Project 1 FYS-STK3155

Lars Johan Brodtkorb October 8, 2018

Contents

Abstract

In this exercise I have tried to figure out which method is to be preferred of ordinary least squares, Ridge and Lasso regressions. The results show a better prediction error for the ridge regression, but that may be due to a faulty implementation of the Lasso method.

1 Introduction

The main aim of this project is to study in more detail various regression methods, including the Ordinary Least Squares (OLS) method, Ridge regression and finally Lasso regression. The methods are in turn combined with resampling techniques.

We will first study how to fit polynomials to a specific two-dimensional function called Franke's function. This is a function which has been widely used when testing various interpolation and fitting algorithms. Furthermore, after having etsablished the model and the method, we will employ resampling techniques such as the cross-validation and/or the bootstrap methods, in order to perform a proper assessment of our models.

The Franke function, which is a weighted sum of four exponentials reads as follows

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

The function will be defined for $x, y \in [0,1]$. Our first step will be to perform an OLS regression analysis of this function, trying out a polynomial fit with an x and y dependence of the form $[x, y, x^2, y^2, xy, \dots]$. We will also include cross-validation and bootstrap as resampling techniques. As in homeworks 1 and 2, we can use a uniform distribution to set up the arrays of values for x and y, or as in the example below just a fix values for x and y with a given step size. In this case we will have two predictors and need to fit a function (for example a polynomial) of x and y. Thereafter we will repeat much of the same procedure using the the Ridge and Lasso regression methods, introducing thus a dependence on the bias (penalty) λ.

Thereafter we are going to use (real) digital terrain data and try to reproduce these data using the same methods. We will also try to go beyond the second-order polynomials metioned above and explore which polynomial fits the data best.

2 Methods

2.1 Measures of prediction accuracy

Find the confidence intervals of the β parameters by computing their variances, evaluate the Mean Squared error (MSE)

$$MSE(\hat{y}, \hat{\tilde{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2,$$

and the R^2 score function. If $\tilde{y}_i$ is the predicted value of the $i-th$ sample and y_i is the corresponding true value, then the score R^2 is defined as

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

where we have defined the mean value of $\hat{y}$ as

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

2.2 Ordinary least square regression

We have an input vector $X^T = (X_1, X_2, X_3, ..., X_p)$ and want to predict a real-valued output Y. The linear regression model has the form: ([2], chapter 3.2)

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j\beta_j \tag{1}$$

The linear model either assumes that the regression function $E(Y|X)$ is linear, or that the linear model is a reasonable approximation. The β_j s are unknown parameters or coefficients, and the variables X_j can come from many different sources. One of them is that of basis expansions such as

$X_2 = X_1^2, X_3 = X_1^3$, leading to a polynomial representation;

We can use the basis expansions as a device to achieve more flexible representations for f(X). Polynomials are an example of this. Having great flexibility in approximation allows the basis expansions to work for many occurences, but their variability may increase heavily if they are applied outside their intended scope. The coefficients to achieve a functional form in one region can cause the function to flap about madly in remote regions ([2], chapter 5.1, p. 140).

The polynomial method produces a dictionary D consisting of typically a very large number of basis functions $|D|$, far more than we can afford to fit to our data. Along with the dictionary we require a method for controlling the complexity of our model, using basis functions from the dictionary. There are three common approaches: Restriction, selection and regularization methods, which we will revisit in the Ridge and Lasso segment.

No matter the source of the X_j, the model is linear in the parameters. Typically we have a set of training data $(x_1, y_1)...(x_N, y_N)$ from which to estimate the parameters β. Each $x_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$ is a vector of feature measurements for the i-th case.

Using linear least squared fittings for $X \in \mathbb{R}^2$ we can get the linear function of X that minimizes the sum of the sum of squared residuals of Y.

A unique solution for β has the form:

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

We have made minimal assumptions about the true distribution of the data. In order to pin down the sampling properties of $\hat{\beta}$, we now assume that the observations y_i are uncorrelated and have constant variance σ^2, and that the x_i are fixed (non random). The matrix of the least squares parameter estimates is then given by:

$$Var(\hat{\beta}) = (\mathbf{X}^T\mathbf{X})^{-1}\sigma^2$$

2.3 K-fold cross-validation algorithm

Cross-validation is probably the simplest and most widely used method for estimating prediction error. This method directly estimates the expected extra-sample error $Err = E[L(Y, f(\hat{X}))]$, the average generalization error when the method $f(\hat{X})$ is applied to an independent test sample from the joint distribution of X and Y. As mentioned earlier, we might hope that cross-validation estimates the conditional error, with the training set T held fixed. Unfortunately, cross-validation typically estimates well only the expected prediction error ([2], chapter 7.10 - 7.12).

K-fold crossvalidation uses part of the available data to fit the model, and a different part to test it. We split the data into K roughly equal-sized parts. For the kth part, we fit the model to the other $K - 1$ parts of the data, and calculate the prediction error of the fitted model when predicting the kth part of the data. We do this for k = 1, 2,...,K and combine the K estimates of prediction error.

Given a set of models $f(x, \alpha)$ indexed by a tuning parameter α, denote $\hat{f}^{-\kappa(x,\alpha)}$ by the α-th model fitted with the k-th part of the data removed. Then for this set of models we define:

$$CV(\hat{f}, \alpha) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{f}^{-\kappa(i)}(x_i, \alpha)) \tag{2}$$

where L is the loss function, κ is the removed data.

The function $CV(\hat{f}, \alpha)$ provides an estimate of the test error curve, and we find the tuning parameter $\hat{\alpha}$ that minimizes it. Our final chosen model is $f(x, \hat{\alpha})$, which we then fit to all the data.

The steps of the cross-validation method:

(a) Find a subset of good predictors that show fairly strong (univariate) correlation with the class labels, using all of the samples except those in fold k.

(b) Using just this subset of predictors, build a multivariate classifier, using all of the samples except those in fold k.

(c) Use the classifier to predict the class labels for the samples in fold k.

2.4 Bootstrap

The bootstrap method has been used in this project to estimate the variance and bias. As with cross-validation, the bootstrap seeks to estimate the conditional error, but typically estimates well only the expected prediction error.

Suppose we have a model fit to a set of training data. We denote the training set by $Z = (z_1, z_2, ..., z_N)$ where $z_i = (x_i, y_i)$. The basic idea is to randomly draw datasets with replacement from the training data, each sample the same size as the original training set. This is done B times (B = 200 in my application), producing B bootstrap datasets. Then we refit the model to each of the bootstrap datasets, and examine the behavior of the fits over the B replications ([2] p.249)

2.5 Ridge and Lasso regression

Ridge regression

Ridge regression is a simple example of a regularization approach, while the lasso is both a regularization and selection method.

Selection methods adaptively scan the dictionary and include only those basis functions h_m that contribute significantly to the fit of the model.

Regularization methods use the entire dictionary but restrict the coefficients. ([2], chapter 5.1 p.141)

The method used follows [3] quite closely. In chapter 1.4.2 on page 8 in [3] we find an estimator for β and variance in the ridge regression:

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

$$Var(\hat{\beta}) = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\sigma^2$$

Ridge regression shrinks the regression coefficients by imposing a penalty on their size. The ridge coefficients minimize a penalized residual sum of squares.

$$\hat{\beta}_{ridge} = \arg\min_{\beta}(\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2) \tag{3}$$

Here $\lambda \geq 0$ is a complexity parameter that controls the amount of shrinkage. The idea of penalizing by the sum-of-squares of the parameters is also used in neural networks, where it is known as weight decay ([2] p. 63)

Lasso regression

The Lasso method is a shrinking method like Ridge. Just as in ridge regression, we can re-parametrize the constant β_0 by standardizing the predictors; the solution for $\hat{\beta_0}$ is $\hat{y}$, and thereafter we fit a model without an intercept. In the signal processing literature, the Lasso is also known as basis pursuit (Chen et al., 1998). We can also write the Lasso problem in the equivalent Lagrangian form:

$$\hat{\beta}_{lasso} = \arg\min_{\beta}(\frac{1}{2}\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|) \tag{4}$$

The difference from the ridge method is that the ridge penalty $\sum_{j=1}^{p}\beta_j^2$ is replaced with the Lasso penalty $\sum_{j=1}^{p}|\beta_j|$

3 Code implementation

For most of the code implementation see attachments for all the code, but I will include the bootstrap method with comments in the report.

3.1 Bootstrap

The bootstrap method was implemented with the following code, noting steps in the comments to the program.

```
def bootstrap_bias_variance(x, y, model, n_bootstraps=200, test_size=0.2):
    # Hold out some test data that is never used in training.
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size)

    # The following (m x n_bootstraps) matrix holds the column vectors y_pred
    # for each bootstrap iteration.
    y_pred = np.empty((y_test.shape[0], n_bootstraps))
    for i in range(n_bootstraps):
        x_, y_ = resample(x_train, y_train)

        # Evaluate the new model on the same test data each time.
        y_pred[:, i] = model.fit(x_, y_).predict(x_test).ravel()

    # Note: Expectations and variances taken w.r.t. different training
    # data sets, hence the axis=1. Subsequent means are taken across the test data
    # set in order to obtain a total value, but before this we have average_error/average_bia
    # calculated per data point in the test set.
    # Note 2: The use of keepdims=True is important in the calculation of average_bias_square
    # maintains the column vector form. Dropping this yields very unexpected results.


    mean_y = np.mean(y_test)

    errors = np.mean((y_test - y_pred) ** 2, axis=1, keepdims=True)
    bias_sqares = (y_test - np.mean(y_pred, axis=1, keepdims=True)) ** 2
    variances = np.var(y_pred, axis=1, keepdims=True)

    r2 =  1.0 - np.mean(errors)/mse(y_test, mean_y)
    error = np.mean( errors )
    bias_squared = np.mean(bias_sqares)
    variance = np.mean( variances )

        return error, bias_squared, variance, r2
```

4 Analysis

4.1 Results

4.1.1 OLS

Below are figures from the ordinary least squares method:

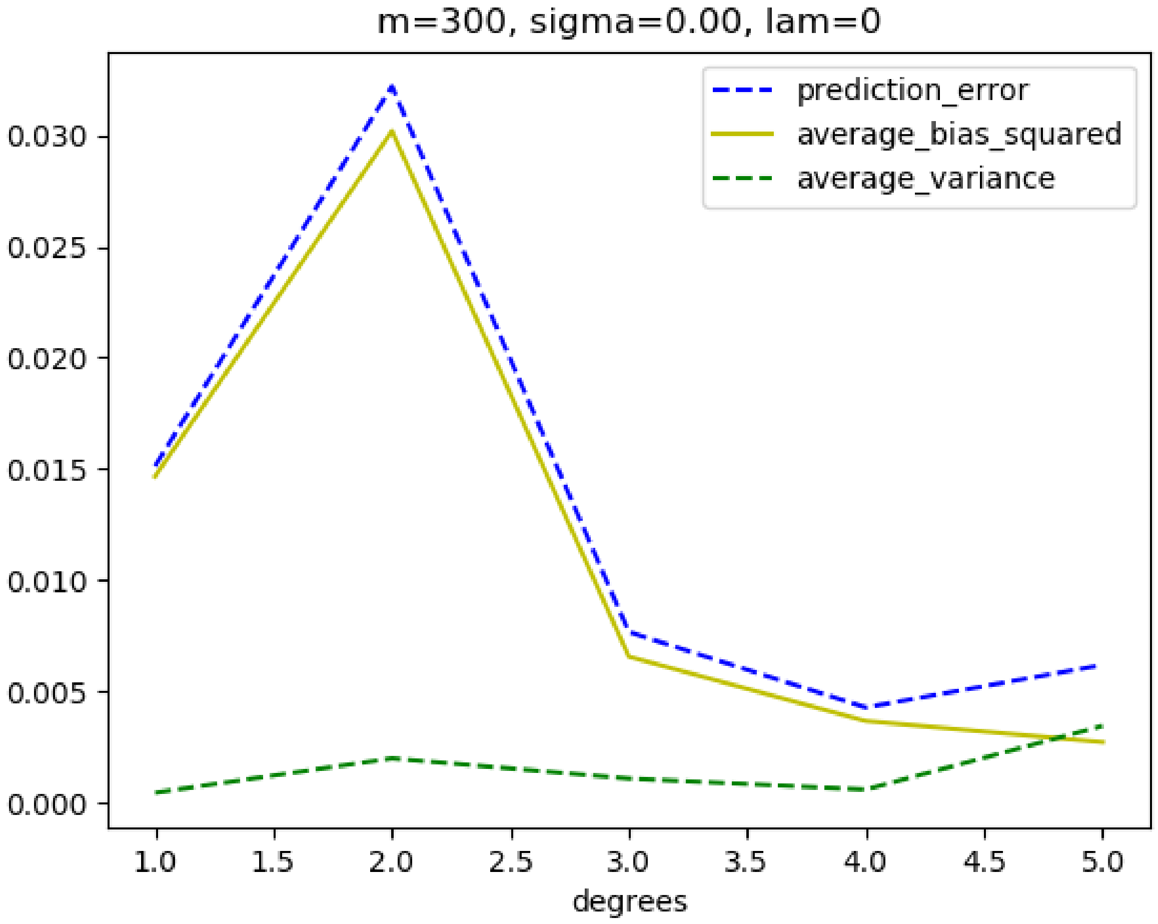

Figure 1: This shows the bias-variance tradeoff as function of the maximum polynomial degree used in the OLS method, with a sample size of 300.

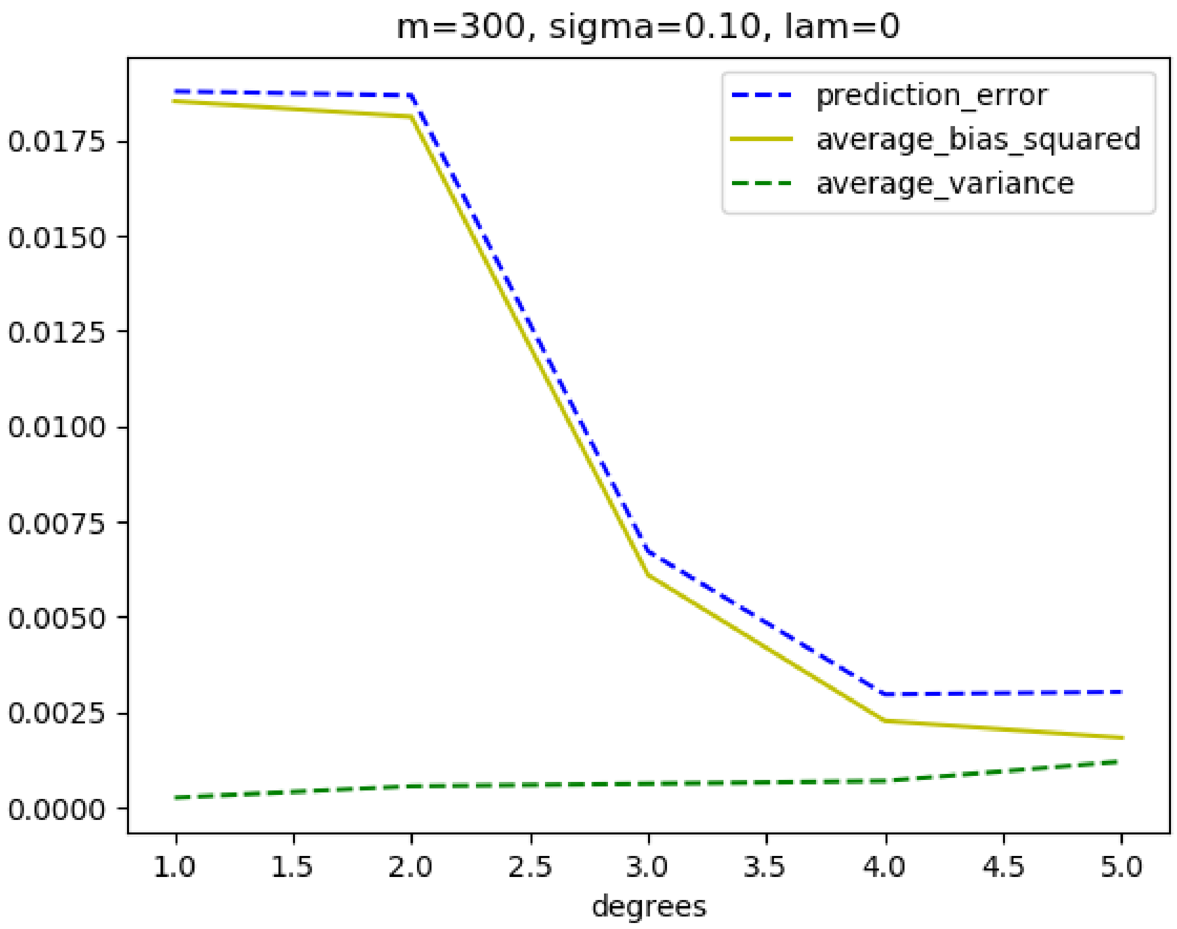

Figure 2: This shows the bias-variance tradeoff as function of the maximum polynomial degree used in the OLS method, with a sample size of 300 and a random gaussian noise with standard deviation of 0.1.

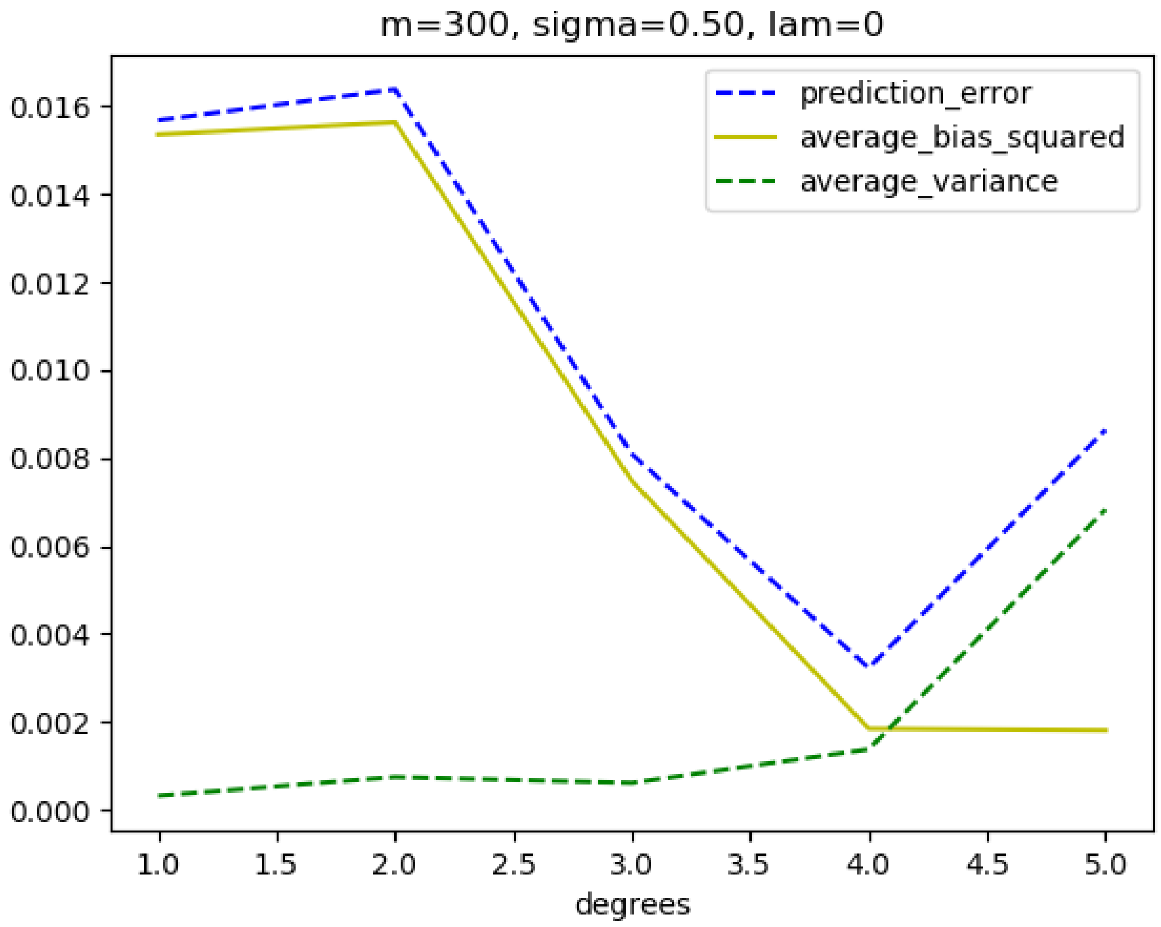

Figure 3: This shows the bias-variance tradeoff as function of the maximum polynomial degree used in the OLS method, with a sample size of 300 and a random gaussian noise with standard deviation of 0.5.

4.1.2 Ridge

Below are the figures for varying sample sizes, degree of polynomials and penalty with Ridge.

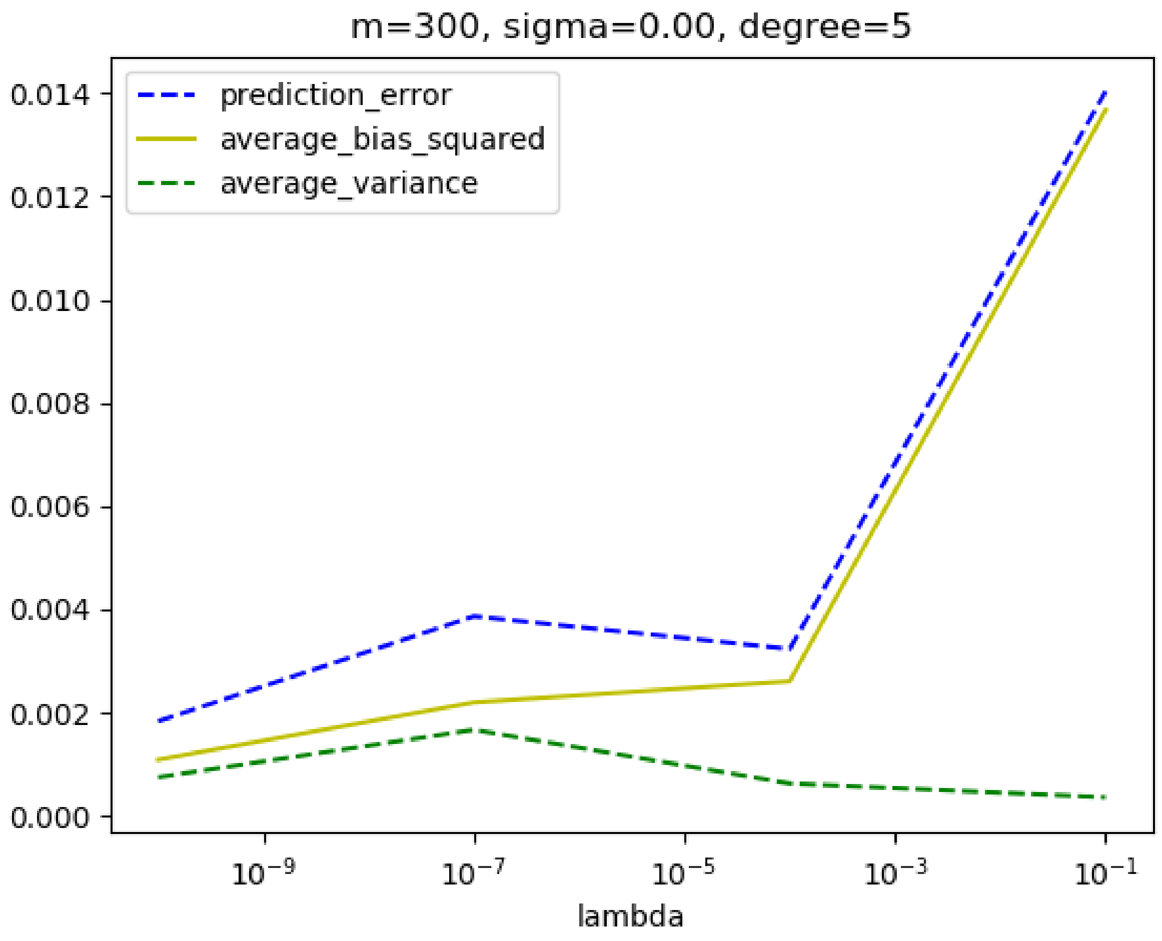

Figure 4: This shows the bias-variance tradeoff as function of the regularization parameter, lambda,for Ridge regression with a sample size of 300 and a polynomial degree of 5

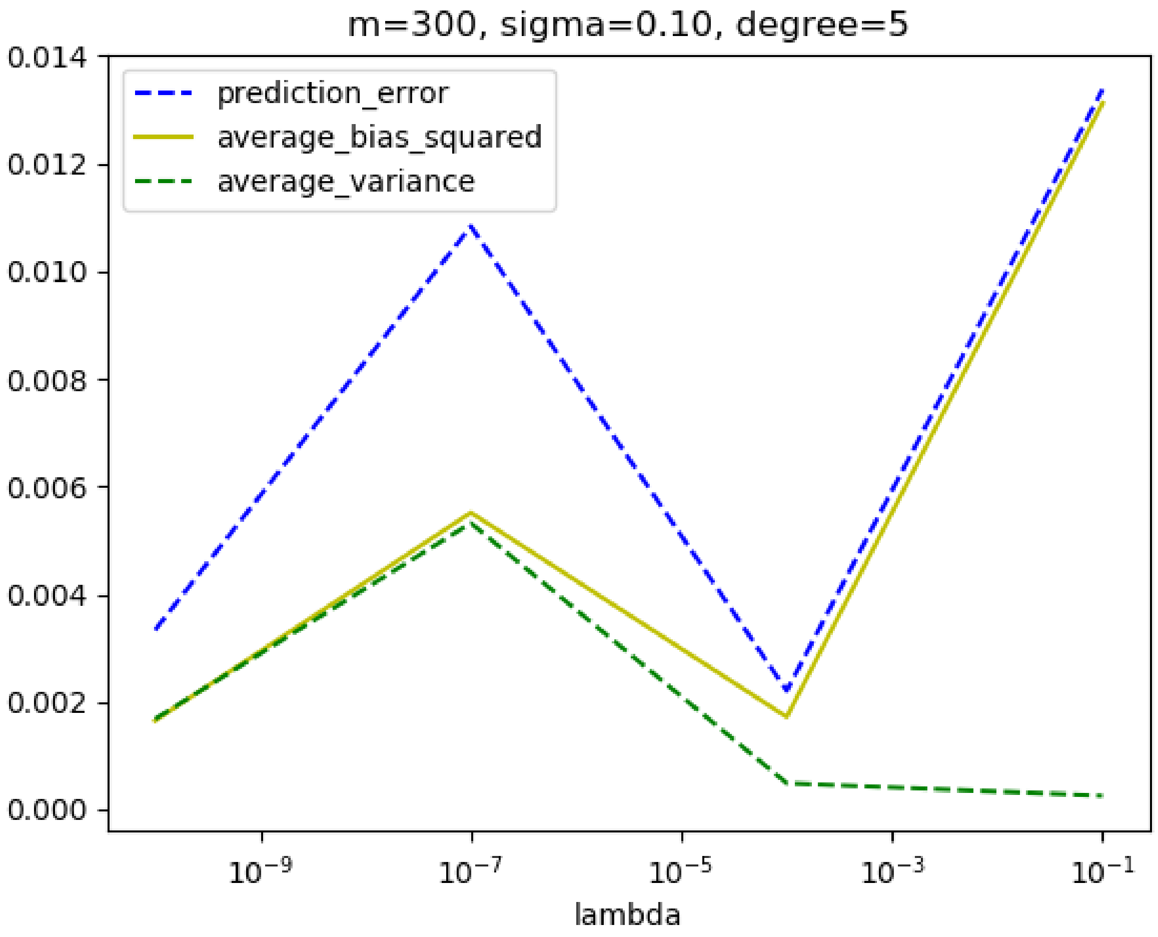

Figure 5: This shows the bias-variance tradeoff as function of the regularization parameter, lambda,for Ridge regression with a sample size of 300, polynomial degree of 5 and standard deviation of 0.1

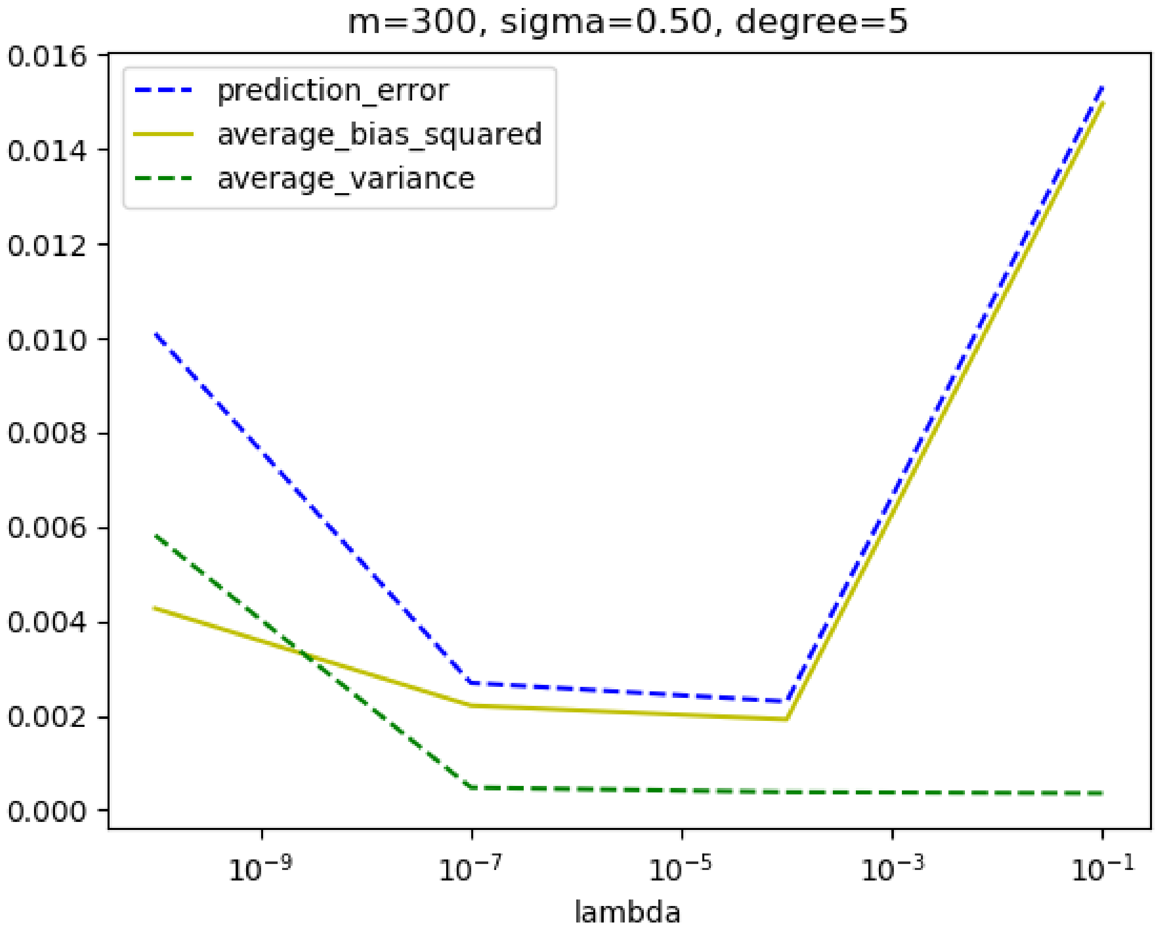

Figure 6: This shows the bias-variance tradeoff as function of the regularization parameter, lambda,for Ridge regression with a sample size of 300, polynomial degree of 5 and standard deviation of 0.5

4.1.3 Lasso

Below are the figures for varying sample sizes, polynomials and penalty with Lasso.

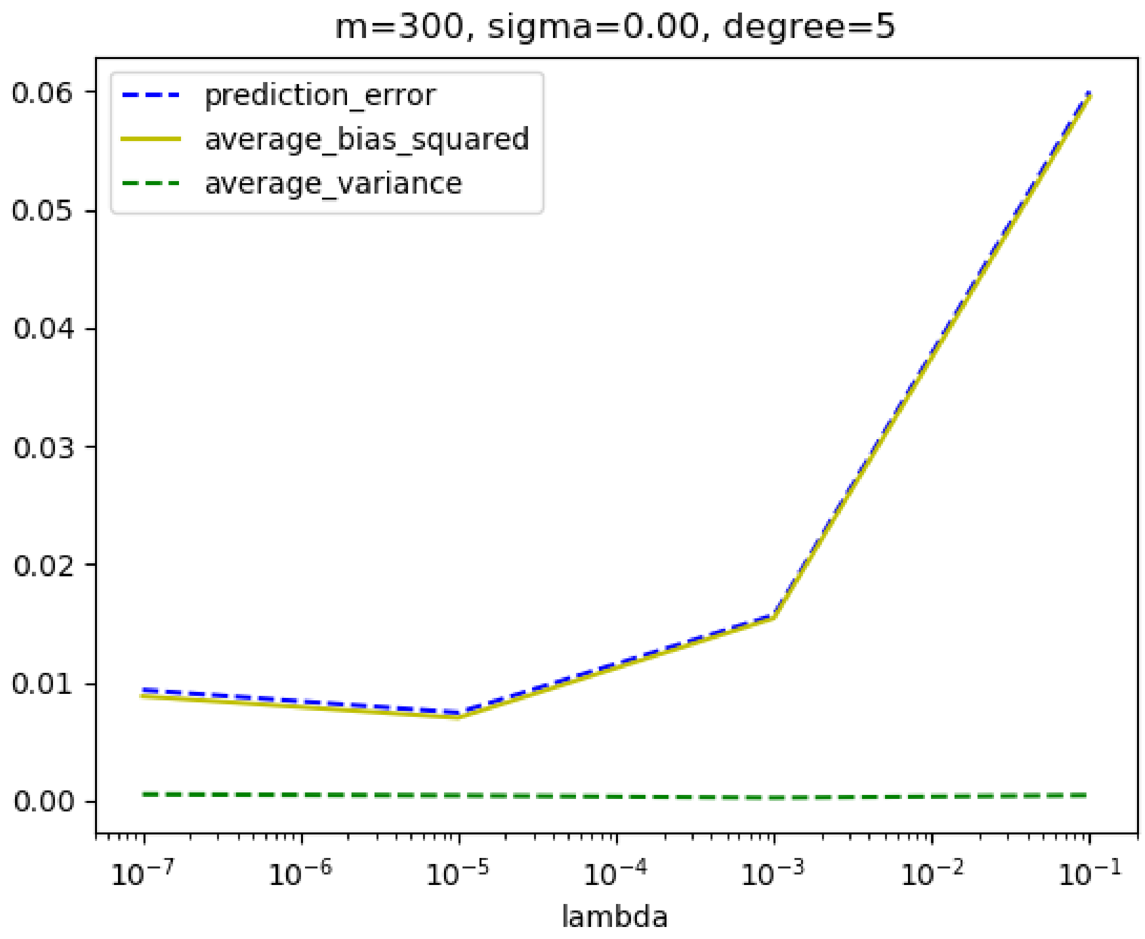

Figure 7: This shows the bias-variance tradeoff as function of the regularization parameter, lambda, for Lasso regression method with a sample size of 300 and a polynomial degree of 5

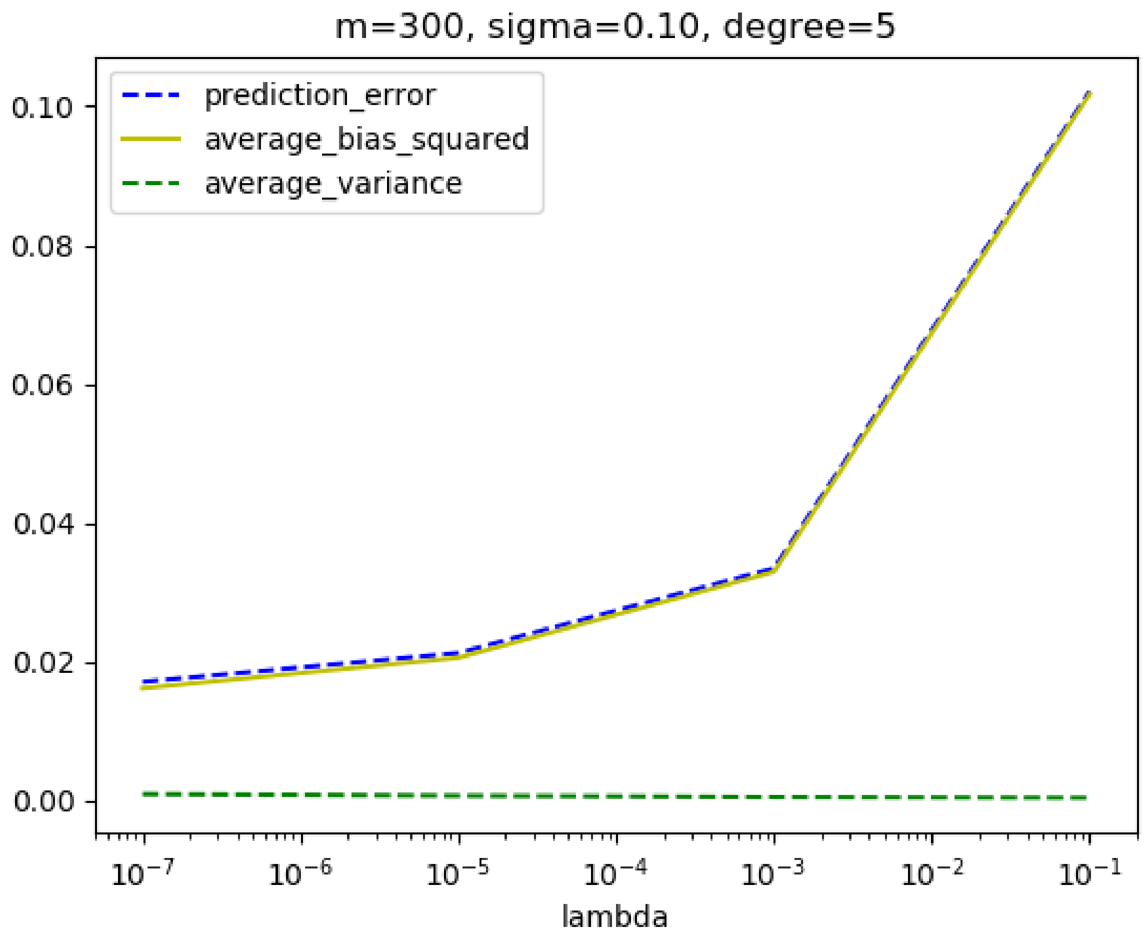

Figure 8: This shows the bias-variance tradeoff as function of the regularization parameter, lambda, for Lasso regression method with a sample size of 300, polynomial degree of 5 and standard deviation of 0.1.

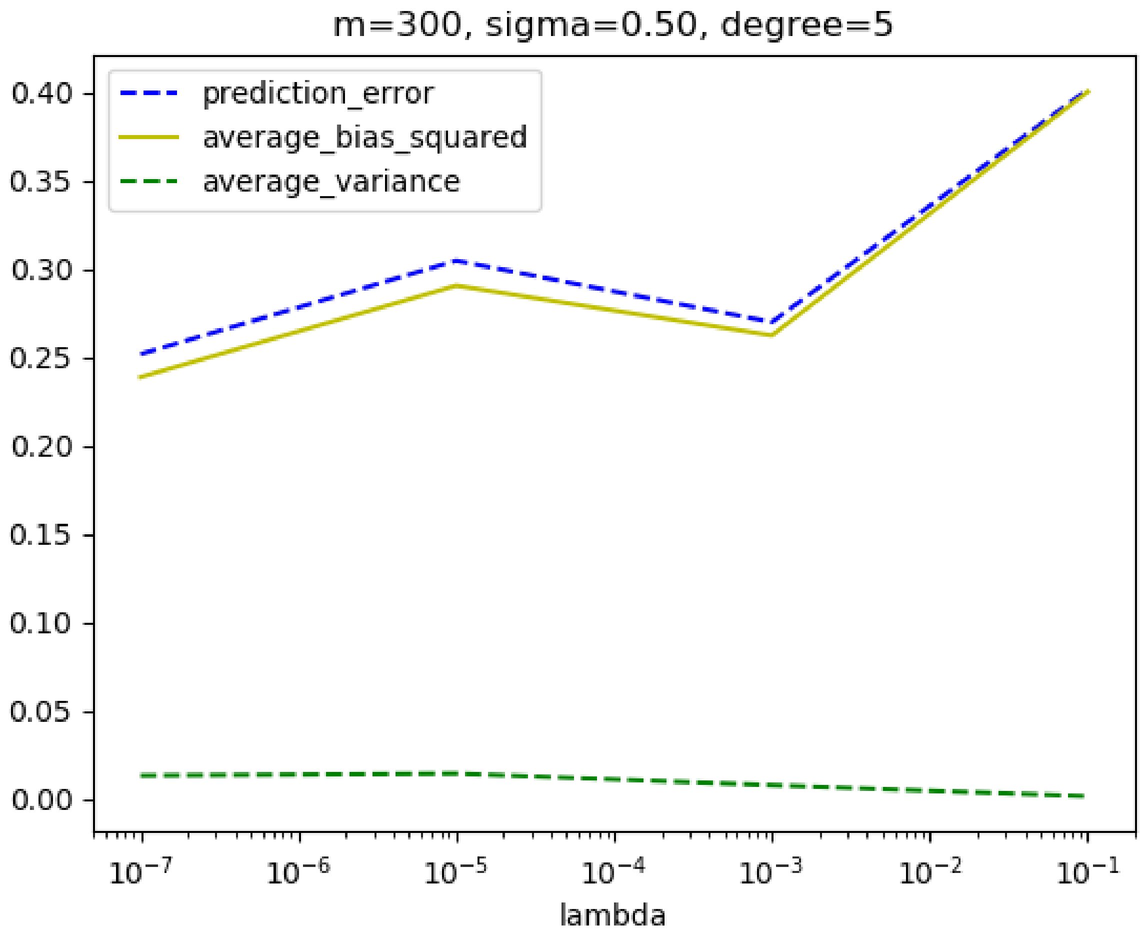

Figure 9: This shows the bias-variance tradeoff as function of the regularization parameter, lambda, for Lasso regression method with a sample size of 300, a polynomial degree of 5 and standard deviation of 0.5

4.2 Discussion

From all the figures I will now consider prediction error, bias and variance for the various methods.

4.2.1 OLS

For the OLS I can see that the minimum prediction error is for a polynomial of degree 4 for all standard deviations 0, 0.1 and 0.5. IN all cases the value of the prediction error is about 0.004. The bias is smallest at 5 degrees in all cases. The variance is smallest for a degree 4 polynomial for standard deviations of 0 and 0.1. For standard deviation of 0.5 the smallest variance is at 3. For a degree 1 polynomial, we do have a low variance, but the bias is really high.

4.2.2 Ridge

For the ridge regression I can see a low prediction error for lambda being 10^{-4} and 10^{-10} for the 0 and 0.1 standard deviation cases, with a value of the prediction error of about 0.0035 and 0.002. The lowest value being for 0.1 standard deviation and lambda 10^{-4}. For the 0.5 standard deviation case, the prediction error is the smallest at about 0.25 for lambda 10^{-4}.

4.2.3 Lasso

For the lasso regression at standard deviation of 0 I can see a minimum prediction error of about 0.009 for lambda 10^{-5}. For standard deviation 0.1 the prediction error is up to 0.019 and for standard deviation 0.5 all the way up to 0.25. This tells me that something might have gone wrong with the implementation of lasso.

5 Conclusion

The results show that the Ridge regression gives a better prediction error than the other two methods. That does not agree with the expected, since the Lasso method should have been better. I tried to figure out the reason why that happened, and I came up with the following explanation:

The formulation of Lasso is the same as for Ridge, but for the extra β term in Ridge. It seems clear from the results that the Ridge method was to be preferred, but on further inspection, it seems that the Lasso method may have been worse because the standardization method was not done correctly for Lasso. I suspect this is the reason for Lasso being worse than the Ridge regression. Ideally I would think Lasso would be the best method because Lasso does regularization and feature selection, setting parameters to 0 if possible by itself ([2] p.63).

6 Bibliography

References

[1] *https://github.com/CompPhysics/MachineLearning/blob/master/doc/Projects/2018/Project1/pdf/Project1.pdf*

[2] *https://github.com/CompPhysics/MachineLearning/blob/master/doc/Textbooks/elementsstat.pdf*

[3] *https://arxiv.org/pdf/1509.09169.pdf*

[4] *http://www.dtic.mil/dtic/tr/fulltext/u2/a081688.pdf*

6.1 Program code project 1

```
import numpy as np
import scipy.stats as ss
from numpy.polynomial.polynomial import polyvander2d, polyval2d
from collections import namedtuple

from sklearn.linear_model import Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

from sklearn.model_selection import train_test_split
from sklearn.utils import resample


fitstats = namedtuple('stats', ['mse', 'r2', 'beta_variance', 'zscore', 'beta_low', 'beta_up'


class Ridge2d(object):
    """
    Ridge regression for 2D polynoms of given degree

    zi = f(xi, yi) + espilon_i

    f(x, y) = sum beta_ij * x**i * y**j for i=0,1,...deg[0], j=0,1,..., deg[1]

    X_ij = x**i * y**j


    """

    def __init__(self, deg=(2, 2), lam=0, alpha=0.05, fulloutput=False):
        self.deg = deg
        self.lam = lam
        self.alpha = alpha
        self.coefficients = None
        self.fulloutput=fulloutput
        self._mean_X = ()
        self._mean_y = 0


    def fit(self, X, y):
        # ndim = len(self.deg)
        orders = [n + 1 for n in self.deg]
        order = np.prod(orders)
        self._mean_X = np.mean(X, axis=0, keepdims=True)
        self._mean_y = np.mean(y)
        X_ = X - self._mean_X
        y_ = y - self._mean_y

        x0 = X_[:,0].reshape(-1, 1)
        x1 = X_[:,1].reshape(-1, 1)
        xb_all = polyvander2d(x0.ravel(), x1.ravel(), deg=self.deg).reshape(-1, order)
        # xb has shape (n, order) where order = (deg[0] +1 ) * (deg[1] +1)
        # x.reshape(-1, 1)  # shape = (nx, ny)  => Change into shape (n, 1) where n = nx*ny
        # so that xb is:
        # xb_all = [np.ones(n, 1), x**1   ,    x**2, ...   , x**deg[0], x**1 * y**1, x**2 * y**1
        xb = xb_all[:, 1:]  # drop the constant term
        xtx_inv = np.linalg.pinv(xb.T.dot(xb) + self.lam * np.eye(order-1))
        # beta = [beta_00, beta_10,  beta_20, ..., beta_01      beta_11,  beta_21, ..., beta_
beta_22, ...]
        beta = np.vstack((0.,
                          xtx_inv.dot(xb.T).dot(y_.reshape(-1, 1)))).reshape(orders)
        self.coefficients = beta

        if self.fulloutput:
            # beta has shape (deg[0] +1, deg[1] +1)
            yhat = polyval2d(x0, x1, beta) + self._mean_y
            # This equal to evaluating the following sum:  sum beta_ij * x**i * y**j for i=0,

            mean_sqared_error = mse(y, yhat)  #
            n = y.size
            sigma =  n * mean_sqared_error / (n-order) # Eq. after Eq. 3.9 on pp 47 in Hastie
            beta_covariance = xtx_inv * sigma # Eq. 3.10 on pp 47 in Hastie etal.  # Is it va
            beta_variance = np.diag(beta_covariance) # .reshape(orders)

            std_error = np.sqrt(beta_variance)
            z_score = beta / std_error
            # 1-alpha confidence interval for beta. Eq 3.14 in Hastie
            z_alpha = -ss.norm.ppf(self.alpha/2)  # inverse of the gaussian cdf function (ss.
            beta_low = beta - z_alpha * std_error
            beta_up = beta + z_alpha * std_error

            self.stats = fitstats(mse=mean_sqared_error,
                                  r2=r_squared(y, yhat),
                                  beta_variance=beta_variance,
                                  zscore=z_score,
                                  beta_low=beta_low,
                                  beta_up=beta_up)
        return self

    def predict(self, X):
        X_  = X - self._mean_X
        return self._mean_y + polyval2d(X_[:,0], X_[:,1], self.coefficients).reshape(-1, 1)


class OLS2d(Ridge2d):
    """
    Ordinary Least Squares for 2D polynoms of degree 'deg'

    zi = f(xi, yi) + espilon_i

    f(x, y) = sum beta_ij * x**i * y**j for i=0,1,...deg[0], j=0,1,..., deg[1]

    X_ij = x**i * y**j


    """

    def __init__(self, deg=(2, 2), alpha=0.05, fulloutput=False):
        super(OLS2d, self).__init__(deg, 0, alpha, fulloutput)

def bootstrap_bias_variance(x, y, model, n_bootstraps=200, test_size=0.2):
    # Hold out some test data that is never used in training.
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size)

    # The following (m x n_bootstraps) matrix holds the column vectors y_pred
    # for each bootstrap iteration.
    y_pred = np.empty((y_test.shape[0], n_bootstraps))
    for i in range(n_bootstraps):
        x_, y_ = resample(x_train, y_train)

        # Evaluate the new model on the same test data each time.
        y_pred[:, i] = model.fit(x_, y_).predict(x_test).ravel()

    # Note: Expectations and variances taken w.r.t. different training
    # data sets, hence the axis=1. Subsequent means are taken across the test data
    # set in order to obtain a total value, but before this we have average_error/average_bia
    # calculated per data point in the test set.
    # Note 2: The use of keepdims=True is important in the calculation of average_bias_square
    # maintains the column vector form. Dropping this yields very unexpected results.


    mean_y = np.mean(y_test)

    errors = np.mean((y_test - y_pred) ** 2, axis=1, keepdims=True)
    bias_sqares = (y_test - np.mean(y_pred, axis=1, keepdims=True)) ** 2
    variances = np.var(y_pred, axis=1, keepdims=True)

    r2 =  1.0 - np.mean(errors)/mse(y_test, mean_y)
    error = np.mean( errors )
    bias_squared = np.mean(bias_sqares)
    variance = np.mean( variances )

        return error, bias_squared, variance, r2


'''
Created on 27. sep. 2018


@author: ljb


The best way to present these confidence intervals is to nail down say the best model.
If you compute the MSE for the different approximations (polynomials), you may find a
behavior like that of fig 5 of Mehta et al, see https://arxiv.org/pdf/1803.08823.pdf,
that is a model where the MSE is at its minimum. If not, I would pick the model with
lowest MSE (and possibly best R2 score) and tell the reader that this is my recommended
model. Then I would present the parameters beta  for the best model (limit yourself to that)
and include the confidence intervals for that model only. Else you'll end up swamping the
report with tons of data. Based on your calculations it is you who recommends the best model,
with its pros and cons.
'''
import pandas as pd
import numpy as np
import scipy.stats as ss
# from random import random, seed
from numpy.polynomial.polynomial import polyvander2d, polyval2d
from collections import namedtuple
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

from sklearn.linear_model import Ridge, Lasso
```

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

from sklearn.model_selection import train_test_split
from sklearn.utils import resample


fitstats = namedtuple('stats', ['mse', 'r2', 'beta_variance', 'zscore', 'beta_low', 'beta_up'
fitstat2 = namedtuple('stats2', ['prediction_r2_score',
                                 'prediction_error',
                                 'average_bias_squared',
                                 'average_variance'])



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


def r_squared(y, yhat):
    mean_y = np.mean(y)
    return 1.0 - mse(y, yhat)/mse(y, mean_y)


def mse(y, yhat):
    return np.mean((y-yhat)**2)


def cross_validate_prediction_error(X, y, fitter=None, k=5):
    """ K fold cross validation of prediction error
    """
    Xi, yi = resample(X, y, replace=False)  # Shuffle the dataset randomly.
    n = len(yi)

    split = n // k  # size

    indices = np.arange(split * k).reshape(k, split)
    prediction_error = []
    for i, test_indices in enumerate(indices):
        train_indices = np.hstack((indices[:i].ravel(),
                                   indices[:i+1].ravel()))
        fitter.fit(Xi[train_indices,: ], yi[train_indices,:])
        yhat = fitter.predict(Xi[test_indices,:])
        prediction_error.append(mse(y[test_indices, :], yhat))
    return np.mean(prediction_error)


def bootstrap_bias_variance(x, y, model, n_bootstraps=200, test_size=0.2, plot=True):
    # Hold out some test data that is never used in training.
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size)

    # The following (m x n_bootstraps) matrix holds the column vectors y_pred
    # for each bootstrap iteration.
    y_pred = np.empty((y_test.shape[0], n_bootstraps))
    for i in range(n_bootstraps):
        x_, y_ = resample(x_train, y_train)

        # Evaluate the new model on the same test data each time.
        y_pred[:, i] = model.fit(x_, y_).predict(x_test).ravel()

    # Note: Expectations and variances taken w.r.t. different training
    # data sets, hence the axis=1. Subsequent means are taken across the test data
    # set in order to obtain a total value, but before this we have average_error/average_bia
    # calculated per data point in the test set.
    # Note 2: The use of keepdims=True is important in the calculation of average_bias_square
    # maintains the column vector form. Dropping this yields very unexpected results.


    mean_y = np.mean(y_test)

    errors = np.mean((y_test - y_pred) ** 2, axis=1, keepdims=True)
    bias_sqared = (y_test - np.mean(y_pred, axis=1, keepdims=True)) ** 2
    variances = np.var(y_pred, axis=1, keepdims=True)

    average_r2 =  1.0 - np.mean(errors)/mse(y_test, mean_y)
    average_error = np.mean( errors )
    average_bias_squared = np.mean( bias_sqared )
    average_variance = np.mean( variances )

#     print(f'R2: {average_r2:0.2g}')
#     print(f'Error: {average_error:0.2g}')
#     print(f'Bias^2: {average_bias_squared:0.2g}', )
#     print(f'Var: {average_variance:0.2g}', )
#     print('{:0.2g} >= {:0.2g} + {:0.2g} = {:0.2g}'.format(average_error, average_bias_square
#                                                average_variance, average_bias_squared+average_varian
    if plot:
        plt.plot(x[::5, :], y[::5, :], label='f(x)')
        plt.scatter(x_test, y_test, label='Data points')
        plt.scatter(x_test, np.mean(y_pred, axis=1), label='Pred')
        plt.legend()

    stats = fitstat2(prediction_r2_score=average_r2,
                     prediction_error=average_error,
                     average_bias_squared=average_bias_squared,
                     average_variance=average_variance)
    return stats



class Ridge2d(object):
    """
    Ridge regression for 2D polynoms of given degree

    zi = f(xi, yi) + espilon_i

    f(x, y) = sum beta_ij * x**i * y**j for i=0,1,...deg[0], j=0,1,..., deg[1]

    X_ij = x**i * y**j


    """
    def __init__(self, deg=(2, 2), lam=0, alpha=0.05, fulloutput=False):
        self.deg = deg
        self.lam = lam
        self.alpha = alpha
        self.coefficients = None
        self.fulloutput=fulloutput
        self._mean_X = ()
        self._mean_y = 0


    def fit(self, X, y):
        # ndim = len(self.deg)
        orders = [n + 1 for n in self.deg]
        order = np.prod(orders)
        self._mean_X = np.mean(X, axis=0, keepdims=True)
        self._mean_y = np.mean(y)
        X_ = X - self._mean_X
        y_ = y - self._mean_y

        x0 = X_[:,0].reshape(-1, 1)
        x1 = X_[:,1].reshape(-1, 1)
        xb_all = polyvander2d(x0.ravel(), x1.ravel(), deg=self.deg).reshape(-1, order)
        # xb has shape (n, order) where order = (deg[0] +1 ) * (deg[1] +1)
        # x.reshape(-1, 1)  # shape = (nx, ny)  => Change into shape (n, 1) where n = nx*ny
        # so that xb is:
        # xb_all = [np.ones(n, 1), x**1   ,   x**2, ...   , x**deg[0], x**1 * y**1, x**2 * y**1
        xb = xb_all[:, 1:]  # drop the constant term
        xtx_inv = np.linalg.pinv(xb.T.dot(xb) + self.lam * np.eye(order-1))
        # beta = [beta_00, beta_10,  beta_20, ..., beta_01       beta_11,  beta_21, ..., beta_
beta_22, ...]
        beta = np.vstack((0.,
                          xtx_inv.dot(xb.T).dot(y_.reshape(-1, 1)))).reshape(orders)
        self.coefficients = beta

        if self.fulloutput:
            # beta has shape (deg[0] +1, deg[1] +1)
            yhat = polyval2d(x0, x1, beta) + self._mean_y
            # This equal to evaluating the following sum:  sum beta_ij * x**i * y**j for i=0,

            mean_sqared_error = mse(y, yhat)  #
            n = y.size
            sigma =  n * mean_sqared_error / (n-order) # Eq. after Eq. 3.9 on pp 47 in Hastie
            beta_covariance = xtx_inv * sigma # Eq. 3.10 on pp 47 in Hastie etal.  # Is it va
            beta_variance = np.diag(beta_covariance) # .reshape(orders)

            std_error = np.sqrt(beta_variance)
            z_score = beta / std_error
            # 1-alpha confidence interval for beta. Eq 3.14 in Hastie
            z_alpha = -ss.norm.ppf(self.alpha/2)  # inverse of the gaussian cdf function (ss.
            beta_low = beta - z_alpha * std_error
            beta_up = beta + z_alpha * std_error

            self.stats = fitstats(mse=mean_sqared_error,
                                  r2=r_squared(y, yhat),
                                  beta_variance=beta_variance,
                                  zscore=z_score,
                                  beta_low=beta_low,
                                  beta_up=beta_up)
        return self

    def predict(self, X):
        X_ = X - self._mean_X
        return self._mean_y + polyval2d(X_[:,0], X_[:,1], self.coefficients).reshape(-1, 1)


class OLS2d(Ridge2d):
    """
    Ordinary Least Squares for 2D polynoms of degree 'deg'

    zi = f(xi, yi) + espilon_i

    f(x, y) = sum beta_ij * x**i * y**j for i=0,1,...deg[0], j=0,1,..., deg[1]

    X_ij = x**i * y**j


    """

    def __init__(self, deg=(2, 2), alpha=0.05, fulloutput=False):
        super(OLS2d, self).__init__(deg, 0, alpha, fulloutput)


def fit_lasso2d(X_, z_, z0_, deg, lam):
    fitter = make_pipeline(PolynomialFeatures(deg[0]), Lasso(alpha=lam, max_iter=4000))
    return _fit_2d(X_, z_, z0_, fitter)


def fit_ridge2d(X_, z_, z0_, deg, lam):
    # fitter = make_pipeline(PolynomialFeatures(deg[0]), Ridge(alpha=lam))
    fitter = Ridge2d(deg=deg, lam=lam, alpha=0.05, fulloutput=False)
    return _fit_2d(X_, z0_, z_, fitter)



def _fit_2d(X_, z_, z0_, fitter):
    stats = bootstrap_bias_variance(X_, z_, fitter, n_bootstraps=200, test_size=0.2, plot=Fals

    return stats


    shape = z0_.shape

    split = shape[0] // 5
    # prepare bootstrap sample
    (x_train, x_test,
     z_train, z_test,
     z0_train, z0_test) = train_test_split(X_, z_, z0_, test_size=split)

    test_data = X_[:split],  z_[:split]
    train_data = X_[split:],  z_[split:]

    prediction_error_from_cross_validation = cross_validate_prediction_error(x_train,
                                                                             z_train, fitter,

    fitter.fit(x_train, z_train)
    # fit_fun = partial(ridge2d, deg=deg, lam=lam)
    # fit_fun = partial(lasso2d, deg=deg, lam=lam)

    # fit_fun = partial(ols2d, deg=deg)  # lam=0
    # beta_avg, beta_var = bootstrap(*train_data, fun=fit_fun)

    zpredict = fitter.predict(x_test)
    prediction_error = mse(z_test, zpredict)
    prediction_r2_score = r_squared(z_test, zpredict)
    # print('R2_cv:{:0.2g}'.format(prediction_r2_score))

    average_bias_squared = mse(z0_test, zpredict)
    stats = fitstat2(prediction_r2_score=r_squared(test_data[1], zpredict),
                     prediction_error=prediction_error,
                     average_bias_squared=average_bias_squared,
                     average_variance=prediction_error - average_bias_squared)

#     print('mean_squared_error: {:.2g}'.format(prediction_error))
#     print('R2: {:.2f}'.format(prediction_r2_score))
#     print('Sigma^2: {:.2g}'.format(sigma ** 2))
#     print('Average Bias**2: {:.2g}'.format(average_bias_squared))
#     print('Average Variance: {:.2g}'.format(average_variance))

    return stats


# def fit_ridge2d(X_, z0_, z_, deg, lam, sigma):
#     shape = z0_.shape
#
#     split = shape[0] // 5
#     # prepare bootstrap sample
#     test_data = X_[:split],  z_[:split]
#     train_data = X_[split:],  z_[split:]
#
#     fitter = make_pipeline(PolynomialFeatures(deg[0]), Ridge(alpha=lam))
#     # fitter = Ridge2d(deg=deg, lam=lam, alpha=0.05, fulloutput=False)
#     prediction_error_from_cross_validation = cross_validate_prediction_error(train_data[0],
#                                                                              train_data[1],
#
#     fitter.fit(train_data[0], train_data[1])
#
#     # fit_fun = partial(ridge2d, deg=deg, lam=lam)
#     # fit_fun = partial(lasso2d, deg=deg, lam=lam)
#
#     # fit_fun = partial(ols2d, deg=deg)  # lam=0
#     # beta_avg, beta_var = bootstrap(*train_data, fun=fit_fun)
#
#     zpredict = fitter.predict(test_data[0])
#     prediction_error = mse(test_data[1], zpredict)
#     average_bias_squared = mse(z0_[:split], zpredict)
#     stats = fitstat2(prediction_r2_score=r_squared(test_data[1], zpredict),
#                      prediction_error_from_cross_validation=prediction_error_from_cross_vali
#                      prediction_error=prediction_error,
#                      average_bias_squared=average_bias_squared,
#                      average_variance=prediction_error - average_bias_squared)
#
##     print('mean_squared_error: {:.2g}'.format(prediction_error))
##     print('R2: {:.2f}'.format(prediction_r2_score))
##      print('Sigma^2: {:.2g}'.format(sigma ** 2))
##      print('Average Bias**2: {:.2g}'.format(average_bias_squared))
##      print('Average Variance: {:.2g}'.format(average_variance))
#
#     return stats



def generate_dataset(m, sigma):
    """
    Return random Franke function data set

    Returns
    -------
        X, z, z0
    where
        z0 = franke_function(X[:, 0], X[:, 1])
    and
        z = z0 + noise

    """

    X = np.random.rand(m, 2)
    z0 = franke_function(X[:, 0], X[:, 1]).reshape(-1, 1)
    z = z0 + sigma * np.random.randn(m, 1)
    return X, z, z0


def plot_prediction_error_vs_degrees(X, z, z0, degrees=2, lam=None, sigma=0, method='ridge'):
    if method == 'ridge':
        fitfun = fit_ridge2d
    elif method == 'lasso':
        fitfun = fit_lasso2d

    m = len(z)
    lam_dict = lam if isinstance(lam, dict) else {}


    stats_all = []
    for degree in degrees:
        if lam_dict:
            # (m, sigma, degree)
            lam = lam_dict[(m, sigma, degree)]
        stats = fitfun(X, z, z0, deg=(degree, degree), lam=lam)
        stats_all.append(tuple(stats))

    stats_all = np.array(stats_all)
    names = list(stats._asdict())

    colors = 'mbygr'
    lines = ['-', '--', '-', '--', '-.']
    if lam==0:
        method_name = dict(ridge='ols').get(method, method)
    else:
        method_name = method
    if lam_dict:
        lam = 'optimum'
    for i, name in enumerate(names):
        sym = colors[i]+lines[i]
        plt.plot(degrees, stats_all[:, i], sym, label=name)
        if i==0:

            plt.title(f'm={m},␣sigma={sigma:0.2f},␣lam={lam}')
            plt.xlabel('degrees')
            plt.axis([0, 5, 0, 1])
            plt.legend()
            plt.savefig('{}_r2_score_vs_degrees_m{}_l{}_s{}.png'.format(method_name, m, int(-n
            # plt.show('hold')
            plt.close()

    plt.title(f'm={m},␣sigma={sigma:0.2f},␣lam={lam}')
    plt.xlabel('degrees')
    # plt.axis([4, 25, 0, 1])
    plt.legend()
    plt.savefig('{}prediction_error_vs_degrees_m{}_l{}_s{}.png'.format(method_name, m, int(-np
    # plt.show('hold')
    plt.close()


def plot_prediction_error_vs_lambdas(X, z, z0, degree=2, lambdas=None, sigma=0, method='ridge
    if method == 'ridge':
        fitfun = fit_ridge2d
    elif method == 'lasso':
        fitfun = fit_lasso2d

    m = len(z)

    stats_all = []
    for lam in lambdas:
        stats = fitfun(X, z, z0, deg=(degree, degree), lam=lam)
        stats_all.append(tuple(stats))

    stats_all = np.array(stats_all)
    names = list(stats._asdict())

    colors = 'mbygr'
    lines = ['-', '--', '-', '--', '-.']
    for i, name in enumerate(names):
        sym = colors[i]+lines[i]
        plt.semilogx(lambdas, stats_all[:, i], sym, label=name)
        if i==0:
            plt.title(f'm={m},␣sigma={sigma:0.2f},␣degree={degree}')
            plt.xlabel('lambda')

            plt.legend()
            plt.savefig('{}_r2_score_m{}_d{}_s{}.png'.format(method, m, degree, int(sigma*100
            plt.close()

    plt.title(f'm={m},␣sigma={sigma:0.2f},␣degree={degree}')
    plt.xlabel('lambda')
    # plt.axis([4, 25, 0, 1])
    plt.legend()

    plt.savefig('{}prediction_error_m{}_d{}_s{}.png'.format(method, m, degree, int(sigma*100)
    # plt.show('hold')
    plt.close()
    lambdas = np.array(lambdas).reshape(-1, 1)
    df = pd.DataFrame.from_records(np.hstack((lambdas, stats_all)), columns=['lambda'] + name
    ix = df['prediction_error'].argmin(axis=0)
    print('lam_min={}'.format(lambdas[ix]))
    filename = '{}_prediction_error_m{}_d{}_s{}.txt'.format(method, m, degree, int(sigma*100)
    df.to_csv(filename, sep='\t', encoding='utf-8')


def test_scikit1d():
    f = lambda x:  np.sin(x) * x

    # generate points used to plot
    x_plot = np.linspace(0, 10, 100)

    # generate points and keep a subset of them
    x = np.linspace(0, 10, 100)
    rng = np.random.RandomState(0)
    rng.shuffle(x)
    x = np.sort(x[:20])
    y = f(x)

    # create matrix versions of these arrays
    X = x[:, np.newaxis]
    X_plot = x_plot[:, np.newaxis]

    colors = ['teal', 'yellowgreen', 'gold']
    lw = 2
    plt.plot(x_plot, f(x_plot), color='cornflowerblue', linewidth=lw,
             label="ground truth")
    plt.scatter(x, y, color='navy', s=30, marker='o', label="training points")

    for count, degree in enumerate([3, 4, 5]):
        model = make_pipeline(PolynomialFeatures(degree), Ridge())
        model.fit(X, y)
        y_plot = model.predict(X_plot)
        plt.plot(x_plot, y_plot, color=colors[count], linewidth=lw,
                 label="degree %d" % degree)

    plt.legend(loc='lower left')

    plt.show()


def example_fit_scikit2d(deg=(4, 4), m=10000, sigma=0.1, lam=1):
    shape = m, 1
    x = np.random.rand(*shape)
    y = np.random.rand(*shape)
    z0 = franke_function(x, y)
    z = z0 + sigma * np.random.randn(*shape)

    X = np.hstack((x, y))
#       poly = PolynomialFeatures(degree=deg[0])
#       t_X = poly3.fit_transform(X)
#       model = linear_model.Lasso(alpha=lam)

    model = make_pipeline(PolynomialFeatures(deg[0]), Lasso(alpha=lam))
    model.fit(X, z)

    xy = np.linspace(0, 1, 201)
    xnew, ynew = np.meshgrid(xy, xy)
    zpredict = model.predict(np.vstack((xnew.ravel(), ynew.ravel())).T).reshape(xnew.shape)


    plot_surface(xnew, ynew, zpredict)

    z_new = franke_function(xnew, ynew)
    print(1-mse(z_new, zpredict)/mse(z_new, z_new.mean()))

    plot_surface(xnew, ynew, z_new)
#     ix = np.argsort(x.ravel())
#     plt.plot(x[ix,:], y[ix, :], 'g-')
#     plt.axis([0, 2.0, 0, 15.0])
#     plt.xlabel(r"$x$")
#     plt.ylabel(r"$y$")
#     plt.title(r'Linear Regression')
    plt.show('hold')


def example_fit_2d(deg=(4, 4), m=10000, sigma=0.1, lam=1):
    shape = m, 1
    x = np.random.rand(*shape)
    y = np.random.rand(*shape)
    z0 = franke_function(x, y)
    z = z0 + sigma * np.random.randn(*shape)

    fitter = Ridge2d(deg=deg, lam=lam, fulloutput=True)
    fitter.fit(np.hstack((x, y)), z)
    print(fitter.stats)

    xy = np.linspace(0, 1, 201)
    xnew, ynew = np.meshgrid(xy, xy)
    zpredict = fitter.predict(np.vstack((xnew.ravel(), ynew.ravel())).T).reshape(xnew.shape)


    plot_surface(xnew, ynew, zpredict)

    z_new = franke_function(xnew, ynew)
    print(mse(z_new, zpredict)/mse(z_new, z_new.mean()))

    plot_surface(xnew, ynew, z_new)
#     ix = np.argsort(x.ravel())
#     plt.plot(x[ix,:], y[ix, :], 'g-')
#     plt.axis([0, 2.0, 0, 15.0])
#     plt.xlabel(r"$x$")
#     plt.ylabel(r"$y$")
#     plt.title(r'Linear Regression')
    plt.show('hold')


def plot_surface(x, y, z):
    fig = plt.figure()
    ax = fig.gca(projection="3d")
    # Plot the surface.
    surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm, linewidth=0, antialiased=False)
    # Customize the z axis.
    ax.set_zlim(-0.10, 1.40)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
    # Add a color bar which maps values to colors.
    fig.colorbar(surf, shrink=0.5, aspect=5)


def task_a(degrees=(1,2,3,4,5),
         lambdas=(0, ),
         sample_sizes=(300, 3000, 30000),
         sigmas=(0, 0.1, 0.5),
         method='ridge'):

    for m in sample_sizes:
        print(f'm={m}')
        for sigma in sigmas:
            X, z, z0 = generate_dataset(m, sigma)
            print(f'    sigma={sigma}')
            for lam in lambdas:
                print(f'        lambda={lam}')
                plot_prediction_error_vs_degrees(X, z, z0, degrees, lam=lam, sigma=sigma, metl



def task_b(degrees=(1,2,3,4,5), lambdas=(1e-10, 1e-7, 1e-4, 1e-1),
           sample_sizes=(300, 3000, 30000),
           sigmas=(0, 0.1, 0.5)):

    task(degrees, lambdas, sample_sizes, sigmas, method='ridge')


def task_c(degrees=(1,2,3,4,5), lambdas=(1e-7, 1e-5, 1e-3, 1e-1),
           sample_sizes=(300, 3000, 30000),
           sigmas=(0, 0.1, 0.5)):
    task(degrees, lambdas, sample_sizes, sigmas, method='lasso')


def task(degrees=(1,2,3,4,5),
         lambdas=(1e-10, 1e-7, 1e-4, 1e-1, 10, 100),
         sample_sizes=(300, 3000, 30000),
         sigmas=(0, 0.1, 0.5),
         method='ridge'):

    for m in sample_sizes:
        print(f'm={m}')
        for sigma in sigmas:
            X, z, z0 = generate_dataset(m, sigma)
            print(f'    sigma={sigma}')
            for degree in degrees:
                print(f'        degree={degree}')
                plot_prediction_error_vs_lambdas(X, z, z0, degree, lambdas=lambdas, sigma=sigm


 # (m, sigma, degree)
RIDGE_OPTIMUM_LAMBDA = {(300,0,1): 1.00000000e-07,
                        (300,0,2): 1.00000000e-10,
                        (300,0,3): 1.00000000e-10,
                        (300,0,4): 1.00000000e-7,
                        (300,0,5): 1.00000000e-7,

                        (300,0.1,1): 1.00000000e-1,
                        (300,0.1,2): 1.00000000e-4,
                        (300,0.1,3): 1.00000000e-10,
                        (300,0.1,4): 1.00000000e-7,
                        (300,0.1,5): 1.00000000e-4,

                        (300,0.5,1): 1.00000000e-4,
                        (300,0.5,2): 1.00000000e-4,
                        (300,0.5,3): 1.00000000e-7,
                        (300,0.5,4): 1.00000000e-7,
                        (300,0.5,5): 1.00000000e-4,

                        (3000,0,1): 1.00000000e-10,
                        (3000,0,2): 1.00000000e-1,
                        (3000,0,3): 1.00000000e-1,
                        (3000,0,4): 1.00000000e-10,
                        (3000,0,5): 1.00000000e-10,

                        (3000,0.1,1): 1.00000000e-10,
                        (3000,0.1,2): 1.00000000e-7,
                        (3000,0.1,3): 1.00000000e-10,
                        (3000,0.1,4): 1.00000000e-7,
                        (3000,0.1,5): 1.00000000e-7,

                        (3000,0.5,1): 1.00000000e-10,
                        (3000,0.5,2): 1.00000000e-10,
                        (3000,0.5,3): 1.00000000e-4,
                        (3000,0.5,4): 1.00000000e-7,
                        (3000,0.5,5): 1.00000000e-7,
                        }


if __name__ == '__main__':

    # example_fit_2d(deg=(5, 5), m=5000, sigma=.0, lam=0)
    task_a(degrees=(1, 2, 3, 4, 5),
           lambdas=(RIDGE_OPTIMUM_LAMBDA, ),
           sample_sizes=(300, 3000),
           sigmas=(0, 0.1, 0.5),
           method='ridge')

    task_b()
    # task_c()
    # test_scikit1d()
    # example_fit_scikit2d(deg=(5,5), m=10000, sigma=0.0, lam=1e-5)
```