exercise 1

getting used to the method of least squares

solutions due

until **November 3, 2023** at **23:59** via **ecampus**

general remarks

Your instructor is an avid proponent of open science and open education and therefore favors working with of open source software.

The coding exercises which accompany this course are therefore to be solved using **python** / **numpy** / **scipy** / **sympy** / **matplotlib**. If you are not yet familiar with these popular components of the data science stack, this course provides an opportunity to gather corresponding experience because **implementations in other languages will not be accepted**.

There are numerous Web resources related to python programming; numpy and scipy are well documented and matplotlib, too, comes with numerous tutorials. The practical problems on this and future exercise sheets are all in all rather simple; if you do not have any ideas for how to solve them, just look around for clues or ideas for how to tackle them. In fact, you may ask modern AIs such as ChatGPT or BARD for help. If you do, please report your experiences.

Remember that you have to achieve at least 50% of the points of the exercises to be eligible to the written exam at the end of the semester.

Your grades (and credits) for this course will be decided on the exam, but —once again— you have to succeed in the exercises to get there.

All your solutions have to be *satisfactory* to count as a success. Your code and results will be checked and need to be convincing.

If your solutions meet the above requirements and you can demonstrate that they work in practice, they are *satisfactory* solutions.

A *very good* solution (one that is rewarded full points) requires additional efforts especially w.r.t. to readability of your code. If your code is neither commented nor well structured, your solution is not good! The same holds for your discussion of your results: these should be concise and convincing and demonstrate that you understood what the respective task was all about. Striving for very good solutions should always be your goal!

task 1.1

beware of numerical instabilities

⚠

Note: Every student in every team should do this task because everybody needs to know about potential pitfalls when using digital computers to get numerical solutions to computational problems!

Note: There are no points for this task, because we will closely guide you through it. Still, we urge you to work through this task since it reveals that mathematical theory and digital computing practice are not always well aligned. This is of course good to know if we develop machine learning solution for real world applications . . .

Consider the following least squares problem

$$\boldsymbol{w}_\star = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left\lVert \boldsymbol{X}^\intercal \boldsymbol{w} - \boldsymbol{y} \right\rVert^2 \tag{1}$$

where matrix $\boldsymbol{X}^\intercal$ and vector $\boldsymbol{y}$ are given by

$$\boldsymbol{X}^\intercal = \begin{bmatrix} 1.00000 & -1.00000 \\ 0.00000 & 0.00001 \\ 0.00000 & 0.00000 \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{y} = \begin{bmatrix} 0.00000 \\ 0.00001 \\ 0.00000 \end{bmatrix} \tag{2}$$

This particular problem is perfectly solved by

$$\boldsymbol{w}_\star = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

because $\boldsymbol{X}^\intercal \boldsymbol{w}_\star = \boldsymbol{y}$ and therefore $\left\lVert \boldsymbol{X}^\intercal \boldsymbol{w}_\star - \boldsymbol{y} \right\rVert^2 = \mathbf{0}$ which is the smallest possible value our (non-negative quadratic) objective function can have.

Solving this problem is easy because it is small enough for us to "see" the solution. In practice, problems rarely are this simple. However, we know that we can compute the solution using

$$\boldsymbol{w}_\star = \left[\boldsymbol{X}\boldsymbol{X}^\intercal\right]^{-1} \boldsymbol{X}\boldsymbol{y} \tag{3}$$

In other words, we know that we can solve least squares problems for any $\boldsymbol{X}^\intercal$ and $\boldsymbol{y}$ (of matching dimensions).

So what happens if we implement the algebraic solution in (3) to use a computer to solve the problem in (1) with the ingredients in (2)?

task 1.1.1

Run the following code snippet and marvel at the output it prodcues

```
import numpy as np
import numpy.linalg as la

matX = np.array([[ 1.00000, 0.00000, 0.00000],
                 [-1.00000, 0.00001, 0.00000]])
vecY = np.array( [ 0.00000, 0.00001, 0.00000] )

vecW = la.inv(matX @ matX.T) @ matX @ vecY
print (vecW)
```

background info

A matrix is called “tall” or “thin” if it has more rows than columns. Matrix $\boldsymbol{X}^\intercal \in \mathbb{R}^{r \times c}$ in (2) is such a matrix and therefore has a QR decomposition

$$\boldsymbol{X}^\intercal = \boldsymbol{Q}\boldsymbol{R}$$

where

$$\begin{aligned} \boldsymbol{Q} &\in \mathbb{R}^{r \times r} && \text{is orthogonal} \\ \boldsymbol{R} &\in \mathbb{R}^{r \times c} && \text{is upper triangular} \end{aligned}$$

Since the $r - c$ bottom rows of an upper triangular matrix only contain 0s, we may partition the product $\boldsymbol{Q}\boldsymbol{R}$ as follows

$$\boldsymbol{Q}\boldsymbol{R} = \begin{bmatrix} \boldsymbol{Q}_1 & \boldsymbol{Q}_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{R}_1 \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{Q}_1 \boldsymbol{R}_1$$

The columns of $\boldsymbol{Q}_1 \in \mathbb{R}^{r \times c}$ are still orthogonal and $\boldsymbol{R}_1 \in \mathbb{R}^{c \times c}$ is still upper triangular.

Note: numpy’s `linalg` module provides a function `qr` whose default is to return $\boldsymbol{Q}_1$ and $\boldsymbol{R}_1$ rather than $\boldsymbol{Q}$ and $\boldsymbol{R}$.

Now, observe the following: if $\boldsymbol{X}^\intercal = \boldsymbol{Q}_1 \boldsymbol{R}_1$ and the columns of $\boldsymbol{Q}_1$ are orthogonal, then

$$\begin{aligned} \boldsymbol{w}_\star = \left[\boldsymbol{X}\boldsymbol{X}^\intercal\right]^{-1} \boldsymbol{X}\boldsymbol{y} &= \left[\boldsymbol{R}_1^\intercal \boldsymbol{Q}_1^\intercal \boldsymbol{Q}_1 \boldsymbol{R}_1\right]^{-1} \boldsymbol{R}_1^\intercal \boldsymbol{Q}_1^\intercal \boldsymbol{y} \\ &= \left[\boldsymbol{R}_1^\intercal \boldsymbol{R}_1\right]^{-1} \boldsymbol{R}_1^\intercal \boldsymbol{Q}_1^\intercal \boldsymbol{y} \\ &= \boldsymbol{R}_1^{-1} \boldsymbol{R}_1^{-\intercal} \boldsymbol{R}_1^\intercal \boldsymbol{Q}_1^\intercal \boldsymbol{y} \\ &= \boldsymbol{R}_1^{-1} \boldsymbol{Q}_1^\intercal \boldsymbol{y} \end{aligned}$$

task 1.1.2

Implement numpy code that computes $\boldsymbol{w}_\star = \boldsymbol{R}_1^{-1}\boldsymbol{Q}_1^\intercal \boldsymbol{y}$ and have a look at the result.

task 1.1.3

Implement numpy code that uses the *`linalg`* function *`lstsq`* to compute $\boldsymbol{w}_\star$ and have a look at the result. What do you think? Does *`lstsq`* solve least square problems via equation (3) or not?

task 1.1.4

Think about what all this means. Here are a few questions to guide your thoughts:

- are real numbers and floating point numbers the same thing?
- what does it mean to speak about machine precision w.r.t. floating point arithmetic?
- what do we have to keep in mind when working with very large and very small floating point numbers?
- what do you find when you use pen and paper to compute the entries of $\boldsymbol{X}\boldsymbol{X}^\intercal$?
- what do you find when you use your computer to compute the entries of $\boldsymbol{X}\boldsymbol{X}^\intercal$?
- can you blindly trust any piece of numerical software that some dude from somewhere has uploaded to github or even to huggingface?

task 1.2 [20 points]

cellular automata, the Boolean Fourier transform, and LSQ

In the lectures, we encountered the idea of feature maps which transform lower dimensional data to higher dimensional representations. Here, we explore this idea in a somewhat unusual context . . .

background info

A one-dimensional (or elemental) cellular automaton consists of

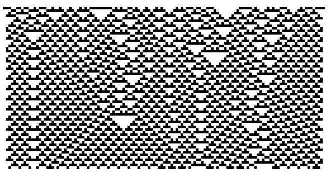
- an infinite sequence of cells $\{x_j\}_{j \in \mathbb{Z}}$ each of which is in either one of two states, namely *off* or *on* which are usually encoded as $x_j \in \{0, 1\}$
- an update rule $f : \{0,1\}^3 \to \{0,1\}$ for how to update a cell based on its and its two neighbors' current states

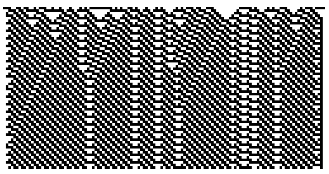
$$x_j[t+1] = f\Big(x_{j-1}[t], x_j[t], x_{j+1}[t]\Big)$$

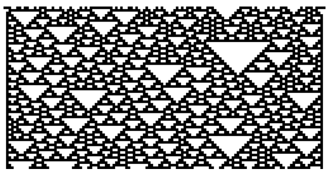
At time $t = 0$, the x_j are (randomly) initialized, and, at any (discrete) time step t, all cells are updated simultaneously. Here is an illustrative example where we plot 0 as □ and 1 as ■

t ···□□□□□□□□□□□□□□□□□■□□□□□□□□□□□□□□□□□···
$t+1$ ···□□□□□□□□□□□□□□□□■□■□□□□□□□□□□□□□□□□···

This update process continues forever and it is common to plot subsequent (finitely sized) state sequences below each other to visualize the evolution of an automaton under a certain rule. Here are some examples of possible evolutions all starting from the same initial configuration


rule 108


rule 110


rule 126

Henceforth, we will reduce notational clutter. That is, instead of writing

$$x_j[t+1] = f\Big(x_{j-1}[t], x_j[t], x_{j+1}[t]\Big)$$

we henceforth simply write

$$y = f(\boldsymbol{x})$$

where $\boldsymbol{x} \in \{0,1\}^3$ and $y \in \{0,1\}$.

The naming convention for the rules which govern the evolution of elemental cellular automata is due to **Stephen Wolfram** and illustrated in the following tables

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

rule 110

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

rule 126

For $\boldsymbol{x} \in \{0,1\}^3$, there are $2^3 = 8$ possible input patterns for each rule. For each input pattern, there are 2 possible outputs or target values. Hence, the total number of possible rules is $2^{2^3} = 256$.

We may collect the possible input patterns in an 8×3 matrix $\boldsymbol{X}^\intercal$ and the target values of each rule in an 8 dimensional target vector $\boldsymbol{y}$. Converting a rule's binary target vector (read from bottom to top) into a decimal number gives the rule's name.

Now, consider a seemingly crazy idea, namely the following substitution

$$\begin{aligned} 0 &\rightarrow +1 \\ 1 &\rightarrow -1 \end{aligned}$$

People like this representation of binary states because there is a simple mapping from the ordered set $\{0,1\}$ to the ordered set $\{+1,-1\}$, namely

$$x \in \{0,1\} \quad \mapsto \quad x' = (-1)^x \in \{+1,-1\}$$

We can therefore convert binary functions $f_{01} : \{0,1\}^n \to \{0,1\}$ into bipolar functions $f_{+-} : \{+1,-1\}^n \to \{+1,-1\}$ by letting

$$f_{+-}\big((-1)^{x_1}, \ldots, (-1)^{x_n}\big) = (-1)^{f_{01}(x_1,\ldots,x_n)}$$

Seen from this point of view, the above two tables representing rule 110 and rule 126 will look like this

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	-1	-1
$+1$	-1	$+1$	-1
$+1$	-1	-1	-1
-1	$+1$	$+1$	$+1$
-1	$+1$	-1	-1
-1	-1	$+1$	-1
-1	-1	-1	$+1$

rule 110

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	-1	-1
$+1$	-1	$+1$	-1
$+1$	-1	-1	-1
-1	$+1$	$+1$	-1
-1	$+1$	-1	-1
-1	-1	$+1$	-1
-1	-1	-1	$+1$

rule 126

task 1.2.1 [5 points]

Given a matrix $\boldsymbol{X}^\intercal$ and a vector $\boldsymbol{y}$ as in the two tables above, implement python / numpy code that first solves the least squares problem

$$\boldsymbol{w}_\star = \underset{\boldsymbol{w} \in \mathbb{R}^3}{\operatorname{argmin}} \left\| \boldsymbol{X}^\intercal \boldsymbol{w} - \boldsymbol{y} \right\|^2$$

and then computes

$$\hat{\boldsymbol{y}} = \boldsymbol{X}^\intercal \boldsymbol{w}_\star$$

Run your code for rules 110 and 126 and print the respective vectors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. Discuss what you observe.

Note: Don't be alarmed if your results seem less than optimal! What you are supposed to do here hardly makes any sense and is mainly intended as a preparation for what comes next. (Can you see *why* what you are supposed to do here hardly makes sense?)

more background info

A Boolean domain $\mathbb{B}$ is a set containing exactly two elements. The most common examples of such domains are $\mathbb{B} = \{0, 1\}$ and $\mathbb{B} = \{-1, +1\}$.

Given this definition, it isn't a surprise that functions of the following form

$$f : \mathbb{B}^n \to \mathbb{B}$$

are called Boolean functions.

Moreover, functions of the form

$$f : \mathbb{B}^n \to \mathbb{R}$$

are called pseudo Boolean functions and we note that, for $\mathbb{B} \subset \mathbb{R}$, the set of pseudo Boolean functions includes the set of Boolean functions.

Above, we already saw that each possible rule $y = f(\boldsymbol{x})$ which may govern the behavior of an elemental cellular automaton can be expressed as a (pseudo) Boolean function

$$f : \{\pm 1\}^n \to \{\pm 1\} \tag{4}$$

where $n = 3$. In what follows, we let $\mathcal{I}$ denote the index set of the entries x_j of $\boldsymbol{x} \in \mathbb{B}^n$, that is

$$\mathcal{I} = \bigl\{1, 2, \ldots, n\bigr\}.$$

The power set $2^{\mathcal{I}}$ of $\mathcal{I}$ is the set of all subsets of $\mathcal{I}$. In other words,

$$2^{\mathcal{I}} = \bigl\{\emptyset, \{1\}, \{2\}, \ldots, \{1, 2\}, \ldots, \{1, 2, \ldots, n\}\bigr\}$$

and we note that $\bigl|2^{\mathcal{I}}\bigr| = 2^n$.

Given these prerequisites, here is why Boolean functions as in (4) are interesting: “One can show” that every pseudo Boolean function

$$f : \{\pm 1\}^n \to \mathbb{R}$$

(and therefore every Boolean function of the form in (4)) can be uniquely expressed as a multilinear polynomial

$$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{\mathcal{I}}} w_{\mathcal{S}} \prod_{j \in \mathcal{S}} x_j \tag{5}$$

The expression in (5) is known as the Boolean Fourier series expansion of the function f. In contrast to the complex exponentials which form the basis functions for conventional Fourier analysis, here the basis functions are the parity functions

$$\varphi_{\mathcal{S}}(\boldsymbol{x}) = \prod_{j \in \mathcal{S}} x_j$$

where by definition

$$\varphi_{\emptyset}(\boldsymbol{x}) = \prod_{j \in \emptyset} x_j = 1$$

Given these definitions of the $\varphi_{\mathcal{S}}(\boldsymbol{x})$, we can rewrite equation (5) as follows

$$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{\mathcal{I}}} w_{\mathcal{S}}\, \varphi_{\mathcal{S}}(\boldsymbol{x}) = \boldsymbol{w}^{\intercal} \boldsymbol{\varphi}(\boldsymbol{x}) \tag{6}$$

This is interesting, because we recognize that any (pseudo) Boolean function of the kind we are currently dealing with is an inner product between two high-dimensional vectors, namely

$$\boldsymbol{w} \in \mathbb{R}^{2^n}$$

and

$$\boldsymbol{\varphi}(\boldsymbol{x}) \in \{+1, -1\}^{2^n}$$

task 1.2.2 [5 points]

Implement a python / numpy function that realizes the transformation

$$\boldsymbol{\varphi} : \{+1, -1\}^n \rightarrow \{+1, -1\}^{2^n}$$

which we implicitly introduced above.

To be specific, for an input vector $\boldsymbol{x} = [x_1, x_2, x_3]^\intercal \in \{+1, -1\}^3$, your code should produce the output vector

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_1\, x_2 \\ x_1\, x_3 \\ x_2\, x_3 \\ x_1\, x_2\, x_3 \end{bmatrix}$$

However, try to implement you function in a more general manner. That is, implement it such that it works for arbitrary $n \in \mathbb{N}$ rather than just for $n = 3$.

Tip: The python standard library contains the module `itertools` which, in turn, provides functionalities that may come in handy for this task.

task 1.2.3 [10 points]

If we finally reconsider the matrices $\boldsymbol{X}^\intercal$ in the two tables you used above, we may think of them as row matrices

$$\boldsymbol{X}^\intercal = \begin{bmatrix} — \ \boldsymbol{x}_0^\intercal \ — \\ — \ \boldsymbol{x}_1^\intercal \ — \\ \vdots \\ — \ \boldsymbol{x}_7^\intercal \ — \end{bmatrix}$$

where

$$\boldsymbol{x}_0 = \begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix} \qquad \boldsymbol{x}_1 = \begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix} \qquad \ldots \qquad \boldsymbol{x}_7 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

Given your code from task 1.2.2, you should thus be able to compute a row matrix

$$\boldsymbol{\Phi}^\intercal = \begin{bmatrix} — \boldsymbol{\varphi}_0^\intercal — \\ — \boldsymbol{\varphi}_1^\intercal — \\ \vdots \\ — \boldsymbol{\varphi}_7^\intercal — \end{bmatrix}$$

where

$$\boldsymbol{\varphi}_j = \boldsymbol{\varphi}\big(\boldsymbol{x}_j\big)$$

Given matrix $\boldsymbol{\Phi}^\intercal$ and the target vector $\boldsymbol{y}$ of a cellular automaton rule, write python / numpy code that first solves the least squares problem

$$\boldsymbol{w}_\star = \underset{\boldsymbol{w}\in\mathbb{R}^8}{\operatorname{argmin}} \left\|\boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y}\right\|^2$$

and then computes

$$\hat{\boldsymbol{y}} = \boldsymbol{\Phi}^\intercal \boldsymbol{w}_\star$$

Run your code for rules 110 and 126 and print the respective vectors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. What do you observe in comparison to your results in task 1.2.1?

What is the “price” you had to pay to obtain these (hopefully) more reasonable results? Discuss this in your own words.

task 1.3 [10 points]

estimating the fractal dimension of objects in pictures

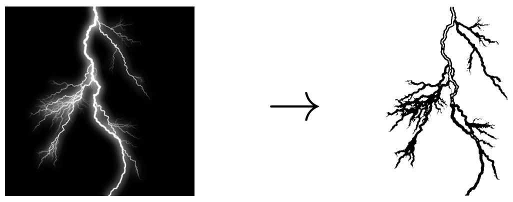
In the lectures, we discussed the use of least squares for linear regression. In this task, we consider a neat practical application of this technique.

background info

Box counting is a method to determine the fractal dimension of an object in an image. For simplicity, we focus on square images whose width w and height h (in pixels) are integer powers of 2. For instance, if $w = h = 512$, then $w = h = 2^L$ where $L = 9$.

Given an image like this, the box counting procedure involves three steps:

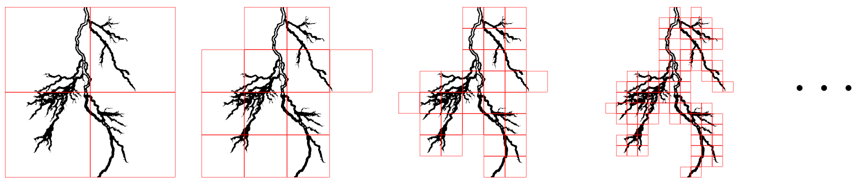
1. apply an appropriate binarization procedure to create a binary image in which foreground pixels are set to 1 and background pixels to 0



2. specify a set S of scaling factors $0 < s_l < 1$, for instance

$$S = \left\{ s_l = \tfrac{1}{2^l} \;\middle|\; l \in \{1, 2, \ldots, L-2\} \right\}$$

and, for each $s_l \in S$, cover the binarized image with boxes of size $s_l\, w \times s_l\, h$ and count the number n_l of boxes which contain at least one foreground pixel

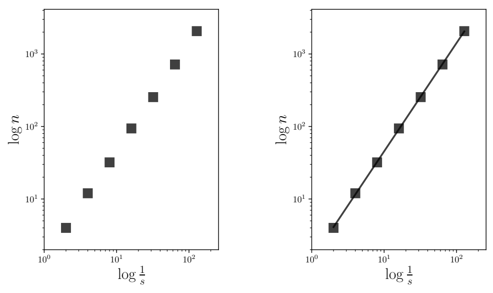


$s_l = \frac{1}{2}, n_l = 4$ $s_l = \frac{1}{4}, n_l = 12$ $s_l = \frac{1}{8}, n_l = 32$ $s_l = \frac{1}{16}, n_l = 94$

3. plot $\log n_l$ against $\log \frac{1}{s_l}$ and fit a line

$$D \cdot \log \frac{1}{s_l} + b = \log n_l$$

   the resulting estimate for the slope D of this line represents the fractal dimension we are after.



In other words, the problem of estimating D is a simple linear regression problem that can of course be tackled using least squares.

task [20 points]

Implement python / numpy / matplotlib code that realizes the box counting method using and run it on the images `tree.png` and `lightning.png`.

What fractal dimensions do you obtain? Which object has the higher one, the tree or the lightning bolt?

Tip: we strongly suggest to work with the following imports

```
import numpy as np
import imageio.v3 as iio
import numpy.linalg as la
import scipy.ndimage as img
```

Tip: to read, say, image `tree.png` into a numpy array, you can then use

```
imgF = iio.imread('tree.png', mode='L').astype(np.float)
```

Note: while we do not really care about how you realize image I/O, we do care about how you binarize the images you read.

This is because after binarization, the outcome of the box counting procedure should be deterministic. I.e. if every team uses the same binarization procedure, every team should obtain the same results and these results should be the same as those your instructors got. Teams getting different results can rest assured they made a mistake.

Therefore, please use the following snippet

```
def binarize(imgF):
    imgD = np.abs(img.filters.gaussian_filter(imgF, sigma=0.50) - \
                  img.filters.gaussian_filter(imgF, sigma=1.00))

    return img.morphology.binary_closing(np.where(imgD < 0.1*imgD.max(), 0, 1)
```

task 1.4

submission of presentation and code

Prepare a presentation / set of slides on your solutions and results for tasks 1.2 and 1.3. These slides should help you to give a scientific presentation of your work (i.e. to give a short talk in front of your fellow students and instructors and answer any questions they may have).

W.r.t. to formalities, please make sure that

- your presentation contains a title slide which lists the names and matriculation numbers of everybody in your team who contributed to the solutions.

W.r.t. content, please make sure that

- your presentation contains about 12 to 15 content slides but not more
- your presentation is concise and clearly structured
- your presentation answers questions such as
  - "what was the task / problem we considered?"
  - "what kind of difficulties (if any) did we encounter?"
  - "how did we solve them?"
  - "what were our results?"
  - "what did we learn?"

Save / export your slides as a PDF file and upload it to eCampus.

Furthermore, please name all your code files in a manner that indicates which task they solve (e.g. `task-1-3.py`) and put them in an archive or a ZIP file.

Upload this archive / ZIP file to eCampus.