Nuclear Talent course on Machine Learning in Nuclear Experiment and Theory

Daniel Bazin[1]

Morten Hjorth-Jensen[1,2]

Michelle Kuchera[3]

Sean Liddick[4]

Raghuram Ramanujan[5]

Physics and Astronomy and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, East Lansing,
[2]Department of Physics and Center for Computing in Science Education, University of Oslo, Oslo, Norway
[3]Physics Department, Davidson College, Davidson, North Carolina, USA
ent of Chemistry and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, East Lansing, Michi
[5]Department of Mathematics and Computer Science, Davidson College, Davidson, North Carolina, USA

Jun 19, 2020

Introduction

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

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

Machine Learning, short overview

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 libraries written in Python for machine learning like Scikit-learn, Tensorflow, PyTorch and Keras, all freely available at their respective GitHub sites, encompass communities of developers in the thousands or more. And the number of code developers and contributors keeps increasing.

A multidisciplinary approach

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

Learning outcomes

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

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

1. Learn about basic data analysis, data optimization and machine learning;

2. Be capable of extending the acquired knowledge to other systems and cases;

3. Have an understanding of central algorithms used in data analysis and machine learning;

4. Understand methods for regression and classification;

5. Methods we will focus on are Linear and Logistic Regression, Decision trees, random forests, bagging and boosting and various variants of deep learning methods, from feed forward neural networks to more advanced methods;

6. Work on numerical examples to illustrate the theory;

Types of Machine Learning

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system. Some 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.

Essential elements of ML

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

The first ingredient is normally our data set (which can be subdivided into training 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.

Choice of Programming Language

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

Software and needed installations

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

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

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

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

Matrix properties reminder.

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

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 $x_0, x_1, x_2, \dots, x_{n-1}$. We could also let (recommended) Numpy to compute the logarithms of a specific array as import numpy as np x = np.log(np.array([4, 7, 8])) print(x)

More Examples

In the last example we used Numpy's unary function *np.log*. This function is highly tuned to compute array elements since the code is vectorized and does not require looping. We 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 $\in [0, 1] A = np.random.rand(n, n) print(A)$

More Examples, Covariance matrix

As we will see throughout these lectures, there are several extremely useful functionalities in Numpy. As an example, consider the discussion of the covariance matrix. Suppose we have defined three vectors $\hat{x}, \hat{y}, \hat{z}$ 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 $\hat{W}$

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

More on the Covariance Matrix

Our matrix is in turn 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)

Brief Reminder on Statistical analysis

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

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

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:

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

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

Statistics, moments

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

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

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

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

Statistics, central moments

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

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

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

$$\begin{aligned} \sigma_X^2 &= \mathrm{var}(X) = \langle (x - \langle x \rangle)^2 \rangle = \int (x - \langle x \rangle)^2 p(x)\, dx && (1) \\ &= \int \left(x^2 - 2x\langle x \rangle^2 + \langle x \rangle^2 \right) p(x)\, dx && (2) \\ &= \langle x^2 \rangle - 2\langle x \rangle \langle x \rangle + \langle x \rangle^2 && (3) \\ &= \langle x^2 \rangle - \langle x \rangle^2 && (4) \end{aligned}$$

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

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

with

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

Statistics, more covariance

If we consider the above covariance as a matrix $C_{ij} = \mathrm{cov}(X_i,\, X_j)$, 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, $(i \neq j)$:

$$\begin{aligned}\mathrm{cov}(X_i,\, X_j) &= \langle (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\rangle && (6)\\ &= \langle x_i x_j - x_i\langle x_j\rangle - \langle x_i\rangle x_j + \langle x_i\rangle\langle x_j\rangle\rangle && (7)\\ &= \langle x_i x_j\rangle - \langle x_i\langle x_j\rangle\rangle - \langle\langle x_i\rangle x_j\rangle + \langle\langle x_i\rangle\langle x_j\rangle\rangle && (8)\\ &= \langle x_i x_j\rangle - \langle x_i\rangle\langle x_j\rangle - \langle x_i\rangle\langle x_j\rangle + \langle x_i\rangle\langle x_j\rangle && (9)\\ &= \langle x_i x_j\rangle - \langle x_i\rangle\langle x_j\rangle && (10)\end{aligned}$$

Covariance example

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

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

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 $\boldsymbol{W}$

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

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

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

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 $U = \sum_i a_i X_i$:

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

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

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

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

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:

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

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. (5) is a measure of the correlation between two stochastic variables, the above defined sample covariance is a measure of the sequential correlation between succeeding measurements of a sample.

These quantities, being known experimental values, differ significantly from and must not be confused with the similarly named quantities for stochastic variables, mean 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 $x_1, x_2, \dots, x_n$ 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^2_X, respectively:

$$\lim_{n\to\infty} p_{\overline{X}_n}(x) = \left(\frac{n}{2\pi \mathrm{var}(X)}\right)^{1/2} e^{-\frac{n(x-\bar{x}_n)^2}{2\mathrm{var}(X)}} \tag{12}$$

Meet the Pandas

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

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

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

Data Frames in Pandas

In the above we have imported **pandas** with the shorthand **pd**, the latter has become the standard way we import **pandas**. We make then a list of various variables and reorganize the aboves lists into a **DataFrame** and then print out a neat table with specific column labels as *Name*, *place of birth* and *date of birth*. Displaying these results, we see that the indices are given by the default numbers from zero to three. **pandas** is extremely flexible and we can easily change the above indices by defining a new type of indexing as data$_{p}andas = pd.DataFrame(data, index = ['Frodo','Bilbo','Aragorn','Sam'])display(data_{p}andas)$ Thereafter we display the content of the row which begins with the index **Aragorn** display(data$_{p}andas.loc['Aragorn'])$

We can easily append data to this, for example new$_{h}obbit = 'FirstName' : ["Peregrin"],'LastName' : ["Too$ $data_{p}andas.append(pd.DataFrame(new_{h}obbit, index = ['Pippin']))display(data_{p}andas)$

More Pandas

Here are other examples where we use the **DataFrame** functionality to handle arrays, now with more interesting features for us, namely numbers. We set up a matrix of dimensionality 10×5 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.

Pandas Series

The **Series** class is another important class included in **pandas**. You can view it as a specialization of **DataFrame** but where we have just a single column of data. It shares many of the same features as $_DataFrame.Aswith$**DataFrame**$, mostoperationsarevectorized, achievin$

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

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

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

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

Simple linear regression model using scikit-learn. We start with perhaps our simplest possible example, using **Scikit-Learn** to perform linear regression analysis on a data set produced by us.

What follows is a simple Python code where we have defined a function y in terms of the variable x. Both are defined as vectors with 100 entries. The numbers in the vector $\hat{x}$ 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.

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

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

The Python code follows here. Importing various packages import numpy as np import matplotlib.pyplot as plt from sklearn.linear$_m odelimportLinearRegression$

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()

Simple linear regression model, what to expect. This example serves several aims. It allows us to demonstrate several aspects of data analysis and later machine learning algorithms. The immediate visualization shows that our linear fit is not impressive. It goes through the data points, but there are many outliers which are not reproduced by our linear regression. We could now play around with this small program and change for example the factor in front of x and the normal distribution. Try to change the function y to

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

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

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

Our first Cost/Loss function encounter. Minimizing the cost function is a central aspect of our discussions to come. Finding its minima as function of the model parameters (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.

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

Our first Cost/Loss function encounter, how do we define them? There are many ways to define the cost/loss function. A simpler approach is to look at the relative difference between the training data and the predicted data, that is we define the relative error (why would we prefer the MSE instead of the relative error?) as

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

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

We can modify easily the above Python code and plot the relative error instead import numpy as np import matplotlib.pyplot as plt from sklearn.linear$_{m}odelimportLinearRegression$

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

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

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

Scikit-Learn functionality. As mentioned above, **Scikit-Learn** has an impressive functionality. We can for example extract the values of 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$_{m}odelimportLinearRegressionfromsklearn.me$

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$_{)}print('Coefficientbeta :', linreg.coef_{)}Themeansquarederrorprint(''Meansquare$
$Explainedvariancescore : 1isperfectpredictionprint('Variancescore : Meansquaredlogerrorprint('Meansqu$
$Meanabsoluteerrorprint('Meanabsoluteerror : plt.plot(x, ypredict, ''r-'')plt.plot(x, y,' ro')plt.axis([0.0, 1.0, 1.$

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 $\tilde{y}_i$ 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

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

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

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

Another quantity taht we will meet again in our discussions of regression analysis is the mean absolute error (MAE), a risk metric corresponding to the expected value of the absolute error loss or what we call the $l1$-norm loss. In our discussion above we presented the relative error. The MAE is defined as follows

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

We present the squared logarithmic (quadratic) error

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

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

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

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

The Huber cost function is defined as

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

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

import matplotlib.pyplot as plt import numpy as np import random from sklearn.linear$_m odelimportRidgefromsklearn.preprocessingimportPolynomialFeaturesfromsklearn.pipelin$

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$_t ransform(x[:, np.newaxis])clf3 = LinearRegression()clf3.fit(X, y)$

Xplot=poly3.fit$_{t}ransform(x[:, np.newaxis])poly3_{p}lot = plt.plot(x, clf3.predict(Xplot), label =' CubicFit')plt.plot(x, yn, color =' red', label = "TrueCubic")plt.scatter(x, y, label =' Data', color =' orange', s = 15)plt.legend()plt.show()$

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

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

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

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

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

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

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

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

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

- There is a volume term $a_1 A$ 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$_modelasskl from sklearn.model_selection import train_test_split from sklearn.metrics import mean_s$

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

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

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

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

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

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

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

infile = open(data$_{p}ath(”MassEval2016.dat”),' r')$

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$_{f}wf(infile, usecols = (2,3,4,6,11), names = ('N','Z','A','Element','Ebinding'), widths = (1,3,5,5,5,1,3,4,1,13,11,11,9,1,2,11,$ $39, index_{c}ol = False)$

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()])

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$_xlabel(r'$A = N + Z$')ax.set_ylabel(r'$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$_fig(''Masses2016'')plt.show()$

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

Decision Tree Regression from sklearn.tree import DecisionTreeRegressor regr$_1 = DecisionTreeRegressor(max_depth = 5)regr_2 = DecisionTreeRegressor(max_depth = 7)regr_3 = DecisionTreeRegressor(max_depth = 9)regr_1.fit(X, Energies)regr_2.fit(X, Energies)regr_3.fit(X,$

y$_1 = regr_1.predict(X)y_2 = regr_2.predict(X)y_3 = regr_3.predict(X)Masses['Eapprox'] = y_3Plottheresultsplt.figure()plt.plot(A, Energies, color = ''blue'', label = ''Data'', linewidth = 2)plt.plot(A, y_1, color = ''red'', label = ''max_depth = 5'', linewidth = 2)plt.plot(A, y_2, color = ''green'', label = ''max_depth = 7'', linewidth = 2)plt.plot(A, y_3, color = ''m'', label = ''max_depth = 9'', linewidth = 2)$

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

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

X$_train = XY_train = Energiesn_hidden_neurons = 100epochs = 100storemodelsforlateruseeta_vals = np.logspace(-5, 1, 7)lmbd_vals = np.logspace(-5, 1, 7)storethemodelsforlateruseDNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype = object)train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals))$ $forj, lmbdinenumerate(lmbd_vals) : dnn = MLPRegressor(hidden_layer_sizes = (n_hidden_neurons), activation =' logistic', alpha = lmbd, learning_rate_init = eta, max_iter = epochs)dnn.fit(X_train, Y_train)DNN_scikit[i][j] = dnntrain_accuracy[i][j] = dnn.score(X_train, Y_train)$

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

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

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

and the proton separation energy reads

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

The two-neutron separation energy is defined as

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

and the two-proton separation energy is given by

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

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

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

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

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

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$_{f}ig('Nucleus')plt.show()$

Prediction versus estimation; correlation versus causation. When you hear these phrases in the context of machine learning, what do you think of? Maybe one thinks of the difference between classifying new data points and generating new data point ssifying new data points and generating new data points. Or perhaps one considers that correlation is a symmetric as- sessment (e.g., if A is correlated with B, then B is correlated with A), but cau- 1 sation is directional (e.g., if A causes B, B does not necessarily cause A) . These concepts are in some sense the difference between machine learning and statistics. In machine learning and prediction based tasks, we are often in- terested in developing algorithms that are capable of learning patterns from given data in an automated fashion, and then using these learned patterns to make predictions or assessments of newly given data. In many cases, our pri- mary concern is the quality of the predictions or assessments, and we are less concerned about the underlying patterns that were learned in order make these predictions.

s. Neural networks are, in some sense, the epitome of this point of view. In various contexts they are incredibly good at making predictions, but

they are often referred to as “black box” methods due to the difficulty in understanding the model by which they make such predictions. For example, the most powerful convolutional neural networks are incredibly good at classify- ing natural images (sometimes even better than humans), but it is very difficult to understand the mechanisms by which they make such predictions. In (classical) statistics and estimation, one is more concerned with the un- derlying model that makes the prediction. In other words, are the parameters of the model that makes the prediction statistically significant? Or could sev- eral other models (i.e., different parameter choices) have made the same pre- diction? This is the correlation versus causation issue. It comes up, for example and perhaps most notably, in medical trials and studies, in which one must not only find correlations and patterns in the data, but one must find the causal factors of a disease, so that one may develop and prescribe treatment.

In science and engineering, one is frequently called upon to infer (or learn) a quantitative model M for a given set of sample points X D fx1; x2; : : : ; xNg RD. For instance, Figure 1.1 shows a simple example in which one is given a set of four sample points in a two-dimensional plane. Obviously, these points can be fit perfectly by a (one-dimensional) straight line L. The line can then be called a “model” for the given points. The reason for inferring such a model is that it serves many useful purposes. On the one hand, the model can reveal information encoded in the data or underlying mechanisms from which the data were generated. In addition, it can simplify the representation of the given data set and help

predict future samples. In the case of the four points shown in Figure 1.1, the line model gives a more compact one-dimensional representation than the original twodimensional plane P. It also suggests that any new point (if generated with a similar mechanism as the existing points) will likely fall on the same line.

A first important consideration to keep in mind is that inferring the “correct” model for a given data set is an elusive, if not impossible, task. The fundamental difficulty is that if we are not specific about what we mean by a “correct” model, there could easily be many different models that fit the given data set “equally well.” For instance, in the example shown in Figure 1.1, any smooth curve that passes through the sample points would seem to be as valid a model as the straight line. Furthermore, if there were noise in the given sample points, then any curve, including the line, passing through the points exactly would unlikely be the “true model.” The question now is this: in what sense can we say that a model is correct or optimal for a given data set? To make the model inference problem well posed, i.e., to guarantee that there is a unique optimal model for the given data, we need to impose additional assumptions or restrictions on the class of models considered. To this end, we should not be looking for just any model that can describe the data. Instead, we should look for a model M that is the best among a restricted class of modelsM.4 In addition, to make the model inference problem computationally tractable, we need to specify how restricted the class of models needs to be. A common strategy, known as the principle of Occam's razor,5 is to try to get away

with the simplest possible class of models that is just necessary to describe the data or solve the problem at hand. More precisely, the model class should be rich enough to contain at least one model that can fit the data to a desired accuracy and yet be restricted enough that it is relatively simple to find the best model for the given data. Thus, in engineering practice, the most popular strategy is to start from the simplest class of models and increase the complexity of the models only when the simpler models become inadequate. For instance, to fit a set of sample points, one may first try the simplest class of models, namely linear models, followed by the class of hybrid (piecewise) linear models (subspaces), and then followed by the class of (piecewise) nonlinear models (submanifolds). One of the goals of this book is to demonstrate that among them, piecewise linear models can already achieve an excellent balance between expressiveness and simplicity for many important practical data sets and problems.

There are essentially two main categories of models and approaches for modeling a data set. Methods of the first category model the data as random samples from a probability distribution and try to learn this distribution from the data. We call such models statistical models. Models of the second category model the overall geometric shape of the data set with deterministic models such as subspaces, smooth manifolds, or topological spaces.6 We call such models geometric models.