

# Data Analysis and Machine Learning:

## Day 1, Linear Regression

Morten Hjorth-Jensen<sup>1,2</sup>

<sup>1</sup>Department of Physics and Center for Computing in Science Education, University of Oslo, Norway

<sup>2</sup>Department of Physics and Astronomy and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, U

Feb 14, 2021

### Introduction

During the last two decades there has been a swift and amazing development of Machine Learning techniques and algorithms that impact many areas in not only Science and Technology but also the Humanities, Social Sciences, Medicine, Law, indeed, almost all possible disciplines. The applications are incredibly many, from self-driving cars to solving high-dimensional differential equations or complicated quantum mechanical many-body problems. Machine Learning is perceived by many as one of the main disruptive techniques nowadays.

Statistics, Data science and Machine Learning form important fields of research in modern science. They describe how to learn and make predictions from data, as well as allowing us to extract important correlations about physical process and the underlying laws of motion in large data sets. The latter, big data sets, appear frequently in essentially all disciplines, from the traditional Science, Technology, Mathematics and Engineering fields to Life Science, Law, education research, the Humanities and the Social Sciences.

### Overview of these introductory notes

The aim of these notes is to give you a birds view over overarching issues on Machine Learning, a brief review of programming with Python and libraries we will use in this course, a reminder on statistics and finally our first encounters of Machine Learning methods applied to an evergreen in Nuclear Physics, fitting nuclear binding energies. If you are familiar with basic Python programming and statistics, you can easily jump some of the introductory material here.

After these introductory words, the set of lectures will contain the following themes:

- Linear Regression

- Logistic Regression
- Statistical analysis and optimization
- Decision Trees, Random Forests, Bagging and Boosting
- Neural Networks
- Deep Learning methods, Convolutional and Recurrent Neural Networks and how to analyze experimental results

## **Machine Learning, short overview**

### **Machine Learning, a small (and probably biased) introduction**

Ideally, machine learning represents the science of giving computers the ability to learn without being explicitly programmed. The idea is that there exist generic algorithms which can be used to find patterns in a broad class of data sets without having to write code specifically for each problem. The algorithm will build its own logic based on the data. You should however always keep in mind that machines and algorithms are to a large extent developed by humans. The insights and knowledge we have about a specific system, play a central role when we develop a specific machine learning algorithm.

### **Machine Learning, an extremely rich field**

Machine learning is an extremely rich field, in spite of its young age. The increases we have seen during the last decades in computational capabilities have been followed by developments of methods and techniques for analyzing and handling large data sets, relying heavily on statistics, computer science and mathematics. The field is rather new and developing rapidly. Popular software libraries written in Python for machine learning like [Scikit-learn](#), [Tensorflow](#), [PyTorch](#) and [Keras](#), all freely available at their respective GitHub sites, encompass communities of developers in the thousands or more. And the number of code developers and contributors keeps increasing.

### **A multidisciplinary approach**

Not all the algorithms and methods can be given a rigorous mathematical justification (for example decision trees and random forests), opening up thereby large rooms for experimenting and trial and error and thereby exciting new developments. However, a solid command of linear algebra, multivariate theory, probability theory, statistical data analysis, understanding errors and Monte Carlo methods are central elements in a proper understanding of many of the algorithms and methods we will discuss.

## Learning outcomes

These sets of lectures aim at giving you an overview of central aspects of statistical data analysis as well as some of the central algorithms used in machine learning. We will introduce a variety of central algorithms and methods essential for studies of data analysis and machine learning.

Hands-on projects and experimenting with data and algorithms play a central role in these lectures, and our hope is, through the various examples discussed in this series of lectures, to expose you to fundamental research problems in these fields, with the aim to reproduce state of the art scientific results. More specifically, you will

1. Learn about basic data analysis, data optimization and machine learning;
2. Be capable of extending the acquired knowledge to other systems and cases;
3. Have an understanding of central algorithms used in data analysis and machine learning;
4. Methods we will focus on are Linear and Logistic Regression, Decision trees, random forests, bagging and boosting and various variants of deep learning methods, from feed forward neural networks to more advanced methods;
5. Work on numerical examples to illustrate the theory;

## Types of Machine Learning

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system. Some authors also operate with a third category, namely *reinforcement learning*. This is a paradigm of learning inspired by behavioral psychology, where learning is achieved by trial-and-error, solely from rewards and punishment.

Another way to categorize machine learning tasks is to consider the desired output of a system. Some of the most common tasks are:

- Classification: Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes. An example is to identify digits based on pictures of hand-written ones. Classification is often supervised learning.
- Regression: Finding a functional relationship between an input data set and a reference data set. The goal is to construct a function that maps input data to continuous output values.

- Clustering: Data are divided into groups with certain common traits, without knowing the different groups beforehand. It is thus a form of unsupervised learning.

## Essential elements of ML

The methods we cover have three main topics in common, irrespective of whether we deal with supervised or unsupervised learning.

- The first ingredient is normally our data set (which can be subdivided into training, validation and test data). Many find the most difficult part of using Machine Learning to be the set up of your data in a meaningful way.
- The second item is a model which is normally a function of some parameters. The model reflects our knowledge of the system (or lack thereof). As an example, if we know that our data show a behavior similar to what would be predicted by a polynomial, fitting our data to a polynomial of some degree would then determine our model.
- The last ingredient is a so-called **cost/loss** function (or error function) which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.

## An optimization/minimization problem

At the heart of basically all Machine Learning algorithms we will encounter so-called minimization or optimization algorithms. A large family of such methods are so-called **gradient methods**.

## A Frequentist approach to data analysis

When you hear phrases like **predictions and estimations** and **correlations and causations**, what do you think of? May be you think of the difference between classifying new data points and generating new data points. Or perhaps you consider that correlations represent some kind of symmetric statements like if  $A$  is correlated with  $B$ , then  $B$  is correlated with  $A$ . Causation on the other hand is directional, that is if  $A$  causes  $B$ ,  $B$  does not necessarily cause  $A$ .

These concepts are in some sense the difference between machine learning and statistics. In machine learning and prediction based tasks, we are often interested in developing algorithms that are capable of learning patterns from given data in an automated fashion, and then using these learned patterns to make predictions or assessments of newly given data. In many cases, our primary concern is the quality of the predictions or assessments, and we are less concerned about the underlying patterns that were learned in order to make these predictions.

In machine learning we normally use a [so-called frequentist approach](#), where the aim is to make predictions and find correlations. We focus less on for example

extracting a probability distribution function (PDF). The PDF can be used in turn to make estimations and find causations such as given  $A$  what is the likelihood of finding  $B$ .

## What is a good model?

In science and engineering we often end up in situations where we want to infer (or learn) a quantitative model  $M$  for a given set of sample points  $\mathbf{X} \in [x_1, x_2, \dots x_N]$ .

As we will see repeatedly in these lectures, we could try to fit these data points to a model given by a straight line, or if we wish to be more sophisticated to a more complex function.

The reason for inferring such a model is that it serves many useful purposes. On the one hand, the model can reveal information encoded in the data or underlying mechanisms from which the data were generated. For instance, we could discover important correlations that relate interesting physics interpretations.

In addition, it can simplify the representation of the given data set and help us in making predictions about future data samples.

A first important consideration to keep in mind is that inferring the *correct* model for a given data set is an elusive, if not impossible, task. The fundamental difficulty is that if we are not specific about what we mean by a *correct* model, there could easily be many different models that fit the given data set *equally well*.

## What is a good model? Can we define it?

The central question is this: what leads us to say that a model is correct or optimal for a given data set? To make the model inference problem well posed, i.e., to guarantee that there is a unique optimal model for the given data, we need to impose additional assumptions or restrictions on the class of models considered. To this end, we should not be looking for just any model that can describe the data. Instead, we should look for a **model**  $M$  that is the best among a restricted class of models. In addition, to make the model inference problem computationally tractable, we need to specify how restricted the class of models needs to be. A common strategy is to start with the simplest possible class of models that is just necessary to describe the data or solve the problem at hand. More precisely, the model class should be rich enough to contain at least one model that can fit the data to a desired accuracy and yet be restricted enough that it is relatively simple to find the best model for the given data.

Thus, the most popular strategy is to start from the simplest class of models and increase the complexity of the models only when the simpler models become inadequate. For instance, if we work with a regression problem to fit a set of sample points, one may first try the simplest class of models, namely linear models, followed obviously by more complex models.

How to evaluate which model fits best the data is something we will come back to over and over again in these set of lectures.

## Practicalities, choice of programming language and other computational issues

### Choice of Programming Language

Python plays nowadays a central role in the development of machine learning techniques and tools for data analysis. In particular, seen the wealth of machine learning and data analysis libraries written in Python, easy to use libraries with immediate visualization (and not the least impressive galleries of existing examples), the popularity of the Jupyter notebook framework with the possibility to run **R** codes or compiled programs written in C++, and much more made our choice of programming language for this series of lectures easy. However, since the focus here is not only on using existing Python libraries such as **Scikit-Learn**, **Tensorflow** and **Pytorch**, but also on developing your own algorithms and codes, we will as far as possible present many of these algorithms either as a Python codes or C++ or Fortran (or other languages) codes.

### Software and needed installations

We will make extensive use of Python as programming language and its myriad of available libraries. You will find Jupyter notebooks invaluable in your work. You can run **R** codes in the Jupyter/IPython notebooks, with the immediate benefit of visualizing your data. You can also use compiled languages like C++, Rust, Julia, Fortran etc if you prefer. The focus in these lectures will be on Python.

If you have Python installed (we strongly recommend Python3) and you feel pretty familiar with installing different packages, we recommend that you install the following Python packages via **pip** as (examples of relevant packages)

1. `pip install numpy scipy matplotlib ipython scikit-learn mglearn sympy pandas pillow`

For OSX users we recommend, after having installed Xcode, to install **brew**. Brew allows for a seamless installation of additional software via for example

1. `brew install python`

For Linux users, with its variety of distributions like for example the widely popular Ubuntu distribution, you can use **pip** as well and simply install Python as

1. `sudo apt-get install python`

etc etc.

## Python installers

If you don't want to perform these operations separately and venture into the hassle of exploring how to set up dependencies and paths, we recommend two widely used distributions which set up all relevant dependencies for Python, namely

- [Anaconda](#),

which is an open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system **conda**.

- [Enthought canopy](#)

is a Python distribution for scientific and analytic computing distribution and analysis environment, available for free and under a commercial license.

Furthermore, [Google's Colab](#) is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. Try it out!

## Useful Python libraries

Here we list several useful Python libraries we strongly recommend (if you use anaconda many of these are already there)

- [NumPy](#) is a highly popular library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays
- [The pandas](#) library provides high-performance, easy-to-use data structures and data analysis tools
- [Xarray](#) is a Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!
- [Scipy](#) (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering.
- [Matplotlib](#) is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- [Autograd](#) can automatically differentiate native Python and Numpy code. It can handle a large subset of Python's features, including loops, ifs, recursion and closures, and it can even take derivatives of derivatives
- [SymPy](#) is a Python library for symbolic mathematics.

- [scikit-learn](#) has simple and efficient tools for machine learning, data mining and data analysis
- [TensorFlow](#) is a Python library for fast numerical computing created and released by Google
- [Keras](#) is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano
- And many more such as [pytorch](#), [Theano](#) etc

## More Practicalities, handling arrays

### Basic Matrix Features, Numpy examples and Important Matrix and vector handling packages

Matrix properties reminder.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The inverse of a matrix is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I}$$

Relations	Name	matrix elements
$A = A^T$	symmetric	$a_{ij} = a_{ji}$
$A = (A^T)^{-1}$	real orthogonal	$\sum_k a_{ik} a_{jk} = \sum_k a_{ki} a_{kj} = \delta_{ij}$
$A = A^*$	real matrix	$a_{ij} = a_{ij}^*$
$A = A^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$A = (A^\dagger)^{-1}$	unitary	$\sum_k a_{ik} a_{jk}^* = \sum_k a_{ki}^* a_{kj} = \delta_{ij}$

### Some famous Matrices

- Diagonal if  $a_{ij} = 0$  for  $i \neq j$
- Upper triangular if  $a_{ij} = 0$  for  $i > j$
- Lower triangular if  $a_{ij} = 0$  for  $i < j$
- Upper Hessenberg if  $a_{ij} = 0$  for  $i > j + 1$
- Lower Hessenberg if  $a_{ij} = 0$  for  $i < j - 1$
- Tridiagonal if  $a_{ij} = 0$  for  $|i - j| > 1$



- Lower banded with bandwidth  $p$ :  $a_{ij} = 0$  for  $i > j + p$
- Upper banded with bandwidth  $p$ :  $a_{ij} = 0$  for  $i < j + p$
- Banded, block upper triangular, block lower triangular....

## More Basic Matrix Features

**Some Equivalent Statements.** For an  $N \times N$  matrix  $\mathbf{A}$  the following properties are all equivalent

- If the inverse of  $\mathbf{A}$  exists,  $\mathbf{A}$  is nonsingular.
- The equation  $\mathbf{Ax} = 0$  implies  $\mathbf{x} = 0$ .
- The rows of  $\mathbf{A}$  form a basis of  $R^N$ .
- The columns of  $\mathbf{A}$  form a basis of  $R^N$ .
- $\mathbf{A}$  is a product of elementary matrices.
- 0 is not eigenvalue of  $\mathbf{A}$ .

## Numpy and arrays

[Numpy](#) provides an easy way to handle arrays in Python. The standard way to import this library is as

```
import numpy as np
```

Here follows a simple example where we set up an array of ten elements, all determined by random numbers drawn according to the normal distribution,

```
n = 10
x = np.random.normal(size=n)
print(x)
```

We defined a vector  $x$  with  $n = 10$  elements with its values given by the Normal distribution  $N(0,1)$ . Another alternative is to declare a vector as follows

```
import numpy as np
x = np.array([1, 2, 3])
print(x)
```

Here we have defined a vector with three elements, with  $x_0 = 1$ ,  $x_1 = 2$  and  $x_2 = 3$ . Note that both Python and C++ start numbering array elements from 0 and on. This means that a vector with  $n$  elements has a sequence of entities  $x_0, x_1, x_2, \dots, x_{n-1}$ . We could also let (recommended) Numpy to compute the logarithms of a specific array as

```
import numpy as np
x = np.log(np.array([4, 7, 8]))
print(x)
```

## More Examples

In the last example we used Numpy's unary function *np.log*. This function is highly tuned to compute array elements since the code is vectorized and does not require looping. We normally recommend that you use the Numpy intrinsic functions instead of the corresponding **log** function from Python's **math** module. The looping is done explicitly by the **np.log** function. The alternative, and slower way to compute the logarithms of a vector would be to write

```
import numpy as np
from math import log
x = np.array([4, 7, 8])
for i in range(0, len(x)):
    x[i] = log(x[i])
print(x)
```

We note that our code is much longer already and we need to import the **log** function from the **math** module. The attentive reader will also notice that the output is `[1, 1, 2]`. Python interprets automatically our numbers as integers (like the **automatic** keyword in C++). To change this we could define our array elements to be double precision numbers as

```
import numpy as np
x = np.log(np.array([4, 7, 8], dtype = np.float64))
print(x)
```

or simply write them as double precision numbers (Python uses 64 bits as default for floating point type variables), that is

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0]))
print(x)
```

To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can simply use the **itemsize** functionality (the array *x* is actually an object which inherits the functionalities defined in Numpy) as

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0]))
print(x)
```

## Matrices in Python

Having defined vectors, we are now ready to try out matrices. We can define a  $3 \times 3$  real matrix **A** as (recall that we use lowercase letters for vectors and uppercase letters for matrices)

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
print(A)
```

If we use the **shape** function we would get (3,3) as output, that is verifying that our matrix is a  $3 \times 3$  matrix. We can slice the matrix and print for example the first column (Python organized matrix elements in a row-major order, see below) as

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[:,0])
```

We can continue this was by printing out other columns or rows. The example here prints out the second column

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[1,:])
```

Numpy contains many other functionalities that allow us to slice, subdivide etc etc arrays. We strongly recommend that you look up the [Numpy website for more details](#). Useful functions when defining a matrix are the **np.zeros** function which declares a matrix of a given dimension and sets all elements to zero

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to zero
A = np.zeros( (n, n) )
print(A)
```

or initializing all elements to

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to one
A = np.ones( (n, n) )
print(A)
```

or as unitarily distributed random numbers (see the material on random number generators in the statistics part)

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to random numbers with  $x \in [0, 1]$ 
A = np.random.rand(n, n)
print(A)
```

## More Examples, Covariance matrix

As we will see throughout these lectures, there are several extremely useful functionalities in Numpy. As an example, consider the discussion of the covariance matrix. Suppose we have defined three vectors  $\mathbf{x}, \mathbf{y}, \mathbf{z}$  with  $n$  elements each. The covariance matrix is defined as

$$\Sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix},$$

where for example

$$\sigma_{xy} = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y}).$$

The Numpy function **np.cov** calculates the covariance elements using the factor  $1/(n-1)$  instead of  $1/n$  since it assumes we do not have the exact mean values. The following simple function uses the **np.vstack** function which takes each vector of dimension  $1 \times n$  and produces a  $3 \times n$  matrix **W**

$$\mathbf{W} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \dots & \dots & \dots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & z_{n-1} \end{bmatrix},$$

## More on the Covariance Matrix

Our matrix is in turn converted into the  $3 \times 3$  covariance matrix **Σ** via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples **x** etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix through the **np.linalg.eig()** function.

```
# Importing various packages
import numpy as np

n = 100
x = np.random.normal(size=n)
print(np.mean(x))
y = 4+3*x+np.random.normal(size=n)
print(np.mean(y))
z = x**3+np.random.normal(size=n)
print(np.mean(z))
W = np.vstack((x, y, z))
Sigma = np.cov(W)
print(Sigma)
Eigvals, Eigvecs = np.linalg.eig(Sigma)
print(Eigvals)
```

## Practicalities, Reminder on Statistics

### Brief Reminder on Statistical Analysis

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  $\mu$ :

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

## 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  $\sigma_X^2$  or  $\text{var}(X)$ :

$$\sigma_X^2 = \text{var}(X) = \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x) dx \quad (1)$$

$$= \int (x^2 - 2x\langle x \rangle + \langle x \rangle^2) p(x) dx \quad (2)$$

$$= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 \quad (3)$$

$$= \langle x^2 \rangle - \langle x \rangle^2 \quad (4)$$

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  $P(x_1, \dots, x_n)$ . The *covariance* of two of the stochastic variables,  $X_i$  and  $X_j$ , is defined as follows:

$$\begin{aligned} \text{cov}(X_i, X_j) &\equiv \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \int \dots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) P(x_1, \dots, x_n) dx_1 \dots dx_n \end{aligned} \quad (5)$$

with

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

## Statistics, more covariance

If we consider the above covariance as a matrix  $C_{ij} = \text{cov}(X_i, X_j)$ , then the diagonal elements are just the familiar variances,  $C_{ii} = \text{cov}(X_i, X_i) = \text{var}(X_i)$ . 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  $X_j$ , ( $i \neq j$ ):

$$\text{cov}(X_i, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \quad (6)$$

$$= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \quad (7)$$

$$= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \quad (8)$$

$$= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \quad (9)$$

$$= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle \quad (10)$$

## 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  $\text{cov}(X_i, X_j) = 0$  ( $i \neq j$ ).

We normally use the acronym **iid** for independent and identically distributed stochastic variables.

## Statistics, more variance

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

$$\text{var}(U) = \sum_{i,j} a_i a_j \text{cov}(X_i, X_j) \quad (11)$$

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

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

and

$$\text{var}\left(\sum_i a_i X_i\right) = \sum_i a_i^2 \text{var}(X_i)$$

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  $\mu_X$  and the variance  $\sigma_X$ .

## 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:

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

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

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

## 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. (5) 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  $\mu_X$ , variance  $\text{var}(X)$  and covariance  $\text{cov}(X, Y)$ .

## 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  $\mu_X$  of the chosen PDF:

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

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

What we need to find out is how good an approximation  $\bar{x}_n$  is to  $\mu_X$ . 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  $\text{err}_X$ . 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  $\bar{X}_n$  for the sample mean  $\bar{x}_n$  is then just a linear combination, already familiar to us:

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

All the coefficients are just equal  $1/n$ . The PDF of  $\bar{X}_n$ , denoted by  $p_{\bar{X}_n}(x)$  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  $x_1, x_2, \dots, x_n$  with the constraint that



the mean of the set  $\{x_i\}$  is  $\bar{x}_n$ :

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

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

## Statistics, central limit theorem

It is generally not possible to express  $p_{\bar{X}_n}(x)$  in a closed form given an arbitrary PDF  $p_X$  and a number  $n$ . But for the limit  $n \rightarrow \infty$  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,  $p_{\bar{X}_n}(x)$  approaches a Gaussian distribution whose mean and variance equal the true mean and variance,  $\mu_X$  and  $\sigma_X^2$ , respectively:

$$\lim_{n \rightarrow \infty} p_{\bar{X}_n}(x) = \left(\frac{n}{2\pi\text{var}(X)}\right)^{1/2} e^{-\frac{n(x-\bar{x}_n)^2}{2\text{var}(X)}} \quad (12)$$

## Covariance example

Suppose we have defined three vectors  $\mathbf{x}, \mathbf{y}, \mathbf{z}$  with  $n$  elements each. The covariance matrix is defined as

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix},$$

where for example

$$\sigma_{xy} = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y}).$$

The Numpy function **np.cov** calculates the covariance elements using the factor  $1/(n-1)$  instead of  $1/n$  since it assumes we do not have the exact mean values.

The following simple function uses the **np.vstack** function which takes each vector of dimension  $1 \times n$  and produces a  $3 \times n$  matrix  $\mathbf{W}$

$$\mathbf{W} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \dots & \dots & \dots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & z_{n-1} \end{bmatrix},$$

which in turn is converted into into the  $3 \times 3$  covariance matrix  $\mathbf{\Sigma}$  via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples  $\mathbf{x}$  etc using the Numpy function **np.mean(x)**. We can also

extract the eigenvalues of the covariance matrix through the `np.linalg.eig()` function.

## Covariance in numpy

```
# Importing various packages
import numpy as np

n = 100
x = np.random.normal(size=n)
print(np.mean(x))
y = 4+3*x+np.random.normal(size=n)
print(np.mean(y))
z = x*3+np.random.normal(size=n)
print(np.mean(z))
W = np.vstack((x, y, z))
Sigma = np.cov(W)
print(Sigma)
```

## Practicalities, Useful Python Packages

### Meet the Pandas



Another useful Python package is [pandas](#), which is an open source library providing high-performance, easy-to-use data structures and data analysis tools for Python. **pandas** stands for panel data, a term borrowed from econometrics and is an efficient library for data analysis with an emphasis on tabular data. **pandas** has two major classes, the **DataFrame** class with two-dimensional data objects and tabular data organized in columns and the class **Series** with a focus on one-dimensional data objects. Both classes allow you to index data easily as we will see in the examples below. **pandas** allows you also to perform mathematical operations on the data, spanning from simple reshaping of vectors and matrices to statistical operations.

The following simple example shows how we can, in an easy way make tables of our data. Here we define a data set which includes names, place of birth and date of birth, and displays the data in an easy to read way. We will see repeated use of **pandas**, in particular in connection with classification of data.

```
import pandas as pd
from IPython.display import display
data = {'First Name': ["Frodo", "Bilbo", "Aragorn II", "Samwise"],
        'Last Name': ["Baggins", "Baggins", "Elessar", "Gamgee"],
        'Place of birth': ["Shire", "Shire", "Eriador", "Shire"],
        'Date of Birth T.A.': [2968, 2890, 2931, 2980]}
data_pandas = pd.DataFrame(data)
display(data_pandas)
```

## Data Frames in Pandas

In the above we have imported **pandas** with the shorthand **pd**, the latter has become the standard way we import **pandas**. We make then a list of various variables and reorganize the aboves lists into a **DataFrame** and then print out a neat table with specific column labels as *Name*, *place of birth* and *date of birth*. Displaying these results, we see that the indices are given by the default numbers from zero to three. **pandas** is extremely flexible and we can easily change the above indices by defining a new type of indexing as

```
data_pandas = pd.DataFrame(data, index=['Frodo', 'Bilbo', 'Aragorn', 'Sam'])
display(data_pandas)
```

Thereafter we display the content of the row which begins with the index **Aragorn**

```
display(data_pandas.loc['Aragorn'])
```

We can easily append data to this, for example

```
new_hobbit = {'First Name': ["Peregrin"],
              'Last Name': ["Took"],
              'Place of birth': ["Shire"],
              'Date of Birth T.A.': [2990]}
data_pandas=data_pandas.append(pd.DataFrame(new_hobbit, index=['Pippin']))
display(data_pandas)
```

## More Pandas

Here are other examples where we use the **DataFrame** functionality to handle arrays, now with more interesting features for us, namely numbers. We set up a matrix of dimensionality  $10 \times 5$  and compute the mean value and standard deviation of each column. Similarly, we can perform mathematical operations like squaring the matrix elements and many other operations.

```
import numpy as np
import pandas as pd
from IPython.display import display
```

```

np.random.seed(100)
# setting up a 10 x 5 matrix
rows = 10
cols = 5
a = np.random.randn(rows,cols)
df = pd.DataFrame(a)
display(df)
print(df.mean())
print(df.std())
display(df**2)

```

Thereafter we can select specific columns only and plot final results

```

df.columns = ['First', 'Second', 'Third', 'Fourth', 'Fifth']
df.index = np.arange(10)

display(df)
print(df['Second'].mean() )
print(df.info())
print(df.describe())

from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

df.cumsum().plot(lw=2.0, figsize=(10,6))
plt.show()

df.plot.bar(figsize=(10,6), rot=15)
plt.show()

```

We can produce a  $4 \times 4$  matrix

```

b = np.arange(16).reshape((4,4))
print(b)
df1 = pd.DataFrame(b)
print(df1)

```

and many other operations.

## Pandas Series

The **Series** class is another important class included in **pandas**. You can view it as a specialization of **DataFrame** but where we have just a single column of data. It shares many of the same features as **DataFrame**. As with **DataFrame**, most operations are vectorized, achieving thereby a high performance when dealing with computations of arrays, in particular labeled arrays. As we will see below it leads also to a very concise code close to the mathematical operations we may be interested in. For multidimensional arrays, we also recommend **xarray**. **xarray** has much of the same flexibility as **pandas**, but allows for the extension to higher dimensions than two. We will see examples later of the usage of both **pandas** and **xarray**.

## Our first Machine Learning Encounter

### Reading Data and Fitting

In order to study various Machine Learning algorithms, we need to access data. Accessing data is an essential step in all machine learning algorithms. In particular, setting up the so-called **design matrix** (to be defined below) is often the first element we need in order to perform our calculations. To set up the design matrix means reading (and later, when the calculations are done, writing) data in various formats. The formats span from reading files from disk, loading data from databases and interacting with online sources like web application programming interfaces (APIs).

In handling various input formats, as discussed above, we will often stay with **pandas**, a Python package which allows us, in a seamless and painless way, to deal with a multitude of formats, from standard **csv** (comma separated values) files, via **excel**, **html** to **hdf5** formats. With **pandas** and the **DataFrame** and **Series** functionalities we are able to convert text data into the calculational formats we need for a specific algorithm. And our code is going to be pretty close the basic mathematical expressions.

Our first data set is going to be a classic from nuclear physics, namely all available data on binding energies. Don't be intimidated if you are not familiar with nuclear physics. It serves simply as an example here of a data set.

We will show some of the strengths of packages like **Scikit-Learn** in fitting nuclear binding energies to specific functions using linear regression first. Then, as a teaser, we will show you how you can easily implement other algorithms like decision trees and random forests and neural networks.

But before we really start with nuclear physics data, let's just look at some simpler polynomial fitting cases, such as, (don't be offended) fitting straight lines!

### 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 a function  $y$  in terms of the variable  $x$ . Both are defined as vectors with 100 entries. The numbers in the vector  $\mathbf{x}$  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.

## Simple linear regression model using scikit-learn, Numpy functions

The Numpy functions are imported using 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 specify 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  $\mu$  equal to zero and variance  $\sigma^2$  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  $(x, y)$  for our training data. The Python package **scikit-learn** has also a functionality which extracts the above fitting parameters  $\alpha$  and  $\beta$  (see below). Later we will distinguish between training data and test data.

## Simple linear regression model using scikit-learn, Matplotlib

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, what to expect

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. Does the fit look better? Indeed, by reducing the role of the noise given by the normal distribution we see immediately that our linear prediction seemingly reproduces better the training set. However, this testing 'by the eye' is obviously 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.

## Simple linear regression model, how to evaluate the model

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  $\chi^2$  function (a variant of the mean-squared error (MSE))

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

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

## Our first Cost/Loss function encounter

Minimizing the cost function is a central aspect of our discussions to come. Finding its minima as function of the model parameters ( $\alpha$  and  $\beta$  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  $\chi^2$  function becomes smaller.

## Our first Cost/Loss function encounter

The terms cost and loss functions are often synonymous, sometimes you will also encounter the usage error function. The more general scenario is to define an objective function first, which we want to optimize. It is common to see statements like this however: **The loss function computes the error for a single training example, while the cost function is the average of the loss functions of the entire training set.**

## Our first Cost/Loss function encounter, how do we define them?

There are many ways to define the cost/loss 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 (why would we prefer the MSE instead of the relative error?) as

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

The squared cost function results in an arithmetic mean-unbiased estimator, and the absolute-value cost function results in a median-unbiased estimator (in the one-dimensional case, and a geometric median-unbiased estimator for the multi-dimensional case). The squared cost function has the disadvantage that it has the tendency to be dominated by outliers.

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.

## Scikit-Learn functionality

As mentioned above, **Scikit-Learn** has an impressive functionality. We can for example extract the values of  $\alpha$  and  $\beta$  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_error

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()
```

The function **coef** gives us the parameter  $\beta$  of our fit while **intercept** yields  $\alpha$ . 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

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

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  $\chi^2$  function defined above.

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  $\mathbf{y}$ , disregarding the input features, would get a  $R^2$  score of 0.0.

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

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

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

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

Another quantity that we will meet again in our discussions of regression analysis is the mean absolute error (MAE), a risk metric corresponding to the expected value of the absolute error loss or what we call the  $l_1$ -norm loss. In our discussion above we presented the relative error. The MAE is defined as follows

$$\text{MAE}(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} |y_i - \tilde{y}_i|.$$

We present the squared logarithmic (quadratic) error

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

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.

Finally, another cost function is the Huber cost function used in robust regression.

The rationale behind this possible cost function is its reduced sensitivity to outliers in the data set. In our discussions on dimensionality reduction and normalization of data we will meet other ways of dealing with outliers.

The Huber cost function is defined as

$$H_\delta(a) = \begin{cases} \frac{1}{2}a^2, & \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

Here  $a = \mathbf{y} - \tilde{\mathbf{y}}$ . We will discuss in more detail these and other functions in the various lectures.

## Cubic Polynomial

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.

```
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))

```

## Getting more serious, fitting Nuclear Binding Energies

### To our real data: nuclear binding energies. Brief reminder on masses and binding energies

Let us now dive into nuclear physics and remind ourselves briefly about some basic features about binding energies. A basic quantity which can be measured for the ground states of nuclei is the atomic mass  $M(N, Z)$  of the neutral atom with atomic mass number  $A$  and charge  $Z$ . The number of neutrons is  $N$ . There are indeed several sophisticated experiments worldwide which allow us to measure this quantity to high precision (parts per million even).

Atomic masses are usually tabulated in terms of the mass excess defined by

$$\Delta M(N, Z) = M(N, Z) - uA,$$

where  $u$  is the Atomic Mass Unit

$$u = M(^{12}\text{C})/12 = 931.4940954(57) \text{ MeV}/c^2.$$

The nucleon masses are

$$m_p = 1.00727646693(9)u,$$

and

$$m_n = 939.56536(8) \text{ MeV}/c^2 = 1.0086649156(6)u.$$

In the 2016 mass evaluation of by W.J.Huang, G.Audi, M.Wang, F.G.Kondev, S.Naimi and X.Xu there are data on masses and decays of 3437 nuclei.

The nuclear binding energy is defined as the energy required to break up a given nucleus into its constituent parts of  $N$  neutrons and  $Z$  protons. In terms of the atomic masses  $M(N, Z)$  the binding energy is defined by

$$BE(N, Z) = ZM_Hc^2 + Nm_nc^2 - M(N, Z)c^2,$$

where  $M_H$  is the mass of the hydrogen atom and  $m_n$  is the mass of the neutron. In terms of the mass excess the binding energy is given by

$$BE(N, Z) = Z\Delta_Hc^2 + N\Delta_nc^2 - \Delta(N, Z)c^2,$$

where  $\Delta_Hc^2 = 7.2890$  MeV and  $\Delta_nc^2 = 8.0713$  MeV.

A popular and physically intuitive model which can be used to parametrize the experimental binding energies as function of  $A$ , is the so-called **liquid drop model**. The ansatz is based on the following expression

$$BE(N, Z) = a_1A - a_2A^{2/3} - a_3\frac{Z^2}{A^{1/3}} - a_4\frac{(N - Z)^2}{A},$$

where  $A$  stands for the number of nucleons and the  $a_i$ s are parameters which are determined by a fit to the experimental data.

To arrive at the above expression we have assumed that we can make the following assumptions:

- There is a volume term  $a_1A$  proportional with the number of nucleons (the energy is also an extensive quantity). When an assembly of nucleons of the same size is packed together into the smallest volume, each interior nucleon has a certain number of other nucleons in contact with it. This contribution is proportional to the volume.
- There is a surface energy term  $a_2A^{2/3}$ . The assumption here is that a nucleon at the surface of a nucleus interacts with fewer other nucleons than one in the interior of the nucleus and hence its binding energy is less. This surface energy term takes that into account and is therefore negative and is proportional to the surface area.
- There is a Coulomb energy term  $a_3\frac{Z^2}{A^{1/3}}$ . The electric repulsion between each pair of protons in a nucleus yields less binding.
- There is an asymmetry term  $a_4\frac{(N-Z)^2}{A}$ . This term is associated with the Pauli exclusion principle and reflects the fact that the proton-neutron interaction is more attractive on the average than the neutron-neutron and proton-proton interactions.

We could also add a so-called pairing term, which is a correction term that arises from the tendency of proton pairs and neutron pairs to occur. An even number of particles is more stable than an odd number.

**Organizing our data.** Let us start with reading and organizing our data. We start with the compilation of masses and binding energies from 2016. After having downloaded this file to our own computer, we are now ready to read the file and start structuring our data.

We start with preparing folders for storing our calculations and the data file over masses and binding energies. We import also various modules that we will find useful in order to present various Machine Learning methods. Here we focus mainly on the functionality of **scikit-learn**.

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

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

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

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

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

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

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

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

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

Our next step is to read the data on experimental binding energies and reorganize them as functions of the mass number  $A$ , the number of protons  $Z$  and neutrons  $N$  using **pandas**. Before we do this it is always useful (unless you have a binary file or other types of compressed data) to actually open the file and simply take a look at it!

In particular, the program that outputs the final nuclear masses is written in Fortran with a specific format. It means that we need to figure out the format and which columns contain the data we are interested in. Pandas comes with a function that reads formatted output. After having admired the file, we are now ready to start massaging it with **pandas**. The file begins with some basic format information.

```
"""
This is taken from the data file of the mass 2016 evaluation.
All files are 3436 lines long with 124 character per line.
Headers are 39 lines long.
col 1      : Fortran character control: 1 = page feed 0 = line feed
format     : a1,i3,i5,i5,i5,1x,a3,a4,1x,f13.5,f11.5,f11.3,f9.3,1x,a2,f11.3,f9.3,1x,i3,1x,f12.5
```

*These formats are reflected in the pandas widths variable below, see the statement widths=(1,3,5,5,5,1,3,4,1,13,11,11,9,1,2,11,9,1,3,1,12,11,1), Pandas has also a variable header, with length 39 in this case.*

The data we are interested in are in columns 2, 3, 4 and 11, giving us the number of neutrons, protons, mass numbers and binding energies, respectively. We add also for the sake of completeness the element name. The data are in fixed-width formatted lines and we will convert them into the **pandas** DataFrame structure.

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

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

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

We have now read in the data, grouped them according to the variables we are interested in. We see how easy it is to reorganize the data using **pandas**. If we were to do these operations in C/C++ or Fortran, we would have had to write various functions/subroutines which perform the above reorganizations for us. Having reorganized the data, we can now start to make some simple fits using both the functionalities in **numpy** and **Scikit-Learn** afterwards.

Now we define five variables which contain the number of nucleons  $A$ , the number of protons  $Z$  and the number of neutrons  $N$ , the element name and finally the energies themselves.

```
A = Masses['A']
Z = Masses['Z']
N = Masses['N']
Element = Masses['Element']
Energies = Masses['Ebinding']
print(Masses)
```

The next step, and we will define this mathematically later, is to set up the so-called **design/feature matrix**. We will throughout label this matrix as **X**. It has dimensionality  $n \times p$ , where  $n$  is the number of data points and  $p$  are the so-called features/predictors. In our case here they are given by the number of polynomials in  $A$  we wish to include in the fit.

```
# Now we set up the design matrix X
X = np.zeros((len(A),5))
```

```

X[:,0] = 1
X[:,1] = A
X[:,2] = A**(2.0/3.0)
X[:,3] = A**(-1.0/3.0)
X[:,4] = A**(-1.0)

```

With **scikitlearn** we are now ready to use linear regression and fit our data.

```

clf = skl.LinearRegression().fit(X, Energies)
fity = clf.predict(X)

```

Pretty simple!

Now we can print measures of how our fit is doing, the coefficients from the fits and plot the final fit together with our data.

```

# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(Energies, fity))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(Energies, fity))
# Mean absolute error
print('Mean absolute error: %.2f' % mean_absolute_error(Energies, fity))
print(clf.coef_, clf.intercept_)

Masses['Eapprox'] = fity
# Generate a plot comparing the experimental with the fitted values values.
fig, ax = plt.subplots()
ax.set_xlabel(r'$A = N + Z$')
ax.set_ylabel(r'$E_{\mathrm{bind}} \backslash, / \mathrm{MeV}$')
ax.plot(Masses['A'], Masses['Ebinding'], alpha=0.7, lw=2,
        label='Ame2016')
ax.plot(Masses['A'], Masses['Eapprox'], alpha=0.7, lw=2, c='m',
        label='Fit')
ax.legend()
save_fig("Masses2016")
plt.show()

```

**Seeing the wood for the trees.** As a teaser, let us now see how we can do this with decision trees using **scikit-learn**. We will discuss the method in more details later.

```

#Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
regr_1=DecisionTreeRegressor(max_depth=5)
regr_2=DecisionTreeRegressor(max_depth=7)
regr_3=DecisionTreeRegressor(max_depth=9)
regr_1.fit(X, Energies)
regr_2.fit(X, Energies)
regr_3.fit(X, Energies)

y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)
y_3=regr_3.predict(X)
Masses['Eapprox'] = y_3
# Plot the results
plt.figure()
plt.plot(A, Energies, color="blue", label="Data", linewidth=2)
plt.plot(A, y_1, color="red", label="max_depth=5", linewidth=2)
plt.plot(A, y_2, color="green", label="max_depth=7", linewidth=2)

```

```

plt.plot(A, y_3, color="m", label="max_depth=9", linewidth=2)

plt.xlabel("$A$")
plt.ylabel("$E$ [MeV]")
plt.title("Decision Tree Regression")
plt.legend()
save_fig("Masses2016Trees")
plt.show()
print(Masses)
print(np.mean( (Energies-y_1)**2))

```

And what about using neural networks? The **seaborn** package allows us to visualize data in an efficient way. Note that we use **scikit-learn**'s multi-layer perceptron (or feed forward neural network) functionality.

```

from sklearn.neural_network import MLPRegressor
from sklearn.metrics import accuracy_score
import seaborn as sns

X_train = X
Y_train = Energies
n_hidden_neurons = 100
epochs = 100
# store models for later use
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
# store the models for later use
DNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)
train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
sns.set()
for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = MLPRegressor(hidden_layer_sizes=(n_hidden_neurons), activation='logistic',
                           alpha=lmbd, learning_rate_init=eta, max_iter=epochs)
        dnn.fit(X_train, Y_train)
        DNN_scikit[i][j] = dnn
        train_accuracy[i][j] = dnn.score(X_train, Y_train)

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

```

## More on flexibility with pandas and xarray

Let us study the  $Q$  values associated with the removal of one or two nucleons from a nucleus. These are conventionally defined in terms of the one-nucleon and two-nucleon separation energies. With the functionality in **pandas**, two to three lines of code will allow us to plot the separation energies. The neutron separation energy is defined as

$$S_n = -Q_n = BE(N, Z) - BE(N - 1, Z),$$



and the proton separation energy reads

$$S_p = -Q_p = BE(N, Z) - BE(N, Z - 1).$$

The two-neutron separation energy is defined as

$$S_{2n} = -Q_{2n} = BE(N, Z) - BE(N - 2, Z),$$

and the two-proton separation energy is given by

$$S_{2p} = -Q_{2p} = BE(N, Z) - BE(N, Z - 2).$$

Using say the neutron separation energies (alternatively the proton separation energies)

$$S_n = -Q_n = BE(N, Z) - BE(N - 1, Z),$$

we can define the so-called energy gap for neutrons (or protons) as

$$\Delta S_n = BE(N, Z) - BE(N - 1, Z) - (BE(N + 1, Z) - BE(N, Z)),$$

or

$$\Delta S_n = 2BE(N, Z) - BE(N - 1, Z) - BE(N + 1, Z).$$

This quantity can in turn be used to determine which nuclei could be interpreted as magic or not. For protons we would have

$$\Delta S_p = 2BE(N, Z) - BE(N, Z - 1) - BE(N, Z + 1).$$

To calculate say the neutron separation we need to multiply our masses with the nucleon number  $A$  (why?). Thereafter we pick the oxygen isotopes and simply compute the separation energies with two lines of code (note that most of the code here is a repeat of what you have seen before).

```
# Common imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

def MakePlot(x,y, styles, labels, axlabels):
    plt.figure(figsize=(10,6))
    for i in range(len(x)):
        plt.plot(x[i], y[i], styles[i], label = labels[i])
    plt.xlabel(axlabels[0])
    plt.ylabel(axlabels[1])
    plt.legend(loc=0)

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

```

DATA_ID = "DataFiles/"

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

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

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

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

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

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

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

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

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

df = pd.DataFrame({'A':A, 'Z':Z, 'N':N, 'Element':Element, 'Energies':Energies})
# Here we pick the oxygen isotopes
Nucleus = df.loc[lambd df: df.Z==8, :]
# drop cases with no number
Nucleus = Nucleus.dropna()
# Here we do the magic and obtain the neutron separation energies, one line of code!!
Nucleus['NeutronSeparationEnergies'] = Nucleus['Energies'].diff(+1)
print(Nucleus)
MakePlot([Nucleus.A], [Nucleus.NeutronSeparationEnergies], ['b'], ['Neutron Separation Energy'],
save_fig('Nucleus')
plt.show()

```

## Linear Regression, basic overview

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

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

We will in particular focus on

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

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

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

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

## Additional Reading

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

## Regression Analysis, Definitions and Aims

### Regression analysis, overarching aims

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

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

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

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

### Regression analysis, overarching aims II

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

The matrix  $\mathbf{X}$  is called the *design matrix*. Additional information of the samples is available in the form of  $\mathbf{y}$  (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  $\boldsymbol{\beta} = [\beta_0, \dots, \beta_{p-1}]^T$  are the *regression parameters*.

Linear regression gives us a set of analytical equations for the parameters  $\beta_j$ .

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

## Examples

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

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

$$BE(A) = a_0 + a_1A + a_2A^{2/3} + a_3A^{-1/3} + a_4A^{-1},$$

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

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

## General linear models

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

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

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

where  $\epsilon_i$  is the error in our approximation.

## Rewriting the fitting procedure as a linear algebra problem

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

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

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

Defining the vectors

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

and

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

and

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

and the design matrix

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

we can rewrite our equations as

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

The above design matrix is called a [Vandermonde matrix](#).

## Generalizing the fitting procedure as a linear algebra problem

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

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

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

## Generalizing the fitting procedure as a linear algebra problem

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

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

and without loss of generality we rewrite again our equations as

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

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

## Optimizing our parameters

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

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

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

## Our model for the nuclear binding energies

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

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

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

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

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

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

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

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

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

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

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

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

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



```

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

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

```

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

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

throughout these lectures.

## Optimizing our parameters, more details

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

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

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

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

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

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

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

It is also common to define the function  $C$  as

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

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

## Interpretations and optimizing our parameters

The function

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

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

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

where  $\langle y_i \rangle$  is the mean value. Keep in mind also that till now we have treated  $y_i$  as the exact value. Normally, the response (dependent or outcome) variable  $y_i$  the outcome of a numerical experiment or another type of experiment and is thus only an approximation to the true value. It is then always accompanied by an error estimate, often limited to a statistical error estimate given by the standard deviation discussed earlier. In the discussion here we will treat  $y_i$  as our exact value for the response variable.

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

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

In practical terms it means we will require

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

which results in

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

or in a matrix-vector form as

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

## Interpretations and optimizing our parameters

We can rewrite

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

as

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

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

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

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

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

### Some useful matrix and vector expressions

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

$$\begin{aligned}\frac{\partial(\mathbf{b}^T \mathbf{a})}{\partial \mathbf{a}} &= \mathbf{b}, \\ \frac{\partial(\mathbf{a}^T \mathbf{A} \mathbf{a})}{\partial \mathbf{a}} &= (\mathbf{A} + \mathbf{A}^T) \mathbf{a}, \\ \frac{\partial \text{tr}(\mathbf{B} \mathbf{A})}{\partial \mathbf{A}} &= \mathbf{B}^T, \\ \frac{\partial \log |\mathbf{A}|}{\partial \mathbf{A}} &= (\mathbf{A}^{-1})^T.\end{aligned}$$

### Interpretations and optimizing our parameters

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

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

and with

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

we have

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

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

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

## Own code for Ordinary Least Squares

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

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

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

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

And finally we plot our fit with and compare with data

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

## Adding error analysis and training set up

We can easily test our fit by computing the  $R^2$  score that we discussed in connection with the functionality of **Scikit-Learn** in the introductory slides. Since we are not using **Scikit-Learn** here we can define our own  $R^2$  function as

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

and we would be using it as

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

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

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

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

and finally the relative error as

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

## The $\chi^2$ function

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

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

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

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

## The $\chi^2$ function

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

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

which results in

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

or in a matrix-vector form as

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

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

## The $\chi^2$ function

We can rewrite

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

as

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

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

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

## The $\chi^2$ function

If we then introduce the matrix

$$\mathbf{H} = (\mathbf{A}^T \mathbf{A})^{-1},$$

we have then the following expression for the parameters  $\beta_j$  (the matrix elements of  $\mathbf{H}$  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  $\beta_j$  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 $\chi^2$ 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  $\chi^2$  with respect to  $\beta_0$  and  $\beta_1$  show that these are given by

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

and

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

## The $\chi^2$ function

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

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

$$\begin{aligned}\gamma_x &= \sum_{i=0}^{n-1} \frac{x_i}{\sigma_i^2}, \\ \gamma_y &= \sum_{i=0}^{n-1} \left( \frac{y_i}{\sigma_i^2} \right), \\ \gamma_{xx} &= \sum_{i=0}^{n-1} \frac{x_i x_i}{\sigma_i^2}, \\ \gamma_{xy} &= \sum_{i=0}^{n-1} \frac{y_i x_i}{\sigma_i^2},\end{aligned}$$

we obtain

$$\begin{aligned}\beta_0 &= \frac{\gamma_{xx}\gamma_y - \gamma_x\gamma_y}{\gamma\gamma_{xx} - \gamma_x^2}, \\ \beta_1 &= \frac{\gamma_{xy}\gamma - \gamma_x\gamma_y}{\gamma\gamma_{xx} - \gamma_x^2}.\end{aligned}$$

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

## Regression Examples

### Fitting an Equation of State for Dense Nuclear Matter

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

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

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

## The code

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

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

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

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

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

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

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

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

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

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

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



```

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

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

```

The above simple polynomial in density  $\rho$  gives an excellent fit to the data.

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

## Splitting our Data in Training and Test data

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

```

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

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

```

```

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

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

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

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

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

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

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

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

```

## The Boston housing data example

The Boston housing data set was originally a part of UCI Machine Learning Repository and has been removed now. The data set is now included in **Scikit-Learn**'s library. There are 506 samples and 13 feature (predictor) variables in

this data set. The objective is to predict the value of prices of the house using the features (predictors) listed here.

The features/predictors are

1. CRIM: Per capita crime rate by town
2. ZN: Proportion of residential land zoned for lots over 25000 square feet
3. INDUS: Proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: Nitric oxide concentration (parts per 10 million)
6. RM: Average number of rooms per dwelling
7. AGE: Proportion of owner-occupied units built prior to 1940
8. DIS: Weighted distances to five Boston employment centers
9. RAD: Index of accessibility to radial highways
10. TAX: Full-value property tax rate per USD10000
11. B:  $1000(Bk - 0.63)^2$ , where  $Bk$  is the proportion of [people of African American descent] by town
12. LSTAT: Percentage of lower status of the population
13. MEDV: Median value of owner-occupied homes in USD 1000s

## Housing data, the code

We start by importing the libraries

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

import pandas as pd
import seaborn as sns
```

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

```
from sklearn.datasets import load_boston

boston_dataset = load_boston()

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

Then we invoke Pandas

```

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

```

and preprocess the data

```

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

```

We can then visualize the data

```

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

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

```

It is now useful to look at the correlation matrix

```

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

```

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

```

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

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

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

```

Now we start training our model

```

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

```

We split the data into training and test sets

```

from sklearn.model_selection import train_test_split

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

```

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

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

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

# model evaluation for training set

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

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

# model evaluation for testing set

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

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

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

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

## Reducing the number of degrees of freedom, overarching view

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

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

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

## Preprocessing our data

Before we proceed however, we will discuss how to preprocess our data. Till now and in connection with our previous examples we have not met so many cases where we are too sensitive to the scaling of our data. Normally the data may need a rescaling and/or may be sensitive to extreme values. Scaling the data renders our inputs much more suitable for the algorithms we want to employ.

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

## More preprocessing

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

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

## Simple preprocessing examples, Franke function and regression

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

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

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

```

os.mkdir(PROJECT_ROOT_DIR)

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

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

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

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

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

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

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

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

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

    return X

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

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

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

scaler = StandardScaler()

```

```

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

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

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

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

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

```

## Singular Value Decomposition Algorithm

### The singular value decomposition

The examples we have looked at so far are cases where we normally can invert the matrix  $\mathbf{X}^T \mathbf{X}$ . Using a polynomial expansion as we did both for the masses and the fitting of the equation of state, leads to row vectors of the design matrix which are essentially orthogonal due to the polynomial character of our model. Obtaining the inverse of the design matrix is then often done via a so-called LU, QR or Cholesky decomposition.

This may however not be the case in general and a standard matrix inversion algorithm based on say LU, QR or Cholesky decomposition may lead to singularities. We will see examples of this below.

There is however a way to partially circumvent this problem and also gain some insight about the ordinary least squares approach.

This is given by the **Singular Value Decomposition** algorithm, perhaps the most powerful linear algebra algorithm. Let us look at a different example where we may have problems with the standard matrix inversion algorithm. Thereafter we dive into the math of the SVD.

### Linear Regression Problems

One of the typical problems we encounter with linear regression, in particular when the matrix  $\mathbf{X}$  (our so-called design matrix) is high-dimensional, are problems with near singular or singular matrices. The column vectors of  $\mathbf{X}$  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 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  $\mathbf{X}$  are linearly dependent. We see this easily since 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  $\mathbf{X}^T \mathbf{X}$  (the matrix we need to 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

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

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

## Fixing the singularity

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

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}, \quad (13)$$

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

A cheap *ad hoc* approach is simply to add a small diagonal component to the matrix to invert, that is we change

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

where  $\mathbf{I}$  is the identity matrix. When we discuss **Ridge** regression this is actually what we end up evaluating. The parameter  $\lambda$  is called a hyperparameter. More about this later.

## Basic math of the SVD

From standard linear algebra we know that a square matrix  $\mathbf{X}$  can be diagonalized if and only if it is a so-called **normal matrix**, that is if  $\mathbf{X} \in \mathbb{R}^{n \times n}$  we have  $\mathbf{X} \mathbf{X}^T = \mathbf{X}^T \mathbf{X}$  or if  $\mathbf{X} \in \mathbb{C}^{n \times n}$  we have  $\mathbf{X} \mathbf{X}^\dagger = \mathbf{X}^\dagger \mathbf{X}$ . The matrix has then a set of eigenpairs

$(\lambda_1, \mathbf{u}_1), \dots, (\lambda_n, \mathbf{u}_n)$ , and the eigenvalues are given by the diagonal matrix  $\boldsymbol{\Sigma} = \text{Diag}(\lambda_1, \dots, \lambda_n)$ .

The matrix  $\mathbf{X}$  can be written in terms of an orthogonal/unitary transformation  $\mathbf{U}$

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T,$$

with  $\mathbf{U}\mathbf{U}^T = \mathbf{I}$  or  $\mathbf{U}\mathbf{U}^\dagger = \mathbf{I}$ .

Not all square matrices are diagonalizable. A matrix like the one discussed above

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

is not diagonalizable, it is a so-called **defective matrix**. It is easy to see that the condition  $\mathbf{X}\mathbf{X}^T = \mathbf{X}^T\mathbf{X}$  is not fulfilled.

## The SVD, a Fantastic Algorithm

However, and this is the strength of the SVD algorithm, any general matrix  $\mathbf{X}$  can be decomposed in terms of a diagonal matrix and two orthogonal/unitary matrices. The **Singular Value Decomposition (SVD) theorem** states that a general  $m \times n$  matrix  $\mathbf{X}$  can be written in terms of a diagonal matrix  $\mathbf{\Sigma}$  of dimensionality  $n \times n$  and two orthogonal matrices  $\mathbf{U}$  and  $\mathbf{V}$ , where the first has dimensionality  $m \times m$  and the last dimensionality  $n \times n$ . We have then

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

As an example, the above defective matrix can be decomposed as

$$\mathbf{X} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T,$$

with eigenvalues  $\sigma_1 = 2$  and  $\sigma_2 = 0$ . The SVD exists always!

## Another Example

Consider the following matrix which can be SVD decomposed as

$$\mathbf{X} = \frac{1}{15} \begin{bmatrix} 14 & 2 \\ 4 & 22 \\ 16 & 13 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 2 & 2 \\ 2 & -1 & 1 \\ 2 & 1 & -2 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \frac{1}{5} \begin{bmatrix} 3 & 4 \\ 4 & -3 \end{bmatrix} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T.$$

This is a  $3 \times 2$  matrix which is decomposed in terms of a  $3 \times 3$  matrix  $\mathbf{U}$ , and a  $2 \times 2$  matrix  $\mathbf{V}$ . It is easy to see that  $\mathbf{U}$  and  $\mathbf{V}$  are orthogonal (how?).

And the SVD decomposition (singular values) gives eigenvalues  $\sigma_i \geq \sigma_{i+1}$  for all  $i$  and for dimensions larger than  $i = 2$ , the eigenvalues (singular values) are zero.

In the general case, where our design matrix  $\mathbf{X}$  has dimension  $n \times p$ , the matrix is thus decomposed into an  $n \times n$  orthogonal matrix  $\mathbf{U}$ , a  $p \times p$  orthogonal matrix  $\mathbf{V}$  and a diagonal matrix  $\mathbf{\Sigma}$  with  $r = \min(n, p)$  singular values  $\sigma_i \geq 0$  on the main diagonal and zeros filling the rest of the matrix. There are at most  $p$

singular values assuming that  $n > p$ . In our regression examples for the nuclear masses and the equation of state this is indeed the case, while for the Ising model we have  $p > n$ . These are often cases that lead to near singular or singular matrices.

The columns of  $U$  are called the left singular vectors while the columns of  $V$  are the right singular vectors.

## Economy-size SVD

If we assume that  $n > p$ , then our matrix  $U$  has dimension  $n \times n$ . The last  $n - p$  columns of  $U$  become however irrelevant in our calculations since they are multiplied with the zeros in  $\Sigma$ .

The economy-size decomposition removes extra rows or columns of zeros from the diagonal matrix of singular values,  $\Sigma$ , along with the columns in either  $U$  or  $V$  that multiply those zeros in the expression. Removing these zeros and columns can improve execution time and reduce storage requirements without compromising the accuracy of the decomposition.

If  $n > p$ , we keep only the first  $p$  columns of  $U$  and  $\Sigma$  has dimension  $p \times p$ . If  $p > n$ , then only the first  $n$  columns of  $V$  are computed and  $\Sigma$  has dimension  $n \times n$ . The  $n = p$  case is obvious, we retain the full SVD. In general the economy-size SVD leads to less FLOPS and still conserving the desired accuracy.

## Codes for the SVD

```
import numpy as np
# SVD inversion
def SVDinv(A):
    ''' Takes as input a numpy matrix A and returns inv(A) based on singular value decomposition
    SVD is numerically more stable than the inversion algorithms provided by
    numpy and scipy.linalg at the cost of being slower.
    '''
    U, s, VT = np.linalg.svd(A)
    # print('test U')
    # print( (np.transpose(U) @ U - U @ np.transpose(U)))
    # print('test VT')
    # print( (np.transpose(VT) @ VT - VT @ np.transpose(VT)))
    print(U)
    print(s)
    print(VT)

    D = np.zeros((len(U),len(VT)))
    for i in range(0,len(VT)):
        D[i,i]=s[i]
    UT = np.transpose(U); V = np.transpose(VT); invD = np.linalg.inv(D)
    return np.matmul(V,np.matmul(invD,UT))

X = np.array([ [1.0, -1.0, 2.0], [1.0, 0.0, 1.0], [1.0, 2.0, -1.0], [1.0, 1.0, 0.0] ])
print(X)
A = np.transpose(X) @ X
print(A)
# Brute force inversion of super-collinear matrix
B = np.linalg.inv(A)
```

```
print(B)
C = SVDinv(A)
print(C)
```

The matrix  $\mathbf{X}$  has columns that are linearly dependent. The first column is the row-wise sum of the other two columns. The rank of a matrix (the column rank) is the dimension of space spanned by the column vectors. The rank of the matrix is the number of linearly independent columns, in this case just 2. We see this from the singular values when running the above code. Running the standard inversion algorithm for matrix inversion with  $\mathbf{X}^T \mathbf{X}$  results in the program terminating due to a singular matrix.

## Mathematical Properties

There are several interesting mathematical properties which will be relevant when we are going to discuss the differences between say ordinary least squares (OLS) and **Ridge** regression.

We have from OLS that the parameters of the linear approximation are given by

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

The matrix to invert can be rewritten in terms of our SVD decomposition as

$$\mathbf{X}^T \mathbf{X} = \mathbf{V} \boldsymbol{\Sigma}^T \mathbf{U}^T \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^T.$$

Using the orthogonality properties of  $\mathbf{U}$  we have

$$\mathbf{X}^T \mathbf{X} = \mathbf{V} \boldsymbol{\Sigma}^T \boldsymbol{\Sigma} \mathbf{V}^T = \mathbf{V} \mathbf{D} \mathbf{V}^T,$$

with  $\mathbf{D}$  being a diagonal matrix with values along the diagonal given by the singular values squared.

This means that

$$(\mathbf{X}^T \mathbf{X}) \mathbf{V} = \mathbf{V} \mathbf{D},$$

that is the eigenvectors of  $(\mathbf{X}^T \mathbf{X})$  are given by the columns of the right singular matrix of  $\mathbf{X}$  and the eigenvalues are the squared singular values. It is easy to show (show this) that

$$(\mathbf{X} \mathbf{X}^T) \mathbf{U} = \mathbf{U} \mathbf{D},$$

that is, the eigenvectors of  $(\mathbf{X} \mathbf{X}^T)$  are the columns of the left singular matrix and the eigenvalues are the same.

Going back to our OLS equation we have

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{X} (\mathbf{V} \mathbf{D} \mathbf{V}^T)^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^T (\mathbf{V} \mathbf{D} \mathbf{V}^T)^{-1} (\mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^T)^T \mathbf{y} = \mathbf{U} \mathbf{U}^T \mathbf{y}.$$

We will come back to this expression when we discuss Ridge regression.

## Beyond Ordinary Least Squares

### Ridge and LASSO Regression

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

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

or we can state it as

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

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

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

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

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

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

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

we have a new optimization equation

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

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

Here we have defined the norm-1 as

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

## More on Ridge Regression

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

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

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

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

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

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

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

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

with  $t$  a finite positive number.

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

Furthermore, if we use the result above in terms of the SVD decomposition (our analysis was done for the OLS method), we get

$$(\mathbf{X}\mathbf{X}^T)\mathbf{U} = \mathbf{U}\mathbf{D}.$$

We can analyse the OLS solutions in terms of the eigenvectors (the columns) of the right singular value matrix  $\mathbf{U}$  as

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{X}(\mathbf{V}\mathbf{D}\mathbf{V}^T)^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T (\mathbf{V}\mathbf{D}\mathbf{V}^T)^{-1} (\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T \mathbf{y} = \mathbf{U}\mathbf{U}^T \mathbf{y}$$

For Ridge regression this becomes

$$\mathbf{X}\boldsymbol{\beta}^{\text{Ridge}} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T (\mathbf{V}\mathbf{D}\mathbf{V}^T + \lambda \mathbf{I})^{-1} (\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T \mathbf{y} = \sum_{j=0}^{p-1} \mathbf{u}_j \mathbf{u}_j^T \frac{\sigma_j^2}{\sigma_j^2 + \lambda} \mathbf{y},$$

with the vectors  $\mathbf{u}_j$  being the columns of  $\mathbf{U}$ .

## Interpreting the Ridge results

Since  $\lambda \geq 0$ , it means that compared to OLS, we have

$$\frac{\sigma_j^2}{\sigma_j^2 + \lambda} \leq 1.$$

Ridge regression finds the coordinates of  $\mathbf{y}$  with respect to the orthonormal basis  $\mathbf{U}$ , it then shrinks the coordinates by  $\frac{\sigma_j^2}{\sigma_j^2 + \lambda}$ . Recall that the SVD has eigenvalues ordered in a descending way, that is  $\sigma_i \geq \sigma_{i+1}$ .

For small eigenvalues  $\sigma_i$  it means that their contributions become less important, a fact which can be used to reduce the number of degrees of freedom. Actually, calculating the variance of  $\mathbf{X}\mathbf{v}_j$  shows that this quantity is equal to  $\sigma_j^2/n$ . With a parameter  $\lambda$  we can thus shrink the role of specific parameters.

## More interpretations

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

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

In this case the standard OLS results in

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

and

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

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

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

## Statistics

### Where are we going?

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

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

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

## Resampling methods

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

Two resampling methods are often used in Machine Learning analyses,

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

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

## Resampling approaches can be computationally expensive

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

## Why resampling methods ?

Statistical analysis.

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



## Statistical analysis

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

## Linking the regression analysis with a statistical interpretation

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

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

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

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

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

## Assumptions made

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

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

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

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

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

## Expectation value and variance

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

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

while its variance is

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

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

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

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

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

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

We can also calculate the variance

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

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

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

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

It is rather straightforward to show that

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

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

We can also compute the variance as

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

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

With this, we can compute the difference

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

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