform(0, 1, mt.floor(n/2))

        x = np.concatenate((x1, x2))
        y = np.concatenate((y1, y2))

        x, y = np.meshgrid(x,y)
        self.x = x.flatten()
        self.y = y.flatten()

        self.ytrue = self.FrankeFunction(self.x, self.y)
        self.yData = self.ytrue + noisefactor*np.random.randn(n**2)*self.ytrue.mean()

        self.n = n
        self.p = p
        self.noisefactor = noisefactor
        self.nfeatures = int(((self.p+1)*(self.p+2))/2)

    def MSE(self, y_data, y_model):
        # Optimal value is 0, with higher values beign worse
        return np.sum((y_data-y_model)**2) / np.size(y_model)

    def R2(self, y_data, y_model):
        # Optimal value is 1, with 0 implying that model performs
        # exactly as well as predicting using the data average would.
        # Lower values imply that predicting using the average would be better
        top = np.sum((y_data - y_model)**2)
        bot = np.sum((y_data - np.mean(y_data))**2)
        return 1 - top/bot

    def FrankeFunction(self, x,y):
        term1 = 0.75*np.exp(-(9*x-2)**2/4.00 - 0.25*((9*y-2)**2))
        term2 = 0.75*np.exp(-(9*x+1)**2/49.0 - 0.10*(9*y+1))
        term3 = 0.50*np.exp(-(9*x-7)**2/4.00 - 0.25*((9*y-3)**2))
        term4 =-0.20*np.exp(-(9*x-4)**2 - (9*y-7)**2)
        return term1 + term2 + term3 + term4

    def generateNewDataset(self, n=None, p=None):
        if n is None: n = self.n
        else: self.n = n
        if p is None: p = self.p
        else: self.p = p

        x1 = y1 = np.linspace(0, 1, mt.ceil(n/2), dtype=int)
        x2 = np.random.uniform(0, 1, mt.floor(n/2))
        y2 = np.random.uniform(0, 1, mt.floor(n/2))

        x = np.concatenate((x1, x2))
        y = np.concatenate((y1, y2))
```

```python

        x, y = np.meshgrid(x,y)
        self.x = x.flatten()
        self.y = y.flatten()

        self.ytrue = self.FrankeFunction(self.x, self.y)
        self.yData = self.ytrue + self.noisefactor*np.random.randn(n**2)*self.ytrue.mean()

    def craftX(self, scaling=True):
        self.nfeatures = int(((self.p+1)*(self.p+2))/2)
        self.X = np.zeros((len(self.x), self.nfeatures))

        ind = 0
        for i in range(self.p+1):
            for j in range(self.p+1-i):
                self.X[:,ind] = self.x**i * self.y**j
                ind += 1

        if scaling:
            self.X[:,1:] -= np.mean(self.X[:,1:], axis=0)


    def OLS_core(self, dataset):
        X_train, X_test, y_train = dataset[:3]
        beta = np.linalg.pinv(X_train.T @ X_train) @ X_train.T @ y_train
        ypredict = X_test @ beta
        return ypredict, beta

    # Ordinary Least Square solution
    def OLS(self, dataset=None):
        if dataset is None:
            self.craftX()
            dataset = train_test_split(self.X, self.yData)
            ypredict, beta = self.OLS_core((dataset))
        else:
            ypredict, beta = self.OLS_core(list(dataset))
        return self.R2(dataset[3], ypredict), self.MSE(dataset[3], ypredict), beta

    # Generates R2- and MSE-values from the OLS function
    def OLSeval(self, reps):
        r2list, SElist = np.zeros((2, reps))
        for i in range(reps):
            r2list[i], SElist[i] = self.OLS()[:2]

        return r2list, SElist

    # Calculates confidence interval of the parameters for OLS
    def confidenceIntervalOLS(self, reps=None):
        if reps is None: reps = 300
        betas = np.zeros((reps, self.nfeatures))
        for rep in range(reps):
            betas[rep,:] = self.OLS()[2]

        # Calculating the standard deviation
        betasAvg = np.sum(betas, axis = 0)/reps
        betasDiff = betas - betasAvg

        STD = np.sqrt((1/(reps-self.p-1))*np.sum(betasDiff**2, axis = 0))

        magicNumberFor95PercentConfidenceInterval = 1.645
        ShortHand = magicNumberFor95PercentConfidenceInterval

        confidenceInterval = [betasAvg-ShortHand*STD, betasAvg+ShortHand, STD**2]

        # returns lower and upper boundaries of confidence interval with the variance
        return confidenceInterval


```

```python

if __name__ == "__main__":
    print("""Task as interpreted:
    (x) Generate a dataset as a bipolynomial to a general order
    (x) Implement and execute OLS on bipolynomial and Franke
    (x) Find confidence interval of parameters beta
    (x) Evaluate R2 and mean square error
    (x) Scale and split the data
    """)



    Npoints = 100
    polydegree = 5
    OLSruns = 18

    OLSlearner = learningA(Npoints, polydegree)

    print("Npoints = ", Npoints, ", giving a number of data points = ", Npoints**2, sep="")
    print("Polynomial degree = ", polydegree, ", giving a number of features = ",int(((polydegree+1)*(polydegree+2))/2), sep="")
    print("OLS run", OLSruns, "times")

    for name, result in zip(["R2", "MSE"], OLSlearner.OLSeval(OLSruns)):
        print("\n" + name +
            ": ( avg =", sum(result)/len(result),
            "), ( max = ",  max(result),
            "), ( min = ", min(result),  ")\n",
            result)

    lower, upper, STD = OLSlearner.confidenceIntervalOLS()
    print("""\n95 Confidence Interval for beta:
    Lower: {}\n
    Upper: {}\n
    STD  : {}""".format(lower,upper,STD))


    # Changing the data generation method
    x = np.linspace(0,1,OLSlearner.n)
    y = np.linspace(0,1,OLSlearner.n)
    x, y = np.meshgrid(x,y)
    OLSlearner.x = x.flatten()
    OLSlearner.y = y.flatten()
    OLSlearner.ytrue = OLSlearner.FrankeFunction(OLSlearner.x, OLSlearner.y)
    OLSlearner.yData = OLSlearner.ytrue + OLSlearner.noisefactor*np.random.randn(OLSlearner.n**2)*OLSlearner.ytrue.mean()


    print("\n\nOLS with data determined just by linspace")
    for name, result in zip(["R2", "MSE"], OLSlearner.OLSeval(OLSruns)):
        print("\n" + name +
            ": ( avg =", sum(result)/len(result),
            "), ( max = ",  max(result),
            "), ( min = ", min(result),  ")\n",
            result)
```

6.1.2 b) Bias-variance trade-off and resamplng techniques

```python
from partA import learningA
import numpy as np
from random import random, seed
from sklearn.model_selection import train_test_split
from matplotlib.pyplot import plot, show, legend, title, ylabel, xlabel, savefig

class learningB(learningA):
    def __init__(self, n, p, noisefactor=None):
        super().__init__(n, p, noisefactor)
        self.imageFilePath  = "../outputs/images/partB/"

    def sample(self, sourceX, sourceY, Nsamples=None):
        if Nsamples is None: Nsamples = 0.8
        if isinstance(Nsamples, float):
            Nsamples = int(Nsamples*sourceX.shape[0])

        sampleArrayX = np.zeros(sourceX[:Nsamples,:].shape)
        sampleArrayY = np.zeros(Nsamples)
        for i in range(Nsamples):
            ind = np.random.randint(Nsamples)
            sampleArrayX[i] = sourceX[ind]
            sampleArrayY[i] = sourceY[ind]

        return sampleArrayX, sampleArrayY

    def plotOLS_trainvtest(self, reps, p_range):
        # Made to be similar to Fig. 2.11 of Hastie, Tibshirani, and Friedman
        p_old = self.p
        p_rangeObject = range(p_range[0], p_range[1])
        SElist_train, SElist_test = np.zeros((2,p_range[1] - p_range[0]))

        for i in p_rangeObject:
            self.p = i + p_old
            self.craftX()

            for _ in range(reps):
                X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)
                ypredict_test, beta = self.OLS_core([X_train, X_test, y_train, y_test])
                ypredict_train = X_train @ beta

                SElist_test[i-p_range[0]]  += self.MSE(y_test,  ypredict_test)/reps
                SElist_train[i-p_range[0]] += self.MSE(y_train, ypredict_train)/reps

        self.p = p_old

        imageFileName = "trainTestMSE"
        plot(p_rangeObject, (SElist_test), label="Test")
        plot(p_rangeObject, (SElist_train), label="Train")
        legend()
        title("Logarithmic train v test plot for MSE")
        xlabel("Polynomial degree")
        ylabel("Mean Square Error")
        savefig(self.imageFilePath + imageFileName + ".png")
        show()

    def bootstrap(self, sampleSize=None, sampleN=None):
        if sampleSize is None: sampleSize = 0.8
        if sampleN is None: sampleN = 5

        self.craftX()
        X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)

```

```
        if isinstance(sampleSize, float): sampleSize = int(sampleSize*X_test.shape[0])

        r2list, SElist = np.zeros((2, sampleN))
        for i in range(sampleN):
            X_sample, y_sample = self.sample(X_train, y_train, sampleSize)
            r2list[i], SElist[i] = self.OLS([X_sample, X_test, y_sample, y_test])[:2]

        return r2list, SElist

    def bias(self, y_data, y_model):
        return np.mean((y_data - np.mean(y_model, axis=1))**2)

    def variance(self, y_model):
        return np.mean(np.var(y_model, axis=1))

    def biasVarianceAnalysis_bootstrap(self, p_range, sampleSize=None, sampleN=None):
        if sampleSize is None: sampleSize=0.8
        if isinstance(sampleSize, float): sampleSize = int(sampleSize*len(self.x))
        if sampleN is None: sampleN=5

        p_old = self.p
        p_rangeObject = range(p_range[0], p_range[1])

        bias = np.zeros(len(p_rangeObject))
        variance = bias.copy()
        SElist = bias.copy()

        # Imported to compare to my own model, see line 102
        from sklearn.linear_model import LinearRegression

        for i in p_rangeObject:
            self.p = i + p_old
            self.craftX()

            X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)

            ypredict_models = np.zeros((y_test.shape[0], sampleN))

            for j in range(sampleN):
                X_sample, y_sample = self.sample(X_train, y_train)


                ypredict = self.OLS_core([X_sample, X_test, y_sample, y_test])[0]
                # My OLS method was compared to ScikitLearns method, and was found not to be the issue
                # reg = LinearRegression().fit(X_sample, y_sample)
                # ypredict = reg.predict(X_test)
                # print("HERE WE GO",np.mean(test_var/ypredict))
                ypredict_models[:,j] = ypredict

            variance[i-p_range[0]] = self.variance(ypredict_models)
            # bias[i-p_range[0]] = np.mean((y_test - np.mean(ypredict_models, axis=1))**2)
            bias[i-p_range[0]] = self.bias(y_test, ypredict_models)

            SElist[i-p_range[0]] = np.mean( np.mean((y_test.reshape(-1,1) - ypredict_models)**2, axis=1))

        print("\nMSE divided by (bias+variance):", SElist/(bias+variance))

        self.p = p_old

        imageFileName = "biasVariance"
        filetype = ".png"

        plot(p_rangeObject, (bias), label="bias")
```

```python
        plot(p_rangeObject, (variance), label="variance")
        plot(p_rangeObject, (SElist), label="MSE")

        legend()
        title("Logarithmic bias-variance plot against complexity\nOLS with bootstrap")
        xlabel("Polynomial degree")
        ylabel("log[bias/variance]")
        savefig(self.imageFilePath + imageFileName + filetype)
        show()






if __name__ == "__main__":
    print("""Task as interpreted:
    (?) "general aim: study bias-variance trade-off by implementing bootstrap resampling"
    (x) Implement OLS with resampling techniques (bootstrap)
    (x) Generate a figure similar to Fig. 2.11, Hastie (showing test&training MSE's)
    (x) perform a bias-variance analysis by: Comparing MSE to complexity (MSE v polydegree), make a graph
    (x) Do some equations (mentioned in the introduction part of the report)
    (x) describe bias and variance as part of those equations;
    (x) Discuss said bias and variance trade-off as a function of complexity, # of data points, and possibly also bootstrap
    """)



    Npoints = 15
    polydegree = 2
    sampleN = 100
    polydegree_range = [0,10]
    trainvtest_reps  = 18*10
    noisefactor = 0

    Bootstraplearner = learningB(Npoints, polydegree, noisefactor)

    for name, result in zip(["R2", "MSE"], Bootstraplearner.bootstrap(sampleN = sampleN)):
        print("\n" + name +
            ": ( avg =", sum(result)/len(result),
            "), ( max = ",  max(result),
            "), ( min = ", min(result),  ")\n",
            result)

    sampleN *= 1

    Bootstraplearner.plotOLS_trainvtest(trainvtest_reps, polydegree_range)
    Bootstraplearner.biasVarianceAnalysis_bootstrap(polydegree_range, sampleN=sampleN)
```

6.1.3 c) Cross-validation as resampling techniques, adding more complexity

```
from partB import learningB
from sklearn.model_selection import train_test_split
import numpy as np

class learningC(learningB):

    # I stole the shit out of this from StackOverflow
    # https://stackoverflow.com/questions/4601373/better-way-to-shuffle-two-numpy-arrays-in-unison
    def unison_shuffled_copies(self, a, b):
        assert len(a) == len(b)
        p = np.random.permutation(len(a))
        return a[p], b[p]

    def kfold_splitter(self, k):
        X_dimx, X_dimy = self.X.shape
        X, y = self.unison_shuffled_copies(self.X, self.yData)
        splitsX = np.zeros((k, int(X_dimx/k), X_dimy))
        splitsY = np.zeros((k, int(X_dimx/k)))
        frac = np.floor(X_dimx/k)

        for i in range(k):
            index_0 = int(i*frac)
            index_1 = int((i+1)*frac)
            try:
                splitsX[i,:,:] = X[index_0:index_1,:]
                splitsY[i,:]   = y[index_0:index_1]
            except ValueError:
                raise Exception("kfold function does not split correctly")

        return splitsX, splitsY

    def kfold_yielder(self, k):
        splitsX, splitsY = self.kfold_splitter(k)

        pointsPerSplit = splitsX.shape[1]

        for i in range(k):
            X_test  = np.zeros((pointsPerSplit, self.nfeatures))
            Y_test  = np.zeros(pointsPerSplit)
            X_train = np.zeros(((k-1) * pointsPerSplit, self.nfeatures))
            Y_train = np.zeros((k-1) * pointsPerSplit)

            X_test[:,:] = splitsX[i,:,:]
            Y_test[:]   = splitsY[i,:]

            flattenedX = splitsX.reshape(-1, splitsX.shape[-1])
            flattenedY = splitsY.reshape(-1)

            X_train[:i*pointsPerSplit,:] = flattenedX[:i*pointsPerSplit,:]
            X_train[i*pointsPerSplit:,:] = flattenedX[(i+1)*pointsPerSplit:,:]

            Y_train[:i*pointsPerSplit] = flattenedY[:i*pointsPerSplit]
            Y_train[i*pointsPerSplit:] = flattenedY[(i+1)*pointsPerSplit:]

            yield X_train, X_test, Y_train, Y_test

    def kfold(self, k, solver=None):
        self.craftX()
        dataset = self.kfold_yielder(k)
        r2list, SElist = np.zeros((2, k))
        for i in range(k):
            data = next(dataset)
```

```
            R2, MSE = self.OLS(data)[:2]
            r2list[i]  = R2
            SElist[i] = MSE
        return r2list, SElist







if __name__ == "__main__":
    print("""Task as interpreted:
    (X) Implement k-fold cross-validation, evaluate MSE from this
    (X) Compare MSE from k-fold and bootstrap
    (X) try 5-10 folds
    """)

    Npoints = 100
    polydegree = 5
    # folds = 10
    for folds in [5,10]:
        sampleN = folds
        kfoldlearner = learningC(Npoints, polydegree)
        kfoldResults = np.asarray(kfoldlearner.kfold(folds)).sum(axis=1)/folds
        print("\nFolds/number of samples: ", folds)
        print("k-fold           [R2, MSE] :", kfoldResults)

        bootstrapResults = np.asarray(kfoldlearner.bootstrap(sampleN =sampleN)).sum(axis=1)/sampleN
        print("Bootstrap        [R2, MSE] :", bootstrapResults)

        print("kfold/bootstrap [R2, MSE] :", kfoldResults/bootstrapResults)
```

6.1.4 d) Ridge Regression on the Franke function with resampling

```python
from partC import learningC
from sklearn.model_selection import train_test_split
import numpy as np
from matplotlib.pyplot import plot, show, legend, title, ylabel, xlabel, figure, savefig


class learningD(learningC):
    def __init__(self, n, p, noisefactor=None):
        super().__init__(n, p, noisefactor)
        self.imageFilePath  = "../outputs/images/partD/"

    def ridge_core(self, dataset):
        X_train, X_test, y_train, _, lamb = dataset
        beta = np.linalg.pinv(X_train.T @ X_train + lamb*np.identity(self.nfeatures)) @ X_train.T @ y_train
        ypredict = X_test @ beta
        return ypredict, beta

    def ridge(self, dataset=None):
        if dataset is None:
            self.craftX()
            dataset = train_test_split(self.X, self.yData)
            ypredict, beta = self.ridge_core((dataset))
        else:
            ypredict, beta = self.ridge_core(list(dataset))
        return self.R2(dataset[3], ypredict), self.MSE(dataset[3], ypredict), beta

    def bootstrapRidge(self, lambSpace, sampleSize=0.8, sampleN=5):
        lambN = len(lambSpace)
        self.craftX()
        if isinstance(sampleSize, float):
            sampleSize = int(sampleSize*len(self.x))
        r2list, SElist = np.zeros((2, lambN))

        X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)

        for lamb in range(lambN):
            for _ in range(sampleN):
                X_sample, y_sample = self.sample(X_train, y_train)

                r2, SE = self.ridge([X_sample, X_test, y_sample, y_test, lambSpace[lamb]])[:2]
                r2list[lamb] += r2/sampleN
                SElist[lamb] += SE/sampleN

        return r2list, SElist

    def kfoldRidge(self, lambSpace, k, solver=None):
        self.craftX()
        lambN = len(lambSpace)
        R2avg, MSEavg = np.zeros((2, lambN))
        for lamb in range(lambN):
            dataset = self.kfold_yielder(k)
            for data in dataset:
                R2, MSE = self.ridge(list(data) + [lambSpace[lamb]])[:2]
                R2avg[lamb]  += R2/k
                MSEavg[lamb] += MSE/k
        return R2avg, MSEavg

    def biasVarianceAnalysis_ridge(self, p_range, sampleSize=None, sampleN=None):
        #TODO: CURRENTLY JUST A COPY OF THE ORIGINAL FUNCTION FROM b), STILL NEEDS TO BE ADAPTED FOR RIDGE
        p_old = self.p
        p_rangeObject = range(p_range[0], p_range[1])
        if sampleSize is None: sampleSize=0.8
```

```
        if isinstance(sampleSize, float): sampleSize = int(sampleSize*len(self.x))
        if sampleN is None: sampleN=5

        bias = np.zeros(len(p_rangeObject))
        variance = bias.copy()
        SElist = bias.copy()

        for i in p_rangeObject:
            self.p = i + p_old
            self.craftX()

            X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)

            ypredict_models = np.zeros((y_test.shape[0], sampleN))

            for j in range(sampleN):
                X_sample, y_sample = self.sample(X_train, y_train)

                ypredict = self.OLS_core([X_sample, X_test, y_sample, y_test])[0]

                ypredict_models[:,j] = ypredict

            test_var = np.var(ypredict_models, axis=1)

            print(test_var, len(test_var), max(test_var), np.mean(test_var), sep="\n", end="\n----------------------------------------\n")
            variance[i-p_range[0]] = self.variance(ypredict_models)
            bias[i-p_range[0]] = np.mean((y_test - np.mean(ypredict_models, axis=1))**2)

            SElist[i-p_range[0]] = np.mean( np.mean((y_test.reshape(-1,1) - ypredict_models)**2, axis=1))

        self.p = p_old

        plot(p_rangeObject, np.log(bias), label="bias")
        plot(p_rangeObject, np.log(variance), label="variance")
        plot(p_rangeObject, np.log(SElist), label="MSE")

        legend()
        title("Logarithmic bias-variance plot against complexity")
        xlabel("Polynomial complexity")
        ylabel("log[bias/variance]")
        show()

if __name__ == "__main__":
    print("""Task as interpreted
    (x) Implement ridge method with kfold and bootstrap
        (-) Change bootstrap and kfold to take method as argument
            Too much fixing required
        (x) Create bootstrapRidge & kfoldRidge
    (x) Compare ridge-Bootstrap, ridge-kfold, Bootstrap, and kfold
    ( ) Study bias-variance trade-off of various values of lambda using bootstrap.
    ( ) Comment on the above
    """)

    Npoints = 100
    polydegree = 5
    lambN = 50
    logLow = -3
    logHigh = 0

    ridgelearner = learningD(Npoints, polydegree)
    lambSpace = np.logspace(logLow, logHigh, num=int(lambN/2))
    lambSpace = np.concatenate((np.flip(-lambSpace), [0], lambSpace))

    folds = sampleN = OLSruns = 5

    print(" "*35 + "*** NORMAL OLS: ***")
```

```python
    for name, result in zip(["OLS, multiple times", "bootstrap", "kfold"],
                            [ridgelearner.OLSeval(OLSruns),
                             ridgelearner.bootstrap(sampleN=sampleN),
                             ridgelearner.kfold(folds)]):
        print("Method used: ", name)
        for name, result in zip(["R2", "MSE"], result):
            print("\n" + name +
                ": ( avg =", sum(result)/len(result),
                "), ( max = ",  max(result),
                "), ( min = ", min(result),  ")\n",
                result)
        print("--------------------"*4+"\n")

    print(" "*35 + "*** RIDGE: ***")
    for name, result in zip(["BRidge", "KRidge"],
                            [ridgelearner.bootstrapRidge(lambSpace, sampleN=sampleN),
                             ridgelearner.kfoldRidge(lambSpace, folds)]):
        print("Method used: ", name)
        for name, result in zip(["R2", "MSE"], result):
            print("\n" + name +
                ": ( avg =", sum(result)/len(result),
                "), ( max = ",  max(result),
                "), ( min = ", min(result),  ")\n",
                result)
        print("--------------------"*4+"\n")

    for name, result in zip(["Bootstrap-Ridge", "Kfold-Ridge"],
                            [ridgelearner.bootstrapRidge(lambSpace, sampleN=sampleN),
                             ridgelearner.kfoldRidge(lambSpace, folds)]):
        # print("Method used: ", name)
        # for i in range(len(lambSpace)):
        #     print("lambda =", lambSpace[i], "R2: ", result[0][i], "MSE: ", result[1][i])
        # print("--------------------"*4)
        figure(1)
        plot(lambSpace, result[0], label=name)
        xlabel("Lambda")
        title("R2 values as a function of lambda\nlambda of logarithmic space between +-10^" + str(logLow) +
          " and +-10^" + str(logHigh) + " \nnumber of lambda points = " + str(len(lambSpace)))
        legend()
        figure(2)
        plot(lambSpace, result[1], label=name)
        title("MSE values as a function of lambda\nlambda of logarithmic space between " + "0" +
          " and " + str(lambSpace[-1]) + " \nnumber of lambda points = " + str(len(lambSpace)))
        xlabel("Lambda")
        legend()


        # show(block=False)
    imageFileNameMSE = "MSERidge"
    imageFileNameR2 = "R2Ridge"
    filetype = ".png"
    figure(1)
    savefig(ridgelearner.imageFilePath + imageFileNameR2 + filetype)
    figure(2)
    savefig(ridgelearner.imageFilePath + imageFileNameMSE + filetype)
    show()

   # ridgelearner.biasVarianceAnalysis_ridge([0,10])
```

6.1.5 e) Lasso Regression on the Franke function with resampling

```
from partD import learningD
from sklearn.model_selection import train_test_split
import sklearn.linear_model as linmod
import numpy as np
from matplotlib.pyplot import plot, show, legend, title, ylabel, xlabel, figure, savefig

class learningE(learningD):
    def __init__(self, n, p, noisefactor=None):
        super().__init__(n, p, noisefactor)
        self.imageFilePath  = "../outputs/images/partE/"


    def lasso(self, alpha=None):
        if alpha is None: alpha = 5
        clf = linmod.Lasso(alpha=alpha)
        self.craftX()
        X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)
        clf.fit(X_train, y_train)
        ypredict = clf.predict(X_test)
        return self.R2(y_test, ypredict), self.MSE(y_test, ypredict)

    def biasVarianceAnalysis_lasso_bootstrap(self, p_range, sampleSize=None, sampleN=None):
        p_old = self.p
        p_rangeObject = range(p_range[0], p_range[1])
        if sampleSize is None: sampleSize=0.8
        if isinstance(sampleSize, float): sampleSize = int(sampleSize*len(self.x))
        if sampleN is None: sampleN=5

        bias = np.zeros(len(p_rangeObject))
        variance = bias.copy()
        SElist = bias.copy()
        SEkfold = bias.copy()
        clf = linmod.Lasso(alpha=0.1)

        for i in p_rangeObject:
            self.p = i + p_old
            self.craftX()

            X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)

            ypredict_models = np.zeros((y_test.shape[0], sampleN))

            for j in range(sampleN):
                X_sample, y_sample = self.sample(X_train, y_train)

                clf.fit(X_sample, y_sample)
                ypredict = clf.predict(X_test)

                ypredict_models[:,j] = ypredict

            variance[i-p_range[0]] = self.variance(ypredict_models)
            bias[i-p_range[0]] = self.bias(y_test, ypredict_models)
            SElist[i-p_range[0]] = np.mean( np.mean((y_test.reshape(-1,1) - ypredict_models)**2, axis=1))
            SEkfold[i-p_range[0]] = np.mean(self.kfold(sampleN)[1])

        self.p = p_old
        filetype = ".png"
        imageFileName = "lassoBiasVariance"

        print("MSE/(bias+variance) =", SElist/(bias+variance))

        plot(p_rangeObject, np.log(bias), label="bias")
        plot(p_rangeObject, np.log(variance), label="variance")
        plot(p_rangeObject, np.log(SElist), label="MSE")
```

```
        plot(p_rangeObject, np.log(SEkfold), label="MSE: K-fold")

        legend()
        title("Logarithmic bias-variance plot against complexity")
        xlabel("Polynomial complexity")
        ylabel("log[bias/variance]")
        savefig(self.imageFilePath + imageFileName + filetype)
        show()










if __name__ == "__main__":
    print("""Task as interpreted:
    (x) Implement lasso from scikit learn
    ( ) Discuss the three methods (assume Lasso, Ridge, and Bootstrap (or OLS?)), and which fits the best
    ( ) Bias-variance trade-off analysis using bootstrap and kfold
    """)

    Npoints = 100
    polydegree = 5
    polydegree_range = [0,10]
    sampleN = 18
    alphas = [0.01,0.1,0.3,1,3]

    Lassolearner = learningE(Npoints, polydegree)
    for alph in alphas:
        print("Lasso evaluation [R2, MSE] at alpha = {:4}:".format(alph), Lassolearner.lasso())
    Lassolearner.biasVarianceAnalysis_lasso_bootstrap(polydegree_range, sampleN=sampleN)
```

6.1.6 f) Introducing real data and preparing the data analysis

```python
import numpy as np
from imageio import imread
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure, title, imshow, xlabel, ylabel, show, savefig
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from partE import learningE

class learningF(learningE):
    def __init__(self, n, p, noisefactor=None):
        super().__init__(n, p, noisefactor)
        self.imageFilePath  = "geodata/"


    def loadTerrainData(self, filename=None):
        if filename is None: filename = "geodata/n45_e069_1arc_v3.tif"
        self.terrainData = imread(filename)

if __name__ == "__main__":
    print("""Task as interpreted:
    (x) download data and manage to import it
    """)

    Npoints = 2
    polydegree = 0
    # Datapoints moot

    terrainlearner = learningF(Npoints, polydegree)
    terrainlearner.loadTerrainData()

    # Show the terrain
    imageFileName = "geodata_area"
    filetype = ".png"
    figure()
    title("Terrain data")
    imshow(terrainlearner.terrainData, cmap="gray")
    xlabel("X")
    ylabel("Y")
    savefig(terrainlearner.imageFilePath + imageFileName + filetype)
    show()
```

6.1.7 g) OLS, Ridge and Lasso regression with resampling

```python
from partF import learningF
from sklearn.model_selection import train_test_split
import numpy as np
import math as mt
from matplotlib.pyplot import plot, figure, title, imshow, xlabel, ylabel, show, savefig, legend


class learningG(learningF):
    def __init__(self, nFrac, p):
        self.loadTerrainData()
        nMax = self.terrainData.shape[0]
        n = int((nMax)*nFrac)

        self.p = p
        self.nfeatures = int(((self.p+1)*(self.p+2))/2)

        self.imageFilePath  = "../outputs/images/partG/"

        if nFrac < 0.5:
            # Includes both random and regular data points
            x1 = y1 = np.linspace(0, nMax-1, mt.ceil(n/2), dtype=int)
            x2 = np.random.randint(0, nMax, mt.floor(n/2))
            y2 = np.random.randint(0, nMax, mt.floor(n/2))

            x = np.concatenate((x1, x2))
            y = np.concatenate((y1, y2))

        else:
            x = y = np.linspace(0, nMax-1, n, dtype=int)

        x, y = np.meshgrid(x,y)
        self.x = x.flatten()
        self.y = y.flatten()
        self.yData = self.terrainData[self.x,self.y]

    def plotkfold_trainvtest(self, k, p_range):
        p_old = self.p
        p_rangeObject = range(p_range[0], p_range[1])
        SElist_train, SElist_test = np.zeros((2,p_range[1] - p_range[0]))

        for i in p_rangeObject:
            self.p = i + p_old
            self.craftX()
            dataset = self.kfold_yielder(k)
            for i in range(k):
                data = next(dataset)
                ypredict = self.OLS_core(data)[0]


            # X_train, X_test, y_train, y_test = train_test_split(self.X, self.yData)
            # ypredict_test, beta = self.OLS_core([X_train, X_test, y_train, y_test])
            # ypredict_train = X_train @ beta

            SElist_test[i-p_range[0]]  += self.MSE(y_test, ypredict_test)/k
            SElist_train[i-p_range[0]] += self.MSE(y_train, ypredict_train)/k

        self.p = p_old

        plot(p_rangeObject, np.log(SElist_test), label="Test")
        plot(p_rangeObject, np.log(SElist_train), label="Train")
        legend()
        title("Logarithmic train v test plot for MSE")
        xlabel("Polynomial complexity")
        ylabel("Mean Square Error")
```

```
        show()


if __name__ == "__main__":
    print("""Tasks as interpreted:
    (x) Parametrize terrain data
    ( ) Apply all three models of [see below] to geographical data
        ( ) OLS (k-fold)
        ( ) Ridge (k-fold)
        ( ) Lasso (k-fold)
    ( ) Critically evaluate results and discuss "the applicabilty of these regression methods to the type of data presented here"
    """)

    polydegree = 5
    datafrac = 0.1
    OLSruns = 6

    finallearner = learningG(datafrac, polydegree)
    # finallearner.OLSeval(OLSruns)

    for name, result in zip(["R2", "MSE"], finallearner.OLSeval(OLSruns)):
        print("\n" + name +
            ": ( avg =", sum(result)/len(result),
            "), ( max = ",  max(result),
            "), ( min = ", min(result),  ")\n",
            result)

    finallearner.plotOLS_trainvtest(OLSruns, [0,20])
```