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

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

2022-10-03

[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

This a preliminary version of a monograph in preparation This version: April 5, 2022.

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 the "Curse Of Dimensionality in PYthon". This library provides the user with a support vector machine (SVM) and encompasses a broad set of applications. The proposed algorithms apply to partial differential equations arising, for instance, in mathematical finance and fluid dynamics, as well as to discrete models arising in machine learning and statistics. Our basic numerical strategy relies on a combination of ideas from the theory of reproducing kernel Hilbert spaces (RKHS) and the theory of optimal transport. The authors have developed this library over the past decade, originally for applications in mathematical finance, and it has now reach a stage where it can be applied to problems of interest to engineering and industry.

We proceed by presenting first, in several tutorial chapters (Chapters 2 to 4), fundamental notions about kernel-based discretizations as we proposed for the CodPy library, and at this stage we only include elementary examples which illustrate the relevance of these fundamental concepts. In the second part of this monograph (Chapters 5 to 8), we apply this general framework and introduce further (and somewhat more involved) discretization techniques while presenting numerical results and applications. Here, we treat several important problems arising in pattern recognition and mathematical finance. Kernel-engineering technique are discussed and aim at formulating support vector machines in a way that makes it easy to adapt them to many problems of interest in engineering and industry.

The methodology applies to the discretization of partial differential operators and leads us to kernel-based building blocks (associated with any given support vector machine) which can be used for the approximation of solutions to partial differential problems, such those arising in fluid dynamics modeling. Importantly, to all of our numerical results we associate quantitative error bounds (or quality tests), which are of crucial importance in applications especially in mathematical finance.

We found it convenient to write this monograph by relying on a combination of Python code, R code, and Latex code. We produced a Jupyter notebook in which all of the numerical tests can be repeated and modified by the reader[1].

1.2 Outline of this monograph

- In Chapter 2 we provide the reader with a brief overview of techniques of machine learning and we introduce the notation and terminology we will use in this monograph while pointing

[1]The CodPy library is available for download.

out some other terminology also used within the machine learning community. Here, we thus discuss the notions relevant for

- the description of numerical algorithms of machine learning,
- the description of performance indicators (or error estimates) which provide a measure of the relevance of any given learning machine, and
- we briefly list the class of libraries currently available.

It is out of the scope of this monograph to cover all of the techniques that are currently available, and we restrict attention to a selection of kernel-based methods of machine learning and specific applications of central interest. By our own presentation of this material, we attempt here to provide a new insight on the subject.

Kernel-based projection operators and kernel-based clustering methods presented in Chapter 3 are novel algorithms that have no equivalent formulation in the existing algorithmic literature. We also advocate here the use of the notion of discrepancy error and kernel-based norms which lead us to performance indicators with good efficiency in the applications.

We provide criteria that can be used to evaluate the performance of an algorithm and do not depend on the specific method in use. This leads us to a suitable benchmark of existing methods. Many methods of machine learning have been proposed in the literature, and we advocate the use of the above indicator in order to systematically benchmark them. * To any given learning problem, represented by a list of input data,
* one should pick up several scenarios, several learning machines, and several performance indicators, and then * systematically run the corresponding tests in order to compare, from the output, the various performance indicators.

We consider here several learning machines, and compare with our strategy first with one- or two-dimensional examples, leading to relevant benchmarks for supervised learning and unsupervised learning. We then proceed with the study of problems of direct interest in the application.

The following topics are covered in this monograph.

- Chapter 2: Brief overview of methods of machine learning
- Chapter 3: Kernel methods for machine learning
- Chapter 4: Kernel methods for optimal transport
- Chapter 5: Application to supervised machine learning
- Chapter 6: Application to unsupervised machine learning
- Chapter 7: Application to optimal transport problems
- Chapter 8: Application to some partial differential equations

1.3 References

Our primarily intention in this monograph is to provide an introduction to our Python library, and only a brief bibliography is included here, while we refer to the research papers cited below for additional references. Indeed, a large literature is available on support vector machines and reproducing kernel Hilbert spaces (RKHS), it is not our purpose to review it here. The interested reader can refer to the textbooks by Berlinet and Thomas-Agnan [3] and Fasshauer [10],[11],[12] which were very influential in the development of the present code and the reader will find therein a background on the subject.

Our own contributions concerning the kernel-based meshfree algorithms presented in this monograph can be found in the research papers by LeFloch and Mercier [28],[29],[30],[31],[32]. Moreover, the unpublished notes [33]–[38] contain earlier versions of the material in this monograph. For additional material on meshfree methods and kernel-based methods and applications in fluid and material dynamics, see for instance [2],[4],[16],[18],[22],[39],[41],[43],[46],[48],[52],[59].

Chapter 2

Brief overview of methods of machine learning

2.1 A framework for machine learning

2.1.1 Prediction machine for supervised/unsupervised learning

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

$$f_z = \mathcal{P}_m(X, Y = [], Z = X, f(X)).$$

We use on a standard Python notation and the brackets above indicate that the variables Y, Z 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 non-optional parameter $X \in \mathbb{R}^{N_x \times D}$ is 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**, while the 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

$$f_Z = \mathcal{P}_m(X, Y = [], Z = X, f(X)) \sim f(Z),$$

[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

$$f_z = \mathcal{P}_m(X, Z = X), \tag{2.1.1}$$

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 being defined, 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 $N_z \times Df$

Moreover, from any machine learning method m we can also compute the gradient of a real valued function $f = f(x_1, \dots, x_D)$ by

$$(\nabla f)_Z = (\nabla_Z \mathcal{P}_m)(X, Y = [], Z = X, f(X) = []) \sim \nabla f(Z),$$

where $\nabla := (\partial_{x_1}, \dots, \partial_{x_D})$, then we say that m is a differentiable learning machine.

2.1.2 Techniques of supervised learning

Supervised learning (2.1.1) corresponds to the situation where the function values $f(X)$ is part of the input data:

$$f_z = \mathcal{P}_m(X, Y = [], Z = X, f(X)). \tag{2.1.2}$$

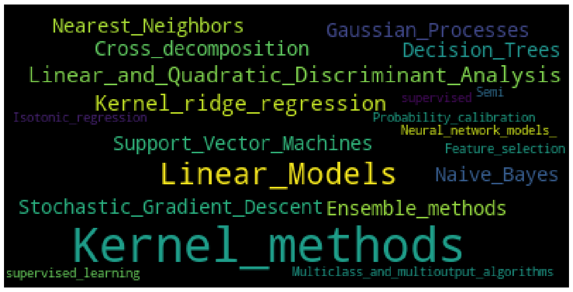
Supervised learning can be best understood as a simple extrapolation procedure: from historical observations of a given function $X, f(X)$, one wants to predict (or extrapolate) the function on a new set of values Z. Concerning the terminology, a method is said to be **multi-class** or multi-output if the function f under consideration can be vector-valued, that is, $D_f \geq 1$ in our notation. Observe that one can always combine learning machines in order to produce multi-class methods. However, this usually comes at a heavy computational cost, and this motivates our definition. Moreover, the input function f can be either:

- 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, or
- mixed (some data being discrete, some data being continuous).

A classification of existing methods of supervised learning can be found at the website https://scikit-learn.org

We distinguish between:

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



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

- Different computational libraries: scikit-learn, TensorFlow,...

2.1.3 Techniques of unsupervised learning

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

$$\mathcal{P}_m(X, Y=[], Z=X). \tag{2.1.3}$$

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 $Y \in \mathbb{R}^{N_y \times D}$.

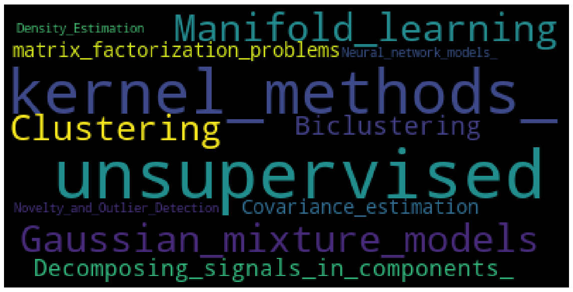
There are natural connections between supervised and unsupervised learning.

- 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 $[1,\dots,N_z] \mapsto [1,\dots,N_y]$.

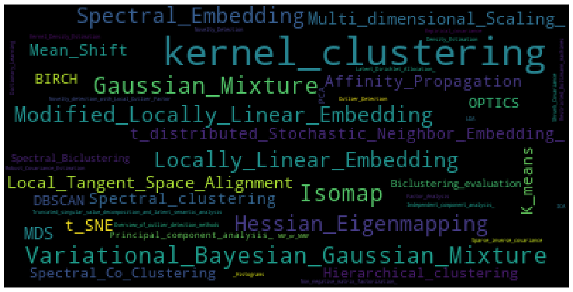
In the literature, many clustering methods are available for performing the task above; see for instance the dedicated Wikipedia page[2].

[2]link to cluster analysis Wikipedia page.

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



- Different particular methods are available: neural networks, Gaussian processes,...



- Different libraries are also available: Scikit-learn,...

Clustering represents one approach to unsupervised learning, and the library Scikit-learn does offer a quite impressive list of clustering methods; see [3]. In Figure 2.1 we provide some illustration:

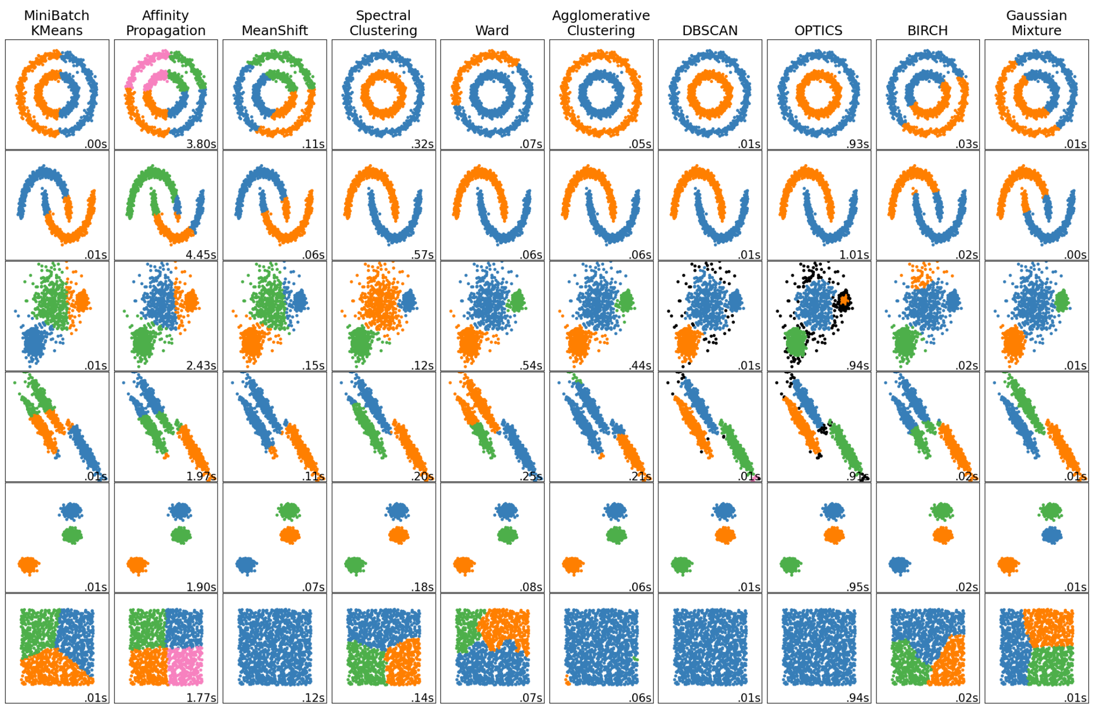


Figure 2.1: List of scikit-learn clustering methods.

[3]link to scikit-learn clustering

- Each column corresponds to a particular clustering algorithm.
- Each row corresponds to a particular clustering, unsupervised problem:
  - Each image scatter shows the training set X and the test set Z, which coincide here.
  - Each image color indicates the predicted values f_z.

2.2 Exploratory data analysis

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 multimodal, 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.

Non-parametric density estimations. The density of the input data is estimated using a kernel density estimate (KDE). Let $(x^1, x^2, \ldots, x^n)$ 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.

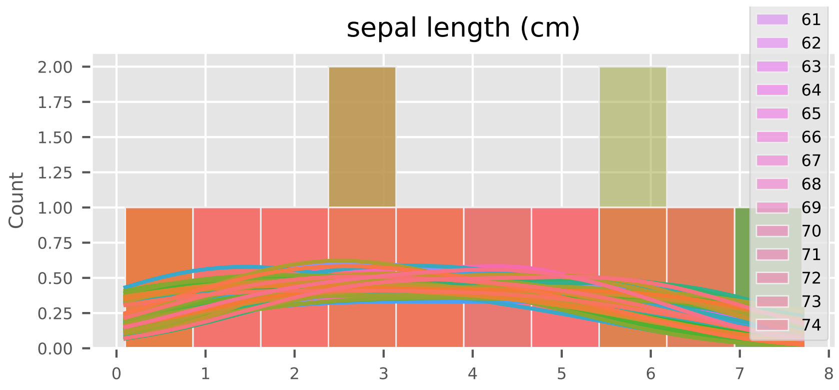


Figure 2.2: Kernel density estimator

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.

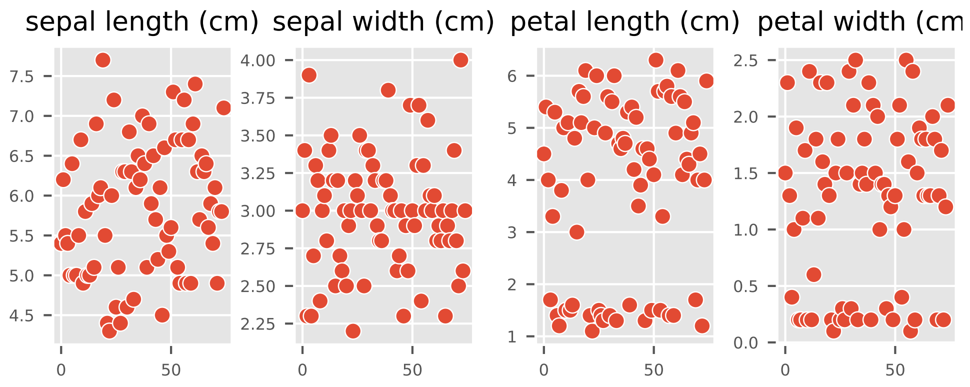


Figure 2.3: Scatter plot

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

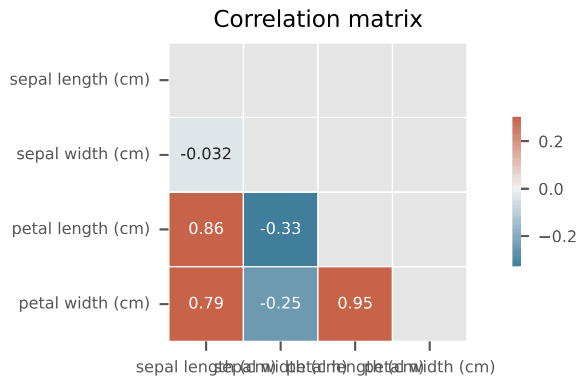


Figure 2.4: Correlation matrix

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 Distances and divergences

f-divergences. The notion of distance between probability distributions has many applications in mathematical statistics, information theory, such as hypothesis and distribution testing, density estimation, etc. One family of well-studied and understood family of distances/divergences between probability distributions are so-called $f-$divergences, we give a brief classification. Let $f : (0, \infty) \mapsto \mathbb{R}$ be a convex function with $f(1) = 0$. Let P and Q be two probability distributions on a discrete measurable space $(\mathcal{X}, \mathcal{F})$. If P is absolutely continuous with respect to Q, then f-divergence is defined as

$$D_f(P||Q) = \mathbb{E}^Q[f\left(\frac{dP}{dQ}\right)] = \sum_x Q(x) f\left(\frac{dP(x)}{dQ(x)}\right)$$

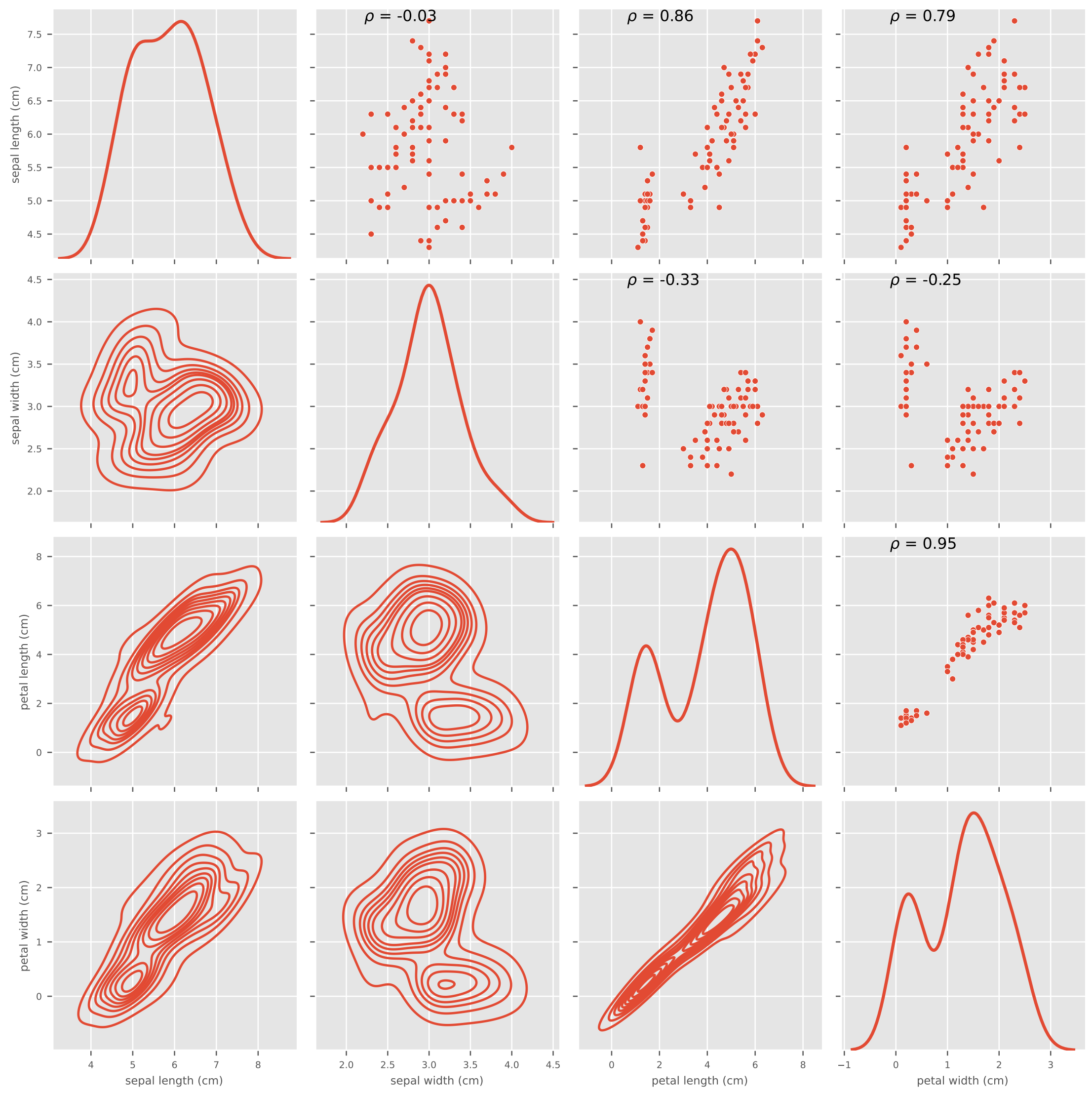


Figure 2.5: Summary plot

We list the following common $f-$divergences:

- **Kullback-Leibler (KL) divergence** with $f(x) = x\log(x)$.
- **Squared Hellinger Distance** with $f(x) = (1-\sqrt{x})^2$. Then the formula of Hellinger distance $\mathcal{H}^2(P,Q)$ is given by

$$\mathcal{H}(P,Q) = \frac{1}{\sqrt{2}}||\sqrt{dP} - \sqrt{dQ}||_2.$$

Maximum mean discrepancy. Another popular family of distances are **the integral probability metrics** (IPMs)[4], that includes Wasserstein or Kantorovich distance, total variation (TVD) or Kolmogorov distance and maximum mean discrepancy (MMD). MMD is defined in Section 3.2.6.

2.3.2 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[5].

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 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

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

  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

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

  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.

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

  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. See the dedicated page on scikit-learn.

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 the following sections. Its common form is

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

[4] A. Muller, "Integral probability metrics and their generating classes of functions", Advances in Applied Probability, vol. 29, pp. 429–443, 1997.

[5] link to scikit-learn metrics.

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 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 summarized up in the following table:

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

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

2.3.3 Indicators for unsupervised learning

Maximum mean discrepancy. Evaluation of clustering algorithms benefits from a lot of performance indicators, a lot of them being implemented in Scikit-learn:see this link.

As an alternative to standard unsupervised learning metrics, we propose to use MMD. It is used primarily to produce worst error estimates, together with the norm of functions, as described in 3.2.5, but 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* algorithm. 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 distances are considered, as the Manhattan or log-entropy, depending upon the problem under consideration. Consider any point $w \in \mathbb{R}^D$. Then w is attached naturally to a point $y^{\sigma(w,y)}$, where the index function $\sigma(w, y)$ is defined as

$$\sigma(w, Y) := \arg \inf_{j=1...N_Y} d(w, y^j).$$

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 the set of **centroids** for k-means algorithms.

2.4 General specification of tests

2.4.1 Preliminaries

We now overview a benchmark methodology and apply it to some supervised learning methods. 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 and restrict attention to toy examples while more practical cases will be studied in Chapter 5.

We begin by describing a general first quality assurance test for supervised learning machines. The goal of this framework is to measure accuracy of any machine learning models, 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 model, 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 five 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
2	2500	2500	2500
2	1600	1600	1600
2	900	900	900
2	400	400	400

For the function f we set a periodic and an increasing function:

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

2.4.2 Extrapolation in one dimension

Description. During this experiment, 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. A validation set Z is distributed over a larger cube, to observe extrapolation and interpolation effects.

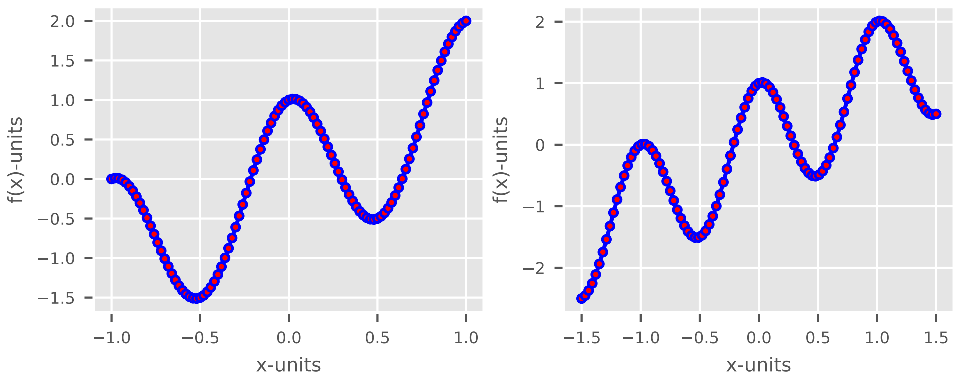


Figure 2.6: training and test set.

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

A comparison between methods. We compare codpy's periodic kernel with following machine learning models: scipy's RBF kernel regression, support vector regression (**SVR**), decision tree (**DT**), adaboost, random forest (**RF**) by scikit-learn library and TensorFlow's neural network (**NN**) model.

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 **??**. For the SVR we chose RBF kernel, for DT we set the maximum depth to 10, for the RF and XGBoost we set the number of estimators to 10 and 5 respectively and the maximum depth to 5. For the feed-forward NN we chose 50 epochs with batch size set to 16, we chose Adam optimization algorithm and mean squared error as the loss function. The NN is composed of two hidden layers (64 cells), one input (8 cells) and one output layers (1 cell) with the following sequence of activation functions: RELU - RELU - RELU - Linear. All other hyperparameters in the models are default set by scikit-learn, SciPy and TensorFlow.

Figure 2.7 visualizes extrapolation of each method. We note that a periodic kernel gives a better extrapolation between $[-1.5, -1]$ and $[1, 1.5]$, that is also confirmed in Figure 2.8 showing RMSE error for different sample size N_x.

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.

2.4.3 Extrapolation in two dimensions

Description. Now we show the fact that the dimension arising in the problem under consideration does not change benchmark methods. To illustrate this point, we simply repeat the previous steps used for the one-dimensional case, but the dimension is set to two, that is $D = 2$, and the reader can test with this parameter. Only data visualization changes.

The data is generated using five scenarios from Table 2.3, corresponding to a two dimensional case. Figure 2.9 shows both graphs $(X, f(X))$ (left, training set),$(Z, f(Z))$ (right, test set) for illustration purposes, f is a two-dimensional periodic function 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 by either setting indices $\tilde{X} := X[index1, index2]$ or performing a PCA over x and setting $\tilde{X} := PCA(X)[index1, index2]$.

A comparison between methods. We compare two models for function's extrapolation:

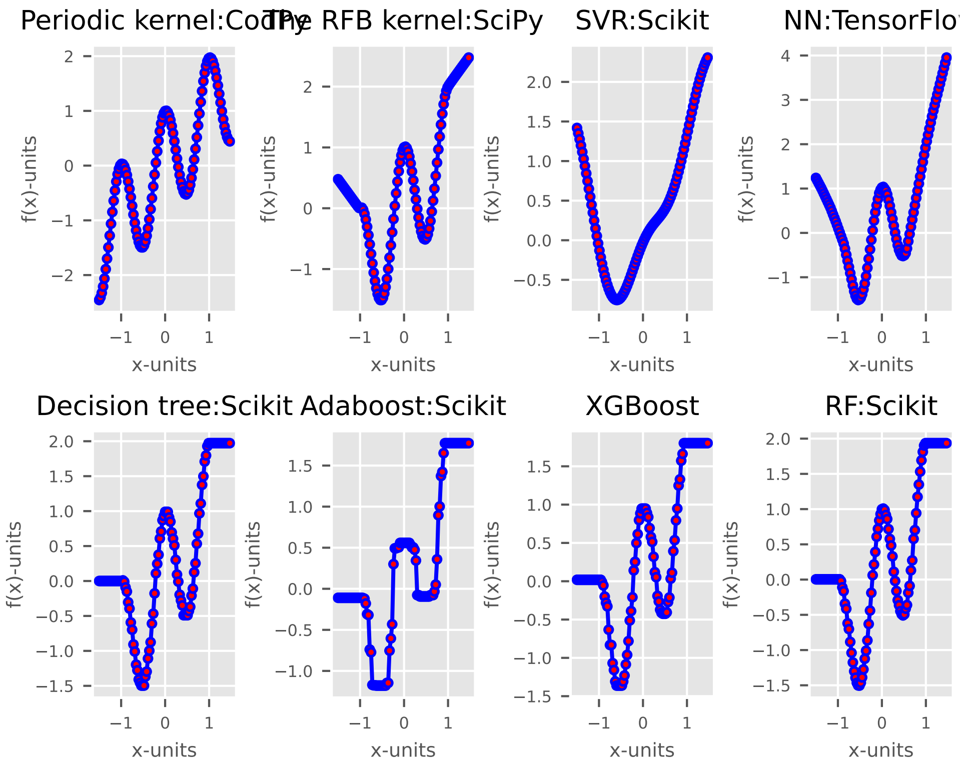


Figure 2.7: Periodic kernel:CodPy, the RBF kernel: SciPy, SVR: Scikit, Neural Network: TensorFlow, Decision tree: Scikit, Adaboost: Scikit, XGBoost, Random Forest: Scikit

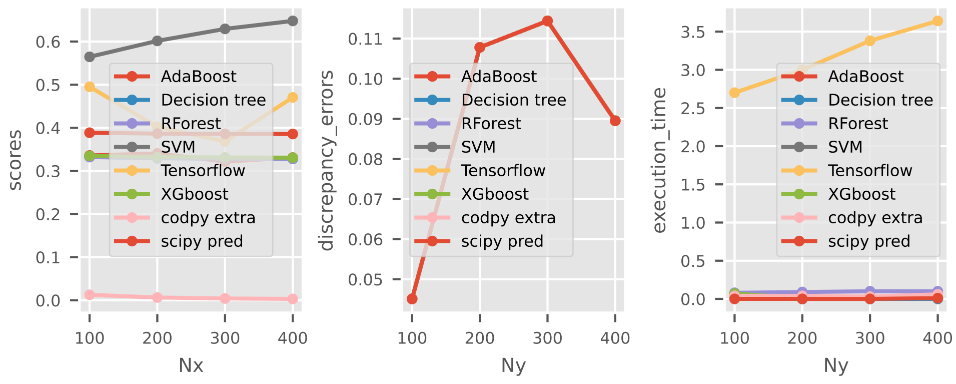


Figure 2.8: RMSE, MMD and execution time

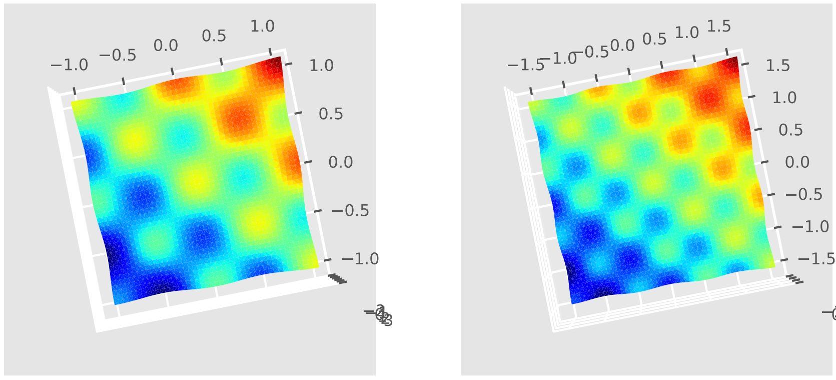


Figure 2.9: Train vs test set.

codpy's periodic Gaussian kernel with SciPy's RBF kernel.

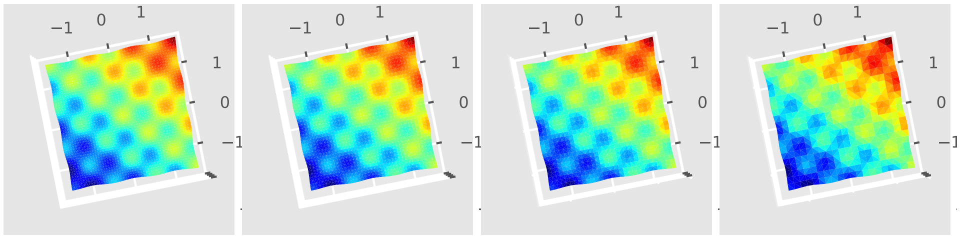


Figure 2.10: RBF (the first and the second), a periodic Gaussian kernel (the third and the forth)

The first two graphs in Figure 2.10 shows RBF's predictions for first two scenarios defined in Table 2.3, and the last two graphs for a periodic Gaussian kernel.

2.4.4 Clustering

Description. 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 multimodal and multivariate Gaussian distribution with a covariance matrix $\Sigma = \sigma I_d$. The problem is to identify the modes of the distribution using a clustering method. In the following we will generate distribution with a predetermined number of modes, it will allow to test validation scores on this toy example.

A comparison between methods. We compute and compare codpy's clustering MMD minimization with Scikit's implementation of k-means algorithm. During this experiment we generate distributions with different number of modes (between 2 and 6).

The two first two graphs in Figure 2.13 correspond to the computed clusters using k-means algorithm and the last two graphs correspond to the MMD minimization.

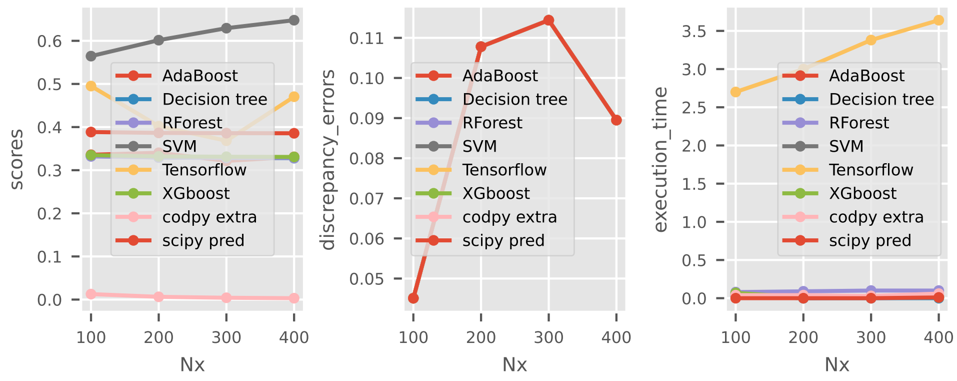


Figure 2.11: A periodic Gaussian kernel vs RBF kernel

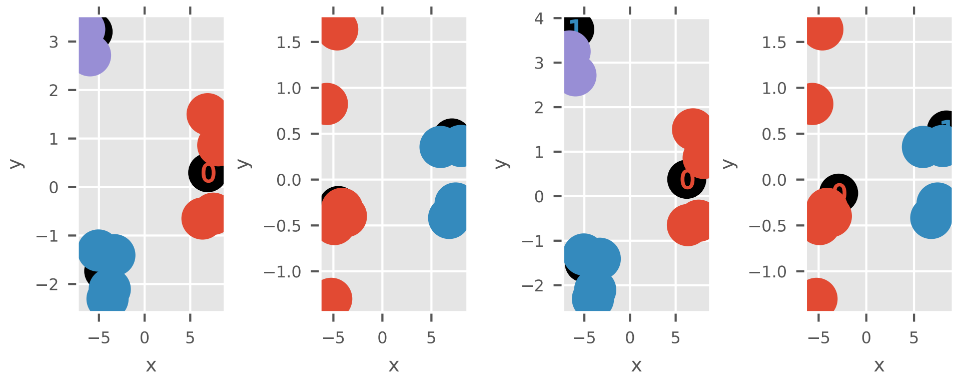


Figure 2.12: Scatter plots of k-means and MMD minimization algorithms

Figure 2.13 illustrates four confusion matrices, the first row corresponds to k-means algorithm and the second to the MMD minimization.

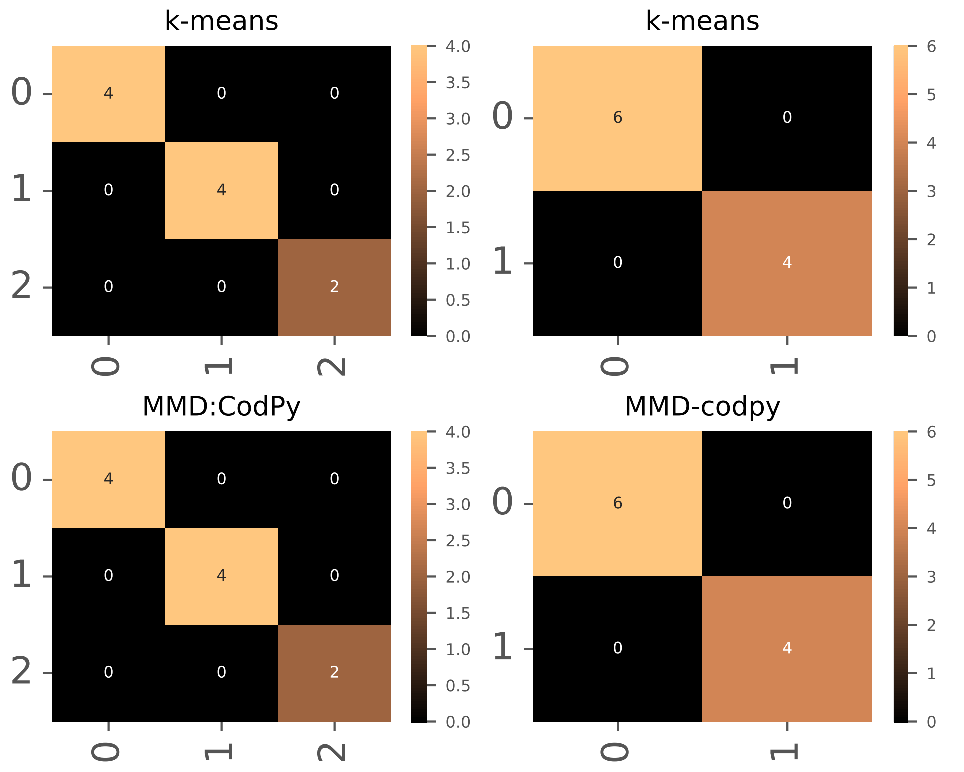


Figure 2.13: Confusion matrices of k-means and MMD minimization algorithms

We compare various methods under consideration, by means of performance indicators, as illustrated by Figure 2.14. In order to avoid confusion of defining best possible clustering, we chose inertia as the metric to evaluate the performance of algorithms. MMD error just indicates the fact that two samples are the same, they coincide at the different levels of sample size. Table 2.6 in the appendix to the chapter resumes the experiment.

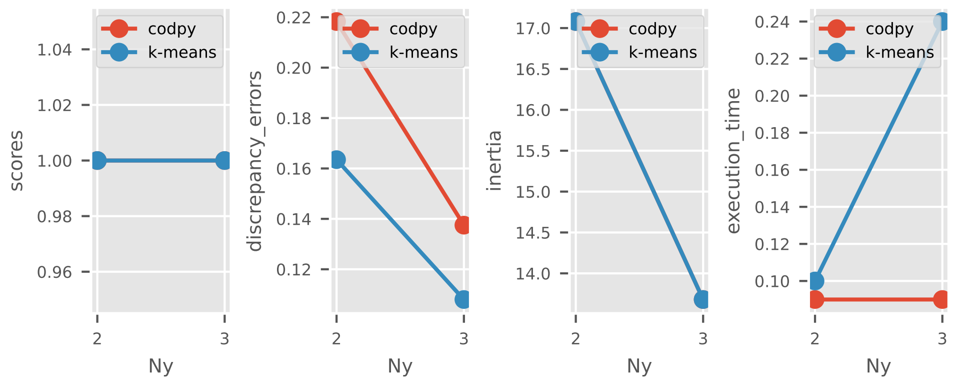


Figure 2.14: benchmark of various performance indicators for clustering.

2.5 Bibliography

XGBoost[6] is a computationally efficient implementation of the original gradient boost algorithm. Standard libraries for neural networks are TensorFlow[7] and Pytorch[8]. Scikit-learn library[9] offers a comprehensive set of models (linear, SVMs, stochastic gradient and feature selection methods). Recently TensorFlow added "TensorFlow probability[10]" library that contains modules on linear algebra, statistics, positive-definite kernels and Gaussian process methods, Monte Carlo and etc.

2.6 Appendix to chapter 2

Results of 1D extrapolation. Table 2.4 illustrates the performance of supervised machine learning models to extrapolate the values of a periodic function define in Section 2.4.1. We compare the performance using four measures: execution time, scores, the norm of the function to be predicted and MMD errors.

Table 2.4: Supervised algorithms performance indicators

predictors	D	N_x	N_y	N_z	D_f	time	RMSE	MMD
codpy extra	1	100	100	100	1	0.04	0.0127	0.0451
codpy extra	1	200	200	200	1	0.04	0.0064	0.1078
codpy extra	1	300	300	300	1	0.03	0.0042	0.1144
codpy extra	1	400	400	400	1	0.06	0.0032	0.0895
scipy pred	1	100	100	100	1	0.00	0.3885	0.0451
scipy pred	1	200	200	200	1	0.00	0.3865	0.1078
scipy pred	1	300	300	300	1	0.00	0.3859	0.1144
scipy pred	1	400	400	400	1	0.01	0.3856	0.0895
SVM	1	100	100	100	1	0.03	0.5645	0.0451
SVM	1	200	200	200	1	0.00	0.6015	0.1078
SVM	1	300	300	300	1	0.01	0.6293	0.1144
SVM	1	400	400	400	1	0.01	0.6478	0.0895
Tensorflow	1	100	100	100	1	2.70	0.4951	0.0451
Tensorflow	1	200	200	200	1	3.00	0.4021	0.1078
Tensorflow	1	300	300	300	1	3.38	0.3683	0.1144
Tensorflow	1	400	400	400	1	3.64	0.4704	0.0895
Decision tree	1	100	100	100	1	0.03	0.3326	0.0451
Decision tree	1	200	200	200	1	0.00	0.3294	0.1078
Decision tree	1	300	300	300	1	0.00	0.3285	0.1144
Decision tree	1	400	400	400	1	0.00	0.3280	0.0895
AdaBoost	1	100	100	100	1	0.05	0.3358	0.0451
AdaBoost	1	200	200	200	1	0.01	0.3404	0.1078
AdaBoost	1	300	300	300	1	0.03	0.3216	0.1144
AdaBoost	1	400	400	400	1	0.03	0.3309	0.0895
XGboost	1	100	100	100	1	0.07	0.3349	0.0451
XGboost	1	200	200	200	1	0.01	0.3320	0.1078
XGboost	1	300	300	300	1	0.01	0.3312	0.1144
XGboost	1	400	400	400	1	0.01	0.3307	0.0895
RForest	1	100	100	100	1	0.08	0.3321	0.0451
RForest	1	200	200	200	1	0.09	0.3297	0.1078

[6] see this dedicated page for a description of XGBoost project
[7] see this dedicated page for a description of TensorFlow neural networks
[8] see this dedicated page for a description of Pytorch's neural networks
[9] see this dedicated page for a description of Scikit's library
[10] see this dedicated page for a description of TensorFlow probability's library

Table 2.4: Supervised algorithms performance indicators *(continued)*

predictors	D	N_x	N_y	N_z	D_f	time	RMSE	MMD
RForest	1	300	300	300	1	0.10	0.3287	0.1144
RForest	1	400	400	400	1	0.10	0.3283	0.0895

Results of 2D extrapolation. Table 2.5 shows the computed indicators after running all scenarios indicated in Table 2.6.

Table 2.5: Supervised algorithms performance indicators

predictors	D	N_x	N_y	N_z	D_f	time	RMSE	MMD
codpy extra	2	2601	2500	2500	1	3.22	0.0000	0.0600
codpy extra	2	1681	1600	1600	1	1.05	0.0000	0.0849
codpy extra	2	961	900	900	1	0.26	0.0000	0.1416
codpy extra	2	441	400	400	1	0.08	0.0000	0.1817
scipy pred	2	2601	2500	2500	1	0.32	0.2009	0.0600
scipy pred	2	1681	1600	1600	1	0.12	0.2036	0.0849
scipy pred	2	961	900	900	1	0.04	0.2083	0.1416
scipy pred	2	441	400	400	1	0.01	0.2182	0.1817

Results of clustering methods. Table 2.6 represents the results obtained during clustering experiments, the performance is measured using four indicators: execution time, scores, MMD and inertia.

Table 2.6: Unsupervised algorithms performance indicators (Clustering)

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD	inertia
k-means	2	10	3	10	1	0.24	1	0.1080	13.68
k-means	2	10	2	10	1	0.10	1	0.1635	17.08
codpy	2	10	3	10	1	0.09	1	0.1375	13.68
codpy	2	10	2	10	1	0.09	1	0.2183	17.08

Chapter 3

Reproducing-kernel methods for machine learning

3.1 Purpose of this chapter

We begin with the description ofreproducing-kernel methods which are adapted to problems arising in the theory of machine learning. We discuss two of the main ingredients required in the design of our CodPy algorithms.

- First of all, our algorithms depend on the choice of a reproducing-kernel space and on the use of transformation maps which we apply to collection of basic kernels in order to adapt them to any particular problem.
- Second, we also define the kernel-based notions of (mesh-free) discrete differential operators which are relevant for machine learning.

The notions presented here provide us with the key building blocks for our basic algorithms of machine learning, but also to deal with problems involving partial differential operators. In the following chapters of this monograph we will next presentmore advanced algorithms and various applications.

For a description of the framework of interest we need some notation. A set of N_x variables in D dimensions is given, denoted by $X \in \mathbb{R}^{N_x \times D}$, together with a D_f-dimensional vector-valued data function $f(X) \in \mathbb{R}^{N_x \times D_f}$ which represents the *training values* associated with the *training set* X. The input data therefore consists of

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

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

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

Note in passing that all of the examples and numerical experiments given below will be based on the following choice of function consisting of the sum of a periodic function and an increasing function (in each direction):

$$f(x) = f(x_1, \dots, x_D) = \prod_{d=1,\dots,D} \cos(4\pi x_d) + \sum_{d=1,\dots,D} x_d, \qquad x \in \mathbb{R}^D. \tag{3.1.2}$$

Table 3.1: choice of dimension for data extrapolation

D	N_x	N_y	N_z
2	529	529	529

At this stage, we do not explain all of our notation but provide some numerical illustration only, concerning the prediction (Z, f_Z) from the training set $(X, f(X))$. In fact, in addition to our notation $X \in \mathbb{R}^{N_x \times D}$ and $Z \in \mathbb{R}^{N_z \times D}$ together with $f(X) \in \mathbb{R}^{N_x \times D_f}$ and $f(Z) \in \mathbb{R}^{N_z \times D_f}$, we also introduce an extra variable denoted by Y below and we distinguish between several cases. The choice $N_x = N_z$ will correspond to a *data extrapolation*, as will be explained in Section 3.2.4 below. On the other hand, a choice $N_y << N_x$ will correspond to a *data projection*, as will be also explained below.

Table 3.2: A choice of dimensions for data projection

D	N_x	N_y	N_z
2	529	18	529

Hence, Figure 3.1 provides us with a typical illustration of results of machine learning, and we will particularly on the choice made in the first test throughout the following discussion. The left-hand plots show the (variables, value) training set $(X, f(X))$, while the right-hand plot displays the (variables, values) test set $(Z, f(Z))$. The plots in the middle display the (variables, values) parameter set $(Y, f(Y))$, whose importance will be explained later on: in short, the choice of Y closely determines not only the overall accuracy of the algorithm, but also its computational cost. Having provided a broad illustration we can now proceed and define all relevant concept in full details.

3.2 Fundamental notions for supervised learning

3.2.1 Preliminaries

Positive kernels and kernel matrices. Let $k = k(x,y) : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ be a symmetric real-valued function, that is, a function satisfying $k(x,y) = k(y,x)$. Given two collection $X = (x^1, \cdots, x^{N_x})$ and $Y = (y^1, \cdots, y^{N_y})$ of points in $\mathbb{R}^D$, we define the associated *kernel matrix* $K(X,Y) := (k(x^n, y^m)) \in \mathbb{R}^{N_x \times N_y}$ by

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

We say that k is a positive kernel if, for any collection of distinct points $X \in \mathbb{R}^{N_x \times D}$ and for any (non-identically vanishing) $c^1, ..., c^{N_x} \in \mathbb{R}^{N_x}$,

$$\sum_{1 \leq i,j \leq N_x} c^i c^j k(x^i, x^j) > 0. \tag{3.2.2}$$

When $N_x = N_y$, the square matrix $K(X,Y)$ is called a Gram matrix. The dimension of the null space of the matrix $K(X,Y)$ is usually $N_x \times N_y$, except typically for some kernels, as can be checked in the section on kernel engineering (Section 3.5).

If $K(X,Y)$ is positive only on a certain sub-manifold of $\mathbb{R}^D$, we say that the kernel is *conditionally positive*, that is, the ositivity condition golds only if X, Y belongs to this sub-manifold.

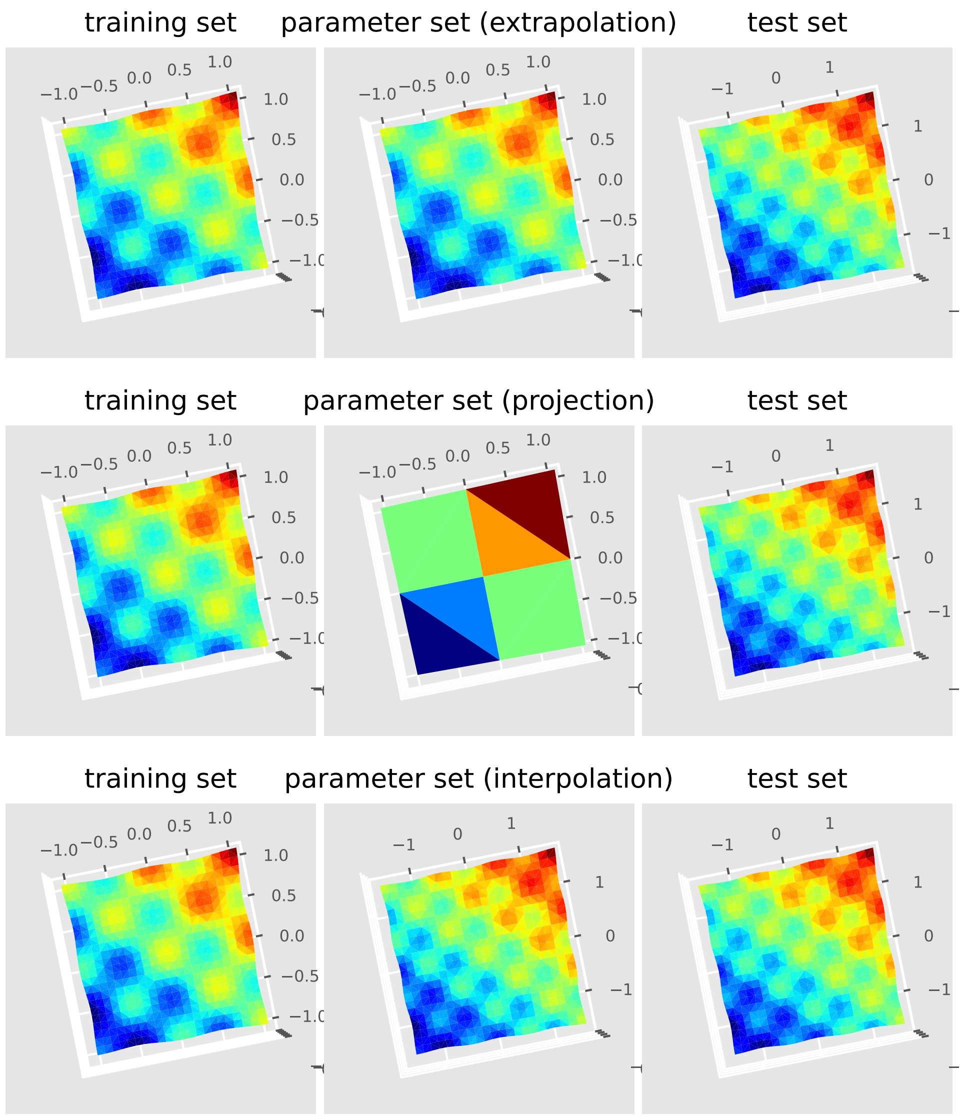


Figure 3.1: Examples of (training, parameter, test) sets for three different choice of Y

We always use positive or conditionally positive kernels. The available kernels in our library are listed in the table below.

	Kernel name	$k(x, y)
1.	Dot product	$k(x, y) = x^T y$
2.	RELU	$k(x, y) = \max(x - y, 0)$
3.	Gaussian	$k(x, y) = \exp(-\pi\|x - y\|^2)$
4.	Periodic Gaussian	$\sigma^2 \exp(-\frac{2}{l^2} \sin^2(\pi \frac{\|x-y\|}{p})$
5.	Matern norm	
6.	Matern tensor	
7.	Matern periodic	
8.	Multiquadric norm	
9.	Multiquadric tensor	
10.	Sincard square tensor	
11.	Sincard tensor	
12.	Tensor norm	
13.	Truncated norm	
14.	Truncated periodic	

Here, for the Gaussian kernel, l denotes the length scale, p the period, and σ the mplitude. A scaling of the basic kernels may be required in order to handle input data, which is exactly the purpose of the maps, discussed below.

Example 3.2.1. *Gaussian kernel reads (and is used by default in the CodPy library)*

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

Example 3.2.2. *A mapping $S : \mathbb{R}^D \mapsto \mathbb{R}^P$ begin given, consider the family of kernels*

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

where g is called an activation function. In particular, the scalar products between the collections of successive powers of x_d and y_d, denoted by

$$k(x, y) =< (1, x, x^T x, ...), (1, y, y^T y, ...) >$$

defines a kernel corresponding to a linear regression over a polynomial basis. It is positive, but the null space of the associated matrix kernel is non-empty. The so-called RELU kernel given by

$$k(x, y) = \max(< x, y > +c, 0)$$

(c being a constant) is a conditionally positive kernel

Let us consider the kernel to be *tensornorm* (discussed below) and let us refer to Section 2 for the description of the relevant parameters in our algorithm. We then display some typical values of the kernel matrix, in Table 3.4,, which are computed using our function *op.Knm* in CodPy.

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

1.003472	1.005208	1.006944	1.008681
1.005208	1.008681	1.012153	1.015625
1.006944	1.012153	1.017361	1.022569
1.008681	1.015625	1.022569	1.029514

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

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

When $X \neq Y$, this inverse is computed using a least-square approach, namely

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

The so-called Tikhonov parameter ϵ represents a regularization parameter (and, by default in CodPy, takes the value $\epsilon = 10^{-8}$).

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

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

0.0004324	0.0004139	0.0003954	0.0003768
0.0004139	0.0003967	0.0003795	0.0003623
0.0003954	0.0003795	0.0003636	0.0003478
0.0003768	0.0003623	0.0003478	0.0003332

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

- If $N_x \neq N_y$.
- If the Tikhonov regularization parameter $\epsilon > 0$ is non-zero. While the user can choose $\epsilon = 0$, some performance issues may arise. Namely, if the kernel is not (unconditionally) positive, then the CodPy library might raise an "exception'', and will switch from the standard inversion of matrices to an adapted inversion (of non-invertible matrices) which 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 3.5). In which case this matrix is a projection over the null space of the matrix $K(X,X)$.

Distance matrices. The notion of distance matrix defined now is a simple and very convenient tool within the framework of kernel methods. To any positive kernel $k : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ we can associated the distance function $d_k(x,y)$ for all $x \in \mathbb{R}^D$ and $y \in \mathbb{R}^D$, defined by

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

For a positive kernel, this expression is continuous, non-negative, and satisfies $d(x,x) = 0$.

For any collections $X = (x^1, ..., x^{N_x})$ and $Y = (y^1, ..., y^{N_y})$ of points in $\mathbb{R}^D$ we define a the associated *distance matrix* $D(X,Y) \in \mathbb{R}^{N_x \times N_y}$ by

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

Table 3.6 displays the first four columns of the kernel-based distance distance matrix $D(X,Y)$, and obviously the diagonal of this matrix vanishes.

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

0.0000000	0.1900826	0.3966942	0.6198347
0.1900826	0.0000000	0.1900826	0.3966942
0.3966942	0.1900826	0.0000000	0.1900826
0.6198347	0.3966942	0.1900826	0.0000000

3.2.2 Methodology of the CodPy algorithms

Our algorithms provide one with general functions in order to make predictions in (3.1.1) from the choice of a kernel. More precisely, the following operator (with $A^{-1} := (A^T A)^{-1} A^T$ 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:

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

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

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 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 **??** 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 refer to this step as **learning process**, as this step is exactly the counterpart of the weight set for the neural network approach. This construction amounts to define a feed-backward machine, analogous to (3.2.6), as

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

Observe that (3.2.6) allows us also to compute the following operation, where $\nabla := (\partial_1, \dots, \partial_D)$ 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 Section 3.4, 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) and **differential learning machine** methods.

3.2.3 Transportation maps

We will use surjective maps $S : \mathbb{R}^T \mapsto \mathbb{R}^D$, referred to as *transportation maps*, and we can distinguish between several types:

- **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 following table.

	Maps	S(X)
1.	Affine	$S(X) =$
2.	Exponential	$S(X) = e^X$
3.	Identity	$S(X) = X$
4.	Log	$S(X) = \log(X)$
5.	Map to grid	$S(X)$
6.	Scale to standard deviation	$S(X) = \frac{x}{\sigma}$, $\sigma = \sqrt{\frac{1}{N_x}\sum_{n<N_x}(x^n - \mu)}$, $\mu = \frac{1}{N_x}\sum_{n<N_x} x^n$
7.	Scale to erf	$S(X) = erf(x)$, erf is the standard error function.
8.	Scale to erfinv	$S(X) = erf^{-1}(x)$, erf^{-1} is the inverse of erf.
9.	Scale to mean distance	$S(X) = \frac{x}{\sqrt{\alpha}}$, $\alpha = \sum_{i,k\leq N_x} \frac{\lvert x^i - x^k\rvert^2}{N_x^2}$.
10.	Scale to min distance	$S(X) = \frac{x}{\sqrt{\alpha}}$, $\alpha = \frac{1}{N_x}\sum_{i\leq N_x} \min_{k\neq i} \lvert x^i - x^k\rvert^2$.
11.	Scale to unit cube	$S(X) = \frac{x - \min_n x^n + \frac{0.5}{N_x}}{\alpha}$, $\alpha := \max_n x^n - \min_n x^n$.

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 Scale to min distance is usually a good choice: this map scales all points to the average min distance. 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.

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

3.2.4 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 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 (3.2.4). 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.

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

3.2.5 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

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

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,

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

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

$$\|f\|_{\mathcal{H}_k}^2 \sim f(X)^T K(X, X)^{-1} f(X)$$

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.

3.2.6 Error estimates based on the generalized maximum mean discrepancy

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

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

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

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

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 functional**, known as the maximum mean discrepancy (MMD), first introduced in [14]. 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*. Section 3.6.5 contains plots and some analysis of this functional. In this book we call generalized MMD and discrepancy error equivalently.

3.3 Dealing with kernels

3.3.1 Maps and kernels

Maps can ruin your prediction. We now 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 $(Z, f_z) \in \mathbb{R}^{N_z\times D}\times\mathbb{R}^{N_z\times D_f}$. In Figure 3.2. we set a different map, called mean distance map, which scales all points to the average distance for a Gaussian kernel. This example illustrates how maps can drastically influence the computation. However, the very same map can be appropriate for other kernels; see Figure 3.2.

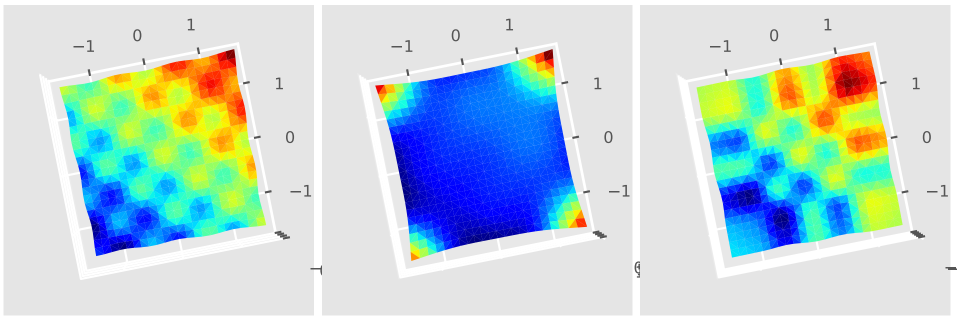


Figure 3.2: A ground truth value (first), Gaussian (second) and Matern kernels (third) with mean distance map

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:

$$\begin{aligned} map\_setters.set\_min\_distance\_map(^{**}kwargs) \\ pipe\_map\_setters.pipe\_erfinv\_map() \\ pipe\_map\_setters.pipe\_unitcube\_map() \end{aligned}$$

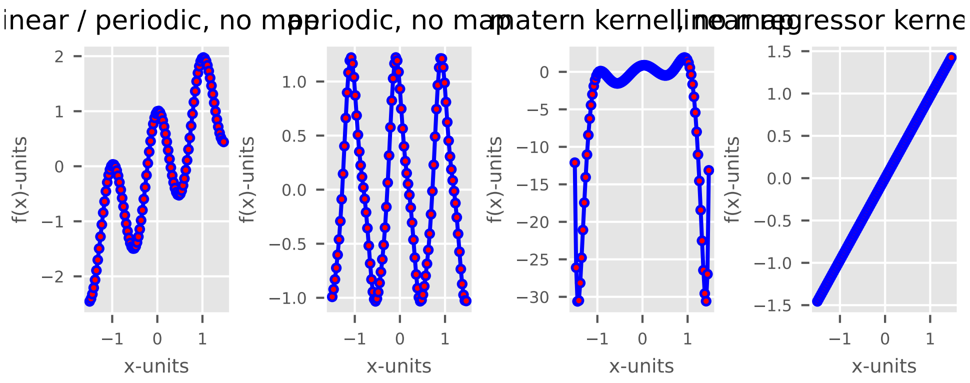
For any $X \in \mathbb{R}^{N_x\times D}$, this composite map performs the following operations: first, rescale all data to the unit cube using scale to unit cube map; second, apply the map defined as $S(X) = erf^{-1}(2X-1)$ and finally use the average min distance map.

3.3.2 Illustration of different kernels predictions

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

- In most situations, a kernel is defined by
  - a positive definite kernel function,
  - a map.
- The choice of the inner parameters set Y. We usually face three main choices here.
  - $Y = X$, that corresponds to the *extrapolation* case; cf. Section 3.2.4. 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 Section **??**. 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.

In order to illustrate the effects of different kernels and maps on learning machines, we compare predictions for the one-dimensional test described for different kernels.



3.4 Discrete differential operators

3.4.1 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:

$$f_Z := K(Z,Y)c_Y, \quad c_Y := K(X,Y)^{-1} f(X) \in \mathbb{R}^{N_Y \times D_f}. \tag{3.4.1}$$

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

$$f_Z := \sum_{n=1}^{N_Y} c_y^n K(Z, y^n), \tag{3.4.2}$$

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

3.4.2 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:

$$\phi : Y \mapsto \left(\phi^1(Y), \dots, \phi^{N_x}(Y)\right) = K(Y,X)K(X,X)^{-1} \in \mathbb{R}^{N_y \times N_x}. \tag{3.4.3}$$

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

$$\phi(x^n) := \Big(0,\dots,1,\dots,0\Big) = \delta_{n,m}, \tag{3.4.4}$$

where $\delta_{n,m} = 1$ if $n = m$, 0 else. For this reason, we call $\phi(x)$ a partition of unity. Figure 3.3 illustrates partitions functions.

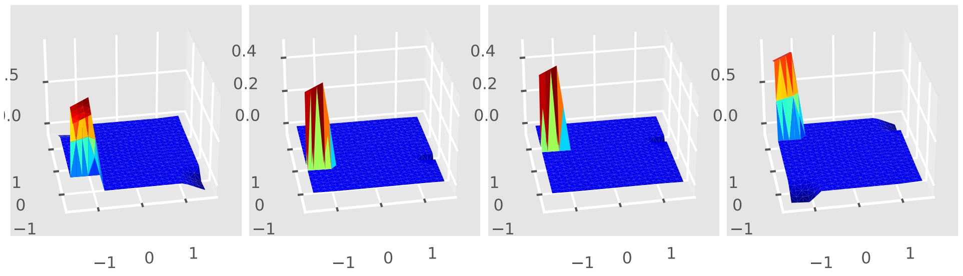


Figure 3.3: Four partitions of unity functions

3.4.3 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 Section 3.2.3, modify the operator ∇_k as follows:

$$\nabla_{k\circ S}(X,Y,Z) := (\nabla S)(Z)\Big(\nabla_1 k)(S(Z),S(Y)\Big)K(S(X),S(Y))^{-1}, \tag{3.4.5}$$

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. Figure 3.4 illustrate a corresponding derivative and compare it to the original one for the first and second dimensions respectively.

3.4.4 Divergence operator

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

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

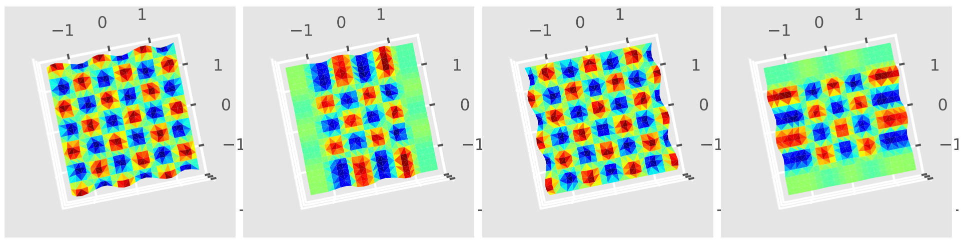


Figure 3.4: The first two graphs correspond to the first dimension (original on the left-hand, computed on the right-hand). The next two graphs correspond to the second dimension (original on the left-hand, computed on the right-hand).

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

$$\int (\nabla f) \cdot g d\mu = - \int f \nabla \cdot (g d\mu). \tag{3.4.6}$$

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

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

To compute this operator, we proceed as follows: starting from the definition of the gradient operator (3.4.5), 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 by

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

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. Figure 3.5 compares the outer product of the gradient to Laplace operator $\nabla_k(X,Y,Z)^T \nabla_k(X,Y,Z) f(X)$ to $\Delta_k(X,Y) f(X)$; see the next section.

3.4.5 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 (3.4.6).

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

3.4.6 Inverse Laplace operator

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

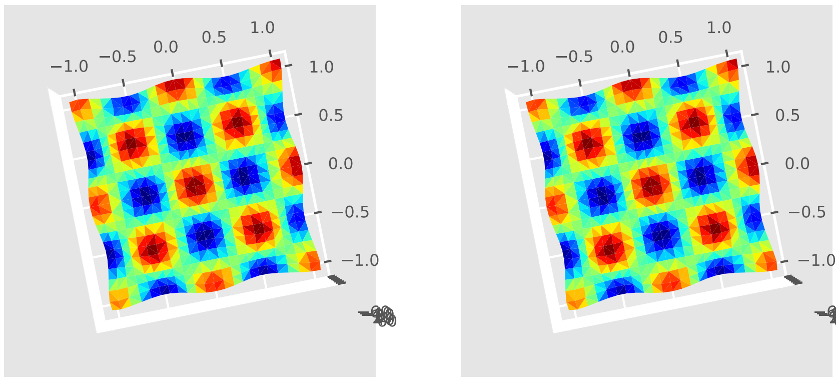


Figure 3.5: Comparison of the outer product of the gradient to Laplace operator

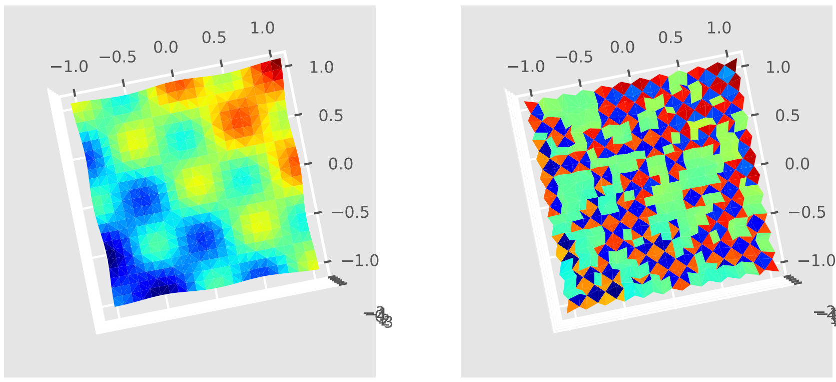


Figure 3.6: Comparison between original function to the product of Laplace and its inverse

A two-dimensional example. Figure 3.6 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 $\Delta_{k,x,y,z}\Delta_{k,x,y,z}^{-1}f(X)$ in Figure 3.7, to check that pseudo-inversion commutes, as highlighted by the following computations:

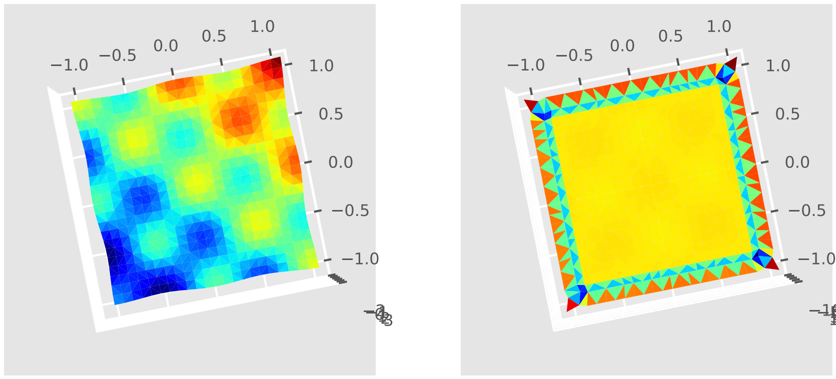


Figure 3.7: Comparison between original function and the product of the inverse of the Laplace operator and the Laplace operator

3.4.7 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:

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

A two-dimensional example. In Figure 3.8 we show that $(\nabla)(\nabla)^{-1}f(X)$ coincides with $f(X)$. Observe however that this latter operation is not equivalent to Figure 3.9, which uses Z as extrapolation set.

3.4.8 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 3.4.6.

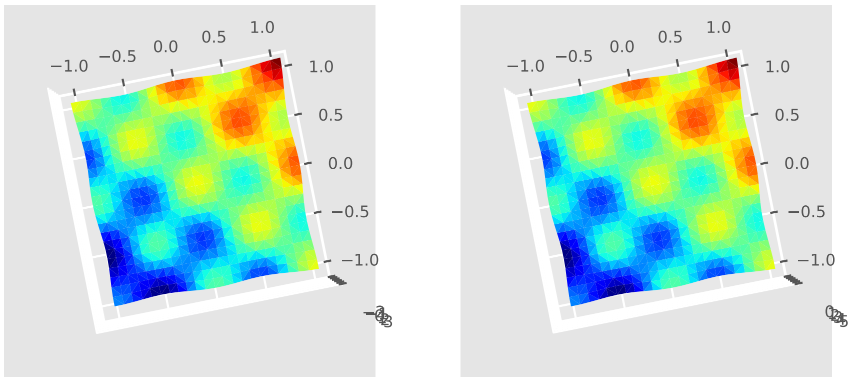


Figure 3.8: Comparison between original function to the product of the gradient operator and its inverse

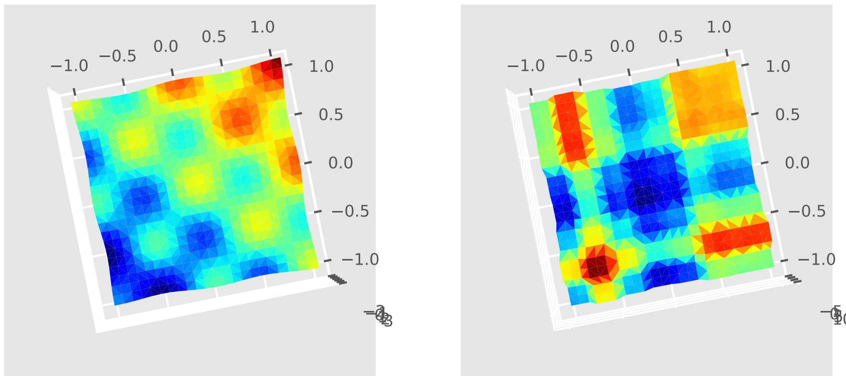


Figure 3.9: Comparison between original function to the product of the inverse of the gradient operator and the gradient operator

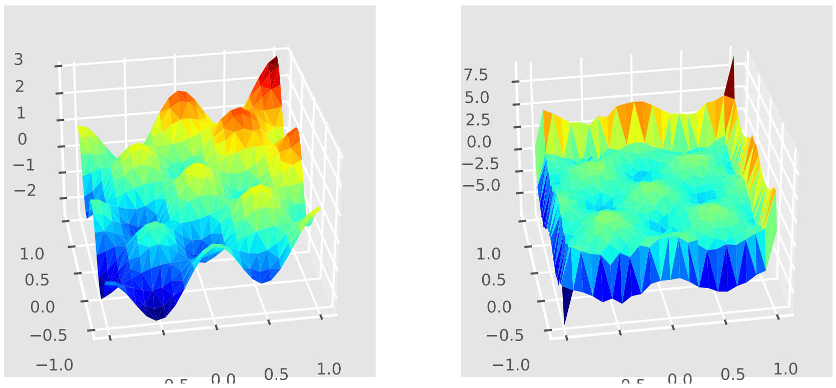


Figure 3.10: Comparison between the product of the divergence operator and its inverse and the product of Laplace operator and its inverse

3.4.9 Leray-orthogonal operator

We define the Leray-orthogonal operator as

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

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 Figure 3.11, we compare this operator to the original function $(\nabla f)(Z)$:

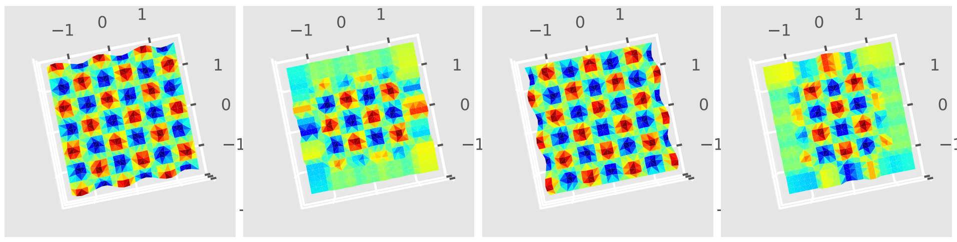


Figure 3.11: Comparing f(z) and the transpose of the Leray operator on each direction

3.4.10 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:

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

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 Figure 3.12 we compare this operator to the original function $(\nabla f)(Z)$.

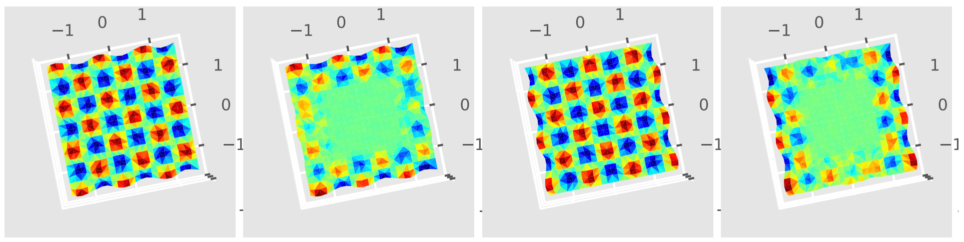


Figure 3.12: Comparing f(z) and the Leray operator in each direction

3.5 Kernel engineering

3.5.1 Manipulating kernels

In this section we describe some general operations on kernels, which allow us to generate new and relevant kernels. These operations preserve a positiveness 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:

$$\mathcal{P}_{k_i}(X,Y,Z) := K_i(Z,Y)K_i(X,Y)^{-1} \in \mathbb{R}^{N_z \times N_x}, i = 1,2 \tag{3.5.1}$$

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.

3.5.2 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 $\mathcal{P}_k(X,Y,Z) \in \mathbb{R}^{N_z \times N_y}$:

$$K(X,Y) := K_1(X,Y) + K_2(X,Y), \quad \mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,Y)^{-1}. \tag{3.5.2}$$

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

$$\mathcal{H}_k = \Big\{ \sum_{1 \le m \le N_x} a^m (k_1(\cdot, x^m) + k_2(\cdot, x^m)) \Big\}. \tag{3.5.3}$$

3.5.3 Multiplicating 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

$$K(X,Y) := K_1(X,Y) \circ K_2(X,Y), \quad \mathcal{P}_k(X,Y,Z) = K(Z,X)K(X,Y)^{-1}, \tag{3.5.4}$$

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

$$\mathcal{H}_k = \Big\{ \sum_{1 \le m \le N_x} a^m (k_1(\cdot, x^m) k_2(\cdot, x^m)) \Big\}. \tag{3.5.5}$$

3.5.4 Convoluting 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

$$K(X,Y) := K_1(X,Y)K_2(Y,Y), \tag{3.5.6}$$

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

$$\mathcal{H}_k = \Big\{ \sum_{1\le m\le N_x} a^m(k(\cdot,x^m)) \Big\}, \tag{3.5.7}$$

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

3.5.5 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 (3.2.4) as follows:

$$\mathcal{P}_k(X,Y,Z) = \mathcal{P}_{k_1}(X,Y,Z)\pi_1(X,Y) + \mathcal{P}_{k_2}(X,Y,Z)\Big(I_d - \pi_1(X,Y)\Big), \tag{3.5.8}$$

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 (3.5.8) is, after (3.2.10),

$$\mathcal{H}^x_k = \Big\{ \sum_{1\le m\le N_x} a^m k_1(\cdot,x^m) + \sum_{1\le m\le N_x} b^m k_2(\cdot,x^m) \Big\}. \tag{3.5.9}$$

Hence, this doubles up the coefficients (3.4.1). We define its inverse concatenating matrix

$$K^{-1}(X,Y) = \Big(K_1(X,Y)^{-1}, K_2(X,Y)^{-1}(I_{N_x} - \pi_1(X,Y))\Big) \in \mathbb{R}^{2N_y \times N_x}. \tag{3.5.10}$$

The kernel matrix associated to a “piped” kernel pair is then

$$K(X,Y) = \Big(K_1(X,Y), K_2(X,Y)\Big) \in \mathbb{R}^{N_x \times 2N_y}. \tag{3.5.11}$$

3.5.6 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. $S(x) = \Big(\varphi_1(x), \dots, \varphi_N(x)\Big)$ and consider the following kernel, called dot product kernel (which is conditionally positive-definite):

$$k_1(x,y) :=< S(x), S(y) > . \tag{3.5.12}$$

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 $S(x) = (1, x_1, \dots)$: it consists of a classical polynomial regression, allowing to perfectly match moments of distributions, since the remaining error is handled by the second kernel.

3.5.7 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 $z_M(y) = y_M k(y_{M-1}, \dots, y_0)$.

3.6 A first application: a clustering algorithm

3.6.1 Distance-based unsupervised learning machines

In this section we describe a kernel-based clustering algorithm. This algorithm, already presented with a toy example in Section 2.4.4, is also benchmarked in a forthcoming section devoted to more concrete problems; see Chapter **??**.

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

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

Suppose that this last problem is well-posed, assuming that the distance functional is convex[1]. Once the cluster set $Y := \Big(y^1, \dots, y^{N_y}\Big)$ is computed, then one can define the index function $\sigma(w, Y) := \arg \inf_{j=1\dots N_Y}\{d(w, y^j)\}$, as for (2.3.3). One we can extend naturally this function, defining a map

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

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 $\sigma(Z, Y)$ is surjective (many-to-one). Hence we can define its injective (one-to-many) inverse, $\sigma(Z, Y)^{-1}(n)$, describing those points of the test set attached to one y^n. This construction defines cells, very similarly to quantization, $C^n := \sigma(\mathbb{R}^D, y^n)^{-1}(n)$, 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 $\sigma(X, Y) \in [1, \dots, N_x]^{N_y}$ 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 a distance-minimization unsupervised algorithm naturally defines a supervised one.

[1]although most of existing distance are not convex

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 MMD or the discrepancy functional (3.2.6). This procedure is separated into two main steps.
  - First solve a discrete version of (3.6.1), namely

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

    where Σ denotes the set of all subsets from $[1, \dots, N_y] \mapsto [1, \dots, N_x]$, and d_k is the discrepancy functional. This minimization problem is described Chapter **??**.
  - 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 (3.2.4), 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

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

  provides an approximation to 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 (4.1), 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

Figure 2.4.4 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 Section 2.4.4, re-running all tests with scenarios for N_y covering [0,100]. Figure 3.13 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.

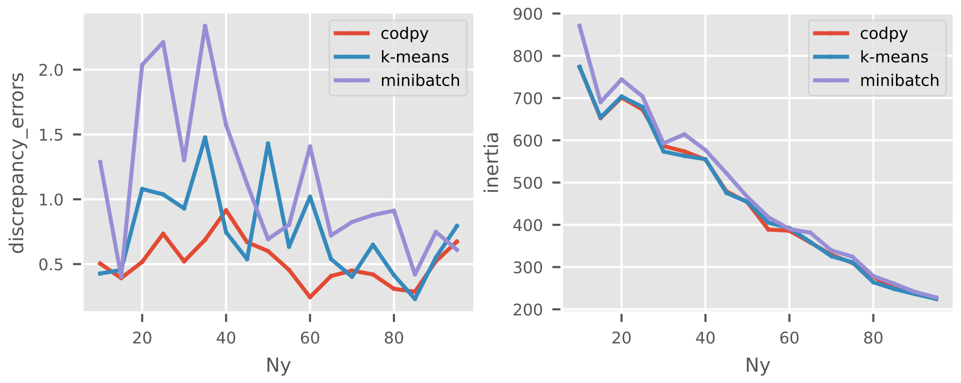


Figure 3.13: benchmark of discrepancy errors and inertia

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 $Y = (y^1, y^2) \in \mathbb{R}^{N_y \times 2}$.

An 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.

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

The following picture shows the function $y \mapsto d_k(x,y)$ in blue. We also display the function $d_k(x,x^n)$, $n = 1 \dots N_x$ in Figure 3.14 in orderto illustrate that this functional is neither convex nor concave.

We see that the functional $y \mapsto d_k(x,y)$ admits a minimum which is close to $y = \frac{1}{2}$, as expected. Figure 3.15 displays $y := (y^1, y^2) \mapsto d_k(x,y)$. We see that this functional admits two minima, and 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 consider a kernel which generate a functional space with less regularity. RELU kernels is the following family of kernels (essentially generating the space of functions with bounded variation):

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

As is clear in Figure 3.14 the function $y \mapsto d_k(x, y)$ is only piecewise differentiable. Hence in some cases, the functional $d_k(x, y)$ might have an infinity of solutions (here on the "flat" segment). Figure 3.15 displays $y := (y^1, y^2) \mapsto d_k(x, y)$ for a two-dimensional example

An example of continuous kernel: Matern. The Matern kernel reads (and generates a space of continuous functions)

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

In Figure 3.14 we see that the function $y \mapsto d_k(x, y)$ admits concave parts, and a gradient-descent algorithm can not give satisfactory results. Figure 3.15 shows a two-dimensional example $y := (y^1, y^2) \mapsto d_k(x, y)$

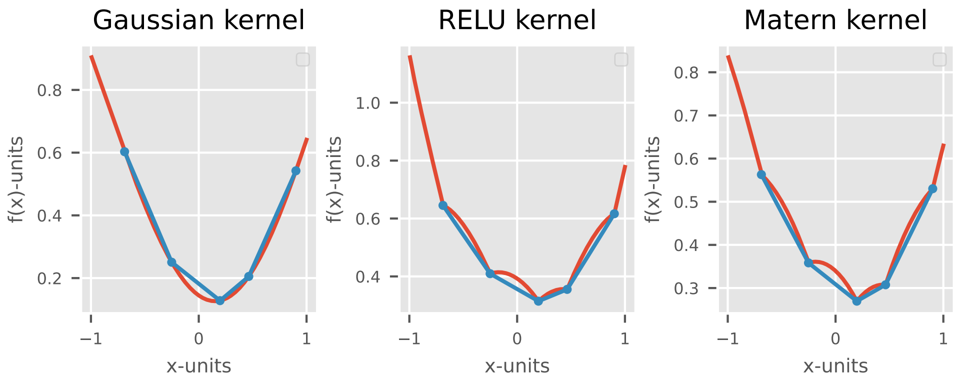


Figure 3.14: Distance functional for the Gaussian, the Matern and the RELU kernels (1D)

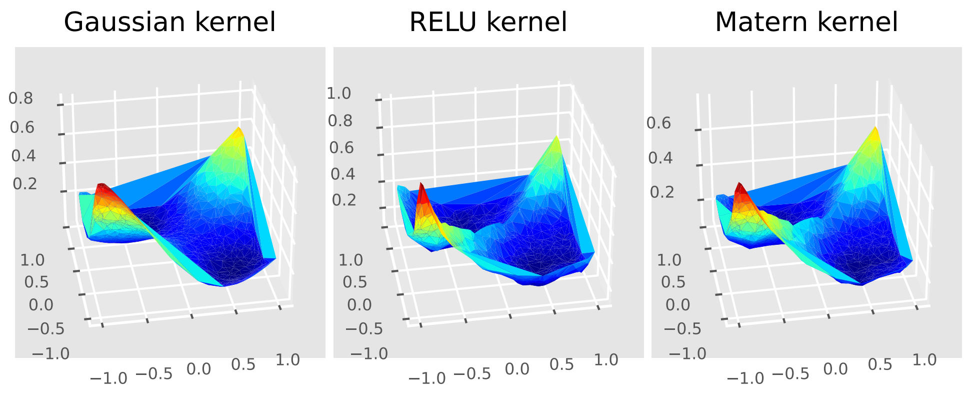


Figure 3.15: Distance functional for the Gaussian, the Matern and the RELU kernels (3D)

3.7 Bibliography

There esxists a vast literature on RKHS methods (cf. the list of refetences at the end of this monograph) and, in particular on kernel regressions. A good reference is the textbook "Elements of Statistical Learning Data Mining, Inference, and Prediction" by R.Tibshirani et al. RKHS methods in statistics and related fields are described in the textbook "Reproducing Kernel Hilbert Spaces in Probability and Statistics" by C. Thomas-Agnan et al. This also studied in "A Hilbert Space Embedding for Distributions" by A. Smola, A. Gretton et al. A dimension reduction technique for kernel least-square models was introduced in "Kernel Partial Least Squares Regression in Reproducing Kernel Hilbert Space" by R. Rosipal and L. Trejo.

3.8 Appendix to Chapter 3

3.8.1 Maps and kernels

In the table below we propose a list of kernels with maps that best fit to each kernel

	Kernels	Maps
1.	Dot Product	
2.	RELU	
3.	Gaussian	Affine map, mean distance map,
4.	Periodic Gaussian	Affine map
5.	Matern norm	
6.	Matern tensor	
7.	Matern periodic	
8.	Multiquadric norm	
9.	Multiquadric tensor	
10.	Sincard square tensor	
11.	sincard tensor	
12.	Tensor norm	Unit cube map
13.	Truncated norm	
14.	Truncated periodic	

Chapter 4

Kernel methods for optimal transport

4.1 A brief overview of discrete optimal transport

In this short introduction we recall basic facts concerning optimal transport, by focusing on the discrete cases discussed in [6], and we explain describe how we connect these standard tools to our error-based learning machines. For a complete review of optimal transport theory, see [56].

Consider a probability measure $\nu \in \mathcal{M}$ on $\mathbb{R}^D$, and a mapping $S : \mathbb{R}^D \mapsto \mathbb{R}^D$. Denote by $\mu \in \mathcal{M}$ the measure defined by the change of variable

$$\int_{\mathbb{R}^D} \varphi(x) d\mu = \int_{\mathbb{R}^D} (\varphi \circ S)(x) d\nu, \qquad \varphi \in \mathcal{C}(\mathbb{R}^D).$$

One then says that S transports ν into μ, and $S_{\#}\nu = \mu$ is referred to as the push-forward. Consider a cost function, that is a positive, scalar-valued, symmetric, $\mathcal{C}^1$-regular function $c = c(\cdot, \cdot)$. The *Monge problem*, given ν, μ, consists in finding a mapping S minimizing the transportation cost from ν to μ, that is

$$\overline{S} = \arg \inf_{S_{\#}\nu = \mu} \int_{\mathbb{R}^D} c(x, S(x)) d\nu.$$

In a discrete point of view, we consider two discrete measures $\mu, \nu = \delta_X, \delta_Z$, in which $X = (x^1, \dots, x^N)$ and $Z = (z^1, \dots, z^N)$ are two sequences of distinct points with the same length. Then the Monge problem (4.1) amounts to determine a permutation $\overline{\sigma} : [1 \dots N] \mapsto [1 \dots N]$ satisfying

$$\overline{S}(x^n) = z^{\overline{\sigma}(n)} \quad \text{with} \quad \overline{\sigma} = \arg \inf_{\sigma \in \Sigma} \sum_{n=1}^{N} c(x^n, z^{\sigma(n)}).$$

Here, Σ is the set of all permutations, and we simply write $\overline{S}(X) = Z^{\overline{\sigma}}$. Consider the matrix $C(X, Z) = \Big(c(x^i, z^j)\Big)_{i=1\dots N}^{j=1\dots N}$. Then the following problem is called the discrete *Kantorovitch problem*

$$\bar{\gamma} = \arg \inf_{\gamma \in \Gamma} C(X, Z) \cdot \gamma,$$

where $A \cdot B$ denotes the Frobenius scalar matrix product, Γ is the set of all bi-stochastic matrix $\gamma \in \mathbb{R}^{N \times N}$, i.e. satisfying $\sum_{n=1}^N \gamma_{m,n} = \sum_{n=1}^N \gamma_{n,m} = 1$ and $\gamma_{n,m} \geq 0$ for all $m = 1, \dots, N$. The minimization problem (4.1) admits a dual expression, called the dual-Kantorovich problem:

$$\overline{\varphi}, \overline{\psi} = \arg \sup_{\varphi, \psi} \sum_{n=1}^{N} \varphi(x^n) - \psi(z^n), \qquad \varphi(x^n) - \psi(z^m) \leq c(x^n, z^m),$$

where $\varphi : X \mapsto \mathbb{R}, \psi : Z \mapsto \mathbb{R}$ are discrete functions. As stated in [6], the three discrete problems above are equivalent. We observe that the discrete Monge problem (4.1) is also known as the **linear sum assignment problem (LSAP)**, and was solved in the 50's by an algorithm due to H.W. Kuhn; it is also known as the Hungarian method[1].

For the continuous case, under suitable conditions on ν, μ (namely, with compact, connected, and smooth support), any transport map $S_{\#}\nu = \mu$ can be *polar-factorized* as

$$S(X) = \overline{S} \circ T(X), \qquad T_{\#}\nu = \nu,$$

where $\overline{S}$ is the unique solution to the Monge problem (4.1), and is the gradient of a $c-$convex potential $\overline{S}(X) = \exp_x\big(- \nabla h(X)\big)$. Here, $\exp_x$ is the standard notation for the exponential map (used in Riemannian geometry). A scalar function is said to be c-convex if $h^{cc} = h$, where $h^c(Z) = \inf_x c(X,Z) - h(X)$ is called the infimal $c-$convolution. Standard convexity coincides with c-convexity for convex cost functions such as the Euclidean function, in which case the following polar factorization holds: $S(X) = (\nabla h) \circ T(X)$ with a convex h. These results go back to [7] (convex distance case) and [25] (general Riemannian distance) in the continuous setting.

We now describe the main connection between these techniques and learning machines 2.4.1. Indeed, consider the cost function defined as $c(X,Z) = d_K(\delta_X, \delta_Z)$, where the discrepancy functional d_k is described in 3.2.6. Consider, as above, two discrete measures $\mu, \nu = \delta_X, \delta_Z$, defining the map $S(x^n) = z^n$. With this notation, finding the map T appearing in the right-hand side of the polar factorization (4.1) consists in finding the permutation

$$\overline{\sigma} = \arg \inf_{\sigma \in \Sigma} \sum_{n=1}^{N} d_K(\delta_{x^n}, \delta_{z^{\sigma(n)}}).$$

Then, considering a differential learning machine 2.4.1, a discrete polar factorization consists in solving the following equation for the unknown potential h

$$Z^{\overline{\sigma}} = \exp_X \Big(- \nabla_X \mathcal{P}_m(X, Y, X, h(X))\Big).$$

Such algorithms can be implemented for any differential, error-based learning machines.

4.2 Linear Sum Assignment Problems (LSAP)

LSAP. The "linear assignment value" problem is a fundamental combinatorial optimization problem, which is used in a number of academic and industrial applications. It is an old and well-documented problem [2].

An illustration of the LSAP problem. Let $A \in \mathbb{R}^{N,M}$ be any, real-valued matrix. A standard way to describe the LSAP problem is to find a permutation $\sigma : [0,..,min(N,M)] \mapsto [0,..,min(N,M)]$ s.t.

$$\sigma = \arg \inf_{\sigma \in \Sigma} Tr(A^\sigma), \quad A^\sigma := A(\sigma(n), m)_{n,m}$$

where Σ is the set of all permutations.

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 $Tr(M)$.

[1]this algorithm seems nowadays credited to a 1890 posthumous paper by Jacobi.
[2]see the Wikipedia page https://en.wikipedia.org/wiki/Assignment_problem

Table 4.1: a 4x4 random matrix

0.2617057	0.2469788	0.9062546	0.2495462
0.2719497	0.7593983	0.4497398	0.7767106
0.0653662	0.4875712	0.0336136	0.0626532
0.9064375	0.1392454	0.5324207	0.4110956

Table 4.2: Total cost before permutation

1.465813

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

Table 4.3: Permutation

1	3	2	0

We use this permutation for the row of the matrix $M^\sigma := M[\sigma]$, and we output the new cost after ordering, that is $Tr(M^\sigma)$. we check in the following that the LSAP algorithm decreased the total cost.

Table 4.4: Total cost after ordering

0.6943549

An illustration of a discrepancy based reordering algorithm. 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:

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

This python function takes as input the following:

- Two distributions of points having shapes
$$X := (x^1, \dots, x^{N_x}) \in \mathbb{R}^{N_x \times D}, \quad Y := (y^1, \dots, y^{N_y}) \in \mathbb{R}^{N_y \times D}$$
- 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 2.4.1.
- 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 Chebyshev 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 symmetric, letting the original distribution X unchanged.

Table 4.7: Cost

393.3725

Table 4.9: Permutation

2	3	1	0

- A permutation σ, represented as a vector $i \mapsto \sigma_i$, $0 \leq i \leq \min(N_x, N_y)$.

A quantitative illustration. We show first the results given by our ordering algorithm on a simple example. We generate two random variables $X \in \mathbb{R}^{4\times5}$, $Y \in \mathbb{R}^{4\times5}$, such that $X \sim \mathcal{N}(\mu, I_5)$ and $Y \sim Unif([0,1]^{4\times5})$ with $\mu = [5, ..., 5]$. The first is generated by multivariate Gaussian distribution centered at μ, the second one by a uniform distribution supported into the unit cube.

Table 4.5 shows the distance matrix D_k induced by the Matern kernel k, and the transportation cost is the trace of the matrix, i.e. $Trace(D_k)$.

Table 4.5: Distance matrix before ordering

98.23496	107.96167	97.30143	102.42566
84.92364	93.94385	84.94698	90.99258
95.84254	106.71865	96.46384	101.69583
101.26279	104.96670	98.23994	104.72984

Table 4.6: Permutation before ordering

1	3	2	0

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

Finally, we output the distance matrix again after ordering in Table 4.8, as well as the permutation σ in Table 4.9

Table 4.8: Distance matrix after ordering

95.84254	106.71865	96.46384	101.69583
101.26279	104.96670	98.23994	104.72984
84.92364	93.94385	84.94698	90.99258
98.23496	107.96167	97.30143	102.42566

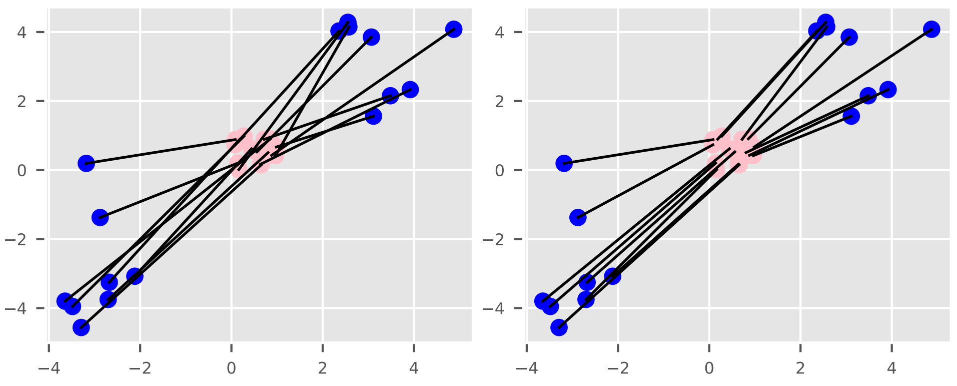
One can check that the sum of the diagonal elements, i.e. the total cost has decreased.

A qualitative illustration. This algorithm can be best illustrated in the two-dimensional case. First we consider a Euclidean distance function $d(x, y) = |x - y|_2$, 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 $X \in \mathbb{R}^{N_x\times D}$ and a random uniform one $Y \in [0,1]^{N_y\times D}$.

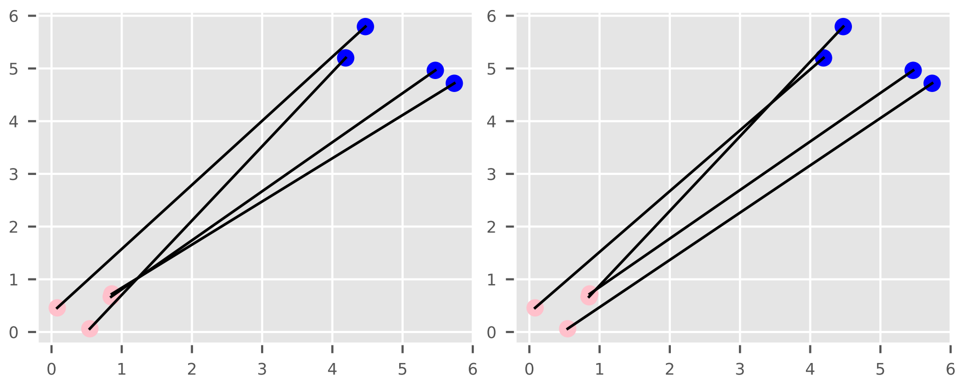
Table 4.10: Total cost after ordering

388.1819

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

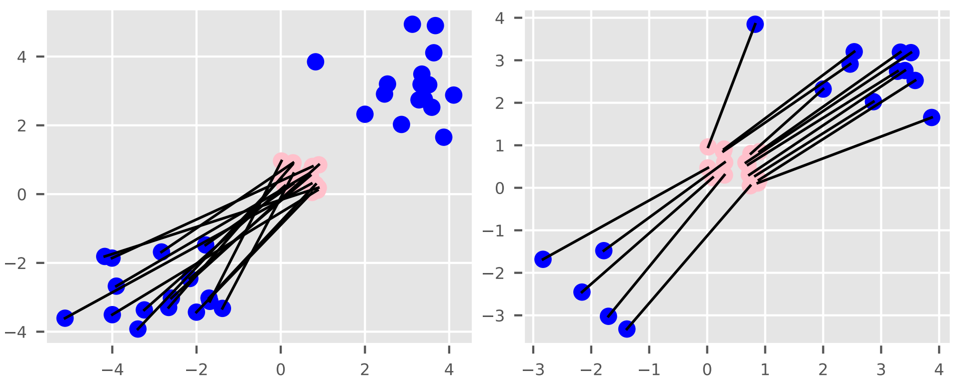


Note however that kernels based distance might lead to different permutations. This is due to the fact that kernels defines distance that might not be Euclidean. Indeed, kernel distance might not respect the triangular inequality. For instance, the kernel selected above defines a distance equivalent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



4.2.1 LSAP extensions

Different input sizes. Next we describe some extensions of the LSAP algorithms that we use in our library. A first quite straightforward extension of the LSAP problem can be found for inputs set of different sizes, without loss of generality $N_y \leq N_x$. Figure **??** illustrates the behavior of our LSAP algorithm in this setting



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 3.6.2, 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

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

4.3 Conditional expectation algorithm

Motivation. Kernel methods to compute conditional expectations were considered a decade ago, see for instance [27]. Indeed, these algorithms are central, in particular, for financial applications, as they are at the heart of pricing technologies. They also have numerous other applications. In this subsection, we propose a general python interface to a function computing conditional expectations problems in arbitrary dimensions, that we named Pi. We also propose a kernel-based implementation of these problems, which is described in [29] - [31].

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 machine learning methods, particularly in the mathematical finance applications, see [17] and ref. therein for instance. In particular, a neural networks approach has been proposed to compute conditional expectation in [20] that we can use as benchmark. Hence a first benchmark is conducted in section 7.5.

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 -

$$f_{Z|X} = \Pi(X, Z, f(Z))$$

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.

The output is a matrix $f_{Z|X}$, representing the conditional expectation

$$f_{Z|X} \sim \mathbb{E}^{X(t^2)}(f(\cdot)|X(t^1)) \in \mathbb{R}^{N_x \times D_f} =:^{not.} f(Z|X). \tag{4.3.1}$$

- 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 $\mathbb{E}^{X(t^1)}(Z|X)$.
- 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 $f(Z|X) := \mathbb{E}^{X(t^1)}(f(Z)|X)$.

4.4 The sampler function and discrete polar factorization

Sampler function. In this paragraph, we illustrate the polar factorization (4.1) through a quite simple algorithm, 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 $X \in \mathbb{R}^{N_x \times D}$, representing i.i.d. samples of a random variable X, $Z \in \mathbb{R}^{[0,1]^{N_z \times D}}$, any i.i.d. of the uniform distribution into the unit cube, and suppose that we solved (4.1) in the following, discrete, sense

$$X = \Big(\nabla f\Big)(Z), \quad f \text{ convex}, \quad X \in \mathbb{R}^{N_x \times D}, Z \in [0,1]^{N_z \times D}.$$

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

$$Y = sampler(X, N_y, seed)$$

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

4.4.1 Examples

One dimensional distributions. Consider two one-dimensional distributions : a bi-modal Gaussian and bi-modal Student's $t-$distribution. The experiment consists of comparing the truth distribution $X \in \mathbb{R}^{1000 \times 1}$ and a computed distribution $Y \in \mathbb{R}^{1000 \times 1}$ using a a sampling function.

Figure 4.1 compares kernel desnity estimates and histogram of the original sample and the distribution generated using a sampling function, the first plot compares to a Gaussian and second to $t-$distribution.

Tables 4.12 and **??** in Appendix show that sampling algorithm generated samples that are very close in skewness, kurtosis and in terms of KL divergence and MMD.

Two dimensional distributions. Next we simply repeat the experiment for a two-dimensional case. Figure 4.2 compares the distributions of $X \in \mathbb{R}^{1000 \times 2}$ and $Y \in \mathbb{R}^{1000 \times 2}$ (original and the computed distribution), the first scatter plot compares to a Gaussian, second to $t-$distribution and third and forth scatters plots are bimodal Gaussian and $t-$distribution respectively with $N_x = N_y = 1000$.

Tables 4.12 and **??** in Appendix to this chapter output third and forth moments of the truth and sampled distributions. On the one hand, the sampling algorithm can not capture the forth moment

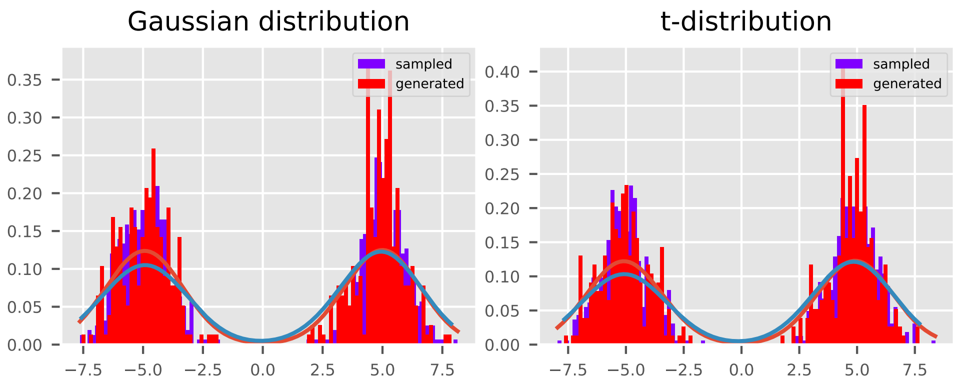


Figure 4.1: Histograms of Bi-modal Gaussian vs sampled (left) and Student's t distribution vs sampled (right)

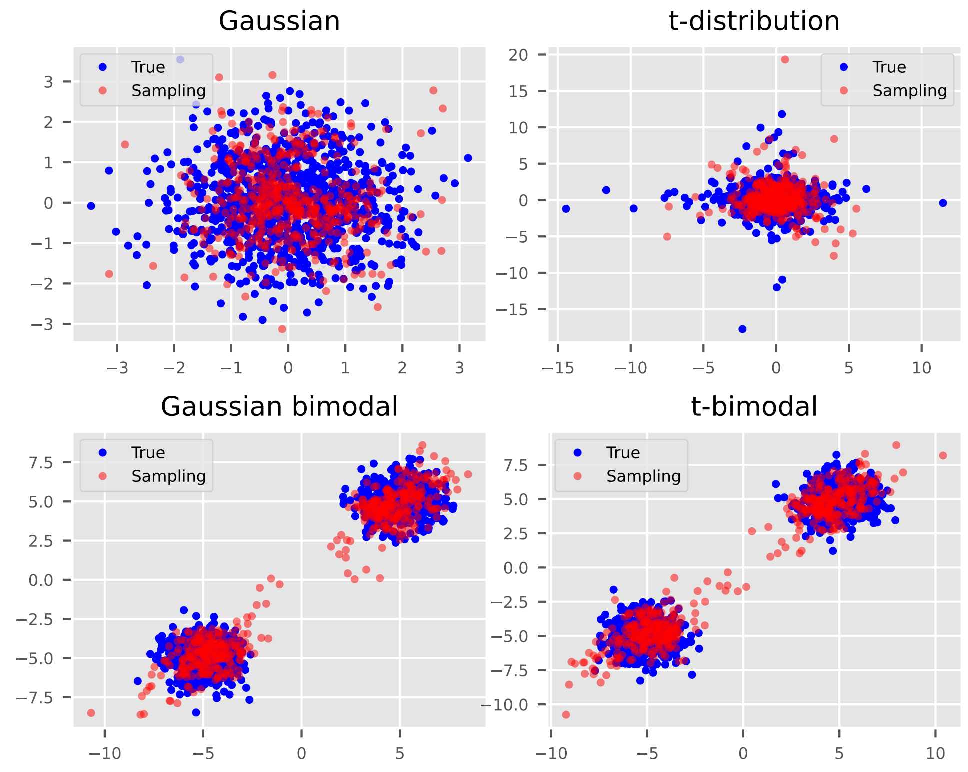


Figure 4.2: 2D Gaussian vs sampled (left) and 2D Student's t distribution vs sampled (center) and 2D bimodal Gaussian vs sampled (right)

for heavy-tailed unimodal distribution, we chose a degree of freedom $df = 3$ for t-distribution. On the other hand, it can capture third and forth moments of light and heavy-tailed distributions, but we can see in Figure 4.2 that there are some samples between two modes.

4.5 Bibliography

There many realizations of LSAP that are available with a python interface. For example, Scipy's optimization and root finding module[3] allows to find LSAP using a Hungarian algorithm when the cost matrix is unbalanced. A python library Lapjv[4] allows to find LSAP using Jonker-Volgenant algorithm[5]. The Sinkhorn algorithm[6],[7] is fast heuristic for the Kantorovich problem, that allows to solve efficiently LSAP, but the matrix obtained by using the Sinkhorn algorithm is not always a permutation matrix. It was implemented for some cases in POT library[8].

4.6 Appendix to Chapter 4

1D distributions. Table 4.11 illustrates the skewness, the kurtosis between $X \in \mathbb{R}^{1000\times 1}$ and $Y \in \mathbb{R}^{1000\times 1}$ for the Gaussian and Student's $t-$distributions from Section 4.4.

Table 4.11: Stats

	Gaussian bimodal 0	t-bimodal 0
Mean	0.021 (0.28)	-0.071 (0.18)
Variance	26 (25)	26 (25)
Skewness	-0.00033 (-0.12)	-0.00072 (-0.13)
Kurtosis	-1.9 (-1.8)	-1.9 (-1.8)
KS test	0.289 (0.05)	0.356 (0.05)

2D distributions. To check numerically some first properties of the generated distribution, We output in Table 4.12 the skewness and kurtosis, probability distances of both $X \in \mathbb{R}^{1000\times 2}$ and $Y \in \mathbb{R}^{1000\times 2}$. Each row represents the truth distribution X and generated distribution using a sampling function labeled as "sampled" Y:

Table 4.12: Summary statistics

	Gaussian 0	1	t-distribution 0	1	Gaussian bimodal 0	1
Mean	-0.032 (0.0084)	-0.0038 (0.058)	-0.11 (-0.034)	0.074 (0.2)	-0.012 (0.061)	-0.038 (0.0027)
Variance	0.95 (0.74)	1 (0.92)	2.7 (2.1)	2.9 (3.8)	26 (25)	26 (24)
Skewness	0.0057 (0.19)	0.076 (0.24)	-1.1 (-0.44)	-0.75 (2.2)	0.0026 (-0.015)	0.00023 (-0.001
Kurtosis	0.12 (0.57)	-0.11 (0.25)	13 (4.2)	21 (20)	-1.9 (-1.7)	-1.9 (-1.8)
KS test	0.12 (0.05)	0.127 (0.05)	0.395 (0.05)	0.222 (0.05)	0.116 (0.05)	0.092 (0.05)

[3]Scipy, see this url. https://github.com/src-d/lapjv

[4]Lapjv, see this url

[5]R. Jonker and A. Volgenant, "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems," Computing, vol. 38, pp. 325-340, 1987.

[6]Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. Pacific Journal of Mathematics, 21-343-348, 1967.

[7]Jason Altschuler, Jonathan Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. CoRR, 2017.(https://arxiv.org/abs/1705.09634)

[8]POT, see this url.

Chapter 5

Application to supervised machine learning

5.1 Aims of this chapter

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[1].

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.

5.2 Regression problem: housing price prediction

Description. This database 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.

A comparison between methods. We compare codpy's extrapolation operator defined in (3.2.8)-left with following machine learning models: decision tree (**DT**) by scikit-learn library and TensorFlow's neural network (**NN**) model. 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 $f(Z)$.

For the feed-forward NN we chose 50 epochs with batch size set to 16, with Adam optimization algorithm and MSE as the loss function. The NN is composed of two hidden (64 cells), one input (8 cells) and one output layers with the following sequence of activation functions: RELU - RELU - RELU - Linear. All the rest hyperparameters in the models are default set by scikit-learn, TensorFlow.

[1]kaggle, see this url.

Table 5.1: scenario list

D	N_x	N_y	N_z
-1	505.0	505.0	-1
-1	456.5	456.5	-1
-1	408.0	408.0	-1
-1	359.5	359.5	-1
-1	311.0	311.0	-1
-1	262.5	262.5	-1
-1	214.0	214.0	-1
-1	165.5	165.5	-1
-1	117.0	117.0	-1
-1	68.5	68.5	-1

The first plot in Figure 5.1 compares methods in term of scores, the second and third plots discrepancy errors and execution time for different scenarii defined in Table 5.1.

We give an interpretation of these results.

- First note that the RKHS-based method *codpy lab extra*, that is the extrapolation method, obtains both best scores and worst execution time.
- Note that if we subtract the discrepancy error from one, the result matches the scores of the method *codpy lab extra*. This indicates that the discrepancy error is an appropriate indicator.
- Another kernel method, *codpy lab proj*, that is the projection method above, is a more balanced method.
- 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.

Observe that function norms and MMD 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.

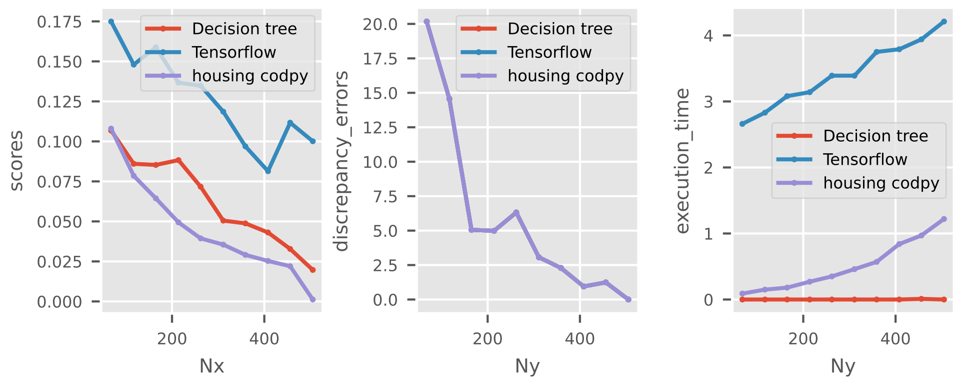


Figure 5.1: MMD and execution time

5.3 Classification problem: handwritten digits

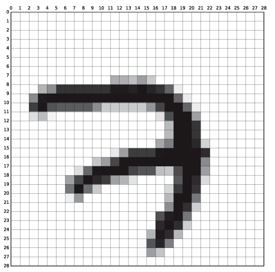
Description. This section contains an example of classification for images, which is a typical academic example referred to as the MNIST problem, and allows us to benchmark our results against more popular methods.

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 database of handwritten images has served as the basis for benchmarking classification algorithms.

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 this dedicated page for a description of the MNIST database, 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.

A comparison between methods. We compare different machine learning models to classify MNIST digits : support vector classifier (**SVC**), decision tree classifier (**DT**), adaboost classifier, random forest classifier(**RF**) by scikit-learn library and TensorFlow's neural network (**NN**) model.

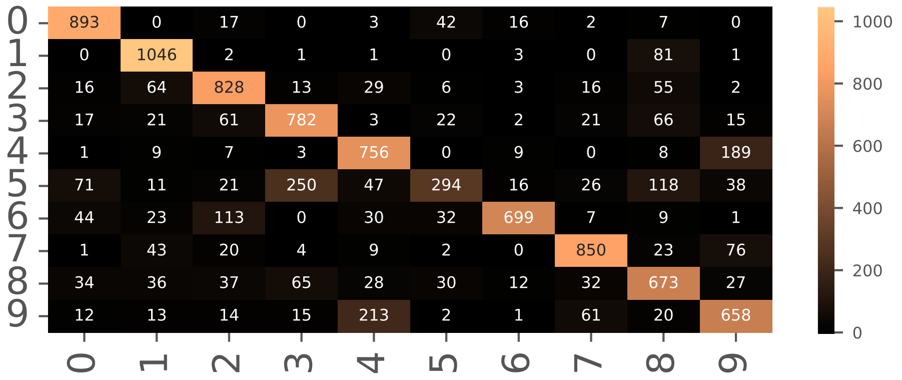
For the feed-forward NN we chose 10 epochs with batch size set to 16, with Adam optimization algorithm and sparse categorial entropy as the loss function. The NN is composed of 128 input and 10 output layers with a RELU activation function. All the rest hyperparameters in the models are default set by scikit-learn, TensorFlow. We also straightforwardly apply the projection operator (3.2.6) with the kernel function defined by a composition of the Gaussian kernel with a mean distance map, 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.

Table 5.2: 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 (2.3.2), a scalar in the interval between 0 and 1, which counts the number of correctly predicted images.

Figure 5.2 is a confusion matrix for the last scenario in Table 5.2 for a neural network.



	0	1	2	3	4	5	6	7	8	9
0	893	0	17	0	3	42	16	2	7	0
1	0	1046	2	1	1	0	3	0	81	1
2	16	64	828	13	29	6	3	16	55	2
3	17	21	61	782	3	22	2	21	66	15
4	1	9	7	3	756	0	9	0	8	189
5	71	11	21	250	47	294	16	26	118	38
6	44	23	113	0	30	32	699	7	9	1
7	1	43	20	4	9	2	0	850	23	76
8	34	36	37	65	28	30	12	32	673	27
9	12	13	14	15	213	2	1	61	20	658

Figure 5.2: Confusion matrix for Neural network: Tensorflow

Figure 5.3 compares methods in term of scores, MMD errors and execution time. We give an interpretation of these results.

- First notice that the kernel method *codpy class. extra* is a multiple-input/multiple-output classifier, which is basically an 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 class. extra*. This indicates that the discrepancy error is a pertinent indicator.
- Another RKHS - based method, *codpy class. proj*, allows to reduce the computational complexity of extrapolation by using a projection of the input data to lower dimensions. It is a more balanced method with respect to accuracy vs complexity.
- Both kernel methods use a standard Gaussian kernel, 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.

Observe that function norms and discrepancy errors are not method-dependent. Clearly, for this example, a periodic 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.

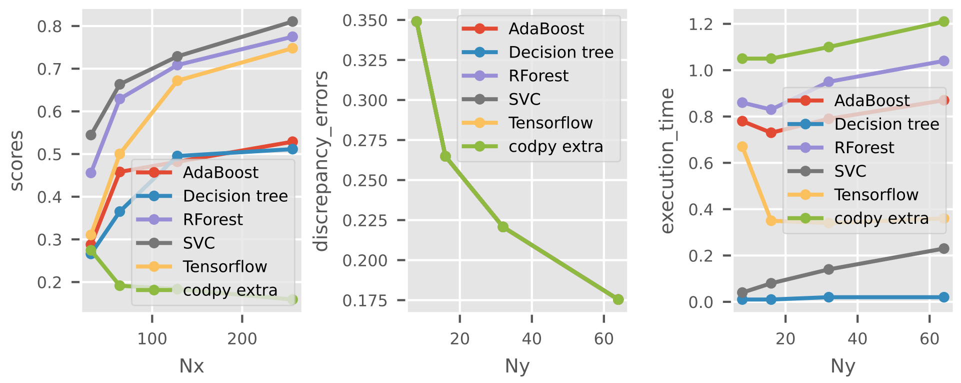


Figure 5.3: MMD and execution time

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

Description. 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-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.

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[2]. 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 (512×512) 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 (256×256) radon transform [3], called a **sinogram** [4]. 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 [5].

An example of training set construction is presented Figure 5.4. Left is the reconstructed image from the "high resolution" sinogram (middle). The low resolution sinogram is plot at right.

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

A comparison between methods. We present here the test resulting from a benchmark of a kernel-based method and the SART algorithm[6]

Following our notations, section 2.1, 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.

[2]the image set is available publicly at this kaggle link.
[3]An introduction to radon transform can be found at this wikipedia page.
[4]We used the standard radon transform from scikit, available at this url.
[5]We used a SART algorithm, 3 iterations, for reconstruction, available at this url.
[6]We did not succeed finding competitive parameters for other methods.

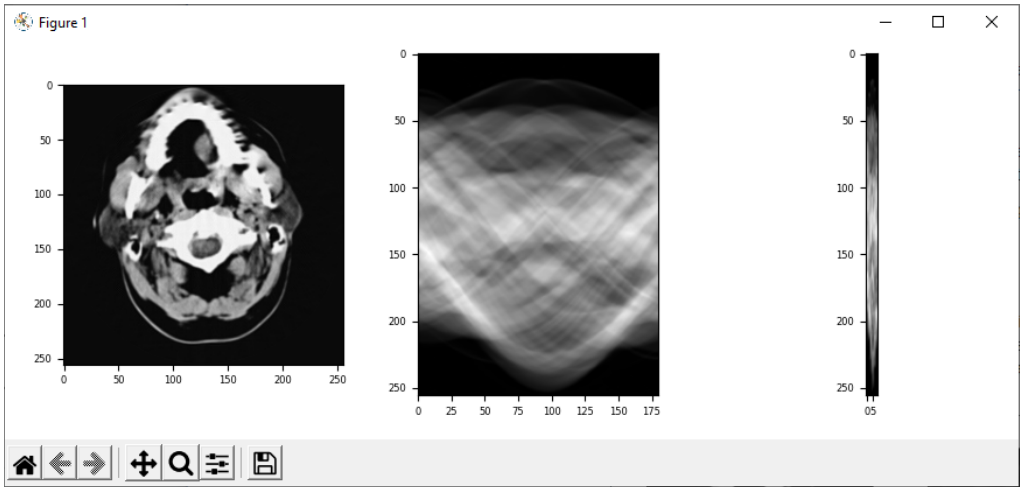


Figure 5.4: high resolution sinogram (middle), low resolution (right), reconstructed image (left)

- 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 identity, D, N_x, N_y, N_z, D_f, the execution time, and the score, computed with the RMSE % error indicator, see (2.3.2).

- 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).

Figure 5.5 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 emphasizes that such methods suit well to pattern recognition problems, as automated tools to support professionals diagnosis.

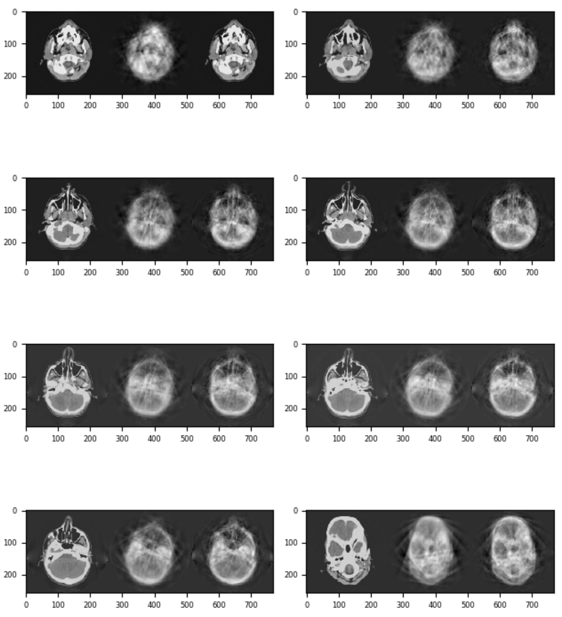


Figure 5.5: Example of reconstruction original (left), sub-sampled SART (middle), kernel extrapolation (right)

5.5 Appendix

Tables 5.3 and 5.4 indicates performance indicators for the Boston housing prices and MNIST datasets.

Table 5.3: Performance indicators for housing prices database

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD
housing codpy	13	505	505	-1	1	1.22	0.0012	0.0000
housing codpy	13	456	456	-1	1	0.97	0.0220	1.2415
housing codpy	13	408	408	-1	1	0.84	0.0253	0.9470
housing codpy	13	359	359	-1	1	0.57	0.0291	2.2870
housing codpy	13	311	311	-1	1	0.46	0.0355	3.0667
housing codpy	13	262	262	-1	1	0.35	0.0394	6.3171
housing codpy	13	214	214	-1	1	0.27	0.0494	4.9851
housing codpy	13	165	165	-1	1	0.18	0.0644	5.0520
housing codpy	13	117	117	-1	1	0.15	0.0785	14.5699
housing codpy	13	68	68	-1	1	0.09	0.1080	20.1727
Tensorflow	13	505	505	-1	1	4.21	0.1001	0.0000
Tensorflow	13	456	456	-1	1	3.94	0.1117	1.2415
Tensorflow	13	408	408	-1	1	3.79	0.0814	0.9470
Tensorflow	13	359	359	-1	1	3.75	0.0969	2.2870
Tensorflow	13	311	311	-1	1	3.39	0.1186	3.0667
Tensorflow	13	262	262	-1	1	3.39	0.1348	6.3171
Tensorflow	13	214	214	-1	1	3.14	0.1367	4.9851
Tensorflow	13	165	165	-1	1	3.08	0.1588	5.0520

Table 5.3: Performance indicators for housing prices database *(continued)*

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD
Tensorflow	13	117	117	-1	1	2.83	0.1480	14.5699
Tensorflow	13	68	68	-1	1	2.66	0.1749	20.1727
Decision tree	13	505	505	-1	1	0.00	0.0197	0.0000
Decision tree	13	456	456	-1	1	0.01	0.0328	1.2415
Decision tree	13	408	408	-1	1	0.00	0.0431	0.9470
Decision tree	13	359	359	-1	1	0.00	0.0488	2.2870
Decision tree	13	311	311	-1	1	0.00	0.0505	3.0667
Decision tree	13	262	262	-1	1	0.00	0.0717	6.3171
Decision tree	13	214	214	-1	1	0.00	0.0883	4.9851
Decision tree	13	165	165	-1	1	0.00	0.0853	5.0520
Decision tree	13	117	117	-1	1	0.00	0.0860	14.5699
Decision tree	13	68	68	-1	1	0.00	0.1068	20.1727

Table 5.4: Performance indicators for MNIST database

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD
Tensorflow	784	32	8	10000	1	0.67	0.3104	0.3490
Tensorflow	784	64	16	10000	1	0.35	0.5004	0.2648
Tensorflow	784	128	32	10000	1	0.34	0.6719	0.2207
Tensorflow	784	256	64	10000	1	0.36	0.7479	0.1754
codpy extra	784	32	8	10000	1	1.05	0.2744	0.3490
codpy extra	784	64	16	10000	1	1.05	0.1915	0.2648
codpy extra	784	128	32	10000	1	1.10	0.1834	0.2207
codpy extra	784	256	64	10000	1	1.21	0.1591	0.1754
SVC	784	32	8	10000	1	0.04	0.5446	0.3490
SVC	784	64	16	10000	1	0.08	0.6634	0.2648
SVC	784	128	32	10000	1	0.14	0.7288	0.2207
SVC	784	256	64	10000	1	0.23	0.8105	0.1754
Decision tree	784	32	8	10000	1	0.01	0.2660	0.3490
Decision tree	784	64	16	10000	1	0.01	0.3652	0.2648
Decision tree	784	128	32	10000	1	0.02	0.4954	0.2207
Decision tree	784	256	64	10000	1	0.02	0.5115	0.1754
AdaBoost	784	32	8	10000	1	0.78	0.2878	0.3490
AdaBoost	784	64	16	10000	1	0.73	0.4581	0.2648
AdaBoost	784	128	32	10000	1	0.79	0.4819	0.2207
AdaBoost	784	256	64	10000	1	0.87	0.5289	0.1754
RForest	784	32	8	10000	1	0.86	0.4558	0.3490
RForest	784	64	16	10000	1	0.83	0.6292	0.2648
RForest	784	128	32	10000	1	0.95	0.7085	0.2207
RForest	784	256	64	10000	1	1.04	0.7749	0.1754

Chapter 6

Applications to unsupervised machine learning

6.1 Aims of this chapter

In this section we apply some clustering methods for a number of use cases.We benchmarked our kernel-based algorithms (see Section 2.4.4 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 algorithm, the distance is called the *inertia*, see section **??**.
- For kernel-based algorithms, the distance is *MMD*, see section 3.2.6.

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-means algorithm usually output different clusters output at different runs.

6.2 Classification problem: handwritten digits

Description. The MNIST test is also studied in the section 5. 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 $Z \in \mathbb{R}^{N_z \times D}$.

A comparison between methods. 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 N_y 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 stabilizes, or the defined number of iterations is reached.

Second we apply codpy's MMD minimization-based algorithm described in **??** using the distance $d_k(x, y)$ induced by a Gaussian kernel: $k(x, y) = \exp(-(x - y)^2)$.

Table 6.1: scenario list

D	N_x	N_y	N_z
-1	1000	128	1000
-1	1000	256	1000

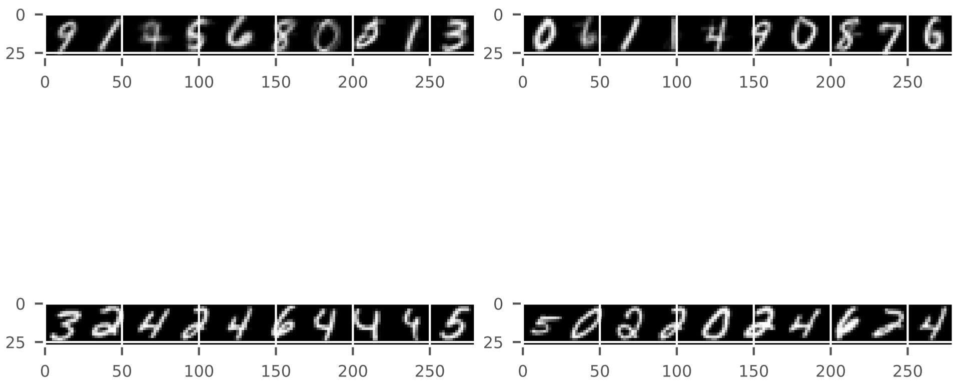
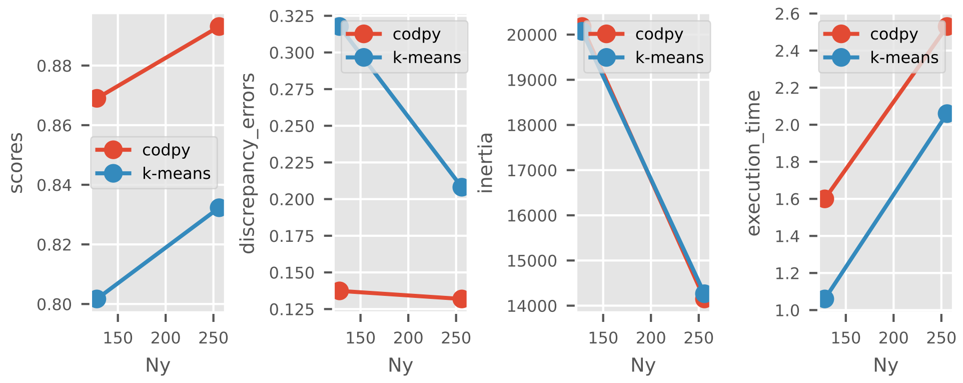


Figure 6.1: Scikit (the first row) and codpy (second row) clusters interpreted as images

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. Figure 6.1 plots some examples of computed clusters, interpreted as images. As can be seen, they are perfectly recognizable.

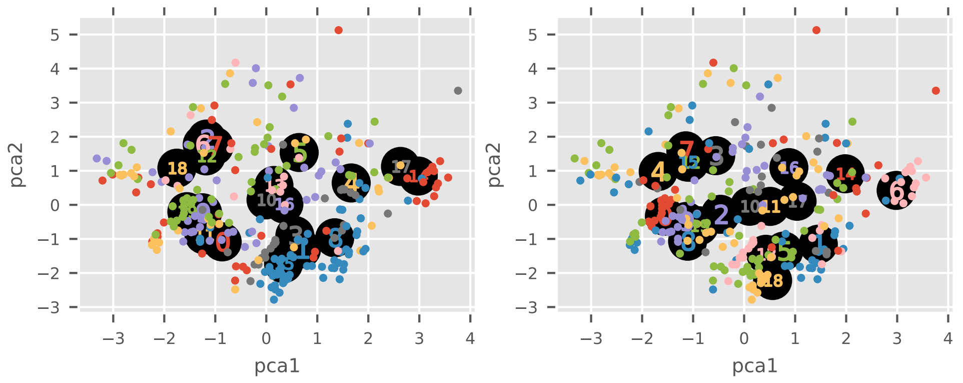
Finally, we illustrate a benchmark plot, displaying the computed performance indicator of scikit’s k-means and codpy’s MMD minimization-based algorithm in terms of MMD, inertia, accuracy scores (when applicable) and execution time, using scenarios in Table 6.1. The higher the scores and the lower are the inertia and MMD the better.



The scores are quite high, compared to supervised methods for similar size of training set, see results section 5. MMD-based minimization have an inertia indicator that is comparable to k-means. This is surprising as k-means algorithms are based on inertia minimization. Moreover, scores seems to indicate that the MMD distance is a more reliable criteria than inertia on this pattern recognition problem.

6.3 German credit risk

Description. The original dataset[1] contains 1000 entries with 20 categorial/symbolic attributes. In this database, 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.



A comparison between methods. The result of k-means and codpy's sharp discrepancy algorithm algorithm is N_y clusters in D dimensions. Notice that the cluster centroids themselves are D-dimensional points.

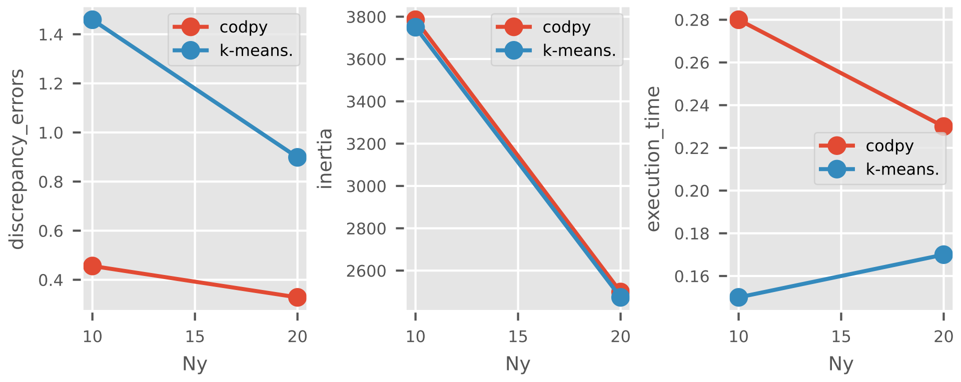
Table 6.2: scenario list

D	N_x	N_y	N_z
-1	-1	10	-1
-1	-1	20	-1

Next we visualize the clusters and corresponding centroids of scikit and codpy's sharp discrepancy algorithm, where we vary the number of clusters N_y 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 $N_y = 8$.

[1]The German credit risk dataset is described in the kaggle page link

Finally, we illustrate a benchmark plot, displaying the computed performance indicators of scikit's k-means and codpy's sharp discrepancy algorithms using scenarios from Table 6.2.



6.4 Credit card marketing strategy

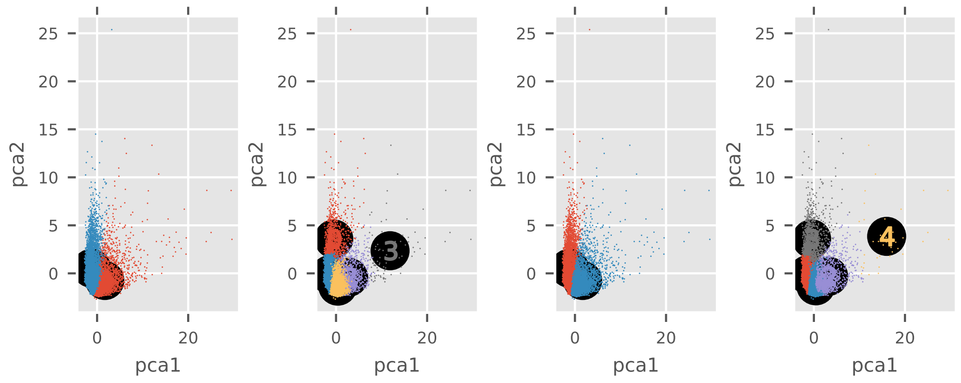
Description. The problem can be formalized as follows. Develop a customer segmentation to define marketing strategy. The sample dataset[2] summarizes the usage behavior of 8,950 active credit card holders during the last 6 months. The database contains 17 features and 8,950 records. The data describes 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.

A comparison between methods. The result of k-means algorithm and codpy's sharp discrepancy algorithm is N_y clusters in D dimensions. Note that the cluster centroids $Y \in \mathbb{R}^{N_y \times D}$ themselves are D-dimensional points.

Table 6.3: scenario list

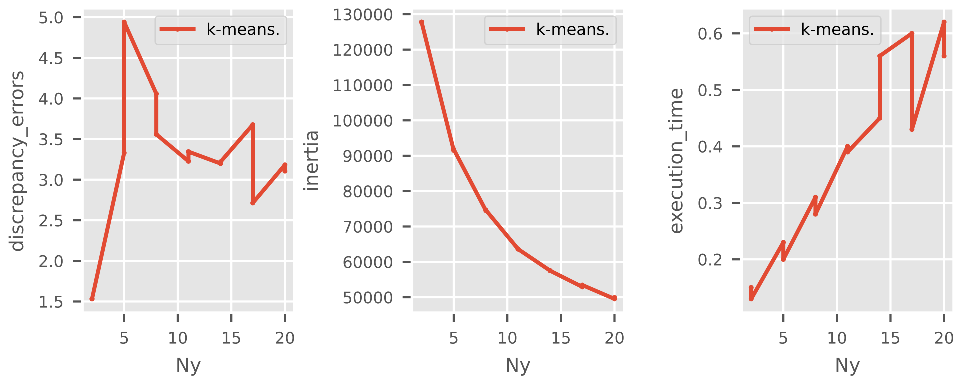
D	N_x	N_y	N_z
-1	-1	2	-1
-1	-1	5	-1
-1	-1	8	-1
-1	-1	11	-1
-1	-1	14	-1
-1	-1	17	-1
-1	-1	20	-1

[2]The credit card marketing strategy dataset is detailed on this dedicated kaggle page.



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

Finally, we illustrate a benchmark plot, displaying the computed performance indicator of scikit's k-means and codpy's sharp discrepancy algorithms using scenarii from Table 6.3.



6.5 Credit card fraud detection

Description. The database[3] 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 database 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 database. 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.

A comparison between methods. Table 6.4 defines different scenarii of our experiment.

[3]You can find more details on this use case following the link kaggle page link.

Table 6.4: scenario list

D	N_x	N_y	N_z
-1	500	15	1000
-1	500	30	1000
-1	500	45	1000
-1	500	60	1000
-1	500	75	1000
-1	500	90	1000

Figure 6.2 illustrates confusion matrices for the last scenario of each approach.

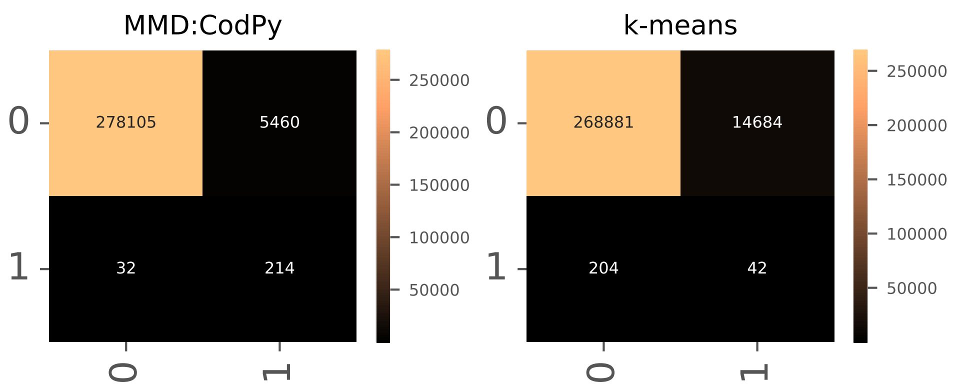
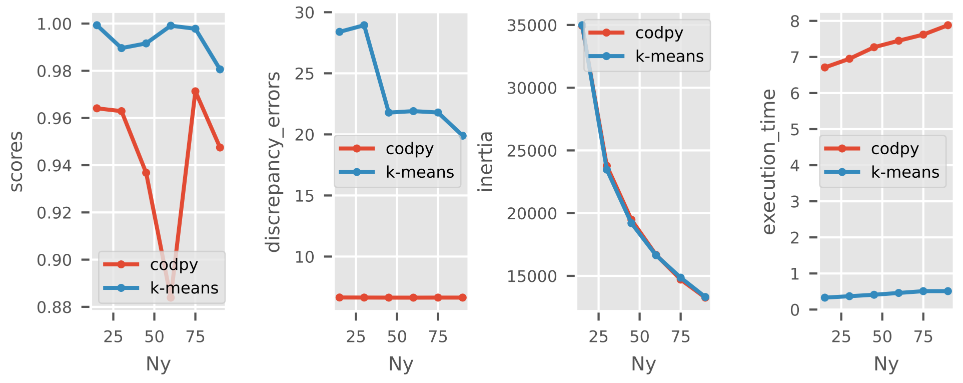


Figure 6.2: confusion matrix for codpy

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.



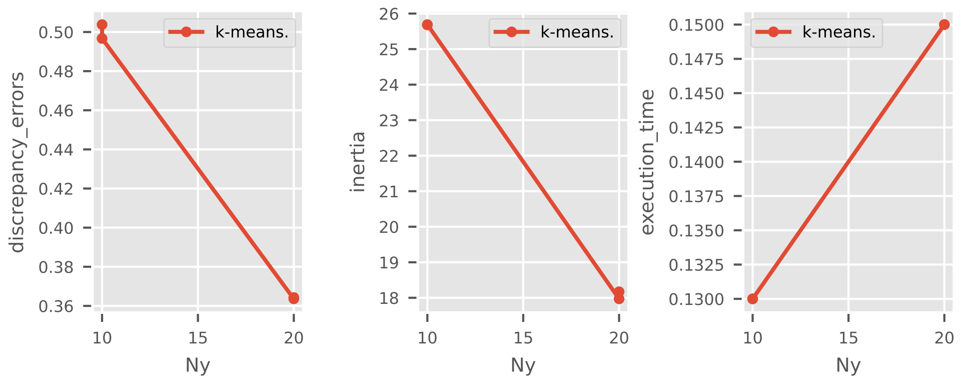
6.6 Portfolio of stock clustering

Description. 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.

Table 6.5: Stock's clustering

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

A comparison between methods. The table with a list of stocks shows that k-means clustering and MMD minimization displays stocks into coherent groups. 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.



6.7 Appendix

Table 6.6: Performance indicators for MNIST dataset

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD	inertia
k-means	784	1000	128	-1	1	1.06	0.8017	0.3177	20073.11
k-means	784	1000	256	-1	1	2.06	0.8323	0.2081	14263.97
codpy	784	1000	128	-1	1	1.60	0.8690	0.1374	20179.44
codpy	784	1000	256	-1	1	2.53	0.8931	0.1319	14148.60

Table 6.7: Performance indicators for German credit database

predictors	D	N_x	N_y	N_z	D_f	time	MMD	inertia
k-means.	24	272	10	-1	0	0.15	1.4592	3750.21
k-means.	24	272	20	-1	0	0.17	0.8986	2473.49
codpy	24	272	10	-1	0	0.28	0.4561	3785.59
codpy	24	272	20	-1	0	0.23	0.3283	2498.88

Table 6.8: Performance indicators for credit card marketing database

predictors	D	N_x	N_y	N_z	D_f	time	MMD	inertia
k-means.	17	8950	2	-1	0	0.15	1.5323	127785.05
k-means.	17	8950	5	-1	0	0.23	3.3306	91502.98
k-means.	17	8950	8	-1	0	0.31	4.0575	74492.65
k-means.	17	8950	11	-1	0	0.40	3.2265	63645.30
k-means.	17	8950	14	-1	0	0.45	3.1983	57496.11
k-means.	17	8950	17	-1	0	0.60	3.6764	52926.03
k-means.	17	8950	20	-1	0	0.62	3.1822	49534.69
k-means.	17	8950	2	-1	0	0.13	1.5320	127784.87
k-means.	17	8950	5	-1	0	0.20	4.9401	91805.97
k-means.	17	8950	8	-1	0	0.28	3.5581	74624.72
k-means.	17	8950	11	-1	0	0.39	3.3447	63618.66
k-means.	17	8950	14	-1	0	0.56	3.2204	57483.26
k-means.	17	8950	17	-1	0	0.43	2.7132	53445.48
k-means.	17	8950	20	-1	0	0.56	3.1055	49898.14

Table 6.9: Performance indicators for credit card fraud database

predictors	D	N_x	N_y	N_z	D_f	time	scores	MMD	inertia
k-means	30	996	15	-1	1	0.33	0.9993	28.3943	34979.77
k-means	30	996	30	-1	1	0.37	0.9896	28.9360	23470.18
k-means	30	996	45	-1	1	0.41	0.9916	21.7840	19200.31
k-means	30	996	60	-1	1	0.46	0.9991	21.9045	16644.38
k-means	30	996	75	-1	1	0.51	0.9978	21.7911	14858.67
k-means	30	996	90	-1	1	0.51	0.9806	19.8914	13316.49
codpy	30	996	15	-1	1	6.71	0.9641	6.6490	34930.21
codpy	30	996	30	-1	1	6.95	0.9629	6.6440	23771.68
codpy	30	996	45	-1	1	7.27	0.9368	6.6387	19474.94
codpy	30	996	60	-1	1	7.45	0.8839	6.6366	16675.26
codpy	30	996	75	-1	1	7.62	0.9713	6.6413	14704.13
codpy	30	996	90	-1	1	7.88	0.9475	6.6409	13263.98

Table 6.10: Performance indicators for stock price

predictors	D	N_x	N_y	N_z	D_f	time	MMD	inertia
k-means.	963	60	10	-1	0	0.13	0.5038	25.68
k-means.	963	60	20	-1	0	0.15	0.3636	17.97
k-means.	963	60	10	-1	0	0.13	0.4967	25.69
k-means.	963	60	20	-1	0	0.15	0.3643	18.17

Chapter 7

Generative models with kernels

7.1 Aim of this section

Synthetic data generation is a data obtained as a result of fitting observed data to a given model. There are many applications using synthetic financial time series data, for risk management, decision tools, or backtesting purposes.

A classical approach to this problem are autoregressive, also called parametric methods, fitting a known process to market observations, as GARCH (Generalized Autoregressive Conditional Heteroscedasticity).

A more recent field of research are non-parametric models, based on neural networks, as for instance GAN (Generative Adversarial Networks). There exists a number of works using GANs in finance for time series prediction, portfolio management or fraud detection, see for instance [9] for a review. However, this approach still needs to prove its efficiency for pricing purposes.

In this paper, we describe an alternative approach to non-parametric models, producing synthetic data using kernel methods. Kernel methods are explainable since we can measure the accuracy of predictions with error estimates. These estimates are based on a distance between measures, that is a natural link to optimal transport theory. This allows to reproduce any random variables based on the observation of their realizations, as well as to quantify theoretically the discretization error.

Indeed, the capability to reproduce a given random variable accurately is key to synthetic data. The section 7 describes our approach, whereas the section 7.2 gives numerical illustrations of our construction.

Finally, we illustrate our approach via two financial applications in section **??**. The first checks that the time series forecast can be used for Monte Carlo pricing. The second is a P&L explanation that can be used for intradays real time P&L approximation of large derivative portfolios. We compute various numerical metrics to show the convergence properties of our methods.

7.1.1 Settings

Let $\mathbb{X}$ be an unknown probability measure, **absolutely continuous** with respect to the Lebesgue measure, supported over a convex set $\mathcal{X} \subset \mathbb{R}^D$, D being the dimension, which is the number of risk sources for financial applications. We focus in this paper on the discrete case, that is, let

$$X := \{x^n_d\}_{n,d=1}^{N_x,D} \in \mathbb{R}^{N_x,D} \tag{7.1.1}$$

be a set of **distinct** points in $\mathcal{X}$, defined as random samples following $\mathbb{X}$, and consider the discrete probability measure $\mathbb{X}_x = \frac{1}{N_x}\sum_{i=1}^{N_x} \delta_{x^i}$, δ_x being the Dirac measure concentrated at x. Consider

$\mathbb{Y}$ another probability measure, with a known law, for instance a uniform distribution over $\mathcal{X}$, and define as for $\mathbb{X}_x$, $\mathbb{Y}_y = \frac{1}{N_y}\sum_{i=1}^{N_y}\delta_{y^i}$.

7.1.2 Kernel review

We refer to [3] for a complete introduction to reproducing kernel Hilbert spaces (RKHS) theory. We call a function $k : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ a kernel if it is symmetric and positive definite (see [3] for a definition). A reproducing kernel Hilbert space $\mathcal{H}_k$ is a Hilbert space, generated by the kernel k, which scalar product satisfies the following reproducing property : $k(x, y) = \langle k(x, \cdot), k(y, \cdot)\rangle_{\mathcal{H}_k}$, $\forall (x, y) \in \mathcal{X} \times \mathcal{X}$, see [15].

The discrepancy between two probability measures $\mathbb{X}$ and $\mathbb{Y}$, induced by a kernel k is

$$\begin{gathered} D_k(\mathbb{X},\mathbb{Y})^2 := \int\int k(x,y)d\mathbb{X}d\mathbb{X}+ \\ \int\int k(x,y)d\mathbb{Y}d\mathbb{Y} - 2\int\int k(x,y)d\mathbb{X}d\mathbb{Y}. \end{gathered} \tag{7.1.2}$$

For the discrete case, this amounts to the following formula, introduced in [14]

$$\begin{gathered} D_k(\mathbb{X}_x,\mathbb{Y}_y)^2 := \alpha\sum_{n,m=1}^{N_x} k(x^n,x^m)+ \\ \beta\sum_{n,m}^{N_y} k(y^n,y^m) - \gamma\sum_{n,m=1}^{N_x,N_y} k(x^n,y^m), \end{gathered} \tag{7.1.3}$$

where $\alpha = \frac{1}{N_x^2}$, $\beta = \frac{1}{N_y^2}$ and $\gamma = \frac{2}{N_x N_y}$. For (7.1.2), or (7.1.3), we define sharp discrepancy sequences (SDS, see [31]), as solutions of the following, non-convex, minimization problem

$$\bar{Y} = \arg\inf_{y\in\mathcal{X}^{N_y}} D_k(\mathbb{X},\mathbb{Y}_y) \tag{7.1.4}$$

We introduce also the discrepancy matrix induced by the kernel k as $M_k(X, Y) := (d_k(x^n, y^m))_{n,m=1}^{N_x,N_y}$, defined as

$$d_k(x,y) = k(x,x) + k(y,y) - 2k(x,y) \tag{7.1.5}$$

Let $f : \mathcal{X} \mapsto \mathbb{R}^{D_f}$ any vector valued function. Kernel methods allow to define a simple interpolation / extrapolation procedure. Denoting $z \mapsto f_z$ the interpolated function and the "ground truth values" $f(z)$, we introduce the following sets, using a classical terminology for a supervised machine learning. Consider a training set $X, f(X) \in \mathcal{X}^{N_x}, \mathbb{R}^{N_x,D_f}$, a test set $Z, f(Z) \in \mathcal{X}^{N_z}, \mathbb{R}^{N_z,D_f}$, as well as a third set $Y \in \mathcal{X}^{N_y}$ of internal parameters (to fix ideas, Y is equivalent to the "weight set" for neural networks). Let $K(X, Y)$ be a kernel matrix, induced by the kernel k, i.e. $K(X, Y) := (k(x^n, y^m))_{n,m=1}^{N_x,N_y}$. We define a projection operator $f_Z := \mathcal{P}_k(X, Y, Z)f(X)$, induced by the kernel k, defined as a matrix through

$$\mathcal{P}_k(X,Y,Z) := K(Y,Z)K(X,Y)^{-1}. \tag{7.1.6}$$

The inverse is computed using a least-squares approach, as follows, $K(X, Y)^{-1} = (K(Y, X)K(X, Y)+ \epsilon I_d)^{-1}K(Y, X)$, where $\epsilon \geq 0$ is a (optional) regularization term. The projection operator (7.1.6) benefits from the following error estimate (see [31]), that are confidence levels

$$\|f(Z) - f_Z\|_{\ell^2} \leq D_k(X,Y,Z)\|f\|_{\mathcal{H}_k}, \tag{7.1.7}$$

where $D_k(X, Y, Z) := D_k(X, Y) + D_k(Y, Z)$. Starting from the formula (7.1.6), we can define all kind of differential operators, as for instance the gradient

$$\nabla f_Z = (\nabla_Z K)(Y,Z)K(X,Y)^{-1}f(X). \tag{7.1.8}$$

7.1.3 Kernel-based transport maps

Consider a map T that transports $\mathbb{Y}_y$ into $\mathbb{X}_x$. Using standard optimal transport definitions, T is a push forward map with notation $T_{\#}\mathbb{Y}_y = \mathbb{X}_x$. To fix ideas, in the discrete case, and $N_x = N_y$, T is defined through any permutation map $\sigma : \{1, ..., N_x\} \mapsto \{1, ..., N_x\}$, as $T(Y) := X^\sigma := \{x^{\sigma(n)}\}_{n=1}^{N_x}$.

To set a well-defined map, we choose to define T as a convex map, with respect to a non-Euclidean metric, which is the discrepancy (7.1.5), as follows

$$\bar{\sigma} = \arg \inf_{\sigma \in \Sigma} Tr(M_k(X^\sigma, Y)), \tag{7.1.9}$$

where Σ is the set of all permutations, and Tr holds for the trace of the matrix M_k. This problem can be solved for instance as a linear sum assignment problem, and the resulting values can be used as initial ones to solve the problem (7.1.4) through a gradient descent algorithm.

Once an optimal permutation $\bar{\sigma}$ is computed, we use the projection operator (7.1.6) to define a continuous map G, as

$$z \mapsto G^{\mathbb{X}_x}(z) := \mathcal{P}_k(Y, Y, z)X^{\bar{\sigma}}. \tag{7.1.10}$$

In particular, suppose that $Z \in \mathcal{X}^{N_z}$ is IID of the same random variable used to sample Y, then $\mathcal{P}_k(Y, Y, Z)X^{\bar{\sigma}}$ is a natural candidate of N_z IID random samples of $\mathbb{X}$.

7.1.4 Time series forecasting

In this paper we consider time series forecasting as fitting a model in order to match a stochastic process $t \mapsto X(t) \in \mathbb{R}^D$, observed on a time grid $t_x^1 < ... < t_x^{T_x}$, the data having the following shape

$$X := \left(x_d^{n,k}\right)_{d=1...D}^{n,k=1...N_x,T_x} \in \mathbb{R}^{N_x,D,T_x}. \tag{7.1.11}$$

In (7.1.11), N_x is the number of observed trajectories, the third component of this 3-dimensional tensor corresponding to the time index, and the dimension is D, or might also be $D+1$, if the time t^k is added to the observation set to take into account time dependencies. Note that market data consists usually in only one trajectory of a stochastic process, hence $N_x = 1$ in this paper. However, in other applications, $N_x \gg 1$, as are for instance customers data.

Feature engineering is a classical approach in machine learning that consists on adding new features to target a model. In our case, we use an injective map $F : \mathbb{R}^{N_x,D,T_x} \mapsto \mathbb{R}^{N_F,D_F}$, assuming that $F(X)$ is a random variable. By reproducing this random variable, one can generate any number N_z of trajectories as follows:

- Consider any input data X having shape (7.1.11), use the map F to retrieve $F(X) \in \mathbb{R}^{N_F \times D_F}$ which are random samples of $\mathbb{X}$.
- Generate samples using (7.1.10), considering $F(X) \in \mathbb{R}^{N_F \times D_F}$ as the training set.
- From these samples, use F^{-1} to output samples having shape $\mathbb{R}^{N_z,D,T_z}$, at any time grid $t_z^1 < ... < t_z^{T_z}$.

In this paper, we consider a simple model assumption [1], adapted to stock markets modeled with Markovian processes, fitting any positive time series to a Markovian process $t \mapsto X_t \in \mathbb{R}^D$ having shape

$$X_t = X_s \exp((t-s)\mu + \sqrt{t-s}\mathbb{X}), \tag{7.1.12}$$

where the unknown random variable, modeling the martingale component of the process, satisfies $\mathbb{E}(\mathbb{X}) = 0$ and is supposed **absolutely continuous** with respect to the Lebesgue measure.

[1]This choice is motivated to provide benchmarks in this paper.

Hence we introduce the log-return map $F(X) : \mathbb{R}^{N_x,D,T_x} \mapsto \mathbb{R}^{N_x\times T_x,D}$, defined as

$$\Big(\frac{\ln(x_d^{n,k}) - \ln(x_d^{n,k-1})}{\sqrt{t^k - t^{k-1}}}\Big)_{d=1\ldots D}^{n,k=1\ldots N_x,T_x}. \tag{7.1.13}$$

Considering any time grid $t_z^1 < \ldots < t_z^{T_z}$, we can define the inverse map $F^{-1}(Z) : \mathbb{R}^{N_z\times T_z,D} \mapsto \mathbb{R}^{N_z,D,T_z}$ as an exponential - integral operator.

7.1.5 Recurrent methods for time series predictions

Let us describe recurrent methods that can be implemented for any predictive machine (2.1.1), and we discuss an example of prediction.

Consider some historical observations X as in (**??**), and two integers H and P, satisfying $H+P \leq T_X$. H is called the historical depth, P the prediction depth. This setting defines a sliding window of size H+P over the data X, used to define the training set as follows (using slicing notations)

$$X^0 = X^{[\cdot,\cdot,i:i+H]} \in \mathbb{R}^{\tilde{N}_X\times D\times H}, f(X^0) = X^{[\cdot,\cdot,i+H:i+H+P]} \in \mathbb{R}^{\tilde{N}_X\times D\times P}$$

for any $i = 1,\ldots,\tilde{N}_X$, with $\tilde{N}_X = (T-H-P)N_X$. We can iterate the procedure, producing at each step P new predicted values, using recursively a predictive machine (2.1.1) as follows

$$X^{k+1} = [X^k, f(X^k)], \qquad f(X^{k+1}) = \mathcal{P}_m(X^k, Y, X^{k+1}, f(X^k)),$$

$[X^k, f(X^k)]$ being the concatenation of these two tensors in the last variable. Such a construction allows to produce predicted values of the temporal series at any future times.

7.2 Numerical illustration

Let X as in (7.1.1). We use kernels together with maps $(k\circ S)(x,y)$, adapted to our sets, to ensure positive-definiteness. In the numerical sections to follow, our kernel choice is

$$k(x,y) = \Pi_{d=1\ldots D}\left(1 - |x_d - y_d|\right), \tag{7.2.1}$$

that is the kernel equivalent of a RELU activation function, together with the following scaling map : $S(x) = \Big(S_d(x_d)\Big)_{d=1\ldots D}$, with

$$S_d(x_d) = \frac{x_d - \overline{x_d}}{\max_{x^n\in X} x_d^n - \min_{x^n\in X} x_d^n}, \tag{7.2.2}$$

$\overline{x_d}$ being the mean $\frac{\sum_{n=1\ldots N_x} x_d^n}{N_x}$.

7.2.1 One dimensional distributions

For illustration goals, we apply the algorithm (7.1.10) for two bi-modal distributions based on a Gaussian and a Student's distribution namely $\mathcal{N}(0,1)$ and $t(\nu=5)$. We use here two distinct sets (training set X and test set Z) to highlight some convergence properties of (7.1.10):

- IID : X, Z are iid samples of $\mathbb{X}$.
- SDS : X, Z are sharp discrepancy sequences (SDS) of $\mathbb{X}$, see (7.1.4).

For both sets, the size of the training set is $N_x = 1000$, whereas the size of the test set is $N_z = 500$. We plot the results computed by the sampler algorithm (7.1.10) in Figure 7.1 (resp. Figure 7.2) for the IID case (resp. SDS case).

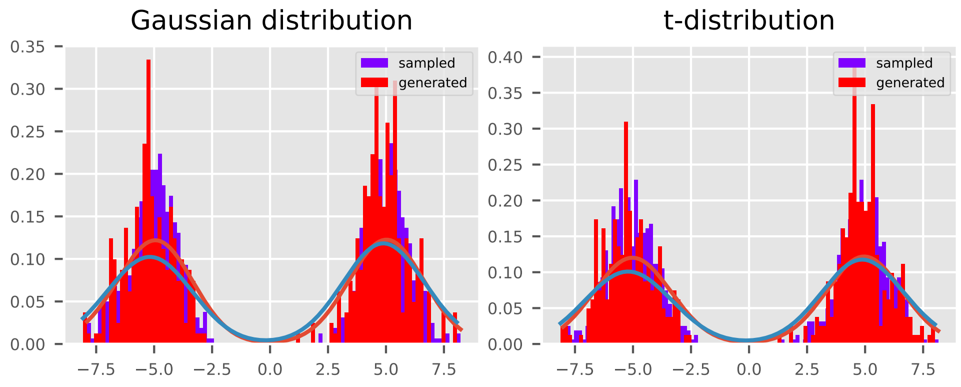


Figure 7.1: Density of generated IID distributions

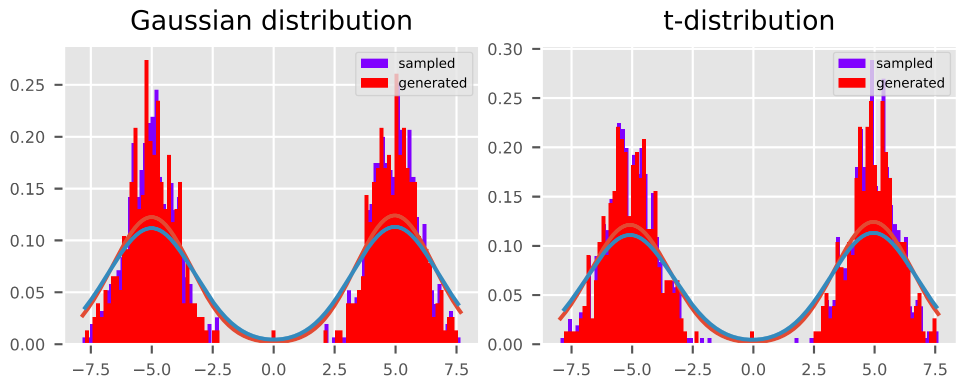


Figure 7.2: Density of generated SDS distributions

To measure a fit of a generated discrete distribution $G^{\mathbb{X}_x}$ to the original distribution $\mathbb{X}$, throughout this paper we compute summary statistics of the generated distribution, together with the original one, as well as the result of the Kolmogorov-Smirnov (KS) test for marginals and, eventually, compute correlation matrices. The computed moment values for the original distribution are quoted in parenthesis. The value for KS tests corresponds to the p-value for a 95% level (a successful KS test is above 0.05).

Table 7.1: Statistics of IID-generated distributions

	Gaussian 0	t-distribution 0
Mean	-0.0056 (0.16)	-0.038 (0.18)
Variance	26 (27)	26 (27)
Skewness	-0.0027 (-0.12)	-0.0026 (-0.12)
Kurtosis	-1.8 (-1.8)	-1.8 (-1.8)
KS test	0.262 (0.05)	0.546 (0.05)

Table 7.2: Statistics of SDS-generated distributions

	Gaussian 0	t-distribution 0
Mean	-0.019 (-0.019)	-0.069 (-0.069)
Variance	26 (26)	26 (26)
Skewness	-0.0039 (-0.0039)	-0.0061 (-0.0061)
Kurtosis	-1.9 (-1.9)	-1.9 (-1.9)
KS test	1.0 (0.05)	1.0 (0.05)

These two tables show how an appropriate choice of sets can drastically improve the convergence performance of (7.1.10). The convergence rate for the set of IID random samples $X \in \mathbb{R}^{N_x,D}$ is of order $\mathcal{O}(\frac{1}{\sqrt{N_x}})$, while SDS's convergence is of order $\mathcal{O}(\frac{1}{N_x^2})$ for smooth distributions, see [31].

7.2.2 Time series forecasting illustration

We illustrate the time series forecast algorithm, see section 7.1.4 with real market data, retrieved from January 1, 2016 to December 31, 2020, for three assets: Google, Apple and Amazon.

After applying the log-return map, we produce samples of this distribution using (7.1.10) and draw both distributions, which represent historical and generated log-returns, in Figures 7.3 and 7.4, projected on two of its components Apple and Google. We check the match between the original and generated distribution using the same table of statistics described in section **??**. Notice that the p-value in K-S test is higher than .05 for the three marginals (see Table 7.3, line K-S).

Table 7.3: Summary statistics for Apple, Amazon and Google

	AAPL	AMZN	GOOGL
Mean	1.3e-03 (8.6e-04)	1.3e-03 (1.7e-03)	8.6e-04 (1.1e-03)
Variance	2.9e-04 (2.8e-04)	2.8e-04 (2.4e-04)	2.3e-04 (1.9e-04)
Skewness	-5.3e-01 (2.5e-02)	1.2e-01 (3.7e-01)	-5.6e-01 (3.0e-02)
Kurtosis	8.0e+00 (3.3e+00)	3.6e+00 (1.9e+00)	7.4e+00 (3.3e+00)
KS test	0.372 (0.05)	0.695 (0.05)	0.672 (0.05)

We also check that the historical correlation matrices Table 7.4 is close to the generated one at Table 7.5.

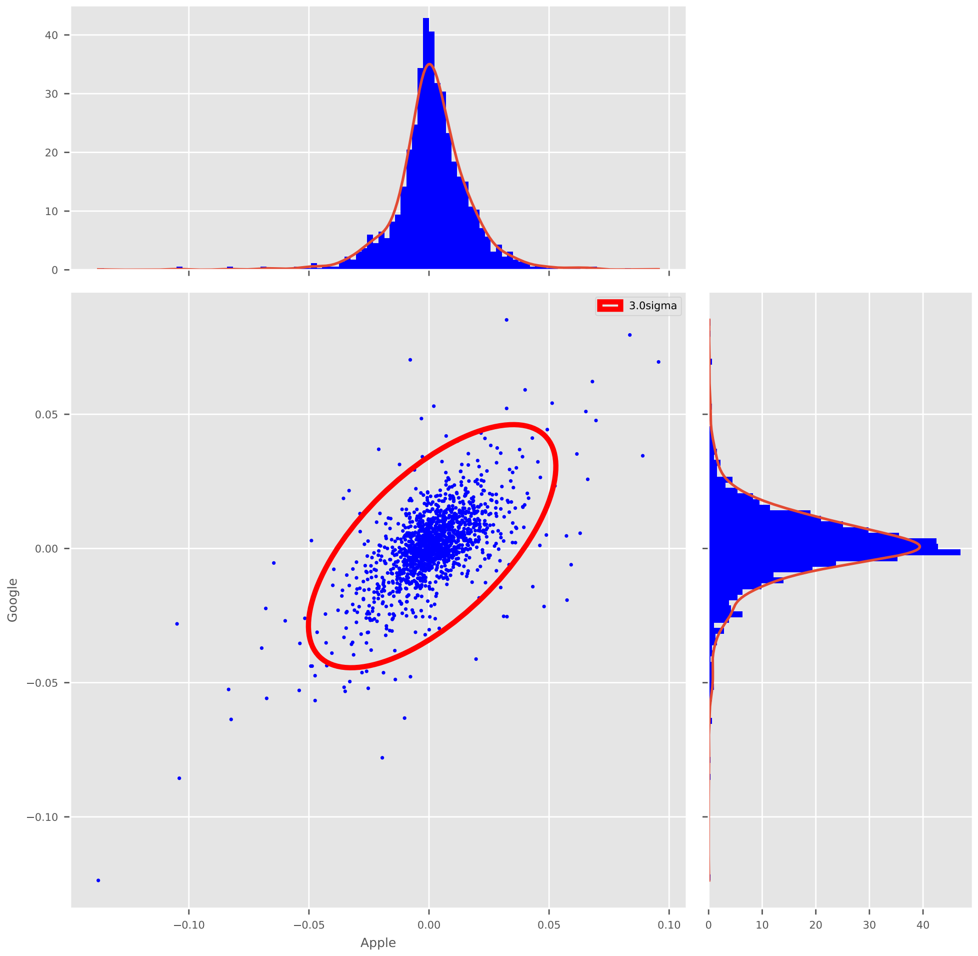


Figure 7.3: Historical log-return distribution for Apple / Google

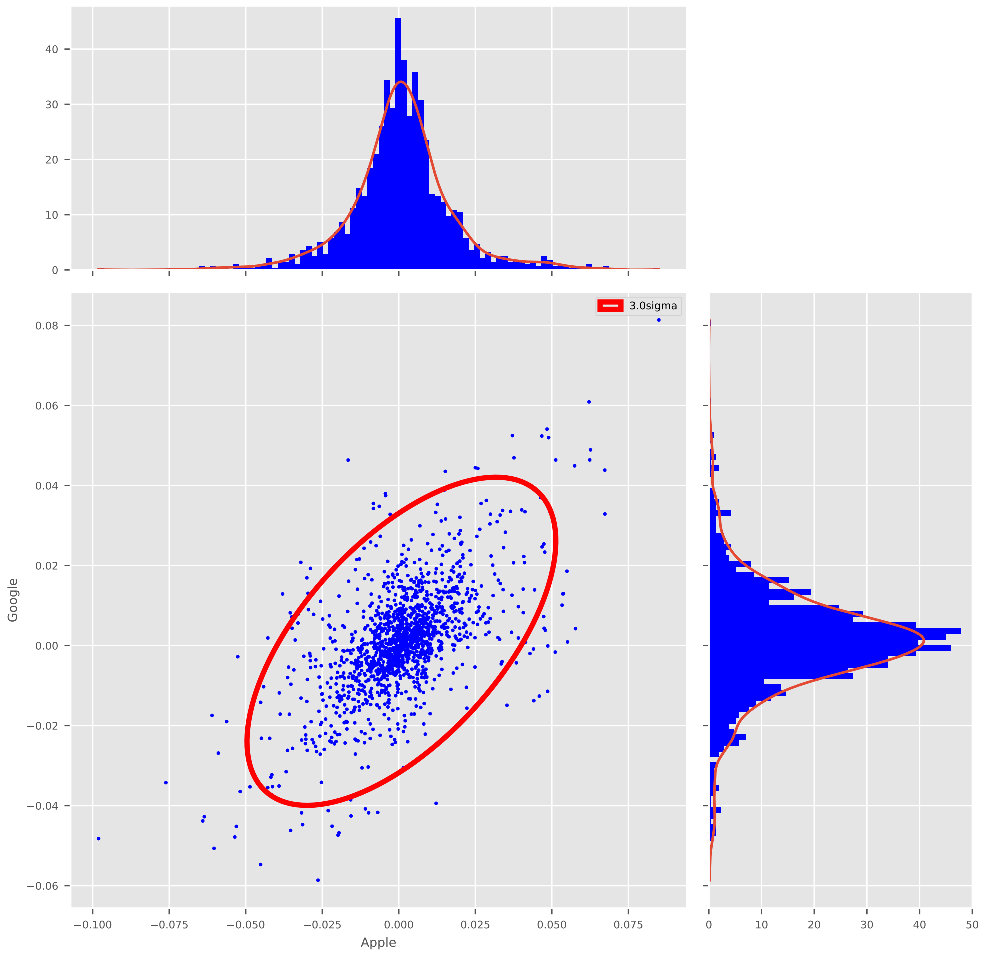


Figure 7.4: Generated log-return distribution for Apple / Google

Table 7.4: Correlation matrix of historical data

	AAPL	AMZN	GOOGL
AAPL	1.0000000	0.6238220	0.6559852
AMZN	0.6238220	1.0000000	0.6932071
GOOGL	0.6559852	0.6932071	1.0000000

Table 7.5: Correlation matrix of generated data

	AAPL	AMZN	GOOGL
AAPL	1.0000000	0.5747237	0.6116212
AMZN	0.5747237	1.0000000	0.6532851
GOOGL	0.6116212	0.6532851	1.0000000

Using the generated distribution, we then reconstruct trajectories from January 1, 2021, to December 31, 2021, as described in section 7.1.4. We illustrate the output of ten generated trajectories for Google, in Figure 7.5, and plot also the historical Google charts to compare with.

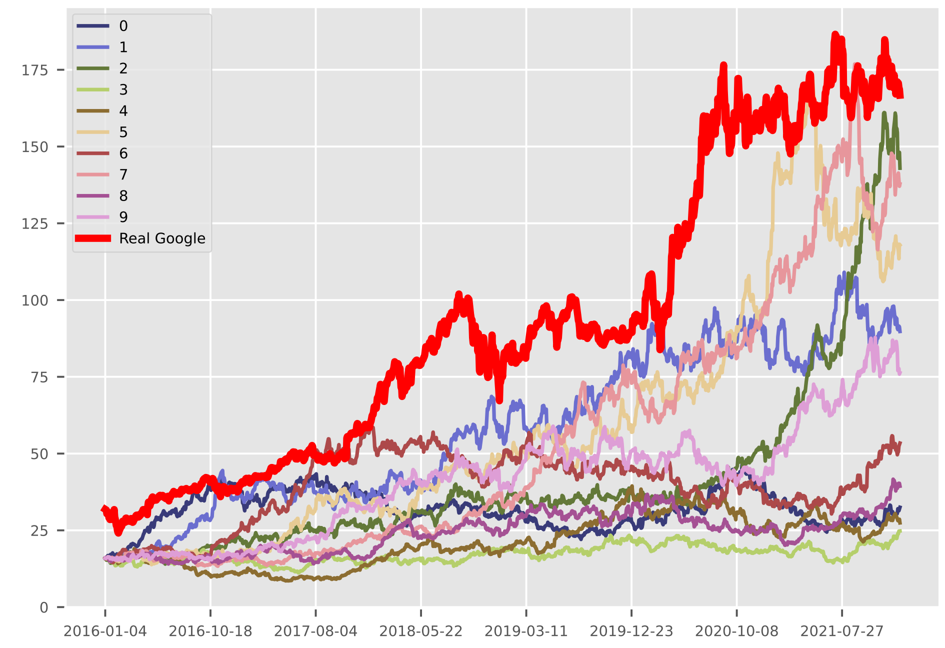


Figure 7.5: Generated Google paths

7.2.3 Recurrent kernels illustration

The recurrent method (7.1.5) allows to draw one trajectory, that can be considered as a iid realization of the temporal series, based on the knowledge of its history. Figure 7.6 shows a toy example of historical temporal series forecast, having two components : the Bitcoin price and the hash-rate values, for which we considered T_X covering daily observations from 01/01/2015 to

23/11/2020, since H and P are set to fit 6 months datas. Hence the settings to produce Figure 7.6 correspond to $N_X = 1, D = 2, T_X = 1460$ in (**??**). Starting from this setting, we predict the temporal series up to 31/12/2021 and compare it with the historically observed one, using a kernel implementation of the scheme (7.1.5).

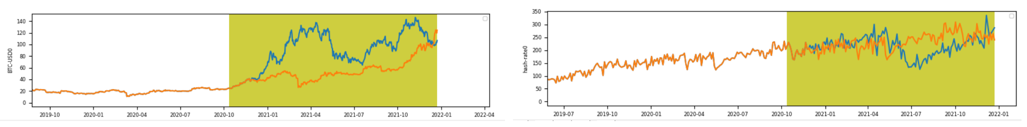


Figure 7.6: Recurrent kernels : Generated (yellow) BTC-USD left / HR right, versus historical (blue).

This method has a lot of forecasting applications, useful for professional purposes. However, in the context of time series forecasts, such a method faces a number of challenges. First, we are left with two extra parameters, H and P. Secondly, 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. Finally, we don't have any argument to ensure the stability of the recurrent scheme 7.1.5.

7.3 Monte Carlo pricing

We show that synthetic data generated in section 7.1.4, illustrated in Figure 7.5, can be used for Monte-Carlo based pricing [2], as they share close statistical properties with simulated paths from known processes.

7.3.1 Experiment settings

Consider a bivariate geometric Brownian motion (gBm) with initial values $X_0 = [100, 120]$:

$$dX_t = \sigma X_t \, dW_t, \tag{7.3.1}$$

where W_t is a two-dimensional Brownian motion with given correlation 0.5 and the volatilities are $\sigma = [0.1, 0.2]$. Consider a standard basket option's payoff as the following function

$$P(x) = \max(x \cdot \omega - K, 0) \tag{7.3.2}$$

where x are the input market data, $\omega = [0.5, 0.5]$ are weights, $K = 108$ is called the option's strike.

We define the reference value for this test as $\mathbb{E}(P(X_T)|X_0)$, the maturity T is set to one year $(1Y)$, which can be computed using Monte-Carlo methods.

7.3.1.1 Reproducing a Bivariate Gaussian

We simulate a trajectory $X \in \mathbb{R}^{1,2,1000}$ of a gBm (7.3.1) path. Following the section 7.1.4, we compute the log-normal returns, which are random samples of a bivariate normal distribution. We then use (7.1.10) to produce 1000 random samples, $Z = G^{\mathbb{X}_x} \in \mathbb{R}^{1000,2}$. Table 7.6 shows that both distributions $\mathbb{X}_x, \mathbb{Z}_z$ match.

[2]An alternative pricing method here could be to consider the Kolmogorov, or Black-Scholes, partial differential equations, similar to a Cox tree, but for any number of underlying assets, as already presented in [30] and ref. therein.

Table 7.6: Stats for BGM

	ASSET0	ASSET1
Mean	-1.9e-04 (-4.4e-04)	7.1e-04 (8.0e-04)
Variance	2.8e-05 (3.4e-05)	1.2e-04 (1.3e-04)
Skewness	-1.1e-01 (-1.3e-01)	-2.1e-01 (-2.0e-01)
Kurtosis	-2.4e-01 (-4.3e-01)	-4.2e-02 (-3.3e-01)
KS test	0.24 (0.05)	0.144 (0.05)

7.3.1.2 Basket option pricing

In this paragraph we study numerically the convergence properties of our approach as the size of the training set, that is the size of the observed gBm samples, increases. We consider as input two sequences of observed gBm X_1^n, X_2^n, having varying size $n = 100, \dots, 100 \times N$, with $N = 10$. The sets X_1^n (resp. X_2^n) correspond to a randomly sampled gBm (resp. sharp-discrepancy sequences), as in section **??**.

Our approach follows then the steps described in section 7.1.4 : we generate IID samples $Z_1^n \in \mathbb{R}^{10000,2}$ (resp. SDS sample $Z_2^n \in \mathbb{R}^{10000,2}$), and we evaluate the option's price at the maturity time T, using $\frac{1}{N_z}\sum_{n=0}^{N_z} P(z^n)$ in order to approximate the option's reference value. The results are plot in Figure 7.7.

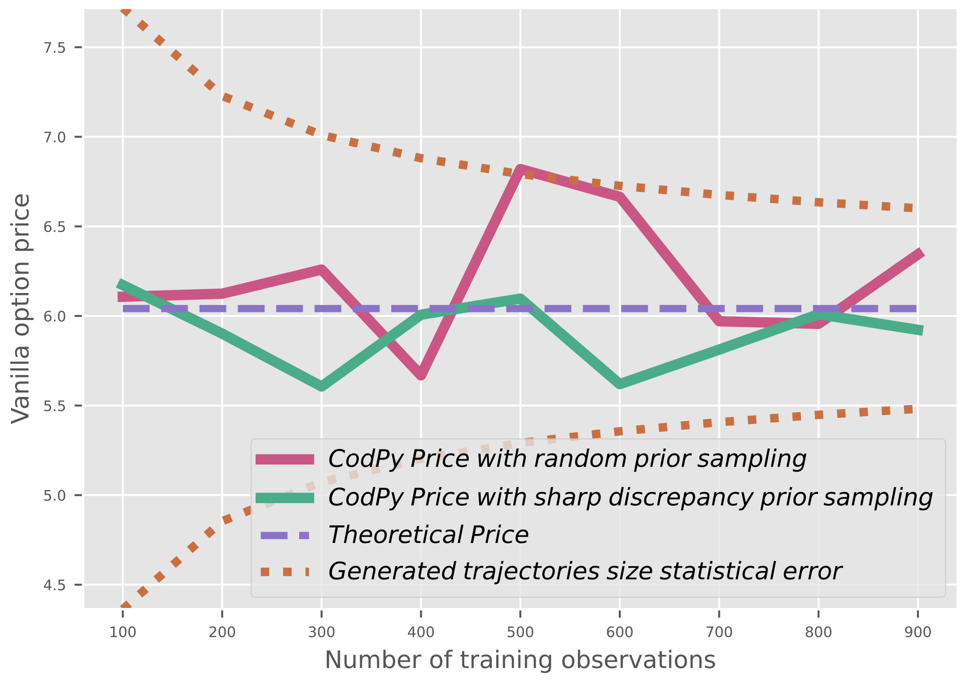


Figure 7.7: Convergence pattern

The blue dashed lines in Figure 7.7 is the reference price computed using a Monte Carlo method, the red (resp. green) lines correspond to the prices computed using a random generated sample Z_1^n (resp. SDS Z_2^n). Note that the results are within the Monte-Carlo's statistical error, that are the dotted line in the figure.

7.4 P&L explanation

7.4.1 Experiment settings

We illustrate our approach with an application of real time P&L explanation for large multi-asset portfolios, which we outline here for a better understanding in the one-dimensional case.

Consider a function $P(t,x) \in \mathbb{R}^{D_P}$, $x \in \mathbb{R}^D$, corresponding to an external engine pricing a portfolio of D_P instruments, assuming that the risk sources values are x at time t. Pricing engines are often computationally intensive and can hardly be used in real-time. This experiment proposes a quicker alternative using nightly batches. For illustrative purposes, we consider a pricing engine taken as a Black-Scholes formula with predefined values, see below.

Consider a historical market data set, for this test consisting of 253 closing values, denoted $x^{-252}, \ldots, x^0$, for the S&P500, during the period of time t^{-252} = June 1, 2021 and t^0 = June 1, 2022, retrieved from Yahoo Finance. Thus, considering (7.1.11), this set is described by a tensor with $N_x = 1, D = 1, T_x = 253$.

We use the historical data set to produce synthetic data at a future horizon date $t^1 = t^0 + 4$ days, following section 7.1.4, simulating night-batch computed Value-at-Risk (VaR) scenario. We also produce similarly a test set $Z \in \mathbb{R}^{N_z}$ with the same method, corresponding to simulated, real time data at date t^1.

To benchmark our approach, we compute the P&L on the test set Z using three methods:

- Analytical P&L : it is computed as $P(t^1, Z) - P(t^0, x^0)$, and is the reference values for our tests.
- Predicted P&L: the price function $P(t^1, Z)$ is approximated using the formula (7.1.6), as $\mathcal{P}_k(X, X, Z)P(t^1, X) - P(t^0, x^0)$.
- Taylor approximation: the price function $P(t^1, Z)$ is computed using a second order Taylor formula approximation around $P(t^0, x^0)$. [3]

7.4.2 Training set

According to (7.1.7), the interpolation error committed by the projection operator P_k (7.1.6), defined on a set X, is driven at any point z by the quantity $d_k(z, X)$. We plot at Figure 7.8 the isocontours of this error function for two distinct sets.

- (a) X is generated as VaR scenarios for the three dates t^{-1}, t^0, t^1.
- (b) X is the historical data set.

The blue dots in Figure 7.8 are the test set Z, and corresponds to simulated, intraday, market values.

It is clear from this picture that the interpolation error is smaller if we consider the VaR scenario dataset on the left-hand side. Indeed, since banks must produce VaR data for regulatory constraints, such data are available, and we considered them as training set in this paper to extrapolate the P&L. We could use only the historical data set, at the expense of less accurate results. Note that this situation might be of interest, if only historical data are available.

Notice finally that there are three sets of red points at Figure 7.8-(a), as we considered VaR scenarios at three different times t^{-1}, t^0, t^1 since we are interested in approximating time derivatives for risk management, as the theta $\partial_t P$, see section below.

[3]We compare to a Taylor approximation, as this method is currently used by some banks to estimate their P&L on a real time basis.

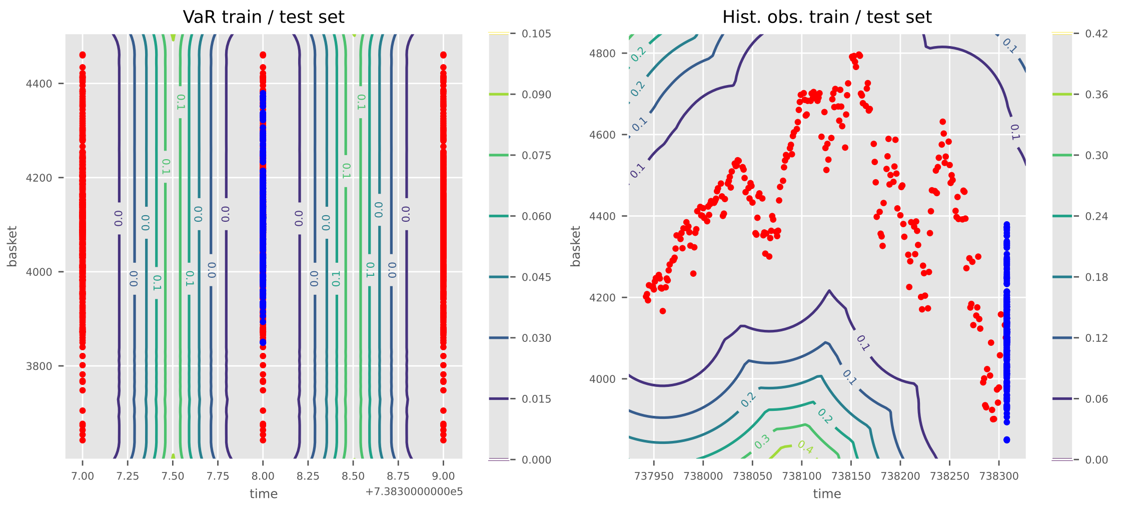


Figure 7.8: Training and test set

7.4.3 P&L explanation of S&P 500 options

To benchmark our results, we considered the following values: the S&P value is $x^0 = 4101$, as of date t^0=June 1, 2022. The pricing engine is taken as a Black-Scholes formula $C(x, K, r, T, \sigma)$, with strike $K = 4050$, which is near to the spot's value, volatility $\sigma = 25\%$, and without risk-free interest rate ($r = 0$). We considered two maturities: a short one $T = 10$ days, and a longer one $T = 365$ days.

European option P&L.

We plot the results of three methods on the test set Z (exact P&L, our approximation, Taylor approximation) in Figure 7.9 for two maturities.

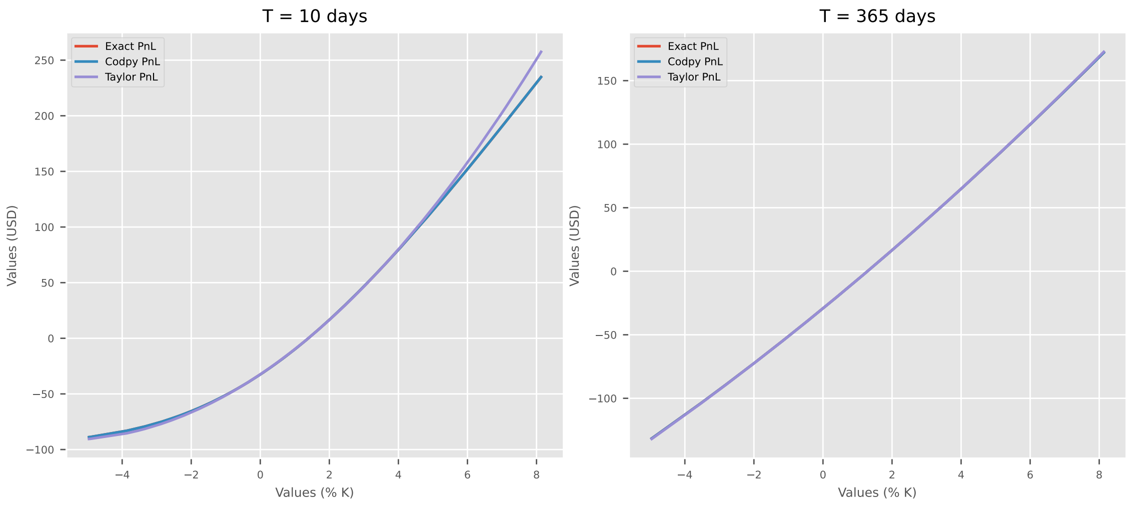


Figure 7.9: The PnL output for 10 and 365 days maturity

We notice that the output values are accurate especially for a short-term maturity where the predicted P&L is more precise than the P&L computed using the Taylor approximation especially deep in the money (DIM) and deep out of the money (DOM). Indeed, this method is more competitive than a Taylor approximation as the pricing function becomes nonlinear.

Table 7.7 shows the values of error between analytical P&L and predicted P&L and also between analytical P&L and Taylor approximation for different maturity scenarios, where the computed error is the relative mean squared error (RMSE) expressed in percentage

$$RMSE(f,g) = \frac{\|f-g\|_{\ell^2}}{\|f\|_{\ell^2} + \|g\|_{\ell^2}} \tag{7.4.1}$$

Table 7.7: PnL error in percentage

Maturity	T = 10 days	T = 365 days
Codpy error	0.00%	0.02%
Taylor error	2.60%	0.10%

European greeks.

Using the differential operators (see (7.1.8)), we approximate the first and second order derivatives of $P(t^1, Z)$, called greeks. The Figure 7.10 plots $\partial_x P$ (called Delta), $\partial_t P$ (theta), and the Figure 7.11 plots the second order derivatives $\partial_x^2 P$ (gamma), $\partial_t \partial_x P$. For the delta, we added the linear, Taylor approximation (the green line). We notice that the results of our machine learning technique are accurate, especially for DIM and DOM where Taylor approximation tends to diverge.

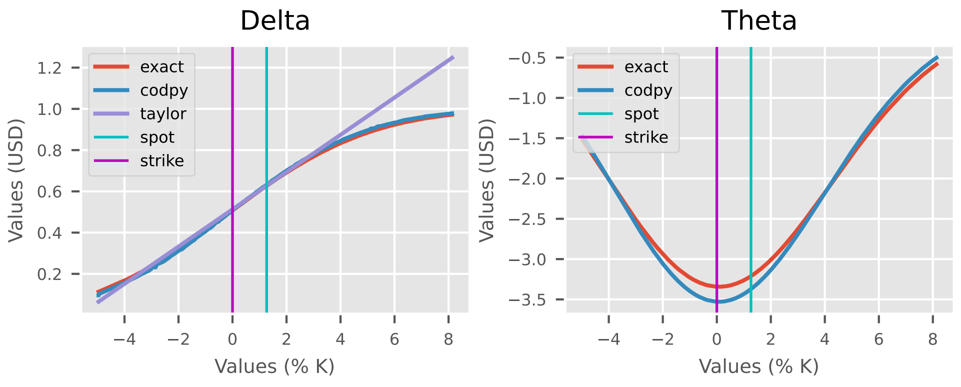


Figure 7.10: First order greeks - T=3M

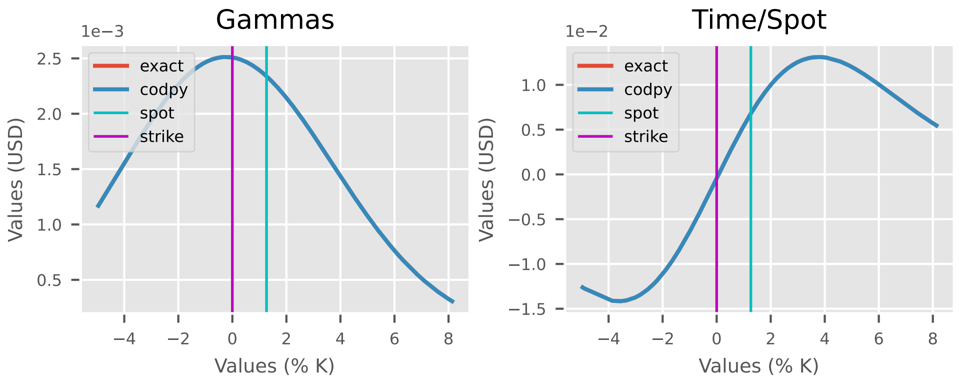


Figure 7.11: Second order greeks - T=3M

7.5 The Bachelier problem

Problem description. This section provides a benchmark of the methods (4.3) approximating the conditional expectation (4.3.1) for the Bachelier problem, which we describe now. 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, which is a function denoted $f(x) = \max(b(x) - K, 0)$, 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.

With these settings, the conditional expectation (4.3.1) can be determined using a closed formula, providing the reference value

$$f(x) = \theta\sqrt{t^2 - t^1} pdf(d) + (b(x) - K)cdf(d), \qquad d(x, K) = \frac{b(x) - K}{\theta\sqrt{t^2 - t^1}},$$

PDF (resp. CDF) holding for the probability density function (resp. cumulative) of the normal law.

7.5.1 Methodology and input/output data

We test different numerical methods implementing (4.3), with the following inputs:

- $X \in \mathbb{R}^{N_X \times D}$, is given by iid samples of the Brownian motion $X(t^1)$ at time $t^1 = 1$. The reference values are $f(Z|X) \in \mathbb{R}^{N_X \times 1}$, computed using (7.5).
- $Z \in \mathbb{R}^{N_Z \times D}$ is an iid realization of the Brownian motion $X(t^2)|X(t^1)$ at time $t^2 = 2$, since $f(Z) \in \mathbb{R}^{N_Z \times 1}$ are the functions values.

For each method, the output are $f_{Z|X} \in \mathbb{R}^{N_Z \times D_f}$ approximating (4.3.1), hence are compared to $f(Z|X)$ in our experiments. We plot the generated learning and test set in picture 7.12, comparing the observed variable f_Z and the reference values $f(Z|X)$. Thus the problem can be stated as : knowing the noisy data in the left-hand side, deduce the one at right.

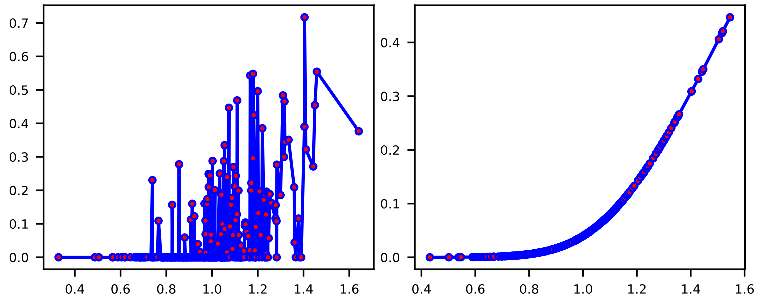


Figure 7.12: Bachelier problem. Left training set $b(Z), f(X)$, right test set $b(X), f(Z|X)$.

7.5.2 Four methods to tackle the Bachelier problem

We compare four methods for the Bachelier problem. Two methods are based on a standard approach, that uses predictive machines of the form (2.1.1), in order to approximate a conditional expectation (4.3) as

$$f_{Z|X} = \mathcal{P}_m(Z, Y, X, f(Z))$$

The first machine m is a neural network method, the second is a kernel one, labeled ANN and CodPy pred in the figures. The third machine solves (**??**), labeled Pi:iid in the figures. The fourth provides a similar approach, but picks up X (resp. Z) as the sharp discrepancy sequences (SDS) of $X(t^1)$ (resp. $X(t^2)$) and is labeled Pi:sharp in our figures.

To illustrate a typical benchmark run of one of these four methods, Figure 7.13 shows the predicted values $f_{Z|X}$ against the exact ones $f(Z|X)$, as functions of the basket values $b(Z)$, for the last method (SDS). We show five runs of the method with $N_X = N_Z = 32, 64, 128, 256, 512$.

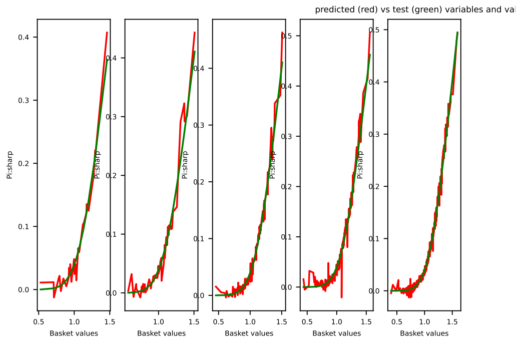


Figure 7.13: Exact and predicted values for sharp discrepancy sequences

Figure 7.14 presents a benchmark for scores, computed accordingly to the RMSE % (2.3.2) (lower is better), for the two dimensional case $D = 2$, however the results are similar whatever the dimensions are.

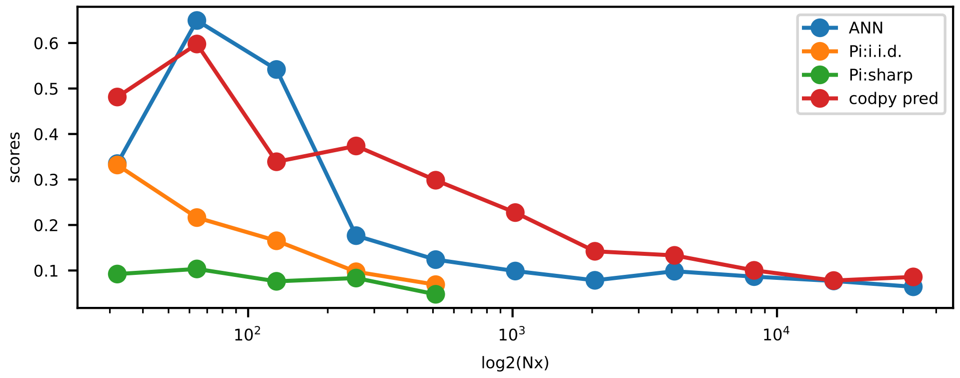


Figure 7.14: Benchmark of scores

7.5.3 Concluding remarks

We emphasize that the axis in Figure 7.14 is in log-scale of the size of the training N_x. This test shows numerically that both predictive methods based on (2.1.1) are not converging. The method Pi:iid (in yellow color) shows a performance profile which has a convergence pattern at the statistical rate $\frac{1}{\sqrt{N_X}}$, that is, the one expected with randomly sampled data. The method Pi:sharp (in green color) is an illustration of performance gains when using the proposed sharp discrepancy sequences.

Chapter 8

Application to partial differential equations

8.1 Automatic algorithmic differentiation.

AAD is a family of techniques for algorithmically computing **exact** derivatives of compositions of differentiable functions. It is a useful tool for several applications in this book, hence we describe it succintly in this section.

Techniques for AAD have been known since at least the 1950s. There are two main variants of AAD: reverse-mode and forward-mode. Reverse-mode AAD computes the derivative of a composition of atomic differentiable functions by computing the sensitivity of an output with respect to the intermediate variables (without materializing the matrices for the intermediate derivatives). In this way, reverse-mode can efficiently compute the derivatives of scalar-valued functions. Forward-mode AAD computes the derivative by calculating the sensitivity of the intermediate variables with respect to an input variable [17].

There are number of high quality implementations of AAD in the libraries, such as[1] TensorFlow , PyTorch, autograd, Zygote and JAX. The JAX supports both reverse-mode and forward-mode AAD.

Codpy provides a simple interface to the Pytorch AAD differentiation framework. Figure 8.1 illustrates the computations of first and second derivatives of a function $f(X) = \frac{1}{6}X^3$ using AAD.

8.2 Differential machines benchmarks

AAD is a natural tool to define a differential machine starting from any predictive machine (2.1.1). In this section, we illustrate a general multi-dimensional benchmark of two differential machines methods. The first one uses the kernel gradient operator (see (3.4.3)). The second one uses a neural network defined with Pytorch together with AAD tools.

An example of one-dimensional testing is shown at figure 8.2, using the same benchmark methodology as in chapter 2. The first row is quite similar to our one-dimensional test. The second row provides also four plots: the first one is the exact gradient of the considered function on the test set, computed using AAD. The second one plot the kernel gradient operator. The two remaining ones plot two different run of the neural network differential machine.

[1]TensorFlow url, PyTorch url, autograd url, Zygote url, JAX url

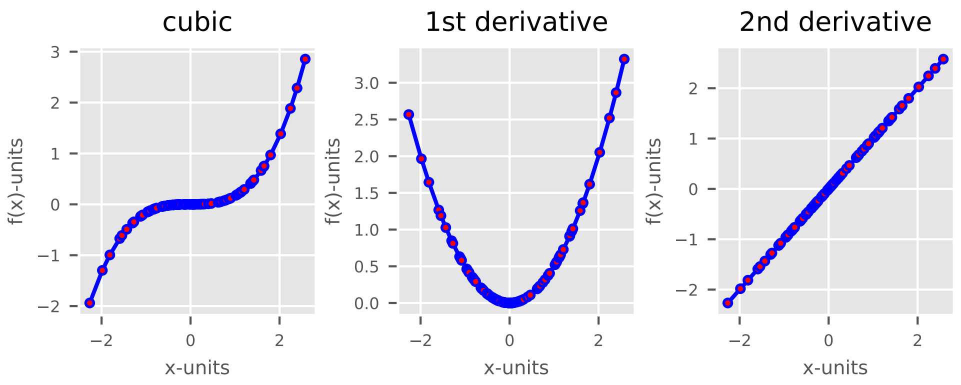


Figure 8.1: A cubic function, exact AAD first order and second order derivatives

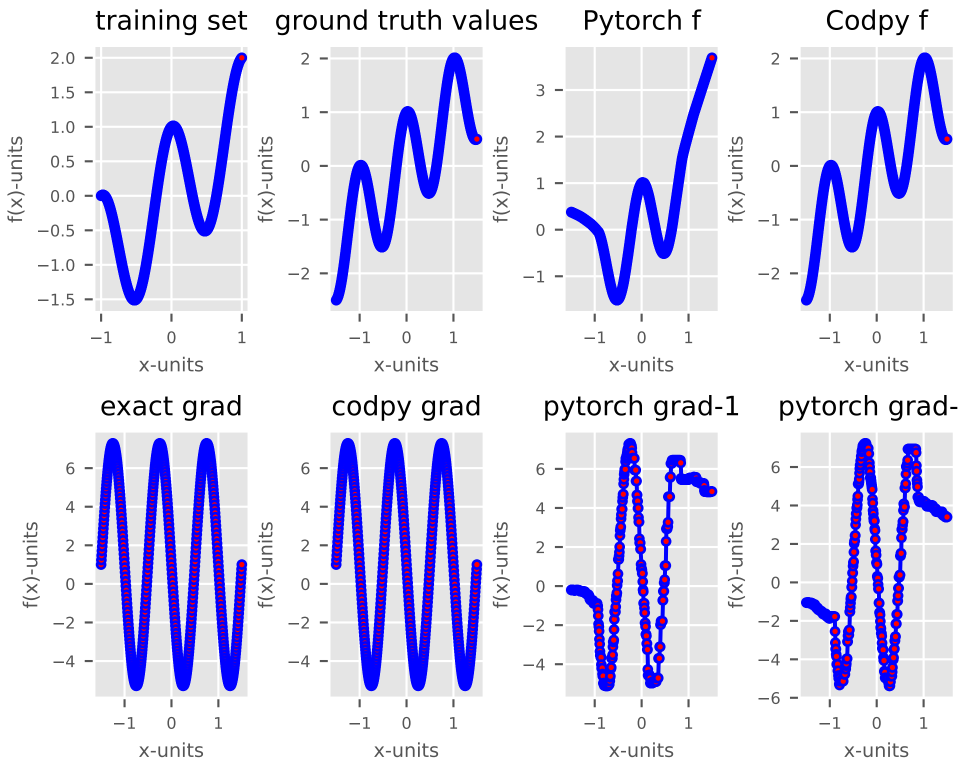


Figure 8.2: A benchmark of one-dimensional differential machines

The same benchmark can be used in any dimension, and we plot the two dimensional test at figure 8.3

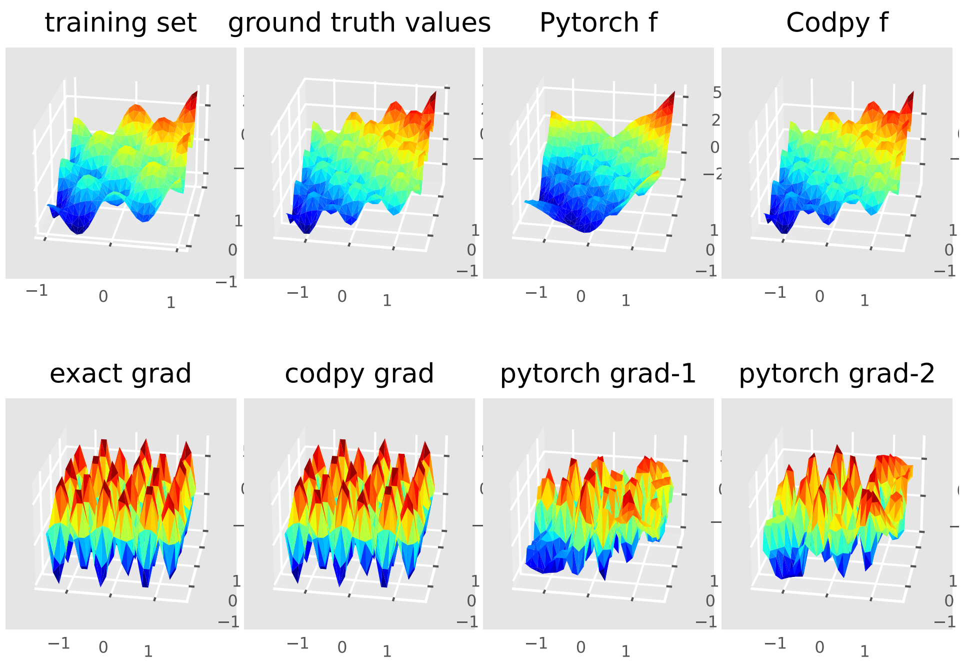


Figure 8.3: A benchmark of two-dimensional differential machines

As noticed in these figures

- Two runs of AAD computations leads to two different results (pytorch-grad1 and 2) : NNs do not define deterministic differential learning machines, due to the stochastic descent algorithm, here Adam optimizer.
- Differential neural networks tends to be less accurate than a kernel-based gradient operator.

8.3 Taylor expansions using differential learning machines

Taylor expansions using differential learning machines are common for several applications, hence we propose a general function to compute them, that we describe in this section. We start to make a reminder of Taylor expansions.

Let us consider a smooth, vector-valued function f, defined over $\mathbb{R}^D$. Considering any sequences of points Z, X having the same length, the following formula is called a Taylor's expansion of order p:

$$f(Z) = f(X) + (Z-X)\cdot(\nabla f)(X) + \frac{1}{2}\Big((Z-X)(Z-X)^T\Big)\cdot(\nabla^2 f)(X) + ... + |Z-X|^p \epsilon(f)$$

where :

- $(z-x) := \Big(z_i - x_i\Big)_{i,j=0..D}$ is a D-dimensional vector.

- $(z-x)(z-x)^T := \Big((z_i - x_i)(z_j - x_j)\Big)_{i,j=0..D}$ is a $D \times D$ matrix.
- $a \cdot b$ denotes the usual Frobenius inner product.
- ∇f,$\nabla^2 f$ holds for the gradient (D-dimensional vector) and the Hessian ($D \times D$ matrix).
- $|z-x|$ is the standard Euclidean distance, $\epsilon(f)$ is a function depending on f and its derivatives that we do not detail here. The term $|Z-X|^3\epsilon(f)$ represents the error committed by this approximation formula.

In this section, we study Taylor formulas using differential learning machines to approximate the derivatives, that is approximating $\nabla f(x)$,$\nabla^2 f(x)$ with

$$\nabla f_x = \nabla_Z \mathcal{P}_m(X, Y, Z = x, f(X)), \nabla^2 f_x = \nabla^2_Z \mathcal{P}_m(X, Y, Z = x, f(X)).$$

following the previous section, we performed a benchmark of a second-order Taylor formula using three approaches

- The first one is the reference value for this test. It uses the AAD to compute both $\nabla f_x, \nabla^2 f_x$.
- The second one, uses a neural network defined with Pytorch together with AAD tools.
- The third one uses the hessian operator from codpy.

The test is genuinely multi-dimensional. We illustrate the results starting from the one-dimensional case in figure 8.4 and Figure **??** illustrate the two dimensional case.

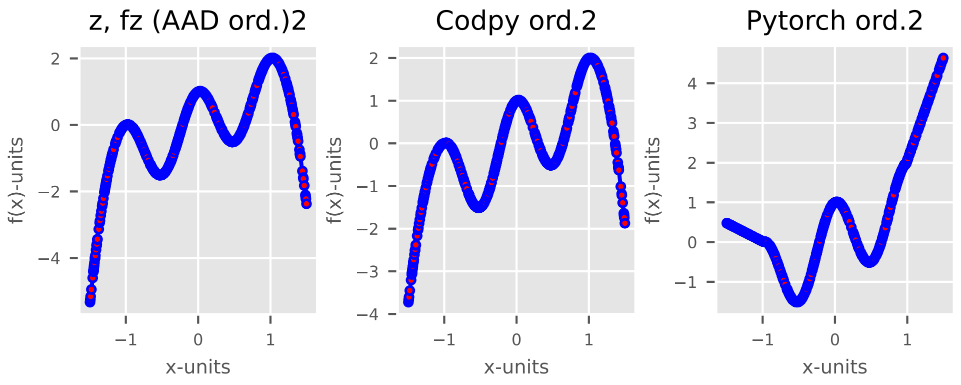


Figure 8.4: A benchmark of one-dimensional learning machine second-order Taylor expansion

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] Eckerli, Florian and Osterrieder, Joerg, Generative Adversarial Networks in finance: an overview, *Comput. Methods Appl. Mech. Engrg.* http://dx.doi.org/10.48550/ARXIV.2106.06364 (2021).

[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] O. Teymur, J. Gorham, M. Riabiz, and C.J. Oates Proc. 24th Int. Conf. on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA, Volume 130, pp. 1027–1035.

[14] A. Gretton, K.M. Borgwardt, M. Rasch, B. Schölkopf, and A.J. Smola, A kernel method for the two sample problems, Proc. 19th Int. Conf. on Neural Information Processing Systems, 2006, pp. 513–520. arXiv:0805.2368

[15] Bernhard Schölkopf, Ralf Herbrich, and Alexander J. Smola. A generalized representer theorem. In Computational learning theory, 416–426, Springer, 2001.

[16] F.C. Günther and W.K. Liu, Implementation of boundary conditions for mesh-less methods, Comput. Methods Appl. Mech. Engrg. 163 (1998), 205–230.

[17] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, 2008.

[18] 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

[19] T. Hofmann, B. Schölkopf, and A. J. Smola, Kernel methods in machine learning, Ann. Statist. 36 (2008), 1171–1220.

[20] B.N. Huge and A. Savine, Differential machine learning, unpublished report, January 2020, available at https://ssrn.com/abstract=3591734

[21] 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

[22] T.F. Korzeniowski and K. Weinberg, A multi-level method for data-driven finite element computations, Comput. Methods Appl. Mech. Engrg. 379 (2021), 113740.

[23] J.J. Koester and J.-S. Chen, Conforming window functions for mesh-free methods, *Comm. Numer. Methods Engrg.* 347 (2019), 588–621.

[24] Y. LeCun, C. Cortes, and C.J.C. Burges, The MNIST database of handwritten digits, http://yann.lecun.com/exdb/mnist/

[25] R. McCann, Polar factorization of maps on Riemannian manifolds, Geom. Funct. Anal. 11 (2001), 589–608.

[26] J.-M. Mercier, Optimally Transported schemes. Applications to Mathematical Finance, unpublished, https://www.researchgate.net/publication/228689632_Optimally_Transported_schemes_Applications_to_Mathematical_Finance

[27] J.-M. Mercier, A High-Dimensional Pricing Framework for Financial Instruments Valuation, DOI:10.2139/ssrn.2432019

[28] P.G. LeFloch and J.-M. Mercier, Revisiting the method of characteristics via a convex hull algorithm, J. Comput. Phys. 298 (2015), 95–112.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: a tutorial, January 2021, available at ssrn.com/abstract=3769804.

[34] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: an advanced tutorial, January 2021, available at ssrn.com/abstract=3769804.

[35] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: a kernel-based reordering algorithm, January 2021, available at ssrn.com/abstract=3770557.

[36] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: RKHS-based polar factorization and sampling algorithm, in preparation.

[37] P.G. LeFloch, J.M. Mercier, and Sh. Miryusupov, CodPy: RKHS-based algorithms and conditional expectations, in preparation.

[38] P.G. LeFloch, J.-M. Mercier, and S. Miryusupov, CodPy: Support Vector Machines (SVM) for (reverse) stress tests in finance, in preparation.

[39] S.F. Li and W.K. Liu, *Mesh-free particle methods,* Springer Verlag, Berlin, 2004.

[40] G.R. Liu, *Mesh-free methods: moving beyond the finite element method,* CRC Press, Boca Raton, FL, 2003.

[41] G.R. Liu, An overview on mesh-free methods for computational solid mechanics, *Int. J. Comp. Methods* 13 (2016), 1630001.

[42] 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.

[43] Y. Nakano, Convergence of mesh-free collocation methods for fully nonlinear parabolic equations, Numer. Math. 136 (2017), 703–723.

[44] 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.

[45] H. Niederreiter, *Random number generation and quasi-Monte Carlo methods,* CBMS-NSF Regional Conf. Series in Applied Math., Soc. Industr. Applied Math., 1992.

[46] 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.

[47] R. Opfer, Multiscale kernels, Adv. Comput. Math. 25 (2006), 357–380.

[48] R. Salehi and M. Dehghan, A moving least square reproducing polynomial mesh-less method, Appl. Numer. Math. 69 (2013), 34–58.

[49] 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.

[50] R. Sinkhorn and P. Knopp, Concerning nonnegative matrices and doubly stochastic matrices, Pacific J. Math. 21 (1967), 343–348.

[51] B.K. Sriperumbudur, A. Gretton, K. Fukumizu, B. Scholkopf, and G.R. Lanckriet, Hilbert space embeddings and metrics on probability measures, J. Mach. Learn. Res. 11 (2010), 1517–1561.

[52] J. Sirignano and K. Spiliopoulos, DGM: a deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018), 1339–1364.

[53] I.M. Sobol, Distribution of points in a cube and approximate evaluation of integrals, U.S.S.R Comput. Maths. Math. Phys. 7 (1967), 86–112.

[54] P. Traccucci, L. Dumontier, G. Garchery, B. Jacot, A Triptych Approach for Reverse Stress Testing of Complex Portfolios. arXiv:1906.11186

[55] R.S. Varga, *Matrix iterative analysis,* Springer Verlag, 2000.

[56] C. Villani, *Optimal transport, old and new,* Springer Verlag, 2009.

[57] 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.

[58] H. Wendland, *Scattered data approximation,* Cambridge Monograph, Applied Comput. Math., Cambridge University, 2005.

[59] J.X. Zhou and M.E. Li, Solving phase field equations using a mesh-less method, Comm. Numer. Methods Engrg. 22 (2006), 1109–1115.

[60] B. Zwicknagl, Power series kernels, Constructive Approx. 29 (2008), 61–84.