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

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
partment of Physics and Astronomy and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, U

Jun 16, 2020

Introduction

Our emphasis throughout this series of lectures is 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.

© 1999-2020, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

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.

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.

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.

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!

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

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

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**.

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`.

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

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

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

Some famous Matrices.

- Diagonal if $a_{ij} = 0$ for $i \neq j$
- Upper triangular if $a_{ij} = 0$ for $i > j$
- Lower triangular if $a_{ij} = 0$ for $i < j$
- Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$
- Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$
- Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$
- Lower banded with bandwidth p: $a_{ij} = 0$ for $i > j + p$
- Upper banded with bandwidth p: $a_{ij} = 0$ for $i < j + p$
- Banded, block upper triangular, block lower triangular....

More Basic Matrix Features.

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

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

Numpy and arrays

Numpy provides an easy way to handle arrays in Python. The standard way to import this library is as

import numpy as np Here follows a simple example where we set up an array of ten elements, all determined by random numbers drawn according to the normal distribution, n = 10 x = np.random.normal(size=n) print(x) We defined a vector x with $n = 10$ elements with its values given by the Normal distribution $N(0, 1)$. Another alternative is to declare a vector as follows import numpy as np x = np.array([1, 2, 3]) print(x) Here we have defined a vector with three elements, with $x_0 = 1$, $x_1 = 2$ and $x_2 = 3$. Note that both Python and C++ start numbering array elements from 0 and on. This means that a vector with n elements has a sequence of entities $x_0, x_1, x_2, \ldots, x_{n-1}$. 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)

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) 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) or simply write them as double precision numbers (Python uses 64 bits as default for floating point type variables), that is import numpy as np x = np.log(np.array([4.0, 7.0, 8.0]) print(x) To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can 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)

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)

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

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] $\text{data}_p andas = pd.DataFrame(data) display(data_p andas)$

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 $\text{data}_p andas = pd.DataFrame(data, index = ['Frodo','Bilbo','Aragorn','Sam']) display(data_p andas)$ Thereafter we display the content of the row which begins with the index **Aragorn** $\text{display(data}_p andas.loc['Aragorn'])$

We can easily append data to this, for example $\text{new}_h obbit = 'FirstName' : ["Peregrin"],' LastName' : ["To$ $data_p andas.append(pd.DataFrame(new_h obbit, index = ['Pippin'])) display(data_p andas)$

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)

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 $_D ataFrame. As with \textbf{DataFrame}, most operations are vectorized, ac$

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$_m odelimportLinearRegression$

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

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

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$_m odel import Linear Regression$

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'$\epsilon_{\text{relative}}$') 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$_m odel import Linear Regression from sklearn.me$

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: ', linreg.intercept$_) print('Coefficient beta :', linreg.coef_) The mean squared error print("Mean square$
$Explained variance score : 1 is perfect prediction print('Variance score : Mean squared log error print('Mean squ$
$Mean absolute error print('Mean absolute error : plt.plot(x, ypredict, "r-") plt.plot(x, y,' ro') plt.axis([0.0, 1.0, 1.$

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

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

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

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

Here $a = \boldsymbol{y} - \tilde{\boldsymbol{y}}$. 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$_m$*odelimportRidgefromsklearn.preprocessingimportPolynomialFeaturesfromsklearn.pipelin*

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$_t$*ransform*(x[:, *np.newaxis*])*clf*3 = *LinearRegression*()*clf*3.*fit*(X, y)

Xplot=poly3.fit$_t$*ransform*(x[:, *np.newaxis*])*poly*3$_p$*lot* = *plt.plot*($x, clf3.predict(Xplot), label =$ *CubicFit*′)*plt.plot*($x, yn, color =$ *red*′, *label* = ”*TrueCubic*”)*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

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

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

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

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

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

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

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

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

- There is a volume term $a_1 A$ proportional with the number of nucleons (the energy is also an extensive quantity). When an assembly of nucleons of the same size is packed together into the smallest volume, each interior nucleon has a certain number of other nucleons in contact with it. This contribution is proportional to the volume.

- There is a surface energy term $a_2 A^{2/3}$. 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$$_asklfromsklearn.model_selectionimporttrain_test_splitfromsklearn.metricsimportmean_s$

Where to save the figures and data files PROJECT$_R$$OOT_DIR$ = "$Results$"$FIGURE_ID$ = "$Results/FigureFiles$"$DATA_ID$ = "$DataFiles/$"

if not os.path.exists(PROJECT$_R$$OOT_DIR$) : $os.mkdir(PROJECT_ROOT_DIR)$

if not os.path.exists(FIGURE$_I$$D$) : $os.makedirs(FIGURE_ID)$

if not os.path.exists(DATA$_I$$D$) : $os.makedirs(DATA_ID)$

def image$_p$$ath(fig_id)$: $returnos.path.join(FIGURE_ID, fig_id)$

def data$_p$$ath(dat_id)$: $returnos.path.join(DATA_ID, dat_id)$

def save$_f$$ig(fig_id)$: $plt.savefig(image_path(fig_id) + ".png", format =' png')$

infile = open(data$_p$$ath("MassEval2016.dat"),' r')$

Before we proceed, we define also a function for making our plots. You can obviously avoid this and simply set up various **matplotlib** commands every time you need them. You may however find it convenient to collect all such commands in one function and simply call this function. from pylab import plt, mpl plt.style.use('seaborn') mpl.rcParams['font.family'] = 'serif'

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

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

In particular, the program that outputs the final nuclear masses is written in Fortran with a specific format. It means that we need to figure out the format and which columns contain the data we are interested in. Pandas comes with a function that reads formatted output. After having admired the file, we are now ready to start massaging it with **pandas**. The file begins with some basic format information. """ This is taken from the data file of the mass 2016 evaluation. All files are 3436 lines long with 124 character per line. Headers are 39 lines long. col 1 : Fortran character control: 1 = page feed 0 = line feed format : a1,i3,i5,i5,i5,1x,a3,a4,1x,f13.5,f11.5,f11.3,f9.3,1x,a2,f11.3,f9.3,1x,i3,1x,f12.5,f11.5 These formats are reflected in the pandas widths variable below, see the statement widths=(1,3,5,5,5,1,3,4,1,13,11,11,9,1,2,11,9,1,3,1,12,11,1), Pandas has also a variable header, with length 39 in this case. """

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$_f wf(infile, usecols =$ $(2, 3, 4, 6, 11), names = ('N',' Z',' A',' Element',' Ebinding'), widths = (1, 3, 5, 5, 5, 1, 3, 4, 1, 13, 11, 11, 9, 1, 2, 11,$ $39, index_col = False)$

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$_n umeric(Masses['Ebinding'], errors =' coerce')Masses =$ $Masses.dropna()ConvertfromkeVtoMeV.Masses['Ebinding']/ = 1000$

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

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

Now we define five variables which contain the number of nucleons A, the number of protons Z and the number of neutrons N, the element name and finally the energies themselves. A = Masses['A'] Z = Masses['Z'] N = Masses['N'] Element = Masses['Element'] Energies = Masses['Ebinding'] print(Masses) The next step, and we will define this mathematically later, is to set up the so-called **design 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: Explained variance score: 1 is perfect prediction print('Variance score: Mean absolute error print('Mean absolute error: 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$_x label(r'$A = N + Z')ax.set_y label(r'E_{bind} /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$_f ig("Masses2016")plt.show()$

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
$\text{regr}_1 = DecisionTreeRegressor(max_depth = 5) regr_2 = DecisionTreeRegressor(max_depth = 7) regr_3 = DecisionTreeRegressor(max_depth = 9) regr_1.fit(X, Energies) regr_2.fit(X, Energies) regr_3.fit(X,$

$y_1 = regr_1.predict(X) y_2 = regr_2.predict(X) y_3 = regr_3.predict(X) Masses['Eapprox'] = y_3 Plottheresultsplt.figure() plt.plot(A, Energies, color = "blue", label = "Data", linewidth = 2) plt.plot(A, y_1, color = "red", label = "max_depth = 5", linewidth = 2) plt.plot(A, y_2, color = "green", label = "max_depth = 7", linewidth = 2) plt.plot(A, y_3, color = "m", label = "max_depth = 9", linewidth = 2)$

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

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 $\text{sklearn.neural}_network import MLPRegressor from sklearn.metrics import accuracy_score import seaborn as sns$

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

fig, ax = plt.subplots(figsize = (10, 10)) $\text{sns.heatmap(train}_accuracy, annot = True, ax = ax, cmap = "viridis") ax.set_title("Training Accuracy") ax.set_ylabel("\eta")$
$\text{ax.set}_xlabel("\lambda")$ plt.show()

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

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

and the proton separation energy reads

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

The two-neutron separation energy is defined as

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

and the two-proton separation energy is given by

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

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

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

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

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

or

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

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

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

To calculate say the neutron separation we need to multiply our masses with the nucleon number A (why?). Thereafter we pick the oxygen isotopes and simply compute the separation energies with two lines of code (note that most of the code here is a repeat of what you have seen before). Common imports import numpy as np import pandas as pd import matplotlib.pyplot as plt import os from pylab import plt, mpl plt.style.use('seaborn') mpl.rcParams['font.family'] = 'serif'

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

Where to save the figures and data files PROJECT$_R OOT_D IR$ = "$Results$"$FIGURE_I D$ = "$Results/FigureFiles$"$DATA_I D$ = "$DataFiles/$"

if not os.path.exists(PROJECT$_R OOT_D IR$) : $os.mkdir(PROJECT_R OOT_D IR)$

if not os.path.exists(FIGURE$_I D$) : $os.makedirs(FIGURE_I D)$

if not os.path.exists(DATA$_I D$) : $os.makedirs(DATA_I D)$

def image$_p ath(fig_i d)$: $returnos.path.join(FIGURE_I D, fig_i d)$

def data$_p ath(dat_i d)$: $returnos.path.join(DATA_I D, dat_i d)$

def save$_f ig(fig_i d)$: $plt.savefig(image_p ath(fig_i d) + ".png", format =' png')$

infile = open(data$_p ath("MassEval2016.dat"),' r')$

Read the experimental data with Pandas Masses = pd.read$_f wf(infile, usecols =$ $(2,3,4,6,11), names = ('N',' Z',' A',' Element',' Ebinding'), widths = (1,3,5,5,5,1,3,4,1,13,11,11,9,1,2,11,$ $39, index_c ol = False)$

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$_n umeric(Masses['Ebinding'], errors =' coerce')Masses =$ $Masses.dropna()ConvertfromkeVtoMeV.Masses['Ebinding']/ = 1000A = Masses['A']Z =$ $Masses['Z']N = Masses['N']Element = Masses['Element']Energies = Masses['Ebinding']*$ A

df = pd.DataFrame('A':A,'Z':Z, 'N':N,'Element':Element,'Energies':Energies) Her we pick the oyxgen isotopes Nucleus = df.loc[lambda df: df.Z==8, :] drop cases with no number Nucleus = Nucleus.dropna() Here we do the

magic and obtain the neutron separation energies, one line of code!! Nucleus['NeutronSeparationEnergies'] = Nucleus['Energies'].diff(+1) print(Nucleus) MakePlot([Nucleus.A], [Nucleus.NeutronSeparationEnergies], ['b'], ['Neutron Separation Energy'], ['A','S_n']) save$_fig('Nucleus')plt.show()$

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.

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 $i = 0, 1, 2, \ldots, n-1$
- p so-called explanatory (independent or predictor) variables $\boldsymbol{x}_i = [x_{i0}, x_{i1}, \ldots, x_{ip-1}]$ with $i = 0, 1, 2, \ldots, n-1$ 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.

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 $\boldsymbol{\beta} = [\beta_0, \ldots, \beta_{p-1}]^T$ are the *regression parameters*.

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

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.

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.

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

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

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

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

we can rewrite our equations as

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

The above design matrix is called a Vandermonde matrix.

Generalizing the fitting procedure as a linear algebra problem

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

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

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

Generalizing the fitting procedure as a linear algebra problem

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

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

and without loss of generality we rewrite again our equations as

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

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

Optimizing our parameters

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

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

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.

Our model for the nuclear binding energies

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

We restate the parts of the code we are most interested in. Common imports import numpy as np import pandas as pd import matplotlib.pyplot as plt from IPython.display import display import os

Where to save the figures and data files PROJECT$_R OOT_D IR$ = "$Results$"$FIGURE_I D$ = "$Results/FigureFiles$"$DATA_I D$ = "$DataFiles/$"

if not os.path.exists(PROJECT$_R OOT_D IR$) : $os.mkdir(PROJECT_R OOT_D IR)$

if not os.path.exists(FIGURE$_I D$) : $os.makedirs(FIGURE_I D)$

if not os.path.exists(DATA$_I D$) : $os.makedirs(DATA_I D)$

def image$_p ath(fig_i d)$: $returnos.path.join(FIGURE_I D, fig_i d)$

def data$_p ath(dat_i d)$: $returnos.path.join(DATA_I D, dat_i d)$

def save$_f ig(fig_i d)$: $plt.savefig(image_p ath(fig_i d) +$ ".png", $format =' png')$

infile = open(data$_p ath($"$MassEval2016.dat$"),$' r')$

Read the experimental data with Pandas Masses = pd.read$_f wf(infile, usecols =$ $(2, 3, 4, 6, 11), names = ('N',' Z',' A',' Element',' Ebinding'), widths = (1, 3, 5, 5, 5, 1, 3, 4, 1, 13, 11, 11, 9, 1, 2, 11,$ $39, index_c ol = False)$

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$_n umeric(Masses['Ebinding'], errors =' coerce')Masses = Masses.dropna()ConvertfromkeVtoMeV.Masses['Ebinding']/ = 1000$

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

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

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

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

throughout these lectures.

Optimizing our parameters, more details

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

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

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

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

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

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

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

It is also common to define the function Q as

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

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

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

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

which results in

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

or in a matrix-vector form as

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

Interpretations and optimizing our parameters

We can rewrite

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

as

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

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

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

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

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

Some useful matrix and vector expressions

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

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

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

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

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

Interpretations and optimizing our parameters

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

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

and with

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

we have

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

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

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

Own code for Ordinary Least Squares

It is rather straightforward to implement the matrix inversion and obtain the parameters $\boldsymbol{\beta}$. 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$_x label(r'$A = N + Z$')ax.set_y label(r'$E_{\mathrm{bind}}$ /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$_f ig("Masses2016OLS")plt.show()$

Adding error analysis and training set up

We can easily test our fit by computing the $R2$ score that we discussed in connection with the functionality of $_S cikit_L earn_i ntheintroductoryslides.Sincewearenotusing_S cikit-Learnherewecandefineourown\mathrm{R2}functionasdefR2(y_d ata, y_m odel) : return1-np.sum((y_d ata-y_m odel)**2)/np.sum((y_d ata-np.mean(y_d ata))**2)andwewouldbeusingitasprint(R2(Energie$

We can easily add our **MSE** score as def MSE(y$_d ata, y_m odel) : n = np.size(y_m odel)returnnp.sum((y_d ata - y_m odel) * *2)/n$

print(MSE(Energies,ytilde)) and finally the relative error as def RelativeError(y$_d ata, y_m odel) : returnabs((y_d ata - y_m odel)/y_d ata)print(RelativeError(Energies, ytilde))$

We could also add the so-called Huber norm, which we defined as

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

with $a = \boldsymbol{y} - \tilde{\boldsymbol{y}}$.

The χ^2 function

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

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

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

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

The χ^2 function

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

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

which results in

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

or in a matrix-vector form as

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

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

The χ^2 function

We can rewrite

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

as

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

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

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

The χ^2 function

If we then introduce the matrix

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

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

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

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

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

resulting in

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

The χ^2 function

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

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

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

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

and

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

The χ^2 function

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

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

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

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

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

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

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.

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.

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$_m odelasskl fromsklearn.metricsimport$

Where to save the figures and data files PROJECT$_R OOT_D IR$ = "$Results$"$FIGURE_I D$ = "$Results/FigureFiles$"$DATA_I D$ = "$DataFiles/$"

if not os.path.exists(PROJECT$_R OOT_D IR$) : $os.mkdir(PROJECT_R OOT_D IR)$

if not os.path.exists(FIGURE$_I D$) : $os.makedirs(FIGURE_I D)$

if not os.path.exists(DATA$_I D$) : $os.makedirs(DATA_I D)$

def image$_p ath(fig_i d)$: $returnos.path.join(FIGURE_I D, fig_i d)$

def data$_p ath(dat_i d)$: $returnos.path.join(DATA_I D, dat_i d)$

def save$_f ig(fig_i d)$: $plt.savefig(image_p ath(fig_i d) + ".png", format =' png')$

infile = open(data$_p ath("EoS.csv"),' r')$

Read the EoS data as csv file and organize the data into two arrays with density and energies EoS = pd.read$_c sv(infile, names = ('Density',' Energy'))EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors =' coerce')EoS = EoS.dropna()Energies = EoS['Energy']Density = EoS['Density']Thedesignmatrixnowasfunctionofvariouspolytrops X = np.zeros((len(Density), 4))X[:, 3] = Density * *(4.0/3.0)X[:, 2] = DensityX[:, 1] = Density * *(2.0/3.0)X[:, 0] = 1$

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: Explained

variance score: 1 is perfect prediction print('Variance score: Mean absolute error print('Mean absolute error: print(clf.coef_,$clf.intercept_$)

The Ridge regression with a hyperparameter lambda = 0.1 $_lambda$ = $0.1clf_ridge = skl.Ridge(alpha =_l\ ambda).fit(X, Energies)yridge = clf_ridge.predict(X)EoS['Eridge'] = yridgeThemeansquarederrorprint("Meansquarederror : Explainedvariancescore : 1isperfectpredictionprint('Variancescore : Meanabsoluteerrorprint('Meanabsoluteerror : print(clf_ridge.coef_,clf_ridge.intercept_)$

fig, ax = plt.subplots() ax.set$_x label(r'\rho$[fm^{-3}]') ax.set$_y label(r'Energyperparticle')ax.plot(EoS['Density'], E$ $0.7, lw = 2, label =' Theoreticaldata')ax.plot(EoS['Density'], EoS['Eols'], alpha =$ $0.7, lw = 2, c =' m', label =' OLS')ax.plot(EoS['Density'], EoS['Eridge'], alpha =$ $0.7, lw = 2, c =' g', label =' Ridge\lambda = 0.1')$ ax.legend() save$_fig$("$EoSfitting")plt.show()$

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

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

Splitting our Data in Training and Test data

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

import os import numpy as np import pandas as pd import matplotlib.pyplot as plt from sklearn.model$_selectionimporttrain_test_splitWheretosavethefiguresanddatafilesPROJECT_ROOT$ $"Results"FIGURE_ID = "Results/FigureFiles"DATA_ID = "DataFiles/"$

if not os.path.exists(PROJECT$_R OOT_D IR) : os.mkdir(PROJECT_ROOT_DIR)$

if not os.path.exists(FIGURE$_I D) : os.makedirs(FIGURE_ID)$

if not os.path.exists(DATA$_I D) : os.makedirs(DATA_ID)$

def image$_path(fig_id) : returnos.path.join(FIGURE_ID, fig_id)$

def data$_path(dat_id) : returnos.path.join(DATA_ID, dat_id)$

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

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

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$_csv(infile, names = ('Density',' Energy'))EoS['Energy'] = pd.to_numeric(EoS['Energy'], errors =' coerce')EoS = EoS.dropna()Energies = EoS['Energy']Density = EoS['Density']Thedesignmatrixnowasfunctionofvariouspolytrops X = np.zeros((len(Density), 5))X[:, 0] = 1X[:, 1] = Density * *(2.0/3.0)X[:, 2] = DensityX[:, 3] = Density**(4.0/3.0)X[:, 4] = Density**(5.0/3.0)Wesplitthedataintestandtrainingdata X_train_test_split(X, Energies, test_size = 0.2)matrixinversiontofindbetabeta =$

$np.linalg.inv(X_train.T.dot(X_train)).dot(X_train.T).dot(y_train)andthenmakethepredictionytilde = X_train@betaprint("TrainingR2")print(R2(y_train, ytilde))print("TrainingMSE")print(MSE(y_train, ytilde X_test@betaprint("TestR2")print(R2(y_test, ypredict))print("TestMSE")print(MSE(y_test, ypredict))$

The Boston housing data example

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

The features/predictors are

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

Housing data, the code

We start by importing the libraries import numpy as np import matplotlib.pyplot as plt

import pandas as pd import seaborn as sns and load the Boston Housing DataSet from **Scikit-Learn**

from sklearn.datasets import load$_boston$

boston$_dataset = load_boston()$

boston$_d$ataset is a dictionary let's check what it contains boston$_d$ataset.keys() Then we invoke Pandas boston = pd.DataFrame(boston$_d$ataset.data, $columns$ = boston$_d$ataset.feature$_n$ames)boston.head()boston['$MEDV'$] = boston$_d$ataset.target and preprocess the data check for missing values in all the columns boston.isnull().sum() We can then visualize the data set the size of the figure sns.set(rc='figure.figsize':(11.7,8.27))

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

It is now useful to look at the correlation matrix compute the pair wise correlation for all columns correlation$_m$atrix = boston.corr().round(2)usetheheatmapfunctionfromseabornto plotth $True$ to print the values inside the squares sns.heatmap(data = correlation$_m$atrix, annot = $True$) From the above coorelation plot we can see that **MEDV** is strongly correlated to **LSTAT** and **RM**. We see also that **RAD** and **TAX** are stronly correlated, but we don't include this in our features together to avoid multicolinearity

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']], c ['$LSTAT'$,'RM'])Y = boston['$MEDV'$] We split the data into training and test sets

from sklearn.model$_s$election import train$_t$est$_s$plit

splits the training and test data set in 80 assign random$_s$tate to any value. This ensures consistency. $X_t$rain, X_te train$_t$est$_s$plit($X, Y, test_size$ = 0.2, random$_s$tate = 5)print($X_t$rain.shape)print($X_t$est.shape)print($Y_t$rain.shap Then we use the linear regression functionality from **Scikit-Learn** from sklearn.linear$_m$odel import $LinearRegression$ from sklearn.metrics import mean$_s$quared$_e$rror, r2$_s$core

lin$_m$odel = $LinearRegression$()lin$_m$odel.fit($X_t$rain, $Y_t$rain)

model evaluation for training set

y$_t$rain$_p$redict = lin$_m$odel.predict($X_t$rain)rmse = (np.sqrt(mean$_s$quared$_e$rror($Y_t$rain, y$_t$rain$_p$redict)))r2 = r2$_s$core($Y_t$rain, y$_t$rain$_p$redict)

print("The model performance for training set") print("————————————") print('RMSE is '.format(rmse)) print('R2 score is '.format(r2)) print("")

model evaluation for testing set

y$_t$est$_p$redict = lin$_m$odel.predict($X_t$est)root mean square error of the model rmse = (np.sqrt(mean$_s$quared$_e$rror($Y_t$est, y$_t$est$_p$redict)))

r-squared score of the model r2 = r2$_s$core($Y_t$est, y$_t$est$_p$redict)

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

plotting the y$_t$est vs y$_p$redict we should have been a straight line plt.scatter($Y_t$est, y$_t$est$_p$redict)plt.show()

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 insight about the ordinary least squares approach.

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

Linear Regression Problems

One of the typical problems we encounter with linear regression, in particular when the matrix $\boldsymbol{X}$ (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.

Fixing the singularity

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

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

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.

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

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

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.

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 $n \times n$ 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!

Another Example

Consider the following matrix which can be SVD decomposed as

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

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

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

In the general case, where our design matrix $\boldsymbol{X}$ 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.

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.

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.

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

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

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 $\boldsymbol{U}$.

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.

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 interpretations 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.

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)

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.

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.

Resampling methods

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

Two resampling methods are often used in Machine Learning analyses,

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

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

Resampling approaches can be computationally expensive

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

Why resampling methods ?

Statistical analysis.

- Our simulations can be treated as *computer experiments*. This is particularly the case for Monte Carlo methods
- The results can be analysed with the same statistical tools as we would use analysing experimental data.

- As in all experiments, we are looking for expectation values and an estimate of how accurate they are, i.e., possible sources for errors.

Statistical analysis

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

Statistics

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

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

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

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

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

Statistics, moments

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

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

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

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

Statistics, central moments

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

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

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{var}(X)$:

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

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

Statistics, covariance

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

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

with

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

Statistics, more covariance

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

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

Covariance example

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.

Covariance in numpy

Importing various packages import numpy as np

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

Statistics, independent variables

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

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

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

By the linearity of the expectation value

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

Statistics, more variance

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

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

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

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

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

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

Statistics and stochastic processes

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

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

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

Statistics and sample variables

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

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

The *sample variance* is:

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

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

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

Statistics, sample variance and covariance

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

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

Statistics, law of large numbers

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

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

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

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

Statistics, more on sample error

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

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

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

Statistics

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

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

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

Statistics, central limit theorem

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

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

Linking the regression analysis with a statistical interpretation

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

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

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

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.

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

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

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

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

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).

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

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

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

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

We can also calculate the variance

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

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

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

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**, 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 a minimum, before starting to increase again. The training error reaches a saturation.

Resampling methods: Jackknife and Bootstrap

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

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

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

Resampling methods: Jackknife

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

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

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

Jackknife code example

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: print("original bias std. error") print("

return t

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

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

Resampling methods: Bootstrap

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

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

Resampling methods: Bootstrap background

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

Resampling methods: More Bootstrap background

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

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

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

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

Resampling methods: Bootstrap approach

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

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

Resampling methods: Bootstrap steps

The independent bootstrap works like this:

1. Draw with replacement n numbers for the observed variables $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$.

2. Define a vector $\boldsymbol{x}^*$ containing the values which were drawn from $\boldsymbol{x}$.

3. Using the vector $\boldsymbol{x}^*$ compute $\widehat{\theta}^*$ by evaluating $\widehat{\theta}$ under the observations $\boldsymbol{x}^*$.

4. Repeat this process k times.

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

Code example for the Bootstrap method

The following code starts with a Gaussian distribution with mean value $\mu = 100$ and variance $\sigma = 15$. We use this to generate the data used in the bootstrap analysis. The bootstrap analysis returns a data set after a given number of bootstrap operations (as many as we have data points). This data set consists of estimated mean values for each bootstrap operation. The histogram generated

by the bootstrap method shows that the distribution for these mean values is also a Gaussian, centered around the mean value $\mu = 100$ but with standard deviation $\sigma/\sqrt{n}$, where n is the number of bootstrap samples (in this case the same as the number of original data points). The value of the standard deviation is what we expect from the central limit theorem.

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

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

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

analysis print("Runtime: print("original bias std. error") print("return t

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

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

plt.show()

Various steps in cross-validation

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

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 $\boldsymbol{\sigma}_{-i}^2(\lambda)$, as

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

- Evaluate the prediction performance of these models on the test set by $\log\{L[y_i, \boldsymbol{X}_{i,*}; \boldsymbol{\beta}_{-i}(\lambda), \boldsymbol{\sigma}_{-i}^2(\lambda)]\}$. Or, by the prediction error $|y_i - \boldsymbol{X}_{i,*}\boldsymbol{\beta}_{-i}(\lambda)|$, the relative error, the error squared or the R2 score function.
- Repeat the first three steps such that each sample plays the role of the test set once.
- Average the prediction performances of the test sets at each grid point of the penalty bias/parameter. It is an estimate of the prediction performance of the model corresponding to this value of the penalty parameter on novel data. It is defined as

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

Cross-validation in brief

For the various values of k

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

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

The code here uses Ridge regression with cross-validation (CV) resampling and k-fold CV in order to fit a specific polynomial. import numpy as np import matplotlib.pyplot as plt from sklearn.model$_s electionimportKFoldfromsklearn.linear_m odelimportRidgefromsklea$

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

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

Cross-validation on Ridge regression using KFold only

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

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

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

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

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

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 $cross_validation_score$ from sklearn along with $KFold$

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

$estimated_mse_sklearn = np.zeros(nlambdas)i = 0forlmbinlambdas : ridge =$
$Ridge(alpha = lmb)$

X = poly.$fit_transform(x[:, np.newaxis])estimated_mse_folds = cross_validation_score(ridge, X, y[:$
$, np.newaxis], scoring =' neg_mean_squared_error', cv = kfold)$

$cross_validation_score$ return an array containing the estimated negative mse for every fold. we have to the the mean of eve
$np.mean(-estimated_mse_folds)$

i += 1

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

plt.figure()

plt.plot(np.log10(lambdas), $estimated_mse_sklearn, label =' cross_validation_score')plt.plot(np.log10(lambdas), estim$
$-', label =' KFold')$

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

plt.legend()

plt.show()

The bias-variance tradeoff

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

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 $\boldsymbol{\epsilon}$.

Example code for Bias-Variance tradeoff

import matplotlib.pyplot as plt import numpy as np from sklearn.linear$_m odelimportLinearRegression, Ridge, L$

np.random.seed(2018)

n = 500 n$_b oostraps = 100degree = 18Aquitehighvalue, justtoshow.noise =$ 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$_t rain, x_t est, y_t rain, y_t est =$ $train_t est_s plit(x, y, test_s ize = 0.2)$

Combine x transformation and model into one operation. Not neccessary, but convenient. model = make$_p ipeline(PolynomialFeatures(degree =$ $degree), LinearRegression(fit_i ntercept = False))$

The following (m x n$_b ootstraps)matrixholdsthecolumnvectorsy_p redforeachbootstrapiteration.y_p red =$ $np.empty((y_t est.shape[0], n_b oostraps))foriinrange(n_b oostraps) : x_,y_=resample(x_t rain, y_t rain)$

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 =' Datapoints')plt.scatter(x_test, np.mean 1), label =' Pred')plt.legend()plt.show()

Understanding what happens

import matplotlib.pyplot as plt import numpy as np from sklearn.linear_model import LinearRegression, Ridge, La

np.random.seed(2018)

n = 40 n_bootstraps = 100maxdegree = 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_bootstraps))foriinrange 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('Polynomialdegree :', degree)print('Error :', error[degree])print('Bias^2 :', bias[degree])print('Var :', variance[degree])print(' >= + = '.format(error[degree], bias[degree], variance[d variance[degree]))

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

Summing up

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

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

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

You may also find this recent article of interest.

Another Example from Scikit-Learn's Repository

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

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

print($_{doc}$)

import numpy as np import matplotlib.pyplot as plt from sklearn.pipeline import Pipeline from sklearn.preprocessing import PolynomialFeatures from sklearn.linear$_{m}odelimportLinearRegressionfromsklearn.model_{s}electionimportcross_{v}al_{s}core$

def true$_{f}un(X) : returnnp.cos(1.5 * np.pi * X)$

np.random.seed(0)

n$_{s}amples = 30degrees = [1, 4, 15]$

X = np.sort(np.random.rand(n$_{s}amples))y = true_{f}un(X)+np.random.randn(n_{s}amples)*$ 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$_{f}eatures = PolynomialFeatures(degree = degrees[i], include_{b}ias = False)linear_{r}egression = LinearRegression()pipeline = Pipeline([("polynomial_{f}eatures", polynomial_{f}eatu , np.newaxis], y)$

Evaluate the models using crossvalidation scores = cross$_val_score(pipeline, X[:
, np.newaxis], y, scoring = "neg$_m$ean$_s$quared$_e$rror", cv = 10)

X$_t$est = np.linspace(0, 1, 100)plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label =
"Model")plt.plot(X_test, true$_f$un(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 = : .2e(+/−: .2e)".format(degrees[i], −scores.mean(), scores.std()))plt.show()

More examples on bootstrap and cross-validation and errors

Common imports import os import numpy as np import pandas as pd import matplotlib.pyplot as plt from sklearn.linear$_m$odelimportLinearRegression, Ridge, Lassofromsklearn.model$_s$electi
"Results"FIGURE$_I$D = "Results/FigureFiles"DATA$_I$D = "DataFiles/"

if not os.path.exists(PROJECT$_R$OOT$_D$IR) : os.mkdir(PROJECT$_R$OOT$_D$IR)

if not os.path.exists(FIGURE$_I$D) : os.makedirs(FIGURE$_I$D)

if not os.path.exists(DATA$_I$D) : os.makedirs(DATA$_I$D)

def image$_p$ath(fig$_i$d) : returnos.path.join(FIGURE$_I$D, fig$_i$d)

def data$_p$ath(dat$_i$d) : returnos.path.join(DATA$_I$D, dat$_i$d)

def save$_f$ig(fig$_i$d) : plt.savefig(image$_p$ath(fig$_i$d) + ".png", format =' png')

infile = open(data$_p$ath("EoS.csv"),' r')

Read the EoS data as csv file and organize the data into two arrays with density and energies EoS = pd.read$_c$sv(infile, names = ('Density',' Energy'))EoS['Energy'] =
pd.to$_n$umeric(EoS['Energy'], errors =' coerce')EoS = EoS.dropna()Energies =
EoS['Energy']Density = EoS['Density']Thedesignmatrixnowasfunctionofvariouspolytrops

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$_t$rain, x$_t$est, y$_t$rain, y$_t$est = train$_t$est$_s$plit(X, Energies, test$_s$ize = 0.2)model =
LinearRegression(fit$_i$ntercept = True).fit(x$_t$rain, y$_t$rain)ypred = model.predict(x$_t$rain)ytilde =
model.predict(x$_t$est)testerror[polydegree]+ = mean$_s$quared$_e$rror(y$_t$est, ytilde)trainingerror[polydegree]+ =
mean$_s$quared$_e$rror(y$_t$rain, ypred)

testerror[polydegree] /= trials trainingerror[polydegree] /= trials print("Degree
of polynomial: print("Mean squared error on training data: print("Mean squared
error on test data:

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

The same example but now with cross-validation

Common imports import os import numpy as np import pandas as pd import matplotlib.pyplot as plt from sklearn.linear$_m$odelimportLinearRegression, Ridge, Lassofromsklearn.metricsimpo

Where to save the figures and data files PROJECT$_R OOT_D IR$ = "$Results$"$FIGURE_I D$ = "$Results/FigureFiles$"$DATA_I D$ = "$DataFiles/$"

if not os.path.exists(PROJECT$_R OOT_D IR$) : $os.mkdir(PROJECT_R OOT_D IR)$

if not os.path.exists(FIGURE$_I D$) : $os.makedirs(FIGURE_I D)$

if not os.path.exists(DATA$_I D$) : $os.makedirs(DATA_I D)$

def image$_p ath(fig_i d)$: $returnos.path.join(FIGURE_I D, fig_i d)$

def data$_p ath(dat_i d)$: $returnos.path.join(DATA_I D, dat_i d)$

def save$_f ig(fig_i d)$: $plt.savefig(image_p ath(fig_i d) + ".png", format =' png')$

infile = open(data$_p ath("EoS.csv"),' r')$

Read the EoS data as csv file and organize the data into two arrays with density and energies EoS = pd.read$_c sv(infile, names = ('Density',' Energy'))EoS['Energy'] = pd.to_n umeric(EoS['Energy'], errors =' coerce')EoS = EoS.dropna()Energies = EoS['Energy']Density = EoS['Density']Thedesignmatrixnowasfunctionofvariouspolytrops$

Maxpolydegree = 30 X = np.zeros((len(Density),Maxpolydegree)) X[:,0] = 1.0 estimated$_m se_s klearn = np.zeros(Maxpolydegree)polynomial = np.zeros(Maxpolydegree)k = 5kfold = KFold(n_s plits = k)$

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$_m se_f olds = cross_v al_s core(OLS, X, Energies, scoring =' neg_m ean_s quared_e rror', cv = kfold)[:, np.newaxis]estimated_m se_s klearn[polydegree] = np.mean(-estimated_m se_f olds)$

plt.plot(polynomial, np.log10(estimated$_m se_s klearn), label =' TestError')plt.xlabel('Polynomialdegree')plt$

Cross-validation with Ridge

import numpy as np import matplotlib.pyplot as plt from sklearn.model$_s electionimportKFoldfromsklearn.linec$

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$_s plits = k)estimated_m se_s klearn = np.zeros(nlambdas)i = 0forlmbinlambdas : ridge = Ridge(alpha = lmb)estimated_m se_f olds = cross_v al_s core(ridge, x, y, scoring =' neg_m ean_s quared_e rror', cv = kfold)estimated_m se_s klearn[i] = np.mean(-estimated_m se_f olds)i+ = 1plt.figure()plt.plot(np.log10(lambdas), estimated_m se_s klearn, label =' cross_v al_s core')plt.xlabel('log10(lambd$

The Ising model

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

$$H = -J \sum_{k}^{L} s_k s_{k+1}, \tag{14}$$

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$_t oolkits.axes_g rid1 importmake_a xes_l ocatableim$
$True) cmap_a rgs = dict(vmin = -1., vmax = 1., cmap =' seismic')$

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.

Reformulating the problem to suit regression

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

$$H = -\sum_{j}^{L} \sum_{k}^{L} s_j s_k J_{jk}. \tag{15}$$

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

$$\boldsymbol{H} = \boldsymbol{X} J, \tag{16}$$

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

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{17}$$

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 $X_t rain, X_t est, y_t rain, y_t est = train_t est_s plit(X, y, test_s ize = 0.2)$

Linear regression

In the ordinary least squares method we choose the cost function

$$C(\boldsymbol{X}, \boldsymbol{\beta}) = \frac{1}{n} \left\{ (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}) \right\}. \tag{18}$$

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

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

$\text{def ols}_inv(x : np.ndarray, y : np.ndarray) -> np.ndarray : returnscl.inv(x.T@x)@(x.T@y) beta = ols_inv(X_train_own, y_train)$

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

$$\boldsymbol{\beta} = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^T\boldsymbol{y}. \tag{19}$$

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$_s vd(x : np.ndarray, y : np.ndarray) - > np.ndarray : u, s, v = scl.svd(x) returnv.T@scl.pinv(scl.diagsvd(s, u.shape[0], v.shape[0]))@u.T@y$

beta = ols$_s vd(X_t rain_o wn, y_t rain)$

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$_a rgs) plt.title("OLS", fontsize =$ 18)$plt.xticks(fontsize = 18) plt.yticks(fontsize = 18) cb = fig.colorbar(im) cb.ax.set_yticklabels(cb.ax.get_ytick$ 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?

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

$$H = -J \sum_k^L s_k s_{k+1}, \tag{20}$$

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$_t oolkits.axes_g rid1 import make_a xes_l ocatable im$ $True) cmap_a rgs = dict(vmin = -1., vmax = 1., cmap =' seismic')$

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

$$H = -\sum_j^L \sum_k^L s_j s_k J_{jk}. \tag{21}$$

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

$$H = XJ, \tag{22}$$

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.

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}. \tag{23}$$

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 $\mathrm{X}_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.96)$

$\mathrm{X}_train_own = np.concatenate((np.ones(len(X_train))[:, np.newaxis], X_train), axis = 1)$

$\mathrm{X}_test_own = np.concatenate((np.ones(len(X_test))[:, np.newaxis], X_test), axis = 1)$

We will do all fitting with **Scikit-Learn**,

clf = skl.LinearRegression().fit($\mathrm{X}_train, y_train$) When extracting the J-matrix we make sure to remove the intercept $\mathrm{J}_sk = clf.coef_.reshape(L, L)$ And then we plot the results fig = plt.figure(figsize=(20, 14)) im = plt.imshow($\mathrm{J}_sk, **cmap_args)plt.title("LinearRegressionfromScikit-learn", fontsize = 18)plt.xticks(fontsize = 18)plt.yticks(fontsize = 18)cb = fig.colorbar(im)cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize = 18)plt.show()$ The results perfectly with our previous discussion where we used our own code.

Ridge regression

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

$$C(\boldsymbol{X}, \boldsymbol{\beta}; \lambda) = (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}) + \lambda \boldsymbol{\beta}^T\boldsymbol{\beta}. \tag{24}$$

$_lambda = 0.1clf_ridge = skl.Ridge(alpha =_l ambda).fit(X_train, y_train)J_ridge_sk = clf_ridge.coef_.reshape(L, L)fig = plt.figure(figsize = (20, 14))im = plt.imshow(J_ridge_sk, ** cmap_args)plt.title("RidgefromScikit-learn", fontsize = 18)plt.xticks(fontsize = 18)plt.yticks(fontsize = 18)cb = fig.colorbar(im)cb.ax.set_yticklabels(cb.ax.get_yticklabels(), fontsize = 18)$

plt.show()

LASSO regression

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

$$C(\boldsymbol{X}, \boldsymbol{\beta}; \lambda) = (\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}) + \lambda\sqrt{\boldsymbol{\beta}^T\boldsymbol{\beta}}. \tag{25}$$

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$_l asso = skl.Lasso(alpha =_l ambda).fit(X_t rain, y_t rain) J_l asso_s k = clf_l asso.coef_.reshape(L, L) fig = plt.figure(figsize = (20, 14)) im = plt.imshow(J_l asso_s k, **cmap_a rgs) plt.title("Lasso from Scikit − learn", fontsize = 18) plt.xticks(fontsize = 18) plt.yticks(fontsize = 18) cb = fig.colorbar(im) cb.ax.set_y ticklabels(cb.ax.get_y ticklabels(), fontsize = 18)$

plt.show()

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

Performance 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$_e rrors$ = "$ols_s k$" : $np.zeros(lambdas.size)$, "$ridge_s k$" : $np.zeros(lambdas.size)$, "$lasso_s k$" : $np.zeros(l$

test$_e rrors$ = "$ols_s k$" : $np.zeros(lambdas.size)$, "$ridge_s k$" : $np.zeros(lambdas.size)$, "$lasso_s k$" : $np.zeros(la$

plot$_c ounter = 1$

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

for i, $_l ambda in enumerate(tqdm.tqdm(lambdas)) : for key, method in zip(["ols_s k", "ridge_s k", "lasso_s k"], [skl$ $ambda), skl.Lasso(alpha =_l ambda)]) : method = method.fit(X_t rain, y_t rain)$

train$_e rrors[key][i] = method.score(X_t rain, y_t rain) test_e rrors[key][i] = method.score(X_t est, y_t est)$

omega = method.coef$_.reshape(L, L)$

plt.subplot(10, 5, plot$_c ounter) plt.imshow(omega, **cmap_a rgs) plt.title(r"plot_c ounter+ =$

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.

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$_s k$" : "r", "$ridge_s k$" : "y", "$lasso_s k$" : "c"

for key in train_errors : $plt.semilogx(lambdas, train_errors[key], colors[key], label = "Train0".format(key), linewidth = 4.0)$

for key in test_errors : $plt.semilogx(lambdas, test_errors[key], colors[key] + "--", label = "Test0".format(key), linewidth = 4.0)plt.legend(loc = "best", fontsize = 18)plt.xlabel(r"\lambda", fontsize=18)$ plt.ylabel(r"R^2", fontsize=18) $\text{plt.tick}_params(labelsize = 18)plt.show()$

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.