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

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Apr 20, 2019

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 the simulation of financial transactions or disease models. 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 packages 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.

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, 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.

Useful Python packages

Here we list several useful Python packages you may wish to install (if you use anaconda many of these are already there)

- Numpy
- Pandas
- Xarray
- Scipy
- Matplotlib
- autodiff
- sympy
- Scikit-learn
- tensorflow and keras
- pytorch

Installing R, C++, cython or Julia

You will also find it convenient to utilize R. Although we will mainly use Python during lectures and in various projects and exercises, we provide a full R set of codes for the same examples. 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 tuned to statistically analysis and allows for an easy usage of the tools 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/IPython 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/IPython 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.

Numpy examples and Important Matrix and vector handling packages

There are several central software packages 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`.

When dealing with matrices and vectors a central issue is memory handling and allocation. If our code is written in Python the way we declare these objects and the way they are handled, interpreted and used by say a linear algebra library, requires codes that interface our Python program with such libraries. For Python programmers, **Numpy** is by now the standard Python package for numerical arrays in Python as well as the source of functions which act on these arrays. These functions span from eigenvalue solvers to functions that compute the mean value, variance or the covariance matrix. If you are not familiar with how arrays are handled in say Python or compiled languages like C++ and Fortran, the sections in this chapter may be useful. For C++ programmer, **Armadillo** is

widely used library for linear algebra and eigenvalue problems. In addition it offers a convenient way to handle and organize arrays. We discuss this library as well. Before we proceed we believe it may be convenient to repeat some basic features of matrices and vectors.

Basic Matrix Features

Matrix properties reminder.

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

Basic Matrix Features

The inverse of a matrix is defined by

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

Basic Matrix Features

Matrix Properties Reminder.

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....

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 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, 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 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

Here we have 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

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

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)

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 We can continue this was by printing out other columns or rows. The example here prints out the second column 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 or initializing all elements to or as unitarily distributed random numbers (see the material on random number generators in the statistics part)

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. For a more in-depth discussion of the covariance and covariance matrix and its meaning, we refer you to the lectures on statistics. 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 $3times3$ covariance matrix $\hat{\Sigma}$ via the Numpy function **np.cov()**. In our review of statistical functions and quantities we will discuss more about the meaning of the covariance matrix. Here we note that we can 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.

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. 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, city of residence and age, 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.

Here are other examples where we use the **DataFrame** functionality to handle arrays

Reading Data and fitting Nuclear Masses

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, 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.

We are going to start with a classic case from nuclear physics, namely all available data on binding energies. We will then show some of the strength of packages like **scikit-learn** in fitting 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 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.

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 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

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

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 as

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

We can modify easily the above Python code and plot the relative error instead

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

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

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

The **r2score** function computes R^2, the coefficient of determination. It provides a measure of how well future samples are likely to be predicted by the

model. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of $\hat{y}$, disregarding the input features, would get a R^2 score of 0.0.

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

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

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

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

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

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

Finally 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.

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.

Similarly, using **R**, we can perform similar studies.

Brief reminder on masses and binding energies. Let us return to 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.

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

$$u = M(^{12}\mathrm{C})/12 = 931.49386 \text{ MeV}/c^2.$$

The nucleon masses are

$$m_p = 938.27203(8)\ \text{MeV}/c^2 = 1.00727646688(13)u,$$

and

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

In the 2016 mass evaluation of by W.J.Huang, G.Audi, M.Wang, F.G.Kondev, S.Naimi and X.Xu there are xxx nuclei measured with an accuracy of xxx MeV or better, and xxx nuclei measured with an accuracy of greater than xxx MeV. For heavy nuclei one observes several chains of nuclei with a constant $N - Z$ value whose masses are obtained from the energy released in α-decay.

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.

Our next step is to read the data on experimental binding energies and reorganize them as functions of the mass number A and 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.

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.

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 **scikitlearn** afterwards. Our first attempt is to make a fit using the standard least squares functionality of **numpy**. Here we will also show functionalities like the computation of the mean values, the variance and the covariance.

Now we define five variables which contain the nucleon number, the number of protons and neutrons, the element name and finally the energies themselves. 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. With **scikitlearn** we are now ready to use linear regression and fit our data. 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.

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

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.

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**, one to two lines of code will allows 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 are 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 or masses with the nucleon number A (why?). The we pick the oxygen isotopes and simply compute the separation energis in two lines of code.

A first summary