CodPy : a Python library for machine learning, mathematical finance, and statistics

Philippe G. LeFloch[1] , Jean-Marc Mercier, and Shohruh Miryusupov[2]

2022-01-18

[1]Laboratoire J.-L. Lions, Sorbonne Université and Centre National de la Recherche Scientifique, 4 Place Jussieu, 75258 Paris, France. Email:contact@philippelefloch.org

[2]MPG-Partners, 136 Boulevard Haussmann, 75008 Paris, France. Email:jean-marc.mercier@mpg-partners.com, shohruh.miryusupov@mpg-partners.com [3]

Contents

Chapter 1

Introduction

1.1 Main objective

This monograph presents the algorithms that are implemented in the Python library CodPy, an acronym that stands for "Curse of dimensionality in Python". This library provides the user with a support vector machine (or SVM in short) which is application-oriented in the sense that it provides a package of techniques relevant for numerous applications. The proposed algorithms apply to systems of partial differential equations arising in mathematical finance and fluid dynamics, as well discrete models of machine learning and statistics. We rely on a numerical strategy based on the theory of reproducing kernel Hilbert spaces (RKHS) which the authors have developed over the past decade, originally for applications in mathematical finance.

We proceed by presenting first, in several tutorial chapters, the basic notions of discretization, as we formulated them in CodPy, and we include elementary examples in order to illustrate the role of these main concepts. In a second part of this monograph, we apply our framework and include more sophisticated discretization techniques and numerical results, while covering applications in pattern recognition and mathematical finance. The proposed kernel engineering technique aims at formulating support vector machines in a way that makes it easy to adapt them to any particular problem. Our methodology encompasses the discretization of differential operators, which we naturally associate with any support vector machine. Indeed, discrete differential operators are building blocks in order to design discrete algorithms for partial differential equations, for instance those arising in fluid dynamics. Importantly, our metholology leads us to error bounds or quality tests, which are of crucial importance in many applications such as mathematical finance.

We found it convenient to write this monograph by relying on a combination of Python code, R code, and Latex code in order to generate a Jupyter notebook. This has led us to a document in which all numerical tests can be repeated and modified by the user[1].

1.2 Outline of this monograph

1.2.1 Aim of Chapter 2: a quick tour to machine learning

- In section **??**, we overview the techniques of learning machines, and introduce the general notation that will be in order for the rest of this monograph. We point out the links between this description and thz standard terminology used in the machine learning community, and we review the numerous methods available in this field. We thus discuss the notions relevant for

[1] CodPy will be made available for all users in the near future.

  - the description of numerical methods for machine learning,
  - the performance indicators that provide a measure to the relevance of any given learning machines, and
  - we mention the class of libraries currently available. It is not our purpose to cover all of the techniques, but to focus on kernel-based methods for machine learning and many other applications, and contribute here with several new aspects of the subject. For instance, discrete projection operators (see section **??**) and kernel-based clustering methods (see 3.6) are novel algorithms. As we also advocate here, the notions of discrepancy error (see (3.2.5)) and kernel-based norms (see Section (**??**)) leads us to performance indicators that are particularly efficient in the applications.

- In Section 2.3, we list a set of criteria that can be used to evaluate the performance of an algorithm and do not depend on the specific method in use. We can thus, with the help of the previous step, automate the benchmark of existing methods. Indeed, due to the vast amount of existing machine learning approaches, we encourage the reader to systematically benchmark them:
  - To a given learning problem, that is materialized as a list of input data.
  - Pick a list of scenarios, a list of learning machines, and a list of performance indicators.
  - Run the tests, output and compare performance indicators.
- The two previous steps allowed us to implement a framework into which we can plug-in a quite large zoo of learning machines, to illustrate numerically with simple, one or two dimensional, examples, our purposes. The section **??** (resp. **??**) contains examples and illustrations for supervised learning (resp. unsupervised) benchmarks.

1.2.2 Aim of Chapter 3

1.2.3 Aim of Chapter 4

1.2.4 Aim of Chapter 5

1.2.5 Aim of Chapter 6

……

1.3 Further references

Since our primarily intention here is to provide a technical introduction to our Python library, we only include a brief bibliography here. While a large literature is available which is devoted to support vector machines and reproducing kernel Hilbert spaces (RKHS), it is not our purpose to review it here. We would like refer the reader to Berlinet and Thomas-Agnan [3] and Fasshauer [10, 11, 12] since they were the most influential in the development of the present code. The reader will find therein a background on the subject, together with many further references.

Our original contributions concerning the class of kernel-based mesh-free algorithms presented in this monograph can be found in the research papers by LeFloch and Mercier [23, 24, 25, 26, 27]. Moreover, [28]–[33]
contain earlier versions of the material in this monograph.

For additional results on kernel techniques, we refer to \cite{NarcowichWardWendland:2005,Niederreiter:1992,Opfer:2006,
Mesh-less methods and kernel-based strategies have been found very useful in fluid dynamics and material dynamics [2, 4, 14, 15, 18, 34, 36, 38, 41, 43, 45, 51].

Chapter 2

Brief overview of methods of machine learning

2.1 A framework for machine learning

2.1.1 Prediction machine for supervised/unsupervised machine learning

Machine learning methods can be roughly split into two main approaches: unsupervised and supervised methods. Both can be described in a general framework, referred to here as a **prediction machine**. In short, a predictor, denoted by $\mathcal{P}_m$, is an extrapolation or interpolation procedure, described by an operator

$$f_z = \mathcal{P}_m(x, y = [], z = x, f(x) = []). \tag{2.1.1}$$

Python notation is used here and the brakets mean that the variables $y, z, f(x)$ are optional input data.

- The choice of the method is indicated by the subscript m. Each method relies on a set of **external parameters**. Fine tuning such parameters is sometimes very cumbersome and provide a source of error and, in fact, some of the strategies in the literature propose to rely on a learning machine in order to determine these external parameters. No performance indicator is provided for this parameter tuning step, and this is an issue to take into account in the applications before selecting up a particular method.
- The input data $x, y, z, f(x)$ are as follows.
  - The only non-optional parameter is the variable $x \in \mathbb{R}^{N_x \times D}$, called the **training set**. The parameter D is usually referred as the **total number of features**.
  - The variable $f(x) \in \mathbb{R}^{N_x \times D_f}$ is called the **training set values**, whilethe parameter D_f is the **number of training features**.
  - The variable $z \in \mathbb{R}^{N_z \times D}$ is called the **test set**. If it is not specified, we tacitly assume that $z = x$.
  - The variable $y \in \mathbb{R}^{N_y \times D}$ is called the **internal parameter set**[1] and is necessary in order to define $\mathcal{P}_m$.
- The output data are as follow:
  - **Supervised learning**: this corresponds to choosing the input function values $f(x)$ and we then write

$$f_z = \mathcal{P}_m(x, y = [], z = x, f(x)),$$

[1] also called weight set in neural network theory

where the values $f_z \in \mathbb{R}^{N_z \times D}$ are called a **prediction**. We distinguish between two cases:

* If the input data y is left empty, then the prediction machine (2.1.1) is called a **feed-backward machine**. In this case, the method computes this set with an internal method and determine f_z.
* If y is specified as input data, then the prediction machine (2.1.1) is referred as a **feed-forward machine**. In this case, the method uses the set of internal parameters and compute the prediction f_z.

– **Unsupervised learning**: we may also choose

$$f_z = \mathcal{P}_m(x, z = x), \tag{2.1.2}$$

where the output values $f_z \in \mathbb{R}^{N_z \times D}$ are sometimes called **clusters** for the so-called clustering methods (described later on).

Other machine learning methods can be described with the same notation. For instance, two methods m_1, m_2 begin given, then the following composition describes a feed-backward machine, which is quite close to the definition of **semi-supervised learning** in the literature and also encompasses feed-backward learning machines:

$$f_z = \mathcal{P}_{m_1}(x, \mathcal{P}_{m_2}(x, f(x)), z, f(x)),$$

We summarize our main notation in Table 2.1. The sizes of the input data, that is, the integers D, N_x, N_y, N_z, D_f, are also considered as input parameters. The distinction between supervised and unsupervised learning is a matter of having, or not, optional input data and the correspondence will be clarified in the rest of this chapter.

Table 2.1: Main parameters for machine learning

x	y	z	$f(x)$	f_z
training set size $N_x \times D$	parameter set size $N_y \times D$	test set size $N_z \times D$	training values size $N_x \times Df$	predictions size $N_z \times Df$

2.1.2 Techniques of supervised learning

Supervised learning (2.1.1) corresponds to the case where the function values $f(x)$ is part of input data.

$$f_z = \mathcal{P}_m(x, y = [], z = x, f(x)). \tag{2.1.3}$$

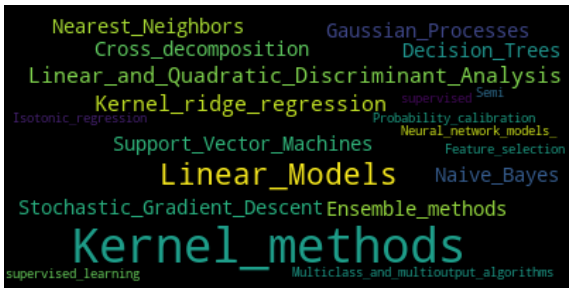
Supervised learning can be best understood as a simple extrapolation procedure: from historical observations of a given function $x, f(x)$, one wants to predict, or extrapolate, the function on a new set of values z. Concerning the terminology, a method is said to be **multi-class** or multi-output if the function f under consideration can be vector-valued, that is, $D_f \geq 1$ with our notations. Note that one can always stack learning machines to produce multi-class methods. However, this comes usually at a quite heavy computational cost, motivating this definition. Moreover, the input function f can be

- discrete, that is the set of unique values $f(\mathbb{R}^D)$ is a discrete set, denoted $Ran(f)$. The set is referred as **labels**, and this set can always be mapped to integer $[1, \dots, \#(Ran(f))]$, where $\#(E)$ denotes the number of elements, or cardinal, of a set.
- continuous.
- mixed (some discrete, some continuous).

A classification of existing methods for supervised learning can be found at scikit-learn

There are

- Different family of methods: linear models, support vector machines, neural networks, ...



- Different methods: neural networks, gaussian processes, etc...

- Different libraries: scikit-learn, Tensorflow, ...

2.1.3 Techniques of unsupervised learning

Unsupervised learning corresponds to the case where the function values $f(x)$ is not part of input data, see (2.1.1) :

$$\mathcal{P}_m(x, y = [], z = x). \tag{2.1.4}$$

Unsupervised learning can be best understood as a simple interpolation procedure: from historical observations of a given distribution x, one wants to extract, or interpolate, N_y features that best represent x. The output data of a standard clustering method are the **cluster set**, denoted $y \in \mathbb{R}^{N_y \times D}$.

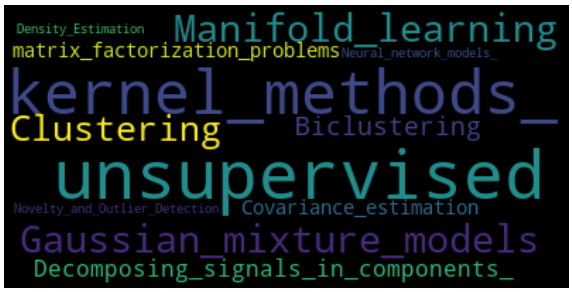
There are natural connections between supervised and unsupervised learning.

- In the context of semi-supervised clustering methods, the clusters y are used in a supervised learning machine to produce a prediction $f_z \in \mathbb{R}^{N_z \times D_f}$, see (2.1.1).
- In the context of unsupervised clustering methods, a prediction $f_z \in \mathbb{R}^{N_z}$ can also be made. This prediction attaches each point z^i of the test set to the cluster set y, producing f_z as a map $[1, \dots, N_z] \mapsto [1, \dots, N_y]$.

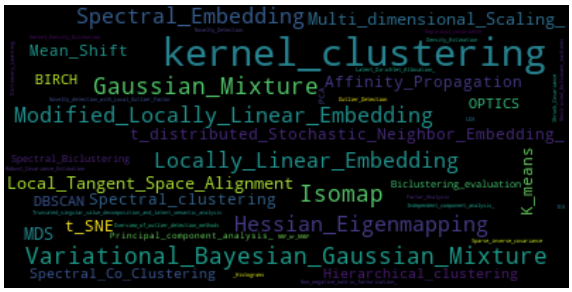
There exists several clustering methods performing this approach, see for instance the dedicated Wikipedia page[2].

- Different family of methods: linear models, support vector machines, neural networks,...

[2] link to cluster analysis Wikipedia page.



- Different methods: neural networks, Gaussian processes, etc..



- Different libraries: Scikit-learn, ...

Clustering is one family of unsupervised learning method. The library Scikit-learn proposes this quite impressive list of clustering methods, see [3]. We extracted the following figure and comment it briefly to illustrate our notation.

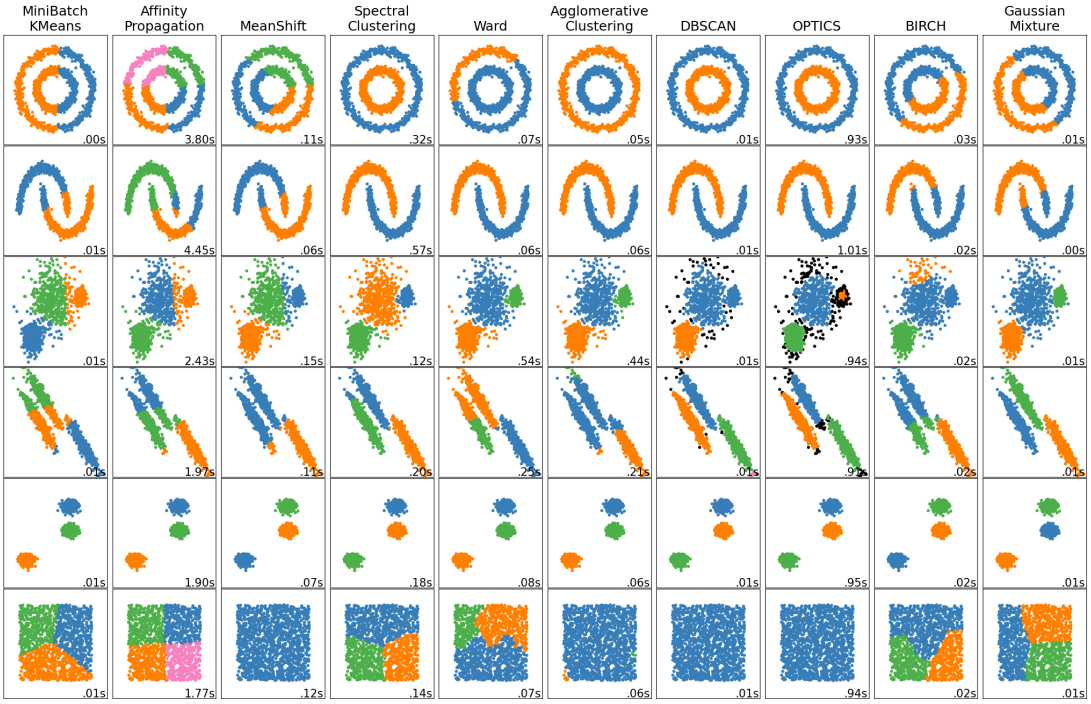


Figure 2.1: list of scikit-learn clustering methods.

- Each column describes a particular clustering algorithm.

[3] link to scikit-learn clustering

- Each row describes a particular clustering, unsupervised problem:
  - Each image scatter plots the training set x and the test set z, that are equals.
  - Each image color codes the predicted values f_z.

2.2 Exploratory data analysis

2.2.1 Preliminaries

Exploratory data analysis plays a central role in data engineering and allows one to understand the structure of a given dataset, including its correlation and statistical properties. For instance, we can study whether a data distribution is multi-modal, skew, or discontinuous, among other features. The technique can help in many different applications and, for instance in unsupervised learning, one can produce a first guess concerning the number of possible clusters associated with a given dataset, or concerning the type of kernels one should choose before applying a kernel regression method.

As an example, we illustrate the visualization tools that we are using, consider the Iris flower data set. Iris data set introduced by the British statistician, eugenicist, and biologist Ronald Fisher in his 1936 paper "The use of multiple measurements in taxonomic problems". The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

2.2.2 Visualization based on non-parametric estimations

The density of the input data is estimated using a kernel density estimate (KDE). Let $(x^1, x^2, \dots, x^n)$ be independent and identically distributed samples, drawn from some univariate distribution with unknown density denoted by f at any given point x. We are interested in estimating the shape of this function f and the kernel density estimator is

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x^i) = \frac{1}{nh}\sum_{i=1}^{n} K\Big(\frac{x - x^i}{h}\Big),$$

where K is a kernel (say any non-negative function) and $h > 0$ is a smoothing parameter called the **bandwidth**. Among the range of possible kernels that are are commonly used, we have: uniform, triangular, biweight, triweight, Epanechnikov, normal, and many others. The ability of the KDE to accurately represent the data depends on the choice of the smoothing bandwidth. An over-smoothed estimate can remove meaningful features, but an under-smoothed estimate can obscure the true shape within the random noise.

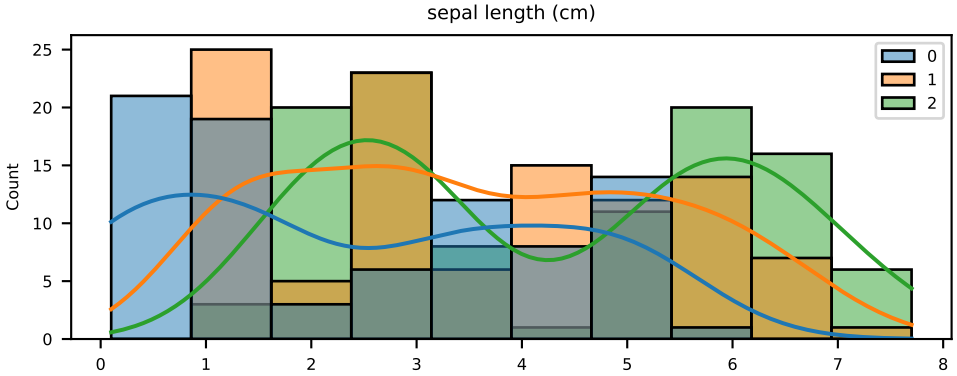


Figure 2.2: Kernel density estimator

2.2.3 Visualization based on scatter plots

Another way to visualize data is to rely on a scatter plot, where the data are displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis.

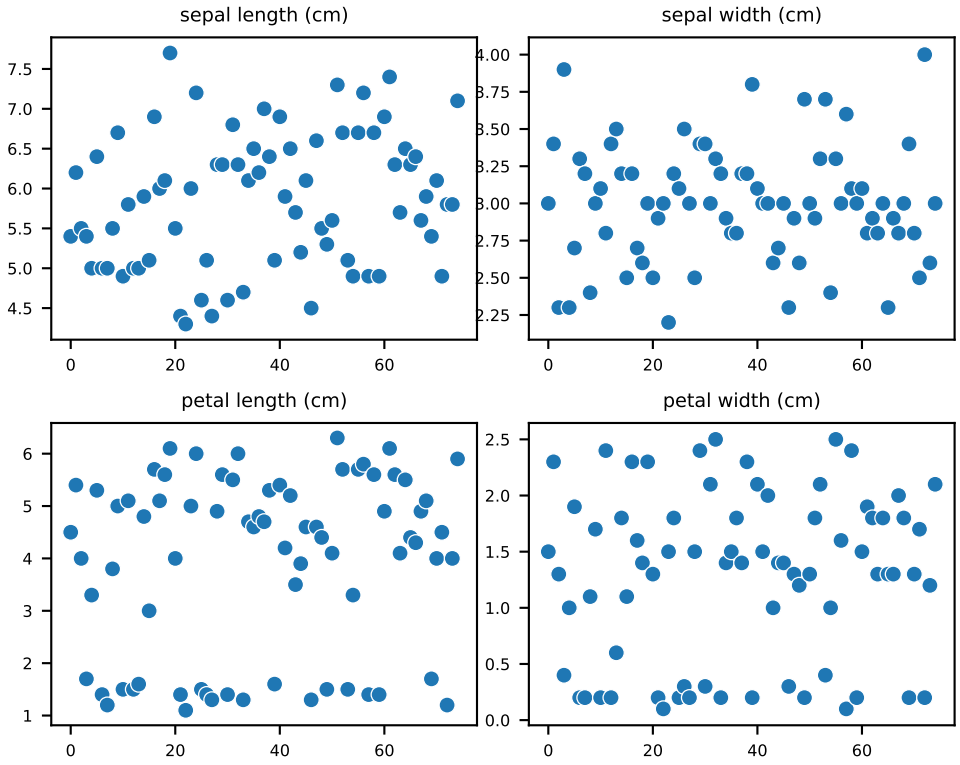


Figure 2.3: Scatter plot

2.2.4 Visualization based on correlation matrices

The correlation matrix of n random variables $x^1, \ldots, x^n$ is the $n \times n$ matrix whose (i, j) entry is $corr(x^i, x^j)$. Thus the diagonal entries are all identically unity.

2.2.5 Visualization based on summary plots

The summary plot visualizes the density of each feature of the data on the diagonal. The KDE plot on the lower diagonal and the scatter plot on the upper diagonal.

2.3 Performance indicators for machine learning

2.3.1 Indicators for supervised learning

Comparison to ground truth values. A huge family of indicators is available in order to evaluate the performance of a learning machine, most of them being readily described and implemented in scikit-learn[4].

We do not discuss them all, but rather overview those that we have included in the CodPy library. First of all, in the context of supervised clustering methods, if the function f is known in

[4]link to scikit-learn metrics.

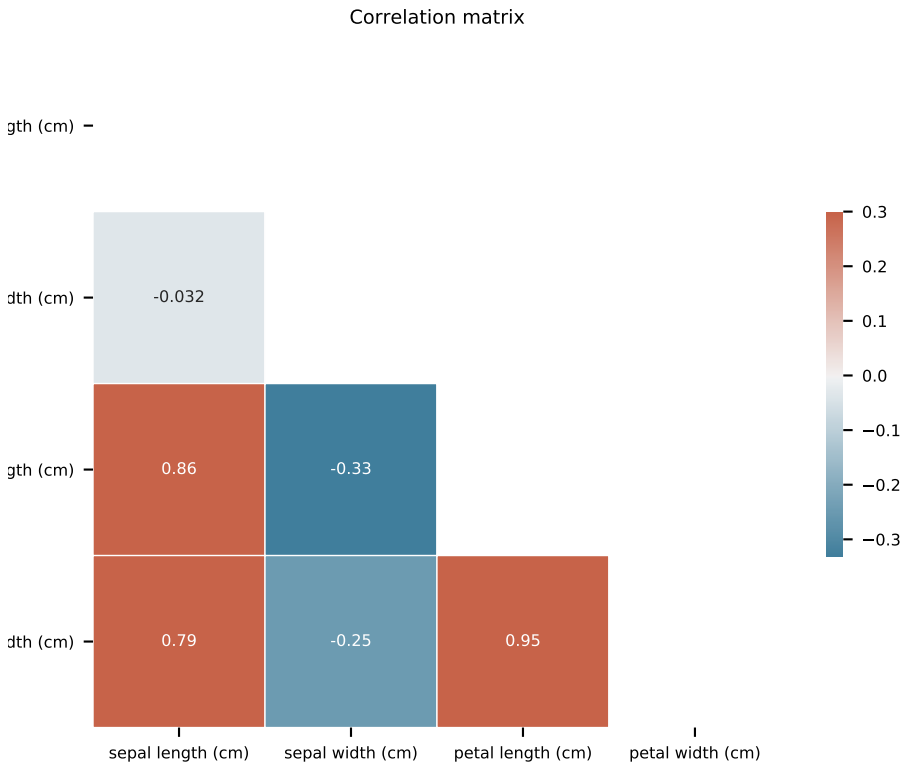


Figure 2.4: Correlation matrix

advance, then predictions of learning machines f_z can be compared with **ground truth values**, $f(z) \in \mathbb{R}^{N_z \times D_f}$. Below we list the main metrics that are used.

- For labeled functions (i.e., discrete functions), a common indicator is the **score**, defined as

$$\frac{1}{N_z} \#\{f_z^n = f(z)^n, n = 1 \dots N_z\}$$

  producing an indicator between 0 and 1, the higher being the better.

- For continuous functions (i.e., discrete functions), a common indicator is ℓ^p norms, defined as

$$\frac{1}{N_z} \|f_z - f(z)\|_{\ell^p}, \ 1 \leq p \leq \infty.$$

  the case $p = 2$ is referred as the *root-mean-square error (RMSE).*

- As the above indicator is not normalized, the following version is preferred.

$$\frac{\|f_z - f(z)\|_{\ell^p}}{\|f_z\|_{\ell^p} + \|f(z)\|_{\ell^p}}, \ 1 \leq p \leq \infty.$$

  producing an indicator between 0 and 1, the smaller being the better, interpreted as error-percentages. In finance, this notion is sometimes referred to as the basis point indicator.

Cross validation scores. The cross validation score consists in randomly selecting a part of the training set and values as test set and values, and to perform a score or RMSE type error analysis on each run[5]

[5]see the dedicated page on scikit-learn.

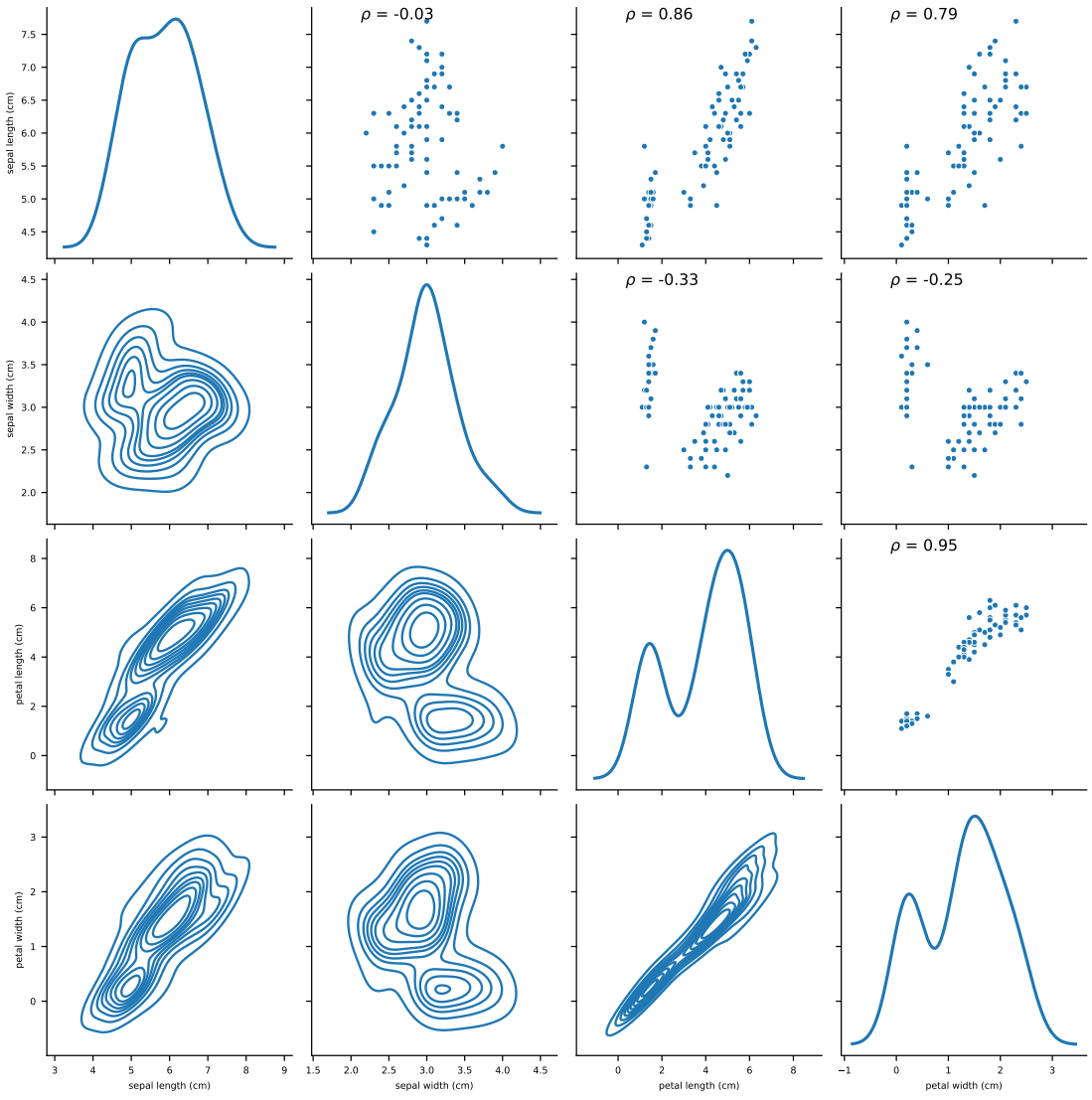


Figure 2.5: Summary plot

Confusion matrix. This indicator is available for labeled, supervised learning, is a matrix representation of the numbers of ground-truth labels in a row, while each column represents the predicted labels in an actual class. Confusion matrix is a quite simple and efficient data error visualization methods, a simple example is shown in Section **??**. Its common form is

$$M(i,j) = \#\{f(z) = i \quad and \quad f_z = j\},$$

representing correct predicted numbers in the matrix diagonal, since off-diagonal elements counts false positive predictions. Note that numerous others performance indicators can be straightforwardly deduced from the confusion matrix, as Rand Index, Fowlkes-Mallows scores, etc…

Norm of output. If no ground truth values are known, the quality of the prediction f_z, depends on **a priori error estimates** or error bounds. Such estimates exist only for kernel methods (to the best of the knowledge of the authors), and are described in the next chapter, see (3.2.5). Such estimates uses the norm of functions described in (**??**), and was proven to be a useful indicator in the applications.

ROC curves. A receiver operating characteristic curve, or ROC curve, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The method was originally developed for operators of military radar receivers starting in 1941, which led to its name.

ROC is the plot of TPR versus FPR by varying the threshold. These metrics are are summed up in the table below:

Metric	Formula	Equivalent
True Positive Rate TPR	$\frac{TP}{TP+FN}$	Recall, sensitivity
False Positive Rate FPR	$\frac{FP}{TN+FP}$	1-specificity

We can use precision score (PRE) to measure the performance across all classes:

$$PRE = \frac{TP}{TP + FP}.$$

In "micro averaging", we calculate the performance, e.g., precision, from the individual true positives, true negatives, false positives, and false negatives of the the k-class model:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}.$$

And in macro-averaging, we average the performances of each individual class

$$PRE_{marco} = \frac{PRE_1 + \cdots + PRE_k}{k}.$$

2.3.2 Indicators for unsupervised learning

Discrepancy error associated to kernel. Evaluation of clustering algorithms benefits from a lot of performance indicators, a lot of them being implemented in Scikit-learn [ˆ206] [ˆ206]:see this link.

We list in this section those that we are computing. First of all, the discrepancy error is an indicator based on a kernel and will be fully described in the next chapter, see (3.2.5). It is used primarily to produce worst error estimates, together with the norm of functions, as described in (**??**). It was also found to be useful as a performance indicator for unsupervised learning machine.

Inertia indicator. The inertia indicator is used for *k-means* type algorithms. We describe it precisely, as it uses a notation that will be used in other parts. It shares some similarities with the

discrepancy error one but is not equivalent. To define inertia, one first pick a distance, denoted $d(x, y)$, as the squared Euclidean one, although other distance are considered, as the Manhattan one or log-entropy, depending upon the problem under consideration. Consider now any point $w \in \mathbb{R}^D$. Then w is attached naturally to a point $y^{\sigma_d(w,y)}$, where the discrete function $\sigma_d(w, y)$ is computed as

$$\sigma_d(w, y) := \{j : d(w, y^j) = \inf_k d(w, y^k)\}.$$

Then the inertia is defined as

$$I(x, y) = \sum_{n=0}^{N_x} (|x^n - y^{\sigma_d(x^n,y)}|^2).$$

Observe that this functional might not be convex, even if the distance under consideration is convex, as is the squared Euclidean distance. For k-means algorithms, the cluster centers y are computed minimizing this functional. The parameter set y is called **centroids** for k-means algorithms.

Homogeneity score. The homogeneity score, see the dedicated scikit-learn for a definition [ˆ492], is a performance indicator that holds for supervised, labeled, clustering problems. This indicator performs a conditional entropy to estimate a score $s(f(z), f_z)$ between 0 and 1 - higher the better. [ˆ492]:see this link

Silhouette coefficient. If the ground truth labels are not known, evaluation must be performed using the model itself. The Silhouette Coefficient is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters.

2.4 General specification of tests

2.4.1 Preliminaries

We now overview a benchmark methodology and apply it to a few methods of supervised learning. For each machine, * we illustrate the prediction function $\mathcal{P}_m$, and * we illustrate the computation of some performance indicators. We then present benchmarks using these indicators. In this section, we restrict attention to toy examples while more significant examples will be studied in Chapter 5.

We begin by describing a general, multi-dimensional, first quality assurance test for supervised learning machines. We illustrate this test framework with one and two-dimensional examples, and the reader can toy with functions and methods. The goal of this framework is to measure accuracy of any learning machines, while using the extrapolation operator . Hence all our unit tests are based on the following input sizes:

$$\text{a function: f , a method: m , five integers: } D, N_x, N_y, N_z, D_f$$

To benchmark our machine, we use a list of scenarios, that is a list of entries D, N_x, N_y, N_z, D_f. Table 2.3 is an example of a list of 5 scenarios.

Table 2.3: scenario list

D	N_x	N_y	N_z
1	100	100	100
1	200	200	200
1	300	300	300
1	400	400	400

For the function f we choose a period and an increasing function:

$$f(x) = \Pi_{d=1..D} \cos(4\pi x_d) + \sum_{d=1..D} x_d. \tag{2.4.1}$$

It is defined in python code of this document, and the reader can change it to any other continuous function.

2.4.2 An example in one dimension

Initialization. For this tutorial, we used a generator, configured to select x (resp. y, z) as N_x (resp. N_y, N_z) points regularly (resp. randomly, regularly) generated on a unit cube. We chose to select z distributed over a larger cube, to observe extrapolation and interpolation effects.

As an illustration, in Figure 2.6 we show both graphs $(x, f(x))$ (left, training set),$(z, f(z))$ (right, test set).

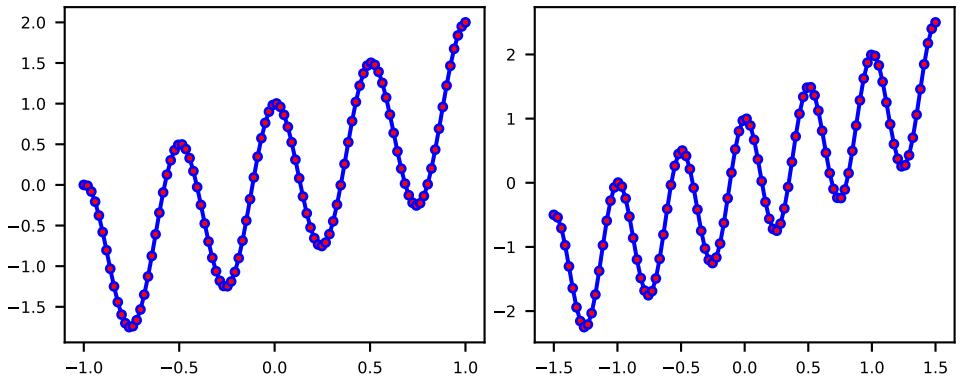


Figure 2.6: training and test set.

2.5 Benchmark methodology: kernel-based predictors

2.5.1 Periodic kernel regression model from CodPy

This test illustrates a kernel-based projection operator, described in Section 3.2. The set of external parameters for kernel-based methods consists simply in picking-up a kernel, and is discussed in the next chapter; see Section 3. We pick-up in the corresponding python chunk a standard periodic Gaussian kernel, with a linear regression kernel, allowing us to fit both periodic and polynomial parts of these data. These settings are explained in Chapter 3.3.

```
set_per_kernel = kernel_setters.kernel_helper(kernel_setters.set_gaussianper_kernel,2,1e-8,None)
```

We then run all the scenarios in Section 2.3.

We plot the first two results of this test in Figure 2.7 : predictions, denoted f_z of the function $f(z)$, see Figure 2.6, for the first two scenarios defined in Section 2.3.

Table 2.4 shows the computed indicators during this test.

Table 2.4: CodPy performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
codpy extra	1	100	100	100	1	0.02	0.0371	0.90	0.0498
codpy extra	1	200	200	200	1	0.01	0.0184	0.95	0.1155
codpy extra	1	300	300	300	1	0.03	0.0122	0.92	0.1194
codpy extra	1	400	400	400	1	0.05	0.0091	0.92	0.0928

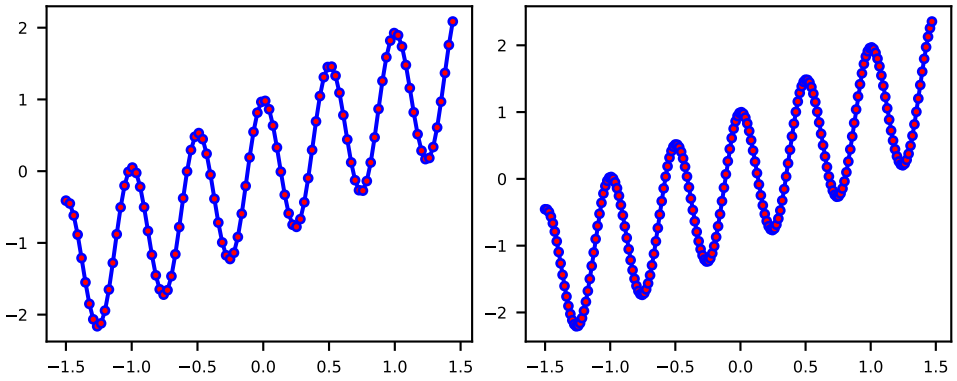


Figure 2.7: periodical kernel : two predictions.

2.5.2 The kernel regression model from SciPy

Scipy proposes a solid and robust kernel regression predictor, see this link. We often benchmark our kernel implementation with it. Let us first set up the external parameters for Scipy.

```
rbf_param = {'function': 'gaussian', 'epsilon':None, 'smooth':1e-8, 'norm':'euclidean'}
```

Indeed, we now proceed by copy-pasting the previous section, to highlight that benchmark methodologies should be method-independent. We then run our scenario list and collect results.

We plot the two first results in Figure 2.8 : these are the predictions, denoted f_z, of the function $f(z)$; see Figure 2.6, for the first two scenarios defined in Section 2.3.

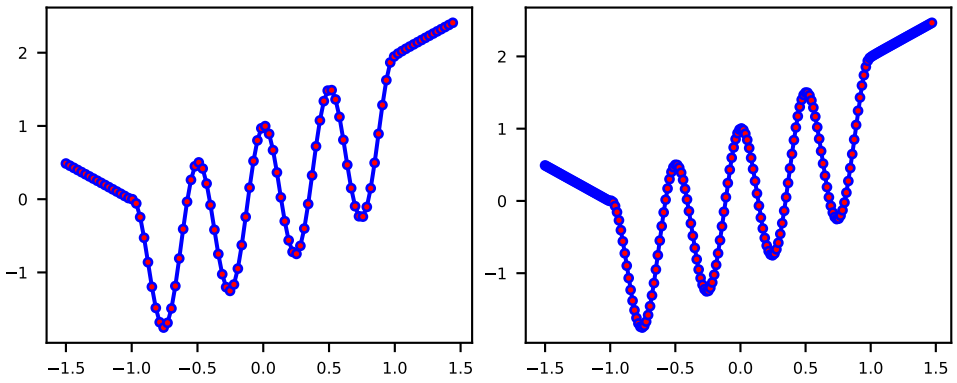


Figure 2.8: scipy : two predictions.

Table 2.5 shows the computed indicators after running all scenarios indicated in the Table 2.3.

Table 2.5: scipy performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
scipy pred	1	100	100	100	1	0.00	0.5532	0.90	0.0498
scipy pred	1	200	200	200	1	0.19	0.5464	0.95	0.1155
scipy pred	1	300	300	300	1	0.19	0.5434	0.92	0.1194
scipy pred	1	400	400	400	1	0.18	0.5419	0.92	0.0928

2.5.3 Support vector regression model

For this test, the interpolation machine is chosen to be a support vector classifier, taken from scikit learn. It specified by a decision function (support vector classifier) and the kernel function associated to it, see this dedicated page for a description of SVC. The reader can tune this set of parameters.

```
svm_param = {'kernel': 'linear', 'gamma': 'auto', 'C': 1}
```

Figure 2.9 shows the results of the first two scenarios of this test.

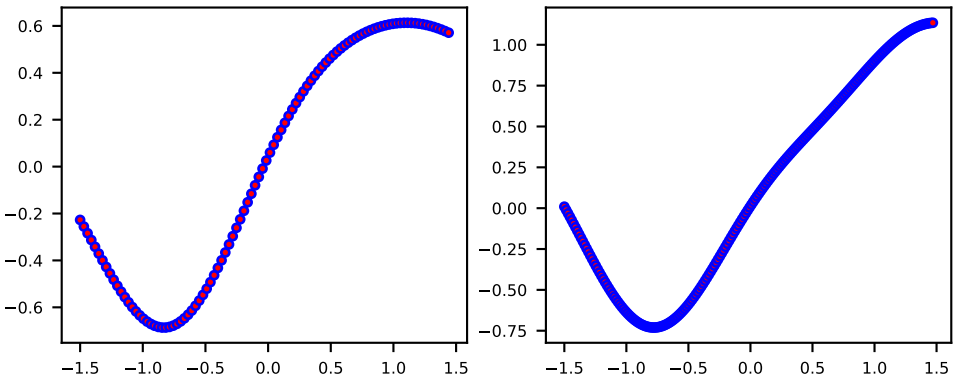


Figure 2.9: SVM

Table 2.6 provides all computed indicators after running all scenarios indicated in the Table 2.3.

Table 2.6: SVM performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
SVM	1	100	100	100	1	0.03	0.5286	0.90	0.0498
SVM	1	200	200	200	1	0.00	0.5201	0.95	0.1155
SVM	1	300	300	300	1	0.00	0.5304	0.92	0.1194
SVM	1	400	400	400	1	0.00	0.5498	0.92	0.0928

2.6 Benchmark methodology: neural network predictors

2.6.1 TensorFlow neural network regression model

For this test, we use as an interpolation machine a standard neural network one, taken from TensorFlow, commonly called **deep learning** method. It consists in a network of *layers* defined by the following settings, see this dedicated page for a description of TensorFlow neural networks. The reader can tune this set of parameters:

```
import tensorflow as tf
codpy_param['tfRegressor'] = {'epochs': 50,
'batch_size':16,
'validation_split':0.1,
'loss':tf.keras.losses.mean_squared_error,
'optimizer':tf.keras.optimizers.Adam(0.001),
'layers':[8,64,64,1],
'activation':['relu','relu','relu','linear'],
'metrics':['mse']}
```

We then run the scenarios. We plot the two first results of this test in Figure 2.10 : these are the predictions, denoted f_z, of the function $f(z)$; see figure 2.6, for the first two scenarios defined in Table 2.3.

```
## WARNING:tensorflow:From C:\informatique\Python37\lib\site-packages\tensorflow_core\python\ops\res
## Instructions for updating:
## If using Keras pass *_constraint arguments to layers.
```

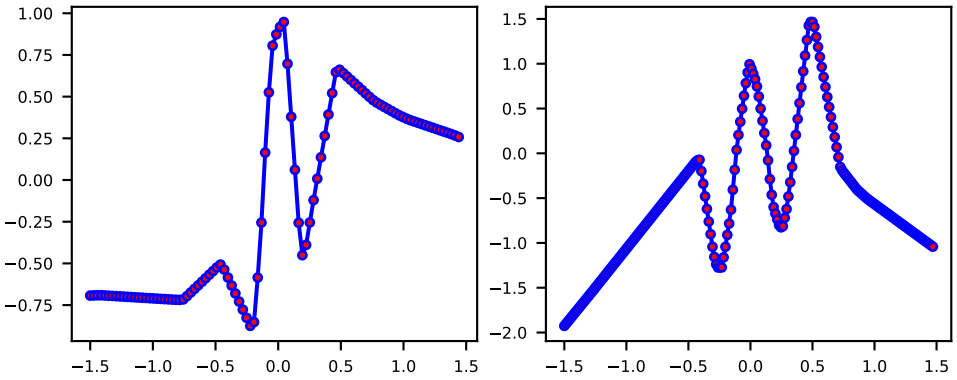


Figure 2.10: TensorFlow : two predictions.

The table 2.7 shows computed indicators after running all scenarios indicated in Table 2.3.

Table 2.7: Tensorflow neural network performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
Tensorflow	1	100	100	100	1	0.70	0.4807	0.90	0.0498
Tensorflow	1	200	200	200	1	1.08	0.6811	0.95	0.1155
Tensorflow	1	300	300	300	1	1.46	0.9161	0.92	0.1194
Tensorflow	1	400	400	400	1	1.80	1.0506	0.92	0.0928

2.6.2 Pytorch neural network regression model

For this test, we use as interpolation machine a standard neural network one, taken from Pytorch. It consists in a network of *layers* defined by the following settings, see this dedicated page for a description of Pytorch neural networks. We constructed the same neural network as in the case of Tensorflow.

```
torch_param = {'PytorchRegressor': {'epochs': 128,
'layers': [8,64,64],
'activation':['relu','linear'],
'batch_size': 16,
'loss': nn.MSELoss(),
'activation': nn.ReLU(),
'optimizer': torch.optim.Adam,
"out_layer": 1}}
```

Figure 2.11 shows the results of first two scenarios of this test.

We run the scenarios and output the results: Table 2.8 provides all computed indicators after running all scenarios indicated in Table 2.3.

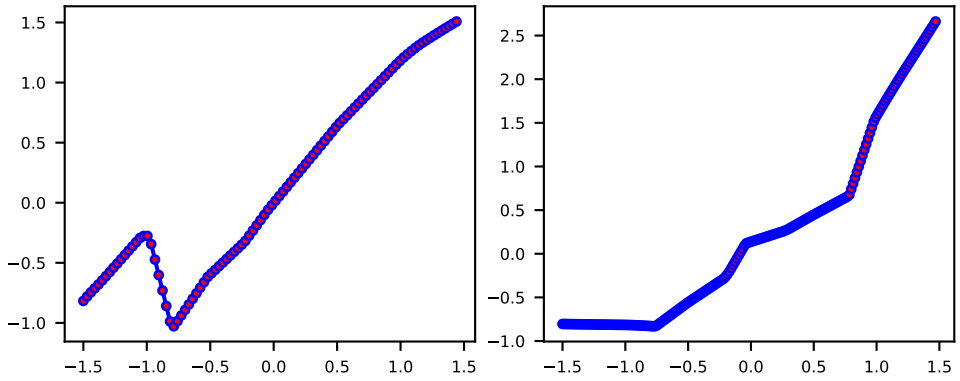


Figure 2.11: Pytorch

Table 2.8: Pytorch performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
Pytorch	1	100	100	100	1	0.59	0.4877	0.90	0.0498
Pytorch	1	200	200	200	1	1.19	0.5156	0.95	0.1155
Pytorch	1	300	300	300	1	1.72	0.7986	0.92	0.1194
Pytorch	1	400	400	400	1	2.24	0.7964	0.92	0.0928

2.7 Benchmark methodology: regression-tree predictors

2.7.1 Decision tree regression

We use as interpolation machine a decision tree, taken from scikit learn. It allows to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation; see this dedicated page for a description of decision trees. (The reader can tune this set of parameters).

```
DT_param = {'max_depth': 10}
```

Figure 2.12 shows the results of the first two scenarios of this test.

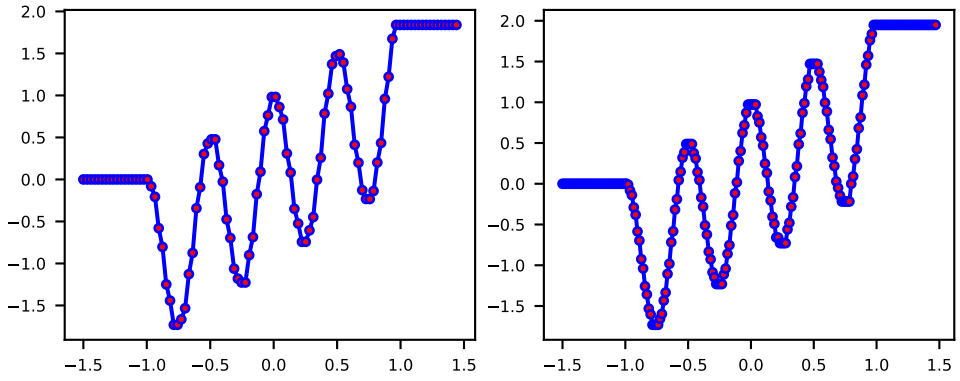


Figure 2.12: Decision Tree

Table 2.9 provides all computed indicators after running all scenarios indicated in Table 2.3.

Table 2.9: Decision Tree performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
Decision tree	1	100	100	100	1	0.03	0.4586	0.90	0.0498
Decision tree	1	200	200	200	1	0.00	0.4621	0.95	0.1155
Decision tree	1	300	300	300	1	0.00	0.4619	0.92	0.1194
Decision tree	1	400	400	400	1	0.00	0.4616	0.92	0.0928

2.7.2 AdaBoost regression

Now, for the interpolation machine we use an AdaBoost algorithm, taken from scikit learn. The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction, see this dedicated page for a description of AdaBoost algorithm. The reader can tune this set of parameters.

```
ada_param = {'tree_no': 50, 'learning_rate': 1}
```

Figure 2.13 shows the results of the first two scenarios of this test.

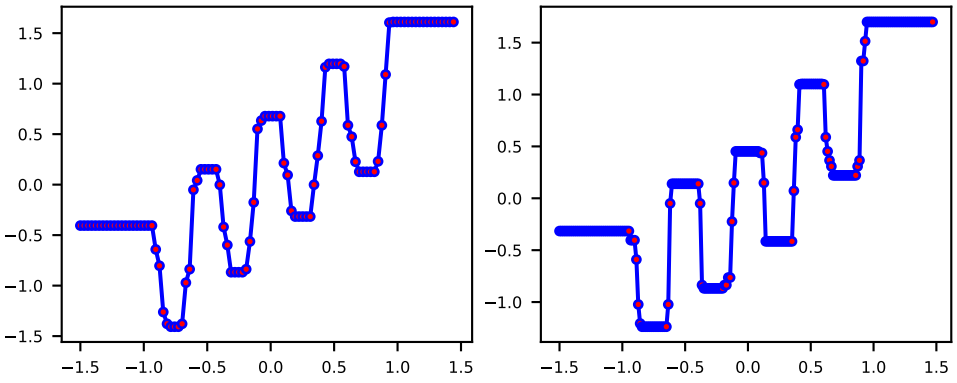


Figure 2.13: AdaBoost

Table 2.10 provides all computed indicators after running all scenarios indicated in Table 2.3.

Table 2.10: AdaBoost performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
AdaBoost	1	100	100	100	1	0.13	0.3814	0.90	0.0498
AdaBoost	1	200	200	200	1	0.01	0.4093	0.95	0.1155
AdaBoost	1	300	300	300	1	0.02	0.4154	0.92	0.1194
AdaBoost	1	400	400	400	1	0.03	0.4169	0.92	0.0928

2.7.3 Gradient boosting regression

For this test, we use as interpolation machine a gradient decision tree boosting (GBDT), taken from scikit learn. It allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function; see this dedicated page for a description of Gradient Tree Boosting. (The reader can tune this set of parameters.)

```
gb_param = {'tree_no': 50, 'learning_rate': 1}
```

Figure 2.14 shows the results of the first two scenarios of this test.

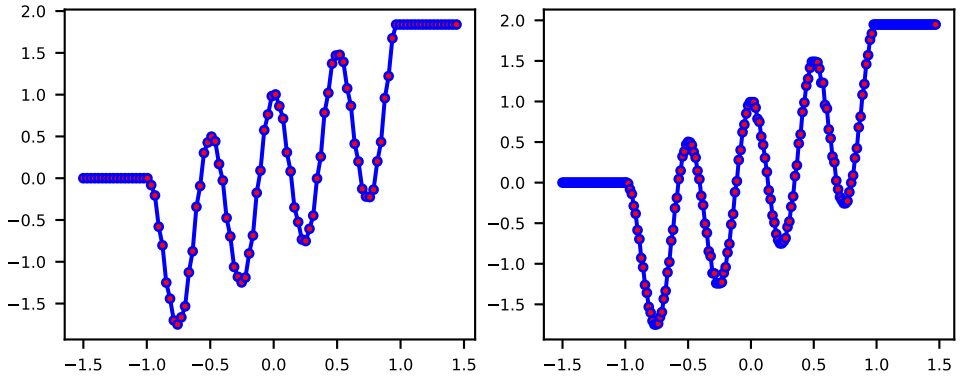


Figure 2.14: Gradient Boosting

Table 2.11 provides all computed indicators after running all scenarios indicated in Table 2.3.

Table 2.11: Gradient Boosting performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
Gradient Boosting	1	100	100	100	1	0.01	0.4586	0.90	0.0498
Gradient Boosting	1	200	200	200	1	0.01	0.4620	0.95	0.1155
Gradient Boosting	1	300	300	300	1	0.02	0.4621	0.92	0.1194
Gradient Boosting	1	400	400	400	1	0.01	0.4611	0.92	0.0928

2.7.4 XGBoost algorithm

For this test, we use as XGBoost as an interpolation machine. It is essentially a computationally efficient implementation of the original gradient boost algorithm, see this dedicated page for a description of XGBoost project. (The reader can tune this set of parameters.)

```
xgb_param = {'max_depth': 5, 'n_estimators': 10}
```

Figure 2.15 shows the results of the first two scenarios of this test.

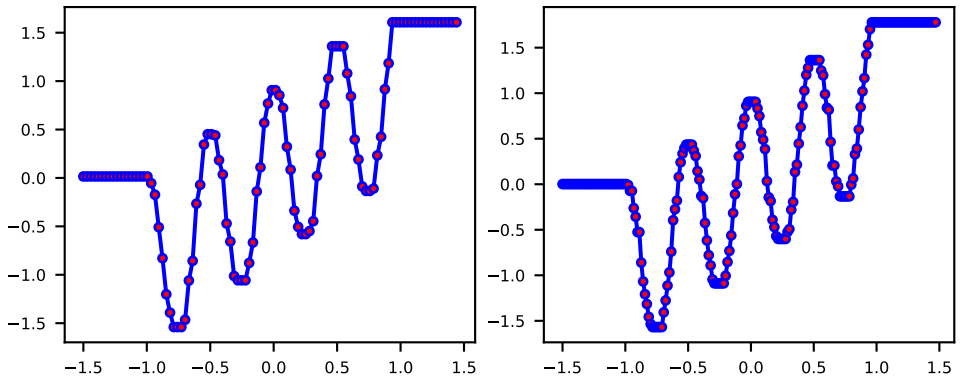


Figure 2.15: XGBoost

Table 2.12 provides all computed indicators after running all scenarios indicated in Table 2.3.

Table 2.12: XGBoost performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
XGboost	1	100	100	100	1	0.16	0.4442	0.90	0.0498
XGboost	1	200	200	200	1	0.01	0.4473	0.95	0.1155
XGboost	1	300	300	300	1	0.01	0.4502	0.92	0.1194
XGboost	1	400	400	400	1	0.01	0.4522	0.92	0.0928

2.7.5 Random forest regression

For this test, as an interpolation machine we use a random forest regression. It operates by constructing a large number of decision trees at training time and producing the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees; see this dedicated page for a description of forests of randomized trees. (The reader can tune this set of parameters.)

```
RF_param = {'max_depth': 5, 'n_estimators': 5}
```

Figure 2.16 shows the results of first two scenarios of this test.

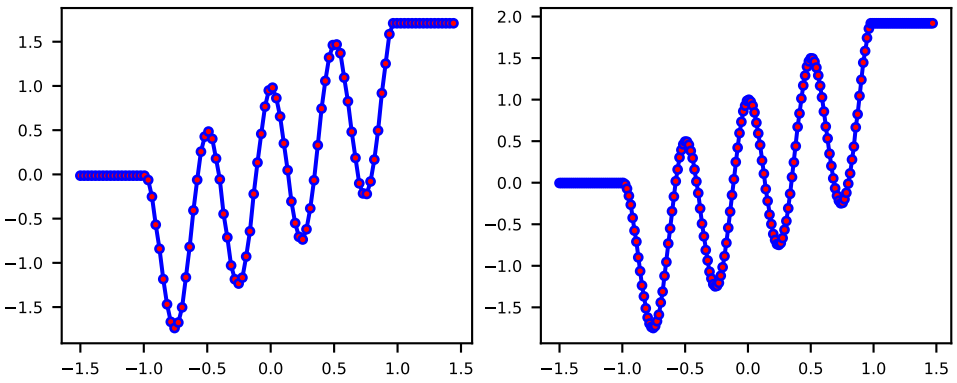


Figure 2.16: Random Forest

Table 2.13 provides all computed indicators after running all scenarios indicated in Table 2.3.

Table 2.13: Random Forest performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
RForest	1	100	100	100	1	0.08	0.4435	0.90	0.0498
RForest	1	200	200	200	1	0.09	0.4579	0.95	0.1155
RForest	1	300	300	300	1	0.10	0.4602	0.92	0.1194
RForest	1	400	400	400	1	0.11	0.4606	0.92	0.0928

2.7.6 A comparison between methods

We benchmark methods, comparing any computed indicators as follows.

Observe that function norms and discrepancy errors are not method-dependent. Clearly, for this example, a periodical kernel-based method outperforms the two other ones. However, it is not our goal to illustrate a particular method supremacy, but a benchmark methodology, particularly in the context of extrapolating test set data far from the training set ones.

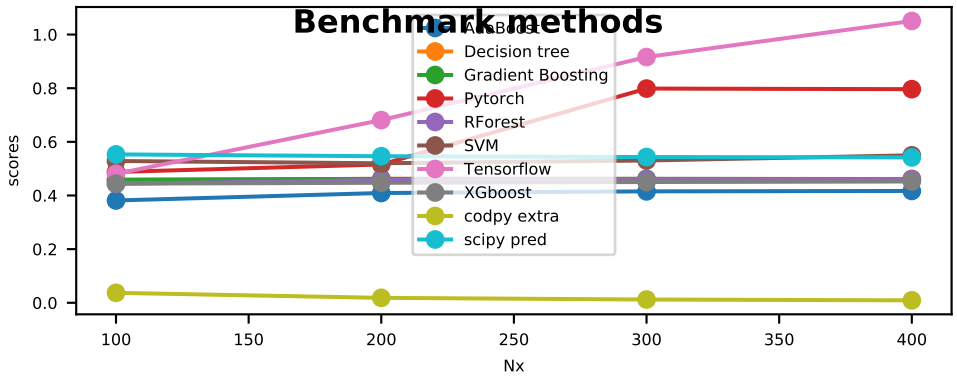


Figure 2.17: Benchmarking scores (RMSE)

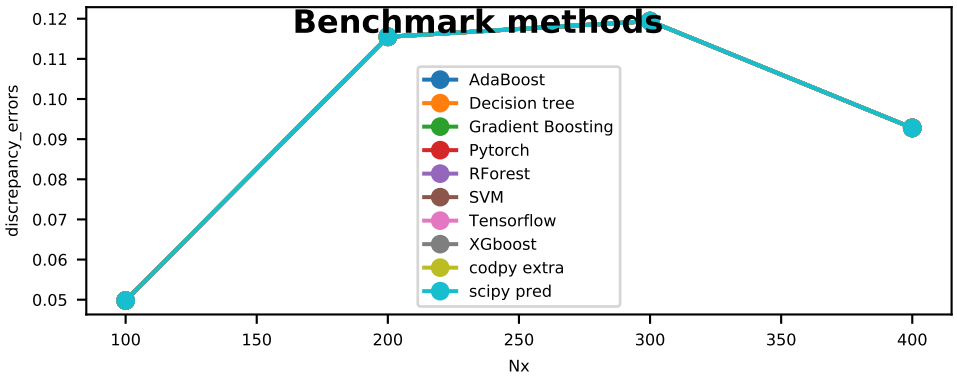


Figure 2.18: Discrepancy error

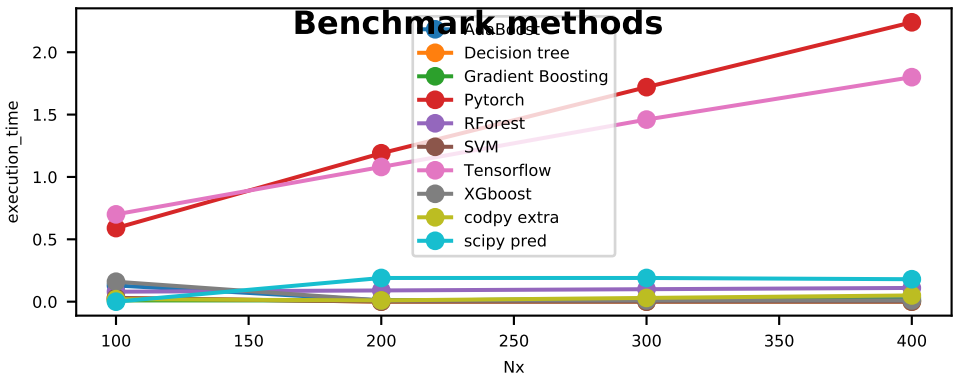


Figure 2.19: Computation time comparison

Table 2.14: scenario list

D	N_x	N_y	N_z
2	2500	2500	2500
2	1600	1600	1600
2	900	900	900
2	400	400	400

2.8 Tutorial in N dimensions

2.8.1 Initialization

Now we illustrate the fact that the dimension arising in the problem under consideration does not change benchmark methods. To illustrate this point, we simply copy/paste the previous step used for the one-dimensional case, but setting the dimension to two, that is $D = 2$, and the user can test with this parameter. Only data visualization changes.

We first pick-up a scenario list, see Table 2.14, to be compared to the one-dimensional scenario Table 2.3.

Then we generate data and in Figure 2.20 we show both graphs $(x, f(x))$ (left, training set),$(z, f(z))$ (right, test set) for illustration purposes, f being defined in Section 2.4.1. Observe that, if the dimension is greater to two, we use a two dimensional visualization, plotting $\tilde{x}, f(x)$, where $\tilde{x}$ is obtained

- either setting indices $\tilde{x} := x[index1, index2]$
- or performing a PCA over x and setting $\tilde{x} := PCA(x)[index1, index2]$.

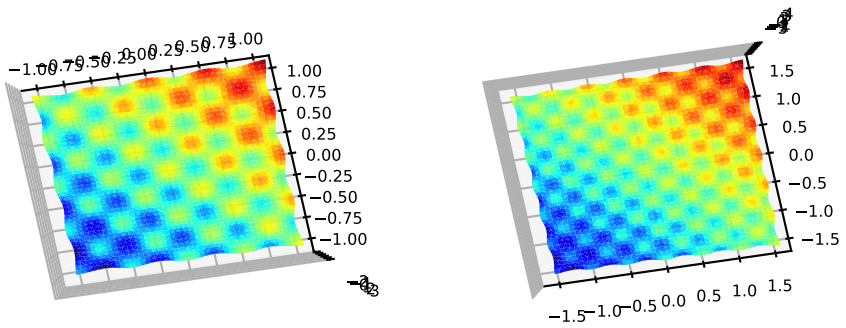


Figure 2.20: train vs test set.

2.8.2 Periodic kernel for machine learning

This defines a standard periodic Gaussian kernel, with a linear regression kernel, allowing us to fit both periodical and polynomial parts of our data.

Table 2.15 shows the computed indicators after running all scenarios indicated in Table 2.3.

We plot the first two results of this test: the predictions, denoted f_z, of the function $f(z)$; see Figure 2.20, for the first two scenarios defined in Table 2.3.

2.8.3 Scipy library

In this section we present the result of an extrapolation using SciPy's function RBF.

Table 2.15: CodPy performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
codpy extra	2	2704	2500	2704	1	3.20	0	1.99	1.7783
codpy extra	2	1764	1600	1764	1	1.28	0	1.98	1.4383
codpy extra	2	1024	900	1024	1	0.39	0	2.00	0.9590
codpy extra	2	484	400	484	1	0.07	0	1.97	0.1795

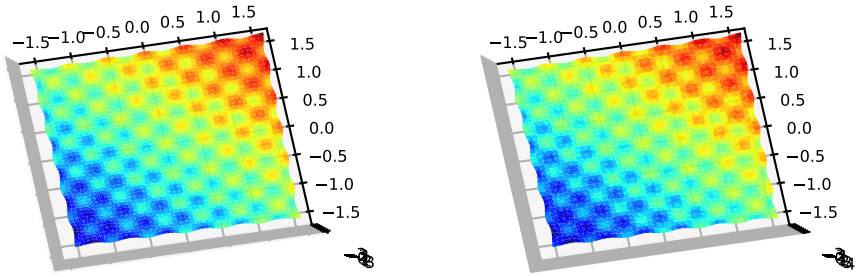


Figure 2.21: Codpy via periodic kernel: train vs test set

We provide all computed indicators after running all scenarios indicated in Table 2.3.

We end this test plotting the two first results of this test, to be compared to Figure 2.20.

2.8.4 A comparison between methods

Methods are compared in the corresponding figure.

2.9 Benchmark methodology for unsupervised learning

2.9.1 Purpose

The goal of this section is to overview our own methodology (which will be fully described in the next chapter).

- We illustrate the prediction function $\mathcal{P}_m$ for some methods in the context of supervised learning.
- We illustrate the computations of some performance indicators, as well as to present a toy benchmark using these indicators.

The data is generated using a multi-modal, multi-variate, Gaussian distribution with a covariance matrix $\Sigma = \sigma I_d$. The problem is to identify the modes of the distribution using clustering method.

Table 2.16: scipy performance indicators

$predictor_{id}$	D	N_x	N_y	N_z	D_f	time	scores	norm function	discr.error
scipy pred	2	2704	2500	2704	1	0.33	0.2771	1.99	1.7783
scipy pred	2	1764	1600	1764	1	0.21	0.2831	1.98	1.4383
scipy pred	2	1024	900	1024	1	0.12	0.2946	2.00	0.9590
scipy pred	2	484	400	484	1	0.12	0.3258	1.97	0.1795

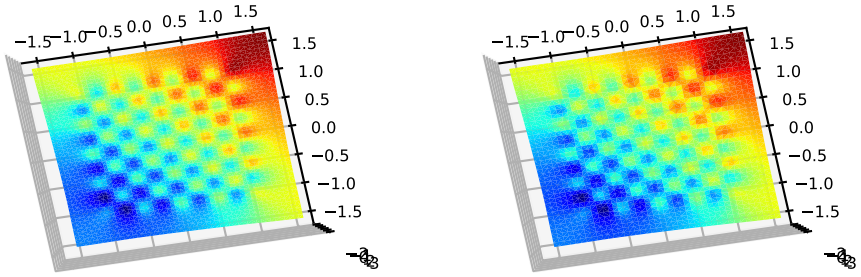


Figure 2.22: Scipy: train set vs test set

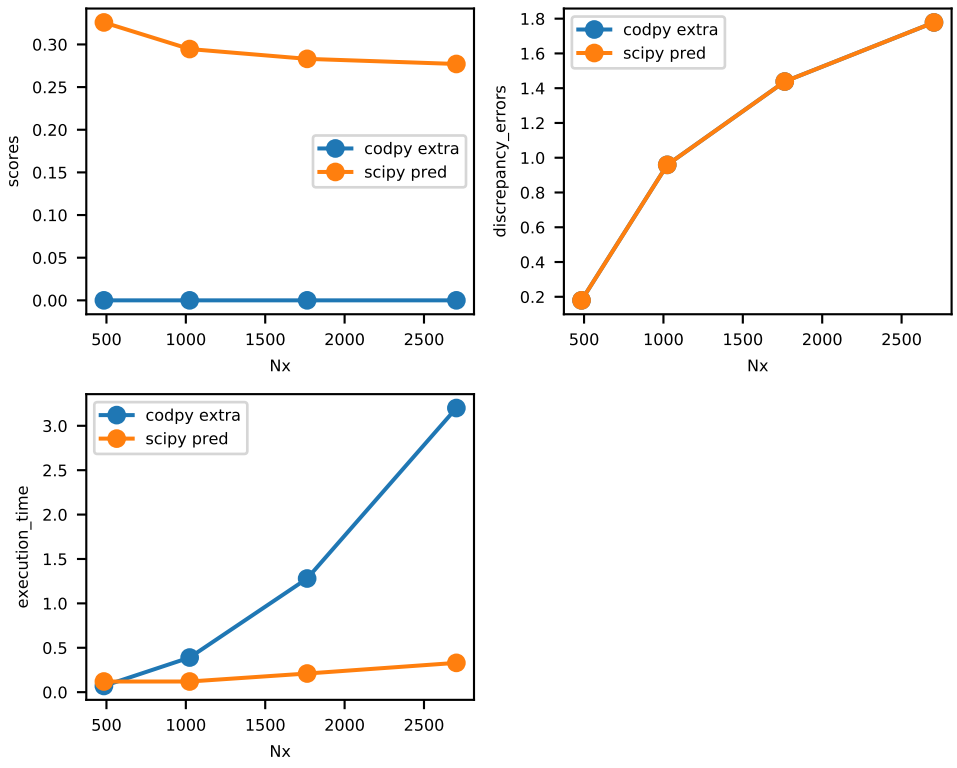


Figure 2.23: benchmark of various performance indicators for supervised learning

Table 2.17: scikit: clusters indicators

$predictor_{id}$	k-means	k-means	k-means	k-means	k-means
D	2	2	2	2	2
N_x	1000	1000	1000	1000	1000
N_y	2	3	4	5	6
N_z	1000	1000	1000	1000	1000
D_f	1	1	1	1	1
time	0.57	0.95	1.16	1.36	2.95
scores	1	0.996	0.981	0.967	0.84
norm function	0.06	0.24	0.97	1.51	6.57
discr.error	0.04	0.0352	0.0581	0.05	0.0346
score calinsky	17799.84	9972.16	5422.82	3816.27	3633.54
score harabazs	0.85	0.72	0.63	0.57	0.46
homogeneity test	1	0.98	0.93	0.91	0.79
inertia	1974.8	1953.95	1962.53	1941.25	1693.22

In the following we will generate distribution with a predetermined number of modes, it will allow to test validation scores on this toy example.

2.9.2 Analysis via k-means clustering

In this paragraph, we compute k-means clustering, using a scikit-learn implementation[6]

We first run all scenarios. We provide all computed indicators after running all scenarios indicated in Table 2.3.

k-means blob visualization. We now plot the first two distributions as well as the corresponding computed clusters.

```
## C:\informatique\github\codpy\apps\common\codpy_tools.py:1235: RuntimeWarning: More than 20 figure
##   fig = plt.figure(figsize = figsize)
```

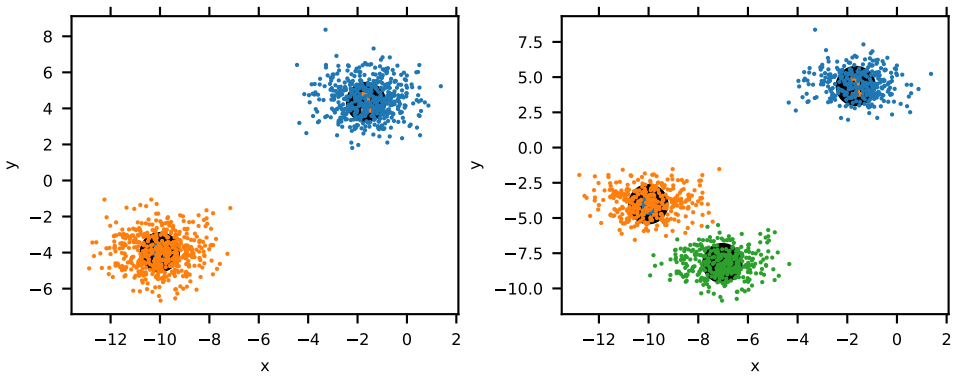


Figure 2.24: Scatter plot of clusters using scikit's k-means

k-means confusion matrix. We next plot the first two confusion matrices.

```
## C:\informatique\github\codpy\apps\common\codpy_tools.py:1235: RuntimeWarning: More than 20 figure
##   fig = plt.figure(figsize = figsize)
```

[6]the scikit-learn implementation is available using this link

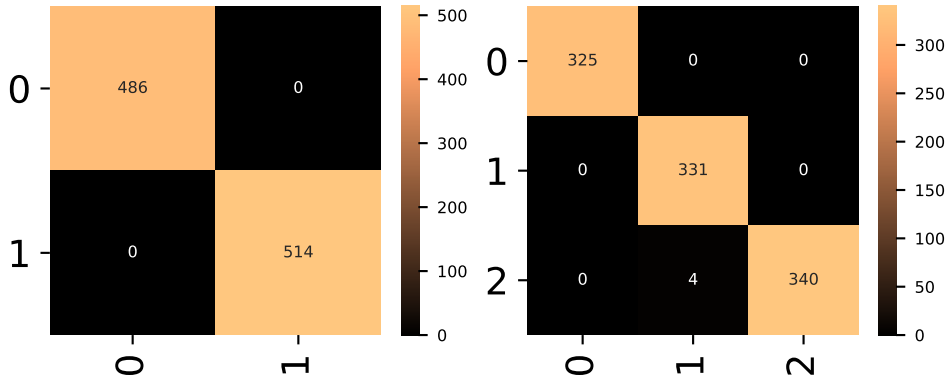


Figure 2.25: Confusion matrices of scikit's kmeans

Table 2.18: Minibatch: clusters indicators

$predictor_{id}$	minibatch	minibatch	minibatch	minibatch	minibatch
D	2	2	2	2	2
N_x	1000	1000	1000	1000	1000
N_y	2	3	4	5	6
N_z	1000	1000	1000	1000	1000
D_f	1	1	1	1	1
time	1.87	1.35	1.12	1.78	1.54
scores	1	0.996	0.982	0.962	0.808
norm function	0.06	0.24	0.97	1.54	4.72
discr.error	0.0389	0.0375	0.0589	0.0882	0.0715
scores calinsky	17799.84	9972.16	5420.6	3762.12	3516.4
score harabazs	0.85	0.72	0.63	0.56	0.46
homogeneity test	1	0.98	0.93	0.9	0.77
inertia	1975.7	1956.49	1966.19	2087.73	1745.04

2.9.3 Analysis via mini-batch clustering

To compute minibatch clustering, we use scikit-learn implementation

We provide all computed indicators after running all scenarios indicated in Table 2.3.

Minibatch blob visualization. We next plot the first two distributions as well as the corresponding computed clusters.

```
## C:\informatique\github\codpy\apps\common\codpy_tools.py:1235: RuntimeWarning: More than 20 figure
##   fig = plt.figure(figsize = figsize)
```

Minibatch confusion matrix. The figure below illustrates two confusion matrices.

```
## C:\informatique\github\codpy\apps\common\codpy_tools.py:1235: RuntimeWarning: More than 20 figure
##   fig = plt.figure(figsize = figsize)
```

2.9.4 Analysis via CodPy clustering

We also provide all the indicators after running all of the scenarios in Table 2.3.

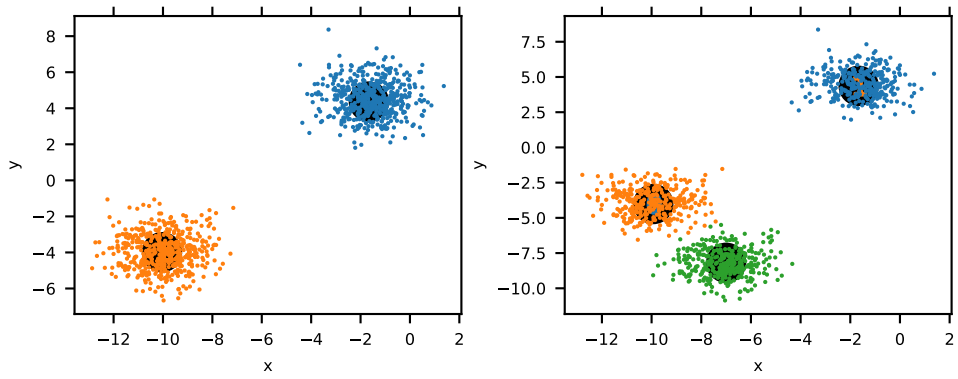


Figure 2.26: Scatter plots of scikit's minibatch kmeans

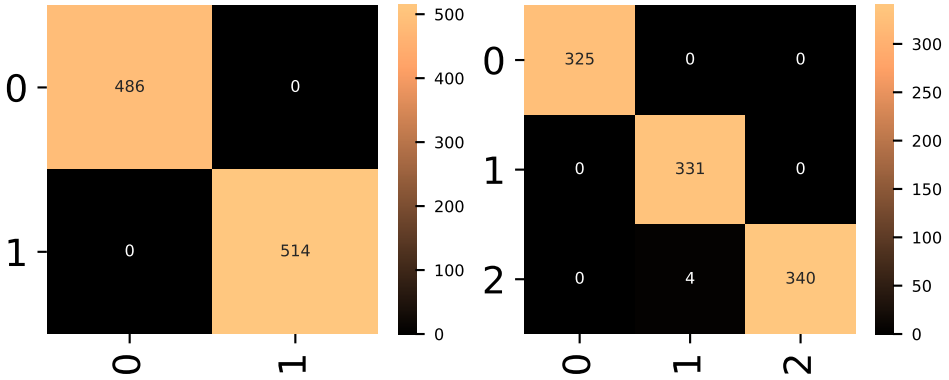


Figure 2.27: Confusion matrices of scikit's minibatch kmeans

Table 2.19: codpy: clusters indicators

$predictor_{id}$	codpy	codpy	codpy	codpy	codpy
D	2	2	2	2	2
N_x	1000	1000	1000	1000	1000
N_y	2	3	4	5	6
N_z	1000	1000	1000	1000	1000
D_f	1	1	1	1	1
time	0.04	0.07	0.09	0.06	0.06
scores	1	0.329	0.98	0.002	0.147
norm function	0.06	0.25	1	1.35	3.66
discr.error	0.0405	0.0375	0.0561	0.0501	0.0339
score calinsky	17799.84	9972.16	5424.4	3730.59	3473.41
score harabazs	0.85	0.72	0.63	0.56	0.47
homogeneity test	1	0.98	0.93	0.91	0.76
inertia	1974.8	1953.95	1962.53	1941.25	1693.94

CodPy blob visualization. We finally plot the two first distributions as well as the corresponding computed clusters

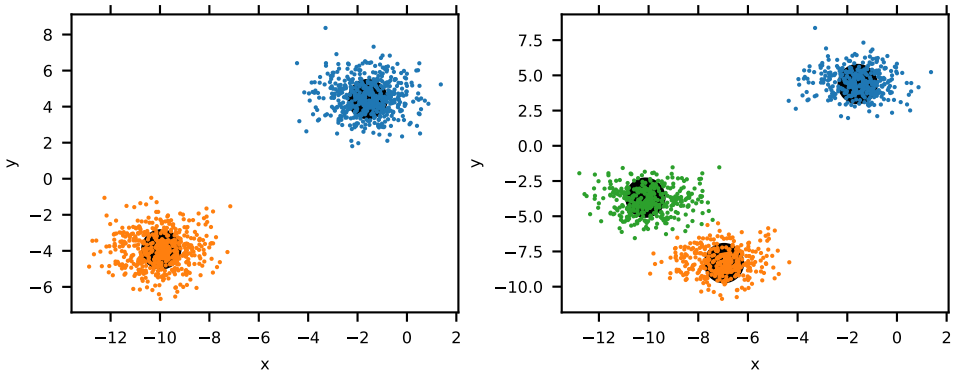


Figure 2.28: Scatter plots of codpy's clustering algorithm

CodPy confusion matrix. The figure below illustrates two confusion matrices.

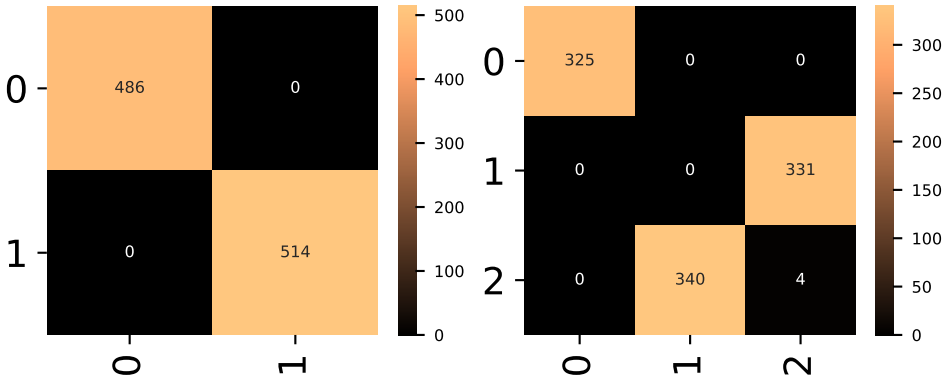


Figure 2.29: Confusion matrices of codpy's clustering algorithm

2.9.5 A comparison between methods

We compare the various methods under consideration, by comparing performance indicators, as illustrated by Figure 2.30.

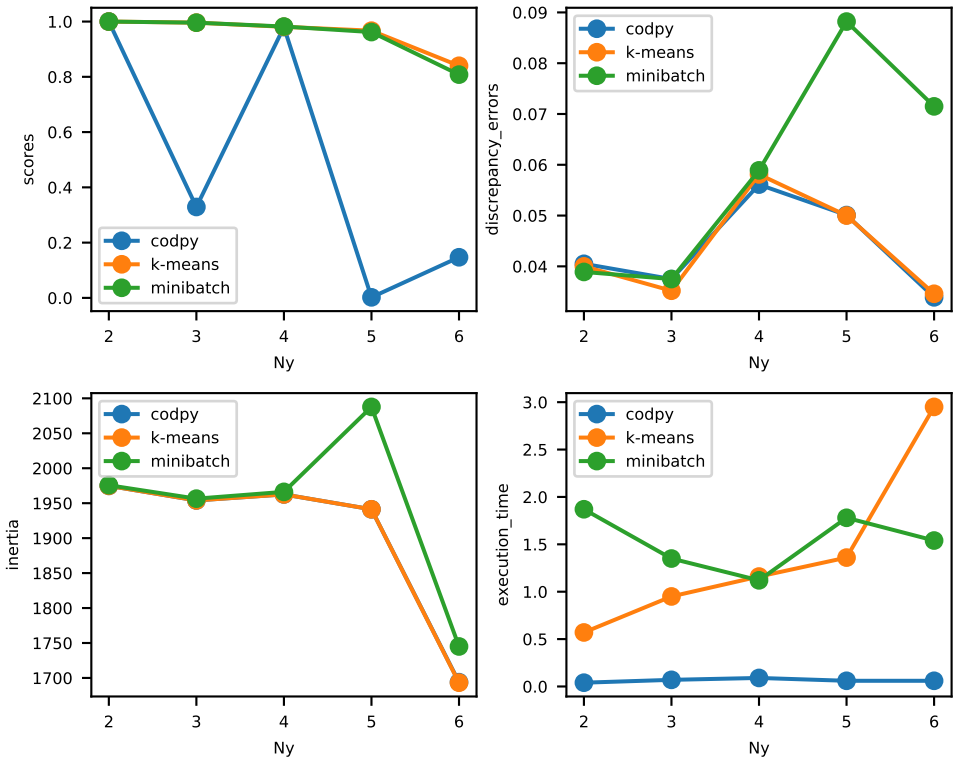


Figure 2.30: benchmark of various performance indicators for clustering.

Chapter 3

Kernel methods for machine learning

3.1 Aim of this chapter

We will present first our techniques for dealing with problems of learning machine and we cover here two main ingredients that are central in the design of our algorithms.

- First, all methods described in this chapter depend on a choice of a kernel and use transformation maps acting on basic kernels in order to adapt them to any particular problem.
- Second, this chapter also provides discrete differentiation operators that are relevant for machine learning as well problems involving partial differential operators.

Importantly, the presented below provides us with the key building blocks of our algorithms and will allow us to formulate more advanced numerical algorithms in the next chapters of this monograph.

For a precise description of our framework we need some further notation. A set of N_x observable data in D dimensions is available, denoted by the symbol $X \in \mathbb{R}^{N_x \times D}$, together with a D_f-dimensional vector-valued function $f(X) : \mathbb{R}^{N_x \times D_f}$ are the training values associated with the training variables. The input dataset is therefore

$$(X, f(X)) := \{x^n, f(x^n)\}_{n=1,\ldots,N_x}, \quad X \in \mathbb{R}^{N_x \times D}, \qquad f(X) \in \mathbb{R}^{N_x \times D_f}.$$

We are interested in predicting test values $f(Z) : \mathbb{R}^{N_z \times D_f}$ on a new set of variables called *a test set* $Z \in \mathbb{R}^{N_z \times D}$:

$$(Z, f_z) := \{z^n, f_z^n\}_{n=1,\ldots,N_z}, \quad Z \in \mathbb{R}^{N_z \times D}, \qquad f_z \in \mathbb{R}^{N_z \times D_f}. \tag{3.1.1}$$

In all examples and numerical experiments in this section we take a matrix $X \in \mathbb{R}^{N_x \times D}$ and use the following periodic and increasing function:

$$f(X) = \prod_{d=1,\ldots,D} \cos(4\pi x_d) + \sum_{d=1,\ldots,D} x_d, \qquad x \in \mathbb{R}^D \tag{3.1.2}$$

We take $X \in \mathbb{R}^{N_x \times D}, Z \in \mathbb{R}^{N_z \times D}$, $f(X) \in \mathbb{R}^{N_x \times D_f}$, $f(Z) \in \mathbb{R}^{N_z \times D_f}$.

Table 3.1: Data's dimensions

D	N_x	N_y	N_z
2	576	576	576

The case $N_x = N_z$ corresponds to data **extrapolation**, as explained in 3.2.3 below. For illustration purposes, we set another set of parameters, corresponding to data **projection**, as explained also in the section 3.2.3, i.e. when $N_y << N_x$.

Table 3.2: Data's dimensions

D	N_x	N_y	N_z
2	576	32	576

The figure 3.1 provides us with an example of machine learning settings. We will rely on the first one throughout the following discussion. Here, the left-hand figures is the training set $(X, f(X))$ of variables and values and the righ-hand figure displays the test set $(Z, f(Z))$ of variable and values, while the middle figure shows the parameter set $(Y, f(Y))$ of variables and values. As mentioned earlier, the latter is a choice, made by a reader, determining not only the overall accuracy, but also computational cost.

3.2 Fundamental notions for supervised learning

3.2.1 Preliminaries

Positive definite kernels and kernel matrices. We briefly give a definition of a kernel.

Let $k : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ be a symmetric real-valued function, i.e. satisfying $k(x, y) = k(y, x)$. For any two sequences of points $X = x^1, \cdots, x^{N_x} \in \mathbb{R}^D$, $Y = y^1, \cdots, y^{N_y} \in \mathbb{R}^D$ we define a kernel matrix $K(X, Y) := \left(k(x^n, y^m)\right)_{n,m} \in \mathbb{R}^{N_x \times N_y}$

$$K(X,Y) = \begin{pmatrix} k(x^1, y^1) & \cdots & k(x^1, y^{N_y}) \\ \ddots & \ddots & \ddots \\ k(x^{N_x}, y^1) & \cdots & k(x^{N_x}, y^{N_y}) \end{pmatrix} \tag{3.2.1}$$

We say that k is a positive-definite kernel if for any sequence of distinct points $X \in \mathbb{R}^{N_x \times D}$ and $c^1, ..., c^{N_x} \in \mathbb{R}^{N_x}$:

$$\sum_{i,j \leq N_x} c^i c^j k(x^i, x^j) \geq 0. \tag{3.2.2}$$

If $N_x = N_y$, the matrix $K(X, Y)$ is called a Gram matrix. The dimension of a kernel matrix $K(X, Y)$ is usually $N_x \times N_y$, except for some combined kernels; see the section on kernel engineering 3.4.

If $K(X, Y)$ is positive-definite on a certain submanifold, we say that the kernel is **conditionally positive-definite** – in the sense that it is positive-definite, conditionally to the fact that X, Y belongs to this submanifold.

As should be expected we can not use an arbitrary symmetric function and in our framework, we always use the kernels that are in the class of positive-definite kernels. The available kernels in our library are listed in the table 3.3:

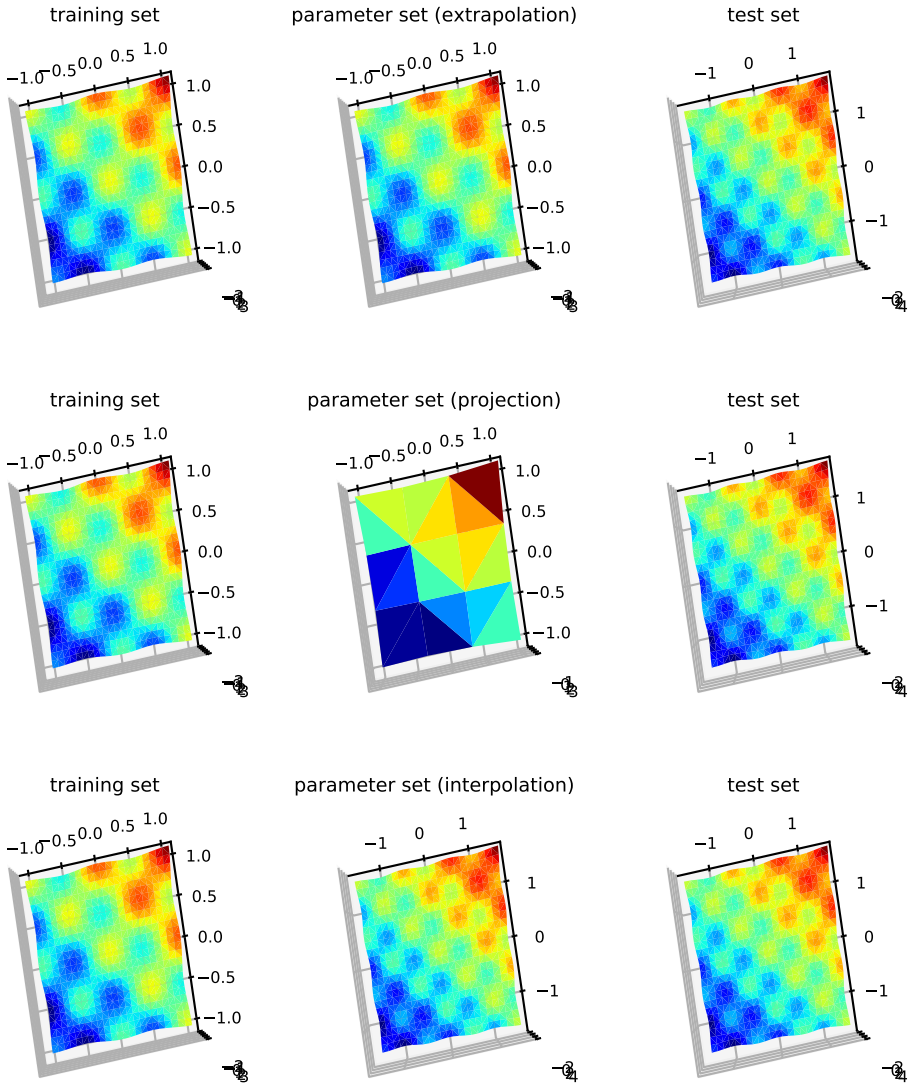


Figure 3.1: Examples of Training set, parameter set, and test set, for three different parameter set y

Table 3.3: A list of available kernels

DotProduct	maternnorm	multiquadrictensor	truncatednorm
RELU	maternper	sincardsquaretensor	truncatedper
gaussian	materntensor	sincardtensor	truncatednorm
gaussianper	multiquadricnorm	tensornorm	truncatedper

Observe that a certain scaling of the kernel may be required in order to handle some input data, which is exactly the purpose of the maps, discussed below.

Example 3.2.1. *Gaussian kernel is used by default in CoDpy library:*

$$k(x, y) = \exp(-\pi|x - y|^2). \tag{3.2.3}$$

Example 3.2.2. *Consider the following family of symmetric functions $k(x, y)$ with $x, y \in \mathbb{R}^D$:*

$$k(x, y) = g(< S(x), S(y) >_{\mathbb{R}^P}), \quad S : \mathbb{R}^D \mapsto \mathbb{R}^P,$$

where g is called an activation function and S is a mapping. In particular, $k(x, y) =< (1, x, x^T x, ...), (1, y, y^T y, ...) >$ defines a kernel corresponding to a linear regression over a polynomial basis, hence is positive definite, but is not strictly positive. The RELU kernel, given as $k(x, y) = \max(< x, y > +c, 0)$ (with c being a constant) is a conditionally positive definite.

Let us choose the kernel to be *tensornorm* (again discussed below) and we refer to the section 2 for the description of external parameters in the described kernel method. Finally, we output some values of the kernel matrix induced by Gaussian kernel in the table 3.4 computed using Codpy's *op.Knm* function.

Table 3.4: First four rows and columns of a kernel matrix $K(X, Y)$

1.00	0.96	0.92	0.88
0.96	1.00	0.96	0.92
0.92	0.96	1.00	0.96
0.88	0.92	0.96	1.00

Inverse kernel matrix. The inverse of a Kernel matrix is denoted $K(X, Y)^{-1}$, and this inverse is computed, if $X = Y$, as follows:

$$K(X, X)^{-1} = (K(X, X) + \epsilon I_d)^{-1}.$$

In the general range $X \neq Y$, it is computed using a least-square inversion, namely

$$K(X, Y)^{-1} = (K(Y, X)K(X, Y) + \epsilon I_d)^{-1} K(Y, X)$$

We refer to Tikhonov regularization parameter ϵ as a REGULARIZATION parameter, and by default takes the value $\epsilon = 10^{-8}$.

The table 3.5 illustrates the first four rows and columns of the kernel matrix's inverse $K(X, Y)^{-1} \in \mathbb{R}^{N_y \times N_x}$, $N_x = N_y$.

Table 3.5: First four rows and columns of an inverted kernel matrix $K(X, Y)^{-1}$

168.0384415	-162.0371111	0.0000783	-0.0000705
-162.0370979	324.0742063	-162.0371706	0.0001191
0.0000542	-162.0371460	324.0741716	-162.0371230
-0.0000459	0.0000957	-162.0371274	324.0741582

The matrix product $K(X,Y)K(X,Y)^{-1}$ in the table 3.5 is just a projection operator. It might not be the identity depending on the experimental setting, for one of the following reasons:

- If $N_x \neq N_y$.
- If the Tikhonov regularization parameter $\epsilon > 0$. One can put $\epsilon = 0$, this choice can be set by the user, but take care of performance issues. If the kernel is not strictly positive, then the library might raise an exception, and switch from a standard inversion of matrix to a non-strictly positive-definite inversion, that can be more computationally costly.
- If the kernel under consideration is such that $K(X,X)K(X,X)^{-1}$ does not have a full rank, for instance if a linear regression kernel is used; see the section on kernel engineering (section 3.4). In which case this matrix is a projection over $Ker(K(X,X))$.

Distance matrices. Distance matrix is simple and very handy tool for kernel methods.

Let $k(\cdot,\cdot) : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ be a positive-definite kernel. Then the distance function $d_k(x,y)$ for $x \in \mathbb{R}^D, y \in \mathbb{R}^D$ is defined as follows:

$$d_k(x,y) = k(x,x) + k(y,y) - 2k(x,y). \tag{3.2.4}$$

Note that for a positive-definite kernel the latter expression is continuous, positive, and satisfies $d(x,x) = 0$.

For any two sequences of points $X = x^1, ..., x^{N_x} \in \mathbb{R}^D$, $Y = y^1, ..., y^{N_y} \in \mathbb{R}^D$ we define a distance matrix $D(X,Y) \in \mathbb{R}^{N_x \times N_y}$:

$$D(X,Y) = \begin{pmatrix} d_k(x^1,y^1) & \cdots & d_k(x^1,y^M) \\ \ddots & \ddots & \ddots \\ d_k(x^N,y^1) & \cdots & d_k(x^N,y^M) \end{pmatrix} \tag{3.2.5}$$

The table 3.6 outputs the first four columns of the kernel-based distance distance matrix $D(X,Y)$.

Table 3.6: First four rows and columns of a kernel-based distance matrix $D(X,Y)$

0.00	0.08	0.16	0.24
0.08	0.00	0.08	0.16
0.16	0.08	0.00	0.08
0.24	0.16	0.08	0.00

CodPy's algorithms. CodPy's algorithms offer general functions in order to get predictions in (3.1.1) from the choice of a kernel. More precisely, the following operator (with $A^{-1} := (A^TA)^{-1}A^T$ denoting the least-square inverse)

$$f_z := \mathcal{P}_k(X,Y,Z)f(X) := K(Z,Y)K(X,Y)^{-1}f(X), \quad K(Z,Y) \in \mathbb{R}^{N_z \times N_y}, K(X,Y) \in \mathbb{R}^{N_x \times N_y} \tag{3.2.6}$$

defines a supervised learning machine which we call a **feed-forward** machine. We also consider $\mathcal{P}_k(X,Y,Z) \in \mathbb{R}^{N_z \times N_x}$ as a **projection operator** and it is well-defined once a kernel k has been chosen. Observe that two factors arise in (3.2.6), namely the so-called Kernel matrix $K(X,Y)$ (discussed below) and the **projection set of variables** denoted by $Y \in \mathbb{R}^{N_y \times D}$. To motivate the role of the later, let us consider two particular operators that do not depend upon Y:

$$\text{Extrapolation operator: } \mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,X)^{-1} \in \mathbb{R}, \tag{3.2.7}$$

$$\text{Interpolation operator: } \mathcal{P}_k(X,Z,Z) = K(Z,Z)K(X,Z)^{-1}. \tag{3.2.8}$$

These operators sometimes generate computational issues, due to the fact that the Kernel matrix $K(X,X) \in \mathbb{R}^{N_x \times N_x}$ must be inverted (3.2.6) and this is a rather costly computational step in presence of a large set of input data. This is our motivation for introducing the additional variable Y

Table 3.7: A list of available transportation maps.

affine_map
exp
identity
log
map_to_grid
scale_std
scale_to_erf
scale_to_erfinv
scale_to_mean_distance
scale_to_min_distance
scale_to_unitcube

which has the effect to lower the computational cost. It reduces the overall algorithmic complexity of (3.2.6) to the order of

$$D\left((N_y)^3 + (N_y)^2 N_x + (N_y)^2 N_z\right).$$

Most importantly, the projection operator $\mathcal{P}_k$ is *linear* in term of, both, input and output data. Hence, while keeping the set Y to a reasonable size, we can consider large set of data, as input or output.

The reader can imagine also that choosing a relevant set Y is a major source of optimization. We use this idea intensively in several applications. For instance, kernel clustering methods described in the section 3.6 aims minimizing the error committed by our learning machine with respect to the set $Y = \mathcal{P}_k(X, Z)$, called **sharp discrepancy sequences** and defined below. We often refer to this step as **learning**, as this step is exactly the counterpart of the weight set for neural network approach. This construction amounts to define a feed-backward machine, analogous to (3.2.6), as

$$f_z := \mathcal{P}_k(X, \mathcal{P}_k(X, Z), Z) f(X)$$

Note that (3.2.6) allows us also to compute the following operation, where $\nabla := (\partial_1, \ldots, \partial_D)$ holds for the gradient

$$(\nabla f)(Z) := (\nabla \mathcal{P}_k)(X, Y, Z) f(X) := (\nabla_z k)(Z, Y) K(X, Y)^{-1} f(X) \in \mathbb{R}^{D \times N_z \times D_f}$$

meaning that $\nabla \mathcal{P}_k \in \mathbb{R}^{D \times N_z \times N_x}$ is interpreted as a tensor operator. This operator is described in the section 3.5, as well as numerous others, as for instance Laplacian, Leray, integral operators, that are based on it. They will be used in designing computational methods for problems involving partial differential equations(PDEs).

3.2.2 Transportation maps

Let us define an important concept of a **transportation map**. A transportation map S is a surjective map

$$S : \mathbb{R}^T \mapsto \mathbb{R}^D.$$

There are several types of transportation maps such as

- **rescaling maps** when $T = D$, in order properly the data X, Y, Z to a given kernel,
- **dimension reduction maps** when $T \leq D$, or
- **dimension augmentation** when $T \geq D$, when adding information to the training set is required. This transformation is sometimes called a **kernel trick**.

The list of maps available in our framework is given in the table 3.7.

Using a map S amounts to change the kernel as $k(x,y) \mapsto k(S(x),S(y))$. For instance, for Gaussian kernels the map *scale_to_min_distance* is usually a good choice: this map scales all points to the average min distance, namely

$$S(x) = \frac{x}{\sqrt{\alpha}}, \qquad \alpha = \frac{1}{N}\sum_{i\leq N} \min_{k\neq i} |x^i - x^k|^2..$$

Let us transform our Gaussian kernel with this map. Observe that, from the signature of the Gaussian setter function, we see that the Gaussian kernel is handled with the default map *set_min_distance_map*. We do not discuss all optional parameters now, but refer the reader to a later section.

$$\begin{aligned} kernel\_setters.set\_gaussian\_kernel(&polynomial\_order : int = 0,\\ &regularization : float = 1e-8,\\ &set\_map = map\_setters.set\_min\_distance\_map) \end{aligned}$$

3.2.3 Extrapolation, interpolation, and projection

The Python function in our framework that describes the projection operator $\mathcal{P}_k$ is based on the definition in (3.2.6), namely

$$f_z = \text{op.projection}(X, Y, Z, f(X) = [], k = None, rescale = False) \in \mathbb{R}^{N_z \times D_f}.$$

The optional values in this function are as follows:

- The function $f(X)$ is optional, meaning that the user can retrieve the whole matrix $\mathcal{P}_k(X,Y,Z) \in \mathbb{R}^{N_z \times N_x}$ if desired.
- The kernel k is optional, meaning that we let to the user the freedom to re-use an already input kernel.
- The optional value *rescale*, defaulted to false, allow to call the map prior of performing the projection operation (3.2.6), in order to compute its internal states for proper data scaling. For instance, a rescaling (3.2.2) computes α according to the set (X,Y,Z).

Interpolation and extrapolation Python functions are, in agreement with (3.2.8), simple decorators toward the operator $\mathcal{P}_k$; see (3.2.3). Obviously, the main question arising at this stage is how good the approximation is f_z compared to $f(Z)$, which is the question addressed in the next section.

$$f_z = \text{op.extrapolation}(X, Z, f(X) = [], ...), \quad f_z = \text{op.interpolation}(X, Z, f(X) = [], ...)$$

3.2.4 Functional spaces and Kolmogorov decomposition.

Given any finite collection of points $X = [x^1 ... x^{N_x}]$, $x^i \in \mathbb{R}^D, i = 1, ..., N_x$, we introduce a (finite dimensional) vector space $\mathcal{H}_k^x$ consisting of all linear combinations of the *basis functions* $x \mapsto k(x, x^n)$. In other words, we set

$$\mathcal{H}_k^x = \Big\{ \sum_{1\leq m\leq N_x} a_m k(\cdot, x^m) \,/\, a = (a^1, ..., a^{N_x}) \in \mathbb{R}^{N_x} \Big\}. \tag{3.2.9}$$

The **functional space** $\mathcal{H}_k$ is (after suitably passing to a limit in (3.2.9))

$$\mathcal{H}_k = \text{Span}\big\{k(\cdot, x) \,/\, x \in \mathbb{R}^D\big\}. \tag{3.2.10}$$

This space consists of all linear combinations of the functions $k(x,\cdot)$ (that is, parametrized by $x \in \mathbb{R}^D$) and is endowed with the scalar product

$$\big\langle k(\cdot, x), k(\cdot, y) \big\rangle_{\mathcal{H}_k} = k(x, y), \qquad x, y \in \mathbb{R}^D. \tag{3.2.11}$$

On every finite dimensional subspace $\mathcal{H}_k^x \subset \mathcal{H}_k$ we can check that, according to the expression of the scalar product,

$$\left\langle k(\cdot, x^i), k(\cdot, x^j) \right\rangle_{\mathcal{H}_k^x} = k(x^i, x) K(X, X)^{-1} k(x, x^j) = k(x^i, x^j), \ i, j = 1, ..., N_x. \tag{3.2.12}$$

The norm of any function f, in view of the functional space $\mathcal{H}_k$, is fully determined by the kernel k. A reasonable approximation of this norm, which is induced by the kernel matrix K is given by

$$\|f\|^2_{\mathcal{H}_k} \sim f(x^i)^T K(X, X)^{-1} f(x^i), \quad x^i \in \mathbb{R}^D, \ i = 1, ..., N_x$$

It is computed via the function in Python:

$$op.norm(X, Y, Z, f(X), set\_codpy\_kernel = None, rescale = True).$$

Again we let the reader the choice to perform a rescaling of the kernel based on a transport map.

3.2.5 Error estimates based on the discrepancy error

Recall the notation for the projection operator (3.2.6). Then the following estimation error holds for any vector-valued function $f : \mathbb{R}^D \mapsto \mathbb{R}^{D_f}$:

$$\Big| \frac{1}{N_x} \sum_{n=1}^{N_x} f(x^n) - \frac{1}{N_z} \sum_{n=1}^{N_z} f_{z^n} \Big| \leq \Big(d_k(X, Y) + d_k(Y, Z) \Big) \|f\|_{\mathcal{H}_k}.$$

Before describing this formula, let us precise that it is a computationally tractable one, that can be systematically applied to check the pertinence of any kernel machine. It is also a quite general one: this formula can be adapted to others kind of error measuring. We have also

$$\Big\| f(Z) - f_z \Big\|_{\ell^2(N_z)^{D_f}} \leq \Big(d_k(X, Y) + d_k(Y, Z) \Big) \|f\|_{\mathcal{H}_k}.$$

This error formula can be split into two parts.

The first part, $\Big(d_k(X, Y) + d_k(Y, Z) \Big)$ measures some kernel-related distance between a set of points, that we call the **discrepancy error**. It is a quite natural quantity, as one expects that the quality of an extrapolation degrades if the extrapolation set Z move away from the sampling set X. Its definition is

$$d_k(X, Y)^2 := \frac{1}{N_x^2} \sum_{n=1,m=1}^{N_x, N_x} k(x^n, x^m) + \frac{1}{N_y^2} \sum_{n=1,m=1}^{N_y, N_y} k(y^n, y^m) - \frac{2}{N_x N_y} \sum_{n=1,m=1}^{N_x, N_y} k(x^n, y^m)$$

and is described in the Python function

$$op.discrepancy(X, Y, Z, set\_codpy\_kernel = None, rescale = True)$$

In particular, the user should pay attention to an undesirable rescaling effect due to the variable *rescale*. The section 3.6.5 contains plots and some analysis of this functional. This distance was named Maximum Mean Discrepancy(MMD) and introduced independently in [13].

3.3 Dealing with kernels

3.3.1 Maps and kernels.

Maps can ruin your prediction. In the figure 3.2 We compare the ground truth values $(Z, f(Z)) \in \mathbb{R}^{N_z \times D} \times \mathbb{R}^{N_z \times D_f}$ and the predicted values $(Z, f_z) \in \mathbb{R}^{N_z \times D} \times \mathbb{R}^{N_z \times D_f}$ figure 3.2.

we set a different map, called set_mean_distance_map, which scales all points to the average distance for a Gaussian kernel:

$$S(Z) = \frac{Z}{\sqrt{\alpha}}, \qquad \alpha = \sum_{i,k \leq N_z} \frac{|z^i - z^k|^2}{N_z^2}. \tag{3.3.1}$$

This example illustrates how maps can drastically influence the computation.

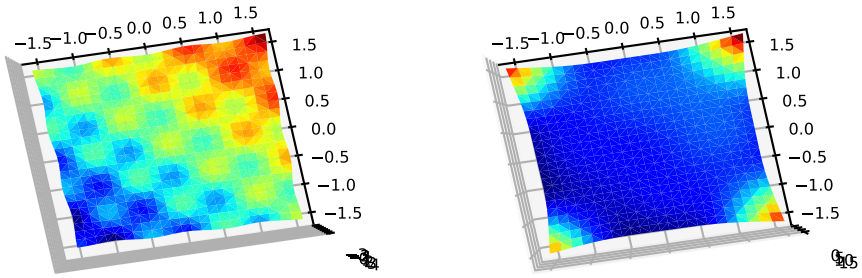


Figure 3.2: Ground truth value (left) and predicted values (right)

However, the very same map can be appropriate for other kernels; see Figure 3.3 with a Matern kernel.

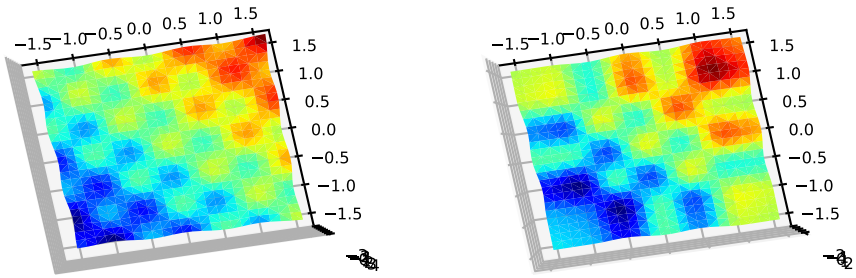


Figure 3.3: Ground truth values (left) and predicted values (right)

Composition of maps. Maps can be composed together. For instance, consider the following Python definition of one of our maps, which we may use as a Swiss-knife map for Gaussian kernels:

For any $X \in \mathbb{R}^{N_x \times D}$, this composite map performs the following operations:

- First rescale all data to the unit cube:

$$S(X) = \frac{x_d - \min_n x_d^n + \frac{0.5}{N_x}}{\alpha_d}, \quad \alpha_d := \max_n x_d^n - \min_n x_d^n \tag{3.3.2}$$

- Then apply the map defined as

$$S(X) = erf^{-1}(2X - 1), \tag{3.3.3}$$

erf^{-1} being the inverse of the standard error function erf.

- Finally use the average min distance map, defined as

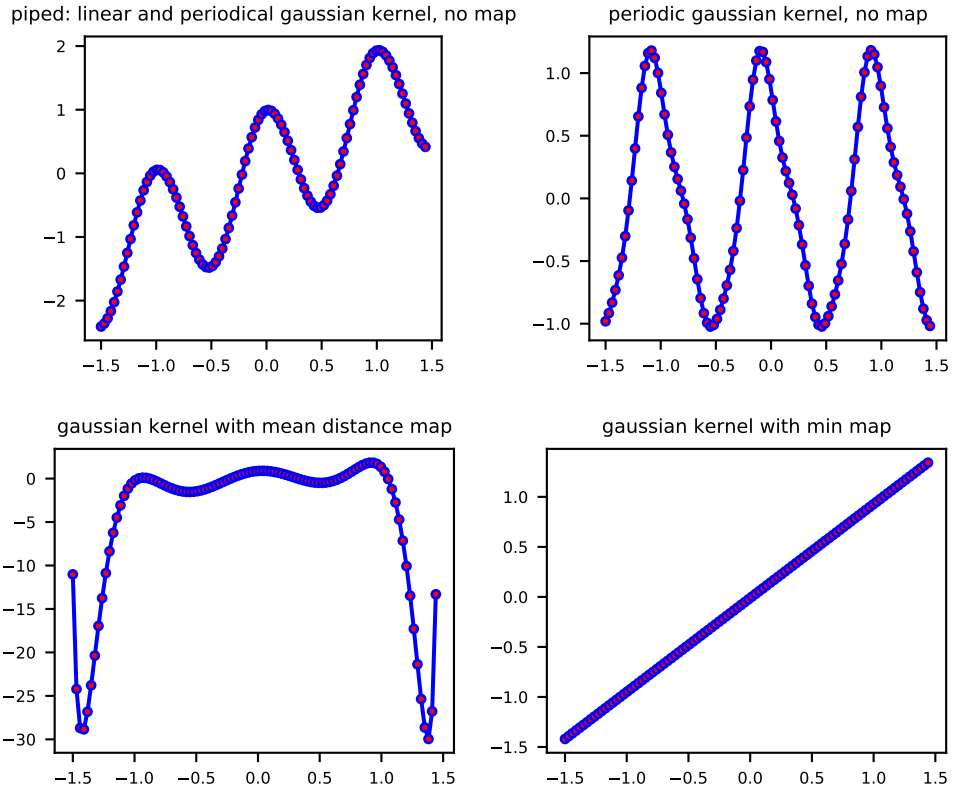
$$S(X) = \frac{X}{\sqrt{\alpha}}, \quad \alpha = \frac{1}{N_x} \sum_{i \leq N_x} \min_{k \neq i} |x_i - x_k|^2. \tag{3.3.4}$$

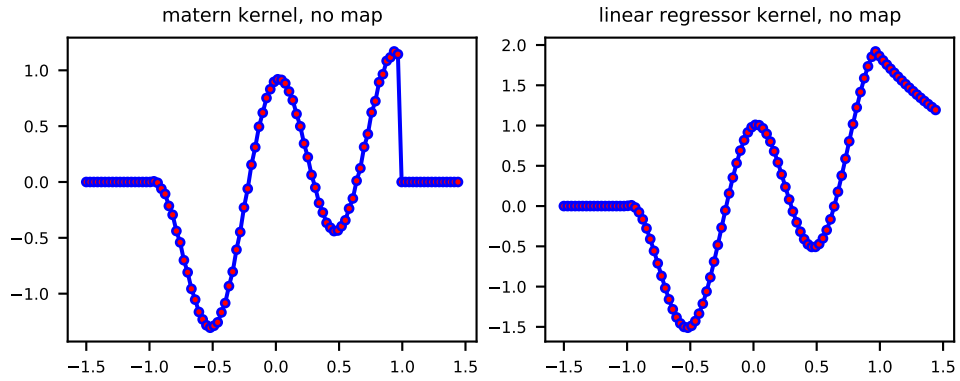
3.3.2 Illustration of different kernels prediction

As is clear from the previous sections, the external parameters of a kernel-based prediction machine are

- The kernel. In most situations, a kernel is defined by
  - A positive definite kernel, that is one element of table 3.3.
  - A map, that is one element of table 3.7.
- The choice of the inner parameters set Y. We usually face three main choices here.
  - $Y = X$, that corresponds to the *extrapolation* case, section 3.2.3. This is the most efficient choice when one seeks for high accuracy results.
  - Y is randomly picked among X. This choice trades accuracy for execution time and is adapted to heavy training set.
  - Y is set as a sharp discrepancy sequence of X, described in the section 3.6. This choice optimizes accuracy versus execution time. These are the most possible accurate machine at a given computational burden but involves a time-consuming algorithm.

To illustrate the effects of different kernels and maps on learning machines, we compare in this section predictions for the one-dimensional test described the section 2.8.2, for different kernels.





3.4 Kernel engineering

In this section we describe some general operations on kernels, which allow us to generate new and relevant kernels. These operations preserve a positivity property required for kernels. For this section, two kernels denoted by $k_i(x,y) : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}, i = 1,2$ are given with corresponding matrices K_1 and K_2. In agreement with (3.2.6), we introduce the corresponding two projection operators:

$$\mathcal{P}_{k_i}(X,Y,Z) := K_i(Z,Y)K_i(X,Y)^{-1} \in \mathbb{R}^{N_z \times N_x}, i = 1,2 \tag{3.4.1}$$

3.4.1 Manipulating kernels

In order to work with two (or more) kernels, we introduced the following Python function, which are basic *setters* and *getters* to kernels: *get_kernel_ptr()* and *set_kernel_ptr(kernel_ptr)*. The first one allows us to recover an already input kernel of our library, while the second one allows us to input it into our framework.

3.4.2 Adding kernels

The first operation, denoted by $k_1 + k_2$ and defined from any two kernels, consists in simply adding two kernels. If K_1 and K_2 are two kernel matrices associated to the kernels k_1 and k_2, then we define the sum of two kernels as $K(X,Y) \in \mathbb{R}^{N_x \times N_y}$ and corresponding projection operator as $\mathcal{P}_k(X,Y,Z) \in \mathbb{R}^{N_z \times N_y}$:

$$K(X,Y) := K_1(X,Y) + K_2(X,Y), \quad \mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,Y)^{-1}. \tag{3.4.2}$$

The functional space generated by $k_1 + k_2$ is then

$$\mathcal{H}_k = \Big\{ \sum_{1 \le m \le N_x} a^m (k_1(\cdot, x^m) + k_2(\cdot, x^m)) \Big\}. \tag{3.4.3}$$

3.4.3 Multiplication kernels

A second operation, denoted by $k_1 \cdot k_2$ and defined from any two kernels, consists in multiplying two kernels together. A kernel matrix $K(X,Y) \in \mathbb{R}^{N_x \times N_y}$ and a projection operator $\mathcal{P}_k(X,Y,Z) \in \mathbb{R}^{N_z \times N_y}$ corresponding to the product of two kernels are defined as

$$K(X,Y) := K_1(X,Y) \circ K_2(X,Y), \quad \mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,Y)^{-1}, \tag{3.4.4}$$

where $\circ$ is the Hadamard product of two matrices. The functional space generated by $k_1 \cdot k_2$ is

$$\mathcal{H}_k = \Big\{ \sum_{1 \le m \le N_x} a^m (k_1(\cdot, x^m) k_2(\cdot, x^m)) \Big\}. \tag{3.4.5}$$

3.4.4 Convolution kernels

Our next operation, denoted by $k_1 * k_2$ and defined from any two kernels, consists in multiplying corresponding kernel matrices K_1 and K_2 as

$$K(X,Y) := K_1(X,Y)K_2(Y,Y), \tag{3.4.6}$$

where $K_1(X,Y)K_2(Y,Y)$ stands for the standard matrix multiplication. The projection operator is given by $\mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,Y)^{-1}$. Suppose that $k_1(x,y) = \varphi_1(x-y)$, $k_2(x,y) = \varphi_2(x-y)$, then the functional space generated by $k_1 * k_2$ is

$$\mathcal{H}_k = \Big\{ \sum_{1 \leq m \leq N_x} a^m (k(\cdot, x^m)) \Big\}, \tag{3.4.7}$$

where $k(x,y) := \Big(\varphi_1 * \varphi_2\Big)(x-y)$ is the convolution of the two kernels.

3.4.5 Piped kernels

Another important operation is referred to here as "piping kernels" and provides yet another route for generating new kernels in a natural and explicit way. It is denoted by $k_1|k_2$ and corresponds to define the projection operator (3.2.3) as follows:

$$\mathcal{P}_k(X,Y,Z) = \mathcal{P}_{k_1}(X,Y,Z)\pi_1(X,Y) + \mathcal{P}_{k_2}(X,Y,Z)\Big(I_d - \pi_1(X,Y)\Big), \tag{3.4.8}$$

where the projection operator here reads

$$\pi_1(X,Y) := K_1(X,Y)K_1(X,Y)^{-1} = \mathcal{P}_{k_1}(X,Y,X).$$

This operation splits the projection operator $\mathcal{P}_k(X,Y,Z)$ into two parts. The first part is handled by a single kernel, while the second kernel handles the remaining error. From a mathematical standpoint point, this is equivalent to a functional Gram-Schmidt orthogonalization process of both functional spaces $\mathcal{H}^x_{k_1}$, $\mathcal{H}^x_{k_2}$, and the corresponding functional space defined by (3.4.8) is, after (3.2.10),

$$\mathcal{H}^x_k = \Big\{ \sum_{1 \leq m \leq N_x} a^m k_1(\cdot, x^m) + \sum_{1 \leq m \leq N_x} b^m k_2(\cdot, x^m) \Big\}. \tag{3.4.9}$$

Hence, this doubles up the coefficients (3.5.1). We define its inverse concatenating matrix

$$K^{-1}(X,Y) = \Big(K_1(X,Y)^{-1}, K_2(X,Y)^{-1}(I_{N_x} - \pi_1(X,Y))\Big) \in \mathbb{R}^{2N_y \times N_x}. \tag{3.4.10}$$

The kernel matrix associated to a "piped" kernel pair is then

$$K(X,Y) = \Big(K_1(X,Y), K_2(X,Y)\Big) \in \mathbb{R}^{N_x \times 2N_y}. \tag{3.4.11}$$

3.4.6 Piping scalar product kernels: an example with a polynomial regression

Let $S : \mathbb{R}^D \mapsto \mathbb{R}^N$ be given by a family of N basis functions φ_n, i.e. $S(x) = \Big(\varphi_1(x), ..., \varphi_N(x)\Big)$ and consider the following kernel, called dot product kernel (which is conditionally positive-definite):

$$k_1(x,y) := < S(x), S(y) > . \tag{3.4.12}$$

Now, given any positive kernel $k_2(x,y)$, consider a "pipe" kernel $k_1|k_2$. In particular, such a construction is very useful with a polynomial basis function $S(x) = \Big(1, x_1, ...\Big)$: it consists of a classical polynomial regression, allowing to perfectly match moments of distributions, since the remaining error is handled by the second kernel.

Table 3.8: First four rows coefficients matrix

5.5258947
8.5381751
4.5974094
-0.6819433

3.4.7 Neural networks viewed as kernel methods.

Our setup describes the strategies developed in deep learning theory, which are based on neural networks. Namely, we consider any feed-forward neural network of depth M, defined by

$$z_m = y_m g_{m-1}(z_{m-1}) \in \mathbb{R}^{N_m}, \qquad y_m \in \mathbb{R}^{N_m \times N_{m-1}}, \qquad z_0 = y_0 \in \mathbb{R}^{N_0},$$

in which $y_0, \dots, y_M$ are considered as weights and g_m as prescribed activation functions. By concatenation, we arrive at the function

$$z_M(y) = y_M z_{M-1}(y_0, \dots, y_{M-1}) : \mathbb{R}^{N_0 \times \dots \times N_M} \mapsto \mathbb{R}^{N_M}.$$

This neural network is thus entirely represented by the kernel composition

$$k(y_m, \dots, y_0) = k_m\Big(y_m, k_{m-1}(\dots, k_1(y_1, y_0))\Big) \in \mathbb{R}^{N_m \times \dots \times N_0}$$

where $k_m(x, y) = g_{m-1}(xy^T)$, indeed $z_M(y) = y_M k(y_{M-1}, \dots, y_0)$.

3.5 Discrete differential operators

3.5.1 Coefficient operator

We start this section by further analyzing the projection operator (3.2.6). We can interpret this operator in a basis function setting:

$$f_z := K(Z, Y)c_y, \quad c_y := K(X, Y)^{-1} f(X) \in \mathbb{R}^{N_y \times D_f}. \tag{3.5.1}$$

The coefficients c_y corresponds to the representation of f in a basis of functions

$$f_z := \sum_{n=1}^{N_y} c_y^n K(Z, y^n), \tag{3.5.2}$$

Coefficients are matrices also, having size $N_y \times D_f$, except for some composite kernels. The table 3.8 shows the first four coefficients of the test function f_z.

3.5.2 Partition of unity

For any $Y \in \mathbb{R}^{N_y \times D}$, consider the projection operator (3.2.6), and the following vector-valued function:

$$\phi : Y \mapsto \Big(\phi^1(Y), \dots, \phi^{N_x}(Y)\Big) = K(Y, X)K(X, X)^{-1} \in \mathbb{R}^{N_y \times N_x}. \tag{3.5.3}$$

that corresponds to the projection operator $\mathcal{P}_k(X, X, Y)$. On every point x^n, one computes (without considering regularization terms)

$$\phi(x^n) := \Big(0, \dots, 1, \dots, 0\Big) = \delta_{n,m}, \tag{3.5.4}$$

where $\delta_{n,m} = 1$ if $n = m$, 0 else. For this reason, we call $\phi(x)$ a partition of unity. The figure 3.4 illustrates partitions functions.

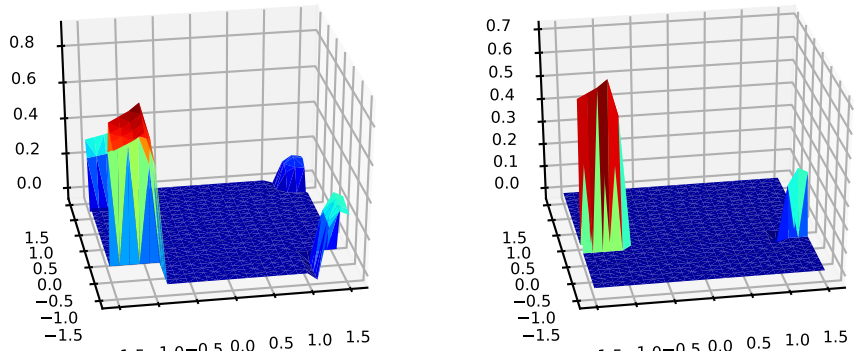


Figure 3.4: Four partitions of unity functions

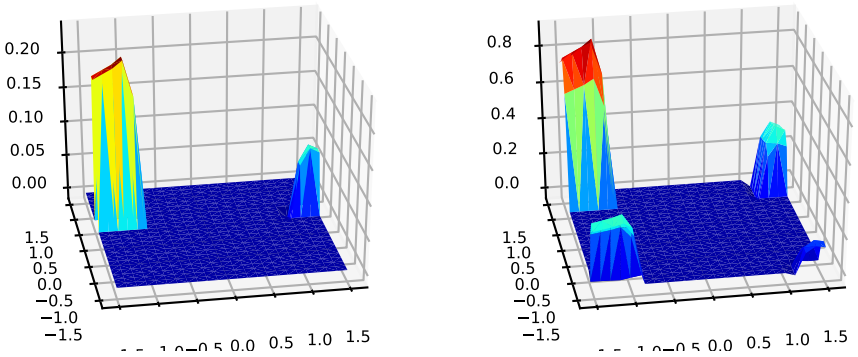


Figure 3.5: Four partitions of unity functions

3.5.3 Gradient operator

For any positive-definite kernel k, and points X, Y, Z, we define ∇ operator as the 3-tensor:

$$\nabla_k(X,Y,Z) = \Big(\nabla_z k\Big)(Z,Y)K(X,Y)^{-1} \in \mathbb{R}^{D\times N_x\times N_z},$$

where $\Big(\nabla_z k\Big)(Z,Y) \in \mathbb{R}^{D\times N_x\times N_y}$ is interpreted as a three-tensor. The gradient of any vector valued function f, is computed as

$$(\nabla f)(Z) \sim (\nabla_k)(Z,Y,Z)f(X) \in \mathbb{R}^{D\times N_z\times D_f},$$

where we omit the dependence $\nabla_k(X,Y,Z)$ for concision. Observe that maps, introduced in the section 3.2.2, modify the operator ∇_k as follows:

$$\nabla_{k\circ S}(X,Y,Z) := (\nabla S)(Z)\Big(\nabla_1 k)(S(Z),S(Y)\Big)K(S(X),S(Y))^{-1}, \tag{3.5.5}$$

where $\Big(\nabla_1 k\Big)(Z,Y) \in \mathbb{R}^{D\times N_z\times N_y}$ is interpreted as a three-tensor, as is $(\nabla S)(Z) := (\partial_d S^j)(Z^{n_z}) \in \mathbb{R}^{D\times D\times N_z}$, representing the Jacobian of the map S, and the multiplication holds for the two first indices.

Two-dimensional example. First we compare the original and extrapolated functions in the figure 3.6:

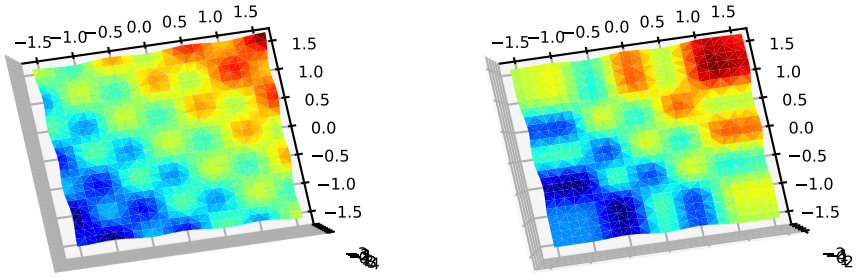


Figure 3.6: Comparison between original values (left) and predicted values (right)

The figures 3.7 and 3.8 illustrate a corresponding derivative and compare it to the original one for the first and second dimensions respectively.

3.5.4 Divergence operator

We define the ∇^T operator as the transpose of the 3-tensor operator ∇:

$$< \nabla_k(X,Y,Z)f(X), g(Z) >=< f(X), \nabla_k(X,Y,Z)^T g(Z) > .$$

Hence, as the left-hand side is homogeneous with, for any smooth function f and vector fields g, and $\nabla\cdot$ denotes the divergence operator

$$\int (\nabla f)\cdot g d\mu = -\int f\nabla\cdot(gd\mu). \tag{3.5.6}$$

The ∇^T operator is thus consistent with the divergence operator

$$\nabla_k(X,Y,Z)^T f(Z) \sim -\nabla\cdot(f\mu)(x)$$

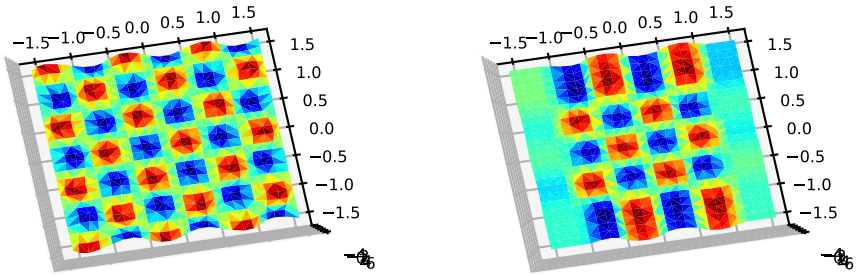


Figure 3.7: Gradient operator. First dimension. Left original, right computed.

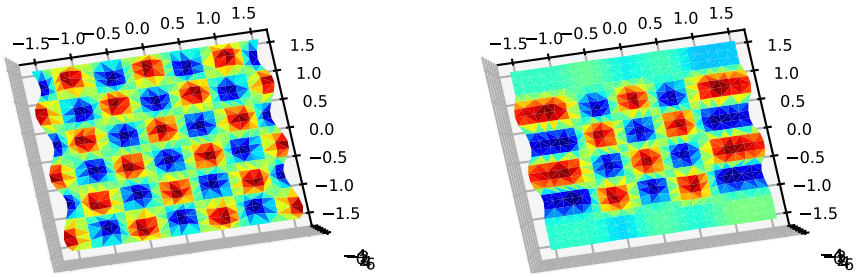


Figure 3.8: Gradient operator. Second dimension. Left original, right computed.

To compute this operator, we proceed as follows: starting from the definition of the gradient operator (3.5.5), we define, for any $f(X) \in \mathbb{R}^{N_x \times D_f}$,$g(Z) \in \mathbb{R}^{D \times N_z \times D_f}$

$$< \Big(\nabla_z K\Big)(Z,Y) K(X,Y)^{-1} f_x, g_z >=< f_x, K(X,Y)^{-T} \Big(\nabla_z K\Big)(Z,Y)^T g_z >$$

Thus the operator $\nabla_k(X,Y,Z)$ is defined as:

$$\nabla_k(X,Y,Z)^T = K(X,Y)^{-T} \Big(\nabla_z K\Big)(Z,Y)^T \in \mathbb{R}^{N_x \times N_z D},$$

where $\nabla_z K(Z,Y)^T \in \mathbb{R}^{N_y \times (N_z D)}$ is the transpose of the matrix $\nabla_z K(Z,Y)$.

A two-dimensional example.

In the figure 3.9 we illustrate the material in this section by comparing $\nabla_k(X,Y,Z)^T \nabla_k(X,Y,Z) f(X)$ to $\Delta_k(X,Y) f(X)$; see the next section.

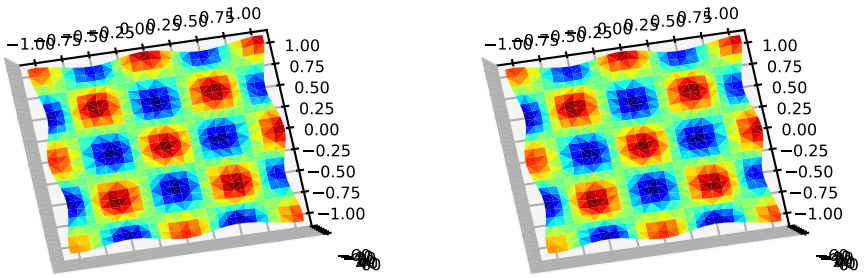


Figure 3.9: Comparison of the outer product of the gradient to Laplace operator

3.5.5 Laplace operator

We define the Laplace operator as the matrix

$$\Delta_k(X,Y) = \Big(\nabla_k(X,Y,X)^T\Big)\Big(\nabla_k(X,Y,X)\Big) \in \mathbb{R}^{N_x \times N_x}.$$

This operator is not consistent with the "true" Laplace operator, but is instead consistent with (3.5.6)

$$-\nabla \cdot (\nabla f \mu).$$

Illustration for this section is done in the figure 3.9.

3.5.6 Inverse Laplace operator

This operator is simply the pseudo-inverse of the Laplacian $\Delta_k(X,Y) \in \mathbb{R}^{N_x \times N_x}$.

A two-dimensional example.

Figure 3.10 compares $f(X)$ with $\Delta_k(X,Y)^{-1} \Delta_k(X,Y) f(X)$. This latter operator is a projection operator (hence is stable).

We also compute the operator $\Delta_{k,x,y,z} \Delta^{-1}_{k,x,y,z} f(X)$ figure 3.11, to check that pseudo-inversion commutes, as highlighted by the following computations:

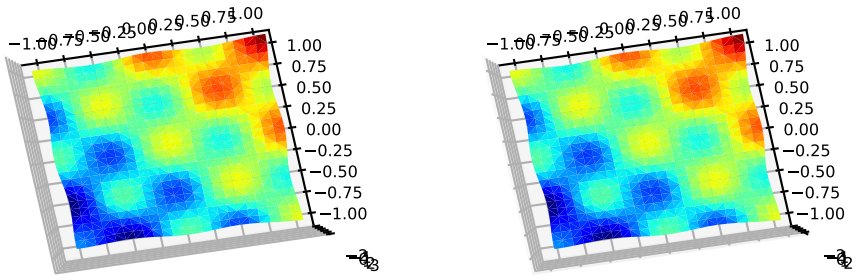


Figure 3.10: Comparison between original function to the product of Laplace and its inverse

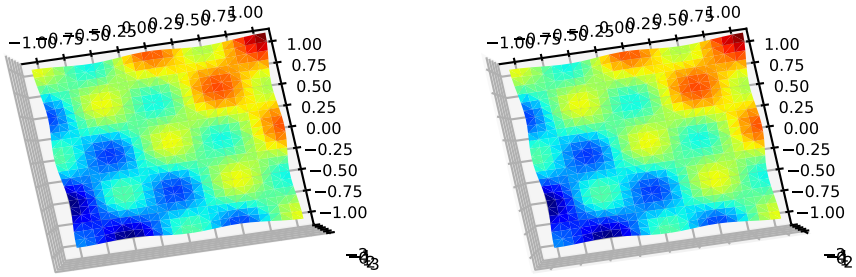


Figure 3.11: Comparison between original function and the product of the inverse of the Laplace operator and the Laplace operator

3.5.7 Integral operator - inverse gradient operator

The following operator ∇_k^{-1} is an integral-type operator

$$\nabla_k^{-1} = \Delta_k^{-1}\nabla_k^T \in \mathbb{R}^{N_x \times DN_z}.$$

It can be interpreted as a matrix, computed first considering $\nabla_k(X,Y,Z) \in \mathbb{R}^{D\times N_z \times N_x}$, down casting it to a matrix $\mathbb{R}^{DN_z \times N_x}$ before performing a least-square inversion. This operator acts on any 3-tensor $v_z \in \mathbb{R}^{D\times N_z \times D_{v_z}}$, and outputs a matrix

$$\nabla_k^{-1}(X,Y,Z)v_z \in \mathbb{R}^{N_x \times D_{v_z}}, \quad v_z \in \mathbb{R}^{D\times N_z \times D_{v_z}}$$

The operator ∇_k^{-1} corresponds to the minimization procedure:

$$h(X) := \arg \inf_{h\in\mathbb{R}^{N_x \times D_{v_z}}} \|\nabla_k(X,Y,Z)h - v_z\|^2_{D,N_z,N_x}.$$

A two-dimensional example.

In the figure 3.12 we show that $(\nabla)(\nabla)^{-1}f(X)$ coincides with $f(X)$. Observe however that this latter operation is not equivalent to Figure 3.13, which uses Z as extrapolation set.

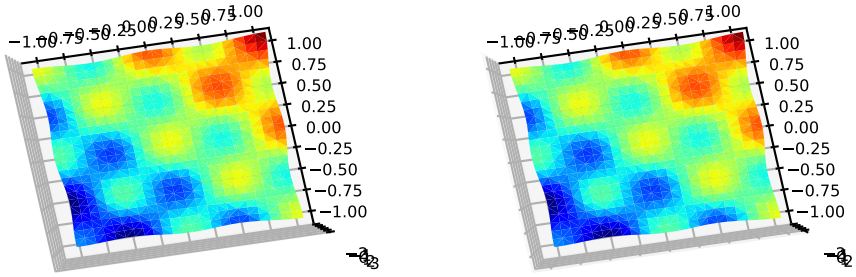


Figure 3.12: Comparison between original function to the product of the gradient operator and its inverse

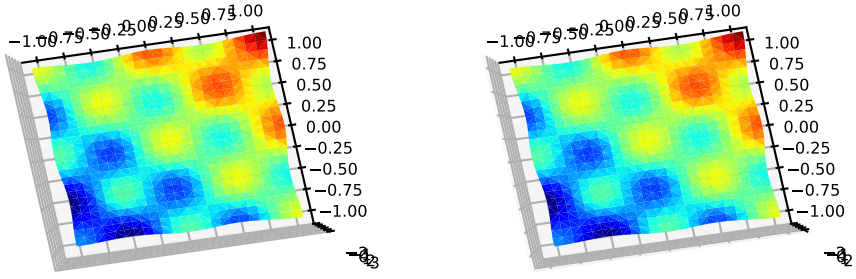


Figure 3.13: Comparison between original function to the product of the inverse of the gradient operator and the gradient operator

3.5.8 Integral operator - inverse divergence operator

The following operator $(\nabla_k^T)^{-1}$ is another integral-type operator of interest. We define the $(\nabla^T)^{-1}$ operator as the pseudo-inverse of the ∇^T operator:

$$(\nabla_k^T(X,Y,Z))^{-1} = \nabla_k(X,Y,Z)\Delta_k(X,Y,Z)^{-1}.$$

A two-dimensional example.

We compute $\nabla_k(X,Y,Z)^T(\nabla_k^T(X,Y,Z))^{-1} = \Delta_k(X,Y,Z)\Delta_k(X,Y,Z)^{-1}$. Thus, the following computations should give comparable results as those obtained in the section concerning the inverse Laplace operator; see section 3.5.6.

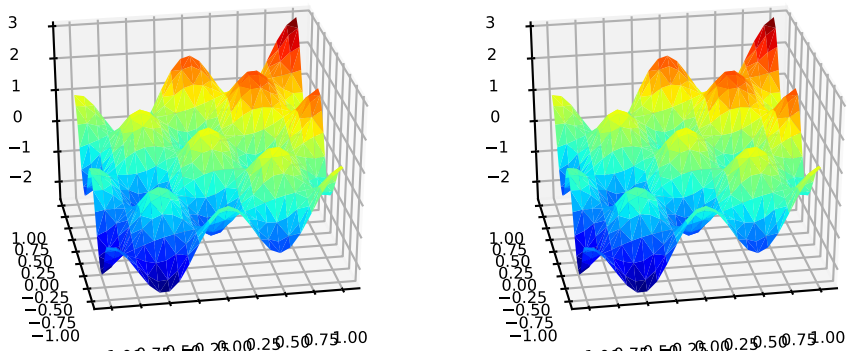


Figure 3.14: Comparison betwenn the product of the divergence operator and its inverse and the product of Laplace operator and its inverse

3.5.9 Leray-orthogonal operator

We define the Leray-orthogonal operator as

$$L_k(X,Y)^{\perp} := \nabla_k(X,Y)\Delta_k(X,Y)^{-1}\nabla^T_{k,x,y,x} = \nabla_k(X,Y,Z)\nabla_k(X,Y,Z)^{-1}.$$

This operator acts on any vector field $f(Z) \in \mathbb{R}^{D\times N_z\times D_f}$, down casting it, performing a matrix multiplication and producing a three-tensor:

$$L_k(X,Y,Z)^{\perp} f(Z) \in \mathbb{R}^{D\times N_z\times D_f}.$$

In the figure 3.15, we compare this operator to the original function $(\nabla f)(Z)$:

3.5.10 Leray operator and Helmholtz-Hodge decomposition

We define the Leray operator as

$$L_k(X,Y,Z) := I_d - L_k(X,Y,Z)^{\perp} = I_d - \nabla_k(X,Y,Z)\Delta_k(X,Y,Z)^{-1}\nabla_k(X,Y,Z)^T,$$

where I_d is the identity. Hence, we get the following orthogonal decomposition of any tensor fields:

$$v_z = L_k(X,Y,Z)v_z + L_k(X,Y,Z)^{\perp}v_z, \quad < L_k(X,Y,Z)v_z, L_k(X,Y,Z)^{\perp}v_z >_{D,N_z,D_v} = 0.$$

This agrees with the Helmholtz-Hodge decomposition, decomposing any vector field into an orthogonal sum of a gradient and a divergence free vector:

$$v = \nabla h + \zeta, \quad \nabla\cdot\zeta = 0, \quad h := \Delta^{-1}\nabla\cdot v$$

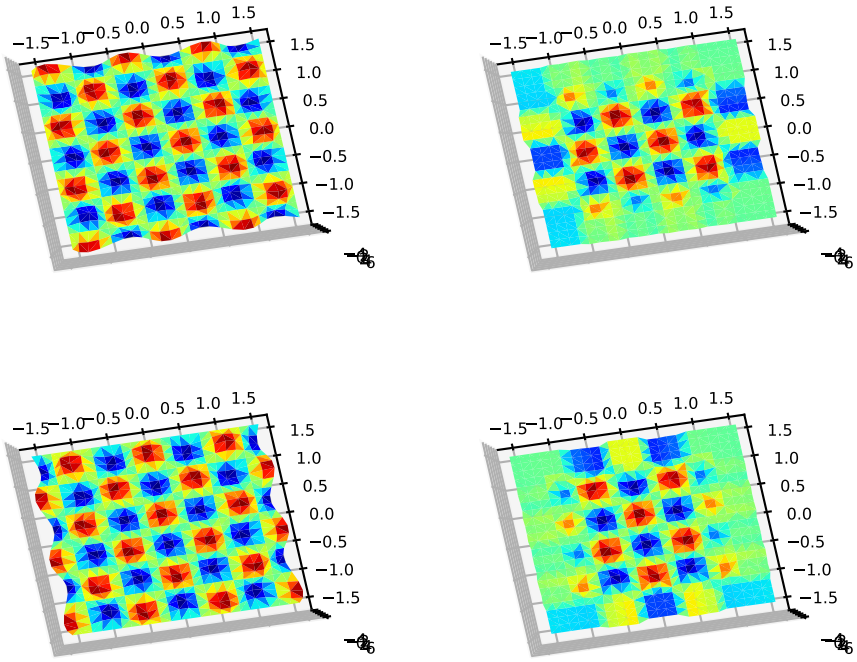


Figure 3.15: comparing f(z) and the transpose of the Leray operator on each direction

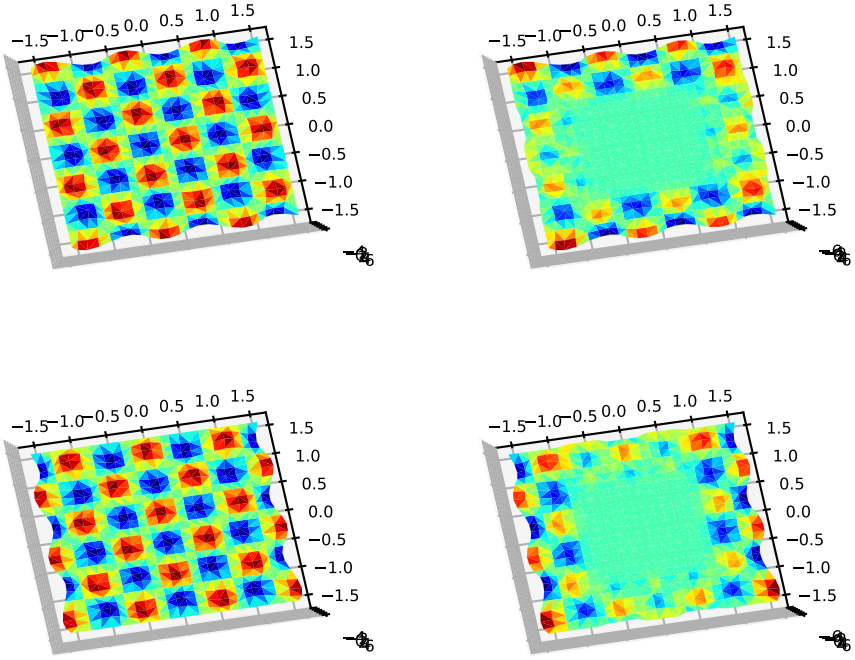
Here we have also an orthogonal decomposition from a numerical point of view:

$$v_z = \nabla_k(X,Y,Z)h_x + \zeta_z, \quad h_x := \nabla_k(X,Y,Z)^{-1}v_z, \quad \zeta_z := L_k(X,Y,Z)v_z,$$

satisfying numerically

$$\nabla_k(X,Y,Z)^T\zeta_z = 0, \quad \langle \zeta_z, \nabla_k(X,Y,Z)h_x \rangle_{D\times N_z \times D_f} = 0.$$

In the following figure, we compare this operator to the original function $(\nabla f)(Z)$ in the figure **??**.



3.6 A kernel-based clustering algorithm

In this section we describe a kernel-based clustering algorithm. This algorithm, already presented with a toy example in the section 2.9, is also benchmarked in a forthcoming section devoted to more concrete problems, see Chapter 6.

3.6.1 Distance-based unsupervised learning machines

Distance-based minimization algorithms can be thought as finding a minimum of a distance between set of points $d_k(X, Y)$, defining equivalently a distance between discrete measures μ_x, μ_y. Within this setting, minimization problem can be expressed as

$$Y = \arg \inf_{Y \in \mathbb{R}^{N_y \times D}} d_k(X, Y)$$

Suppose that this last problem is well-posed, assuming that the distance functional is convex[1]. Once the cluster set $Y := \left(y^1, \dots, y^{N_y}\right)$ is computed, then one can define the index function $\sigma_d(w, Y) := \{j : d(w, y^j) = \inf_k d(w, y^k)\}$, as for (2.3.2). One we can extend naturally this function, defining a map

$$\sigma_d(Z, Y) := \{\sigma_d(z^1, Y), \dots, \sigma_d(z^{N_z}, Y)\} \in [1, \dots, N_y]^{N_z}, \tag{3.6.1}$$

that acts on the indices of the test set Z. This allows to compare this prediction to a given, user-desired, partition of $f(Z)$, if needed.

Note that the function $\sigma_d(Z, Y)$ is surjective (many-to-one). Hence we can define its injective (one-to-many) inverse, $\sigma_d(Z, Y)^{-1}(n)$, describing those points of the test set attached to one y^n.

[1] although most of existing distance are not convex

This construction defines cells, very similarly to quantization, $C^n := \sigma_d(\mathbb{R}^D, y^n)^{-1}(n)$, defining a partition of unity of the space $\mathbb{R}^D$. A last remark: consider, in the context of supervised clustering methods, the training set and its values $X, f(X)$ and the index map $\sigma_d(X, Y) \in [1, ..., N_x]^{N_y}$ defined above. One can always define a prediction on the test set Z as

$$f_z := f(X^{\sigma(Y^{\sigma(Z,Y)},X)}),$$

showing that distance minimization unsupervised algorithm naturally defines supervised ones.

3.6.2 Sharp discrepancy sequences

Our kernel-based algorithm for clustering can be described as follows:

- The unsupervised algorithm aims to solve the minimization problem (3.6.1) with the discrepancy functional (3.2.5). This procedure is separated into two main steps.
  - First solve a discrete version of (3.6.1), namely

    $$X^\sigma = \arg\inf_{\sigma\in\Sigma} d(X, X^\sigma),$$

    where Σ denotes the set of all subsets from $[1, ..., N_y] \mapsto [1, ..., N_x]$. This minimization problem is described Chapter 4.1.2.2.
  - Depending on kernels, this step is completed by a simple gradient descent algorithm. The initial state for this minimization is chosen to be X^σ.
  - The resulting solution Y is named **sharp discrepancy sequences**.
- The supervised algorithm consists then simply to compute the projection operator (3.2.6), that we recall here.

  $$f_z := \mathcal{P}_k(X, Y, Z)f(X)$$

  using the python function (3.2.3), where the *weight set* Y is taken as the sharp discrepancy sequence computed above.

3.6.3 Python functions

- The unsupervised clustering algorithm is given by the Python function

  $$sharp\_discrepancy(X, Y = [], N_y = 0, set\_codpy\_kernel = None, rescale = False, nmax = 10)$$

- Let $X \in \mathbb{R}^{N_x\times D}$, $Y \in \mathbb{R}^{N_y\times D}$ any two distributions of points and $k(x, y)$ a positive-definite kernel. The following Python function

  $$codpy.alg.match(Y, X, nmax = 10)$$

  approximate the following problem

  $$\arg\inf_{Y\in\mathbb{R}^{N_y\times D}} d_k(X, Y)^2$$

  via a simple descent algorithm: starting from the input distribution Y, the algorithm performs $nmax$ steps of a descent algorithm and output the resulting distribution.
- The computation of index associations (3.6.1), that is the function $\sigma_{d_k}(X, Y)$, is given by

  $$alg.distance\_labelling(X, Y, set\_codpy\_kernel = None, rescale = True).$$

  This function relies on the distance matrix $D(X, Y)$; see 3.2.

3.6.4 Impact of sharp discrepancy sequences on discrepancy errors

The figure 2.9 presented a first illustration of the impact of computing discrepancy errors on several toy "blob" examples. In this paragraph, we fix the number of "blobs" to two, and the number of generated points N_x to 100. We then follow the test methodology of the section 2.9, re-running all tests with scenarios for N_y covering [0,100]. The figure 3.16 compare the results for discrepancy errors of the three methods. One can check visually that discrepancy errors is zero, whatever the clustering method is, when the number of clusters N_y tends to N_x. Note also that our kernel clustering methods shows quite good inertia performance indicators. This is surprising, as our method is based on discrepancy error minimization, not inertia. An interpretation could be that the inertia functional is bounded by the discrepancy error one.

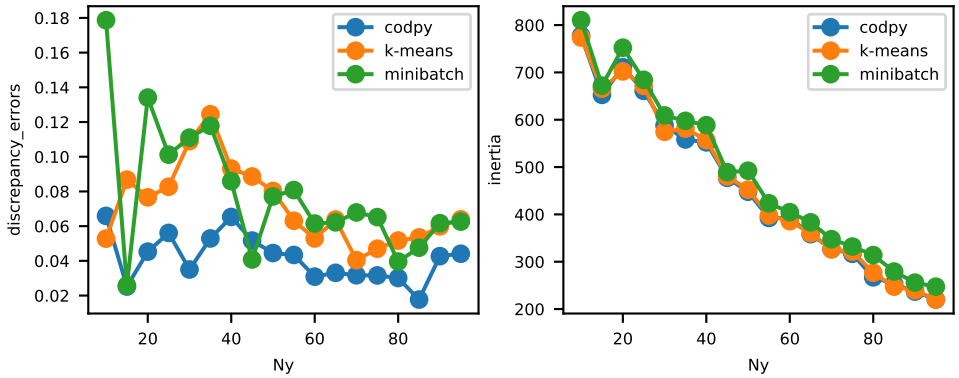


Figure 3.16: benchmark of discrepancy errors and inertia

3.6.5 A study of the discrepancy functional

As stated in the previous section, we first compute a discrete minimizing problem and denote X^σ its solution. We eventually complete this step with a simple gradient descent algorithm. This section explains and motivate this choice. Indeed, the minimizing properties $d_k(X,Y)$ relies heavily on the kernel definition $k(x,y)$, and we face an alternative, depending on regularity of kernels, that we illustrate numerically in this section:

- If the kernel is smooth, then the distance functional $d_k(X,Y)$ also is, and a descent algorithm based on gradients computations is an efficient option.
- If the kernel is only continuous, or piecewise derivable, then we assume that the minimum is attained by the discrete minimum solution X^σ.

Hence, we study in this section the effect of some classical kernel over this functional for a better understanding. To that aim, let us produce some random distributions $x \in \mathbb{R}^{N_x}$ in one dimension, we will study then for three kernels the following functionals:

$$\mathbb{R}^{N_y} \ni y \mapsto d_k(x,y),$$

$$\mathbb{R}^{N_y \times 2} \ni Y = (y^1, y^2) \mapsto d_k(x,Y).$$

We generate uniform random variables $x \in \mathbb{R}^{N_x}, y \in \mathbb{R}^{N_y}$ and $Y = (y^1, y^2) \in \mathbb{R}^{N_y \times 2}$.

A example of smooth kernels: Gaussian. We start our study of the discrepancy functional with a Gaussian kernel. The Gaussian kernels is a family of kernels based upon the following kernel, generating functional spaces of smooth functions.

$$k(x,y) = \exp(-(x-y)^2)$$

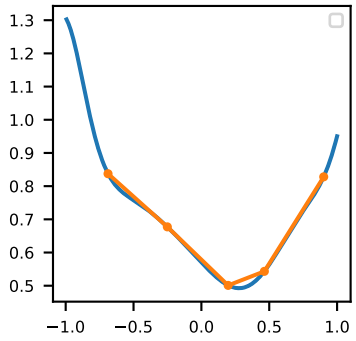


Figure 3.17: Distance functional for the Gaussian kernel

The following picture plot the function $y \mapsto d_k(x, y)$ in blue. We also output the function $d_k(x, x^n)$, $n = 1 \dots N_x$, figure 3.17 to illustrate that this functional is neither convex nor concave.

We see that the functional $y \mapsto d_k(x, y)$ admits a minimum close to $y = \frac{1}{2}$ as expected. The next picture 3.18 plots $y := (y^1, y^2) \mapsto d_k(x, y)$

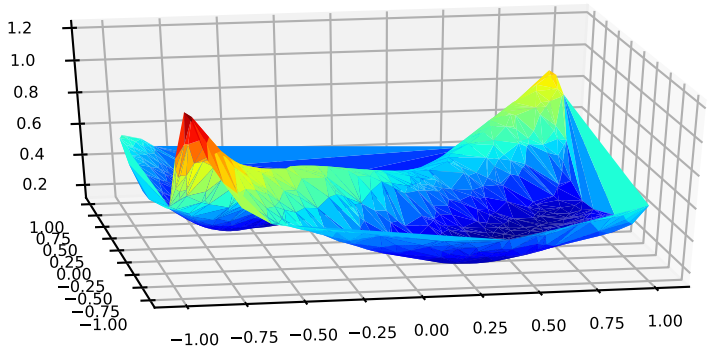


Figure 3.18: Distance functional for the Gaussian kernel

We see that this functional admits two minima. This reflects the fact that the functional $y \mapsto d_k(x, y)$ is invariant by permutation of the indices of y.

An example of Lipschitz continuous kernels: RELU. Let us now study a kernel generating functional spaces having less regularity. The norm kernels is a family of kernels based upon the following kernel, generating functional spaces of bounded variation functions.

$$k(x, y) = 1 - |x - y|.$$

Indeed, in view of Figure 3.19 it is clear that the function $y \mapsto d_k(x, y)$ is piecewise differentiable. Hence in some situations, the functional $d_k(x, y)$ might have an infinity of solutions (here on the "flat" segment). The next figure 3.20 plots $y := (y^1, y^2) \mapsto d_k(x, y)$ for the two dimensional case.

A example of continuous kernel: Matern. The Matern kernel is based upon the following kernel, generating usually functional spaces of continuous functions.

$$k(x, y) = \exp(-|x - y|)$$

Indeed, in the figure 3.21 we see that the functional $y \mapsto d_k(x, y)$ is here almost everywhere concave,

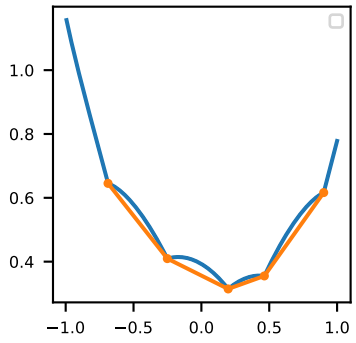


Figure 3.19: Distance functional for the norm kernel

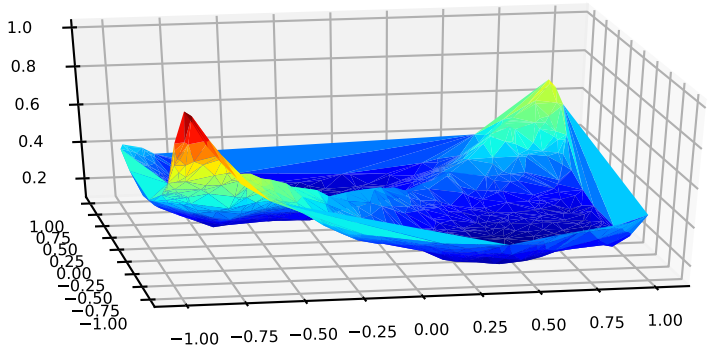


Figure 3.20: Distance functional for the norm kernel

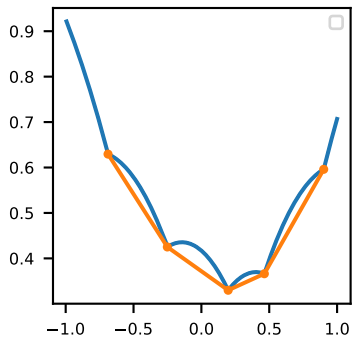


Figure 3.21: Distance functional for the Matern kernel

and a gradient-descent algorithm can't give good results in this case. The next picture 3.22 plots the two dimensional case $y := (y^1, y^2) \mapsto d_k(x, y)$

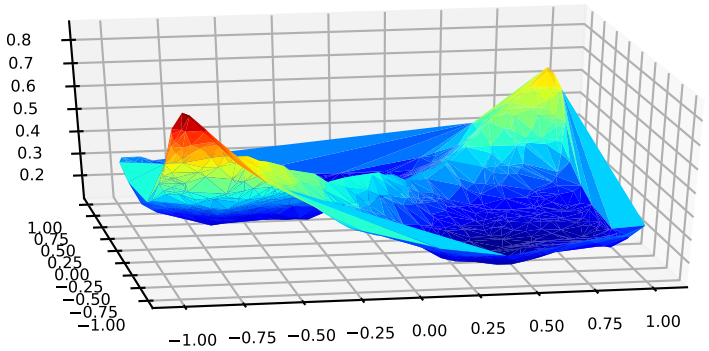


Figure 3.22: Distance functional for the Matern kernel

Chapter 4

Kernel methods for optimal transportation

4.1 Discrete ordering algorithms

We describe in this section discrete ordering algorithms based on the kernel-based distance.

4.1.1 Linear Sum Assignment Problems (LSAP)

In this section, we describe a ordering algorithm, which we discovered while working in the development of algorithms based on the theory of Reproducing Kernel Hilbert Space (RKHS). This algorithm is motivated by the "linear assignment value' ' problem, and is used in a number of our Academic and industrial applications;

The Linear Sum Assignment problem (LSAP) is a fundamental problem of combinatorial optimization. It is an old and well-documented problem [1], which leads to a large number of important industrial applications. It has been solved in the early 30's by H.W. Kuhn and is often called the Hungarian method in order to highlight that it derives from two older[2] results by two Hungarians mathematicians, Koenig (Math Ann 77:453 465, 1916) and Egervry (Mat Fiz Lapok 38:1628, 1931). In particular, the approach adopted in the present text relies on the LSAP. As our library embeds a different algorithm than the Hungarian one for performance purposes, we describe and exhibit our interface on this problem in the next section.

Our basic idea here is to use a ordering algorithm and order any two sets of points $x \in \mathbb{R}^{N_x \times D}$, $y \in \mathbb{R}^{N_y \times D}$ with respect to the *discrepancy distance matrix* $d_k(x, y) \in \mathbb{R}^{N_x \times N_y}$ associated with a given kernel, introduced in (**??**). Without loss of generality, suppose that $N_x > N_y$. The above distance allows us to compute a permutation σ with length N_y, with the help of the LSAP algorithm. Then, we reorder and output the distribution x with this permutation. Our motivation comes from optimal transportation; see for instance [48] for a review of optimal transport. Indeed, once computed, the map

$$x^n \mapsto y^{\sigma_n}$$

defines an optimal map, with respect to the kernel-induced *discrepancy distance* $d_k(x, y)$, described in section **??**, transporting the measure $\mu_x := \sum_n \delta_{x^n}$ into the measure $\mu_y := \sum_n \delta_{y^{\sigma_n}}$. This equivalence between the LSAP and the Monge-Kantorovich problem, used since some time, has only been recently rigorously proven in [6].

[1] see the wikipedia page https://en.wikipedia.org/wiki/Assignment_problem

[2] These algorithms might be credited to Jacobi posthumous papers [17].

Table 4.1: a 4x4 random matrix

0	1	2	3
0.8231103	0.0261180	0.2107706	0.6184218
0.0982845	0.6201313	0.0538902	0.9606541
0.9804294	0.5211277	0.6365533	0.7647569
0.7649553	0.4176856	0.7688053	0.4232018

4.1.1.1 Description of the problem

Linear sum assignment problems can be well described using graph theory : it consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is a minimum.

To make this definition clear, let us introduce some notations : consider any real-valued matrix $M \in \mathbb{R}^{N_x \times N_y}$, called a *cost* matrix. A linear assignment problem consist in finding a permutation $\sigma : [1 \dots \min(N_x, N_y)] \mapsto [1 \dots \min(N_x, N_y)]$ such that

$$\sigma = \arg\inf_{\sigma \in \Sigma} \sum_{n \leq \min(N_x, N_y)} M(n, \sigma(n)),$$

where Σ holds here for the set of all permutations. Another equivalent formulation is the following one

$$\sigma = \arg\inf_{\sigma \in \Sigma} \sigma \cdot M$$

where $A \cdot B$ holds here for the Frobenius scalar product and Σ is the set of permutations, using a matrix representation : $\sigma \in \mathbb{R}^{N_x \times N_y}$ $\sum_n \sigma(n,m) = \sum_m \sigma(n,m) = 1, \sigma(n,m) \in \{0,1\}$.

Let us give a quick illustration for better understanding to this problem. We fill out a matrix with random values in table 4.1, and output also its cost, that is $Tr(M)$.

```
## total cost: 2.5029967458342304
```

Then we compute the permutation σ. The python interface to this function is simply $\sigma = \text{lsap}(M)$.

```
## permutation:  [1, 0, 2, 3]
```

We reorder $\tilde{M} = \sigma M$, and we ouput the new cost after ordering, that is $Tr(\tilde{M})$. we check in the following that the lsap algorithm decreased the total cost.

```
## total cost: 1.184157542154346
```

4.1.1.2 Description of the algorithm

Our ordering algorithm is quite straightfoward. Consider any two set of points $x \in \mathbb{R}^{N_x \times D}$, $y \in \mathbb{R}^{N_y \times D}$, a kernel $k(x,y)$, and the *discrepancy* distance $d_k(x,y) \in \mathbb{R}^{N_x \times N_y}$ introduced in @ref{distance-matrices}. Suppose wlog $N_y < N_x$. We compute first the permutation

$$\sigma = \text{lsap}(d_k(x,y)), \quad \sigma \in \mathbb{R}^{N_y}$$

Then output the reordered set

$$x^\sigma := (x^{\sigma_1}, \dots, x^{\sigma_{N_y}}), \quad y, \quad \text{if } N_x \geq N_y$$

4.1.1.3 Description of the python function

The ordering algorithm takes two distributions in input, and output a permutation of one of its input data (x or y), as well as the permutation σ:

Table 4.2: a random gaussian distribution x

0	1	2	3	4
5.216782	5.507742	5.973072	4.643358	4.211705
4.767422	4.152499	3.251747	5.808072	6.239795
5.560636	4.457183	4.515850	6.319828	6.445540
5.051817	6.628679	4.634000	4.975758	4.173435

Table 4.3: a random uniform distribution y

0	1	2	3	4
0.4343886	0.2460579	0.8616407	0.0200226	0.4508267
0.0474229	0.4977275	0.8587740	0.3348157	0.9015900
0.1228876	0.1574337	0.7873853	0.6649391	0.7202042
0.5392553	0.4719475	0.9006875	0.3745125	0.5277864

$$x^\sigma, y^\sigma, \sigma = alg.reordering(x, y, set\_codpy\_kernel, rescale, distance = None)$$

This algorithm takes in input the following:

- Two distributions of points having shapes

$$x := (x^1, ..., x^{N_x}) \in \mathbb{R}^{N_x \times D}, \quad y := (y^1, ..., y^{N_y}) \in \mathbb{R}^{N_y \times D}$$

- A positive kernel $k(x, y)$, defined through the input variable set_codpy_kernel. This defines the cost matrix as being $M = d_k(x, y)$, where the distance matrix is defined in **??**.
- Alternatively an optional parameter *distance* taking values among
  - "norm1", in which case the sorting is done accordingly to the Manhattan distance $d(x, y) = |x - y|_1$
  - "norm2", in which case the sorting is done accordingly to the Euclidean distance $d(x, y) = |x - y|_2$
  - "normifty", in which case the sorting is done accordingly to the sup-distance $d(x, y) = |x - y|_\infty$

This function outputs :

- Two distributions x^σ, y^σ having length N_y. If $N_x > N_y$, then $y^\sigma = y$. The case $N_y > N_x$ is symetrical, letting the original distribution x unchanged.
- A permutation σ, represented as a vector $i \mapsto \sigma_i$, $0 \leq i \leq \min(N_x, N_y)$.

4.1.1.4 Illustration for the square matrix case

4.1.1.4.1 A quantitative illustration We show first the results given by our ordering algorithm on a simple example. We generate two distributions of 4 points in $\mathbb{R}^5$. The first is generated by multivariate Gaussian distribution centered in $(5, .., 5)$, the second one by a uniform distribution supported into the unit cube.

We output x in table 4.2 and y in table 4.3

Let us first pick up a kernel k, here a Matern kernel.

Then we compute the distance matrix, and output the transportation cost $\sum_{n=0}^{N} d_k(n, n)$

```
## cost: 11.072754623017211
```

Table 4.4: Matrix after ordering (samples)

0	1	2	3
2.785269	2.730067	2.751811	2.701832
2.707658	2.630662	2.641554	2.619985
3.003731	2.930162	2.939942	2.913812
2.805791	2.742673	2.767694	2.716881

Table 4.5: permutation

0	1	2	3

We output the distance matrix in table **??**.

0	1	2	3
2.785269	2.730067	2.751811	2.701832
2.707658	2.630662	2.641554	2.619985
3.003731	2.930162	2.939942	2.913812
2.805791	2.742673	2.767694	2.716881

We then invoke the ordering algorithm and output the cost after ordering.

```
## cost: 11.072754623017211
```

Finally, we output the distance matrix again after ordering in table 4.4, as well as the permutation σ in table 4.5

One can check that the sum of the diagonal elements has decreased.

4.1.1.4.2 A qualitative illustration This algorithm can be best illustrated in the two-dimensional case.

We first consider the Euclidean distance function $d(x, y) = |x - y|$, in which case this algorithm corresponds to a classical rearrangement, i.e. the one corresponding to the Wasserstein distance. To illustrate this behavior, let us generate a bi-modal type distribution $x \in \mathbb{R}^{N \times D}$ and a random uniform one $y \in [0, 1]^{N \times D}$.

For a convex distance, this algorithm is characterized by a ordering where characteristic lines do not cross each others, as plot in the picture 4.1, plotting both edges $x^i \mapsto y^i$, before and after the ordering algorithm.

Note however that kernels based distance might lead to different permutations. This is due to the fact that kernels defines distance that might not be euclidean. Indeed, kernel distance might not respect the triangular inequality. For instance, the kernel selected above defines a distance equivalent to $d(x, y) = \Pi_d |x_d - y_d|$, and leads to a ordering for which some characteristics should cross

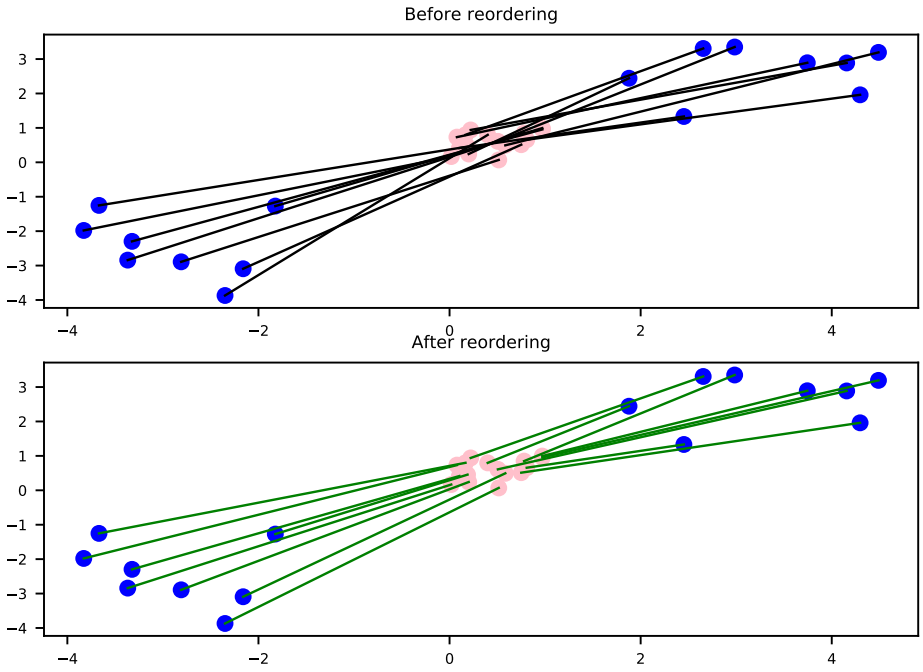
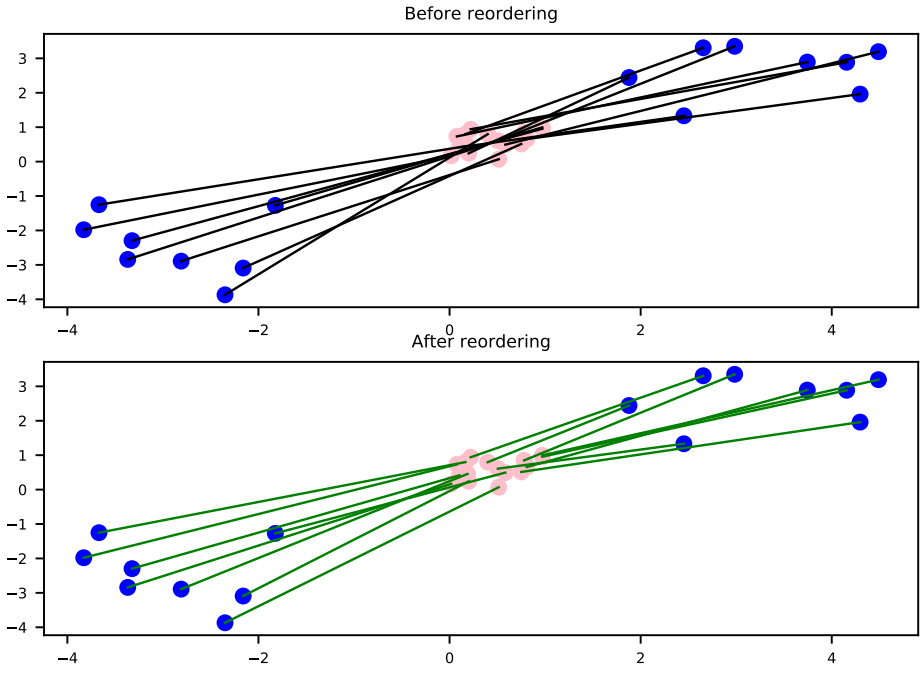


Figure 4.1: LSAP with different input sizes



4.1.2 LSAP extensions

In this section we describe some extensions of the LSAP algorithms that we use in our library.

4.1.2.1 Different input sizes

A first quite straightforward extension of LSAP problem can be found for inputs set of different sizes, wlog $N_y \leq N_x$. The figure 4.2 illustrates the behavior of our LSAP algorithm in this setting

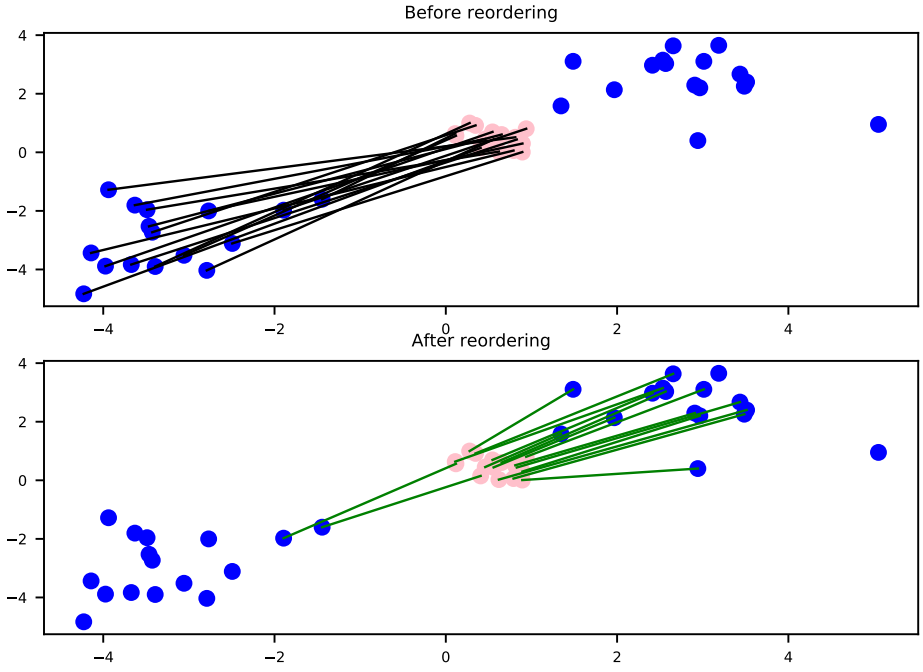


Figure 4.2: LSAP with different input sizes

4.1.2.2 General cost functions and motivations

Consider any real-valued matrix $M \in \mathbb{R}^{N \times N}$. In situations of interests, we consider cost functional $c(M)$ that generalizes the classical cost functional for LSAP problem $c(M) = \sum_n M(n,n)$. Our algorithm generalizes to these cases, finding a permutation $\sigma : [1 \dots N] \mapsto [1 \dots N]$ such that

$$\bar{\sigma} = \arg\inf_{\sigma \in \Sigma} c(M^\sigma), \quad M^\sigma = m(n, \sigma(n))$$

An example of such a LSAP problem extension arised with kernel methods in section 3.6.2. It corresponds to compute the minimum of the discrepancy functional @ref{eq:dk}, for the particular choice where $x^\sigma \subset x$ is a subset of x having length $N_y < N_x$. We used the notations $x^\sigma = (x^{\sigma_1}, \dots, x^{\sigma_{N_y}})$, with $\sigma : [1 \dots N_y] \mapsto [1 \dots N_x]$. In this context, the matrix is defined as $M(n,m) = k(x^n, x^m)$, and the cost function is

$$d_k(x, x^\sigma)^2 = c(M) = \frac{1}{N_x^2} \sum_{n=1,m=1}^{N_x,N_x} M(n,m) + \frac{1}{N_y^2} \sum_{n=1,m=1}^{N_y,N_y} M(\sigma(n),\sigma(m)) - \frac{2}{N_x N_y} \sum_{n=1,m=1}^{N_x,N_y} k(n,\sigma(m)).$$

So that our target minimization problem can be described as finding a permutation $\bar{\sigma}$ such that

$$\overline{\sigma} = \arg \inf_{\sigma:[1...N_y]\mapsto[1...N_x]} c(M^\sigma), \quad M^\sigma(n,m) = k(x^n, x^{\sigma(m)})$$

4.2 Conditional expectation algorithm

4.2.1 Introduction

In this section, we propose a general interface to a python function computing conditional expectations problems in arbitrary dimensions, that we named Pi. We also propose a kernel-based implementation of these problems, which algorithm is described in [24] - [26].

Kernel methods to compute conditional expectations started to be considered a decade ago, see for instance [22]. Indeed, these algorithms are centrals, particularly for finance applications, as they are the heart of pricing technologies. They also have numerous other applications.

Benchmarking such algorithms is a difficult task, as the literature did not provide competitor algorithms to compute conditional expectations to kernel-based methods, for arbitrary dimensions, to our knowledge. Indeed, these algorithms are tightly concerned with the so called *curse of dimensionality*, as we are dealing with arbitrary dimensions algorithms.

However, there is a recent, but impressively fast-growing, literature, devoted to the study of Artificial Intelligence methods (AI), particularly for Finance applications, see [**?**] and ref. therein for instance. In particular, a Neural Networks (NN) approach has been proposed to compute conditional expectation in [**?**] that we can use as benchmark. Hence a first benchmark is conducted in section **??**.

4.2.2 The Pi function

Consider any martingale process $t \mapsto X(t)$, and any positive definite kernel k, we define the operator Π - using python notations -

$$f_{z|x} = \Pi(x, z, f(z) = []) \tag{4.2.1}$$

where

- $x \in \mathbb{R}^{N_x \times D}$ is any set of points generated by a i.i.d sample of $X(t^1)$ where t^1 is any time.
- $z \in \mathbb{R}^{N_z \times D}$ is any set of points, generated by a i.i.d sample of $X(t^2)$ at any time $t^2 > t^1$.
- $f(z) \in \mathbb{R}^{N_z \times D_f}$ is any, optional, function, representing payoff values.

The output is

- if $f(z)$ is let empty, the output $f_{z|x} \in \mathbb{R}^{N_z \times N_x}$ is a matrix, representing a convergent approximation of the stochastic matrix $\mathbb{E}^X(z|x)$.
- if $f(z) \in \mathbb{R}^{N_z \times D_f}$ is not empty, $f_{z|x} \in \mathbb{R}^{N_z \times D_f}$ is a matrix, representing the conditional expectation $f(z|x) := \mathbb{E}^X(f(z)|x)$.

4.3 Polar factorization algorithms

Consider any mapping $S : \mathbb{R}^D \mapsto \mathbb{R}^D$, and a distance function, that is positive, scalar valued, $\mathcal{C}^1$ function $d(\cdot, \cdot)$.

The polar factorization algorithm amounts to find a **scalar, convex** function f, and a volume preserving map $T : \mathbb{R}^D \mapsto \mathbb{R}^D$, satisfying

$$S = \Big(\nabla f\Big) \circ T(y), \quad f \text{ convex}, \quad T_{\#}m = m$$

A volume preserving map is a mapping satisfying $\int \varphi dx = \int \varphi \circ T dx$ for any continuous function φ, dx being the Lebesgue measure. Existence and unicity of this decomposition is discussed in Brenier seminal's paper [**?**].

4.3.1 Discrete Polar factorization and linear sum assignment problem

Let us start from any distributions $x \in \mathbb{R}^{N_x \times D}$, $z \in \mathbb{R}^{N_y \times D}$, and consider a distance function, that is positive, scalar valued, $\mathcal{C}^1$ function $d(\cdot, \cdot)$, and we naturally extend this function to a matrix valued one $d(x, z) \in \mathbb{R}^{N_x \times N_z}$. Let us first make some reminder about optimal transportation type problems.

4.3.2 The Monge-Kantorovitch problem

Polar factorization are linked to the Monge-Kantorovitch problem, that is an optimal transportation one. Consider any two measures μ_x, μ_z, and consider a distance function, that is positive, symmetrical, convex, scalar valued, $\mathcal{C}^1$ function $d(\cdot, \cdot)$.

The following discrete problem is called the **Monge** problem

$$\bar{\gamma} = \inf_{\gamma \in \Gamma} d(x, z) \cdot \gamma$$

where

- $A \cdot B$ denotes the Frobenius scalar matrix product
- Γ denotes the set of all bi-stochastic matrix $\gamma \in \mathbb{R}^{N \times N}$, that is satisfying,

$$\sum_{n=1...N} \gamma_{m,n} = \sum_{n=1...N} \gamma_{n,m} = 1, \quad \gamma_{n,m} \geq 0, \quad \text{for all } n, m = 1, ..., N.$$

This minimization problem owns a dual expression, called the **Kantorovitch** problem

$$\sup_{\varphi, \psi} \sum_{n}^{N} \varphi(x^n) - \psi(z^n), \quad \varphi(x^n) - \psi(z^n) \leq d(x^n, z^n)$$

where φ, ψ are the unknown functions.

Note that any permutation $\sigma : [1 ... N] \mapsto [1 ... N]$ is a stochastic matrix. In particular, the following discrete problem, that is a Linear assignment problem, described in section 4.1.1, consists in a first approach to (4.3.2):

$$\bar{\gamma} = \inf_{\gamma \in \Sigma} d(x, z) \cdot \gamma$$

Indeed, all problems (4.3.2)-(4.3.2)-(4.3.2) are equivalent, see [6].

4.3.3 Motivation: the sampler function

In many applications we would like to fit the scattered data to a given model that best represents them. To be specific, consider any distributions of points $x \in \mathbb{R}^{N \times D}$, representing i.i.d. samples of a random variable X, $z \in \mathbb{R}^{[0,1]^{N \times D}}$, any i.i.d. of the uniform distribution into the unit cube, and suppose that we solved (4.3) in the following, discrete, sense

Table 4.6: one-dimensional bi-modal distribution

samples bimodal distribution
-4.443787
-5.892410
-6.278320
-3.363557
-5.366850
-5.984819

$$x = \Big(\nabla f\Big)(z), \quad f \text{ convex}, \quad x \in \mathbb{R}^{N\times D}, z \in [0,1]^{N\times D}.$$

Then the function

$$y \mapsto \Big(\nabla f\Big)(y),$$

where $y \in \mathbb{R}^{[0,1]^{N_y \times D}}$ provides us with a natural candidate for others i.i.d. realization of the random variable X.

Hence this section illustrates the following python function

$$y = sampler(x, M, seed)$$

that outputs M values $y \in \mathbb{R}^{N\times D}$ of a distribution sharing close statistical properties with the discrete distribution x, that we discuss in the next paragraph.

4.3.3.1 Statistical tests

We exhibit three statistical indicators to support our claims, measuring each some kind of distance between the two distributions x and y. The two first tests are one-dimensional based tests. We check it on every axes. The third one is based on the discrepancy error.

- Kolmogorov-Smirnov based tests. These are one-dimensional tests, based on the cumulative distribution function. The test is

$$\|cdf_x - cdf_z\|_\ell^\infty(\mathbb{R}^N) \geq \frac{c_N}{\sqrt{N}}$$

  where c_N is a confidence level.
- Hellinger distance tests. To measure the closedness of pdf_x to pdf_z which are the PDFs of x and y repectively. The Hellinger distance is defined as

$$\mathcal{H}(x,z) = \frac{1}{\sqrt{2}}||\sqrt{pdf_x} - \sqrt{pdf_z}||_2$$

- Discrepancy errors, defined in section **??**.

4.3.4 One dimensional Examples

4.3.4.1 Bimodal Gaussian distribution

In this section we study a bi-modal gaussian distribution.

We output some values of x in the following lines

Let us call the sampling function, filling up $y \in \mathbb{R}^{M\times 1}$.

We output some values of y in the following lines

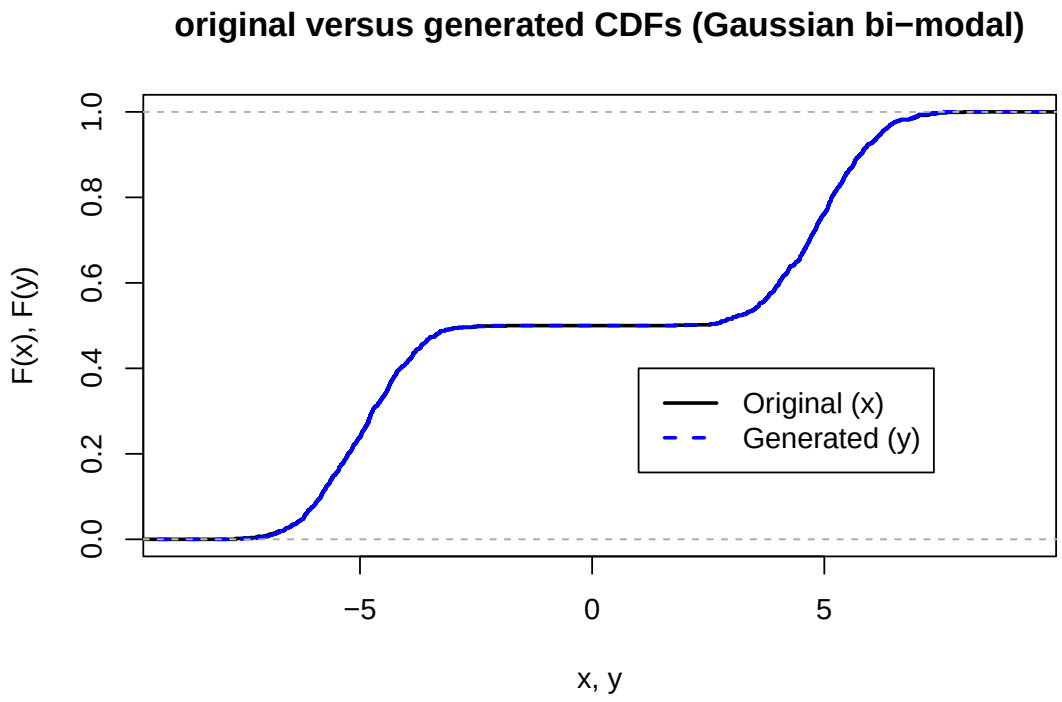
Table 4.7: sampled distribution

generated samples
-2.142562
-2.556046
-2.872753
-3.005417
-3.088909
-3.163469

Table 4.8: distributions characteristics

skew0	kurtosis0
0.0038324	-1.846861
0.0036805	-1.850400

To check our results, let us compare both cdf of x and y in the following figure –>



To check numerically some first properties of the generated distribution, We output in the following table the skewness and kurtosis of both x and y

Table 4.9: KS Test

statistic	pvalue
0.005	1

Table 4.10: one-dimensional bi-modal distribution

samples bimodal distribution
-4.309976
-3.643491
-4.368566
-4.844745
-6.134673
-4.140688

Table 4.11: sampled distribution

generated samples
-0.3026662
-0.5739486
-0.7570987
-1.0403230
-1.3104252
-1.6400824

4.3.4.2 Bimodal t-distribution

In this section we study a bi-modal t - distribution.

We output some values of x in the following lines

Let us call the sampling function, filling up $y \in \mathbb{R}^{M\times 1}$.

We output some values of y in the following lines

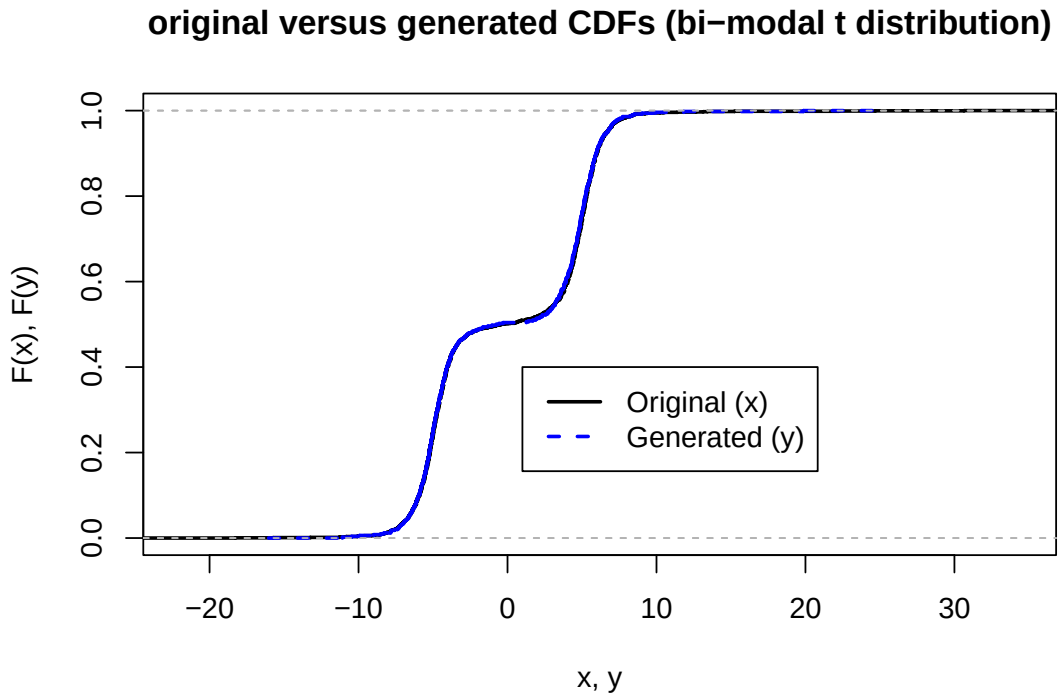
To check our results, let us compare both cdf of x and y in the following figure –>

Table 4.12: distributions characteristics

skew0	kurtosis0
0.1595258	-0.6986219
0.0952795	-1.3211845

Table 4.13: KS Test

statistic	pvalue
0.015	0.9999863



To check numerically some first properties of the generated distribution, We output in the following table the skewness and kurtosis of both x and y

4.3.5 N dimensional Examples

Let us call the sampling function, filling up $y \in \mathbb{R}^{M\times 1}$.

To check our results, let us compare both cdf of x and y in the following figure

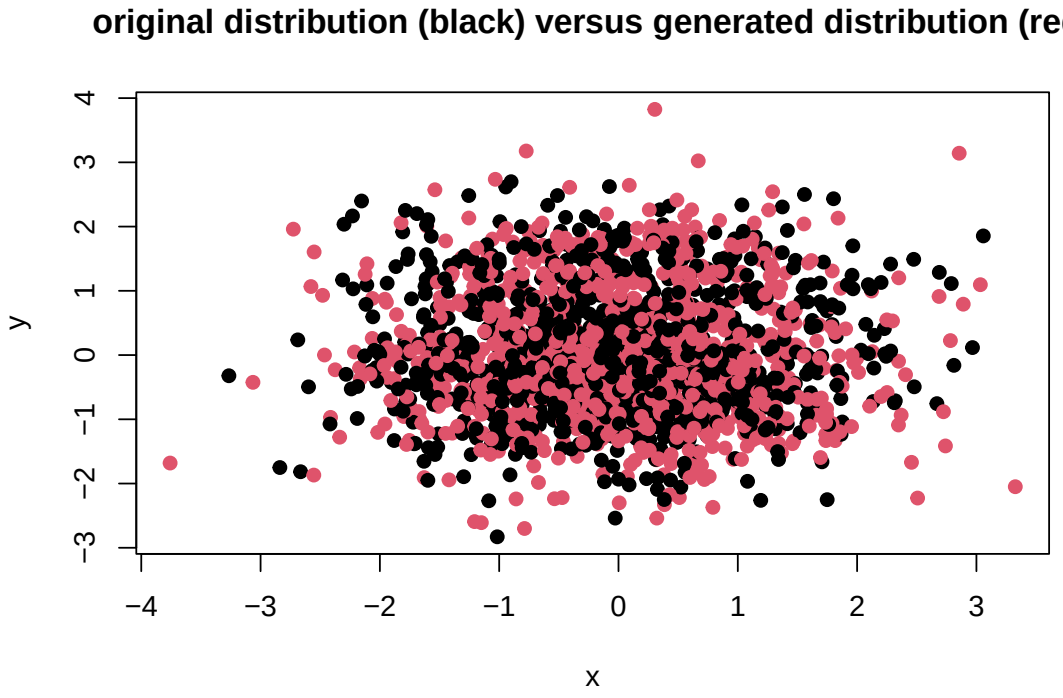
–>

Table 4.14: distributions characteristics

skew0	kurtosis0	skew1	kurtosis1
-0.0722898	0.0196230	0.1187547	-0.2172157
0.2644525	-0.3137455	0.3225859	-0.0308898

Table 4.15: KS Test

statistic	pvalue
0.065	0.0292253
0.055	0.0971035



To check numerically some first properties of the generated distribution, We output in the following table the skewness and kurtosis of both x and y

4.3.5.1 ND t - distribution

Let us call the sampling function, filling up $y \in \mathbb{R}^{M\times 1}$.

To check our results, let us compare both cdf of x and y in the following figure –>

Table 4.16: (#tab:dicrepancy error)Hell. dist. / discrepancy err.

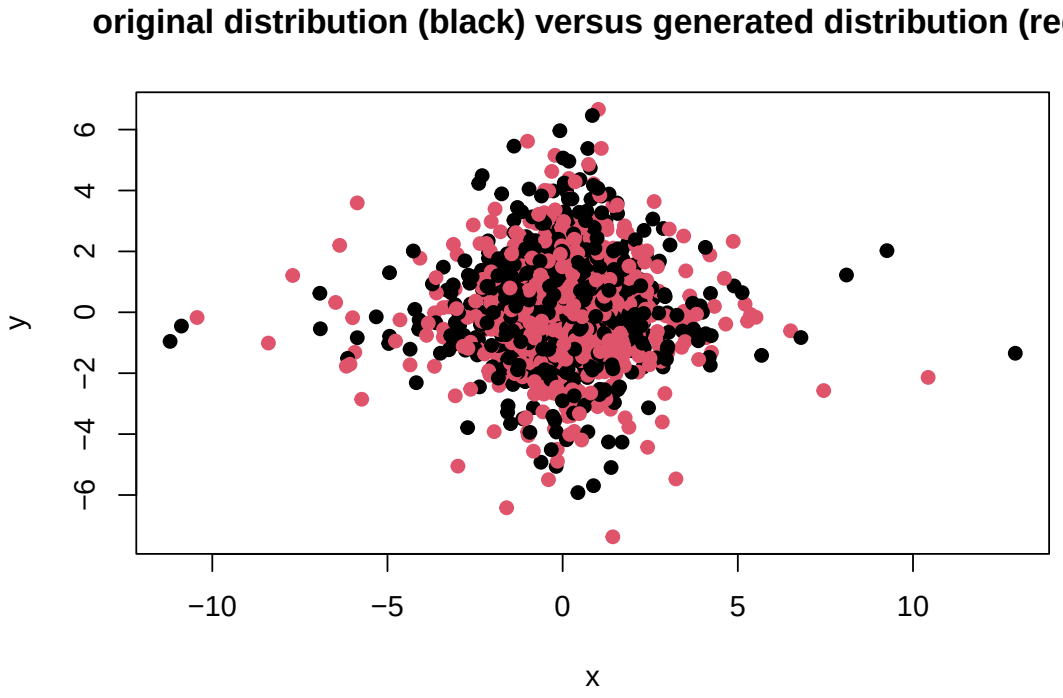
Hellingers	discrepancy
0.6457827	0.1343717

Table 4.17: distributions characteristics

skew0	kurtosis0	skew1	kurtosis1
-0.3370146	8.717153	-0.0291514	8.457726
0.1155473	2.018977	0.0256401	1.543392

Table 4.18: KS Test

statistic	pvalue
0.049	0.1811645
0.045	0.2634717



To check numerically some first properties of the generated distribution, We output in the following table the skewness and kurtosis of both x and y

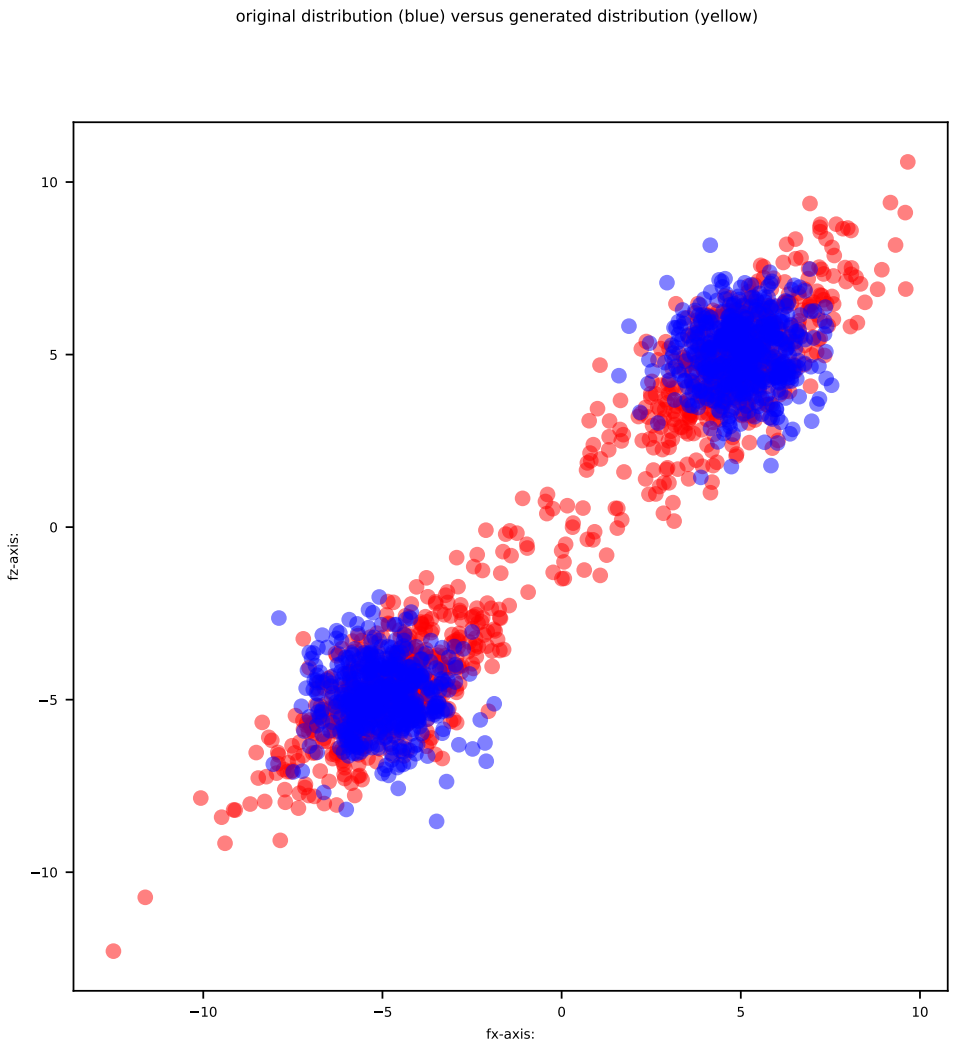
4.3.5.2 Nd-Bimodal Gaussian distribution

In this section we study a bi-modal gaussian distribution.

Let us call the sampling function, filling up $y \in \mathbb{R}^{M\times 1}$, and let us plot both original and generated

Table 4.19: (#tab:dicrepancy error)Hell. dist. / discrepancy err.

Hellingers	discrepancy
0.3571977	0.102778



samples.

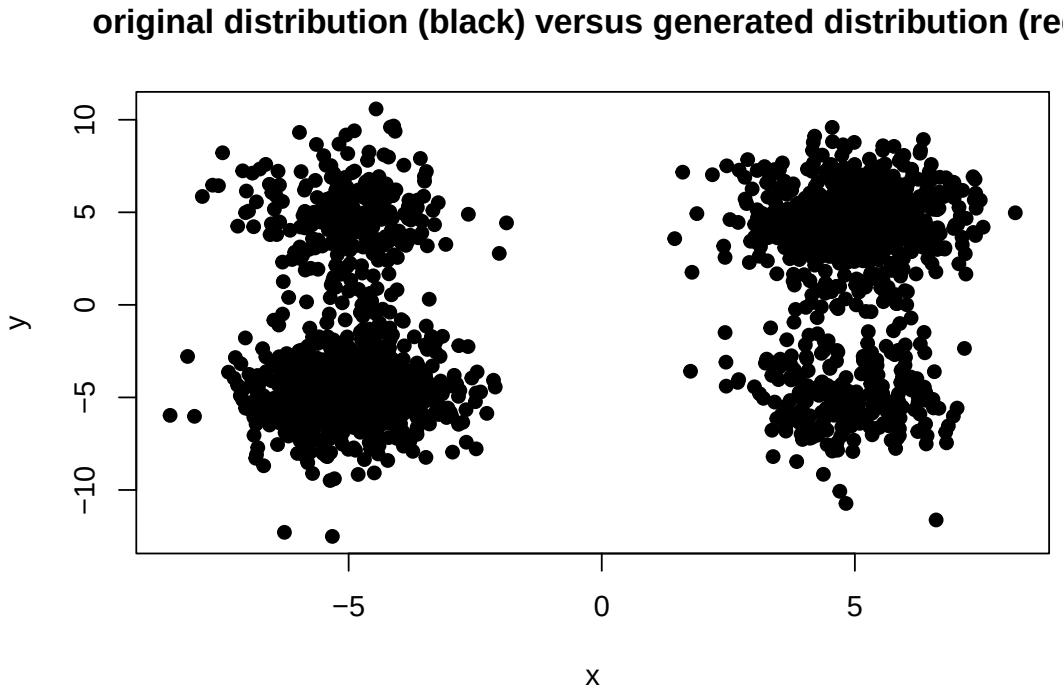
To check our results, let us compare both cdf of x and y in the following figure

Table 4.20: distributions characteristics

skew0	kurtosis0	skew1	kurtosis1
0.0003263	-1.851935	0.0039503	-1.844948
-0.0593544	-1.523340	-0.0211978	-1.543494

Table 4.21: KS Test

statistic	pvalue
0.109	1.36e-05
0.096	1.97e-04



To check numerically some first properties of the generated distribution, We output in the following table the skewness and kurtosis of both x and y

Table 4.22: (#tab:dicrepancy error)Hell. dist. / discrepancy err.

Hellingers	discrepancy
1.151721	0.1693971

Chapter 5

Application to supervised machine learning

In this chapter and the following ones, we present some examples of more concrete learning machines problems. Some of these tests are taken from kaggle, see this url.

Supervised learning problems can be split into Regression and Classification problems. Both problems have as goal the construction of a model that can predict the value of the output from the input variables. In the case of regression the output is a real valued variable, whereas in the case of classification the output is category (e.g. "disease" or "no disease"). Codpy's extrapolate and projection function can be used to treat each of above mentioned problems.

We present two cases corresponding two each typical problems in supervised learning: Boston housing prices prediction and MNIST classification.

5.1 Regression problem: housing price prediction

This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. There are 506 cases and 13 attributes (features) with a target column (price). More details can be found in the article published by Harrison, D. and Rubinfeld, D.L. "Hedonic prices and the demand for clean air", J. Environ. Economics & Management, vol.5, 81-102, 1978.

5.1.1 Codpy's extrapolation

Starting from the training set $x \in \mathbb{R}^{N_x \times D}$, we extrapolate the labels f_z, and compare to test set labels $f(z)$, using the extrapolation operator defined in (3.2.8)-left.

We output at table 5.1 the list of performance indicators for this test.

Table 5.1: Codpy: indicators for Boston housing prices

	0	1	2	3	4	5
predictor_id	housing codpy	housing codpy	housing codpy	housing codpy	housing codpy	housing co
D	13	13	13	13	13	13
Nx	505	456	408	359	311	262
Ny	505	456	408	359	311	262
Nz	506	506	506	506	506	506
Df	1	1	1	1	1	1
execution_time	1.31	1.09	0.88	0.69	0.53	0.41
scores	0.0017	0.0311	0.0358	0.0412	0.0502	0.0557
discrepancy_errors	0	1.2415	0.947	2.287	3.0667	6.3171

5.1.2 Tensorflow

The benchmark method is described chapter **??**. The following lines defines a standard neural network for a regression model.

```
tf_param = {'tfRegressor': {'epochs': 50,
'batch_size':16,
'validation_split':0.1,
'loss':tf.keras.losses.mean_squared_error,
'optimizer':tf.keras.optimizers.Adam(0.001),
'layers':[8,64,64,1],
'activation':['relu','relu','relu','linear'],
'metrics':['mse']}
}
scenarios.run_scenarios(scenarios_list,data_generator_, tfRegressor(set_kernel = set_kernel), data_a
results = scenarios.accumulator.get_output_datas().dropna(axis=1).T
df_results = results
```

We output at table 5.2 the list of performance indicators for this test.

Table 5.2: Tensorflow Neural Network: indicators for Boston housing prices

	0	1	2	3	4	5	6	7
predictor_id	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Te
D	13	13	13	13	13	13	13	13
Nx	505	456	408	359	311	262	214	16
Ny	505	456	408	359	311	262	214	16
Nz	506	506	506	506	506	506	506	50
Df	1	1	1	1	1	1	1	1
execution_time	2.39	2.14	2.01	1.98	1.86	1.71	1.5	1.
scores	0.1328	0.1349	0.1377	0.1299	0.1341	0.1584	0.1726	0.
discrepancy_errors	0	1.2415	0.947	2.287	3.0667	6.3171	4.9851	5.

5.1.3 Pytorch

The Pytorch neural network model is described chapter **??**. We use this parameters set to define this Pytorch regression model, defined below

```
torch_param = {'PytorchRegressor': {'epochs': 50,
'layers': [8,64,64],
'loss': nn.MSELoss(),
```

```
'batch_size': 16,
'loss': nn.MSELoss(),
'activation': nn.ReLU(),
'optimizer': torch.optim.Adam,
'out_layer': 1}}
scenarios.run_scenarios(scenarios_list,data_generator_, PytorchRegressor(set_kernel = set_kernel), d
results = scenarios.accumulator.get_output_datas().dropna(axis=1).T
df_results = pd.concat([df_results,results.T],axis=0)
```

We output at table 5.3 the list of performance indicators for this test.

Table 5.3: Pytorch Neural Network: indicators for Boston housing prices

	0	1	2	3	4	5	6	7	8
predictor_id	Pytorch	Pytorch	Pytorch	Pytorch	Pytorch	Pytorch	Pytorch	Pytorch	Pytorch
D	13	13	13	13	13	13	13	13	13
Nx	505	456	408	359	311	262	214	165	117
Ny	505	456	408	359	311	262	214	165	117
Nz	506	506	506	506	506	506	506	506	506
Df	1	1	1	1	1	1	1	1	1
execution_time	1.1	0.98	0.89	0.78	0.67	0.58	0.47	0.37	0.26
scores	0.1488	0.1461	0.1697	0.2125	0.1997	0.1981	0.1706	0.2049	0.2161
discrepancy_errors	0	1.2415	0.947	2.287	3.0667	6.3171	4.9851	5.052	14.5699

5.1.4 Decision tree

The decision tree model is described chapter **??**.

We output at table 5.4 the list of performance indicators for this test.

Table 5.4: Decision tree: indicators for Boston housing prices

	0	1	2	3	4	5	6
predictor_id	Decision tree	Decision tree	Decision tree	Decision tree	Decision tree	Decision tree	Decis
D	13	13	13	13	13	13	13
Nx	505	456	408	359	311	262	214
Ny	505	456	408	359	311	262	214
Nz	506	506	506	506	506	506	506
Df	1	1	1	1	1	1	1
execution_time	0.01	0.01	0	0	0	0	0
scores	0.0279	0.0476	0.0538	0.0671	0.0738	0.1005	0.131

5.1.5 Methods comparison

The following picture compares methods in term of scores Figure 5.5, discrepancy errors Figure 5.6, and execution time Figure 5.7. We give an interpretation of these results.

- First notice that the kernel method *codpy lab extra*, that is the extrapolation method, obtains both best scores and worst execution time.
- Notice also that one, minus the discrepancy error, matches the scores of the method *codpy lab extra*. This indicates that the discrepancy error is a pertinent indicator.

- Another kernel method, *codpy lab proj*, that is the projection method above, is a more balanced method [1].
- Both kernel methods are shipped with a very standard kernel, that is the Gaussian one, that is the only parameter for kernel methods. We emphasize that Kernel engineering can easily improves these results. We do not present these improved kernel methods, as our purposes is to benchmark standard methods.

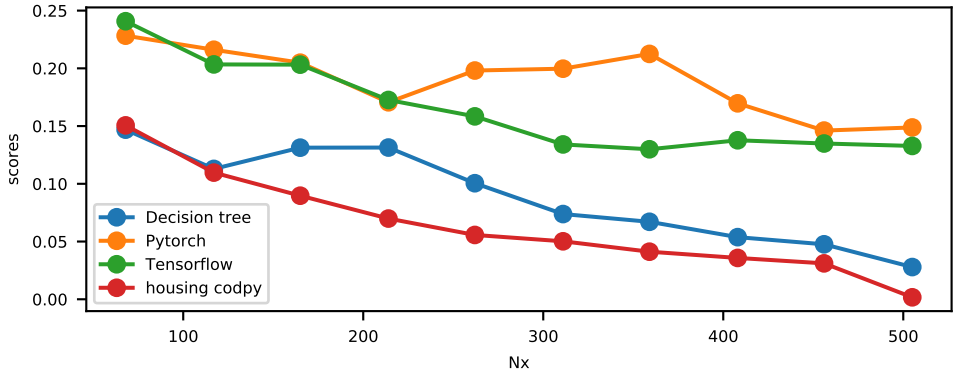


Figure 5.1: Benchmark scores

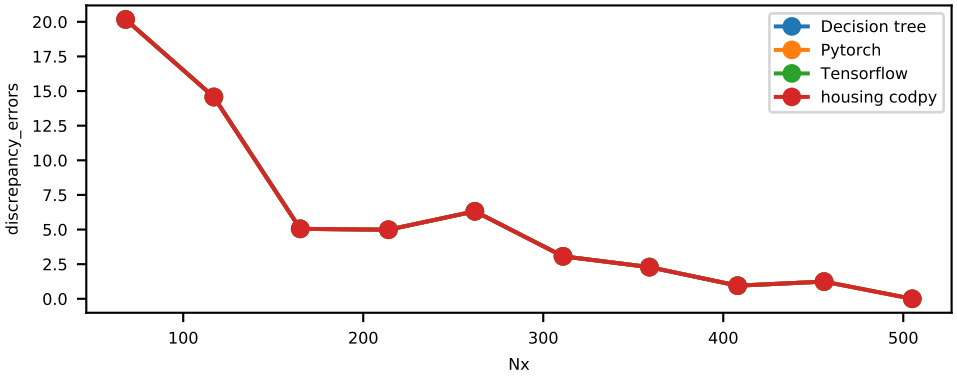


Figure 5.2: Discrepancy errors

5.2 Classification problem: handwritten digits

This section contains an example of classification for images, which is a typical academic example referred to as the MNIST problem, and allows us to benchmark our results against more popular methods.

MNIST ("Modified National Institute of Standards and Technology") contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms.

In this section, we propose a benchmark of several machine learning methods, including kernel ones. Our goals, above benchmarking our methods against popular alternatives, are to demonstrate that

[1] except Gradient boosting method, for which we did not succeed retrieving a competitive set of parameters for this test.

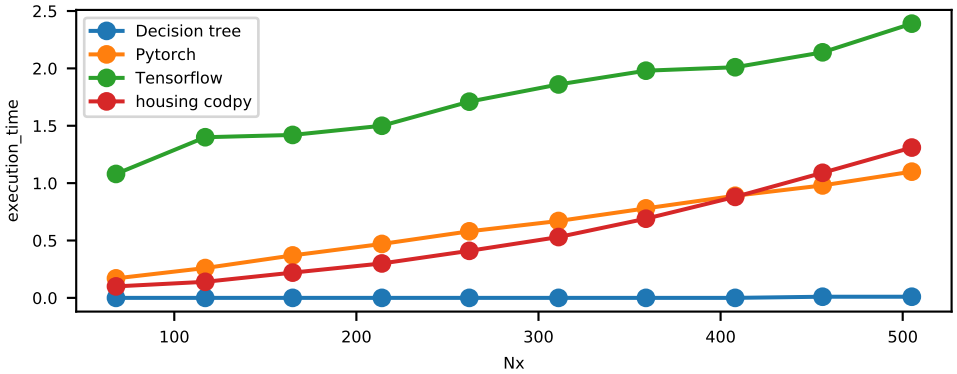


Figure 5.3: Benchmarks execution time

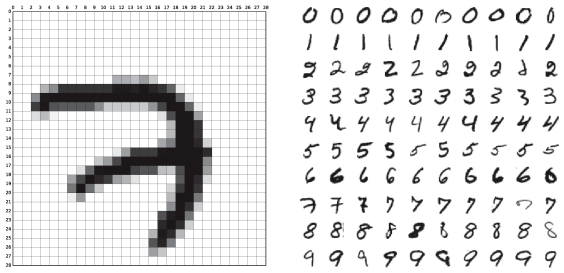
all these tests are problem dependent, not method dependents. To illustrate this fact, we purposely almost copy paste each test, to test another method. The motivation here is also to provide to our users a bank of code, where they can just copy paste one section of this document to test their own learning machines.

5.2.1 Short introduction to MNIST

The MNIST dataset is composed of $60,000$ images defining a training set of handwritten digits. Each image is a vector having dimensions 784 (a 28×28 grayscale image flattened in row-order). There are 10 digits 0–9. The test set is composed of $10,000$ images with their labels.

We formalize the problem as follows. Given the test set represented by a matrix $x \in \mathbb{R}^{N_x \times D}$, $D = 784$, the labels $f(x) \in \mathbb{R}^{N_x \times D_f}$, $D_f = 10$, and the test set $z \in \mathbb{R}^{N_z \times D}$, $N_z = 10000$, predict the label function $f(z) \in \mathbb{R}^{N_z \times D_f}$. Data are retrieved from Y. LeCun MNIST home page [**?**], and we will test different values for N_x.

The following picture shows an image of hand-written number, that is the first image x^1, as well as numerous others



(a) MNIST sample belonging to the digit '7'.

(b) 100 samples from the MNIST training set.

The following line defines our scenario list

The table **??** output this scenario list

D	Nx	Ny	Nz
784	32	8	10000
784	64	16	10000
784	128	32	10000
784	256	64	10000

Scores are computed using the formula (2.3.1), a scalar in the interval between 0 and 1, which counts the number of correctly predicted images.

Our kernel setup for this MNIST test is the following

```
set_mnist_kernel = kernel_setters.kernel_helper(kernel_setters.set_gaussian_kernel, 0,1e-8 ,map_sett
```

5.2.1.1 Keras Tensorflow scores

The benchmark method is described chapter **??**. The following lines defines a standard neural network for studying the MNIST problem.

```
import tensorflow as tf
tf_param = {'tfClassifier' : {'epochs': 10,
'batch_size':16,
'validation_split':0.1,
'loss': tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
'optimizer':tf.keras.optimizers.Adam(0.001),
'activation':['relu',''],
'layers':[128,10],
'metrics':[tf.keras.metrics.SparseCategoricalAccuracy()]} }
```

We then run the benchmarks

We output at table 5.5 the list of performance indicators for this test.

Table 5.5: tensorflow: indicators for MNIST

	0	1	2	3
predictor_id	Tensorflow	Tensorflow	Tensorflow	Tensorflow
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.62	0.55	0.66	0.64
scores	0.37	0.4488	0.6122	0.7576
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

Finally, we output as well the confusion matrix for the last scenario in figure 5.4.

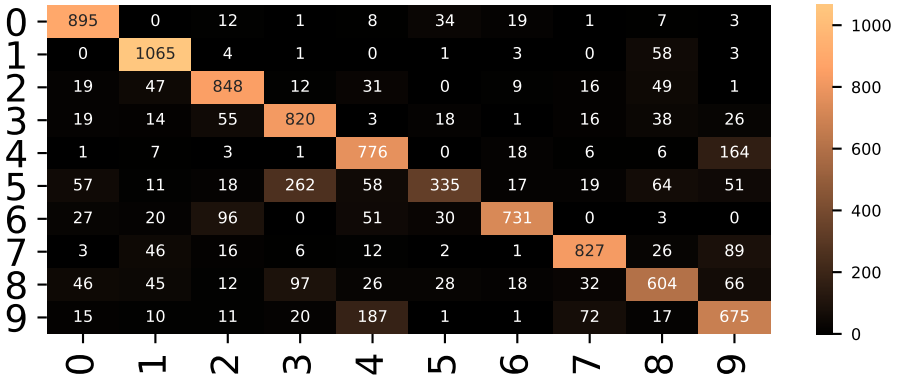


Figure 5.4: confusion matrix for tensorflow

5.2.1.2 CodPy scores extrapolation

Starting from the training set $x \in \mathbb{R}^{N_x \times 784}$, we extrapolate the labels f_z, and compare to test set labels $f(z)$, using the extrapolation operator defined in (3.2.8)-left.

We output at table 5.6 the list of performance indicators for this test.

Table 5.6: codpy extrapolation: indicators for MNIST

	0	1	2	3
predictor_id	codpy lab extra	codpy lab extra	codpy lab extra	codpy lab extra
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.51	0.55	0.63	0.75
scores	0.5671	0.685	0.7361	0.8213
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

Finally, we output as well the confusion matrix for the last scenario in figure 7.1.

5.2.1.3 CodPy scores projection

In this section we apply straightfowardly the projection operator (3.2.6), where the training set is $x \in \mathbb{R}^{N_x \times 784}$, and $y \in \mathbb{R}^{N_y \times 784} \subset x$ is randomly chosen. Then we use the projection operator defined in (3.2.6).

We output at table 5.7 the list of performance indicators for this test.

Table 5.7: codpy extrapolation: indicators for MNIST

	0	1	2	3
predictor_id	codpy lab pred	codpy lab pred	codpy lab pred	codpy lab pred
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.45	0.51	0.58	0.56
scores	0.4129	0.5573	0.6614	0.7467
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

Finally, we output as well the confusion matrix for the last scenario in figure **??**.

5.2.1.4 Pytorch

The Pytorch neural network model is described chapter **??**. We use this parameters set to define this Pytorch machine.

```
torch_param = {'PytorchClassifier': {'epochs': 10,
'layers': [128],
'batch_size': 16,
'loss': nn.CrossEntropyLoss(),
```

```
'activation': nn.ReLU(),
'optimizer': torch.optim.Adam,
"datatype": "long",
"prediction": "labeled",
"out_layer": 10}}
```

We output at table 5.8 the list of performance indicators for this test.

Table 5.8: Pytorch Neural Network: indicators for MNIST

	0	1	2	3
predictor_id	Pytorch	Pytorch	Pytorch	Pytorch
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.45	0.42	0.54	0.54
scores	0.4785	0.6326	0.6979	0.7941
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

5.2.1.5 Decision Tree

The decision tree model is described chapter **??**.

We output at table 5.9 the list of performance indicators for this test.

Table 5.9: Decision tree classifier: indicators for MNIST

	0	1	2	3
predictor_id	Decision tree	Decision tree	Decision tree	Decision tree
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.02	0.02	0.02	0.02
scores	0.303	0.3942	0.4359	0.5058
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

5.2.1.6 AdaBoost

The Adaboost model is described chapter **??**.

We output at table 5.10 the list of performance indicators for this test.

Table 5.10: AdaBoost classifier: indicators for MNIST

	0	1	2	3
predictor_id	AdaBoost	AdaBoost	AdaBoost	AdaBoost
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.77	0.83	0.88	0.93
scores	0.2878	0.4581	0.4819	0.5289
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

5.2.1.7 Gradient Boosting

The gradient boosting model is described chapter **??**.

We output at table 5.11 the list of performance indicators for this test.

Table 5.11: Gradient Boosting classifier: indicators for MNIST

	0	1	2	3	4
predictor_id	Gradient Boosting	Gradient Boosting	Gradient Boosting	Gradient Boosting	Gradient B
D	784	784	784	784	784
Nx	32	64	128	256	512
Ny	8	16	32	64	128
Nz	10000	10000	10000	10000	10000
Df	1	1	1	1	1
execution_time	0.61	0.91	1.69	3.1	6.31
scores	0.2554	0.4089	0.5304	0.656	0.7595
discrepancy_errors	0.3524	0.2627	0.2192	0.1731	0.1497

5.2.1.8 XGBoost

The XGBoost model is described chapter **??**. We set its parameters as follows.

```
xgb_param = {'epochs': 5,
'max_depth': 3,
'eta' : 0.3,
'objective': 'multi:softmax',
'num_class': 10}
```

5.2.1.9 Random Forest

The random forest model and its parameter set are described chapter **??**.

We output at table 5.12 the list of performance indicators for this test.

Table 5.12: Random Forest classifier: indicators for MNIST

	0	1	2	3
predictor_id	RForest	RForest	RForest	RForest
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.7	0.79	0.76	0.83
scores	0.4578	0.6212	0.7118	0.7698
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

5.2.1.10 Support vector classifier

The SVC model and its parameter set are described chapter **??**.

We output at table 5.13 the list of performance indicators for this test.

Table 5.13: SVC classifier: indicators for MNIST

	0	1	2	3
predictor_id	SVC	SVC	SVC	SVC
D	784	784	784	784
Nx	32	64	128	256
Ny	8	16	32	64
Nz	10000	10000	10000	10000
Df	1	1	1	1
execution_time	0.18	0.33	0.63	1.17
scores	0.5446	0.6634	0.7288	0.8105
discrepancy_errors	0.3524	0.2627	0.2192	0.1731

Table 5.14: SVC classifier: indicators for MNIST

0	1	2	3	4	5	6	7	8
Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow	Tensorflow
13	13	13	13	13	13	13	13	13
505	456	408	359	311	262	214	165	117
505	456	408	359	311	262	214	165	117
506	506	506	506	506	506	506	506	506
1	1	1	1	1	1	1	1	1
2.39	2.14	2.01	1.98	1.86	1.71	1.5	1.42	1.4
0.1328	0.1349	0.1377	0.1299	0.1341	0.1584	0.1726	0.2032	0.2034
0	1.2415	0.947	2.287	3.0667	6.3171	4.9851	5.052	14.5699
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

5.2.2 Comparing methods

The following picture compares methods in term of scores Figure 5.5, discrepancy errors Figure 5.6, and execution time Figure 5.7. We give an interpretation of these results.

- First notice that the kernel method *codpy lab extra*, that is the extrapolation method, obtains both best scores and worst execution time.
- Notice also that one, minus the dicrepancy error, matches the scores of the method *codpy lab extra*. This indicates that the discrepancy error is a pertinent indicator.
- Another kernel method, *codpy lab proj*, that is the projection method above, is a more balanced method [2].
- Both kernel methods are shipped with a very standard kernel, that is the gaussian one, that is the only parameter for kernel methods. We emphasize that Kernel engineering can easily improves these results. We do not present these improved kernel methods, as our purposes is to benchmark standard methods.

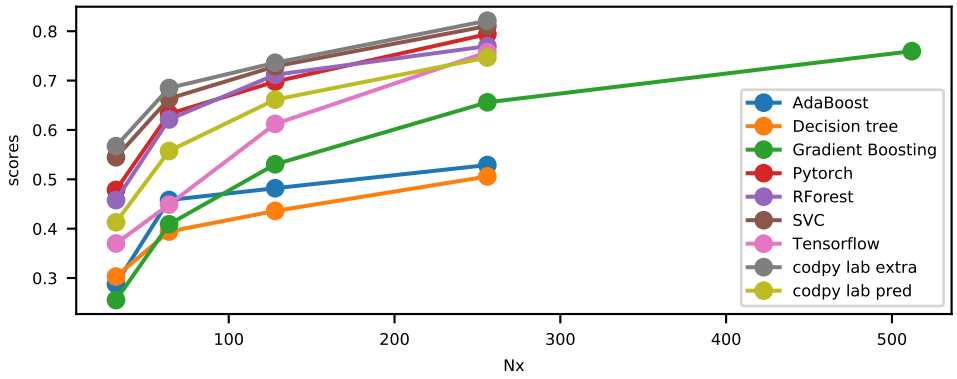


Figure 5.5: Benchmark scores

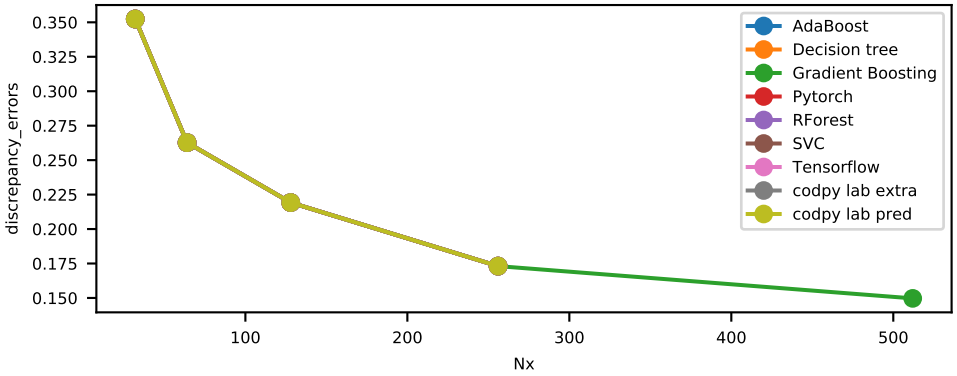


Figure 5.6: Discrepancy errors

5.3 Reconstruction problems : learning from sub-sampled signals in tomography.

This numerical experience illustrates an interesting capability of learning machines to reconstruction problems from sub-sampled signals. Indeed, in this test, we will be learning from a well-

[2] except Gradient boosting method, for which we did not succeed retrieving a competitive set of parameters for this test.

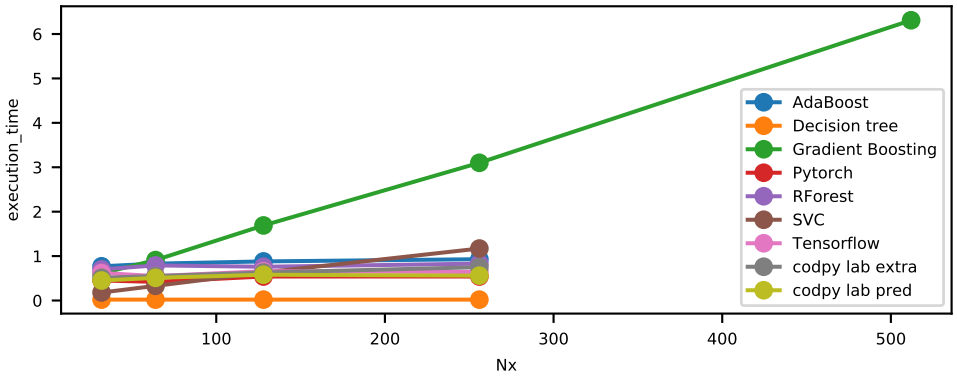


Figure 5.7: Benchmarks execution time

established algorithm, that is the SART one, to fasten the reconstruction.

There are many applications of such problems. We illustrate this section with a problem coming from a medical image reconstruction, that can be used also as a medical helping diagnosis decision tool. However, such problems occur in a wide variety of other situations: biology, oceanography, astrophysics, ...

Poor input signal quality can sometimes be a choice. For instance, in nuclear medicine, it is better to work with lower radioisotopes concentration for obvious health reasons. Another interesting motivation for sub-sampling signals can be also accelerating data acquisition processes from expensive machines.

We illustrate this section with an example of such a reconstruction coming from reconstructing a signal from a sub-sampled SPEC (tomography) problem that we describe now.

5.3.1 A problem coming from SPECT tomography

The purpose of this test is to illustrate a sub-sampling reconstruction in the context of medical imagery, more precisely from sub-sampled SPECT images. To that aim, we start from collecting a set of *high resolution* images[3]. The set itself is not really important for our illustration sake in this section. However it should be chosen carefully for real, production problem.

This database image consists in high resolution (512x512) images, consisting in approximately 30 images of 82 patients. The training set is built on the first 81 patient. The 82-th patient is used for the test set. We first transform the training set database to produce our data. For each image in the training set (2470 images):

- We perform a "high" resolution (256x256) radon transform [4], called a **sinogram** [5]. A sinogram is quite close to a Fourier transform of the original image, generating sinusoids.
- We perform a "low" resolution (8x256) radon transform.
- We reconstruct the original image from the high resolution sinogram to simulate high resolution SPECT images from these data. The reconstruction algorithm consists in computing an inverse radon transform [6].

An example of training set construction is presented Figure 5.8. Left is the reconstructed image from the "high resolution" sinogram (middle). The low resolution sinogram is plot at right.

[3] the image set is available publicly at this kaggle link.
[4] An introduction to radon transform can be found at this wikipedia page.
[5] We used the standard radon transform from scikit, available at this url.
[6] We used a SART algorithm, 3 iterations, for reconstruction, available at this url.

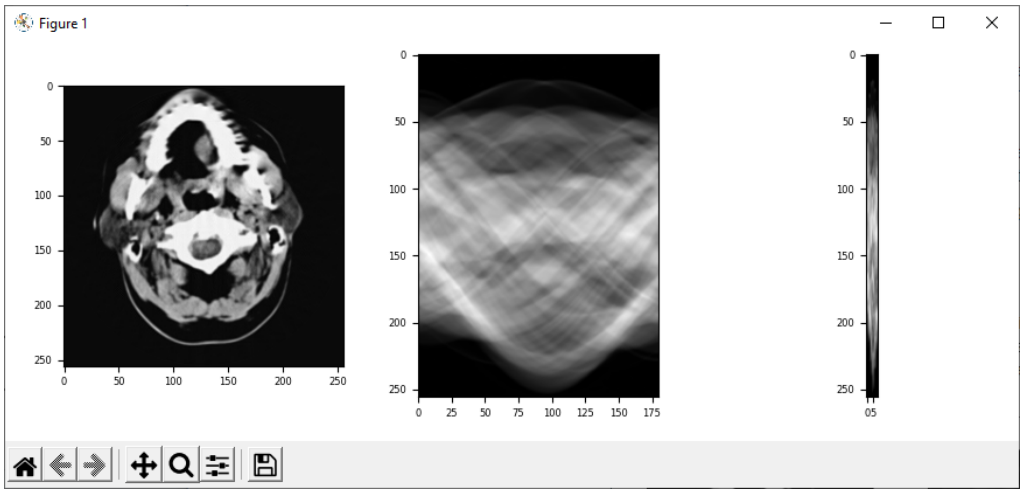


Figure 5.8: high resolution sinogram (middle), low resolution (right), reconstructed image (left)

The test consists then in reconstructing all images of the 82-th patient using low-resolution sinograms.

5.3.2 Performing the test

We present here the test resulting from a benchmark of a kernel-based method and the SART algorithm[7]

Following our notations, section **??**, we introduce

- The training set $x \in \mathbb{R}^{2473\times 2304}$, consisting in 2473 sinograms having resolution 8×256, consisting in all low-resolution sinograms of the 81 first patients, plus the first one of the 82-th patient. This last figure is added to check an important feature in these problems : the learning machine must be able to retrieve an already input example.
- The test set $z \in \mathbb{R}^{29\times 2304}$, consisting in 29 sinograms of the 82-th patient, having resolution 8×256.
- The training values set $f_x \in \mathbb{R}^{2473\times 65536}$, consisting in the 2473 images in "high-resolution".
- The ground truth values $f(z) \in \mathbb{R}^{29\times 65536}$, consists in 29 images in "high-resolution".

We perform the tests and output the results in Table **??**. The columns are the predictor identifiant, D, N_x, N_y, N_z, D_f, the execution time, and the score, computed with the RMSE % error indicator, see (2.3.1).

- The first line, named *exact*, simply output the original figures, leading to zero error.
- The second one, named *SART*, reconstruct the figures from the SART algorithm with sub-sampled data.
- The third one, named *codpy*, reconstruct the figures from the sub-sampled data with the kernel extrapolation method (3.2.8).

The figure 5.9 plots the first 8 images, presenting the original one at left, the reconstruction from SART algorithm, middle, and our algorithm, right. One can check visually that this kernel method better reconstruct the original image. It would be erroneous to conclude that this reconstruction process performs better than the SART algorithm, and it is not at all our speech here. We simply illustrate here the capacity of our algorithm to recognize existing patterns: indeed, note that the first image is perfectly reconstructed, as it is part of the training set. This property

[7]We did not succeed finding competitive parameters for other methods.

emphasizes that such methods suit well to pattern recognition problems, as automated tools to support professionals diagnosis.

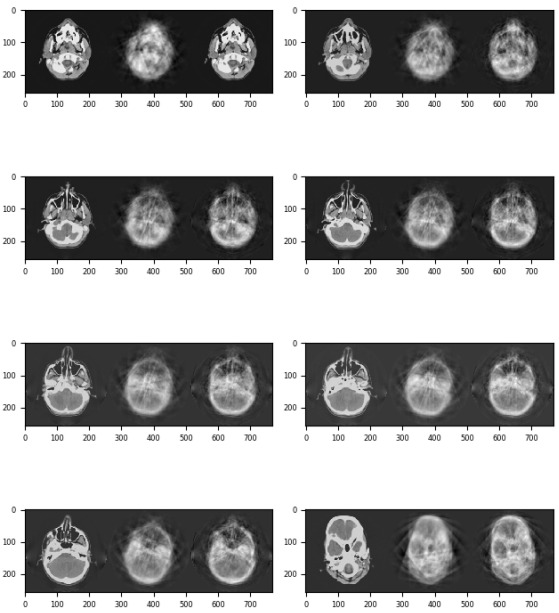


Figure 5.9: Example of reconstruction original (left), sub-sampled SART (middle), kernel extrapolation (right)

Chapter 6

Application for unsupervised machine learning

In this section we apply some clustering methods for a number of use cases.

We benchmarked our kernel-based algorithms, see section 3.6 against the popular k-means algorithms. Both are distance-based minimization algorithms, aiming to solve the problem @ref{eq:dist}, that we recall here

$$y = \arg \inf_{y \in \mathbb{R}^{N_y \times D}} d(x, y)$$

The clusters $y \in \mathbb{R}^{N_y \times D}$ are the results of this minimization algorithm, where :

- For k-means based algorithms, the distance is called the *inertia*, see section **??**.
- For kernel-based algorithms, the distance is called the *discrepancy error*, see section **??**.

Importantly, if the distance functional $d(x, y)$ is not convex, then a solution to (3.6.1) might not be unique. For instance, a k-mean based algorithm usually output different clusters output at different runs.

6.1 Classification problem: handwritten digits

The MNIST test is also studied in the section **??**. Here we consider it as a semi-supervised learning: we use the train set $x \in \mathbb{R}^{N_x \times D}$ to compute the cluster's centroids $y \in \mathbb{R}^{N_y \times D}$. Then we use these clusters to predict the test labels $f_z \in \mathbb{R}^{N_z \times D_f}$, corresponding to the test set $z \in \mathbb{R}^{N_z \times D}$.

The following lines define our setting for this test.

6.1.1 Scikit k-means

First we use Scikit's k-means algorithm implementation, which is simply partitioning the input data $x \in \mathbb{R}^{N_x \times D}$ into Ny sets so as to minimize the within-cluster sum of squares, which is defined as "inertia", see **??**. The inertia represents the sum of distances of all points to the centroid $y \in \mathbb{R}^{N_y \times D}$ in a cluster. K-means algorithm starts with a group of randomly initialized centroids and then performs iterative calculations to optimize the position of centroids until the centroids stabilizes, or the defined number of iterations is reached.

The result of k-means algorithm is N_y clusters in $D = 784$ dimensions, i.e. $y \in \mathbb{R}^{N_y \times D}$. Note that the cluster centroids themselves are 784-dimensional points, and can themselves be interpreted as

the "typical" digit within the cluster. The figure 6.1 plots some examples of computed clusters, interpreted as images. As can be seen, they are perfectly recognizable.

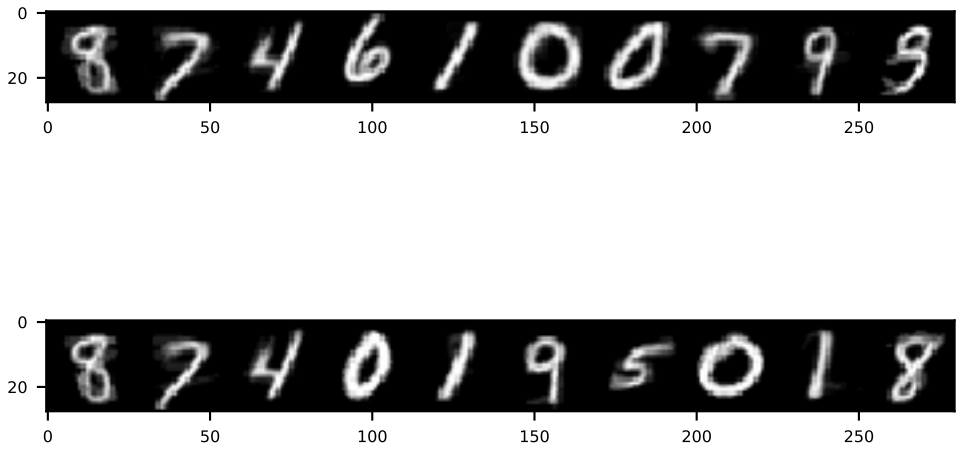


Figure 6.1: k-means scikit clusters interpreted as images

The table 6.1 displays the dimension of the problem D, the size of training set Nx, the number of clusters Ny, the size of the test set Nz, the execution time, inertia and discrepancy errors, scores. The higher the scores and the lower are the inertia and discrepancy errors the better.

Table 6.1: Scikit: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	scores	discrepancy_errors	inertia
k-means	784	1000	128	1000	1	4.27	0.784	0.2232	20135.37
k-means	784	1000	256	1000	1	4.69	0.793	0.1600	14233.02

6.1.2 Scikit minibatch k-mean

We replicate our previous tests for a different scikit algorithm, that is mini batch. Minibatch is a k-mean algorithm that is optimized for computational time.

The figure 6.2 plots some examples of computed clusters, interpreted as images.

The table 6.2 displays the performance indicator for this test.

Table 6.2: Scikit: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	scores	discrepancy_errors	inertia
minibatch	784	1000	128	1000	1	2.64	0.765	0.1894	21824.12
minibatch	784	1000	256	1000	1	2.33	0.747	0.1505	17149.84

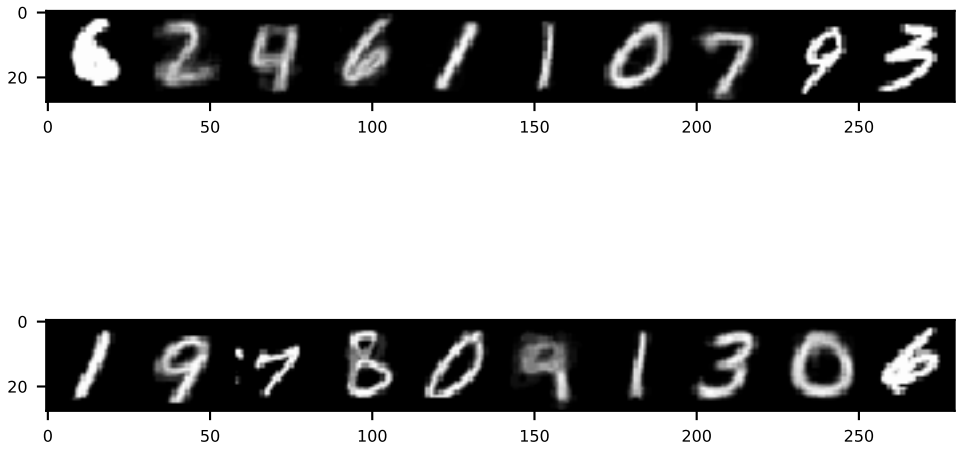


Figure 6.2: k-means scikit clusters interpreted as images

6.1.3 Codpy

In this section we apply codpy's algorithm described in 3.6 using the distance $d_k(x, y)$ induced by a Gaussian kernel:

$$k(x, y) = \exp(-(x - y)^2)$$

We repeat the same test as in the previous section. We first run all scenarios.

Then we dispay figure 6.3 some computed clusters as images, and as for k-means, notice that these are recognizable numbers.

The table 6.3 displays computed performance indicators for all scenarios.

Table 6.3: codpy: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	scores	discrepancy_errors	inertia
codpy	784	1000	128	1000	1	3.57	0.836	0.1325	20137.13
codpy	784	1000	256	1000	1	4.35	0.856	0.1292	14238.05

6.1.4 Benchmarks results

Finally, we illustrate a benchmark plot, displaying the computed performance indicator of Scikit's k-means and codpy's sharp discrepancy algorithms in terms of discrepancy errors, inertia, accuracy scores (when applicable) and execution time.

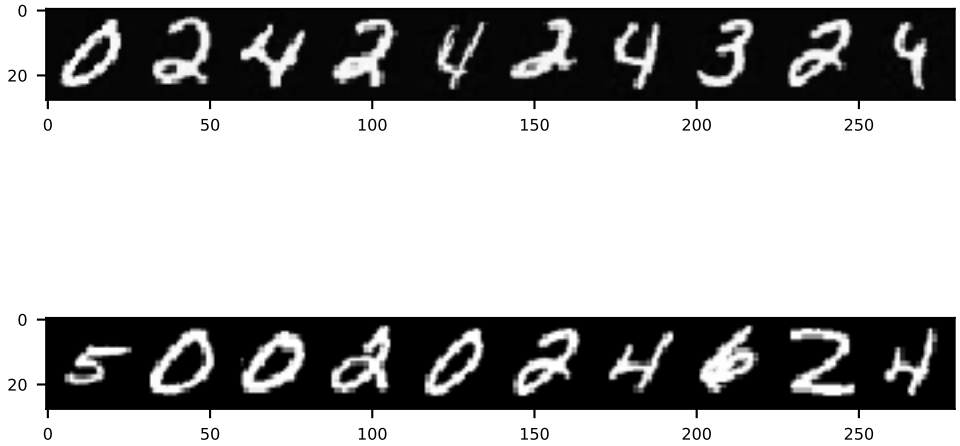
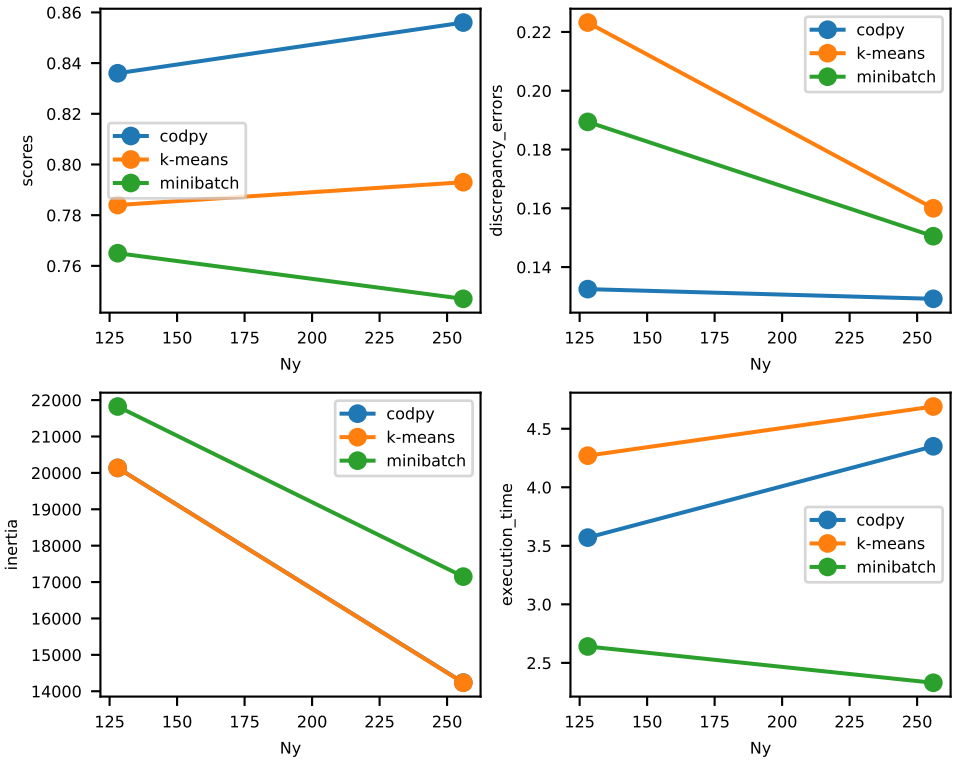


Figure 6.3: codpy clusters interpreted as images



Notice that the score are quite high, when compared to supervised methods for similar size of

Table 6.4: performance indicator for scikit

predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_error	sinertia
k-means.	24	272	10	272	0	2.58	0.2580	3773.76
k-means.	24	272	20	272	0	1.73	0.1726	2502.04

training set, see results section **??**. Notice also that codpy, which algorithm relies on a discrepancy distance minimization, displays an inertia indicator that is lower than minibatch, and quite comparable to k-means. This is surprizing as k-means algorithms are based on inertia minimization. Moreover, scores seems to indicate than the disrepancy distance is a more reliable criteria than inertia on this pattern recognition problem.

6.2 German credit risk

The original dataset contains 1000 entries with 20 categorial/symbolic attributes. In this dataset, each entry represents a person who takes a credit by a bank. The goal is to categorize each person as good or bad credit risks according to the set of attributes. The dataset is described on kaggle page link.

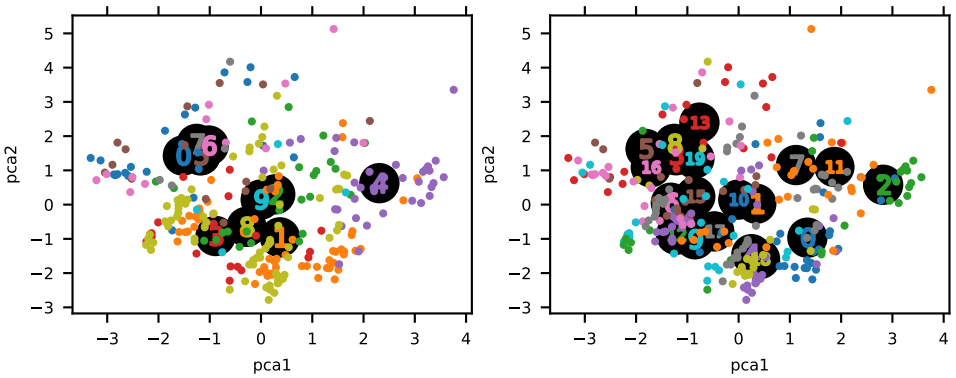
The following lines define our setting for this test.

6.2.1 Scikit

The tests follows the very same method as in the previous section. We first run our scenarios in the following line.

The result of k-means algorithm is Ny clusters in D dimensions. Notice that the cluster centroids themselves are D-dimensional points.

Next we visualize the clusters and corresponding centroids of scikit, where we vary the number of clusters Ny from 1 to 8. Obviously in this example we see that the high number of clusters leads to overfitting and one is unable to interpret the resulting clusters when $Ny = 8$.



The table 6.4 displays inertia, discrepancy errors and execution time performance indicators.

6.2.2 Codpy

In this section we apply the same methodology as in the previous sections.

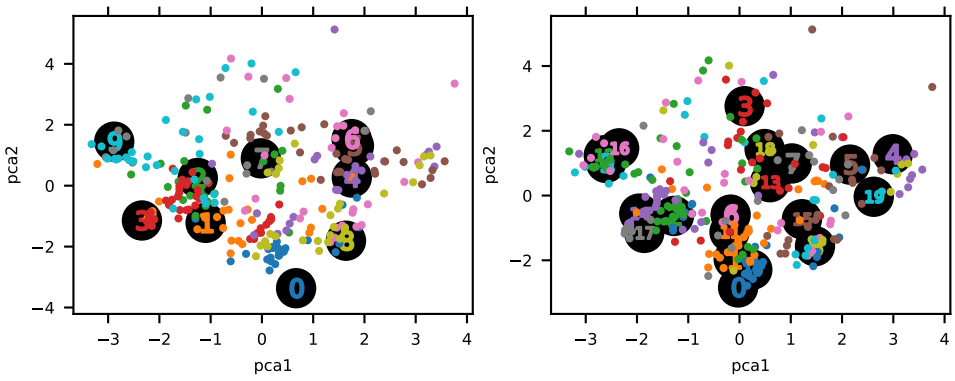
The result of codpy's sharp discrepancy algorithm is Ny clusters in D dimensions. Notice that the cluster centroids themselves are D-dimensional points.

Next we visualize the clusters and corresponding centroids computed using codpy's sharp discrepancy algorithm, where we vary the number of clusters Ny from 1 to 8. Obviously in this example we

Table 6.5: performance indicator for codpy

predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_errors	inertia
codpy	24	272	10	272	0	0.04	0.1557	3742.88
codpy	24	272	20	272	0	0.05	0.0823	2474.09

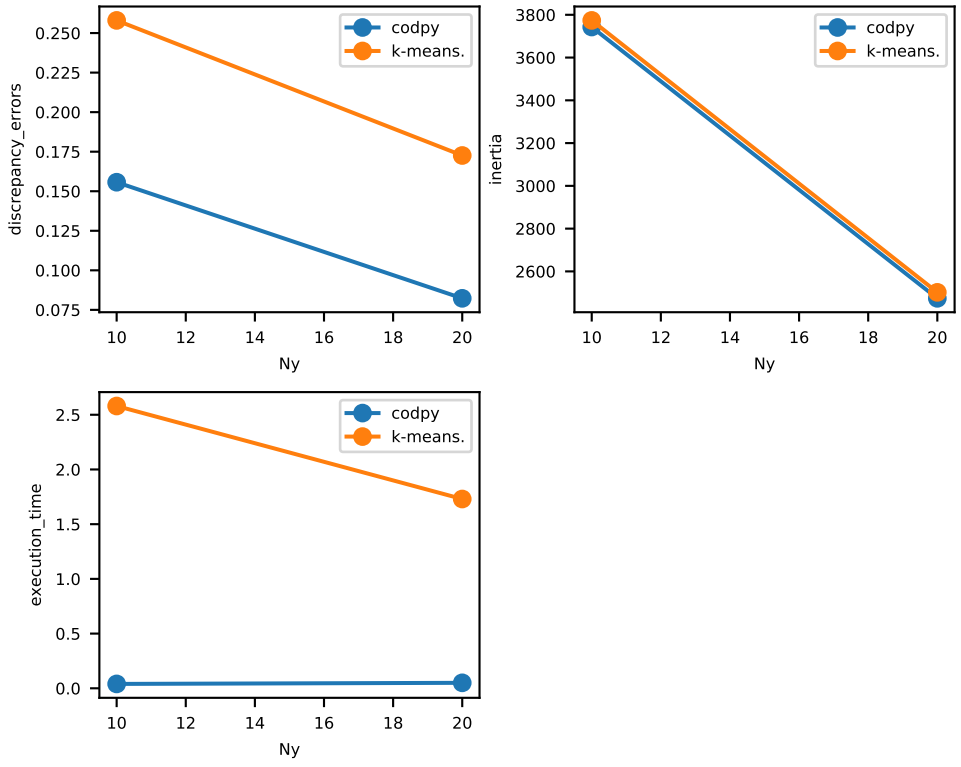
see that the high number of clusters leads to an overfitting and one is unable to interpret the resulting clusters when $Ny = 8$.



The table 6.5 displays inertia, discrepancy errors and execution time performance indicators.

6.2.3 Benchmarks results

Finally, we illustrate a benchmark plot, displaying the computed performance indicator of Scikit's k-means and codpy's sharp discrepancy algorithms in terms of discrepancy errors, inertia, accuracy scores (when applicable) and execution time.



6.3 Credit card marketing strategy

The problem can be formalized as follows. Develop a customer segmentation to define marketing strategy. The sample dataset summarizes the usage behavior of 8,950 active credit card holders during the last 6 months. The dataset contains 17 features and 8,950 records. The data describes customer's purchase and payment habits, such as how often a customer installment purchases, or how often they make cash advances, how much payments are made, etc. By inspecting each customer, we can find which type of purchase he/she is keen on, or if he/she prefers cash advance over purchases. The dataset is detailed on this dedicated kaggle page.

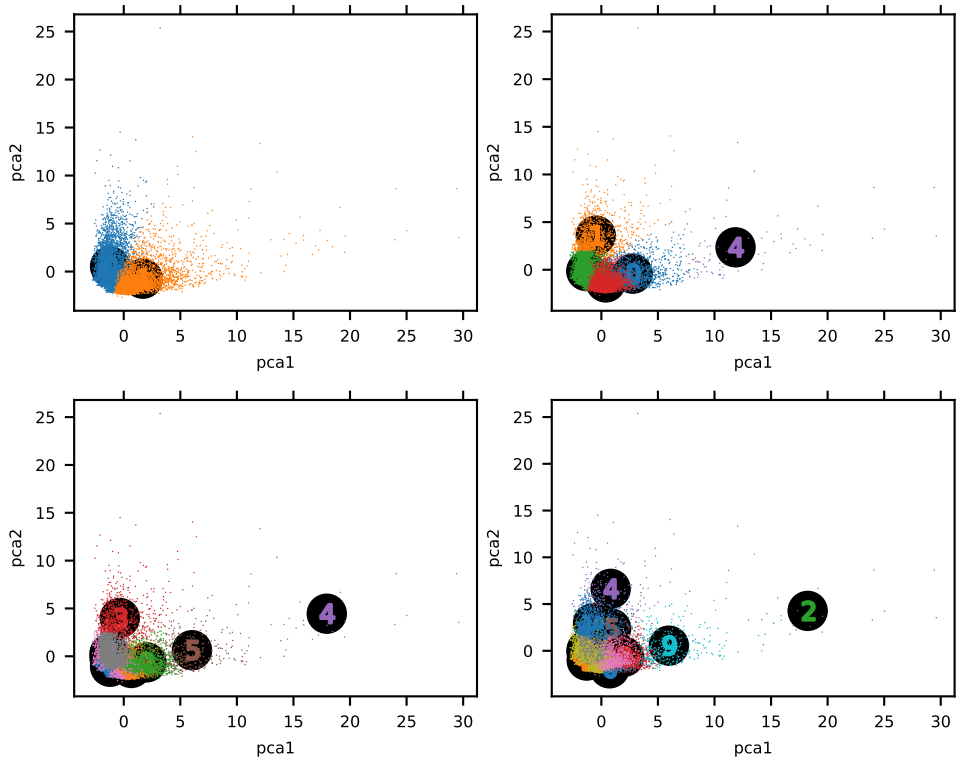
The following lines define our setting for this test.

6.3.1 Scikit

First we use Scikit's k-means algorithm implementation, which is simply partitioning the input data $x \in \mathbb{R}^{N_x \times D}$ into Ny sets so as to minimize the within-cluster sum of squares, which is defined as "inertia". The inertia represents the sum of distances of all points to the centroid $y \in \mathbb{R}^{N_y \times D}$ in a cluster. K-means algorithm starts with a group of randomly initialized centroids and then performs iterative calculations to optimize the position of centroids until the centroids stabilize, or the defined number of iterations is reached.

The result of k-means algorithm is Ny clusters in D dimensions. Notice that the cluster centroids $y \in \mathbb{R}^{N_y \times D}$ themselves are D-dimensional points.

Next we visualize the clusters and corresponding centroids of scikit's k-means implementation, where we vary the number of clusters Ny from 2 to 4.



The table below demonstrates the performance of Scikit's k-means algorithm in terms of inertia, discrepancy errors and time.

Table 6.6: Scikit:2,4 clusters

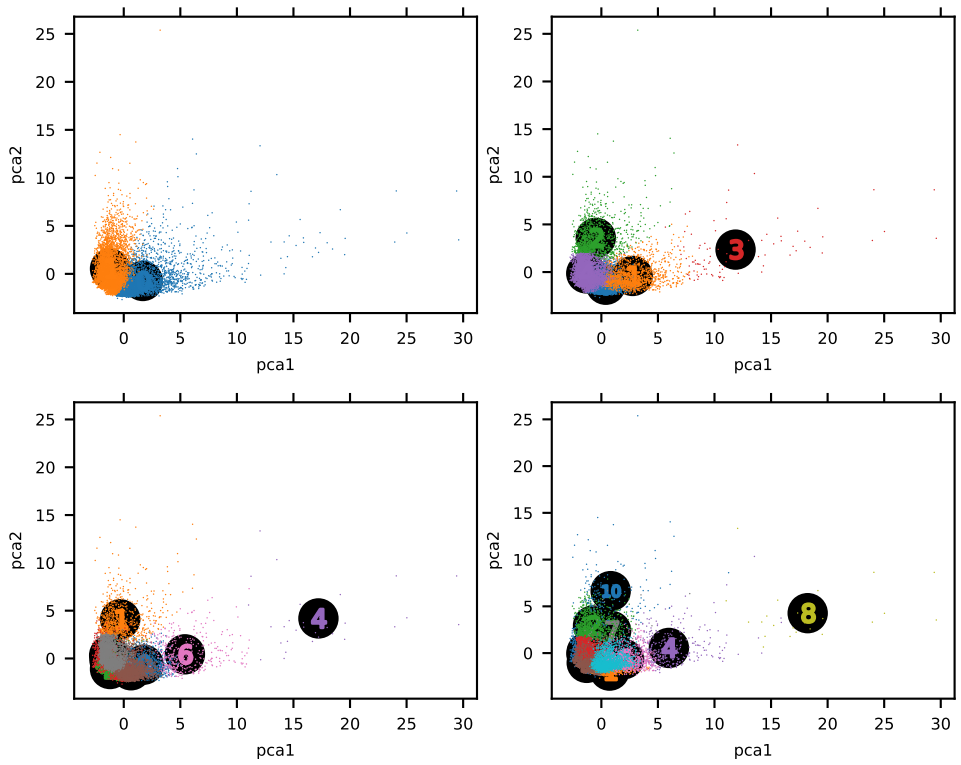
predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_errors	inertia
k-means.	17	8950	2	8950	0	4.89	0.4929	127785.29
k-means.	17	8950	5	8950	0	5.02	0.3085	91502.98
k-means.	17	8950	8	8950	0	5.80	0.2427	74492.63
k-means.	17	8950	11	8950	0	9.59	0.2760	63633.03
k-means.	17	8950	14	8950	0	11.27	0.2670	57500.31
k-means.	17	8950	17	8950	0	10.75	0.2514	53305.03
k-means.	17	8950	20	8950	0	7.61	0.2350	49690.91

6.3.2 Codpy

In this section we apply the same methodology as in the previous sections.

The result of codpy's sharp discrepancy algorithm is Ny clusters in D dimensions. Notice that the cluster centroids themselves are D-dimensional points.

Next we visualize the clusters and corresponding centroids of codpy's sharp discrepancy algorithm, where we vary the number of clusters Ny from 2 to 4 clusters.



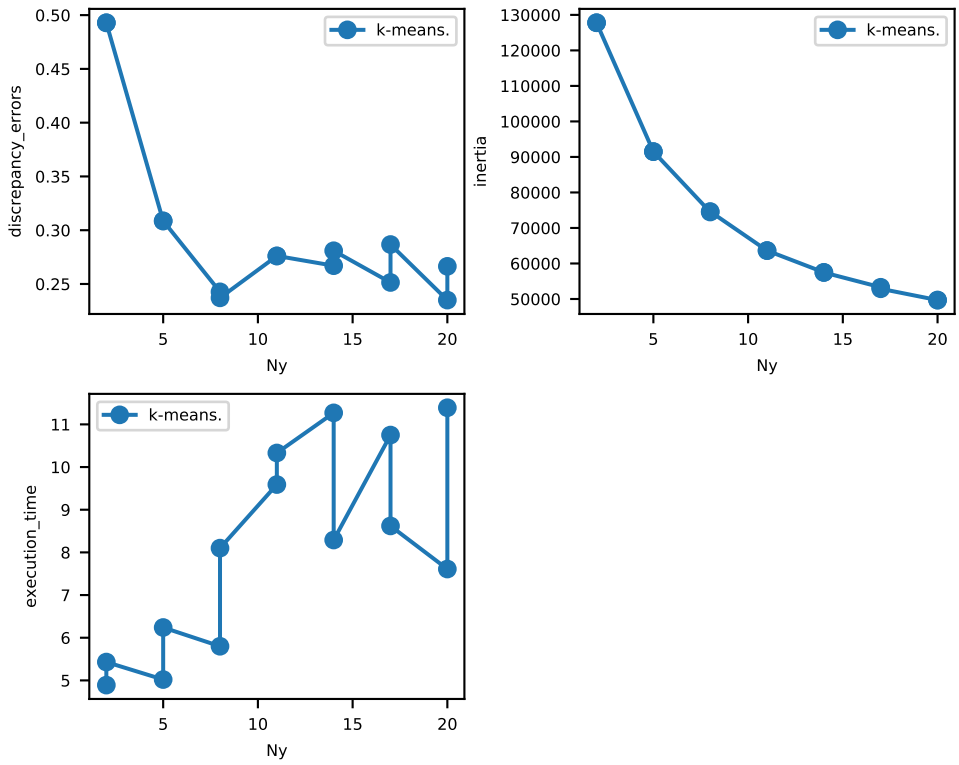
The table below demonstrates the performance of codpy's sharp discrepancy algorithm in terms of inertia, discrepancy errors and time.

Table 6.7: Codpy:2,4 clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_errors	inertia
k-means.	17	8950	2	8950	0	5.43	0.4928	127785.42
k-means.	17	8950	5	8950	0	6.24	0.3087	91502.96
k-means.	17	8950	8	8950	0	8.10	0.2371	74625.00
k-means.	17	8950	11	8950	0	10.33	0.2761	63633.12
k-means.	17	8950	14	8950	0	8.29	0.2809	57460.63
k-means.	17	8950	17	8950	0	8.62	0.2867	52877.36
k-means.	17	8950	20	8950	0	11.39	0.2664	49685.45

6.3.3 Benchmarks results

Finally, we illustrate a benchmark plot, displaying the computed performance indicator of Scikit's k-means and codpy's sharp discrepancy algorithms in terms of discrepancy errors, inertia, accuracy scores (when applicable) and execution time.



6.4 Credit card fraud detection

The dataset contains transactions made by credit cards in September 2013 by European cardholders. It presents transactions that occurred in two days, where we have 492 frauds out of 284, 807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

The study addresses the fraud detection system to analyze the customer transactions in order to identify the patterns that lead to frauds. In order to facilitate this pattern recognition work, the k-means clustering algorithm is used which is an unsupervised learning algorithm and applied to find out the normal usage patterns of credit card users based on their past activity [**?**].

It contains only numerical input variables which are the result of a PCA transformation. The only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning.

Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise. You can find more details on this Credit Card Fraud use case following this kaggle page link.

6.4.1 Scikit

We run our tests in the following line.

The table 6.8 displays inertia, discrepancy errors and execution time performance indicators.

Table 6.8: Scikit: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	scores	discrepancy_errors	inertia
k-means	30	499	15	284308	1	2.30	0.9496	0.6953	21363.21
k-means	30	499	30	284308	1	2.46	0.9296	0.6598	13892.58
k-means	30	499	45	284308	1	2.84	0.9407	0.6145	10898.61
k-means	30	499	60	284308	1	2.73	0.9794	0.5669	9082.48
k-means	30	499	75	284308	1	2.47	0.9326	0.5966	7609.51
k-means	30	499	90	284308	1	2.48	0.9801	0.5515	6446.03

Finally, we output as well the confusion matrix for the last scenario in figure 6.4.

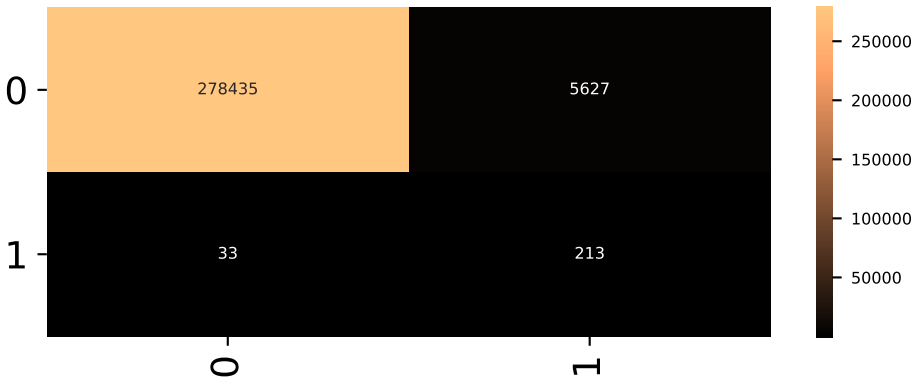


Figure 6.4: confusion matrix for k-means

6.4.2 Codpy

We repeat the same benchmark methodology as in the previous sections. We first run all scenarios.

The table 6.9 displays computed performance indicators for all scenarios.

Table 6.9: codpy: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	scores	discrepancy_errors	inertia
codpy	30	499	15	284308	1	0.74	0.9678	0.3248	21499.18
codpy	30	499	30	284308	1	0.78	0.9753	0.3145	13930.79
codpy	30	499	45	284308	1	0.86	0.9720	0.3132	11054.61
codpy	30	499	60	284308	1	0.96	0.9658	0.3136	9084.62
codpy	30	499	75	284308	1	0.98	0.9714	0.3115	7657.18
codpy	30	499	90	284308	1	1.08	0.9658	0.3087	6520.76

Finally, we output as well the confusion matrix for the last scenario in figure 6.5.

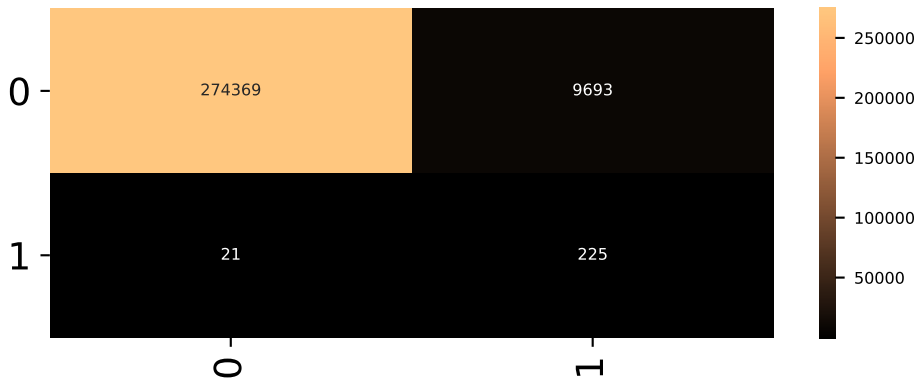
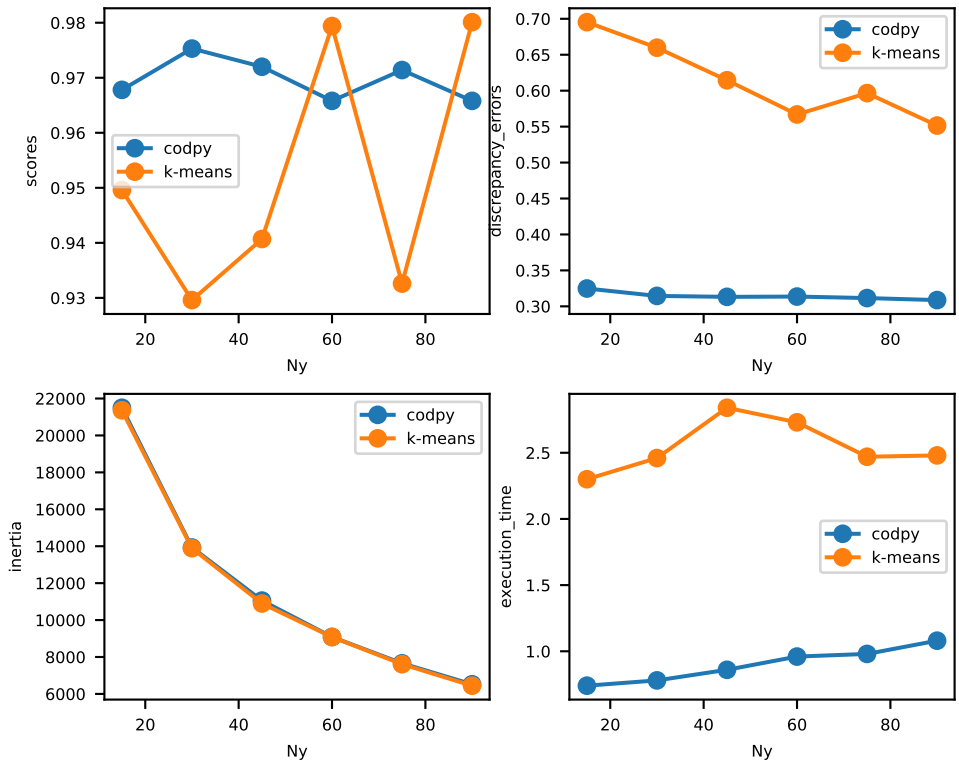


Figure 6.5: confusion matrix for codpy

6.4.3 Benchmarks results

Finally, we illustrate a benchmark plot, that shows the performance of Scikit's k-means and codpy's sharp discrepancy algorithms in terms of discrepancy errors, inertia, accuracy scores (when applicable) and execution time.



6.5 Portfolio of stock clustering

This case represents daily stock price movements $x \in \mathbb{R}^{N_x \times D}$ (i.e. the dollar difference between the closing and opening prices for each trading day) from 2010 to 2015.

6.5.1 Scikit

The tests follows the very same method as in the previous section. We first run our scenarios in the following line.

The table 6.10 displays inertia, discrepancy errors and execution time performance indicators.

Table 6.10: Scikit: Ny clusters

	x
0	Pfizer , Sanofi-Aventis
1	Cisco , Microsoft
2	Canon , Ford , Honda , Mitsubishi, Sony , Toyota
3	Boeing , Lookheed Martin , Northrop Grumman
4	British American Tobacco, ConocoPhillips , Chevron , GlaxoSmithKline , Novartis , Royal Dutch Shell , SAP , S
5	American express , Caterpillar , DuPont de Nemours, General Electrics, Home Depot , IBM , Johnson & Johnson
6	Walgreen
7	AIG , Bank of America, Goldman Sachs , JPMorgan Chase , Wells Fargo
8	Colgate-Palmolive, Kimberly-Clark , Procter Gamble , Wal-Mart
9	Navistar
10	Philip Morris
11	Yahoo
12	Dell, HP
13	Valero Energy
14	Apple
15	Amazon , Google/Alphabet
16	Intel , Taiwan Semiconductor Manufacturing, Texas instruments
17	MasterCard
18	Coca Cola, Pepsi
19	Xerox

k-means clusters displays stocks into coherent groups. The performance indicators for this step are the following.

Table 6.11: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_errors	inertia
k-means.	963	60	10	60	0	0.69	0.3807	25.58
k-means.	963	60	20	60	0	0.62	0.2067	18.47

6.5.2 Codpy

The tests follows the very same method as in the previous section. We first run our scenarios in the following line.

The table 6.12 displays inertia, discrepancy errors and execution time performance indicators.

Table 6.12: Scikit: Ny clusters

predictor_id	D	Nx	Ny	Nz	Df	execution_time	discrepancy_errors	inertia
k-means.	963	60	10	60	0	0.73	0.4004	25.19
k-means.	963	60	20	60	0	0.61	0.2126	18.03

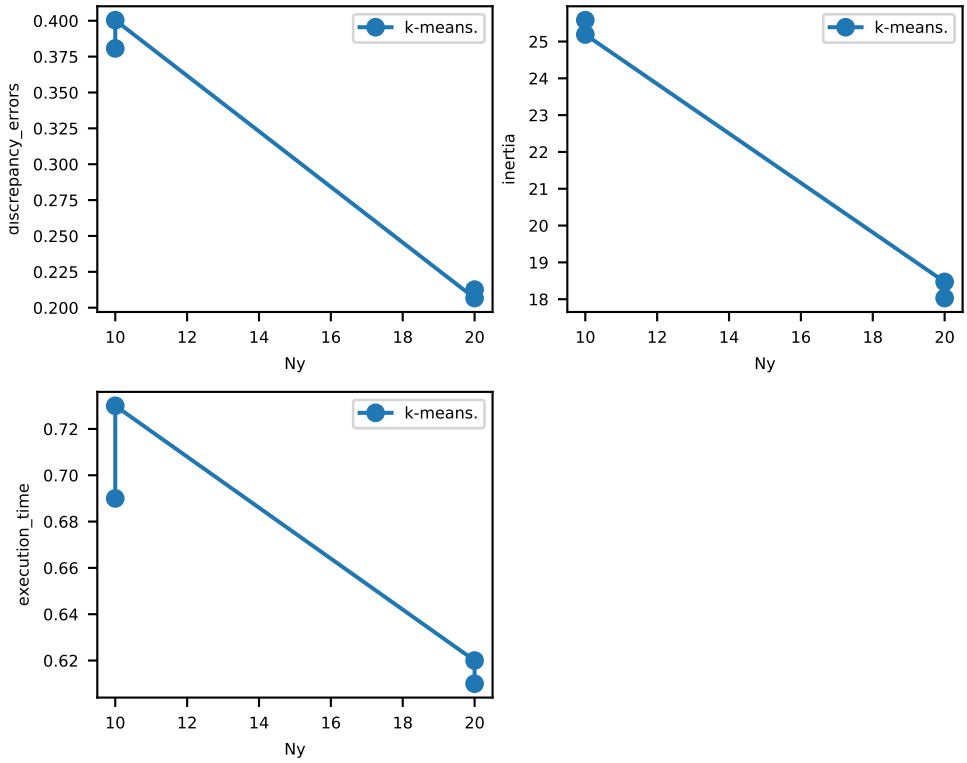
k-means clusters displays stocks into coherent groups. The performance indicators for this step are the following.

Table 6.13: Codpy: Stocks' clusters

	Stocks
0	American express , Bank of America , Ford , General Electrics, Goldman Sachs , JPMorgan Chase , Wells Fargo
1	British American Tobacco, GlaxoSmithKline , Novartis , Royal Dutch Shell , SAP , Sanofi-Aventis , Total , Unile
2	Philip Morris
3	Colgate-Palmolive, Kimberly-Clark , Procter Gamble
4	Mitsubishi
5	Caterpillar , ConocoPhillips , Chevron , DuPont de Nemours, IBM , 3M , Schlumberger , Valero Energy , Exxon
6	Coca Cola, Pepsi
7	Canon , Honda , Sony , Toyota
8	Navistar
9	Intel , Taiwan Semiconductor Manufacturing, Texas instruments
10	McDonalds
11	Johnson & Johnson, Pfizer , Walgreen , Wal-Mart
12	Boeing , Lookheed Martin , Northrop Grumman
13	Apple , Amazon , Google/Alphabet
14	Yahoo
15	Dell, HP
16	Cisco , Microsoft, Symantec
17	AIG
18	MasterCard
19	Home Depot

6.5.3 Benchmarks results

Finally, we illustrate a benchmark plot, that shows the performance of Scikit's k-means and codpy's sharp discrepancy algorithms in terms of discrepancy errors, inertia, accuracy scores (when applicable) and execution time.




```
## time_series loaded
```

Chapter 7

Application to optimal transport

7.1 Bachelier problem

7.1.1 Introduction

The purpose of this test is to benchmark the Pi-function, see section **??** with the Bachelier problem, that is described in the next section.

It is known since a decade that deep learning methods can be described by kernel methods, see for instance [**?**]. We illustrate this fact with a kernel method, designed with a quite comparable spirit to the Neural network approach. Indeed, this method gives quite comparable results to the NN one : both methods are not convergent, and we do not advise their use for critical applications.

7.1.2 Problem description

- Consider a martingale process $t \mapsto X(t) \in \mathbb{R}^D$, given by the Brownian motion $dX = \sigma dW_t$, where the matrix $\sigma \in \mathbb{R}^{D\times D}$ is randomly generated. The initial condition is $X(0) = (1, \cdots, 1)$, w.l.o.g.
- Consider two times $1 = t^1 < t^2 = 2$, t^2 being the maturity of an option, that is a function denoted $P(x) = \max(b(x) - K, 0)$, where $K = 1.1$, $b(x) := x \cdot a$ with random weights $a \in \mathbb{R}^D$. It is straightforward to verify that $b(x)$ follows a Brownian motion $db = \theta dW_t$. To get a fixed value for θ (fixed to 0.2 in our tests), we normalize the diffusion matrix σ above.
- The goal of this test is to benchmark numerical methods aiming to compute the following conditional expectation
$$f(z) := \mathbb{E}^{X(t^2)}\Big(P(\cdot)|X(t^1) = z\Big).$$
For the Bachelier problem, this last quantity can be determined using a closed formula : the reference value is computed as
$$f(z) = \theta\sqrt{t^2 - t^1}pdf(d) + (b(x) - K)cdf(d), \quad d(x, K) := \frac{b(x) - K}{\theta\sqrt{t^2 - t^1}} \tag{7.1.1}$$
pdf (resp. cdf) holding for the probability density function (resp. cumulative) of the normal law.

7.1.3 Methodology, Notations, input and output data

For our tests, we use the following notations, and precise their signification for this report, and more generally for Finance applications:

- $x \in \mathbb{R}^{N_x \times D}$ denotes the training set of variables. For our test, this set is given by iid samples of the brownian motion $X(t)$ at time $t^1 = 1$.
  - For quantitative Finance applications, this set typically consists in N_x iid samples of a stochastic process $t \mapsto X(t) \in \mathbb{R}^D$ at a time t. Such samples might be generated by discretization of stochastic processes using Euler methods for instance.
- $f(x) \in \mathbb{R}^{N_x \times D_f}$ denotes the training set of values. It is generated as $P(X(t^2)|x)$, P being the payoff of the option described in the previous section, x being the training set.
  - For Finance applications, $f \in \mathcal{C}^1(\mathbb{R}^D)^M$ is usually a derivable function, having vector valued values corresponding to payoffs, or investment strategies, of portfolios.
- $z \in \mathbb{R}^{N_z \times D}$ denotes the test set of variables. It consists for our test as another iid realization of the brownian motion $X(t)$ at time $t^1 = 1$.
  - This set represent usually a set of user defined samples of underlying risks, chosen accordingly to its needs.
- $f(z) \in \mathbb{R}^{N_z \times D_f}$ is the set of reference values, computed using (7.1.1) ($D_f = 1$ in this experiment)
  - This set consists in reference - ground truth - values, approximating $P(z) := \mathbb{E}(f(x_{t^2}|z))$.

To these set we added another one, used for internal computations

- $y \in \mathbb{R}^{N_y \times D}$, with $N_y << N_x$.
  - This set corresponds to the weight set for neural networks methods.
  - This set corresponds to what we call a "projection set" for kernel methods (see [**?**] for a definition).

Output data are

- $f_z \in \mathbb{R}^{N_z \times D_f}$ the set of predicted values. These are the values that are benchmarked against $f(z)$ in our experiments.

For each numerical experiments, we output a table summarizing the values of N_x, N_y, N_z, D.

7.1.4 Four methods to tackle the Bachelier problem

In this technical report, we compare four methods for tackling the computation of conditional expectations for the Bachelier problem :

1. The first is a Neural Network one, using Tensorflow to retrieve results. Code can be downloaded at [**?**], the method itself is described in [**?**].
2. The second one is a standard kernel method, quite comparable to the previous approach, using codpy implementation of kernel methods, that is the textit {projection} function

$$P^k(x, y, z, f(z) = []) = k(z, y)k(x, y)^{-1} f(z) \tag{7.1.2}$$

where y is a N_y-size random shuffling of the initial set x, and $k(x, y)$ is a Gram matrix, see [**?**] for a more detailed description.

3. The third one uses the Pi function above, where x (resp. z) are iid sequences of $X(t^1)$ (resp. $X(t^2)$), as presented in the above section.
4. The fourth one uses also the Pi function above, but choosing x (resp. z) as sharp discrepancy sequences of $X(t^1)$, (resp. $X(t^2)$) see [**?**] - [**?**].

7.1.5 Test specification

A single test relies on 8 parameters, that we list below. We will be running several scenario to benchmark our results.

Table 7.1: A test specification

Nx	Ny	Nz	D	s1	s2	s3
# xs	# ys	# zs	Dimension	Generator x	Generator z | x	Generator z

Hence this numerical experiment uses s_1, s_2, s_3, three seeds for random generators:

- s_1 is used to generate iid samples of $X(t^1)$ for the training set of variables x.
- s_2 is used to generate iid samples of the conditional sampling $X(t^2)|X(t^1) = x$ for the training set of variables.
- s_3 is used to generate iid samples of $X(t^1)$ for the test set of variables z.

For instance, if $s_1 = s_3$, and $N_z < N_x$, then the test set is a subset of the training set $z \subset x$.

To summarize the methodology, for each scenario of a given 8-uple $N_x, N_y, N_z, D, D_f, s_1, s_2, s_3$, each of our three methods output a prediction f_z, that is benchmarked against the ground-truth value $f(z)$.

To measure errors, we use the percentage RMSE error, expressed as a real number between 0 and 1, called **basis point error**, as follows:

$$RMSE\%(f_z, f(z)) = \frac{\|f_z - f(z)\|_{\ell^2}}{\|f_z\|_{\ell^2} + \|f(z)\|_{\ell^2}}$$

We presents three tests. Two of them are two-dimensional ($D = 2$), allowing graphical representation of input data and output errors to best illustrate our three methods. The third one is concerned with higher dimensional case.

7.1.5.1 Input data settings

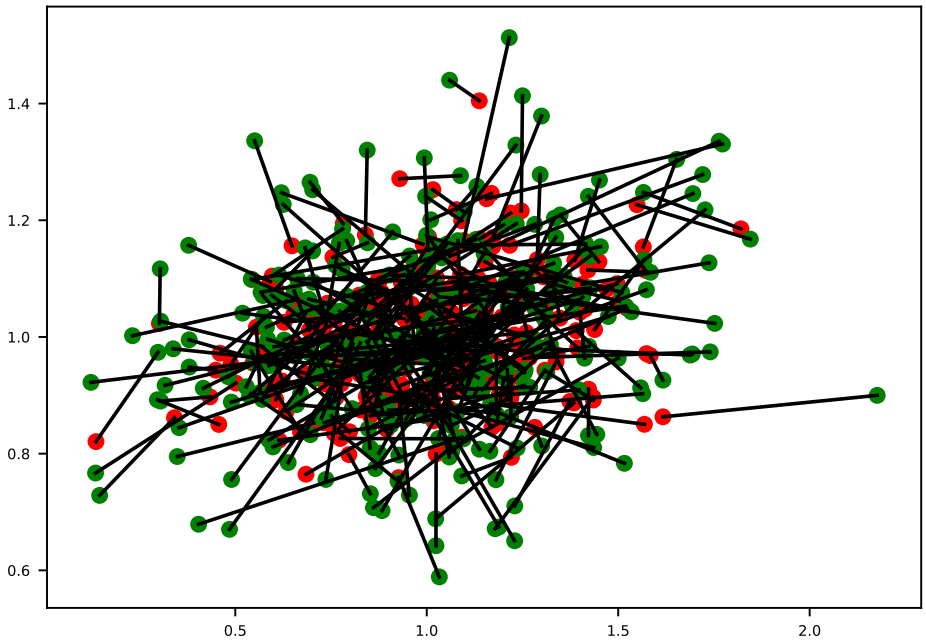
We generate our data set for the test, accordingly to the description given in the previous paragraph.

7.1.5.1.1 Training set for different times We plot $x = X(t^1)$ (training set), generated at time $t^1 = 1$ with $X(t^2)|X(t^1)$, that are the trajectories generated at time $t^2 = 2$. We plot a line between each x and $z|x$

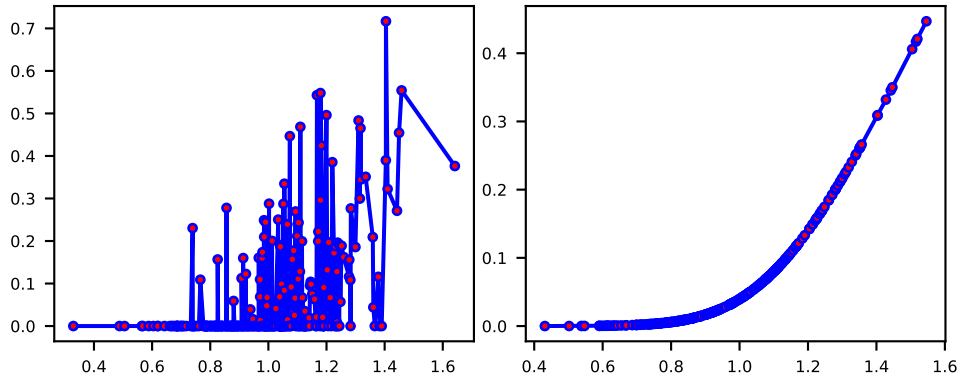
```
## <matplotlib.collections.PathCollection object at 0x0000000358F3A948>
```

```
## <matplotlib.collections.PathCollection object at 0x000000035934F948>
```

```
## [<matplotlib.lines.Line2D object at 0x0000000357DEBEC8>, <matplotlib.lines.Line2D object at 0x000
```



7.1.5.1.2 Training values and ground test values distributions We plot the generated learning and test set in the following picture, comparing the variable $f(x)$ and the exact to predict $f(z)$, taking as x-axis the corresponding values of $b(x), b(z)$.



7.1.6 Running the tests

7.1.6.1 Standard Neural Network

This test uses part of the code available at [**?**]. Our chosen scenarios are listed in Table 7.2

The Table 7.3 output the values of this test

```
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
```

Table 7.2: scenario list

D	Nx	Ny	Nz
2	32	80	512
2	64	80	512
2	128	80	512
2	256	80	512
2	512	80	512
2	1024	80	512
2	2048	80	512
2	4096	80	512
2	8192	80	512
2	16384	80	512
2	32768	80	512

Table 7.3: tensorflow indicators

	0	1	2	3	4	5	6	7	8	9	10
predictor_id	ANN	ANN	ANN	ANN	ANN	ANN	ANN	ANN	ANN	ANN	ANN
D	2	2	2	2	2	2	2	2	2	2	2
Nx	32	64	128	256	512	1024	2048	4096	8192	16384	32768
Ny	32	64	80	80	80	80	80	80	80	80	80
Nz	512	512	512	512	512	512	512	512	512	512	512
Df	1	1	1	1	1	1	1	1	1	1	1
execution_time	0.2	0.22	0.22	0.24	0.27	0.55	0.54	0.95	1.35	1.51	2.1
scores	0.2039	0.5269	0.1735	0.1751	0.0801	0.0971	0.0787	0.104	0.0868	0.0777	0.0682
norm_function	0.34	0.21	0.23	0.25	0.24	0.28	0.29	0.23	0.34	0.32	0.26
discrepancy_errors	0.1272	0.0725	0.0956	0.1067	0.1166	0.211	0.2112	0.2314	0.1915	0.2375	0.2127

```
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
##
## HBox(children=(IntProgress(value=0, description='standard training', style=ProgressStyle(descript
```

We output the predicted values f_z against the exact ones $f(z)$, as functions of the basket values $b(z)$ in Figure 7.1

7.1.6.2 Standard codpy kernel

We provide the same approach with the kernel projection operator. The list of scenario for this test is Table 7.4

The Table 7.5 output the values of this test

We output the predicted values f_z against the exact ones $f(z)$, as functions of the basket values $b(z)$ in Figure 7.2

7.1.6.3 Pi function

We provide the same approach with the Pi function. The list of scenario for this test is Table 7.6

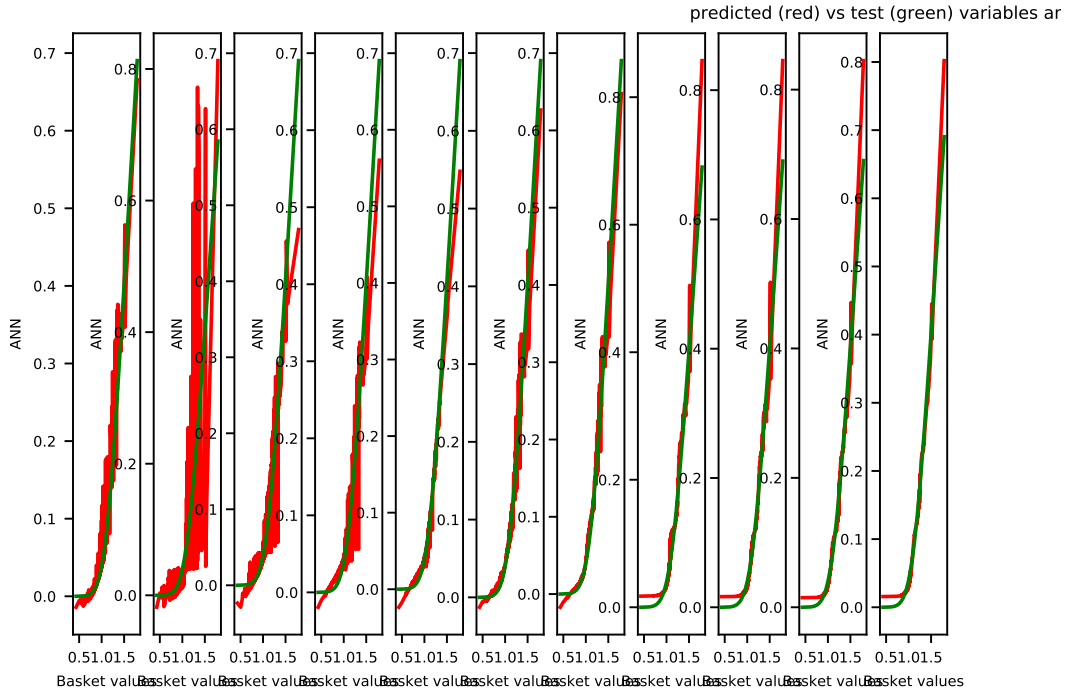


Figure 7.1: exact and predicted values for Tensoflow

Table 7.4: scenario list

D	Nx	Ny	Nz
2	32	80	512
2	64	80	512
2	128	80	512
2	256	80	512
2	512	80	512
2	1024	80	512
2	2048	80	512
2	4096	80	512
2	8192	80	512
2	16384	80	512
2	32768	80	512

Table 7.5: codpy predictor indicators

	0	1	2	3	4	5	6	7	8	9	10
predictor_id	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred	codpy pred
D	2	2	2	2	2	2	2	2	2	2	2
Nx	32	64	128	256	512	1024	2048	4096	8192	16384	32768
Ny	32	64	80	80	80	80	80	80	80	80	80
Nz	512	512	512	512	512	512	512	512	512	512	512
Df	1	1	1	1	1	1	1	1	1	1	1
execution_time	0	0	0	0	0.01	0.01	0.01	0.02	0.03	0.05	0.09
scores	0.4813	0.5978	0.339	0.3737	0.2984	0.2273	0.1422	0.1332	0.1003	0.0779	0.0859
norm_functions	0.34	0.21	0.23	0.25	0.24	0.28	0.29	0.23	0.34	0.32	0.26
discrepancy_errors	0.1272	0.0725	0.0956	0.1067	0.1166	0.211	0.2112	0.2314	0.1915	0.2375	0.2127

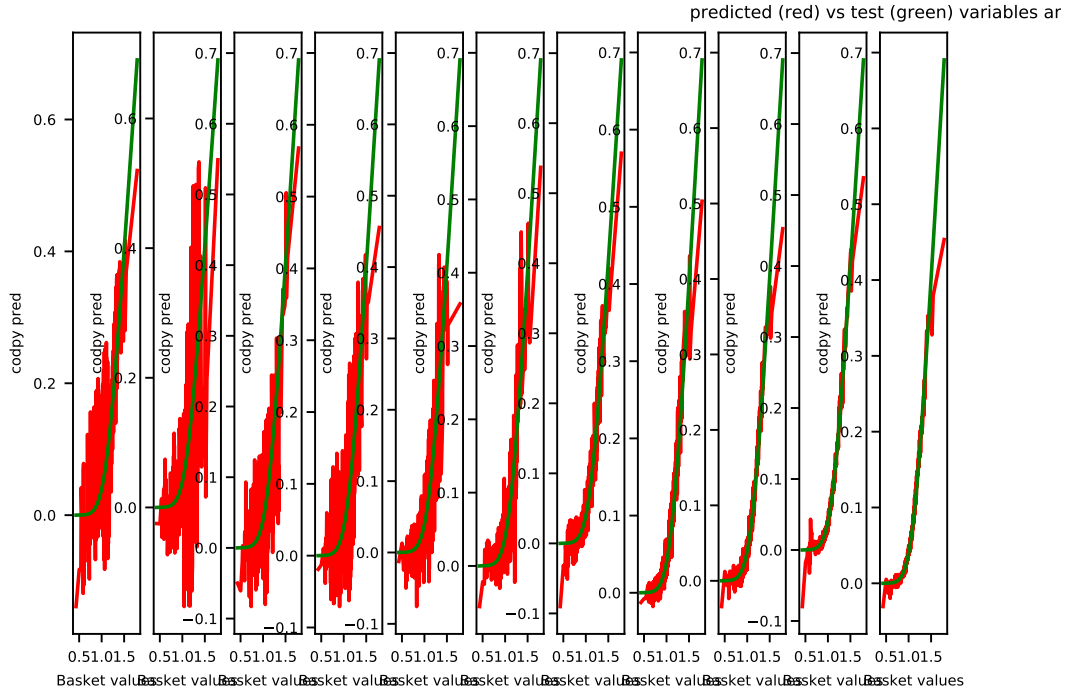


Figure 7.2: exact and predicted values for projection

Table 7.7: Pi indicators

	0	1	2	3	4
predictor_id	Pi:i.i.d.	Pi:i.i.d.	Pi:i.i.d.	Pi:i.i.d.	Pi:i.i.d.
D	2	2	2	2	2
Nx	32	64	128	256	512
Ny	32	64	128	256	512
Nz	32	64	128	256	512
Df	1	1	1	1	1
execution_time	0.15	0.33	1.52	6.19	29.02
scores	0.3319	0.2163	0.1653	0.0972	0.0689
norm_function	0.01	0.01	0.01	0.02	0.01
discrepancy_errors	0.1466	0.1287	0.1034	0.0422	0.0404

Table 7.6: scenario list

D	Nx	Ny	Nz
2	32	32	32
2	64	64	64
2	128	128	128
2	256	256	256
2	512	512	512

The Table 7.7 output the values of the tests

We output the predicted values f_z against the exact ones $f(z)$, as functions of the basket values $b(z)$ in Figure 7.3

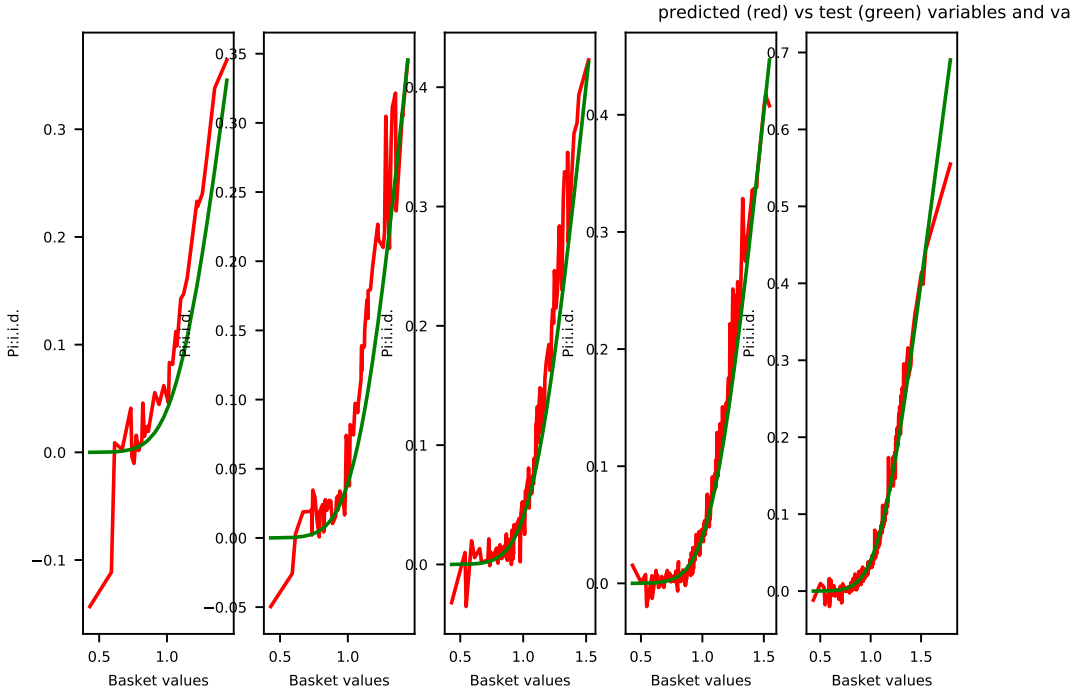


Figure 7.3: exact and predicted values for Pi

Table 7.8: scenario list

D	Nx	Ny	Nz
2	32	32	32
2	64	64	64
2	128	128	128
2	256	256	256
2	512	512	512

7.1.6.4 Pi function - discrepancy sequences

We provide the same approach with the Pi function, with sharp discrepancy sequences. The list of scenario for this test is Table 7.8

The Table 7.9 output the values of the tests

We output the predicted values f_z against the exact ones $f(z)$, as functions of the basket values $b(z)$ in Figure 7.4

7.1.7 Comparing methods

The Figure 7.5 presents a benchmark for scores, computed accordingly to (7.1.5). Axis are in log-scale of the size of the training N_x.

The Figure 7.6 presents a benchmark regarding execution times in seconds. Axis are in log-scale of the size of the training N_x.

Table 7.9: Pi-sharp indicators

	0	1	2	3	4
predictor_id	Pi:sharp	Pi:sharp	Pi:sharp	Pi:sharp	Pi:sharp
D	2	2	2	2	2
Nx	32	64	128	256	512
Ny	32	64	128	256	512
Nz	32	64	128	256	512
Df	1	1	1	1	1
execution_time	0.07	0.26	1.71	6.02	30.16
scores	0.0922	0.1034	0.0762	0.0834	0.0477
norm_function	0.28	0.28	0.35	0.41	0.46
discrepancy_errors	0	0	0	0	0

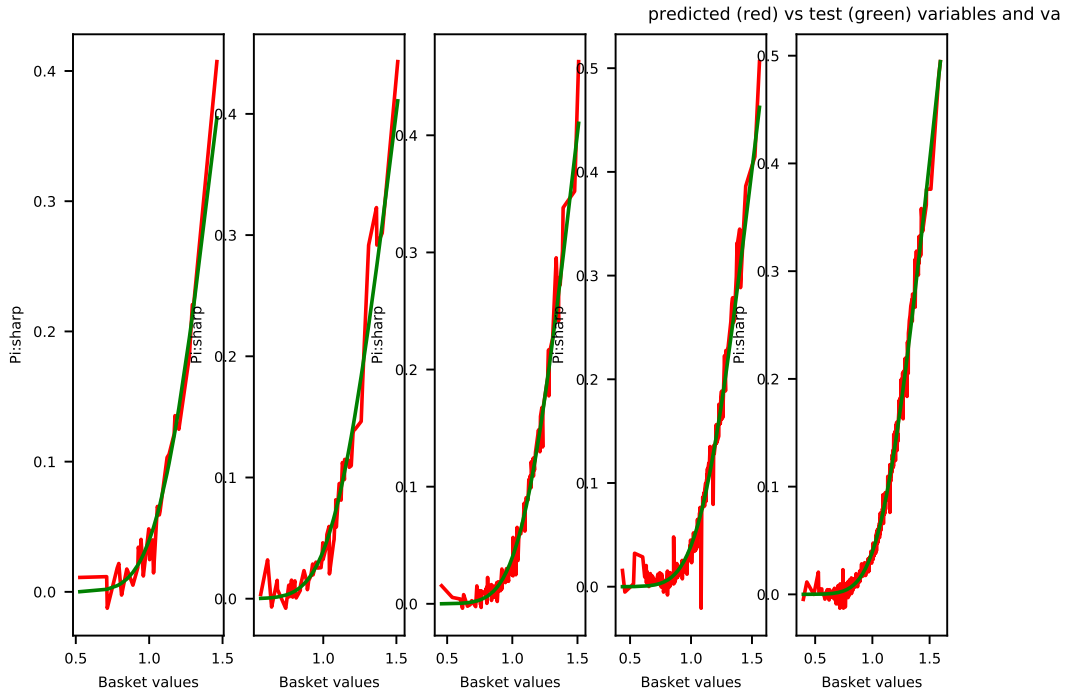


Figure 7.4: exact and predicted values for Pi-sharp

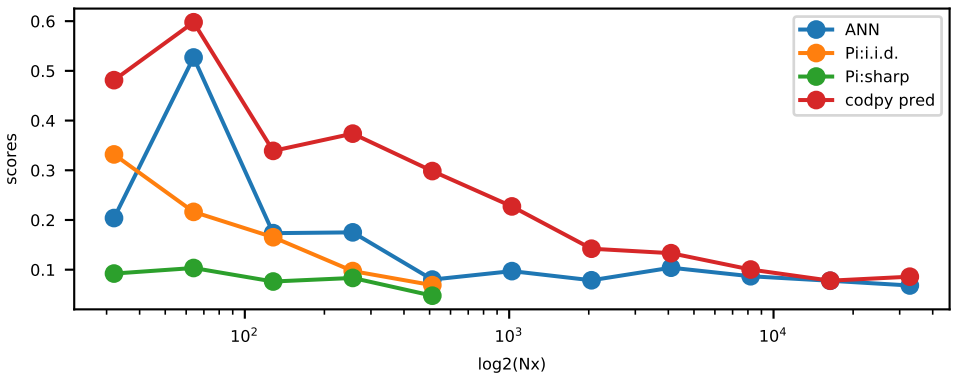


Figure 7.5: Benchmark of scores

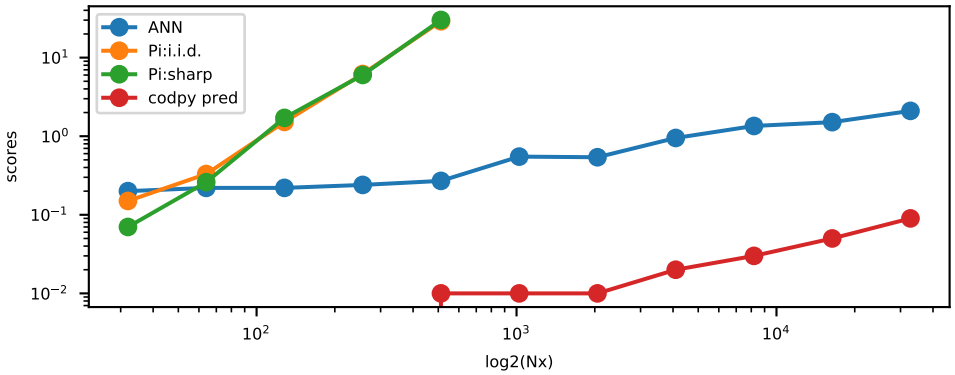


Figure 7.6: Benchmark of execution times

7.2 Time series

This section remains to write properly.

7.2.1 Recurrent kernels

The implemented method is defined using two integer values : H and P. H is called the historical depth, P the prediction depth. This setting defines a sliding window of size H+P over the dataset, used to define the training set. If the dataset contains N vectors, then the training set can be of size N-H-P. We can iterate the procedure, producing at each step P new predicted values. This allows, theoretically, to produce predicted values of the temporal series at any future times.

This method allows to draw one trajectory, that can be considered as a iid realization of the temporal series, based on the knowledge of its history. On the following example, H and P are set to 360 days. Here the separation date is the 23/11/2020.

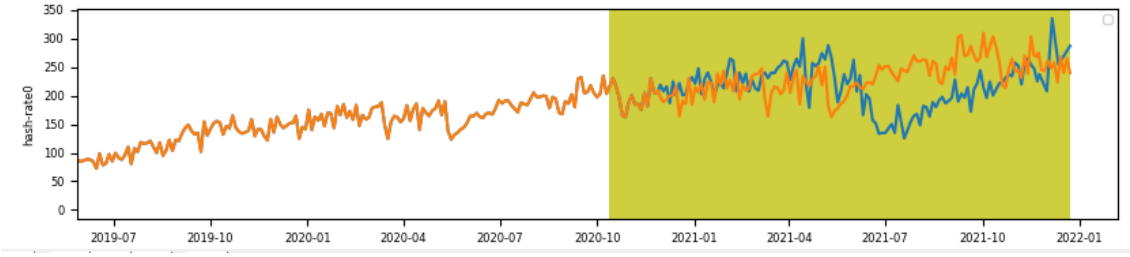


Figure 7.7: The generated BTC-USD curve is the yellow one.

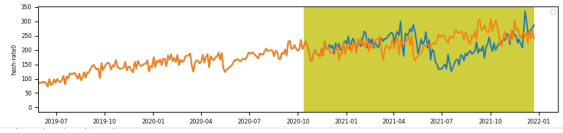


Figure 7.8: The generated hash-rate curve is the yellow one.

This method has a lot of forecasting applications, and we do use it for professional purposes. However, in the context of temporal series forecasting, such a method faces a number of questions. For instance :

- It is not clear how to generate other realizations of the studied temporal series.
- As a consequence, it is not clear neither how to generate a pertinent mean estimator using this construction.

Even if long-short term memory, or recurrent networks can produce credible generated samples, beware to unstability issues. We have no theoretical references to support these methods.

7.2.2 Optimal transport methods for time series

Kernel methods can link easily with optimal transport theory. Using the polar factorization of maps, we can also compute explicitly the quantile of the original distribution, and extrapolate it on any random trajectory set, and we can draw "equi-probable" trajectories (i.e. iid realizations of the underlying process).

We have also a quite clear interpretation of a mean estimator and the method is quite performing.

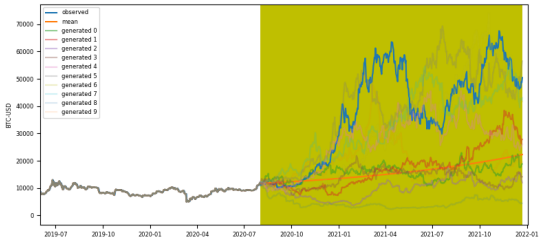


Figure 7.9: Generated sample and mean estimator for the BTC hash-rate curve.

7.3 Stress and reverse stress tests

Chapter 8

Application to partial differential equations

Bibliography

[1] A. Antonov and M. Konikov and M. Spector, The free boundary SABR: natural extension to negative rates, unpublished report, January 2015, available at https://ssrn.com/abstract=2557046.

[2] I. Babuska, U. Banerjee, and J.E. Osborn, Survey of mesh-less and generalized finite element methods: a unified approach, Acta Numer. 12 (2003), 1–125.

[3] A. Berlinet and C. Thomas-Agnan, *Reproducing kernel Hilbert spaces in probability and statistics,* Springer US, Kluwer Academic Publishers, 2004.

[4] M.A. Bessa, and J.T. Foster, T. Belytschko, and W.K. Liu, A mesh-free unification: reproducing kernel peridynamics, Comput. Mech. 53 (2014), 1251–1264.

[5] A. Brace, and D. Gatarek and M. Musiela, The market model of interest rate dynamics, Math. Finance 7 (1997), 127–154.

[6] H. Brezis, Remarques sur le problème de Monge–Kantorovich dans le cas discret, Comptes Rendus Mathematique 356 (2018), 207–213.

[7] Y. Brenier, Polar factorization and monotone rearrangement of vector-valued functions, Comm. Pure Applied Math. XLIV (1991), 375–417.

[8] H. Buehler, Volatility and dividends: volatility modeling with cash dividends and simple credit risk, February 2010, available at: https://ssrn.com/abstract=1141877.

[9] Financial Service Authority CP 08/24, PS 09/20 and GL32

[10] G.E. Fasshauer, *Mesh-free methods,* in "Handbook of Theoretical and Computational Nanotechnology", Vol. 2, 2006.

[11] G.E. Fasshauer, *Mesh-free approximation methods with Matlab,* Interdisciplinary Math. Sciences, Vol. 6, World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2007.

[12] G.E. Fasshauer, Positive definite kernels: past, present and future, unpublished report, available at http://www.math.iit.edu/ fass/PDKernels.pdf.

[13] A. Gretton, K. Borgwardt, M. J. Rasch, B. Scholkopf, A. J. Smola, A Kernel Method for the Two-Sample Problem, arXiv:0805.2368

[14] F.C. Günther and W.K. Liu, Implementation of boundary conditions for mesh-less methods, Comput. Methods Appl. Mech. Engrg. 163 (1998), 205–230.

[15] E. Haghighat, M. Raissib, A. Moure, H. Gomez, and R. Juanes, A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics, Comput. Methods Appl. Mech. Engrg. 379 (2021), 113741

[16] B.N. Huge and A. Savine, Differential machine learning, unpublished report, January 2020, available at https://ssrn.com/abstract=3591734

[17] Charles Gustave Jacob Jacobi, «De investigando ordine systematis aequationum differentialum vulgarium cujuscunque», herausgegeben von K. Weierstrass, Berlin, Bruck und Verlag von Georg Reimer, 1890, p.193-216

[18] T.F. Korzeniowski and K. Weinberg, A multi-level method for data-driven finite element computations, Comput. Methods Appl. Mech. Engrg. 379 (2021), 113740.

[19] J.J. Koester and J.-S. Chen, Conforming window functions for mesh-free methods, *Comm. Numer. Methods Engrg.* 347 (2019), 588–621.

[20] Y. LeCun, C. Cortes, and C.J.C. Burges, The MNIST database of handwritten digits, http://yann.lecun.com/exdb/mnist/

[21] J.-M. Mercier, Optimally Transported schemes. Applications to Mathematical Finance, unpublished, https://www.researchgate.net/publication/228689632_Optimally_Transported_schemes_Applications_to_Mathematical_Finance

[22] J.-M. Mercier, A High-Dimensional Pricing Framework for Financial Instruments Valuation, DOI:10.2139/ssrn.2432019

[23] P.G. LeFloch and J.-M. Mercier, Revisiting the method of characteristics via a convex hull algorithm, J. Comput. Phys. 298 (2015), 95–112.

[24] P.G. LeFloch and J.-M. Mercier, A new method for solving Kolmogorov equations in mathematical finance, C. R. Math. Acad. Sci. Paris 355 (2017), 680–686.

[25] P.G. LeFloch and J.-M. Mercier, The Transport-based Mesh-free Method (TMM). A short review, The Wilmott journal 109 (2020), 52–57. Available at ArXiv:1911.00992.

[26] P.G. LeFloch and J.-M. Mercier, Mesh-free error integration in arbitrary dimensions: a numerical study of discrepancy functions, Comput. Methods Appl. Mech. Engrg. 369 (2020), 113245.

[27] P.G. LeFloch and J.-M. Mercier, A class of mesh-free algorithms for mathematical finance, machine learning, and fluid dynamics, Preprint February 2021. Available at ssrn.com/abstract=3790066.

[28] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: a tutorial, January 2021, available at ssrn.com/abstract=3769804.

[29] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: an advanced tutorial, January 2021, available at ssrn.com/abstract=3769804.

[30] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: a kernel-based reordering algorithm, January 2021, available at ssrn.com/abstract=3770557.

[31] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: RKHS-based polar factorization and sampling algorithm, in preparation.

[32] P.G. LeFloch, J.M. Mercier, and Sh. Miryusupov, CodPy: RKHS-based algorithms and conditional expectations, in preparation.

[33] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: Support Vector Machines (SVM) for (reverse) stress tests in finance, in preparation.

[34] S.F. Li and W.K. Liu, *Mesh-free particle methods,* Springer Verlag, Berlin, 2004.

[35] G.R. Liu, *Mesh-free methods: moving beyond the finite element method,* CRC Press, Boca Raton, FL, 2003.

[36] G.R. Liu, An overview on mesh-free methods for computational solid mechanics, *Int. J. Comp. Methods* 13 (2016), 1630001.

[37] J.-M. Mercier and Sh. Miryusupov, Hedging strategies for net interest income and economic values of equity, unpublished report, September 2019, available at: https://ssrn.com/abstract=3454813.

[38] Y. Nakano, Convergence of mesh-free collocation methods for fully nonlinear parabolic equations, Numer. Math. 136 (2017), 703–723.

[39] F. Narcowich, J. Ward, and H. Wendland, Sobolev bounds on functions with scattered zeros, with applications to radial basis function surface fitting, *Math. of Comput.* 74 (2005), 743–763.

[40] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods,* CBMS-NSF Regional Conf. Series in Applied Math., Soc. Industr. Applied Math., 1992.

[41] H.S. Oh, C. Davis, and J.W. Jeong, Mesh-free particle methods for thin plates, *Comput. Methods Appl. Mech. Engrg.* 209/212 (2012), 156–171.

[42] R. Opfer, Multiscale kernels, Adv. Comput. Math. 25 (2006), 357–380.

[43] R. Salehi and M. Dehghan, A moving least square reproducing polynomial mesh-less method, Appl. Numer. Math. 69 (2013), 34–58.

[44] M. Sathyapriya, Dr. V. Thiagarasu, A cluster-based approach for credit card fraud detection system using Hmm with the implementation of big data technology, Report 2019.

[45] J. Sirignano and K. Spiliopoulos, DGM: a deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018), 1339–1364.

[46] P. Traccucci, L. Dumontier, G. Garchery, B. Jacot, A Triptych Approach for Reverse Stress Testing of Complex Portfolios. arXiv:1906.11186

[47] R.S. Varga, *Matrix iterative analysis,* Springer Verlag, 2000.

[48] C. Villani, *Optimal transport, old and new,* Springer Verlag, 2009.

[49] H. Wendland, Sobolev-type error estimates for interpolation by radial basis functions, in "Surface fitting and multiresolution methods" (Chamonix-Mont-Blanc, 1996), Vanderbilt Univ. Press, Nashville, TN, 1997, pp. 337–344.

[50] H. Wendland, *Scattered data approximation,* Cambridge Monograph, Applied Comput. Math., Cambridge University, 2005.

[51] J.X. Zhou and M.E. Li, Solving phase field equations using a mesh-less method, Comm. Numer. Methods Engrg. 22 (2006), 1109–1115.

[52] B. Zwicknagl, Power series kernels, Constructive Approx. 29 (2008), 61–84.