Applied Machine Learning and Data Analysis

Morten Hjorth-Jensen

Dec 20, 2020

SUPERVISED LEARNING

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

ONE

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

TWO

MACHINE LEARNING, A SMALL (AND PROBABLY BIASED) INTRODUCTION

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

Machine learning 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.

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.

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.

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.

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

THREE

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

FOUR

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*.

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

FIVE

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

SIX

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.

6.1 Elements of Probability Theory and Statistical Data Analysis

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

6.1.1 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*

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

6.1.2 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

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

6.1.3 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).

6.1.4 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

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

6.1.5 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.

6.1.6 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

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

6.1.7 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

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

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

6.1.8 Important distributions, the uniform distribution

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

$$p(x) = \frac{1}{b-a}\theta(x-a)\theta(b-x). \quad (6.1)$$

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.

6.1.9 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**

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

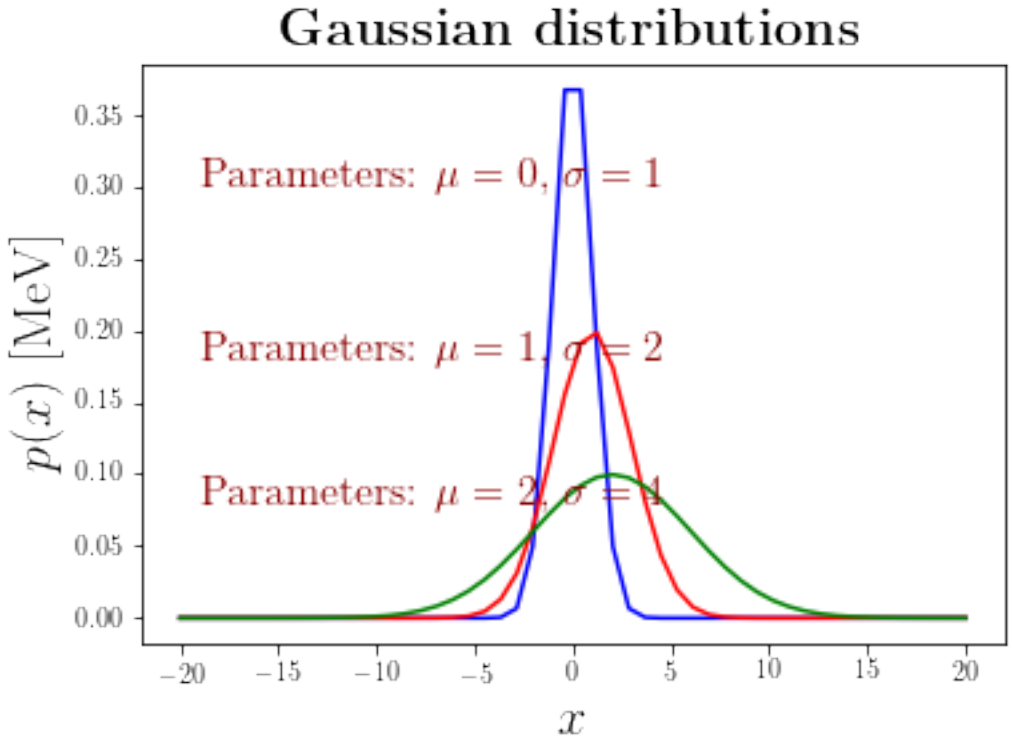
The following simple Python code plots the above distribution for different values of μ and σ.

```
%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()
```



6.1.10 Exponential distribution

Another important distribution in science is the exponential distribution

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

6.1.11 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

$$\langle h \rangle_X \equiv \int h(x)p(x)\,dx \quad (6.2)$$

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

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

6.1.12 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 $p(x)$.

6.1.13 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_X^2 or

$\mathrm{Var}(X)$

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

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.

6.1.14 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.

6.1.15 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.

6.1.16 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

$$p(x) = \alpha \exp\left(-\alpha x\right),$$

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

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

with variance

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

6.1.17 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

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

6.1.18 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

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right). \quad (6.3)$$

6.1.19 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.

6.1.20 Probability Distribution Functions, other important distribution

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

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

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 $1/2$.

6.1.21 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 $\sigma^2 = ny(1-y)$.

6.1.22 Probability Distribution Functions, Poisson's distribution

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

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

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 $\sigma^2 = \lambda$.

6.1.23 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

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

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

$$\mathrm{Cov}(X_i,\, X_j) = \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \quad (6.4)$$

$$= \int \cdots \int (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) P(x_1, \ldots, x_n)\, dx_1 \ldots dx_n, \quad (6.5)$$

with

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

6.1.25 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.

6.1.26 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)
```

```
2.56416115970942
[[ 5.90838144e+00  6.98010863e+00 -1.93947492e-01  1.33851317e+00
   8.51355225e+00  8.12403403e+00  4.82725395e+00  7.54223480e+00
   9.38189254e+00  5.58473257e+00]
 [ 6.98010863e+00  8.24623749e+00 -2.29127820e-01  1.58130740e+00
   1.00578340e+01  9.59766065e+00  5.70287435e+00  8.91032829e+00
   1.10836834e+01  6.59775278e+00]
 [-1.93947492e-01 -2.29127820e-01  6.36648631e-03 -4.39377982e-02
  -2.79464371e-01 -2.66678115e-01 -1.58458590e-01 -2.47580075e-01
  -3.07968357e-01 -1.83323451e-01]
 [ 1.33851317e+00  1.58130740e+00 -4.39377982e-02  3.03233215e-01
   1.92870110e+00  1.84045777e+00  1.09358934e+00  1.70865417e+00
   2.12541909e+00  1.26519220e+00]
 [ 8.51355225e+00  1.00578340e+01 -2.79464371e-01  1.92870110e+00
   1.22674158e+01  1.17061481e+01  6.95572537e+00  1.08678173e+01
   1.35186316e+01  8.04719753e+00]
 [ 8.12403403e+00  9.59766065e+00 -2.66678115e-01  1.84045777e+00
   1.17061481e+01  1.11705599e+01  6.63748198e+00  1.03705850e+01
   1.29001174e+01  7.67901631e+00]
 [ 4.82725395e+00  5.70287435e+00 -1.58458590e-01  1.09358934e+00
   6.95572537e+00  6.63748198e+00  3.94395333e+00  6.16214154e+00
   7.66517501e+00  4.56282700e+00]
 [ 7.54223480e+00  8.91032829e+00 -2.47580075e-01  1.70865417e+00
   1.08678173e+01  1.03705850e+01  6.16214154e+00  9.62790003e+00
   1.19762810e+01  7.12908684e+00]
 [ 9.38189254e+00  1.10836834e+01 -3.07968357e-01  2.12541909e+00
   1.35186316e+01  1.29001174e+01  7.66517501e+00  1.19762810e+01
   1.48974653e+01  8.86797194e+00]
 [ 5.58473257e+00  6.59775278e+00 -1.83323451e-01  1.26519220e+00
   8.04719753e+00  7.67901631e+00  4.56282700e+00  7.12908684e+00
```

(continues on next page)

(continued from previous page)

```
   8.86797194e+00  5.27881252e+00]]
```

6.1.27 Meet the covariance, uncorrelated events

Consider the stochastic variables X_i and X_j, $(i \neq j)$. 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 $i \neq j$), we have that

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

leading to

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

6.1.28 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

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

6.1.29 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.

6.1.30 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*.

6.1.31 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

$$\langle X_m\rangle = \frac{1}{m}\sum_{\alpha=1}^{m}x_\alpha = \frac{1}{mn}\sum_{\alpha,k}x_{\alpha,k}, \quad (6.6)$$

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

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k}-\langle X_m\rangle)(x_{\alpha,l}-\langle X_m\rangle). \quad (6.7)$$

6.1.32 Numerical experiments and the covariance, the sample variance

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

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n}(x_{\alpha,k}-\langle X_m\rangle)^2. \quad (6.8)$$

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 $\mathrm{Cov}(X,Y)$.

6.1.33 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.

6.1.34 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 $k \neq l$.

6.1.35 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.

6.1.36 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

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

6.1.37 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**

$$\kappa_d = \frac{f_d}{\sigma^2} \tag{6.9}$$

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

6.1.38 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

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{n} \cdot \sigma^2 \sum_{d=1}^{n-1} \frac{f_d}{\sigma^2} = \left(1 + 2\sum_{d=1}^{n-1} \kappa_d\right) \frac{1}{n}\sigma^2 = \frac{\tau}{n} \cdot \sigma^2 \tag{6.10}$$

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*

$$\tau = 1 + 2\sum_{d=1}^{n-1} \kappa_d \tag{6.11}$$

For a correlation free experiment, τ equals 1.

6.1.39 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

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

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.

6.1.40 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)
```

```
-0.06895120441963641
3.74382855355186
-0.2186383556244991
0.8186111112149481 8.041491608761895 5.824398367359107
2.4292738814902837 1.6663000899456695 4.915994880963659
[[0.81861111 2.42927388 1.66630009]
 [2.42927388 8.04149161 4.91599488]
 [1.66630009 4.91599488 5.82439837]]
[12.69915666  0.06805927  1.91728516]
```

6.1.41 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.

6.1.42 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.

6.1.43 Random number generator RNG

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

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

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 $(13)\text{MOD}(9)$, the outcome is the remainder of the division $13/9$, namely 4.

6.1.44 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

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

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

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

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

6.1.45 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,

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

6.1.46 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

$$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69), \quad (6.12)$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}), \quad (6.13)$$

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

6.1.47 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 $m \wedge n$.

6.1.48 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

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

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

6.1.49 Random number generator RNG, RAN0

To see how this works we note first that

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\mathrm{MOD}(M), \quad (6.14)$$

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.

6.1.50 Random number generator RNG, RAN0

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

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\mathrm{MOD}(M), \quad (6.15)$$

which results in

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\ \mathrm{MOD}(M), \quad (6.16)$$

yielding

$$(aN_{i-1})\text{MOD}(M) = (a(N_{i-1}\text{MOD}(q)) - [N_{i-1}/q]r))\,\text{MOD}(M). \quad (6.17)$$

6.1.51 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 $N_{i-1}\text{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\text{MOD}(q)) < aq$. 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.

6.1.52 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()
```

6.1.53 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 $\mu = \langle x \rangle$.

6.1.54 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

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

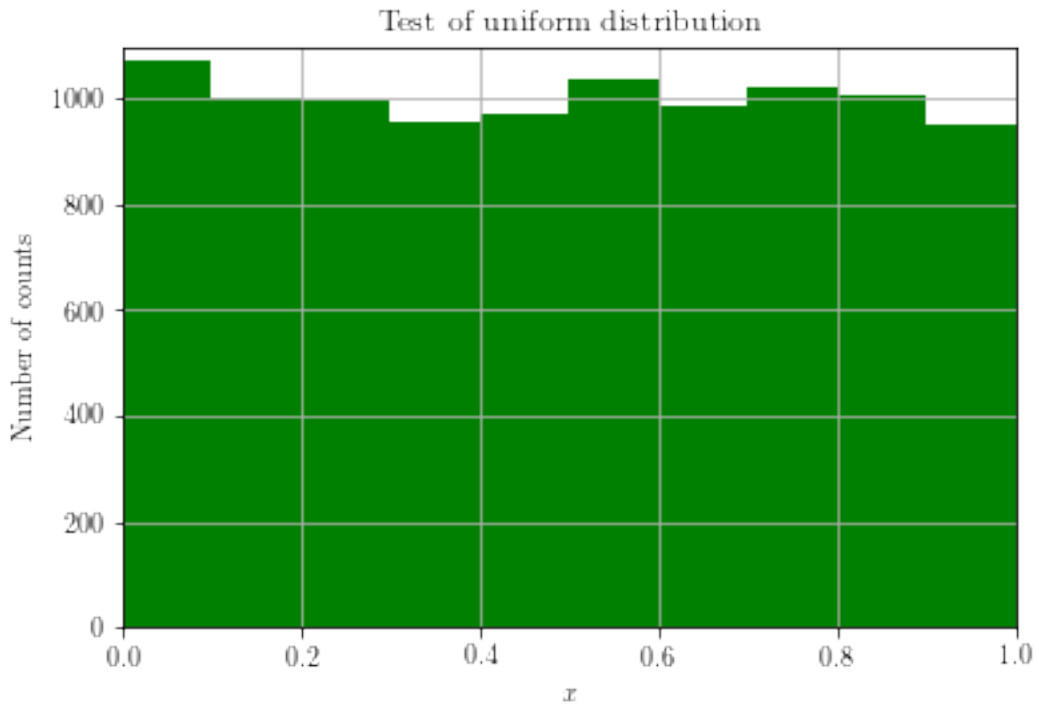
6.1.55 Properties of Selected Random Number Generators

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

6.1.56 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.

```
#!/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()
```



6.1.57 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.

6.1.58 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 $N(0, 1)$.

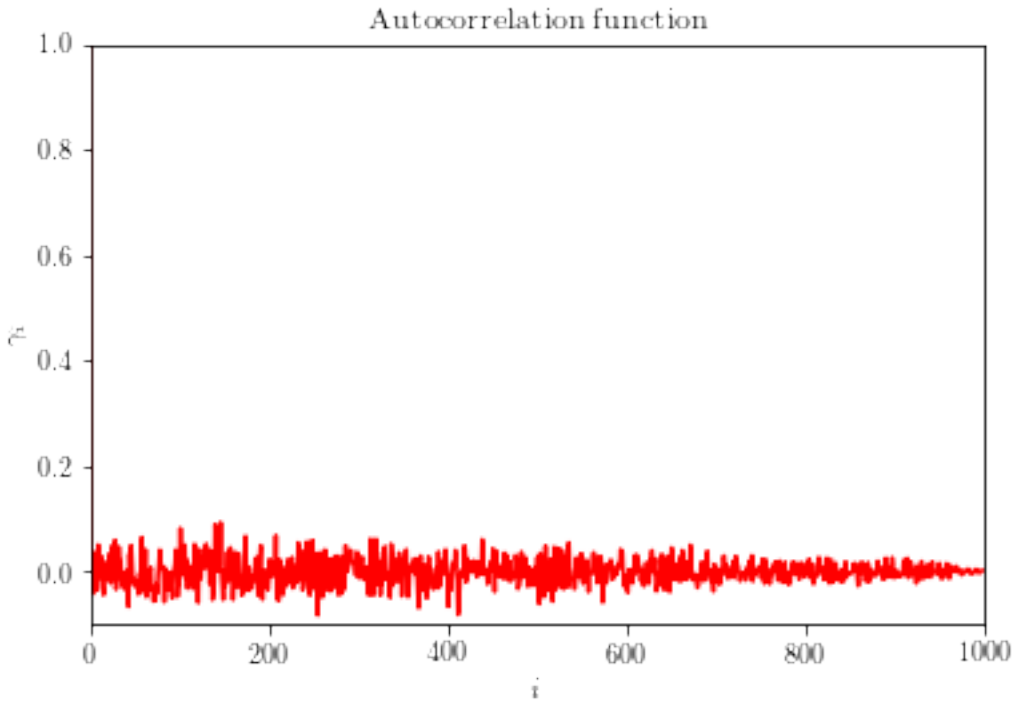
```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
```

(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()
```

```
0.052276839521249285 0.9675166677604984
```



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.

6.1.59 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
```

6.1.60 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

6.1.61 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);
```

6.1.62 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).

6.1.63 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)
```

6.1.64 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
```

6.1.65 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 τ is the correlation time (the time between a sample and the next uncorrelated sample) and Δt is time between each sample

6.1.66 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
```

6.1.67 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
```

6.1.68 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
```

6.1.69 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
```

6.1.70 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.

6.1.71 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.

6.1.72 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

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

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 $\angle\theta\rangle = \tilde{\theta} \pm \sigma_{\tilde{\theta}}$.

6.1.73 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)
```

6.1.74 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)
```

6.2 Getting started, our first data and Machine Learning encounters

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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!

6.2.6 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

6.2.7 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

6.2.8 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**.

6.2.9 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.

6.2.10 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 $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.
- The rows of $\mathbf{A}$ form a basis of R^N.
- The columns of $\mathbf{A}$ form a basis of R^N.
- $\mathbf{A}$ is a product of elementary matrices.
- 0 is not eigenvalue of $\mathbf{A}$.

6.2.11 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)
```

```
[ 0.99499832 -0.89728339 -1.69744895 -1.03875025 -0.07638981  0.18716123
 -0.27804028  0.72149922  1.25862131 -0.7970463 ]
```

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 $x_0, x_1, x_2, \ldots, x_{n-1}$. 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)
```

```
  File "<ipython-input-7-f6d7a289d493>", line 3
    print(x)
    ^
SyntaxError: invalid syntax
```

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)
```

6.2.12 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)
```

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])
```

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

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

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

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

or initializing all elements to

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

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

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

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.

```
# 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)
```

```
%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()
```

6.2.13 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)
```

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

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

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

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

We can easily append data to this, for example

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

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())
display(df**2)
```

Thereafter we can select specific columns only and plot final results

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

(continues on next page)

(continued from previous page)

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

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

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


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

We can produce a 4×4 matrix

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

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**.

6.2.14 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.

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

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

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

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

The function **coef** gives us the parameter 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

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

We present the squared logarithmic (quadratic) error

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

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

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 $a = \boldsymbol{y}-\tilde{\boldsymbol{y}}$. 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
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))
```

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

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

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

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

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

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

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

where A stands for the number of nucleons and the a_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_1A proportional with the number of nucleons (the energy is also an extensive quantity). When an assembly of nucleons of the same size is packed together into the smallest volume, each interior nucleon has a certain number of other nucleons in contact with it. This contribution is proportional to the volume.
- There is a surface energy term $a_2A^{2/3}$. The assumption here is that a nucleon at the surface of a nucleus interacts with fewer other nucleons than one in the interior of the nucleus and hence its binding energy is less. This surface energy term takes that into account and is therefore negative and is proportional to the surface area.
- There is a Coulomb energy term $a_3\frac{Z^2}{A^{1/3}}$. The electric repulsion between each pair of protons in a nucleus yields less binding.

- There is an asymmetry term $a_4\frac{(N-Z)^2}{A}$. This term is associated with the Pauli exclusion principle and reflects the fact that the proton-neutron interaction is more attractive on the average than the neutron-neutron and proton-proton interactions.

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

Organizing our data

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

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

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

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

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

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

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

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

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

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

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

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')
```

(continues on next page)

(continued from previous page)

```
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.
↪
"""
```

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
```

(continues on next page)

(continued from previous page)

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

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

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

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

The next step, and we will define this mathematically later, is to set up the so-called **design 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')
```

(continues on next page)

(continued from previous page)

```
ax.plot(Masses['A'], Masses['Eapprox'], alpha=0.7, lw=2, c='m',
            label='Fit')
ax.legend()
save_fig("Masses2016")
plt.show()
```

Seeing the wood for the trees

As a teaser, let us now see how we can do this with decision trees using **scikit-learn**. 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)

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

And what about using neural networks?

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

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

X_train = X
Y_train = Energies
n_hidden_neurons = 100
```

(continues on next page)

(continued from previous page)

```
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()
```

6.2.15 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.

6.3 Linear Regression and more Advanced Regression Analysis

6.3.1 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.

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.3.5 General linear models

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

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

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

where ϵ_i is the error in our approximation.

6.3.6 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

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

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

Defining the vectors

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

and

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

and

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

and the design matrix

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

we can rewrite our equations as

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

The above design matrix is called a Vandermonde matrix.

6.3.8 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

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

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

6.3.9 Generalizing the fitting procedure as a linear algebra problem

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

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

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?

6.3.10 Optimizing our parameters

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

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

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.

6.3.11 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.

```
%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.

6.3.12 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

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

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

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

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

It is also common to define the function C as

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

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

6.3.13 Interpretations and optimizing our parameters

The function

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

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

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

In practical terms it means we will require

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

which results in

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

or in a matrix-vector form as

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

6.3.14 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?

6.3.15 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.

2 6

< < < ! ! M A T H _ B L O C K

2 7

< < < ! ! M A T H _ B L O C K

2 8

< < < ! ! M A T H _ B L O C K

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

6.3.16 Interpretations and optimizing our parameters

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

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

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.

6.3.17 Own code for Ordinary Least Squares

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

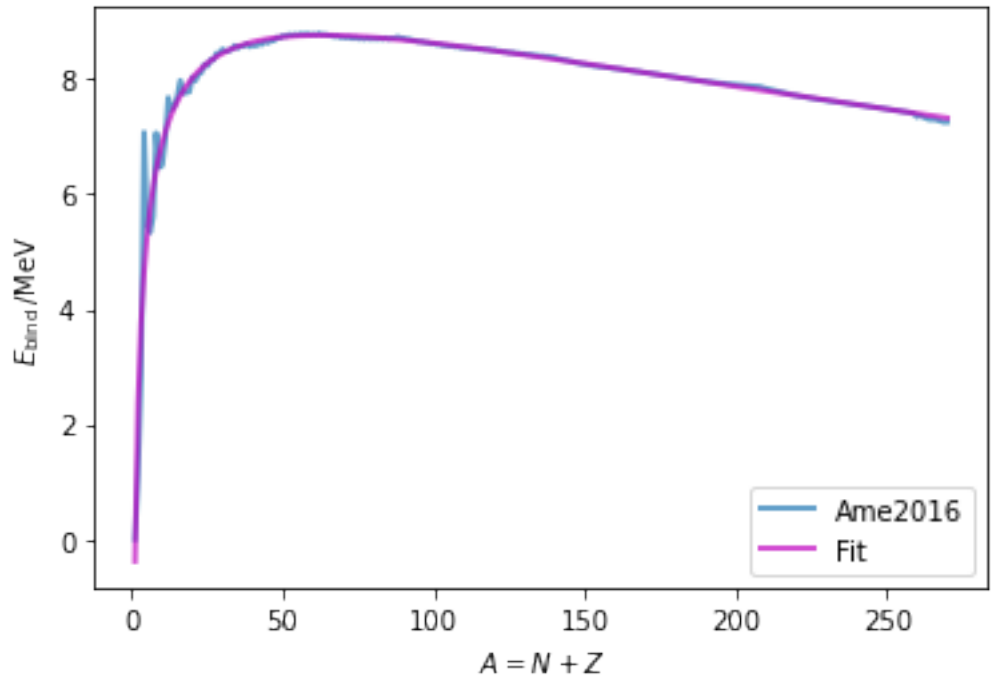
```
# matrix inversion to find beta
beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Energies)
# and then make the prediction
ytilde = X @ beta
```

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()
```



6.3.18 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
```

6.3.19 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

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

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

6.3.20 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

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

which results in

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

or in a matrix-vector form as

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

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 $b_i = y_i/\sigma_i$.

6.3.21 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

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

6.3.22 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

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

6.3.23 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

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

and

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

6.3.24 The χ^2 function

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

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

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

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

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

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

we obtain

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

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

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

6.3.25 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.

6.3.26 The code

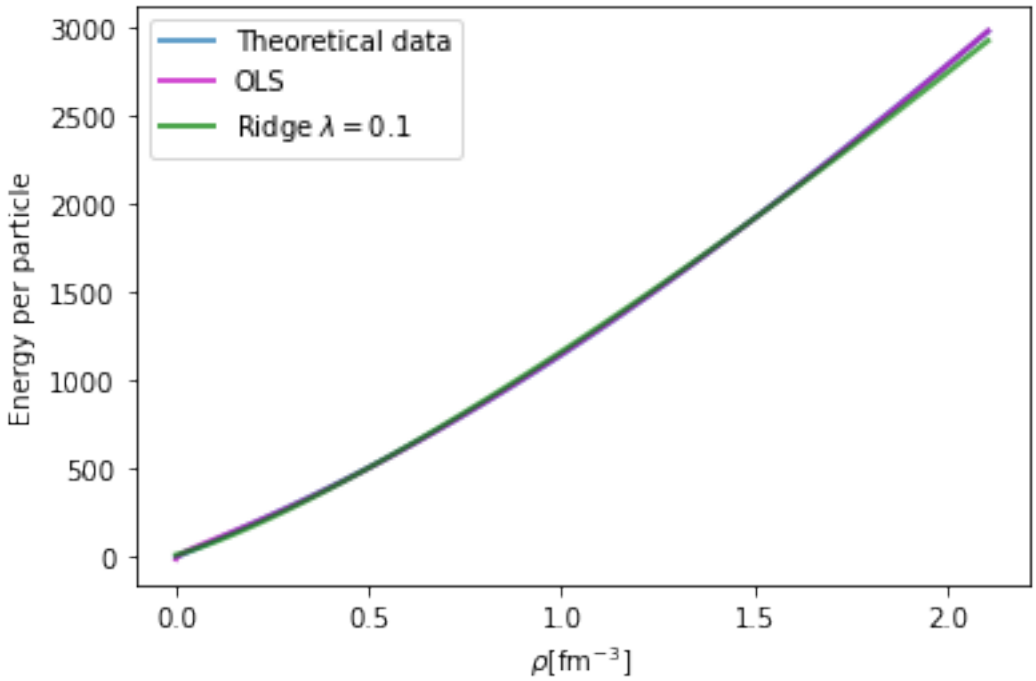
```
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
import sklearn.linear_model as skl
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

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

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

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

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

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

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

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

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

# Read the EoS data as  csv file and organize the data into two arrays with density␣
↪and energies
EoS = pd.read_csv(infile, names=('Density', 'Energy'))
EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors='coerce')
EoS = EoS.dropna()
Energies = EoS['Energy']
```

(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.

6.3.27 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
0.9999886705644145
Training MSE
3.90943518299982
Test R2
0.9999697792755088
```

(continues on next page)

(continued from previous page)

```
Test MSE
25.32441051671905
```

6.3.28 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

6.3.29 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
```

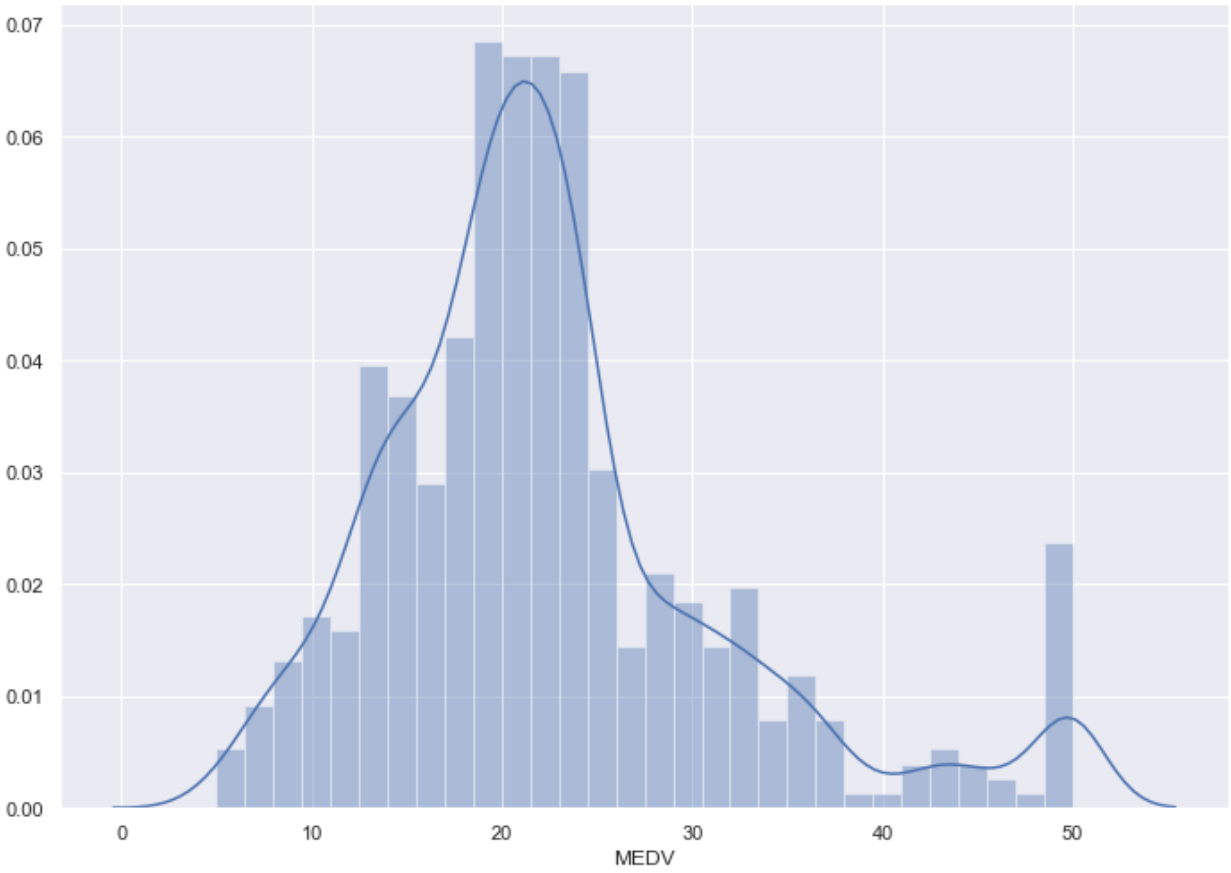
and preprocess the data

```
# 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()
```



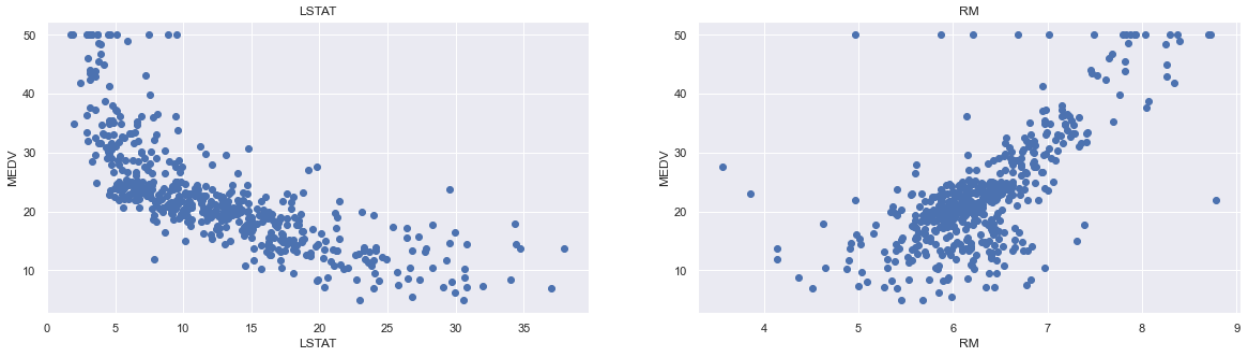
It is now useful to look at the correlation matrix

```
# 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 0x7fb5be1db040>
```

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
```

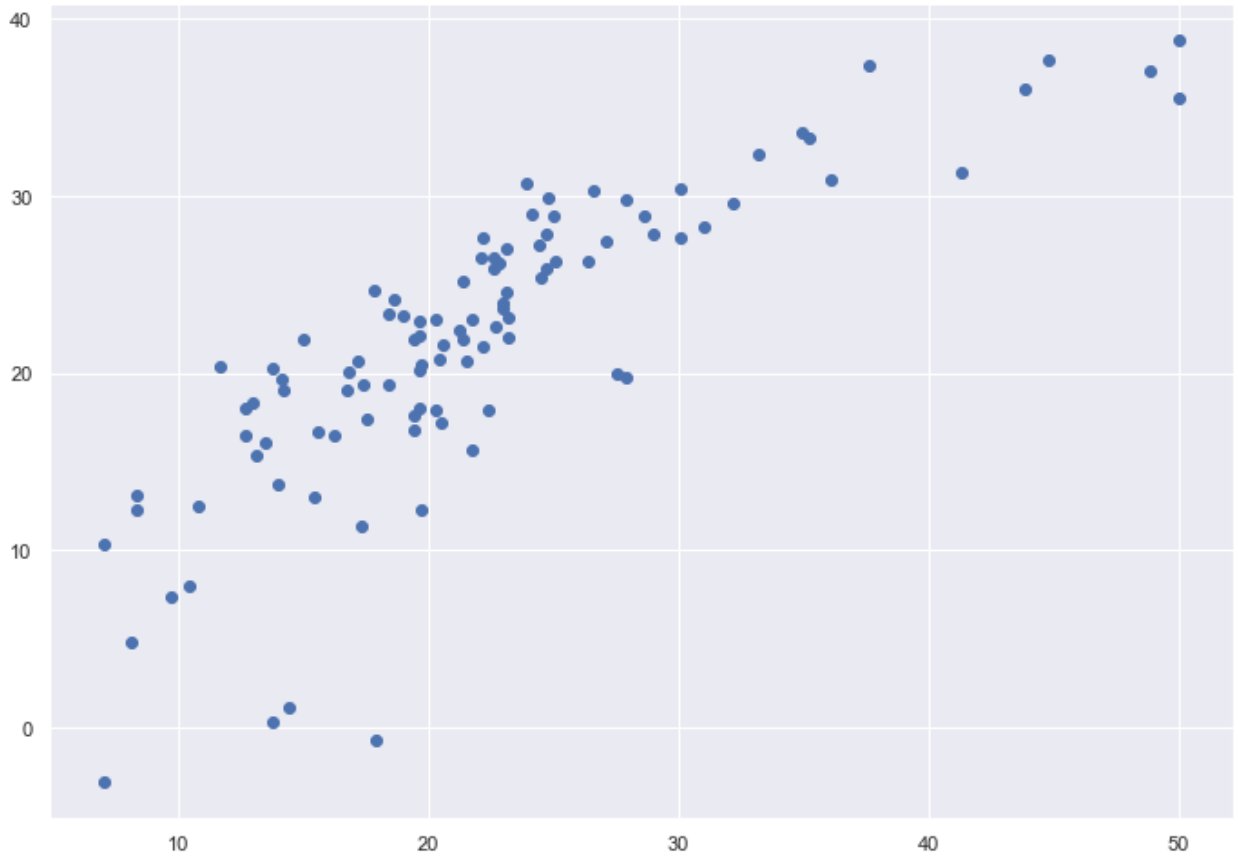
(continues on next page)

(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()
```



6.3.30 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.

6.3.31 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

6.3.32 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.

6.3.33 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 1.97624658e-03 6.76071445e-04 3.90555053e-06
 1.33608388e-06 4.57072598e-07 7.71833086e-09 2.64043119e-09
 9.03288157e-10 3.09013732e-10 1.52533249e-11 5.21814310e-12
 1.78512013e-12 6.10687329e-13 2.08915360e-13 3.01443312e-14
 1.03123374e-14 3.52783754e-15 1.20686874e-15 4.12868265e-16
 1.41241709e-16]
Feature max values before scaling:
 [1.         0.99729116 0.99990303 0.99458965 0.99719445 0.99980607
 0.99189546 0.9944932  0.99709775 0.99970911 0.98920857 0.99179928
 0.99439677 0.99700106 0.99961217 0.98652896 0.98911265 0.9917031
 0.99430034 0.99690438 0.99951524]
Feature min values after scaling:
 [ 0.         -1.72006556 -1.76752166 -1.10773734 -1.11611153 -1.12503152
 -0.87804483 -0.88146268 -0.88496565 -0.8885653  -0.75100539 -0.75305518
 -0.75511928 -0.75719979 -0.75929911 -0.66720819 -0.66863343 -0.67005915
 -0.67148589 -0.67291428 -0.67434497]
Feature max values after scaling:
 [0.         1.74774217 1.73485006 2.2506647  2.23491553 2.21893993
 2.6752011  2.65930564 2.64314172 2.62671501 3.05237641 3.03669249
 3.02075875 3.00457804 2.98815333 3.39653406 3.38082542 3.36489022
 3.34873108 3.33235061 3.31575145]
MSE after  scaling: 0.00
R2 score for  scaled data: 0.99
```

6.3.34 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.

6.3.35 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

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

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.

6.3.36 Fixing the singularity

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

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}, \quad (6.18)$$

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.

6.3.37 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.

6.3.38 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.

6.3.39 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.

6.3.40 Codes for the SVD

```
import numpy as np
# SVD inversion
def SVDinv(A):
    ''' Takes as input a numpy matrix A and returns inv(A) based on singular value␣
↪decomposition (SVD).
    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.

6.3.41 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.

$\tilde{y}^{OLS} = \boldsymbol{X}\hat{\beta}^{OLS} = \sum_{j=1}^{p}\boldsymbol{u}_j\boldsymbol{u}_j^T\boldsymbol{y}$ and for Ridge we have

$\tilde{y}^{Ridge} = \boldsymbol{X}\hat{\beta}^{Ridge} = \sum_{j=1}^{p}\boldsymbol{u}_j\frac{\sigma_j^2}{\sigma_j^2+\lambda}\boldsymbol{u}_j^T\boldsymbol{y}$.

It is indeed the economy-sized SVD, note the summation runs up tp $ponlyandnotn$.

Here we have that $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T, with\Sigma beingann \times pmatrixand\boldsymbol{V}beingap \times pmatrix.Wealsohaveassumedherethatn > p$.

6.3.42 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

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

or we can state it as

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

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

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

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

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

which leads to the Ridge regression minimization problem where we require that $||\boldsymbol{\beta}||_2^2\leq t$, 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

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

6.3.43 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

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

with t a finite positive number.

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

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

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

For Ridge regression this becomes

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

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

6.3.44 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.

6.3.45 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

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

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

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.

6.3.46 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.

6.3.47 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

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

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 $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}$.

6.3.48 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**.

6.3.49 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

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

6.3.50 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)
```

```
0.12208685625303164
4.452659449239899
[[ 1.00910422  3.12769989]
 [ 3.12769989 10.63920917]]
```

6.3.51 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)
```

```
0.08487462066865184
1.7716882265595972
[[1.         0.74332853]
 [0.74332853 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**.

6.3.52 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)
```

```
[[-0.63821798 -2.03548189]
 [ 0.98355854  2.40965456]
 [ 0.48870683  2.68995497]
 [ 0.44655566  1.28908336]
 [ 0.3871261  -0.67155367]
 [-0.32256574 -0.55849157]
 [ 1.3507663   2.75066843]
 [-1.44489727 -5.37462032]
 [-0.42087991  1.26840089]
 [-0.83015254 -1.76761477]]
          0         1
0 -0.638218 -2.035482
1  0.983559  2.409655
2  0.488707  2.689955
3  0.446556  1.289083
4  0.387126 -0.671554
5 -0.322566 -0.558492
6  1.350766  2.750668
7 -1.444897 -5.374620
8 -0.420880  1.268401
9 -0.830153 -1.767615
          0         1
0  1.000000  0.877156
1  0.877156  1.000000
```

We expand this model to the Franke function discussed above.

6.3.53 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  0.085618  0.079643  0.085857  0.084047  0.081888  0.076531  0.075718
2   0.0  0.079643  0.075359  0.078265  0.077130  0.075774  0.069143  0.068658
3   0.0  0.085857  0.078265  0.090778  0.088194  0.085138  0.083648  0.082461
4   0.0  0.084047  0.077130  0.088194  0.085938  0.083258  0.080985  0.079981
5   0.0  0.081888  0.075774  0.085138  0.083258  0.081011  0.077856  0.077054
6   0.0  0.076531  0.069143  0.083648  0.080985  0.077856  0.078889  0.077645
7   0.0  0.075718  0.068658  0.082461  0.079981  0.077054  0.077645  0.076512
8   0.0  0.074845  0.068154  0.081161  0.078881  0.076175  0.076275  0.075261
9   0.0  0.073877  0.067608  0.079704  0.077645  0.075189  0.074736  0.073852
```

(continues on next page)

(continued from previous page)

```
10  0.0  0.067084  0.060409  0.075015  0.072536  0.069633  0.071991  0.070821
11  0.0  0.066572  0.060088  0.074312  0.071948  0.069168  0.071268  0.070173
12  0.0  0.066065  0.059784  0.073591  0.071348  0.068699  0.070516  0.069500
13  0.0  0.065552  0.059492  0.072838  0.070724  0.068215  0.069723  0.068789
14  0.0  0.065022  0.059205  0.072037  0.070061  0.067706  0.068871  0.068023

           8         9         10        11        12        13        14
0   0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
1   0.074845  0.073877  0.067084  0.066572  0.066065  0.065552  0.065022
2   0.068154  0.067608  0.060409  0.060088  0.059784  0.059492  0.059205
3   0.081161  0.079704  0.075015  0.074312  0.073591  0.072838  0.072037
4   0.078881  0.077645  0.072536  0.071948  0.071348  0.070724  0.070061
5   0.076175  0.075189  0.069633  0.069168  0.068699  0.068215  0.067706
6   0.076275  0.074736  0.071991  0.071268  0.070516  0.069723  0.068871
7   0.075261  0.073852  0.070821  0.070173  0.069500  0.068789  0.068023
8   0.074139  0.072869  0.069531  0.068962  0.068371  0.067746  0.067072
9   0.072869  0.071752  0.068081  0.067595  0.067092  0.066559  0.065985
10  0.069531  0.068081  0.066617  0.065944  0.065240  0.064493  0.063686
11  0.068962  0.067595  0.065944  0.065324  0.064676  0.063985  0.063237
12  0.068371  0.067092  0.065240  0.064676  0.064083  0.063452  0.062765
13  0.067746  0.066559  0.064493  0.063985  0.063452  0.062881  0.062258
14  0.067072  0.065985  0.063686  0.063237  0.062765  0.062258  0.061702
```

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.

6.3.54 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}]to indicate that this is the covariance of the vectors \boldsymbol{x} of the design/feature matrix \b

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

6.3.55 Linking with SVD

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

6.3.56 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.

6.3.57 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.

6.3.58 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.

6.3.59 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.

6.3.60 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.

6.3.61 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.:

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

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

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

6.3.62 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 $\tilde{\boldsymbol{y}}$ where we want to minimize $(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2$, our MSE, with

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

6.3.63 Expectation value and variance

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

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

while its variance is

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

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).

6.3.64 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

$$to$$

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

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

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

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.

6.3.65 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.

6.3.66 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.

6.3.67 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 $\vec{X}_i$.

6.3.68 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: 0.430516 sec
Jackknife Statistics :
original           bias      std. error
 100.186        100.176        0.153008
```

6.3.69 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).

6.3.70 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.

6.3.71 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 $p(\boldsymbol{t})$.

6.3.72 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 $\boldsymbol{X}$.

6.3.73 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 $\widehat{\theta}^*$.

6.3.74 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()
```

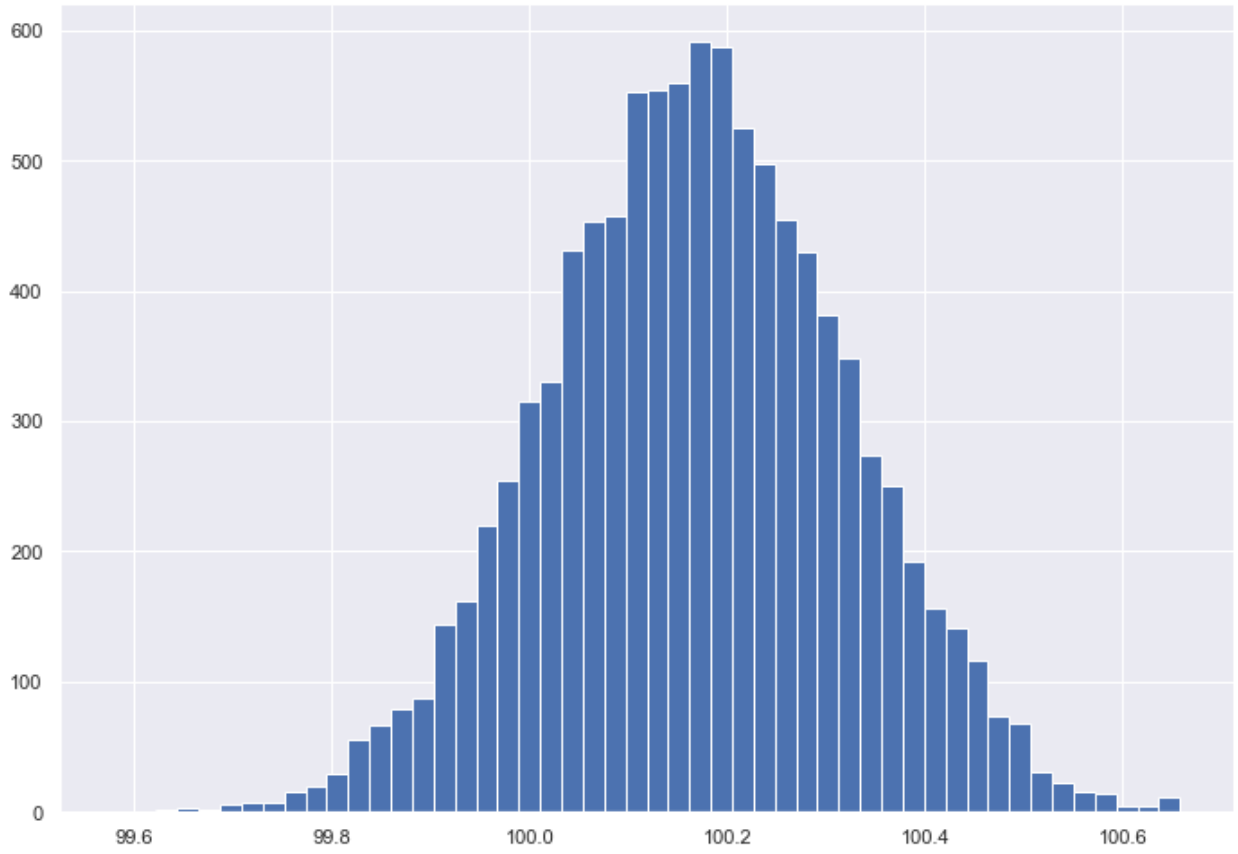
```
Runtime: 2.20285 sec
Bootstrap Statistics :
original           bias      std. error
 100.167   14.919        100.169        0.150847
```

```
---------------------------------------------------------------------------
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'
```



6.3.75 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).

6.3.76 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 $\{1, \ldots, n\} \setminus i$ and $\{i\}$, respectively.
- Fit the linear regression model by means of ridge estimation for each λ in the grid using the training set, and the corresponding estimate of the error variance $\boldsymbol{\sigma}^2_{-i}(\lambda)$, as

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

- Evaluate the prediction performance of these models on the test set by $\log\{L[y_i, \boldsymbol{X}_{i,*}; \boldsymbol{\beta}_{-i}(\lambda), \boldsymbol{\sigma}^2_{-i}(\lambda)]\}$. 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

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

6.3.77 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

6.3.78 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()
```

6.3.79 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 $\mathbf{X}_{\mathcal{L}} = \{(y_j, \boldsymbol{x}_j), j = 0 \dots n-1\}$.

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

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

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

We can rewrite this as

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

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 $\tilde{\boldsymbol{y}}$. We use a more compact notation in terms of the expectation value

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

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

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

which, using the abovementioned expectation values can be rewritten as

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

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

6.3.80 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()
```

6.3.81 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 = 40
n_boostraps = 100
maxdegree = 14


# Make data set.
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)
error = np.zeros(maxdegree)
bias = np.zeros(maxdegree)
variance = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

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

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

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

6.3.82 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.

6.3.83 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
```

(continues on next page)

(continued from previous page)

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

np.random.seed(0)

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

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

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

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

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

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

6.3.84 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/"
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
    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()
```

6.3.85 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
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']
```

(continues on next page)

(continued from previous page)

```
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')
plt.xlabel('Polynomial degree')
plt.ylabel('log10[MSE]')
plt.legend()
plt.show()
```

6.3.86 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:
```

(continues on next page)

(continued from previous page)

```
    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()
```

6.3.87 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.

6.3.88 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)
```

6.3.89 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)
```

6.3.90 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?

6.3.91 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)

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.

6.3.92 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 β. This results in a penalized regression problem. The cost function is given by

1 3 6

< < < ! ! M A T H _ B L O C K

```
_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)
```

(continues on next page)

(continued from previous page)

```
cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize=18)

plt.show()
```

6.3.93 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 $J_{j,j+1} = -1$.

6.3.94 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
```

(continues on next page)

(continued from previous page)

```
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 λ, but will "wither" when we increase λ too much. Ridge is more stable over a larger range of values for λ, but eventually also fades away.

6.3.95 Finding the optimal value of λ

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

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

colors = {
    "ols_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),
```

(continues on next page)

(continued from previous page)

```
        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 λ.

6.4 Logistic Regression

6.4.1 Introduction

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.

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.

6.4.2 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 $k = 0, \ldots, K-1$ (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

$$y_i = \begin{bmatrix} 0 & \text{no} \\ 1 & \text{yes} \end{bmatrix}.$$

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.

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.

6.4.3 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,

$$p(t) = \frac{1}{1 + \exp -t} = \frac{\exp t}{1 + \exp t}.$$

Note that $1 - p(t) = p(-t)$.

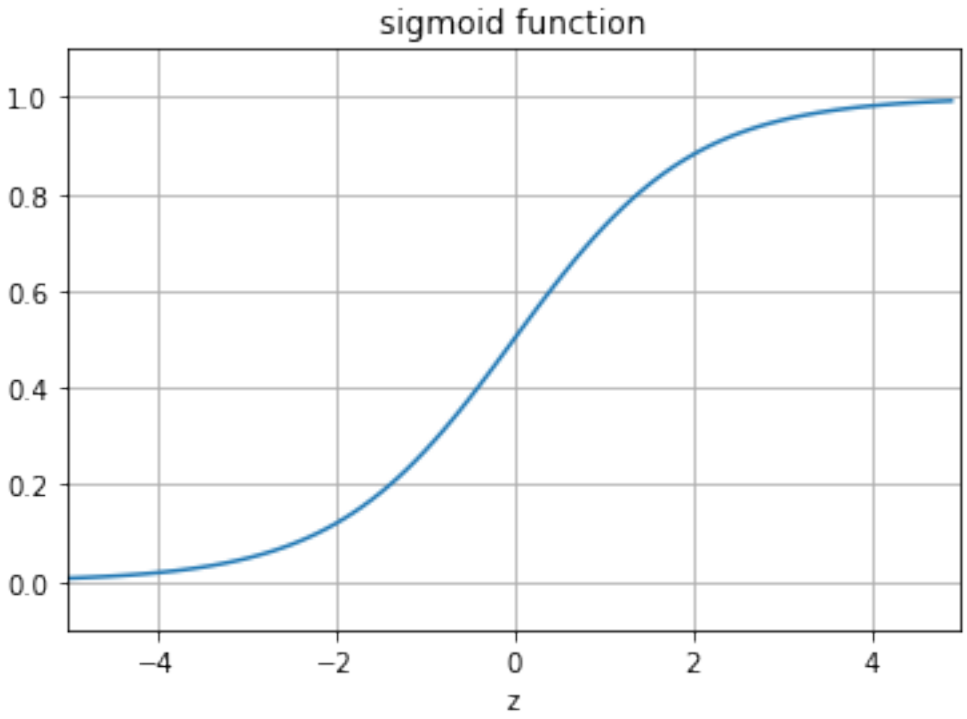
The following code plots the logistic function, the step function and other functions we will encounter from here and on.

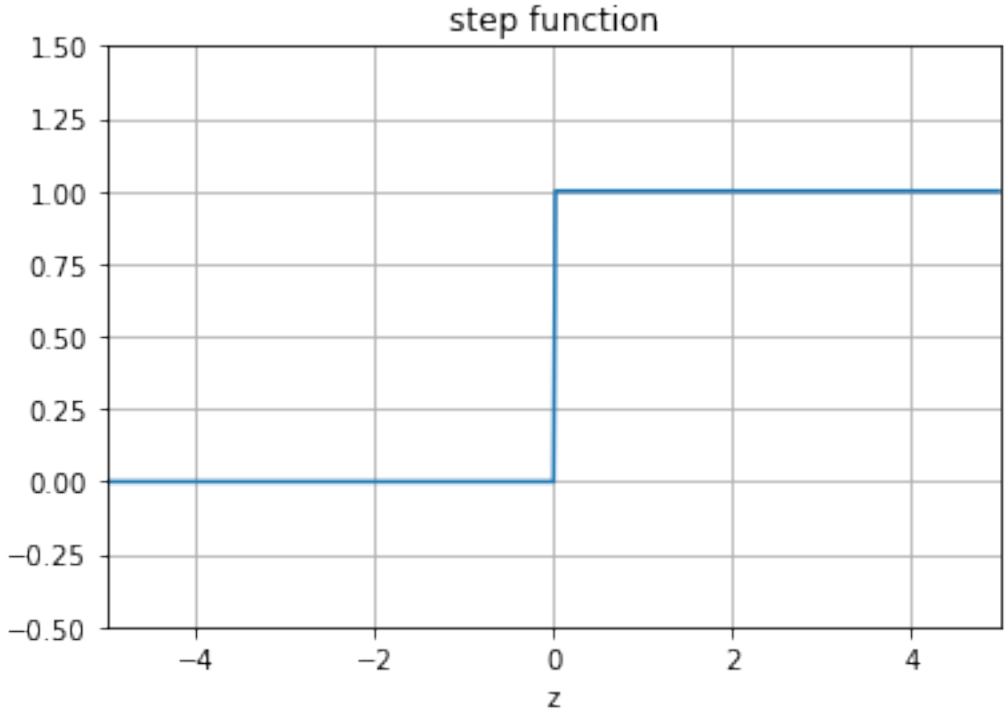
```
%matplotlib inline

"""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."""
```

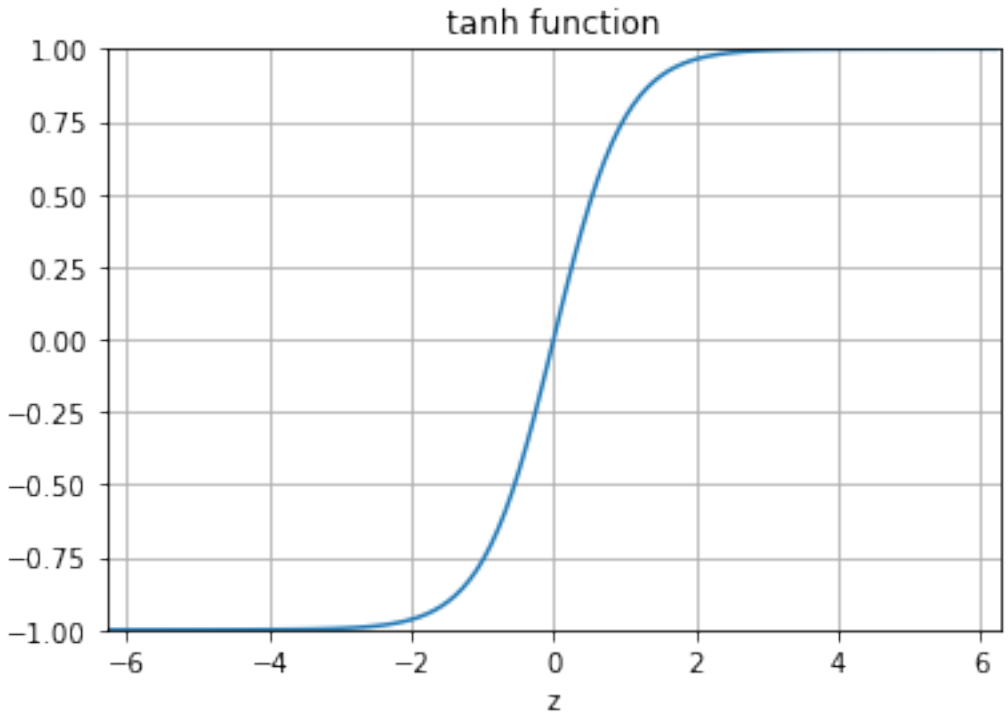
(continues on next page)

(continued from previous page)

```
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])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('tanh function')

plt.show()
```


sigmoid function
1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4
z


step function
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−4
−2
0
2
4
z



6.4.4 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

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

where $\hat{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

Note that we used

$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}).$$

6.4.5 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

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i \log p(y_i = 1|x_i, \hat{\beta}) + (1 - y_i) \log \left[1 - p(y_i = 1|x_i, \hat{\beta}))\right]\right).$$

Reordering the logarithms, we can rewrite the **cost/loss** function as

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

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

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

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.

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

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right),$$

and

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i\frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right).$$

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 $p(y_i|x_i, \hat{\beta})$. 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 $p(y_i|x_i, \hat{\beta})(1 - p(y_i|x_i, \hat{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}\partial \hat{\beta}^T} = \hat{X}^T\hat{W}\hat{X}.$$

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log\frac{p(\hat{\beta}\hat{x})}{1 - p(\hat{\beta}\hat{x})} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Here we defined $\hat{x} = [1, x_1, x_2, \ldots, x_p]$ and $\hat{\beta} = [\beta_0, \beta_1, \ldots, \beta_p]$ leading to

$$p(\hat{\beta}\hat{x}) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}{1 + \exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}.$$

6.4.6 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

<<<!! M A T H _ B L O C K

$$\log \frac{p(C=2|x)}{p(K|x)} = \beta_{20} + \beta_{21}x_1,$$

and so on till the class $C = K - 1$ class

$$\log \frac{p(C=K-1|x)}{p(K|x)} = \beta_{(K-1)0} + \beta_{(K-1)1}x_1,$$

and the model is specified in term of $K - 1$ so-called log-odds or **logit** transformations.

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):

$$p(C=k|\mathbf{x}) = \frac{\exp\left(\beta_{k0} + \beta_{k1}x_1\right)}{1 + \sum_{l=1}^{K-1}\exp\left(\beta_{l0} + \beta_{l1}x_1\right)}.$$

It is easy to extend to more predictors. The final class is

$$p(C=K|\mathbf{x}) = \frac{1}{1 + \sum_{l=1}^{K-1}\exp\left(\beta_{l0} + \beta_{l1}x_1\right)},$$

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.

6.5 Neural networks, from the simple perceptron to deep learning

6.5.1 To do list

- write code for single perceptron model and make link with linear regression
- revise initial info and add references
- Update tensorflow material, with keras
- think of adding material about pytorch
- rework pulsar example and breast cancer example
- add ising model example for both regression and classification
- make data on gravitational problem, add reference to articles on uncovering physical laws from ML
- think of genetic data

6.5.2 Neural networks

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable.

6.5.3 Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u)$$

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons.

Conceptually, it is helpful to divide neural networks into four categories:

1. general purpose neural networks for supervised learning,
2. neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs),
3. neural networks for sequential data such as Recurrent Neural Networks (RNNs), and
4. neural networks for unsupervised learning such as Deep Boltzmann Machines.

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs, topological phases, and even non-equilibrium many-body localization. Representing quantum states as DNNs quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study of quantum systems.

In quantum information theory, it has been shown that one can perform gate decompositions with the help of neural.

The applications are not limited to the natural sciences. There is a plethora of applications in essentially all disciplines, from the humanities to life science and medicine.

6.5.4 Neural network types

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons, or nodes or units. We will refer to these interchangeably as units or nodes, and sometimes as neurons.

It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different ANNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods we discussed earlier.

6.5.5 Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of ANNs that were devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable (figure to come). We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

6.5.6 Convolutional Neural Network

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation. (figure to come)

Convolutional neural networks emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

6.5.7 Recurrent neural networks

So far we have only mentioned ANNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

6.5.8 Other types of networks

There are many other kinds of ANNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer (''linear" here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a separate type of NN due the unusual activation functions.

6.5.9 Multilayer perceptrons

One uses often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs).

6.5.10 Why multilayer perceptrons?

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**.

Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

6.5.11 Mathematical model

The output y is produced via the activation function f

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z),$$

This function receives x_i as inputs. Here the activation $z = (\sum_{i=1}^{n} w_i x_i + b_i)$. In an FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

6.5.12 Mathematical model

First, for each node i in the first hidden layer, we calculate a weighted sum z_i^1 of the input coordinates x_j,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1$$

Here b_i is the so-called bias which is normally needed in case of zero activation weights or inputs. How to fix the biases and the weights will be discussed below. The value of z_i^1 is the argument to the activation function f_i of each

node i, The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right)$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f. In general, we could assume in the more general case that different layers have different activation functions. In this case we would identify these functions with a superscript l for the l-th layer,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right)$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

6.5.13 Mathematical model

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right)$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right]$$

where we have substituted y_k^1 with the inputs x_k. Finally, the ANN output reads

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right)$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right]$$

6.5.14 Mathematical model

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\ldots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right]$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n.

6.5.15 Mathematical model

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\hat{x} \in \mathbb{R}^n \rightarrow \hat{y} \in \mathbb{R}^m$.

Furthermore, the flexibility and universality of an MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$f(x) = c_1 f(c_2 x + c_3) + c_4$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

Matrix-vector notation

We can introduce a more convenient notation for the activations in an A NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\hat{b}_l$ and $\hat{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that W_l is an $N_{l-1} \times N_l$ matrix, while $\hat{b}_l$ and $\hat{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 (assuming three nodes for simplicity) as

$$\hat{y}_2 = f_2(\mathrm{W}_2\hat{y}_1 + \hat{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right).$$

$w_{13}^2 w_{21}^2$
$w_{23}^2 w_{31}^2$
w_{33}^2

$$\cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}.$$

Matrix-vector notation and activation

The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right).$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l\hat{y}_{l-1}$ we move forward one layer.

Activation functions

A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

Activation functions, Logistic and Hyperbolic ones

The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1 + e^{-x}},$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x)$$

Relevance

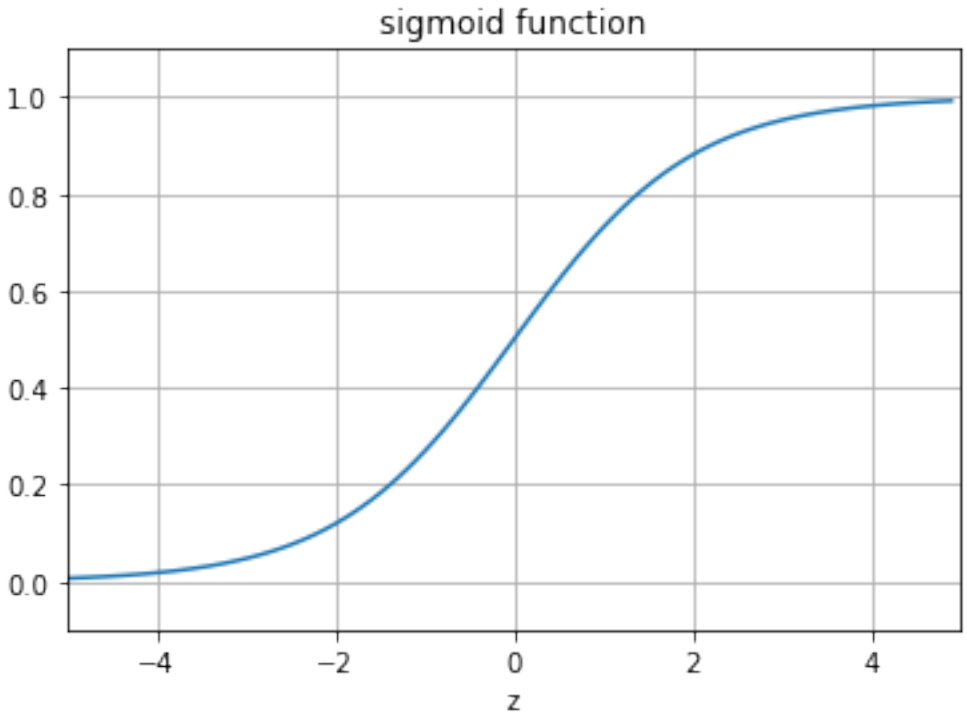
The *sigmoid* function are more biologically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown that the hyperbolic tangent performs better than the sigmoid for training MLPs. has become the most popular for *deep neural networks*

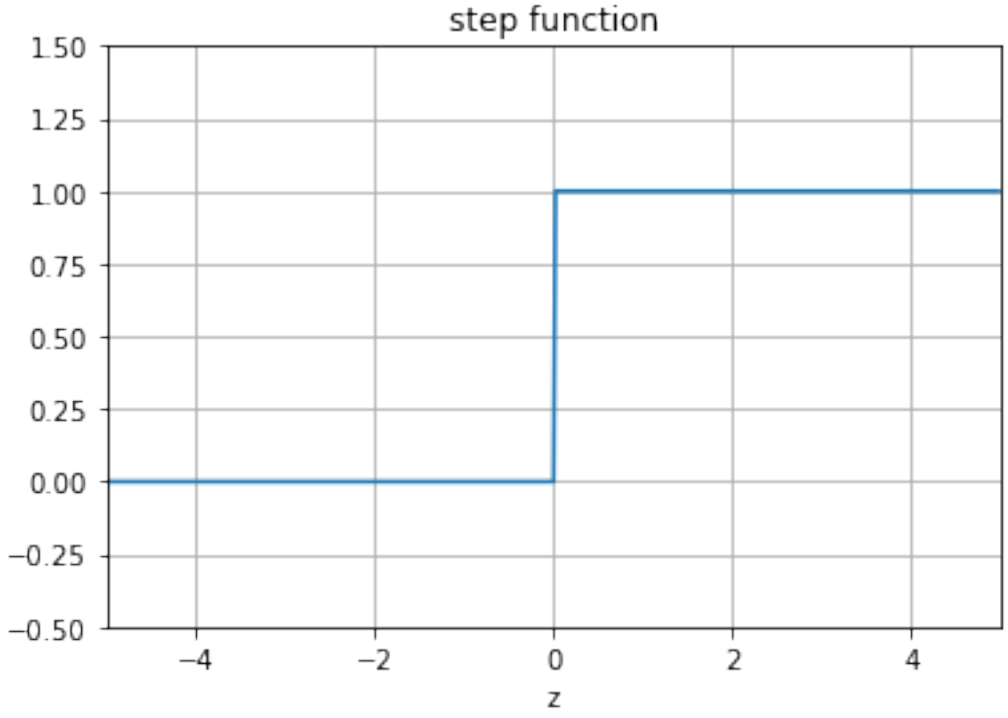
```
%matplotlib inline

"""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)
```

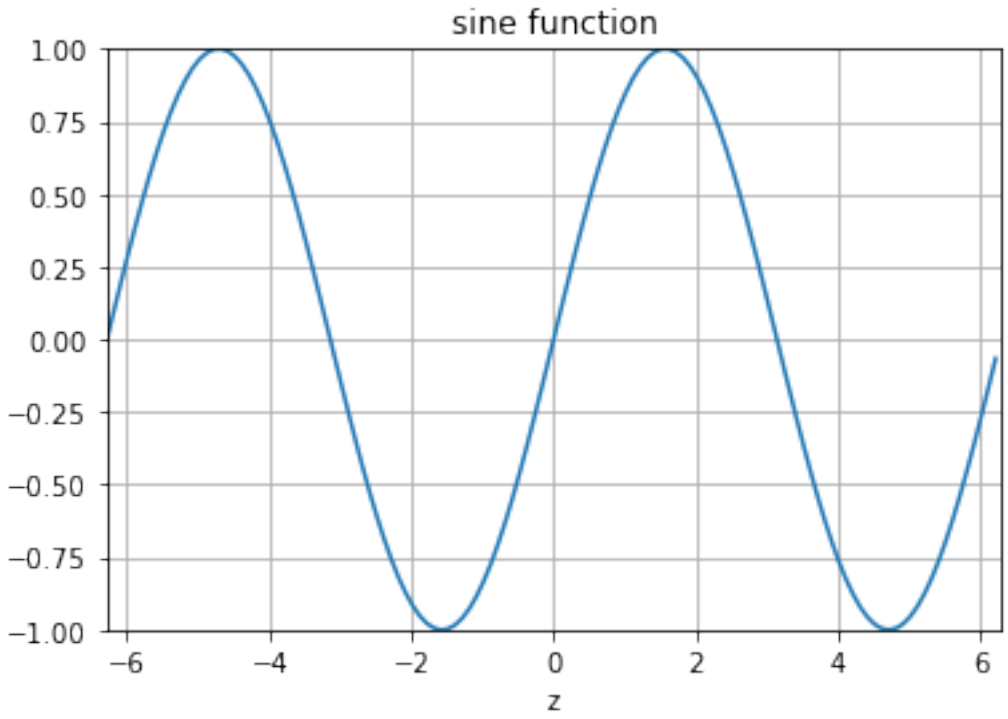
(continues on next page)

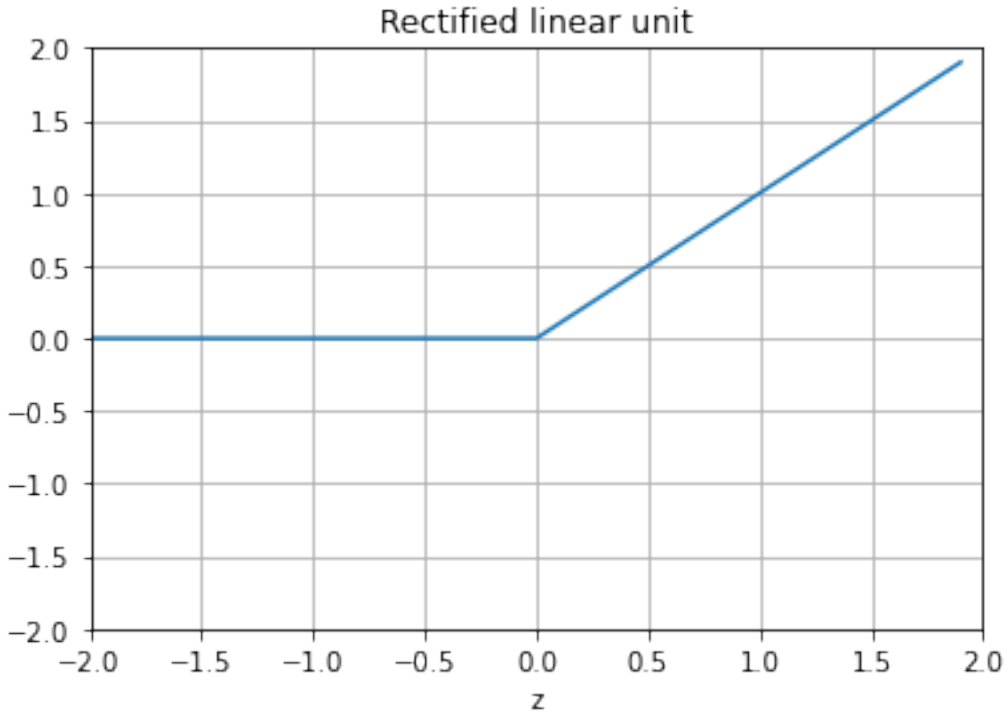
(continued from previous page)

```
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()

"""Sine Function"""
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.sin(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])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sine function')

plt.show()

"""Plots a graph of the squashing function used by a rectified linear
unit"""
z = numpy.arange(-2, 2, .1)
zero = numpy.zeros(len(z))
y = numpy.max([zero, z], axis=0)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, y)
ax.set_ylim([-2.0, 2.0])
ax.set_xlim([-2.0, 2.0])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('Rectified linear unit')

plt.show()
```


sigmoid function
1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4
z


step function
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−4
−2
0
2
4
z


sine function
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−6
−4
−2
0
2
4
6
z


Rectified linear unit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
z

6.5.16 The multilayer perceptron (MLP)

The multilayer perceptron is a very popular, and easy to implement approach, to deep learning. It consists of

1. A neural network with one or more layers of nodes between the input and the output nodes.
2. The multilayer network structure, or architecture, or topology, consists of an input layer, one or more hidden layers, and one output layer.
3. The input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

As a convention it is normal to call a network with one layer of input units, one layer of hidden units and one layer of output units as a two-layer network. A network with two layers of hidden units is called a three-layer network etc etc.

For an MLP network there is no direct connection between the output nodes/neurons/units and the input nodes/neurons/units. Hereafter we will call the various entities of a layer for nodes. There are also no connections within a single layer.

The number of input nodes does not need to equal the number of output nodes. This applies also to the hidden layers. Each layer may have its own number of nodes and activation functions.

The hidden layers have their name from the fact that they are not linked to observables and as we will see below when we define the so-called activation $\hat{z}$, we can think of this as a basis expansion of the original inputs $\hat{x}$. The difference however between neural networks and say linear regression is that now these basis functions (which will correspond to the weights in the network) are learned from data. This results in an important difference between neural networks and deep learning approaches on one side and methods like logistic regression or linear regression and their modifications on the other side.

6.5.17 From one to many layers, the universal approximation theorem

A neural network with only one layer, what we called the simple perceptron, is best suited if we have a standard binary model with clear (linear) boundaries between the outcomes. As such it could equally well be replaced by standard linear regression or logistic regression. Networks with one or more hidden layers approximate systems with more complex boundaries.

As stated earlier, an important theorem in studies of neural networks, restated without proof here, is the universal approximation theorem.

It states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. It is the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators.

6.5.18 Deriving the back propagation code for a multilayer perceptron model

Note: figures will be inserted later!

As we have seen now in a feed forward network, we can express the final output of our network in terms of basic matrix-vector multiplications. The unknowwn quantities are our weights w_{ij} and we need to find an algorithm for changing them so that our errors are as small as possible. This leads us to the famous back propagation algorithm.

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights?

To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\hat{W}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - t_i\right)^2,$$

where the t_is are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\hat{x}$ are given by y_i. Below we will demonstrate how the basic equations arising from the back propagation algorithm can be modified in order to study classification problems with K classes.

6.5.19 Definitions

With our definition of the targets $\hat{t}$, the outputs of the network $\hat{y}$ and the inputs $\hat{x}$ we define now the activation z_j^l of node/neuron/unit j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs $\hat{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure here illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l.$$

With the activation values $\hat{z}^l$ we can in turn define the output of layer l as $\hat{a}^l = f(\hat{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = f(z_j^l) = \frac{1}{1+\exp -(z_j^l)}.$$

6.5.20 Derivatives and the chain rule

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1-a_j^l) = f(z_j^l)(1-f(z_j^l)).$$

6.5.21 Derivative of the cost function

With these definitions we can now compute the derivative of the cost function in terms of the weights.

Let us specialize to the output layer $l = L$. Our cost function is

$$\mathcal{C}(\hat{W^L}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - t_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right) \frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1-a_j^L)a_k^{L-1},$$

6.5.22 Bringing it together, first back propagation equation

We have thus

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right) a_j^L(1-a_j^L)a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1-a_j^L)\left(a_j^L - t_j\right) = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\hat{\delta}^L = f'(\hat{z}^L) \circ \frac{\partial \mathcal{C}}{\partial (\hat{a}^L)}.$$

This is an important expression. The second term on the right handside measures how fast the cost function is changing as a function of the jth output activation. If, for example, the cost function doesn't depend much on a particular output node j, then δ_j^L will be small, which is what we would expect. The first term on the right, measures how fast the activation function f is changing at a given activation value z_j^L.

Notice that everything in the above equations is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it is only a small additional overhead to compute $f'(z_j^L)$. The exact form of the derivative with respect to the output depends on the form of the cost function. However, provided the cost function is known there should be little trouble in calculating

$$\frac{\partial \mathcal{C}}{\partial (a_j^L)}$$

With the definition of δ_j^L we have a more compact definition of the derivative of the cost function in terms of the weights, namely

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}.$$

6.5.23 Derivatives in terms of z_j^L

It is also easy to see that our previous equation can be written as

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L},$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, namely

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

6.5.24 Bringing it together

We have now three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

The starting equations.

$$\frac{\partial \mathcal{C}(\hat{W^L})}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1},$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)},$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L},$$

An interesting consequence of the above equations is that when the activation a_k^{L-1} is small, the gradient term, that is the derivative of the cost function with respect to the weights, will also tend to be small. We say then that the weight learns slowly, meaning that it changes slowly when we minimize the weights via say gradient descent. In this case we say the system learns slowly.

Another interesting feature is that is when the activation function, represented by the sigmoid function here, is rather flat when we move towards its end values 0 and 1 (see the above Python codes). In these cases, the derivatives of the activation function will also be close to zero, meaning again that the gradients will be small and the network learns slowly again.

We need a fourth equation and we are set. We are going to propagate backwards in order to the determine the weights and biases. In order to do so we need to represent the error in the layer before the final one $L-1$ in terms of the errors in the final output layer.

6.5.25 Final back propagating equation

We have that (replacing L with a general layer l)

$$\delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l+1$. Using the chain rule and summing over all k entries we have

$$\delta_j^l = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

with M_l being the number of nodes in layer l, we obtain

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l),$$

This is our final equation.

We are now ready to set up the algorithm for back propagation and learning the weights and biases.

6.5.26 Setting up the Back propagation algorithm

The four equations provide us with a way of computing the gradient of the cost function. Let us write this out in the form of an algorithm.

First, we set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^1$.

Secondly, we perform then the feed forward till we reach the output layer and compute all $\hat{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^l$ for $l = 2, 3, \ldots, L$.

Thereafter we compute the ouput error $\hat{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L) \frac{\partial \mathcal{C}}{\partial (a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 2$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

The parameter η is the learning parameter discussed in connection with the gradient descent methods. Here it is convenient to use stochastic gradient descent (see the examples below) with mini-batches with an outer loop that steps through multiple epochs of training.

6.5.27 Setting up a Multi-layer perceptron model for classification

We are now gong to develop an example based on the MNIST data base. This is a classification problem and we need to use our cross-entropy function we discussed in connection with logistic regression. The cross-entropy defines our cost function for the classificaton problems with neural networks.

In binary classification with two classes $(0, 1)$ we define the logistic/sigmoid function as the probability that a particular input is in class 0 or 1. This is possible because the logistic function takes any input from the real numbers and inputs a number between 0 and 1, and can therefore be interpreted as a probability. It also has other nice properties, such as a derivative that is simple to calculate.

For an input $\boldsymbol{a}$ from the hidden layer, the probability that the input $\boldsymbol{x}$ is in class 0 or 1 is just. We let θ represent the unknown weights and biases to be adjusted by our equations). The variable x represents our activation values z. We have

$$P(y = 0 \mid \hat{x}, \hat{\theta}) = \frac{1}{1 + \exp(-\hat{x})},$$

and

$$P(y = 1 \mid \hat{x}, \hat{\theta}) = 1 - P(y = 0 \mid \hat{x}, \hat{\theta}),$$

where $y \in \{0, 1\}$ and $\hat{\theta}$ represents the weights and biases of our network.

6.5.28 Defining the cost function

Our cost function is given as (see the Logistic regression lectures)

$$\mathcal{C}(\hat{\theta}) = -\ln P(\mathcal{D} \mid \hat{\theta}) = -\sum_{i=1}^{n} y_i \ln[P(y_i = 0)] + (1 - y_i) \ln[1 - P(y_i = 0)] = \sum_{i=1}^{n} \mathcal{L}_i(\hat{\theta}).$$

This last equality means that we can interpret our *cost* function as a sum over the *loss* function for each point in the dataset $\mathcal{L}_i(\hat{\theta})$.The negative sign is just so that we can think about our algorithm as minimizing a positive number, rather than maximizing a negative number.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$y = 5 \quad \rightarrow \quad \hat{y} = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$, and

$y = 1 \quad \rightarrow \quad \hat{y} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$,

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset (numbers from 0 to 9)..

If $\hat{x}_i$ is the i-th input (image), y_{ic} refers to the c-th component of the i-th output vector $\hat{y}_i$.The probability of $\hat{x}_i$ being in class c will be given by the softmax function:

$$P(y_{ic} = 1 \mid \hat{x}_i, \hat{\theta}) = \frac{\exp((\hat{a}_i^{hidden})^T \hat{w}_c)}{\sum_{c'=0}^{C-1} \exp((\hat{a}_i^{hidden})^T \hat{w}_{c'})},$$

which reduces to the logistic function in the binary case.The likelihood of this C-class classifier is now given as:

$$P(\mathcal{D} \mid \hat{\theta}) = \prod_{i=1}^{n} \prod_{c=0}^{C-1} [P(y_{ic} = 1)]^{y_{ic}}.$$

Again we take the negative log-likelihood to define our cost function:

$$\mathcal{C}(\hat{\theta}) = -\log P(\mathcal{D} \mid \hat{\theta}).$$

See the logistic regression lectures for a full definition of the cost function.

The back propagation equations need now only a small change, namely the definition of a new cost function. We are thus ready to use the same equations as before!

6.5.29 Example: binary classification problem

As an example of the above, relevant for project 2 as well, let us consider a binary class. As discussed in our logistic regression lectures, we defined a cost function in terms of the parameters β as

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i \log p(y_i|x_i,\hat{\beta}) + (1-y_i) \log 1 - p(y_i|x_i,\hat{\beta}) \right),$$

where we had defined the logistic (sigmoid) function

$$p(y_i = 1|x_i,\hat{\beta}) = \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1+\exp\left(\beta_0 + \beta_1 x_i\right)},$$

and

$$p(y_i = 0|x_i,\hat{\beta}) = 1 - p(y_i = 1|x_i,\hat{\beta}).$$

The parameters $\hat{\beta}$ were defined using a minimization method like gradient descent or Newton-Raphson's method.

Now we replace x_i with the activation z_i^l for a given layer l and the outputs as $y_i = a_i^l = f(z_i^l)$, with z_i^l now being a function of the weights w_{ij}^l and biases b_i^l. We have then

$$a_i^l = y_i = \frac{\exp\left(z_i^l\right)}{1+\exp\left(z_i^l\right)},$$

with

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l,$$

where the superscript $l-1$ indicates that these are the outputs from layer $l-1$. Our cost function at the final layer $l = L$ is now

$$\mathcal{C}(\hat{W}) = -\sum_{i=1}^{n} \left(t_i \log a_i^L + (1-t_i) \log\left(1-a_i^L\right) \right),$$

where we have defined the targets t_i. The derivatives of the cost function with respect to the output a_i^L are then easily calculated and we get

$$\frac{\partial \mathcal{C}(\hat{W})}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1-a_i^L)}.$$

In case we use another activation function than the logistic one, we need to evaluate other derivatives.

6.5.30 The Softmax function

In case we employ the more general case given by the Softmax equation, we need to evaluate the derivative of the activation function with respect to the activation z_i^l, that is we need

$$\frac{\partial f(z_i^l)}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} a_k^{l-1}.$$

For the Softmax function we have

$$f(z_i^l) = \frac{\exp\left(z_i^l\right)}{\sum_{m=1}^{K}\exp\left(z_m^l\right)}.$$

Its derivative with respect to z_j^l gives

$$\frac{\partial f(z_i^l)}{\partial z_j^l} = f(z_i^l)\left(\delta_{ij} - f(z_j^l)\right),$$

which in case of the simply binary model reduces to having $i = j$.

6.5.31 Developing a code for doing neural networks with back propagation

One can identify a set of key steps when using neural networks to solve supervised learning problems:

1. Collect and pre-process data
2. Define model and architecture
3. Choose cost function and optimizer
4. Train the model
5. Evaluate model performance on test data
6. Adjust hyperparameters (if necessary, network architecture)

6.5.32 Collect and pre-process data

Here we will be using the MNIST dataset, which is readily available through the **scikit-learn** package. You may also find it for example here.The *MNIST* (Modified National Institute of Standards and Technology) database is a large database of handwritten digits that is commonly used for training various image processing systems.The MNIST dataset consists of 70 000 images of size 28×28 pixels, each labeled from 0 to 9.The scikit-learn dataset we will use consists of a selection of 1797 images of size 8×8 collected and processed from this database.

To feed data into a feed-forward neural network we need to represent the inputs as a design/feature matrix $X = (n_{inputs}, n_{features})$. Each row represents an *input*, in this case a handwritten digit, and each column represents a *feature*, in this case a pixel. The correct answers, also known as *labels* or *targets* are represented as a 1D array of integers $Y = (n_{inputs}) = (5, 3, 1, 8, ...)$.

As an example, say we want to build a neural network using supervised learning to predict Body-Mass Index (BMI) from measurements of height (in m)and weight (in kg). If we have measurements of 5 people the design/feature matrix could be for example:

$$X = \begin{bmatrix} 1.85 & 81 \\ 1.71 & 65 \\ 1.95 & 103 \\ 1.55 & 42 \\ 1.63 & 56 \end{bmatrix},$$

and the targets would be:

$$Y = (23.7, 22.2, 27.1, 17.5, 21.1)$$

Since each input image is a 2D matrix, we need to flatten the image (i.e. "unravel" the 2D matrix into a 1D array) to turn the data into a design/feature matrix. This means we lose all spatial information in the image, such as locality and translational invariance. More complicated architectures such as Convolutional Neural Networks can take advantage of such information, and are most commonly applied when analyzing images.

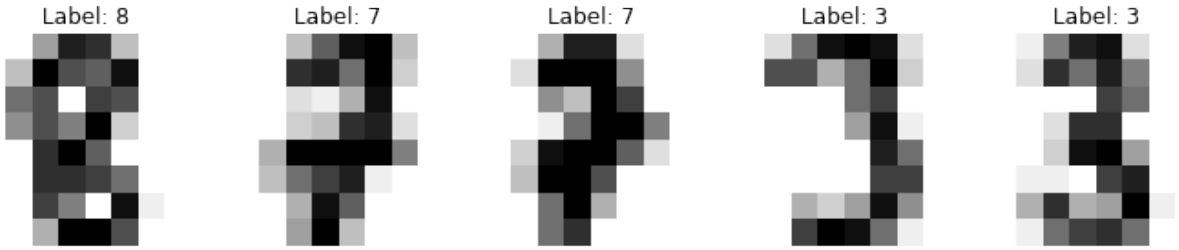
```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
```

(continues on next page)

(continued from previous page)

```
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

print("inputs = (n_inputs, pixel_width, pixel_height) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))


# flatten the image
# the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64
n_inputs = len(inputs)
inputs = inputs.reshape(n_inputs, -1)
print("X = (n_inputs, n_features) = " + str(inputs.shape))


# choose some random images to display
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

```
inputs = (n_inputs, pixel_width, pixel_height) = (1797, 8, 8)
labels = (n_inputs) = (1797,)
X = (n_inputs, n_features) = (1797, 64)
```



6.5.33 Train and test datasets

Performing analysis before partitioning the dataset is a major error, that can lead to incorrect conclusions.

We will reserve 80% of our dataset for training and 20% for testing.

It is important that the train and test datasets are drawn randomly from our dataset, to ensure no bias in the sampling.Say you are taking measurements of weather data to predict the weather in the coming 5 days. You don't want to train your model on measurements taken from the hours 00.00 to 12.00, and then test it on data collected from 12.00 to 24.00.

```
from sklearn.model_selection import train_test_split

# one-liner from scikit-learn library
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_
↪size,
                                                    test_size=test_size)

# equivalently in numpy
def train_test_split_numpy(inputs, labels, train_size, test_size):
    n_inputs = len(inputs)
    inputs_shuffled = inputs.copy()
    labels_shuffled = labels.copy()

    np.random.shuffle(inputs_shuffled)
    np.random.shuffle(labels_shuffled)

    train_end = int(n_inputs*train_size)
    X_train, X_test = inputs_shuffled[:train_end], inputs_shuffled[train_end:]
    Y_train, Y_test = labels_shuffled[:train_end], labels_shuffled[train_end:]

    return X_train, X_test, Y_train, Y_test

#X_train, X_test, Y_train, Y_test = train_test_split_numpy(inputs, labels, train_size,
↪ test_size)

print("Number of training images: " + str(len(X_train)))
print("Number of test images: " + str(len(X_test)))
```

```
Number of training images: 1437
Number of test images: 360
```

6.5.34 Define model and architecture

Our simple feed-forward neural network will consist of an *input* layer, a single *hidden* layer and an *output* layer. The activation y of each neuron is a weighted sum of inputs, passed through an activation function. In case of the simple perceptron model we have

$$z = \sum_{i=1}^{n} w_i a_i,$$

$$y = f(z),$$

where f is the activation function, a_i represents input from neuron i in the preceding layer and w_i is the weight to input i.The activation of the neurons in the input layer is just the features (e.g. a pixel value).

The simplest activation function for a neuron is the *Heaviside* function:

$$f(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

A feed-forward neural network with this activation is known as a *perceptron*.For a binary classifier (i.e. two classes, 0 or 1, dog or not-dog) we can also use this in our output layer.This activation can be generalized to k classes (using e.g. the *one-against-all* strategy), and we call these architectures *multiclass perceptrons*.

However, it is now common to use the terms Single Layer Perceptron (SLP) (1 hidden layer) andMultilayer Perceptron (MLP) (2 or more hidden layers) to refer to feed-forward neural networks with any activation function.

Typical choices for activation functions include the sigmoid function, hyperbolic tangent, and Rectified Linear Unit (ReLU).We will be using the sigmoid function $\sigma(x)$:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}},$$

which is inspired by probability theory (see logistic regression) and was most commonly used until about 2011. See the discussion below concerning other activation functions.

6.5.35 Layers

- Input

Since each input image has 8x8 = 64 pixels or features, we have an input layer of 64 neurons.

- Hidden layer

We will use 50 neurons in the hidden layer receiving input from the neurons in the input layer.Since each neuron in the hidden layer is connected to the 64 inputs we have 64x50 = 3200 weights to the hidden layer.

- Output

If we were building a binary classifier, it would be sufficient with a single neuron in the output layer, which could output 0 or 1 according to the Heaviside function. This would be an example of a *hard* classifier, meaning it outputs the class of the input directly. However, if we are dealing with noisy data it is often beneficial to use a *soft* classifier, which outputs the probability of being in class 0 or 1.

For a soft binary classifier, we could use a single neuron and interpret the output as either being the probability of being in class 0 or the probability of being in class 1. Alternatively we could use 2 neurons, and interpret each neuron as the probability of being in each class.

Since we are doing multiclass classification, with 10 categories, it is natural to use 10 neurons in the output layer. We number the neurons $j = 0, 1, ..., 9$. The activation of each output neuron j will be according to the *softmax* function:

$$P(\text{class } j \mid \text{input } \hat{a}) = \frac{\exp\left(\hat{a}^T \hat{w}_j\right)}{\sum_{c=0}^{9} \exp\left(\hat{a}^T \hat{w}_c\right)},$$

i.e. each neuron j outputs the probability of being in class j given an input from the hidden layer $\hat{a}$, with $\hat{w}_j$ the weights of neuron j to the inputs.The denominator is a normalization factor to ensure the outputs (probabilities) sum up to 1.The exponent is just the weighted sum of inputs as before:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

Since each neuron in the output layer is connected to the 50 inputs from the hidden layer we have 50x10 = 500 weights to the output layer.

6.5.36 Weights and biases

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias unit has an output of 1, and a weight to each neuron j, b_j:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

The bias weights $\hat{b}$ are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

```
# building our neural network

n_inputs, n_features = X_train.shape
n_hidden_neurons = 50
n_categories = 10

# we make the weights normally distributed using numpy.random.randn

# weights and bias in the hidden layer
hidden_weights = np.random.randn(n_features, n_hidden_neurons)
hidden_bias = np.zeros(n_hidden_neurons) + 0.01

# weights and bias in the output layer
output_weights = np.random.randn(n_hidden_neurons, n_categories)
output_bias = np.zeros(n_categories) + 0.01
```

6.5.37 Feed-forward pass

Denote F the number of features, H the number of hidden neurons and C the number of categories.For each input image we calculate a weighted sum of input features (pixel values) to each neuron j in the hidden layer l:

$$z_j^l = \sum_{i=1}^{F} w_{ij}^l x_i + b_j^l,$$

this is then passed through our activation function

$$a_j^l = f(z_j^l).$$

We calculate a weighted sum of inputs (activations in the hidden layer) to each neuron j in the output layer:

$$z_j^L = \sum_{i=1}^{H} w_{ij}^L a_i^l + b_j^L.$$

Finally we calculate the output of neuron j in the output layer using the softmax function:

$$a_j^L = \frac{\exp\left(z_j^L\right)}{\sum_{c=0}^{C-1} \exp\left(z_c^L\right)}.$$

6.5.38 Matrix multiplications

Since our data has the dimensions $X = (n_{inputs}, n_{features})$ and our weights to the hidden layer have the dimensions$W_{hidden} = (n_{features}, n_{hidden})$, we can easily feed the network all our training data in one go by taking the matrix product

$$XW^h = (n_{inputs}, n_{hidden}),$$

and obtain a matrix that holds the weighted sum of inputs to the hidden layer for each input image and each hidden neuron.We also add the bias to obtain a matrix of weighted sums to the hidden layer Z^h:

$$\hat{z}^l = \hat{X}\hat{W}^l + \hat{b}^l,$$

meaning the same bias (1D array with size equal number of hidden neurons) is added to each input image.This is then passed through the activation:

$$\hat{a}^l = f(\hat{z}^l).$$

This is fed to the output layer:

$$\hat{z}^L = \hat{a}^L\hat{W}^L + \hat{b}^L.$$

Finally we receive our output values for each image and each category by passing it through the softmax function:

$$output = softmax(\hat{z}^L) = (n_{inputs}, n_{categories}).$$

```
# setup the feed-forward pass, subscript h = hidden layer

def sigmoid(x):
    return 1/(1 + np.exp(-x))

def feed_forward(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    return probabilities

probabilities = feed_forward(X_train)
print("probabilities = (n_inputs, n_categories) = " + str(probabilities.shape))
print("probability that image 0 is in category 0,1,2,...,9 = \n" +␣
↪str(probabilities[0]))
print("probabilities sum up to: " + str(probabilities[0].sum()))
print()

# we obtain a prediction by taking the class with the highest likelihood
def predict(X):
    probabilities = feed_forward(X)
```

(continues on next page)

(continued from previous page)

```
    return np.argmax(probabilities, axis=1)

predictions = predict(X_train)
print("predictions = (n_inputs) = " + str(predictions.shape))
print("prediction for image 0: " + str(predictions[0]))
print("correct label for image 0: " + str(Y_train[0]))
```

```
probabilities = (n_inputs, n_categories) = (1437, 10)
probability that image 0 is in category 0,1,2,...,9 =
[5.41511965e-04 2.17174962e-03 8.84355903e-03 1.44970586e-03
 1.10378326e-04 5.08318298e-09 2.03256632e-04 1.92507116e-03
 9.84443254e-01 3.11507992e-04]
probabilities sum up to: 1.0

predictions = (n_inputs) = (1437,)
prediction for image 0: 8
correct label for image 0: 6
```

6.5.39 Choose cost function and optimizer

To measure how well our neural network is doing we need to introduce a cost function.We will call the function that gives the error of a single sample output the *loss* function, and the function that gives the total error of our network across all samples the *cost* function. A typical choice for multiclass classification is the *cross-entropy* loss, also known as the negative log likelihood.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$$y = 5 \quad \rightarrow \quad \hat{y} = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0),$$

$$y = 1 \quad \rightarrow \quad \hat{y} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0),$$

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset.

Let y_{ic} denote the c-th component of the i-th one-hot vector.We define the cost function $\mathcal{C}$ as a sum over the cross-entropy loss for each point $\hat{x}_i$ in the dataset.

In the one-hot representation only one of the terms in the loss function is non-zero, namely the probability of the correct category c'(i.e. the category c' such that $y_{ic'} = 1$). This means that the cross entropy loss only punishes you for how wrong you got the correct label. The probability of category c is given by the softmax function. The vector $\hat{\theta}$ represents the parameters of our network, i.e. all the weights and biases.

6.5.40 Optimizing the cost function

The network is trained by finding the weights and biases that minimize the cost function. One of the most widely used classes of methods is *gradient descent* and its generalizations. The idea behind gradient descent is simply to adjust the weights in the direction where the gradient of the cost function is large and negative. This ensures we flow toward a *local* minimum of the cost function.Each parameter θ is iteratively adjusted according to the rule

$$\theta_{i+1} = \theta_i - \eta \nabla \mathcal{C}(\theta_i),$$

where η is known as the *learning rate*, which controls how big a step we take towards the minimum.This update can be repeated for any number of iterations, or until we are satisfied with the result.

A simple and effective improvement is a variant called *Batch Gradient Descent*.Instead of calculating the gradient on the whole dataset, we calculate an approximation of the gradient on a subset of the data called a *minibatch*.If there are

N data points and we have a minibatch size of M, the total number of batches is N/M.We denote each minibatch B_k, with $k = 1, 2, ..., N/M$. The gradient then becomes:

$$\nabla \mathcal{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla \mathcal{L}_i(\theta) \quad \rightarrow \quad \frac{1}{M}\sum_{i \in B_k} \nabla \mathcal{L}_i(\theta),$$

i.e. instead of averaging the loss over the entire dataset, we average over a minibatch.

This has two important benefits:

1. Introducing stochasticity decreases the chance that the algorithm becomes stuck in a local minima.
2. It significantly speeds up the calculation, since we do not have to use the entire dataset to calculate the gradient.

The various optmization methods, with codes and algorithms, are discussed in our lectures on Gradient descent approaches.

6.5.41 Regularization

It is common to add an extra term to the cost function, proportional to the size of the weights. This is equivalent to constraining the size of the weights, so that they do not grow out of control. Constraining the size of the weights means that the weights cannot grow arbitrarily large to fit the training data, and in this way reduces *overfitting*.

We will measure the size of the weights using the so called *L2-norm*, meaning our cost function becomes:

$$\mathcal{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\theta) \quad \rightarrow \quad \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\theta) + \lambda ||\hat{w}||_2^2 = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\theta) + \lambda \sum_{ij} w_{ij}^2,$$

i.e. we sum up all the weights squared. The factor λ is known as a regularization parameter.

In order to train the model, we need to calculate the derivative of the cost function with respect to every bias and weight in the network. In total our network has $(64 + 1) \times 50 = 3250$ weights in the hidden layer and $(50 + 1) \times 10 = 510$ weights to the output layer ($+1$ for the bias), and the gradient must be calculated for every parameter. We use the *backpropagation* algorithm discussed above. This is a clever use of the chain rule that allows us to calculate the gradient efficently.

6.5.42 Matrix multiplication

To more efficently train our network these equations are implemented using matrix operations.The error in the output layer is calculated simply as, with $\hat{t}$ being our targets,

$$\delta_L = \hat{t} - \hat{y} = (n_{inputs}, n_{categories}).$$

The gradient for the output weights is calculated as

$$\nabla W_L = \hat{a}^T \delta_L = (n_{hidden}, n_{categories}),$$

where $\hat{a} = (n_{inputs}, n_{hidden})$. This simply means that we are summing up the gradients for each input.Since we are going backwards we have to transpose the activation matrix.

The gradient with respect to the output bias is then

$$\nabla \hat{b}_L = \sum_{i=1}^{n_{inputs}} \delta_L = (n_{categories}).$$

The error in the hidden layer is

$$\Delta_h = \delta_L W_L^T \circ f'(z_h) = \delta_L W_L^T \circ a_h \circ (1 - a_h) = (n_{inputs}, n_{hidden}),$$

where $f'(a_h)$ is the derivative of the activation in the hidden layer. The matrix products mean that we are summing up the products for each neuron in the output layer. The symbol $\circ$ denotes the *Hadamard product*, meaning element-wise multiplication.

This again gives us the gradients in the hidden layer:

$$\nabla W_h = X^T \delta_h = (n_{features}, n_{hidden}),$$

$$\nabla b_h = \sum_{i=1}^{n_{inputs}} \delta_h = (n_{hidden}).$$

```
# to categorical turns our integer vector into a onehot representation
from sklearn.metrics import accuracy_score

# one-hot in numpy
def to_categorical_numpy(integer_vector):
    n_inputs = len(integer_vector)
    n_categories = np.max(integer_vector) + 1
    onehot_vector = np.zeros((n_inputs, n_categories))
    onehot_vector[range(n_inputs), integer_vector] = 1

    return onehot_vector

#Y_train_onehot, Y_test_onehot = to_categorical(Y_train), to_categorical(Y_test)
Y_train_onehot, Y_test_onehot = to_categorical_numpy(Y_train), to_categorical_numpy(Y_
↪test)

def feed_forward_train(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    # for backpropagation need activations in hidden and output layers
    return a_h, probabilities

def backpropagation(X, Y):
    a_h, probabilities = feed_forward_train(X)

    # error in the output layer
    error_output = probabilities - Y
    # error in the hidden layer
    error_hidden = np.matmul(error_output, output_weights.T) * a_h * (1 - a_h)

    # gradients for the output layer
    output_weights_gradient = np.matmul(a_h.T, error_output)
```

(continues on next page)

(continued from previous page)

```
    output_bias_gradient = np.sum(error_output, axis=0)

    # gradient for the hidden layer
    hidden_weights_gradient = np.matmul(X.T, error_hidden)
    hidden_bias_gradient = np.sum(error_hidden, axis=0)

    return output_weights_gradient, output_bias_gradient, hidden_weights_gradient,␣
↪hidden_bias_gradient

print("Old accuracy on training data: " + str(accuracy_score(predict(X_train), Y_
↪train)))

eta = 0.01
lmbd = 0.01
for i in range(1000):
    # calculate gradients
    dWo, dBo, dWh, dBh = backpropagation(X_train, Y_train_onehot)

    # regularization term gradients
    dWo += lmbd * output_weights
    dWh += lmbd * hidden_weights

    # update weights and biases
    output_weights -= eta * dWo
    output_bias -= eta * dBo
    hidden_weights -= eta * dWh
    hidden_bias -= eta * dBh

print("New accuracy on training data: " + str(accuracy_score(predict(X_train), Y_
↪train)))
```

```
Old accuracy on training data: 0.1440501043841336
```

```
<ipython-input-5-16b8e3cda33a>:4: RuntimeWarning: overflow encountered in exp
  return 1/(1 + np.exp(-x))
```

```
New accuracy on training data: 0.09951287404314545
```

6.5.43 Improving performance

As we can see the network does not seem to be learning at all. It seems to be just guessing the label for each image.In order to obtain a network that does something useful, we will have to do a bit more work.

The choice of *hyperparameters* such as learning rate and regularization parameter is hugely influential for the performance of the network. Typically a *grid-search* is performed, wherein we test different hyperparameters separated by orders of magnitude. For example we could test the learning rates $\eta = 10^{-6}, 10^{-5}, ..., 10^{-1}$ with different regularization parameters $\lambda = 10^{-6}, ..., 10^{-0}$.

Next, we haven't implemented minibatching yet, which introduces stochasticity and is though to act as an important regularizer on the weights. We call a feed-forward + backward pass with a minibatch an *iteration*, and a full training period going through the entire dataset (n/M batches) an *epoch*.

If this does not improve network performance, you may want to consider altering the network architecture, adding more neurons or hidden layers.Andrew Ng goes through some of these considerations in this video. You can find a summary of the video here.

6.5.44 Full object-oriented implementation

It is very natural to think of the network as an object, with specific instances of the network being realizations of this object with different hyperparameters. An implementation using Python classes provides a clean structure and interface, and the full implementation of our neural network is given below.

```
class NeuralNetwork:
    def __init__(
            self,
            X_data,
            Y_data,
            n_hidden_neurons=50,
            n_categories=10,
            epochs=10,
            batch_size=100,
            eta=0.1,
            lmbd=0.0):

        self.X_data_full = X_data
        self.Y_data_full = Y_data

        self.n_inputs = X_data.shape[0]
        self.n_features = X_data.shape[1]
        self.n_hidden_neurons = n_hidden_neurons
        self.n_categories = n_categories

        self.epochs = epochs
        self.batch_size = batch_size
        self.iterations = self.n_inputs // self.batch_size
        self.eta = eta
        self.lmbd = lmbd

        self.create_biases_and_weights()

    def create_biases_and_weights(self):
        self.hidden_weights = np.random.randn(self.n_features, self.n_hidden_neurons)
        self.hidden_bias = np.zeros(self.n_hidden_neurons) + 0.01

        self.output_weights = np.random.randn(self.n_hidden_neurons, self.n_
↪categories)
        self.output_bias = np.zeros(self.n_categories) + 0.01

    def feed_forward(self):
        # feed-forward for training
        self.z_h = np.matmul(self.X_data, self.hidden_weights) + self.hidden_bias
        self.a_h = sigmoid(self.z_h)

        self.z_o = np.matmul(self.a_h, self.output_weights) + self.output_bias

        exp_term = np.exp(self.z_o)
        self.probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    def feed_forward_out(self, X):
        # feed-forward for output
        z_h = np.matmul(X, self.hidden_weights) + self.hidden_bias
        a_h = sigmoid(z_h)

        z_o = np.matmul(a_h, self.output_weights) + self.output_bias
```

(continues on next page)

(continued from previous page)

```
        exp_term = np.exp(z_o)
        probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)
        return probabilities

    def backpropagation(self):
        error_output = self.probabilities - self.Y_data
        error_hidden = np.matmul(error_output, self.output_weights.T) * self.a_h * (1␣
↪- self.a_h)

        self.output_weights_gradient = np.matmul(self.a_h.T, error_output)
        self.output_bias_gradient = np.sum(error_output, axis=0)

        self.hidden_weights_gradient = np.matmul(self.X_data.T, error_hidden)
        self.hidden_bias_gradient = np.sum(error_hidden, axis=0)

        if self.lmbd > 0.0:
            self.output_weights_gradient += self.lmbd * self.output_weights
            self.hidden_weights_gradient += self.lmbd * self.hidden_weights

        self.output_weights -= self.eta * self.output_weights_gradient
        self.output_bias -= self.eta * self.output_bias_gradient
        self.hidden_weights -= self.eta * self.hidden_weights_gradient
        self.hidden_bias -= self.eta * self.hidden_bias_gradient

    def predict(self, X):
        probabilities = self.feed_forward_out(X)
        return np.argmax(probabilities, axis=1)

    def predict_probabilities(self, X):
        probabilities = self.feed_forward_out(X)
        return probabilities

    def train(self):
        data_indices = np.arange(self.n_inputs)

        for i in range(self.epochs):
            for j in range(self.iterations):
                # pick datapoints with replacement
                chosen_datapoints = np.random.choice(
                    data_indices, size=self.batch_size, replace=False
                )

                # minibatch training data
                self.X_data = self.X_data_full[chosen_datapoints]
                self.Y_data = self.Y_data_full[chosen_datapoints]

                self.feed_forward()
                self.backpropagation()
```

6.5.45 Evaluate model performance on test data

To measure the performance of our network we evaluate how well it does it data it has never seen before, i.e. the test data.We measure the performance of the network using the *accuracy* score.The accuracy is as you would expect just the number of images correctly labeled divided by the total number of images. A perfect classifier will have an accuracy score of 1.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(\hat{y}_i = y_i)}{n},$$

where I is the indicator function, 1 if $\hat{y}_i = y_i$ and 0 otherwise.

```
epochs = 100
batch_size = 100

dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmbd, epochs=epochs, batch_
↪size=batch_size,
                    n_hidden_neurons=n_hidden_neurons, n_categories=n_categories)
dnn.train()
test_predict = dnn.predict(X_test)

# accuracy score from scikit library
print("Accuracy score on test set: ", accuracy_score(Y_test, test_predict))

# equivalent in numpy
def accuracy_score_numpy(Y_test, Y_pred):
    return np.sum(Y_test == Y_pred) / len(Y_test)

#print("Accuracy score on test set: ", accuracy_score_numpy(Y_test, test_predict))
```

```
Accuracy score on test set:  0.9416666666666667
```

6.5.46 Adjust hyperparameters

We now perform a grid search to find the optimal hyperparameters for the network.Note that we are only using 1 layer with 50 neurons, and human performance is estimated to be around 98% (2% error rate).

```
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
# store the models for later use
DNN_numpy = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

# grid search
for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmbd,␣
↪epochs=epochs, batch_size=batch_size,
                            n_hidden_neurons=n_hidden_neurons, n_categories=n_
↪categories)
        dnn.train()

        DNN_numpy[i][j] = dnn

        test_predict = dnn.predict(X_test)

        print("Learning rate  = ", eta)
```

(continues on next page)

(continued from previous page)

```
        print("Lambda = ", lmbd)
        print("Accuracy score on test set: ", accuracy_score(Y_test, test_predict))
        print()
```

```
Learning rate  =  1e-05
Lambda =  1e-05
Accuracy score on test set:  0.11666666666666667
```

```
Learning rate  =  1e-05
Lambda =  0.0001
Accuracy score on test set:  0.20833333333333334
```

```
Learning rate  =  1e-05
Lambda =  0.001
Accuracy score on test set:  0.12222222222222222
```

```
Learning rate  =  1e-05
Lambda =  0.01
Accuracy score on test set:  0.14722222222222223
```

```
Learning rate  =  1e-05
Lambda =  0.1
Accuracy score on test set:  0.17777777777777778
```

```
Learning rate  =  1e-05
Lambda =  1.0
Accuracy score on test set:  0.16111111111111112
```

```
Learning rate  =  1e-05
Lambda =  10.0
Accuracy score on test set:  0.20277777777777778
```

```
Learning rate  =  0.0001
Lambda =  1e-05
Accuracy score on test set:  0.5305555555555556
```

```
Learning rate  =  0.0001
Lambda =  0.0001
Accuracy score on test set:  0.5944444444444444
```

```
Learning rate  =  0.0001
Lambda =  0.001
Accuracy score on test set:  0.5888888888888889
```

```
Learning rate  =  0.0001
Lambda =  0.01
Accuracy score on test set:  0.6111111111111112
```

```
Learning rate  =  0.0001
Lambda =  0.1
Accuracy score on test set:  0.5222222222222223
```

```
Learning rate  =  0.0001
Lambda =  1.0
Accuracy score on test set:  0.5555555555555556
```

```
Learning rate  =  0.0001
Lambda =  10.0
Accuracy score on test set:  0.8055555555555556
```

```
Learning rate  =  0.001
Lambda =  1e-05
Accuracy score on test set:  0.85
```

```
Learning rate  =  0.001
Lambda =  0.0001
Accuracy score on test set:  0.85
```

```
Learning rate  =  0.001
Lambda =  0.001
Accuracy score on test set:  0.875
```

```
Learning rate  =  0.001
Lambda =  0.01
Accuracy score on test set:  0.8666666666666667
```

```
Learning rate  =  0.001
Lambda =  0.1
Accuracy score on test set:  0.8638888888888889
```

```
Learning rate  =  0.001
Lambda =  1.0
Accuracy score on test set:  0.9555555555555556
```

```
Learning rate  =  0.001
Lambda =  10.0
Accuracy score on test set:  0.925
```

```
Learning rate  =  0.01
Lambda =  1e-05
Accuracy score on test set:  0.9416666666666667
```

```
Learning rate  =  0.01
Lambda =  0.0001
Accuracy score on test set:  0.9277777777777778
```

```
Learning rate  =  0.01
Lambda =  0.001
Accuracy score on test set:  0.9305555555555556
```

```
Learning rate  =  0.01
Lambda =  0.01
Accuracy score on test set:  0.9361111111111111
```

```
Learning rate  =  0.01
Lambda =  0.1
Accuracy score on test set:  0.9555555555555556
```

```
Learning rate  =  0.01
Lambda =  1.0
Accuracy score on test set:  0.95
```

```
Learning rate  =  0.01
Lambda =  10.0
Accuracy score on test set:  0.49166666666666664
```

```
<ipython-input-5-16b8e3cda33a>:4: RuntimeWarning: overflow encountered in exp
  return 1/(1 + np.exp(-x))
```

```
Learning rate  =  0.1
Lambda =  1e-05
Accuracy score on test set:  0.09166666666666666
```

```
Learning rate  =  0.1
Lambda =  0.0001
Accuracy score on test set:  0.10555555555555556
```

```
Learning rate  =  0.1
Lambda =  0.001
Accuracy score on test set:  0.08888888888888889
```

```
Learning rate  =  0.1
Lambda =  0.01
Accuracy score on test set:  0.08888888888888889
```

```
Learning rate  =  0.1
Lambda =  0.1
Accuracy score on test set:  0.08611111111111111
```

```
Learning rate  =  0.1
Lambda =  1.0
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  0.1
Lambda =  10.0
Accuracy score on test set:  0.09166666666666666
```

```
<ipython-input-7-2572e3a4b38d>:43: RuntimeWarning: overflow encountered in exp
  exp_term = np.exp(self.z_o)
<ipython-input-7-2572e3a4b38d>:44: RuntimeWarning: invalid value encountered in true_
↪divide
  self.probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)
```

```
Learning rate  =  1.0
Lambda =  1e-05
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  0.0001
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  0.001
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  0.01
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  0.1
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  1.0
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  1.0
Lambda =  10.0
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  1e-05
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  0.0001
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  0.001
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  0.01
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  0.1
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  1.0
Accuracy score on test set:  0.07777777777777778
```

```
Learning rate  =  10.0
Lambda =  10.0
Accuracy score on test set:  0.07777777777777778
```

6.5.47 Visualization

```
# visual representation of grid search
# uses seaborn heatmap, you can also do this with matplotlib imshow
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_numpy[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
        test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


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()

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

6.5.48 scikit-learn implementation

scikit-learn focuses more on traditional machine learning methods, such as regression, clustering, decision trees, etc. As such, it has only two types of neural networks: Multi Layer Perceptron outputting continuous values, *MPLRegressor*, and Multi Layer Perceptron outputting labels, *MLPClassifier*. We will see how simple it is to use these classes.

scikit-learn implements a few improvements from our neural network, such as early stopping, a varying learning rate, different optimization methods, etc. We would therefore expect a better performance overall.

```
from sklearn.neural_network import MLPClassifier
# store models for later use
DNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = MLPClassifier(hidden_layer_sizes=(n_hidden_neurons), activation=
↪'logistic',
                            alpha=lmbd, learning_rate_init=eta, max_iter=epochs)
        dnn.fit(X_train, Y_train)
```

(continues on next page)

(continued from previous page)

```
        DNN_scikit[i][j] = dnn

        print("Learning rate  = ", eta)
        print("Lambda = ", lmbd)
        print("Accuracy score on test set: ", dnn.score(X_test, Y_test))
        print()
```

6.5.49 Visualization

```
# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_scikit[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
        test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


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()

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

6.5.50 Building neural networks in Tensorflow and Keras

Now we want to build on the experience gained from our neural network implementation in NumPy and scikit-learn and use it to construct a neural network in Tensorflow. Once we have constructed a neural network in NumPy and Tensorflow, building one in Keras is really quite trivial, though the performance may suffer.

In our previous example we used only one hidden layer, and in this we will use two. From this it should be quite clear how to build one using an arbitrary number of hidden layers, using data structures such as Python lists or NumPy arrays.

6.5.51 Tensorflow

Tensorflow is an open source library machine learning library developed by the Google Brain team for internal use. It was released under the Apache 2.0 open source license in November 9, 2015.

Tensorflow is a computational framework that allows you to construct machine learning models at different levels of abstraction, from high-level, object-oriented APIs like Keras, down to the C++ kernels that Tensorflow is built upon. The higher levels of abstraction are simpler to use, but less flexible, and our choice of implementation should reflect the problems we are trying to solve.

Tensorflow uses so-called graphs to represent your computation in terms of the dependencies between individual operations, such that you first build a Tensorflow *graph* to represent your model, and then create a Tensorflow *session* to run the graph.

In this guide we will analyze the same data as we did in our NumPy and scikit-learn tutorial, gathered from the MNIST database of images. We will give an introduction to the lower level Python Application Program Interfaces (APIs), and see how we use them to build our graph. Then we will build (effectively) the same graph in Keras, to see just how simple solving a machine learning problem can be.

To install tensorflow on Unix/Linux systems, use pip as

```
pip3 install tensorflow
```

and/or if you use **anaconda**, just write (or install from the graphical user interface)

```
conda install tensorflow
```

6.5.52 Collect and pre-process data

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()
```

(continues on next page)

(continued from previous page)

```
# define inputs and labels
inputs = digits.images
labels = digits.target

print("inputs = (n_inputs, pixel_width, pixel_height) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))


# flatten the image
# the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64
n_inputs = len(inputs)
inputs = inputs.reshape(n_inputs, -1)
print("X = (n_inputs, n_features) = " + str(inputs.shape))


# choose some random images to display
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

```
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# one-hot representation of labels
labels = to_categorical(labels)

# split into train and test data
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_
↪size,
                                                    test_size=test_size)
```

6.5.53 Using TensorFlow backend

1. Define model and architecture
2. Choose cost function and optimizer

```
import tensorflow as tf

class NeuralNetworkTensorflow:
    def __init__(
            self,
            X_train,
            Y_train,
            X_test,
            Y_test,
```

(continues on next page)

(continued from previous page)

```
            n_neurons_layer1=100,
            n_neurons_layer2=50,
            n_categories=2,
            epochs=10,
            batch_size=100,
            eta=0.1,
            lmbd=0.0):

        # keep track of number of steps
        self.global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name=
↪'global_step')

        self.X_train = X_train
        self.Y_train = Y_train
        self.X_test = X_test
        self.Y_test = Y_test

        self.n_inputs = X_train.shape[0]
        self.n_features = X_train.shape[1]
        self.n_neurons_layer1 = n_neurons_layer1
        self.n_neurons_layer2 = n_neurons_layer2
        self.n_categories = n_categories

        self.epochs = epochs
        self.batch_size = batch_size
        self.iterations = self.n_inputs // self.batch_size
        self.eta = eta
        self.lmbd = lmbd

        # build network piece by piece
        # name scopes (with) are used to enforce creation of new variables
        # https://www.tensorflow.org/guide/variables
        self.create_placeholders()
        self.create_DNN()
        self.create_loss()
        self.create_optimiser()
        self.create_accuracy()

    def create_placeholders(self):
        # placeholders are fine here, but "Datasets" are the preferred method
        # of streaming data into a model
        with tf.name_scope('data'):
            self.X = tf.placeholder(tf.float32, shape=(None, self.n_features), name=
↪'X_data')
            self.Y = tf.placeholder(tf.float32, shape=(None, self.n_categories), name=
↪'Y_data')

    def create_DNN(self):
        with tf.name_scope('DNN'):
            # the weights are stored to calculate regularization loss later

            # Fully connected layer 1
            self.W_fc1 = self.weight_variable([self.n_features, self.n_neurons_
↪layer1], name='fc1', dtype=tf.float32)
            b_fc1 = self.bias_variable([self.n_neurons_layer1], name='fc1', dtype=tf.
↪float32)
            a_fc1 = tf.nn.sigmoid(tf.matmul(self.X, self.W_fc1) + b_fc1)
```

(continues on next page)

(continued from previous page)

```
            # Fully connected layer 2
            self.W_fc2 = self.weight_variable([self.n_neurons_layer1, self.n_neurons_
↪layer2], name='fc2', dtype=tf.float32)
            b_fc2 = self.bias_variable([self.n_neurons_layer2], name='fc2', dtype=tf.
↪float32)
            a_fc2 = tf.nn.sigmoid(tf.matmul(a_fc1, self.W_fc2) + b_fc2)

            # Output layer
            self.W_out = self.weight_variable([self.n_neurons_layer2, self.n_
↪categories], name='out', dtype=tf.float32)
            b_out = self.bias_variable([self.n_categories], name='out', dtype=tf.
↪float32)
            self.z_out = tf.matmul(a_fc2, self.W_out) + b_out

    def create_loss(self):
        with tf.name_scope('loss'):
            softmax_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_
↪v2(labels=self.Y, logits=self.z_out))

            regularizer_loss_fc1 = tf.nn.l2_loss(self.W_fc1)
            regularizer_loss_fc2 = tf.nn.l2_loss(self.W_fc2)
            regularizer_loss_out = tf.nn.l2_loss(self.W_out)
            regularizer_loss = self.lmbd*(regularizer_loss_fc1 + regularizer_loss_fc2␣
↪+ regularizer_loss_out)

            self.loss = softmax_loss + regularizer_loss

    def create_accuracy(self):
        with tf.name_scope('accuracy'):
            probabilities = tf.nn.softmax(self.z_out)
            predictions = tf.argmax(probabilities, axis=1)
            labels = tf.argmax(self.Y, axis=1)

            correct_predictions = tf.equal(predictions, labels)
            correct_predictions = tf.cast(correct_predictions, tf.float32)
            self.accuracy = tf.reduce_mean(correct_predictions)

    def create_optimiser(self):
        with tf.name_scope('optimizer'):
            self.optimizer = tf.train.GradientDescentOptimizer(learning_rate=self.
↪eta).minimize(self.loss, global_step=self.global_step)

    def weight_variable(self, shape, name='', dtype=tf.float32):
        initial = tf.truncated_normal(shape, stddev=0.1)
        return tf.Variable(initial, name=name, dtype=dtype)

    def bias_variable(self, shape, name='', dtype=tf.float32):
        initial = tf.constant(0.1, shape=shape)
        return tf.Variable(initial, name=name, dtype=dtype)

    def fit(self):
        data_indices = np.arange(self.n_inputs)

        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(self.epochs):
```

(continues on next page)

(continued from previous page)

```
                for j in range(self.iterations):
                    chosen_datapoints = np.random.choice(data_indices, size=self.
↪batch_size, replace=False)
                    batch_X, batch_Y = self.X_train[chosen_datapoints], self.Y_
↪train[chosen_datapoints]

                    sess.run([DNN.loss, DNN.optimizer],
                        feed_dict={DNN.X: batch_X,
                                   DNN.Y: batch_Y})
                    accuracy = sess.run(DNN.accuracy,
                        feed_dict={DNN.X: batch_X,
                                   DNN.Y: batch_Y})
                    step = sess.run(DNN.global_step)

            self.train_loss, self.train_accuracy = sess.run([DNN.loss, DNN.accuracy],
                feed_dict={DNN.X: self.X_train,
                           DNN.Y: self.Y_train})

            self.test_loss, self.test_accuracy = sess.run([DNN.loss, DNN.accuracy],
                feed_dict={DNN.X: self.X_test,
                           DNN.Y: self.Y_test})
```

6.5.54 Optimizing and using gradient descent

```
epochs = 100
batch_size = 100
n_neurons_layer1 = 100
n_neurons_layer2 = 50
n_categories = 10
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
```

```
DNN_tf = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        DNN = NeuralNetworkTensorflow(X_train, Y_train, X_test, Y_test,
                                      n_neurons_layer1, n_neurons_layer2, n_
↪categories,
                                      epochs=epochs, batch_size=batch_size, eta=eta,␣
↪lmbd=lmbd)
        DNN.fit()

        DNN_tf[i][j] = DNN

        print("Learning rate = ", eta)
        print("Lambda = ", lmbd)
        print("Test accuracy: %.3f" % DNN.test_accuracy)
        print()
```

```
# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        DNN = DNN_tf[i][j]

        train_accuracy[i][j] = DNN.train_accuracy
        test_accuracy[i][j] = DNN.test_accuracy


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()

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

```
# optional
# we can use log files to visualize our graph in Tensorboard
writer = tf.summary.FileWriter('logs/')
writer.add_graph(tf.get_default_graph())
```

6.5.55 Using Keras

Keras is a high level neural network that supports Tensorflow, CTNK and Theano as backends.If you have Tensorflow installed Keras is available through the *tf.keras* module.If you have Anaconda installed you may run the following command

```
conda install keras
```

Alternatively, if you have Tensorflow or one of the other supported backends install you may use the pip package manager:

```
pip3 install keras
```

or look up the instructions here.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l2
from keras.optimizers import SGD

def create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories, eta,
↪ lmbd):
```

(continues on next page)

(continued from previous page)

```
    model = Sequential()
    model.add(Dense(n_neurons_layer1, activation='sigmoid', kernel_
↪regularizer=l2(lmbd)))
    model.add(Dense(n_neurons_layer2, activation='sigmoid', kernel_
↪regularizer=l2(lmbd)))
    model.add(Dense(n_categories, activation='softmax'))

    sgd = SGD(lr=eta)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy
↪'])

    return model
```

```
DNN_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        DNN = create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_
↪categories,
                                          eta=eta, lmbd=lmbd)
        DNN.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)
        scores = DNN.evaluate(X_test, Y_test)

        DNN_keras[i][j] = DNN

        print("Learning rate = ", eta)
        print("Lambda = ", lmbd)
        print("Test accuracy: %.3f" % scores[1])
        print()
```

```
# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        DNN = DNN_keras[i][j]

        train_accuracy[i][j] = DNN.evaluate(X_train, Y_train)[1]
        test_accuracy[i][j] = DNN.evaluate(X_test, Y_test)[1]


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()

fig, ax = plt.subplots(figsize = (10, 10))
```

(continues on next page)

(continued from previous page)

```
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

6.5.56 Which activation function should I use?

The Back propagation algorithm we derived above works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent (GD) step.

Unfortunately for us, the gradients often get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is known in the literature as **the vanishing gradients problem**.

In other cases, the opposite can happen, namely the the gradients can grow bigger and bigger. The result is that many of the layers get large updates of the weights the algorithm diverges. This is the **exploding gradients problem**, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients, different layers may learn at widely different speeds

6.5.57 Is the Logistic activation function (Sigmoid) our choice?

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it.

A paper titled Understanding the Difficulty of Training Deep Feedforward Neural Networks by Xavier Glorot and Yoshua Bengio found that the problems with the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a normal distribution with a mean of 0 and a standard deviation of 1.

They showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

6.5.58 The derivative of the Logistic funtion

Looking at the logistic activation function, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

6.5.59 The RELU function family

The ReLU activation function suffers from a problem known as the dying ReLUs: during training, some neurons effectively die, meaning they stop outputting anything other than 0.

In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, nowadays practitioners use a variant of the ReLU function, such as the leaky ReLU discussed above or the so-called exponential linear unit (ELU) function

$$ELU(z) = \begin{cases} \alpha\left(\exp\left(z\right) - 1\right) & z < 0, \\ z & z \geq 0. \end{cases}$$

6.5.60 Which activation function should we use?

In general it seems that the ELU activation function is better than the leaky ReLU function (and its variants), which is better than ReLU. ReLU performs better than $\tanh$ which in turn performs better than the logistic function.

If runtime performance is an issue, then you may opt for the leaky ReLU function over the ELU function If you don't want to tweak yet another hyperparameter, you may just use the default α of 0.01 for the leaky ReLU, and 1 for ELU. If you have spare time and computing power, you can use cross-validation or bootstrap to evaluate other activation functions.

6.5.61 A top-down perspective on Neural networks

The first thing we would like to do is divide the data into two or three parts. A training set, a validation or dev (development) set, and a test set. The test set is the data on which we want to make predictions. The dev set is a subset of the training data we use to check how well we are doing out-of-sample, after training the model on the training dataset. We use the validation error as a proxy for the test error in order to make tweaks to our model. It is crucial that we do not use any of the test data to train the algorithm. This is a cardinal sin in ML. Then:

- Estimate optimal error rate
- Minimize underfitting (bias) on training data set.
- Make sure you are not overfitting.

If the validation and test sets are drawn from the same distributions, then a good performance on the validation set should lead to similarly good performance on the test set.

However, sometimes the training data and test data differ in subtle ways because, for example, they are collected using slightly different methods, or because it is cheaper to collect data in one way versus another. In this case, there can be a mismatch between the training and test data. This can lead to the neural network overfitting these small differences between the test and training sets, and a poor performance on the test set despite having a good performance on the validation set. To rectify this, Andrew Ng suggests making two validation or dev sets, one constructed from the training data and one constructed from the test data. The difference between the performance of the algorithm on these two

validation sets quantifies the train-test mismatch. This can serve as another important diagnostic when using DNNs for supervised learning.

6.5.62 Limitations of supervised learning with deep networks

Like all statistical methods, supervised learning using neural networks has important limitations. This is especially important when one seeks to apply these methods, especially to physics problems. Like all tools, DNNs are not a universal solution. Often, the same or better performance on a task can be achieved by using a few hand-engineered features (or even a collection of random features).

Here we list some of the important limitations of supervised neural network based models.

- **Need labeled data**. All supervised learning methods, DNNs for supervised learning require labeled data. Often, labeled data is harder to acquire than unlabeled data (e.g. one must pay for human experts to label images).
- **Supervised neural networks are extremely data intensive.** DNNs are data hungry. They perform best when data is plentiful. This is doubly so for supervised methods where the data must also be labeled. The utility of DNNs is extremely limited if data is hard to acquire or the datasets are small (hundreds to a few thousand samples). In this case, the performance of other methods that utilize hand-engineered features can exceed that of DNNs.
- **Homogeneous data.** Almost all DNNs deal with homogeneous data of one type. It is very hard to design architectures that mix and match data types (i.e. some continuous variables, some discrete variables, some time series). In applications beyond images, video, and language, this is often what is required. In contrast, ensemble models like random forests or gradient-boosted trees have no difficulty handling mixed data types.
- **Many problems are not about prediction.** In natural science we are often interested in learning something about the underlying distribution that generates the data. In this case, it is often difficult to cast these ideas in a supervised learning setting. While the problems are related, it is possible to make good predictions with a *wrong* model. The model might or might not be useful for understanding the underlying science.

Some of these remarks are particular to DNNs, others are shared by all supervised learning methods. This motivates the use of unsupervised methods which in part circumvent these problems.

6.6 Support Vector Machines, overarching aims

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning method, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small-sized or medium-sized datasets.

The case with two well-separated classes only can be understood in an intuitive way in terms of lines in a two-dimensional space separating the two classes (see figure below).

The basic mathematics behind the SVM is however less familiar to most of us. It relies on the definition of hyperplanes and the definition of a **margin** which separates classes (in case of classification problems) of variables. It is also used for regression problems.

With SVMs we distinguish between hard margin and soft margins. The latter introduces a so-called softening parameter to be discussed below. We distinguish also between linear and non-linear approaches. The latter are the most frequent ones since it is rather unlikely that we can separate classes easily by say straight lines.

6.6.1 Hyperplanes and all that

The theory behind support vector machines (SVM hereafter) is based on the mathematical description of so-called hyperplanes. Let us start with a two-dimensional case. This will also allow us to introduce our first SVM examples. These will be tailored to the case of two specific classes, as displayed in the figure here based on the usage of the petal data.

We assume here that our data set can be well separated into two domains, where a straight line does the job in the separating the two classes. Here the two classes are represented by either squares or circles.

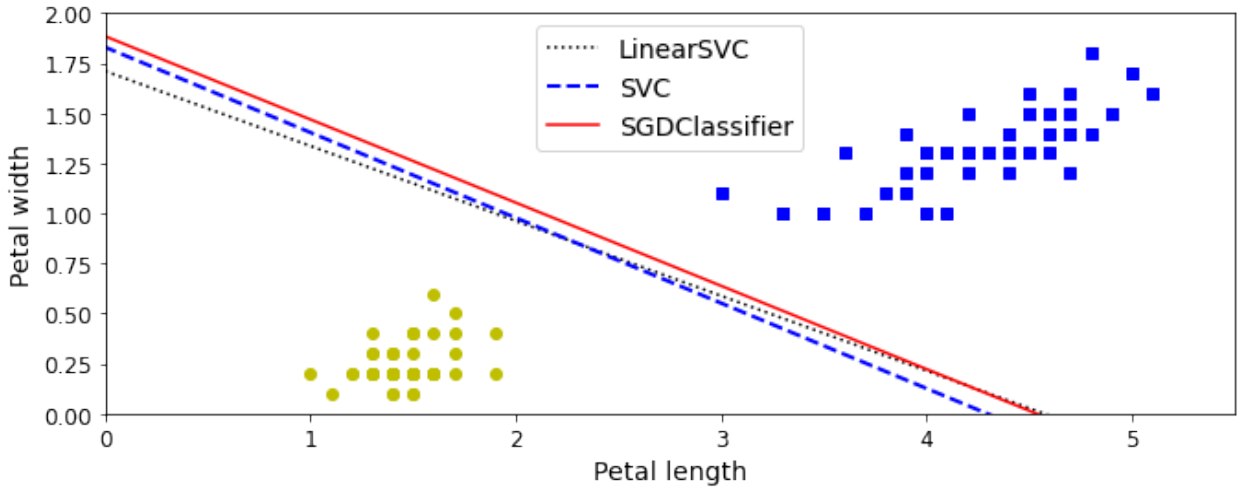
```
%matplotlib inline

from sklearn import datasets
from sklearn.svm import SVC, LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12


iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]



C = 5
alpha = 1 / (C * len(X))

lin_clf = LinearSVC(loss="hinge", C=C, random_state=42)
svm_clf = SVC(kernel="linear", C=C)
sgd_clf = SGDClassifier(loss="hinge", learning_rate="constant", eta0=0.001,␣
↪alpha=alpha,
                        max_iter=100000, random_state=42)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

lin_clf.fit(X_scaled, y)
svm_clf.fit(X_scaled, y)
sgd_clf.fit(X_scaled, y)

print("LinearSVC:                   ", lin_clf.intercept_, lin_clf.coef_)
print("SVC:                         ", svm_clf.intercept_, svm_clf.coef_)
print("SGDClassifier(alpha={:.5f}):".format(sgd_clf.alpha), sgd_clf.intercept_, sgd_
↪clf.coef_)

# Compute the slope and bias of each decision boundary
w1 = -lin_clf.coef_[0, 0]/lin_clf.coef_[0, 1]
b1 = -lin_clf.intercept_[0]/lin_clf.coef_[0, 1]
w2 = -svm_clf.coef_[0, 0]/svm_clf.coef_[0, 1]
```

(continues on next page)

(continued from previous page)

```
b2 = -svm_clf.intercept_[0]/svm_clf.coef_[0, 1]
w3 = -sgd_clf.coef_[0, 0]/sgd_clf.coef_[0, 1]
b3 = -sgd_clf.intercept_[0]/sgd_clf.coef_[0, 1]

# Transform the decision boundary lines back to the original scale
line1 = scaler.inverse_transform([[-10, -10 * w1 + b1], [10, 10 * w1 + b1]])
line2 = scaler.inverse_transform([[-10, -10 * w2 + b2], [10, 10 * w2 + b2]])
line3 = scaler.inverse_transform([[-10, -10 * w3 + b3], [10, 10 * w3 + b3]])

# Plot all three decision boundaries
plt.figure(figsize=(11, 4))
plt.plot(line1[:, 0], line1[:, 1], "k:", label="LinearSVC")
plt.plot(line2[:, 0], line2[:, 1], "b--", linewidth=2, label="SVC")
plt.plot(line3[:, 0], line3[:, 1], "r-", label="SGDClassifier")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs") # label="Iris-Versicolor"
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo") # label="Iris-Setosa"
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper center", fontsize=14)
plt.axis([0, 5.5, 0, 2])

plt.show()
```

```
LinearSVC:                    [0.28475098] [[1.05364854 1.09903804]]
SVC:                          [0.31896852] [[1.1203284  1.02625193]]
SGDClassifier(alpha=0.00200): [0.117] [[0.77714169 0.72981762]]
```



6.6.2 What is a hyperplane?

The aim of the SVM algorithm is to find a hyperplane in a p-dimensional space, where p is the number of features that distinctly classifies the data points.

In a p-dimensional space, a hyperplane is what we call an affine subspace of dimension of $p - 1$. As an example, in two dimension, a hyperplane is simply as straight line while in three dimensions it is a two-dimensional subspace, or stated simply, a plane.

In two dimensions, with the variables x_1 and x_2, the hyperplane is defined as

$$b + w_1 x_1 + w_2 x_2 = 0,$$

where b is the intercept and w_1 and w_2 define the elements of a vector orthogonal to the line $b + w_1 x_1 + w_2 x_2 = 0$. In two dimensions we define the vectors $\boldsymbol{x} = [x1, x2]$ and $\boldsymbol{w} = [w1, w2]$. We can then rewrite the above equation as

$$\boldsymbol{x}^T \boldsymbol{w} + b = 0.$$

6.6.3 A p-dimensional space of features

We limit ourselves to two classes of outputs y_i and assign these classes the values $y_i = \pm 1$. In a p-dimensional space of say p features we have a hyperplane defines as

$$b + w x_1 + w_2 x_2 + \cdots + w_p x_p = 0.$$

If we define a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p]$ of dimension $n \times p$, where n represents the observations for each feature and each vector x_i is a column vector of the matrix $\boldsymbol{X}$,

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \cdots \\ \cdots \\ x_{ip} \end{bmatrix}.$$

If the above condition is not met for a given vector $\boldsymbol{x}_i$ we have

$$b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} > 0,$$

if our output $y_i = 1$. In this case we say that $\boldsymbol{x}_i$ lies on one of the sides of the hyperplane and if

$$b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} < 0,$$

for the class of observations $y_i = -1$, then $\boldsymbol{x}_i$ lies on the other side.

Equivalently, for the two classes of observations we have

$$y_i \left(b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip}\right) > 0.$$

When we try to separate hyperplanes, if it exists, we can use it to construct a natural classifier: a test observation is assigned a given class depending on which side of the hyperplane it is located.

6.6.4 The two-dimensional case

Let us try to develop our intuition about SVMs by limiting ourselves to a two-dimensional plane. To separate the two classes of data points, there are many possible lines (hyperplanes if you prefer a more strict naming)that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

What a linear classifier attempts to accomplish is to split the feature space into two half spaces by placing a hyperplane between the data points. This hyperplane will be our decision boundary. All points on one side of the plane will belong to class one and all points on the other side of the plane will belong to the second class two.

Unfortunately there are many ways in which we can place a hyperplane to divide the data. Below is an example of two candidate hyperplanes for our data sample.

6.6.5 Getting into the details

Let us define the function

$$f(x) = \boldsymbol{w}^T\boldsymbol{x} + b = 0,$$

as the function that determines the line L that separates two classes (our two features), see the figure here.

Any point defined by $\boldsymbol{x}_i$ and $\boldsymbol{x}_2$ on the line L will satisfy $\boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$.

The signed distance δ from any point defined by a vector $\boldsymbol{x}$ and a point $\boldsymbol{x}_0$ on the line L is then

$$\delta = \frac{1}{||\boldsymbol{w}||}(\boldsymbol{w}^T\boldsymbol{x} + b).$$

6.6.6 First attempt at a minimization approach

How do we find the parameter b and the vector $\boldsymbol{w}$? What we could do is to define a cost function which now contains the set of all misclassified points M and attempt to minimize this function

$$C(\boldsymbol{w}, b) = -\sum_{i \in M} y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b).$$

We could now for example define all values $y_i = 1$ as misclassified in case we have $\boldsymbol{w}^T\boldsymbol{x}_i + b < 0$ and the opposite if we have $y_i = -1$. Taking the derivatives gives us

$$\frac{\partial C}{\partial b} = -\sum_{i \in M} y_i,$$

and

$$\frac{\partial C}{\partial \boldsymbol{w}} = -\sum_{i \in M} y_i x_i.$$

6.6.7 Solving the equations

We can now use the Newton-Raphson method or different variants of the gradient descent family (from plain gradient descent to various stochastic gradient descent approaches) to solve the equations

$$b \leftarrow b + \eta\frac{\partial C}{\partial b},$$

and

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\frac{\partial C}{\partial \boldsymbol{w}},$$

where η is our by now well-known learning rate.

6.6.8 Code Example

The equations we discussed above can be coded rather easily (the framework is similar to what we developed for logistic regression). We are going to set up a simple case with two classes only and we want to find a line which separates them the best possible way.

6.6.9 Problems with the Simpler Approach

There are however problems with this approach, although it looks pretty straightforward to implement. When running the above code, we see that we can easily end up with many diffeent lines which separate the two classes.

For small gaps between the entries, we may also end up needing many iterations before the solutions converge and if the data cannot be separated properly into two distinct classes, we may not experience a converge at all.

6.6.10 A better approach

A better approach is rather to try to define a large margin between the two classes (if they are well separated from the beginning).

Thus, we wish to find a margin M with $\boldsymbol{w}$ normalized to $||\boldsymbol{w}|| = 1$ subject to the condition

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M \; \forall i = 1, 2, \ldots, p.$$

All points are thus at a signed distance from the decision boundary defined by the line L. The parameters b and w_1 and w_2 define this line.

We seek thus the largest value M defined by

$$\frac{1}{||\boldsymbol{w}||} y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M \; \forall i = 1, 2, \ldots, n,$$

or just

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M||\boldsymbol{w}|| \; \forall i.$$

If we scale the equation so that $||\boldsymbol{w}|| = 1/M$, we have to find the minimum of $\boldsymbol{w}^T\boldsymbol{w} = ||\boldsymbol{w}||$ (the norm) subject to the condition

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 \; \forall i.$$

We have thus defined our margin as the invers of the norm of $\boldsymbol{w}$. We want to minimize the norm in order to have a as large as possible margin M. Before we proceed, we need to remind ourselves about Lagrangian multipliers.

6.6.11 A quick Reminder on Lagrangian Multipliers

Consider a function of three independent variables $f(x, y, z)$. For the function f to be an extreme we have

$$df = 0.$$

A necessary and sufficient condition is

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

due to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz.$$

In many problems the variables x, y, z are often subject to constraints (such as those above for the margin) so that they are no longer all independent. It is possible at least in principle to use each constraint to eliminate one variable and to proceed with a new and smaller set of independent varables.

The use of so-called Lagrangian multipliers is an alternative technique when the elimination of variables is incovenient or undesirable. Assume that we have an equation of constraint on the variables x, y, z

$$\phi(x, y, z) = 0,$$

resulting in

$$d\phi = \frac{\partial \phi}{\partial x}dx + \frac{\partial \phi}{\partial y}dy + \frac{\partial \phi}{\partial z}dz = 0.$$

Now we cannot set anymore

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

if $df = 0$ is wanted because there are now only two independent variables! Assume x and y are the independent variables. Then dz is no longer arbitrary.

6.6.12 Adding the Multiplier

However, we can add to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz,$$

a multiplum of $d\phi$, viz. $\lambda d\phi$, resulting in

$$df + \lambda d\phi = (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial x})dx + (\frac{\partial f}{\partial y} + \lambda\frac{\partial \phi}{\partial y})dy + (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z})dz = 0.$$

Our multiplier is chosen so that

$$\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z} = 0.$$

We need to remember that we took dx and dy to be arbitrary and thus we must have

$$\frac{\partial f}{\partial x} + \lambda\frac{\partial \phi}{\partial x} = 0,$$

and

$$\frac{\partial f}{\partial y} + \lambda\frac{\partial \phi}{\partial y} = 0.$$

When all these equations are satisfied, $df = 0$. We have four unknowns, x, y, z and λ. Actually we want only x, y, z, λ needs not to be determined, it is therefore often called Lagrange's undetermined multiplier. If we have a set of constraints ϕ_k we have the equations

$$\frac{\partial f}{\partial x_i} + \sum_k \lambda_k \frac{\partial \phi_k}{\partial x_i} = 0.$$

6.6.13 Setting up the Problem

In order to solve the above problem, we define the following Lagrangian function to be minimized

$$\mathcal{L}(\lambda, b, \boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{n} \lambda_i \left[y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1\right],$$

where λ_i is a so-called Lagrange multiplier subject to the condition $\lambda_i \geq 0$.

Taking the derivatives with respect to b and $\boldsymbol{w}$ we obtain

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \lambda_i y_i = 0,$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 = \boldsymbol{w} - \sum_i \lambda_i y_i \boldsymbol{x}_i.$$

Inserting these constraints into the equation for $\mathcal{L}$ we obtain

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j,$$

subject to the constraints $\lambda_i \geq 0$ and $\sum_i \lambda_i y_i = 0$. We must in addition satisfy the Karush-Kuhn-Tucker (KKT) condition

$$\lambda_i \left[y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - 1 \right] \ \forall i.$$

1. If $\lambda_i > 0$, then $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1$ and we say that x_i is on the boundary.
2. If $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) > 1$, we say x_i is not on the boundary and we set $\lambda_i = 0$.

When $\lambda_i > 0$, the vectors $\boldsymbol{x}_i$ are called support vectors. They are the vectors closest to the line (or hyperplane) and define the margin M.

6.6.14 The problem to solve

We can rewrite

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j,$$

and its constraints in terms of a matrix-vector problem where we minimize w.r.t. λ the following problem

$$\frac{1}{2}\boldsymbol{\lambda}^T \begin{bmatrix} y_1 y_1 \boldsymbol{x}_1^T \boldsymbol{x}_1 & y_1 y_2 \boldsymbol{x}_1^T \boldsymbol{x}_2 & \dots & \dots & y_1 y_n \boldsymbol{x}_1^T \boldsymbol{x}_n \\ y_2 y_1 \boldsymbol{x}_2^T \boldsymbol{x}_1 & y_2 y_2 \boldsymbol{x}_2^T \boldsymbol{x}_2 & \dots & \dots & y_1 y_n \boldsymbol{x}_2^T \boldsymbol{x}_n \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ y_n y_1 \boldsymbol{x}_n^T \boldsymbol{x}_1 & y_n y_2 \boldsymbol{x}_n^T \boldsymbol{x}_2 & \dots & \dots & y_n y_n \boldsymbol{x}_n^T \boldsymbol{x}_n \end{bmatrix} \boldsymbol{\lambda} - \mathbb{1}\boldsymbol{\lambda},$$

subject to $\boldsymbol{y}^T \boldsymbol{\lambda} = 0$. Here we defined the vectors $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_n]$ and $\boldsymbol{y} = [y_1, y_2, \dots, y_n]$.

6.6.15 The last steps

Solving the above problem, yields the values of λ_i. To find the coefficients of your hyperplane we need simply to compute

$$\boldsymbol{w} = \sum_i \lambda_i y_i \boldsymbol{x}_i.$$

With our vector $\boldsymbol{w}$ we can in turn find the value of the intercept b (here in two dimensions) via

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1,$$

resulting in

$$b = \frac{1}{y_i} - \boldsymbol{w}^T \boldsymbol{x}_i,$$

or if we write it out in terms of the support vectors only, with N_s being their number, we have

$$b = \frac{1}{N_s} \sum_{j \in N_s} \left(y_j - \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_j \right).$$

With our hyperplane coefficients we can use our classifier to assign any observation by simply using

$$y_i = \text{sign}(\boldsymbol{w}^T \boldsymbol{x}_i + b).$$

Below we discuss how to find the optimal values of λ_i. Before we proceed however, we discuss now the so-called soft classifier.

6.6.16 A soft classifier

Till now, the margin is strictly defined by the support vectors. This defines what is called a hard classifier, that is the margins are well defined.

Suppose now that classes overlap in feature space, as shown in the figure here. One way to deal with this problem before we define the so-called **kernel approach**, is to allow a kind of slack in the sense that we allow some points to be on the wrong side of the margin.

We introduce thus the so-called **slack** variables $\boldsymbol{\xi} = [\xi_1, x_2, \ldots, x_n]$ and modify our previous equation

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1,$$

to

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1 - \xi_i,$$

with the requirement $\xi_i \geq 0$. The total violation is now $\sum_i \xi$. The value ξ_i in the constraint the last constraint corresponds to the amount by which the prediction $y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1$ is on the wrong side of its margin. Hence by bounding the sum $\sum_i \xi_i$, we bound the total amount by which predictions fall on the wrong side of their margins.

Misclassifications occur when $\xi_i > 1$. Thus bounding the total sum by some value C bounds in turn the total number of misclassifications.

6.6.17 Soft optmization problem

This has in turn the consequences that we change our optmization problem to finding the minimum of

$$\mathcal{L} = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{n} \lambda_i \left[y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - (1 - \xi_) \right] + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \gamma_i \xi_i,$$

subject to

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1 - \xi_i \; \forall i,$$

with the requirement $\xi_i \geq 0$.

Taking the derivatives with respect to b and $\boldsymbol{w}$ we obtain

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \lambda_i y_i = 0,$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 = \boldsymbol{w} - \sum_i \lambda_i y_i \boldsymbol{x}_i,$$

and

$$\lambda_i = C - \gamma_i \; \forall i.$$

Inserting these constraints into the equation for $\mathcal{L}$ we obtain the same equation as before

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j,$$

but now subject to the constraints $\lambda_i \geq 0$, $\sum_i \lambda_i y_i = 0$ and $0 \leq \lambda_i \leq C$. We must in addition satisfy the Karush-Kuhn-Tucker condition which now reads

5 0

< < < ! ! M A T H _ B L O C K

$$\gamma_i \xi_i = 0,$$

and

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - (1 - \xi_) \geq 0 \; \forall i.$$

6.6.18 Kernels and non-linearity

The cases we have studied till now, were all characterized by two classes with a close to linear separability. The classifiers we have described so far find linear boundaries in our input feature space. It is possible to make our procedure more flexible by exploring the feature space using other basis expansions such as higher-order polynomials, wavelets, splines etc.

If our feature space is not easy to separate, as shown in the figure here, we can achieve a better separation by introducing more complex basis functions. The ideal would be, as shown in the next figure, to, via a specific transformation to obtain a separation between the classes which is almost linear.

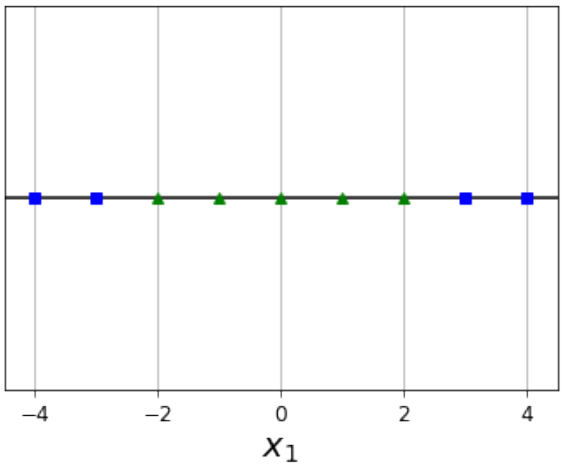
The change of basis, from $x \rightarrow z = \phi(x)$ leads to the same type of equations to be solved, except that we need to introduce for example a polynomial transformation to a two-dimensional training set.

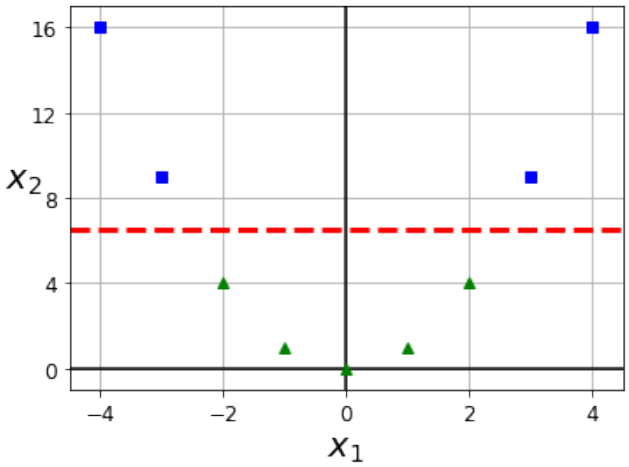
```
import numpy as np
import os

np.random.seed(42)

# To plot pretty figures
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12


from sklearn.svm import SVC
from sklearn import datasets
```

(continues on next page)

(continued from previous page)

```
X1D = np.linspace(-4, 4, 9).reshape(-1, 1)
X2D = np.c_[X1D, X1D**2]
y = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.plot(X1D[:, 0][y==0], np.zeros(4), "bs")
plt.plot(X1D[:, 0][y==1], np.zeros(5), "g^")
plt.gca().get_yaxis().set_ticks([])
plt.xlabel(r"$x_1$", fontsize=20)
plt.axis([-4.5, 4.5, -0.2, 0.2])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(X2D[:, 0][y==0], X2D[:, 1][y==0], "bs")
plt.plot(X2D[:, 0][y==1], X2D[:, 1][y==1], "g^")
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"$x_2$", fontsize=20, rotation=0)
plt.gca().get_yaxis().set_ticks([0, 4, 8, 12, 16])
plt.plot([-4.5, 4.5], [6.5, 6.5], "r--", linewidth=3)
plt.axis([-4.5, 4.5, -1, 17])
plt.subplots_adjust(right=1)
plt.show()
```





6.6.19 The equations

Suppose we define a polynomial transformation of degree two only (we continue to live in a plane with x_i and y_i as variables)

$$z = \phi(x_i) = \left(x_i^2, y_i^2, \sqrt{2}x_i y_i\right).$$

With our new basis, the equations we solved earlier are basically the same, that is we have now (without the slack option for simplicity)

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i\lambda_j y_i y_j \boldsymbol{z}_i^T\boldsymbol{z}_j,$$

subject to the constraints $\lambda_i \geq 0$, $\sum_i \lambda_i y_i = 0$, and for the support vectors

$$y_i(\boldsymbol{w}^T\boldsymbol{z}_i + b) = 1 \; \forall i,$$

from which we also find b. To compute $\boldsymbol{z}_i^T\boldsymbol{z}_j$ we define the kernel $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ as

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{z}_i^T\boldsymbol{z}_j = \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j).$$

For the above example, the kernel reads

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = [x_i^2, y_i^2, \sqrt{2}x_i y_i]^T \begin{bmatrix} x_j^2 \\ y_j^2 \\ \sqrt{2}x_j y_j \end{bmatrix} = x_i^2x_j^2 + 2x_i x_j y_i y_j + y_i^2 y_j^2.$$

We note that this is nothing but the dot product of the two original vectors $(\boldsymbol{x}_i^T\boldsymbol{x}_j)^2$. Instead of thus computing the product in the Lagrangian of $\boldsymbol{z}_i^T\boldsymbol{z}_j$ we simply compute the dot product $(\boldsymbol{x}_i^T\boldsymbol{x}_j)^2$.

This leads to the so-called kernel trick and the result leads to the same as if we went through the trouble of performing the transformation $\phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ during the SVM calculations.

6.6.20 The problem to solve

Using our definition of the kernel We can rewrite again the Lagrangian

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i\lambda_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{z}_j,$$

subject to the constraints $\lambda_i \geq 0$, $\sum_i \lambda_i y_i = 0$ in terms of a convex optimization problem

$$\frac{1}{2}\boldsymbol{\lambda}^T \begin{bmatrix} y_1y_1K(\boldsymbol{x}_1,\boldsymbol{x}_1) & y_1y_2K(\boldsymbol{x}_1,\boldsymbol{x}_2) & \dots & \dots & y_1y_nK(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ y_2y_1K(\boldsymbol{x}_2,\boldsymbol{x}_1) & y_2y_2(\boldsymbol{x}_2,\boldsymbol{x}_2) & \dots & \dots & y_1y_nK(\boldsymbol{x}_2,\boldsymbol{x}_n) \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ y_ny_1K(\boldsymbol{x}_n,\boldsymbol{x}_1) & y_ny_2K(\boldsymbol{x}_n\boldsymbol{x}_2) & \dots & \dots & y_ny_nK(\boldsymbol{x}_n,\boldsymbol{x}_n) \end{bmatrix} \boldsymbol{\lambda} - \mathbb{1}\boldsymbol{\lambda},$$

subject to $\boldsymbol{y}^T\boldsymbol{\lambda} = 0$. Here we defined the vectors $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_n]$ and $\boldsymbol{y} = [y_1, y_2, \dots, y_n]$. If we add the slack constants this leads to the additional constraint $0 \leq \lambda_i \leq C$.

We can rewrite this (see the solutions below) in terms of a convex optimization problem of the type

$$\begin{aligned} &\min_\lambda \; \frac{1}{2}\boldsymbol{\lambda}^T\boldsymbol{P}\boldsymbol{\lambda} + \boldsymbol{q}^T\boldsymbol{\lambda}, \\ &\text{subject to } \; \boldsymbol{G}\boldsymbol{\lambda} \preceq \boldsymbol{h} \;\; \wedge \boldsymbol{A}\boldsymbol{\lambda} = f. \end{aligned}$$

Below we discuss how to solve these equations. Here we note that the matrix $\boldsymbol{P}$ has matrix elements $p_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Given a kernel K and the targets y_i this matrix is easy to set up. The constraint $\boldsymbol{y}^T\boldsymbol{\lambda} = 0$ leads to $f = 0$ and $\boldsymbol{A} = \boldsymbol{y}$. How to set up the matrix $\boldsymbol{G}$ is discussed later. Here note that the inequalities $0 \leq \lambda_i \leq C$ can be split up into $0 \leq \lambda_i$ and $\lambda_i \leq C$. These two inequalities define then the matrix $\boldsymbol{G}$ and the vector $\boldsymbol{h}$.

6.6.21 Different kernels and Mercer's theorem

There are several popular kernels being used. These are

1. Linear: $K(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{x}^T\boldsymbol{y}$,
2. Polynomial: $K(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}^T\boldsymbol{y}+\gamma)^d$,
3. Gaussian Radial Basis Function: $K(\boldsymbol{x},\boldsymbol{y}) = \exp\left(-\gamma||\boldsymbol{x}-\boldsymbol{y}||^2\right)$,
4. Tanh: $K(\boldsymbol{x},\boldsymbol{y}) = \tanh\left(\boldsymbol{x}^T\boldsymbol{y}+\gamma\right)$,

and many other ones.

An important theorem for us is Mercer's theorem. The theorem states that if a kernel function K is symmetric, continuous and leads to a positive semi-definite matrix $\boldsymbol{P}$ then there exists a function ϕ that maps $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ into another space (possibly with much higher dimensions) such that

$$K(\boldsymbol{x}_i,\boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j).$$

So you can use K as a kernel since you know ϕ exists, even if you don't know what ϕ is.

Note that some frequently used kernels (such as the Sigmoid kernel) don't respect all of Mercer's conditions, yet they generally work well in practice.

6.6.22 The moons example

```
from __future__ import division, print_function, unicode_literals

import numpy as np
np.random.seed(42)

import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12


from sklearn.svm import SVC
from sklearn import datasets



from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC


from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
```

(continues on next page)

(continued from previous page)

```
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()

from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

plt.show()


from sklearn.svm import SVC

poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)

poly100_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))
    ])
poly100_kernel_svm_clf.fit(X, y)

plt.figure(figsize=(11, 4))

plt.subplot(121)
plot_predictions(poly_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.title(r"$d=3, r=1, C=5$", fontsize=18)

plt.subplot(122)
plot_predictions(poly100_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
```

(continues on next page)

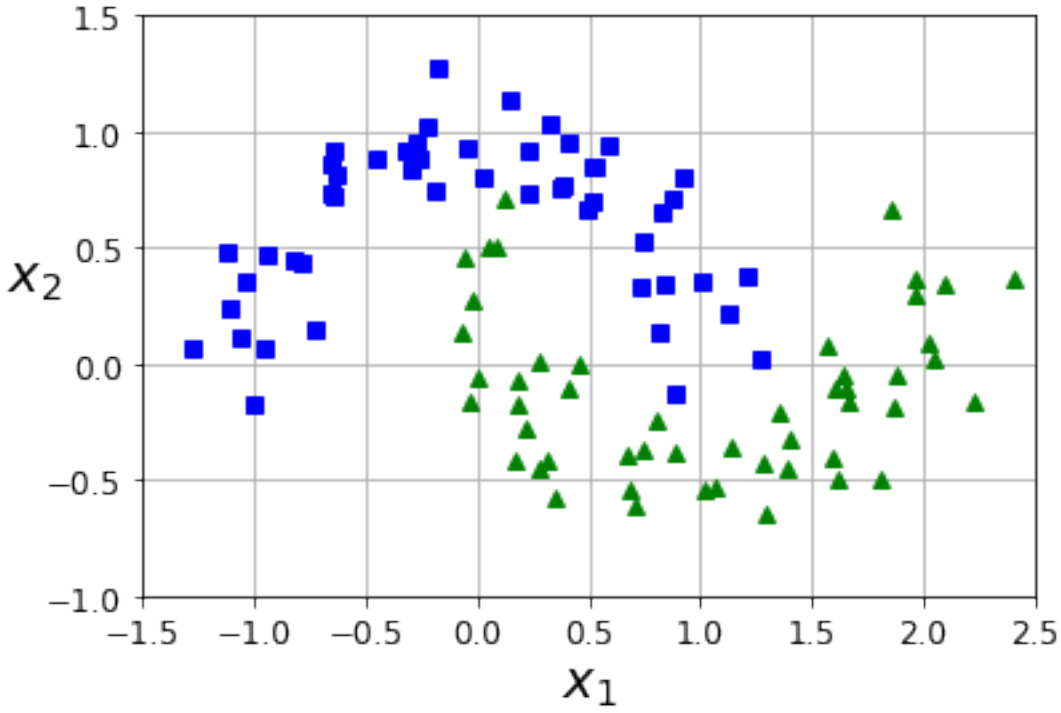
(continued from previous page)

```
plt.title(r"$d=10, r=100, C=5$", fontsize=18)

plt.show()

def gaussian_rbf(x, landmark, gamma):
    return np.exp(-gamma * np.linalg.norm(x - landmark, axis=1)**2)

gamma = 0.3

x1s = np.linspace(-4.5, 4.5, 200).reshape(-1, 1)
x2s = gaussian_rbf(x1s, -2, gamma)
x3s = gaussian_rbf(x1s, 1, gamma)

XK = np.c_[gaussian_rbf(X1D, -2, gamma), gaussian_rbf(X1D, 1, gamma)]
yk = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.scatter(x=[-2, 1], y=[0, 0], s=150, alpha=0.5, c="red")
plt.plot(X1D[:, 0][yk==0], np.zeros(4), "bs")
plt.plot(X1D[:, 0][yk==1], np.zeros(5), "g^")
plt.plot(x1s, x2s, "g--")
plt.plot(x1s, x3s, "b:")
plt.gca().get_yaxis().set_ticks([0, 0.25, 0.5, 0.75, 1])
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"Similarity", fontsize=14)
plt.annotate(r'$\mathbf{x}$',
             xy=(X1D[3, 0], 0),
             xytext=(-0.5, 0.20),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.text(-2, 0.9, "$x_2$", ha="center", fontsize=20)
plt.text(1, 0.9, "$x_3$", ha="center", fontsize=20)
plt.axis([-4.5, 4.5, -0.1, 1.1])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(XK[:, 0][yk==0], XK[:, 1][yk==0], "bs")
plt.plot(XK[:, 0][yk==1], XK[:, 1][yk==1], "g^")
plt.xlabel(r"$x_2$", fontsize=20)
plt.ylabel(r"$x_3$  ", fontsize=20, rotation=0)
plt.annotate(r'$\phi\left(\mathbf{x}\right)$',
             xy=(XK[3, 0], XK[3, 1]),
             xytext=(0.65, 0.50),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.plot([-0.1, 1.1], [0.57, -0.1], "r--", linewidth=3)
plt.axis([-0.1, 1.1, -0.1, 1.1])
```

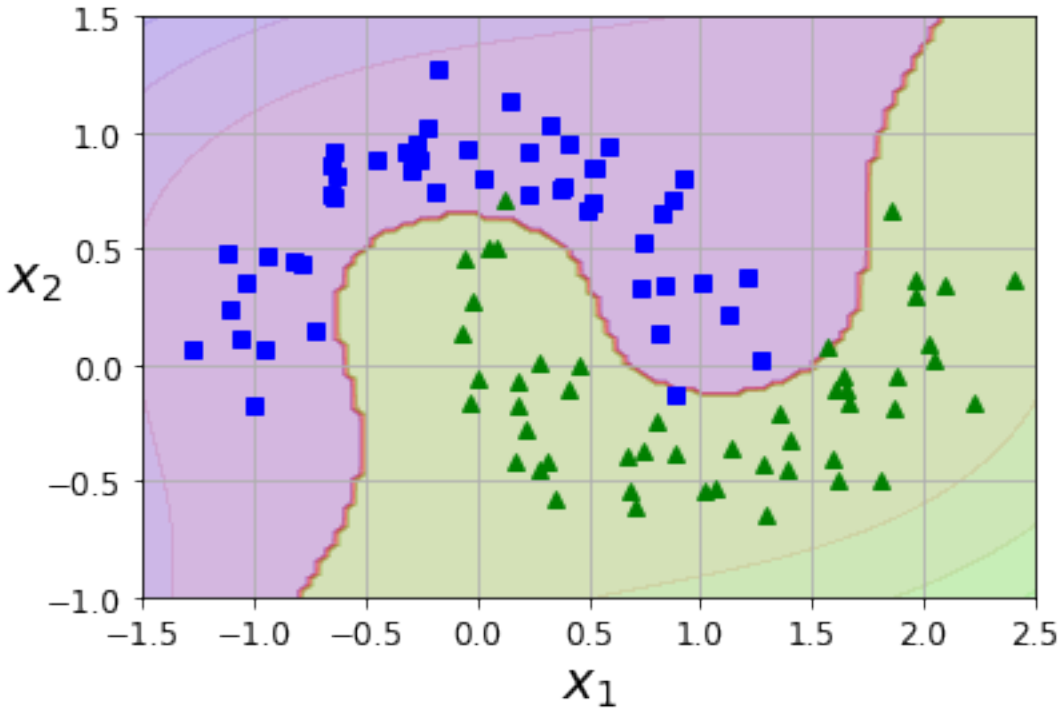
(continues on next page)

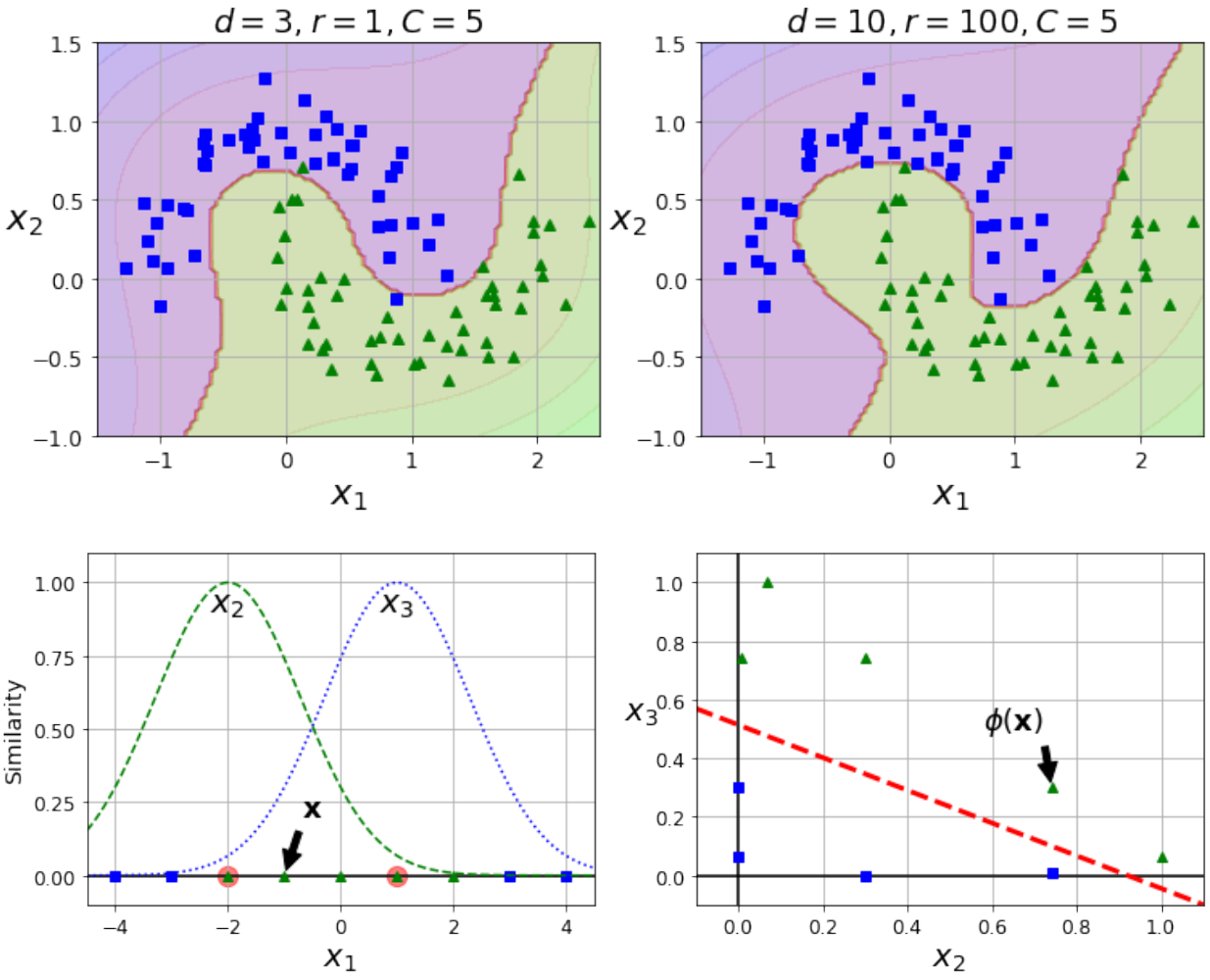
(continued from previous page)

```
plt.subplots_adjust(right=1)

plt.show()


x1_example = X1D[3, 0]
for landmark in (-2, 1):
    k = gaussian_rbf(np.array([[x1_example]]), np.array([[landmark]]), gamma)
    print("Phi({}, {}) = {}".format(x1_example, landmark, k))

rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)


from sklearn.svm import SVC

gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ])
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)

plt.figure(figsize=(11, 7))

for i, svm_clf in enumerate(svm_clfs):
    plt.subplot(221 + i)
    plot_predictions(svm_clf, [-1.5, 2.5, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsize=16)

plt.show()
```

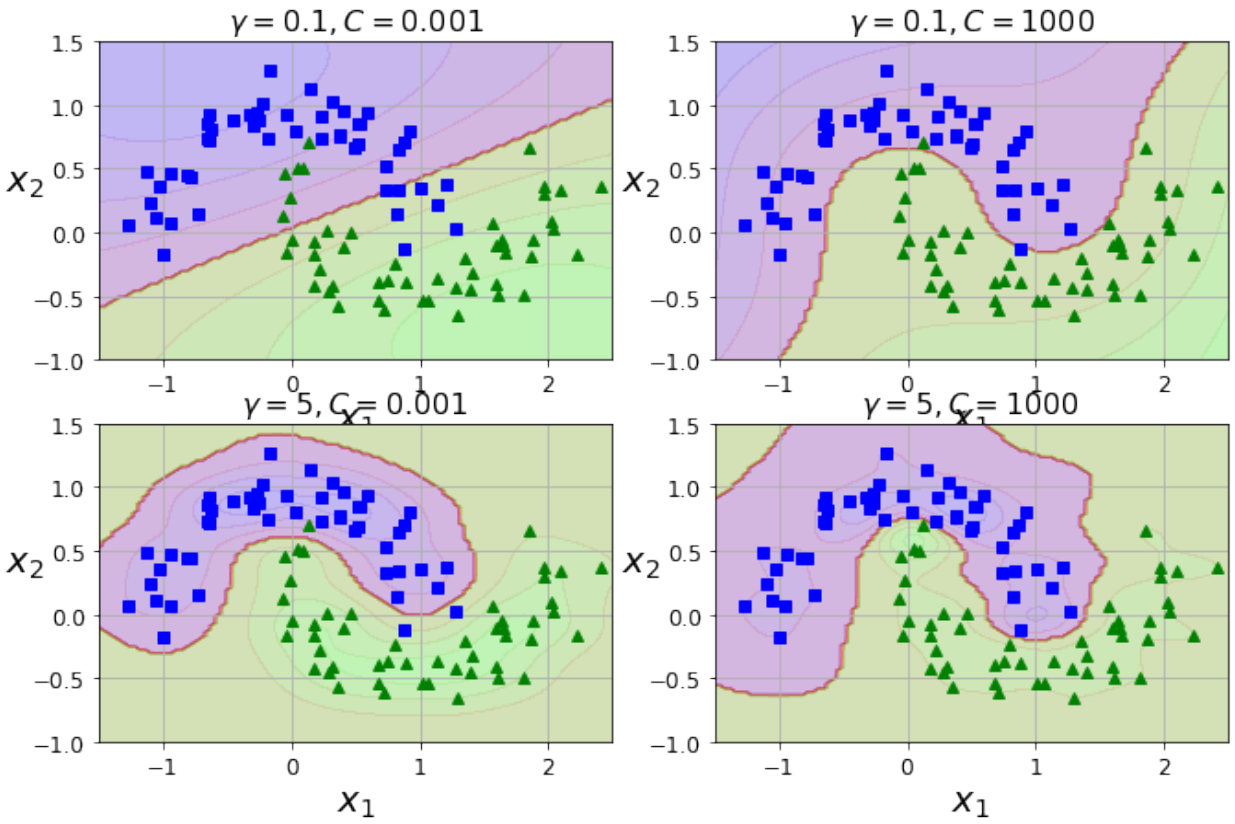



```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/svm/_base.py:976:␣
↪ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```






```
Phi(-1.0, -2) = [0.74081822]
Phi(-1.0, 1) = [0.30119421]
```



6.6.23 Mathematical optimization of convex functions

A mathematical (quadratic) optimization problem, or just optimization problem, has the form

$$\min_\lambda \ \frac{1}{2}\boldsymbol{\lambda}^T\boldsymbol{P}\boldsymbol{\lambda} + \boldsymbol{q}^T\boldsymbol{\lambda},$$
$$\text{subject to } \ \boldsymbol{G}\boldsymbol{\lambda} \preceq \boldsymbol{h} \wedge \boldsymbol{A}\boldsymbol{\lambda} = f.$$

subject to some constraints for say a selected set $i = 1, 2, \ldots, n$. In our case we are optimizing with respect to the Lagrangian multipliers λ_i, and the vector $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_n]$ is the optimization variable we are dealing with.

In our case we are particularly interested in a class of optimization problems called convex optmization problems. In our discussion on gradient descent methods we discussed at length the definition of a convex function.

Convex optimization problems play a central role in applied mathematics and we recommend strongly Boyd and Vandenberghe's text on the topics.

6.6.24 How do we solve these problems?

If we use Python as programming language and wish to venture beyond **scikit-learn**, **tensorflow** and similar software which makes our lives so much easier, we need to dive into the wonderful world of quadratic programming. We can, if we wish, solve the minimization problem using say standard gradient methods or conjugate gradient methods. However, these methods tend to exhibit a rather slow converge. So, welcome to the promised land of quadratic programming.

The functions we need are contained in the quadratic programming package **CVXOPT** and we need to import it together with **numpy** as

```
import numpy
import cvxopt
```

This will make our life much easier. You don't need t write your own optimizer.

6.6.25 A simple example

We remind ourselves about the general problem we want to solve

$$\begin{aligned}&\min_x \ \frac{1}{2}\boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x}+\boldsymbol{q}^T\boldsymbol{x},\\ &\text{subject to } \ \boldsymbol{G}\boldsymbol{x}\preceq\boldsymbol{h}\wedge\boldsymbol{A}\boldsymbol{x}=f.\end{aligned}$$

Let us show how to perform the optmization using a simple case. Assume we want to optimize the following problem

$$\begin{aligned}&\min_x \ \frac{1}{2}x^2+5x+3y\\ &\text{subjectto}\\ &x,y\geq 0\\ &x+3y\geq 15\\ &2x+5y\leq 100\\ &3x+4y\leq 80.\end{aligned}$$

The minimization problem can be rewritten in terms of vectors and matrices as (with x and y being the unknowns)

$$\frac{1}{2}\begin{bmatrix}x\\y\end{bmatrix}^T\begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}3\\4\end{bmatrix}^T\begin{bmatrix}x\\y\end{bmatrix}.$$

Similarly, we can now set up the inequalities (we need to change $\geq$ to $\leq$ by multiplying with -1 on bot sides) as the following matrix-vector equation

$$\begin{bmatrix}-1&0\\0&-1\\-1&-3\\2&5\\3&4\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}\preceq\begin{bmatrix}0\\0\\-15\\100\\80\end{bmatrix}.$$

We have collapsed all the inequalities into a single matrix $\boldsymbol{G}$. We see also that our matrix

$$\boldsymbol{P}=\begin{bmatrix}1&0\\0&0\end{bmatrix}$$

is clearly positive semi-definite (all eigenvalues larger or equal zero). Finally, the vector $\boldsymbol{h}$ is defined as

$$\boldsymbol{h}=\begin{bmatrix}0\\0\\-15\\100\\80\end{bmatrix}.$$

Since we don't have any equalities the matrix $\boldsymbol{A}$ is set to zero The following code solves the equations for us

```
# Import the necessary packages
import numpy
from cvxopt import matrix
from cvxopt import solvers
P = matrix(numpy.diag([1,0]), tc='d')
q = matrix(numpy.array([3,4]), tc='d')
G = matrix(numpy.array([[-1,0],[0,-1],[-1,-3],[2,5],[3,4]]), tc='d')
h = matrix(numpy.array([0,0,-15,100,80]), tc='d')
# Construct the QP, invoke solver
sol = solvers.qp(P,q,G,h)
# Extract optimal value and solution
sol['x']
sol['primal objective']
```

```
  File "<ipython-input-5-c46dd114b2af>", line 5
    P = matrix(numpy.diag([1,0]), tc='d')
                                      ^
SyntaxError: invalid character in identifier
```

6.6.26 Back to the more realistic cases

We are now ready to return to our setup of the optmization problem for a more realistic case. Introducing the **slack** parameter C we have

$$\frac{1}{2}\boldsymbol{\lambda}^T \begin{bmatrix} y_1y_1K(\boldsymbol{x}_1,\boldsymbol{x}_1) & y_1y_2K(\boldsymbol{x}_1,\boldsymbol{x}_2) & \dots & \dots & y_1y_nK(\boldsymbol{x}_1,\boldsymbol{x}_n) \\ y_2y_1K(\boldsymbol{x}_2,\boldsymbol{x}_1) & y_2y_2K(\boldsymbol{x}_2,\boldsymbol{x}_2) & \dots & \dots & y_1y_nK(\boldsymbol{x}_2,\boldsymbol{x}_n) \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ y_ny_1K(\boldsymbol{x}_n,\boldsymbol{x}_1) & y_ny_2K(\boldsymbol{x}_n\boldsymbol{x}_2) & \dots & \dots & y_ny_nK(\boldsymbol{x}_n,\boldsymbol{x}_n) \end{bmatrix} \boldsymbol{\lambda} - \mathbb{I}\boldsymbol{\lambda},$$

subject to $\boldsymbol{y}^T\boldsymbol{\lambda} = 0$. Here we defined the vectors $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_n]$ and $\boldsymbol{y} = [y_1, y_2, \dots, y_n]$. With the slack constants this leads to the additional constraint $0 \leq \lambda_i \leq C$.

code will be added

6.7 Dimensionality Reduction

6.7.1 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.

Here we will discuss some of the most popular dimensionality reduction techniques: the principal component analysis (PCA), Kernel PCA, and Locally Linear Embedding (LLE). Furthermore, we will start by looking at some simple preprocessing of the data which allow us to rescale the data.

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.

6.7.2 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

6.7.3 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.

6.7.4 Simple preprocessing examples, Franke function and regression

```
%matplotlib inline

# 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
from sklearn.svm import SVR

# 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)
```

(continues on next page)

(continued from previous page)

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

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

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


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


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

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

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

        return X


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


svm = SVR(gamma='auto',C=10.0)
svm.fit(X_train, y_train)

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

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
```

(continues on next page)

(continued from previous page)

```
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)))

svm = SVR(gamma='auto',C=10.0)
svm.fit(X_train_scaled, y_train)

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

```
MSE before scaling: 0.01
R2 score before scaling 0.93
Feature min values before scaling:
 [1.00000000e+00 7.43297505e-04 1.67887686e-04 5.52491181e-07
 1.24790498e-07 2.81862750e-08 4.10665316e-10 9.27564656e-11
 2.09507879e-11 4.73212847e-12 3.05246505e-13 6.89456494e-14
 1.55726683e-14 3.51737928e-15 7.94466097e-16 2.26888965e-16
 5.12471292e-17 1.15751255e-17 2.61445925e-18 5.90524667e-19
 1.33381074e-19]
Feature max values before scaling:
 [1.         0.99422559 0.99481826 0.98848453 0.98907377 0.98966337
 0.98277662 0.98336246 0.98394865 0.98453519 0.97710167 0.97768412
 0.97826693 0.97885008 0.97943358 0.97145948 0.97203858 0.97261802
 0.9731978  0.97377793 0.97435841]
Feature min values after scaling:
 [ 0.         -1.7697784  -1.72415276 -1.14649434 -1.12953807 -1.11320874
 -0.90535485 -0.8968819  -0.88864254 -0.88062733 -0.77069333 -0.76574003
 -0.76089806 -0.7561637  -0.75153337 -0.68210821 -0.6789547  -0.67586334
 -0.67283231 -0.66985986 -0.66694428]
Feature max values after scaling:
 [0.         1.68453745 1.71039356 2.1618639  2.17707405 2.19168422
 2.54536814 2.55730765 2.56887875 2.58009023 2.87398064 2.88442648
 2.89461285 2.90454384 2.91422324 3.16472629 3.17449812 3.18407693
 3.19346461 3.20266288 3.21167335]
MSE after  scaling: 0.00
R2 score for  scaled data: 0.97
```

6.7.5 Simple preprocessing examples, breast cancer data and classification, Support Vector Machines

We show here how we can use a simple regression case on the breast cancer data using support vector machines (SVM) as algorithm for classification.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
cancer = load_breast_cancer()
```

(continues on next page)

(continued from previous page)

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_
↪state=0)
print(X_train.shape)
print(X_test.shape)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test,y_test)))

from sklearn.preprocessing import MinMaxScaler, StandardScaler
scaler = MinMaxScaler()
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 before scaling:\n {}".format(X_train_scaled.min(axis=0)))
print("Feature max values before scaling:\n {}".format(X_train_scaled.max(axis=0)))


svm.fit(X_train_scaled, y_train)
print("Test set accuracy scaled data with Min-Max scaling: {:.2f}".format(svm.score(X_
↪test_scaled,y_test)))

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

svm.fit(X_train_scaled, y_train)
print("Test set accuracy scaled data with Standar Scaler: {:.2f}".format(svm.score(X_
↪test_scaled,y_test)))
```

```
(426, 30)
(143, 30)
Test set accuracy: 0.94
Feature min values before scaling:
 [6.981e+00 9.710e+00 4.379e+01 1.435e+02 5.263e-02 1.938e-02 0.000e+00
 0.000e+00 1.060e-01 4.996e-02 1.115e-01 3.628e-01 7.570e-01 7.228e+00
 1.713e-03 2.252e-03 0.000e+00 0.000e+00 7.882e-03 8.948e-04 7.930e+00
 1.202e+01 5.041e+01 1.852e+02 7.117e-02 2.729e-02 0.000e+00 0.000e+00
 1.565e-01 5.504e-02]
Feature max values before scaling:
 [2.811e+01 3.381e+01 1.885e+02 2.501e+03 1.447e-01 3.114e-01 4.268e-01
 2.012e-01 3.040e-01 9.744e-02 2.873e+00 4.885e+00 2.198e+01 5.422e+02
 2.333e-02 1.064e-01 3.960e-01 5.279e-02 6.146e-02 2.984e-02 3.604e+01
 4.954e+01 2.512e+02 4.254e+03 2.226e-01 1.058e+00 1.252e+00 2.903e-01
 6.638e-01 2.075e-01]
Feature min values before scaling:
 [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.]
```

```
Feature max values before scaling:
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1.]
Test set accuracy scaled data with Min-Max scaling: 0.97
Test set accuracy scaled data with Standar Scaler: 0.96
```

6.7.6 More on Cancer Data, now with Logistic Regression

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()

# Set up training data
X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_
↪state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(logreg.score(X_test,y_test)))

# Scale data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
logreg.fit(X_train_scaled, y_train)
print("Test set accuracy scaled data: {:.2f}".format(logreg.score(X_test_scaled,y_
↪test)))
```

```
Test set accuracy: 0.95
Test set accuracy scaled data: 0.96
```

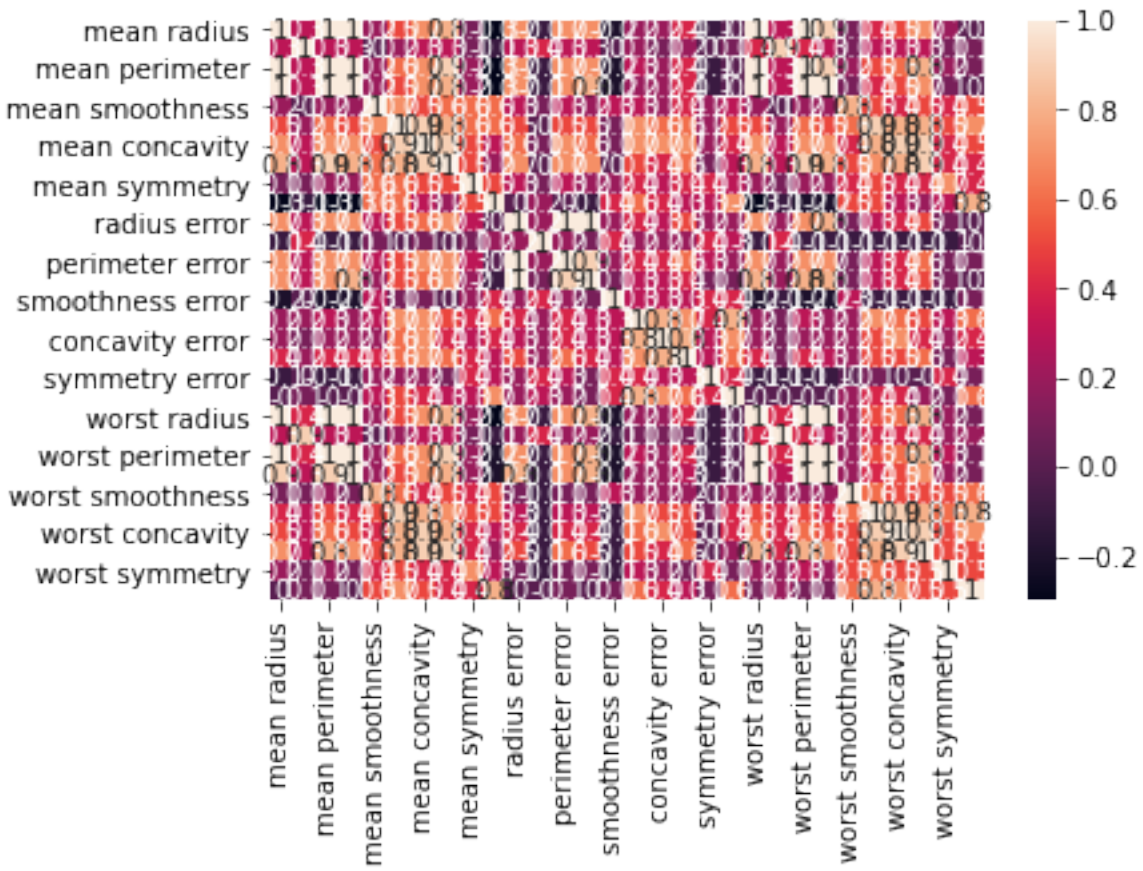
```
/Users/hjensen/opt/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_
↪logistic.py:762: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

6.7.7 Why should we think of reducing the dimensionality

In addition to the plot of the features, we study now also the covariance (and the correlation matrix). We use also **Pandas** to compute the correlation matrix.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()
import pandas as pd
# Making a data frame
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)

fig, axes = plt.subplots(15,2,figsize=(10,20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]
ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:,i], bins =50)
    ax[i].hist(malignant[:,i], bins = bins, alpha = 0.5)
    ax[i].hist(benign[:,i], bins = bins, alpha = 0.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["Malignant", "Benign"], loc ="best")
fig.tight_layout()
plt.show()

import seaborn as sns
correlation_matrix = cancerpd.corr().round(1)
# 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)
plt.show()

#print eigvalues of correlation matrix
EigValues, EigVectors = np.linalg.eig(correlation_matrix)
print(EigValues)
```


mean radius
Malignant
Benign
Frequency
Feature magnitude
mean texture
mean perimeter
mean area
mean smoothness
mean compactness
mean concavity
mean concave points
mean symmetry
mean fractal dimension
radius error
texture error
perimeter error
area error
smoothness error
compactness error
concavity error
concave points error
symmetry error
fractal dimension error
worst radius
worst texture
worst perimeter
worst area
worst smoothness
worst compactness
worst concavity
worst concave points
worst symmetry
worst fractal dimension




```
[ 1.33026664e+01  5.69238112e+00  2.83259341e+00  1.99126621e+00
  1.69095941e+00  1.19614372e+00  7.13037918e-01  5.77804918e-01
  5.14549644e-01  4.30579019e-01  3.58977964e-01 -1.91269450e-01
  2.86195387e-01 -1.59310961e-01 -1.38483045e-01  2.44786221e-01
 -8.79318674e-02  1.91811329e-01  1.70896266e-01 -7.17683028e-02
  1.39860289e-01  1.18721828e-01 -4.83194721e-02  8.92983892e-02
  6.81402316e-02  6.01968016e-02  3.31819744e-02  1.56432255e-02
 -1.61028757e-02 -6.50565973e-03]
```

In the above example we note two things. In the first plot we display the overlap of benign and malignant tumors as functions of the various features in the Wisconsing breast cancer data set. We see that for some of the features we can distinguish clearly the benign and malignant cases while for other features we cannot. This can point to us which features may be of greater interest when we wish to classify a benign or not benign tumour.

In the second figure we have computed the so-called correlation matrix, which in our case with thirty features becomes a 30×30 matrix.

We constructed this matrix using **pandas** via the statements

```
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)
```

and then

```
correlation_matrix = cancerpd.corr().round(1)
```

Diagonalizing this matrix we can in turn say something about which features are of relevance and which are not. But before we proceed we need to define covariance and correlation matrices. This leads us to the classical Principal

Component Analysis (PCA) theorem with applications.

6.7.8 Basic ideas of the Principal Component Analysis (PCA)

The principal component analysis deals with the problem of fitting a low-dimensional affine subspace S of dimension d much smaller than the totaldimension D of the problem at hand (our data set). Mathematically it can be formulated as a statistical problem or a geometric problem. In our discussion of the theorem for the classical PCA, we will stay with a statistical approach. This is also what set the scene historically which for the PCA.

We have a data set defined by a design/feature matrix $\boldsymbol{X}$ (see below for its definition)

- Each data point is determined by p extrinsic (measurement) variables
- We may want to ask the following question: Are there fewer intrinsic variables (say $d << p$) that still approximately describe the data?
- If so, these intrinsic variables may tell us something important and finding these intrinsic variables is what dimension reduction methods do.

6.7.9 Introducing the Covariance and Correlation functions

Before we discuss the PCA theorem, 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**.

6.7.10 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

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

6.7.11 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)
```

```
0.039184456674535545
4.128680426693387
[[ 1.12297057  3.2186233 ]
 [ 3.2186233  10.11517976]]
```

6.7.12 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)
```

```
0.08474873038505544
1.7239002399681738
[[1.         0.62968416]
 [0.62968416 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**.

6.7.13 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)
```

```
[[ 0.29439863  1.90169666]
 [ 1.67736015  3.10465335]
 [-0.55604865 -0.71001501]
 [-0.86385893 -3.79843757]
 [-0.32231733  0.0602184 ]
 [-1.30134141 -3.01442901]
 [-0.46095356  0.49571687]
 [ 0.83524925  1.94721304]
 [-0.41069583 -1.37424606]
 [ 1.10820768  1.38762934]]
          0         1
0  0.294399  1.901697
1  1.677360  3.104653
2 -0.556049 -0.710015
3 -0.863859 -3.798438
4 -0.322317  0.060218
5 -1.301341 -3.014429
6 -0.460954  0.495717
7  0.835249  1.947213
8 -0.410696 -1.374246
9  1.108208  1.387629
          0         1
0  1.000000  0.883341
1  0.883341  1.000000
```

We expand this model to the Franke function discussed above.

6.7.14 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  0.088096  0.086427  0.083845  0.082640  0.081425  0.073689  0.072606
2   0.0  0.086427  0.085149  0.082400  0.081356  0.080291  0.072444  0.071452
3   0.0  0.083845  0.082400  0.085937  0.084688  0.083422  0.079032  0.077855
4   0.0  0.082640  0.081356  0.084688  0.083531  0.082352  0.077865  0.076753
5   0.0  0.081425  0.080291  0.083422  0.082352  0.081256  0.076682  0.075632
6   0.0  0.073689  0.072444  0.079032  0.077865  0.076682  0.074903  0.073780
7   0.0  0.072606  0.071452  0.077855  0.076753  0.075632  0.073780  0.072708
8   0.0  0.071547  0.070480  0.076701  0.075660  0.074600  0.072677  0.071655
9   0.0  0.070513  0.069529  0.075570  0.074589  0.073587  0.071595  0.070622
```

(continues on next page)

(continued from previous page)

```
10  0.0  0.064543  0.063450  0.071358  0.070294  0.069218  0.069100  0.068064
11  0.0  0.063581  0.062551  0.070288  0.069275  0.068248  0.068065  0.067072
12  0.0  0.062646  0.061677  0.069247  0.068283  0.067304  0.067057  0.066105
13  0.0  0.061738  0.060828  0.068235  0.067318  0.066385  0.066075  0.065164
14  0.0  0.060857  0.060003  0.067250  0.066378  0.065490  0.065119  0.064247

           8         9         10        11        12        13        14
0   0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
1   0.071547  0.070513  0.064543  0.063581  0.062646  0.061738  0.060857
2   0.070480  0.069529  0.063450  0.062551  0.061677  0.060828  0.060003
3   0.076701  0.075570  0.071358  0.070288  0.069247  0.068235  0.067250
4   0.075660  0.074589  0.070294  0.069275  0.068283  0.067318  0.066378
5   0.074600  0.073587  0.069218  0.068248  0.067304  0.066385  0.065490
6   0.072677  0.071595  0.069100  0.068065  0.067057  0.066075  0.065119
7   0.071655  0.070622  0.068064  0.067072  0.066105  0.065164  0.064247
8   0.070650  0.069664  0.067046  0.066096  0.065170  0.064268  0.063389
9   0.069664  0.068722  0.066047  0.065138  0.064251  0.063387  0.062545
10  0.067046  0.066047  0.064786  0.063821  0.062880  0.061964  0.061071
11  0.066096  0.065138  0.063821  0.062893  0.061989  0.061108  0.060249
12  0.065170  0.064251  0.062880  0.061989  0.061120  0.060273  0.059447
13  0.064268  0.063387  0.061964  0.061108  0.060273  0.059458  0.058665
14  0.063389  0.062545  0.061071  0.060249  0.059447  0.058665  0.057901
```

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.

6.7.15 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

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

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

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

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 $\$\boldsymbol{C}[\boldsymbol{x}_0, \boldsymbol{x}_1] = \boldsymbol{C}[\boldsymbol{x}] to indicate that this is the covariance of the vectors \boldsymbol{x} of the design/feature matrix \b$

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

6.7.16 Towards the PCA theorem

We have that the covariance matrix (the correlation matrix involves a simple rescaling) is given as

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

Let us now assume that we can perform a series of orthogonal transformations where we employ some orthogonal matrices $\boldsymbol{S}$. These matrices are defined as $\boldsymbol{S} \in \mathbb{R}^{p\times p}$ and obey the orthogonality requirements $\boldsymbol{S}\boldsymbol{S}^T = \boldsymbol{S}^T\boldsymbol{S} = \boldsymbol{I}$. The matrix can be written out in terms of the column vectors $\boldsymbol{s}_i$ as $\boldsymbol{S} = [\boldsymbol{s}_0, \boldsymbol{s}_1, \dots, \boldsymbol{s}_{p-1}]$ and $\boldsymbol{s}_i \in \mathbb{R}^p$.

Assume also that there is a transformation $\boldsymbol{S}\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T = \boldsymbol{C}[\boldsymbol{y}]$ such that the new matrix $\boldsymbol{C}[\boldsymbol{y}]$ is diagonal with elements $[\lambda_0, \lambda_1, \lambda_2, \dots, \lambda_{p-1}]$.

That is we have

$$\boldsymbol{C}[\boldsymbol{y}] = \mathbb{E}[\boldsymbol{S}\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{S}^T] = \boldsymbol{S}\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T,$$

since the matrix $\boldsymbol{S}$ is not a data dependent matrix. Multiplying with $\boldsymbol{S}^T$ from the left we have

$$\boldsymbol{S}^T\boldsymbol{C}[\boldsymbol{y}] = \boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}^T,$$

and since $\boldsymbol{C}[\boldsymbol{y}]$ is diagonal we have for a given eigenvalue i of the covariance matrix that

$$\boldsymbol{S}_i^T\lambda_i = \boldsymbol{C}[\boldsymbol{x}]\boldsymbol{S}_i^T.$$

In the derivation of the PCA theorem we will assume that the eigenvalues are ordered in descending order, that is $\lambda_0 > \lambda_1 > \cdots > \lambda_{p-1}$.

The eigenvalues tell us then how much we need to stretch the corresponding eigenvectors. Dimensions with large eigenvalues have thus large variations (large variance) and define therefore useful dimensions. The data points are more spread out in the direction of these eigenvectors. Smaller eigenvalues mean on the other hand that the corresponding eigenvectors are shrunk accordingly and the data points are tightly bunched together and there is not much variation in these specific directions. Hopefully then we could leave it out dimensions where the eigenvalues are very small. If p is very large, we could then aim at reducing p to $l << p$ and handle only l features/predictors.

6.7.17 The Algorithm before theorem

Here's how we would proceed in setting up the algorithm for the PCA, see also discussion below here.

- Set up the datapoints for the design/feature matrix $\boldsymbol{X}$ with $\boldsymbol{X} \in \mathbb{R}^{n\times p}$, with the predictors/features p referring to the column numbers and the entries n being the row elements.

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

- Center the data by subtracting the mean value for each column. This leads to a new matrix $\boldsymbol{X} \rightarrow \overline{\boldsymbol{X}}$.
- Compute then the covariance/correlation matrix $\mathbb{E}[\overline{\boldsymbol{X}\boldsymbol{X}}^T]$.
- Find the eigenpairs of $\boldsymbol{C}$ with eigenvalues $[\lambda_0, \lambda_1, \dots, \lambda_{p-1}]$ and eigenvectors $[\boldsymbol{s}_0, \boldsymbol{s}_1, \dots, \boldsymbol{s}_{p-1}]$.
- Order the eigenvalue (and the eigenvectors accordingly) in order of decreasing eigenvalues.
- Keep only those l eigenvalues larger than a selected threshold value, discarding thus $p - l$ features since we expect small variations in the data here.

6.7.18 Writing our own PCA code

We will use a simple example first with two-dimensional data drawn from a multivariate normal distribution with the following mean and covariance matrix:

$$\mu = (-1, 2) \qquad \Sigma = \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix}$$

Note that the mean refers to each column of data. We will generate $n = 1000$ points $X = \{x_1, \ldots, x_N\}$ from this distribution, and store them in the 1000×2 matrix $\boldsymbol{X}$.

The following Python code aids in setting up the data and writing out the design matrix. Note that the function **multivariate** returns also the covariance discussed above and that it is defined by dividing by $n - 1$ instead of n.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
n = 10000
mean = (-1, 2)
cov = [[4, 2], [2, 2]]
X = np.random.multivariate_normal(mean, cov, n)
```

Now we are going to implement the PCA algorithm. We will break it down into various substeps.

Compute the sample mean and center the data

The first step of PCA is to compute the sample mean of the data and use it to center the data. Recall that the sample mean is

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and the mean-centered data $\bar{X} = \{\bar{x}_1, \ldots, \bar{x}_n\}$ takes the form

$$\bar{x}_i = x_i - \mu_n.$$

When you are done with these steps, print out μ_n to verify it is close to μ and plot your mean centered data to verify it is centered at the origin! Compare your code with the functionality from **Scikit-Learn** discussed above. The following code elements perform these operations using **pandas** or using our own functionality for doing so. The latter, using **numpy** is rather simple through the **mean()** function.

```
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df -df.mean()
# we center it ourselves
X_centered = X - X.mean(axis=0)
```

Alternatively, we could use the functions we discussed earlier for scaling the data set. That is, we could have used the **StandardScaler** function in **Scikit-Learn**, a function which 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). You would then not get the same results, since we divide by the variance. The diagonal covariance matrix elements will then be one, while the non-diagonal ones need to be divided by $2\sqrt{2}$ for our specific case.

Compute the sample covariance

Now we are going to use the mean centered data to compute the sample covariance of the data by using the following equation

$$\Sigma_n = \frac{1}{n-1}\sum_{i=1}^{n} \bar{x}_i^T \bar{x}_i = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \mu_n)^T (x_i - \mu_n)$$

where the data points $x_i \in \mathbb{R}^p$ (here in this example $p = 2$) are column vectors and x^T is the transpose of x. We can write our own code or simply use either the functionaly of **numpy** or that of **pandas**, as follows

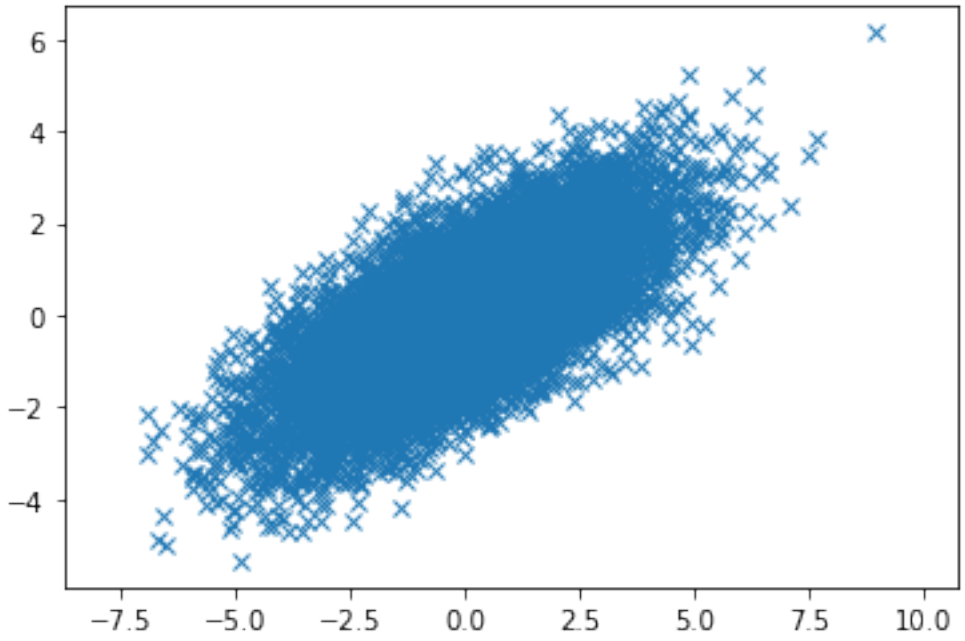
```
print(df.cov())
print(np.cov(X_centered.T))
```

```
          0         1
0  4.050693  2.010827
1  2.010827  1.974163
[[4.050693   2.01082738]
 [2.01082738 1.97416255]]
```

Note that the way we define the covariance matrix here has a factor $n - 1$ instead of n. This is included in the **cov()** function by **numpy** and **pandas**. Our own code here is not very elegant and asks for obvious improvements. It is tailored to this specific 2×2 covariance matrix.

```
# extract the relevant columns from the centered design matrix of dim n x 2
x = X_centered[:,0]
y = X_centered[:,1]
Cov = np.zeros((2,2))
Cov[0,1] = np.sum(x.T@y)/(n-1.0)
Cov[0,0] = np.sum(x.T@x)/(n-1.0)
Cov[1,1] = np.sum(y.T@y)/(n-1.0)
Cov[1,0]= Cov[0,1]
print("Centered covariance using own code")
print(Cov)
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```

```
Centered covariance using own code
[[4.050693   2.01082738]
 [2.01082738 1.97416255]]
```



Depending on the number of points n, we will get results that are close to the covariance values defined above. The plot shows how the data are clustered around a line with slope close to one. Is this expected?

Diagonalize the sample covariance matrix to obtain the principal components

Now we are ready to solve for the principal components! To do so we diagonalize the sample covariance matrix Σ. We can use the function **np.linalg.eig** to do so. It will return the eigenvalues and eigenvectors of Σ. Once we have these we can perform the following tasks:

- We compute the percentage of the total variance captured by the first principal component
- We plot the mean centered data and lines along the first and second principal components
- Then we project the mean centered data onto the first and second principal components, and plot the projected data.
- Finally, we approximate the data as

$$x_i \approx \tilde{x}_i = \mu_n + \langle x_i, v_0 \rangle v_0$$

where v_0 is the first principal component.

Collecting all these steps we can write our own PCA function and compare this with the functionality included in **Scikit-Learn**.

The code here outlines some of the elements we could include in the analysis. Feel free to extend upon this in order to address the above questions.

```
# diagonalize and obtain eigenvalues, not necessarily sorted
EigValues, EigVectors = np.linalg.eig(Cov)
# sort eigenvectors and eigenvalues
#permute = EigValues.argsort()
#EigValues = EigValues[permute]
#EigVectors = EigVectors[:,permute]
print("Eigenvalues of Covariance matrix")
for i in range(2):
```

(continues on next page)

(continued from previous page)

```
    print(EigValues[i])
FirstEigvector = EigVectors[:,0]
SecondEigvector = EigVectors[:,1]
print("First eigenvector")
print(FirstEigvector)
print("Second eigenvector")
print(SecondEigvector)
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2Dsl = pca.fit_transform(X)
print("Eigenvector of largest eigenvalue")
print(pca.components_.T[:, 0])
```

```
Eigenvalues of Covariance matrix
5.275483542917728
0.7493720026008108
First eigenvector
[0.85404598 0.52019753]
Second eigenvector
[-0.52019753  0.85404598]
Eigenvector of largest eigenvalue
[0.85404598 0.52019753]
```

This code does not contain all the above elements, but it shows how we can use **Scikit-Learn** to extract the eigenvector which corresponds to the largest eigenvalue. Try to address the questions we pose before the above code. Try also to change the values of the covariance matrix by making one of the diagonal elements much larger than the other. What do you observe then?

6.7.19 Classical PCA Theorem

We assume now that we have a design matrix $\boldsymbol{X}$ which has been centered as discussed above. For the sake of simplicity we skip the overline symbol. The matrix is defined in terms of the various column vectors $[\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_{p-1}]$ each with dimension $\boldsymbol{x} \in \mathbb{R}^n$.

We assume also that we have an orthogonal transformation $\boldsymbol{W} \in \mathbb{R}^{p \times p}$. We define the reconstruction error (which is similar to the mean squared error we have seen before) as

$$J(\boldsymbol{W}, \boldsymbol{Z}) = \frac{1}{n} \sum_i (\boldsymbol{x}_i - \overline{\boldsymbol{x}}_i)^2,$$

with $\overline{\boldsymbol{x}}_i = \boldsymbol{W}\boldsymbol{z}_i$, where $\boldsymbol{z}_i$ is a row vector with dimension $\mathbb{R}^n$ of the matrix $\boldsymbol{Z} \in \mathbb{R}^{p \times n}$. When doing PCA we want to reduce this dimensionality.

The PCA theorem states that minimizing the above reconstruction error corresponds to setting $\boldsymbol{W} = \boldsymbol{S}$, the orthogonal matrix which diagonalizes the empirical covariance(correlation) matrix. The optimal low-dimensional encoding of the data is then given by a set of vectors $\boldsymbol{z}_i$ with at most l vectors, with $l << p$, defined by the orthogonal projection of the data onto the columns spanned by the eigenvectors of the covariance(correlations matrix).

The proof which follows will be updated by mid January 2020.

6.7.20 Proof of the PCA Theorem

To show the PCA theorem let us start with the assumption that there is one vector $\boldsymbol{w}_0$ which corresponds to a solution which minimized the reconstruction error J. This is an orthogonal vector. It means that we now approximate the reconstruction error in terms of $\boldsymbol{w}_0$ and $\boldsymbol{z}_0$ as

$$J(\boldsymbol{w}_0,\boldsymbol{z}_0) = \frac{1}{n}\sum_i (\boldsymbol{x}_i - z_{i0}\boldsymbol{w}_0)^2 = \frac{1}{n}\sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - 2z_{i0}\boldsymbol{w}_0^T\boldsymbol{x}_i + z_{i0}^2\boldsymbol{w}_0^T\boldsymbol{w}_0),$$

which we can rewrite due to the orthogonality of $\boldsymbol{w}_i$ as

$$J(\boldsymbol{w}_0,\boldsymbol{z}_0) = \frac{1}{n}\sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - 2z_{i0}\boldsymbol{w}_0^T\boldsymbol{x}_i + z_{i0}^2).$$

Minimizing J with respect to the unknown parameters z_{0i} we obtain that

$$z_{i0} = \boldsymbol{w}_0^T\boldsymbol{x}_i,$$

where the vectors on the rhs are known.

6.7.21 PCA Proof continued

We have now found the unknown parameters z_{i0}. These correspond to the projected coordinates and we can write

$$J(\boldsymbol{w}_0) = \frac{1}{p}\sum_i (\boldsymbol{x}_i^T\boldsymbol{x}_i - z_{i0}^2) = \text{const} - \frac{1}{n}\sum_i z_{i0}^2.$$

We can show that the variance of the projected coordinates defined by $\boldsymbol{w}_0^T\boldsymbol{x}_i$ are given by

$$\text{var}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \frac{1}{n}\sum_i z_{i0}^2,$$

since the expectation value of

$$\mathbb{E}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \mathbb{E}[z_{i0}] = \boldsymbol{w}_0^T\mathbb{E}[\boldsymbol{x}_i] = 0,$$

where we have used the fact that our data are centered.

Recalling our definition of the covariance as

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

we have thus that

$$\text{var}[\boldsymbol{w}_0^T\boldsymbol{x}_i] = \frac{1}{n}\sum_i z_{i0}^2 = \boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0.$$

We are almost there, we have obtained a relation between minimizing the reconstruction error and the variance and the covariance matrix. Minimizing the error is equivalent to maximizing the variance of the projected data.

6.7.22 The final step

We could trivially maximize the variance of the projection (and thereby minimize the error in the reconstruction function) by letting the norm-2 of $\boldsymbol{w}_0$ go to infinity. However, this norm since we want the matrix $\boldsymbol{W}$ to be an

orthogonal matrix, is constrained by $||\boldsymbol{w}_0||_2^2 = 1$. Imposing this condition via a Lagrange multiplier we can then in turn maximize

$$J(\boldsymbol{w}_0) = \boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 + \lambda_0(1 - \boldsymbol{w}_0^T\boldsymbol{w}_0).$$

Taking the derivative with respect to $\boldsymbol{w}_0$ we obtain

$$\frac{\partial J(\boldsymbol{w}_0)}{\partial \boldsymbol{w}_0} = 2\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 - 2\lambda_0\boldsymbol{w}_0 = 0,$$

meaning that

$$\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 = \lambda_0\boldsymbol{w}_0.$$

The direction that maximizes the variance (or minimizes the construction error) is an eigenvector of the covariance matrix! If we left multiply with $\boldsymbol{w}_0^T$ we have the variance of the projected data is

$$\boldsymbol{w}_0^T\boldsymbol{C}[\boldsymbol{x}]\boldsymbol{w}_0 = \lambda_0.$$

If we want to maximize the variance (minimize the construction error) we simply pick the eigenvector of the covariance matrix with the largest eigenvalue. This establishes the link between the minimization of the reconstruction function J in terms of an orthogonal matrix and the maximization of the variance and thereby the covariance of our observations encoded in the design/feature matrix $\boldsymbol{X}$.

The proof for the other eigenvectors $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$ can be established by applying the above arguments and using the fact that our basis of eigenvectors is orthogonal, see Murphy chapter 12.2. The discussion in chapter 12.2 of Murphy's text has also a nice link with the Singular Value Decomposition theorem. For categorical data, see chapter 12.4 and discussion therein.

Additional part of the proof for the other eigenvectors will be added by mid January 2020.

6.7.23 Geometric Interpretation and link with Singular Value Decomposition

This material will be added by mid January 2020.

6.7.24 Principal Component Analysis

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

The following Python code uses NumPy's **svd()** function to obtain all the principal components of the training set, then extracts the first two principal components. First we center the data using either **pandas** or our own code

```
import numpy as np
import pandas as pd
from IPython.display import display
np.random.seed(100)
# setting up a 10 x 5 vanilla matrix
rows = 10
cols = 5
X = np.random.randn(rows,cols)
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df -df.mean()
display(df)
```

(continues on next page)

(continued from previous page)

```
# we center it ourselves
X_centered = X - X.mean(axis=0)
# Then check the difference between pandas and our own set up
print(X_centered-df)
#Now we do an SVD
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
print(X2D)
```

```
          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
```

```
     0    1    2    3    4
0  0.0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0  0.0
4  0.0  0.0  0.0  0.0  0.0
5  0.0  0.0  0.0  0.0  0.0
6  0.0  0.0  0.0  0.0  0.0
7  0.0  0.0  0.0  0.0  0.0
8  0.0  0.0  0.0  0.0  0.0
9  0.0  0.0  0.0  0.0  0.0
[[-1.5378811  -0.94639099]
 [ 0.86145244  0.89288636]
 [-0.00445655  0.81633628]
 [ 0.07145103 -1.00433417]
 [ 2.03707133 -0.48476997]
 [ 0.72174172 -1.4557763 ]
 [-0.55854694  1.60673226]
 [ 1.6999536   0.43766686]
 [-1.10405456  0.31718909]
 [-2.18673098 -0.17953942]]
```

PCA assumes that the dataset is centered around the origin. Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

```
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
```

6.7.25 PCA and scikit-learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
print(X2D)
```

```
[[ 1.5378811  -0.94639099]
 [-0.86145244  0.89288636]
 [ 0.00445655  0.81633628]
 [-0.07145103 -1.00433417]
 [-2.03707133 -0.48476997]
 [-0.72174172 -1.4557763 ]
 [ 0.55854694  1.60673226]
 [-1.6999536   0.43766686]
 [ 1.10405456  0.31718909]
 [ 2.18673098 -0.17953942]]
```

After fitting the PCA transformer to the dataset, you can access the principal components using the components variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to

```
pca.components_.T[:, 0].
```

```
  File "<ipython-input-19-17314f270d45>", line 1
    pca.components_.T[:, 0].
                            ^
SyntaxError: invalid syntax
```

Another very useful piece of information is the explained variance ratio of each principal component, available via the $explained\_variance\_ratio$ variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

6.7.26 Back to the Cancer Data

We can now repeat the above but applied to real data, in this case our breast cancer data. Here we compute performance scores on the training data using logistic regression.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_
↪state=0)
```

(continues on next page)

(continued from previous page)

```
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Train set accuracy from Logistic Regression: {:.2f}".format(logreg.score(X_
↪train,y_train)))
# We scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Then perform again a log reg fit
logreg.fit(X_train_scaled, y_train)
print("Train set accuracy scaled data: {:.2f}".format(logreg.score(X_train_scaled,y_
↪train)))
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D_train = pca.fit_transform(X_train_scaled)
# and finally compute the log reg fit and the score on the training data
logreg.fit(X2D_train,y_train)
print("Train set accuracy scaled and PCA data: {:.2f}".format(logreg.score(X2D_train,
↪y_train)))
```

We see that our training data after the PCA decomposition has a performance similar to the non-scaled data.

6.7.27 More on the PCA

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization — in that case you will generally want to reduce the dimensionality down to 2 or 3. The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

You could then set $n\_components = d$ and run PCA again. However, there is a much better option: instead of specifying the number of principal components you want to preserve, you can set $n\_components$ to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
```

6.7.28 Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, Incremental PCA (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one minibatch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

6.7.29 Randomized PCA

Scikit-Learn offers yet another option to perform PCA, called Randomized PCA. This is a stochastic algorithm that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2)+O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n.

6.7.30 Kernel PCA

The kernel trick is a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (kPCA). It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold. For example, the following code uses Scikit-Learn's KernelPCA class to perform kPCA with an

```
from sklearn.decomposition import KernelPCA
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

6.7.31 LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly).

6.7.32 Other techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn.

Here are some of the most popular:

- **Multidimensional Scaling (MDS)** reduces dimensionality while trying to preserve the distances between the instances.
- **Isomap** creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances between the instances.
- **t-Distributed Stochastic Neighbor Embedding** (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).

- Linear Discriminant Analysis (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as a Support Vector Machine (SVM) classifier discussed in the SVM lectures.