Data Analysis and Machine Learning: Day 2, Ridge and Lasso Regression and Resampling Methods

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway

partment of Physics and Astronomy and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, U

Feb 17, 2021

Plans for Day 2

- Basics of linear regression
- Statistics, probability theory and resampling methods
- Shrinkage methods: Ridge and Lasso Regression

Linear Regression, basic overview

The aim of this set of lectures is to introduce basic aspects of linear regression, a widely applied set of methods used to fit continuous functions.

We will also use these widely popular methods to introduce resampling techniques like bootstrapping and cross-validation.

We will in particular focus on

- Ordinary linear regression
- Ridge regression
- Lasso regression
- Resampling techniques
- Bias-variance tradeoff

Why Linear Regression (aka Ordinary Least Squares and family)?

Fitting a continuous function with linear parameterization in terms of the parameters $\boldsymbol{\beta}$.

- Method of choice for fitting a continuous function!
- Gives an excellent introduction to central Machine Learning features with **understandable pedagogical** links to other methods like **Neural Networks**, **Support Vector Machines** etc
- Analytical expression for the fitting parameters $\boldsymbol{\beta}$
- Analytical expressions for statistical propertiers like mean values, variances, confidence intervals and more
- Analytical relation with probabilistic interpretations
- Easy to introduce basic concepts like bias-variance tradeoff, cross-validation, resampling and regularization techniques and many other ML topics
- Easy to code! And links well with classification problems and logistic regression and neural networks
- Allows for **easy** hands-on understanding of gradient descent methods. These methods are at the heart of all essentially all Machine Learning methods.
- and many more features

Additional Reading

For more discussions of Ridge and Lasso regression, Wessel van Wieringen's article is highly recommended. Similarly, Mehta et al's article is also recommended. The textbook by Hastie, Tibshirani, and Friedman on The Elements of Statistical Learning Data Mining, chapter 3 is highly recommended. Also Bishop's text, chapter 3 is an excellent read.

Regression Analysis, Definitions and Aims

Regression analysis, overarching aims

Regression modeling deals with the description of the sampling distribution of a given random variable y and how it varies as function of another variable or a set of such variables $\boldsymbol{x} = [x_0, x_1, \dots, x_{n-1}]^T$. The first variable is called the **dependent**, the **outcome** or the **response** variable while the set of variables $\boldsymbol{x}$ is called the independent variable, or the predictor variable or the explanatory variable.

A regression model aims at finding a likelihood function $p(\boldsymbol{y}|\boldsymbol{x})$, that is the conditional distribution for $\boldsymbol{y}$ with a given $\boldsymbol{x}$. The estimation of $p(\boldsymbol{y}|\boldsymbol{x})$ is made using a data set with

- n cases $i = 0, 1, 2, \dots, n-1$
- Response (target, dependent or outcome) variable y_i with $i = 0, 1, 2, \dots, n-1$
- p so-called explanatory (independent or predictor) variables $\boldsymbol{x}_i = [x_{i0}, x_{i1}, \dots, x_{ip-1}]$ with $i = 0, 1, 2, \dots, n-1$ and explanatory variables running from 0 to $p-1$. See below for more explicit examples. These variables are also called features or predictors.

The goal of the regression analysis is to extract/exploit relationship between $\boldsymbol{y}$ and $\boldsymbol{x}$ in or to infer causal dependencies, approximations to the likelihood functions, functional relationships and to make predictions, making fits and many other things.

Regression analysis, overarching aims II

Consider an experiment in which p characteristics of n samples are measured. The data from this experiment, for various explanatory variables p are normally represented by a matrix $\mathbf{X}$.

The matrix $\mathbf{X}$ is called the *design matrix*. Additional information of the samples is available in the form of $\boldsymbol{y}$ (also as above). The variable $\boldsymbol{y}$ is generally referred to as the *response variable*. The aim of regression analysis is to explain $\boldsymbol{y}$ in terms of $\boldsymbol{X}$ through a functional relationship like $y_i = f(\mathbf{X}_{i,*})$. When no prior knowledge on the form of $f(\cdot)$ is available, it is common to assume a linear relationship between $\boldsymbol{X}$ and $\boldsymbol{y}$. This assumption gives rise to the *linear regression model* where $\boldsymbol{\beta} = [\beta_0, \dots, \beta_{p-1}]^T$ are the *regression parameters*.

Linear regression gives us a set of analytical equations for the parameters β_j.

Note: The optimal values of the parameters $\boldsymbol{\beta}$ are obtained by minimizing a chosen **cost/risk/loss** function. We will label these as $\hat{\boldsymbol{\beta}}$ or as $\boldsymbol{\beta}^{\mathrm{opt}}$.

Examples

In order to understand the relation among the predictors p, the set of data n and the target (outcome, output etc) $\boldsymbol{y}$, consider the model we discussed for describing nuclear binding energies.

There we assumed that we could parametrize the data using a polynomial approximation based on the liquid drop model. Assuming

$$BE(A) = a_0 + a_1 A + a_2 A^{2/3} + a_3 A^{-1/3} + a_4 A^{-1},$$

we have five predictors, that is the intercept, the A dependent term, the $A^{2/3}$ term and the $A^{-1/3}$ and A^{-1} terms. This gives $p = 0, 1, 2, 3, 4$. Furthermore we have n entries for each predictor. It means that our design matrix is an $n \times p$ matrix $\boldsymbol{X}$.

Here the predictors are based on a model we have made. A popular data set which is widely encountered in ML applications is the so-called credit card default data from Taiwan. The data set contains data on $n = 30000$ credit card holders with predictors like gender, marital status, age, profession, education, etc. In total there are 24 such predictors or attributes leading to a design matrix of dimensionality 24×30000. This is however a classification problem and we will come back to it when we discuss Logistic Regression.

General linear models

Before we proceed let us study a case from linear algebra where we aim at fitting a set of data $\boldsymbol{y} = [y_0, y_1, \dots, y_{n-1}]$. We could think of these data as a result of an experiment or a complicated numerical experiment. These data are functions of a series of variables $\boldsymbol{x} = [x_0, x_1, \dots, x_{n-1}]$, that is $y_i = y(x_i)$ with $i = 0, 1, 2, \dots, n-1$. The variables x_i could represent physical quantities like time, temperature, position etc. We assume that $y(x)$ is a smooth function.

Since obtaining these data points may not be trivial, we want to use these data to fit a function which can allow us to make predictions for values of y which are not in the present set. The perhaps simplest approach is to assume we can parametrize our function in terms of a polynomial of degree $n-1$ with n points, that is

$$y = y(x) \to y(x_i) = \tilde{y}_i + \epsilon_i = \sum_{j=0}^{n-1} \beta_j x_i^j + \epsilon_i,$$

where ϵ_i is the error in our approximation.

Rewriting the fitting procedure as a linear algebra problem

For every set of values y_i, x_i we have thus the corresponding set of equations

$$\begin{aligned}
y_0 &= \beta_0 + \beta_1 x_0^1 + \beta_2 x_0^2 + \cdots + \beta_{n-1} x_0^{n-1} + \epsilon_0 \\
y_1 &= \beta_0 + \beta_1 x_1^1 + \beta_2 x_1^2 + \cdots + \beta_{n-1} x_1^{n-1} + \epsilon_1 \\
y_2 &= \beta_0 + \beta_1 x_2^1 + \beta_2 x_2^2 + \cdots + \beta_{n-1} x_2^{n-1} + \epsilon_2 \\
&\dots\dots \\
y_{n-1} &= \beta_0 + \beta_1 x_{n-1}^1 + \beta_2 x_{n-1}^2 + \cdots + \beta_{n-1} x_{n-1}^{n-1} + \epsilon_{n-1}.
\end{aligned}$$

Rewriting the fitting procedure as a linear algebra problem, more details

Defining the vectors

$$\boldsymbol{y} = [y_0, y_1, y_2, \dots, y_{n-1}]^T,$$

and

$$\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, \dots, \beta_{n-1}]^T,$$

and

$$\boldsymbol{\epsilon} = [\epsilon_0, \epsilon_1, \epsilon_2, \dots, \epsilon_{n-1}]^T,$$

and the design matrix

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_0^1 & x_0^2 & \dots & \dots & x_0^{n-1} \\ 1 & x_1^1 & x_1^2 & \dots & \dots & x_1^{n-1} \\ 1 & x_2^1 & x_2^2 & \dots & \dots & x_2^{n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1}^1 & x_{n-1}^2 & \dots & \dots & x_{n-1}^{n-1} \end{bmatrix}$$

we can rewrite our equations as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}.$$

The above design matrix is called a Vandermonde matrix.

Generalizing the fitting procedure as a linear algebra problem

We are obviously not limited to the above polynomial expansions. We could replace the various powers of x with elements of Fourier series or instead of x_i^j we could have $\cos(jx_i)$ or $\sin(jx_i)$, or time series or other orthogonal functions. For every set of values y_i, x_i we can then generalize the equations to

$$\begin{aligned} y_0 &= \beta_0 x_{00} + \beta_1 x_{01} + \beta_2 x_{02} + \dots + \beta_{n-1} x_{0n-1} + \epsilon_0 \\ y_1 &= \beta_0 x_{10} + \beta_1 x_{11} + \beta_2 x_{12} + \dots + \beta_{n-1} x_{1n-1} + \epsilon_1 \\ y_2 &= \beta_0 x_{20} + \beta_1 x_{21} + \beta_2 x_{22} + \dots + \beta_{n-1} x_{2n-1} + \epsilon_2 \\ &\dots\dots \\ y_i &= \beta_0 x_{i0} + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_{n-1} x_{in-1} + \epsilon_i \\ &\dots\dots \\ y_{n-1} &= \beta_0 x_{n-1,0} + \beta_1 x_{n-1,2} + \beta_2 x_{n-1,2} + \dots + \beta_{n-1} x_{n-1,n-1} + \epsilon_{n-1}. \end{aligned}$$

Note that we have used $p = n$ here. The matrix is thus quadratic (it may be symmetric). This is generally not the case!

Generalizing the fitting procedure as a linear algebra problem

We redefine in turn the matrix $\boldsymbol{X}$ as

$$\boldsymbol{X}=\begin{bmatrix} x_{00} & x_{01} & x_{02} & \dots & \dots & x_{0,n-1} \\ x_{10} & x_{11} & x_{12} & \dots & \dots & x_{1,n-1} \\ x_{20} & x_{21} & x_{22} & \dots & \dots & x_{2,n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots & x_{n-1,n-1} \end{bmatrix}$$

and without loss of generality we rewrite again our equations as

$$\boldsymbol{y}=\boldsymbol{X}\boldsymbol{\beta}+\boldsymbol{\epsilon}.$$

The left-hand side of this equation is kwown. Our error vector $\boldsymbol{\epsilon}$ and the parameter vector $\boldsymbol{\beta}$ are our unknown quantities. How can we obtain the optimal set of β_i values?

Optimizing our parameters

We have defined the matrix $\boldsymbol{X}$ via the equations

$$\begin{aligned} y_0 &= \beta_0x_{00}+\beta_1x_{01}+\beta_2x_{02}+\dots+\beta_{n-1}x_{0n-1}+\epsilon_0 \\ y_1 &= \beta_0x_{10}+\beta_1x_{11}+\beta_2x_{12}+\dots+\beta_{n-1}x_{1n-1}+\epsilon_1 \\ y_2 &= \beta_0x_{20}+\beta_1x_{21}+\beta_2x_{22}+\dots+\beta_{n-1}x_{2n-1}+\epsilon_1 \\ \dots\dots & \\ y_i &= \beta_0x_{i0}+\beta_1x_{i1}+\beta_2x_{i2}+\dots+\beta_{n-1}x_{in-1}+\epsilon_1 \\ \dots\dots & \\ y_{n-1} &= \beta_0x_{n-1,0}+\beta_1x_{n-1,2}+\beta_2x_{n-1,2}+\dots+\beta_{n-1}x_{n-1,n-1}+\epsilon_{n-1}. \end{aligned}$$

As we noted above, we stayed with a system with the design matrix $\boldsymbol{X}\in\mathbb{R}^{n\times n}$, that is we have $p=n$. For reasons to come later (algorithmic arguments) we will hereafter define our matrix as $\boldsymbol{X}\in\mathbb{R}^{n\times p}$, with the predictors refering to the column numbers and the entries n being the row elements.

Our model for the nuclear binding energies

In our introductory notes we looked at the so-called liquid drop model. Let us remind ourselves about what we did by looking at the code.

We restate the parts of the code we are most interested in.

```
# Common imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
```

```python
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("MassEval2016.dat"),'r')


# Read the experimental data with Pandas
Masses = pd.read_fwf(infile, usecols=(2,3,4,6,11),
              names=('N', 'Z', 'A', 'Element', 'Ebinding'),
              widths=(1,3,5,5,5,1,3,4,1,13,11,11,9,1,2,11,9,1,3,1,12,11,1),
              header=39,
              index_col=False)

# Extrapolated values are indicated by '#' in place of the decimal place, so
# the Ebinding column won't be numeric. Coerce to float and drop these entries.
Masses['Ebinding'] = pd.to_numeric(Masses['Ebinding'], errors='coerce')
Masses = Masses.dropna()
# Convert from keV to MeV.
Masses['Ebinding'] /= 1000

# Group the DataFrame by nucleon number, A.
Masses = Masses.groupby('A')
# Find the rows of the grouped DataFrame with the maximum binding energy.
Masses = Masses.apply(lambda t: t[t.Ebinding==t.Ebinding.max()])
A = Masses['A']
Z = Masses['Z']
N = Masses['N']
Element = Masses['Element']
Energies = Masses['Ebinding']

# Now we set up the design matrix X
X = np.zeros((len(A),5))
X[:,0] = 1
X[:,1] = A
X[:,2] = A**(2.0/3.0)
X[:,3] = A**(-1.0/3.0)
X[:,4] = A**(-1.0)
# Then nice printout using pandas
DesignMatrix = pd.DataFrame(X)
```

```
DesignMatrix.index = A
DesignMatrix.columns = ['1', 'A', 'A^(2/3)', 'A^(-1/3)', '1/A']
display(DesignMatrix)
```

With $\boldsymbol{\beta} \in \mathbb{R}^{p\times 1}$, it means that we will hereafter write our equations for the approximation as

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta},$$

throughout these lectures.

Optimizing our parameters, more details

With the above we use the design matrix to define the approximation $\tilde{\boldsymbol{y}}$ via the unknown quantity $\boldsymbol{\beta}$ as

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta},$$

and in order to find the optimal parameters β_i instead of solving the above linear algebra problem, we define a function which gives a measure of the spread between the values y_i (which represent hopefully the exact values) and the parameterized values $\tilde{y}_i$, namely

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \frac{1}{n}\left\{(\boldsymbol{y}-\tilde{\boldsymbol{y}})^T(\boldsymbol{y}-\tilde{\boldsymbol{y}})\right\},$$

or using the matrix $\boldsymbol{X}$ and in a more compact matrix-vector notation as

$$C(\boldsymbol{\beta}) = \frac{1}{n}\left\{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})\right\}.$$

This function is one possible way to define the so-called cost function.

It is also common to define the function C as

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

since when taking the first derivative with respect to the unknown parameters β, the factor of 2 cancels out.

Interpretations and optimizing our parameters

The function

$$C(\boldsymbol{\beta}) = \frac{1}{n}\left\{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})\right\},$$

can be linked to the variance of the quantity y_i if we interpret the latter as the mean value. Below we will show that

$$y_i = \langle y_i \rangle = \beta_0 x_{i,0} + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_{n-1}x_{i,n-1} + \epsilon_i,$$

where $\langle y_i \rangle$ is the mean value. Keep in mind also that till now we have treated y_i as the exact value. Normally, the response (dependent or outcome) variable

y_i the outcome of a numerical experiment or another type of experiment and is thus only an approximation to the true value. It is then always accompanied by an error estimate, often limited to a statistical error estimate given by the standard deviation discussed earlier. In the discussion here we will treat y_i as our exact value for the response variable.

In order to find the parameters β_i we will then minimize the spread of $C(\boldsymbol{\beta})$, that is we are going to solve the problem

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}\left\{\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right)^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right)\right\}.$$

In practical terms it means we will require

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \beta_j} = \frac{\partial}{\partial \beta_j}\left[\frac{1}{n}\sum_{i=0}^{n-1}\left(y_i-\beta_0x_{i,0}-\beta_1x_{i,1}-\beta_2x_{i,2}-\dots-\beta_{n-1}x_{i,n-1}\right)^2\right]=0,$$

which results in

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \beta_j} = -\frac{2}{n}\left[\sum_{i=0}^{n-1}x_{ij}\left(y_i-\beta_0x_{i,0}-\beta_1x_{i,1}-\beta_2x_{i,2}-\dots-\beta_{n-1}x_{i,n-1}\right)\right]=0,$$

or in a matrix-vector form as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0 = \boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right).$$

Interpretations and optimizing our parameters

We can rewrite

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0 = \boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right),$$

as

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\beta},$$

and if the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is invertible we have the solution

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

We note also that since our design matrix is defined as $\boldsymbol{X}\in\mathbb{R}^{n\times p}$, the product $\boldsymbol{X}^T\boldsymbol{X}\in\mathbb{R}^{p\times p}$. In the above case we have that $p \ll n$, in our case $p=5$ meaning that we end up with inverting a small 5×5 matrix. This is a rather common situation, in many cases we end up with low-dimensional matrices to invert. The methods discussed here and for many other supervised learning algorithms like classification with logistic regression or support vector machines, exhibit dimensionalities which allow for the usage of direct linear algebra methods such as **LU** decomposition or **Singular Value Decomposition** (SVD) for finding the inverse of the matrix $\boldsymbol{X}^T\boldsymbol{X}$.

Small question: Do you think the example we have at hand here (the nuclear binding energies) can lead to problems in inverting the matrix $\boldsymbol{X}^T\boldsymbol{X}$? What kind of problems can we expect?

Some useful matrix and vector expressions

The following matrix and vector relation will be useful here and for the rest of the course. Vectors are always written as boldfaced lower case letters and matrices as upper case boldfaced letters. Here we list some useful expressions

$$\frac{\partial (\boldsymbol{b}^T\boldsymbol{a})}{\partial \boldsymbol{a}} = \boldsymbol{b},$$

$$\frac{\partial (\boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a})}{\partial \boldsymbol{a}} = (\boldsymbol{A}+\boldsymbol{A}^T)\boldsymbol{a},$$

$$\frac{\partial tr(\boldsymbol{B}\boldsymbol{A})}{\partial \boldsymbol{A}} = \boldsymbol{B}^T,$$

$$\frac{\partial \log{\vert\boldsymbol{A}\vert}}{\partial \boldsymbol{A}} = (\boldsymbol{A}^{-1})^T.$$

Interpretations and optimizing our parameters

The residuals $\boldsymbol{\epsilon}$ are in turn given by

$$\boldsymbol{\epsilon} = \boldsymbol{y}-\boldsymbol{\tilde{y}} = \boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta},$$

and with

$$\boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right)= 0,$$

we have

$$\boldsymbol{X}^T\boldsymbol{\epsilon}=\boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right)= 0,$$

meaning that the solution for $\boldsymbol{\beta}$ is the one which minimizes the residuals. Later we will link this with the maximum likelihood approach.

Let us now return to our nuclear binding energies and simply code the above equations.

Own code for Ordinary Least Squares

It is rather straightforward to implement the matrix inversion and obtain the parameters $\boldsymbol{\beta}$. After having defined the matrix $\boldsymbol{X}$ we simply need to write

```
# matrix inversion to find beta
beta = np.linalg.inv(X.T @ X) @ X.T @ Energies
# or in a more old-fashioned way
# beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Energies)
# and then make the prediction
ytilde = X @ beta
```

Alternatively, you can use the least squares functionality in **Numpy** as

```
fit = np.linalg.lstsq(X, Energies, rcond =None)[0]
ytildenp = np.dot(fit,X.T)
```

And finally we plot our fit with and compare with data

```
Masses['Eapprox']  = ytilde
# Generate a plot comparing the experimental with the fitted values values.
fig, ax = plt.subplots()
ax.set_xlabel(r'$A = N + Z$')
ax.set_ylabel(r'$E_\mathrm{bind}\,/\mathrm{MeV}$')
ax.plot(Masses['A'], Masses['Ebinding'], alpha=0.7, lw=2,
            label='Ame2016')
ax.plot(Masses['A'], Masses['Eapprox'], alpha=0.7, lw=2, c='m',
            label='Fit')
ax.legend()
save_fig("Masses2016OLS")
plt.show()
```

Adding error analysis and training set up

We can easily test our fit by computing the $R2$ score that we discussed in connection with the functionality of **Scikit-Learn** in the introductory slides. Since we are not using **Scikit-Learn** here we can define our own $R2$ function as

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

and we would be using it as

```
print(R2(Energies,ytilde))
```

We can also add our **MSE** score as

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

print(MSE(Energies,ytilde))
```

and finally the relative error as

```
def RelativeError(y_data,y_model):
    return abs((y_data-y_model)/y_data)
print(RelativeError(Energies, ytilde))
```

The χ^2 function

Normally, the response (dependent or outcome) variable y_i is the outcome of a numerical experiment or another type of experiment and is thus only an approximation to the true value. It is then always accompanied by an error estimate, often limited to a statistical error estimate given by the standard deviation discussed earlier. In the discussion here we will treat y_i as our exact value for the response variable.

Introducing the standard deviation σ_i for each measurement y_i, we define now the χ^2 function (omitting the $1/n$ term) as

$$\chi^2(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{(y_i-\tilde{y}_i)^2}{\sigma_i^2} = \frac{1}{n}\left\{(\boldsymbol{y}-\tilde{\boldsymbol{y}})^T\frac{1}{\boldsymbol{\Sigma^2}}(\boldsymbol{y}-\tilde{\boldsymbol{y}})\right\},$$

where the matrix $\boldsymbol{\Sigma}$ is a diagonal matrix with σ_i as matrix elements.

The χ^2 function

In order to find the parameters β_i we will then minimize the spread of $\chi^2(\boldsymbol{\beta})$ by requiring

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\beta_j} = \frac{\partial}{\partial\beta_j}\left[\frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i-\beta_0x_{i,0}-\beta_1x_{i,1}-\beta_2x_{i,2}-\dots-\beta_{n-1}x_{i,n-1}}{\sigma_i}\right)^2\right]=0,$$

which results in

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\beta_j} = -\frac{2}{n}\left[\sum_{i=0}^{n-1}\frac{x_{ij}}{\sigma_i}\left(\frac{y_i-\beta_0x_{i,0}-\beta_1x_{i,1}-\beta_2x_{i,2}-\dots-\beta_{n-1}x_{i,n-1}}{\sigma_i}\right)\right]=0,$$

or in a matrix-vector form as

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = 0 = \boldsymbol{A}^T\left(\boldsymbol{b}-\boldsymbol{A}\boldsymbol{\beta}\right).$$

where we have defined the matrix $\boldsymbol{A}=\boldsymbol{X}/\boldsymbol{\Sigma}$ with matrix elements $a_{ij}=x_{ij}/\sigma_i$ and the vector $\boldsymbol{b}$ with elements $b_i=y_i/\sigma_i$.

The χ^2 function

We can rewrite

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = 0 = \boldsymbol{A}^T\left(\boldsymbol{b}-\boldsymbol{A}\boldsymbol{\beta}\right),$$

as

$$\boldsymbol{A}^T\boldsymbol{b} = \boldsymbol{A}^T\boldsymbol{A}\boldsymbol{\beta},$$

and if the matrix $\boldsymbol{A}^T\boldsymbol{A}$ is invertible we have the solution

$$\boldsymbol{\beta} = \left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1}\boldsymbol{A}^T\boldsymbol{b}.$$

The χ^2 function

If we then introduce the matrix

$$\boldsymbol{H} = \left(\boldsymbol{A}^T\boldsymbol{A}\right)^{-1},$$

we have then the following expression for the parameters β_j (the matrix elements of $\boldsymbol{H}$ are h_{ij})

$$\beta_j = \sum_{k=0}^{p-1}h_{jk}\sum_{i=0}^{n-1}\frac{y_i}{\sigma_i}\frac{x_{ik}}{\sigma_i} = \sum_{k=0}^{p-1}h_{jk}\sum_{i=0}^{n-1}b_ia_{ik}$$

We state without proof the expression for the uncertainty in the parameters β_j as (we leave this as an exercise)

$$\sigma^2(\beta_j) = \sum_{i=0}^{n-1} \sigma_i^2 \left(\frac{\partial \beta_j}{\partial y_i}\right)^2,$$

resulting in

$$\sigma^2(\beta_j) = \left(\sum_{k=0}^{p-1} h_{jk} \sum_{i=0}^{n-1} a_{ik}\right)\left(\sum_{l=0}^{p-1} h_{jl} \sum_{m=0}^{n-1} a_{ml}\right) = h_{jj}!$$

The χ^2 function

The first step here is to approximate the function y with a first-order polynomial, that is we write

$$y = y(x) \to y(x_i) \approx \beta_0 + \beta_1 x_i.$$

By computing the derivatives of χ^2 with respect to β_0 and β_1 show that these are given by

$$\frac{\partial \chi^2(\boldsymbol{\beta})}{\partial \beta_0} = -2\left[\frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i - \beta_0 - \beta_1 x_i}{\sigma_i^2}\right)\right] = 0,$$

and

$$\frac{\partial \chi^2(\boldsymbol{\beta})}{\partial \beta_1} = -\frac{2}{n}\left[\sum_{i=0}^{n-1} x_i \left(\frac{y_i - \beta_0 - \beta_1 x_i}{\sigma_i^2}\right)\right] = 0.$$

The χ^2 function

For a linear fit (a first-order polynomial) we don't need to invert a matrix!! Defining

$$\gamma = \sum_{i=0}^{n-1} \frac{1}{\sigma_i^2},$$

$$\gamma_x = \sum_{i=0}^{n-1} \frac{x_i}{\sigma_i^2},$$

$$\gamma_y = \sum_{i=0}^{n-1} \left(\frac{y_i}{\sigma_i^2}\right),$$

$$\gamma_{xx} = \sum_{i=0}^{n-1} \frac{x_i x_i}{\sigma_i^2},$$

$$\gamma_{xy} = \sum_{i=0}^{n-1}\frac{y_ix_i}{\sigma_i^2},$$

we obtain

$$\beta_0 = \frac{\gamma_{xx}\gamma_y-\gamma_x\gamma_y}{\gamma\gamma_{xx}-\gamma_x^2},$$

$$\beta_1 = \frac{\gamma_{xy}\gamma-\gamma_x\gamma_y}{\gamma\gamma_{xx}-\gamma_x^2}.$$

This approach (different linear and non-linear regression) suffers often from both being underdetermined and overdetermined in the unknown coefficients β_i. A better approach is to use the Singular Value Decomposition (SVD) method discussed below. Or using Lasso and Ridge regression. See below.

Regression Examples

Fitting an Equation of State for Dense Nuclear Matter

Before we continue, let us introduce yet another example. We are going to fit the nuclear equation of state using results from many-body calculations. The equation of state we have made available here, as function of density, has been derived using modern nucleon-nucleon potentials with the addition of three-body interactions. This time the file is presented as a standard **csv** file.

The beginning of the Python code here is similar to what you have seen before, with the same initializations and declarations. We use also **pandas** again, rather extensively in order to organize our data.

The difference now is that we use **Scikit-Learn's** regression tools instead of our own matrix inversion implementation. Furthermore, we sneak in **Ridge** regression (to be discussed below) which includes a hyperparameter λ, also to be explained below.

The code

```
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import sklearn.linear_model as skl
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)
```

```
if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("EoS.csv"),'r')

# Read the EoS data as  csv file and organize the data into two arrays with density and energies
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
Density = EoS['Density']
#  The design matrix now as function of various polytrops
X = np.zeros((len(Density),4))
X[:,3] = Density**(4.0/3.0)
X[:,2] = Density
X[:,1] = Density**(2.0/3.0)
X[:,0] = 1

# We use now Scikit-Learn's linear regressor and ridge regressor
# OLS part
clf = skl.LinearRegression().fit(X, Energies)
ytilde = clf.predict(X)
EoS['Eols']  = ytilde
# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(Energies, ytilde))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(Energies, ytilde))
# Mean absolute error
print('Mean absolute error: %.2f' % mean_absolute_error(Energies, ytilde))
print(clf.coef_, clf.intercept_)

# The Ridge regression with a hyperparameter lambda = 0.1
_lambda = 0.1
clf_ridge = skl.Ridge(alpha=_lambda).fit(X, Energies)
yridge = clf_ridge.predict(X)
EoS['Eridge']  = yridge
# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(Energies, yridge))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(Energies, yridge))
# Mean absolute error
print('Mean absolute error: %.2f' % mean_absolute_error(Energies, yridge))
print(clf_ridge.coef_, clf_ridge.intercept_)

fig, ax = plt.subplots()
ax.set_xlabel(r'$\rho[\mathrm{fm}^{-3}]$')
ax.set_ylabel(r'Energy per particle')
ax.plot(EoS['Density'], EoS['Energy'], alpha=0.7, lw=2,
            label='Theoretical data')
ax.plot(EoS['Density'], EoS['Eols'], alpha=0.7, lw=2, c='m',
            label='OLS')
ax.plot(EoS['Density'], EoS['Eridge'], alpha=0.7, lw=2, c='g',
            label='Ridge $\lambda = 0.1$')
ax.legend()
save_fig("EoSfitting")
plt.show()
```

The above simple polynomial in density ρ gives an excellent fit to the data.

We note also that there is a small deviation between the standard OLS and the Ridge regression at higher densities. We discuss this in more detail below.

Splitting our Data in Training and Test data

It is normal in essentially all Machine Learning studies to split the data in a training set and a test set (sometimes also an additional validation set). **Scikit-Learn** has an own function for this. There is no explicit recipe for how much data should be included as training data and say test data. An accepted rule of thumb is to use approximately 2/3 to 4/5 of the data as training data. We will postpone a discussion of this splitting to the end of these notes and our discussion of the so-called **bias-variance** tradeoff. Here we limit ourselves to repeat the above equation of state fitting example but now splitting the data into a training set and a test set.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

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

infile = open(data_path("EoS.csv"),'r')

# Read the EoS data as  csv file and organized into two arrays with density and energies
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
Density = EoS['Density']
#  The design matrix now as function of various polytrops
X = np.zeros((len(Density),5))
X[:,0] = 1
X[:,1] = Density**(2.0/3.0)
X[:,2] = Density
X[:,3] = Density**(4.0/3.0)
X[:,4] = Density**(5.0/3.0)
# We split the data in test and training data
X_train, X_test, y_train, y_test = train_test_split(X, Energies, test_size=0.2)
# matrix inversion to find beta
beta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
# and then make the prediction
ytilde = X_train @ beta
print("Training R2")
print(R2(y_train,ytilde))
print("Training MSE")
print(MSE(y_train,ytilde))
ypredict = X_test @ beta
print("Test R2")
print(R2(y_test,ypredict))
print("Test MSE")
print(MSE(y_test,ypredict))
```

The Boston housing data example

The Boston housing data set was originally a part of UCI Machine Learning Repository and has been removed now. The data set is now included in **Scikit-Learn**'s library. There are 506 samples and 13 feature (predictor) variables in this data set. The objective is to predict the value of prices of the house using the features (predictors) listed here.

The features/predictors are

1. CRIM: Per capita crime rate by town
2. ZN: Proportion of residential land zoned for lots over 25000 square feet
3. INDUS: Proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: Nitric oxide concentration (parts per 10 million)
6. RM: Average number of rooms per dwelling
7. AGE: Proportion of owner-occupied units built prior to 1940

8. DIS: Weighted distances to five Boston employment centers
9. RAD: Index of accessibility to radial highways
10. TAX: Full-value property tax rate per USD10000
11. B: $1000(Bk - 0.63)^2$, where Bk is the proportion of [people of African American descent] by town
12. LSTAT: Percentage of lower status of the population
13. MEDV: Median value of owner-occupied homes in USD 1000s

Housing data, the code

We start by importing the libraries

```
import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns
```

and load the Boston Housing DataSet from **Scikit-Learn**

```
from sklearn.datasets import load_boston

boston_dataset = load_boston()

# boston_dataset is a dictionary
# let's check what it contains
boston_dataset.keys()
```

Then we invoke Pandas

```
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
boston['MEDV'] = boston_dataset.target
```

and preprocess the data

```
# check for missing values in all the columns
boston.isnull().sum()
```

We can then visualize the data

```
# set the size of the figure
sns.set(rc={'figure.figsize':(11.7,8.27)})

# plot a histogram showing the distribution of the target values
sns.distplot(boston['MEDV'], bins=30)
plt.show()
```

It is now useful to look at the correlation matrix

```
# compute the pair wise correlation for all columns
correlation_matrix = boston.corr().round(2)
# use the heatmap function from seaborn to plot the correlation matrix
# annot = True to print the values inside the square
sns.heatmap(data=correlation_matrix, annot=True)
```

From the above coorelation plot we can see that **MEDV** is strongly correlated to **LSTAT** and **RM**. We see also that **RAD** and **TAX** are stronly correlated, but we don't include this in our features together to avoid multi-colinearity

```
plt.figure(figsize=(20, 5))

features = ['LSTAT', 'RM']
target = boston['MEDV']

for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
    x = boston[col]
    y = target
    plt.scatter(x, y, marker='o')
    plt.title(col)
    plt.xlabel(col)
    plt.ylabel('MEDV')
```

Now we start training our model

```
X = pd.DataFrame(np.c_[boston['LSTAT'], boston['RM']], columns = ['LSTAT','RM'])
Y = boston['MEDV']
```

We split the data into training and test sets

```
from sklearn.model_selection import train_test_split

# splits the training and test data set in 80% : 20%
# assign random_state to any value.This ensures consistency.
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state=5)
print(X_train.shape)
print(X_test.shape)
print(Y_train.shape)
print(Y_test.shape)
```

Then we use the linear regression functionality from **Scikit-Learn**

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

lin_model = LinearRegression()
lin_model.fit(X_train, Y_train)

# model evaluation for training set

y_train_predict = lin_model.predict(X_train)
rmse = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
r2 = r2_score(Y_train, y_train_predict)

print("The model performance for training set")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
print("\n")

# model evaluation for testing set

y_test_predict = lin_model.predict(X_test)
# root mean square error of the model
rmse = (np.sqrt(mean_squared_error(Y_test, y_test_predict)))

# r-squared score of the model
r2 = r2_score(Y_test, y_test_predict)

print("The model performance for testing set")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))

# plotting the y_test vs y_pred
# ideally should have been a straight line
plt.scatter(Y_test, y_test_predict)
plt.show()
```

Reducing the number of degrees of freedom, overarching view

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we will see. This problem is often referred to as the curse of dimensionality. Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one.

Later we will discuss some of the most popular dimensionality reduction techniques: the principal component analysis (PCA), Kernel PCA, and Locally Linear Embedding (LLE).

Principal component analysis and its various variants deal with the problem of fitting a low-dimensional affine subspace to a set of of data points in a high-dimensional space. With its family of methods it is one of the most used tools in data modeling, compression and visualization.

Preprocessing our data

Before we proceed however, we will discuss how to preprocess our data. Till now and in connection with our previous examples we have not met so many cases where we are too sensitive to the scaling of 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

More preprocessing

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.

Simple preprocessing examples, Franke function and regression

```python
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.linear_model as skl
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import  train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler, Normalizer

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')
```

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


def create_X(x, y, n ):
        if len(x.shape) > 1:
                x = np.ravel(x)
                y = np.ravel(y)

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

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

        return X


# Making meshgrid of datapoints and compute Franke's function
n = 5
N = 1000
x = np.sort(np.random.uniform(0, 1, N))
y = np.sort(np.random.uniform(0, 1, N))
z = FrankeFunction(x, y)
X = create_X(x, y, n=n)
# split in training and test data
X_train, X_test, y_train, y_test = train_test_split(X,z,test_size=0.2)


clf = skl.LinearRegression().fit(X_train, y_train)

# The mean squared error and R2 score
print("MSE before scaling: {:.2f}".format(mean_squared_error(clf.predict(X_test), y_test)))
print("R2 score before scaling {:.2f}".format(clf.score(X_test,y_test)))

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Feature min values before scaling:\n {}".format(X_train.min(axis=0)))
print("Feature max values before scaling:\n {}".format(X_train.max(axis=0)))

print("Feature min values after scaling:\n {}".format(X_train_scaled.min(axis=0)))
print("Feature max values after scaling:\n {}".format(X_train_scaled.max(axis=0)))

clf = skl.LinearRegression().fit(X_train_scaled, y_train)


print("MSE after  scaling: {:.2f}".format(mean_squared_error(clf.predict(X_test_scaled), y_test))
print("R2 score for  scaled data: {:.2f}".format(clf.score(X_test_scaled,y_test)))
```

Beyond Ordinary Least Squares

Ridge and LASSO Regression

Let us remind ourselves about the expression for the standard Mean Squared Error (MSE) which we used to define our cost function and the equations for the ordinary least squares (OLS) method, that is our optimization problem is

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p}\frac{1}{n}\left\{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})\right\}.$$

or we can state it as

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p}\frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2=\frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2,$$

where we have used the definition of a norm-2 vector, that is

$$||\boldsymbol{x}||_2=\sqrt{\sum_i x_i^2}.$$

By minimizing the above equation with respect to the parameters $\boldsymbol{\beta}$ we could then obtain an analytical expression for the parameters $\boldsymbol{\beta}$. We can add a regularization parameter λ by defining a new cost function to be optimized, that is

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p}\frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2+\lambda||\boldsymbol{\beta}||_2^2$$

which leads to the Ridge regression minimization problem where we require that $||\boldsymbol{\beta}||_2^2\le t$, where t is a finite number larger than zero. By defining

$$C(\boldsymbol{X},\boldsymbol{\beta})=\frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2+\lambda||\boldsymbol{\beta}||_1,$$

we have a new optimization equation

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p}\frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2+\lambda||\boldsymbol{\beta}||_1$$

which leads to Lasso regression. Lasso stands for least absolute shrinkage and selection operator.

Here we have defined the norm-1 as

$$||\boldsymbol{x}||_1=\sum_i|x_i|.$$

More on Ridge Regression

Using the matrix-vector expression for Ridge regression (we drop the $1/n$ factor),

$$C(\boldsymbol{X},\boldsymbol{\beta}) = \left\{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})\right\} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta},$$

by taking the derivatives with respect to $\boldsymbol{\beta}$ we obtain

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0 = 2\boldsymbol{X}^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}\right) - 2\lambda\boldsymbol{\beta}.$$

We obtain a slightly modified matrix inversion problem which for finite values of λ does not suffer from singularity problems, that is

$$\boldsymbol{\beta}^{\mathrm{Ridge}} = \left(\boldsymbol{X}^T\boldsymbol{X}+\lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y},$$

with $\boldsymbol{I}$ being a $p\times p$ identity matrix with the constraint that

$$\sum_{i=0}^{p-1}\beta_i^2 \leq t,$$

with t a finite positive number.

We see that Ridge regression is nothing but the standard OLS with a modified diagonal term added to $\boldsymbol{X}^T\boldsymbol{X}$. The consequences, in particular for our discussion of the bias-variance tradeoff are rather interesting.

Simple Ridge interpretations

For the sake of simplicity, let us assume that the design matrix is orthonormal, that is

$$\boldsymbol{X}^T\boldsymbol{X} = (\boldsymbol{X}^T\boldsymbol{X})^{-1} = \boldsymbol{I}.$$

In this case the standard OLS results in

$$\boldsymbol{\beta}^{\mathrm{OLS}} = \boldsymbol{X}^T\boldsymbol{y} = \sum_{i=0}^{p-1}\boldsymbol{u}_j\boldsymbol{u}_j^T\boldsymbol{y},$$

and

$$\boldsymbol{\beta}^{\mathrm{Ridge}} = \left(\boldsymbol{I}+\lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y} = (1+\lambda)^{-1}\boldsymbol{\beta}^{\mathrm{OLS}},$$

that is the Ridge estimator scales the OLS estimator by the inverse of a factor $1+\lambda$, and the Ridge estimator converges to zero when the hyperparameter goes to infinity.

For more discussions of Ridge and Lasso regression, Wessel van Wieringen's article is highly recommended. Similarly, Mehta et al's article is also recommended.

Statistics

Where are we going?

Before we proceed, we need to rethink what we have been doing. In our eager to fit the data, we have omitted several important elements in our regression analysis. In what follows we will

1. remind ourselves about some statistical properties, including a discussion of mean values, variance and the so-called bias-variance tradeoff
2. introduce resampling techniques like cross-validation, bootstrapping and jackknife and more

This will allow us to link the standard linear algebra methods we have discussed above to a statistical interpretation of the methods.

Linking the regression analysis with a statistical interpretation

We are going to discuss several statistical properties which can be obtained in terms of analytical expressions. The advantage of doing linear regression is that we actually end up with analytical expressions for several statistical quantities. Standard least squares and Ridge regression allow us to derive quantities like the variance and other expectation values in a rather straightforward way.

It is assumed that $\varepsilon_i \sim \mathcal{N}(0,\sigma^2)$ and the ε_i are independent, i.e.:

$$\mathrm{Cov}(\varepsilon_{i_1},\varepsilon_{i_2}) = \left\{ \begin{array}{ll} \sigma^2 & \text{if} \quad i_1 = i_2, \\ 0 & \text{if} \quad i_1 \neq i_2. \end{array} \right.$$

The randomness of ε_i implies that $\mathbf{y}_i$ is also a random variable. In particular, $\mathbf{y}_i$ is normally distributed, because $\varepsilon_i \sim \mathcal{N}(0,\sigma^2)$ and $\mathbf{X}_{i,*}\,\boldsymbol{\beta}$ is a non-random scalar. To specify the parameters of the distribution of $\mathbf{y}_i$ we need to calculate its first two moments.

Recall that $\boldsymbol{X}$ is a matrix of dimensionality $n\times p$. The notation above $\mathbf{X}_{i,*}$ means that we are looking at the row number i and perform a sum over all values p.

Assumptions made

The assumption we have made here can be summarized as (and this is going to be useful when we discuss the bias-variance trade off) that there exists a function $f(\boldsymbol{x})$ and a normal distributed error $\boldsymbol{\varepsilon}\sim \mathcal{N}(0, \sigma^2)$ which describe our data

$$\boldsymbol{y} = f(\boldsymbol{x})+\boldsymbol{\varepsilon}$$

We approximate this function with our model from the solution of the linear regression equations, that is our function f is approximated by $\boldsymbol{\tilde{y}}$ where we want to minimize $(\boldsymbol{y}-\boldsymbol{\tilde{y}})^2$, our MSE, with

$$\boldsymbol{\tilde{y}} = \boldsymbol{X}\boldsymbol{\beta} \approx f(\boldsymbol{x}).$$

Note that we reserve the design matrix $\boldsymbol{X}$ to represent our specific rewrite of the input variables $\boldsymbol{x}$.

Expectation value and variance

We can calculate the expectation value of $\boldsymbol{y}$ for a given element i

$$\mathbb{E}(y_i) = \mathbb{E}(\mathbf{X}_{i,*}\,\boldsymbol{\beta}) + \mathbb{E}(\varepsilon_i) \quad = \quad \mathbf{X}_{i,*}\,\beta,$$

while its variance is

$$\begin{aligned}\mathrm{Var}(y_i) &= \mathbb{E}\{[y_i - \mathbb{E}(y_i)]^2\} \quad = \quad \mathbb{E}(y_i^2) - [\mathbb{E}(y_i)]^2 \\ &= \mathbb{E}[(\mathbf{X}_{i,*}\,\beta + \varepsilon_i)^2] - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\ &= \mathbb{E}[(\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\varepsilon_i \mathbf{X}_{i,*}\,\boldsymbol{\beta} + \varepsilon_i^2] - (\mathbf{X}_{i,*}\,\beta)^2 \\ &= (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\mathbb{E}(\varepsilon_i)\mathbf{X}_{i,*}\,\boldsymbol{\beta} + \mathbb{E}(\varepsilon_i^2) - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\ &= \mathbb{E}(\varepsilon_i^2) \quad = \quad \mathrm{Var}(\varepsilon_i) \quad = \quad \sigma^2.\end{aligned}$$

Hence, $y_i \sim \mathcal{N}(\mathbf{X}_{i,*}\,\boldsymbol{\beta}, \sigma^2)$, that is $\boldsymbol{y}$ follows a normal distribution with mean value $\boldsymbol{X\beta}$ and variance σ^2 (not be confused with the singular values of the SVD).

Expectation value and variance for $\boldsymbol{\beta}$

With the OLS expressions for the parameters $\boldsymbol{\beta}$ we can evaluate the expectation value

$$\mathbb{E}(\boldsymbol{\beta}) = \mathbb{E}[(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}] = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbb{E}[\mathbf{Y}] = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \boldsymbol{\beta}.$$

This means that the estimator of the regression parameters is unbiased.

We can also calculate the variance

The variance of $\boldsymbol{\beta}$ is

$$\begin{aligned}\mathrm{Var}(\boldsymbol{\beta}) &= \mathbb{E}\{[\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})][\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})]^T\} \\ &= \mathbb{E}\{[(\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{Y} - \boldsymbol{\beta}]\,[(\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{Y} - \boldsymbol{\beta}]^T\} \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbb{E}\{\mathbf{Y}\,\mathbf{Y}^T\}\,\mathbf{X}\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\,\mathbf{X}^T\,\{\mathbf{X}\,\boldsymbol{\beta}\,\boldsymbol{\beta}^T\,\mathbf{X}^T + \sigma^2\}\,\mathbf{X}\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \\ &= \boldsymbol{\beta}\,\boldsymbol{\beta}^T + \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^T \quad = \quad \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1},\end{aligned}$$

where we have used that $\mathbb{E}(\mathbf{Y}\mathbf{Y}^T) = \mathbf{X}\,\boldsymbol{\beta}\,\boldsymbol{\beta}^T\,\mathbf{X}^T + \sigma^2\,\mathbf{I}_{nn}$. From $\mathrm{Var}(\boldsymbol{\beta}) = \sigma^2\,(\mathbf{X}^T\mathbf{X})^{-1}$, one obtains an estimate of the variance of the estimate of the j-th regression coefficient: $\boldsymbol{\sigma}^2(\hat{\beta}_j) = \boldsymbol{\sigma}^2\sqrt{[(\mathbf{X}^T\mathbf{X})^{-1}]_{jj}}$. This may be used to construct a confidence interval for the estimates.

In a similar way, we can obtain analytical expressions for say the expectation values of the parameters $\boldsymbol{\beta}$ and their variance when we employ Ridge regression, allowing us again to define a confidence interval.

It is rather straightforward to show that

$$\mathbb{E}\big[\boldsymbol{\beta}^{\mathrm{Ridge}}\big] = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_{pp})^{-1}(\mathbf{X}^\top\mathbf{X})\boldsymbol{\beta}^{\mathrm{OLS}}.$$

We see clearly that $\mathbb{E}\big[\boldsymbol{\beta}^{\mathrm{Ridge}}\big] \neq \boldsymbol{\beta}^{\mathrm{OLS}}$ for any $\lambda > 0$. We say then that the ridge estimator is biased.

We can also compute the variance as

$$\mathrm{Var}[\boldsymbol{\beta}^{\mathrm{Ridge}}] = \sigma^2[\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}]^{-1}\mathbf{X}^T\mathbf{X}\{[\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}]^{-1}\}^T,$$

and it is easy to see that if the parameter λ goes to infinity then the variance of Ridge parameters $\boldsymbol{\beta}$ goes to zero.

With this, we can compute the difference

$$\mathrm{Var}[\boldsymbol{\beta}^{\mathrm{OLS}}] - \mathrm{Var}(\boldsymbol{\beta}^{\mathrm{Ridge}}) = \sigma^2[\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I}]^{-1}[2\lambda\mathbf{I}+\lambda^2(\mathbf{X}^T\mathbf{X})^{-1}]\{[\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I}]^{-1}\}^T.$$

The difference is non-negative definite since each component of the matrix product is non-negative definite. This means the variance we obtain with the standard OLS will always for $\lambda > 0$ be larger than the variance of $\boldsymbol{\beta}$ obtained with the Ridge estimator. This has interesting consequences when we discuss the so-called bias-variance trade-off tomorrow.

Why resampling methods

Before we proceed, we need to rethink what we have been doing. In our eager to fit the data, we have omitted several important elements in our regression analysis. In what follows we will

1. look at statistical properties, including a discussion of mean values, variance and the so-called bias-variance tradeoff
2. introduce resampling techniques like cross-validation, bootstrapping and jackknife and more

This will allow us to link the standard linear algebra methods we have discussed above to a statistical interpretation of the methods.

Resampling methods

Resampling methods are an indispensable tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. For example, in order to estimate the variability of a linear regression fit, we can repeatedly draw different samples from the training data, fit a linear regression to each new sample, and then examine the extent to which the resulting fits differ. Such an approach may allow us to obtain information that would not be available from fitting the model only once using the original training sample.

Two resampling methods are often used in Machine Learning analyses,

1. The **bootstrap method**
2. and **Cross-Validation**

In addition there are several other methods such as the Jackknife and the Blocking methods. We will discuss in particular cross-validation and the bootstrap method.

Resampling approaches can be computationally expensive

Resampling approaches can be computationally expensive, because they involve fitting the same statistical method multiple times using different subsets of the training data. However, due to recent advances in computing power, the computational requirements of resampling methods generally are not prohibitive. In this chapter, we discuss two of the most commonly used resampling methods, cross-validation and the bootstrap. Both methods are important tools in the practical application of many statistical learning procedures. For example, cross-validation can be used to estimate the test error associated with a given statistical learning method in order to evaluate its performance, or to select the appropriate level of flexibility. The process of evaluating a model's performance is known as model assessment, whereas the process of selecting the proper level of flexibility for a model is known as model selection. The bootstrap is widely used.

Why resampling methods ?

Statistical analysis.

- Our simulations can be treated as *computer experiments*. This is particularly the case for Monte Carlo methods
- The results can be analysed with the same statistical tools as we would use analysing experimental data.
- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

Statistical analysis

- As in other experiments, many numerical experiments have two classes of errors:
  - Statistical errors
  - Systematical errors
- Statistical errors can be estimated using standard tools from statistics
- Systematical errors are method specific and must be treated differently from case to case.

Resampling methods: Jackknife and Bootstrap

Two famous resampling methods are the **independent bootstrap** and **the jackknife**.

The jackknife is a special case of the independent bootstrap. Still, the jackknife was made popular prior to the independent bootstrap. And as the popularity of the independent bootstrap soared, new variants, such as **the dependent bootstrap**.

The Jackknife and independent bootstrap work for independent, identically distributed random variables. If these conditions are not satisfied, the methods will fail. Yet, it should be said that if the data are independent, identically distributed, and we only want to estimate the variance of $\overline{X}$ (which often is the case), then there is no need for bootstrapping.

Resampling methods: Jackknife

The Jackknife works by making many replicas of the estimator $\widehat{\theta}$. The jackknife is a resampling method where we systematically leave out one observation from the vector of observed values $\boldsymbol{x} = (x_1, x_2, \cdots, X_n)$. Let $\boldsymbol{x}_i$ denote the vector

$$\boldsymbol{x}_i = (x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n),$$

which equals the vector $\boldsymbol{x}$ with the exception that observation number i is left out. Using this notation, define $\widehat{\theta}_i$ to be the estimator $\widehat{\theta}$ computed using $\vec{X}_i$.

Jackknife code example

```python
from numpy import *
from numpy.random import randint, randn
from time import time

def jackknife(data, stat):
    n = len(data);t = zeros(n); inds = arange(n); t0 = time()
    ## 'jackknifing' by leaving out an observation for each i
    for i in range(n):
        t[i] = stat(delete(data,i) )

    # analysis
    print("Runtime: %g sec" % (time()-t0)); print("Jackknife Statistics :")
    print("original           bias      std. error")
    print("%8g %14g %15g" % (stat(data),(n-1)*mean(t)/n, (n*var(t))**.5))

    return t


# Returns mean of data samples
def stat(data):
    return mean(data)


mu, sigma = 100, 15
datapoints = 10000
x = mu + sigma*random.randn(datapoints)
```

```
# jackknife returns the data sample
t = jackknife(x, stat)
```

Resampling methods: Bootstrap

Bootstrapping is a nonparametric approach to statistical inference that substitutes computation for more traditional distributional assumptions and asymptotic results. Bootstrapping offers a number of advantages:

1. The bootstrap is quite general, although there are some cases in which it fails.

2. Because it does not require distributional assumptions (such as normally distributed errors), the bootstrap can provide more accurate inferences when the data are not well behaved or when the sample size is small.

3. It is possible to apply the bootstrap to statistics with sampling distributions that are difficult to derive, even asymptotically.

4. It is relatively simple to apply the bootstrap to complex data-collection plans (such as stratified and clustered samples).

Resampling methods: Bootstrap background

Since $\widehat{\theta} = \widehat{\theta}(\boldsymbol{X})$ is a function of random variables, $\widehat{\theta}$ itself must be a random variable. Thus it has a pdf, call this function $p(\boldsymbol{t})$. The aim of the bootstrap is to estimate $p(\boldsymbol{t})$ by the relative frequency of $\widehat{\theta}$. You can think of this as using a histogram in the place of $p(\boldsymbol{t})$. If the relative frequency closely resembles $p(\vec{t})$, then using numerics, it is straight forward to estimate all the interesting parameters of $p(\boldsymbol{t})$ using point estimators.

Resampling methods: More Bootstrap background

In the case that $\widehat{\theta}$ has more than one component, and the components are independent, we use the same estimator on each component separately. If the probability density function of X_i, $p(x)$, had been known, then it would have been straight forward to do this by:

1. Drawing lots of numbers from $p(x)$, suppose we call one such set of numbers $(X_1^*, X_2^*, \cdots, X_n^*)$.

2. Then using these numbers, we could compute a replica of $\widehat{\theta}$ called $\widehat{\theta}^*$.

By repeated use of (1) and (2), many estimates of $\widehat{\theta}$ could have been obtained. The idea is to use the relative frequency of $\widehat{\theta}^*$ (think of a histogram) as an estimate of $p(\boldsymbol{t})$.

Resampling methods: Bootstrap approach

But unless there is enough information available about the process that generated $X_1, X_2, \cdots, X_n$, $p(x)$ is in general unknown. Therefore, Efron in 1979 asked the question: What if we replace $p(x)$ by the relative frequency of the observation X_i; if we draw observations in accordance with the relative frequency of the observations, will we obtain the same result in some asymptotic sense? The answer is yes.

Instead of generating the histogram for the relative frequency of the observation X_i, just draw the values $(X_1^*, X_2^*, \cdots, X_n^*)$ with replacement from the vector $\boldsymbol{X}$.

Resampling methods: Bootstrap steps

The independent bootstrap works like this:

1. Draw with replacement n numbers for the observed variables $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$.
2. Define a vector $\boldsymbol{x}^*$ containing the values which were drawn from $\boldsymbol{x}$.
3. Using the vector $\boldsymbol{x}^*$ compute $\widehat{\theta}^*$ by evaluating $\widehat{\theta}$ under the observations $\boldsymbol{x}^*$.
4. Repeat this process k times.

When you are done, you can draw a histogram of the relative frequency of $\widehat{\theta}^*$. This is your estimate of the probability distribution $p(t)$. Using this probability distribution you can estimate any statistics thereof. In principle you never draw the histogram of the relative frequency of $\widehat{\theta}^*$. Instead you use the estimators corresponding to the statistic of interest. For example, if you are interested in estimating the variance of $\widehat{\theta}$, apply the etsimator $\widehat{\sigma}^2$ to the values $\widehat{\theta}^*$.

Code example for the Bootstrap method

The following code starts with a Gaussian distribution with mean value $\mu = 100$ and variance $\sigma = 15$. We use this to generate the data used in the bootstrap analysis. The bootstrap analysis returns a data set after a given number of bootstrap operations (as many as we have data points). This data set consists of estimated mean values for each bootstrap operation. The histogram generated by the bootstrap method shows that the distribution for these mean values is also a Gaussian, centered around the mean value $\mu = 100$ but with standard deviation $\sigma/\sqrt{n}$, where n is the number of bootstrap samples (in this case the same as the number of original data points). The value of the standard deviation is what we expect from the central limit theorem.

```
from numpy import *
from numpy.random import randint, randn
from time import time
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

# Returns mean of bootstrap samples
def stat(data):
    return mean(data)

# Bootstrap algorithm
def bootstrap(data, statistic, R):
    t = zeros(R); n = len(data); inds = arange(n); t0 = time()
    # non-parametric bootstrap
    for i in range(R):
        t[i] = statistic(data[randint(0,n,n)])

    # analysis
    print("Runtime: %g sec" % (time()-t0)); print("Bootstrap Statistics :")
    print("original           bias      std. error")
    print("%8g %8g %14g %15g" % (statistic(data), std(data),mean(t),std(t)))
    return t


mu, sigma = 100, 15
datapoints = 10000
x = mu + sigma*random.randn(datapoints)
# bootstrap returns the data sample
t = bootstrap(x, stat, datapoints)
# the histogram of the bootstrapped  data
n, binsboot, patches = plt.hist(t, 50, normed=1, facecolor='red', alpha=0.75)

# add a 'best fit' line
y = mlab.normpdf( binsboot, mean(t), std(t))
lt = plt.plot(binsboot, y, 'r--', linewidth=1)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.axis([99.5, 100.6, 0, 3.0])
plt.grid(True)

plt.show()
```

Various steps in cross-validation

When the repetitive splitting of the data set is done randomly, samples may accidently end up in a fast majority of the splits in either training or test set. Such samples may have an unbalanced influence on either model building or prediction evaluation. To avoid this k-fold cross-validation structures the data splitting. The samples are divided into k more or less equally sized exhaustive and mutually exclusive subsets. In turn (at each split) one of these subsets plays the role of the test set while the union of the remaining subsets constitutes the training set. Such a splitting warrants a balanced representation of each sample in both training and test set over the splits. Still the division into the k subsets involves a degree of randomness. This may be fully excluded when choosing $k = n$. This particular case is referred to as leave-one-out cross-validation (LOOCV).

Cross-validation in brief

For the various values of k

1. shuffle the dataset randomly.
2. Split the dataset into k groups.
3. For each unique group:
   (a) Decide which group to use as set for test data
   (b) Take the remaining groups as a training data set
   (c) Fit a model on the training set and evaluate it on the test set
   (d) Retain the evaluation score and discard the model
4. Summarize the model using the sample of model evaluation scores

Code Example for Cross-validation and k-fold Cross-validation

The code here uses Ridge regression with cross-validation (CV) resampling and k-fold CV in order to fit a specific polynomial.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import PolynomialFeatures

# A seed just to ensure that the random numbers are the same for every run.
# Useful for eventual debugging.
np.random.seed(3155)

# Generate the data.
nsamples = 100
x = np.random.randn(nsamples)
y = 3*x**2 + np.random.randn(nsamples)

## Cross-validation on Ridge regression using KFold only

# Decide degree on polynomial to fit
poly = PolynomialFeatures(degree = 6)

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

# Initialize a KFold instance
k = 5
kfold = KFold(n_splits = k)

# Perform the cross-validation to estimate MSE
scores_KFold = np.zeros((nlambdas, k))
```

```
i = 0
for lmb in lambdas:
    ridge = Ridge(alpha = lmb)
    j = 0
    for train_inds, test_inds in kfold.split(x):
        xtrain = x[train_inds]
        ytrain = y[train_inds]

        xtest = x[test_inds]
        ytest = y[test_inds]

        Xtrain = poly.fit_transform(xtrain[:, np.newaxis])
        ridge.fit(Xtrain, ytrain[:, np.newaxis])

        Xtest = poly.fit_transform(xtest[:, np.newaxis])
        ypred = ridge.predict(Xtest)

        scores_KFold[i,j] = np.sum((ypred - ytest[:, np.newaxis])**2)/np.size(ypred)

        j += 1
    i += 1


estimated_mse_KFold = np.mean(scores_KFold, axis = 1)

## Cross-validation using cross_val_score from sklearn along with KFold

# kfold is an instance initialized above as:
# kfold = KFold(n_splits = k)

estimated_mse_sklearn = np.zeros(nlambdas)
i = 0
for lmb in lambdas:
    ridge = Ridge(alpha = lmb)

    X = poly.fit_transform(x[:, np.newaxis])
    estimated_mse_folds = cross_val_score(ridge, X, y[:, np.newaxis], scoring='neg_mean_squared_er

    # cross_val_score return an array containing the estimated negative mse for every fold.
    # we have to the the mean of every array in order to get an estimate of the mse of the model
    estimated_mse_sklearn[i] = np.mean(-estimated_mse_folds)

    i += 1

## Plot and compare the slightly different ways to perform cross-validation

plt.figure()

plt.plot(np.log10(lambdas), estimated_mse_sklearn, label = 'cross_val_score')
plt.plot(np.log10(lambdas), estimated_mse_KFold, 'r--', label = 'KFold')

plt.xlabel('log10(lambda)')
plt.ylabel('mse')

plt.legend()

plt.show()
```

The bias-variance tradeoff

We will discuss the bias-variance tradeoff in the context of continuous predictions such as regression. However, many of the intuitions and ideas discussed here also carry over to classification tasks. Consider a dataset $\mathcal{L}$ consisting of the data $\mathbf{X}_{\mathcal{L}} = \{(y_j, \boldsymbol{x}_j), j = 0 \ldots n-1\}$.

Let us assume that the true data is generated from a noisy model

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$$

where ϵ is normally distributed with mean zero and standard deviation σ^2.

In our derivation of the ordinary least squares method we defined then an approximation to the function f in terms of the parameters $\boldsymbol{\beta}$ and the design matrix $\boldsymbol{X}$ which embody our model, that is $\boldsymbol{\tilde{y}} = \boldsymbol{X}\boldsymbol{\beta}$.

Thereafter we found the parameters $\boldsymbol{\beta}$ by optimizing the means squared error via the so-called cost function

$$C(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \boldsymbol{\tilde{y}})^2\right].$$

We can rewrite this as

$$\mathbb{E}\left[(\boldsymbol{y} - \boldsymbol{\tilde{y}})^2\right] = \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\boldsymbol{\tilde{y}}\right])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}\left[\boldsymbol{\tilde{y}}\right])^2 + \sigma^2.$$

The three terms represent the square of the bias of the learning method, which can be thought of as the error caused by the simplifying assumptions built into the method. The second term represents the variance of the chosen model and finally the last terms is variance of the error $\boldsymbol{\epsilon}$.

To derive this equation, we need to recall that the variance of $\boldsymbol{y}$ and $\boldsymbol{\epsilon}$ are both equal to σ^2. The mean value of $\boldsymbol{\epsilon}$ is by definition equal to zero. Furthermore, the function f is not a stochastics variable, idem for $\boldsymbol{\tilde{y}}$. We use a more compact notation in terms of the expectation value

$$\mathbb{E}\left[(\boldsymbol{y} - \boldsymbol{\tilde{y}})^2\right] = \mathbb{E}\left[(\boldsymbol{f} + \boldsymbol{\epsilon} - \boldsymbol{\tilde{y}})^2\right],$$

and adding and subtracting $\mathbb{E}\left[\boldsymbol{\tilde{y}}\right]$ we get

$$\mathbb{E}\left[(\boldsymbol{y} - \boldsymbol{\tilde{y}})^2\right] = \mathbb{E}\left[(\boldsymbol{f} + \boldsymbol{\epsilon} - \boldsymbol{\tilde{y}} + \mathbb{E}\left[\boldsymbol{\tilde{y}}\right] - \mathbb{E}\left[\boldsymbol{\tilde{y}}\right])^2\right],$$

which, using the abovementioned expectation values can be rewritten as

$$\mathbb{E}\left[(\boldsymbol{y} - \boldsymbol{\tilde{y}})^2\right] = \mathbb{E}\left[(\boldsymbol{y} - \mathbb{E}\left[\boldsymbol{\tilde{y}}\right])^2\right] + \mathrm{Var}\left[\boldsymbol{\tilde{y}}\right] + \sigma^2,$$

that is the rewriting in terms of the so-called bias, the variance of the model $\boldsymbol{\tilde{y}}$ and the variance of $\boldsymbol{\epsilon}$.

Overfitting, Training and Test data before bias-variance trade-off analysis

Let us just analyze the means-squared error as function of model complexity by comparing the training data against the test data. The first example does also not contain any resampling.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline

np.random.seed(2018)

n = 80
maxdegree = 18
# preparing for the next example
n_boostraps = 1

# 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)
TrainMSE = np.zeros(maxdegree)
TestMSE = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
# Split data in train and test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
for degree in range(maxdegree):
    model = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression(fit_intercept=False
    y_TrainPred = np.empty((y_train.shape[0], n_boostraps))
    y_TestPred = np.empty((y_test.shape[0], n_boostraps))
    y_TrainPred[:, i] = model.fit(x_train, y_train).predict(x_train).ravel()
    y_TestPred[:, i] = model.predict(x_test).ravel()

    polydegree[degree] = degree
    TestMSE[degree] = np.mean( np.mean((y_test - y_TestPred)**2, axis=1, keepdims=True) )
    TrainMSE[degree] = np.mean( np.mean((y_train - y_TrainPred)**2, axis=1, keepdims=True) )
    print('Polynomial degree:', degree)
    print('Training MSE:', TrainMSE[degree])
    print('Test MSE:', TestMSE[degree])

plt.plot(polydegree, TrainMSE, label='Train MSE')
plt.plot(polydegree, TestMSE, label='Test MSE')
plt.legend()
plt.show()
```

Understanding what happens

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample

np.random.seed(2018)
```

```
n = 40
n_boostraps = 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)
error = np.zeros(maxdegree)
bias = np.zeros(maxdegree)
variance = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

for degree in range(maxdegree):
    model = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression(fit_intercept=False)
    y_pred = np.empty((y_test.shape[0], n_boostraps))
    for i in range(n_boostraps):
        x_, y_ = resample(x_train, y_train)
        y_pred[:, i] = model.fit(x_, y_).predict(x_test).ravel()

    polydegree[degree] = degree
    error[degree] = np.mean( np.mean((y_test - y_pred)**2, axis=1, keepdims=True) )
    bias[degree] = np.mean( (y_test - np.mean(y_pred, axis=1, keepdims=True))**2 )
    variance[degree] = np.mean( np.var(y_pred, axis=1, keepdims=True) )
    print('Polynomial degree:', degree)
    print('Error:', error[degree])
    print('Bias^2:', bias[degree])
    print('Var:', variance[degree])
    print('{} >= {} + {} = {}'.format(error[degree], bias[degree], variance[degree], bias[degree]

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

Summing up

The bias-variance tradeoff summarizes the fundamental tension in machine learning, particularly supervised learning, between the complexity of a model and the amount of training data needed to train it. Since data is often limited, in practice it is often useful to use a less-complex model with higher bias, that is a model whose asymptotic performance is worse than another model because it is easier to train and less sensitive to sampling noise arising from having a finite-sized training dataset (smaller variance).

The above equations tell us that in order to minimize the expected test error, we need to select a statistical learning method that simultaneously achieves low variance and low bias. Note that variance is inherently a nonnegative quantity, and squared bias is also nonnegative. Hence, we see that the expected test MSE can never lie below $Var(\epsilon)$, the irreducible error.

What do we mean by the variance and bias of a statistical learning method? The variance refers to the amount by which our model would change if we estimated it using a different training data set. Since the training data are used to fit the statistical learning method, different training data sets will result in a different estimate. But ideally the estimate for our model should not vary too much between training sets. However, if a method has high variance then small changes in the training data can result in large changes in the model. In general, more flexible statistical methods have higher variance.

You may also find this recent article of interest.

Another Example from Scikit-Learn's Repository

```
"""
============================
Underfitting vs. Overfitting
============================

This example demonstrates the problems of underfitting and overfitting and
how we can use linear regression with polynomial features to approximate
nonlinear functions. The plot shows the function that we want to approximate,
which is a part of the cosine function. In addition, the samples from the
real function and the approximations of different models are displayed. The
models have polynomial features of different degrees. We can see that a
linear function (polynomial with degree 1) is not sufficient to fit the
training samples. This is called **underfitting**. A polynomial of degree 4
approximates the true function almost perfectly. However, for higher degrees
the model will **overfit** the training data, i.e. it learns the noise of the
training data.
We evaluate quantitatively **overfitting** / **underfitting** by using
cross-validation. We calculate the mean squared error (MSE) on the validation
set, the higher, the less likely the model generalizes correctly from the
training data.
"""

print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score


def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                             include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_features),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y,
                             scoring="neg_mean_squared_error", cv=10)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))
plt.show()
```

More examples on bootstrap and cross-validation and errors

```
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.model_selection import train_test_split
from sklearn.utils import resample
from sklearn.metrics import mean_squared_error
# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("EoS.csv"),'r')
```

```python
# Read the EoS data as  csv file and organize the data into two arrays with density and energies
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
Density = EoS['Density']
#  The design matrix now as function of various polytrops

Maxpolydegree = 30
X = np.zeros((len(Density),Maxpolydegree))
X[:,0] = 1.0
testerror = np.zeros(Maxpolydegree)
trainingerror = np.zeros(Maxpolydegree)
polynomial = np.zeros(Maxpolydegree)

trials = 100
for polydegree in range(1, Maxpolydegree):
    polynomial[polydegree] = polydegree
    for degree in range(polydegree):
        X[:,degree] = Density**(degree/3.0)

# loop over trials in order to estimate the expectation value of the MSE
    testerror[polydegree] = 0.0
    trainingerror[polydegree] = 0.0
    for samples in range(trials):
        x_train, x_test, y_train, y_test = train_test_split(X, Energies, test_size=0.2)
        model = LinearRegression(fit_intercept=True).fit(x_train, y_train)
        ypred = model.predict(x_train)
        ytilde = model.predict(x_test)
        testerror[polydegree] += mean_squared_error(y_test, ytilde)
        trainingerror[polydegree] += mean_squared_error(y_train, ypred)

    testerror[polydegree] /= trials
    trainingerror[polydegree] /= trials
    print("Degree of polynomial: %3d"% polynomial[polydegree])
    print("Mean squared error on training data: %.8f" % trainingerror[polydegree])
    print("Mean squared error on test data: %.8f" % testerror[polydegree])

plt.plot(polynomial, np.log10(trainingerror), label='Training Error')
plt.plot(polynomial, np.log10(testerror), label='Test Error')
plt.xlabel('Polynomial degree')
plt.ylabel('log10[MSE]')
plt.legend()
plt.show()
```

The same example but now with cross-validation

```python
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score


# Where to save the figures and data files
```

```
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("EoS.csv"),'r')

# Read the EoS data as  csv file and organize the data into two arrays with density and energies
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
Density = EoS['Density']
#  The design matrix now as function of various polytrops

Maxpolydegree = 30
X = np.zeros((len(Density),Maxpolydegree))
X[:,0] = 1.0
estimated_mse_sklearn = np.zeros(Maxpolydegree)
polynomial = np.zeros(Maxpolydegree)
k =5
kfold = KFold(n_splits = k)

for polydegree in range(1, Maxpolydegree):
    polynomial[polydegree] = polydegree
    for degree in range(polydegree):
        X[:,degree] = Density**(degree/3.0)
        OLS = LinearRegression()
# loop over trials in order to estimate the expectation value of the MSE
    estimated_mse_folds = cross_val_score(OLS, X, Energies, scoring='neg_mean_squared_error', cv=k
#[:, np.newaxis]
    estimated_mse_sklearn[polydegree] = np.mean(-estimated_mse_folds)

plt.plot(polynomial, np.log10(estimated_mse_sklearn), label='Test Error')
plt.xlabel('Polynomial degree')
plt.ylabel('log10[MSE]')
plt.legend()
plt.show()
```

Cross-validation with Ridge

```
import numpy as np
import matplotlib.pyplot as plt
```

```python
from sklearn.model_selection import KFold
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import PolynomialFeatures

# A seed just to ensure that the random numbers are the same for every run.
np.random.seed(3155)
# Generate the data.
n = 100
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)
# Decide degree on polynomial to fit
poly = PolynomialFeatures(degree = 10)

# Decide which values of lambda to use
nlambdas = 500
lambdas = np.logspace(-3, 5, nlambdas)
# Initialize a KFold instance
k = 5
kfold = KFold(n_splits = k)
estimated_mse_sklearn = np.zeros(nlambdas)
i = 0
for lmb in lambdas:
    ridge = Ridge(alpha = lmb)
    estimated_mse_folds = cross_val_score(ridge, x, y, scoring='neg_mean_squared_error', cv=kfold
    estimated_mse_sklearn[i] = np.mean(-estimated_mse_folds)
    i += 1
plt.figure()
plt.plot(np.log10(lambdas), estimated_mse_sklearn, label = 'cross_val_score')
plt.xlabel('log10(lambda)')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

*

Exercise 1: making your own data and exploring scikit-learn

We will 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 $N(0,1)$. The following simple Python instructions define our x and y values (with 100 data points).

```python
x = np.random.rand(100,1)
y = 2.0+5*x*x+0.1*np.random.randn(100,1)
```

1. Write your own code (following the examples under the regression slides) for computing the parametrization of the data set fitting a second-order polynomial.

2. Use thereafter **scikit-learn** (see again the examples in the regression slides) and compare with your own code.

3. Using scikit-learn, compute also the mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error defined as

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

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

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

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

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

You can use the functionality included in scikit-learn. If you feel for it, you can use your own program and define functions which compute the above two functions. Discuss the meaning of these results. Try also to vary the coefficient in front of the added stochastic noise term and discuss the quality of the fits.

Solution. The code here is an example of where we define our own design matrix and fit parameters β.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

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

x = np.random.rand(100)
y = 2.0+5*x*x+0.1*np.random.randn(100)


#  The design matrix now as function of a given polynomial
X = np.zeros((len(x),3))
X[:,0] = 1.0
X[:,1] = x
X[:,2] = x**2
# We split the data in test and training data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# matrix inversion to find beta
beta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
print(beta)
# and then make the prediction
ytilde = X_train @ beta
print("Training R2")
print(R2(y_train,ytilde))
print("Training MSE")
print(MSE(y_train,ytilde))
ypredict = X_test @ beta
print("Test R2")
print(R2(y_test,ypredict))
print("Test MSE")
print(MSE(y_test,ypredict))
```

*

Exercise 2: mean values and variances in linear regression

This exercise deals with various mean values ad variances in linear regression method (here it may be useful to look up chapter 3, equation (3.8) of Trevor Hastie, Robert Tibshirani, Jerome H. Friedman, The Elements of Statistical Learning, Springer).

The assumption we have made is that there exists a function $f(\boldsymbol{x})$ and a normal distributed error $\boldsymbol{\varepsilon} \sim \mathcal{N}(0, \sigma^2)$ which describes our data

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$$

We then approximate this function with our model from the solution of the linear regression equations (ordinary least squares OLS), that is our function f is approximated by $\tilde{\boldsymbol{y}}$ where we minimized $(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2$, with

$$f(\boldsymbol{x}) \approx \tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta}.$$

The matrix $\boldsymbol{X}$ is the so-called design matrix.

aragraph!paragraph>paragraph>-0.5em

a) Show that the expected value of $\boldsymbol{y}$ for a given element i

$$\mathbb{E}(y_i) = \mathbf{X}_{i,*}\,\beta,$$

and that its variance is

$$\mathrm{Var}(y_i) = \sigma^2.$$

Hence, $y_i \sim \mathcal{N}(\mathbf{X}_{i,*}\,\boldsymbol{\beta}, \sigma^2)$, that is $\boldsymbol{y}$ follows a normal distribution with mean value $\boldsymbol{X}\boldsymbol{\beta}$ and variance σ^2.

Solution. We can calculate the expected value of $\boldsymbol{y}$ for a given element i

$$\mathbb{E}(y_i) = \mathbb{E}(\mathbf{X}_{i,*}\,\boldsymbol{\beta}) + \mathbb{E}(\varepsilon_i) \quad = \quad \mathbf{X}_{i,*}\,\beta,$$

while its variance is

$$\begin{aligned} \mathrm{Var}(y_i) &= \mathbb{E}\{[y_i - \mathbb{E}(y_i)]^2\} \quad = \quad \mathbb{E}(y_i^2) - [\mathbb{E}(y_i)]^2 \\ &= \mathbb{E}[(\mathbf{X}_{i,*}\,\beta + \varepsilon_i)^2] - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\ &= \mathbb{E}[(\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\varepsilon_i\mathbf{X}_{i,*}\,\boldsymbol{\beta} + \varepsilon_i^2] - (\mathbf{X}_{i,*}\,\beta)^2 \\ &= (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 + 2\mathbb{E}(\varepsilon_i)\mathbf{X}_{i,*}\,\boldsymbol{\beta} + \mathbb{E}(\varepsilon_i^2) - (\mathbf{X}_{i,*}\,\boldsymbol{\beta})^2 \\ &= \mathbb{E}(\varepsilon_i^2) \quad = \quad \mathrm{Var}(\varepsilon_i) \quad = \quad \sigma^2. \end{aligned}$$

Hence, $y_i \sim \mathcal{N}(\mathbf{X}_{i,*}\,\boldsymbol{\beta}, \sigma^2)$, that is $\boldsymbol{y}$ follows a normal distribution with mean value $\boldsymbol{X}\boldsymbol{\beta}$ and variance σ^2 (not be confused with the singular values of the SVD).
aragraph!paragraph>paragraph>-0.5em

b) With the OLS expressions for the optimal parameters $\boldsymbol{\beta}^{\mathrm{opt}}$ show that

$$\mathbb{E}(\boldsymbol{\beta}^{\mathrm{opt}}) = \boldsymbol{\beta}.$$

Solution.

$$\mathbb{E}(\boldsymbol{\beta}^{\mathrm{opt}}) = \mathbb{E}[(\mathbf{X}^{\top}\mathbf{X})^{-1}\mathbf{X}^{T}\mathbf{Y}] = (\mathbf{X}^{T}\mathbf{X})^{-1}\mathbf{X}^{T}\mathbb{E}[\mathbf{Y}] = (\mathbf{X}^{T}\mathbf{X})^{-1}\mathbf{X}^{T}\mathbf{X}\boldsymbol{\beta} = \boldsymbol{\beta}.$$

This means that the estimator of the regression parameters is unbiased.
aragraph!paragraph>paragraph>-0.5em

c) Show finally that the variance of $\boldsymbol{\beta}$ is

$$\mathrm{Var}(\boldsymbol{\beta}^{\mathrm{opt}}) \quad = \quad \sigma^2\,(\mathbf{X}^{T}\mathbf{X})^{-1}.$$

Solution. The variance of $\boldsymbol{\beta}$ is

$$\begin{aligned}
\mathrm{Var}(\boldsymbol{\beta}^{\mathrm{opt}}) &= \mathbb{E}\{[\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})][\boldsymbol{\beta} - \mathbb{E}(\boldsymbol{\beta})]^{T}\} \\
&= \mathbb{E}\{[(\mathbf{X}^{T}\mathbf{X})^{-1}\,\mathbf{X}^{T}\mathbf{Y} - \boldsymbol{\beta}]\,[(\mathbf{X}^{T}\mathbf{X})^{-1}\,\mathbf{X}^{T}\mathbf{Y} - \boldsymbol{\beta}]^{T}\} \\
&= (\mathbf{X}^{T}\mathbf{X})^{-1}\,\mathbf{X}^{T}\,\mathbb{E}\{\mathbf{Y}\,\mathbf{Y}^{T}\}\,\mathbf{X}\,(\mathbf{X}^{T}\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^{T} \\
&= (\mathbf{X}^{T}\mathbf{X})^{-1}\,\mathbf{X}^{T}\,\{\mathbf{X}\,\boldsymbol{\beta}\,\boldsymbol{\beta}^{T}\,\mathbf{X}^{T} + \sigma^2\}\,\mathbf{X}\,(\mathbf{X}^{T}\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^{T} \\
&= \boldsymbol{\beta}\,\boldsymbol{\beta}^{T} + \sigma^2\,(\mathbf{X}^{T}\mathbf{X})^{-1} - \boldsymbol{\beta}\,\boldsymbol{\beta}^{T} \quad = \quad \sigma^2\,(\mathbf{X}^{T}\mathbf{X})^{-1},
\end{aligned}$$

where we have used that $\mathbb{E}(\mathbf{Y}\mathbf{Y}^{T}) = \mathbf{X}\,\boldsymbol{\beta}\,\boldsymbol{\beta}^{T}\,\mathbf{X}^{T} + \sigma^2\,\mathbf{I}_{nn}$. From $\mathrm{Var}(\boldsymbol{\beta}) = \sigma^2\,(\mathbf{X}^{T}\mathbf{X})^{-1}$, one obtains an estimate of the variance of the estimate of the j-th regression coefficient: $\boldsymbol{\sigma}^2(\hat{\beta}_j) = \boldsymbol{\sigma}^2\sqrt{[(\mathbf{X}^{T}\mathbf{X})^{-1}]_{jj}}$. This may be used to construct a confidence interval for the estimates.

In a similar way, we can obtain analytical expressions for say the expectation values of the parameters $\boldsymbol{\beta}$ and their variance when we employ Ridge regression, allowing us again to define a confidence interval.

*

Exercise 3: Adding Ridge and Lasso Regression
This exercise is a continuation of exercise 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)
y = 2.0+5*x*x+0.1*np.random.randn(100)
```

aragraph!paragraph>paragraph>-0.5em

a) 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 3. Study the dependence on λ while also varying the strength of the noise in your expression for $y(x)$.

Solution. The code here allows you to perform your own Ridge calculation and perform calculations for various values of the regularization parameter λ. This program can easily be extended upon.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

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


# A seed just to ensure that the random numbers are the same for every run.
# Useful for eventual debugging.
np.random.seed(3155)

x = np.random.rand(100)
y = 2.0+5*x*x+0.1*np.random.randn(100)

# number of features p (here degree of polynomial
p = 3
#  The design matrix now as function of a given polynomial
X = np.zeros((len(x),p))
X[:,0] = 1.0
X[:,1] = x
X[:,2] = x*x
# We split the data in test and training data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# matrix inversion to find beta
OLSbeta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
```

```python
print(OLSbeta)
# and then make the prediction
ytildeOLS = X_train @ OLSbeta
print("Training R2 for OLS")
print(R2(y_train,ytildeOLS))
print("Training MSE for OLS")
print(MSE(y_train,ytildeOLS))
ypredictOLS = X_test @ OLSbeta
print("Test R2 for OLS")
print(R2(y_test,ypredictOLS))
print("Test MSE OLS")
print(MSE(y_test,ypredictOLS))

# Repeat now for Ridge regression and various values of the regularization parameter
I = np.eye(p,p)
# Decide which values of lambda to use
nlambdas = 20
MSEPredict = np.zeros(nlambdas)
MSETrain = np.zeros(nlambdas)
lambdas = np.logspace(-4, 1, nlambdas)
for i in range(nlambdas):
    lmb = lambdas[i]
    Ridgebeta = np.linalg.inv(X_train.T @ X_train+lmb*I) @ X_train.T @ y_train
    # and then make the prediction
    ytildeRidge = X_train @ Ridgebeta
    ypredictRidge = X_test @ Ridgebeta
    MSEPredict[i] = MSE(y_test,ypredictRidge)
    MSETrain[i] = MSE(y_train,ytildeRidge)
# Now plot the resulys
plt.figure()
plt.plot(np.log10(lambdas), MSETrain, label = 'MSE Ridge train')
plt.plot(np.log10(lambdas), MSEPredict, 'r--', label = 'MSE Ridge Test')
plt.xlabel('log10(lambda)')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

aragraph!paragraph>paragraph>-0.5em

b) 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.

Solution. To use **scikit-learn** with Ridge, we simply need to add the relevant function **Ridge()**, as done in the code here.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import sklearn.linear_model as skl

def R2(y_data, y_model):
    return 1 - np.sum((y_data - y_model) ** 2) / np.sum((y_data - np.mean(y_data)) ** 2)
def MSE(y_data,y_model):
    n = np.size(y_model)
    return np.sum((y_data-y_model)**2)/n
```

```
# A seed just to ensure that the random numbers are the same for every run.
# Useful for eventual debugging.
np.random.seed(3155)

x = np.random.rand(100)
y = 2.0+5*x*x+0.1*np.random.randn(100)

# number of features p (here degree of polynomial
p = 3
#  The design matrix now as function of a given polynomial
X = np.zeros((len(x),p))
X[:,0] = 1.0
X[:,1] = x
X[:,2] = x*x
# We split the data in test and training data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# matrix inversion to find beta
OLSbeta = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
print(OLSbeta)
# and then make the prediction
ytildeOLS = X_train @ OLSbeta
print("Training R2 for OLS")
print(R2(y_train,ytildeOLS))
print("Training MSE for OLS")
print(MSE(y_train,ytildeOLS))
ypredictOLS = X_test @ OLSbeta
print("Test R2 for OLS")
print(R2(y_test,ypredictOLS))
print("Test MSE OLS")
print(MSE(y_test,ypredictOLS))

# Repeat now for Ridge regression and various values of the regularization parameter
I = np.eye(p,p)
# Decide which values of lambda to use
nlambdas = 100
MSEPredict = np.zeros(nlambdas)
MSEPredictSKL = np.zeros(nlambdas)
MSETrain = np.zeros(nlambdas)
lambdas = np.logspace(-4, 0, nlambdas)
for i in range(nlambdas):
    lmb = lambdas[i]
    # add ridge
    clf_ridge = skl.Ridge(alpha=lmb).fit(X_train, y_train)
    yridge = clf_ridge.predict(X_test)
    Ridgebeta = np.linalg.inv(X_train.T @ X_train+lmb*I) @ X_train.T @ y_train
    # and then make the prediction
    ytildeRidge = X_train @ Ridgebeta
    ypredictRidge = X_test @ Ridgebeta
    MSEPredict[i] = MSE(y_test,ypredictRidge)
    MSEPredictSKL[i] = MSE(y_test,yridge)
    MSETrain[i] = MSE(y_train,ytildeRidge)
#then plot the results
plt.figure()
plt.plot(np.log10(lambdas), MSETrain, label = 'MSE Ridge train')
plt.plot(np.log10(lambdas), MSEPredict, 'r--', label = 'MSE Ridge Test')
plt.plot(np.log10(lambdas), MSEPredictSKL, 'g--', label = 'MSE Ridge sickit-learn Test')
plt.xlabel('log10(lambda)')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

aragraph!paragraph>paragraph>-0.5em

c) 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 parameters (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 figures 3.10 and 3.11. **Scikit-Learn** may not provide the variance of the parameters β. This needs to be checked. With your own code you can however do so.

Solution. import numpy as np import pandas as pd import matplotlib.pyplot as plt from sklearn.model$_selectionimporttrain_test_splitfromsklearn.preprocessingimportStandardScalerimpo$

def R2(y$_data, y_model) : return1-np.sum((y_data-y_model)**2)/np.sum((y_data-np.mean(y_data))**2)defMSE(y_data, y_model) : n = np.size(y_model)returnnp.sum((y_data-y_model)**2)/n$

np.random.seed(3155)

x = np.random.rand(100) y = 2.0+5*x*x+0.1*np.random.randn(100)

p = 3 X = np.zeros((len(x),p)) X[:,0] = 1.0 X[:,1] = x X[:,2] = x*x X$_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)scaler = StandardScaler()scaler.fit(X_train)X_train_scaled = scaler.transform(X_train)X_test_scaled = scaler.transform(X_test)$

OLSbeta = np.linalg.inv(X$_train.T@X_train)@X_train.T@y_trainprint(OLSbeta)print(np.linalg.inv(X_train.$

I = np.eye(p,p) nlambdas = 10 MSEPredict = np.zeros(nlambdas) MSEPredictSKL = np.zeros(nlambdas) MSETrain = np.zeros(nlambdas) lambdas = np.logspace(-4, 0, nlambdas) for i in range(nlambdas): lmb = lambdas[i] Ridgebeta = np.linalg.inv(X$_train.T@X_train+lmb*I)@X_train.T@y_trainprint(np.linalg.inv(X_train.T@X_train+lmb*I))$

aragraph!paragraph>paragraph>-0.5em

d) 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. Also, you do not need to compute the variance of the parameters β but you can extract their values and study their behavior as functions of the regularization parameter λ.

Solution. import numpy as np import pandas as pd import matplotlib.pyplot as plt from sklearn.model$_selectionimporttrain_test_splitfromsklearn.preprocessingimportStandardScalerimpo$
$return1-np.sum((y_data-y_model)**2)/np.sum((y_data-np.mean(y_data))*$

$*2)defMSE(y_data, y_model) : n = np.size(y_model)returnnp.sum((y_data{-}y_model)*$
$*2)/n$

np.random.seed(3155)

x = np.random.rand(100) y = 2.0+5*x*x+0.1*np.random.randn(100)

p = 3 X = np.zeros((len(x),p)) X[:,0] = 1.0 X[:,1] = x X[:,2] = x*x X$_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)scaler = StandardScaler()scaler.fit(X_train)X_train{}_scaled = scaler.transform(X_train)X_test{}_scaled = scaler.transform(X_test)$

OLSbeta = np.linalg.inv(X$_train.T@X_train)@X_train.T@y_trainprint(OLSbeta)ytildeOLS = X_train@OLSbetaprint(”TrainingR2forOLS”)print(R2(y_train, ytildeOLS))print(”TrainingMSEforOLS”)$
$X_test@OLSbetaprint(”TestR2forOLS”)print(R2(y_test, ypredictOLS))print(”TestMSEOLS”)print(MSE($

I = np.eye(p,p) nlambdas = 100 MSEPredictLasso = np.zeros(nlambdas) MSEPredictRidge = np.zeros(nlambdas) lambdas = np.logspace(-4, 0, nlambdas) for i in range(nlambdas): lmb = lambdas[i] add ridge clf$_ridge = skl.Ridge(alpha = lmb).fit(X_train, y_train)clf_lasso = skl.Lasso(alpha = lmb).fit(X_train, y_train)yridge = clf_ridge.predict(X_test)ylasso = clf_lasso.predict(X_test)MSEPredictLasso[i] = MSE(y_test, ylasso)MSEPredictRidge[i] = MSE(y_test, yridge)plt.figure()plt.plot(np.log10(lambdas), MSE$
$-', label =' MSERidgeTest')plt.plot(np.log10(lambdas), MSEPredictLasso,' g-$
$-', label =' MSELassoTest')plt.xlabel('log10(lambda)')plt.ylabel('MSE')plt.legend()plt.show()$

aragraph!paragraph>paragraph>-0.5em

e) 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.

Solution. These results can all be studied with the codes we have above. These scores are included in the codes above.

*

Exercise 4: Normalizing our data

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 standard scaler 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.

aragraph!paragraph>paragraph>-0.5em

a) 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.

Solution.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline


np.random.seed(2018)
n = 50
maxdegree = 5
# 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)
TestError = np.zeros(maxdegree)
TrainError = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
scaler = StandardScaler()
scaler.fit(X_train)
x_train_scaled = scaler.transform(x_train)
x_test_scaled = scaler.transform(x_test)

for degree in range(maxdegree):
    model = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression(fit_intercept=False
    clf = model.fit(x_train_scale,y_train)
    y_fit = clf.predict(x_train_scaled)
    y_pred = clf.predict(x_test_scaled)
    polydegree[degree] = degree
    TestError[degree] = np.mean( np.mean((y_test - y_pred)**2) )
    TrainError[degree] = np.mean( np.mean((y_train - y_fit)**2) )

plt.plot(polydegree, TestError, label='Test Error')
plt.plot(polydegree, TrainError, label='Train Error')
plt.legend()
plt.show()
```

aragraph!paragraph>paragraph>-0.5em

b) 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.

Solution. This requires a simple extension to the above code where you simply add a statement calling the $R2$ function included in the same code.
aragraph!paragraph>paragraph>-0.5em

c) 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)?

Solution. Here you simply need to change the degree of the polynomial in the above code to $n = 15$.
aragraph!paragraph>paragraph>-0.5em

d) 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.

Solution. Here you need to add for example the same loop over the parameters λ as you did in the first exercise, that is add

```
nlambdas = 100
MSEPredictRidge = np.zeros(nlambdas)
lambdas = np.logspace(-4, 0, nlambdas)
for i in range(nlambdas):
    lmb = lambdas[i]
    # add ridge
    clf_ridge = skl.Ridge(alpha=lmb).fit(X_train_scaled, y_train)
```

*

Exercise 5: Bias-Variance tradeoff and Bootstrap
This exercise is a continuation of exercise 2 from the second homework set. In that exercise we computed the MSE-score for the training data and the test data as functions of the complexity of a polynomial, that is the degree of a given polynomial.
One of the aims of that exercise was to reproduce Figure 2.11 of Hastie et al.
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.

Part (1a) Proving the bias-variance tradeoff. Consider a dataset $\mathcal{L}$ consisting of the data $\mathbf{X}_{\mathcal{L}} = \{(y_j, \boldsymbol{x}_j), j = 0 \ldots n-1\}$.

Let us assume that the true data is generated from a noisy model

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}.$$

Here ϵ is normally distributed with mean zero and standard deviation σ^2.

In our derivation of the ordinary least squares method we defined then an approximation to the function f in terms of the parameters $\boldsymbol{\beta}$ and the design matrix $\boldsymbol{X}$ which embody our model, that is $\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta}$.

The parameters $\boldsymbol{\beta}$ are in turn found by optimizing the means squared error via the so-called cost function

$$C(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right].$$

Show that you can rewrite this as

$$\mathbb{E}\left[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2\right] = \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}\left[\tilde{\boldsymbol{y}}\right])^2 + \sigma^2.$$

Explain what the terms mean, which one is the bias and which one is the variance and discuss their interpretations.

Part (1b) Adding Bootstrap and Bias-Variance Tradeoff. Add now bootstrapping as discussed in the Regression lectures (scroll down to the bias-variance code). Add also the expressions for the bias and the variance as discussed above.

Discuss the bias and variance tradeoff as function of your model complexity (the degree of the polynomial) and the number of data points, and possibly also your training and test data.

Try to make a figure similar to Fig. 2.11 of Hastie et al. You should include an analysis of the bias and variance for the test results. Figure 2.11 displays only the test and training MSEs while indicating regions of low/high bias and variance. You will most likely not get an equally smooth curve! Note also that when you calculate the bias, in all applications you don't know the function values f_i. You would hence replace them with the actual data points y_i.

*

Exercise 6: Linear Regression for a two-dimensional function

This is a longer exercise and the aim 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 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 established the model and the method, we will employ resamling like the bootstrap from the previous exercise in order to perform a proper assessment of our models. We will also study in detail the so-called Bias-Variance trade off.

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 (or bootstrap) as resampling technique. 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 set of fixed values for x and y with a given step size. We will fit a function (for example a polynomial) of x and y. Thereafter we will repeat much of the same procedure using the Ridge and Lasso regression methods, introducing thus a dependence on the bias (penalty) λ.

Finally 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.

The Python fucntion for the Franke function is included here (it performs also a three-dimensional plot of it)

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
from random import random, seed

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
x = np.arange(0, 1, 0.05)
y = np.arange(0, 1, 0.05)
x, y = np.meshgrid(x,y)


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


z = FrankeFunction(x, y)
```

```
# 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)

plt.show()
```

(a) Ordinary Least Square on the Franke function with resampling. We will generate our own dataset for a function FrankeFunction(x, y) with $x, y \in [0, 1]$. The function $f(x, y)$ is the Franke function. You should explore also the addition an added stochastic noise to this function using the normal distribution $\mathcal{N}(\prime, \infty)$.

Write your own code (using either a matrix inversion or a singular value decomposition from e.g., **numpy**) or use your code from homeworks 1 and 2 and perform a standard least square regression analysis using polynomials in x and y up to fifth order. You can use **scikit-learn** as well.

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 $\tilde{\hat{y}}_i$ 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

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

To set up the design matrix, the following code can be used

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


def create_X(x, y, n ):
        if len(x.shape) > 1:
                x = np.ravel(x)
                y = np.ravel(y)

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

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

        return X

# Making meshgrid of datapoints and compute Franke's function
n = 5
N = 1000
x = np.sort(np.random.uniform(0, 1, N))
y = np.sort(np.random.uniform(0, 1, N))
z = FrankeFunction(x, y)
X = create_X(x, y, n=n)
```

Part (b) Resampling techniques, adding more complexity. Perform a resampling of the data where you split the data in training data and test data. Here you can write your own function or use the function for splitting training data provided by **Scikit-Learn**. This function is called *train_test_split*. You should also renormalize your data.

It is normal in essentially all Machine Learning studies to split the data in a training set and a test set (sometimes also an additional validation set). There is no explicit recipe for how much data should be included as training data and say test data. An accepted rule of thumb is to use approximately 2/3 to 4/5 of the data as training data.

Use then the $_bootstrapcodeyoudevelopedinthepreviousexercisetoresampleyourdataandevaluateagaintheM$

Part (c): Bias-variance tradeoff. With a code which does OLS and includes bootstrap we will now discuss the bias-variance tradeoff in the context of continuous predictions such as regression. However, many of the intuitions and ideas discussed here also carry over to classification tasks and basically all Machine Learning algorithms.

Use the code from exercise 1 above and implement the bootstrap resampling and perform a bias-variance tradeoff analysis like you did in exercise 1.

Part (d): Ridge Regression on the Franke function with resampling. Write your own code for the Ridge method, either using matrix inversion or the singular value decomposition ir use **scikit-learn** Perform the same analysis as in the previous three steps (for the same polynomials and include resampling techniques) but now for different values of λ. Compare and analyze your results with those obtained in parts 2a-2c). Study the dependence on λ.

Study also the bias-variance tradeoff as function of various values of the parameter λ. Comment your results.

Part (e): Lasso Regression on the Franke function with resampling. This part is essentially a repeat of the previous ones, but now with Lasso regression. Write either your own code or use the functionalities of **Scikit-Learn** (recommended). Give a critical discussion of the three methods and a judgement of which model fits the data best.