Day 3: Homework 2

Data Analysis and Machine Learning

May 22, 2020

Day three exercises

Exercise 1. This exercise is a continuation of exercise 3 from homework 1. We will use the same function to generate our data set, still staying with a simple function $y(x)$ which we want to fit using linear regression, but now extending the analysis to include the Ridge and the Lasso regression methods. You can use the code under the Regression as an example on how to use the Ridge and the Lasso methods, see the regression slides).

We will thus again generate our own dataset for a function $y(x)$ where $x \in [0, 1]$ and defined by random numbers computed with the uniform distribution. The function y is a quadratic polynomial in x with added stochastic noise according to the normal distribution $\mathcal{N}(\prime, \infty)$.

The following simple Python instructions define our x and y values (with 100 data points). x = np.random.rand(100,1) y = 5*x*x+0.1*np.random.randn(100,1)

1. (1a) Write your own code for the Ridge method (see chapter 3.4 of Hastie *et al.*, equations (3.43) and (3.44)) and compute the parametrization for different values of λ. Compare and analyze your results with those from exercise 2. Study the dependence on λ while also varying the strength of the noise in your expression for $y(x)$.

2. (1b) Repeat the above but using the functionality of **Scikit-Learn**. Compare your code with the results from **Scikit-Learn**. Remember to run with the same random numbers for generating x and y.

3. (1c) Our next step is to study the variance of the parameters β_1 and β_2 (assuming that we are parameterizing our function with a second-order polynomial). We will use standard linear regression and the Ridge regression. You can now opt for either writing your own function or using **Scikit-Learn** to find the parameters β. From your results calculate the variance of these paramaters (recall that this is equal to the diagonal elements of the matrix $(\hat{X}^T\hat{X}) + \lambda\hat{I})^{-1}$). Discuss the results of these variances as functions of λ. In particular, try to link your discussion with the discussion in Hastie *et al.* and their figure 3.11.

4. (1d) Repeat the previous step but add now the Lasso method, see equation (3.53) of Hastie *et al.*. Discuss your results and compare with standard regression and the Ridge regression results. You can write your own code or use the functionality of **scikit-learn**. We recommend the latter since we have not yet discussed how to solve the Lasso equations numerically.

5. (1e) Finally, using **Scikit-Learn** or your own code, compute also the mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error defined as

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

and the R^2 score function. If $\tilde{\hat{y}}_i$ 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

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

Discuss these quantities as functions of the variable λ in the Ridge and Lasso regression methods.

Exercise 2

A much used approach before starting to train the data is to preprocess our data. Normally the data may need a rescaling and/or may be sensitive to extreme values. Scaling the data renders our inputs much more suitable for the algorithms we want to employ.

Scikit-Learn has several functions which allow us to rescale the data, normally resulting in much better results in terms of various accuracy scores. The **StandardScaler** function in **Scikit-Learn** ensures that for each feature/predictor we study the mean value is zero and the variance is one (every column in the design/feature matrix). This scaling has the drawback that it does not ensure that we have a particular maximum or minimum in our data set. Another function included in **Scikit-Learn** is the **MinMaxScaler** which ensures that all features are exactly between 0 and 1. The

The **Normalizer** scales each data point such that the feature vector has a euclidean length of one. In other words, it projects a data point on the circle (or sphere in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of it's length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

The **RobustScaler** works similarly to the StandardScaler in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the RobustScaler uses the median and quartiles, instead of mean and variance. This makes the RobustScaler ignore data points that are very different from the rest (like measurement errors). These odd data points are also called outliers, and might often lead to trouble for other scaling techniques.

It also common to split the data in a **training** set and a **testing** set. A typical split is to use 80% of the data for training and the rest for testing. This can be done as follows with our design matrix $\boldsymbol{X}$ and data $\boldsymbol{y}$ (remember to import **scikit-learn**) split in training and test data X$_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)$ Then we can use the stadndard scale to scale our data as scaler = StandardScaler() scaler.fit(X$_train) X_train_scaled = scaler.transform(X_train) X_test_scaled = scaler.transform(X_test)$

In this exercise we want you to to compute the MSE for the training data and the test data as function of the complexity of a polynomial, that is the degree of a given polynomial. We want you also to compute the $R2$ score as function of the complexity of the model for both training data and test data. You should also run the calculation with and without scaling.

One of the aims is to reproduce Figure 2.11 of Hastie et al. We will also use Ridge and Lasso regression.

Our data is defined by $x \in [-3, 3]$ with a total of for example 100 data points. np.random.seed() n = 100 maxdegree = 14 Make data set. x = np.linspace(-3, 3, n).reshape(-1, 1) y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape) where y is the function we want to fit with a given polynomial.

1. (2a) Write a first code which sets up a design matrix X defined by a fifth-order polynomial. Scale your data and split it in training and test data.

2. (2b) Perform an ordinary least squares and compute the means squared error and the $R2$ factor for the training data and the test data, with and without scaling.

3. (2c) Add now a model which allows you to make polynomials up to degree 15. Perform a standard OLS fitting of the training data and compute the MSE and $R2$ for the training and test data and plot both test and training data MSE and $R2$ as functions of the polynomial degree. Compare what you see with Figure 2.11 of Hastie et al. Comment your results. For which polynomial degree do you find an optimal MSE (smallest value)?

4. (2d) Repeat part (2c) but now using Ridge regressions with various hyperparameters λ. Make the same plots for the optimal λ value for each polynomial degree. Compare these results with those from the standard OLS approach.

Example of how to solve the previous exercise

import matplotlib.pyplot as plt import numpy as np from sklearn.model$_{s}electionimporttrain_{t}est_{s}plitfromsklear$

n = 100 Make data set. x = np.linspace(-3, 3, n).reshape(-1, 1) y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape) X$_{t}rain, X_{t}est, y_{t}rain, y_{t}est = train_{t}est_{s}plit(x, y, test_{s}ize = 0.2)$

from sklearn.preprocessing import StandardScaler scaler = StandardScaler() scaler.fit(X$_{t}rain)X_{t}rain_{s}caled = scaler.transform(X_{t}rain)X_{t}est_{s}caled = scaler.transform(X_{t}est)$

Decide which values of lambda to use nlambdas = 500 lambdas = np.logspace(-3, 5, nlambdas)

estimated$_{m}se_{s}klearn = np.zeros(nlambdas)i = 0forlmbinlambdas : clf_{r}idge = Ridge(alpha = lmb).fit(X_{t}rain_{s}caled, y_{t}rain)yridge = clf_{r}idge.predict(X_{t}est_{s}caled)estimated_{m}se_{s}klearn[i] =$
$mean_{s}quared_{e}rror(y_{t}est, yridge)i+ = 1plt.figure()plt.plot(np.log10(lambdas), estimated_{m}se_{s}klearn, label ='$
$RidgeMSE')plt.xlabel('log10(lambda)')plt.ylabel('MSE')plt.legend()plt.show()$

And now with OLS only and Bootstrap

import matplotlib.pyplot as plt import numpy as np from sklearn.model$_{s}electionimporttrain_{t}est_{s}plitfromsklear$

n = 100 n$_{b}oostraps = 100maxdegree = 14$

Make data set. x = np.linspace(-3, 3, n).reshape(-1, 1) y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape) error = np.zeros(maxdegree) bias = np.zeros(maxdegree) variance = np.zeros(maxdegree) polydegree = np.zeros(maxdegree)

Make data set. x = np.linspace(-3, 3, n).reshape(-1, 1) y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape) X$_{t}rain, X_{t}est, y_{t}rain, y_{t}est = train_{t}est_{s}plit(x, y, test_{s}ize = 0.2)$

from sklearn.preprocessing import StandardScaler scaler = StandardScaler() scaler.fit(X$_{t}rain)X_{t}rain_{s}caled = scaler.transform(X_{t}rain)X_{t}est_{s}caled = scaler.transform(X_{t}est)$

for degree in range(maxdegree): model = make$_{p}ipeline(PolynomialFeatures(degree = degree), LinearRegression(fit_{i}ntercept = False))y_{p}red = np.empty((y_{t}est.shape[0], n_{b}oostraps))foriinrange$
$x_{,}y_{=}resample(X_{t}rain_{s}caled, y_{t}rain)y_{p}red[:, i] = model.fit(x_{,}y_{)}.predict(X_{t}est_{s}caled).ravel()$

polydegree[degree] = degree error[degree] = np.mean(np.mean((y$_{t}est - y_{p}red) * *2, axis = 1, keepdims = True))bias[degree] = np.mean((y_{t}est - np.mean(y_{p}red, axis = 1, keepdims = True))**2)variance[degree] = np.mean(np.var(y_{p}red, axis = 1, keepdims = True))print('Polynomialdegree :', degree)print('Error :', error[degree])print('Bias^{2} :'$
$, bias[degree])print('Var :', variance[degree])print(' >= + = '.format(error[degree], bias[degree], variance[d$
$variance[degree]))$

plt.plot(polydegree, error, label='Error') plt.plot(polydegree, bias, label='bias') plt.plot(polydegree, variance, label='Variance') plt.legend() plt.show()