Machine learning estimators: implementation and comparsion in Python

Fabian Merle*

March 14, 2023

Abstract

We compare different *machine learning* estimators, and present details about their implementation in Python. The computational studies are conducted for *classification*, as well as *regression* problems. Moreover, as one of the founding problems of machine learning, we present the specific *classification* task of *handwritten digit recognition* from the scratch. In this connection, we are responsive to the mathematical formulation, and of course to the implementational details of this problem. All corresponding Python codes are fully provided on request and can be downloaded from the author's GitHub page [7][1].

Contents

*Mathematisches Institut, Universität Tübingen, Auf der Morgenstelle 10, D-72076 Tübingen, Germany. email: merle@na.uni-tuebingen.de / private email: fabimer@web.de

[1]In case you are interested, please write me an email. I will then grant you access to the *private* repository in [7], from where the Python codes can be downloaded. Note that you need to have an own GitHub account for this procedure.

1 Introduction and Setting

'Artificial' generation of knowledge based on experience — this is essentially what machine learning (ML) is about. Seen as a part of artificial intelligence (AI), ML provides/builds methods that leverage data to help systems to automatically *learn* and improve. An artificial system which has *learned* through experience is able to make *general* predictions and/or decisions without being explicitly programmed to do so. In this connection, involved ML algorithms build a statistical model which relies on given data — referred to as *training data* — to perform certain tasks, where conventional algorithms are unfeasible or might fail. In doing so, ML algorithms are eager to detect patterns and principles coming from the training data rather than memorizing those ('overfitting'), which, in particular, enables a smart evaluation of unknown (new) data to optimize underlying processes. Due to their huge practical relevance in a broad variety of applications, ML algorithms are indispensable nowadays. Typical examples are *image recognition* (detect an object or feature in a digital image), *speech recognition* ('Alexa' (Amazon) or 'Siri' (Apple)), *prediction of traffic patterns* (identify the fastest route to circumvent traffic jam), *product recommendations* (customer behavior based on past purchases and/or browsing habits is tracked and similar products are recommended to buy), *sentiment analysis* (determine the emotion or opinion of a speaker or writer), *self-driving cars* (cars collect informations from cameras and sensors about its surroundings and automatically choose what actions to perform), *stock market and day trading* (decide when to buy and/or sell stocks), *social media* (automatic friend tagging suggestions), *language translation* (Google translate), *personalized medication* (disease assessment), *genetics and genomics* (identify the impact of heredity on human health), *cancer prognosis and prediction* (identify critical traits in complicated datasets to help to model the evolution and therapy of malignant diseases), *drug discovery/manufacturing* (speed up drug discovery processes), *fraud detection* (spot patterns for unusual behavior, *e.g.* in finance) and many more. All these examples go back to different algorithmic approaches, which are essentially divided into three categories:

- *supervised learning*: algorithms use known data to detect patterns and principles, and to learn a general rule that maps inputs to outputs $\rightarrow$ *e.g. classification* and *regression* problems.
- *unsupervised learning*: (hidden) patterns and principles are detected independently without providing already known data to the algorithms $\rightarrow$ *e.g. clustering*, *density estimation*.
- *reinforcement learning*: algorithms interact with a dynamic environment and learn through rewards $\rightarrow$ *e.g. self-driving cars*.

However, in this manuscript, we mainly focus on *supervised learning* tasks such as classification and regression. The goal here is to give a more mathematical-based introduction of these tasks in combination with fully provided Python codes, which are meant to help the reader to (better) understand the related classification resp. regression estimators presented in Section 2 from a practical viewpoint. Therefore, we encourage the reader to download and test the corresponding Python codes from the author's GitHub page; see [7].
In the following, we formalize the setting and present a motivating example with which we illustrate the main problems and goals of *supervised learning*.

Let $n, d \in \mathbb{N}$. Throughout this manuscript, we assume that we have given 'data' $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\{\mathbf{x}_i\}_{i=1}^n \subset \mathbb{R}^d$ are called points, and $\{y_i\}_{i=1}^n \subset \mathcal{Y}$ are called 'labels' or 'target values'. If $\mathcal{Y}$ is a discrete set, *e.g.* $\mathcal{Y} = \{0,1\}$, we deal with a *classification* problem. On the other hand, if $\mathcal{Y} = \mathbb{R}$, we are concerned with a *regression* problem. Technically speaking, $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ can be considered as single realizations of *i.i.d.* random variables $\{(\mathbf{X}_i, Y_i)\}_{i=1}^n$ on a given probability space $(\Omega, \mathcal{F}, \mathbb{P})$, *i.e.*, $\mathbf{X}_i(\omega) = \mathbf{x}_i$ and $Y_i(\omega) = y_i$ $(i = 1, ..., n)$, where the distribution of $\{(\mathbf{X}_i, Y_i)\}_{i=1}^n$ is in general **unknown**.
A first question, which naturally arises now is '*where to get data from?*'. Essentially, there are two possibilities to get data from:

1) Generate, 'create' or collect data from *e.g.* own experiments.

2) Import (already known) data from a (given) data sheet, file, etc.

Once data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ are available, they are saved in a $\big(n \times (d+1)\big)-$matrix '`Data`', given by

$$\texttt{Data} = \left[\begin{array}{c|c} \mathbf{x}_1 & y_1 \\ \vdots & \vdots \\ \mathbf{x}_n & y_n \end{array}\right], \tag{1.1}$$

where $\mathbf{x}_i = \big(x_i^{(1)}, ..., x_i^{(d)}\big) \in \mathbb{R}^d$ for every $i = 1, ..., n$.

To get a better, more concrete insight, we start with a motivating example, which is one of the 'founding problems' of ML: **Handwritten digit recognition**. Its goal is to make the 'computer' recognize human handwritten digits, which come from different sources, like images, papers, letters, etc., and classify them into the class labels $\{0,1,2,3,4,5,6,7,8,9\}$. We consider the following experiment: (automatically) recognize the postal code in the address field of a letter. We first take a picture of the address (see Figure 1 (a)); then we segment the image into individual letters and digits (see Figure 1 (b)). Suppose the image of each individual digit has the form of a 28×28 greyscale image (if not, we can transform and resize it; see Figure 1 (d)).

The 28×28 greyscale image corresponds to a vector with $28 \cdot 28 = 784$ entries in $[0,1]$, where $0 \equiv$'black' and $1 \equiv$'white'. Our goal is now to find out the digits in the postal code; if put in mathematical terms: we want to find an 'exact' mapping $f^* : [0,1]^{784} \to \{0,1,2,3,4,5,6,7,8,9\}$ that has *true risk* as small as possible, *i.e.*, has *true risk* close to the *Bayes risk*. By using the notation above, we have $d = 784$ and $\mathcal{Y} = \{0,1,2,3,4,5,6,7,8,9\}$. Here, the *true risk* of a (prediction) function $f : \mathbb{R}^d \to \mathcal{Y}$ (in the experiment: $f : [0,1]^{784} \to \{0,1,2,3,4,5,6,7,8,9\}$) is defined as

$$\texttt{Risk}(f) := \mathbb{E}\Big[\ell\big(\mathbf{X}, Y, f(\mathbf{X})\big)\Big],$$

where the expectation is over a random draw $(\mathbf{X}, Y)$ according to the (unknown) distribution $\mathbb{P}$, and $\ell : \mathbb{R}^d \times \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$ is a given *loss-function*. For classification problems (*i.e.*, when $\mathcal{Y}$ is a discrete set) one can use the '$0-1-$*loss function*'

$$\ell(\mathbf{x}_i, y_i, \hat{y}_i) = \begin{cases} 0, & y_i = \hat{y}_i \\ 1, & \text{otherwise}. \end{cases} \tag{1.2}$$

(a) Postal code

(b) Digit extracted from postal code

(c) Greyscale image

(d) 28×28 greyscale image

Figure 1: (a) Handwritten postal code. (b) Digit '4' extracted from the postal code. (c) Greyscale image of '4'. (d) 28×28 greyscale pixel image of the handwritten '4'.

For regression problems (*i.e.*, when $\mathcal{Y} = \mathbb{R}$) one can use the '*squared loss function*'

$$\ell(\mathbf{x}_i, y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2 . \tag{1.3}$$

The *Bayes risk* in this respect is defined by (see [4, p. 22/23, Def. 2.3])

$$\texttt{Risk}^* := \inf\{\texttt{Risk}(f) \,|\, f : \mathbb{R}^d \to \mathcal{Y},\ f \text{ measurable}\} . \tag{1.4}$$

In case the infimum in (1.4) is attained, we call

$$f^* := \arg\min \texttt{Risk}(f) \quad \Leftrightarrow \quad \texttt{Risk}(f^*) = \min_{f:\mathbb{R}^d \to \mathcal{Y}} \texttt{Risk}(f) = \texttt{Risk}^*$$

the *Bayes classifier*.
Returning to our *classification* experiment, we thus want to find a function f^* such that

$$\mathbb{P}[f^*(\mathbf{X}) \neq Y] = \min_{f:[0,1]^{784} \to \{0,1,2,3,4,5,6,7,8,9\}} \mathbb{P}[f(\mathbf{X}) \neq Y] .$$

By [3, p. 9], the *Bayes classifier* f^* is given by

$$f^*(\mathbf{x}) = \operatorname*{arg\,max}_{k \in \{0,1,2,3,4,5,6,7,8,9\}} \mathbb{P}[Y = k \,|\, \mathbf{X} = \mathbf{x}] .$$

For $k \in \{0,1,2,3,4,5,6,7,8,9\}$, we here denote by

$$\mathbb{P}[Y = k \,|\, \mathbf{X} = \mathbf{x}] = \mathbb{E}[\mathbb{1}_{\{Y=k\}} \,|\, \mathbf{X} = \mathbf{x}] =: m^{(k)}(\mathbf{x}) \tag{1.5}$$

the *a posteriori* probabilities. However, since the distribution of $(\mathbf{X}, Y)$ is unknown in practice, we have to estimate f^* via given data ('training examples') $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ by estimating $m^{(k)}(\cdot)$, $k \in \{0,1,2,3,4,5,6,7,8,9\}$. By following this *supervised learning* approach, we use the given data $\mathbf{D}_n$ drawn from $\{(\mathbf{X}_i, Y_i)\}_{i=1}^n$, with $\mathbf{x}_i \in [0,1]^{784}$ representing a resized greyscale image, and $y_i \in \{0,1,2,3,4,5,6,7,8,9\}$ representing the true class label to construct a function $f_n^* : [0,1]^{784} \to \{0,1,2,3,4,5,6,7,8,9\}$ that has *true risk* close to the *Bayes risk*, *i.e.*, $\texttt{Risk}(f_n^*) \approx \texttt{Risk}^*$. To represent f_n^*, one can use a plug-in estimate

$$f_n^*(\mathbf{x}) = \underset{k \in \{0,1,2,3,4,5,6,7,8,9\}}{\arg\max} m_n^{(k)}(\mathbf{x}), \tag{1.6}$$

where $m_n^{(k)}(\cdot)$ is an estimation of $m^{(k)}(\cdot)$. Given the data $\mathbf{D}_n$, the estimators $m_n^{(k)}(\cdot)$ can be constructed via the data set $\mathbf{D}_n^{(k)} = \{(\mathbf{x}_i, \mathbb{1}_{\{y_i=k\}})\}_{i=1}^n$. In the next section, we present some particular estimators $m_n(\cdot) \equiv m_n^{(k)}(\cdot)$ to estimate $m^{(k)}$, and consequently f^*. We refer to Subsection 3.3 for a continuification of this experiment.

2 Estimators and their implementation

In this section we present the **partitioning estimator**, the **kernel estimator** and the $k-$**Nearest Neighbor (kNN) estimator**, and give details about their implementation. In this conncection, we also present parts of the corresponding Python codes, with which we further explain/discuss implementational details. We refer to [7], where full Python codes are available.

2.1 Uniform partitioning estimator

The realization of the uniform partitioning estimator may be classified into three steps. We follow a *Binary Tree Cuboid (BTC)* structure; see *e.g.* [1], [2] and/or [3], which allows for an efficient way of implementing the estimator; see also Figure 2. Note that in the next subsection, we present a *data-dependent* partitioning estimator, which is much more favorable when the dimension of the state space $\mathbb{R}^d$ is large. We refer to Subsection 2.2 for a more detailed discussion in this direction.

STEP 1: (Partition of the state space $\mathbb{R}^d$) Fix $\kappa \in \mathbb{N}$. We partition the state space $\mathbb{R}^d$ into 2^κ many (uniform) rectangles. In the following, we briefly sketch the procedure (see Source Code 1 below):

(1) Specify a rectangle $\mathbf{R_{0,0}} = [a_1^{(1)}, a_2^{(1)}] \times ... \times [a_1^{(d)}, a_2^{(d)}]$, which contains all points $\{\mathbf{x}_i\}_{i=1}^n \subset \mathbb{R}^d$ (see Figure 3(a)).

(2) Compute $\texttt{MAX} := \max\big(a_2^{(1)} - a_1^{(1)}, ..., a_2^{(d)} - a_1^{(d)}\big)$ in order to find out along which axis to divide the rectangle $\mathbf{R_{0,0}}$ into two (new) rectangles $\mathbf{R_{1,0}}$ and $\mathbf{R_{1,1}}$ (see Figure 3(b)).

(3) Proceed in a similar way for each new rectangle until the amount of (new) rectangles is 2^κ. We denote these 2^κ many rectangles by $\boldsymbol{\mathcal{A}_1} := \mathbf{R}_{\kappa,\mathbf{0}}, ..., \boldsymbol{\mathcal{A}_{2^\kappa}} := \mathbf{R}_{\kappa,\mathbf{2^\kappa - 1}}$; see also Figure 2.

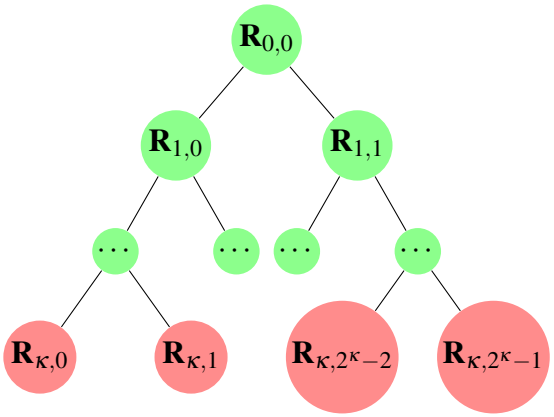


Figure 2: Binary tree structure of the partitioning strategy.

The informations needed to specify a particular rectangle can be saved in a $\big((\kappa+1)\times 2^\kappa\big)-$'cell-matrix'[2] RR, given by

$$\mathtt{RR} = \begin{bmatrix} RR_{0,0} & \dots & \dots & \dots \\ RR_{1,0} & RR_{1,1} & \dots & \dots \\ \vdots & & \ddots & \\ RR_{\kappa,0} & \dots & \dots & RR_{\kappa,2^\kappa-1} \end{bmatrix}.$$

In this connection, the cell-matrix RR 'reflects' the binary tree structure in Figure 2. Each $RR_{i,j}$ $(i=0,\dots,\kappa\,,\ j=0,\dots,2^\kappa-1)$ is a $(d\times 2)-$matrix, which contains the information to specify the rectangle $\mathbf{R_{i,j}}$. Ultimately, in order to conduct the partition of $\mathbb{R}^d$, we only need the last row of RR, *i.e.*, we are only interested in the matrices $RR_{\kappa,0}$,....,$RR_{\kappa,2^\kappa-1}$.

In the following, we illustrate this partitioning procedure by means of an example.

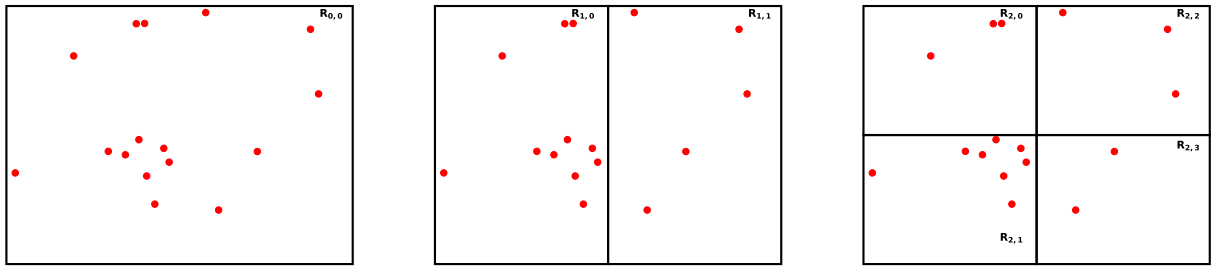
Example 2.1. Let $n=16$, $d=2$ and $\kappa=2$. Figure 3 below illustrates the partition of $\mathbb{R}^2$ into 2^2 many rectangles. According to the procedure from above, we have

$$\mathtt{RR} = \begin{bmatrix} RR_{0,0} & \dots & \dots & \dots \\ RR_{1,0} & RR_{1,1} & \dots & \dots \\ RR_{2,0} & RR_{2,1} & RR_{2,2} & RR_{2,3} \end{bmatrix},$$

where

$$RR_{0,0} = \begin{bmatrix} a_1^{(1)} & a_2^{(1)} \\ a_1^{(2)} & a_2^{(2)} \end{bmatrix}, \quad RR_{1,0} = \begin{bmatrix} a_1^{(1)} & (a_1^{(1)}+a_2^{(1)})/2 \\ a_1^{(2)} & a_2^{(2)} \end{bmatrix}, \quad RR_{1,1} = \begin{bmatrix} (a_1^{(1)}+a_2^{(1)})/2 & a_2^{(1)} \\ a_1^{(2)} & a_2^{(2)} \end{bmatrix}, \dots$$

[2] A 'cell-matrix' is a matrix whose entries consists of matrices.



(a) $\mathbf{R_{0,0}}$ (b) $\mathbf{R_{1,0}}$, $\mathbf{R_{1,1}}$ (c) $\mathbf{R_{2,0}}, \mathbf{R_{2,1}}, \mathbf{R_{2,2}}, \mathbf{R_{2,3}}$

Figure 3: (a) Specification of rectangle $\mathbf{R_{0,0}} = [a_1^{(1)}, a_2^{(1)}] \times [a_1^{(2)}, a_2^{(2)}]$, which contains all points $\{x_i\}_{i=1}^n$ highlighted in red. (b) Specification of rectangles $\mathbf{R_{1,0}} = [a_1^{(1)}, (a_1^{(1)}+a_2^{(1)})/2] \times [a_1^{(2)}, a_2^{(2)}]$ and $\mathbf{R_{1,1}} = [(a_1^{(1)}+a_2^{(1)})/2, a_2^{(1)}] \times [a_1^{(2)}, a_2^{(2)}]$ according to (2) from above. (c) Final result of the partition of $\mathbb{R}^2$ into 2^2 many rectangles.

```
# The following function partitions the state space (which is determined by the minimum and maximum value
# of the {x_i}_{i=1,...,n} & and a given tolerance) into 2^kappa many uniform rectangles.
def partitioning(kappa,A,tolerance): # Define partitioining function
    RR=np.empty(shape=(kappa+1,2**(kappa)),dtype='object') # Initialize cell-matrix RR
    RR1=np.zeros((d,2)) # initialize first entry of RR
    for i in range(d):
        # Set up the first rectangle (resp. corresponding informations), which contains all points. 'tolerance' determines the distance of the rectangle to the furthest points, which all are located inside the rectangle
        RR1[i,0]=np.amin(A)-tolerance
        RR1[i,1]=np.amax(A)+tolerance
    RR[0][0]= RR1 # specification of first rectangle; see Subsection 2.1 STEP 1: (1) in the manuscript
    # Successively specify subsequent rectangles according to a Binary Tree Cuboid structure; see Subsection 2.1 STEP 1: (2) in the manuscript
    for p in range(kappa):
        for q in range(2**p):
            maxi=np.copy(RR[p][q][0][1]-RR[p][q][0][0])
            iter=0
            for i in range(d):
                comp=np.copy(RR[p][q][i][1]-RR[p][q][i][0])
                if comp > maxi:
                    iter=i
                    maxi=np.copy(comp)
            RR[p+1][2*q] = np.copy(RR[p][q])
            RR[p+1][2*q][iter][1]=(RR[p][q][iter][0]+RR[p][q][iter][1])/2
            RR[p+1][2*q+1] = np.copy(RR[p][q])
```

```
            RR[p+1][2*q+1][iter][0] = (RR[p][q][iter][0] + RR[p][q][iter][1]) / 2
    return RR
```

Source Code 1: Uniform partition of $\mathbb{R}^d$.

Source Code 1 generates the cell-matrix RR by following (1) – (3) from **STEP 1**.

STEP 2: (Localization) We choose a point point $\in \mathbb{R}^d$. Then, we determine the rectangle $\boldsymbol{\mathcal{A}_1}$,...,$\boldsymbol{\mathcal{A}_{2^\kappa}}$ in which point is located.

```
# For a given Point 'point' in the state space, the following function determines the rectangle (resp.
# corresponding informations) in which 'point' is included. Here, the localization procedure successively checks every rectangle if 'point' is included, and then stops.
def localization(RR,point):  # define localization function
    for i in range(len(RR[-1,:])): # successively check for each rectangle (all rectangles are saved in the last line 'RR[-1,:]' of cell-matrix RR), if 'point' is included
        if (RR[-1,i][:,0] <= point ).all()==True and ( point <= RR[-1,i][:,1] ).all()==True:
            index=i
            break  # stop, if rectangle resp. corresponding index is found, in which 'point' is included
    return index
```

Source Code 2: Localization of point by successively checking every rectangle.

The strategy in Source Code 2 above successively checks every rectangle $\boldsymbol{\mathcal{A}_1}$,...,$\boldsymbol{\mathcal{A}_{2^\kappa}}$, if point is located in it, and thus requires at most $\mathcal{O}(2^\kappa)$ many checks. However, a more convenient localization strategy is given in Source Code 3 below, which (recursively) utilizes the binary tree structure of the cell-matrix RR; *cf.* also Figure 2, and thus requires $\mathcal{O}(\kappa)$ many checks.

```
# For a given Point 'point' in the state space, the cell-matrix RR, two indices 'counter' and 'index' ('counter' refers to the row-index in RR; 'index' refers to the column-index of RR) and kappa, the following function determines the rectangle (resp.
# corresponding informations) in which 'point' is included. The function is defined recursively and utilizes the binary tree structure of RR.
# Note: to start, set counter=0, index=0
def localization(RR,point,counter,index,kappa): # define localization function
    if counter == kappa: # stop, if last row of RR is reached
        return index
    # Recursively utilize binary tree structure of RR
    if (RR[counter+1,2*index][:,0] <= point ).all()==True and ( point <= RR[counter+1,2*index][:,1] ).all()==True:
        newindex=int(2*index)
        return localization(RR,point,counter+1,newindex,kappa)
    else:
```

```
        newindex=int(2*index+1)
        return localization(RR, point, counter + 1, newindex, kappa)
```

Source Code 3: Localization of `point` by utilizing the tree structure.

STEP 3: (Estimator) Compute the partitioning estimator '$m_n(\texttt{point})$' in order to estimate the label of the given point $\texttt{point} \in \mathbb{R}^d$. Suppose `point` is located in the rectangle $\boldsymbol{\mathcal{A}_j}$ $(\boldsymbol{j} = 1, ..., 2^\kappa)$, then

$$m_n(\texttt{point}) = \frac{\sum_{i=1}^{n} \mathbb{1}_{\{\mathbf{x}_i \in \boldsymbol{\mathcal{A}_j}\}} \cdot y_i}{\sum_{i=1}^{n} \mathbb{1}_{\{\mathbf{x}_i \in \boldsymbol{\mathcal{A}_j}\}}}. \tag{2.1}$$

Source Code 4 below illustrates the computation via (2.1).

```
#index=localization(RR,point) # needed from Localization-Algorithm
index=localization(RR,point,0,0,kappa) # needed from Localization-
    Algorithm
##############################################
# The following function realizes the partitioning estimator m_n=m_n(
    point), and yields the estimated label of
# the given 'point'.
def partEstimator(RR,index,Data): # define partitioning estimator
    factor1=0
    factor2=0
    # Compute numerator and denominator in equation (2.1) in the
    manuscript
    for i in range(n):
        if (RR[-1,index][:,0] <= Data[i,0:d] ).all()==True and ( Data[i
    ,0:d] <= RR[-1,index][:,1] ).all()==True:
            factor1 += Data[i,d]
            factor2 +=1
    if factor2 ==0:
        m_n=0
    else:
        m_n=factor1/factor2
    return m_n
```

Source Code 4: Estimation of label of `point`.

2.2 Data-dependent partitioning estimator

Since the computational complexity of a uniform partition of the state space $\mathbb{R}^d$ grows exponentially with the dimension d, the *uniform partitioning estimator* from Subsection 2.1 may not be a good choice for tackling classification/regression problems when the underlying state space is high-dimensional. However, using a *data-dependent* partition of $\mathbb{R}^d$ instead, which, in some sense, incorporates the 'distribution' of the given points in $\mathbb{R}^d$, drastically cuts down the complexity

and is much more favorable, especially when d is large. In the following, we present such a *data-dependent* partitioning strategy (see [2]), which partitions $\mathbb{R}^d$ based on *statistically equivalent blocks*, *i.e.*, where each rectangle/compartement in the partition contains the same amount of points; see [3, p. 243 *ff.*] for further informations.

Similar to steps **1**, **2** and **3** in Subsection 2.1, we realize the *data-dependent* partitioning estimator. Compared to the uniform partitioning estimator, the main difference here is the *data-dependent* partition in **STEP 1** below. The localization procedure in **STEP 2**, and the computation of the estimator in **STEP 3** below are similar to those in Subsection 2.1.

STEP 1: (Data-dependent partition of the state space $\mathbb{R}^d$) Fix $\kappa \in \mathbb{N}$, and assume $n = 2^N$ with $\kappa \leq N \in \mathbb{N}$. We partition the state space $\mathbb{R}^d$ into 2^κ many rectangles, where each rectangle contains $\frac{n}{2^\kappa}$ points. Based on a coordinate-wise empirical standard deviation criterion of $\{\mathbf{x}_i\}_{i=1}^n$, the following algorithm divides the given points into 2^κ many disjoint groups; see Source Code 5 for an implementation in Python, and also [7]. Each particular group of points corresponds to a rectangle/compartement in the partition in which they are located inside; see Source code 6.

Algorithm 2.2 (BTC-Algorithm; see [2])**.** Choose $\kappa \in \mathbb{N}$. Let $\mathcal{S}_{0,0} = [\mathbf{x}_1, ..., \mathbf{x}_n]^\top \in \mathbb{R}^{n \times d}$.
For $p = 0, ..., \kappa - 1$ do:
For $q = 0, ..., 2^p - 1$ do:

(I) Set $\boldsymbol{\mathcal{S}} := \mathcal{S}_{p,q}$.

(II) Find the component $\ell \in \{1, ..., d\}$ in $\boldsymbol{\mathcal{S}}$ which possesses the largest empirical standard deviation $\hat{\sigma}_\ell \in \mathbb{R}$; denote this component by $\ell_{p,q} \in \{1, ..., d\}$.

(III) Compute median $med_{p,q} \in \mathbb{R}$ of $\boldsymbol{\mathcal{S}}[:, \ell_{p,q}]$.

(IV) Divide $\boldsymbol{\mathcal{S}}$ into two equal parts $\boldsymbol{\mathcal{S}} = \mathcal{S}_{p+1,2q} \cup \mathcal{S}_{p+1,2q+1}$ according to the criterion $x_i^{(\ell_{p,q})} \leq med_{p,q}$ (for those points, whose indices i belong to $\boldsymbol{\mathcal{S}}$).

```
# BTC-ALGORITHM: Realization of Algorithm 2.2 in the manuscript
def MEDpart(kappa,x):
    SS=np.empty(shape=(kappa+1,2**(kappa)),dtype='object') # initialize
    cell-matrix SS, in which all 'S_{p,q}' are saved
    MED=np.zeros((kappa+1,2**(kappa))) # initialize matrix 'MED', in
    which  all values
    # med_{p,q} are saved
    LL=np.zeros((kappa+1,2**(kappa))) # initialize matrix 'LL', in which
    all values
    # l_{p,q} are saved
    SS[0][0]=x # set S_{0,0}
    for p in range(kappa):
        for q in range(2**p):
            S=np.copy(SS[p][q])
            maxi=np.copy(np.std(S[:,0]))
            l=0
```

```
            # Find out the component which posseses largest empirical
    standard deviation, cf. (II) in Algorithm 2.2 in the manuscript
            for k in range(d):
                if maxi<np.std(S[:,k]):
                    maxi=np.copy(np.std(S[:,k]))
                    l=np.copy(k)
            med=np.median(S[:,l]) # cf. (III) in Algorithm 2.2 in the
    manuscript
            # Realize (IV) in Algorithm 2.2 in the manuscript
            B=np.copy(S[S[:, l].argsort()]) # sort matrix for appropriate
     dividing according to (IV)
            S=np.copy(B)
            t=int(len(S[:,0])/2)
            SS[p + 1][2*q]=np.copy(S[t:n,:])
            SS[p + 1][2*q + 1]=np.copy(S[0:t,:])
            MED[p][q]=np.copy(med)
            LL[p][q]=np.copy(l)
     Scell=SS[-1,:] # last row of cell-matrix SS contains all pair of
    points, which each are included in the particular rectangles/
    compartements in the partition
     return MED, LL, Scell
```

Source Code 5: BTC-Algorithm.

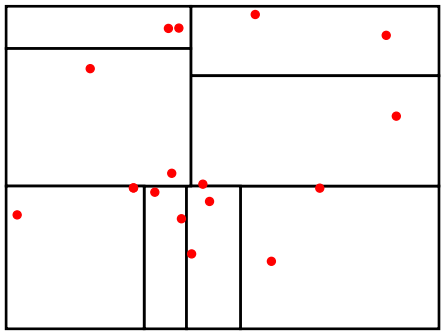


Figure 4: Data-dependent partition of $\mathbb{R}^d$ based on the points $\{\mathbf{x}_i\}_{i=1}^n$ (highlighted in red) from Example 2.1: Here, $d = 2$, $n = 2^4$, $\kappa = 3$. Each rectangle contains $\frac{2^4}{2^3} = 2$ points.

The informations needed to specify a particular rectangle can (again) be saved in the cell-matrix RR from Subsection 2.1. Source Code 6 below generates this matrix RR in case of a *data-dependent* partitioning.

```
MED, LL, Scell = MEDpart(kappa,Data[:,0:d]) # needed from BTC-Algorithm
############################################################
# The following function partitions the state space (based on Algorithm
    2.2 in the manuscript) --- which is determined by the minimum and
    maximum value
# of the {x_i}_{i=1,...,n} & and a given tolerance --- into R=2^(kappa)
    many rectangles.
```

```
def partitioning(kappa,A,tolerance,MED,LL):
    RR=np.empty(shape=(kappa+1,2**(kappa)),dtype='object') # initialize
    cell-matrix RR
    RR1=np.zeros((d,2)) # initialize first entry of RR
    for i in range(d):
    # Set up the first rectangle (resp. correspondinhg informations)
    which contains all points. 'tolerance' determines the distance of the
     rectangle to the furthest points, which all are located inside the
    rectangle
        RR1[i,0]=np.amin(A)-tolerance
        RR1[i,1]=np.amax(A)+tolerance
    RR[0][0]= RR1 # specification of first rectangle
    # Successfully specify subsequent rectangles/compratements according
    to a Binary Tree Cuboid structure
    for p in range(kappa):
        for q in range(2**p):
            for i in range(d):
               if LL[p][q] == i:
                  RR[p+1][2*q] = np.copy(RR[p][q])
                  RR[p+1][2*q][i][0] = np.copy(MED[p][q])
                  RR[p+1][2*q + 1] = np.copy(RR[p][q])
                  RR[p+1][2*q + 1][i][1] = np.copy(MED[p][q])
    return RR
```

Source Code 6: Data-dependent partition of $\mathbb{R}^d$.

STEP 2 and STEP 3: See steps **2** and **3** in Subsection 2.1.

We now present an example, where the *data-dependent* partitioning estimator is used for classification. All corresponding Python codes to reconstruct Example 2.3, *i.e.*, to realize steps **1**, **2** and **3** above can be found in [7].

Example 2.3 (Classification via data-dependent partitioning estimator)**.** Let $d = 2$, $n = 2^{10}$ and $\mathcal{Y} = \{1,2,3,4\}$. We use [7, DataGeneration1.py] to generate data $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$; see Figure 5 for a realization. According to steps **1**, **2** and **3** from Subsection 2.2, we choose $\kappa = 8$ and fix $\texttt{point} = (1, 1.35)^\top$. An implementation of the partitioning estimator yields $m_n(\texttt{point}) = 1$; see [7, Example2.3_PartEstimator.py] and also Figure 6.

2.3 Kernel estimator

Opposed to the implementation of the partitioning estimator from Subsections 2.1 and 2.2, the realization of the kernel estimator is much simpler and can be classified into two steps:

STEP 1: (Kernel function) Choose a *kernel function* $K : \mathbb{R}^d \to [0,\infty)$, which is used to express the similarity of two points in $\mathbb{R}^d$; for instance

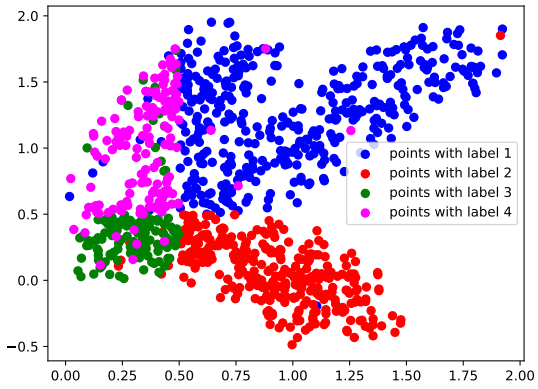
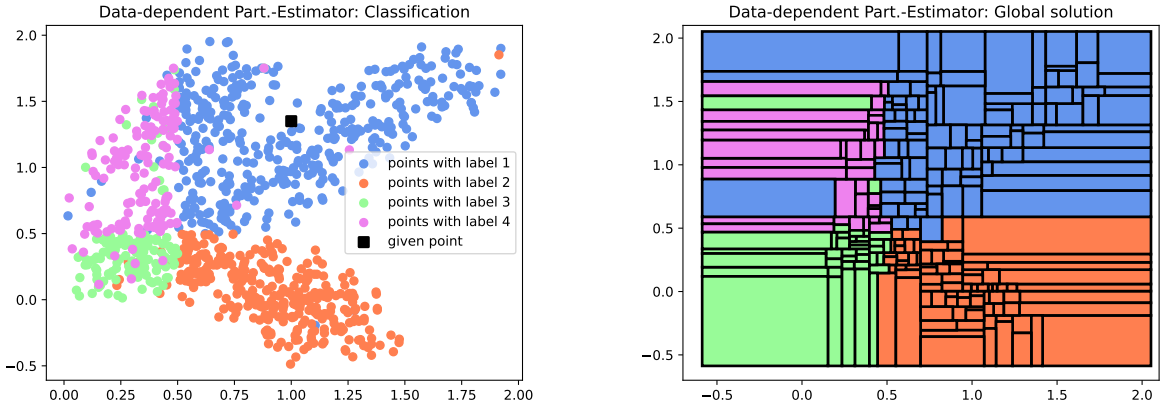


Figure 5: Data generated via [7, DataGeneration1.py].



(a) Estimation (b) Global solution

Figure 6: (a) Visualization of data and `point` (highlighted in black), whose label is estimated. (b) Visualization of the global solution.

- **Naive Kernel:**

$$K(\mathbf{x}) = \mathbb{1}_{\{\|\mathbf{x}\|_{\mathbb{R}^d} \leq 1\}}.$$

- **Epanechnikov Kernel:**

$$K(\mathbf{x}) = \begin{cases} \frac{3}{4}\left(1 - \|\mathbf{x}\|_{\mathbb{R}^d}^2\right), & \|\mathbf{x}\|_{\mathbb{R}^d} \leq 1 \\ 0, & \text{otherwise.} \end{cases}$$

STEP 2: (Estimator) Fix a bandwidth $h > 0$, and a point $\texttt{point} \in \mathbb{R}^d$, whose corresponding label

is estimated by the kernel estimator

$$m_n(\texttt{point}) = \frac{\sum_{i=1}^{n} K\left(\frac{\texttt{point}-\mathbf{x}_i}{h}\right) \cdot y_i}{\sum_{i=1}^{n} K\left(\frac{\texttt{point}-\mathbf{x}_i}{h}\right)}. \tag{2.2}$$

Source Code 7 below realizes steps **1** and **2** from above: first a kernel function is defined, then the kernel estimator in (2.2) is computed.

```
# Define Kernel-function: Naive-Kernel
def naivkernel(a):
    if np.linalg.norm(a) <= 1:
        value =1
    else:
        value = 0
    return value
############################################################
# Define Kernel-function: Epanechnikov-Kernel
def epanechnikovkernel(a):
    if np.linalg.norm(a)<=1:
        value=(3/4)*(1-(np.linalg.norm(a)**2))
    else:
        value=0
    return value
############################################################
# Define Kernel-Estimator:
def kernelestimator(point,Data,h):
    factor1=0
    factor2=0
    for i in range(len(Data)):
        #factor1 += naivkernel((point-Data[i,0:d])/h)*Data[i,d]
        #factor2 += naivkernel((point-Data[i,0:d])/h)
        factor1 += epanechnikovkernel((point - Data[i, 0:d]) / h) * Data[
    i, d]
        factor2 += epanechnikovkernel((point - Data[i, 0:d]) / h)
    if factor2 == 0:
        m_n = 0
    else:
        m_n = factor1 / factor2
    return m_n
############################################################
```

Source Code 7: Estimation of label of point

We now consider an example, which is similar to Example 2.3, but uses the kernel estimator for the underlying classification task. Th full Python code to reconstruct Example 2.4 is provided in [7].

Example 2.4. (Classification via kernel estimator) Suppose the same framework as in Example 2.3; *i.e.*, $d = 2$, $n = 2^{10}$, $\mathcal{Y} = \{1,2,3,4\}$, the same data, and $\texttt{point} = (1, 1.35)^\top$. For the estimation of the label of `point`, we choose the *Epanechnikov kernel* as a kernel function and fix $h = 0.1$. An implementation of the kernel estimator yields $m_n(\texttt{point}) = 1$; see [7, Example2.4_KernelEstimator.py].

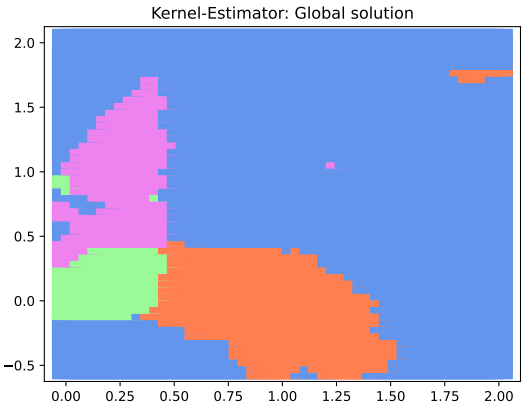


Figure 7: Visualization of the global solution.

2.4 $k-$Nearest Neighbor (kNN) estimator

The realization of the kNN estimator may be classified into three steps:

STEP 1: (Setup) Fix $k \in \{1,...,n\}$, and fix a point $\texttt{point} \in \mathbb{R}^d$ whose corresponding label should be estimated. Compute all distances $\|\texttt{point} - \mathbf{x}_i\|_{\mathbb{R}^d}$ $(i = 1,...,n)$, and save the results in an extra column of the matrix '`Data`' in (1.1):

$$\texttt{Datadist} = \left[\begin{array}{c|c|c} \mathbf{x}_1 & y_1 & \|\texttt{point} - \mathbf{x}_1\|_{\mathbb{R}^d} \\ \vdots & \vdots & \vdots \\ \mathbf{x}_n & y_n & \|\texttt{point} - \mathbf{x}_n\|_{\mathbb{R}^d} \end{array}\right] \in \mathbb{R}^{n\times(d+2)}.$$

STEP 2: (Sort and extract) Sort the rows of matrix '`Datadist`' in ascending order based on the value in the last column:

$$\texttt{Datadistsort} = \left[\begin{array}{c|c|c} \mathbf{x}_{i_1} & y_{i_1} & \|\texttt{point} - \mathbf{x}_{i_1}\|_{\mathbb{R}^d} \\ \vdots & \vdots & \vdots \\ \mathbf{x}_{i_k} & y_{i_k} & \|\texttt{point} - \mathbf{x}_{i_k}\|_{\mathbb{R}^d} \\ \mathbf{x}_{i_{k+1}} & y_{i_{k+1}} & \|\texttt{point} - \mathbf{x}_{i_{k+1}}\|_{\mathbb{R}^d} \\ \vdots & \vdots & \vdots \\ \mathbf{x}_{i_n} & y_{i_n} & \|\texttt{point} - \mathbf{x}_{i_n}\|_{\mathbb{R}^d} \end{array}\right] \in \mathbb{R}^{n\times(d+2)}.$$

Extract from matrix '`Datadistsort`' the first k rows:

$$\texttt{Datadistsortk} = \left[\begin{array}{c|c|c} \mathbf{x}_{i_1} & y_{i_1} & \|\texttt{point} - \mathbf{x}_{i_1}\|_{\mathbb{R}^d} \\ \vdots & \vdots & \vdots \\ \mathbf{x}_{i_k} & y_{i_k} & \|\texttt{point} - \mathbf{x}_{i_k}\|_{\mathbb{R}^d} \end{array}\right] \in \mathbb{R}^{k\times(d+2)}.$$

The points $\mathbf{x}_{i_1},...,\mathbf{x}_{i_k}$ are the $k-$nearest neighbors of `point`.

```
# Compute distances:
def distancemeasure(Data,point,k):
    Dataplusdist=np.zeros((len(Data),d+2)) # initialize matrix
    Dataplusdist[:, 0:d+1] = np.copy(Data)
    for i in range(len(Data)):
        Dataplusdist[i,d+1]=np.linalg.norm(point-Data[i,0:d]) # compute
    distance
    Dataplusdist=np.copy(Dataplusdist[Dataplusdist[:, d+1].argsort()]) #
    sort distances in ascending order
    Dataplusdistk=Dataplusdist[0:k,:] # extract the first k rows
    return Dataplusdistk
```

Source Code 8: Implementation of STEP 1 and STEP 2

STEP 3: (Estimator) In order to estimate the label of `point`, we compute the kNN estimator via

$$m_n(\texttt{point}) = \frac{1}{k}\sum_{\ell=1}^{k} y_{i_\ell}, \tag{2.3}$$

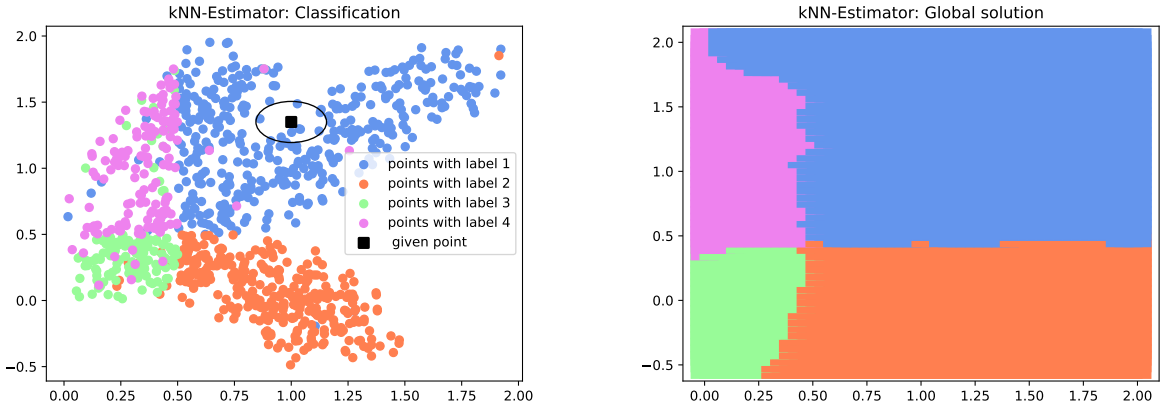
where the bracket $[\cdot]$ either rounds the expression inside it up, or down.

```
# Define kNN-Estimator:
def kNNestimator(Dataplusdistk,k):
    sum=0
    for i in range(len(Dataplusdistk)):
        sum += Dataplusdistk[i,d]
    m_n=sum/k
    return m_n
```

Source Code 9: Estimation of label of point

The following example is similar to Example 2.3 resp. 2.4, but uses the kNN estimator for the underlying classification task. Th full Python code to reconstruct Example 2.5 is again provided in [7].

Example 2.5. (Classification via kNN estimator) Suppose the same framework as in Example 2.3; *i.e.*, $d = 2$, $n = 2^{10}$, $\mathcal{Y} = \{1,2,3,4\}$, the same data, and $\texttt{point} = (1, 1.35)^\top$. For the estimation of the label of `point`, we fix $k = 9$. An implementation of the kNN estimator yields $f_n^*(\texttt{point}) = 1$; see [7, Example2.5_kNNEstimator.py].



(a) Estimation (b) Global solution

Figure 8: (a) Visualization of data and `point` (highlighted in black), whose label is estimated. The circle around `point` involves the $k-$nearest neighbors. (b) Visualization of the global solution.

In the following remark, we compare the three estimators presented in this section from an algorithmical viewpoint.

Remark 2.1. As far as preliminary work is concerned, the partitioning estimator clearly leads the list. Before realizing a classification resp. regression task, a (data-dependent) partition of the state space $\mathbb{R}^d$ has to be carried out first, which has a complexity of roughly $\mathcal{O}(d2^{\kappa+1})$ (uniform partition) resp. $\mathcal{O}(dn2^{\kappa+1})$ (data-dependent partition), *cf.* Source Codes 1, 5 and 6. In order to estimate the label of a given point according to (2.1), we first localize it; *cf.* Source Code 3, which requires $\mathcal{O}(\kappa)$ checks. The evaluation via (2.1) then requires $\mathcal{O}(n)$ further checks; *cf.* Source Code 4. Second on the list is the kNN estimator with a preliminary work complexity of $\mathcal{O}(nd)$; *cf.* Source Code 8. The estimation via (2.3) then has a complexity of $\mathcal{O}(k)$, *cf.* Source Code 9. In contrast, the kernel estimator does not require any algorithmical preliminary work, which is why its implementation is straightforward; *cf.* Source Code 7, and the 'easiest' compared to the other estimators. However, the estimation via (2.2) is the most time consuming with a complexity of roughly $\mathcal{O}(nd)$, which, in particular, has an influence on the computation of a global solution. We refer to Subsection 3.2 for a further discussion in this direction, and for a comparison of the above estimators from a more computational/simulational viewpoint.

3 Data-dependent selection of parameters in the estimation

3.1 K-fold cross validation

The parameters $\kappa \in \mathbb{N}$, $h > 0$ and $k \in \mathbb{N}$ involved in the error estimators in Section 2 determine the 'accuracy' of the estimators. Improper choices of κ, h, and k being too 'small' or too 'large' might lead to underfitting resp. overfitting phenomenons. A question which naturally arises now is '*How to choose the 'optimal' accuracy-parameter(s)?*'. We answer this question by considering

the following machine learning experiment:

Suppose we have given data $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. In a first step, we shuffle and then split the data into 'learning/training data' $\mathbf{D}_{n_\mathrm{L}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n_\mathrm{L}}$ and 'test data' $\mathbf{D}_{n_\mathrm{T}} = \{(\mathbf{x}_i, y_i)\}_{i=n_\mathrm{L}+1}^{n_\mathrm{T}}$, where $n = n_\mathrm{L} + n_\mathrm{T}$. This strategy has been analyzed in [3, Chap. 7,8]. For an implementation, see Source code 10 below, which yields the data $\mathbf{D}_{n_\mathrm{L}}$ and $\mathbf{D}_{n_\mathrm{T}}$.

```
# Generate (random) training set & test set; shuffel and split the data:
# Suppose the (n x (d+1))-matrix 'Data' is given (cf. (1.1) in the
    manuscript).
fac=0.7 # fac is a number between 0 and 1; Example: fac=0.7 splits the
    data into 70% training data and
# 30% test data
randomtrainindices=np.random.choice(n,int(fac*n),replace=False) #
    generate int(fac*n)-many different
# random indices from 0 to n
randomtestindices=list(set(range(n))-set(randomtrainindices)) # save the
    other indices (not chosen above)
TrainData=np.zeros((len(randomtrainindices),len(Data[0]))) # initialize
    the training data
TestData=np.zeros((len(randomtestindices),len(Data[0]))) # initialize the
     test data
# The training data are built via the randomly chosen indices in '
    randomtrainidices':
n_L=len(randomtrainindices)
for i in range(n_L):
    TrainData[i,:]=Data[randomtrainindices[i],:]
# The test data are built via the indices in 'randomtestidices':
n_T=len(randomtestindices)
for i in range(n_T):
    TestData[i,:]=Data[randomtestindices[i],:]
```

Source Code 10: Generate random training and test data

The training data are used to train the estimator, *i.e.*, to find out the optimal 'accuracy-patrameter(s)'. The test data are independent from the training data and will be used to evaluate the success of the estimator. In this regard, we introduce the **training error** and **test error**:

Training error: We predict the labels of all training points $\{\mathbf{x}_i\}_{i=1}^{n_\mathrm{L}}$ (although the true labels are known) via the estimator: call the predicted labels $\hat{y}_i$. We then compute the error on each training point $\{\mathbf{x}_i\}_{i=1}^{n_\mathrm{L}}$

$$\mathtt{err}(\mathbf{x}_i, y_i, \hat{y}_i) = \ell(\mathbf{x}_i, y_i, \hat{y}_i)\,,$$

where $\ell : \mathbb{R}^d \times \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$ is a given *loss-function*; see (1.2) and (1.3).
The training error is now defined by

$$\mathrm{Err}_{\mathrm{train}} := \frac{1}{n_\mathrm{L}} \sum_{i=1}^{n_\mathrm{L}} \mathtt{err}(\mathbf{x}_i, y_i, \hat{y}_i)\,.$$

Test error: We predict the labels of all test points $\{\mathbf{x}_i\}_{i=n_{\mathrm{L}}+1}^{n_{\mathrm{T}}}$ (although the true labels are known) via the estimator: call the predicted labels $\hat{y}_i$. We then compute the error on each test point: $\mathtt{err}(\mathbf{x}_i, y_i, \hat{y}_i) = \ell(\mathbf{x}_i, y_i, \hat{y}_i)$. The test error can now be defined by

$$\mathtt{Err}_{\mathrm{test}} := \frac{1}{n_{\mathrm{T}}} \sum_{i=n_{\mathrm{L}}+1}^{n_{\mathrm{T}}} \mathtt{err}(\mathbf{x}_i, y_i, \hat{y}_i)\,.$$

Next, we illustrate the concept of $K-$fold cross validation, which is one of the most important methods to determine the optimal accuracy parameter(s), in the case of the partitioning estimator. We refer to the Python files 'Example2.3_PartEstimator.py', 'Example2.4_KernelEstimator.py' and 'Example2.5_kNNEstimator.py' in [7], in which $K-$fold cross validation is each included in the setting of *classification* via the partitioning estimator, kernel estimator and kNN estimator.

Algorithm 3.1. ($K-$fold cross validation for the partitioning estimator)
Input: $K \in \mathbb{N}$, training data $\mathbf{D}_{n_{\mathrm{L}}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n_{\mathrm{L}}}$, and a set $\mathcal{P}_n$ of different parameter combinations, *i.e.*, a set of different values of κ.

(I) Partition the training set into K parts that are equally large. These parts are called 'folds'.

(II) **For** $\kappa \in \mathcal{P}_n$ do:
For $j = 1, ..., K$ do:

- Build one training set out of folds $1, ..., j-1, j+1, ..., K$; call this fold `trainingfold`; call the leftover set `testfold`
- Realize the partitioning estimator with a (data-dependent) partition based on all points in `trainingfold`
- Compute the *validation error* on fold j ($\equiv$ `testfold`):

$$\mathtt{err}_{\mathrm{val}}(\kappa, j) := \frac{1}{\#\{\text{points in } \mathtt{testfold}\}} \sum_{\mathbf{x}_i \in \mathtt{testfold}} \ell(\mathbf{x}_i, y_i, \hat{y}_i)$$

(III) Compute the average validation error over folds:

$$\mathtt{err}(\kappa) := \frac{1}{K} \sum_{j=1}^{K} \mathtt{err}_{\mathrm{val}}(\kappa, j)$$

(IV) Determine the optimal $\kappa^* := \arg\min_{\kappa \in \mathcal{S}} \mathtt{err}(\kappa)$.

Output: κ^*.

After the $K-$fold cross validation procedure, we realize the partitioning estimator with a (data-dependent) partition into 2^{κ^*} many rectangles based on all training points $\{\mathbf{x}_i\}_{i=1}^{n_{\mathrm{L}}}$. Then, and **only** then, we use a completely new test set to compute the test error.

Remark 3.1. **1.** It is not allowed to use the test set in the training procedure!
2. The choice of the number of folds K is not so critical. In practice, people often use $K = 5$ or $K = 10$.
3. The $K-$fold cross validation procedure is computationally expensive, but there is no other systematic method to choose 'accuracy parameters' in a useful way.
4. When dealing with a *classification* problem, we predict the labels '$\hat{y}_i$' via '$f^*_{n_{\mathrm{L}}}(\mathbf{x}_i) = \hat{y}_i$' according to (1.6), and choose (1.2) as *loss-function* in Algorithm 3.1. On the other hand, when dealing with a *regression* problem, we predict the labels '$\hat{y}_i$' via '$m_{n_{\mathrm{L}}}(\mathbf{x}_i) = \hat{y}_i$' according to the estimators presented in Section 2, which, in particular, approximate (3.1) below, and choose (1.3) as *loss-function* in Algorithm 3.1.

3.2 Computational comparison

In this subsection, we compare the different error estimators presented in Section 2 on the basis of further examples.

We start with a comparison of Exmaples 2.3, 2.4 and 2.5, where for the same data set (see Figure 5) a global solution is computed via the partitioning estimator, the kernel estimator and the kNN estimator; see Figures 6 (b), 7 and 8 (b). Table 1 below compares the computational time each estimator (with corresponding optimal 'accuracy' parameter determined via $5-$fold cross validation) needs for the generation of the global solution. There, we can clearly see that the partitioning estimator outclasses the other two estimators as far as computational time is concerned, which is due to the 'character'/setup of the estimators: the partitioning estimator automatically yields a 'rectanglewise'/'compartementwise' global solution of the given problem (*i.e.*, in the sense that the label in each rectangle/compartment is the same); whereas the kernel estimator and the kNN estimator yield a pointwise solution (*i.e.*, at every given point in the state space $\mathbb{R}^d$ the corresponding label has to be estimated again and again, which is computationally more costly). These temporal differences increase more drastically with growing dimension of the state space $\mathbb{R}^d$.

Estimators	**Computational time**	**(Training) error**
Partitioning estimator	4 sec	≈ 0.0712
Kernel estimator	81 sec	≈ 0.0517
kNN estimator	33 sec	≈ 0.0576

Table 1: Comparisons of the computational time for the global solution and the (training) errors of Examples 2.3, 2.4 and 2.5. Here, we used all the data as training data.

Next, we consider an example which deals with *regression*. Similar to the mathematical framework explained in the 'handwritten digit recognition–experiment' for *classification* in the introduction, the goal in *regression* analysis is to find a (regression) function $f^* : \mathbb{R}^d \to \mathbb{R}$ (note that

$\mathcal{Y} = \mathbb{R}$), such that

$$\mathbb{E}\left[|f^*(\mathbf{X}) - Y|^2\right] = \min_{f:\mathbb{R}^d \to \mathbb{R}} \mathbb{E}\left[|f(\mathbf{X}) - Y|^2\right],$$

where $(\mathbf{X}, Y) \in \mathbb{R}^d \times \mathbb{R}$ is a random variable on a given probability space $(\Omega, \mathcal{F}, \mathbb{P})$. By [3, p. 2], $f^*(\cdot) = m(\cdot)$, where $m : \mathbb{R}^d \to \mathbb{R}$ is given by

$$m(\mathbf{x}) := \mathbb{E}[Y \,|\, \mathbf{X} = \mathbf{x}] \qquad \text{forall } \mathbf{x} \in \mathbb{R}^d. \tag{3.1}$$

Again, since the distribution of $(\mathbf{X}, Y)$ is unknown in general, we have to estimate the *regression* function $m(\cdot)$ by $m_n(\cdot)$ based on given data $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ in practice. We refer to Section 2 for different estimators '$m_n(\cdot)$'.

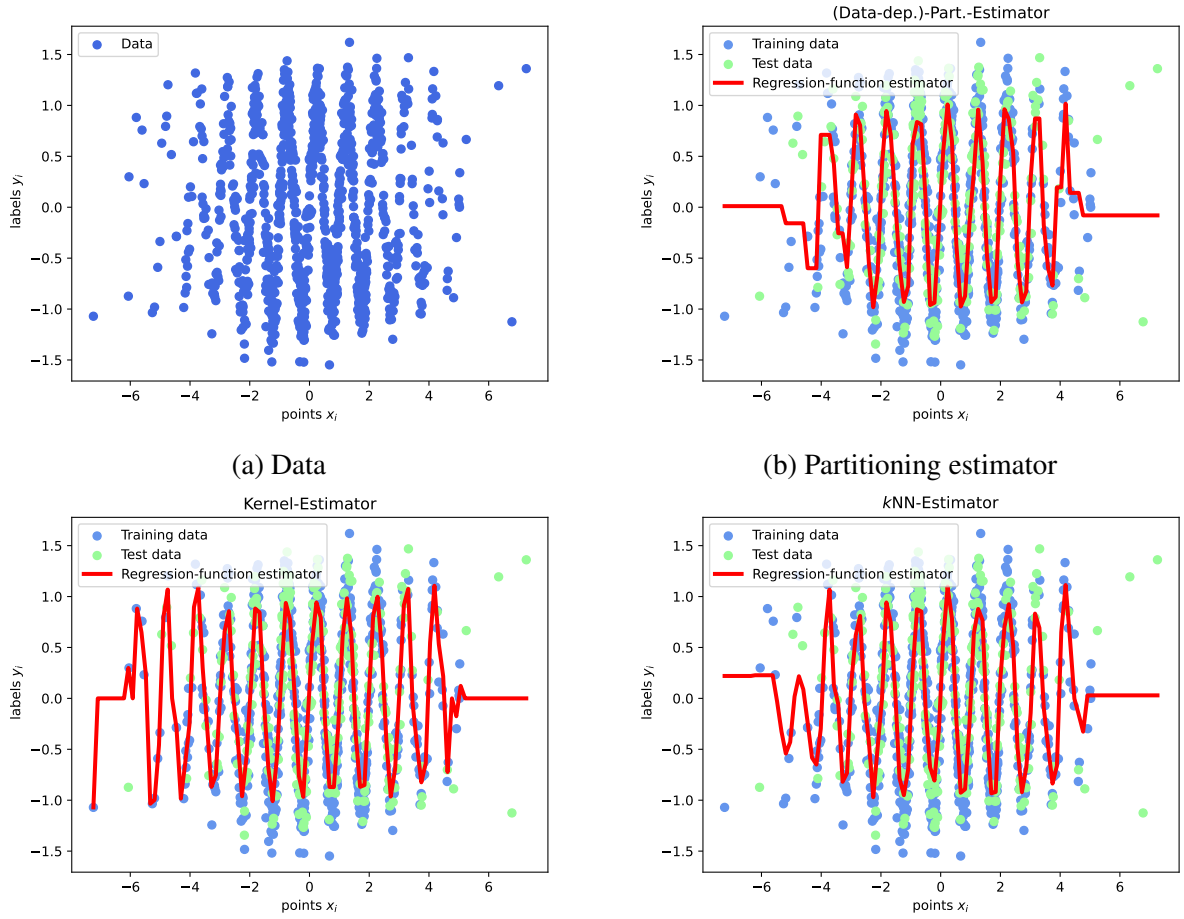
Example 3.2. (Regression) Let $d = 1$, $n = 2^{10}$ and $\mathcal{Y} = \mathbb{R}$. We use [7, DataGeneration2.py] to generate data $\mathbf{D}_n = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$; see Figure 9 (a) for a realization. Table 2 below compares the performance of the partitioning estimator, kernel estimator and kNN estimator in the framework of this example. Figures 9 (b) – (d) shows the regression function emerging from each estimator with optimal chosen 'accuracy parameter' according to the $5-$fold cross validation procedure in Subsection 3.1. The corresponding Python files 'Example.3.2_PartEst.py', 'Example.3.2_KernelEst.py' and 'Example.3.2_kNNEst.py' can be found in [7].

ESTIMATORS	**Partitioning estimator**	**Kernel estimator**	**kNN estimator**
Opt. parameter via cross val.	$\kappa^* = 7$	$h^* = 0.1$	$k^* = 5$
Training error	≈ 0.077	≈ 0.055	≈ 0.056
Test error	≈ 0.112	≈ 0.093	≈ 0.11
Comp. time with cross val.	25 sec	57 sec	14 sec
Comp. time without cross val.	1.5 sec	3 sec	1 sec

Table 2: Comparison of the performance of the estimators in Example 3.2.

3.3 Handwritten digit recognition

In this subsection, we immediately tie in with the 'handwritten digit recognition–experiment' from the introduction, and present further details. In order to construct the (prediction) function f_n^* in (1.6), we first need a suitable data set. Here, we make use of the *MNIST* data set (*Modified National Institute of Standards and Technology database*), which provides (informations of) a collection of $n = 70000$ 28×28 greyscale (pixel) images of handwritten digits (*normalized* and *anti-aliased*; see [5, 6]) from 0 to 9; see Figure 10 (a). The data set $\mathbf{D}_n = \{(\tilde{\mathbf{x}}_i, y_i)\}_{i=1}^n$ is already divided into $n_{\mathrm{L}} = 60000$ learning/training data $\mathbf{D}_{n_{\mathrm{L}}} = \{(\tilde{\mathbf{x}}_i, y_i)\}_{i=1}^{n_{\mathrm{L}}}$ and $n_{\mathrm{T}} = 10000$ test data $\mathbf{D}_{n_{\mathrm{T}}} = \{(\tilde{\mathbf{x}}_i, y_i)\}_{i=n_{\mathrm{T}}+1}^{n_{\mathrm{T}}}$. Here, each $\tilde{\mathbf{x}}_i$ corresponds to a matrix in $\mathbb{R}^{28\times 28}$ with values between 0 and 255. We proceed as follows:



(a) Data

(b) Partitioning estimator

(c) Kernel estimator

(d) kNN estimator

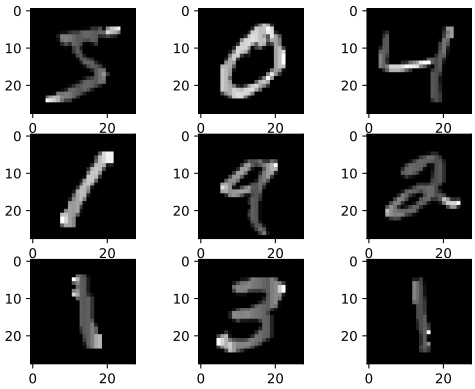
Figure 9: (a) Data generated via [7, DataGeneration2.py]. (b) Regression-function via partitioning estimator with $\kappa = 7$. (2c) Regression-function via kernel estimator with $h = 0.1$ and *Epanechnikov kernel*. (d) Regression-function via kNN estimator with $k = 5$.

1) **Prepare data set:** Via the Python file 'datasetPREP.py' in [7], we first scale down the entries in each matrix $\tilde{\mathbf{x}}_i$ to values in $[0,1]$: each entry of $\tilde{\mathbf{x}}_i$ (after the scale down procedure) corresponds to a pixel in the corresponding 28×28 greyscale image, where $0 \equiv$'black' and $1 \equiv$'white'. Secondly (after the scale down procedure), we transform each matrix $\tilde{\mathbf{x}}_i$ into a vector $\mathbf{x}_i$ with values in $[0,1]^{784}$. Then, via 'datasetPREP.py', the (prepared) data is saved (according to (1.1)) in the matrices $\texttt{TrainData} \in \mathbb{R}^{n_{\mathrm{L}} \times 785}$ and $\texttt{TestData} \in \mathbb{R}^{n_{\mathrm{T}} \times 785}$.

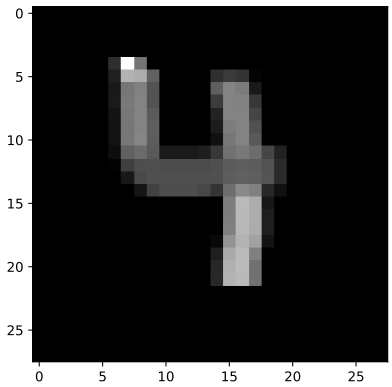
2) **Prepare input image:** We use the Python file 'Test_Imgage_PREP.py' in [7] to 'read' and to transform a given input/test image in 'jpg' or 'png' format with equal width and height (see Figure 1 (b)) into a 28×28 greyscale (pixel) image in 'png' format (see Figure 10 (b)). Then, via 'Test_Imgage_PREP.py', we prepare resp. extract informations from the new resulting image and save them in the vector $\texttt{testdigit} \in [0,1]^{784}$, which encodes these informations.

Note that it is crucial, that the given test image is prepared in such a way, that it is similar to the 'structure' of the digits in the *MNIST* data set. As one can see in Figure 10 (a), the digits are provided as 'white on black'; so the prepared test image has also to be represented in this way; see Figure 10 (b). An improper representation 'black on white' as seen in Figure 1 (d) might lead to troubles and spoiled predictions in the (upcoming) classification task!

3) **Prediction:** We now use the Python file 'kNN_digit_classifier.py' in [7], which predicts the digit/label of the original input image in Figure 1 (b); *i.e.*, encoded in the vector `testdigit`. In this connection, the kNN estimator is used to estimate the *a posteriori* probabilites in (1.5) by $m_{n_L}^{(i)}(\texttt{testdigit})$, where $i = 0, ..., 9$. Then, according to (1.6), 'kNN_digit_classifier.py' outputs the predicted label.



(a) Training examples of handwritten digits



(b) Prepared image via 'Test_Imgage_PREP.py'

Figure 10: (a) Digits from the *MNIST* data set. (b) Prepared image from Figure 1 (b).

References

[1] L. Devroye, L. Györfi, and G. Lugosi, *A probabilistic theory of pattern recognition*, volume 31 of *Applications of Mathematics (New York)*, Springer-Verlag New York (1996).

[2] T. Dunst, A. Prohl, *The forward-backward stochastic heat equation: numerical analysis and simulation*, *SIAM J. Sci. Comput.* **38**, (2016).

[3] L. Györfi, M. Kohler, A. Krzyzak and H. Walk, *A distribution-free theory of nonparametric regression*, *Springer Series in Statistics*, Springer-Verlag New York (2002).

[4] I. Steinwart, A. Christmann, *Support vector machines*, *Information Science and Statistics*, Springer, New York (2008).

[5] `https://en.wikipedia.org/wiki/Normalization_(image_processing)`

[6] https://en.wikipedia.org/wiki/Spatial_anti-aliasing

[7] https://github.com/Fab1Fatal