École Polytechnique
Promotion X2012
Dubois Sébastien

École polytechnique - Massachusetts Institute of Technology

Rapport de Stage de Recherche

Deep Mining : Copula-based Hyper-Parameter Optimization for Machine Learning Pipelines

- Non Confidentiel -

Département : Informatique
Champ : INF591 Informatique
Directeur de l'option : Olivier Bournez
Maître de stage : Kalyan Veeramachaneni
Dates du stage : 30/03/2015 - 17/07/2015
Nom de l'organisme : Massachusetts Institute of Technology - CSAIL
32 Vassar Street, Cambridge, MA, USA

Déclaration d'intégrité relative au plagiat

Je soussigné Dubois Sébastien certifie sur l'honneur :

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport.
3. Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les référencer selon les règles bibliographiques préconisées.

Je déclare que ce travail ne peut être suspecté de plagiat.

Date: 08/08/2015 Signature: [illegible]

Deep Mining : Copula-based Hyper-Parameter Optimization for Machine Learning Pipelines

by
Sébastien Dubois

Abstract

Every machine learning model has several hyper-parameters that need to be carefully chosen for they can hugely impact its quality. While grid search was the first automatic approach to tackle this problem, random search has proved to be much faster for such tasks. Recently some sequential models have been proposed in order to leverage the information acquired during the search process. We build our work upon such a technique which models the performances yielded by the hyper-parameters with a Gaussian Process (GP).
First, we present a novel non-parametric approach for Gaussian Copula Processes (nGCP), and use it for hyper-parameter optimization. In addition, we present a framework to auto-tune machine learning pipelines (MLP). Specifically, we consider noisy performance evaluations which speeds up the search and paves the way for parallel designs.
We finally demonstrate that nGCP outperforms GP for regression. We also test nGCP-based hyper-parameter optimization techniques on two classic problems involving text data and images. We show that even with noisy estimations our techniques outperform random search, and that nGCP-based methods are a bit faster than GP-based ones.

Résumé

Tout modèle de *Machine Learning* (Apprentissage Automatisé) possède des hyper-paramètres qui doivent être choisis avec soin, car ceux-ci peuvent nettement en impacter la qualité. Après la méthode du *grid search* (recherche sur une grille) et de la recherche aléatoire, déjà plus efficace, des modèles séquentiels ont été proposés afin de tirer profit de l'information acquise durant l'exploration d'hyper-paramètres. Le travail présenté dans ce rapport se fonde sur une telle méthode, qui modélise les performances des hyper-paramètres par processus gaussien (GP).
Nous présentons d'abord une nouvelle approche non-paramétrique du *Gaussian Copula Process* (processus de copule gaussienne, nGCP), et l'utilisons dans le cadre de l'optimisation d'hyper-paramètres. Nous présentons également une méthode pour ajuster automatiquement les hyper-paramètres d'un modèle complet de *Machine Learning.* Plus précisément, nous considérons des évaluations bruitées de performance, ce qui accélère l'exploration et facilite la transition vers un système parallélisé.
Enfin, nous démontrons que les nGCP sont plus performants que les GP en régression. Nous avons également testé l'optimisation d'hyper-paramètres par nGCP sur deux problèmes classiques impliquant des données textuelles et des images. Nous montrons ainsi que ces méthodes séquentielles, même avec des observations bruitées, sont largement plus performantes que la recherche aléatoire, et qu'elles sont légèrement plus rapides lorsqu'elles reposent sur un nGCP plutôt qu'un GP.

Acknowledgements

I would like to acknowledge my supervisor Kalyan Veeramachaneni for the wonderful time I had during my visit in his lab, the Alfa Group. He proposed this great subject to me and always let me focus on what I wanted to study. I learned a lot during these four months and Kalyan was always there to give helpful advice.

I would also like to thank Sebastien Boyer for interesting and helpful discussions about our projects, but also for introducing me to the MIT and Cambridge.

Finally, thanks to Max Kanter for using my software in his project and for providing useful feedback.

And to all Alfa labmates for the great time we had together.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Machine Learning Pipeline

The field of Machine Learning lies at the intersection of Computer Science and Statistics and studies how *machines* can acquire knowledge, *ie.* provide insights from some phenomenon and predict other phenomena based on its experience. According to Tom Mitchell in [1], Computer Science aims at building machines that can solve problems while Statistics aim at inferring from data. Finally he described the Machine Learning as the field which tries to answer the following question: "How can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes ?"

The applications of machine learning are infinite but in practice we can often observe a general *pipeline*:

- Aggregate real-world data
- Extract features from the data
- Design a predictive model
- Predict, classify, evaluate ...

While real-world data can be very different in nature (text, images, databases, ...), statistical models usually need real vectors as input. This conversion step is called *feature extraction* or *feature engineering* and basically aims at representing the original data as vectors in $\mathbb{R}^n$. Of course this step should be done carefully as it will considerably impact the ability to build a powerful statistical model.

Designing a predictive model consists in choosing a model and *fitting* it to the data. This is generally done by minimizing a cost function. For example, linear Support Vector Machines (SVM) is a model that tries to best separate the data with hyperplanes, and fitting an SVM means defining the hyperplanes (see Figure 1-1). Once this is done, one can for example predict the class of a new

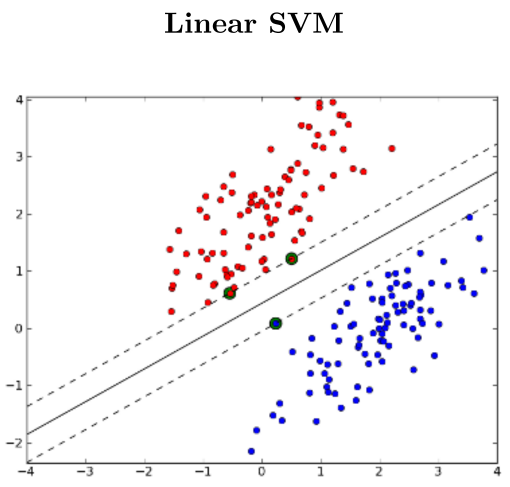

Figure 1-1: Two-class linear SVM in a two-dimensional space.

point in the input space given its position with regards to the hyperplanes.

So the main work of a data scientist, when s/he has the data and a problem to tackle, is to design such a pipeline, which means find out the best features to extract and which classification model to use. However such models actually have several parameters that need to be set: these are called *hyper-parameters* to distinguish them from the parameters that are automatically set during the fitting process. We can even think as each model (*eg.* choosing a linear SVM against Random Forests) as a categorical hyper-parameter. These hyper-parameters can have a huge impact on the prediction accuracy and are usually set by an expert hand after a lot of testing.

Figure 1-2 shows an example of a pipeline and some hyper-parameters that can be tuned. This refers to a classic machine learning problem of handwritten digits recognition using the MNIST dataset. So here the real-world data are images of handwritten digits and the aim is to predict which digit is written given the image. We propose a pipeline in which we first blur the data, reduce the dimension by Principal Component Analysis (PCA), and finally classify through a polynomial SVM (see Appendix A for details on these techniques).

1.2 Motivation

We cannot emphasize enough that tuning a machine learning pipeline is a difficult and time-consuming task. Most of the time the dependencies between the hyper-parameters and the pipeline's performances are hard to understand and thus there is high chance the user does not take the most benefits of its pipeline. A solution could be to test all hyper-parameter combinations possible. This is called *grid search*: discrete values are set for each categorical/integer/floating point hyper-parameter, thus making a grid over the space of hyper-parameter sets, and each point of this grid is tested. However this grid growths exponentially and that makes this method almost practically

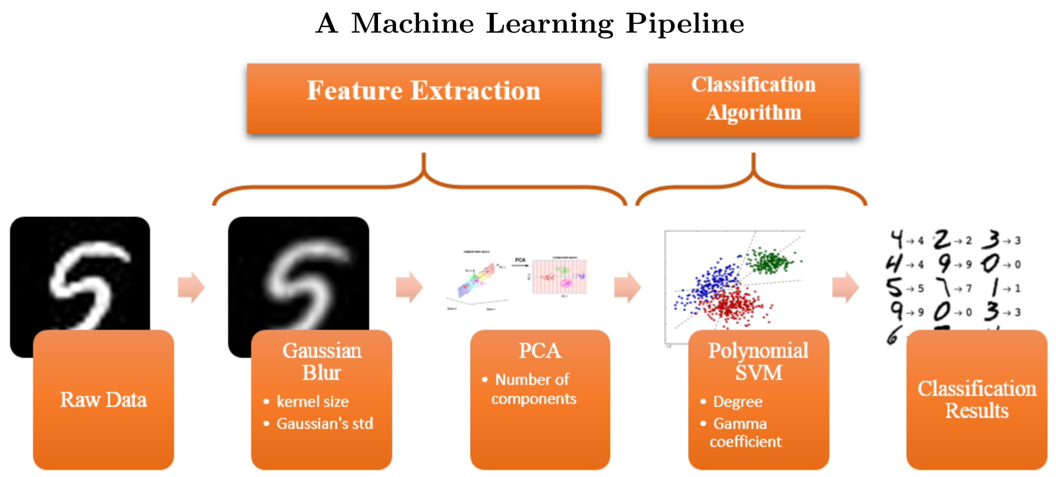

Figure 1-2: A pipeline example and some tunable hyper-parameters for the handwritten digits recognition problem.

unfeasible. For example with the pipeline in Fig. 1-2, we propose to test 2, 5, 26, 4, 4 values for respectively the Gaussian blur kernel size, standard deviation, PCA dimension, SVM kernel degree and gamma coefficient, which makes 4160 hyper-parameter sets. Testing a hyper-parameter set can takes around a minute, so the time needed to test all the possible values is about three days (and we must keep in mind that this is only one pipeline example that does not have many hyper-parameters).

In [2], Bergstra and Bengio show that random search is much faster than grid-search for hyper-parameter optimization, and recommend to investigate on *adaptive* search, *ie.* search models that take into account previous tests to decide which part of the space to explore. Such sequential techniques have been studied in [3, 4, 5] and compared in [6]. However to the best of our knowledge there has not been much effort put into the design of an automated machine that would tune a whole machine learning pipeline; most of the previous work focuses on tuning the hyper-parameters of classification algorithms. We are thus interested in building such a machine. This goal can be compared to the work of Will Drevo [7] who designed a machine that automatically tests several classification models and hyper-parameter sets in a sequential manner, in order to find the *best* model.

In our opinion, auto-tuning the whole machine learning pipeline has not been done yet because the feature extraction part is the most time-consuming and present here the most significant challenge. It is indeed now common to deal with several millions of images representing hundreds of gigabytes of memory, and each of them must be processed to extract some features like SIFT descriptors [8]. So such an automated machine should take the most advantages of current hyper-parameter optimization methods but also include pratical considerations, like thinking of a parallel design.

Regarding the hyper-parameter optimization technique, we focus on Gaussian Process (GP)-

based methods such as [9, 3]. We are also interested in Gaussian Copula Processes (GCP) [10] for they should present better modeling power than GP. Thus our main goal was to:

- Design a GCP-based hyper-parameter optimization technique
- Use this technique to auto-tune machine learning pipelines (MLP)

1.3 Contributions

Our main contributions are the following:

- Design a new regression technique based on GCP that we call non-parametric GCP (nGCP)
- Design an auto-tuning machine for machine learning pipelines based on noisy performance estimations
- Build this machine through nGCP modeling
- Compare several nGCP-based hyper-parameter optimization methods for machine learning pipelines on two real-world problems : on sentiment analysis and handwritten digits recognition.

Chapter 2

Non-Parametric Gaussian Copula Processes (nGCP)

In this section, we first recall the fundamentals of Gaussian Processes (GP) for machine learning and present previous methods used to design Gaussian Copula Processes (GCP). We then introduce our novel non-parametric approach to GCP (nGCP) and finally propose a latent model for nGCP (nLGCP).

2.1 Gaussian Process (GP)

This is a brief review of Gaussian Processes for regression; see Rasmussen and Williams [11] for a complete overview of GP.

The Gaussian Process in Machine Learning is a particular prior to perform Bayesian learning. Specifically if the goal is to predict the values of a function $f : \mathcal{X} \to \mathbb{R}$, this means that we model f such that for any finite set of N points x_n in $\mathcal{X}$, $\{f(x_i)\}_{i=1}^N$ has a multivariate Gaussian distribution on $\mathbb{R}^N$.

The properties of such a process are determined by the mean function $m : \mathcal{X} \to \mathbb{R}$ and the covariance function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$:

$$m(x) = \mathbb{E}[f(x)] \tag{2.1}$$

$$k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]. \tag{2.2}$$

Note that these functions depend on x only because x is a proxy to index the random variable $f(x)$.

Usually the mean function is fixed as the average of the known f values (on the training data), so we will take it to be zero assuming the data are centered. The covariance function could also be fixed but most of the time it is parametrized and the values of its parameters are learned while fitting the Gaussian Process to the data. This last step can be done through Maximum Likelihood Estimation [11].

In the Bayesian framework predictions are simply made by looking at the posterior distribution, which is the prior conditioned on the training data $(\bar{x}_t, \bar{f}_t)$. Thus the predicted value f_* of the function f at point x_* would be:

$$f_* = \mathbb{E}(\ f(x_*) \mid \bar{x}_t,\ \bar{f}_t\ ), \tag{2.3}$$

where :

$$\begin{aligned} f(x_*) \sim \mathcal{N}(\ & k(x_*, \bar{x}_t)K_t^{-1}\bar{f}_t, \\ & k(x_*, x_*) - k(x_*, \bar{x}_t)K_t^{-1}k(\bar{x}_t, x_*)). \end{aligned} \tag{2.4}$$

and $K_t = k(\bar{x}_t, \bar{x}_t)$ is the square covariance matrix of the training data.

2.2 Gaussian Copula Process (GCP)

The Gaussian Copula Process, as introduced in [10], is a prior based on a GP that can model more precisely the multivariate distribution of $\{f(x_i)\}_{i=1}^N$. This is done through a mapping

$$\Psi : \mathbb{R} \to \mathbb{R}$$

that transforms f into a new function g to which we set a GP prior:

$$g(x) = \Psi \circ f(x) \tag{2.5}$$

$$g(\bar{x}) \sim \mathcal{N}(0, k(\bar{x}, \bar{x})) \tag{2.6}$$

where $g(\bar{x})$ denotes the vector $[\ g(x_1)\ ...\ g(x_N)\ ]^T$.

By doing this, we actually change the assumed Gaussian marginal distribution of each $f(x)$ into a more complex one. Precisely, the Gaussian prior on $g(x)$ yields the prior for $f(x)$ given by the following cumulative distribution function:

$$F(t) = \Phi \circ \Psi(t), \tag{2.7}$$

where $F(t) = \mathbb{P}(f(x) \leq t)$ and Φ is the standard univariate Gaussian cumulative distribution function.

So far in the literature, *eg.* [10, 12], a parametric mapping is learned so that $g(x)$ is best modeled by a Gaussian Process. In [10], Wilson and Ghahramani propose to parametrize Ψ^{-1} by $\{a_j, b_j, c_j\}$ such that:

$$\Psi^{-1}(x; \{a_j, b_j, c_j\}_{j=1}^K) = \sum_{j=1}^{K} a_j log\big(e^{b_j(x+c_j)} + 1\big),\ a_j, b_j > 0. \tag{2.8}$$

In [10] they are interested in predicting the values of a positive function f. So in the general case

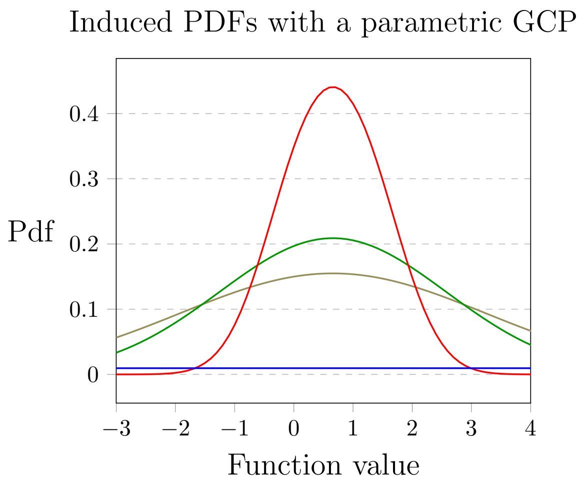

Figure 2-1: Yielded probability density functions for multiple trials with a parametric GCP, where the mapping function is defined by eq. (2.9). The function to model is a custom function shown in blue in Fig. 2-3.

we can add another variable m:

$$\Psi^{-1}(x; \{a_j, b_j, c_j\}_{j=1}^{K}, m) = \sum_{j=1}^{K} a_j log\big(e^{b_j(x+c_j)} + 1\big) - m,\ m, a_j, b_j > 0.$$

Thus for $K = 1$ we have :

$$\Psi(t) = \frac{log(e^{\frac{t+m}{a}} - 1)}{b} - c,\ m, a, b > 0. \tag{2.9}$$

However, we have found that such a mapping was unstable: for many trials on a same dataset, different mappings where learned, as shown in Fig. 2-1. Moreover the induced univariate distribution for $f(x)$ was most of the time almost Gaussian and the parametric mapping could not offer a great flexibility. We indeed see in eq. (2.8) that for $K = 1$, if $bx >> bc, 1$, then $\Psi^{-1}(x; a, b, c) \sim abx$, *ie.* the mapping is linear and the GCP is actually a GP.

Thus we introduce a novel approach where a marginal distribution is learned from the observed data through kernel density estimation [13] of F. Then the mapping Ψ is numerically computed from equation (2.7), so that the observations of the training data $g(x_{t,i})$ have a Gaussian distribution:

$$\Psi(t) = \Phi^{-1} \circ F_{est}(t). \tag{2.10}$$

As the mapping function is learned in a non-parametric manner, we call this novel approach *Non-Parametric Gaussian Copula Process* and note it nGCP.

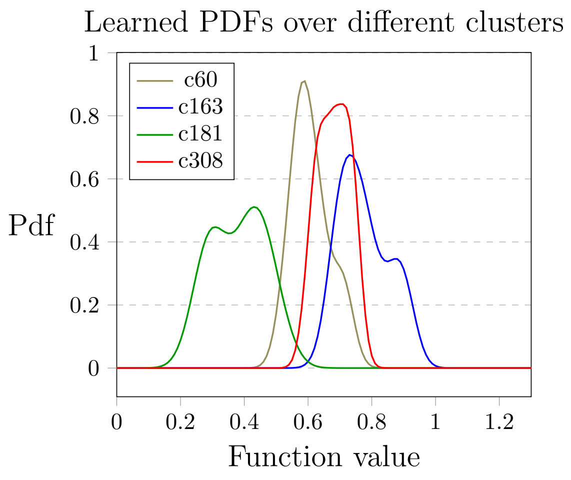

Figure 2-2: Density estimations with a Gaussian kernel of the probability density function on 4 clusters. cXXX in the legend refers to a cluster centered on x=XXX. The function used is a custom function shown in blue in Fig. 2-3.

2.3 Non-Parametric Latent GCP (nLGCP)

As said in section 2.1, the prior mean of a Gaussian process is usually fixed as the empirical mean of the observations $f(x_{t,i})$. However in the context of hyper-parameter optimization, we can easily feel that there should be some region where the hyper-parameters would be *rather good* and others where they would be *rather bad*. Thus it appears that it may be convenient to set a different mean for the prior depending of the region in which the hyper-parameter is. Regarding the GP, scientists would easily argue that this would not have much impact as the role of the covariance function is exactly to induce such smoothness. However with GCP we do not only fix the mean function but the mapping function. We recall that the mapping function reflects the distribution of the data. Thus with nGCP a latent model aims at learning several distributions of $f(x)$ over the input space. In particular we think this can have a positive influence when trying to locate the good regions to explore in a Bayesian optimization process.

Thus we introduce a non-parametric *Latent Gaussian Copula Process* prior(nLGCP), where the mapping function depends also on the input x. Intuitively the goal is to include in the prior not only the distribution $\mathcal{D}$ of $f(x)$ on the entire space $\mathcal{X}$ but the distributions $\mathcal{D}_1$, ..., $\mathcal{D}_k$ of $f(x)$ on k regions of $\mathcal{X}$. This way, we design a prior that *really* depends on x (whereas in the previous equations x was only an index to denote the random variable $f(x)$).

To design the nLGCP prior, we are looking for a mapping function that depends on the input x *and* output $f(x)$,

$$\Psi : \mathcal{X} \times \mathbb{R} \to \mathbb{R}.$$

To this purpose, the training data $\{(x_i, f(x_i))\}$ are clustered in $\mathcal{X} \times \mathbb{R} \subset \mathbb{R}^{m+1}$ using *K-means*. For each cluster k a mapping Ψ_k is learned as described in section 2.2. Then for each x in $\mathcal{X}$, the

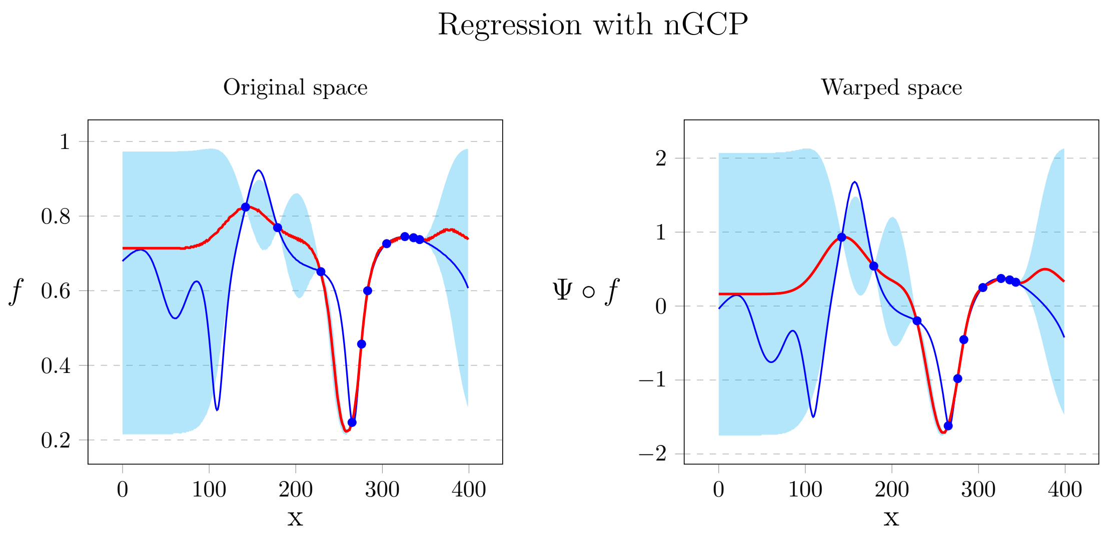

Figure 2-3: nGCP predictions and 95% confidence bounds. The unknown function is shown in blue, and the blue points correspond to the training data, randomly chosen in the interval $[150, 350]$. Predictions are shown in red while the cyan zone corresponds to the 95% confidence interval. In particular we see that with the GP prior on $\Psi \circ f$, the upper and lower bounds are symmetric about the posterior's expectation, while this is not the case with the nGCP prior over f.

final mapping Ψ is computed as:

$$\Psi(x, y) = \sum \alpha_k(x).\Psi_k(y), \tag{2.11}$$

where :

$$\alpha_k(x) = exp(-s \sum (\frac{d_k}{\sigma_k})^2) \tag{2.12}$$

$$d_k = dist_{\mathcal{X}}(x, c_k) \tag{2.13}$$

$$\sigma_k = std_{\mathcal{X}}(\mathcal{C}_k) \tag{2.14}$$

and s is a smoothing coefficient, c_k is the projected center of the cluster $\mathcal{C}_k$ on $\mathcal{X}$.

2.4 Predictions with n(L)GCP

In this section, we do not distinguish Ψ for nGCP or nLGCP when this is useless in order to simplify the equations. So with nLGCP we write Ψ to mean $\Psi(x_*, t)$ where x_* is the point of $\mathcal{X}$ where we want to predict f.

Predictions with GP are straightforward given equations (2.3,2.4) but it is not the case anymore with n(L)GCP. A faster but approximate way to compute the predicted value of $f(x_*)$ for a given x_*, is to set $f_* = \Psi^{-1}(g_*)$ where g_* is the GP prediction of $\Psi \circ f(x_*)$.

For a fully Bayesian prediction of $f(x_*)$ as given by eq. (2.3), for an input x_*, we can recall that GP analysis give us the posterior:

$$g(x_*) \sim \mathcal{N}(\mu_*, \sigma_*)$$

where μ_*, σ_* correspond to the detailed expressions of equation (2.4). Thus from equation (2.7) we get that $F_* = \Phi_{\mu_*,\sigma_*} \circ \Psi$, where F_* denotes the predicted cumulative distribution function of $f(x_*)$, and so the prediction is:

$$f_* = \int_{u=-\infty}^{\infty} u.\Psi'(u).\phi_{\mu_*,\sigma_*} \circ \Psi(u).du \tag{2.15}$$

where ϕ_{μ_*,σ_*} denotes the probability density function of a univariate Gaussian $\mathcal{N}(\mu_*, \sigma_*)$.

The particular expression of Ψ in eq. (2.10) for the nGCP prior finally enables us to express directly its derivative:

$$\Psi'_{nGCP}(t) = \frac{d_{est}(t)}{\phi \circ \Psi(t)}, \tag{2.16}$$

where ϕ and d_{est} are respectively the probability density function of the standard univariate Gaussian and the one corresponding to F_{est} defined in section 2.2.
Fortunately we have a similar expression for the derivative of Ψ for nLGCP:

$$\Psi'_{nLGCP}(x,t) = \sum_{cluster\ k} \alpha_k(x).\Psi'_k(t) \tag{2.17}$$

$$\Psi'_{nLGCP}(x,t) = \sum_{cluster\ k} \alpha_k(x).\frac{d_{est,k}(t)}{\phi \circ \Psi_k(t)}. \tag{2.18}$$

Chapter 3

Tuning a Machine Learning Pipeline

Ultimately we aim at finding the sets of hyper-parameters that yield machine learning pipelines which should achieve high classification accuracy on whatever test set when trained on (almost) whatever training set. In this section, we first present the hyper-parameter optimization process, then explain how we will test a set of hyper-parameters and finally which hyper-parameters to choose when the process is done.

For simplification purposes, we will write *hyper-parameter* instead of *a set of hyper-parameters*, so that a hyper-parameter is a vector corresponding to the pipeline's hyper-parameters that we would like to tune.

3.1 Hyper-parameter Optimization Process

We suppose here that a *performance function* is given which enables us to test a hyper-parameter and retrieve an estimation of its quality. Thus we try to find the hyper-parameters that maximize this function.

We need to train and test an entire machine learning pipeline to estimate the quality of a hyper-parameter, which takes a lot of time and computations, so this is a particular type of *costly function optimization*. The method used follows an iterative process with three steps: *model & predict*, *select*, *evaluate*.

1. Model & Predict :
   Given a set of observations $\mathcal{O} = \{(\bar{p}_i, o_i)\}_{i=1}^{N}$, fit a model to express the correlations between a hyper-parameter $\bar{p}_i$ and the performance induced o_i.
   Sample randomly $\{\bar{p}'_i\}_{i=1}^{k}$ in $\mathcal{P}$ and predict the performances $\{o'_i\}_{i=1}^{k}$.

2. Select :
   Given an *acquisition function* a, select $\bar{p}'^{*}$ that maximizes $a(o')$. The goal and the form of the acquisition function are discussed in section 4.3.

3. Evaluate :
   Compute the performance of the hyper-parameter $\bar{p}'^*$ and add the result to $\mathcal{O}$.

Each iteration corresponds to exactly one evaluation of the performance function. Usually we have an amount of time and our goal is to find the best hyper-parameters within this time. Thus we will refer to this process as the *Smart Search process* because it aims at sampling hyper-parameters p in $\mathcal{P}$ while maximizing our chance to find good hyper-parameters.

3.2 Noisy performance estimations

In order to speed up the tuning process, but also to make it easily parallelizable, we decided to run each hyper-parameter test on a random subset d of the whole dataset available $\mathcal{D}$.

A machine learning pipeline can be split into two main components: *feature extraction* and *classification algorithm*. As discussed before, processing a dataset to extract proper features usually takes a lot of time, and even considerably more than testing a classification algorithm. This means that we will allow us to test more a classification algorithm rather than the feature extraction model.

We thus have two layers to nuance: 1) the quality of the feature extraction model depending on the data, and 2) the quality of the classification algorithm depending on the transformed data. To nuance the first layer we could for example split the dataset d before learning the representation and then extract the features on the rest. Or we could sub-sample several d from $\mathcal{D}$ at each hyper-parameter test. But this would yield too many computations and we decide instead to model the evaluation as *noisy performance estimations*.

Precisely, given the data $\mathcal{D}$ of size $s_{\mathcal{D}}$ and a sub-sample size $s_d < s_{\mathcal{D}}$ we test the hyper-parameter $\bar{p}$ by the following method:

- Sub-sample data d of size s_d from $\mathcal{D}$
- Extract the features from d according to $\bar{p}$
- Perform k-fold cross-validation on d with a classification algorithm given by $\bar{p}$
- Return the k performance estimations

3.3 Ranking the hyper-parameters

We described in section 3.1 the generic process used, and in the former section how we evaluate the quality of a machine learning pipeline, and here we discuss how we can select good hyper-parameters based on this.

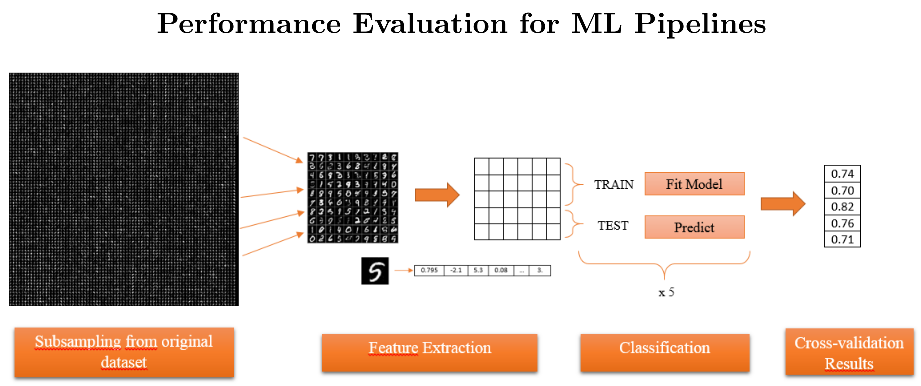

Figure 3-1: Method used to estimate the quality of a hyper-parameter. The feature extraction method and the classification algorithm depend on the hyper-parameter, but data sub-sampling is done randomly and uniformly.

Usually we would simply set a trade-off between the mean μ and the standard deviation σ of the multiple cross-validation estimations $\bar{o}$:

$$\mathcal{Q}(\bar{o}) = \mu(\bar{o}) - \alpha\,\sigma(\bar{o}), \tag{3.1}$$

where α is a balance coefficient to define. We could then select the hyper-parameter, from those tested during the process, that maximizes this function $\mathcal{Q}$.

However we propose a more complex approach in order to deal with:

- the noise in the observations $\bar{o}$, introduced by testing the machine learning pipelines only on (small) sub-samples of the dataset,
- the desire to have a high confidence in the selected hyper-parameters.

Note that the standard deviation of the cross-validation results is usually a good way to evaluate the confidence one can have in a model but is not here sufficient to estimate the confidence we can have in a hyper-parameter because we sub-sampled the data. For example, it does not take into account the number of times a given hyper-parameter has been tested (*ie.* the number of sub-sampled datasets on which it has been tested).

3.3.1 Consider only significant differences

We use Welch's t-test [14, 15] in order to consider only significant differences in the mean values of the performance estimations. The idea is to replace $\mu(\bar{o})$ in eq. (3.1) by $\tilde{\mu}(\bar{o})$ where $\tilde{\mu}$ can only take its values in a set $\mathcal{S}_\mu = \{\mu_1, ..., \mu_w\}$. $\mathcal{S}_\mu$ is computed through these steps:

i Sort the observation vectors $\mathcal{O} = \{\bar{o}_i\}$ by increasing mean.
  Set $O_1 = \{o_{1,1}, ..., o_{1,k}\}$, $j = 1$.

ii For $\bar{o}$ in $\mathcal{O}$,
if $\mathcal{W}(\bar{o}, O_j) > \tau$, add $o_1, ..., o_k$ to O_j
else, $j \leftarrow j+1, O_j = \{o_1, ..., o_k\}$
where τ is a threshold and $\mathcal{W}$ refers to the p-value of Welch's t-test.

iii $w \leftarrow j$ and for each O_j, set μ_j as its mean.

3.3.2 Consider hyper-parameters' neighborhood

To address the confidence problem, we propose to look at the hyper-parameters' neighborhood. We indeed expect the pipelines' performances to be somewhat smooth, *ie.* close hyper-parameters yield close performances. Thus the performances of a hyper-parameter's neighbors can inform us on its quality.

Ultimately we would select a hyper-parameter that yields high performances and which has *close* hyper-parameters tested during the optimization process, and which also yield high performances. The fact that those neighbors are close is quite important as the distance directly impacts the informativeness of a neighbor. Given a number n_c of neighbors to consider and a radius r_c, we define the smooth mean function $\tilde{\mu}_s$ by:

$$\tilde{\mu}_s(o_{\bar{p}}) = \frac{\mu_{neigh}(o_{\bar{p}}) + \beta\,\tilde{\mu}(o_{\bar{p}})}{1+\beta},$$
$$\mu_{neigh}(o_{\bar{p}}) = \frac{1}{n_c}\sum_{i=1}^{n_c} e^{-\left(\frac{d_i}{r_c}\right)^2}\tilde{\mu}(o_{\bar{p}_i}),$$

where $d_i = dist(\bar{p}, \bar{p}_i)$ and β is a trade-off parameter.

In the following we will estimate the quality of a hyper-parameter $\bar{p}$, of observed performances $o_{\bar{p}}$, through the functions $\tilde{\mathcal{Q}}$ and $\tilde{\mathcal{Q}}_s$:

$$\tilde{\mathcal{Q}}(\bar{p}) = \tilde{\mu}(o_{\bar{p}}) - \alpha\,\sigma(o_{\bar{p}}) \tag{3.2}$$
$$\tilde{\mathcal{Q}}_s(\bar{p}) = \tilde{\mu}_s(o_{\bar{p}}) - \alpha\,\sigma(o_{\bar{p}}) \tag{3.3}$$

3.4 Two pipeline examples

3.4.1 Sentiment analysis

The first problem we tackle is based on text analysis, where the aim is to predict the probability for a movie review to be positive or negative, *ie.* to understand the global emotion expressed in the review. We used Kaggle's competition *Bag of Words Meets Bags of Popcorn* which provides 25,000 labeled training data from IMDB. The labels where attributed based on the rating accompanying the review: 0 for a rating lower than 5 and 1 for a rating greater or equal to 5.
Pipeline description:

i Transform the reviews in Bags of n-grams ii Transform the n-grams count vectors in tf-idf vectors

iii Feature selection based on the χ^2 test

iv Classify with a multinomial Naïve Bayes classifier

The hyper-parameters:

i Bag of n-grams transformation:

- Maximum number of features to keep $n_f \in [13000; 32000]$.
- Consider only the terms with a document frequency between df_{min} and df_{max}, where $df_{min} \in [0; 0.1], df_{max} \in [0.4; 1]$.
- Consider n-grams terms for n between 1 and $n_{ngram,max}$, where $n_{ngram,max} \in [1; 4]$.

ii Tf-idf transformation: we choose between one of the following methods

- Norm L1
- Norm L2
- Norm L2, sub-linear tf scaling

iii Feature selection: we only keep the κ% best features, where $\kappa \in [50; 100]$.

iv Classification: we use a Lidstone smoothing parameter $\alpha \in [0.1; 1]$.

3.4.2 Handwritten digits

The second problem tackled is the famous handwritten digits recognition problem from the MNIST dataset. Once again we used Kaggle's platform and so our training dataset consists in 42,000 images of size 28x28.
Pipeline description:

i Blurring the training images with a Gaussian kernel of size $k_{size} \in \{3; 5\}$ and standard deviation $k_\sigma \in [0; 4]$ (the blur is not applied when k_σ is null).

ii The image matrices are reduced by Principal Component Analysis, and the number of components $d_{PCA} \in [50; 300]$.

iii Classification is made through polynomial SVMs of degree $d_{SVM} \in [1; 4]$ and gamma coefficient $\gamma = 10^g$, $g \in [-3; 1]$.

Average cross-validation results for the sentiment analysis problem

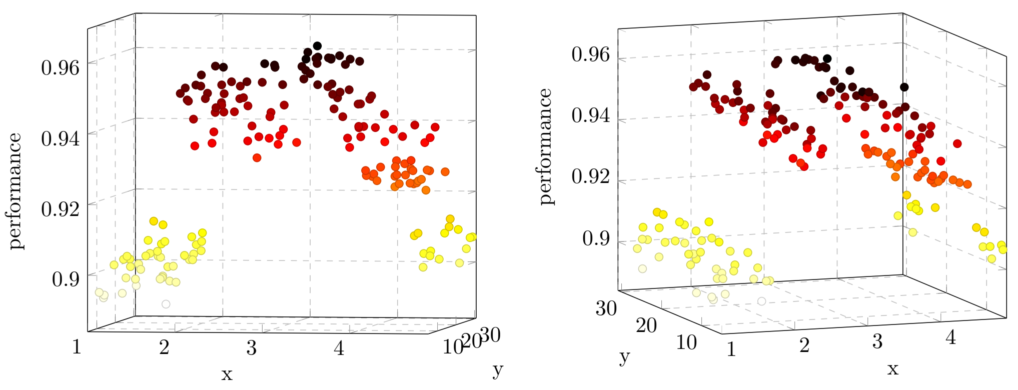

Figure 3-2: Mean performances observed on the hyper-parameter space for the sentiment analysis problem. For visualization purposes, the 5-dimensional hyper-parameters are projected on two dimensions so that $x = d + 0.5 * k_{size} + 0.1 * k_{\sigma}; y = 0.1 * d_{PCA} + 0.1 * (g + 3)$. In addition the color of the points corresponds to their value on the z-axis (the performance).

Chapter 4

Practical use of n(L)GCP for MLP optimization

4.1 Predict with noisy observations

In chapter 2 we introduced the non-parametric (Latent) Gaussian Copula Process and explained how we could learned a *mapping function* to build such a prior. In that chapter we considered having noise-free observations of the function f. However we usually have instead noisy observations $y = f(x) + \epsilon$ of f, and the discussion in the previous chapter largely points out that we cannot think of a MLP performance estimations as a clear representation of its true quality.

Figure 4-1 displays the deviation of each observation to its estimated true value. The latter is computed as the observed mean of all cross-validation results given for a single hyper-parameter. On this figure are represented both the histogram of values observed and a Gaussian estimation of them. We can see that this *noise* indeed seem to be Gaussian, which encourage us to really model the multiple cross-validation estimations of a single hyper-parameter as noisy observations of the same random variable.

When using a GP prior, considering noisy observations does not change much, as it is taken independent and identically distributed with variance σ_n^2. Thus it suffices to replace the covariance matrix K_t in eq. (2.4) as follow :

$$K_{t,n} = K_t + \sigma_n^2 I$$

where I is the identity matrix [11].

However with n(L)GCP, it is not clear how to deal with such noise, and here are a few questions that arise:

- Would it help or not to consider all cross-validation estimations or should we rather only consider the mean ?
- Should we use all cross-validation estimations to learn the mapping function Ψ ?

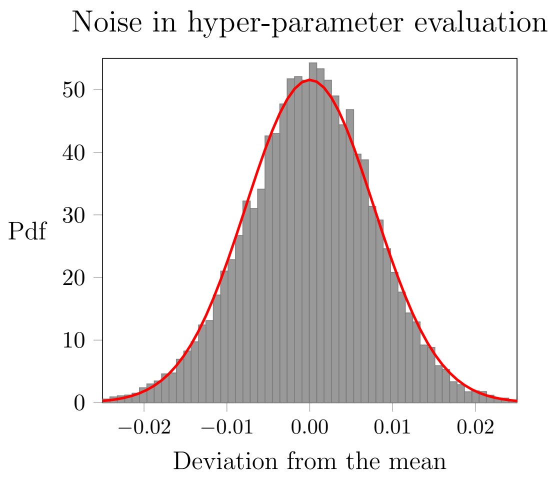

Figure 4-1: Histogram of the observed deviations from empirical mean of cross-validation estimations, compared to its Gaussian interpolation in red.

- How would the noise be transformed by Ψ ?
- Can we transform the noisy observations of $f(x)$ to get noisy observations of $g(x)$?

4.1.1 Learning the mapping function

By using all cross-validation estimations to learn the mapping function, we would hope to get $g(x)$ values with an almost Gaussian distribution, thus facilitating the GP estimation for g. However with only few training data with a relatively high standard deviations in the cross-validation estimations, we observed that this method could almost erase the benefit of the nGCP by learning a non-discriminative mapping function (the distribution of the performance estimations seems Gaussian).

4.1.2 Transforming the noise

It is interesting to see that transforming all cross-validation estimations y with Ψ in $z = \Psi(f(x) + \epsilon) = g(x) + \epsilon'$ completely deforms the noise ϵ. Actually if ϵ is indeed Gaussian, this is the exact opposite case of chapter 2 as we have $\Psi^{-1}(z) \sim \mathcal{N}(f(x), \sigma_n)$. Thus equation (2.7) yields:

$$F_z(t) = \Phi_{f(x),\sigma_n} \circ \Psi^{-1}(t).$$

Whereas such a transformation was desired to refine the prior over f, its side effect is to non-linearly transform the noise without any control. We thus should guess $f(x)$ from y to control the noise.

4.1.3 Reproducing a Gaussian noise

So we consider only the averages of the cross-validation estimations as a proxy for $f(x)$ and suppose that it gives us observations of $g(x)$ with a Gaussian noise. Formally, let $\{y_{i,j}\}_{j=1}^{J}$ be multiple cross-validation estimations of a single hyper-parameter $\bar{p}_i$ and f_i their mean. We consider $\{z_{i,j}\}_{j=1}^{J}$ as noisy observations of $g(\bar{p}_i) = \Psi \circ f(\bar{p}_i)$ where

$$z_{i,j} = g_i + \frac{y_{i,j} - f_i}{\sigma_f}, \tag{4.1}$$

$$g_i = \Psi(f_i), \tag{4.2}$$

$$\sigma_f = std(f_i, i = 1, ..., N). \tag{4.3}$$

σ_f is a scaling parameter to take into account that g is supposed to have a Gaussian distribution with standard deviation equals to one.
Note that $z_i = g_i + \frac{\epsilon}{\sigma_f}$ where ϵ is the Gaussian noise around f_i. Thus $z_i = g_i + \epsilon'$ where

$$\epsilon' \sim \mathcal{N}\big(0, \frac{\sigma_n}{\sigma_f}\big). \tag{4.4}$$

Finally we modify accordingly the covariance function for the GP estimation of g:

$$K_{t,n} = K_t + \big(\frac{\sigma_n}{\sigma_f}\big)^2 I.$$

4.1.4 Estimating the noise

As said above we can decide to estimate the noise σ_n or to fix it identical for all x, which is usually done with GP. In this particular context of hyper-parameter optimization we propose to set the noise from the standard deviation of the cross-validation estimations, and we will refer to this method as the *Estimated Gaussian Noise (EGN)*. This way we have:

$$K_{t,n} = K_t + \frac{1}{\sigma_f^2} \begin{pmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_N^2 \end{pmatrix},$$

$$\sigma_i = std(y_{i,j}, j = 1, .., J).$$

4.2 Covariance Function

We used both the *Squared Exponential (SE)* covariance function and another one inspired from [11] that we will call *Exponential Periodic (EP)*, defined as follow:

$$k_{SE}(x, x') = exp(-\sum_{i=1}^{m} \theta_i.(x_i - x'_i)^2),$$

$$k_{EP}(x, x') = \theta_0.k_0(x, x') + \theta_1.k_1(x, x') + \theta_2.k_2(x, x')$$

with :

$$x = [x_1, ..., x_m]^T \in \mathcal{X} \subset \mathbb{R}^m,$$

and :

$$k_0(x, x') = exp(-\sum_{i=1}^{m} \theta_{3,i}.(x_i - x'_i)^2) \tag{4.5}$$

$$k_1(x, x') = exp(-\sum_{i=1}^{m} \frac{(x_i - x'_i)^2}{2.\theta_{4,i}} - \sum_{i=1}^{m} 2\Big(\frac{\sin(\pi\frac{(x_i - x'_i)}{\theta_{5,i}})}{\theta_{6,i}}\Big)^2) \tag{4.6}$$

$$k_2(x, x') = \prod_{i=1}^{m} \Big(1 + \frac{(x_i - x'_i)^2}{\theta_{8,i}}\Big)^{-\theta_{7,i}} \tag{4.7}$$

The value of the parameter θ is set while fitting the n(L)GCP to the training data. This is actually done by first computing the mapping function and the corresponding g values. Then we only consider the $(x, g(x))$ data and fit a GP by maximizing the estimated likelihood.

4.3 Acquisition Function

4.3.1 Expected Improvement (EI)

The expected improvement is one of the most widely used acquisition function [9] when dealing with GP. Given a set of previously tested hyper-parameters $\mathcal{P}_t$, let $\bar{p}_+ = \text{argmax}_{\bar{p} \in \mathcal{P}_t} f(\bar{p})$. We define the improvement function as:

$$I_{GP}(\bar{p}) = max\{0, g_{pred}(\bar{p}) - g(\bar{p}_+)\}$$

and the likelihood of improvement I for a random variable $g_{pred}(\bar{p}) \sim \mathcal{N}(\mu(\bar{p}), \sigma(\bar{p}))$ is:

$$d_{I_{GP}}(I) = \frac{1}{\sqrt{2\pi}\sigma} exp\big(-\frac{(\mu - g(\bar{p}_+) - I)^2}{2\sigma^2}\big).$$

Finally the expected improvement is the integral:

$$\mathbb{E}(I_{GP}) = \int_{u=0}^{\infty} u.d_{I_{GP}}(u).du.$$

While the expected improvement has a very simple expression for a GP, it is more reasonable in the n(L)GCP context to compute the improvement on the original function f:

$$I(\bar{p}) = max\{0, f_{pred}(\bar{p}) - f(\bar{p}_+)\}$$

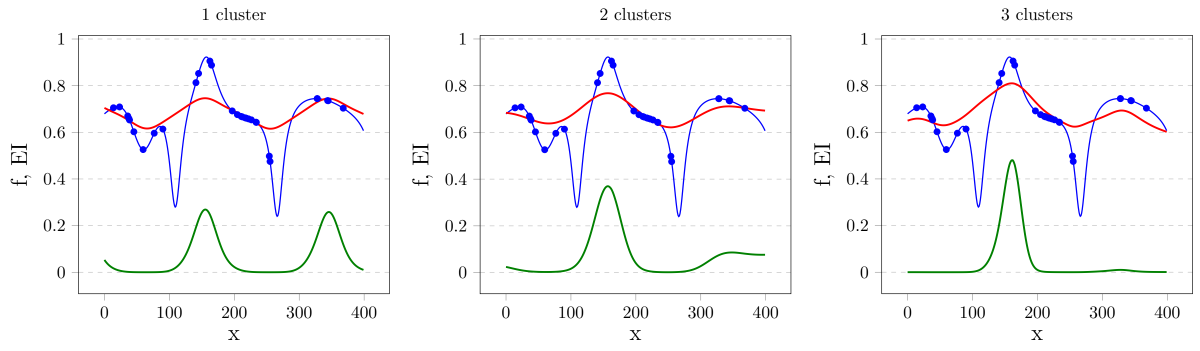

Figure 4-2: nLGCP models of an unknown function (in blue). Predictions (in red) and the expected improvement (in green, multiplied by 100) are computed based on the training points (blue points).

The likelihood of improvement is actually easily derived from the above expression:

$$d_I(I) = \frac{\Psi'(f_+ + I)}{\sqrt{2\pi}\sigma} exp\Big(- \frac{\big(\mu - \Psi(f_+ + I)\big)^2}{2\sigma^2} \Big).$$

where $f_+ = f(x_+)$.

We recall from eq. (2.16,2.18) the expressions of Ψ' for nGCP and nLGCP:

$$\Psi'_{nGCP}(t) = \frac{d_{est}(t)}{\phi \circ \Psi(t)}, \tag{4.8}$$

$$\Psi'_{nLGCP}(\bar{p}, t) = \sum_{cluster\ k} \alpha_k(\bar{p}). \frac{d_{est,k}(t)}{\phi \circ \Psi_k(t)}, \tag{4.9}$$

so that we can compute the expected improvement:

$$a_{EI} = \mathbb{E}(I) = \int_{u=0}^{\infty} u.d_I(u).du.$$

4.3.2 Upper Confidence Bound (UCB)

Another well used acquisition function is the Upper Confidence Bound. With a GP model, we can easily express the confidence bound through the posterior mean μ and standard deviation σ:

$$UCB_{GP,\kappa}(\bar{p}) = \mu(\bar{p}) + \kappa\sigma(\bar{p})$$

where $\kappa \geq 0$ is a parameter to control the balance between exploitation and exploration.

With an n(L)GCP model, we are thus interested in maximizing $UCB_{nGCP,\kappa}(\bar{p}) = \Psi^{-1} \circ$

$UCB_{GP,\kappa}(\bar{p})$:

$$a_{UCB} = \Psi^{-1}(\mu(\bar{p}) + \kappa\sigma(\bar{p})).$$

As Ψ is an increasing function (see eq. (2.10,2.11)), we only have to maximize $UCB_{GP,\kappa}$ when dealing with nGCP (although this is not true for nLGCP). This is useful as the costly numerical inversion of Ψ becomes useless in this case.

4.4 Number of clusters for an nLGCP

When we introduced the Latent Gaussian Copula Process in section 2.3, we said that data are clustered into k clusters using K-Means without any advice on how to choose k. In our hyper-parameter optimization framework, we propose to increase the number of clusters with the iterations. We recall that the number of iterations is exactly the number of already tested hyper-parameters $\bar{p}$ used to fit the nLGCP. With this method we can easily control the complexity of the model with respect to the current knowledge.

In the tests reported in chapter 6 we added one cluster every 30 iterations.

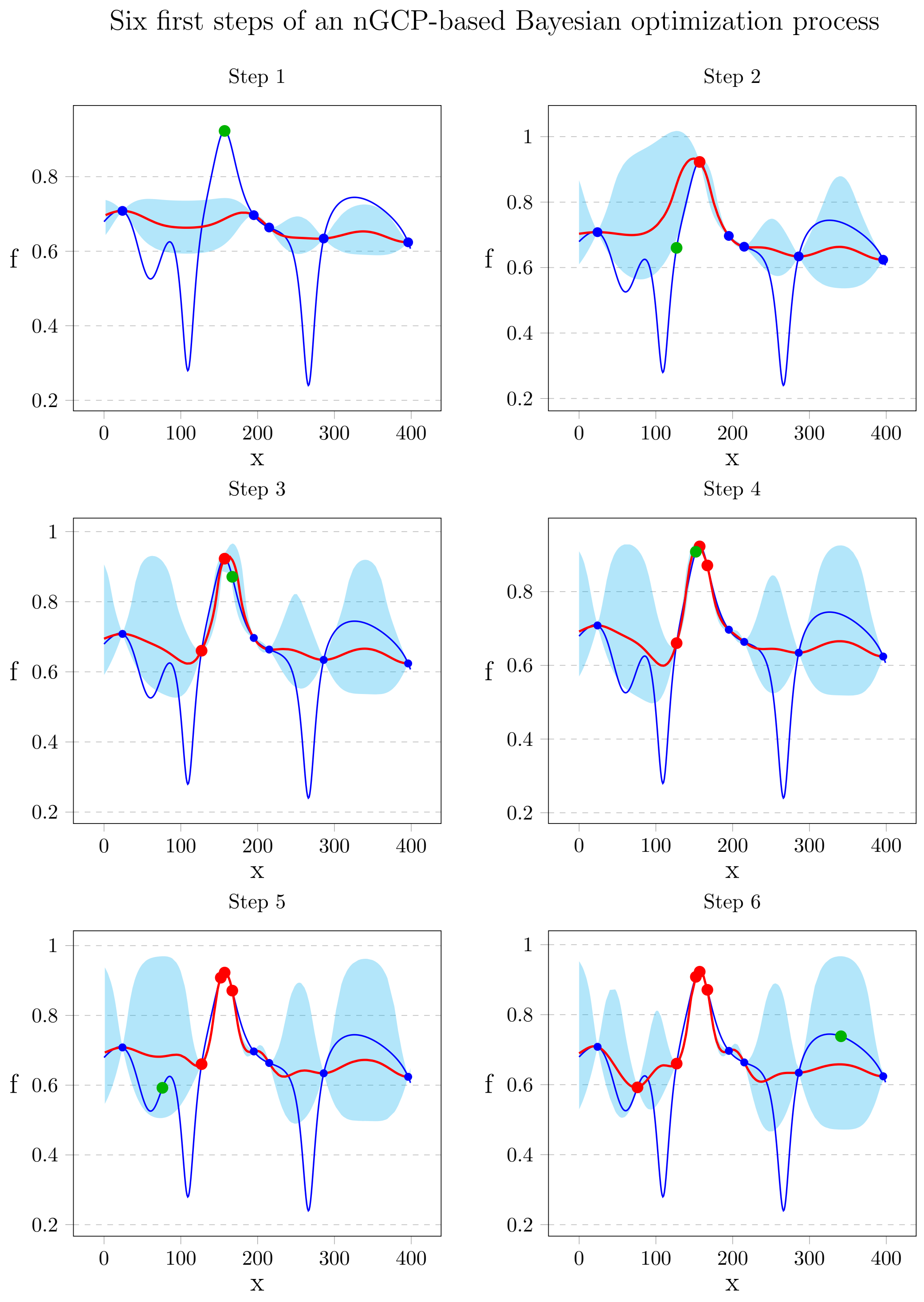

Figure 4-3: Six first steps of a Smart Search process with the UCB acquisition function (confidence bounds are shown in cyan). The process begins with the 5 blue points and the goal is to find the maximum of the unknown blue function f. At each step, f is modeled by an nGCP, which is shown by the red line, and the selected point (next point to test, *ie.* the point maximizing the acquisition function) is in green.

Chapter 5

Algorithm

5.1 n(L)GCP

Let X be the matrix of inputs (each row is a unique hyper-parameter) and Y the outputs (performance estimations).

i Normalize the data:
$\tilde{X} := [\tilde{X}_1|...|\tilde{X}_m]$ where $\tilde{X}_i = \frac{1}{\sigma_i}(X_i - m_i[1...1]^T)$ and m_i, σ_i are respectively the mean and standard deviation of $\{X_{j,i}|j = 1,..,N\}$. $\tilde{Y} := \frac{1}{\sigma}(Y - m[1...1]^T)$.

ii Compute clusters' attributes:
Perform K-Means on $Z := [\tilde{X}_1|...|\tilde{X}_m|\tilde{Y}]$ to compute the k clusters $C_1, ..., C_k$. For each cluster C_l, let $d_{est,l}$ be the Kernel Density Estimation, with a Gaussian kernel, of $\tilde{Y}_{C_l}$, the sub-matrix of $\tilde{Y}$ made from the rows j such that the rows Z_j are in cluster C_l.

iii Compute the mapping function:
Define the mapping function $\Psi(x, y) = \sum_{l=1}^{k} \alpha_l(x) * \Psi_l(y)$ where $\alpha_l(x) = exp(-s \sum (\frac{d_l}{\sigma_l})^2)$ (see eq. 2.12) is a coefficient that decreases with the distance between x and the cluster C_l; $\Psi_l(y) = \Phi^{-1}(\int_{-\infty}^{y} d_{est,l}(u).du)$ and Φ is the cumulative density function of the standard univariate Gaussian.

iv Transform the original data:
Consider now the vector $W := [w_1, .., w_N]^T$ where $w_j = \Psi(\bar{x}_j, y_j)$ and $\bar{x}_j, y_j$ are respectively the jth row of $\tilde{X}$ and $\tilde{Y}$.

v Fit a Gaussian Process:
Fit a GP on the data ($\tilde{X}$,W) via Maximum Likelihood Estimation. Choose either the *squared exponential* or *exponential periodic* covariance function.

vi Predict:
To predict the value y at a given point $x = [x_1, .., x_m]$, first normalize x, $\tilde{x} := [\tilde{x}_1, .., \tilde{x}_m]$

where $\tilde{x}_i = \frac{x_i - m_i}{\sigma_i}$. Then use the fitted GP to predict the values of $\mu(\tilde{x})$ and $\sigma(\tilde{x})$ representing the mean and standard deviation of the Gaussian posterior at point $\tilde{x}$. Finally numerically inverse $\Psi(\tilde{x}, .)$ and compute the expectation $\tilde{y}$ from eq. (2.15).

vii Return $y := m + \sigma * \tilde{y}$.

5.2 Handling cross-validation results and multiple evaluations

The algorithm above needs to be slightly updated when we consider the hyper-parameter optimization process. Indeed when dealing with cross-validation results, we have a vector of observations $\bar{y}$ instead of a real one y. Moreover, it is possible that the Smart Search process select more than once a given hyper-parameter, and in that case we would have more than one CV results. So actually Y is a list for which each entry may have a different length. The first solution is to simply replace $\bar{y}$ by its mean and use the algorithm presented above. However we have proposed some methods to leverage such information, and here is the description of how it modifies the nGCP algorithm:

- Use all CV results to build the mapping function:
  The aim is to compute the KDE estimation of the density function by taking all pairs (hyper-parameter,one of the CV results) into account. To do that, we duplicate the rows in $\tilde{X}$ as many times as we have observation for the corresponding hyper-parameter (a hyper-parameter is a row of X) to obtain the matrix X', and consider the vector Y' of all the CV results that appear in Y. We then use X' and Y' as $\tilde{X}$ and Y in point *ii.* in the algorithm above.

- Model the noise through the EGN method:
  We compute the values of $\sigma_1, .., \sigma_N$ in eq. (4.5) as the standard deviation of each list entry in Y, *ie.* the standard deviation of all CV results for a given hyper-parameter. Then we modify accordingly the covariance matrix when fitting the GP, as shown in eq. (4.5).

- Fit the GP with all CV results:
  In that case we completely replace $\tilde{X}$ and Y by X' and Y'. We only compute the means and standard deviations in step *i.* with X and Y_μ where Y_μ is the mean of each list entry in Y. Note that this method is not compatible with the EGN.

5.3 Smart Search

Process overview:

i Random initialization:
Sample uniformly inside the bounded hyper-parameter space N_{init} hyper-parameters and compute a performance evaluation for each of them.

ii Perform N_{smart} *smart* iterations:
Model the performance function with an n(L)GCP given all the hyper-parameters already tested and their performance evaluations. Use the fitted n(L)GCP to compute the value of the acquisition at $n_{param-sampling}$ random candidates uniformly sampled in the hyper-parameter space. Select the candidate that maximizes the acquisition function and compute its performance evaluation.

iii End with N_{final} *smart* iterations for which the acquisition function is simply the predicted performance.

Chapter 6

Experiments and Results

6.1 Comparing n(L)GCP and GP for regression

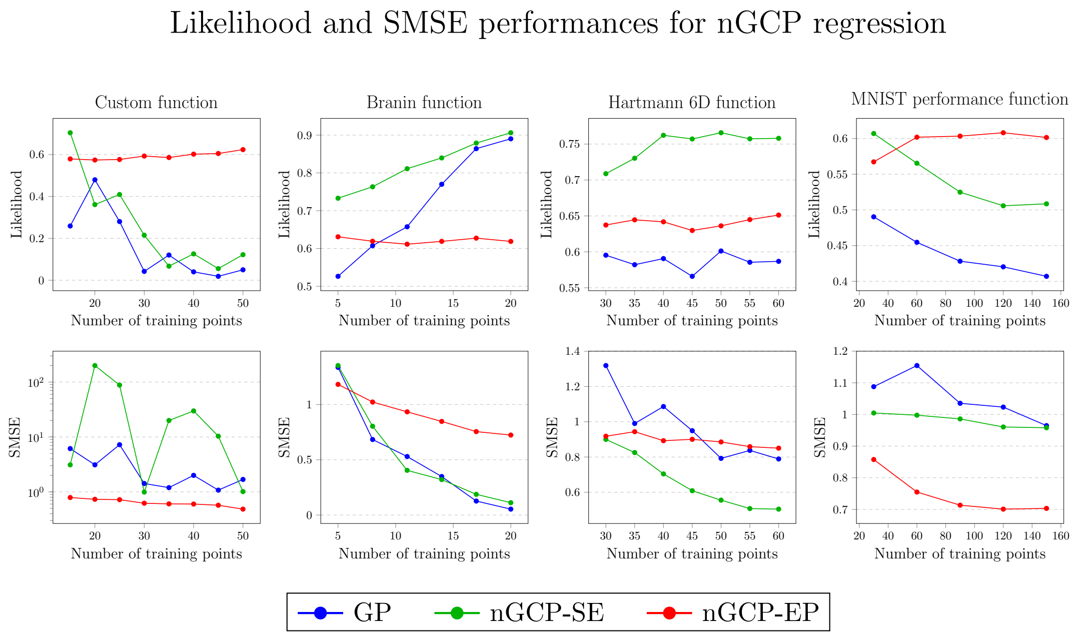

Figure 6-1: The first row displays the likelihood of the training data given the fitted model. The second row displays the Standardized Mean Squared Error, which is computed from predictions at 1000 randomly chosen points. The results shown are the average over 20 trials. SE and EP refers to the covariance function used.

In this section, we compare n(L)GCP and GP through their fitting and predictive power in a regression perspective. For its common use in Bayesian learning, we use the likelihood of the training data given the fitted model to assess the ability of a model to fit the data. Its computation with Gaussian processes is described in [16]. As the mapping function Ψ is part of the model with n(L)GCP, we can compute the likelihood with the same method, although the GP is fitted on $(X, \Psi \circ f(X))$. In addition we compute the Standardized Mean Squared Error (SMSE) to assess the prediction accuracy. This is the square of the difference between the predicted and the true values of the function, divided by the variance of the true values.

Figure 6-1 compares the nGCP with the GP on four test functions. The *custom function* is the one-dimensional function used in Fig. 2-3. The Branin and Hartmann functions are two commonly used functions to test optimization processes; the input spaces are respectively two- and six-dimensional.

Figure 6-2 also compares nLGCP through likelihood and SMSE for 1, 2 and 3 clusters to model the *custom function*. We observed that increasing the number of clusters always decreased the model's likelihood. However this was not necesseraly correlated to a lower predictive accuracy.

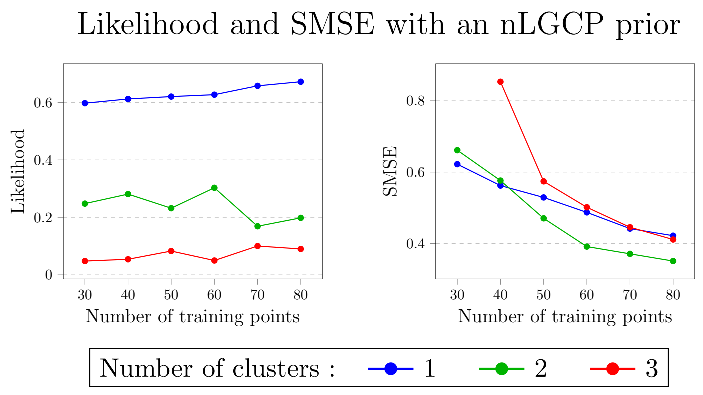

Figure 6-2: Likelihood and SMSE for an nLGCP prior with 1, 2 and 3 clusters to model the *custom function*. The results shown are the average over 20 trials, with the exponential periodic kernel, and predictions are made at 1000 random points.

6.2 Experiments' Methodology for Hyper-Parameter Optimization

We describe in this section how we ran the experiments. The methodology used can be divided into two main parts: an *off-line* one, which consists in aggregating as much information as we can about the pipeline, and an *on-line* one which consists in sampling hyper-parameters in the optimization framework. The first part aims at running a lot of experiments (the second part) much more quickly.

6.2.1 Aggregating information on a pipeline

We actually restrict the hyper-parameter space to a grid, and the aim is basically to get a performance score for each point of the grid. To do so, we pick iteratively, at random, a hyper-parameter $\bar{p}$ and evaluate it as described in section 3.2: this returns an evaluation E, which depends on the sub-sampled data, of k-fold cross-validation results $c_1, ..., c_k$. After running this for a while, we get a database where each entry corresponds to a hyper-parameter $\bar{p}$ and contains a certain number of evaluations $E_1, ..., E_{n_{\bar{p}}}$. We do this during enough time to get approximately at least a score per hyper-parameter of the space, so that we have multiple evaluations for a good proportion of them.

6.2.2 On-line optimization process

The optimization process asks iteratively for the performance evaluation of hyper-parameters. Thanks to the off-line work, this only consists in retrieving a value from the database. The result retrieved for $\bar{p}$ corresponds to all cross-validation results $c_1, ..., c_k$ of a given evaluation E of the nearest neighbors of $\bar{p}$ that has at least one entry in the database. If the corresponding hyper-parameter has multiple entries E, we choose at random. However we do not mix the cross-validation results c_i which belong to different evaluations. In particular this prevents from returning misleading results, for example only high c_i for a hyper-parameter that over-fits.

6.2.3 Presentation of the results

When analyzing a hyper-parameter optimization process, we are interested in observing:

i The quality of the hyper-parameters explored during the process.

ii The ability to select hyper-parameters that are *really* good, with the limited knowledge of the process.

We recall that when running the optimization process, we only get evaluations of the hyper-parameters on sub-samples of the dataset, which means that we have a limited knowledge on the hyper-parameters. However to analyze the results, we suppose that we know the ground truth thanks to the off-line computations described above.
So point *i.* can be done by considering either the limited knowledge of an experiment or the ground truth, and point *ii.* can be done by ranking the hyper-parameters through the different quality functions described in section 3.3.

Moreover, we can interpret the quality of a hyper-parameter through several tools:

i The absolute performances achieved by the corresponding pipeline through:

- the mean of all cross-validation results,
- a balanced score between mean and standard deviation,
- the values given by the functions $\tilde{\mathcal{Q}}$ and $\tilde{\mathcal{Q}}_s$ (3.3).

ii The relative gain on the hyper-parameter space, where a 0% gain corresponds to the minimum performance and 100% corresponds to the maximum reachable. The performance can still be computed by one of the ways listed in *i.*

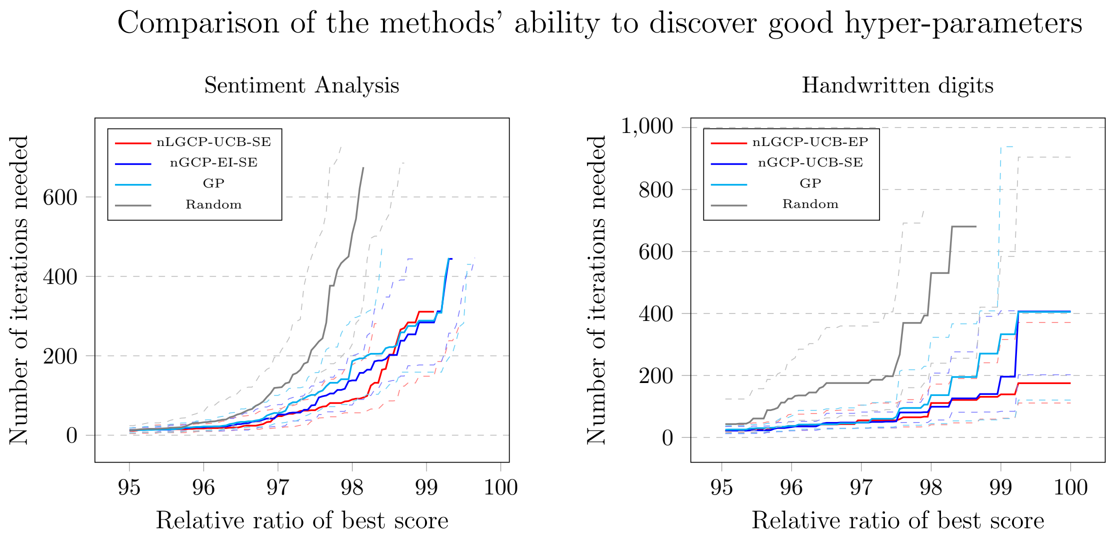

Figure 6-3: Number of hyper-parameters tested to attain a certain gain, defined as the relative improvement in performance from the minimum to the maximum reachable on the grid space. The median over several trials are displayed, as well as the firsts and thirds quartiles in dashed lines. UCB/EI refers to the acquisition function while SE/EP refers to the covariance function.

6.3 Results

6.3.1 On the quality of tested hyper-parameters

Figure 6-3 shows the performances of the n(L)GCP model compared to a GP-based optimization process and a random grid search. This figure shows the number of iterations needed (*ie.* the number of hyper-parameters to test) in order to reach a given relative gain (as defined in section 6.2.3). More results are shown in Appendix B.

Figure 6-4 also displays the number of iterations needed but divided by the number of iterations needed with a GP-based method to ease comparison.

Figure 6-5 presents how the different methods explore the hyper-parameter space during the process. While the random grid search visits unbiasedly each hyper-parameter, it is encouraging to see that *smart* processes visit more good hyper-parameters.

We observe that n(L)GCP-based methods outperform similar GP-based ones. In particular, Fig. 6-5 shows that the GP-based method visits less the good hyper-parameters. Although this could come from a different acquisition function (which could insist more on exploration than exploitation), one should note that here it only comes from less accurate predictions, as the only difference with the nGCP-UCB trials is the GP or nGCP model.

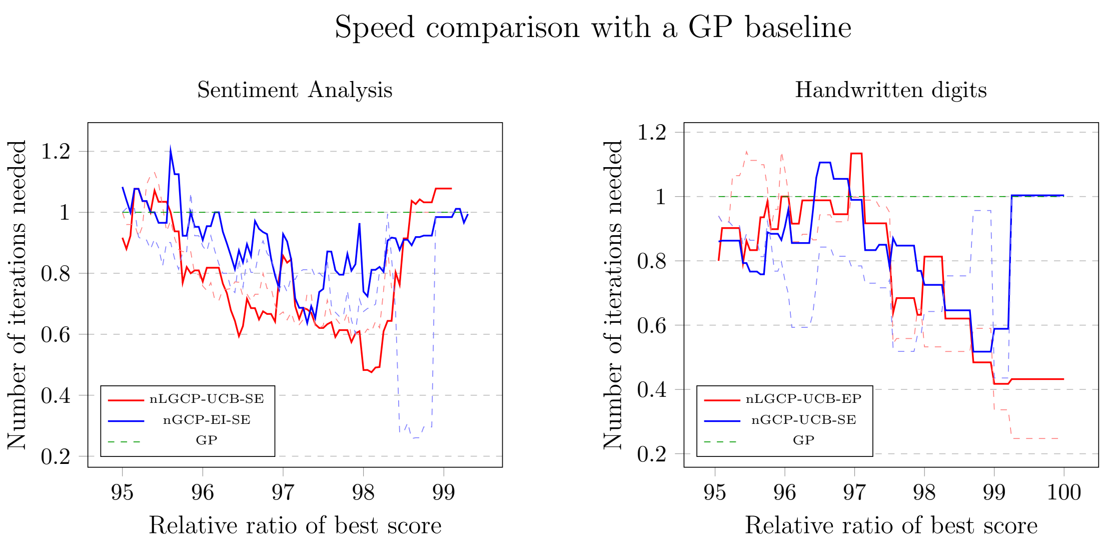

Figure 6-4: Number of hyper-parameters tested to attain a certain gain, in comparison to the number of iterations needed with a GP. The median over several trials are displayed, as well as the thirds quartiles in dashed lines.

6.3.2 On information gain

As we can see on Figure 6-6, using an nGCP provides a greater confidence gain. We evaluate the confidence/information acquired during the search through the maximum of the quality function $\tilde{\mathcal{Q}}_s$. As a matter of fact this function measures both the quality of a hyper-parameter but also takes into account the neighbors tested.

One of the most significant weaknesses of the random grid search is also shown here: it does not test enough the neighbors of good hyper-parameters. Thus such a process provide much less information on important hyper-parameters, which is even more crucial when dealing with noisy estimations.

6.3.3 On ranking hyper-parameters with a trial's limited knowledge

Figures 6-7 and 6-8 show the correspondence between the rankings made with the limited knowledge of a trial, through functions $\tilde{\mathcal{Q}}$ and $\tilde{\mathcal{Q}}_s$, and the truth values (computed thanks to the off-line work described in section 6.2) of function $\tilde{\mathcal{Q}}$. Fig. 6-7 displays the ratio between the true score and the best true score attained during the trial. Fig. 6-8 displays the true rank (based on true scores). We can see ranks are kept with more difficulty, but what really matters is the quality of the hyper-parameters. Regarding this, we see that the maximum reachable during the trial is almost always in the two first positions. Finally we observe that there is not a great difference between $\tilde{\mathcal{Q}}$ and $\tilde{\mathcal{Q}}_s$, although $\tilde{\mathcal{Q}}_s$ can still help a little in selecting good hyper-parameters.

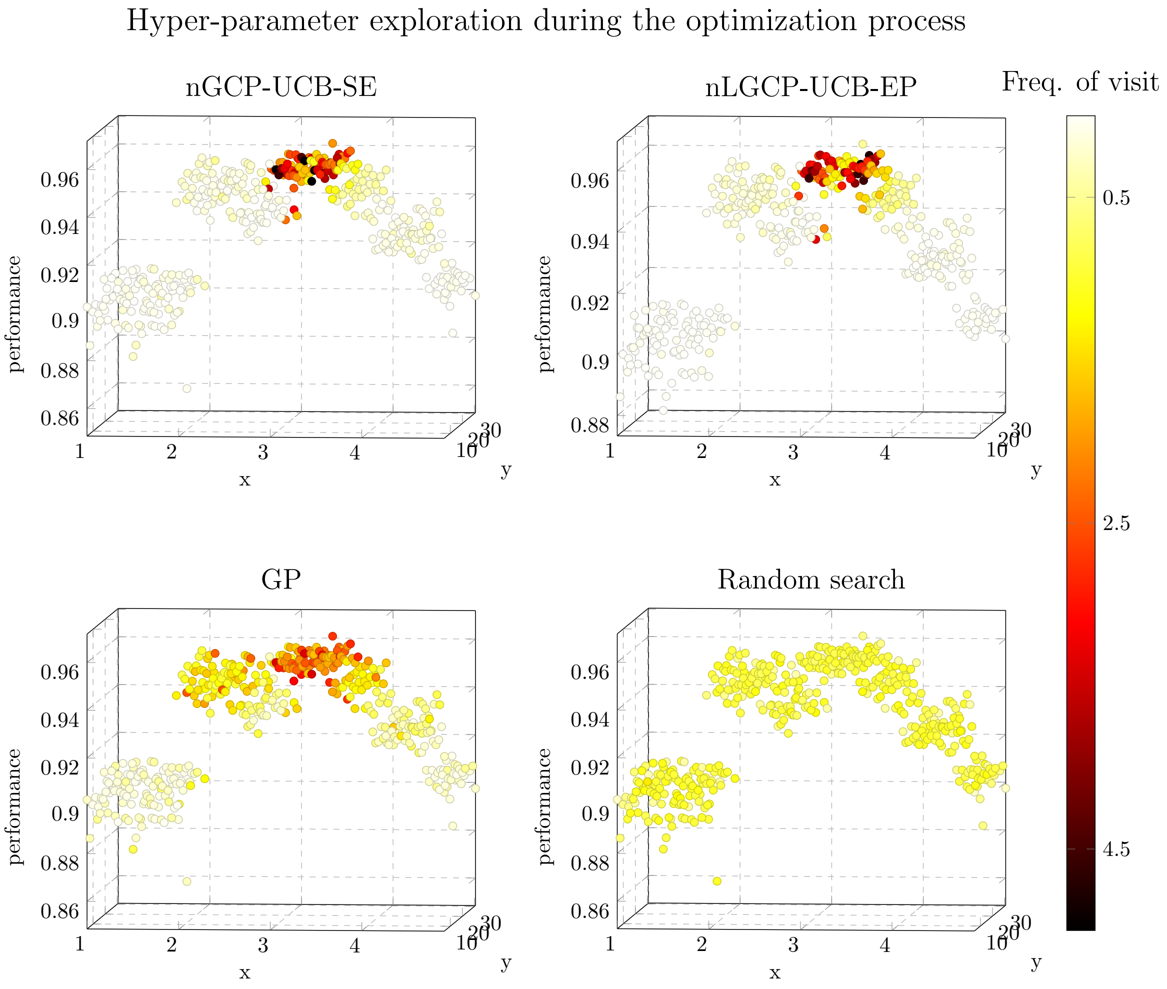

Figure 6-5: Hyper-parameters tested during several trials on the sentiment analysis problem. Hyper-parameters are represented as in Fig. 3-2 in a 2D space, while the z-axis corresponds to the real score of the hyper-parameter. Finally the color corresponds to the frequency of the visits (for visualization purposes, close points are clustered and the frequencies summed).

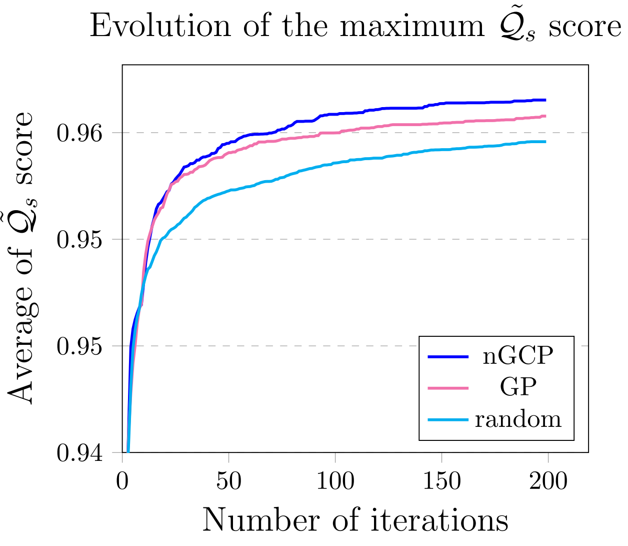

Figure 6-6: Information gain - Average of absolute $\tilde{\mathcal{Q}}_s$ score on several trials. The score is computed with the limited knowledge of a trial, and using only the neighbors already tested at iteration i.

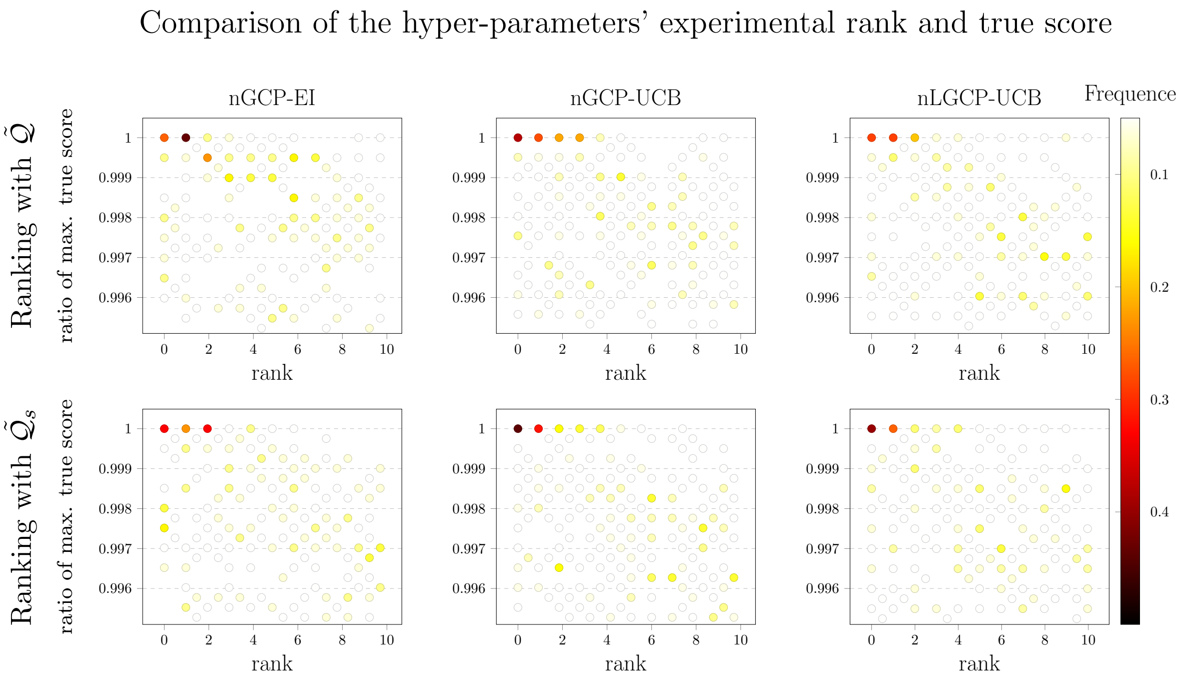

Figure 6-7: On each trial, parameters are ranked with $\tilde{\mathcal{Q}}$ or $\tilde{\mathcal{Q}}_s$, and the percentage of the best *real* score reached during the hyper-parameter exploration is displayed as a function of the rank. These are the results on several trials on the sentiment analysis problem, for different sub-methods. The color corresponds to the frequency of a given pair (rank,percentage of best real score reached) over the multiple trials. The parameters used are: $\tau = 0.5, \alpha = 0.5, n_c = 3, r_c = 200, \beta = 5$.

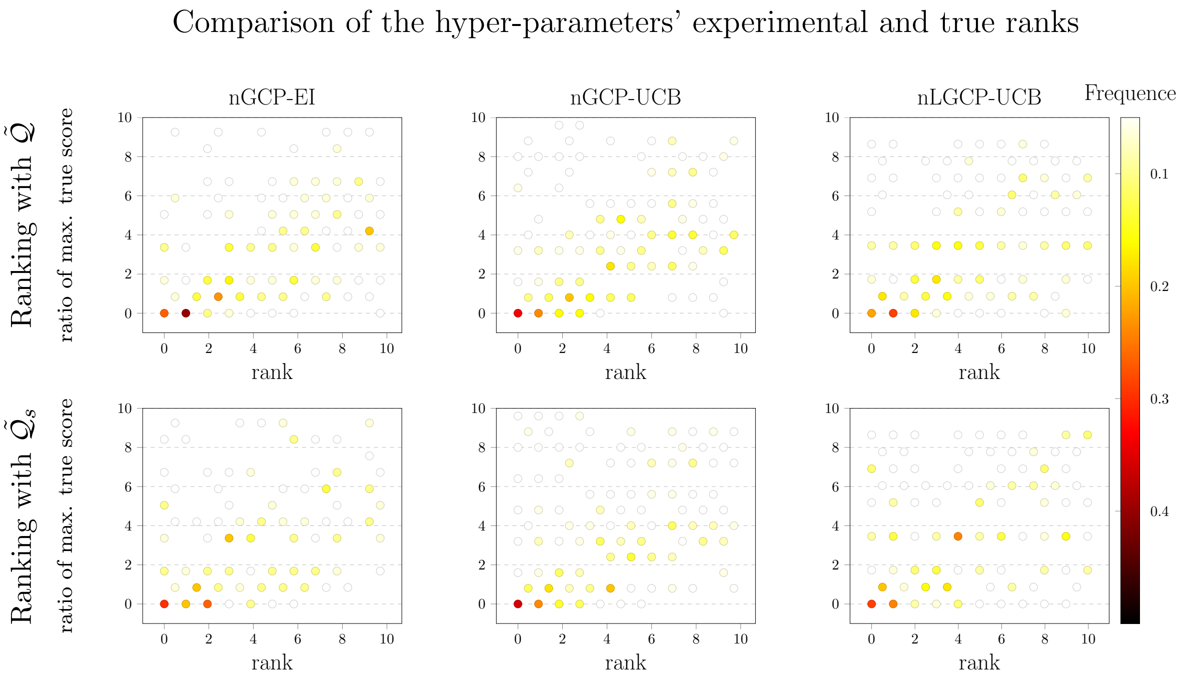

Figure 6-8: The *real* rank, computed from the *real* scores, as a function of the rank computed from $\tilde{\mathcal{Q}}$ or $\tilde{\mathcal{Q}}_s$ and the limited knowledge of a trial. These are the results on several trials on the sentiment analysis problem, for different sub-methods. The color corresponds to the frequency of a given pair (rank,real rank) over the multiple trials. The parameters used are: $\tau = 0.5, \alpha = 0.5, n_c = 3, r_c = 200, \beta = 5$.

Chapter 7

Conclusion and future work

7.1 Conclusion

We introduced the n(L)GCP, a new machine learning method based on Gaussian Copula Processes. Comparisons with Gaussian Processes for regression purposes proved that nGCP usually fit better to the data, and thus should yield higher prediction accuracy. We also presented how to use n(L)GCP for Bayesian Optimization. While the latent model usually yield lower likelihoods, meaning that the model is supposed to fit worse to the data, we saw that it could yield better prediction accuracy. In particular, latent models seemed to leverage the data's distribution learning through the acquisition functions: for example in Figure 4-2 we see that even if predictions do not change much when the number of clusters increases, the expected improvement curves change a lot and become more accurate.

We also presented a complete framework to perform hyper-parameter optimization on machine learning pipelines. By introducing the *noisy performance estimation* approach, this framework becomes easily parallelizable. This particular approach also implied to focus on noise management for n(L)GCP models.

Finally we tested our methods on two real-world problems, proving that sequential optimization processes outperform the randomized search, *ie.* that machine learning pipelines can and thus *must* be mined.

7.2 Future work

We hope to see a great appeal in this work and present here some ideas of what we think would benefit to this domain:

- Boost the computations to learn the mapping function for n(L)GCP.
- Try other latent models, for example by changing the definition of the α coefficients, and compare them. Design an automated method to learn the number of clusters and the smoothing coefficient.

- Consider a fully Bayesian treatment of the n(L)GCP model (*ie.* set a prior/posterior distribution of the model's parameters instead of learning them by maximum likelihood estimation) to compute the acquisition functions. Consider also direct approaches to maximize the acquisition function instead of sampling random candidates.
- Investigate in methods that take the evaluation time in consideration.
- Design a parallel auto-tuning machine for scaling purposes.

Appendix A

Machine Learning - Complement

A.1 Gaussian Blur

The Gaussian Blur is a smoothing method frequently used in image processing.
This operation is done by applying a filter to the input image. A linear filter transforms the input pixel values $f(i, j)$ into $g(i, j)$ by the following rule :

$$g(i, j) = \sum_{x,y} f(i + y, j + y)K(x, y),$$

for x, y in the filter's window. K is the called the kernel and shapes the blur by defining the smoothing coefficient in function of the distance with the input pixel.

A Gaussian blur means that the kernel is a Gaussian :

$$K(x, y) = Aexp\big(- \frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\big).$$

A.2 Bag-of-ngrams

The *Bag-of-Words* is a method to represent text data. The idea is to represent a document (a sequence of words and tokens) as a real vector. To do so, a dictionary is first built from an ensemble of documents, and documents are then represented as a set of words with their occurrence counts. An *n-gram* is simply a contiguous sequence of n items.

A.3 Tf-Idf

Tf-Idf means Term Frequency - Inverse Document Frequency, and aims at computing the relevance of a document given a query. The query is decomposed in terms (the words) and the total score is computed as the sum of each term's score.

Term frequency refers to the number of times a term occurs in a document. But only taking into account the term frequency has the drawback to incorrectly give too much importance in terms that are very frequent in the language (like 'a', 'the', 'of', ...).

To counter this, each term t is weighted through its inverse document frequency $idf_t = log(\frac{N_d}{df_t})$, where N_d is the number of documents and df_t is the number of documents that contain t. This way, rare words will have a high *idf* score and frequent words will have a low *idf* score.

A.4 Principal Component Analysis (PCA)

The aim of PCA is to reduce the dimension of the data while preserving the maximum variance possible among the studied data. Variance is indeed useful in machine learning as it should ease clustering and pulling apart, but reducing the dimension is often useful (*curse of dimensionality*, computation costs,...).

The eigenvectors of the data's covariance matrix form an orthogonal basis. The eigenvector with the highest eigenvalue actual corresponds to the direction in which the data have the highest variance. The projection of the data on this line is the first component. Following this idea, a PCA in dimension d is done by projecting the data on the subspace linearly spanned by the first d eigenvectors (ranked by decreasing eigenvalues).

A.5 χ^2 test

In machine learning, the chi-squared (χ^2) test is used to measure how much a variable is correlated with the result. This is mainly used for *feature selection* purposes, *ie.* to select the most relevant subset of features in order to reduce the dimension of the data without loosing useful information.

For categorical data, the test counts the number of instances $O_{i,j}$ that have the value i and belong to the class j. If $E_{i,j}$ if the expectation of $O_{i,j}$, *ie.* $N.p_i.p_j$ where N is the total number of instances, p_i is the probability for the feature to have the value i and p_j is the probability for the instance to belong to class j, then the χ^2 score is :

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}.$$

The higher this score is, the more correlated the feature is with the class. Thus this enables to rank the features by decreasing order so that the first features should be the most relevant to predict the class of an instance.

A.6 Polynomial Support Vector Machines (SVM)

With a polynomial kernel, the scalar product $\langle x, x' \rangle = k(x, x')$ where:

$$k(x, x') = (\gamma \langle x, x' \rangle + r)^d$$

A.7 Multinomial Naïve Bayes Classifier

This presentation comes from Scikit-Learn's documentation [1].

A.7.1 Naïve Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship: $P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots x_n|y)}{P(x_1, \dots, x_n)}$ Using the naive independence assumption that $P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y)$, for all i, this relationship is simplified to :

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}.$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \Rightarrow \hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate P(y) and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set. The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of$P(x_i \mid y)$. In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters.

A.7.2 Multinomial Naïve Bayes

Multinomial Naïve bayes is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts or tf-idf vectors). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y. The parameters θ_y are estimated by a smoothed version of maximum likelihood, *ie.* relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n},$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y. The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

[1] Available at : http://scikit-learn.org/stable/modules/naive_bayes.html

A.8 Cross-validation

Cross-validation is a technique used to assess the quality of a predictive model. Basically the idea is to learn a model on a *training* set and measure its performance on a *test* set. But always taking the same training/test sets provide a biased evaluation of the model. That is why cross-validation aims at partitioning the data into several training/test pairs.

In particular, in k-fold cross-validation, the original set is randomly partitioned into k equal sized sub-samples. The model is then trained and tested k times and at each time, one of the k sub-samples is used for testing while the other $k-1$ are used for training. The advantage is that all the data are used at least once for training and testing.

Usually the average of 5-fold or 10-fold cross-validation results is used to assess the quality of the model. But one can also, for example, take the standard deviation of the k results (the smaller, the better).

A.9 Maximum Likelihood Estimation

The *Maximum Likelihood Estimation* is a method used to perform *model selection*. Given a model parametrized by θ to represent the random variables $X_1, ..., X_n$, and some observed values $x_1, ..., x_n$, the likelihood of θ is defined as :

$$L(\theta) = \mathbb{P}(X_1 = x_1, ..., X_n = x_n|\theta).$$

So the maximum likelihood estimate of θ is the value that maximizes L.

A.10 Kernel Density Estimation (KDE)

KDE is a non-parametric technique to estimate the probability density function p of a random variable. Given the data $x_1, ...x_n$, p is computed as :

$$p(x) = \sum_{i=1}^{n} K(\frac{x - x_i}{h}), \tag{A.1}$$

where K is a kernel and h is a smoothing parameter called the bandwidth. The Gaussian kernel K_G is such that $K_G(x) \propto \exp(-\frac{x^2}{2})$ and p integrates to one.

Appendix B

Results - Complement

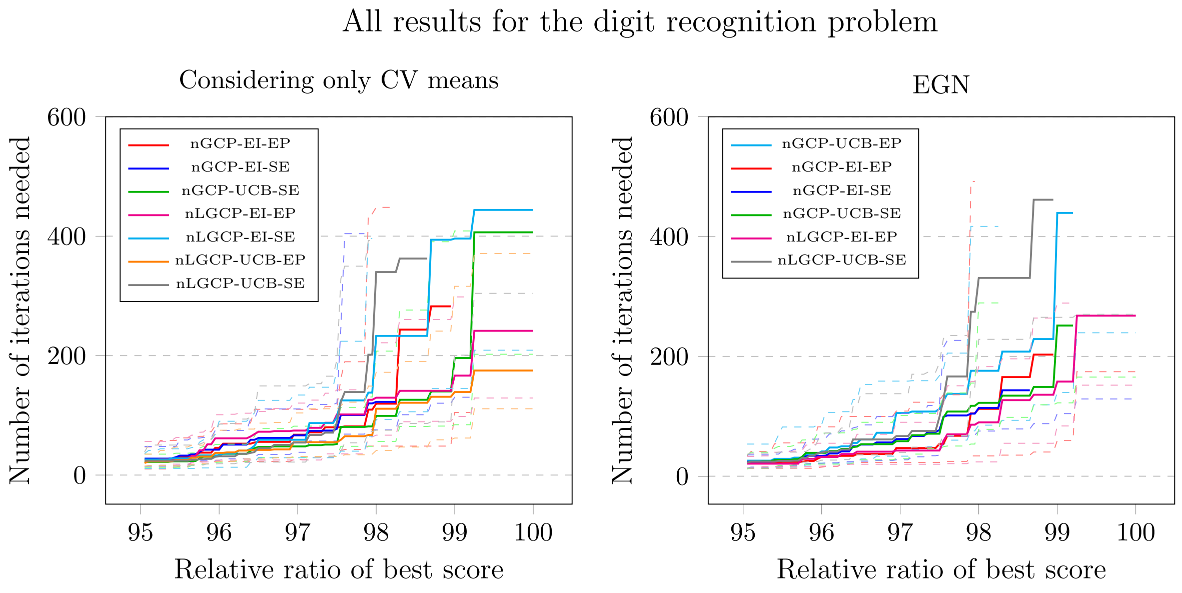

Figure B-1: Number of hyper-parameters tested to attain a certain gain, defined as the relative improvement in performance from the minimum to the maximum reachable on the grid space. The median over several trials are displayed, as well as the firsts and thirds quartiles in dashed lines. UCB/EI refers to the acquisition function while SE/EP refers to the covariance function. On the left plot we considered only the means of all cross-validation results while on the right plot we used the EGN method.

Appendix C

Code

The code is publicly available on Github at https://github.com/HDI-Project/DeepMining.
It is entirely written in Python and fully compatible with Scikit-Learn. As shown below it is really easy to use with Scikit-Learn's pipeline implementation, but it is also possible to define one's own estimator (a function that returns a (noisy or not) performance evaluation of a hyper-parameter set).

C.1 Run SmartSearch

```
# load data
from sklearn.datasets import fetch_20newsgroups
categories = ['alt.atheism','talk.religion.misc']
data = fetch_20newsgroups(subset='train', categories=categories)

# define a pipeline combining a text feature extractor
# with a simple classifier
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])

# define parameters
parameters = {
    'vect__max_df': ['float',[0.5,1.]],
    'vect__ngram_range': ['cat',[(1, 1), (1, 2)]],
    'tfidf__norm': ['cat',('l1', 'l2')],
    'clf__alpha': ['float',[0.000001, 0.0001]],
    'clf__penalty': ['cat',['l2', 'elasticnet']]
}

search = SmartSearch(parameters,
                        estimator=pipeline,
                        X=data.data, y=data.target,
                        n_iter=30)
search._fit()
```

C.2 SmartSearch Log

```
Loading categories
# ['alt.atheism',
# 'talk.religion.misc']
# 857 documents
Start random init
# Step 0 :
# {'vect__ngram_range': (1,2),
# clf__penalty':'l2',
# 'vect__max_df': 0.948133,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 4.57342e-05}
Score : 0.564769481844
# Step 1 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.573541,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 3.86617e-05}
Score : 0.949836801306
# Step 2 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.935701,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 4.98774e-05}
Score : 0.946300829593
# Step 3 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.619286,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 9.76227e-05}
Score : 0.753862369101
# Step 4 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.5643,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 1.84138e-05}
Score : 0.7445124439
# Step 5 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.990196,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 8.50129e-05}
Score : 0.949823201414
# Step 6 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.818634,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 6.16835e-05}
Score : 0.942826057392
# Step 7 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.571205,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 9.29654e-05}
Score : 0.772405820753
# Step 8 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.851345,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 3.8907e-05}
Score : 0.935849313205
# Step 9 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.744579,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 3.74336e-05}
Score : 0.924167006664

Start smart search
# Step 10 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.870831,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 2.16659e-06}
Score : 0.938147694818
# Step 11 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.544334,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 2.99991e-06}
Score : 0.93352373181
# Step 12 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.500498,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.87529e-05}
Score : 0.955671154631
# Step 13 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.696791,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 4.05229e-06}
Score : 0.932340541276
# Step 14 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.590258,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.7677e-05}
Score : 0.954501563987
# Step 15 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.512882,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 7.27651e-05}
Score : 0.954515163879
# Step 16 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.538971,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 7.60604e-05}
Score : 0.953318373453
# Step 17 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.503184,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.11519e-05}
Score : 0.954494764042
# Step 18 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.890506,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.9789e-05}
Score : 0.943975248198

# Step 19 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.998721,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 3.80934e-05}
Score : 0.945144838841

# Step 20 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.659229,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.60697e-05}
Score : 0.954494764042

# Step 21 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.665456,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.98543e-05}
Score : 0.949836801306

# Step 22 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.914834,
# 'tfidf__norm': 'l1',
# 'clf__alpha': 2.0786e-06}
Score : 0.933482932137

# Step 23 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.88294,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 5.63376e-05}
Score : 0.953311573507

# Step 24 :
# {'vect__ngram_range': (1,1),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.507396,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.30728e-05}
Score : 0.952169182647

# Step 25 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'elasticnet',
# 'vect__max_df': 0.554181,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.46619e-05}
Score : 0.954494764042

# Step 26 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.584624,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 9.31281e-05}
Score : 0.954481164151

# Step 27 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.987069,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 4.98404e-05}
Score : 0.948646810826

# Step 28 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.726414,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 6.28071e-05}
Score : 0.956820345437

# Step 29 :
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.74765,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 6.53554e-05}
Score : 0.955664354685

Tested 30 parameters
Max cv score 0.956820345437
# {'vect__ngram_range': (1,2),
# 'clf__penalty':'l2',
# 'vect__max_df': 0.726414,
# 'tfidf__norm': 'l2',
# 'clf__alpha': 6.28071e-05}
```

C.3 Scikit-Learn

This presentation is an excerpt of [17].

C.3.1 Introduction

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies and is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings. Source code, binaries, and documentation can be downloaded from http://scikit-learn.sourceforge.net.

C.3.2 Project Vision

- Code quality. Rather than providing as many features as possible, the project's goal has been to provide solid implementations. Code quality is ensured with unit tests—as of release 0.8, test coverage is 81%—and the use of static analysis tools such as pyflakes and pep8. Finally, we strive to use consistent naming for the functions and parameters used throughout a strict adherence to the Python coding guidelines and numpy style documentation.

- BSD licensing. Most of the Python ecosystem is licensed with non-copyleft licenses. While such policy is beneficial for adoption of these tools by commercial projects, it does impose some restrictions: we are unable to use some existing scientific code, such as the GSL.

- Bare-bone design and API. To lower the barrier of entry, we avoid framework code and keep the number of different objects to a minimum, relying on numpy arrays for data containers.

- Community-driven development. We base our development on collaborative tools such as git, github and public mailing lists. External contributions are welcome and encouraged.

- Documentation. Scikit-learn provides a ∼300 page user guide including narrative documentation, class references, a tutorial, installation instructions, as well as more than 60 examples, some featuring real-world applications. We try to minimize the use of machine-learning jargon, while maintaining precision with regards to the algorithms employed.

Bibliography

[1] T. M. Mitchell, *The discipline of machine learning*, vol. 17. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[2] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[3] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, pp. 2951–2959, 2012.

[4] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

[5] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*, pp. 507–523, Springer, 2011.

[6] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown, "Towards an empirical foundation for assessing bayesian optimization of hyperparameters," in *NIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

[7] W. Drevo, *Delphi: A Distributed Multi-algorithm, Multi-user, Self Optimizing Machine Learning System.* PhD thesis, Master's thesis, Massachusetts Institute of Technology, 2014.

[8] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2, pp. 1150–1157, Ieee, 1999.

[9] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *CoRR*, vol. abs/1012.2599, 2010.

[10] A. Wilson and Z. Ghahramani, "Copula processes," in *Advances in Neural Information Processing Systems*, pp. 2460–2468, 2010.

[11] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning.* Adaptive Computation And Machine Learning, MIT Press, 2005.

[12] E. Snelson, C. E. Rasmussen, and Z. Ghahramani, "Warped gaussian processes," *Advances in neural information processing systems*, vol. 16, pp. 337–344, 2004.

[13] B. W. Silverman, *Density estimation for statistics and data analysis*, vol. 26. CRC press, 1986.

[14] B. L. Welch, “The significance of the difference between two means when the population variances are unequal,” *Biometrika*, vol. 29, no. 3-4, pp. 350–362, 1938.

[15] Student, “The probable error of a mean,” *Biometrika*, pp. 1–25, 1908.

[16] H. B. Nielsen, S. N. Lophaven, and J. Sondergaard, “Dace, a matlab kriging toolbox,” *Informatics and mathematical modelling. Lyngby–Denmark: Technical University of Denmark, DTU*, 2002.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.