project 2:

least squares regression and nearest neighbor classifiers

solution(s) due:

Dec 21, 2017 at **12:00** via email to **bauckhag@bit.uni-bonn.de**

problem specification:

task 2.1: least squares regression for missing value prediction
Download the file `whData.dat`, remove the outliers (as in the previous project) and collect the remaining height and weight data in two vectors

$$\boldsymbol{x} = [x_1\ x_2\ \ldots\ x_n]^T \quad \text{and} \quad \boldsymbol{y} = [y_1\ y_2\ \ldots\ y_n]^T,$$

respectively.

Use the method of least squares to fit polynomial models

$$y(x) = \sum_{j=0}^{d} w_j x^j$$

to the data. In particular, fit models for $d \in \{1, 5, 10\}$ and plot your results. Use each of your resulting models to predict weight values for the outliers (i.e. the data points in `whData.dat` where the weight value is -1).

note:

There are numerous ways of how this can be done. However, depending on how you implement your solution, you may run into numerical problems (you should recognize them when they occur). In this case, it is not acceptable to give up and simply claim that the method did not work. Rather, you should either double your efforts and try to come up with an implementation that works or provide an in-depth analysis as to why and where your code failed and point out possible solutions.

task 2.2: conditional expectation for missing value prediction
Fit a bi-variate Gaussian to the height and weight data in $\boldsymbol{x}$ and $\boldsymbol{y}$ to model the joint density $p(x, y)$ of heights and weights.

Given your fitted bi-variate Gaussian, use the idea of conditional expectation to predict the weight values for the outliers. That is, let x_o denote the available height data of an outlier and compute

$$\mathbb{E}\big[y \mid x_o\big] = \int y\, p\big(y \mid x_o\big)\, dy$$

Do this either analytically as discussed in the lecture or numerically and report your results.

task 2.3: Bayesian regression for missing value prediction
Use the method of Bayesian regression to fit a fifth degree polynomial

$$y(x) = \sum_{j=0}^{5} w_j x^j$$

to the height and weight data in $\boldsymbol{x}$ and $\boldsymbol{y}$. Assume a Gaussian prior

$$p(\boldsymbol{w}) \sim \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}_0, \sigma_0^2\, \boldsymbol{I})$$

for the parameter vector $\boldsymbol{w}$ where $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\sigma_0^2 = 3$. Plot you resulting model and compare it to the corresponding model ($d = 5$) from task 2.1

task 2.4: Boolean functions and the Boolean Fourier transform
In the supplementary material for lecture 07, we already briefly discussed elemental cellular automata.

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

- an infinite sequence of cells $\{x_i\}_{i\in\mathbb{N}}$ in two possible states; typically these states are assumed to be 0 or 1, here, however, we consider

$$x_i \in \{+1, -1\}$$

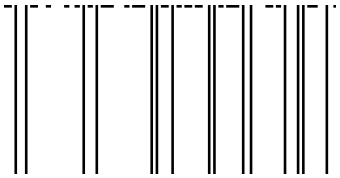
- a update rule for how to update a cell based on its and its two neighbors' current states

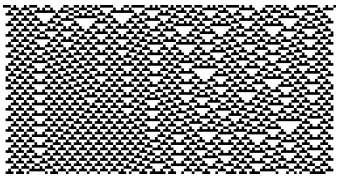
$$x_i(t+1) = f\big(x_{i-1}(t), x_i(t), x_{i+1}(t)\big)$$

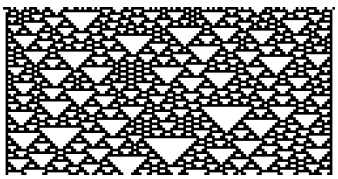
At time $t = 0$, the x_i are (randomly) initialized, and, at any (discrete) time step t, all cells are updated simultaneously. Here is an illustrative example where we use $+1 = \blacksquare$ and $-1 = \square$

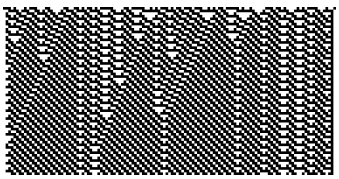
t $\cdots\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\blacksquare\square\square\square\square\square\square\square\square\square\square\square\square\square\cdots$
$t+1$ $\cdots\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\blacksquare\square\blacksquare\square\square\square\square\square\square\square\square\square\square\square\square\cdots$

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


rule 32


rule 108


rule 126


rule 110

From now on, we remove notational clutter. That is, instead of

$$x_i(t+1) = f\big(x_{i-1}(t), x_i(t), x_{i+1}(t)\big)$$

we simply write

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

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

The naming convention for the rules of elemental cellular automata is due to Wolfram (1984) and illustrated in the following tables.

$\boldsymbol{X}$			$\boldsymbol{y}$	$(\boldsymbol{y}+\mathbf{1})/2$
$+1$	$+1$	$+1$	-1	0
$+1$	$+1$	-1	$+1$	1
$+1$	-1	$+1$	$+1$	1
$+1$	-1	-1	$+1$	1
-1	$+1$	$+1$	-1	0
-1	$+1$	-1	$+1$	1
-1	-1	$+1$	$+1$	1
-1	-1	-1	-1	0

rule 110

$\boldsymbol{X}$			$\boldsymbol{y}$	$(\boldsymbol{y}+\mathbf{1})/2$
$+1$	$+1$	$+1$	-1	0
$+1$	$+1$	-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	-1	-1	0

rule 126

For $\boldsymbol{x} \in \{+1, -1\}^3$, 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 $2^{2^3} = 256$.

Using the language of pattern recognition, we may collect the possible input patterns in an 8×3 design matrix $\boldsymbol{X}$ and the target values of each rule in an 8 dimensional target vector $\boldsymbol{y}$. Converting a rule's target vector from bipolar to binary yields a byte representation of a number between 0 and 255 and provides the rule's name.

Here is your first sub-task: For rules 110 and 126, determine

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \big\| \boldsymbol{X}\boldsymbol{w} - \boldsymbol{y} \big\|^2$$

and then compute

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}^*$$

Now, compare $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. What do you observe?

note:

Don't be alarmed! What we did here hardly makes sense and was mainly intended to prepare ourselves for what comes next.

Each possible rule $y = f(\boldsymbol{x})$ governing the behavior of a cellular automaton is an instance of a *Boolean function*

$$f : \{-1, +1\}^m \to \{-1, +1\} \tag{1}$$

where in our case $m = 3$. In what follows, we let $\mathcal{I}$ denote the index set of the elements x_i of $\boldsymbol{x}$, that is

$$\mathcal{I} = \big\{1, 2, \ldots, m\big\}.$$

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

$$2^{\mathcal{I}} = \big\{\emptyset, \{1\}, \{2\}, \ldots, \{1, 2\}, \ldots, \{1, 2, \ldots, m\}\big\}$$

and we note that $|2^{\mathcal{I}}| = 2^m$.

Every Boolean function f of the form in (1) can be uniquely expressed as a multilinear polynomial

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

which is known as the *Boolean Fourier series expansion* of f. In contrast to the complex exponentials which form the basis functions for conventional Fourier analysis, here the basis functions are the parity functions

$$\varphi_{\mathcal{S}}(\boldsymbol{x}) = \prod_{i \in \mathcal{S}} x_i$$

where by definition

$$\varphi_{\emptyset}(\boldsymbol{x}) = \prod_{i \in \emptyset} x_i = 1.$$

Given these definitions, we can rewrite (2) as follows

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

and recognize that any Boolean function is an inner product between two vectors $\boldsymbol{w} \in \mathbb{R}^{2^m}$ and $\boldsymbol{\varphi} \in \{+1, -1\}^{2^m}$.

Here is your second sub-task: Implement a function

$$\varphi : \{+1, -1\}^m \to \{+1, -1\}^{2^m}$$

such that $\boldsymbol{\varphi} = \varphi(\boldsymbol{x})$. To be specific, for $\boldsymbol{x} = [x_1, x_2, x_3]^T \in \{+1, -1\}^3$, your function should produce

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

However, implement it such that is works for arbitrary $m \in \mathbb{N}$ and not just for $m = 3$. This will come in handy in the next project.

Here is your third sub-task: For rules 110 and 126, turn the design matrix $\boldsymbol{X}$ into a "feature" design matrix $\boldsymbol{\Phi}$, then determine

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left\| \boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{y} \right\|^2$$

and finally compute

$$\hat{\boldsymbol{y}} = \boldsymbol{\Phi}\boldsymbol{w}^*$$

Now, compare $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. What do you observe?

task 2.4: nearest neighbor classifier
Implement a function that realizes an n-nearest neighbor classifier, i.e. a function that decides the class label of a test data point from the majority of the labels among the n nearest training data.

Download the files `data2-train.dat` and `data2-test.dat` of labeled 2D data and determine the recognition accuracy (percentage of correctly classified data points) of your classifier for $n \in \{1, 3, 5\}$.

Determine the overall run time for computing the 1-nearest neighbor of every data in `data2-test.dat`.

task 2.5: computing a kD-tree
Implement a function that computes *and plots* up to four different kinds of kD-trees of the data in `data2-train.dat` where $k = 2$.

Determine overall run time for computing the 1-nearest neighbor of every data in `data2-test.dat` by means of your kD-tree.

note:

Since `scipy` provides functions for kD-tree computation, you may use those for your run time evaluations. However, just using these functions is of course too easy. Therefore, implement your own function for building and plotting a tree (you may ignore the problem of searching the tree). Your function for building the tree should allow for choosing how to determine the dimension for splitting and for choosing how to determine the split point. In particular, you should enable the following two methods for selecting the splitting dimension:

1. alternate between the x and the y dimension
2. split the data along the dimension of higher variance

With respect to computing the split point, you should enable the following to ideas:

1. split at the midpoint of the data
2. split at the median of the data

Given the data in `data2-train.dat` create all four possible kD-trees and plot them. what do you observe?

general hints and remarks

- Send all your solutions (code, plots, slides) in a ZIP archive to

  bauckhag@bit.uni-bonn.de

- If you insist on using a language other than python, you have to figure out elementary functions/toolboxes in these languages by yourself. **Implementations in MATLAB will not be accepted.**

- Remember that you have to complete all practical projects (and the tasks therein) to be eligible to the written exam at the end of the semester. Your grades (and credits) for this course will be decided based on the exam only, but –once again– you have to succeed in the projects to get there.

- Not handing in a solution implies failing the course.

- Your project work needs to be *satisfactory* to count as a success. Your code and results will be checked and your presentation needs to be convincing.

- If your solutions meets the above requirements and you can demonstrate that they work in practice, it is a *satisfactory* solution.

- A *good* to *very good* solution requires additional efforts especially w.r.t. to elegance and readability of your code. If your code is neither commented nor well structured, your solution is not good! A very good solution requires additional efforts towards the quality of your project presentation in the colloquium. Your presentation should be well timed, consistent, and convincing. Striving for very good solutions should always be your goal!