Data Analysis and Machine Learning: Linear Regression and more Advanced Regression Analysis

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Oct 18, 2018

Regression analysis, overarching aims

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

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

- n cases $i = 0, 1, 2, \dots, n-1$
- Response (dependent or outcome) variable y_i with $i = 0, 1, 2, \dots, n-1$
- p Explanatory (independent or predictor) variables $\hat{x}_i = [x_{i0}, x_{i1}, \dots, x_{ip}]$ with $i = 0, 1, 2, \dots, n-1$

The goal of the regression analysis is to extract/exploit relationship between y_i and $\hat{x}_i$ in or to infer causal dependencies, approximations to the likelihood functions, functional relationships and to make predictions .

Regression analysis, overarching aims II

Consider an experiment in which p characteristics of n samples are measured. The data from this experiment are denoted $\mathbf{X}$, with $\mathbf{X}$ as above. The matrix $\mathbf{X}$ is called the *design matrix*. Additional information of the samples is available in the form of $\mathbf{Y}$ (also as above). The variable $\mathbf{Y}$ is generally referred to as the *response*

variable. The aim of regression analysis is to explain $\mathbf{Y}$ in terms of $\mathbf{X}$ 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 $\mathbf{X}$ and $\mathbf{Y}$. This assumption gives rise to the *linear regression model* where $\beta = (\beta_1, \ldots, \beta_p)^\top$ is the *regression parameter*. The parameter β_j, $j = 1, \ldots, p$, represents the effect size of covariate j on the response. That is, for each unit change in covariate j (while keeping the other covariates fixed) the observed change in the response is equal to β_j.

General linear models

Before we proceed let us study a case from linear algebra where we aim at fitting a set of data $\hat{y} = [y_0, y_1, \ldots, y_{n-1}]$. 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 $\hat{x} = [x_0, x_1, \ldots, x_{n-1}]$, that is $y_i = y(x_i)$ with $i = 0, 1, 2, \ldots, n-1$. 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

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

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

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

Rewriting the fitting procedure as a linear algebra problem, follows

Defining the vectors

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

and

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

and

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

and the matrix

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

we can rewrite our equations as

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

Generalizing the fitting procedure as a linear algebra problem

We are obviously not limited to the above polynomial. We could replace the various powers of x with elements of Fourier series, that is, 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

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

Generalizing the fitting procedure as a linear algebra problem

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

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

and without loss of generality we rewrite again our equations as

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

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

Optimizing our parameters

We have defined the matrix $\hat{X}$

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

Optimizing our parameters, more details

We well use this matrix to define the approximation $\hat{\tilde{y}}$ via the unknown quantity $\hat{\beta}$ as

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

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 parametrized values $\tilde{y}_i$, namely

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

or using the matrix $\hat{X}$ as

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

Interpretations and optimizing our parameters

The function

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

can be linked to the variance of the quantity y_i if we interpret the latter as the mean value of for example a numerical experiment. When linking below with

the maximum likelihood approach below, we will indeed interpret y_i as a mean value

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

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 $Q(\hat{\beta})$ by requiring

$$\frac{\partial Q(\hat{\beta})}{\partial \beta_j} = \frac{\partial}{\partial \beta_j}\left[\sum_{i=0}^{n-1} \left(y_i - \beta_0 x_{i,0} - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \cdots - \beta_{n-1} x_{i,n-1}\right)^2\right] = 0,$$

which results in

$$\frac{\partial Q(\hat{\beta})}{\partial \beta_j} = -2\left[\sum_{i=0}^{n-1} x_{ij}\left(y_i - \beta_0 x_{i,0} - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \cdots - \beta_{n-1} x_{i,n-1}\right)\right] = 0,$$

or in a matrix-vector form as

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

Interpretations and optimizing our parameters

We can rewrite

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

as

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

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

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

Interpretations and optimizing our parameters

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

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

and with

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

we have

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

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

The χ^2 function

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.

Introducing the standard deviation σ_i for each measurement y_i, we define now the χ^2 function as

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

where the matrix $\hat{\Sigma}$ 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 $\chi^2(\hat{\beta})$ by requiring

$$\frac{\partial\chi^2(\hat{\beta})}{\partial\beta_j} = \frac{\partial}{\partial\beta_j}\left[\sum_{i=0}^{n-1}\left(\frac{y_i - \beta_0 x_{i,0} - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \cdots - \beta_{n-1}x_{i,n-1}}{\sigma_i}\right)^2\right] = 0,$$

which results in

$$\frac{\partial\chi^2(\hat{\beta})}{\partial\beta_j} = -2\left[\sum_{i=0}^{n-1}\frac{x_{ij}}{\sigma_i}\left(\frac{y_i - \beta_0 x_{i,0} - \beta_1 x_{i,1} - \beta_2 x_{i,2} - \cdots - \beta_{n-1}x_{i,n-1}}{\sigma_i}\right)\right] = 0,$$

or in a matrix-vector form as

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

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

The χ^2 function

We can rewrite

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

as

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

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

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

The χ^2 function

If we then introduce the matrix

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

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

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

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

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

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

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

and

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

The χ^2 function

For a linear fit we don't need to invert a matrix!! Defining

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

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

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

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

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

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.

Simple regression model

We are now ready to write our first program which aims at solving the above linear regression equations. We start with data we have produced ourselves, in this case normally distributed random numbers along the x-axis. These numbers define then the value of a function $y(x) = 4 + 3x + N(0,1)$. Thereafter we order the x values and employ our linear regression algorithm to set up the best fit. Here we find it useful to use the numpy function $c\_$ arrays where arrays are stacked along their last axis after being upgraded to at least two dimensions with ones post-pended to the shape. The following examples help in understanding what happens

```
import numpy as np
print(np.c_[np.array([1,2,3]), np.array([4,5,6])])
print(np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])])
```

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

x = 2*np.random.rand(100,1)
y = 4+3*x+np.random.randn(100,1)

xb = np.c_[np.ones((100,1)), x]
beta = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)
xnew = np.array([[0],[2]])
xbnew = np.c_[np.ones((2,1)), xnew]
ypredict = xbnew.dot(beta)

plt.plot(xnew, ypredict, "r-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Linear Regression')
plt.show()
```

We see that, as expected, a linear fit gives a seemingly (from the graph) good representation of the data.

Simple regression model, now using scikit-learn

We can repeat the above algorithm using **scikit-learn** as follows

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

x = 2*np.random.rand(100,1)
y = 4+3*x+np.random.randn(100,1)
linreg = LinearRegression()
linreg.fit(x,y)
xnew = np.array([[0],[2]])
ypredict = linreg.predict(xnew)

plt.plot(xnew, ypredict, "r-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Random numbers ')
plt.show()
```

Simple linear regression model using scikit-learn

We start with perhaps our simplest possible example, using **scikit-learn** to perform linear regression analysis on a data set produced by us. What follows is a simple Python code where we have defined function y in terms of the variable x. Both are defined as vectors of dimension 1×100. The entries to the vector $\hat{x}$ are given by random numbers generated with a uniform distribution with entries

$x_i \in [0,1]$ (more about probability distribution functions later). These values are then used to define a function $y(x)$ (tabulated again as a vector) with a linear dependence on x plus a random noise added via the normal distribution.

The Numpy functions are imported used the **import numpy as np** statement and the random number generator for the uniform distribution is called using the function **np.random.rand()**, where we specificy that we want 100 random variables. Using Numpy we define automatically an array with the specified number of elements, 100 in our case. With the Numpy function **randn()** we can compute random numbers with the normal distribution (mean value μ equal to zero and variance σ^2 set to one) and produce the values of y assuming a linear dependence as function of x

$$y = 2x + N(0,1),$$

where $N(0,1)$ represents random numbers generated by the normal distribution. From **scikit-learn** we import then the **LinearRegression** functionality and make a prediction $\tilde{y} = \alpha + \beta x$ using the function **fit(x,y)**. We call the set of data $(\hat{x}, \hat{y})$ for our training data. The Python package **scikit-learn** has also a functionality which extracts the above fitting parameters α and β (see below). Later we will distinguish between training data and test data.

For plotting we use the Python package matplotlib which produces publication quality figures. Feel free to explore the extensive gallery of examples. In this example we plot our original values of x and y as well as the prediction **ypredict** ($\tilde{y}$), which attempts at fitting our data with a straight line.

The Python code follows here.

```
# Importing various packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

x = np.random.rand(100,1)
y = 2*x+np.random.randn(100,1)
linreg = LinearRegression()
linreg.fit(x,y)
xnew = np.array([[0],[1]])
ypredict = linreg.predict(xnew)

plt.plot(xnew, ypredict, "r-")
plt.plot(x, y ,'ro')
plt.axis([0,1.0,0, 5.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Simple Linear Regression')
plt.show()
```

Simple linear regression model

This example serves several aims. It allows us to demonstrate several aspects of data analysis and later machine learning algorithms. The immediate visualization shows that our linear fit is not impressive. It goes through the data points, but

there are many outliers which are not reproduced by our linear regression. We could now play around with this small program and change for example the factor in front of x and the normal distribution. Try to change the function y to

$$y = 10x + 0.01 \times N(0, 1),$$

where x is defined as before.

Less noise

Does the fit look better? Indeed, by reducing the role of the normal distribution we see immediately that our linear prediction seemingly reproduces better the training set. However, this testing 'by the eye' is obviouly not satisfactory in the long run. Here we have only defined the training data and our model, and have not discussed a more rigorous approach to the **cost** function.

How to study our fits

We need more rigorous criteria in defining whether we have succeeded or not in modeling our training data. You will be surprised to see that many scientists seldomly venture beyond this 'by the eye' approach. A standard approach for the *cost* function is the so-called χ^2 function

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

where σ_i^2 is the variance (to be defined later) of the entry y_i. We may not know the explicit value of σ_i^2, it serves however the aim of scaling the equations and make the cost function dimensionless.

Minimizing the cost function

Minimizing the cost function is a central aspect of our discussions to come. Finding its minima as function of the model parameters (α and β in our case) will be a recurring theme in these series of lectures. Essentially all machine learning algorithms we will discuss center around the minimization of the chosen cost function. This depends in turn on our specific model for describing the data, a typical situation in supervised learning. Automatizing the search for the minima of the cost function is a central ingredient in all algorithms. Typical methods which are employed are various variants of **gradient** methods. These will be discussed in more detail later. Again, you'll be surprised to hear that many practitioners minimize the above function "by the eye', popularly dubbed as 'chi by the eye'. That is, change a parameter and see (visually and numerically) that the χ^2 function becomes smaller.

Relative error

There are many ways to define the cost function. A simpler approach is to look at the relative difference between the training data and the predicted data, that is we define the relative error as

$$\epsilon_{\mathrm{relative}} = \frac{|\hat{y} - \hat{\tilde{y}}|}{|\hat{y}|}.$$

We can modify easily the above Python code and plot the relative error instead

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

x = np.random.rand(100,1)
y = 5*x+0.01*np.random.randn(100,1)
linreg = LinearRegression()
linreg.fit(x,y)
ypredict = linreg.predict(x)

plt.plot(x, np.abs(ypredict-y)/abs(y), "ro")
plt.axis([0,1.0,0.0, 0.5])
plt.xlabel(r'$x$')
plt.ylabel(r'$\epsilon_{\mathrm{relative}}$')
plt.title(r'Relative error')
plt.show()
```

Depending on the parameter in front of the normal distribution, we may have a small or larger relative error. Try to play around with different training data sets and study (graphically) the value of the relative error.

The richness of scikit-learn

As mentioned above, **scikit-learn** has an impressive functionality. We can for example extract the values of α and β and their error estimates, or the variance and standard deviation and many other properties from the statistical data analysis.

Here we show an example of the functionality of scikit-learn.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_squared_log_error, mean_absolute_er

x = np.random.rand(100,1)
y = 2.0+ 5*x+0.5*np.random.randn(100,1)
linreg = LinearRegression()
linreg.fit(x,y)
ypredict = linreg.predict(x)
print('The intercept alpha: \n', linreg.intercept_)
print('Coefficient beta : \n', linreg.coef_)
# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(y, ypredict))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(y, ypredict))
```

```
# Mean squared log error
print('Mean squared log error: %.2f' % mean_squared_log_error(y, ypredict) )
# Mean absolute error
print('Mean absolute error: %.2f' % mean_absolute_error(y, ypredict))
plt.plot(x, ypredict, "r-")
plt.plot(x, y ,'ro')
plt.axis([0.0,1.0,1.5, 7.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Linear Regression fit ')
plt.show()
```

Functions in scikit-learn

The function **coef** gives us the parameter β of our fit while **intercept** yields α. Depending on the constant in front of the normal distribution, we get values near or far from $alpha = 2$ and $\beta = 5$. Try to play around with different parameters in front of the normal distribution. The function **meansquarederror** gives us the mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error or loss defined as

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

The smaller the value, the better the fit. Ideally we would like to have an MSE equal zero. The attentive reader has probably recognized this function as being similar to the χ^2 function defined above.

Other functions in scikit-learn

The **r2score** function computes R^2, the coefficient of determination. It provides a measure of how well future samples are likely to be predicted by the model. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of $\hat{y}$, disregarding the input features, would get a R^2 score of 0.0.

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

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

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

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

The mean absolute error and other functions in scikit-learn

Another quantity will meet again in our discussions of regression analysis is mean absolute error (MAE), a risk metric corresponding to the expected value

of the absolute error loss or what we call the $l1$-norm loss. In our discussion above we presented the relative error. The MAE is defined as follows

$$\mathrm{MAE}(\hat{y},\hat{\tilde{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}\left|y_i-\tilde{y}_i\right|.$$

Finally we present the squared logarithmic (quadratic) error

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

where $\log_e(x)$ stands for the natural logarithm of x. This error estimate is best to use when targets having exponential growth, such as population counts, average sales of a commodity over a span of years etc.

Cubic polynomial in scikit-learn

We will discuss in more detail these and other functions in the various lectures. We conclude this part with another example. Instead of a linear x-dependence we study now a cubic polynomial and use the polynomial regression analysis tools of scikit-learn. Add description of the various python commands.

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

x=np.linspace(0.02,0.98,200)
noise = np.asarray(random.sample((range(200)),200))
y=x**3*noise
yn=x**3*100
poly3 = PolynomialFeatures(degree=3)
X = poly3.fit_transform(x[:,np.newaxis])
clf3 = LinearRegression()
clf3.fit(X,y)

Xplot=poly3.fit_transform(x[:,np.newaxis])
poly3_plot=plt.plot(x, clf3.predict(Xplot), label='Cubic Fit')
plt.plot(x,yn, color='red', label="True Cubic")
plt.scatter(x, y, label='Data', color='orange', s=15)
plt.legend()
plt.show()

def error(a):
    for i in y:
        err=(y-yn)/yn
    return abs(np.sum(err))/len(err)

print (error(y))
```

Using **R**, we can perform similar studies.

Polynomial Regression

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

m = 100
x = 2*np.random.rand(m,1)+4.
y = 4+3*x*x+ +x-np.random.randn(m,1)

xb = np.c_[np.ones((m,1)), x]
theta = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)
xnew = np.array([[0],[2]])
xbnew = np.c_[np.ones((2,1)), xnew]
ypredict = xbnew.dot(theta)

plt.plot(xnew, ypredict, "r-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Random numbers ')
plt.show()
```

Linking the regression analysis with a statistical interpretation

Before we proceed, and to link with our discussions of Bayesian statistics to come, it is useful the derive the standard regression analysis equations using a statistical interpretation. This allows us also 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.:

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

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 $\mathbf{X}_{i,*}\,\beta$ is a non-random scalar. To specify the parameters of the distribution of $\mathbf{Y}_i$ we need to calculate its first two moments.

Expectation value and variance

Its expectation equals:

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

while its variance is

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

Hence, $Y_i \sim \mathcal{N}(\mathbf{X}_{i,*}\,\beta,\sigma^2)$.

The singular value decompostion

A general $m\times n$ matrix $\hat{A}$ can be written in terms of a diagonal matrix $\hat{D}$ of dimensionality $n\times n$ and two orthognal matrices $\hat{U}$ and $\hat{V}$, where the first has dimensionality $m\times m$ and the last dimensionality $n\times n$. We have then

$$\hat{A}=\hat{U}\hat{D}\hat{V}^T$$

From standard regression to Ridge regressions

One of the typical problems we encounter with linear regression, in particular when the matrix $\hat{X}$ (our so-called design matrix) is high-dimensional, are problems with near singular or singular matrices. The column vectors of $\hat{X}$ may be linearly dependent, normally referred to as super-collinearity. This means that the matrix may be rank deficient and it is basically impossible to to model the data using linear regression. As an example, consider the matrix

$$\mathbf{X}=\begin{bmatrix}1 & -1 & 2\\ 1 & 0 & 1\\ 1 & 2 & -1\\ 1 & 1 & 0\end{bmatrix}$$

The columns of $\hat{X}$ are linearly dependent. We se this easily since the the first column is the row-wise sum of the other two columns. The rank (more correct, the column rank) of a matrix is the dimension of the space spanned by the column vectors. Hence, the rank of $\mathbf{X}$ is equal to the number of linearly independent columns. In this particular case the matrix has rank 2.

Super-collinearity of an $(n\times p)$-dimensional design matrix $\mathbf{X}$ implies that the inverse of the matrix $\hat{X}^T\hat{x}$ (the matrix we needto invert to solve the linear regression equations) is non-invertible. If we have a square matrix that does not have an inverse, we say this matrix singular. The example here demonstrates this

$$\hat{X}=\begin{bmatrix}1 & -1\\ 1 & -1\end{bmatrix}.$$

We see easily that $\det(\hat{X})=x_{11}x_{22}-x_{12}x_{21}=1\times(-1)-1\times(-1)=0$. Hence, $\mathbf{X}$ is singular and its inverse is undefined. This is equivalent to saying that the matrix $\hat{X}$ has at least an eigenvalue which is zero.

Fixing the singularity

If our design matrix $\hat{X}$ which enters the linear regression problem

$$\hat{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T \hat{y}, \tag{1}$$

has linearly dependent column vectors, we will not be able to compute the inverse of $\hat{X}^T \hat{X}$ and we cannot find the parameters (estimators) β_i. The estimators are only well-defined if $(\hat{X}^T \hat{X})^{-1}$ exits. This is more likely to happen when the matrix $\hat{X}$ is high-dimensional. In this case it is likely to encounter a situation where the regression parameters β_i cannot be estimated.

The *ad hoc* approach which was introduced in the 70s was simply to add a diagonal component to the matrix to invert, that is we change

$$\hat{X}^T \hat{X} \rightarrow \hat{X}^T \hat{X} + \lambda \hat{I},$$

where $\hat{I}$ is the identity matrix.

Fitting vs. predicting when data is in the model class

We start by considering the case $f(x) = 2x$.

Then the data is clearly generated by a model that is contained within all three model classes we are using to make predictions (linear models, third order polynomials, and tenth order polynomials).

Run the code for the following cases:

1. For $f(x) = 2x$, $Ntrain = 10$ and $\sigma = 0$ (noiseless case), train the three classes of models (linear, third-order polynomial, and tenth order polynomial) for a training set when $x \in [0, 1]$. Make graphs comparing fits for different order of polynomials. Which model fits the data the best?

2. Do you think that the data that has the least error on the training set will also make the best predictions? Why or why not? Can you try to discuss and formalize your intuition? What can go right and what can go wrong?

3. Check your answer by seeing how well your fits predict newly generated test data (including on data outside the range you fit on, for example $x \in [0, 1.2]$) using the code below. How well do you do on points in the range of x where you trained the model? How about points outside the original training data set?

4. Repeat the above for $f(x) = 2x$, $Ntrain = 10$, and $\sigma = 1$. What changes?

Repeat the exercises above for $f(x) = 2x$, $Ntrain = 100$, and $\sigma = 1$. What changes? Summarize what you have learned about the relationship between model complexity (number of parameters), goodness of fit on training data, and the ability to predict well.

Fitting versus predicting when data is not in the model class

Thus far, we have considered the case where the data is generated using a model contained in the model class. Now consider $f(x) = 2x - 10x^5 + 15x^{10}$. Notice that the for linear and third-order polynomial the true model $f(x)$ is not contained in model class.

1. Do better fits lead to better predictions?
2. What is the relationship between the true model for generating the data and the model class that has the most predictive power? How is this related to the model complexity? How does this depend on the number of data points $Ntrain$ and σ?

Summarize what you think you learned about the relationship of knowing the true model class and predictive power.

An example code without the model assessment part

```
import numpy as np
import sklearn as sk
from sklearn import datasets, linear_model
from sklearn.preprocessing import PolynomialFeatures

import matplotlib as mpl
from matplotlib import pyplot as plt

%matplotlib notebook

# The Training Data

N_train=100

sigma_train=1;

# Train on integers
x=np.linspace(0.05,0.95,N_train)
# Draw random noise
s = sigma_train*np.random.randn(N_train)

#linear
y=2*x+s

#Tenth Order
#y=2*x-10*x**5+15*x**10+s

p1=plt.plot(x,y, "o",ms=15, label='Training')

#Linear Regression
# Create linear regression object
clf = linear_model.LinearRegression()

# Train the model using the training sets
clf.fit(x[:, np.newaxis], y)
# The coefficients
```

```
xplot=np.linspace(0.02,0.98,200)
linear_plot=plt.plot(xplot, clf.predict(xplot[:, np.newaxis]),label='Linear')

#Polynomial Regression


poly3 = PolynomialFeatures(degree=3)
X = poly3.fit_transform(x[:,np.newaxis])
clf3 = linear_model.LinearRegression()
clf3.fit(X,y)


Xplot=poly3.fit_transform(xplot[:,np.newaxis])
poly3_plot=plt.plot(xplot, clf3.predict(Xplot), label='Poly 3')



#poly5 = PolynomialFeatures(degree=5)
#X = poly5.fit_transform(x[:,np.newaxis])
#clf5 = linear_model.LinearRegression()
#clf5.fit(X,y)

#Xplot=poly5.fit_transform(xplot[:,np.newaxis])
#plt.plot(xplot, clf5.predict(Xplot), 'r--',linewidth=1)

poly10 = PolynomialFeatures(degree=10)
X = poly10.fit_transform(x[:,np.newaxis])
clf10 = linear_model.LinearRegression()
clf10.fit(X,y)

Xplot=poly10.fit_transform(xplot[:,np.newaxis])
poly10_plot=plt.plot(xplot, clf10.predict(Xplot), label='Poly 10')

axes = plt.gca()
axes.set_ylim([-7,7])

handles, labels=axes.get_legend_handles_labels()
plt.legend(handles,labels, loc='lower center')
plt.xlabel("$x$")
plt.ylabel("$y$")
Title="$N=$"+str(N_train)+", $\sigma=$"+str(sigma_train)
plt.title(Title+" (train)")
plt.tight_layout()
plt.show()
```

Generating test data

```
# Generate Test Data

#Number of test data
N_test=20

sigma_test=sigma_train

max_x=1.2
x_test=max_x*np.random.random(N_test)
# Draw random noise
s_test = sigma_test*np.random.randn(N_test)
```

```
#Linear
y_test=2*x_test+s_test
#Tenth order
#y_test=2*x_test-10*x_test**5+15*x_test**10+s_test

#Make design matrices for prediction
x_plot=np.linspace(0,max_x, 200)
X3 = poly3.fit_transform(x_plot[:,np.newaxis])
X10 = poly10.fit_transform(x_plot[:,np.newaxis])

%matplotlib notebook

fig = plt.figure()
p1=plt.plot(x_test,y_test.transpose(), 'o', ms=12, label='data')
p2=plt.plot(x_plot,clf.predict(x_plot[:,np.newaxis]), label='linear')
p3=plt.plot(x_plot,clf3.predict(X3), label='3rd order')
p10=plt.plot(x_plot,clf10.predict(X10), label='10th order')


plt.legend(loc=2)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend(loc='best')
plt.title(Title+" (pred.)")
plt.tight_layout()
plt.show()
```

How can we effectively evaluate the various models?

In Ridge regression and the subsequent discussion of its properties the bias or penalty parameter is considered known or 'given'. In practice, it is unknown and the user needs to make an informed decision on its value. How do we do that? Much of the same considerations apply to the Lasso method.

Code examples for Ridge and Lasso Regression

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import linear_model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

#creating data with random noise
x=np.arange(50)

delta=np.random.uniform(-2.5,2.5, size=(50))
np.random.shuffle(delta)
y =0.5*x+5+delta

#arranging data into 2x50 matrix
a=np.array(x) #inputs
b=np.array(y) #outputs

#Split into training and test
X_train=a[:37, np.newaxis]
X_test=a[37:, np.newaxis]
y_train=b[:37]
y_test=b[37:]
```

```
print ("X_train: ", X_train.shape)
print ("y_train: ", y_train.shape)
print ("X_test: ", X_test.shape)
print ("y_test: ", y_test.shape)

print ("------------------------------------")

print ("Ordinary Least Squares")
#Add Ordinary Least Squares fit
reg=LinearRegression()
reg.fit(X_train, y_train)
pred=reg.predict(X_test)
print ("Prediction Shape: ", pred.shape)

print('Coefficients: \n', reg.coef_)
# The mean squared error
print("Mean squared error: %.2f"
      % mean_squared_error(y_test, pred))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(y_test, pred))

#plot
plt.scatter(X_test,y_test,color='green', label="Training Data")
plt.plot(X_test, pred, color='black', label="Fit Line")
plt.legend()
plt.show()

print ("------------------------------------")

print ("Ridge Regression")

ridge=linear_model.RidgeCV(alphas=[0.1,1.0,10.0])
ridge.fit(X_train,y_train)
print ("Ridge Coefficient: ",ridge.coef_)
print ("Ridge Intercept: ", ridge.intercept_)
#Look into graphing with Ridge fit

print ("------------------------------------")

print ("Lasso")
lasso=linear_model.Lasso(alpha=0.1)
lasso.fit(X_train,y_train)
predl=lasso.predict(X_test)
print("Lasso Coefficient: ", lasso.coef_)
print("Lasso Intercept: ", lasso.intercept_)
plt.scatter(X_test,y_test,color='green', label="Training Data")
plt.plot(X_test, predl, color='blue', label="Lasso")
plt.legend()
plt.show()
```

A second-order polynomial with Ridge and Lasso

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score

np.random.seed(4155)
```

```
n_samples = 100

x = np.random.rand(n_samples,1)
y = 5*x*x + 0.1*np.random.rand(n_samples,1)

# Centering  x and y.
x_ = x - np.mean(x)
y_ = y - np.mean(y) # beta_0 = mean(y)

X = np.c_[np.ones((n_samples,1)), x, x**2]
X_ = np.c_[x_, x_**2]


### 1.
lmb_values = [1e-4, 1e-3, 1e-2, 10, 1e2, 1e4]
num_values = len(lmb_values)

## Ridge-regression of centered and not centered data
beta_ridge = np.zeros((3,num_values))
beta_ridge_centered = np.zeros((3,num_values))

I3 = np.eye(3)
I2 = np.eye(2)

for i,lmb in enumerate(lmb_values):
    beta_ridge[:,i] = (np.linalg.inv( X.T @ X + lmb*I3) @ X.T @ y).flatten()
    beta_ridge_centered[1:,i] = (np.linalg.inv( X_.T @ X_ + lmb*I2) @ X_.T @ y_).flatten()

# sett beta_0 = np.mean(y)
beta_ridge_centered[0,:] = np.mean(y)

## OLS (ordinary least squares) solution
beta_ls = np.linalg.inv( X.T @ X ) @ X.T @ y

## Evaluate the models
pred_ls = X @ beta_ls
pred_ridge =  X @ beta_ridge
pred_ridge_centered =  X_ @ beta_ridge_centered[1:] + beta_ridge_centered[0,:]

## Plot the results

# Sorting
sort_ind = np.argsort(x[:,0])

x_plot = x[sort_ind,0]
x_centered_plot = x_[sort_ind,0]

pred_ls_plot = pred_ls[sort_ind,0]
pred_ridge_plot = pred_ridge[sort_ind,:]
pred_ridge_centered_plot = pred_ridge_centered[sort_ind,:]

# Plott not centered
plt.plot(x_plot,pred_ls_plot,label='ls')

for i in range(num_values):
    plt.plot(x_plot,pred_ridge_plot[:,i],label='ridge, lmb=%g'%lmb_values[i])

plt.plot(x,y,'ro')

plt.title('linear regression on un-centered data')
```

```
plt.legend()

# Plott centered
plt.figure()

for i in range(num_values):
    plt.plot(x_centered_plot,pred_ridge_centered_plot[:,i],label='ridge, lmb=%g'%lmb_values[i])

plt.plot(x_,y,'ro')

plt.title('linear regression on centered data')
plt.legend()


# 2.

pred_ridge_scikit =  np.zeros((n_samples,num_values))
for i,lmb in enumerate(lmb_values):
    pred_ridge_scikit[:,i] = (Ridge(alpha=lmb,fit_intercept=False).fit(X,y).predict(X)).flatten()

plt.figure()

plt.plot(x_plot,pred_ls_plot,label='ls')

for i in range(num_values):
    plt.plot(x_plot,pred_ridge_scikit[sort_ind,i],label='scikit-ridge, lmb=%g'%lmb_values[i])

plt.plot(x,y,'ro')
plt.legend()
plt.title('linear regression using scikit')

plt.show()

### R2-score of the results
for i in range(num_values):
    print('lambda = %g'%lmb_values[i])
    print('r2 for scikit: %g'%r2_score(y,pred_ridge_scikit[:,i]))
    print('r2 for own code, not centered: %g'%r2_score(y,pred_ridge[:,i]))
    print('r2 for own, centered: %g\n'%r2_score(y,pred_ridge_centered[:,i]))
```

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.

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.

Statistics

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur:

$$p(x) = \text{prob}(X = x)$$

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$ then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a,\, b]$ is then just the integral:

$$\text{prob}(a \leq X \leq b) = \int_a^b p(x)dx$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

Statistics, moments

A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows:

$$\langle x^n \rangle \equiv \int x^n p(x)\, dx$$

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ:

$$\langle x \rangle = \mu \equiv \int x p(x)\, dx$$

Statistics, central moments

A special version of the moments is the set of *central moments*, the n-th central moment defined as:

$$\langle (x - \langle x \rangle)^n \rangle \equiv \int (x - \langle x \rangle)^n p(x)\, dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular

interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{var}(X)$:

$$\begin{aligned}\sigma_X^2 &= \mathrm{var}(X) = \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)\, dx && (2)\\ &= \int \left(x^2 - 2x\langle x \rangle^2 + \langle x \rangle^2 \right) p(x)\, dx && (3)\\ &= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 && (4)\\ &= \langle x^2 \rangle - \langle x \rangle^2 && (5)\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x - \langle x \rangle)^2 \rangle}$ is called the *standard deviation* of p. It is clearly just the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the *spread* of p around its mean.

Statistics, covariance

Another important quantity is the so called covariance, a variant of the above defined variance. Consider again the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1, \ldots, x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows:

$$\begin{aligned}\mathrm{cov}(X_i, X_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle)\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n \qquad (6)\end{aligned}$$

with

$$\langle x_i \rangle = \int \cdots \int x_i\, P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n$$

Statistics, more covariance

If we consider the above covariance as a matrix $C_{ij} = \mathrm{cov}(X_i, X_j)$, then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{cov}(X_i, X_i) = \mathrm{var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated. This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$:

$$\begin{aligned}\mathrm{cov}(X_i, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle && (7)\\ &= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle && (8)\\ &= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle && (9)\\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle && (10)\\ &= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle && (11)\end{aligned}$$

Statistics, independent variables

If X_i and X_j are independent, we get $\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle$, resulting in $\mathrm{cov}(X_i, X_j) = 0 \quad (i \neq j)$.

Also useful for us is the covariance of linear combinations of stochastic variables. Let $\{X_i\}$ and $\{Y_i\}$ be two sets of stochastic variables. Let also $\{a_i\}$ and $\{b_i\}$ be two sets of scalars. Consider the linear combination:

$$U = \sum_i a_i X_i \qquad V = \sum_j b_j Y_j$$

By the linearity of the expectation value

$$\mathrm{cov}(U, V) = \sum_{i,j} a_i b_j \mathrm{cov}(X_i, Y_j)$$

Statistics, more variance

Now, since the variance is just $\mathrm{var}(X_i) = \mathrm{cov}(X_i, X_i)$, we get the variance of the linear combination $U = \sum_i a_i X_i$:

$$\mathrm{var}(U) = \sum_{i,j} a_i a_j \mathrm{cov}(X_i, X_j) \tag{12}$$

And in the special case when the stochastic variables are uncorrelated, the off-diagonal elements of the covariance are as we know zero, resulting in:

$$\mathrm{var}(U) = \sum_i a_i^2 \mathrm{cov}(X_i, X_i) = \sum_i a_i^2 \mathrm{var}(X_i)$$

$$\mathrm{var}(\sum_i a_i X_i) = \sum_i a_i^2 \mathrm{var}(X_i)$$

which will become very useful in our study of the error in the mean value of a set of measurements.

Statistics and stochastic processes

A *stochastic process* is a process that produces sequentially a chain of values:

$$\{x_1, x_2, \dots x_k, \dots\}.$$

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment. We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

Statistics and sample variables

In practical situations a sample is always of finite size. Let that size be n. The expectation value of a sample, the *sample mean*, is then defined as follows:

$$\bar{x}_n \equiv \frac{1}{n}\sum_{k=1}^{n} x_k$$

The *sample variance* is:

$$\mathrm{var}(x) \equiv \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2$$

its square root being the *standard deviation of the sample*. The *sample covariance* is:

$$\mathrm{cov}(x) \equiv \frac{1}{n}\sum_{kl}(x_k - \bar{x}_n)(x_l - \bar{x}_n)$$

Statistics, sample variance and covariance

Note that the sample variance is the sample covariance without the cross terms. In a similar manner as the covariance in Eq. (6) is a measure of the correlation between two stochastic variables, the above defined sample covariance is a measure of the sequential correlation between succeeding measurements of a sample.

These quantities, being known experimental values, differ significantly from and must not be confused with the similarly named quantities for stochastic variables, mean μ_X, variance $\mathrm{var}(X)$ and covariance $\mathrm{cov}(X, Y)$.

Statistics, law of large numbers

The law of large numbers states that as the size of our sample grows to infinity, the sample mean approaches the true mean μ_X of the chosen PDF:

$$\lim_{n\to\infty} \bar{x}_n = \mu_X$$

The sample mean $\bar{x}_n$ works therefore as an estimate of the true mean μ_X.

What we need to find out is how good an approximation $\bar{x}_n$ is to μ_X. In any stochastic measurement, an estimated mean is of no use to us without a measure of its error. A quantity that tells us how well we can reproduce it in another experiment. We are therefore interested in the PDF of the sample mean itself. Its standard deviation will be a measure of the spread of sample means, and we will simply call it the *error* of the sample mean, or just sample error, and denote it by err_X. In practice, we will only be able to produce an *estimate* of the sample error since the exact value would require the knowledge of the true PDFs behind, which we usually do not have.

Statistics, more on sample error

Let us first take a look at what happens to the sample error as the size of the sample grows. In a sample, each of the measurements x_i can be associated with its own stochastic variable X_i. The stochastic variable $\overline{X}_n$ for the sample mean $\bar{x}_n$ is then just a linear combination, already familiar to us:

$$\overline{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

All the coefficients are just equal $1/n$. The PDF of $\overline{X}_n$, denoted by $p_{\overline{X}_n}(x)$ is the desired PDF of the sample means.

Statistics

The probability density of obtaining a sample mean $\bar{x}_n$ is the product of probabilities of obtaining arbitrary values $x_1, x_2, \ldots, x_n$ with the constraint that the mean of the set $\{x_i\}$ is $\bar{x}_n$:

$$p_{\overline{X}_n}(x) = \int p_X(x_1)\cdots\int p_X(x_n)\ \delta\left(x - \frac{x_1 + x_2 + \cdots + x_n}{n}\right) dx_n \cdots dx_1$$

And in particular we are interested in its variance $\mathrm{var}(\overline{X}_n)$.

Statistics, central limit theorem

It is generally not possible to express $p_{\overline{X}_n}(x)$ in a closed form given an arbitrary PDF p_X and a number n. But for the limit $n \to \infty$ it is possible to make an approximation. The very important result is called *the central limit theorem*. It tells us that as n goes to infinity, $p_{\overline{X}_n}(x)$ approaches a Gaussian distribution whose mean and variance equal the true mean and variance, μ_X and σ_X^2, respectively:

$$\lim_{n\to\infty} p_{\overline{X}_n}(x) = \left(\frac{n}{2\pi\mathrm{var}(X)}\right)^{1/2} e^{-\frac{n(x-\bar{x}_n)^2}{2\mathrm{var}(X)}} \tag{13}$$

Statistics, more technicalities

The desired variance $\mathrm{var}(\overline{X}_n)$, i.e. the sample error squared err_X^2, is given by:

$$\mathrm{err}_X^2 = \mathrm{var}(\overline{X}_n) = \frac{1}{n^2}\sum_{ij}\mathrm{cov}(X_i, X_j) \tag{14}$$

We see now that in order to calculate the exact error of the sample with the above expression, we would need the true means μ_{X_i} of the stochastic variables X_i. To calculate these requires that we know the true multivariate PDF of all the X_i. But this PDF is unknown to us, we have only got the measurements of

one sample. The best we can do is to let the sample itself be an estimate of the PDF of each of the X_i, estimating all properties of X_i through the measurements of the sample.

Statistics

Our estimate of μ_{X_i} is then the sample mean $\bar{x}$ itself, in accordance with the the central limit theorem:

$$\mu_{X_i} = \langle x_i \rangle \approx \frac{1}{n} \sum_{k=1}^{n} x_k = \bar{x}$$

Using $\bar{x}$ in place of μ_{X_i} we can give an *estimate* of the covariance in Eq. (14)

$$\mathrm{cov}(X_i, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \approx \langle (x_i - \bar{x})(x_j - \bar{x}) \rangle,$$

resulting in

$$\frac{1}{n} \sum_{l}^{n} \left(\frac{1}{n} \sum_{k}^{n} (x_k - \bar{x}_n)(x_l - \bar{x}_n) \right) = \frac{1}{n} \frac{1}{n} \sum_{kl} (x_k - \bar{x}_n)(x_l - \bar{x}_n) = \frac{1}{n} \mathrm{cov}(x)$$

Statistics and sample variance

By the same procedure we can use the sample variance as an estimate of the variance of any of the stochastic variables X_i

$$\mathrm{var}(X_i) = \langle x_i - \langle x_i \rangle \rangle \approx \langle x_i - \bar{x}_n \rangle,$$

which is approximated as

$$\mathrm{var}(X_i) \approx \frac{1}{n} \sum_{k=1}^{n} (x_k - \bar{x}_n) = \mathrm{var}(x) \tag{15}$$

Now we can calculate an estimate of the error err_X of the sample mean $\bar{x}_n$:

$$\begin{aligned} \mathrm{err}_X^2 &= \frac{1}{n^2} \sum_{ij} \mathrm{cov}(X_i, X_j) \\ &\approx \qquad\qquad \frac{1}{n^2} \sum_{ij} \frac{1}{n} \mathrm{cov}(x) = \frac{1}{n^2} n^2 \frac{1}{n} \mathrm{cov}(x) \\ &= \frac{1}{n} \mathrm{cov}(x) \end{aligned} \tag{16}$$

which is nothing but the sample covariance divided by the number of measurements in the sample.

Statistics, uncorrelated results

In the special case that the measurements of the sample are uncorrelated (equivalently the stochastic variables X_i are uncorrelated) we have that the off-diagonal elements of the covariance are zero. This gives the following estimate of the sample error:

$$\mathrm{err}_X^2 = \frac{1}{n^2}\sum_{ij}\mathrm{cov}(X_i, X_j) = \frac{1}{n^2}\sum_{i}\mathrm{var}(X_i),$$

resulting in

$$\mathrm{err}_X^2 \approx \frac{1}{n^2}\sum_{i}\mathrm{var}(x) = \frac{1}{n}\mathrm{var}(x) \tag{17}$$

where in the second step we have used Eq. (15). The error of the sample is then just its standard deviation divided by the square root of the number of measurements the sample contains. This is a very useful formula which is easy to compute. It acts as a first approximation to the error, but in numerical experiments, we cannot overlook the always present correlations.

Statistics, computations

For computational purposes one usually splits up the estimate of err_X^2, given by Eq. (16), into two parts

$$\mathrm{err}_X^2 = \frac{1}{n}\mathrm{var}(x) + \frac{1}{n}(\mathrm{cov}(x) - \mathrm{var}(x)),$$

which equals

$$\frac{1}{n^2}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 + \frac{2}{n^2}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) \tag{18}$$

The first term is the same as the error in the uncorrelated case, Eq. (17). This means that the second term accounts for the error correction due to correlation between the measurements. For uncorrelated measurements this second term is zero.

Statistics, more on computations of errors

Computationally the uncorrelated first term is much easier to treat efficiently than the second.

$$\mathrm{var}(x) = \frac{1}{n}\sum_{k=1}^{n}(x_k - \bar{x}_n)^2 = \left(\frac{1}{n}\sum_{k=1}^{n}x_k^2\right) - \bar{x}_n^2$$

We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Statistics, wrapping up 1

Let us analyze the problem by splitting up the correlation term into partial sums of the form:

$$f_d = \frac{1}{n-d}\sum_{k=1}^{n-d}(x_k - \bar{x}_n)(x_{k+d} - \bar{x}_n)$$

The correlation term of the error can now be rewritten in terms of f_d

$$\frac{2}{n}\sum_{k<l}(x_k - \bar{x}_n)(x_l - \bar{x}_n) = 2\sum_{d=1}^{n-1} f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the sample samples. Notice that for $d = 0$, f is just the sample variance, $\mathrm{var}(x)$. If we divide f_d by $\mathrm{var}(x)$, we arrive at the so called *autocorrelation function*

$$\kappa_d = \frac{f_d}{\mathrm{var}(x)}$$

which gives us a useful measure of pairwise correlations starting always at 1 for $d = 0$.

Statistics, final expression

The sample error (see eq. (18)) can now be written in terms of the autocorrelation function:

$$\begin{aligned}
\mathrm{err}_X^2 &= \frac{1}{n}\mathrm{var}(x) + \frac{2}{n}\cdot\mathrm{var}(x)\sum_{d=1}^{n-1}\frac{f_d}{\mathrm{var}(x)} \\
&= \left(1 + 2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\mathrm{var}(x) \\
&= \frac{\tau}{n}\cdot\mathrm{var}(x) \qquad (19)
\end{aligned}$$

and we see that err_X can be expressed in terms the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*:

$$\tau = 1 + 2\sum_{d=1}^{n-1}\kappa_d \qquad (20)$$

Statistics, effective number of correlations

For a correlation free experiment, τ equals 1. From the point of view of eq. (19) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes:

$$n_{\mathrm{eff}} = \frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\mathrm{err}_X^2 \approx \mathrm{var}(x)/n$ to be less than the true sample error. The estimate of the error will be too *good*. On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large.

Log-likelihood

A popular strategy is to choose a penalty parameter that yields a good but parsimonious model. Information criteria measure the balance between model fit and model complexity. One possibility is Aikaike's information criterion (AIC). The AIC measures model fit by the log-likelihood and model complexity is measured by the number of parameters used by the model. The number of model parameters in regular regression simply corresponds to the number of covariates in the model. Or, by the degrees of freedom consumed by the model, which is equivalent to the trace of the hat matrix. For ridge regression it thus seems natural to define model complexity analogously by the trace of the ridge hat matrix. This yields the AIC for the linear regression model with ridge estimates:

$$\begin{aligned}
\mathrm{AIC}(\lambda) &= 2\,p - 2\log(\hat{L}) \\
&= 2\,\mathrm{tr}[\mathbf{H}(\lambda)] - 2\log\{L[\hat{\beta}(\lambda), \hat{\sigma}^2(\lambda)]\} \\
&= 2\sum_{j=1}^{p} \frac{d_{jj}^2}{d_{jj}^2 + \lambda} + 2n\,\log[\sqrt{2\,\pi}\,\hat{\sigma}(\lambda)] + \frac{1}{\hat{\sigma}^2(\lambda)} \sum_{i=1}^{n} [y_i - \mathbf{X}_{i,*}\,\hat{\beta}(\lambda)]^2.
\end{aligned}$$

The value of λ which minimizes $\mathrm{AIC}(\lambda)$ corresponds to the 'optimal' balance of model complexity and overfitting.

Cross-validation

Instead of choosing the penalty parameter to balance model fit with model complexity, cross-validation requires it (i.e. the penalty parameter) to yield a model with good prediction performance. Commonly, this performance is evaluated on novel data. Novel data need not be easy to come by and one has to make do with the data at hand. The setting of 'original' and novel data is then mimicked by sample splitting: the data set is divided into two (groups of samples). One of these two data sets, called the *training set*, plays the role of 'original' data on which the model is built. The second of these data sets, called the *test set*, plays the role of the 'novel' data and is used to evaluate the prediction performance (often operationalized as the log-likelihood or the prediction error or its square or the R2 score) of the model built on the training data set. This procedure (model building and prediction evaluation on training and test set, respectively) is done for a collection of possible penalty parameter choices. The penalty parameter that yields the model with the best prediction performance

is to be preferred. The thus obtained performance evaluation depends on the actual split of the data set. To remove this dependence the data set is split many times into a training and test set. For each split the model parameters are estimated for all choices of λ using the training data and estimated parameters are evaluated on the corresponding test set. The penalty parameter that on average over the test sets performs best (in some sense) is then selected.

Computationally expensive

The validation set approach is conceptually simple and is easy to implement. But it has two potential drawbacks:

- The validation estimate of the test error rate can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the validation set.
- In the validation approach, only a subset of the observations, those that are included in the training set rather than in the validation set are used to fit the model. Since statistical methods tend to perform worse when trained on fewer observations, this suggests that the validation set error rate may tend to overestimate the test error rate for the model fit on the entire data set.

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).

How to set up the cross-validation for Ridge and/or Lasso

- Define a range of interest for the penalty parameter.
- Divide the data set into training and test set comprising samples $\{1, \ldots, n\} \setminus i$ and $\{i\}$, respectively.

- Fit the linear regression model by means of ridge estimation for each λ in the grid using the training set, and the corresponding estimate of the error variance $\hat{\sigma}^2_{-i}(\lambda)$, as

$$\hat{\beta}_{-i}(\lambda) = (\hat{X}^\top_{-i,*}\hat{X}_{-i,*} + \lambda\hat{I}_{pp})^{-1}\hat{X}^\top_{-i,*}\hat{y}_{-i}$$

- Evaluate the prediction performance of these models on the test set by $\log\{L[y_i, \hat{X}_{i,*}; \hat{\beta}_{-i}(\lambda), \hat{\sigma}^2_{-i}(\lambda)]\}$. Or, by the prediction error $|y_i - \hat{X}_{i,*}\hat{\beta}_{-i}(\lambda)|$, the relative error, the error squared or the R2 score function.

- Repeat the first three steps such that each sample plays the role of the test set once.

- Average the prediction performances of the test sets at each grid point of the penalty bias/parameter by computing the *cross-validated log-likelihood*. It is an estimate of the prediction performance of the model corresponding to this value of the penalty parameter on novel data. It is defined as

$$\frac{1}{n}\sum_{i=1}^{n}\log\{L[y_i, \mathbf{X}_{i,*}; \hat{\beta}_{-i}(\lambda), \hat{\sigma}^2_{-i}(\lambda)]\}.$$

- The value of the penalty parameter that maximizes the cross-validated log-likelihood is the value of choice. Or we can use the MSE or the R2 score functions.

Predicted Residual Error Sum of Squares

Another approach in the LOOCV scheme is to the use the so-called Predicted Residual Error Sum of Squares (PRESS).

We can define the optimal penalty parameter to minimize

$$\lambda_{\text{opt}} = \arg\min_{\lambda}\frac{1}{n}\sum_{i=1}^{n}[y_i - \hat{X}_{i,*}\hat{\beta}_{-i}(\lambda)]^2.$$

The LOOCV prediction performance can be expressed analytically in terms of the known quantities derived from the design matrix and the parameters β.

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, we explained that this happens by scrambling the data in some way. When using the jackknife, this is done by systematically leaving out one observation from the vector of observed values $\hat{x} = (x_1, x_2, \cdots, X_n)$. Let $\hat{x}_i$ denote the vector

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

which equals the vector $\hat{x}$ 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 $\vec{X}_i$.

Resampling methods: Jackknife estimator

To get an estimate for the bias and standard error of $\widehat{\theta}$, use the following estimators for each component of $\widehat{\theta}$

$$\widehat{\text{Bias}}(\widehat{\theta}, \theta) = (n-1)\left(-\widehat{\theta} + \frac{1}{n}\sum_{i=1}^{n}\widehat{\theta}_i\right) \qquad \text{and} \qquad \widehat{\sigma}^2_{\widehat{\theta}} = \frac{n-1}{n}\sum_{i=1}^{n}(\widehat{\theta}_i - \frac{1}{n}\sum_{j=1}^{n}\widehat{\theta}_j)^2.$$

Jackknife code example

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

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 $\hat{X}$.

Resampling methods: Bootstrap steps

The independent bootstrap works like this:

1. Draw with replacement n numbers for the observed variables $\hat{x} = (x_1, x_2, \cdots, x_n)$.
2. Define a vector $\hat{x}^*$ containing the values which were drawn from $\hat{x}$.
3. Using the vector $\hat{x}^*$ compute $\widehat{\theta}^*$ by evaluating $\widehat{\theta}$ under the observations $\hat{x}^*$.
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
# the histogram of the bootstrapped  data

# 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()
```

Resampling methods: Blocking

The blocking method was made popular by Flyvbjerg and Pedersen (1989) and has become one of the standard ways to estimate $V(\widehat{\theta})$ for exactly one $\widehat{\theta}$, namely $\widehat{\theta} = \overline{X}$.

Assume $n = 2^d$ for some integer $d > 1$ and $X_1, X_2, \cdots, X_n$ is a stationary time series to begin with. Moreover, assume that the time series is asymptotically uncorrelated. We switch to vector notation by arranging $X_1, X_2, \cdots, X_n$ in an n-tuple. Define:

$$\hat{X} = (X_1, X_2, \cdots, X_n).$$

The strength of the blocking method is when the number of observations, n is large. For large n, the complexity of dependent bootstrapping scales poorly, but the blocking method does not, moreover, it becomes more accurate the larger n is.

Blocking Transformations

We now define blocking transformations. The idea is to take the mean of subsequent pair of elements from $\vec{X}$ and form a new vector $\vec{X}_1$. Continuing in the same way by taking the mean of subsequent pairs of elements of $\vec{X}_1$ we obtain $\vec{X}_2$, and so on. Define $\vec{X}_i$ recursively by:

$$\begin{aligned}(\vec{X}_0)_k &\equiv (\vec{X})_k \\ (\vec{X}_{i+1})_k &\equiv \frac{1}{2}\Big((\vec{X}_i)_{2k-1} + (\vec{X}_i)_{2k}\Big) \qquad \text{for all} \qquad 1 \leq i \leq d-1 \end{aligned} \tag{21}$$

The quantity $\vec{X}_k$ is subject to k **blocking transformations**. We now have d vectors $\vec{X}_0, \vec{X}_1, \cdots, \vec{X}_{d-1}$ containing the subsequent averages of observations. It turns out that if the components of $\vec{X}$ is a stationary time series, then the components of $\vec{X}_i$ is a stationary time series for all $0 \leq i \leq d-1$

We can then compute the autocovariance, the variance, sample mean, and number of observations for each i. Let $\gamma_i, \sigma_i^2, \overline{X}_i$ denote the autocovariance, variance and average of the elements of $\vec{X}_i$ and let n_i be the number of elements of $\vec{X}_i$. It follows by induction that $n_i = n/2^i$.

Blocking Transformations

Using the definition of the blocking transformation and the distributive property of the covariance, it is clear that since $h = |i-j|$ we can define

$$\begin{aligned}\gamma_{k+1}(h) &= cov\left((X_{k+1})_i, (X_{k+1})_j\right) \\ &= \frac{1}{4} cov\left((X_k)_{2i-1} + (X_k)_{2i}, (X_k)_{2j-1} + (X_k)_{2j}\right) \\ &= \frac{1}{2}\gamma_k(2h) + \frac{1}{2}\gamma_k(2h+1) \text{ h} = 0 \qquad (22) \\ &= \frac{1}{4}\gamma_k(2h-1) + \frac{1}{2}\gamma_k(2h) + \frac{1}{4}\gamma_k(2h+1) \quad \text{else} \qquad (23)\end{aligned}$$

The quantity $\hat{X}$ is asymptotic uncorrelated by assumption, $\hat{X}_k$ is also asymptotic uncorrelated. Let's turn our attention to the variance of the sample mean $V(\overline{X})$.

Blocking Transformations, getting there

We have

$$V(\overline{X}_k) = \frac{\sigma_k^2}{n_k} + \underbrace{\frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1 - \frac{h}{n_k}\right)\gamma_k(h)}_{\equiv e_k} = \frac{\sigma_k^2}{n_k} + e_k \quad \text{if} \quad \gamma_k(0) = \sigma_k^2. \tag{24}$$

The term e_k is called the **truncation error**:

$$e_k = \frac{2}{n_k}\sum_{h=1}^{n_k-1}\left(1-\frac{h}{n_k}\right)\gamma_k(h). \tag{25}$$

We can show that $V(\overline{X}_i) = V(\overline{X}_j)$ for all $0 \leq i \leq d-1$ and $0 \leq j \leq d-1$.

Blocking Transformations, final expressions

We can then wrap up

$$\begin{aligned} n_{j+1}\overline{X}_{j+1} &= \sum_{i=1}^{n_{j+1}}(\hat{X}_{j+1})_i = \frac{1}{2}\sum_{i=1}^{n_j/2}(\hat{X}_j)_{2i-1} + (\hat{X}_j)_{2i} \\ &= \frac{1}{2}\left[(\hat{X}_j)_1 + (\hat{X}_j)_2 + \cdots + (\hat{X}_j)_{n_j}\right] = \underbrace{\frac{n_j}{2}}_{=n_{j+1}}\overline{X}_j = n_{j+1}\overline{X}_j. \end{aligned} \tag{26}$$

By repeated use of this equation we get $V(\overline{X}_i) = V(\overline{X}_0) = V(\overline{X})$ for all $0 \leq i \leq d-1$. This has the consequence that

$$V(\overline{X}) = \frac{\sigma_k^2}{n_k} + e_k \qquad \text{for all} \qquad 0 \leq k \leq d-1. \tag{27}$$

Fyvbjerg and Petersen demonstrated that the sequence $\{e_k\}_{k=0}^{d-1}$ is decreasing, and conjecture that the term e_k can be made as small as we would like by making k (and hence d) sufficiently large. The sequence is decreasing (Master of Science thesis by Marius Jonsson, UiO 2018). It means we can apply blocking transformations until e_k is sufficiently small, and then estimate $V(\overline{X})$ by $\widehat{\sigma}_k^2/n_k$.

Code examples for Blocking, Jackknife and bootstrap

```
from sys import argv
from os import mkdir, path
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter
from matplotlib.font_manager import FontProperties

# Timing Decorator
def timeFunction(f):
    def wrap(*args):
        time1 = time.time()
        ret = f(*args)
        time2 = time.time()
        print '%s Function Took: \t %0.3f s' % (f.func_name.title(), (time2-time1))
        return ret
    return wrap

class dataAnalysisClass:
```

```
# General Init functions
def __init__(self, fileName, size=0):
    self.inputFileName = fileName
    self.loadData(size)
    self.createOutputFolder()
    self.avg = np.average(self.data)
    self.var = np.var(self.data)
    self.std = np.std(self.data)

def loadData(self, size=0):
    if size != 0:
        with open(self.inputFileName) as inputFile:
            self.data = np.zeros(size)
            for x in xrange(size):
                self.data[x] = float(next(inputFile))
    else:
        self.data = np.loadtxt(self.inputFileName)

# Statistical Analysis with Multiple Methods
def runAllAnalyses(self):
    if len(self.data) <= 100000:
        print "Autocorrelation..."
        self.autocorrelation()
    print "Bootstrap..."
    self.bootstrap()
    print "Jackknife..."
    self.jackknife()
    print "Blocking..."
    self.blocking()

# Standard Autocorrelation
@timeFunction
def autocorrelation(self):
    self.acf = np.zeros(len(self.data)/2)
    for k in range(0, len(self.data)/2):
        self.acf[k] = np.corrcoef(np.array([self.data[0:len(self.data)-k], \
                                            self.data[k:len(self.data)]]))[0,1]

# Bootstrap
@timeFunction
def bootstrap(self, nBoots = 1000):
    bootVec = np.zeros(nBoots)
    for k in range(0,nBoots):
        bootVec[k] = np.average(np.random.choice(self.data, len(self.data)))
    self.bootAvg = np.average(bootVec)
    self.bootVar = np.var(bootVec)
    self.bootStd = np.std(bootVec)

# Jackknife
@timeFunction
def jackknife(self):
    jackknVec = np.zeros(len(self.data))
    for k in range(0,len(self.data)):
        jackknVec[k] = np.average(np.delete(self.data, k))
    self.jackknAvg = self.avg - (len(self.data) - 1) * (np.average(jackknVec) - self.avg)
    self.jackknVar = float(len(self.data) - 1) * np.var(jackknVec)
    self.jackknStd = np.sqrt(self.jackknVar)

# Blocking
@timeFunction
def blocking(self, blockSizeMax = 500):
```

```
    blockSizeMin = 1

    self.blockSizes = []
    self.meanVec = []
    self.varVec = []

    for i in range(blockSizeMin, blockSizeMax):
        if(len(self.data) % i != 0):
            pass#continue
        blockSize = i
        meanTempVec = []
        varTempVec = []
        startPoint = 0
        endPoint = blockSize

        while endPoint <= len(self.data):
            meanTempVec.append(np.average(self.data[startPoint:endPoint]))
            startPoint = endPoint
            endPoint += blockSize
        mean, var = np.average(meanTempVec), np.var(meanTempVec)/len(meanTempVec)
        self.meanVec.append(mean)
        self.varVec.append(var)
        self.blockSizes.append(blockSize)

    self.blockingAvg = np.average(self.meanVec[-200:])
    self.blockingVar = (np.average(self.varVec[-200:]))
    self.blockingStd = np.sqrt(self.blockingVar)



# Plot of Data, Autocorrelation Function and Histogram
def plotAll(self):
    self.createOutputFolder()
    if len(self.data) <= 100000:
        self.plotAutocorrelation()
    self.plotData()
    self.plotHistogram()
    self.plotBlocking()

# Create Output Plots Folder
def createOutputFolder(self):
    self.outName = self.inputFileName[:-4]
    if not path.exists(self.outName):
        mkdir(self.outName)

# Plot the Dataset, Mean and Std
def plotData(self):
    # Far away plot
    font = {'fontname':'serif'}
    plt.plot(range(0, len(self.data)), self.data, 'r-', linewidth=1)
    plt.plot([0, len(self.data)], [self.avg, self.avg], 'b-', linewidth=1)
    plt.plot([0, len(self.data)], [self.avg + self.std, self.avg + self.std], 'g--', linewidth
    plt.plot([0, len(self.data)], [self.avg - self.std, self.avg - self.std], 'g--', linewidth
    plt.ylim(self.avg - 5*self.std, self.avg + 5*self.std)
    plt.gca().yaxis.set_major_formatter(FormatStrFormatter('%.4f'))
    plt.xlim(0, len(self.data))
    plt.ylabel(self.outName.title() + ' Monte Carlo Evolution', **font)
    plt.xlabel('MonteCarlo History', **font)
    plt.title(self.outName.title(), **font)
    plt.savefig(self.outName + "/data.eps")
    plt.savefig(self.outName + "/data.png")
```

```
        plt.clf()

    # Plot Histogram of Dataset and Gaussian around it
    def plotHistogram(self):
        binNumber = 50
        font = {'fontname':'serif'}
        count, bins, ignore = plt.hist(self.data, bins=np.linspace(self.avg - 5*self.std, self.avg
        plt.plot([self.avg, self.avg], [0,np.max(count)+10], 'b-', linewidth=1)
        plt.ylim(0,np.max(count)+10)
        plt.ylabel(self.outName.title() + ' Histogram', **font)
        plt.xlabel(self.outName.title() , **font)
        plt.title('Counts', **font)

        #gaussian
        norm = 0
        for i in range(0,len(bins)-1):
            norm += (bins[i+1]-bins[i])*count[i]
        plt.plot(bins,  norm/(self.std * np.sqrt(2 * np.pi)) * np.exp( - (bins - self.avg)**2 / (2
        plt.savefig(self.outName + "/hist.eps")
        plt.savefig(self.outName + "/hist.png")
        plt.clf()

    # Plot the Autocorrelation Function
    def plotAutocorrelation(self):
        font = {'fontname':'serif'}
        plt.plot(range(1, len(self.data)/2), self.acf[1:], 'r-')
        plt.ylim(-1, 1)
        plt.xlim(0, len(self.data)/2)
        plt.ylabel('Autocorrelation Function', **font)
        plt.xlabel('Lag', **font)
        plt.title('Autocorrelation', **font)
        plt.savefig(self.outName + "/autocorrelation.eps")
        plt.savefig(self.outName + "/autocorrelation.png")
        plt.clf()

    def plotBlocking(self):
        font = {'fontname':'serif'}
        plt.plot(self.blockSizes, self.varVec, 'r-')
        plt.ylabel('Variance', **font)
        plt.xlabel('Block Size', **font)
        plt.title('Blocking', **font)
        plt.savefig(self.outName + "/blocking.eps")
        plt.savefig(self.outName + "/blocking.png")
        plt.clf()

    # Print Stuff to the Terminal
    def printOutput(self):
        print "\nSample Size:    \t", len(self.data)
        print "\n========================================\n"
        print "Sample Average: \t", self.avg
        print "Sample Variance:\t", self.var
        print "Sample Std:     \t", self.std
        print "\n========================================\n"
        print "Bootstrap Average: \t", self.bootAvg
        print "Bootstrap Variance:\t", self.bootVar
        print "Bootstrap Error:   \t", self.bootStd
        print "\n========================================\n"
        print "Jackknife Average: \t", self.jackknAvg
        print "Jackknife Variance:\t", self.jackknVar
        print "Jackknife Error:   \t", self.jackknStd
        print "\n========================================\n"
```

```
        print "Blocking Average: \t", self.blockingAvg
        print "Blocking Variance:\t", self.blockingVar
        print "Blocking Error:   \t", self.blockingStd, "\n"

# Initialize the class
if len(argv) > 2:
    dataAnalysis = dataAnalysisClass(argv[1], int(argv[2]))
else:
    dataAnalysis = dataAnalysisClass(argv[1])

# Run Analyses
dataAnalysis.runAllAnalyses()

# Plot the data
dataAnalysis.plotAll()

# Print Some Output
dataAnalysis.printOutput()
```

The bias-variance tradeoff

We begin with an unknown function $y = f(x)$ and fix a *hypothesis set* $\mathcal{H}$ consisting of all functions we are willing to consider, defined also on the domain of f. This set may be uncountably infinite (e.g. if there are real-valued parameters to fit). The choice of which functions to include in $\mathcal{H}$ usually depends on our intuition about the problem of interest. The function $f(x)$ produces a set of pairs (x_i, y_i), $i = 1 \dots N$, which serve as the observable data. Our goal is to select a function from the hypothesis set $h \in \mathcal{H}$ which approximates $f(x)$ as best as possible, namely, we would like to find $h \in \mathcal{H}$ such that $h \approx f$ in some strict mathematical sense which we specify below. If this is possible, we say that we *learned* $f(x)$. But if the function $f(x)$ can, in principle, take any value on *unobserved* inputs, how is it possible to learn in any meaningful sense?

Training and testing data

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 $\mathbf{X}_{\mathcal{L}} = \{(y_j, \boldsymbol{x}_j), j = 1 \dots N\}$. Let us assume that the true data is generated from a noisy model

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

where ϵ is normally distributed with mean zero and standard deviation σ_ϵ.

Procedure to find a predictor

We have a statistical procedure (e.g. least-squares regression) for forming a predictor $\hat{g}_{\mathcal{L}}(\boldsymbol{x})$ that gives the prediction of our model for a new data point $\boldsymbol{x}$. This estimator is chosen by minimizing a cost function which we take to be the squared error

$$\mathcal{C}(\boldsymbol{X}, \hat{g}(\boldsymbol{x})) = \sum_i (y_i - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2.$$

What we want

We are interested in the generalization error on all data drawn from the true model, not just the error on the particular training dataset $\mathcal{L}$ that we have in hand. This is just the expectation of the cost function over many different data sets $\{\mathcal{L}_j\}$. Denote this expectation value by $E_{\mathcal{L}}$. In other words, we can view $\hat{g}_{\mathcal{L}}$ as a stochastic functional that depends on the dataset $\mathcal{L}$ and we can think of $E_{\mathcal{L}}$ as the expected value of the functional if we drew an infinite number of datasets $\{\mathcal{L}_1, \mathcal{L}_2, \ldots\}$.

The expected generalization error

We would also like to average over different instances of the "noise" ϵ and we denote the expectation value over the noise by E_ϵ. Thus, we can decompose the expected generalization error as

$$\begin{aligned}
E_{\mathcal{L},\epsilon}[\mathcal{C}(\boldsymbol{X}, \hat{g}(\boldsymbol{x}))] &= E_{\mathcal{L},\epsilon}\left[\sum_i (y_i - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2\right] \\
&= E_{\mathcal{L},\epsilon}\left[\sum_i (y_i - f(\boldsymbol{x}_i) + f(\boldsymbol{x}_i) - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2\right] \\
&= \sum_i E_\epsilon[(y_i - f(\boldsymbol{x}_i))^2] + E_{\mathcal{L},\epsilon}[(f(\boldsymbol{x}_i) - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2] + 2E_\epsilon[y_i - f(\boldsymbol{x}_i)]E_{\mathcal{L}}[f(\boldsymbol{x}_i) - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)] \\
&= \sum_i \sigma_\epsilon^2 + E_{\mathcal{L}}[(f(\boldsymbol{x}_i) - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2], \qquad (28)
\end{aligned}$$

where in the last line we used the fact that our noise has zero mean and variance σ_ϵ^2 and the sum over i applies to all terms.

Elaborating a little bit more

It is also helpful to further decompose the second term as follows:

$$\begin{aligned}
E_{\mathcal{L}}[(f(\boldsymbol{x}_i) - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2] &= E_{\mathcal{L}}[(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)] + E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)] - \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i))^2] \\
&= E_{\mathcal{L}}[(f(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2] + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2] \\
&\quad + 2E_{\mathcal{L}}[(f(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])(\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])] \\
&= (f(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2 + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2]. \qquad (29)
\end{aligned}$$

The bias

The first term is called the bias

$$Bias^2 = \sum_i (f(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2$$

and measures the deviation of the expectation value of our estimator (i.e. the asymptotic value of our estimator in the infinite data limit) from the true value.

The variance

The second term is called the variance

$$Var = \sum_i E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i)])^2],$$

and measures how much our estimator fluctuates due to finite-sample effects. Combining these expressions, we see that the expected out-of-sample error of our model can be decomposed as

$$E_{\text{out}} = E_{\mathcal{L},\epsilon}[\mathcal{C}(\boldsymbol{X}, \hat{g}(\boldsymbol{x}))] = Bias^2 + Var + Noise.$$

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 – 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).

Summing up

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.

The one-dimensional Ising model, project 2

The one-dimensional Ising model with nearest neighbor interaction, no external field and a constant coupling constant J is given by

$$H = -J \sum_{k}^{L} s_k s_{k+1}, \tag{30}$$

where $s_i \in \{-1, 1\}$ and $s_{N+1} = s_1$. The number of spins in the system is determined by L. For the one-dimensional system there is no phase transition.

We will look at a system of $L = 40$ spins with a coupling constant of $J = 1$. To get enough training data we will generate 10000 states with their respective energies.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import seaborn as sns
import scipy.linalg as scl
from sklearn.model_selection import train_test_split
import sklearn.linear_model as skl
import tqdm
sns.set(color_codes=True)
cmap_args=dict(vmin=-1., vmax=1., cmap='seismic')

L = 40
n = int(1e4)

spins = np.random.choice([-1, 1], size=(n, L))
J = 1.0

energies = np.zeros(n)

for i in range(n):
    energies[i] = - J * np.dot(spins[i], np.roll(spins[i], 1))
```

Here we use linear (ordinary least squares), ridge and LASSO regression to predict the energy in the nearest neighbor one-dimensional Ising model on a ring, i.e., the endpoints wrap around. We will use the linear regression models to fit a value for the coupling constant to achieve this.

Reformulating the problem to suit regression

A more general form for the one-dimensional Ising model is

$$H = -\sum_{j}^{L} \sum_{k}^{L} s_j s_k J_{jk}. \tag{31}$$

Here we allow for interactions beyond the nearest neighbors and a more adaptive coupling matrix. This latter expression can be formulated as a matrix-

product on the form

$$H = XJ, \tag{32}$$

where $X_{jk} = s_j s_k$ and J is the matrix consisting of the elements $-J_{jk}$. This form of writing the energy fits perfectly with the form utilized in linear regression, viz.

$$y = X\omega + \epsilon, \tag{33}$$

```
X = np.zeros((n, L ** 2))
for i in range(n):
    X[i] = np.outer(spins[i], spins[i]).ravel()
y = energies
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.96)

X_train_own = np.concatenate(
    (np.ones(len(X_train))[:, np.newaxis], X_train),
    axis=1
)

X_test_own = np.concatenate(
    (np.ones(len(X_test))[:, np.newaxis], X_test),
    axis=1
)
```

Linear regression

The problem at hand is to try to fit the equation

$$y = f(x) + \epsilon, \tag{34}$$

where $f(x)$ is some unknown function of the data x and ϵ is normally distributed with mean zero noise with standard deviation σ_ϵ. Our job is to try to find a predictor which estimates the function $f(x)$. In linear regression we assume that we can formulate the problem as

$$y = X\omega + \epsilon, \tag{35}$$

where X and ω are now matrices. Our job at hand is now to find a **cost function** C, which we wish to minimize in order to find the best estimate of ω.

Ordinary least squares

In the ordinary least squares method we choose the cost function

$$C(X, \omega) = ||X\omega - y||^2 = (X\omega - y)^T (X\omega - y) \tag{36}$$

We then find the extremal point of C by taking the derivative with respect to ω and setting it to zero, i.e.,

$$\frac{\mathrm{d}C}{\mathrm{d}\omega} = 0. \tag{37}$$

This yields the expression for ω to be

$$\omega = \frac{X^T y}{X^T X}, \tag{38}$$

which immediately imposes some requirements on X as there must exist an inverse of $X^T X$. If the expression we are modelling contains an intercept, i.e., a constant expression we must make sure that the first column of X consists of 1.

```
def get_ols_weights_naive(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    return scl.inv(x.T @ x) @ (x.T @ y)
omega = get_ols_weights_naive(X_train_own, y_train)
```

Singular Value decomposition

Doing the inversion directly turns out to be a bad idea as the matrix $X^T X$ is singular. An alternative approach is to use the **singular value decomposition**. Using the definition of the Moore-Penrose pseudoinverse we can write the equation for ω as

$$\omega = X^+ y, \tag{39}$$

where the pseudoinverse of X is given by

$$X^+ = \frac{X^T}{X^T X}. \tag{40}$$

Using singular value decomposition we have that $X = U\Sigma V^T$, where $X^+ = V\Sigma^+ U^T$. This reduces the equation for ω to

$$\omega = V\Sigma^+ U^T y. \tag{41}$$

Note that solving this equation by actually doing the pseudoinverse (which is what we will do) is not a good idea as this operation scales as $\mathcal{O}(n^3)$, where n is the number of elements in a general matrix. Instead, doing QR-factorization and solving the linear system as an equation would reduce this down to $\mathcal{O}(n^2)$ operations.

```
def get_ols_weights(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    u, s, v = scl.svd(x)
    return v.T @ scl.pinv(scl.diagsvd(s, u.shape[0], v.shape[0])) @ u.T @ y
```

Before passing in the data to the function we append a column with ones to the training data.

```
omega = get_ols_weights(X_train_own,y_train)
```

Fitting with scikit-learn

Next we fit a `LinearRegression`-model from Scikit-learn for comparison.

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

Extracting the J-matrix from both our own method and the Scikit-learn model where we make sure to remove the intercept.

```
J_own = omega[1:].reshape(L, L)
J_sk = clf.coef_.reshape(L, L)
```

A way of looking at the coefficients in J is to plot the matrices as images.

```
fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_own, **cmap_args)
plt.title("Home-made OLS", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_sk, **cmap_args)
plt.title("LinearRegression from Scikit-learn", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)
plt.show()
```

We can see that our model for the least squares method performes close to the benchmark from Scikit-learn. It is interesting to note that OLS considers both $J_{j,j+1} = -0.5$ and $J_{j,j-1} = -0.5$ as valid matrix elements for J.

Ridge regression

Having explored the ordinary least squares we move on to ridge regression. In ridge regression we include a **regularizer**. This involves a new cost function which leads to a new estimate for the weights ω. This results in a penalized regression problem. The cost function is given by

$$C(X, \omega; \lambda) = ||X\omega - y||^2 + \lambda||\omega||^2 = (X\omega - y)^T(X\omega - y) + \lambda\omega^T\omega. \tag{42}$$

Finding the extremum of this function yields the weights

$$\omega(\lambda) = \frac{X^T y}{X^T X + \lambda} \rightarrow \frac{\omega_{\text{LS}}}{1 + \lambda}, \tag{43}$$

where ω_{LS} is the weights from ordinary least squares. The last assumption assumes that X is orthogonal, which it is not. We will therefore resort to solving the equation as it stands on the left hand side.

```
def get_ridge_weights(x: np.ndarray, y: np.ndarray, _lambda: float) -> np.ndarray:
    return x.T @ y @ scl.inv(
        x.T @ x + np.eye(x.shape[1], x.shape[1]) * _lambda
    )
_lambda = 0.1
omega_ridge = get_ridge_weights(X_train_own, y_train, np.array([_lambda]))
clf_ridge = skl.Ridge(alpha=_lambda).fit(X_train, y_train)
J_ridge_own = omega_ridge[1:].reshape(L, L)
J_ridge_sk = clf_ridge.coef_.reshape(L, L)
fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_ridge_own, **cmap_args)
plt.title("Home-made ridge regression", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_ridge_sk, **cmap_args)
plt.title("Ridge from Scikit-learn", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

plt.show()
```

LASSO regression

In the **Least Absolute Shrinkage and Selection Operator** (LASSO)-method we get a third cost function.

$$C(X,\omega;\lambda) = ||X\omega - y||^2 + \lambda||\omega|| = (X\omega - y)^T(X\omega - y) + \lambda\sqrt{\omega^T\omega}. \qquad (44)$$

Finding the extremal point of this cost function is not so straight-forward as in least squares and ridge. We will therefore rely solely on the function "Lasso" from Scikit-learn.

```
clf_lasso = skl.Lasso(alpha=_lambda).fit(X_train, y_train)
J_lasso_sk = clf_lasso.coef_.reshape(L, L)
fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_lasso_sk, **cmap_args)
plt.title("Lasso from Scikit-learn", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

plt.show()
```

It is quite striking how LASSO breaks the symmetry of the coupling constant as opposed to ridge and OLS. We get a sparse solution with $J_{j,j+1} = -1$.

Performance of the different models

In order to judge which model performs best at varying values of λ (for ridge and LASSO) we compute R^2 which is given by

$$R^2 = 1 - \frac{(y - \hat{y})^2}{(y - \bar{y})^2}, \tag{45}$$

where y is a vector with the true values of the energy, $\hat{y}$ is the predicted values of y from the models and $\bar{y}$ is the mean of $\hat{y}$.

```
def r_squared(y, y_hat):
    return 1 - np.sum((y - y_hat) ** 2) / np.sum((y - np.mean(y_hat)) ** 2)
```

This is the same metric used by Scikit-learn for their regression models when scoring.

```
y_hat = clf.predict(X_test)
r_test = r_squared(y_test, y_hat)
sk_r_test = clf.score(X_test, y_test)

assert abs(r_test - sk_r_test) < 1e-2
```

Performance as function of the regularization parameter

We see how the different models perform for a different set of values for λ.

```
lambdas = np.logspace(-4, 5, 10)

train_errors = {
    "ols_own": np.zeros(lambdas.size),
    "ols_sk": np.zeros(lambdas.size),
    "ridge_own": np.zeros(lambdas.size),
    "ridge_sk": np.zeros(lambdas.size),
    "lasso_sk": np.zeros(lambdas.size)
}

test_errors = {
    "ols_own": np.zeros(lambdas.size),
    "ols_sk": np.zeros(lambdas.size),
    "ridge_own": np.zeros(lambdas.size),
    "ridge_sk": np.zeros(lambdas.size),
    "lasso_sk": np.zeros(lambdas.size)
}

plot_counter = 1

fig = plt.figure(figsize=(32, 54))

for i, _lambda in enumerate(tqdm.tqdm(lambdas)):
    omega = get_ols_weights(X_train_own, y_train)
    y_hat_train = X_train_own @ omega
    y_hat_test = X_test_own @ omega

    train_errors["ols_own"][i] = r_squared(y_train, y_hat_train)
    test_errors["ols_own"][i] = r_squared(y_test, y_hat_test)

    plt.subplot(10, 5, plot_counter)
    plt.imshow(omega[1:].reshape(L, L), **cmap_args)
    plt.title("Home made OLS")
    plot_counter += 1
```

```
        omega = get_ridge_weights(X_train_own, y_train, _lambda)
        y_hat_train = X_train_own @ omega
        y_hat_test = X_test_own @ omega

        train_errors["ridge_own"][i] = r_squared(y_train, y_hat_train)
        test_errors["ridge_own"][i] = r_squared(y_test, y_hat_test)

        plt.subplot(10, 5, plot_counter)
        plt.imshow(omega[1:].reshape(L, L), **cmap_args)
        plt.title(r"Home made ridge, $\lambda = %.4f$" % _lambda)
        plot_counter += 1

        for key, method in zip(
            ["ols_sk", "ridge_sk", "lasso_sk"],
            [skl.LinearRegression(), skl.Ridge(alpha=_lambda), skl.Lasso(alpha=_lambda)]
        ):
            method = method.fit(X_train, y_train)

            train_errors[key][i] = method.score(X_train, y_train)
            test_errors[key][i] = method.score(X_test, y_test)

            omega = method.coef_.reshape(L, L)

            plt.subplot(10, 5, plot_counter)
            plt.imshow(omega, **cmap_args)
            plt.title(r"%s, $\lambda = %.4f$" % (key, _lambda))
            plot_counter += 1

plt.show()
```

We can see that LASSO quite fast reaches a good solution for low values of λ, but will "wither" when we increase λ too much. Ridge is more stable over a larger range of values for λ, but eventually also fades away.

Finding the optimal value of λ

To determine which value of λ is best we plot the accuracy of the models when predicting the training and the testing set. We expect the accuracy of the training set to be quite good, but if the accuracy of the testing set is much lower this tells us that we might be subject to an overfit model. The ideal scenario is an accuracy on the testing set that is close to the accuracy of the training set.

```
fig = plt.figure(figsize=(20, 14))

colors = {
    "ols_own": "b",
    "ridge_own": "g",
    "ols_sk": "r",
    "ridge_sk": "y",
    "lasso_sk": "c"
}

for key in train_errors:
    plt.semilogx(
        lambdas,
        train_errors[key],
```

```
            colors[key],
            label="Train {0}".format(key),
            linewidth=4.0
        )

    for key in test_errors:
        plt.semilogx(
            lambdas,
            test_errors[key],
            colors[key] + "--",
            label="Test {0}".format(key),
            linewidth=4.0
        )
    #plt.semilogx(lambdas, train_errors["ols_own"], label="Train (OLS own)")
    #plt.semilogx(lambdas, test_errors["ols_own"], label="Test (OLS own)")

    plt.legend(loc="best", fontsize=18)
    plt.xlabel(r"$\lambda$", fontsize=18)
    plt.ylabel(r"$R^2$", fontsize=18)
    plt.tick_params(labelsize=18)
    plt.show()
```

From the above figure we can see that LASSO with $\lambda = 10^{-2}$ achieve a very good accuracy on the test set. This by far surpases the other models for all values of λ.