Week 34: Introduction to the course, Logistics and Practicalities

Morten Hjorth-Jensen[1]

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway[1]

Week 34, August 18-22, 2025

Overview of first week

1. The sessions on Tuesdays and Wednesdays last four hours for each group (four groups in total) and will include lectures in a flipped mode (promoting active learning) and work on exercices and projects.
2. The sessions will begin with lectures, discussions, questions and answers about the material to be covered every week. Videos and teaching material will be announced in due time.
3. There are four groups:
   - Tuesdays 815am-12pm and 1215pm-4pm
   - Wednesdays 815am-12pm and 1215pm-4pm.
4. On Mondays we have a regular lecture which will be organized as a mix of active learning sessions and regular lectures. These lectures/active learning sessions start at 215pm and end at 4pm and serve the aims of giving an overview over various topics as well as solving specific problems. These lectures will also be recorded. Lectures can be attended in person or via zoom at `https://uio.zoom.us/my/mortenhj`

Videos and learning material with reading suggestions will be made available before each week starts.

Schedule first week

- August 18: Lecture: Presentation of course, Linear regression, examples and theory
- August 19: Introduction to software and repetition of Python Programming, linear algebra and basic elements of statistics. Please select group.
- August 20: Introduction to software and repetition of Python Programming, linear algebra and basic elements of statistics. Please select group.

Lectures and ComputerLab

- Mondays: regular lectures/active learning sessions (2.15pm-4pm)
- The sessions on Tuesdays and Wednesdays last four hours and will include partly lectures and discussions in the beginning.
- Weekly reading assignments and videos needed to solve projects and exercises.
- Weekly exercises. You can hand in exercises if you want and get an extra score, see below.
- Detailed lecture notes, exercises, all programs presented, projects etc can be found at the homepage of the course.
- Weekly plans and all other information are on the official website. This info will also be conveyed via weekly emails.
- No final exam, three projects that are graded and have to be approved.

Communication channels

- Communications (email and more) via `canvas.uio.no`

Course Format

- Three compulsory projects. Electronic reports only using Canvas to hand in projects and git as version control software and GitHub for repository (or GitLab) of all your material.
- Evaluation and grading: The three projects are graded and each counts 1/3 of the final mark. No final written or oral exam.
  1. For the last project each group/participant submits a proposal or works with suggested (by us) proposals for the project.
  2. If possible, we would like to organize the last project as a workshop where each group presents this to all other participants of the course
  3. Based on feedback etc, each group finalizes the report and submits for grading.
- Python is the default programming language, but feel free to use C/C++, Julia and/or Fortran or other programming languages. All source codes discussed during the lectures can be found at the webpage and github address of the course.

Teachers

- Morten Hjorth-Jensen, morten.hjorth-jensen@fys.uio.no
  - **Phone**: +47-48257387
  - **Office**: Department of Physics, University of Oslo, Eastern wing, room FØ470
  - **Office hours**: *Anytime*! Individual or group office hours can be arranged either in person or via zoom. Feel free to send an email for planning.
- Ida Torkjellsdatter Storehaug, i.t.storehaug@fys.uio.no
- Oskar Leinonen, oskarlei@fys.uio.no
- Mia-Katrin Ose Kvalsund, m.k.o.kvalsund@fys.uio.no
- Karl Henrik Fredly, k.h.fredly@fys.uio.no
- Eir Eline Hørlyk, e.e.horlyk@fys.uio.no
- Britt S. Haanen, b.s.m.haanen@fys.uio.no

Deadlines for projects (tentative)

1. Project 1: October 6 (available September 1) graded with feedback)
2. Project 2: November 3 (available October 6, graded with feedback)
3. Project 3: December 8 (available November 3, graded with feedback)

Extra Credit (not mandatory), weekly exercise assignments, 10 in total (due Friday same week), 10% additional score. The extra credit assignments are due each Friday and can be uploaed to **Canvas** in your preferred format (although we prefer jupyter-notebooks). First assignment is for week 35. Each weekly exercise set gives one additional point to the final score, see below on grading.

Grading

Grades are awarded on a scale from A to F, where A is the best grade and F is a fail. There are three projects which are graded and each project counts 1/3 of the final grade. The total score is thus the average from all three projects.

The final number of points is based on the average of all projects and the grade follows the following table:

- 92-100 points: A
- 77-91 points: B
- 58-76 points: C
- 46-57 points: D
- 40-45 points: E
- 0-39 points: F-failed

In addition you can get an extra score for weekly assignments (10 in total and due each Friday). Each weekly assignment counts 1 point. As an example, this means that if your average after three projects is 88 points and you have handed in and gotten approved four weekly exercises, the total score is 88+4=92, which translates into an A.

Reading material

The lecture notes are collected as a jupyter-book at https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.
In addition to the lecture notes, we recommend the books of Rasckha et al and Goodfellow et al. We will follow these texts closely and the weekly reading assignments refer to these texts.
The text by Hastie et al is also widely used in the Machine Learning community. See next slide for link to textbooks.

Main textbooks

- Goodfellow, Bengio, and Courville (GBC), Deep Learning https://www.deeplearningbook.org/
- Sebastian Raschka, Yuxi Lie, and Vahid Mirjalili (RLM), Machine Learning with PyTorch and Scikit-Learn at https://www.packtpub.com/product/machine-learning-with-pytorch-and-scikit-learn/9781801819312, see also https://sebastianraschka.com/blog/2022/ml-pytorch-book.html

The weekly reading suggestions are all from these two texts. The text by GBC can be accessed chapter by chapter from the abovementioned URL. Each chapter of RLM gives access to the pertinent notebooks. These notebooks are highly recommended.

Other popular texts

Other texts

- Christopher M. Bishop (CB), Pattern Recognition and Machine Learning
- Hastie, Tibshirani, and Friedman (HTF), The Elements of Statistical Learning, Springer.
- Aurelien Geron (AG), Hands-On Machine Learning with Scikit-Learn and TensorFlow, O'Reilly. This text is very useful since it contains many code examples and hands-on applications of all algorithms discussed in this course.
- Kevin Murphy (KM), Probabilistic Machine Learning, an Introduction
- David Foster (DF), Generative Deep Learning, `https://www.oreilly.com/library/view/generative-deep-learning/9781098134174/`
- Babcock and Gavras (BG), Generative AI with Python and TensorFlow, `https://github.com/PacktPublishing/Hands-On-Generative-AI-with-Python-and-TensorFlow-2`

Reading suggestions week 34

This week: Refresh linear algebra, GBC chapter 2. Install scikit-learn. See lecture notes for week 34 at https://compphysics.github.io/MachineLearning/doc/web/course.html (these notes).

Prerequisites

Basic knowledge in programming and mathematics, with an emphasis on linear algebra. Knowledge of Python or/and C++ as programming languages is strongly recommended and experience with Jupiter notebook is recommended. Required courses are the equivalents to the University of Oslo mathematics courses MAT1100, MAT1110, MAT1120 and at least one of the corresponding computing and programming courses INF1000/INF1110 or MAT-INF1100/MAT-INF1100L/BIOS1100/KJM-INF1100. Most universities offer nowadays a basic programming course (often compulsory) where Python is the recurring programming language.

Topics covered in this course: Statistical analysis and optimization of data

The course has two central parts

1. Statistical analysis and optimization of data
2. Machine learning

These topics will be scattered thorughout the course and may not necessarily be taught separately. Rather, we will often take an approach (during the lectures and project/exercise sessions) where say elements from statistical data analysis are mixed with specific Machine Learning algorithms.

Statistical analysis and optimization of data

We plan to cover the following topics:

- Basic concepts, expectation values, variance, covariance, correlation functions and errors;
- Simpler models, binomial distribution, the Poisson distribution, simple and multivariate normal distributions;
- Central elements of Bayesian statistics and modeling;
- Gradient methods for data optimization;
- Monte Carlo methods, Markov chains, Gibbs sampling and Metropolis-Hastings sampling (tentative);
- Estimation of errors and resampling techniques such as the cross-validation, blocking, bootstrapping and jackknife methods;
- Principal Component Analysis (PCA) and its mathematical foundation;

Machine Learning

- Pre deep-learning revolution (2008 approx)
  - Linear Regression and Logistic Regression, classification and regression problems;
  - Bayesian linear and logistic regression, kernel regression;
  - Decisions trees, Random Forests, Bagging and Boosting methods;
  - Support vector machines (only survey);
  - Unsupervised learning and dimensionality reduction, from PCA to clustering;

Deep learning methods

- Deep learning
  - Neural networks and deep learning;
  - Convolutional neural networks;
  - Recurrent neural networks;
  - Autoencoders
  - Generative methods with an emphasis on Boltzmann Machines, Variational Autoencoders and Generalized Adversarial Networks(covered by FYS5429);

Hands-on demonstrations, exercises and projects aim at deepening your understanding of these topics.

Extremely useful tools, strongly recommended

and discussed at the lab sessions

- GIT for version control, and GitHub or GitLab as repositories, highly recommended. This will be discussed during the first exercise session
- Anaconda and other Python environments, see intro slides and links to programming resources at
  `https://computationalscienceuio.github.io/RefreshProgrammingSkills/intro.html`

Other courses on Data science and Machine Learning at UiO

- FYS5419 Quantum Computing and Quantum Machine Learning
- FYS5429 Advanced Machine Learning for the Physical Sciences
- STK2100 Machine learning and statistical methods for prediction and classification.
- IN3050/4050 Introduction to Artificial Intelligence and Machine Learning. Introductory course in machine learning and AI with an algorithmic approach.
- STK-INF3000/4000 Selected Topics in Data Science. The course provides insight into selected contemporary relevant topics within Data Science.
- IN4080 Natural Language Processing. Probabilistic and machine learning techniques applied to natural language processing.

Other courses on Data science and Machine Learning at UiO, contn

- STK-IN4300 Statistical learning methods in Data Science. An advanced introduction to statistical and machine learning. For students with a good mathematics and statistics background.
- IN3310/4310 Deep Learnig for Image Analysis
- STK4051 Computational Statistics
- STK4021 Applied Bayesian Analysis and Numerical Methods

Learning outcomes

- Learn about basic data analysis, statistical analysis, Bayesian statistics, Monte Carlo sampling, data optimization and machine learning;
- Be capable of extending the acquired knowledge to other systems and cases;
- Have an understanding of central algorithms used in data analysis and machine learning;
- Understand linear methods for regression and classification, from ordinary least squares, via Lasso and Ridge to Logistic regression;
- Learn about neural networks and deep learning methods for supervised and unsupervised learning. Emphasis on feed forward neural networks, convolutional and recurrent neural networks;
- Learn about about decision trees, random forests, bagging and boosting methods;

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.

Essential elements of ML

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

- The first ingredient is normally our data set (which can be subdivided into training, validation and test data). Many find the most difficult part of using Machine Learning to be the set up of your data in a meaningful way.
- The second item is a model which is normally a function of some parameters. The model reflects our knowledge of the system (or lack thereof). As an example, if we know that our data show a behavior similar to what would be predicted by a polynomial, fitting our data to a polynomial of some degree would then determin our model.
- The last ingredient is a so-called **cost/loss** function (or error or risk function) which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.

Essential elements of ML

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

- The first ingredient is normally our data set (which can be subdivided into training, validation and test data). Many find the most difficult part of using Machine Learning to be the set up of your data in a meaningful way.
- The second item is a model which is normally a function of some parameters. The model reflects our knowledge of the system (or lack thereof). As an example, if we know that our data show a behavior similar to what would be predicted by a polynomial, fitting our data to a polynomial of some degree would then determin our model.
- The last ingredient is a so-called **cost/loss** function (or error or risk function) which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.

Essential elements of ML

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

- The first ingredient is normally our data set (which can be subdivided into training, validation and test data). Many find the most difficult part of using Machine Learning to be the set up of your data in a meaningful way.
- The second item is a model which is normally a function of some parameters. The model reflects our knowledge of the system (or lack thereof). As an example, if we know that our data show a behavior similar to what would be predicted by a polynomial, fitting our data to a polynomial of some degree would then determin our model.
- The last ingredient is a so-called **cost/loss** function (or error or risk function) which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.

Essential elements of ML

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

- The first ingredient is normally our data set (which can be subdivided into training, validation and test data). Many find the most difficult part of using Machine Learning to be the set up of your data in a meaningful way.
- The second item is a model which is normally a function of some parameters. The model reflects our knowledge of the system (or lack thereof). As an example, if we know that our data show a behavior similar to what would be predicted by a polynomial, fitting our data to a polynomial of some degree would then determin our model.
- The last ingredient is a so-called **cost/loss** function (or error or risk function) which allows us to present an estimate on how good our model is in reproducing the data it is supposed to train.

An optimization/minimization problem

At the heart of basically all Machine Learning algorithms we will encounter so-called minimization or optimization algorithms. A large family of such methods are so-called **gradient methods**.

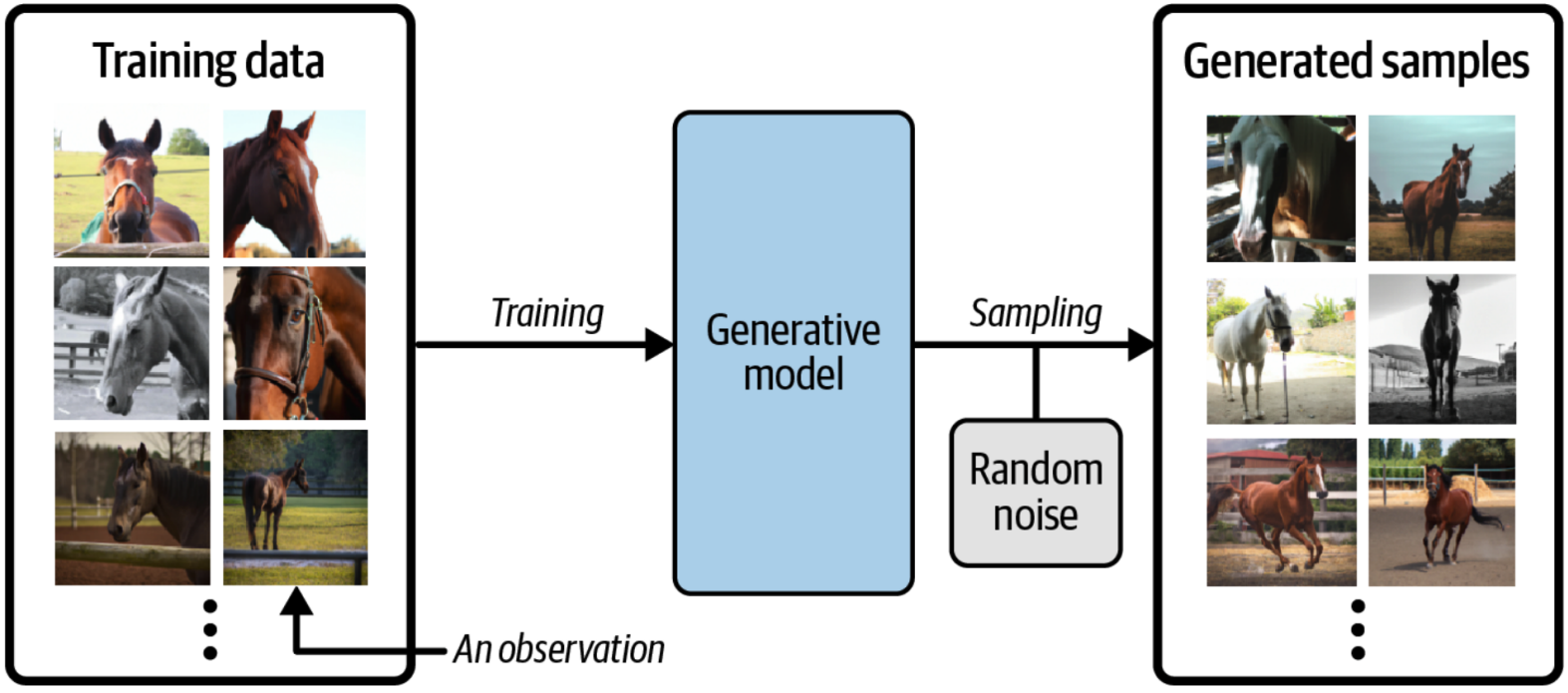
The plethora of machine learning algorithms/methods

1. Deep learning: Neural Networks (NNs), Convolutional NNs, Recurrent NNs, Transformers, Boltzmann machines, autoencoders and variational autoencoders and generative adversarial networks and other generative models
2. Bayesian statistics and Bayesian Machine Learning, Bayesian experimental design, Bayesian Regression models, Bayesian neural networks, Gaussian processes and much more
3. Dimensionality reduction (Principal component analysis), Clustering Methods and more
4. Ensemble Methods, Random forests, bagging and voting methods, gradient boosting approaches
5. Linear and logistic regression, Kernel methods, support vector machines and more
6. Reinforcement Learning; Transfer Learning and more

What Is Generative Modeling?

Generative modeling can be broadly defined as follows:
Generative modeling is a branch of machine learning that involves training a model to produce new data that is similar to a given dataset.
What does this mean in practice? Suppose we have a dataset containing photos of horses. We can train a generative model on this dataset to capture the rules that govern the complex relationships between pixels in images of horses. Then we can sample from this model to create novel, realistic images of horses that did not exist in the original dataset.

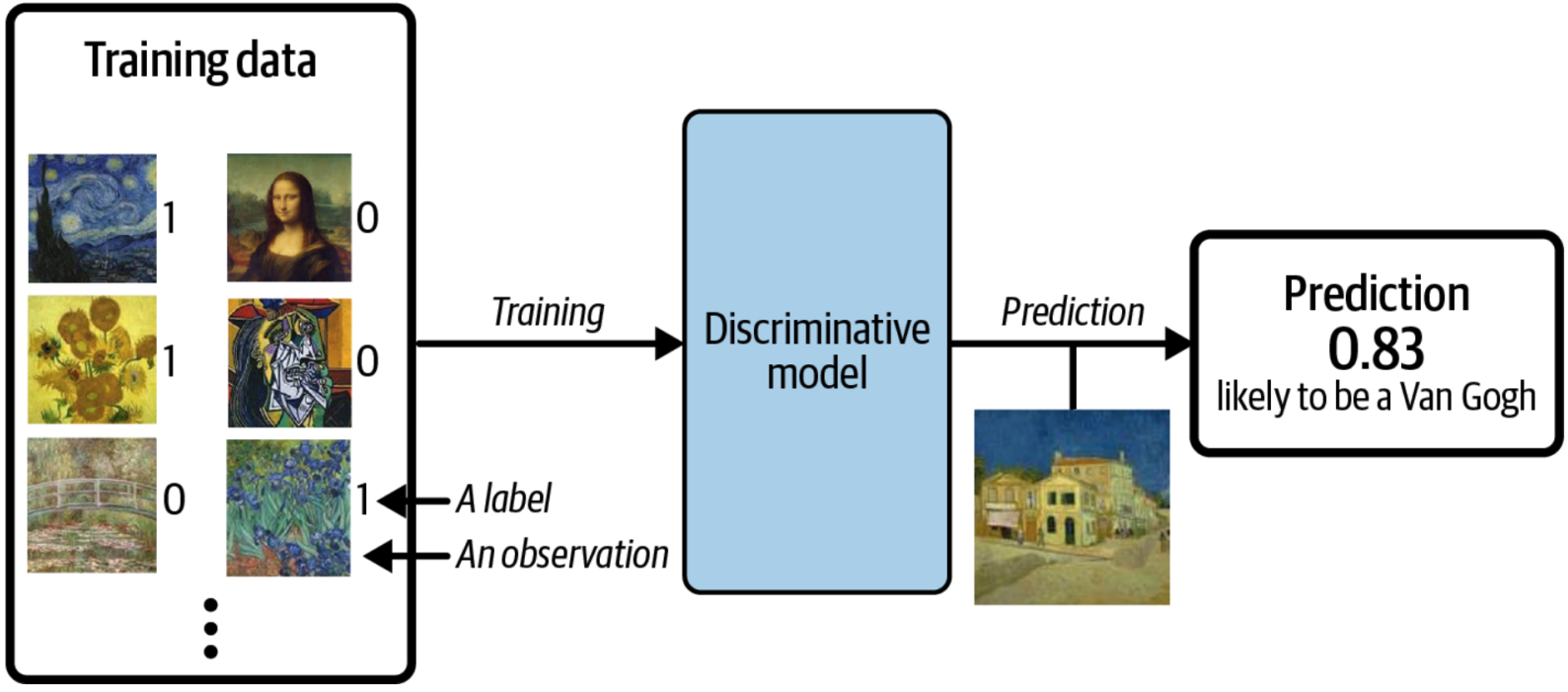
Example of generative modeling, taken from Generative Deep Learning by David Foster



Generative Versus Discriminative Modeling

In order to truly understand what generative modeling aims to achieve and why this is important, it is useful to compare it to its counterpart, discriminative modeling. If you have studied machine learning, most problems you will have faced will have most likely been discriminative in nature.

Example of discriminative modeling, taken from Generative Deeep Learning by David Foster



Discriminative Modeling

When performing discriminative modeling, each observation in the training data has a label. For a binary classification problem such as our data could be labeled as ones and zeros. Our model then learns how to discriminate between these two groups and outputs the probability that a new observation has label 1 or 0
In contrast, generative modeling doesn't require the dataset to be labeled because it concerns itself with generating entirely new data (for example an image), rather than trying to predict a label for say a given image.

A Frequentist approach to data analysis

When you hear phrases like **predictions and estimations** and **correlations and causations**, what do you think of? May be you think of the difference between classifying new data points and generating new data points. Or perhaps you consider that correlations represent some kind of symmetric statements like if A is correlated with B, then B is correlated with A. Causation on the other hand is directional, that is if A causes B, B does not necessarily cause A.

These concepts are in some sense the difference between machine learning and statistics. In machine learning and prediction based tasks, we are often interested in developing algorithms that are capable of learning patterns from given data in an automated fashion, and then using these learned patterns to make predictions or assessments of newly given data. In many cases, our primary concern is the quality of the predictions or assessments, and we are less concerned about the underlying patterns that were learned in order to make these predictions.

In machine learning we normally use a so-called frequentist approach, where the aim is to make predictions and find

What is a good model?

In science and engineering we often end up in situations where we want to infer (or learn) a quantitative model M for a given set of sample points $\boldsymbol{X} \in [x_1, x_2, \ldots x_N]$.

As we will see repeatedely in these lectures, we could try to fit these data points to a model given by a straight line, or if we wish to be more sophisticated to a more complex function.

The reason for inferring such a model is that it serves many useful purposes. On the one hand, the model can reveal information encoded in the data or underlying mechanisms from which the data were generated. For instance, we could discover important corelations that relate interesting physics interpretations.

In addition, it can simplify the representation of the given data set and help us in making predictions about future data samples.

A first important consideration to keep in mind is that inferring the *correct* model for a given data set is an elusive, if not impossible, task. The fundamental difficulty is that if we are not specific about what we mean by a *correct* model, there could easily be many different models that fit the given data set *equally well*.

What is a good model? Can we define it?

The central question is this: what leads us to say that a model is correct or optimal for a given data set? To make the model inference problem well posed, i.e., to guarantee that there is a unique optimal model for the given data, we need to impose additional assumptions or restrictions on the class of models considered. To this end, we should not be looking for just any model that can describe the data. Instead, we should look for a **model** M that is the best among a restricted class of models. In addition, to make the model inference problem computationally tractable, we need to specify how restricted the class of models needs to be. A common strategy is to start with the simplest possible class of models that is just necessary to describe the data or solve the problem at hand. More precisely, the model class should be rich enough to contain at least one model that can fit the data to a desired accuracy and yet be restricted enough that it is relatively simple to find the best model for the given data. Thus, the most popular strategy is to start from the simplest class of models and increase the complexity of the models only when the simpler models become inadequate. For instance, if we work with a

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

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

Installing R, C++, cython or Julia

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

Installing R, C++, cython, Numba etc

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

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

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

And to add more versatility, the Python package SymPy is a Python library for symbolic mathematics. It aims to become a full-featured

Numpy examples and Important Matrix and vector handling packages

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

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

Numpy and arrays

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

```
import numpy as np
```

Here follows a simple example where we set up an array of ten elements, all determined by random numbers drawn according to the normal distribution,

```
n = 10
x = np.random.normal(size=n)
print(x)
```

We defined a vector x with $n = 10$ elements with its values given by the Normal distribution $N(0, 1)$. Another alternative is to declare a vector as follows

```
import numpy as np
x = np.array([1, 2, 3])
print(x)
```

Here we have defined a vector with three elements, with $x_0 = 1$, $x_1 = 2$ and $x_2 = 3$. Note that both Python and C++ start numbering array elements from 0 and on. This means that a vector with n elements has a sequence of entities $x_0, x_1, x_2, \ldots, x_{n-1}$. We

Matrices in Python

Having defined vectors, we are now ready to try out matrices. We can define a 3×3 real matrix $\boldsymbol{A}$ 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
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
# 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
# print the first column, row-major order and elements start with 0
print(A[1,:])
```

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

Pandas AI

Try out Pandas AI

A first summary

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 $\boldsymbol{x}$ 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

Why Linear Regression (aka Ordinary Least Squares and family)

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

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

Regression analysis, overarching aims

Regression modeling deals with the description of the sampling distribution of a given random variable y and how it varies as function of another variable or a set of such variables $\boldsymbol{x} = [x_0, x_1, \dots, x_{n-1}]^T$. The first variable is called the **dependent**, the **outcome** or the **response** variable while the set of variables $\boldsymbol{x}$ is called the independent variable, or the predictor variable or the explanatory variable, or simply just the **inputs**.
A regression model aims at finding a likelihood function $p(\boldsymbol{y}|\boldsymbol{x})$ or in the more traditional sense a function $\boldsymbol{y}(\boldsymbol{x})$, that is the conditional distribution for $\boldsymbol{y}$ with a given $\boldsymbol{x}$. The estimation of $p(\boldsymbol{y}|\boldsymbol{x})$ is made using a data set with

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

Regression analysis, overarching aims II

Consider an experiment in which p characteristics/features of n samples are measured. The data from this experiment, for various explanatory variables p are normally represented by a matrix X. The matrix X is called the *design matrix*. Additional information of the samples is available in the form of $\boldsymbol{y}$ (also as above). The variable $\boldsymbol{y}$ is generally referred to as the *response variable*. The aim of regression analysis is to explain $\boldsymbol{y}$ in terms of $\boldsymbol{X}$ through a functional relationship like $y_i = f(\mathrm{X}_{i,*})$. When no prior knowledge on the form of $f(\cdot)$ is available, it is common to assume a linear relationship between $\boldsymbol{X}$ and $\boldsymbol{y}$. This assumption gives rise to the *linear regression model* where $\boldsymbol{\beta} = [\beta_0, \ldots, \beta_{p-1}]^T$ are the *regression parameters*.

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

Examples

In order to understand the relation among the predictors (or features or properties) p, the set of data n and the target (outcome, output etc) $\boldsymbol{y}$, consider the model we discussed for describing nuclear binding energies.
There we assumed that we could parametrize the data using a polynomial approximation based on the liquid drop model.
Assuming

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

we have five predictors, that is the intercept, the A dependent term, the $A^{2/3}$ term and the $A^{-1/3}$ and A^{-1} terms. This gives $p = 0, 1, 2, 3, 4$. Furthermore we have n entries for each predictor. It means that our design matrix is a $p \times n$ matrix $\boldsymbol{X}$.
Here the predictors are based on a model we have made. A popular data set which is widely encountered in ML applications is the so-called credit card default data from Taiwan. The data set contains data on $n = 30000$ credit card holders with predictors like

General linear models and linear algebra

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

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

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

where ϵ_i is the error in our approximation.

Rewriting the fitting procedure as a linear algebra problem

For every set of values y_i, x_i we have thus the corresponding set of equations

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

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

Defining the vectors

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

and

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

and

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

and the design matrix

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

we can rewrite our equations as

Generalizing the fitting procedure as a linear algebra problem

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

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

Note that we have $p = n$ here. The matrix is symmetric.

Generalizing the fitting procedure as a linear algebra problem

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

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

and without loss of generality we rewrite again our equations as

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

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

Optimizing our parameters

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

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

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

Our model for the nuclear binding energies

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

We restate the parts of the code we are most interested in.

```
# Common imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
import os

# Where to save the figures and data files
PROJECT_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):
```

Optimizing our parameters, more details

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

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

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

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

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

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

Interpretations and optimizing our parameters

The function

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

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

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

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

Interpretations and optimizing our parameters

We can rewrite

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

as

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

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

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

We note also that since our design matrix is defined as $\boldsymbol{X} \in \mathbb{R}^{n \times p}$, the product $\boldsymbol{X}^T\boldsymbol{X} \in \mathbb{R}^{p \times p}$. In the above case we have that $p \ll n$, in our case $p = 5$ meaning that we end up with inverting a small 5×5 matrix. This is a rather common situation, in many cases we end up with low-dimensional matrices to invert. The methods discussed here and for many other supervised learning algorithms like classification with logistic regression or support vector

Interpretations and optimizing our parameters

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

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

and with

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

we have

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

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

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

Own code for Ordinary Least Squares

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

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

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

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

And finally we plot our fit with and compare with data

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

Adding error analysis and training set up

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

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

and we would be using it as

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

We can easily add our **MSE** score as

```
def MSE(y_data,y_model):
    n = np.size(y_model)
    return np.sum((y_data-y_model)**2)/n

print(MSE(Energies,ytilde))
```

and finally the relative error as

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

The χ^2 function

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

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

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

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

The χ^2 function

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

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

which results in

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

or in a matrix-vector form as

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

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

The χ^2 function

We can rewrite

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

as

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

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

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

The χ^2 function

If we then introduce the matrix

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

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

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

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

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

resulting in

The χ^2 function

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

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

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

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

and

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

The χ^2 function

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

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

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

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

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

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

Fitting an Equation of State for Dense Nuclear Matter

Before we continue, let us introduce yet another example. We are going to fit the nuclear equation of state using results from many-body calculations. The equation of state we have made available here, as function of density, has been derived using modern nucleon-nucleon potentials with the addition of three-body forces. This time the file is presented as a standard **csv** file.
The beginning of the Python code here is similar to what you have seen before, with the same initializations and declarations. We use also **pandas** again, rather extensively in order to organize our data. The difference now is that we use **Scikit-Learn's** regression tools instead of our own matrix inversion implementation. Furthermore, we sneak in **Ridge** regression (to be discussed below) which includes a hyperparameter λ, also to be explained below.

The code

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

# 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)
```

Splitting our Data in Training and Test data

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

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

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