Applied Machine Learning and Data Analysis

Morten Hjorth-Jensen

Sep 20, 2020

CONTENTS

Morten Hjorth-Jensen, Department of Physics, University of Oslo and Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Date: **Sep 19, 2020**

CHAPTER ONE

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.

It has become more and more common to see research projects on big data in for example the Social Sciences where extracting patterns from complicated survey data is one of many research directions. Having a solid grasp of data analysis and machine learning is thus becoming central to scientific computing in many fields, and competences and skills within the fields of machine learning and scientific computing are nowadays strongly requested by many potential employers. The latter cannot be overstated, familiarity with machine learning has almost become a prerequisite for many of the most exciting employment opportunities, whether they are in bioinformatics, life science, physics or finance, in the private or the public sector. This author has had several students or met students who have been hired recently based on their skills and competences in scientific computing and data science, often with marginal knowledge of machine learning.

Machine learning is a subfield of computer science, and is closely related to computational statistics. It evolved from the study of pattern recognition in artificial intelligence (AI) research, and has made contributions to AI tasks like computer vision, natural language processing and speech recognition. Many of the methods we will study are also strongly rooted in basic mathematics and physics research.

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 is an extremely rich field, in spite of its young age. The increases we have seen during the last three decades in computational capabilities have been followed by developments of methods and techniques for analyzing and handling large date sets, relying heavily on statistics, computer science and mathematics. The field is rather new and developing rapidly. Popular software packages 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. Not all the algorithms and methods can be given a rigorous mathematical justification, 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 algorithms and methods we will discuss.

CHAPTER TWO

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 plays a central role in these lectures, and our hope is, through the various projects and exercises, to expose you to fundamental research problems in these fields, with the aim to reproduce state of the art scientific results. You will learn to develop and structure codes for studying these systems, get acquainted with computing facilities and learn to handle large scientific projects. A good scientific and ethical conduct is emphasized throughout the course. More specifically, you will

1. Learn about basic data analysis, Bayesian statistics, Monte Carlo methods, 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. Gain knowledge of central aspects of Monte Carlo methods, Markov chains, Gibbs samplers and their possible applications, from numerical integration to simulation of stock markets;
5. Understand methods for regression and classification;
6. Learn about neural network, genetic algorithms and Boltzmann machines;
7. Work on numerical projects to illustrate the theory. The projects play a central role and you are expected to know modern programming languages like Python or C++, in addition to a basic knowledge of linear algebra (typically taught during the first one or two years of undergraduate studies).

There are several topics we will cover here, spanning from statistical data analysis and its basic concepts such as expectation values, variance, covariance, correlation functions and errors, via well-known probability distribution functions like the uniform distribution, the binomial distribution, the Poisson distribution and simple and multivariate normal distributions to central elements of Bayesian statistics and modeling. We will also remind the reader about central elements from linear algebra and standard methods based on linear algebra used to optimize (minimize) functions (the family of gradient descent methods) and the Singular-value decomposition and least square methods for parameterizing data.

We will also cover Monte Carlo methods, Markov chains, well-known algorithms for sampling stochastic events like the Metropolis-Hastings and Gibbs sampling methods. An important aspect of all our calculations is a proper estimation of errors. Here we will also discuss famous resampling techniques like the blocking, the bootstrapping and the jackknife methods and the infamous bias-variance tradeoff.

The second part of the material covers several algorithms used in machine learning.

CHAPTER

THREE

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.

CHAPTER

FOUR

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 date 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.

CHAPTER FIVE

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.

CHAPTER

SIX

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 authours 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.

CHAPTER

SEVEN

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 determin 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.

CHAPTER

EIGHT

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**.

CHAPTER

NINE

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.

CHAPTER TEN

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 $\boldsymbol{X} \in [x_1, x_2, \dots x_N]$.

As we will see repeatedely 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 corelations 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*.

CHAPTER

ELEVEN

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.

CHAPTER

TWELVE

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.

CHAPTER

THIRTEEN

DATA HANDLING, MACHINE LEARNING AND ETHICAL ASPECTS

In most of the cases we will study, we will either generate the data to analyze ourselves (both for supervised learning and unsupervised learning) or we will recur again and again to data present in say **Scikit-Learn** or **Tensorflow**. Many of the examples we end up dealing with are from a privacy and data protection point of view, rather inoccuous and boring results of numerical calculations. However, this does not hinder us from developing a sound ethical attitude to the data we use, how we analyze the data and how we handle the data.

The most immediate and simplest possible ethical aspects deal with our approach to the scientific process. Nowadays, with version control software like Git and various online repositories like Github, Gitlab etc, we can easily make our codes and data sets we have used, freely and easily accessible to a wider community. This helps us almost automagically in making our science reproducible. The large open-source development communities involved in say Scikit-Learn, Tensorflow, PyTorch and Keras, are all excellent examples of this. The codes can be tested and improved upon continuosly, helping thereby our scientific community at large in developing data analysis and machine learning tools. It is much easier today to gain traction and acceptance for making your science reproducible. From a societal stand, this is an important element since many of the developers are employees of large public institutions like universities and research labs. Our fellow taxpayers do deserve to get something back for their bucks.

However, this more mechanical aspect of the ethics of science (in particular the reproducibility of scientific results) is something which is obvious and everybody should do so as part of the dialectics of science. The fact that many scientists are not willing to share their codes or data is detrimental to the scientific discourse.

Before we proceed, we should add a disclaimer. Even though we may dream of computers developing some kind of higher learning capabilities, at the end (even if the artificial intelligence community keeps touting our ears full of fancy futuristic avenues), it is we, yes you reading these lines, who end up constructing and instructing, via various algorithms, the machine learning approaches. Self-driving cars for example, rely on sofisticated programs which take into account all possible situations a car can encounter. In addition, extensive usage of training data from GPS information, maps etc, are typically fed into the software for self-driving cars. Adding to this various sensors and cameras that feed information to the programs, there are zillions of ethical issues which arise from this.

For self-driving cars, where basically many of the standard machine learning algorithms discussed here enter into the codes, at a certain stage we have to make choices. Yes, we , the lads and lasses who wrote a program for a specific brand of a self-driving car. As an example, all carmakers have as their utmost priority the security of the driver and the accompanying passengers. A famous European carmaker, which is one of the leaders in the market of self-driving cars, had **if** statements of the following type: suppose there are two obstacles in front of you and you cannot avoid to collide with one of them. One of the obstacles is a monstertruck while the other one is a kindergarten class trying to cross the road. The self-driving car algo would then opt for the hitting the small folks instead of the monstertruck, since the likelihood of surving a collision with our future citizens, is much higher.

This leads to serious ethical aspects. Why should we opt for such an option? Who decides and who is entitled to make such choices? Keep in mind that many of the algorithms you will encounter in this series of lectures or hear about later, are indeed based on simple programming instructions. And you are very likely to be one of the people who may end up writing such a code. Thus, developing a sound ethical attitude to what we do, an approach well beyond the simple mechanistic one of making our science available and reproducible, is much needed. The example of the self-driving cars is just one of infinitely many cases where we have to make choices. When you analyze data on economic inequalities, who guarantees that you are not weighting some data in a particular way, perhaps because you dearly want a specific conclusion which may support your political views? Or what about the recent claims that a famous IT company like Apple has a sexist bias on the their recently launched credit card?

We do not have the answers here, nor will we venture into a deeper discussions of these aspects, but we want you think over these topics in a more overarching way. A statistical data analysis with its dry numbers and graphs meant to guide the eye, does not necessarily reflect the truth, whatever that is. As a scientist, and after a university education, you are supposedly a better citizen, with an improved critical view and understanding of the scientific method, and perhaps some deeper understanding of the ethics of science at large. Use these insights. Be a critical citizen. You owe it to our society.

13.1 Data Analysis and Machine Learning: Getting started, our first data and Machine Learning encounters

Morten Hjorth-Jensen, Department of Physics, University of Oslo and Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Date: **Dec 25, 2019**

13.1.1 Introduction

Our emphasis throughout this series of lecturesis on understanding the mathematical aspects of different algorithms used in the fields of data analysis and machine learning.

However, where possible we will emphasize the importance of using available software. We start thus with a hands-on and top-down approach to machine learning. The aim is thus to start with relevant data or data we have produced and use these to introduce statistical data analysis concepts and machine learning algorithms before we delve into the algorithms themselves. The examples we will use in the beginning, start with simple polynomials with random noise added. We will use the Python software package Scikit-Learn and introduce various machine learning algorithms to make fits of the data and predictions. We move thereafter to more interesting cases such as data from say experiments (below we will look at experimental nuclear binding energies as an example). These are examples where we can easily set up the data and then use machine learning algorithms included in for example **Scikit-Learn**.

These examples will serve us the purpose of getting started. Furthermore, they allow us to catch more than two birds with a stone. They will allow us to bring in some programming specific topics and tools as well as showing the power of various Python libraries for machine learning and statistical data analysis.

Here, we will mainly focus on two specific Python packages for Machine Learning, Scikit-Learn and Tensorflow (see below for links etc). Moreover, the examples we introduce will serve as inputs to many of our discussions later, as well as allowing you to set up models and produce your own data and get started with programming.

13.1.2 What is Machine Learning?

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.

It has become more and more common to see research projects on big data in for example the Social Sciences where extracting patterns from complicated survey data is one of many research directions. Having a solid grasp of data analysis and machine learning is thus becoming central to scientific computing in many fields, and competences and skills within the fields of machine learning and scientific computing are nowadays strongly requested by many potential employers. The latter cannot be overstated, familiarity with machine learning has almost become a prerequisite for many of the most exciting employment opportunities, whether they are in bioinformatics, life science, physics or finance, in the private or the public sector. This author has had several students or met students who have been hired recently based on their skills and competences in scientific computing and data science, often with marginal knowledge of machine learning.

Machine learning is a subfield of computer science, and is closely related to computational statistics. It evolved from the study of pattern recognition in artificial intelligence (AI) research, and has made contributions to AI tasks like computer vision, natural language processing and speech recognition. Many of the methods we will study are also strongly rooted in basic mathematics and physics research.

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 is an extremely rich field, in spite of its young age. The increases we have seen during the last three decades in computational capabilities have been followed by developments of methods and techniques for analyzing and handling large date sets, relying heavily on statistics, computer science and mathematics. The field is rather new and developing rapidly. Popular software packages 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. Not all the algorithms and methods can be given a rigorous mathematical justification, 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 algorithms and methods we will discuss.

13.1.3 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 authours 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 typically 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.

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 and test data), 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 determin our model.

The last ingredient is a so-called **cost** function which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.At the heart of basically all ML algorithms there are so-called minimization algorithms, often we end up with various variants of **gradient** methods.

13.1.4 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

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

For Python3, replace **pip** with **pip3**.

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 python3

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 python3 (or python for pyhton2.7)

etc etc.

13.1.5 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 distrubutions 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!

13.1.6 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 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

13.1.7 Installing R, C++, cython or Julia

You will also find it convenient to utilize **R**. We will mainly use Python during our lectures and in various projects and exercises. Those of you already familiar with **R** should feel free to continue using **R**, keeping however an eye on the parallel Python set ups. Similarly, if you are a Python afecionado, feel free to explore **R** as well. Jupyter/Ipython notebook allows you to run **R** codes interactively in your browser. The software library **R** is really tailored for statistical data analysis and allows for an easy usage of the tools and algorithms we will discuss in these lectures.

To install **R** with Jupyter notebook follow the link here

13.1.8 Installing R, C++, cython, Numba etc

For the C++ aficionados, Jupyter/IPython notebook allows you also to install C++ and run codes written in this language interactively in the browser. Since we will emphasize writing many of the algorithms yourself, you can thus opt for either Python or C++ (or Fortran or other compiled languages) as programming languages.

To add more entropy, **cython** can also be used when running your notebooks. It means that Python with the jupyter notebook setup allows you to integrate widely popular softwares and tools for scientific computing. Similarly, the Numba Python package delivers increased performance capabilities with minimal rewrites of your codes. With its versatility, including symbolic operations, Python offers a unique computational environment. Your jupyter notebook can easily be converted into a nicely rendered **PDF** file or a Latex file for further processing. For example, convert to latex as

```
pycod jupyter nbconvert filename.ipynb --to latex
```

And to add more versatility, the Python package SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) and is entirely written in Python.

Finally, if you wish to use the light mark-up language doconce you can convert a standard ascii text file into various HTML formats, ipython notebooks, latex files, pdf files etc with minimal edits. These lectures were generated using **doconce**.

13.1.9 Numpy examples and Important Matrix and vector handling packages

There are several central software libraries for linear algebra and eigenvalue problems. Several of the more popular ones have been wrapped into ofter software packages like those from the widely used text **Numerical Recipes**. The original source codes in many of the available packages are often taken from the widely used software package LAPACK, which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. We describe them shortly here.

- LINPACK: package for linear equations and least square problems.
- LAPACK:package for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website http://www.netlib.org it is possible to download for free all source codes from this library. Both C/C++ and Fortran versions are available.
- BLAS (I, II and III): (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Blas I is vector operations, II vector-matrix operations and III matrix-matrix operations. Highly parallelized and efficient codes, all available for download from http://www.netlib.org.

13.1.10 Basic Matrix Features

Matrix properties reminder.

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

The inverse of a matrix is defined by

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

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 $\mathbf{A}\mathbf{x} = 0$ 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 $\mathbf{A}$.

13.1.11 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)
print(x[1])
```

```
[ 0.66109153  0.97132954 -0.91470448  1.81588452  0.49665195  0.77698678
  1.04576423  1.65832096 -0.25613699 -0.7554783 ]
0.971329541323762
```

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)
```

```
[1 2 3]
```

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)
```

```
[1.38629436 1.94591015 2.07944154]
```

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 normaly recommend that you use the Numpy intrinsic functions instead of the corresponding **log** function from Python's **math** module. The looping is done explicitely 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)
```

```
[1 1 2]
```

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 automagically 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)
```

```
[1.38629436 1.94591015 2.07944154]
```

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)
```

```
[1.38629436 1.94591015 2.07944154]
```

To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can use simple 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.itemsize)
```

```
8
```

13.1.12 Matrices in Python

Having defined vectors, we are now ready to try out matrices. We can define a 3×3 real matrix $\hat{A}$ as (recall that we user 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)
```

```
[[1.38629436 1.94591015 2.07944154]
 [1.09861229 2.30258509 2.39789527]
 [1.38629436 1.60943791 1.94591015]]
```

If we use the **shape** function we would get $(3, 3)$ as output, that is verifying that our matrix is a 3×3 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])
```

```
[1.38629436 1.09861229 1.38629436]
```

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,:])
```

```
[1.09861229 2.30258509 2.39789527]
```

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)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

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.eye( n )
print(A)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

(continues on next page)

(continued from previous page)

```
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

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)
```

```
[[0.88645063 0.21176899 0.23897722 0.06720011 0.38272048 0.88809424
  0.0878077  0.73425375 0.72093518 0.19764225]
 [0.74839206 0.31545538 0.14076925 0.59046117 0.89437867 0.81232389
  0.20782531 0.52852411 0.10079113 0.07130568]
 [0.64573187 0.71051155 0.65626927 0.92475512 0.58540859 0.52327428
  0.28610035 0.69580848 0.94590985 0.98668663]
 [0.01101253 0.38937429 0.60430671 0.81789875 0.2280742  0.17786312
  0.34053316 0.19374203 0.01520576 0.2466458 ]
 [0.60613717 0.19314313 0.959787   0.6376673  0.73798553 0.63486051
  0.15742577 0.39450088 0.07204545 0.46013967]
 [0.23509059 0.79672774 0.81910307 0.53487394 0.50992632 0.112086
  0.55751711 0.10662789 0.70911163 0.59072491]
 [0.0259775  0.26815865 0.5090044  0.41809535 0.32232515 0.72743704
  0.32110496 0.04202295 0.18265714 0.70084503]
 [0.07736581 0.57300172 0.97099592 0.62357997 0.36579921 0.37062279
  0.74683978 0.59433006 0.35540585 0.97384662]
 [0.25205634 0.7601454  0.40639052 0.24504215 0.21745329 0.95788719
  0.7490677  0.29789413 0.81839129 0.93822723]
 [0.6302074  0.47072129 0.16215217 0.53540566 0.91760002 0.14146861
  0.37860338 0.8099108  0.07897476 0.69254202]]
```

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 $\hat{x}, \hat{y}, \hat{z}$ with n elements each. The covariance matrix is defined as

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

where for example

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

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 $\hat{W}$

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

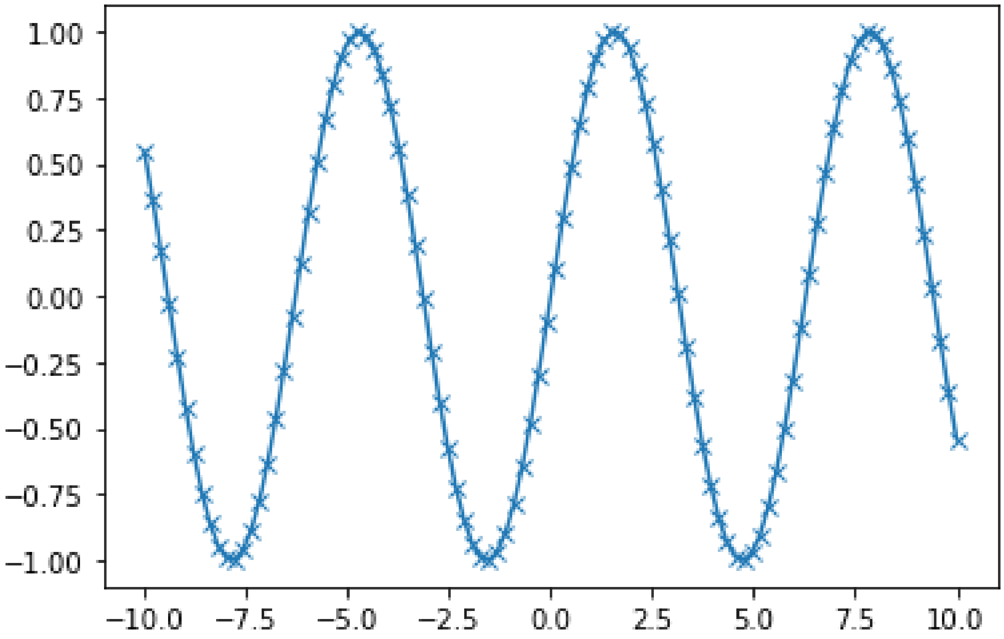
which in turn is converted into into the 3×3 covariance matrix $\hat{\Sigma}$ via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples $\hat{x}$ etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix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

```python
# 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)
```

```
-0.13489611340279922
3.7349937337703607
-0.6150223388222948
[[ 1.15822308  3.44928776  3.99889831]
 [ 3.44928776 11.11151113 12.25819175]
 [ 3.99889831 12.25819175 21.09855273]]
[30.28557034  0.07629455  3.00642205]
```

```python
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
eye = np.eye(4)
print(eye)
sparse_mtx = sparse.csr_matrix(eye)
print(sparse_mtx)
x = np.linspace(-10,10,100)
y = np.sin(x)
plt.plot(x,y,marker='x')
plt.show()
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
  (0, 0)	1.0
  (1, 1)	1.0
  (2, 2)	1.0
  (3, 3)	1.0
```



13.1.13 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 reshapings 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)
```

```
   First Name Last Name Place of birth  Date of Birth T.A.
0       Frodo   Baggins          Shire                2968
1       Bilbo   Baggins          Shire                2890
2  Aragorn II   Elessar        Eriador                2931
3     Samwise    Gamgee          Shire                2980
```

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)
```

```
         First Name Last Name Place of birth  Date of Birth T.A.
Frodo         Frodo   Baggins          Shire                2968
Bilbo         Bilbo   Baggins          Shire                2890
Aragorn  Aragorn II   Elessar        Eriador                2931
Sam         Samwise    Gamgee          Shire                2980
```

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

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

```
First Name            Aragorn II
Last Name                Elessar
Place of birth           Eriador
Date of Birth T.A.          2931
Name: Aragorn, dtype: object
```

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)
```

```
         First Name Last Name Place of birth  Date of Birth T.A.
Frodo         Frodo   Baggins          Shire                2968
Bilbo         Bilbo   Baggins          Shire                2890
Aragorn  Aragorn II   Elessar        Eriador                2931
Sam         Samwise    Gamgee          Shire                2980
Pippin     Peregrin      Took          Shire                2990
```

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 10×5 and compute the mean value and standard deviation of each column. Similarly, we can perform mathematial 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())
```

(continues on next page)

(continued from previous page)

```
display(df**2)
print(df-df.mean())
```

```
          0         1         2         3         4
0 -1.749765  0.342680  1.153036 -0.252436  0.981321
1  0.514219  0.221180 -1.070043 -0.189496  0.255001
2 -0.458027  0.435163 -0.583595  0.816847  0.672721
3 -0.104411 -0.531280  1.029733 -0.438136 -1.118318
4  1.618982  1.541605 -0.251879 -0.842436  0.184519
5  0.937082  0.731000  1.361556 -0.326238  0.055676
6  0.222400 -1.443217 -0.756352  0.816454  0.750445
7 -0.455947  1.189622 -1.690617 -1.356399 -1.232435
8 -0.544439 -0.668172  0.007315 -0.612939  1.299748
9 -1.733096 -0.983310  0.357508 -1.613579  1.470714
```

```
0   -0.175300
1    0.083527
2   -0.044334
3   -0.399836
4    0.331939
dtype: float64
0    1.069584
1    0.965548
2    1.018232
3    0.793167
4    0.918992
dtype: float64
```

```
          0         1         2         3         4
0  3.061679  0.117430  1.329492  0.063724  0.962990
1  0.264421  0.048920  1.144993  0.035909  0.065026
2  0.209789  0.189367  0.340583  0.667239  0.452553
3  0.010902  0.282259  1.060349  0.191963  1.250636
4  2.621102  2.376547  0.063443  0.709698  0.034047
5  0.878123  0.534362  1.853835  0.106431  0.003100
6  0.049462  2.082875  0.572069  0.666597  0.563167
7  0.207888  1.415201  2.858185  1.839818  1.518895
8  0.296414  0.446453  0.000054  0.375694  1.689345
9  3.003620  0.966899  0.127812  2.603636  2.162999
```

```
          0         1         2         3         4
0 -1.574465  0.259153  1.197370  0.147400  0.649382
1  0.689519  0.137652 -1.025709  0.210340 -0.076938
2 -0.282727  0.351636 -0.539261  1.216683  0.340782
3  0.070889 -0.614808  1.074067 -0.038300 -1.450257
4  1.794282  1.458078 -0.207545 -0.442600 -0.147420
5  1.112383  0.647473  1.405890  0.073598 -0.276263
6  0.397700 -1.526744 -0.712018  1.216290  0.418506
7 -0.280647  1.106095 -1.646283 -0.956563 -1.564374
8 -0.369139 -0.751699  0.051649 -0.213103  0.967809
9 -1.557795 -1.066837  0.401842 -1.213743  1.138775
```

Thereafter we can select specific columns only and plot final results

```
df.columns = ['First', 'Second', 'Third', 'Fourth', 'Fifth']
```

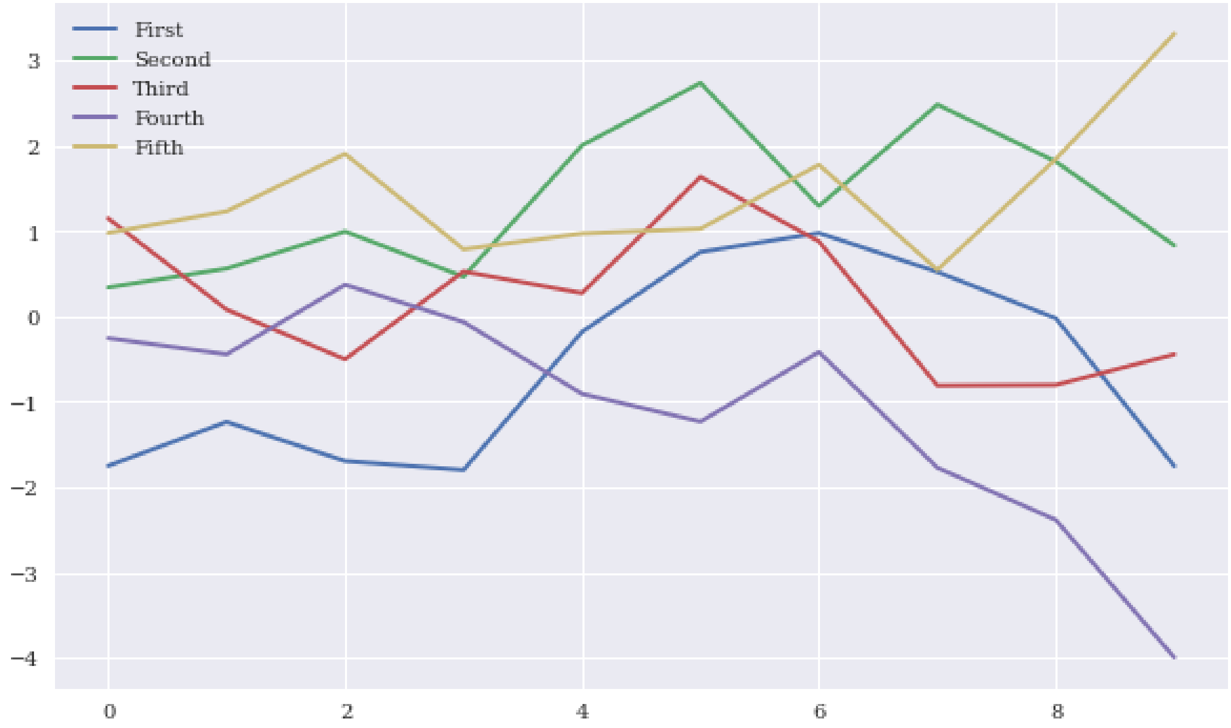
(continues on next page)

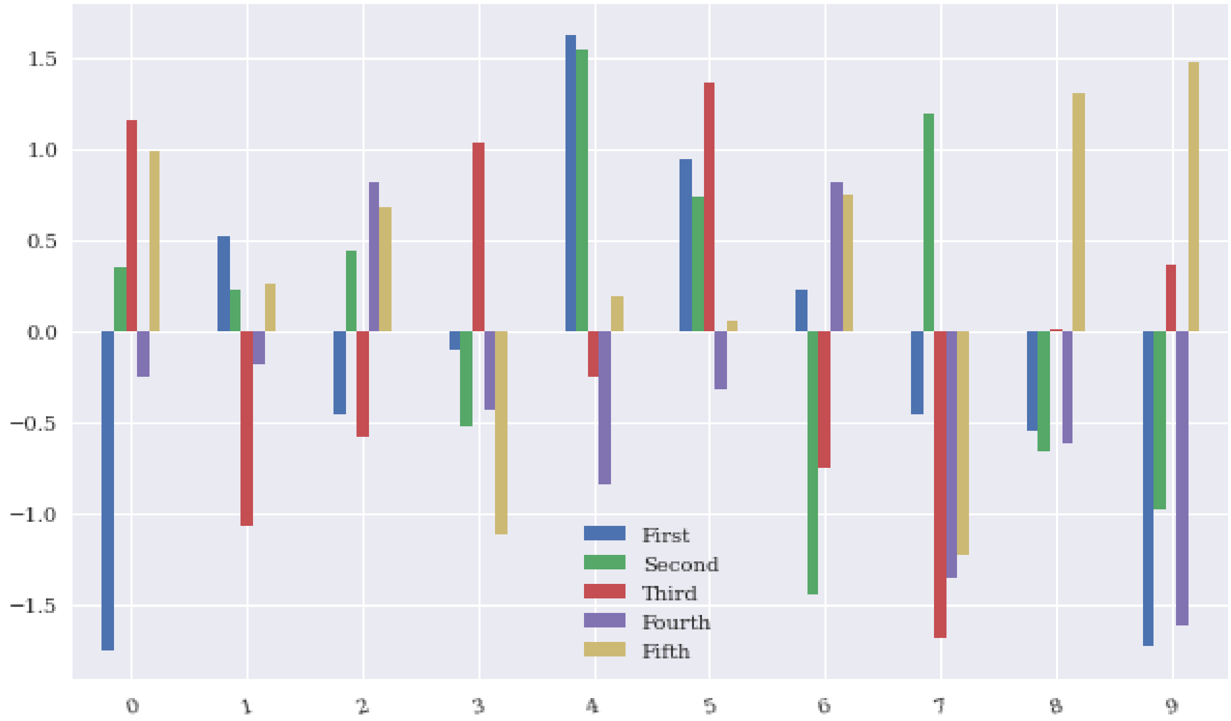
(continued from previous page)

```
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()
```

```
      First    Second     Third    Fourth     Fifth
0 -1.749765  0.342680  1.153036 -0.252436  0.981321
1  0.514219  0.221180 -1.070043 -0.189496  0.255001
2 -0.458027  0.435163 -0.583595  0.816847  0.672721
3 -0.104411 -0.531280  1.029733 -0.438136 -1.118318
4  1.618982  1.541605 -0.251879 -0.842436  0.184519
5  0.937082  0.731000  1.361556 -0.326238  0.055676
6  0.222400 -1.443217 -0.756352  0.816454  0.750445
7 -0.455947  1.189622 -1.690617 -1.356399 -1.232435
8 -0.544439 -0.668172  0.007315 -0.612939  1.299748
9 -1.733096 -0.983310  0.357508 -1.613579  1.470714
```

```
0.08352721390288316
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 0 to 9
Data columns (total 5 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   First    10 non-null     float64
 1   Second   10 non-null     float64
 2   Third    10 non-null     float64
 3   Fourth   10 non-null     float64
 4   Fifth    10 non-null     float64
dtypes: float64(5)
memory usage: 480.0 bytes
None
           First     Second      Third     Fourth      Fifth
count  10.000000  10.000000  10.000000  10.000000  10.000000
mean   -0.175300   0.083527  -0.044334  -0.399836   0.331939
std     1.069584   0.965548   1.018232   0.793167   0.918992
min    -1.749765  -1.443217  -1.690617  -1.613579  -1.232435
25%    -0.522836  -0.633949  -0.713163  -0.785061   0.087887
50%    -0.280179   0.281930  -0.122282  -0.382187   0.463861
75%     0.441264   0.657041   0.861676  -0.205231   0.923602
max     1.618982   1.541605   1.361556   0.816847   1.470714
```





We can produce a 4×4 matrix

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

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
```

and many other operations.

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 concice code close to the mathematical operations we may be interested in. For multidimensional arrays, we recommend strongly 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**.

13.1.14 Reading Data and fitting

In order to study various Machine Learning algorithms, we need to access data. Acccessing 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 mainly 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 $\hat{x}$ are given by random numbers generated with a uniform distribution with entries $x_i \in [0, 1]$ (more about probability distribution functions later). These values are then used to define a function $y(x)$ (tabulated again as a vector) with a linear dependence on x plus a random noise added via the normal distribution.

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

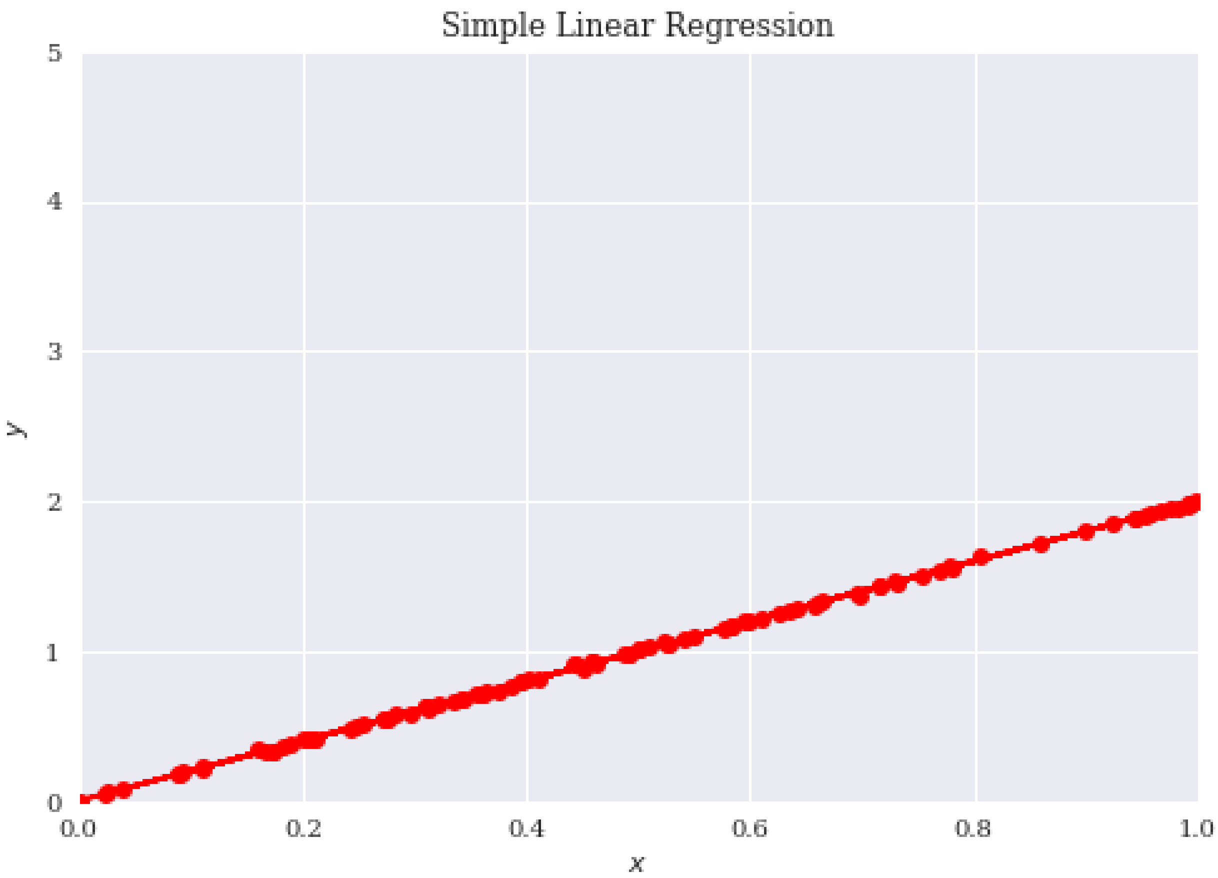
$$y = 2x + N(0,1),$$

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

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

The Python code follows here.

```python
# 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+0.01*np.random.randn(100,1)
linreg = LinearRegression()
linreg.fit(x,y)
#ynew = linreg.predict(x)
#xnew = np.array([[0],[1]])
ypredict = linreg.predict(x)

plt.plot(x, 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()
```



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 obviouly not satisfactory in the long run. Here we have only defined the training data and our model, and have not discussed a more rigorous approach to the **cost** function.

We need more rigorous criteria in defining whether we have succeeded or not in modeling our training data. You will be surprised to see that many scientists seldomly venture beyond this 'by the eye' approach. A standard approach for the *cost* function is the so-called χ^2 function (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 σ_i^2 is the variance (to be defined later) of the entry y_i. We may not know the explicit value of σ_i^2, it serves however the aim of scaling the equations and make the cost function dimensionless.

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

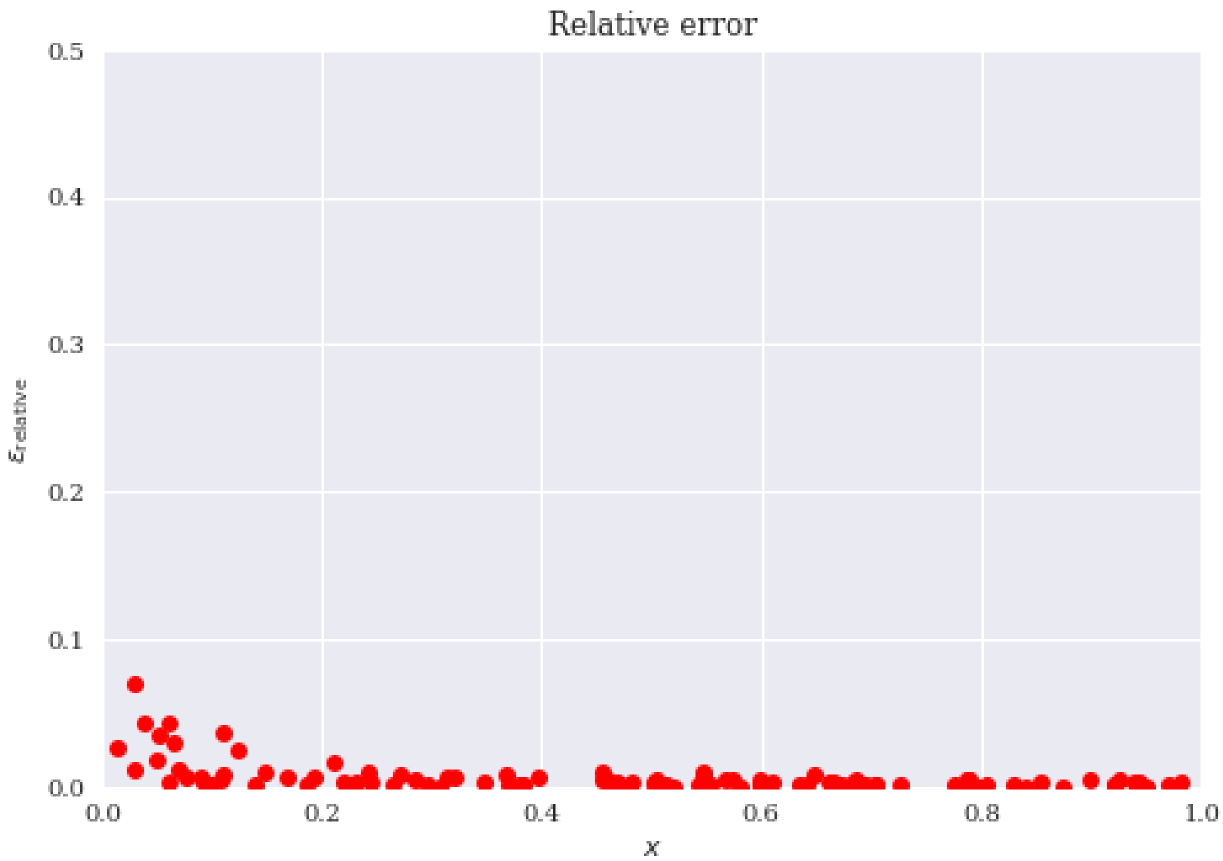
There are many ways to define the cost function. A simpler approach is to look at the relative difference between the training data and the predicted data, that is we define the relative error (why would we prefer the MSE instead of the relative error?) as

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

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.

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

Here we show an example of the functionality of **Scikit-Learn**.

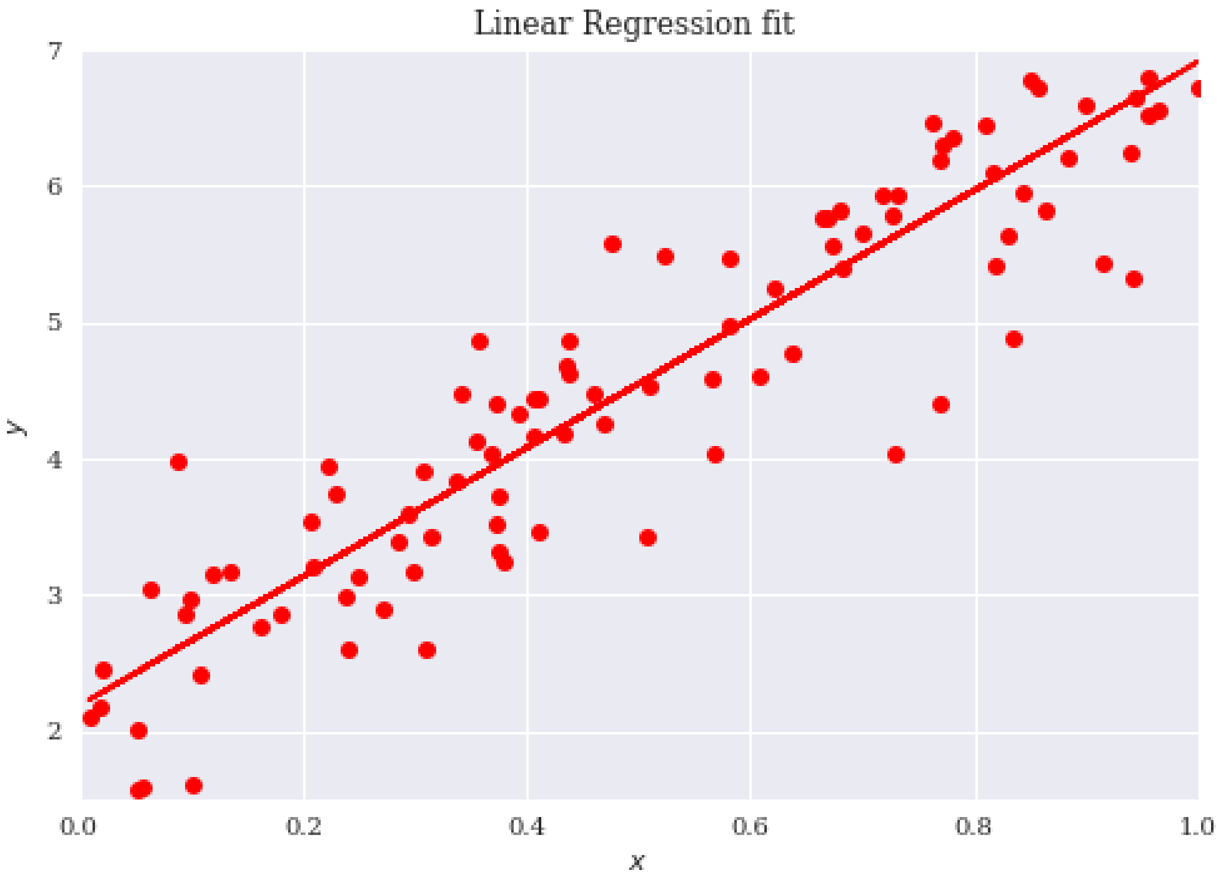
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_squared_log_error, mean_absolute_error

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
```

(continues on next page)

(continued from previous page)

```
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 intercept alpha:
 [2.18780801]
Coefficient beta :
 [[4.72228205]]
Mean squared error: 0.37
Variance score: 0.83
Mean squared log error: 0.01
Mean absolute error: 0.47
```



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

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

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

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

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

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

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

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

Another quantity taht 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 $l1$-norm loss. In our discussion above we presented the relative error. The MAE is defined as follows

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

We present the squared logarithmic (quadratic) error

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

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

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

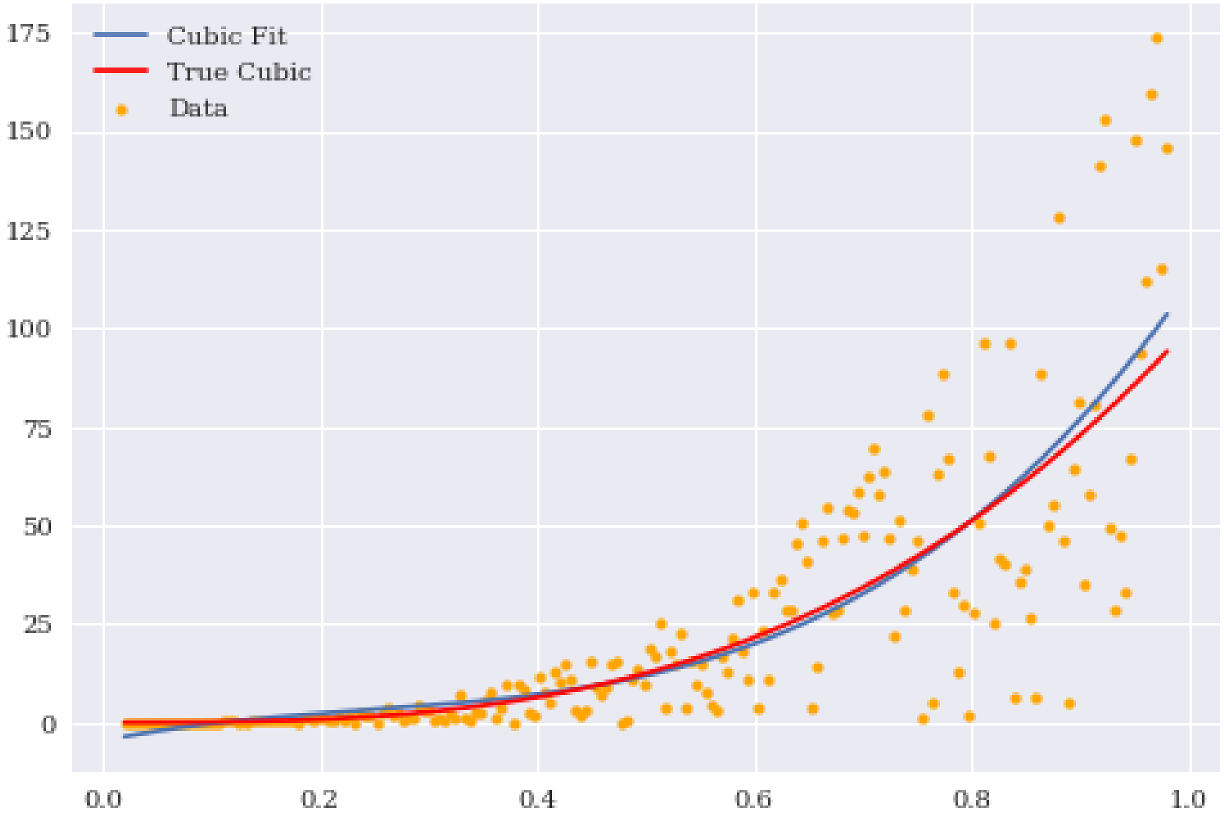
. Here $a=\boldsymbol{y} - \boldsymbol{\tilde{y}}$. We will discuss in more detail these and other functions in the various lectures. We conclude this part with another example. Instead of a linear x-dependence we study now a cubic polynomial and use the polynomial regression analysis tools of scikit-learn.

```
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
```

(continues on next page)

(continued from previous page)

```
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))
```



```
0.005000000000000007
```

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

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

The nucleon masses are

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

and

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

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

$$BE(N, Z) = ZM_H c^2 + Nm_n c^2 - M(N, Z)c^2,$$

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

$$BE(N, Z) = Z\Delta_H c^2 + N\Delta_n c^2 - \Delta(N, Z)c^2,$$

where $\Delta_H c^2 = 7.2890$ MeV and $\Delta_n c^2 = 8.0713$ 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

$$BE(N, Z) = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(N - Z)^2}{A},$$

where A stands for the number of nucleons and the a_is 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 $a_1 A$ 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**.

```python
# 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')
```

Before we proceed, we define also a function for making our plots. You can obviously avoid this and simply set up various **matplotlib** commands every time you need them. You may however find it convenient to collect all such commands in one function and simply call this function.

```
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)
```

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,f11.5
   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.     ␣
↪
"""
```

```
'                                                                      ␣
↪                                     \nThis is taken from the data file of the mass␣
↪2016 evaluation.                                                  \
↪nAll files are 3436 lines long with 124 character per line.         ␣
↪                                        \n        Headers are 39 lines long.  ␣
↪                                                                    \
↪n   col 1      :  Fortran character control: 1 = page feed   0 = line feed   ␣
↪                                        \n    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,f11.5      ␣
↪\n   These formats are reflected in the pandas widths variable below, see the␣
↪statement                                       \n   widths=(1,3,5,5,5,1,3,4,1,13,
↪11,11,9,1,2,11,9,1,3,1,12,11,1),                                     ␣
↪        \n   Pandas has also a variable header, with length 39 in this case.␣
↪                                                     \n'
```

(continues on next page)

(continued from previous page)

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 covert 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)
```

```
            N    Z    A Element  Ebinding
A
1   0       0    1    1       H  0.000000
2   1       1    1    2       H  1.112283
3   2       2    1    3       H  2.827265
4   6       2    2    4      He  7.073915
5   9       3    2    5      He  5.512132
...       ...  ...  ...     ...       ...
264 3304  156  108  264      Hs  7.298375
265 3310  157  108  265      Hs  7.296247
266 3317  158  108  266      Hs  7.298273
269 3338  159  110  269      Ds  7.250154
270 3344  160  110  270      Ds  7.253775

[267 rows x 5 columns]
```

The next step, and we will define this mathematically later, is to set up the so-called **design matrix**. We will throughout call this matrix $\boldsymbol{X}$. It has dimensionality $p \times n$, where n is the number of data points and p are the so-called 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)
```

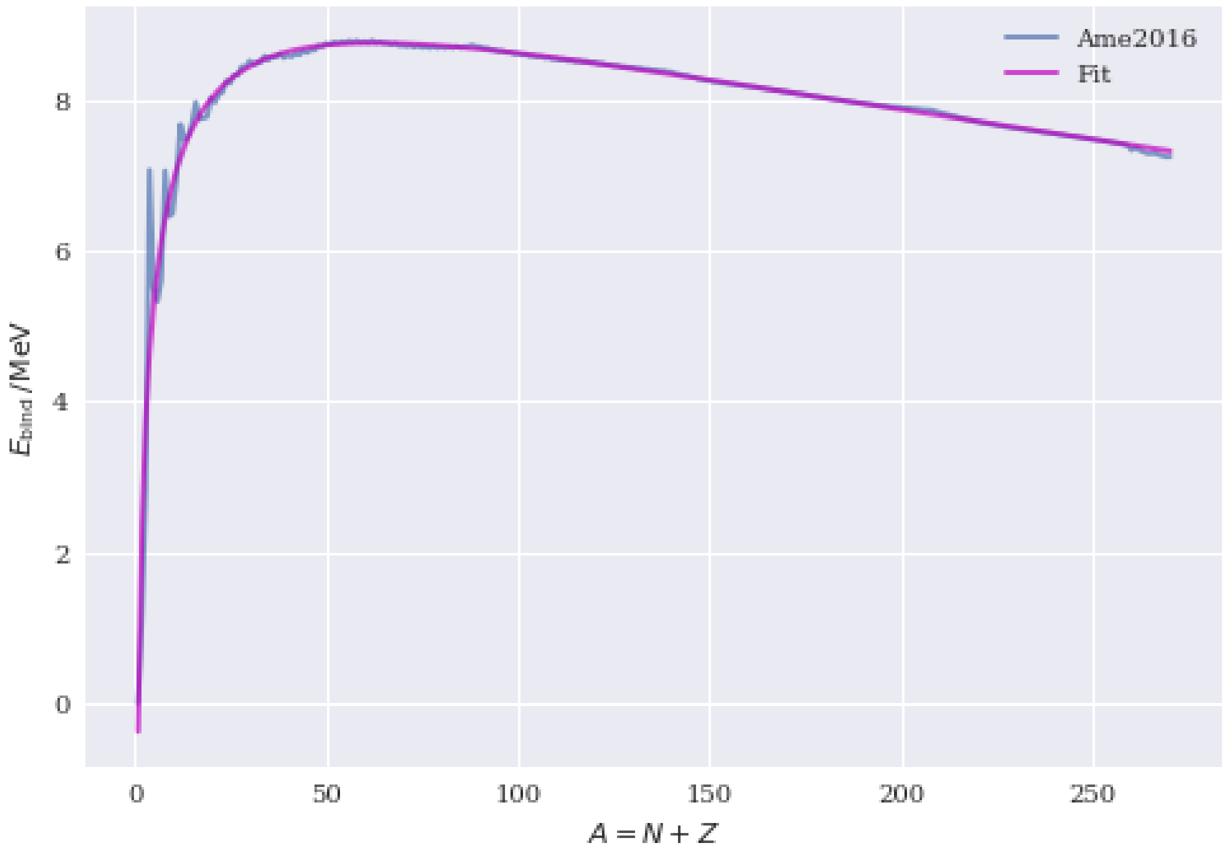
With **scikitlearn** we are now ready to use linear regression and fit our data.

```
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'$E_\mathrm{bind}\,/\mathrm{MeV}$')
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()
```

```
Mean squared error: 0.04
Variance score: 0.95
Mean absolute error: 0.05
[ 0.00000000e+00  7.06492086e-03 -1.73091052e-01 -1.66020213e+01
  1.17385778e+00] 15.212327334149508
```



Seeing the wood for the trees

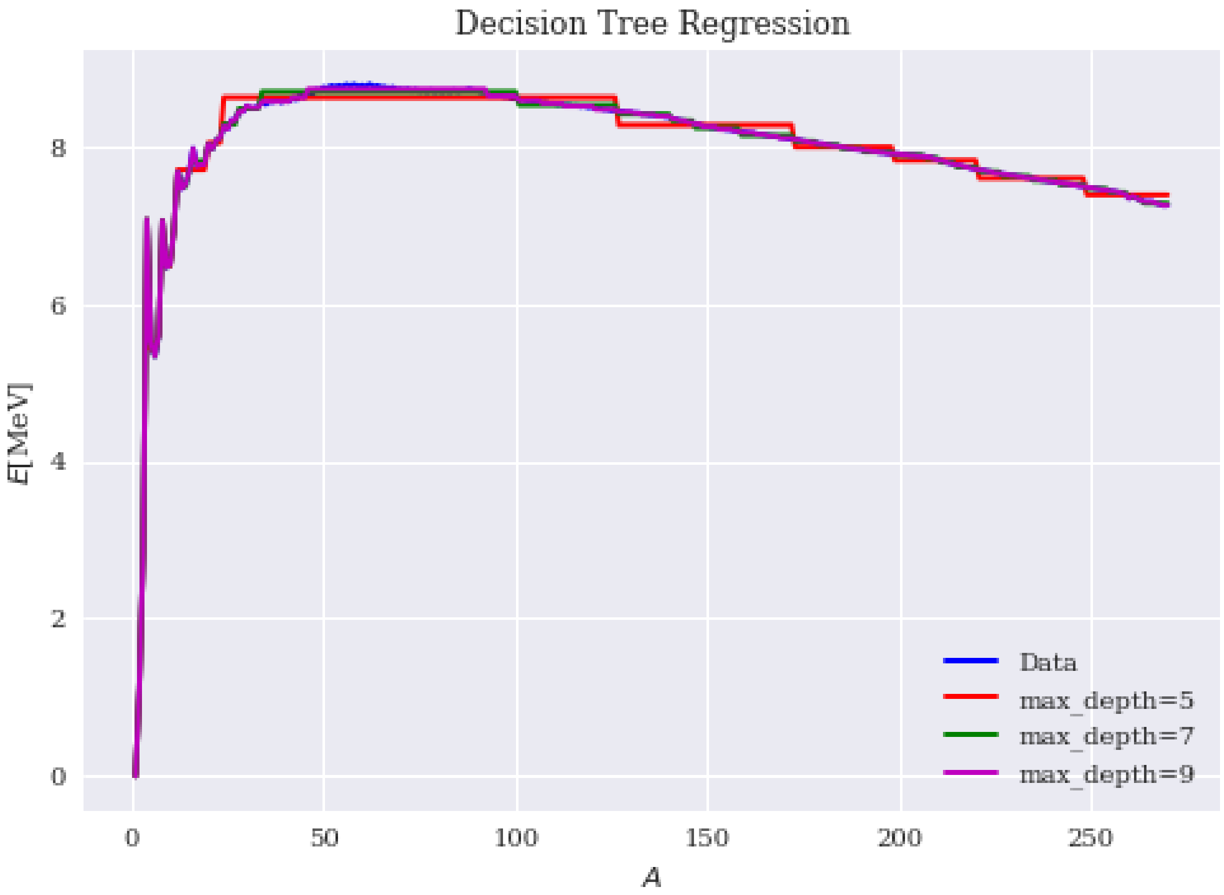
As a teaser, let us now see how we can do this with decision trees using **scikit-learn**. Later we will switch to so-called **random forests**!

```
#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)
```

(continues on next page)

(continued from previous page)

```
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))
```



```
            N    Z    A Element  Ebinding   Eapprox
A
1   0       0    1    1       H  0.000000  0.000000
2   1       1    1    2       H  1.112283  1.112283
3   2       2    1    3       H  2.827265  2.827265
4   6       2    2    4      He  7.073915  7.073915
5   9       3    2    5      He  5.512132  5.512132
...       ...  ...  ...     ...       ...       ...
264 3304  156  108  264      Hs  7.298375  7.298375
265 3310  157  108  265      Hs  7.296247  7.297260
266 3317  158  108  266      Hs  7.298273  7.297260
269 3338  159  110  269      Ds  7.250154  7.250154
270 3344  160  110  270      Ds  7.253775  7.253775

[267 rows x 6 columns]
0.00988361564671618
```

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()
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

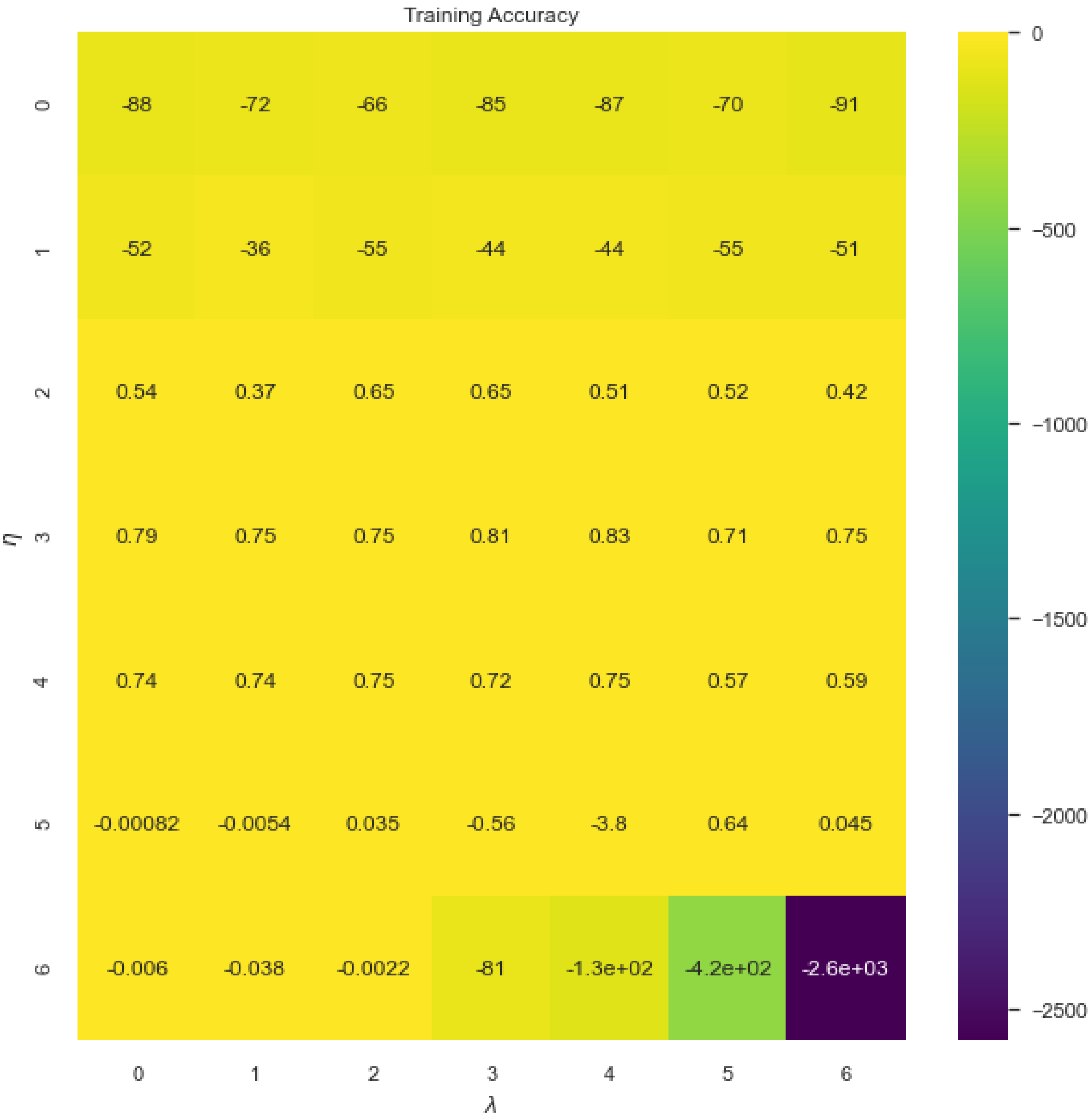
```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:582: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```



13.1.15 A first summary

The aim behind these introductory words was to present to you various Python libraries and their functionalities, in particular libraries like **numpy**, **pandas**, **xarray** and **matplotlib** and other that make our life much easier in handling various data sets and visualizing data.

Furthermore, **Scikit-Learn** allows us with few lines of code to implement popular Machine Learning algorithms for supervised learning. Later we will meet **Tensorflow**, a powerful library for deep learning. Now it is time to dive more into the details of various methods. We will start with linear regression and try to take a deeper look at what it entails.

13.2 Data Analysis and Machine Learning: Elements of Probability Theory and Statistical Data Analysis

Morten Hjorth-Jensen, Department of Physics, University of Oslo and Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Date: **Sep 20, 2020**

13.2.1 To do list

- add math about MVN and define MLE and other quantities
- rewrite about covariance matrix
- add KL theorem

13.2.2 Domains and probabilities

Consider the following simple example, namely the tossing of two dice, resulting in the following possible values

$$\{2,3,4,5,6,7,8,9,10,11,12\}.$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$\{1/36,2/36/,3/36,4/36,5/36,6/36,5/36,4/36,3/36,2/36,1/36\}.$$

13.2.3 Tossing the dice

The numbers in the domain are the outcomes of the physical process of tossing say two dice. We cannot tell beforehand whether the outcome is 3 or 5 or any other number in this domain. This defines the randomness of the outcome, or unexpectedness or any other synonimous word which encompasses the uncertitude of the final outcome.

The only thing we can tell beforehand is that say the outcome 2 has a certain probability.If our favorite hobby is to spend an hour every evening throwing dice and registering the sequence of outcomes, we will note that the numbers in the above domain

$$\{2,3,4,5,6,7,8,9,10,11,12\},$$

appear in a random order. After 11 throws the results may look like

$$\{10,8,6,3,6,9,11,8,12,4,5\}.$$

13.2.4 Stochastic variables

Random variables are characterized by a domain which contains all possible values that the random value may take. This domain has a corresponding probability distribution function(PDF).

13.2.5 Stochastic variables and the main concepts, the discrete case

There are two main concepts associated with a stochastic variable. The *domain* is the set $\mathbb{D} = \{x\}$ of all accessible values the variable can assume, so that $X \in \mathbb{D}$. An example of a discrete domain is the set of six different numbers that we may get by throwing of a dice, $x \in \{1,\, 2,\, 3,\, 4,\, 5,\, 6\}$.

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

$$p(x) = \mathrm{Prob}(X = x).$$

13.2.6 Stochastic variables and the main concepts, the continuous case

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

$$\mathrm{Prob}(a \leq X \leq b) = \int_a^b p(x)dx.$$

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.

13.2.7 The cumulative probability

Of interest to us is the *cumulative probability distribution function* (**CDF**), $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x

$$P(x) = \mathrm{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'.$$

The relation between a CDF and its corresponding PDF is then

$$p(x) = \frac{d}{dx}P(x).$$

13.2.8 Properties of PDFs

There are two properties that all PDFs must satisfy. The first one is positivity (assuming that the PDF is normalized)

$$0 \leq p(x) \leq 1.$$

Naturally, it would be nonsensical for any of the values of the domain to occur with a probability greater than 1 or less than 0. Also, the PDF must be normalized. That is, all the probabilities must add up to unity. The probability of "anything" to happen is always unity. For both discrete and continuous PDFs, this condition is

$$\sum_{x_i\in\mathbb{D}} p(x_i) = 1,$$

$$\int_{x\in\mathbb{D}} p(x)\,dx = 1.$$

13.2.9 Important distributions, the uniform distribution

The first one is the most basic PDF; namely the uniform distribution

$$p(x) = \frac{1}{b-a}\theta(x-a)\theta(b-x).(13.1)$$

For $a = 0$ and $b = 1$ we have

$$p(x)dx = dx \quad \in [0,1].$$

The latter distribution is used to generate random numbers. For other PDFs, one needs normally a mapping from this distribution to say for example the exponential distribution.

13.2.10 Gaussian distribution

The second one is the Gaussian Distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp{(-\frac{(x-\mu)^2}{2\sigma^2})},$$

with mean value μ and standard deviation σ. If $\mu = 0$ and $\sigma = 1$, it is normally called the **standard normal distribution**

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp{(-\frac{x^2}{2})},$$

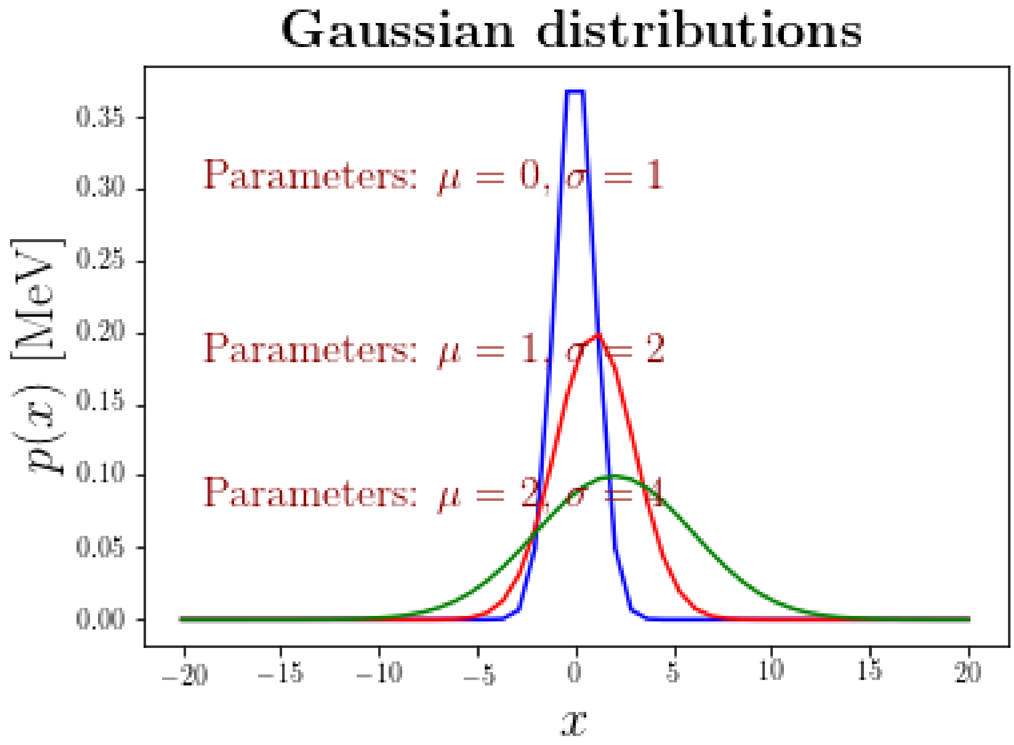
The following simple Python code plots the above distribution for different values of μ and σ.

```
%matplotlib inline

import numpy as np
from math import acos, exp, sqrt
from  matplotlib import pyplot as plt
from matplotlib import rc, rcParams
import matplotlib.units as units
import matplotlib.ticker as ticker
rc('text',usetex=True)
rc('font',**{'family':'serif','serif':['Gaussian distribution']})
font = {'family' : 'serif',
        'color'  : 'darkred',
        'weight' : 'normal',
        'size'   : 16,
        }
pi = acos(-1.0)
mu0 = 0.0
sigma0 = 1.0
mu1= 1.0
sigma1 = 2.0
mu2 = 2.0
sigma2 = 4.0
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-20.0, 20.0)
v0 = np.exp(-(x*x-2*x*mu0+mu0*mu0)/(2*sigma0*sigma0))/sqrt(2*pi*sigma0*sigma0)
v1 = np.exp(-(x*x-2*x*mu1+mu1*mu1)/(2*sigma1*sigma1))/sqrt(2*pi*sigma1*sigma1)
v2 = np.exp(-(x*x-2*x*mu2+mu2*mu2)/(2*sigma2*sigma2))/sqrt(2*pi*sigma2*sigma2)
plt.plot(x, v0, 'b-', x, v1, 'r-', x, v2, 'g-')
plt.title(r'{\bf Gaussian distributions}', fontsize=20)
plt.text(-19, 0.3, r'Parameters: $\mu = 0$, $\sigma = 1$', fontdict=font)
plt.text(-19, 0.18, r'Parameters: $\mu = 1$, $\sigma = 2$', fontdict=font)
plt.text(-19, 0.08, r'Parameters: $\mu = 2$, $\sigma = 4$', fontdict=font)
plt.xlabel(r'$x$',fontsize=20)
plt.ylabel(r'$p(x)$ [MeV]',fontsize=20)

# Tweak spacing to prevent clipping of ylabel                                         ␣
↪
plt.subplots_adjust(left=0.15)
plt.savefig('gaussian.pdf', format='pdf')
plt.show()
```



13.2.11 Exponential distribution

Another important distribution in science is the exponential distribution

$$p(x) = \alpha \exp -(\alpha x).$$

13.2.12 Expectation values

Let $h(x)$ be an arbitrary continuous function on the domain of the stochastic variable X whose PDF is $p(x)$. We define the *expectation value* of h with respect to p as follows

$\langle h\rangle_X \equiv \int h(x)p(x)\, dx(13.2)$

Whenever the PDF is known implicitly, like in this case, we will drop the index X for clarity.A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows

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

13.2.13 Stochastic variables and the main concepts, mean values

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

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

for a continuous distribution and

$$\langle x\rangle = \mu \equiv \sum_{i=1}^{N} x_i p(x_i),$$

for a discrete distribution. Qualitatively it represents the centroid or the average value of the PDF and is therefore simply called the expectation value of $p(x)$.

13.2.14 Stochastic variables and the main concepts, central moments, the variance

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

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

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ^2_X or

$\mathrm{Var}(X)$

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

The square root of the variance, $\sigma = \sqrt{\langle (x-\langle x\rangle)^2\rangle}$ is called the **standard deviation** of p. It is 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.

13.2.15 Probability Distribution Functions

The following table collects properties of probability distribution functions. In our notation we reserve the label $p(x)$ for the probability of a certain event, while $P(x)$ is the cumulative probability.

13.2.16 Probability Distribution Functions

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k\rangle = \sum_{i=1}^{N} x_i^k p(x_i),$$

if we have a discrete PDF or

$$\langle x^k\rangle = \int_a^b x^k p(x)dx,$$

in the case of a continuous PDF. We have already defined the mean value μ and the variance σ^2.

13.2.17 The three famous Probability Distribution Functions

There are at least three PDFs which one may encounter. These are the

Uniform distribution

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$

yielding probabilities different from zero in the interval $[a,b]$.

The exponential distribution

$$p(x) = \alpha \exp{(-\alpha x)},$$

yielding probabilities different from zero in the interval $[0,\infty)$ and with mean value

$$\mu = \int_0^\infty xp(x)dx = \int_0^\infty x\alpha \exp{(-\alpha x)}dx = \frac{1}{\alpha},$$

with variance

$$\sigma^2 = \int_0^\infty x^2 p(x)dx - \mu^2 = \frac{1}{\alpha^2}.$$

13.2.18 Probability Distribution Functions, the normal distribution

Finally, we have the so-called univariate normal distribution, or just the **normal distribution**

$$p(x) = \frac{1}{b\sqrt{2\pi}}\exp\left(-\frac{(x-a)^2}{2b^2}\right)$$

with probabilities different from zero in the interval $(-\infty,\infty)$. The integral $\int_{-\infty}^{\infty}\exp\left(-(x^2\right)dx$ appears in many calculations, its value is $\sqrt{\pi}$, a result we will need when we compute the mean value and the variance. The mean value is

$$\mu = \int_0^{\infty} xp(x)dx = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} x\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

which becomes with a suitable change of variables

$$\mu = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(a+b\sqrt{2}y)\exp -y^2 dy = a.$$

13.2.19 Probability Distribution Functions, the normal distribution

Similarly, the variance becomes

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}(x-\mu)^2\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$

and inserting the mean value and performing a variable change we obtain

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(b\sqrt{2}y)^2\exp\left(-y^2\right)dy = \frac{2b^2}{\sqrt{\pi}}\int_{-\infty}^{\infty} y^2\exp\left(-y^2\right)dy,$$

and performing a final integration by parts we obtain the well-known result $\sigma^2 = b^2$. It is useful to introduce the standard normal distribution as well, defined by $\mu = a = 0$, viz. a distribution centered around zero and with a variance $\sigma^2 = 1$, leading to

$p(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right)$.(13.3)

13.2.20 Probability Distribution Functions, the cumulative distribution

The exponential and uniform distributions have simple cumulative functions, whereas the normal distribution does not, being proportional to the so-called error function $erf(x)$, given by

$$P(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}\exp\left(-\frac{t^2}{2}\right)dt,$$

which is difficult to evaluate in a quick way.

13.2.21 Probability Distribution Functions, other important distribution

Some other PDFs which one encounters often in the natural sciences are the binomial distribution

$$p(x) = \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} \qquad x = 0, 1, \dots, n,$$

where y is the probability for a specific event, such as the tossing of a coin or moving left or right in case of a random walker. Note that x is a discrete stochastic variable.

The sequence of binomial trials is characterized by the following definitions

- Every experiment is thought to consist of N independent trials.
- In every independent trial one registers if a specific situation happens or not, such as the jump to the left or right of a random walker.
- The probability for every outcome in a single trial has the same value, for example the outcome of tossing (either heads or tails) a coin is always $1/2$.

13.2.22 Probability Distribution Functions, the binomial distribution

In order to compute the mean and variance we need to recall Newton's binomial formula

$$(a+b)^m = \sum_{n=0}^{m} \left(\begin{array}{c} m \\ n \end{array} \right) a^n b^{m-n},$$

which can be used to show that

$$\sum_{x=0}^{n} \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = (y+1-y)^n = 1,$$

the PDF is normalized to one. The mean value is

$$\mu = \sum_{x=0}^{n} x \left(\begin{array}{c} n \\ x \end{array} \right) y^x (1-y)^{n-x} = \sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} y^x (1-y)^{n-x},$$

resulting in

$$\mu = \sum_{x=0}^{n} x \frac{(n-1)!}{(x-1)!(n-1-(x-1))!} y^{x-1} (1-y)^{n-1-(x-1)},$$

which we rewrite as

$$\mu = ny \sum_{\nu=0}^{n} \left(\begin{array}{c} n-1 \\ \nu \end{array} \right) y^\nu (1-y)^{n-1-\nu} = ny(y+1-y)^{n-1} = ny.$$

The variance is slightly trickier to get. It reads $\sigma^2 = ny(1-y)$.

13.2.23 Probability Distribution Functions, Poisson's distribution

Another important distribution with discrete stochastic variables x isthe Poisson model, which resembles the exponential distribution and reads

$$p(x) = \frac{\lambda^x}{x!} e^{-\lambda} \qquad x = 0, 1, \dots, ; \lambda > 0.$$

In this case both the mean value and the variance are easier to calculate,

$$\mu = \sum_{x=0}^{\infty} x \frac{\lambda^x}{x!} e^{-\lambda} = \lambda e^{-\lambda} \sum_{x=1}^{\infty} \frac{\lambda^{x-1}}{(x-1)!} = \lambda,$$

and the variance is $\sigma^2 = \lambda$.

13.2.24 Probability Distribution Functions, Poisson's distribution

An example of applications of the Poisson distribution could be the counting of the number of α-particles emitted from a radioactive source in a given time interval. In the limit of $n \to \infty$ and for small probabilities y, the binomial distribution approaches the Poisson distribution. Setting $\lambda = ny$, with y the probability for an event in the binomial distribution we can show that

$$\lim_{n\to\infty} \left(\begin{array}{c} n \\ x \end{array}\right) y^x(1-y)^{n-x}e^{-\lambda} = \sum_{x=1}^{\infty}\frac{\lambda^x}{x!}e^{-\lambda}.$$

13.2.25 Meet the covariance!

An important quantity in a statistical analysis is the so-called covariance.

Consider 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

$\mathrm{Cov}(X_i,\, X_j) = \langle (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\rangle$(13.4)

$= \int\cdots\int (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)P(x_1,\dots,x_n)\,dx_1\dots dx_n,$(13.5)

with

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

13.2.26 Meet the covariance in matrix disguise

If we consider the above covariance as a matrix

$$C_{ij} = \mathrm{Cov}(X_i,\, X_j),$$

then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{Cov}(X_i,\, X_i) = \mathrm{Var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated.

13.2.27 Covariance

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

def covariance(x, y, n):
    sum = 0.0
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    for i in range(0, n):
        sum += (x[(i)]-mean_x)*(y[i]-mean_y)
    return  sum/n

n = 10

x=np.random.normal(size=n)
y = 4+3*x+np.random.normal(size=n)
covxy = covariance(x,y,n)
print(covxy)
z = np.vstack((x, y))
c = np.cov(z.T)

print(c)
```

```
1.3311287509199639
[[14.31659816 13.99268172 19.21873233 14.42933266 18.41189432 16.6094975
  11.00141571  4.4034975  14.34205163  7.63509233]
 [13.99268172 13.67609397 18.78390394 14.10286557 17.99532083 16.23370367
  10.75250606  4.30386732 14.0175593   7.46234655]
 [19.21873233 18.78390394 25.79940208 19.37006815 24.71629535 22.29674138
  14.76840108  5.91129532 19.2529013  10.24941779]
 [14.42933266 14.10286557 19.37006815 14.54295487 18.55687678 16.74028719
  11.08804516  4.43817236 14.45498656  7.69521404]
 [18.41189432 17.99532083 24.71629535 18.55687678 23.67865945 21.36068284
  14.14839622  5.66312819 18.44462881  9.81912823]
 [16.6094975  16.23370367 22.29674138 16.74028719 21.36068284 19.26962007
  12.76336631  5.1087472  16.63902752  8.85790364]
 [11.00141571 10.75250606 14.76840108 11.08804516 14.14839622 12.76336631
   8.45390408  3.38381409 11.02097512  5.86709383]
 [ 4.4034975   4.30386732  5.91129532  4.43817236  5.66312819  5.1087472
   3.38381409  1.35442722  4.41132648  2.34840076]
 [14.34205163 14.0175593  19.2529013  14.45498656 18.44462881 16.63902752
  11.02097512  4.41132648 14.36755036  7.64866676]
 [ 7.63509233  7.46234655 10.24941779  7.69521404  9.81912823  8.85790364
   5.86709383  2.34840076  7.64866676  4.07182169]]
```

13.2.28 Meet the covariance, uncorrelated events

Consider the stochastic variables X_i and X_j, $(i \ne j)$. We have

$$
\begin{aligned}
Cov(X_i,\, X_j) &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\
&= \langle x_i x_j - x_i \langle x_j \rangle - \langle x_i \rangle x_j + \langle x_i \rangle \langle x_j \rangle \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \langle x_j \rangle \rangle - \langle \langle x_i \rangle x_j \rangle + \langle \langle x_i \rangle \langle x_j \rangle \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle - \langle x_i \rangle \langle x_j \rangle + \langle x_i \rangle \langle x_j \rangle \\
&= \langle x_i x_j \rangle - \langle x_i \rangle \langle x_j \rangle
\end{aligned}
$$

If X_i and X_j are independent (assuming $i \ne j$), we have that

$$
\langle x_i x_j \rangle = \langle x_i \rangle \langle x_j \rangle,
$$

leading to

$$
Cov(X_i, X_j) = 0 \; (i \ne j).
$$

13.2.29 Numerical experiments and the covariance

Now that we have constructed an idealized mathematical framework, let us try to apply it to empirical observations. Examples of relevant physical phenomena may be spontaneous decays of nuclei, or a purely mathematical set of numbers produced by some deterministic mechanism. It is the latter we will deal with, using so-called pseudo-random number generators. In general our observations will contain only a limited set of observables. We remind the reader that a *stochastic process* is a process that produces sequentially a chain of values

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

13.2.30 Numerical experiments and the covariance

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

13.2.31 Numerical experiments and the covariance, actual situations

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

$$
\langle x_\alpha \rangle \equiv \frac{1}{n} \sum_{k=1}^{n} x_{\alpha,k}.
$$

The *sample variance* is:

$$
\mathrm{Var}(x) \equiv \frac{1}{n} \sum_{k=1}^{n} (x_{\alpha,k} - \langle x_\alpha \rangle)^2,
$$

with its square root being the *standard deviation of the sample*.

13.2.32 Numerical experiments and the covariance, our observables

You can think of the above observables as a set of quantities which define a given experiment. This experiment is then repeated several times, say m times. The total average is then

$$\langle X_m\rangle = \frac{1}{m}\sum_{\alpha=1}^{m} x_\alpha = \frac{1}{mn}\sum_{\alpha,k} x_{\alpha,k}, (13.6)$$

where the last sums end at m and n. The total variance is

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}(\langle x_\alpha\rangle - \langle X_m\rangle)^2,$$

which we rewrite as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m\rangle)(x_{\alpha,l} - \langle X_m\rangle).(13.7)$$

13.2.33 Numerical experiments and the covariance, the sample variance

We define also the sample variance σ^2 of all mn individual experiments as

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n}(x_{\alpha,k} - \langle X_m\rangle)^2.(13.8)$$

These quantities, being known experimental values or the results from our calculations, may differ, in some cases significantly, from the similarly named exact values for the mean value μ_X, the variance $\mathrm{Var}(X)$ and the covariance $\mathrm{Cov}(X,Y)$.

13.2.34 Numerical experiments and the covariance, central limit theorem

The central limit theorem states that the PDF $\tilde{p}(z)$ of the average of m random values corresponding to a PDF $p(x)$ is a normal distribution whose mean is the mean value of the PDF $p(x)$ and whose variance is the variance of the PDF $p(x)$ divided by m, the number of values used to compute z.

The central limit theorem leads then to the well-known expression for the standard deviation, given by

$$\sigma_m = \frac{\sigma}{\sqrt{m}}.$$

In many cases the above estimate for the standard deviation, in particular if correlations are strong, may be too simplistic. We need therefore a more precise defintion of the error and the variance in our results.

13.2.35 Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

$$\mu_X \approx X_m = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n} x_{\alpha,k}.$$

We can then use Eq. (7)

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

and rewrite it as

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle),$$

where the first term is the sample variance of all mn experiments divided by n and the last term is nothing but the covariance which arises when $k \neq l$.

13.2.36 Definition of Correlation Functions and Standard Deviation

Our estimate of the true average μ_X is the sample mean $\langle X_m \rangle$

If the observables are uncorrelated, then the covariance is zero and we obtain a total variance which agrees with the central limit theorem. Correlations may often be present in our data set, resulting in a non-zero covariance. The first term is normally called the uncorrelated contribution. Computationally the uncorrelated first term is much easier to treat efficiently than the second. We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

13.2.37 Definition of Correlation Functions and Standard Deviation

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

$$f_d = \frac{1}{nm}\sum_{\alpha=1}^{m}\sum_{k=1}^{n-d}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,k+d} - \langle X_m \rangle),$$

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

$$\frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,l} - \langle X_m \rangle) = \frac{2}{n}\sum_{d=1}^{n-1} f_d$$

13.2.38 Definition of Correlation Functions and Standard Deviation

The value of f_d reflects the correlation between measurements separated by the distance d in the samples. Notice that for $d = 0$, f is just the sample variance, σ^2. If we divide f_d by σ^2, we arrive at the so called **autocorrelation function**

$$\kappa_d = \frac{f_d}{\sigma^2}(13.9)$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d = 0$.

13.2.39 Definition of Correlation Functions and Standard Deviation, sample variance

The sample variance of the mn experiments can now be written in terms of the autocorrelation function

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{n} \cdot \sigma^2 \sum_{d=1}^{n-1} \frac{f_d}{\sigma^2} = \left(1 + 2\sum_{d=1}^{n-1} \kappa_d\right) \frac{1}{n}\sigma^2 = \frac{\tau}{n} \cdot \sigma^2(13.10)$$

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

$$\tau = 1 + 2\sum_{d=1}^{n-1} \kappa_d(13.11)$$

For a correlation free experiment, τ equals 1.

13.2.40 Definition of Correlation Functions and Standard Deviation

From the point of view of Eq. (*10*) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes

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

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\sigma_m^2 \approx \sigma^2/n$ to be less than the true sample error. The estimate of the error will be too "good". On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large. The solution to this problem is given by more practically oriented methods like the blocking technique.

13.2.41 Code to compute the Covariance matrix and the Covariance

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

# Sample covariance, note the factor 1/(n-1)
def covariance(x, y, n):
    sum = 0.0
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    for i in range(0, n):
        sum += (x[(i)]-mean_x)*(y[i]-mean_y)
    return  sum/(n-1.)

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))
covxx = covariance(x,x,n)
covyy = covariance(y,y,n)
covzz = covariance(z,z,n)
covxy = covariance(x,y,n)
covxz = covariance(x,z,n)
covyz = covariance(y,z,n)
print(covxx,covyy, covzz)
print(covxy,covxz, covyz)
w = np.vstack((x, y, z))
#print(w)
c = np.cov(w)
print(c)
#eigen = np.zeros(n)
Eigvals, Eigvecs = np.linalg.eig(c)
print(Eigvals)
```

```
0.2642716177294466
4.940580327634602
1.1829871582902483
1.015877605436128 9.696167748586095 17.72713727297909
2.9616975181296192 3.4095763130714993 9.375087003275128
[[ 1.01587761  2.96169752  3.40957631]
 [ 2.96169752  9.69616775  9.375087  ]
 [ 3.40957631  9.375087   17.72713727]]
[24.75536438  0.07216528  3.61165297]
```

13.3 Random Numbers

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think random numbers are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work.

13.4 Random Numbers, better name: pseudo random numbers

A disclaimer is however appropriate. It should be fairly obvious that something as deterministic as a computer cannot generate purely random numbers.

Numbers generated by any of the standard algorithms are in reality pseudo random numbers, hopefully abiding to the following criteria:

- they produce a uniform distribution in the interval [0,1].
- correlations between random numbers are negligible
- the period before the same sequence of random numbers is repeated is as large as possible and finally
- the algorithm should be fast.

13.5 Random number generator RNG

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\mathrm{MOD}(M),$$

which yield a number in the interval [0,1] through

$$x_i = N_i/M$$

The number M is called the period and it should be as large as possible and N_0 is the starting value, or seed. The function MOD means the remainder, that is if we were to evaluate $(13)\mathrm{MOD}(9)$, the outcome is the remainder of the division $13/9$, namely 4.

13.6 Random number generator RNG and periodic outputs

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.

Consider the following example

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, ...$, i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \dots$, a period of just 2.

13.7 Random number generator RNG and its period

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose lth number is the sum of the $l-i$th and $l-j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M).$$

13.8 Random number generator RNG, other examples

Such a generator again produces a sequence of pseudorandom numbers but this time with a period much larger than M. It is also possible to construct more elaborate algorithms by including more than two past terms in the sum of each iteration. One example is the generator of Marsaglia and Zaman which consists of two congruential relations

$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69), (13.12)$

followed by

$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}), (13.13)$

which according to the authors has a period larger than 2^{94}.

13.9 Random number generator RNG, other examples

Instead of using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j})$$

where the bitwise action of $\oplus$ means that if $N_{l-i} = N_{l-j}$ the result is 0 whereas if $N_{l-i} \neq N_{l-j}$ the result is 1. As an example, consider the case where $N_{l-i} = 6$ and $N_{l-j} = 11$. The first one has a bit representation (using 4 bits only) which reads 0110 whereas the second number is 1011. Employing the $\oplus$ operator yields 1101, or $2^3 + 2^2 + 2^0 = 13$.

In Fortran90, the bitwise $\oplus$ operation is coded through the intrinsic function $\mathrm{IEOR}(m, n)$ where m and n are the input numbers, while in C it is given by $m \wedge n$.

13.10 Random number generator RNG, RAN0

We show here how the linear congruential algorithm can be implemented, namely

$$N_i = (aN_{i-1})\mathrm{MOD}(M).$$

However, since a and N_{i-1} are integers and their multiplication could become greater than the standard 32 bit integer, there is a trick via Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

where we have defined

$$q = [M/a],$$

and

$$r = M \text{ MOD } a.$$

where the brackets denote integer division. In the code below the numbers q and r are chosen so that $r < q$.

13.11 Random number generator RNG, RAN0

To see how this works we note first that

$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\mathrm{MOD}(M),$ (13.14)

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\mathrm{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M.

13.12 Random number generator RNG, RAN0

We can now rewrite Eq. (*14*) as

$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\mathrm{MOD}(M),$ (13.15)

which results in

$(aN_{i-1})\mathrm{MOD}(M) = \left(a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r)\right)\mathrm{MOD}(M),$ (13.16) yielding

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1}\mathrm{MOD}(q)) - [N_{i-1}/q]r))\,\mathrm{MOD}(M).(13.17)$$

13.13 Random number generator RNG, RAN0

The term $[N_{i-1}/q]r$ is always smaller or equal $N_{i-1}(r/q)$ and with $r < q$ we obtain always a number smaller than N_{i-1}, which is smaller than M. And since the number $N_{i-1}\mathrm{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\mathrm{MOD}(q)) < aq$. Combined with our definition of $q = [M/a]$ ensures that this term is also smaller than M meaning that both terms fit into a 32-bit signed integer. None of these two terms can be negative, but their difference could. The algorithm below adds M if their difference is negative. Note that the program uses the bitwise $\oplus$ operator to generate the starting point for each generation of a random number. The period of $ran0$ is $\sim 2.1 \times 10^9$. A special feature of this algorithm is that is should never be called with the initial seed set to 0.

13.14 Random number generator RNG, RAN0 code

```
    /*
     ** The function
     **           ran0()
     ** is an "Minimal" random number generator of Park and Miller
     ** Set or reset the input value
     ** idum to any integer value (except the unlikely value MASK)
     ** to initialize the sequence; idum must not be altered between
     ** calls for sucessive deviates in a sequence.
     ** The function returns a uniform deviate between 0.0 and 1.0.
     */
double ran0(long &idum)
{
   const int a = 16807, m = 2147483647, q = 127773;
   const int r = 2836, MASK = 123459876;
   const double am = 1./m;
   long     k;
   double   ans;
   idum ^= MASK;
   k = (*idum)/q;
   idum = a*(idum - k*q) - r*k;
   // add m if negative difference
   if(idum < 0) idum += m;
   ans=am*(idum);
   idum ^= MASK;
   return ans;
} // End: function ran0()
```

13.14.1 Properties of Selected Random Number Generators

As mentioned previously, the underlying PDF for the generation of random numbers is the uniform distribution, meaning that the probability for finding a number $x$ in the interval [0,1] is $p(x) = 1$.

A random number generator should produce numbers which are uniformly distributed in this interval. The table shows the distribution of $N = 10000$ random numbers generated by the functions in the program library. We note in this table that the number of points in the various intervals $0.0 - 0.1$, $0.1 - 0.2$ etc are fairly close to 1000, with some minor deviations.

Two additional measures are the standard deviation σ and the mean $\mu = \langle x \rangle$.

13.14.2 Properties of Selected Random Number Generators

For the uniform distribution, the mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886.$$

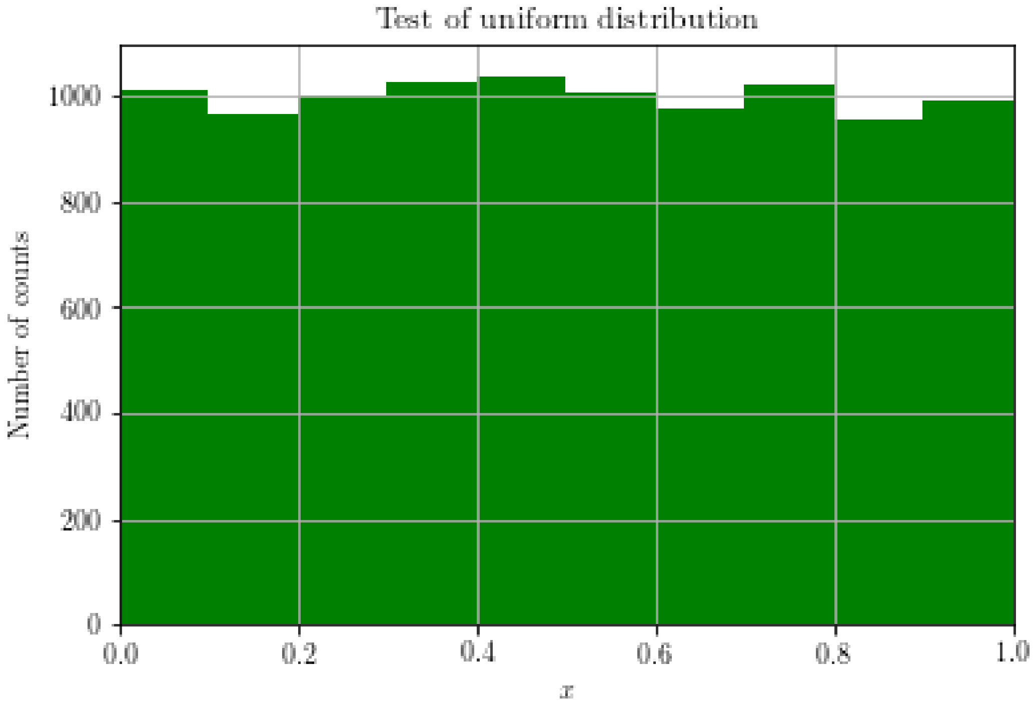
13.14.3 Properties of Selected Random Number Generators

The various random number generators produce results which agree rather well with these limiting values.

13.14.4 Simple demonstration of RNGs using python

The following simple Python code plots the distribution of the produced random numbers using the linear congruential RNG employed by Python. The trend displayed in the previous table is seen rather clearly.

```python
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random

# initialize the rng with a seed
random.seed()
counts = 10000
values = np.zeros(counts)
for i in range (1, counts, 1):
    values[i] = random.random()

# the histogram of the data
n, bins, patches = plt.hist(values, 10, facecolor='green')

plt.xlabel('$x$')
plt.ylabel('Number of counts')
plt.title(r'Test of uniform distribution')
plt.axis([0, 1, 0, 1100])
plt.grid(True)
plt.show()
```



13.14.5 Properties of Selected Random Number Generators

Since our random numbers, which are typically generated via a linear congruential algorithm, are never fully independent, we can then define an important test which measures the degree of correlation, namely the so-calledauto-correlation function defined previously, see again Eq. (*9*). We rewrite it here as

$$C_k = \frac{f_d}{\sigma^2},$$

with $C_0 = 1$. Recall that $\sigma^2 = \langle x_i^2 \rangle - \langle x_i \rangle^2$ and that

$$f_d = \frac{1}{nm}\sum_{\alpha=1}^{m}\sum_{k=1}^{n-d}(x_{\alpha,k} - \langle X_m \rangle)(x_{\alpha,k+d} - \langle X_m \rangle),$$

The non-vanishing of C_k for $k \neq 0$ means that the random numbers are not independent. The independence of the random numbers is crucial in the evaluation of other expectation values. If they are not independent, our assumption for approximating σ_N is no longer valid.

13.14.6 Autocorrelation function

This program computes the autocorrelation function as discussed in the equation on the previous slide for random numbers generated with the normal distribution $N(0, 1)$.

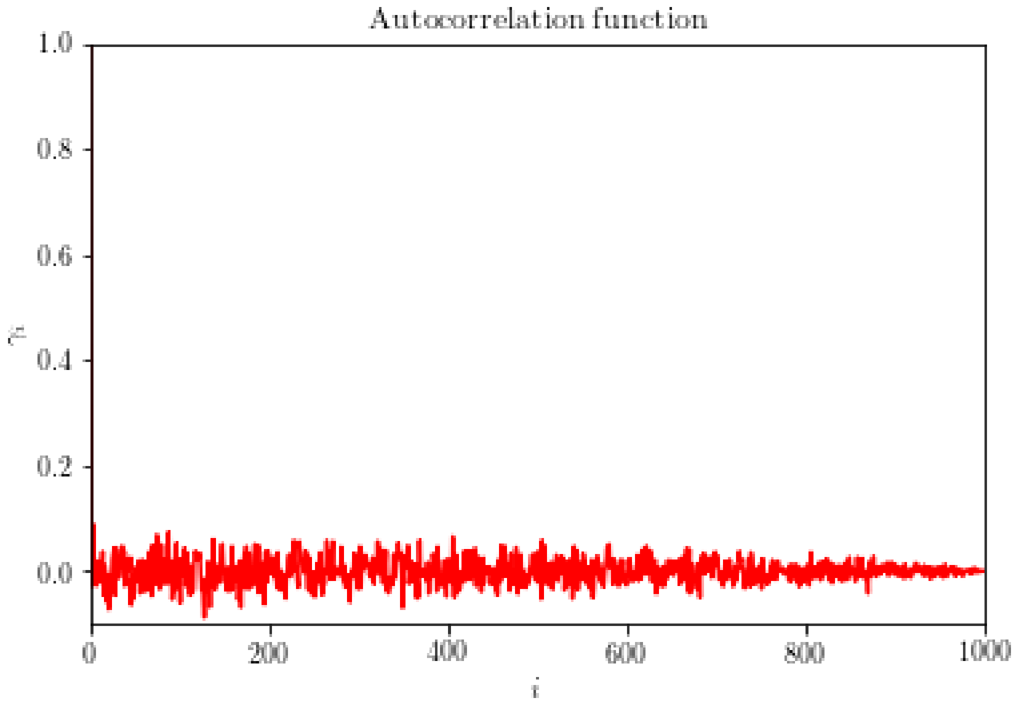
```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```
def autocovariance(x, n, k, mean_x):
    sum = 0.0
    for i in range(0, n-k):
        sum += (x[(i+k)]-mean_x)*(x[i]-mean_x)
    return  sum/n

n = 1000
x=np.random.normal(size=n)
autocor = np.zeros(n)
figaxis = np.zeros(n)
mean_x=np.mean(x)
var_x = np.var(x)
print(mean_x, var_x)
for i in range (0, n):
    figaxis[i] = i
    autocor[i]=(autocovariance(x, n, i, mean_x))/var_x

plt.plot(figaxis, autocor, "r-")
plt.axis([0,n,-0.1, 1.0])
plt.xlabel(r'$i$')
plt.ylabel(r'$\gamma_i$')
plt.title(r'Autocorrelation function')
plt.show()
```

```
0.042290321404441689 0.9337417995235178
```



As can be seen from the plot, the first point gives back the variance and a value of one. For the remaining values we notice that there are still non-zero values for the auto-correlation function.

13.14.7 Correlation function and which random number generators should I use

The program here computes the correlation function for one of the standard functions included with the c++ compiler.

```
//  This function computes the autocorrelation function for
//  the standard c++ random number generator

#include <fstream>
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;
// output file as global variable
ofstream ofile;

//     Main function begins here
int main(int argc, char* argv[])
{
     int n;
     char *outfilename;

     cin >> n;
     double MCint = 0.;      double MCintsqr2=0.;
     double invers_period = 1./RAND_MAX; // initialise the random number generator
     srand(time(NULL));  // This produces the so-called seed in MC jargon
     // Compute the variance and the mean value of the uniform distribution
     // Compute also the specific values x for each cycle in order to be able to
     // the covariance and the correlation function
     // Read in output file, abort if there are too few command-line arguments
     if( argc <= 2 ){
       cout << "Bad Usage: " << argv[0] <<
         " read also output file and number of cycles on same line" << endl;
       exit(1);
     }
     else{
       outfilename=argv[1];
     }
     ofile.open(outfilename);
     // Get  the number of Monte-Carlo samples
     n = atoi(argv[2]);
     double *X;
     X = new double[n];
     for (int i = 0;  i < n; i++){
           double x = double(rand())*invers_period;
           X[i] = x;
           MCint += x;
           MCintsqr2 += x*x;
     }
     double Mean = MCint/((double) n );
     MCintsqr2 = MCintsqr2/((double) n );
     double STDev = sqrt(MCintsqr2-Mean*Mean);
     double Variance = MCintsqr2-Mean*Mean;
//   Write mean value and standard deviation
     cout << " Standard deviation= " << STDev << " Integral = " << Mean << endl;

     // Now we compute the autocorrelation function
     double *autocor;  autocor = new double[n];
     for (int j = 0; j < n; j++){
```

(continues on next page)

(continued from previous page)

```
           double sum = 0.0;
           for (int k = 0; k < (n-j); k++){
             sum  += (X[k]-Mean)*(X[k+j]-Mean);
           }
           autocor[j] = sum/Variance/((double) n );
           ofile << setiosflags(ios::showpoint | ios::uppercase);
           ofile << setw(15) << setprecision(8) << j;
           ofile << setw(15) << setprecision(8) << autocor[j] << endl;
         }
         ofile.close();  // close output file
         return 0;
    }  // end of main program
```

13.15 Which RNG should I use?

- C++ has a class called **random**. The random class contains a large selection of RNGs and is highly recommended. Some of these RNGs have very large periods making it thereby very safe to use these RNGs in case one is performing large calculations. In particular, the Mersenne twister random number engine has a period of 2^{19937}.
- Add RNGs in Python

13.15.1 How to use the Mersenne generator

The following part of a c++ code (from project 4) sets up the uniform distribution for $x \in [0, 1]$.

```
    /*

    //  You need this
    #include <random>

    // Initialize the seed and call the Mersienne algo
    std::random_device rd;
    std::mt19937_64 gen(rd());
    // Set up the uniform distribution for x \in [[0, 1]
    std::uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);

    // Now use the RNG
    int ix = (int) (RandomNumberGenerator(gen)*NSpins);
```

13.15.2 Why blocking?

Statistical analysis.

```
* Monte Carlo simulations can be treated as *computer experiments*

* 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.
```

A very good article which explains blocking is H. Flyvbjerg and H. G. Petersen, *Error estimates on averages of correlated data*, Journal of Chemical Physics 91, 461-466 (1989).

13.15.3 Why blocking?

Statistical analysis.

```
* As in other experiments, Monte Carlo 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. (In VMC a common source is the step length or time step in importance␣
↪sampling)
```

13.15.4 Code to demonstrate the calculation of the autocorrelation function

The following code computes the autocorrelation function, the covariance and the standard deviation for standard RNG. The following file gives the code.

```
    //  This function computes the autocorrelation function for
    //  the Mersenne random number generator with a uniform distribution
    #include <iostream>
    #include <fstream>
    #include <iomanip>
    #include <cstdlib>
    #include <random>
    #include <armadillo>
    #include <string>
    #include <cmath>
    using namespace  std;
    using namespace arma;
    // output file
    ofstream ofile;

    //     Main function begins here
    int main(int argc, char* argv[])
    {
      int MonteCarloCycles;
      string filename;
      if (argc > 1) {
        filename=argv[1];
        MonteCarloCycles = atoi(argv[2]);
        string fileout = filename;
        string argument = to_string(MonteCarloCycles);
        fileout.append(argument);
        ofile.open(fileout);
      }

      // Compute the variance and the mean value of the uniform distribution
```

(continues on next page)

(continued from previous page)

```
      // Compute also the specific values x for each cycle in order to be able to
      // compute the covariance and the correlation function

      vec X  = zeros<vec>(MonteCarloCycles);
      double MCint = 0.;      double MCintsqr2=0.;
      std::random_device rd;
      std::mt19937_64 gen(rd());
      // Set up the uniform distribution for x \in [[0, 1]
      std::uniform_real_distribution<double> RandomNumberGenerator(0.0,1.0);
      for (int i = 0;  i < MonteCarloCycles; i++){
        double x =   RandomNumberGenerator(gen);
        X(i) = x;
        MCint += x;
        MCintsqr2 += x*x;
      }
      double Mean = MCint/((double) MonteCarloCycles );
      MCintsqr2 = MCintsqr2/((double) MonteCarloCycles );
      double STDev = sqrt(MCintsqr2-Mean*Mean);
      double Variance = MCintsqr2-Mean*Mean;
      //   Write mean value and variance
      cout << " Sample variance= " << Variance  << " Mean value = " << Mean << endl;
      // Now we compute the autocorrelation function
      vec autocorrelation = zeros<vec>(MonteCarloCycles);
      for (int j = 0; j < MonteCarloCycles; j++){
        double sum = 0.0;
        for (int k = 0; k < (MonteCarloCycles-j); k++){
          sum  += (X(k)-Mean)*(X(k+j)-Mean);
        }
        autocorrelation(j) = sum/Variance/((double) MonteCarloCycles );
        ofile << setiosflags(ios::showpoint | ios::uppercase);
        ofile << setw(15) << setprecision(8) << j;
        ofile << setw(15) << setprecision(8) << autocorrelation(j) << endl;
      }
      // Now compute the exact covariance using the autocorrelation function
      double Covariance = 0.0;
      for (int j = 0; j < MonteCarloCycles; j++){
        Covariance  += autocorrelation(j);
      }
      Covariance *=  2.0/((double) MonteCarloCycles);
      // Compute now the total variance, including the covariance, and obtain the␣
↪standard deviation
      double TotalVariance = (Variance/((double) MonteCarloCycles ))+Covariance;
      cout << "Covariance =" << Covariance << "Totalvariance= " << TotalVariance <<
↪"Sample Variance/n= " << (Variance/((double) MonteCarloCycles )) << endl;
      cout << " STD from sample variance= " << sqrt(Variance/((double)␣
↪MonteCarloCycles )) << " STD with covariance = " << sqrt(TotalVariance) << endl;

      ofile.close();  // close output file
      return 0;
    }  // end of main program
```

13.15.5 What is blocking?

Blocking.

```
* Say that we have a set of samples from a Monte Carlo experiment

* Assuming (wrongly) that our samples are uncorrelated our best estimate of the␣
↪standard deviation of the mean $\langle \mathbf{M}\rangle$ is given by
```

$$\sigma=\sqrt{\frac{1}{n}\left(\langle \mathbf{M}^2\rangle-\langle \mathbf{M}\rangle^2\right)}$$

- If the samples are correlated we can rewrite our results to show that

$$\sigma=\sqrt{\frac{1+2\tau/\Delta t}{n}\left(\langle \mathbf{M}^2\rangle-\langle \mathbf{M}\rangle^2\right)}$$

where $\tau$ is the correlation time (the time between a sample and the next uncorrelated sample) and Δt is time between each sample

13.15.6 What is blocking?

Blocking.

```
* If $\Delta t\gg\tau$ our first estimate of $\sigma$ still holds

* Much more common that $\Delta t<\tau$

* In the method of data blocking we divide the sequence of samples into blocks

* We then take the mean $\langle \mathbf{M}_i\rangle$ of block $i=1\ldots n_{blocks}$␣
↪to calculate the total mean and variance

* The size of each block must be so large that sample $j$ of block $i$ is not␣
↪correlated with sample $j$ of block $i+1$

* The correlation time $\tau$ would be a good choice
```

13.15.7 What is blocking?

Blocking.

```
* Problem: We don't know $\tau$ or it is too expensive to compute

* Solution: Make a plot of std. dev. as a function of blocksize

* The estimate of std. dev. of correlated data is too low $\to$ the error will␣
↪increase with increasing block size until the blocks are uncorrelated, where we␣
↪reach a plateau

* When the std. dev. stops increasing the blocks are uncorrelated
```

13.15.8 Implementation

```
* Do a Monte Carlo simulation, storing all samples to file

* Do the statistical analysis on this file, independently of your Monte Carlo program

* Read the file into an array

* Loop over various block sizes

* For each block size $n_b$, loop over the array in steps of $n_b$ taking the mean of elements $i n_b,\ldots,(i+1) n_b$

* Take the mean and variance of the resulting array

* Write the results for each block size to file for later
  analysis
```

13.15.9 Actual implementation with code, main function

When the file gets large, it can be useful to write your data in binary mode instead of ascii characters. The following python file reads data from file with the output from every Monte Carlo cycle.

```
# Blocking
    @timeFunction
    def blocking(self, blockSizeMax = 500):
        blockSizeMin = 1

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

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

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

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

```
  File "<ipython-input-6-2ff97f4bf03b>", line 2
    @timeFunction
    ^
IndentationError: unexpected indent
```

13.15.10 The Bootstrap method

The Bootstrap resampling method is also very popular. It is very simple:

1. Start with your sample of measurements and compute the sample variance and the mean values
2. Then start again but pick in a random way the numbers in the sample and recalculate the mean and the sample variance.
3. Repeat this K times.

It can be shown, see the article by Efron that it produces the correct standard deviation.

This method is very useful for small ensembles of data points.

13.15.11 Bootstrapping

Given a set of N data, assume that we are interested in some observable θ which may be estimated from that set. This observable can also be for example the result of a fit based on all N raw data. Let us call the value of the observable obtained from the original data set $\hat{\theta}$. One recreates from the sample repeatedly other samples by choosing randomly N data out of the original set. This costs essentially nothing, since we just recycle the original data set for the building of new sets.

13.15.12 Bootstrapping, recipe

Let us assume we have done this K times and thus have K sets of N data values each. Of course some values will enter more than once in the new sets. For each of these sets one computes the observable θ resulting in values θ_k with $k = 1, ..., K$. Then one determines

$$\tilde{\theta} = \frac{1}{K}\sum_{k=1}^{K}\theta_k,$$

and

$$sigma^2_{\tilde{\theta}} = \frac{1}{K}\sum_{k=1}^{K}\left(\theta_k - \tilde{\theta}\right)^2.$$

These are estimators for $\angle\theta\rangle$ and its variance. They are not unbiased and therefore $\tilde{\theta} \neq \hat{\theta}$ for finite K.

The difference is called bias and gives an idea on how far away the result may be from the true $\angle\theta\rangle$. As final result for the observable one quotes $\angle\theta\rangle = \tilde{\theta} \pm \sigma_{\tilde{\theta}}$.

13.15.13 Bootstrapping, code

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

13.15.14 Jackknife, code

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

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

Morten Hjorth-Jensen, Department of Physics, University of Oslo and Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Date: **Sep 11, 2020**

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

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

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

- Allows for **easy** hands-on understanding of gradient descent methods
- and many more features

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.

13.16.2 Regression analysis, overarching aims

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

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

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

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

13.16.3 Regression analysis, overarching aims II

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

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

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

13.16.4 Examples

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

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

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

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

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

13.16.5 General linear models

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

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

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

where ϵ_i is the error in our approximation.

13.16.6 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

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

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

Defining the vectors

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

and

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

and

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

and the design matrix

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

we can rewrite our equations as

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

The above design matrix is called a Vandermonde matrix.

13.16.8 Generalizing the fitting procedure as a linear algebra problem

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

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

Note that we have $p = n$ here. The matrix is symmetric. This is generally not the case!

13.16.9 Generalizing the fitting procedure as a linear algebra problem

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

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

and without loss of generality we rewrite again our equations as

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

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

13.16.10 Optimizing our parameters

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

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

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

13.16.11 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.

```python
%matplotlib inline

# 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()
```

(continues on next page)

(continued from previous page)

```
# 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)
```

```
       1      A    A^(2/3)  A^(-1/3)       1/A
A
1    1.0    1.0   1.000000  1.000000  1.000000
2    1.0    2.0   1.587401  0.793701  0.500000
3    1.0    3.0   2.080084  0.693361  0.333333
4    1.0    4.0   2.519842  0.629961  0.250000
5    1.0    5.0   2.924018  0.584804  0.200000
..   ...    ...        ...       ...       ...
264  1.0  264.0  41.153106  0.155883  0.003788
265  1.0  265.0  41.256962  0.155687  0.003774
266  1.0  266.0  41.360688  0.155491  0.003759
269  1.0  269.0  41.671089  0.154911  0.003717
270  1.0  270.0  41.774300  0.154720  0.003704

[267 rows x 5 columns]
```

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

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

throughout these lectures.

13.16.12 Optimizing our parameters, more details

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

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

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

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

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

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

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

It is also common to define the function C as

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

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

13.16.13 Interpretations and optimizing our parameters

The function

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

can be linked to the variance of the quantity y_i if we interpret the latter as the mean value. When linking (see the discussion below) with the maximum likelihood approach below, we will indeed interpret y_i as a mean value

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

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

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

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

In practical terms it means we will require

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

which results in

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

or in a matrix-vector form as

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

13.16.14 Interpretations and optimizing our parameters

We can rewrite

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

as

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

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

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

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

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

13.16.15 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.

26

<<<!!MATH_BLOCK

27

<<<!!MATH_BLOCK

28

<<<!!MATH_BLOCK

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

13.16.16 Interpretations and optimizing our parameters

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

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

and with

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

we have

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

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

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

13.16.17 Own code for Ordinary Least Squares

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

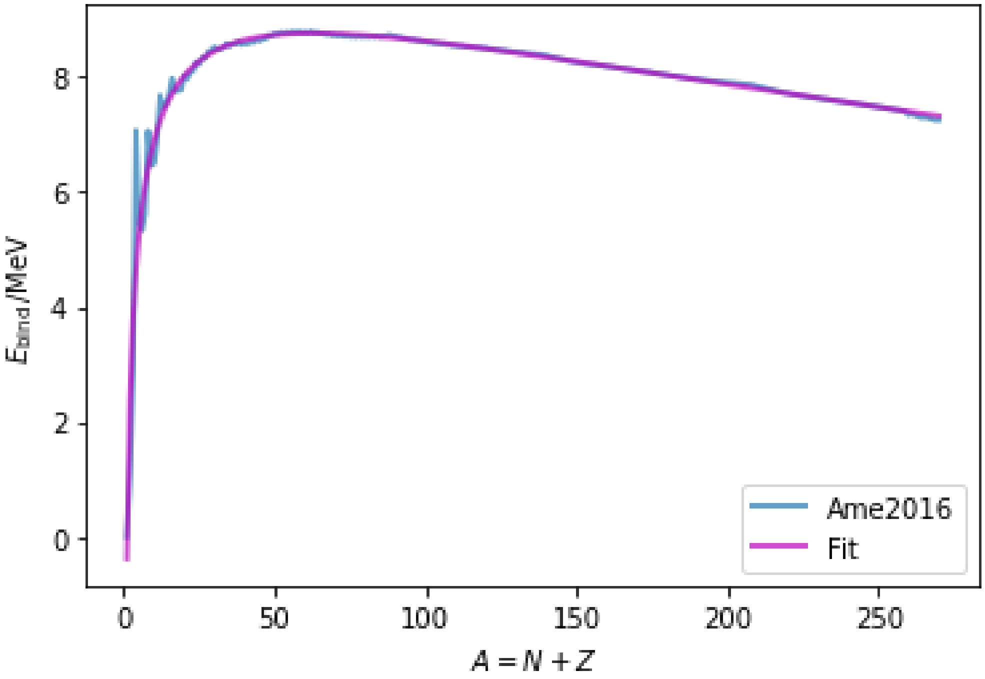
```
# matrix inversion to find beta
beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Energies)
# and then make the prediction
ytilde = X @ beta
```

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

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

And finally we plot our fit with and compare with data

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



13.16.18 Adding error analysis and training set up

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

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

and we would be using it as

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

```
0.9547578478889096
```

We can easily 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))
```

```
0.03787596148305239
```

and finally the relative error as

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

```
A
1    0           inf
2    1      1.123190
3    2      0.327631
4    6      0.344172
5    9      0.044402
              ...
264  3304   0.009911
265  3310   0.009154
266  3317   0.007824
269  3338   0.011347
270  3344   0.009790
Name: Ebinding, Length: 267, dtype: float64
```

13.16.19 The χ^2 function

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

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

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

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

13.16.20 The χ^2 function

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

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

which results in

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

or in a matrix-vector form as

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

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

13.16.21 The χ^2 function

We can rewrite

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

as

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

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

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

13.16.22 The χ^2 function

If we then introduce the matrix

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

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

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

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

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

resulting in

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

13.16.23 The χ^2 function

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

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

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

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\beta_0} = -2\left[\frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i-\beta_0-\beta_1x_i}{\sigma_i^2}\right)\right] = 0,$$

and

$$\frac{\partial\chi^2(\boldsymbol{\beta})}{\partial\beta_1} = -\frac{2}{n}\left[\sum_{i=0}^{n-1}x_i\left(\frac{y_i-\beta_0-\beta_1x_i}{\sigma_i^2}\right)\right] = 0.$$

13.16.24 The χ^2 function

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

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

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

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

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

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

we obtain

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

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

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

13.16.25 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 forces. This time the file is presented as a standard **csv** file.

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

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

13.16.26 The code

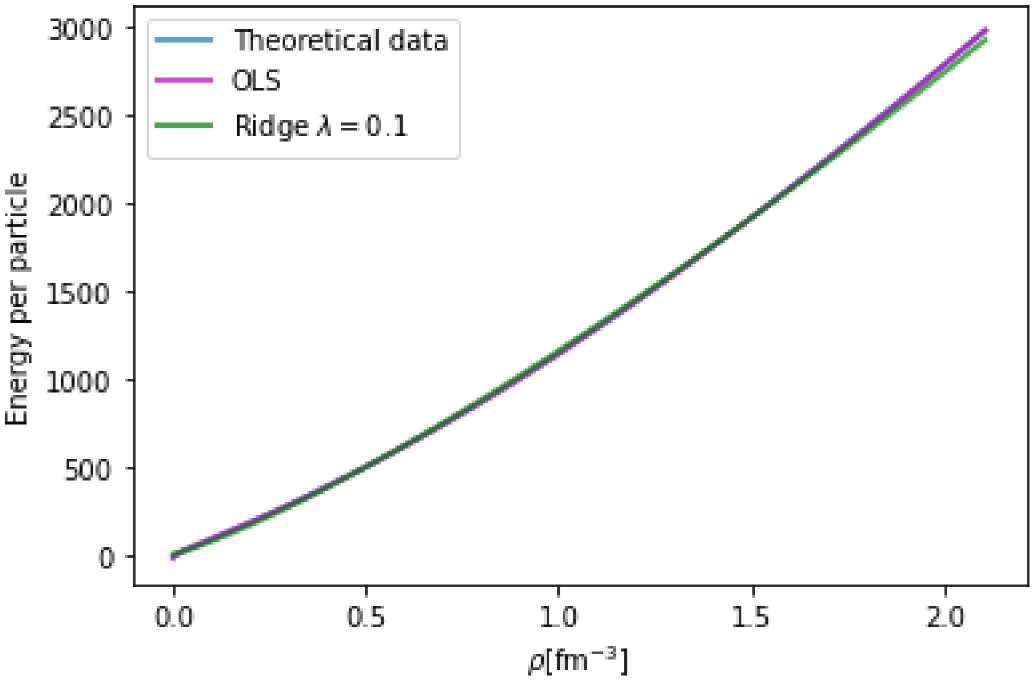
```python
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import sklearn.linear_model as skl
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# 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']
```

(continues on next page)

(continued from previous page)

```
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()
```

```
Mean squared error: 12.36
Variance score: 1.00
Mean absolute error: 2.83
[   0.          618.32047562 -861.13519106 1404.91549644] -11.057088709963637
Mean squared error: 197.93
Variance score: 1.00
Mean absolute error: 11.69
[  0.          28.18220995 282.79902342 842.30879705] 12.946893955206974
```



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

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

13.16.27 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):
```

(continues on next page)

(continued from previous page)

```
    os.makedirs(DATA_ID)

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

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

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

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

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

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

```
Training R2
0.9999941778221182
Training MSE
2.1663059697316176
Test R2
0.9999188193958993
```

(continues on next page)

(continued from previous page)

```
Test MSE
57.32209754560856
```

13.16.28 The Boston housing data example

The Boston housingdata 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

13.16.29 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()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

Then we invoke Pandas

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

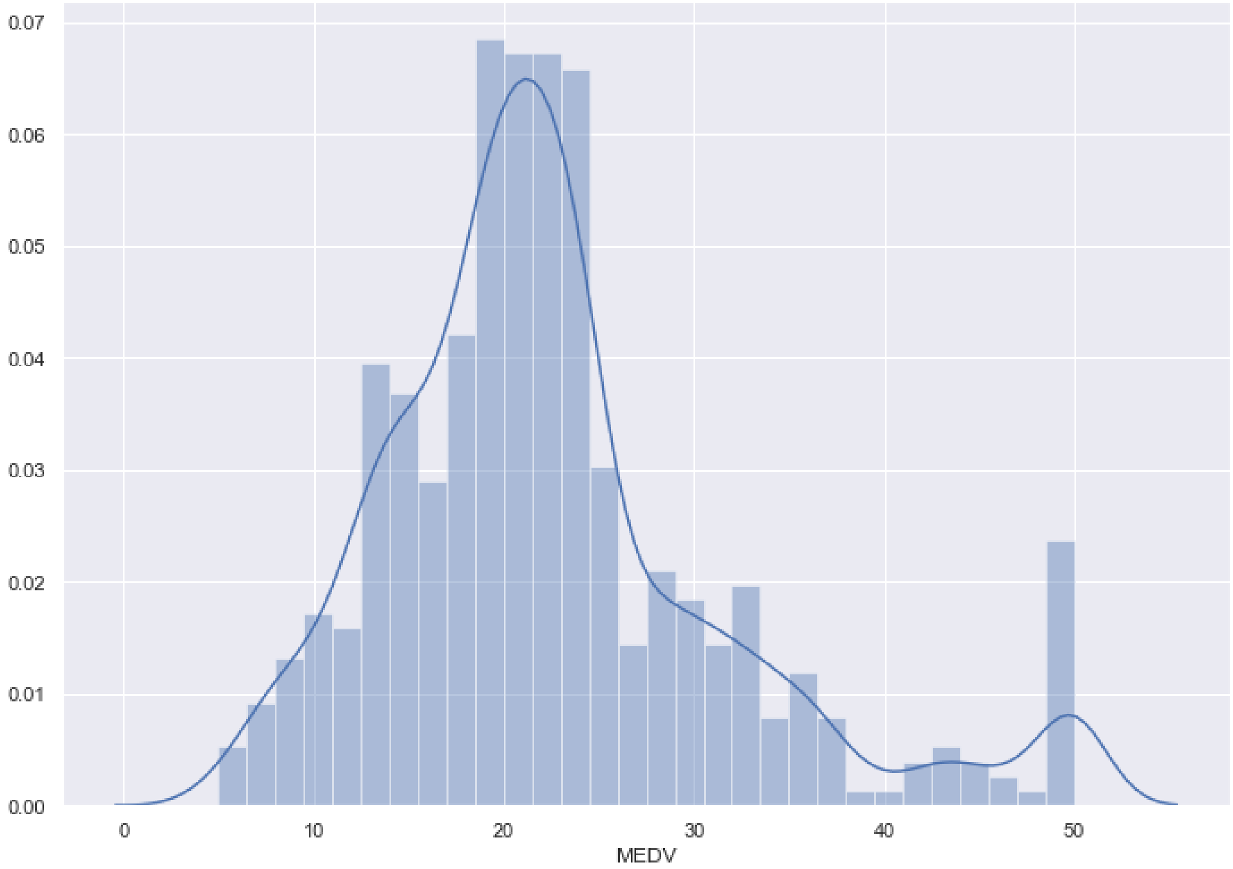
and preprocess the data

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

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

We can then visualize the data

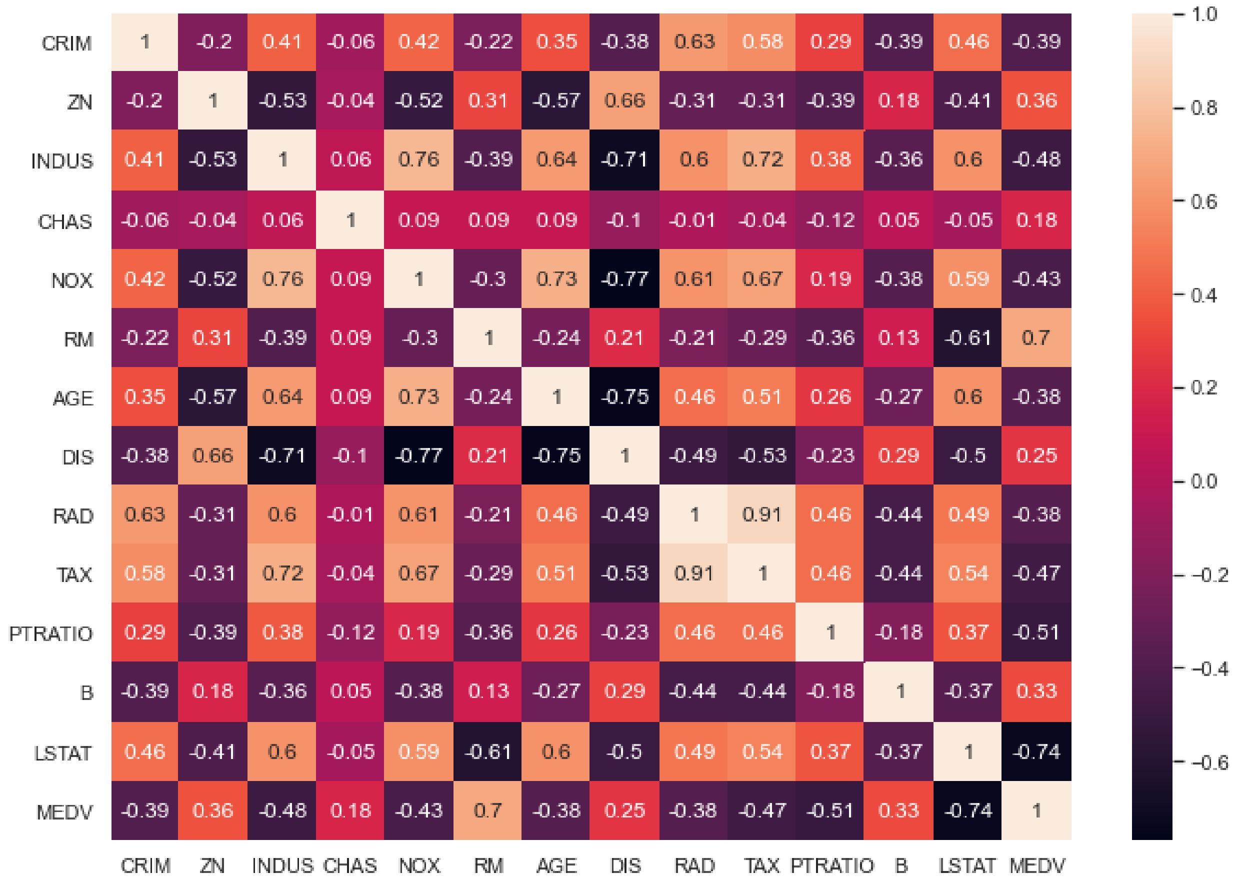
```
# set the size of the figure
sns.set(rc={'figure.figsize':(11.7,8.27)})

# 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)
```

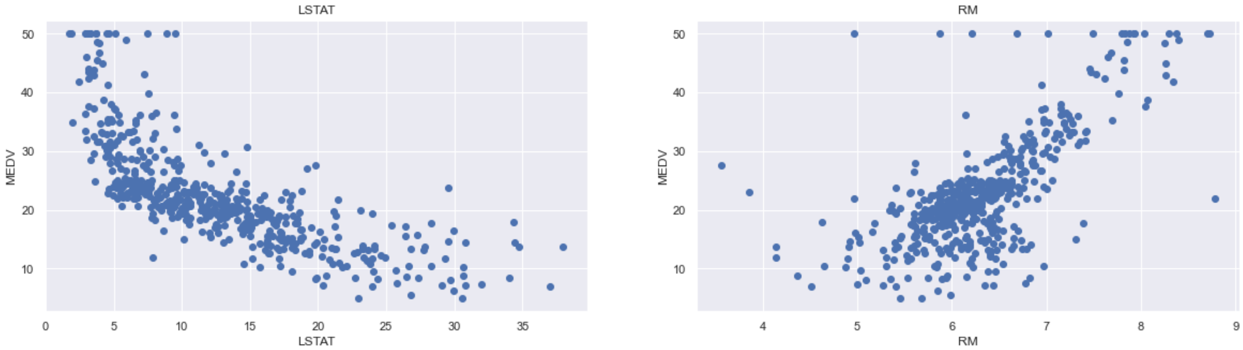
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa71a4449d0>
```



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
CRIM	1	-0.2	0.41	-0.06	0.42	-0.22	0.35	-0.38	0.63	0.58	0.29	-0.39	0.46	-0.39
ZN	-0.2	1	-0.53	-0.04	-0.52	0.31	-0.57	0.66	-0.31	-0.31	-0.39	0.18	-0.41	0.36
INDUS	0.41	-0.53	1	0.06	0.76	-0.39	0.64	-0.71	0.6	0.72	0.38	-0.36	0.6	-0.48
CHAS	-0.06	-0.04	0.06	1	0.09	0.09	0.09	-0.1	-0.01	-0.04	-0.12	0.05	-0.05	0.18
NOX	0.42	-0.52	0.76	0.09	1	-0.3	0.73	-0.77	0.61	0.67	0.19	-0.38	0.59	-0.43
RM	-0.22	0.31	-0.39	0.09	-0.3	1	-0.24	0.21	-0.21	-0.29	-0.36	0.13	-0.61	0.7
AGE	0.35	-0.57	0.64	0.09	0.73	-0.24	1	-0.75	0.46	0.51	0.26	-0.27	0.6	-0.38
DIS	-0.38	0.66	-0.71	-0.1	-0.77	0.21	-0.75	1	-0.49	-0.53	-0.23	0.29	-0.5	0.25
RAD	0.63	-0.31	0.6	-0.01	0.61	-0.21	0.46	-0.49	1	0.91	0.46	-0.44	0.49	-0.38
TAX	0.58	-0.31	0.72	-0.04	0.67	-0.29	0.51	-0.53	0.91	1	0.46	-0.44	0.54	-0.47
PTRATIO	0.29	-0.39	0.38	-0.12	0.19	-0.36	0.26	-0.23	0.46	0.46	1	-0.18	0.37	-0.51
B	-0.39	0.18	-0.36	0.05	-0.38	0.13	-0.27	0.29	-0.44	-0.44	-0.18	1	-0.37	0.33
LSTAT	0.46	-0.41	0.6	-0.05	0.59	-0.61	0.6	-0.5	0.49	0.54	0.37	-0.37	1	-0.74
MEDV	-0.39	0.36	-0.48	0.18	-0.43	0.7	-0.38	0.25	-0.38	-0.47	-0.51	0.33	-0.74	1

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

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

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

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



Now we start training our model

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

We split the data into training and test sets

```
from sklearn.model_selection import train_test_split

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

```
(404, 2)
(102, 2)
(404,)
(102,)
```

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
```

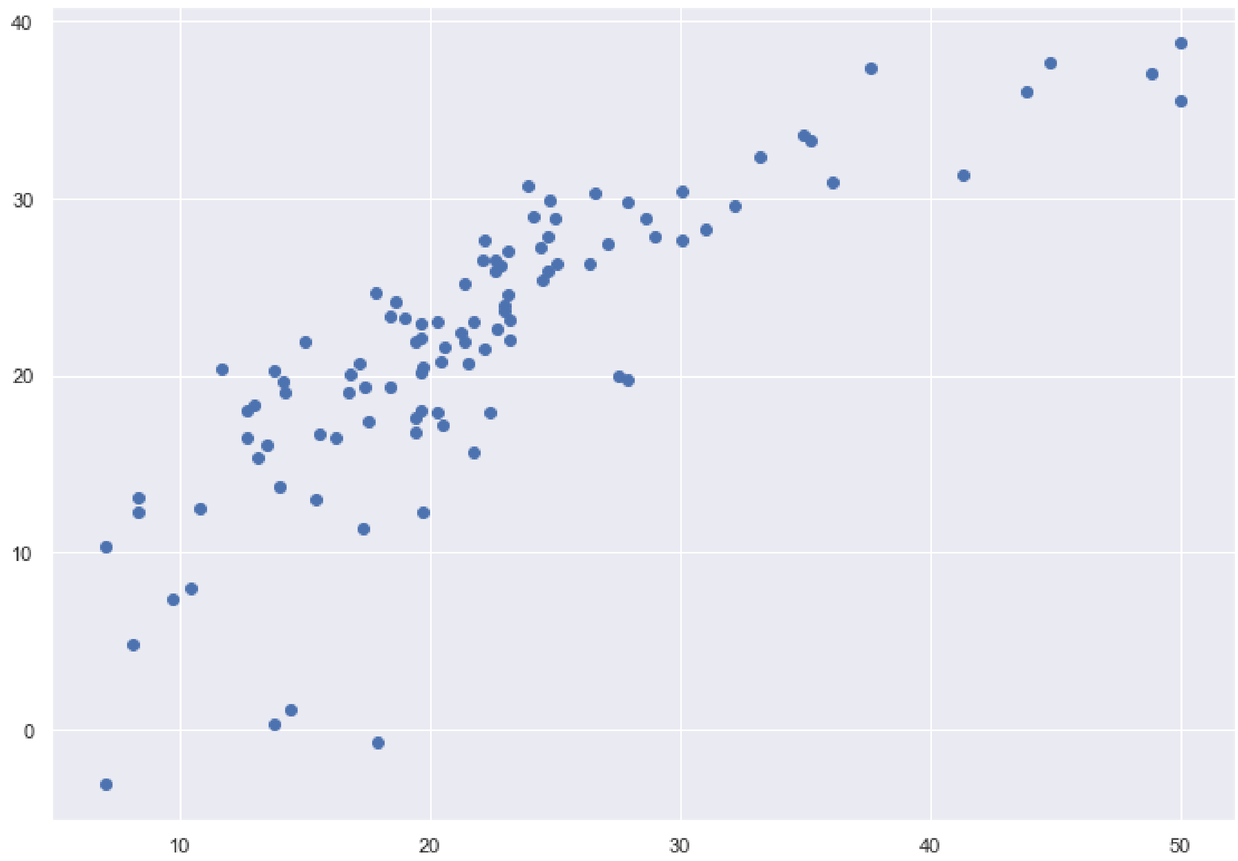
(continues on next page)

(continued from previous page)

```
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))
```

```
The model performance for training set
--------------------------------------
RMSE is 5.6371293350711955
R2 score is 0.6300745149331701


The model performance for testing set
--------------------------------------
RMSE is 5.137400784702912
R2 score is 0.6628996975186952
```

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



13.16.30 Reducing the number of degrees of freedom, overarching view

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

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

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

13.16.31 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

13.16.32 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.

13.16.33 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))
```

(continues on next page)

(continued from previous page)

```
        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)))
```

```
MSE before scaling: 0.00
R2 score before scaling 0.99
Feature min values before scaling:
 [1.00000000e+00 3.31070195e-03 4.26950921e-04 1.09607474e-05
 1.41350724e-06 1.82287089e-07 3.62877676e-08 4.67970118e-09
 6.03498219e-10 7.78276403e-11 1.20137983e-10 1.54930958e-11
 1.99800273e-12 2.57664120e-13 3.32285826e-14 3.97741054e-13
 5.12930224e-14 6.61479151e-15 8.53049103e-16 1.10009933e-16
 1.41869739e-17]
Feature max values before scaling:
 [1.         0.99858638 0.99964819 0.99717477 0.99823507 0.99929651
 0.99576514 0.99682395 0.99788388 0.99894494 0.99435752 0.99541483
 0.99647326 0.99753282 0.99859351 0.99295188 0.99400769 0.99506463
 0.99612269 0.99718188 0.99824219]
Feature min values after scaling:
 [ 0.         -1.7247216  -1.76366767 -1.11821924 -1.1207329  -1.12330932
 -0.88380004 -0.88090074 -0.87801917 -0.87516765 -0.75358163 -0.74975133
 -0.74594528 -0.74216924 -0.73842878 -0.66775774 -0.66415352 -0.66058072
 -0.65704182 -0.65353917 -0.65007499]
Feature max values after scaling:
 [0.         1.71439421 1.7725581  2.19850437 2.24574687 2.29247479
 2.59145801 2.63154972 2.67112014 2.71016871 2.93008405 2.96590898
 3.00128975 3.03623229 3.07074298 3.23004415 3.26340549 3.29641768
 3.32908665 3.36141837 3.39341886]
MSE after  scaling: 0.00
R2 score for  scaled data: 0.99
```

13.16.34 The singular value decomposition

The examples we have looked at so far are cases where we normally can invert the matrix $\boldsymbol{X}^T\boldsymbol{X}$. 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 the be 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 insights about the ordinary least squares approach, and later shrinkage methods like Ridge and Lasso regressions.

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.

13.16.35 Linear Regression Problems

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

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

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

Super-collinearity of an $(n \times p)$-dimensional design matrix $\mathbf{X}$ implies that the inverse of the matrix $\boldsymbol{X}^T\boldsymbol{X}$ (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

$$\boldsymbol{X} = \left[\begin{array}{cc} 1 & -1 \\ 1 & -1 \end{array} \right].$$

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

13.16.36 Fixing the singularity

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

$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y},$ (13.18)

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

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

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

13.16.37 Basic math of the SVD

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

$$(\lambda_1, \boldsymbol{u}_1), \dots, (\lambda_n, \boldsymbol{u}_n),$$

and the eigenvalues are given by the diagonal matrix

$$\boldsymbol{\Sigma} = \mathrm{Diag}(\lambda_1, \dots, \lambda_n).$$

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

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

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

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

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

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

13.16.38 The SVD, a Fantastic Algorithm

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

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

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

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

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

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

In the general case, where our design matrix $\boldsymbol{X}$ has dimension $n \times p$, the matrix is thus decomposed into an $n \times n$ orthogonal matrix $\boldsymbol{U}$, a $p \times p$ orthogonal matrix $\boldsymbol{V}$ and a diagonal matrix $\boldsymbol{\Sigma}$ 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 $\boldsymbol{U}$ are called the left singular vectors while the columns of $\boldsymbol{V}$ are the right singular vectors.

13.16.39 Economy-size SVD

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

The economy-size decomposition removes extra rows or columns of zeros from the diagonal matrix of singular values, $\boldsymbol{\Sigma}$, along with the columns in either $\boldsymbol{U}$ or $\boldsymbol{V}$ 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 $\boldsymbol{U}$ and $\boldsymbol{\Sigma}$ has dimension $p \times p$. If $p > n$, then only the first n columns of $\boldsymbol{V}$ are computed and $\boldsymbol{\Sigma}$ 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.

13.16.40 Codes for the SVD

```python
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).
    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)
```

```
[[ 1. -1.  2.]
 [ 1.  0.  1.]
 [ 1.  2. -1.]
 [ 1.  1.  0.]]
[[ 4.  2.  2.]
 [ 2.  6. -4.]
 [ 2. -4.  6.]]
[[-9.57425734e-17  8.16496581e-01 -5.77350269e-01]
 [-7.07106781e-01  4.08248290e-01  5.77350269e-01]
 [ 7.07106781e-01  4.08248290e-01  5.77350269e-01]]
[1.00000000e+01 6.00000000e+00 9.10898112e-32]
[[ 3.33066907e-17 -7.07106781e-01  7.07106781e-01]
 [ 8.16496581e-01  4.08248290e-01  4.08248290e-01]
 [ 5.77350269e-01 -5.77350269e-01 -5.77350269e-01]]
[[-3.65939208e+30  3.65939208e+30  3.65939208e+30]
 [ 3.65939208e+30 -3.65939208e+30 -3.65939208e+30]
 [ 3.65939208e+30 -3.65939208e+30 -3.65939208e+30]]
```

The matrix $\boldsymbol{X}$ 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 $\boldsymbol{X}^T\boldsymbol{X}$ results in the program terminating due to a singular matrix.

13.16.41 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

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

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

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

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

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

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

This means that

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

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

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

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

Going back to our OLS equation we have

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

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

13.16.42 Ridge and LASSO Regression

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

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

or we can state it as

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

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

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

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

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

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

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

we have a new optimization equation

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

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

Here we have defined the norm-1 as

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

13.16.43 More on Ridge Regression

Using the matrix-vector expression for Ridge regression,

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

by taking the derivatives with respect to $\boldsymbol{\beta}$ we obtain then a slightly modified matrix inversion problem which for finite values of λ does not suffer from singularity problems. We obtain

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

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

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

with t a finite positive number.

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

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

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

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

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

For Ridge regression this becomes

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

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

13.16.44 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 $\boldsymbol{y}$ with respect to the orthonormal basis $\boldsymbol{U}$, 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 σ_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 $\boldsymbol{X}\boldsymbol{v}_j$ shows that this quantity is equal to σ_j^2/n. With a parameter λ we can thus shrink the role of specific parameters.

13.16.45 More interpretations

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

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

In this case the standard OLS results in

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

and

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

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

We will come back to more interpreations after we have gone through some of the statistical analysis part.

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.

13.16.46 A better understanding of regularization

The parameter λ that we have introduced in the Ridge (and Lasso as well) regression is often called a regularization parameter or shrinkage parameter. It is common to call it a hyperparameter. What does it mean mathemtically?

Here we will first look at how to analyze the difference between the standard OLS equations and the Ridge expressions in terms of a linear algebra analysis using the SVD algorithm. Thereafter, we will link (see the material on the bias-variance tradeoff below) these observation to the statisical analysis of the results. In particular we consider how the variance of the parameters $\boldsymbol{\beta}$ is affected by changing the parameter λ.

13.16.47 Decomposing the OLS and Ridge expressions

We have our design matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$. With the SVD we decompose it as

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

with $\boldsymbol{U} \in \mathbb{R}^{n\times n}$, $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times p}$ and $\boldsymbol{V} \in \mathbb{R}^{p\times p}$.

The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary/orthonormal matrices, that is in case the matrices are real we have $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}$ and $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}$.

13.16.48 Introducing the Covariance and Correlation functions

Before we discuss the link between for example Ridge regression and the singular value decomposition, we need to remind ourselves about the definition of the covariance and the correlation function. These are quantities

Suppose we have defined two vectors $\hat{x}$ and $\hat{y}$ with n elements each. The covariance matrix $\boldsymbol{C}$ is defined as

$$\boldsymbol{C}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} \mathrm{cov}[\boldsymbol{x},\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{cov}[\boldsymbol{y},\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{y},\boldsymbol{y}] \end{bmatrix},$$

where for example

$$\mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] = \frac{1}{n}\sum_{i=0}^{n-1}(x_i-\overline{x})(y_i-\overline{y}).$$

With this definition and recalling that the variance is defined as

$$\mathrm{var}[\boldsymbol{x}] = \frac{1}{n}\sum_{i=0}^{n-1}(x_i-\overline{x})^2,$$

we can rewrite the covariance matrix as

$$\boldsymbol{C}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}] & \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{cov}[\boldsymbol{x},\boldsymbol{y}] & \mathrm{var}[\boldsymbol{y}] \end{bmatrix}.$$

The covariance takes values between zero and infinity and may thus lead to problems with loss of numerical precision for particularly large values. It is common to scale the covariance matrix by introducing instead the correlation matrix defined via the so-called correlation function

$$\mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] = \frac{\mathrm{cov}[\boldsymbol{x},\boldsymbol{y}]}{\sqrt{\mathrm{var}[\boldsymbol{x}]\mathrm{var}[\boldsymbol{y}]}}.$$

The correlation function is then given by values $\mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] \in [-1,1]$. This avoids eventual problems with too large values. We can then define the correlation matrix for the two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as

$$\boldsymbol{K}[\boldsymbol{x},\boldsymbol{y}] = \begin{bmatrix} 1 & \mathrm{corr}[\boldsymbol{x},\boldsymbol{y}] \\ \mathrm{corr}[\boldsymbol{y},\boldsymbol{x}] & 1 \end{bmatrix},$$

In the above example this is the function we constructed using **pandas**.

13.16.49 Correlation Function and Design/Feature Matrix

In our derivation of the various regression algorithms like **Ordinary Least Squares** or **Ridge regression** we defined the design/feature matrix $\boldsymbol{X}$ as

$$\boldsymbol{X}=\begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1}\\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1}\\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1}\\ \dots & \dots & \dots & \dots\dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1}\\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1}\\ \end{bmatrix},$$

with $\boldsymbol{X} \in \mathbb{R}^{n\times p}$, with the predictors/features p refering to the column numbers and the entries n being the row elements. We can rewrite the design/feature matrix in terms of its column vectors as

$$\boldsymbol{X}=\begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \dots & \boldsymbol{x}_{p-1}\end{bmatrix},$$

with a given vector

$$\boldsymbol{x}_i^T = \begin{bmatrix}x_{0,i} & x_{1,i} & x_{2,i}& \dots & \dots x_{n-1,i}\end{bmatrix}.$$

With these definitions, we can now rewrite our 2×2 correaltion/covariance matrix in terms of a moe general design/feature matrix $\boldsymbol{X}\in \mathbb{R}^{n\times p}$. This leads to a $p\times p$ covariance matrix for the vectors $\boldsymbol{x}_i$ with $i=0,1,\dots,p-1$

$$\boldsymbol{C}[\boldsymbol{x}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_0,\boldsymbol{x}_{p-1}]\\ \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_0] & \mathrm{var}[\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_1,\boldsymbol{x}_{p-1}]\\ \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_1] & \mathrm{var}[\boldsymbol{x}_2] & \dots & \dots & \mathrm{cov}[\boldsymbol{x}_2,\boldsymbol{x}_{p-1}]\\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_1] & \mathrm{cov}[\boldsymbol{x}_{p-1},\boldsymbol{x}_{2}] & \dots & \dots & \mathrm{var}[\boldsymbol{x}_{p-1}]\\ \end{bmatrix},$$

and the correlation matrix

$$\boldsymbol{K}[\boldsymbol{x}] = \begin{bmatrix} 1 & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_1] & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_2] & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_0,\boldsymbol{x}_{p-1}]\\ \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_0] & 1 & \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_2] & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_1,\boldsymbol{x}_{p-1}]\\ \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_0] & \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_1] & 1 & \dots & \dots & \mathrm{corr}[\boldsymbol{x}_2,\boldsymbol{x}_{p-1}]\\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_0] & \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_1] & \mathrm{corr}[\boldsymbol{x}_{p-1},\boldsymbol{x}_{2}] & \dots & \dots & 1\\ \end{bmatrix},$$

13.16.50 Covariance Matrix Examples

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 $2\times n$ matrix $\boldsymbol{W}$

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

which in turn is converted into into the 2×2 covariance matrix $\boldsymbol{C}$ via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples $\boldsymbol{x}$ 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))
W = np.vstack((x, y))
C = np.cov(W)
print(C)
```

```
0.04144248019085487
4.194659964041423
[[ 1.10772753  3.41408669]
 [ 3.41408669 11.38266616]]
```

13.16.51 Correlation Matrix

The previous example can be converted into the correlation matrix by simply scaling the matrix elements with the variances. We should also subtract the mean values for each column. This leads to the following code which sets up the correlations matrix for the previous example in a more brute force way. Here we scale the mean values for each column of the design matrix, calculate the relevant mean values and variances and then finally set up the 2 × 2 correlation matrix (since we have only two vectors).

```
import numpy as np
n = 100
# define two vectors                                                                    ␣
↪
x = np.random.random(size=n)
y = 4+3*x+np.random.normal(size=n)
#scaling the x and y vectors                                                            ␣
↪
x = x - np.mean(x)
y = y - np.mean(y)
variance_x = np.sum(x@x)/n
variance_y = np.sum(y@y)/n
print(variance_x)
print(variance_y)
cov_xy = np.sum(x@y)/n
cov_xx = np.sum(x@x)/n
cov_yy = np.sum(y@y)/n
C = np.zeros((2,2))
C[0,0]= cov_xx/variance_x
C[1,1]= cov_yy/variance_y
C[0,1]= cov_xy/np.sqrt(variance_y*variance_x)
C[1,0]= C[0,1]
print(C)
```

```
0.09052813127366242
1.5055457430392476
[[1.         0.56806976]
 [0.56806976 1.        ]]
```

We see that the matrix elements along the diagonal are one as they should be and that the matrix is symmetric. Furthermore, diagonalizing this matrix we easily see that it is a positive definite matrix.

The above procedure with **numpy** can be made more compact if we use **pandas**.

13.16.52 Correlation Matrix with Pandas

We whow here how we can set up the correlation matrix using **pandas**, as done in this simple code

```
import numpy as np
import pandas as pd
n = 10
x = np.random.normal(size=n)
x = x - np.mean(x)
y = 4+3*x+np.random.normal(size=n)
y = y - np.mean(y)
X = (np.vstack((x, y))).T
print(X)
Xpd = pd.DataFrame(X)
print(Xpd)
correlation_matrix = Xpd.corr()
print(correlation_matrix)
```

```
[[ 0.19619662 -0.78402323]
 [ 0.31757369  1.22304781]
 [-0.69338544 -1.67201132]
 [-0.83049344 -3.62540546]
 [ 1.07466493  3.158262  ]
 [-0.63779754 -0.28923407]
 [-0.50263516 -2.66065903]
 [ 0.59499124  1.87464536]
 [-0.05490614  0.80385431]
 [ 0.53579124  1.97152362]]
          0         1
0  0.196197 -0.784023
1  0.317574  1.223048
2 -0.693385 -1.672011
3 -0.830493 -3.625405
4  1.074665  3.158262
5 -0.637798 -0.289234
6 -0.502635 -2.660659
7  0.594991  1.874645
8 -0.054906  0.803854
9  0.535791  1.971524
          0         1
0  1.000000  0.896919
1  0.896919  1.000000
```

We expand this model to the Franke function discussed above.

13.16.53 Correlation Matrix with Pandas and the Franke function

```
# Common imports
import numpy as np
import pandas as pd


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 = 4
N = 100
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)

Xpd = pd.DataFrame(X)
# subtract the mean values and set up the covariance matrix
Xpd = Xpd - Xpd.mean()
covariance_matrix = Xpd.cov()
print(covariance_matrix)
```

```
     0         1         2         3         4         5         6         7  \
0  0.0  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
1  0.0  0.089183  0.084797  0.089147  0.085535  0.082266  0.080527  0.077480
2  0.0  0.084797  0.082978  0.087978  0.085822  0.083785  0.081609  0.079503
3  0.0  0.089147  0.087978  0.094876  0.092810  0.090814  0.088924  0.086714
4  0.0  0.085535  0.085822  0.092810  0.091736  0.090594  0.088276  0.086777
5  0.0  0.082266  0.083785  0.090814  0.090594  0.090194  0.087493  0.086618
6  0.0  0.080527  0.081609  0.088924  0.088276  0.087493  0.085397  0.084155
7  0.0  0.077480  0.079503  0.086714  0.086777  0.086618  0.084155  0.083459
8  0.0  0.074802  0.077623  0.084723  0.085401  0.085786  0.082993  0.082776
9  0.0  0.072449  0.075952  0.082939  0.084154  0.085015  0.081923  0.082128
```

(continues on next page)

(continued from previous page)

```
10  0.0  0.071557  0.073969  0.080992  0.081340  0.081431  0.079151  0.078671
11  0.0  0.069117  0.072168  0.079031  0.079900  0.080455  0.077870  0.077808
12  0.0  0.066984  0.070584  0.077297  0.078621  0.079584  0.076722  0.077031
13  0.0  0.065125  0.069200  0.075772  0.077496  0.078821  0.075706  0.076344
14  0.0  0.063508  0.067997  0.074439  0.076518  0.078164  0.074816  0.075749

          8         9        10        11        12        13        14
0   0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
1   0.074802  0.072449  0.071557  0.069117  0.066984  0.065125  0.063508
2   0.077623  0.075952  0.073969  0.072168  0.070584  0.069200  0.067997
3   0.084723  0.082939  0.080992  0.079031  0.077297  0.075772  0.074439
4   0.085401  0.084154  0.081340  0.079900  0.078621  0.077496  0.076518
5   0.085786  0.085015  0.081431  0.080455  0.079584  0.078821  0.078164
6   0.082993  0.081923  0.079151  0.077870  0.076722  0.075706  0.074816
7   0.082776  0.082128  0.078671  0.077808  0.077031  0.076344  0.075749
8   0.082517  0.082246  0.078172  0.077682  0.077236  0.076846  0.076519
9   0.082246  0.082311  0.077682  0.077525  0.077378  0.077256  0.077172
10  0.078172  0.077682  0.074339  0.073633  0.072989  0.072416  0.071916
11  0.077682  0.077525  0.073633  0.073260  0.072914  0.072607  0.072349
12  0.077236  0.077378  0.072989  0.072914  0.072835  0.072771  0.072733
13  0.076846  0.077256  0.072416  0.072607  0.072771  0.072926  0.073089
14  0.076519  0.077172  0.071916  0.072349  0.072733  0.073089  0.073437
```

We note here that the covariance is zero for the first rows and columns since all matrix elements in the design matrix were set to one (we are fitting the function in terms of a polynomial of degree n).

This means that the variance for these elements will be zero and will cause problems when we set up the correlation matrix. We can simply drop these elements and construct a correlation matrix without these elements.

13.16.54 Rewriting the Covariance and/or Correlation Matrix

We can rewrite the covariance matrix in a more compact form in terms of the design/feature matrix $\boldsymbol{X}$ as

$$\boldsymbol{C}[\boldsymbol{x}] = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X} = \mathbb{E}[\boldsymbol{X}^T\boldsymbol{X}].$$

To see this let us simply look at a design matrix $\boldsymbol{X} \in \mathbb{R}^{2\times 2}$

$$\boldsymbol{X} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 \end{bmatrix}.$$

If we then compute the expectation value

$$\mathbb{E}[\boldsymbol{X}^T\boldsymbol{X}] = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X} = \begin{bmatrix} x_{00}^2 + x_{01}^2 & x_{00}x_{10} + x_{01}x_{11} \\ x_{10}x_{00} + x_{11}x_{01} & x_{10}^2 + x_{11}^2 \end{bmatrix},$$

which is just

$$\boldsymbol{C}[\boldsymbol{x}_0, \boldsymbol{x}_1] = \boldsymbol{C}[\boldsymbol{x}] = \begin{bmatrix} \mathrm{var}[\boldsymbol{x}_0] & \mathrm{cov}[\boldsymbol{x}_0, \boldsymbol{x}_1] \\ \mathrm{cov}[\boldsymbol{x}_1, \boldsymbol{x}_0] & \mathrm{var}[\boldsymbol{x}_1] \end{bmatrix},$$

where we wrote \$$\boldsymbol{C}[\boldsymbol{x}_0, \boldsymbol{x}_1] = \boldsymbol{C}[\boldsymbol{x}] toindicatethatthisthecovarianceofthevectors$\boldsymbol{x}$ofthedesign/featurematrix$\b

It is easy to generalize this to a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$.

13.16.55 Linking with SVD

See lecture september 11. More text to be added here soon.

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

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

13.16.57 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.

13.16.58 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.

13.16.59 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.

13.16.60 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.

13.16.61 Linking the regression analysis with a statistical interpretation

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

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

$$\mbox{Cov}(\varepsilon_{i_1}, \varepsilon_{i_2}) = \left\{ \begin{array}{lcc} \sigma^2 & \mbox{if} & i_1 = i_2, \\ 0 & \mbox{if} & i_1 \not= i_2. \end{array} \right.$$

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

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

13.16.62 Assumptions made

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

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

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

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

13.16.63 Expectation value and variance

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

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

while its variance is

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

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

13.16.64 Expectation value and variance for β

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

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

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

We can also calculate the variance

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

$$to$$

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

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

$$\boldsymbol{\beta}\boldsymbol{\beta}^T + \sigma^2 \, (\mathbf{X}^T\mathbf{X})^{-1} - \boldsymbol{\beta}\boldsymbol{\beta}^T \;\; = \;\; \sigma^2 \, (\mathbf{X}^T\mathbf{X})^{-1},$$

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

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

It is rather straightforward to show that

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

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

We can also compute the variance as

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

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

With this, we can compute the difference

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

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

13.16.65 Resampling methods

With all these analytical equations for both the OLS and Ridge regression, we will now outline how to assess a given model. This will lead us to a discussion of the so-called bias-variance tradeoff (see below) and so-called resampling methods.

One of the quantities we have discussed as a way to measure errors is the mean-squared error (MSE), mainly used for fitting of continuous functions. Another choice is the absolute error.

In the discussions below we will focus on the MSE and in particular since we will split the data into test and training data, we discuss the

1. prediction error or simply the **test error** $\mathrm{Err}_{\mathrm{Test}}$, where we have a fixed training set and the test error is the MSE arising from the data reserved for testing. We discuss also the
2. training error $\mathrm{Err}_{\mathrm{Train}}$, which is the average loss over the training data.

As our model becomes more and more complex, more of the training data tends to used. The training may thence adapt to more complicated structures in the data. This may lead to a decrease in the bias (see below for code example) and a slight increase of the variance for the test error. For a certain level of complexity the test error will reach minimum, before starting to increase again. The training error reaches a saturation.

13.16.66 Resampling methods: Jackknife and Bootstrap

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

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

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

13.16.67 Resampling methods: Jackknife

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

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

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

13.16.68 Jackknife code example

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

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

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

    return t


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

(continues on next page)

(continued from previous page)

```
mu, sigma = 100, 15
datapoints = 10000
x = mu + sigma*random.randn(datapoints)
# jackknife returns the data sample                                          ␣
↪                                                                            ␣
↪
t = jackknife(x, stat)
```

```
Runtime: 0.389789 sec
Jackknife Statistics :
original           bias      std. error
 100.259        100.249        0.149656
```

13.16.69 Resampling methods: Bootstrap

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

1. The bootstrap is quite general, although there are some cases in which it fails.
2. Because it does not require distributional assumptions (such as normally distributed errors), the bootstrap can provide more accurate inferences when the data are not well behaved or when the sample size is small.
3. It is possible to apply the bootstrap to statistics with sampling distributions that are difficult to derive, even asymptotically.
4. It is relatively simple to apply the bootstrap to complex data-collection plans (such as stratified and clustered samples).

13.16.70 Resampling methods: Bootstrap background

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

13.16.71 Resampling methods: More Bootstrap background

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

1. Drawing lots of numbers from $p(x)$, suppose we call one such set of numbers $(X_1^*, X_2^*, \cdots, X_n^*)$.
2. Then using these numbers, we could compute a replica of $\widehat{\theta}$ called $\widehat{\theta}^*$.

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

13.16.72 Resampling methods: Bootstrap approach

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

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

13.16.73 Resampling methods: Bootstrap steps

The independent bootstrap works like this:

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

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

13.16.74 Code example for the Bootstrap method

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

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

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

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

(continues on next page)

(continued from previous page)

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


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

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

plt.show()
```

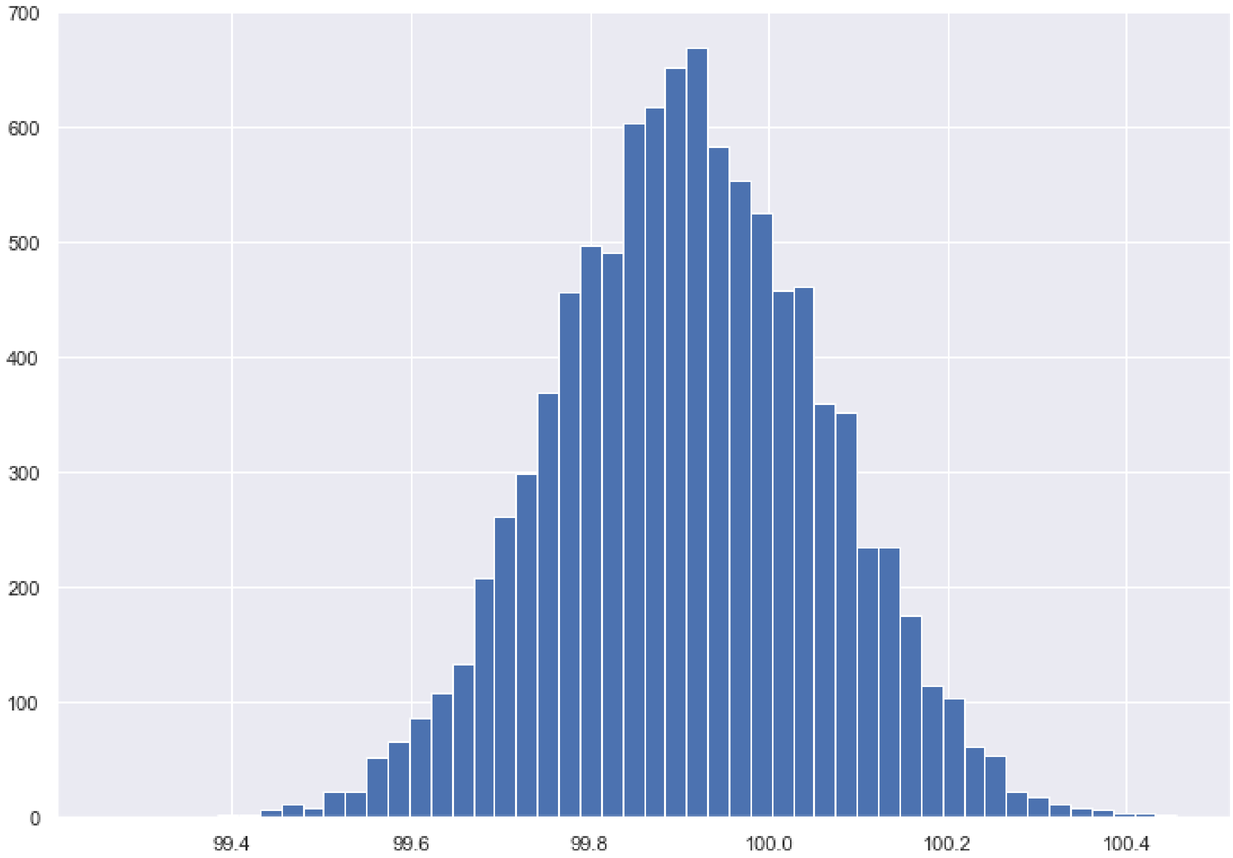
```
Runtime: 2.00056 sec
Bootstrap Statistics :
original           bias      std. error
 99.9087  15.0416        99.9098        0.150465
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-29-53990135e988> in <module>
     29 t = bootstrap(x, stat, datapoints)
     30 # the histogram of the bootstrapped  data
---> 31 n, binsboot, patches = plt.hist(t, 50, normed=1, facecolor='red', alpha=0.75)
     32
     33 # add a 'best fit' line

~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/pyplot.py in hist(x, bins,␣
↪range, density, weights, cumulative, bottom, histtype, align, orientation, rwidth,␣
↪log, color, label, stacked, data, **kwargs)
   2603         orientation='vertical', rwidth=None, log=False, color=None,
   2604         label=None, stacked=False, *, data=None, **kwargs):
-> 2605     return gca().hist(
   2606         x, bins=bins, range=range, density=density, weights=weights,
   2607         cumulative=cumulative, bottom=bottom, histtype=histtype,

~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/__init__.py in inner(ax, data,␣
↪*args, **kwargs)
   1563     def inner(ax, *args, data=None, **kwargs):
   1564         if data is None:
-> 1565             return func(ax, *map(sanitize_sequence, args), **kwargs)
   1566
   1567         bound = new_sig.bind(ax, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/axes/_axes.py in hist(self, x,␣
↪bins, range, density, weights, cumulative, bottom, histtype, align, orientation,␣
↪rwidth, log, color, label, stacked, **kwargs)
   6817                 if patch:
   6818                     p = patch[0]
-> 6819                     p.update(kwargs)
   6820                     if lbl is not None:
   6821                         p.set_label(lbl)

~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/artist.py in update(self,␣
↪props)
   1004
   1005         with cbook._setattr_cm(self, eventson=False):
-> 1006             ret = [_update_property(self, k, v) for k, v in props.items()]
   1007
   1008         if len(ret):

~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/artist.py in <listcomp>(.0)
   1004
   1005         with cbook._setattr_cm(self, eventson=False):
-> 1006             ret = [_update_property(self, k, v) for k, v in props.items()]
   1007
   1008         if len(ret):

~/opt/anaconda3/lib/python3.8/site-packages/matplotlib/artist.py in _update_
↪property(self, k, v)
    999                     func = getattr(self, 'set_' + k, None)
   1000                     if not callable(func):
-> 1001                         raise AttributeError('{!r} object has no property {!r}'
   1002                                              .format(type(self).__name__, k))
   1003                     return func(v)

AttributeError: 'Rectangle' object has no property 'normed'
```



13.16.75 Various steps in cross-validation

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

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

- Define a range of interest for the penalty parameter.
- Divide the data set into training and test set comprising samples $\{1, \dots, n\} \setminus i$ and $\{i\}$, respectively.
- Fit the linear regression model by means of ridge estimation for each λ in the grid using the training set, and the corresponding estimate of the error variance $\boldsymbol{\sigma}_{-i}^2(\lambda)$, as

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

- Evaluate the prediction performance of these models on the test set by $\log\{L[y_i, \boldsymbol{X}_{i,*}; \boldsymbol{\beta}_{-i}(\lambda), \boldsymbol{\sigma}_{-i}^2(\lambda)]\}$. Or, by the prediction error $|y_i - \boldsymbol{X}_{i,*}\boldsymbol{\beta}_{-i}(\lambda)|$, the relative error, the error squared or the R2 score function.
- Repeat the first three steps such that each sample plays the role of the test set once.

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

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

13.16.77 Cross-validation in brief

For the various values of k

1. shuffle the dataset randomly.
2. Split the dataset into k groups.
3. For each unique group:

a. Decide which group to use as set for test data

b. Take the remaining groups as a training data set

c. Fit a model on the training set and evaluate it on the test set

d. Retain the evaluation score and discard the model

1. Summarize the model using the sample of model evaluation scores

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

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

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

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

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

## Cross-validation on Ridge regression using KFold only

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

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

# Initialize a KFold instance
```

(continues on next page)

(continued from previous page)

```
k = 5
kfold = KFold(n_splits = k)

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

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

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

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

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

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

        j += 1
    i += 1


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

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

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

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

    X = poly.fit_transform(x[:, np.newaxis])
    estimated_mse_folds = cross_val_score(ridge, X, y[:, np.newaxis], scoring='neg_
↪mean_squared_error', cv=kfold)

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

    i += 1

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

plt.figure()
```

(continues on next page)

(continued from previous page)

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

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

plt.legend()

plt.show()
```

13.16.79 The bias-variance tradeoff

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

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

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

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

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

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

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

We can rewrite this as

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

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

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

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

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

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

which, using the abovementioned expectation values can be rewritten as

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

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

13.16.80 Example code for Bias-Variance tradeoff

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

np.random.seed(2018)

n = 500
n_boostraps = 100
degree = 18  # A quite high value, just to show.
noise = 0.1

# Make data set.
x = np.linspace(-1, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2) + np.random.normal(0, 0.1, x.shape)

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

# Combine x transformation and model into one operation.
# Not neccesary, but convenient.
model = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression(fit_
↪intercept=False))

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

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

# Note: Expectations and variances taken w.r.t. different training
# data sets, hence the axis=1. Subsequent means are taken across the test data
# set in order to obtain a total value, but before this we have error/bias/variance
# calculated per data point in the test set.
# Note 2: The use of keepdims=True is important in the calculation of bias as this
# maintains the column vector form. Dropping this yields very unexpected results.
error = np.mean( np.mean((y_test - y_pred)**2, axis=1, keepdims=True) )
bias = np.mean( (y_test - np.mean(y_pred, axis=1, keepdims=True))**2 )
variance = np.mean( np.var(y_pred, axis=1, keepdims=True) )
print('Error:', error)
print('Bias^2:', bias)
print('Var:', variance)
print('{} >= {} + {} = {}'.format(error, bias, variance, bias+variance))

plt.plot(x[::5, :], y[::5, :], label='f(x)')
plt.scatter(x_test, y_test, label='Data points')
plt.scatter(x_test, np.mean(y_pred, axis=1), label='Pred')
plt.legend()
plt.show()
```

13.16.81 Understanding what happens

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

np.random.seed(2018)

n = 400
n_boostraps = 100
maxdegree = 30


# Make data set.
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)
error = np.zeros(maxdegree)
bias = np.zeros(maxdegree)
variance = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

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

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

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

```
Polynomial degree: 0
Error: 0.2937910450030775
Bias^2: 0.2929212799917661
Var: 0.0008697650113114119
0.2937910450030775 >= 0.2929212799917661 + 0.0008697650113114119 = 0.2937910450030775
Polynomial degree: 1
```

(continues on next page)

(continued from previous page)

```
Error: 0.06894146856540674
Bias^2: 0.06832043024896824
Var: 0.0006210383164384989
0.06894146856540674 >= 0.06832043024896824 + 0.0006210383164384989 = 0.
↪06894146856540674
Polynomial degree: 2
Error: 0.06106765054837855
Bias^2: 0.060547654220995305
Var: 0.0005199963273832372
0.06106765054837855 >= 0.060547654220995305 + 0.0005199963273832372 = 0.
↪061067650548378545
Polynomial degree: 3
Error: 0.03346202229536659
Bias^2: 0.0331409564680546
Var: 0.00032106582731199456
0.03346202229536659 >= 0.0331409564680546 + 0.00032106582731199456 = 0.
↪03346202229536659
Polynomial degree: 4
Error: 0.0335277871704832
Bias^2: 0.03311607538577367
Var: 0.0004117117847095335
0.0335277871704832 >= 0.03311607538577367 + 0.0004117117847095335 = 0.
↪03352778717048321
Polynomial degree: 5
Error: 0.025517151530854786
Bias^2: 0.024968890209256463
Var: 0.0005482613215983259
0.025517151530854786 >= 0.024968890209256463 + 0.0005482613215983259 = 0.
↪02551715153085479
Polynomial degree: 6
Error: 0.01994607606842793
Bias^2: 0.019502076889868637
Var: 0.00044399917855929527
0.01994607606842793 >= 0.019502076889868637 + 0.00044399917855929527 = 0.
↪019946076068427934
Polynomial degree: 7
Error: 0.018695928655417676
Bias^2: 0.01797984009000237
Var: 0.0007160885654153078
0.018695928655417676 >= 0.01797984009000237 + 0.0007160885654153078 = 0.
↪01869592865541768
Polynomial degree: 8
Error: 0.010736105188369479
Bias^2: 0.010376602508045063
Var: 0.00035950268032441344
0.010736105188369479 >= 0.010376602508045063 + 0.00035950268032441344 = 0.
↪010736105188369477
Polynomial degree: 9
Error: 0.01101329065273084
Bias^2: 0.010539027867197629
Var: 0.0004742627855332104
0.01101329065273084 >= 0.010539027867197629 + 0.0004742627855332104 = 0.
↪01101329065273084
Polynomial degree: 10
Error: 0.010972468815261078
Bias^2: 0.010593565969983903
Var: 0.00037890284527716995
```

(continues on next page)

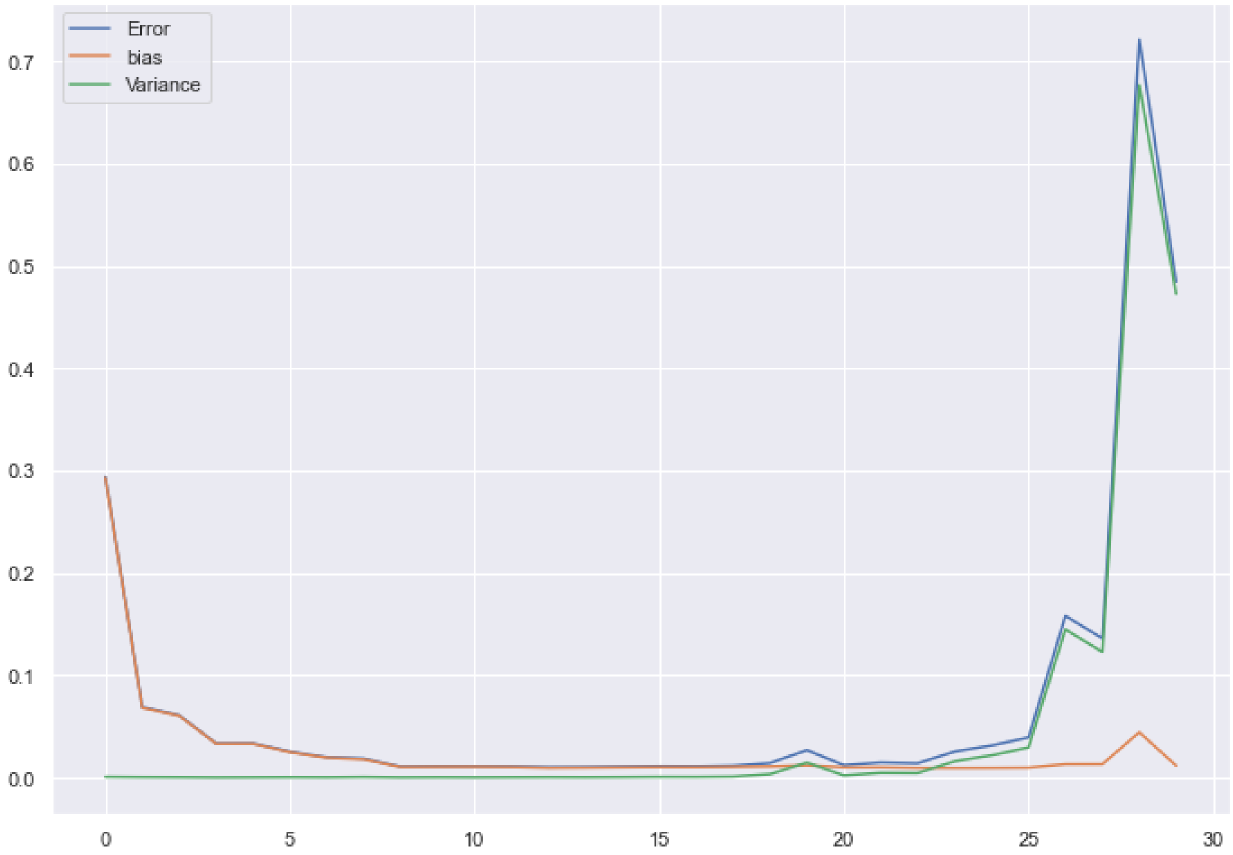
(continued from previous page)

```
0.010972468815261078 >= 0.010593565969983903 + 0.00037890284527716995 = 0.
↪010972468815261073
Polynomial degree: 11
Error: 0.01084055593776807
Bias^2: 0.010348475861989281
Var: 0.0004920800757787882
0.01084055593776807 >= 0.010348475861989281 + 0.0004920800757787882 = 0.
↪01084055593776807
Polynomial degree: 12
Error: 0.010192472149429362
Bias^2: 0.009610568640072627
Var: 0.0005819035093567355
0.010192472149429362 >= 0.009610568640072627 + 0.0005819035093567355 = 0.
↪010192472149429362
Polynomial degree: 13
Error: 0.010312285920590011
Bias^2: 0.009802534263801815
Var: 0.0005097516567881938
0.010312285920590011 >= 0.009802534263801815 + 0.0005097516567881938 = 0.
↪01031228592059001
Polynomial degree: 14
Error: 0.010722455299595876
Bias^2: 0.01008891676024437
Var: 0.0006335385393515036
0.010722455299595876 >= 0.01008891676024437 + 0.0006335385393515036 = 0.
↪010722455299595875
Polynomial degree: 15
Error: 0.011155437503231998
Bias^2: 0.010311761228670724
Var: 0.0008436762745612778
0.011155437503231998 >= 0.010311761228670724 + 0.0008436762745612778 = 0.
↪011155437503232002
Polynomial degree: 16
Error: 0.011028026782676708
Bias^2: 0.010223572382311492
Var: 0.0008044544003652116
0.011028026782676708 >= 0.010223572382311492 + 0.0008044544003652116 = 0.
↪011028026782676703
Polynomial degree: 17
Error: 0.011628743129658555
Bias^2: 0.010533948734129592
Var: 0.001094794395528961
0.011628743129658555 >= 0.010533948734129592 + 0.001094794395528961 = 0.
↪011628743129658553
Polynomial degree: 18
Error: 0.014371682171531027
Bias^2: 0.010922362242870073
Var: 0.0034493199286609573
0.014371682171531027 >= 0.010922362242870073 + 0.0034493199286609573 = 0.
↪01437168217153103
Polynomial degree: 19
Error: 0.026986306199342624
Bias^2: 0.01214176442858653
Var: 0.014844541770756087
0.026986306199342624 >= 0.01214176442858653 + 0.014844541770756087 = 0.
↪026986306199342617
Polynomial degree: 20
```

(continues on next page)

(continued from previous page)

```
Error: 0.012249244024160728
Bias^2: 0.01006785246285396
Var: 0.002181391561306766
0.012249244024160728 >= 0.01006785246285396 + 0.002181391561306766 = 0.
↪012249244024160727
Polynomial degree: 21
Error: 0.014973172820830053
Bias^2: 0.010154371176360328
Var: 0.00481880164446972
0.014973172820830053 >= 0.010154371176360328 + 0.00481880164446972 = 0.
↪014973172820830048
Polynomial degree: 22
Error: 0.014186606932681737
Bias^2: 0.009594131981212376
Var: 0.0045924749514693625
0.014186606932681737 >= 0.009594131981212376 + 0.0045924749514693625 = 0.
↪014186606932681738
Polynomial degree: 23
Error: 0.025574552577788824
Bias^2: 0.009477519033249752
Var: 0.016097033544539077
0.025574552577788824 >= 0.009477519033249752 + 0.016097033544539077 = 0.
↪02557455257778883
Polynomial degree: 24
Error: 0.03147298632679604
Bias^2: 0.009565267585507206
Var: 0.021907718741288846
0.03147298632679604 >= 0.009565267585507206 + 0.021907718741288846 = 0.
↪03147298632679605
Polynomial degree: 25
Error: 0.03929027799369515
Bias^2: 0.009776269005896726
Var: 0.029514008987798424
0.03929027799369515 >= 0.009776269005896726 + 0.029514008987798424 = 0.
↪03929027799369515
Polynomial degree: 26
Error: 0.15813256009613183
Bias^2: 0.013239726753028333
Var: 0.14489283334310352
0.15813256009613183 >= 0.013239726753028333 + 0.14489283334310352 = 0.
↪15813256009613186
Polynomial degree: 27
Error: 0.1360840943498259
Bias^2: 0.013266085921451169
Var: 0.12281800842837416
0.1360840943498259 >= 0.013266085921451169 + 0.12281800842837416 = 0.13608409434982585
Polynomial degree: 28
Error: 0.7210723692205014
Bias^2: 0.04436186918146108
Var: 0.6767105000390408
0.7210723692205014 >= 0.04436186918146108 + 0.6767105000390408 = 0.7210723692205019
Polynomial degree: 29
Error: 0.48454430745837984
Bias^2: 0.011809368338879722
Var: 0.4727349391195001
0.48454430745837984 >= 0.011809368338879722 + 0.4727349391195001 = 0.48454430745837984
```



13.16.82 Summing up

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

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

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

You may also find this recent article of interest.

13.16.83 Another Example from Scikit-Learn's Repository

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

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

print(__doc__)

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


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

np.random.seed(0)

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

13.16.84 More examples on bootstrap and cross-validation and errors

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

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

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

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

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

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

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

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

# Read the EoS data as  csv file and organize the data into two arrays with density and energies
```

(continues on next page)

(continued from previous page)

```
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
Density = EoS['Density']
#  The design matrix now as function of various polytrops

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

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

# loop over trials in order to estimate the expectation value of the MSE
    testerror[polydegree] = 0.0
    trainingerror[polydegree] = 0.0
    for samples in range(trials):
        x_train, x_test, y_train, y_test = train_test_split(X, Energies, test_size=0.
↪2)
        model = LinearRegression(fit_intercept=True).fit(x_train, y_train)
        ypred = model.predict(x_train)
        ytilde = model.predict(x_test)
        testerror[polydegree] += mean_squared_error(y_test, ytilde)
        trainingerror[polydegree] += mean_squared_error(y_train, ypred)

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

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

13.16.85 The same example but now with cross-validation

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

(continues on next page)

(continued from previous page)

```
from sklearn.model_selection import cross_val_score


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

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

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

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

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

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

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

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

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

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

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

plt.plot(polynomial, np.log10(estimated_mse_sklearn), label='Test Error')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Polynomial degree')
plt.ylabel('log10[MSE]')
plt.legend()
plt.show()
```

13.16.86 Cross-validation with Ridge

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

# A seed just to ensure that the random numbers are the same for every run.
np.random.seed(3155)
# Generate the data.
n = 100
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)
# Decide degree on polynomial to fit
poly = PolynomialFeatures(degree = 10)

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

13.16.87 The Ising model

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

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

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

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

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

L = 40
n = int(1e4)

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

energies = np.zeros(n)

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

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

13.16.88 Reformulating the problem to suit regression

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

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

Here we allow for interactions beyond the nearest neighbors and a state dependent coupling constant. This latter expression can be formulated as a matrix-product

$$\boldsymbol{H} = \boldsymbol{X}J,$$

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

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

We split the data in training and test data as discussed in the previous example

```
X = np.zeros((n, L ** 2))
for i in range(n):
    X[i] = np.outer(spins[i], spins[i]).ravel()
y = energies
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

13.16.89 Linear regression

In the ordinary least squares method we choose the cost function

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

We then find the extremal point of C by taking the derivative with respect to $\boldsymbol{\beta}$ as discussed above. This yields the expression for $\boldsymbol{\beta}$ to be

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

which immediately imposes some requirements on $\boldsymbol{X}$ as there must exist an inverse of $\boldsymbol{X}^T\boldsymbol{X}$. If the expression we are modeling contains an intercept, i.e., a constant term, we must make sure that the first column of $\boldsymbol{X}$ consists of 1. We do this here

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

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

13.16.90 Singular Value decomposition

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

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

where the pseudoinverse of $\boldsymbol{X}$ is given by

$$\boldsymbol{X}^+ = \frac{\boldsymbol{X}^T}{\boldsymbol{X}^T\boldsymbol{X}}.$$

Using singular value decomposition we can decompose the matrix $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal(unitary) matrices and $\boldsymbol{\Sigma}$ contains the singular values (more details below). where $X^+ = V\Sigma^+U^T$. This reduces the equation for ω to

$$\boldsymbol{\beta} = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^T\boldsymbol{y}.$$

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

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

```
beta = ols_svd(X_train_own,y_train)
```

When extracting the J-matrix we need to make sure that we remove the intercept, as is done here

```
J = beta[1:].reshape(L, L)
```

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

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

It is interesting to note that OLS considers both $J_{j,j+1} = -0.5$ and $J_{j,j-1} = -0.5$ as valid matrix elements for J. In our discussion below on hyperparameters and Ridge and Lasso regression we will see that this problem can be removed, partly and only with Lasso regression.

In this case our matrix inversion was actually possible. The obvious question now is what is the mathematics behind the SVD?

13.16.91 The one-dimensional Ising model

Let us bring back the Ising model again, but now with an additional focus on Ridge and Lasso regression as well. We repeat some of the basic parts of the Ising model and the setup of the training and test data. The one-dimensional Ising model with nearest neighbor interaction, no external field and a constant coupling constant J is given by

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

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

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

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

L = 40
n = int(1e4)

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

energies = np.zeros(n)
```

(continues on next page)

(continued from previous page)

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

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

$$H = - \sum_j^L \sum_k^L s_j s_k J_{jk}.$$

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

$$H = XJ,$$

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

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

We organize the data as we did above

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

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

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

We will do all fitting with **Scikit-Learn**,

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

When extracting the J-matrix we make sure to remove the intercept

```
J_sk = clf.coef_.reshape(L, L)
```

And then we plot the results

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

The results perfectly with our previous discussion where we used our own code.

13.16.92 Ridge regression

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

136

<<<!!MATH_BLOCK

```
_lambda = 0.1
clf_ridge = skl.Ridge(alpha=_lambda).fit(X_train, y_train)
J_ridge_sk = clf_ridge.coef_.reshape(L, L)
fig = plt.figure(figsize=(20, 14))
im = plt.imshow(J_ridge_sk, **cmap_args)
plt.title("Ridge from Scikit-learn", fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
cb = fig.colorbar(im)
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

plt.show()
```

13.16.93 LASSO regression

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

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

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

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

plt.show()
```

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

13.16.94 Performance as function of the regularization parameter

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

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

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

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

plot_counter = 1

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

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

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

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

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

plt.show()
```

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

13.16.95 Finding the optimal value of $\lambda$

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

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

colors = {
    "ols_sk": "r",
    "ridge_sk": "y",
    "lasso_sk": "c"
}

for key in train_errors:
    plt.semilogx(
        lambdas,
        train_errors[key],
        colors[key],
        label="Train {0}".format(key),
        linewidth=4.0
    )

for key in test_errors:
    plt.semilogx(
        lambdas,
        test_errors[key],
        colors[key] + "--",
        label="Test {0}".format(key),
        linewidth=4.0
    )
plt.legend(loc="best", fontsize=18)
plt.xlabel(r"$\lambda$", fontsize=18)
plt.ylabel(r"$R^2$", fontsize=18)
plt.tick_params(labelsize=18)
plt.show()
```

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

13.17 Data Analysis and Machine Learning: Logistic Regression

Morten Hjorth-Jensen, Department of Physics, University of Oslo and Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Date: **Oct 17, 2019**

13.17.1 Logistic Regression

In linear regression our main interest was centered on learning the coefficients of a functional fit (say a polynomial) in order to be able to predict the response of a continuous variable on some unseen data. The fit to the continuous variable y_i is based on some independent variables $\hat{x}_i$. Linear regression resulted in analytical expressions for standard ordinary Least Squares or Ridge regression (in terms of matrices to invert) for several quantities, ranging from the variance and thereby the confidence intervals of the parameters $\hat{\beta}$ to the mean squared error. If we can invert the product of the design matrices, linear regression gives then a simple recipe for fitting our data.

Classification problems, however, are concerned with outcomes taking the form of discrete variables (i.e. categories). We may for example, on the basis of DNA sequencing for a number of patients, like to find out which mutations are important for a certain disease; or based on scans of various patients' brains, figure out if there is a tumor or not; or given a specific physical system, we'd like to identify its state, say whether it is an ordered or disordered system (typical situation in solid state physics); or classify the status of a patient, whether she/he has a stroke or not and many other similar situations.

The most common situation we encounter when we apply logistic regression is that of two possible outcomes, normally denoted as a binary outcome, true or false, positive or negative, success or failure etc.

13.17.2 Optimization and Deep learning

Logistic regression will also serve as our stepping stone towards neural network algorithms and supervised deep learning. For logistic learning, the minimization of the cost function leads to a non-linear equation in the parameters $\hat{\beta}$. The optimization of the problem calls therefore for minimization algorithms. This forms the bottle neck of all machine learning algorithms, namely how to find reliable minima of a multi-variable function. This leads us to the family of gradient descent methods. The latter are the working horses of basically all modern machine learning algorithms.

We note also that many of the topics discussed here on logistic regression are also commonly used in modern supervised Deep Learning models, as we will see later.

13.17.3 Basics

We consider the case where the dependent variables, also called the responses or the outcomes, y_i are discrete and only take values from $k = 0, \dots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $\hat{X} \in \mathbb{R}^{n\times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and $y_i = 1$. Our outcomes could represent the status of a credit card user that could default or not on her/his credit card debt. That is

$$y_i = \begin{bmatrix} 0 & \mathrm{no} \\ 1 & \mathrm{yes} \end{bmatrix}.$$

13.17.4 Linear classifier

Before moving to the logistic model, let us try to use our linear regression model to classify these two outcomes. We could for example fit a linear model to the default case if $y_i > 0.5$ and the no default case $y_i \leq 0.5$.

We would then have our weighted linear combination, namely

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

where $\hat{y}$ is a vector representing the possible outcomes, $\hat{X}$ is our $n \times p$ design matrix and $\hat{\beta}$ represents our estimators/predictors.

13.17.5 Some selected properties

The main problem with our function is that it takes values on the entire real axis. In the case of logistic regression, however, the labels y_i are discrete variables. A typical example is the credit card data discussed below here, where we can set the state of defaulting the debt to $y_i = 1$ and not to $y_i = 0$ for one the persons in the data set (see the full example below).

One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to values $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 if otherwise. We will encounter this model in our first demonstration of neural networks. Historically it is called the “perceptron” model in the machine learning literature. This model is extremely simple. However, in many cases it is more favorable to use a ``soft” classifier that outputs the probability of a given category. This leads us to the logistic function.

13.17.6 The logistic function

The perceptron is an example of a ``hard classification” model. We will encounter this model when we discuss neural networks as well. Each datapoint is deterministically assigned to a category (i.e $y_i = 0$ or $y_i = 1$). In many cases, it is favorable to have a “soft” classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category k. Logistic regression is the most common example of a so-called soft classifier. In logistic regression, the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so-called logit function (or Sigmoid) which is meant to represent the likelihood for a given event,

$$p(t) = \frac{1}{1 + \exp{-t}} = \frac{\exp{t}}{1 + \mathrm{exp}t}.$$

Note that $1 - p(t) = p(-t)$.

13.17.7 Examples of likelihood functions used in logistic regression and nueral networks

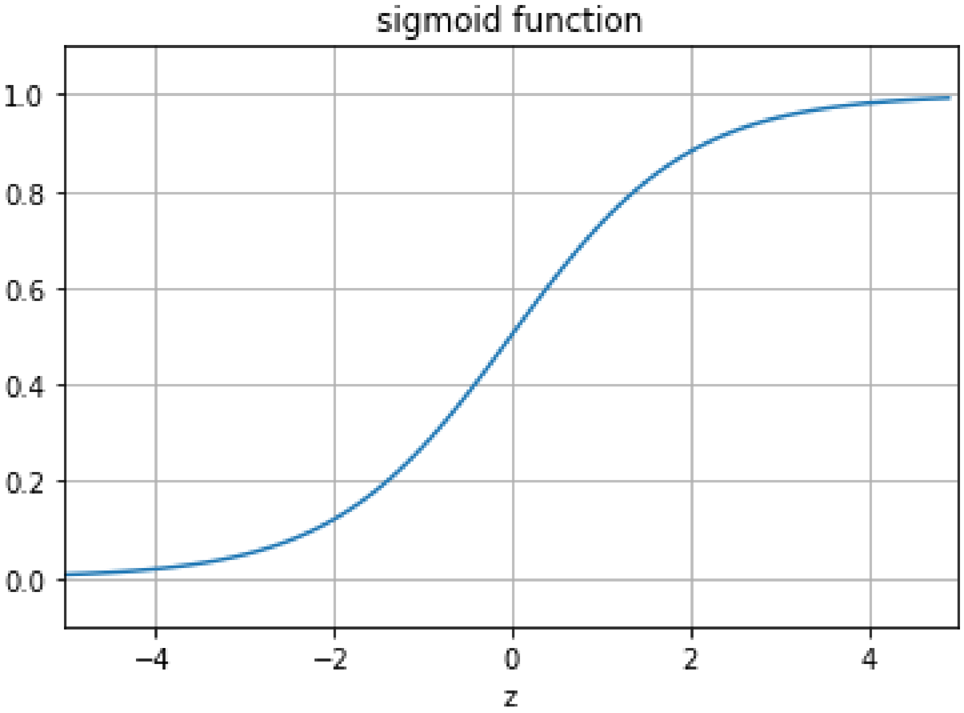
The following code plots the logistic function, the step function and other functions we will encounter from here and on.

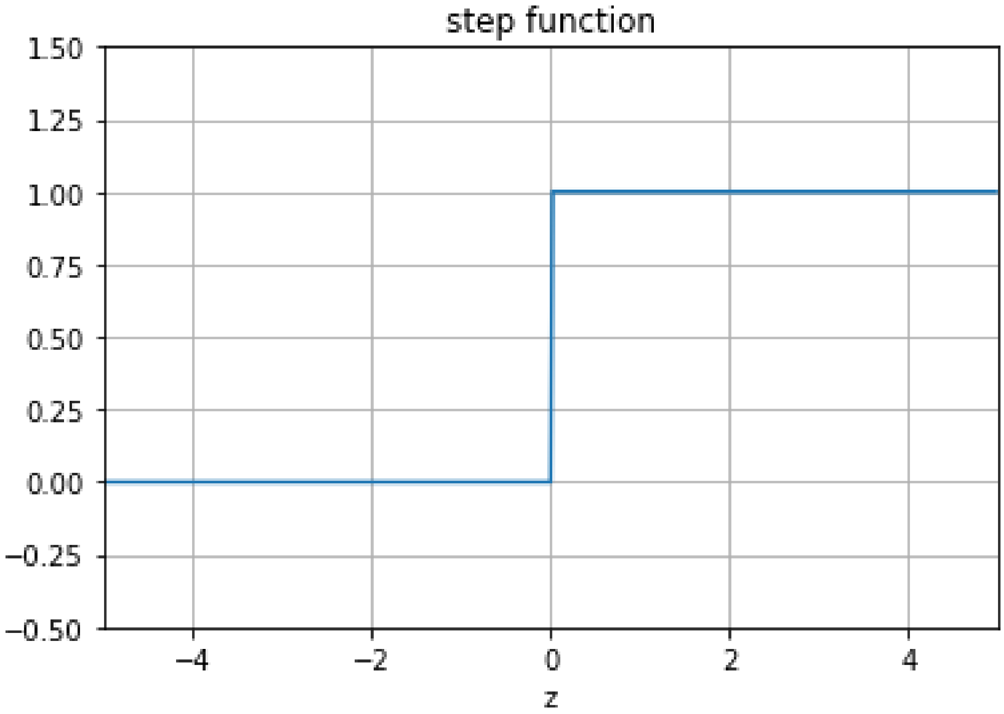
```
%matplotlib inline

"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative scale.
The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')

plt.show()

"""Step Function"""
z = numpy.arange(-5, 5, .02)
step_fn = numpy.vectorize(lambda z: 1.0 if z >= 0.0 else 0.0)
step = step_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, step)
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('step function')

plt.show()

"""tanh Function"""
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.tanh(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, t)
ax.set_ylim([-1.0, 1.0])
ax.set_xlim([-2*mt.pi,2*mt.pi])
```

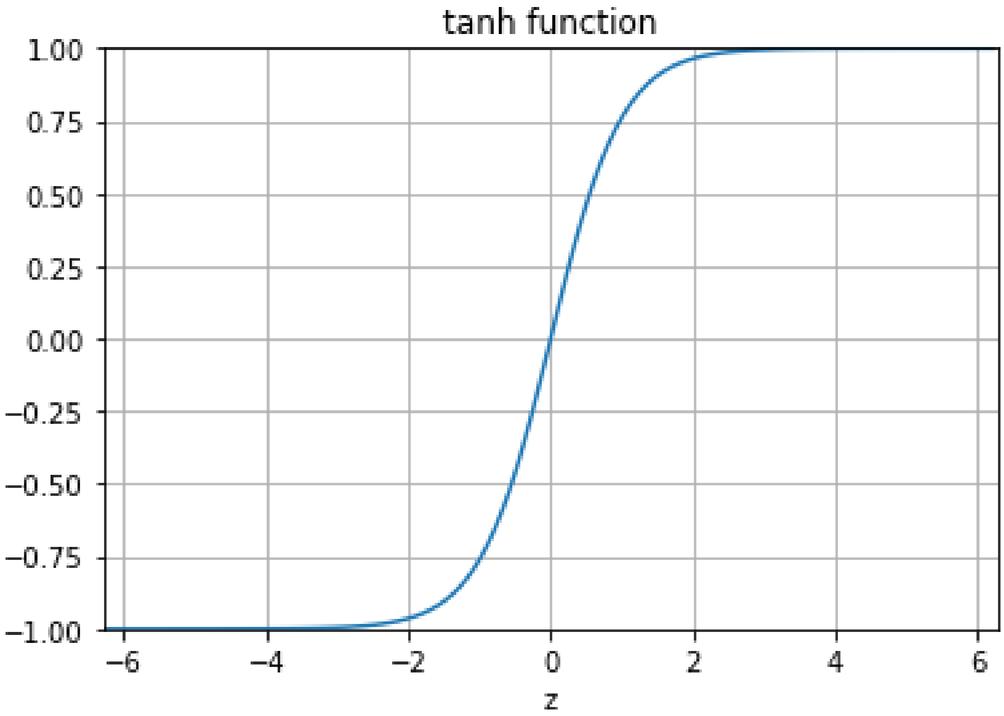
(continues on next page)

(continued from previous page)

```
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('tanh function')

plt.show()
```







13.17.8 Two parameters

We assume now that we have two classes with y_i either 0 or 1. Furthermore we assume also that we have only two parameters β in our fitting of the Sigmoid function, that is we define probabilities

$$\begin{aligned} p(y_i = 1|x_i,\hat{\beta}) &= \frac{\exp{(\beta_0+\beta_1x_i)}}{1+\exp{(\beta_0+\beta_1x_i)}},\\ p(y_i = 0|x_i,\hat{\beta}) &= 1 - p(y_i = 1|x_i,\hat{\beta}), \end{aligned}$$

where $\hat{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

Note that we used

$$p(y_i = 0|x_i,\hat{\beta}) = 1 - p(y_i = 1|x_i,\hat{\beta}).$$

13.17.9 Maximum likelihood

In order to define the total likelihood for all possible outcomes from adataset $\mathcal{D}=\{(y_i,x_i)\}$, with the binary labels $y_i\in\{0,1\}$ and where the data points are drawn independently, we use the so-called Maximum Likelihood Estimation (MLE) principle. We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is

$$P(\mathcal{D}|\hat{\beta})= \prod_{i=1}^n \left[p(y_i=1|x_i,\hat{\beta})\right]^{y_i}\left[1-p(y_i=1|x_i,\hat{\beta}))\right]^{1-y_i}$$

from which we obtain the log-likelihood and our **cost/loss** function

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^n \left(y_i\log{p(y_i=1|x_i,\hat{\beta})} + (1-y_i)\log\left[1-p(y_i=1|x_i,\hat{\beta}))\right]\right).$$

13.17.10 The cost function rewritten

Reordering the logarithms, we can rewrite the **cost/loss** function as

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i(\beta_0+\beta_1 x_i) - \log{(1+\exp{(\beta_0+\beta_1 x_i)})}\right).$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. Since the cost (error) function is just the negative log-likelihood, for logistic regression we have that

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0+\beta_1 x_i) - \log{(1+\exp{(\beta_0+\beta_1 x_i)})}\right).$$

This equation is known in statistics as the **cross entropy**. Finally, we note that just as in linear regression, in practice we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization as we did for Ridge and Lasso regression.

13.17.11 Minimizing the cross entropy

The cross entropy is a convex function of the weights $\hat{\beta}$ and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to the two parameters β_0 and β_1 we obtain

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp{(\beta_0+\beta_1 x_i)}}{1+\exp{(\beta_0+\beta_1 x_i)}}\right),$$

and

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i\frac{\exp{(\beta_0+\beta_1 x_i)}}{1+\exp{(\beta_0+\beta_1 x_i)}}\right).$$

13.17.12 A more compact expression

Let us now define a vector $\hat{y}$ with n elements y_i, an $n\times p$ matrix $\hat{X}$ which contains the x_i values and a vector $\hat{p}$ of fitted probabilities $p(y_i\vert x_i,\hat{\beta})$. We can rewrite in a more compact form the first derivative of cost function as

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T\left(\hat{y}-\hat{p}\right).$$

If we in addition define a diagonal matrix $\hat{W}$ with elements $p(y_i\vert x_i,\hat{\beta})(1-p(y_i\vert x_i,\hat{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}\partial \hat{\beta}^T} = \hat{X}^T\hat{W}\hat{X}.$$

13.17.13 Extending to more predictors

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log\frac{p(\hat{\beta}\hat{x})}{1-p(\hat{\beta}\hat{x})} = \beta_0+\beta_1x_1+\beta_2x_2+\dots+\beta_px_p.$$

Here we defined $\hat{x}=[1,x_1,x_2,\dots,x_p]$ and $\hat{\beta}=[\beta_0,\beta_1,\dots,\beta_p]$ leading to

$$p(\hat{\beta}\hat{x})=\frac{\exp{(\beta_0+\beta_1x_1+\beta_2x_2+\dots+\beta_px_p)}}{1+\exp{(\beta_0+\beta_1x_1+\beta_2x_2+\dots+\beta_px_p)}}.$$

13.17.14 Including more classes

Till now we have mainly focused on two classes, the so-called binary system. Suppose we wish to extend to K classes. Let us for the sake of simplicity assume we have only two predictors. We have then following model

1 5

<<<!!MATH_BLOCK

$$\log\frac{p(C=2\vert x)}{p(K\vert x)} = \beta_{20}+\beta_{21}x_1,$$

and so on till the class $C=K-1$ class

$$\log\frac{p(C=K-1\vert x)}{p(K\vert x)} = \beta_{(K-1)0}+\beta_{(K-1)1}x_1,$$

and the model is specified in term of $K-1$ so-called log-odds or **logit** transformations.

13.17.15 More classes

In our discussion of neural networks we will encounter the above again in terms of a slightly modified function, the so-called **Softmax** function.

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression (also known as softmax regression), multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks. Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the predicted probability for the k-th class given a sample vector $\hat{x}$ and a weighting vector $\hat{\beta}$ is (with two predictors):

$$p(C=k\vert \mathbf{x})=\frac{\exp{(\beta_{k0}+\beta_{k1}x_1)}}{1+\sum_{l=1}^{K-1}\exp{(\beta_{l0}+\beta_{l1}x_1)}}.$$

It is easy to extend to more predictors. The final class is

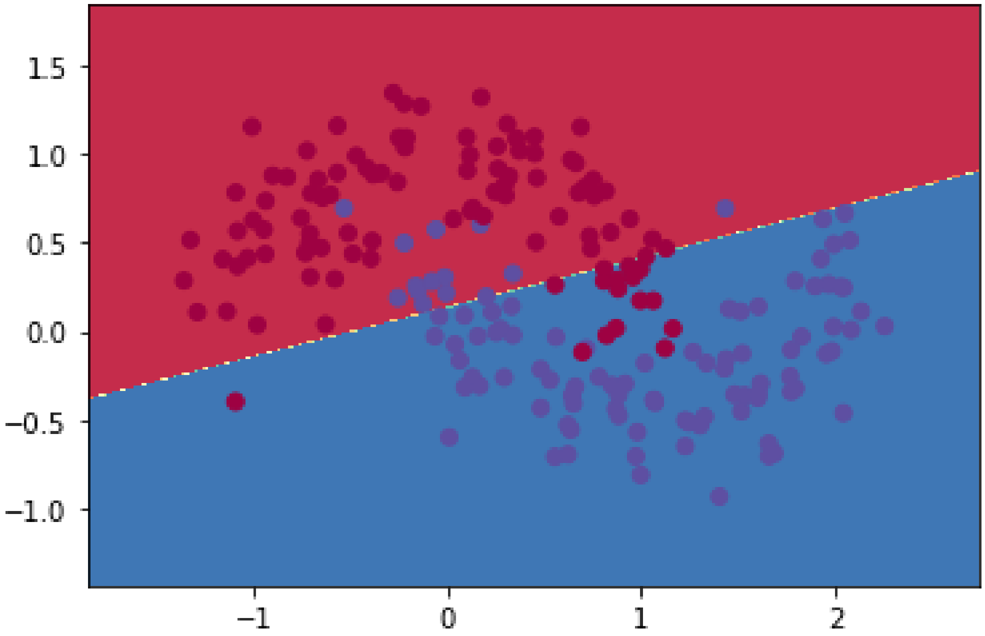
$$p(C=K\vert \mathbf{x})=\frac{1}{1+\sum_{l=1}^{K-1}\exp{(\beta_{l0}+\beta_{l1}x_1)}},$$

and they sum to one. Our earlier discussions were all specialized to the case with two classes only. It is easy to see from the above that what we derived earlier is compatible with these equations.

To find the optimal parameters we would typically use a gradient descent method. Newton's method and gradient descent methods are discussed in the material on optimization methods.

13.17.16 A simple classification problem

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


def generate_data():
    np.random.seed(0)
    X, y = datasets.make_moons(200, noise=0.20)
    return X, y


def visualize(X, y, clf):
    plot_decision_boundary(lambda x: clf.predict(x), X, y)

def plot_decision_boundary(pred_func, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
    plt.show()


def classify(X, y):
    clf = linear_model.LogisticRegressionCV()
    clf.fit(X, y)
    return clf


def main():
    X, y = generate_data()
    # visualize(X, y)
    clf = classify(X, y)
    visualize(X, y, clf)

if __name__ == "__main__":
    main()
```



13.17.17 The Credit Card example

Here we use the the credit card data. The data are from an extensive database from Taiwan and include more than ten predictors.

For categorical data -Scikit-Learn- provides a so-called **one-hot encoder**. This is called one-hot encoding, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). **Scikit-Learn** provides a OneHotEncoder encoder to convert integer categorical values into one-hot

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
```

13.17.18 How to read the Credit Card data

```
import pandas as pd
import os
import numpy as np


from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score

# Trying to set the seed
np.random.seed(0)
import random
random.seed(0)

# Reading file into data frame
```

(continues on next page)

(continued from previous page)

```
cwd = os.getcwd()
filename = cwd + '/default of credit card clients.xls'
nanDict = {}
df = pd.read_excel(filename, header=1, skiprows=0, index_col=0, na_values=nanDict)

df.rename(index=str, columns={"default payment next month": "defaultPaymentNextMonth"}
↪, inplace=True)

# Features and targets
X = df.loc[:, df.columns != 'defaultPaymentNextMonth'].values
y = df.loc[:, df.columns == 'defaultPaymentNextMonth'].values

# Categorical variables to one-hot's
onehotencoder = OneHotEncoder(categories="auto")

X = ColumnTransformer(
    [("", onehotencoder, [3]),],
    remainder="passthrough"
).fit_transform(X)

y.shape

# Train-test split
trainingShare = 0.5
seed  = 1
XTrain, XTest, yTrain, yTest=train_test_split(X, y, train_size=trainingShare, \
                                              test_size = 1-trainingShare,
                                             random_state=seed)

# Input Scaling
sc = StandardScaler()
XTrain = sc.fit_transform(XTrain)
XTest = sc.transform(XTest)

# One-hot's of the target vector
Y_train_onehot, Y_test_onehot = onehotencoder.fit_transform(yTrain), onehotencoder.
↪fit_transform(yTest)

# Remove instances with zeros only for past bill statements or paid amounts
'''
df = df.drop(df[(df.BILL_AMT1 == 0) &
                (df.BILL_AMT2 == 0) &
                (df.BILL_AMT3 == 0) &
                (df.BILL_AMT4 == 0) &
                (df.BILL_AMT5 == 0) &
                (df.BILL_AMT6 == 0) &
                (df.PAY_AMT1 == 0) &
                (df.PAY_AMT2 == 0) &
                (df.PAY_AMT3 == 0) &
                (df.PAY_AMT4 == 0) &
                (df.PAY_AMT5 == 0) &
                (df.PAY_AMT6 == 0)].index)
'''
df = df.drop(df[(df.BILL_AMT1 == 0) &
                (df.BILL_AMT2 == 0) &
                (df.BILL_AMT3 == 0) &
                (df.BILL_AMT4 == 0) &
```

(continues on next page)

(continued from previous page)

```
                (df.BILL_AMT5 == 0) &
                (df.BILL_AMT6 == 0)].index)

df = df.drop(df[(df.PAY_AMT1 == 0) &
                (df.PAY_AMT2 == 0) &
                (df.PAY_AMT3 == 0) &
                (df.PAY_AMT4 == 0) &
                (df.PAY_AMT5 == 0) &
                (df.PAY_AMT6 == 0)].index)

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

lambdas=np.logspace(-5,7,13)
parameters = [{'C': 1./lambdas, "solver":["lbfgs"]}]#*len(parameters)}]
scoring = ['accuracy', 'roc_auc']
logReg = LogisticRegression()
gridSearch = GridSearchCV(logReg, parameters, cv=5, scoring=scoring, refit='roc_auc')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-4-4edad39a9543> in <module>
     19 filename = cwd + '/default of credit card clients.xls'
     20 nanDict = {}
---> 21 df = pd.read_excel(filename, header=1, skiprows=0, index_col=0, na_
↪values=nanDict)
     22 
     23 df.rename(index=str, columns={"default payment next month":
↪"defaultPaymentNextMonth"}, inplace=True)

~/opt/anaconda3/lib/python3.8/site-packages/pandas/io/excel/_base.py in read_excel(io,
↪ sheet_name, header, names, index_col, usecols, squeeze, dtype, engine, converters,␣
↪true_values, false_values, skiprows, nrows, na_values, keep_default_na, verbose,␣
↪parse_dates, date_parser, thousands, comment, skipfooter, convert_float, mangle_
↪dupe_cols, **kwds)
    302 
    303     if not isinstance(io, ExcelFile):
--> 304         io = ExcelFile(io, engine=engine)
    305     elif engine and engine != io.engine:
    306         raise ValueError(

~/opt/anaconda3/lib/python3.8/site-packages/pandas/io/excel/_base.py in __init__(self,
↪ io, engine)
    822         self._io = stringify_path(io)
    823 
--> 824         self._reader = self._engines[engine](self._io)
    825 
    826     def __fspath__(self):

~/opt/anaconda3/lib/python3.8/site-packages/pandas/io/excel/_xlrd.py in __init__(self,
↪ filepath_or_buffer)
     19         err_msg = "Install xlrd >= 1.0.0 for Excel support"
     20         import_optional_dependency("xlrd", extra=err_msg)
---> 21         super().__init__(filepath_or_buffer)
     22 
     23     @property
```

(continues on next page)

(continued from previous page)

```
~/opt/anaconda3/lib/python3.8/site-packages/pandas/io/excel/_base.py in __init__(self, filepath_or_buffer)
    351                 self.book = self.load_workbook(filepath_or_buffer)
    352             elif isinstance(filepath_or_buffer, str):
--> 353                 self.book = self.load_workbook(filepath_or_buffer)
    354             elif isinstance(filepath_or_buffer, bytes):
    355                 self.book = self.load_workbook(BytesIO(filepath_or_buffer))

~/opt/anaconda3/lib/python3.8/site-packages/pandas/io/excel/_xlrd.py in load_workbook(self, filepath_or_buffer)
     34                 return open_workbook(file_contents=data)
     35             else:
---> 36                 return open_workbook(filepath_or_buffer)
     37 
     38     @property

~/opt/anaconda3/lib/python3.8/site-packages/xlrd/__init__.py in open_workbook(filename, logfile, verbosity, use_mmap, file_contents, encoding_override, formatting_info, on_demand, ragged_rows)
    109     else:
    110         filename = os.path.expanduser(filename)
--> 111         with open(filename, "rb") as f:
    112             peek = f.read(peeksz)
    113     if peek == b"PK\x03\x04": # a ZIP file

FileNotFoundError: [Errno 2] No such file or directory: '/Users/hjensen/Teaching/FYS-STK4150/doc/src/LectureNotes/default of credit card clients.xls'
```