Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master's Thesis

Bayesian optimization in machine learning

José Jiménez Luna

supervised by
Josep Ginebra
Departament Estadística i Investigació Operativa. ETSEIB.

May 30, 2017

Universitat de Barcelona

Universitat Politècnica de Catalunya
BarcelonaTech

Abstract

Bayesian optimization has risen over the last few years as a very attractive approach to find the optimum of noisy, expensive to evaluate, and possibly black-box functions. One of the fields where these functions are common is in machine-learning, where one typically has to fit a particular model by minimizing a specified form of loss. In this Master's thesis we first focus on reviewing the most recent literature on Gaussian Processes as well as Bayesian optimization methods, then we benchmark said methods against several real case machine-learning scenarios and lastly we provide open source software that will allow researchers to apply these strategies in other problems.

Keywords: machine-learning, bayesian, optimization

There ain't no such thing as a free lunch.
Friedman. M. (1975)

Be brave, be bayesian.

Contents

Chapter 1

Organization of this work

1.1 Introduction

This Master's thesis aims to be a multi-objective optimization task:

- The first objective of the thesis is to provide the reader with an introduction to Gaussian Process regression and Bayesian optimization. While there are vast pieces of work for both Gaussian Processes and Bayesian Optimization, this work aims to bridge the gap between them. I try to cover as much literature as needed to provide the reader with enough background to understand and implement the theoretical work presented here. Explanations are accompanied by comprehensive coding implementations and examples that help understand the material.

- To show the Bayesian Optimization framework works in several real-world machine learning tasks. This is done by selecting several datasets related to open computational chemistry problems, following said methodology and finally comparing its performance to other already existing strategies.

- Finally, to write a complete software package for users to apply Bayesian Optimization in their research. This comes in the form of a Python (>3.5) package named pyGPGO. The code can either be obtained through its GitHub repository `https://github.com/hawk31/pyGPGO` or the Python Package Index (PyPI). The entire software package is MIT licensed. All the examples and code snippets throughout this manual are based on this software. While certainly there are a couple implementations of Global Optimization software in Python, the software developed here is modular, easy to use and requires minimal dependencies, while still being feature-wise competitive.

We begin by describing the title of this thesis. Bayesian Optimization focuses on the global optimization of a function $f : \mathbb{R}^n \to \mathbb{R}$ over a compact set $\mathcal{A}$. The problem can be formalized as:

$$\max_{\boldsymbol{x} \in \mathcal{A}} f(x) \tag{1.1}$$

Most optimization procedures (local based ones such as gradient ascent, for example) assume that the function f is closed-form, that is, it has an analytical expression, that it is convex, with known first or second order derivatives or cheap to evaluate. Bayesian optimization focuses on all these problems proposing a very elegant solution. By the use of a surrogate model, a Gaussian Process, a Bayesian optimization procedure can help find the global minimum of non-necessarily convex, expensive functions that are expensive to evaluate. These methods shine also where there is no closed-form expression to evaluate or derivatives.

At the same time, in machine learning (or statistical learning), we are usually interested in minimizing a loss function $\mathcal{L}$ over a subset of data. These losses can take many forms, e.g. when doing regression, a typical loss might be the mean squared error between predictions and observed values on a holdout test set.

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_i (y_i - \hat{y_i})^2 \tag{1.2}$$

In binary classification, for example, a very popular choice is the logarithmic loss:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{n} = \sum_i (y_i \log(\hat{y_i}) + (1 - y_i) \log(1 - \hat{y_i})) \tag{1.3}$$

Notice in any case, that these losses are typically defined in a subset of $\mathbb{R}$. In the work described here we focus on the supervised setting of machine learning. Depending on the problem at hand, even evaluating these losses can be very expensive from a computational point of view. This may have to do with the machine learning algorithm used or dataset size. These machine learning algorithms typically have *hyperparameters* that have to be tuned in a sensible way to get the best performance possible out of these models. In the machine learning community it is common for practitioners to perform hyperparameter grid lookups or randomized searches to reach reasonable solutions. However, with the advent of big-data and more computationally hungry strategies, the training of a single model could already take substantial resources in terms of CPU cycles or memory, that is translated in higher wall-clock waiting times. Therefore we would like to have a more efficient and cheap way to optimize these hyperparameters. Bayesian optimization will let us do that by proposing the next candidate hyperparameter set $\boldsymbol{x}$ to evaluate according to several criteria.

1.2 Organization of the thesis

The whole thesis is organized in 5 self-contained chapters. First, all the theoretical work is presented, for both Gaussian Processes and Bayesian optimization, then benchmarking of the method is presented, and finally we describe the developed piece of software. I briefly describe the content of each chapter here:

Chapter 2 focuses on a swift but thorough introduction to regression problems using Gaussian Processes. These are the surrogate models we will use for Bayesian Optimization in Chapter 3. We will mostly cover the theory behind them from a functional point of view. We will also explain different covariance functions and their role in these models. Special attention is given to different approaches towards covariance hyperparameter treatment. Further theoretical aspects are also discussed.

Chapter 3 is about the main topic in this work, Bayesian Optimization. Once we have laid down all the foundations of Gaussian Processes, we can start explaining the theory of Bayesian optimization using these as surrogate models. The role of several acquisition functions, i.e. functions that will propose the next point to evaluate will be thoroughly discussed, as well as their advantages or disadvantages. Different modelling choices are then further presented. References on this chapter will be very diverse, as I will try to summarize several recent publications on the field.

Chapter 4 covers experiments using the software provided alongside this manual. These are mostly mid-sized regression or classification problems where we will compare the performance of Bayesian Optimization of hyperparameters with several regressors/classifiers with other strategies, such as random search. Most of these datasets are related to the experimental sciences, in particular chemistry, and some of them were used for other benchmarking purposes in other studies.

Chapter 5 holds no theoretical content nor testing content. It will cover technical explanations of pyGPGO, the software developed alongside this manual. Usage examples are also provided.

Chapter 2

Gaussian Process regression

In this chapter we will focus on regression problems. Assume we have some labelled data

$$\mathcal{D} = \{(\boldsymbol{x_i}, y_i) \,|\, i = 1, \ldots, n\}\,, \tag{2.1}$$

where $\boldsymbol{x}$ is a vector of covariates and y denotes a continuous objective variable. We wish to learn a predictive distribution over new values of $\boldsymbol{y}$ given $\boldsymbol{x}$, so that we can make predictions and inference over these. In practice, for simplicity we write that $\mathcal{D} = \{X, \boldsymbol{y}\}$, where X is our predictor matrix.

One can interpret a Gaussian Process in several ways. The most widely known is the function space view, which is the one we will cover first here and the one we will assume for the rest for the thesis. In this view, we consider a Gaussian Process to be a stochastic process, hence, a distribution over functions, instead of over values. Inference takes place directly in this space. For completeness, we will also provide a weight-space view second, that might be more appealing to readers familiar with Bayesian linear regression.

2.1 A function space view for Gaussian Processes

We start by formally defining a Gaussian Process:

Definition 1 *A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution. This process is totally defined by two functions. Its mean function:*

$$m(\boldsymbol{x}) = \mathbb{E}\left[f(\boldsymbol{x})\right] \tag{2.2}$$

and its covariance function:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \mathbb{E}\left[\left(f(\boldsymbol{x}) - m(\boldsymbol{x})\right)\left(f(\boldsymbol{x'}) - m(\boldsymbol{x'})\right)\right] \tag{2.3}$$

We say that f is a Gaussian Process with mean $m(\boldsymbol{x})$ and covariance function $k(x, x')$ and write:

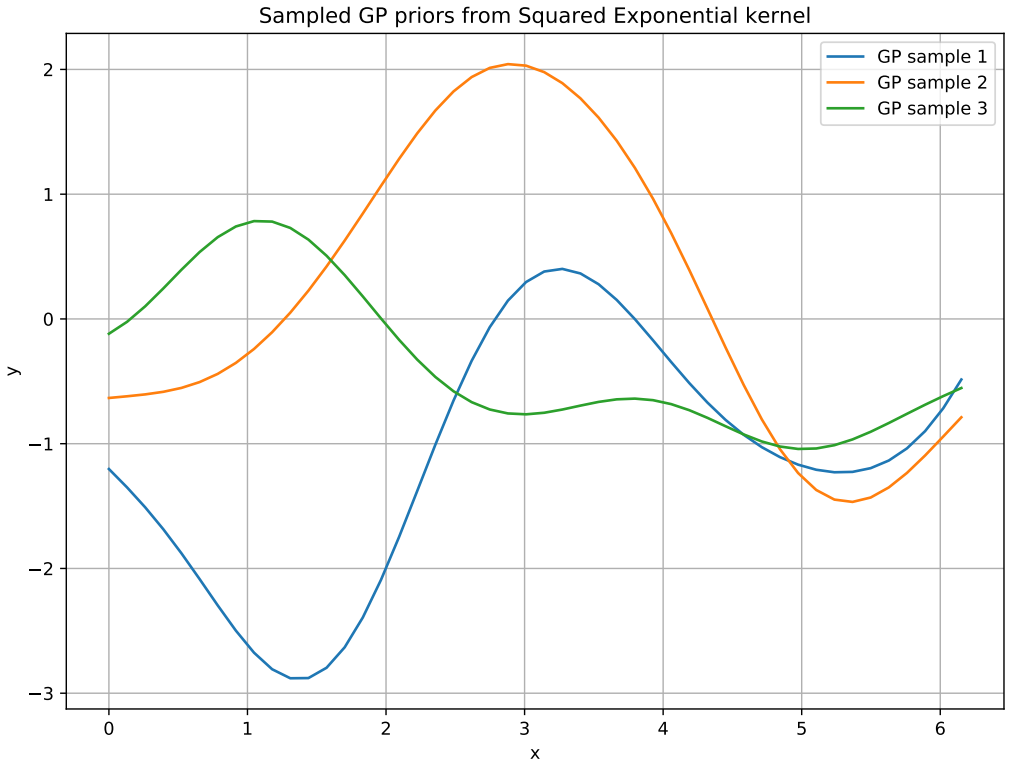
$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})\right) \tag{2.4}$$

In practice, for simplicity we will take $m(\boldsymbol{x}) = 0$, but this can be specified otherwise. Furthermore, a Gaussian Process fulfils the marginalization property, that is to say that if the the GP specifies $(y_1, y_2) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ then it follows that $y_1 \sim \mathcal{N}(\mu_1, \Sigma_{11})$. A Gaussian multivariate distribution is just a finite index set of a given Gaussian Process.

Define then a covariance function, such as the *squared exponential* kernel, written as:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}|\boldsymbol{x} - \boldsymbol{x}'|^2\right) \tag{2.5}$$

Figure 2.1: Three sampled Gaussian Process priors using the Squared Exponential kernel.



where $|.|$ denotes the standard L_2 norm. Most of the covariance functions that we will see here are a function of this norm, therefore it is much more comfortable to write $r = |\boldsymbol{x} - \boldsymbol{x}'|$ and therefore the squared exponential kernel becomes:

$$k(r) = \exp\left(-\frac{1}{2}r^2\right) \tag{2.6}$$

It is straightforward to draw samples from a Gaussian Process. In particular, since we work with a finite number of points, choose an arbitrary number of examples X_* and compute the squared exponential kernel (assuming $m(\boldsymbol{x}) = \mathbf{0}$). Then the procedure is simplified to sampling from the following multivariate Gaussian:

$$\boldsymbol{f}_* \sim \mathcal{N}\left(\mathbf{0}, K(X_*, X_*)\right) \tag{2.7}$$

We have written a very simple script to illustrate this point, which is available in Appendix A.1, producing Figure 2.1. Most of the code examples presented throughout the rest of the text were programmed using pyGPGO, the software developed alongside this thesis.

Before we move on, notice that the drawn functions in Figure 2.1 seem to have a characteristic length-scale. This can be interpreted as the distance one has to move in input space before the function value changes significantly. By default, the squared exponential kernel uses a characteristic length-scale of 1 ($l = 1$). To change this behaviour, it is sufficient to consider r/l instead of r in Equation 2.5. This can be thought as an hyperparameter to optimize. We will return to this problem in Section 2.5.

2.2 A weight space view for Gaussian Processes

In this section I will try to draw connections between Bayesian linear regression [2] and Gaussian Processes, through the use of kernel functions.

2.2.1 Standard Bayesian linear regression

A Bayesian linear regression model with Gaussian error can be formulated as:

$$y = X^T \boldsymbol{w} + \epsilon \tag{2.8}$$

where we typically assume $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$. This noise assumption directly implies a Gaussian likelihood, thus it can be easily proven that:

$$p(\boldsymbol{y}|X, \boldsymbol{w}) \sim \mathcal{N}(X^T \boldsymbol{w}, \sigma_n^2 I) \tag{2.9}$$

Assume now a Gaussian prior on the weights $\boldsymbol{w}$:

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \Sigma_p) \tag{2.10}$$

We are interested now on the posterior distribution of w, given both X and y, and assuming the model in Equation 2.7, that is:

$$p(\boldsymbol{w}|\boldsymbol{y}, X) = \frac{p(\boldsymbol{y}|X, \boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{y}|X)} \tag{2.11}$$

One can solve this problem by means of sampling procedures like Markov Chain Monte Carlo, but in this particular case, there is a closed-form solution. It can be proven that:

$$p(\boldsymbol{w}|X, \boldsymbol{y}) \sim \mathcal{N}\left(\frac{1}{\sigma_n^n} A^{-1} X \boldsymbol{y}, A^{-1}\right) \tag{2.12}$$

where $A = \sigma^{-2} X X^T + \Sigma_p^{-1}$. Notice that a simple MAP (maximum a posteriori) estimate of the weights can be obtained by just computing the mean of this distribution. Now, to make predictions for a particular test case $\boldsymbol{x_*}$, we average over all possible parameter values, hence we get a whole predictive distribution. Again, it can be shown that:

$$f_*|\boldsymbol{x_*}, X, \boldsymbol{y} \sim \mathcal{N}\left(\frac{1}{\sigma_n^2} \boldsymbol{x_*}^T A^{-1} X \boldsymbol{y}, \boldsymbol{x_*}^T A^{-1} \boldsymbol{x_*}\right) \tag{2.13}$$

2.2.2 Kernel functions in feature space

We have presented a very simple Bayesian approach to linear regression in the previous section. While useful, it lacks expressiveness due to its linearity. A very simple idea is to project this data into a higher dimension, where it may be more easily separated by a linear model of this sort. This is known as using the kernel trick [4]. We can do this through a covariance (or kernel) function $\phi(\boldsymbol{x})$. Note by $\Phi(X)$ the aggregation of columns after computing this kernel function in the entire dataset at hand.

The model becomes now:

$$f(\boldsymbol{x}) = \phi(\boldsymbol{x})^T \boldsymbol{w} \tag{2.14}$$

where we assume the same prior over $\boldsymbol{w}$ as in Equation 2.9. All the math presented in the previous section applies here, just placing $\phi(\boldsymbol{x})$ instead of $\boldsymbol{x}$. The predictive distribution over y becomes now, for example:

$$f_*|\boldsymbol{x}_*, X, \boldsymbol{y} \sim \mathcal{N}\left(\frac{1}{\sigma_n^2} \phi(\boldsymbol{x_*})^T A^{-1} \Phi \boldsymbol{y}, \phi(\boldsymbol{x_*})^T A^{-1} \phi(\boldsymbol{x_*})\right) \tag{2.15}$$

where for simplicity we have written $\Phi = \Phi(X)$ and $A = \sigma_n^{-2}\Phi\Phi^T + \Sigma_p^{-1}$. The predictive distribution needs to invert $N \times N$ matrix. Equation 2.14 can be rewritten as:

$$f_*|\boldsymbol{x}_*, X, \boldsymbol{y} \sim \mathcal{N}\left(\boldsymbol{\phi_*}^T\Sigma_p\Phi(K+\sigma_n^2 I)^{-1}\boldsymbol{y}, \boldsymbol{\phi_*}\Sigma_p\boldsymbol{\phi_*} - \boldsymbol{\phi_*}^T\Sigma_p\Phi(K+\sigma_n^2 I)^{-1}\Phi^T\Sigma_p\boldsymbol{\phi_*}\right) \tag{2.16}$$

where we have again simplified notation by $\boldsymbol{\phi_*} = \phi(\boldsymbol{x_*})$ and $K = \Phi^T\Sigma_p\Phi$. Now notice that the entries of K for both train and test set are of the form $\phi(\boldsymbol{x_*}^T)\Sigma_p\phi(\boldsymbol{x_*})$. We have implicitly defined now a *covariance function* of the form $k(\boldsymbol{x}, \boldsymbol{x'}) = \phi(\boldsymbol{x_*}^T)\Sigma_p\phi(\boldsymbol{x_*})$. This is in fact an inner product with respect to Σ_p. That is if we define $\psi(\boldsymbol{x}) = \Sigma_p^{1/2}(\boldsymbol{x})$, then a simple dot product representation of a covariance function is:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \psi(\boldsymbol{x})^T\psi(\boldsymbol{x'}) \tag{2.17}$$

where $\Sigma_p^{1/2}$ can be defined by means of a singular value decomposition. We then replace the original feature vectors by these dot products, *lifting* to a higher space. In the next Section, we will perform the same calculations detailed here, but using the function space view that will be used throughout the rest of the text.

2.3 Prediction using a Gaussian Process prior

In this particular section, arguably the most important one in the chapter, we will learn how to incorporate the knowledge of training data $\mathcal{D} = \{(\boldsymbol{x_i}, y_i)\,|i = 1, \ldots, n\}$ into our Gaussian Process to obtain a posterior predictive distribution. We will start considering the case that we have a noiseless function, that is to say, when $\sigma_n^2 = 0$. Let us define $K(X, X_*)$, the covariance function evaluated on train and test points, $K(X, X)$ the covariance function evaluated at only the training points, $K(X_*, X_*)$ equivalently defined for the test values. Notice the last two have to be square matrices by definition.

Let us also use the following theorem:

Theorem 1 *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be jointly Gaussian:*

$$\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\boldsymbol{\mu_x}\\ \boldsymbol{\mu_y}\end{bmatrix}, \begin{bmatrix}A & C\\ C^T & B\end{bmatrix}\right) \tag{2.18}$$

Then $\boldsymbol{x}|\boldsymbol{y} \sim \left(\boldsymbol{\mu_x} + CB^{-1}(\boldsymbol{y} - \boldsymbol{\mu_y}), A - CB^{-1}C^T\right)$

Similarly as in Equation 2.6, assume that $\boldsymbol{f}$ and $\boldsymbol{f_*}$ are jointly Gaussian:

$$\begin{bmatrix}\boldsymbol{f}\\ \boldsymbol{f_*}\end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix}K(X,X) & K(X,X_*)\\ K(X_*,X) & K(X_*,X_*)\end{bmatrix}\right) \tag{2.19}$$

We are interested now in the distribution of $\boldsymbol{f_*}|\boldsymbol{f}$. Simply applying Theorem 1, we can obtain:

$$\boldsymbol{f_*}|\boldsymbol{f} \sim \mathcal{N}\left(K(X_*,X)K(X,X)^{-1}\boldsymbol{f}, K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)\right) \tag{2.20}$$

This covers all the basics for a Gaussian Process regression model. Notice that now we have a complete predictive distribution over test values $\boldsymbol{f_*}$, and this provides us with plenty of choices. For example, one could obtain an estimate of this function by drawing samples from a multivariate normal with the computed posterior parameters, or obtain a MAP estimate using the posterior mean.

Let us now consider the scenario where observations are not noise-free, that is, each time the function is queried it comes with i.i.d Gaussian error with mean 0 and variance $\sigma_n^2 > 0$. Assume now the following prior on the noisy observations:

$$Cov(\boldsymbol{y}) = K(X,X) + \sigma_n^2 I \tag{2.21}$$

Following the exact operations as before, but taking into account this new term, we got the following joint distribution:

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f_*} \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{2.22}$$

And conditioning again $\boldsymbol{f_*}$ on $\boldsymbol{y}$, we obtain our final predictive distribution:

$$\boldsymbol{f_*}|\boldsymbol{y} \sim \mathcal{N}(\overline{\boldsymbol{f_*}}, Cov(\boldsymbol{f_*})) \tag{2.23}$$

where now:

$$\begin{aligned} \overline{\boldsymbol{f_*}} &= K(X_*,X)\left(K(X,X) + \sigma_n^2 I\right)^{-1}\boldsymbol{y} \\ Cov(\boldsymbol{f_*}) &= K(X_*,X_*) - K(X_*,X)\left(K(X,X) + \sigma_n^2 I\right)^{-1}K(X,X_*) \end{aligned} \tag{2.24}$$

It will probably be useful to note that a Gaussian Process model can be written in terms of a Bayesian hierarchical model, since:

$$\begin{aligned} \boldsymbol{y}|\boldsymbol{f} &\sim \mathcal{N}(\boldsymbol{f}, \sigma_n^2 I) \\ \boldsymbol{f}|X &\sim \mathcal{N}(\boldsymbol{0}, K(X,X)) \end{aligned} \tag{2.25}$$

In fact, one can also assume other priors, even over σ_n^2. This representation may help us understand the introduction of the *marginal likelihood*. This marginal likelihood in a Gaussian Process setting is defined as:

$$p(\boldsymbol{y}|X) = \int p(\boldsymbol{y}|\boldsymbol{f}, X)p(\boldsymbol{f}|X)d\boldsymbol{f} \tag{2.26}$$

Using the results from Equations 2.24 we can derive the integral analytically to obtain:

$$\log p(\boldsymbol{y}|X) = -\frac{1}{2}\boldsymbol{y}^T(K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi \tag{2.27}$$

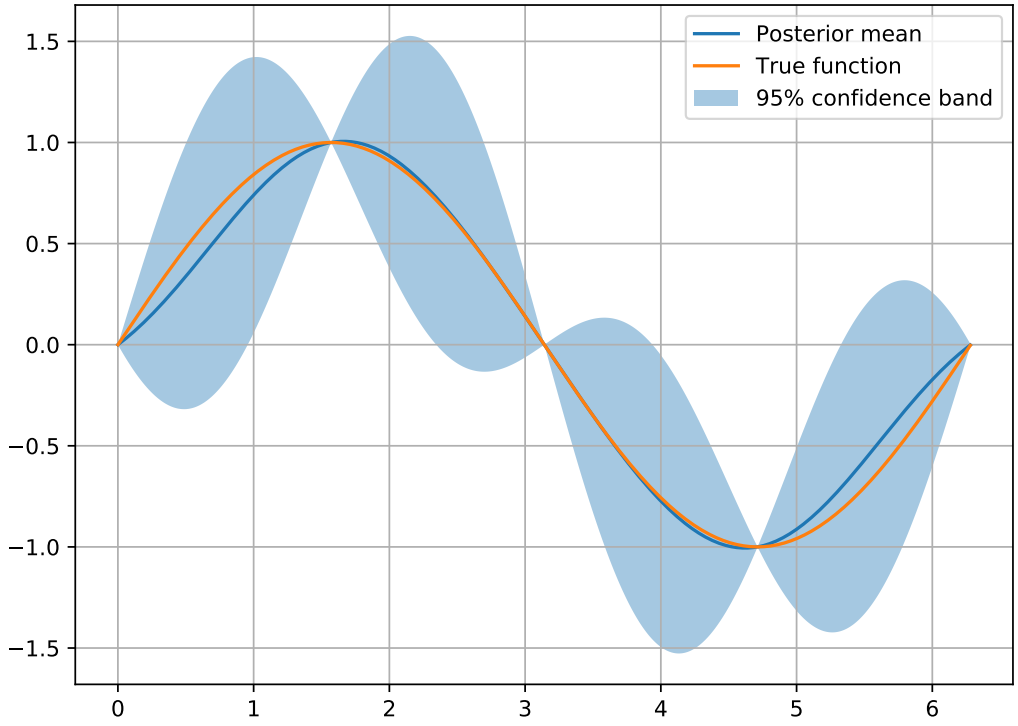
Notice this result is equivalent to the log-density of $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{0}, K + \sigma_n^2 I)$. We have now all the necessary ingredients to lay down pseudo-code for the implementation of a Gaussian Process regressor, as presented in Algorithm 7. It makes use of several tricks for computational stability, such as a Cholesky decomposition and several linear system of equations to avoid directly inverting matrices.

Algorithm 1 Gaussian regressor pseudo-code.

function GaussianProcess(X, $\boldsymbol{y}$, k, σ_n^2, $\boldsymbol{x_*}$)
  $L \leftarrow \text{chol}(K + \sigma_n^2 I)$
  $\boldsymbol{\alpha} \leftarrow \text{linsolve}\left(L^T, \text{linsolve}(L, \boldsymbol{y})\right)$
  $\overline{f_*} \leftarrow \boldsymbol{k_*}^T\boldsymbol{\alpha}$
  $\boldsymbol{v} \leftarrow \text{linsolve}(L, \boldsymbol{k_*})$
  $\mathbb{V}[f_*] \leftarrow k(\boldsymbol{x_*}, \boldsymbol{x_*}) - \boldsymbol{v}^T\boldsymbol{v}$
  $\log p(\boldsymbol{y}|X) \leftarrow -\frac{1}{2}\boldsymbol{y}^T\boldsymbol{\alpha} - \sum_i \log L_{ii} - \frac{n}{2}\log 2\pi$
end function

pyGPGO includes an implementation of a Gaussian Process regressor under the `surrogates.GaussianProcess` module. The entire module along with its functionality will be detailed in Section 5.1.

Figure 2.2: A fitted Gaussian Process regressor to samples of the sine function.



2.3.1 A toy example of Gaussian Process regression

Now that we have both the algorithm and the tools at hand, it may be interesting how a Gaussian Process regressor behaves with a toy example. We try to approximate a simple sine function in the interval $[0, 2\pi]$, and plot both the posterior mean and a 95% confidence band using the posterior variance of the fitted process. The code in Appendix A.2 produces Figure 2.2.

2.3.2 Picking a winner

In the previous section we have shown how to compute predictive posterior distribution for function outputs y_* given a new input $\boldsymbol{x}_*$. These are given by a Gaussian distribution with a certain mean and variance. In plenty of production settings, however, it is more common to provide a single value, or estimate y_{guess} that is *optimal* in some sense. To define a sense of optimality, define a loss function $\mathcal{L}(y_{\text{true}}, y_{\text{guess}})$. This, defines a penalty incurred by taking the decision to use y_{guess} when the true value is y_{true}. For example, this could be the mean square or mean absolute error function. In the Bayesian setting, there is no clear mention of a loss function in any stage. In the frequentist setting, however, a model is usually trained by minimizing this loss. Furthermore, there is a clear separation between loss and likelihood in the Bayesian setting, the latter used for training, with prior information. The loss function however only captures the consequences of making a single specific choice given a true state.

Again, we would like to somehow pick a *winner* y_{guess} that minimizes our loss. Without knowing y_{true}, our best choice is to minimize the expected loss, averaging with respect to our model:

$$\mathcal{R}_{\mathcal{L}}(y_{\text{guess}}|\boldsymbol{x}_*) = \int \mathcal{L}(y_*, y_{\text{guess}})p(y_*|\boldsymbol{x}_*, \mathcal{D})dy \tag{2.28}$$

Our optimal value is the one that minimizes this expected loss:

$$y_{\text{optimal}}|\boldsymbol{x}_* = \arg\min_{y_{\text{guess}}} \mathcal{R}_{\mathcal{L}}(y_{\text{guess}}|\boldsymbol{x}_*) \tag{2.29}$$

It can be proven that the value y_{guess} that minimizes Equation 2.28 for the absolute loss function is the median of $p(y_*|\boldsymbol{x}_*, \mathcal{D})$. For the squared loss function, it is the mean of the same distribution. Since in our case we are dealing with the Gaussian distribution, median and mean coincide, and the most reasonable *winner* will therefore be the specific value of the posterior mean.

2.4 On covariance functions

A covariance function [11], like the squared exponential kernel that we have been using as a example throughout the chapter encodes our assumptions of similarity between inputs from $\boldsymbol{x}$. We assume *similar* items in input space to have similar values of the target value y. Not all functions of $\boldsymbol{x}$ and $\boldsymbol{x'}$ can be defined as covariance function. Covariance functions (though not all) tend to satisfy different properties:

- *Weak stationarity.* A covariance function is said to be weakly stationary if it is a function of $\boldsymbol{x} - \boldsymbol{x'}$. That is to say that it is invariant to translations in the input space. Most of the covariance functions we will see fall into this category.
- *Isotropy.* A covariance function is said to be isotropic if it is *only* function of $|\boldsymbol{x} - \boldsymbol{x'}|$. Therefore, every isotropic covariance function is stationary.
- Dot-product. Some covariance functions are functionals of the dot-product $|\boldsymbol{x}^T\boldsymbol{x'}|$. These kernels, while invariant to rotations are not invariant to translations.

There is an excellent theoretical analysis of covariance functions in [9]. We will not cover this here since it falls beyond the scope of this thesis. However, we will start providing examples of the most common covariance functions. All covariance functions described here are implemented in the software developed alongside this thesis, pyGPGO, in the `covfunc` module. We will describe their functionality in Section 5.1.

The *Squared Exponential* covariance function is the one that we have been using so far. It is also arguably the most used in practice. It takes the general form:

$$k_{SE}(r) = \exp\left(-\frac{r^2}{2l^2}\right) \tag{2.30}$$

where l is the parameter controlling its characteristic length-scale. It is useful to define these functions in terms of r since we can abstract this calculation to another function.

The *Matérn class* of covariance functions [5] takes the form:

$$k_{\text{Matern}}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}r}{l}\right)^{\nu} K_{\nu}\left(\frac{\sqrt{2\nu}r}{l}\right) \tag{2.31}$$

with $\nu, l > 0$ and K_ν is a modified Bessel function of the second kind [1]. Simple functional forms can be obtained when ν is half integer, that is $\nu = p + 1/2$ for p non-negative integer. In particular, if $\nu = 1/2$, we obtain the a simple exponential kernel and if we take limit $\nu \to \infty$ we obtain the squared exponential covariance function. Popular values are $\nu = 3/2$ (once-differentiable) and $\nu = 5/2$ (twice-differentiable).

The γ-*exponential* covariance function, of which the squared exponential is a special case, takes the general form:

$$k_{\mathrm{GE}}(r) = \exp\left(-\left(\frac{r}{l}\right)^{\gamma}\right) \tag{2.32}$$

for $0 < \gamma \leq 2$.

The *Rational Quadratic* covariance function can be written as:

$$k_{RQ}(r) = \left(1 + \frac{r^2}{2\alpha l^2}\right)^{-\alpha} \tag{2.33}$$

with $\alpha, l > 0$. This covariance function can be seen as a scale mixture of squared exponential kernels with different length scales.

The *arcSin* kernel is an example of a dot product covariance function, therefore non-stationary:

$$k_{\mathrm{arcSin}}(\boldsymbol{x}, \boldsymbol{x'}) = \frac{2}{\pi}\sin^{-1}\left(\frac{2\boldsymbol{x}\Sigma\boldsymbol{x'}}{\sqrt{(1 + 2\boldsymbol{x}^T\Sigma\boldsymbol{x})(1 + 2\boldsymbol{x'}^T\Sigma\boldsymbol{x})}}\right) \tag{2.34}$$

Normally, these are the covariance functions that are used for the noiseless case of observation, that is, we know precisely that $f(\boldsymbol{x}_i) = y_i, \quad i = 1 \dots n$. In general, our covariance functions will take the form:

$$k^y(\boldsymbol{x}_p, \boldsymbol{x}_q) = \sigma_f^2 k(\boldsymbol{x}_p, \boldsymbol{x}_q) + \sigma_n^2 \delta_{pq} \tag{2.35}$$

where σ_f^2 is the signal variance, and controls the overall scale of our covariance matrix, σ_n^2 is the noise variance and δ_{pq} is a Kronecker delta function. Notice we note now k^y instead of k to account for noisy observations. In practice, all covariance function internal parameters plus $\left\{\sigma_n^2, \sigma_f^2\right\}$ can be considered unknowns of our particular problem. Several different treatments of hyperparameters can be considered, for example, one may choose to fix them manually, try to optimize them in a maximum-likelihood fashion (Section 2.5) or take the full Bayesian approach and marginalize over them (Section 2.6.3).

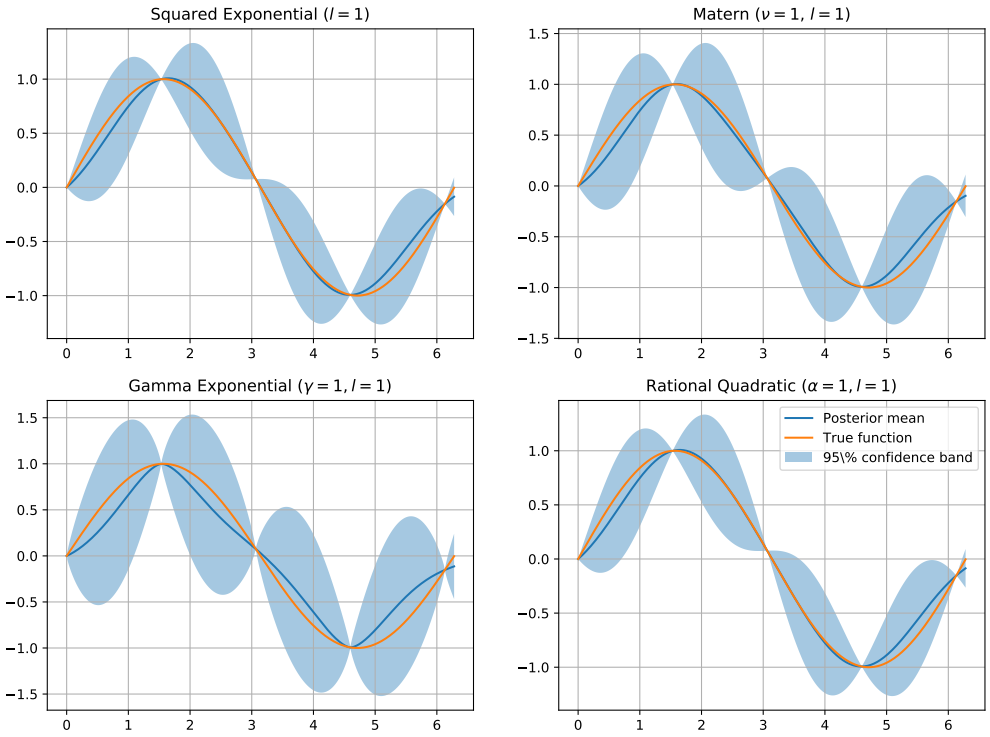
2.4.1 Visualizing different covariance functions

We have seen plenty of covariance function specifications in the last section. Remember that these control the degree of *similarity* between input points. As an exercise, it would be good to recreate the same sine function example that we saw before, using four different stationary different covariance functions. The choice of parameters is the default one in pyGPGO. The script detailed in Appendix A.3 below produces Figure 2.3.

2.5 Hyperparameter optimization

As seen in the previous sections, different covariance functions have different *hyperparameters*. These control how the kernel measures similarity among different instances of $\boldsymbol{x}$. So far, we have chosen these hyperparameters according to those set default in pyGPGO, but one may want to choose these according to training data. Depending on the situation, good parameter choices should lead to better models, either in terms of accuracy or interpretability. There are several ways to select these hyperparameters, by optimizing the marginal log-likelihood or via cross-validation. A full Bayesian treatment of hyperparameters is discussed in Section 2.6.3.

Figure 2.3: Behaviour of different stationary covariance functions with the default parameters in pyGPGO.



2.5.1 Type II Maximum Likelihood

This is the empirical Bayes [10] analytical approach to optimizing hyperparameters. One may quickly notice that Gaussian Processes are non-parametric models, in the sense that apart from the quantities set in the covariance functions, there is nothing else to optimize for. First we will provide a small background on Bayesian model selection. Assume that we have a model $\mathcal{H}_i$ with *parameters* $\boldsymbol{w}$, *hyperparameters* $\boldsymbol{\theta}$, and we have some training data $X, \boldsymbol{y}$. The posterior over the parameters is given by Bayes's theorem:

$$p(\boldsymbol{w}|\boldsymbol{y}, X, \boldsymbol{\theta}, \mathcal{H}_i) = \frac{p(\boldsymbol{y}|X, \boldsymbol{w}, \mathcal{H}_i)p(\boldsymbol{w}|\boldsymbol{\theta}, \mathcal{H}_i)}{p(\boldsymbol{y}|X, \boldsymbol{\theta}, \mathcal{H}_i)} \tag{2.36}$$

where $p(\boldsymbol{y}|X, \boldsymbol{w}, \mathcal{H}_i)$ is the likelihood, $p(\boldsymbol{w}|\boldsymbol{\theta}, \mathcal{H}_i)$ our prior distribution over the parameters and $p(\boldsymbol{y}|X, \boldsymbol{\theta}, \mathcal{H}_i)$ is called the *evidence* or marginal likelihood. Notice that this last quantity is nothing but the integral over parameter $\boldsymbol{w}$ space of the numerator in Equation 2.35.

We can do the same at the next level of inference for the hyperparameters. The posterior of hyperparameters is defined as:

$$p(\boldsymbol{\theta}|\boldsymbol{y}, X, \mathcal{H}_i) = \frac{p(\boldsymbol{y}|X, \boldsymbol{\theta}, \mathcal{H}_i)p(\boldsymbol{\theta}|\mathcal{H}_i)}{p(\boldsymbol{y}|X, \mathcal{H}_i)} \tag{2.37}$$

where now $p(\boldsymbol{\theta}|\mathcal{H}_i)$ is our prior over hyperparameters. We are interested however in optimizing the denominator in Equation 2.36 with respect to the hyperparameters. Typically, in Bayesian inference to perform the kind of integrals presented before, one has to resort to sampling procedures related to Markov Chain Monte Carlo, such as the Gibbs sampler. In the case of Gaussian processes, all computations are analytically tractable. In fact, the expression of the marginal likelihood was presented in Section 2.3. We reproduce the expression here for completeness:

$$\log p(\boldsymbol{y}|X) = -\frac{1}{2}\boldsymbol{y}^T(K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi \tag{2.38}$$

For typical local optimization methods to work fairly well, we may also need an specification of the derivative of the log-marginal likelihood w.r.t. the hyperparameters.

$$\frac{\partial}{\partial\theta_j}\log p(\boldsymbol{y}|X, \boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{y}^T K^{-1}\frac{\partial K}{\partial\theta_j}K^{-1}\boldsymbol{y} - \frac{1}{2}\mathrm{tr}\left(K^{-1}\frac{\partial K}{\partial\theta_j}\right) \tag{2.39}$$

where $\dfrac{\partial K}{\partial\theta_j}$ denotes the derivative of the selected covariance function, evaluated at each pair of instances of the training set. For optimization, one may choose to make use of this expression or not, depending on both of the optimization algorithm (gradient ascent, L-BFGS-B...) or on the cost of evaluation of the derivative. In pyGPGO, most of the covariance functions are implemented with a method `gradK` to return the gradient.

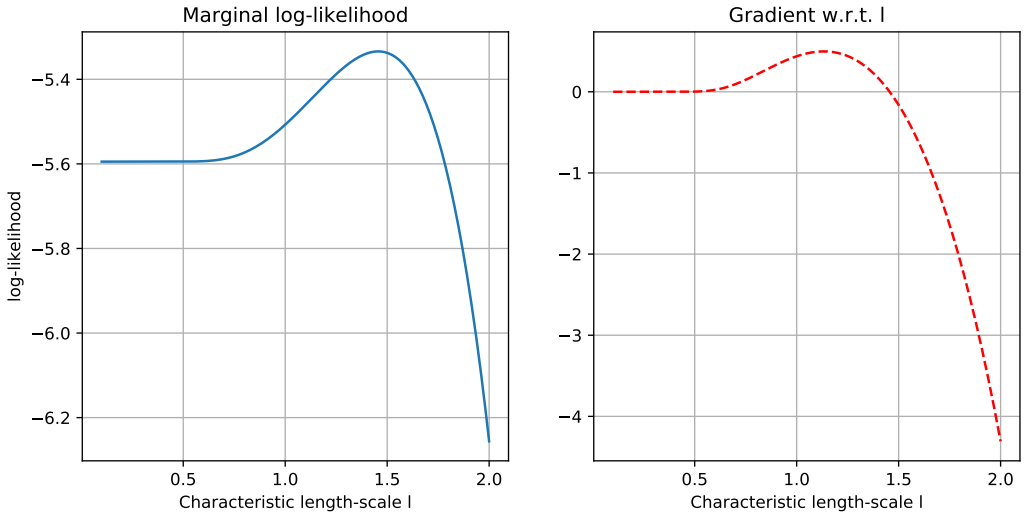
Another toy example: Optimizing the characteristic length-scale

To illustrate the previous point, it may be a good idea to see the behaviour of the marginal log-likelihood and its gradient when we modify the characteristic length scale l in the squared exponential covariance function. The sine function will also serve as playground here. The code in Appendix A.4 produces Figure 2.4.

2.5.2 Cross validation

We first lay down some very basic ideas related to model selection from a general machine learning perspective. The concepts presented here are more useful if one plans to use Gaussian Processes purely as a regression model. To evaluate the performance of hyperparameters of a machine-learning model $\boldsymbol{\theta}$, in $\mathcal{D} = \{X, \boldsymbol{y}\}$ one could do the following:

Figure 2.4: Log-marginal likelihood and its gradient w.r.t to the characteristic length-scale. Notice there seems to be an optimal point at around $l = 1.4$.



- **Holdout test**. Consider $\mathcal{D} = \{\mathcal{D}_T, \mathcal{D}_V\}$ as a training and validation set from your data and $\boldsymbol{\theta}$ some chosen hyperparameters to test. Train your Gaussian Process regressor on $\mathcal{D}_T$ with hyperparameters $\boldsymbol{\theta}$ and test its performance according to some loss metric $\mathcal{L}$ on $\mathcal{D}_V$. Repeat as many times as needed with different hyperparameter choices. Choose the set of hyperparameters yielding the lowest loss.
- k**-fold cross validation**. Instead of considering a single test set $\mathcal{D}_V$, partition $\mathcal{D} = \mathcal{D}_1, \ldots, \mathcal{D}_k$. Train your model iteratively on $k-1$ partitions of the data and test on the remaining one. Consider an average of losses for each hyperparameter choice.

When $k = n$, the resulting method is called *jackniffe*, and theoretical analyses can be provided in the case of Gaussian Processes. The predictive log-density when leaving out training case i is simply:

$$\log p(y_i|X, \boldsymbol{y}_{-i}, \boldsymbol{\theta}) = -\frac{1}{2}\log\sigma_i^2 - \frac{(y_i - \mu_i)^2}{2\sigma_i^2} - \frac{1}{2}\log 2\pi \tag{2.40}$$

where $\boldsymbol{y}_{-i}$, means all target values excluding i and μ_i, σ_i^2 are computed according to the equations detailed in Section 2.3 considering $\mathcal{D} = \left\{X_{-i}, \boldsymbol{y}_{-i}\right\}$ as training sets. Consequently, the likelihood when this is computed for all training cases is given by:

$$L_{\text{JK}}(X, \boldsymbol{y}, \boldsymbol{\theta}) = \sum_{i=1}^{n} \log p(y_i|X, \boldsymbol{y}_{-i}, \boldsymbol{\theta}) \tag{2.41}$$

This last quantity is sometimes called *pseudo*-likelihood. Notice that to compute this quantity, one would need to fit n GPs and therefore inverting matrices for each training case. This can be avoided by noticing that the computations in subsequent fittings are very similar, by using inversion by partitioning. In particular, the expressions for μ_i and σ_i can be expressed in terms of the full GP:

$$\mu_i = y_i - \frac{[K^{-1}\boldsymbol{y}]_i}{[K^{-1}]_{ii}} \qquad\qquad \sigma_i^2 = \frac{1}{[K^{-1}]_{ii}} \tag{2.42}$$

We can obtain derivatives w.r.t. hyperparameters from Equation 2.40 to perform gradient-based optimization. In particular, let us first define the derivatives of μ_i and σ_i^2:

$$\frac{\partial\mu_i}{\partial\theta_j} = \frac{[Z_j\boldsymbol{\alpha}]_i}{[K^{-1}]_{ii}} - \frac{\boldsymbol{\alpha}_i[Z_jK^{-1}]_{ii}}{[K^{-1}]_{ii}^2} \qquad\qquad \frac{\partial\sigma_i^2}{\partial\theta_j} = \frac{[Z_jK^{-1}]_{ii}}{[K^{-1}]_{ii}^2} \tag{2.43}$$

where $\boldsymbol{\alpha} = K^{-1}\boldsymbol{y}$ and $Z_j = K^{-1}\frac{\partial K}{\partial \theta_j}$. Finally, our gradient can be written as:

$$\frac{\partial L_{\text{JK}}}{\partial \theta_j} = \sum_{i=1}^{n} \frac{\left(\alpha_i [Z_j \boldsymbol{\alpha}]_i - \frac{1}{2}\left(1 + \frac{\alpha_i^2}{[K^{-1}]_{ii}}\right)[Z_j K^{-1}]_{ii}\right)}{[K^{-1}]_{ii}} \tag{2.44}$$

One may ask under which circumstances the jackniffe approach might be preferable to direct marginal likelihood optimization, since computationally they are almost identical. Some have argued [12] that the cross-validated approach should be more robust to model mis-specification.

2.6 Further theoretical aspects

In this section, we will briefly mention other theoretical aspects of Gaussian Processes. This includes for example, how Gaussian Process can be seen as *linear smoothers*, by means of a spectral analysis or how to incorporate explicit basis functions into the model. Finally, we will discuss a full Bayesian treatment of covariance function hyperparameters, by the use of different MCMC sampling strategies.

2.6.1 Gaussian processes as linear smoothers

As many machine learning algorithms, the main objective of a Gaussian Process regressor is to reconstruct the underlying signal f by removing noise ϵ. It does this by computing a weighted average of the values $\boldsymbol{y}$. In particular, as seen in 2.3, it can be written as:

$$\overline{f}(\boldsymbol{x}_*) = \boldsymbol{k}(\boldsymbol{x}_*)^T (K + \sigma_n^2 I)^{-1} \boldsymbol{y} \tag{2.45}$$

Therefore, one can see a Gaussian Process regressor as a linear smoother [3]. We can study this smoothing in terms of spectral analysis. Again, for training points, predicted training points $\overline{\boldsymbol{f}}$ are:

$$\overline{\boldsymbol{f}} = K(K + \sigma_n^2 I)^{-1} \boldsymbol{y} \tag{2.46}$$

Write K using its eigenvalue decomposition $K = \sum_{i=1}^{n} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T$, with λ_i and $\boldsymbol{u}_i$ its i-th eigenvalue and eigenvector respectively. Since K is a covariance matrix, its is symmetric positive semidefinite, and therefore has positive eigenvalues. If we note $\gamma_i = \boldsymbol{u}_i^T \boldsymbol{y}$, then:

$$\overline{\boldsymbol{f}} = \sum_{i=1}^{n} \frac{\gamma_i \lambda_i}{\lambda_i + \sigma_n^2} \boldsymbol{u}_i \tag{2.47}$$

For the covariance functions we have studied in section 2.4, the eigenvalues are larger for slowly varying eigenvectors, so the more frequent items in $\boldsymbol{y}$ get smoothed-out. The effective number of degrees of freedom in a Gaussian Process model can be defined as the number of used eigenvectors:

$$\text{df}(K) = \text{tr}(K(K + \sigma_n^2 I)^{-1}) = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda_i + \sigma_n^2} \tag{2.48}$$

To make the explanation clearer, let us define $\boldsymbol{h}(\boldsymbol{x}_*) = (K + \sigma_n^2 I)^{-1} \boldsymbol{k}(\boldsymbol{x}_*)$. So for a new given point, prediction is defined as $\overline{f}(\boldsymbol{x}_*)^T \boldsymbol{y}$, that is, a linear combination of $\boldsymbol{y}$, with weights $\boldsymbol{h}(\boldsymbol{x}_*)$. A Gaussian Process regressor is a linear smoother, since the weight function $\boldsymbol{h}$ does not depend directly on $\boldsymbol{y}$. While a regular linear model defines a linear combination of the inputs, a linear smoother defines a linear combination of the targets. This weight function depends directly on the specific location of the n training points, by means of the matrix inversion of $K + \sigma_n^2 I$, therefore observations close in input space are smoothed out.

2.6.2 Explicit basis functions

Notice that during the entire chapter, we have considered a Gaussian Process prior with mean $m(\boldsymbol{x}) = 0$ for simplicity reasons. One may want, however, to define a different mean value for the prior. On the other hand, imposing $m(\boldsymbol{x}) = 0$ is not a strong assumption, since the posterior is not constrained to be zero as well. With an explicit mean function $m(\boldsymbol{x}) \neq 0$, the prior becomes:

$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x^*})\right) \tag{2.49}$$

and the mean of the posterior predictive distribution then becomes, very naturally:

$$\boldsymbol{f}_* = \boldsymbol{m}(X_*) + k(X_*, X)K^{-1}\left(\boldsymbol{y} - \boldsymbol{m}(X)\right) \tag{2.50}$$

The variance of the posterior predictive distribution remains the same as in Equation 2.3. In practice, however, it may not be clear how to specify a prior mean function for the process. In some cases it may be useful to define a few parametric basis function, whose parameters $\boldsymbol{\beta}$ we have to estimate from training data. Formally:

$$g(\boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{h}(\boldsymbol{x})^T\boldsymbol{\beta} \tag{2.51}$$

where $f(\boldsymbol{x})$ is a regular zero-mean Gaussian Process prior, $\boldsymbol{h}(\boldsymbol{x})$ are our chosen basis functions, and β are our parameters. For example, if we are interested in polynomial regression, then $\boldsymbol{h}(x) = (1, x, x^2, \dots)$. One could consider optimizing $\boldsymbol{\beta}$ the same way as with our kernel hyperparameters, but if we assume a Gaussian prior $\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{b}, B)$, we can solve analytically to obtain another Gaussian Process:

$$g(\boldsymbol{x}) \sim \mathcal{GP}\left(\boldsymbol{h}(\boldsymbol{x}^T\boldsymbol{b}, k(\boldsymbol{x}, \boldsymbol{x}^*) + \boldsymbol{h}(\boldsymbol{x})^T B\boldsymbol{h}(\boldsymbol{x}^*)\right) \tag{2.52}$$

Notice that now we have an extra term in the covariance function. This is caused by the uncertainty in the parameters of the mean. Now predictions are made by substituting these parameters into Equation 2.50. An explicit version for the mean and covariance is given by:

$$\overline{\boldsymbol{g}}(X_*) = H_*^T\hat{\boldsymbol{\beta}} + K_*^TK^{-1}(\boldsymbol{y} - H^T\hat{\boldsymbol{\beta}}) = \overline{\boldsymbol{f}}(X_*) + R^T\hat{\boldsymbol{\beta}} \tag{2.53}$$

$$\mathrm{Cov}(\boldsymbol{g_*}) = K_{**} + R^T(B^{-1} + HK^{-1}H^T)^{-1}R \tag{2.54}$$

where H and H_* are matrices containing the evaluation of the chosen basis functions over training and testing points respectively, $\hat{\boldsymbol{\beta}} = (B^{-1} + HK^{-1}H^T)^{-1}(HK^{-1}\boldsymbol{y} + B^{-1}\boldsymbol{b})$ and $R = H_* - HK^{-1}K_*$. The posterior process parameters can be interpreted as such: $\hat{\boldsymbol{\beta}}$ is a mean of the model linear parameters, a compromise between the prior and the likelihood provided by the data. The mean of the process is simply $\hat{\boldsymbol{\beta}}$ plus our typical Gaussian Process prediction of the residuals. The covariance matrix is just the addition of our regular expression and a non-negative term.

Consider the limit of B^{-1} as it approaches O (O being a zero-filled matrix), that is when the prior is vague. We then get a predictive distribution independent of $\boldsymbol{b}$:

$$\overline{\boldsymbol{g}}(X_*) = \overline{\boldsymbol{f}}(X_*) + R^T\hat{\boldsymbol{\beta}} \tag{2.55}$$

$$\mathrm{Cov}(\boldsymbol{g_*}) = K_{**} + R^T\left(HK^{-1}H^T\right)^{-1}R \tag{2.56}$$

where now $\hat{\boldsymbol{\beta}} = \left(HK^{-1}H^T\right)^{-1}HK^{-1}\boldsymbol{y}$.

We explore now the behaviour of the marginal log-likelihood under this model where we assume a Gaussian prior $\boldsymbol{\beta} \sim \mathcal{N}(\boldsymbol{b}, B)$. Formally:

$$\log p(\boldsymbol{y}|X, \boldsymbol{b}, B) = -\frac{1}{2}\log|K + H^TBH| - \frac{n}{2}\log 2\pi \tag{2.57}$$

In the same way as before, exploring the limit $B^{-1} \to O$, the prior becomes irrelevant, so we can safely assume that $\boldsymbol{b} = 0$, yielding:

$$\log p(\boldsymbol{y}|X, \boldsymbol{b}=\boldsymbol{0}, B) = -\frac{1}{2}\boldsymbol{y}^T K^{-1}\boldsymbol{y} + \frac{1}{2}\boldsymbol{y}^T C\boldsymbol{y} \tag{2.58}$$

$$-\frac{1}{2}\left(\log|K| + \log|B| + \log|A| + n\log 2\pi\right) \tag{2.59}$$

where $A = B^{-1} + HK^{-1}H^T$ and $C = K^{-1}H^T A^{-1} HK^{-1}$.

2.6.3 Marginalizing over hyperparameters

In the full Bayesian framework, the covariance matrix Σ_θ can be defined without explicitly specifying hyperparameters. Integrating out these considers different possible explanations of the data when making predictions. This is typically done using MCMC techniques. Here we present several techniques for capturing this uncertainty, based on the Metropolis-Hastings criterion and slice sampling [6]. We first assume a prior distribution on hyperparameters:

$$\theta \sim p_h(\theta) \tag{2.60}$$

And remembering now notation from Section 2.3:

$$\boldsymbol{f} \sim \mathcal{N}(0, \Sigma_\theta) \tag{2.61}$$

We forget the conditioning on X here for simplicity. Fix the data $\boldsymbol{y}$ and consider it a function of f. Define the *conditional likelihood* to be the first part of the integrand of the marginal likelihood described in Equation 2.25, that is:

$$\mathcal{L}(\boldsymbol{f}) = p(\text{data}|\boldsymbol{f}) = p(\boldsymbol{y}|\boldsymbol{f}) \tag{2.62}$$

The objective here is to sample from the joint posterior under unknowns:

$$p(\boldsymbol{f}, \theta|\,\text{data}) \propto \mathcal{L}(\boldsymbol{f})p(\boldsymbol{f})p_h(\theta) \tag{2.63}$$

Notice that we would like an unifying approach for different likelihoods $\mathcal{L}$ and covariance priors p_h. A simple algorithm that updates the hyperparameters for fixed latent variables $\boldsymbol{f}$ that leaves invariant the conditional posterior:

$$p(\theta|\boldsymbol{f}) \propto p(\boldsymbol{f})p_h(\theta) \tag{2.64}$$

is the standard Metropolis-Hastings algorithm presented in Algorithm 2. However, the resulting Markov chain from this algorithm can be very slow exploring the joint distribution. It has also been shown that the samples generated by standard MH, are highly informative, which limits the amount of space a Markov chain can cover.

Algorithm 2 Standard Metropolis-Hastings updated for fixed $\boldsymbol{f}$.

Require: Current $\boldsymbol{f}$ and θ, proposal distribution q, covariance function Σ_θ
1: Draw $\theta' \sim q(\theta', \theta)$
2: Draw $u \sim \text{Uniform}(0, 1)$
3: **if** $u < \dfrac{p(\boldsymbol{f}|\theta')p_h(\theta')q(\theta, \theta')}{p(\boldsymbol{f}|\theta)p_h(\theta)q(\theta', \theta)}$ **then return** θ'
4: **else return** θ
5: **end if**

In the extreme limit in which there is no data $\mathcal{L}$ is constant, that is $p(\boldsymbol{f}, \theta) = p(\boldsymbol{f})p_h(\theta)$, for which both both distributions are strongly coupled, and therefore alternating sampling for $\boldsymbol{f}$ and θ does not work. An alternative is to reparametrize the model so that the unknown variables are independent under the prior. An independent random multivariate normal vector $\boldsymbol{\nu}$ is sampled and compute:

$$\begin{aligned} \boldsymbol{\nu} &\sim \mathcal{N}(0, \mathbb{I}) \\ \boldsymbol{f} &= L\boldsymbol{\nu} \end{aligned} \tag{2.65}$$

where $LL^T = \Sigma_\theta$, L being a lower-diagonal matrix taken from a Cholesky decomposition of Σ_θ. Then we update hyperparameters based on a fixed $\boldsymbol{\nu}$ rather than $\boldsymbol{f}$. Since the latent variable $\boldsymbol{f}$ is determined by θ, updates will change both θ and $\boldsymbol{f}$. This is described in Algorithm 3.

Algorithm 3 Standard Metropolis-Hastings updated for fixed $\boldsymbol{\nu}$.

Require: Current $\boldsymbol{f}$ and θ, proposal distribution q, covariance function Σ_θ
1: Compute $\boldsymbol{\nu} = L^{-1}\boldsymbol{f}$
2: Draw $\theta^{'} \sim q(\theta^{'}, \theta)$
3: Compute $\boldsymbol{f}^{'} = L\boldsymbol{\nu}$
4: Draw $u \sim \text{Uniform}(0, 1)$
5: **if** $u < \dfrac{\mathcal{L}(\boldsymbol{f}^{'})p_h(\theta^{'})q(\theta, \theta^{'})}{\mathcal{L}(\boldsymbol{f})p_h(\theta)q(\theta^{'}, \theta)}$ **then return** $\theta^{'}, \boldsymbol{f'}$
6: **else return** $\theta, \boldsymbol{f}$
7: **end if**

In practice, none of the two mentioned solutions are ideal for applications. Algorithm 3 is ideal in the weak data limit, where $\boldsymbol{f}$ is almost identically distributed as the prior. In the strong data limit, a simple Metropolis-Hastings like proposed in Algorithm 2 is ideal, since the $\boldsymbol{f}$ carries most weight from the likelihood. Notice, that the latter, however, only updates θ but does not propose any updates over $\boldsymbol{f}$. A recent alternative considers using an augmented data model, introducing surrogate Gaussian observations that will guide proposals of both hyperparameters and latent variables. The idea is to augment the Gaussian latent model with noisy auxiliary variable $\boldsymbol{g}$:

$$\boldsymbol{g}|\boldsymbol{f}, \theta \sim \mathcal{N}(\boldsymbol{f}, S_\theta) \tag{2.66}$$

where S_θ is an arbitrary parameter that can be either set manually or depending of current θ. Integrating out $\boldsymbol{f}$ yields:

$$\boldsymbol{g}|\theta \sim \mathcal{N}(0, \Sigma_\theta + S_\theta) \tag{2.67}$$

The original latent model on $\boldsymbol{f}$ implies a joint auxiliary distribution $p(\boldsymbol{f}, \boldsymbol{g}|\theta)$, conditioning on $\boldsymbol{g}$ we obtain:

$$\boldsymbol{f}|\boldsymbol{g}, \theta \sim \mathcal{N}(\boldsymbol{m}_{\theta,\boldsymbol{g}}, R_\theta) \tag{2.68}$$

where:

$$\begin{aligned} R_\theta &= S_\theta - S_\theta(S_\theta + \Sigma_\theta)^{-1}S_\theta \\ \boldsymbol{m}_{\theta,\boldsymbol{g}} &= R_\theta S_\theta^{-1}\boldsymbol{g} \end{aligned} \tag{2.69}$$

Under this surrogate modelling, the latent variables are drawn from their posterior given $\boldsymbol{g}$. The sampling procedure is very similar then to our previous MH sampler. It is detailed in Algorithm 4.

Our discussed Metropolis-Hastings algorithms, while efficient, require selecting a proposal distribution q that also has to be tuned. Instead, some authors [7, 6] have proposed slice sampling as an alternative. Slice sampling [8] is an adaptive procedure that are more robust to the choice of scale in our proposal distribution. A procedure from this family is detailed in Algorithm 5.

Algorithm 4 Surrogate model Metropolis-Hastings.

Require: Current $\boldsymbol{f}$ and θ, proposal distribution q, covariance function Σ_θ

Draw $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{f}, S_\theta)$
Compute $\boldsymbol{\nu} = L_{R_\theta}^{-1}(\boldsymbol{f} - \boldsymbol{m}_{\theta,\boldsymbol{g}})$
Draw $\theta^{'} \sim q(\theta^{'}, \theta)$
Compute $\boldsymbol{f}^{'} = L_{R_{\theta'}}\boldsymbol{\nu} + \boldsymbol{m}_{\theta',\boldsymbol{g}}$
Draw $u \sim \text{Uniform}(0, 1)$
if $u < \dfrac{\mathcal{L}(\boldsymbol{f})p_{\boldsymbol{g}|\theta'}(\boldsymbol{g})p_h(\theta^{'})q(\theta, \theta^{'})}{\mathcal{L}(\boldsymbol{f})p_{\boldsymbol{g}|\theta}(\boldsymbol{g})p_h(\theta)q(\theta', \theta)}$ **then return** $\theta^{'}, \boldsymbol{f}^{'}$
else return $\theta, \boldsymbol{f}$
end if

Algorithm 5 Surrogate model slice sampling.

Require: Current $\boldsymbol{f}$ and θ, scale σ, covariance function Σ_θ

Draw surrogate $\boldsymbol{g} \sim \mathcal{N}(\boldsymbol{f}, S_\theta)$
Compute $\boldsymbol{\nu} = L_{R_\theta}^{-1}(\boldsymbol{f} - \boldsymbol{m}_{\theta,\boldsymbol{g}})$
Center bracket $v \sim \text{Uniform}(0, 1)$
$\theta_{\min} = \theta - v$
$\theta_{\max} = \theta_{\min} + \sigma$
Draw $u \sim \text{Uniform}(0, 1)$
Compute threshold $y = \mathcal{L}(\boldsymbol{f})p_{\boldsymbol{g}|\theta}(\boldsymbol{g})p_h(\theta)$
Draw $\theta^{'} \sim \text{Uniform}(\theta_{\min}, \theta_{\max})$
Compute $\boldsymbol{f}^{'} = L_{R_\theta}\boldsymbol{\nu} + \boldsymbol{m}_{\theta',\boldsymbol{g}}$
if $\mathcal{L}(\boldsymbol{f}^{'})p_{\boldsymbol{g}|\theta'}(\boldsymbol{g})p_h(\theta) > y$ **then return** $\boldsymbol{f'}, \theta^{'}$
else if $\theta^{'} < \theta$ **then**
    Shrink bracket min. $\theta_{\min} = \theta^{'}$
else
    Shrink bracket max. $\theta_{\max} = \theta^{'}$
end if
goto 8.

Bibliography

[1] George Arfken. *Mathematical Methods for Physicists 6th*. Vol. 40. 4. 2005, p. 642. ISBN: 0120598760. DOI: 10.1119/1.1988084. arXiv: arXiv:1011.1669v3.

[2] William M Bolstad. "Bayesian Inference for Simple Linear Regression". In: *Introduction to Bayesian Statistics* (2007), pp. 267–295. DOI: 10.1002/9780470181188.ch14. URL: http://dx.doi.org/10.1002/9780470181188.ch14.

[3] A. Buja, T. Hastie, and R. Tibshirani. "Linear smoothers and additive models". In: *The Annals of Statistic* 17.2 (1989), pp. 453–555. ISSN: 1098-6596. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[4] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. *Kernel methods in machine learning*. 2008. DOI: 10.1214/009053607000000677. arXiv: 0701907v3 [arXiv:math].

[5] Budiman Minasny and Alex B. McBratney. "The matern function as a general model for soil variograms". In: *Geoderma* 128 (2005), pp. 192–207. ISSN: 00167061. DOI: 10.1016/j.geoderma.2005.04.003.

[6] Iain Murray and Ryan Prescott Adams. "Slice sampling covariance hyperparameters of latent Gaussian models". In: *Advances in Neural Information Processing ...* 2.1 (2010), p. 9. arXiv: 1006.0868. URL: http://papers.nips.cc/paper/4114-slice-sampling-covariance-hyperparameters-of-latent-gaussian-models%7B%5C%%7D5Cnhttp://arxiv.org/abs/1006.0868.

[7] Iain Murray, Ryan Prescott RP Adams, and DJC David J. C. MacKay. "Elliptical slice sampling". In: *arXiv preprint arXiv:1001.0175* 2 (2009), p. 8. ISSN: 15324435. arXiv: 1001.0175. URL: http://www.jmlr.org/proceedings/papers/v9/murray10a/murray10a.pdf$%5Cbackslash$nhttp://arxiv.org/abs/1001.0175.

[8] Radford M. Neal. *Slice sampling: Rejoinder*. 2003. DOI: 10.1214/aos/1056562461. arXiv: 1003.3201v1.

[9] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Vol. 14. 2. 2004, pp. 69–106. ISBN: 026218253X. DOI: 10.1142/S0129065704001899. arXiv: 026218253X. URL: http://www.gaussianprocess.org/gpml/chapters/RW.pdf.

[10] Peter E. Rossi, Greg M. Allenby, and Robert McCulloch. *Bayesian Statistics and Marketing*. 2006, pp. 1–348. ISBN: 9780470863695. DOI: 10.1002/0470863692.

[11] H. Wackernagel. "Multivariate geostatistics: an introduction with applications". In: *Multivariate geostatistics: an introduction with applications* (1995). ISSN: 3540601279 (ISBN). DOI: 10.1016/S0098-3004(97)87526-7. arXiv: arXiv:1011.1669v3.

[12] Grace Wahba. *Spline Models for Observational Data*. Vol. 33. 3. 1991, pp. 502–502. ISBN: 0-89871-244-0. DOI: 10.1137/1033124.

Chapter 3

Bayesian optimization

3.1 Preliminaries

In this chapter we will deal with the main topic of this master's thesis, Bayesian Optimization. Here, we approach global optimization from the viewpoint of Bayesian theory, as a sequential problem. For the moment, imagine that we have a very expensive function to evaluate $f : \mathbb{R}^n :\to \mathbb{R}$. This function, for the purposes of this work, will be the negative of a loss function in a machine learning problem, or any other fitness function that we wish to maximize. Formally, we wish to maximize over a compact set $\mathcal{A}$.

$$\max_{\boldsymbol{x}\in\mathcal{A}} f(\boldsymbol{x}) \tag{3.1}$$

For technical reasons, we also assume that the function is *Lipschitz-continuous*, that is, there exists some constant C such that $\forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{A}$:

$$\|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\| \leq C\|\boldsymbol{x}_1 - \boldsymbol{x}_2\| \tag{3.2}$$

We are also interested in global optimization instead of local, since loss functions do not have to be convex over hyperparameter space. That is, we can not assume that we can find a point $\boldsymbol{x^*}$ such that:

$$f(\boldsymbol{x}^*) \geq f(\boldsymbol{x}), \forall \boldsymbol{x} \text{ s.t.} \|\boldsymbol{x}^* - \boldsymbol{x}\| < \epsilon \tag{3.3}$$

The function we are typically interested may not have an analytical expression that we can analyse, take derivatives etc. Most we will assume here is that we can just query the function over any point to evaluate $\boldsymbol{x} \in \mathcal{A}$ and some bounds to optimize over. This is normally called a *black box* function. Moreover, the function response can be noisy. This is the case when optimizing a loss or fitness function in machine learning on a holdout test, for example, only having an estimation of its real value.

Bayesian optimization has risen over the last few years as a very attractive method to optimize expensive to evaluate black box functions [30, 6]. It has grabbed the attention of machine learning researchers over simpler model hyperparameter optimization strategies, such as grid search or random search [4]. Bayesian optimization uses prior information and evidence to define a posterior distribution over the space of functions. The model we will use to model this posterior is Gaussian Process regression, for which we have studied its basics in the previous chapter.

3.2 The bayesian optimization framework

Assume that we have sampled our function f to optimize a small number of times k. Notice this can be treated as a regression problem where $\boldsymbol{x}_k$ is the k-th point we have sampled and y_k its (possibly noisy) function evaluation. We can fit a Gaussian Process regression model over the set of sampled points and evaluations. Remember from Section 2.3 that this gives us a posterior distribution over all possible values in $\mathcal{A}$. Basically, we will use this information to optimize the function efficiently. Note by

$$\mathcal{D}_n = \{\boldsymbol{x}_i, y_i, i = 1, \dots, n\}\,. \tag{3.4}$$

the set of training values.

Bayesian optimization is a sequential model-based approach for optimization. The posterior distribution facilitated by the Gaussian Process allows us to define what we will call an *acquisition function* α that will guide the search for the most promising point from $\mathcal{A}$ to evaluate each step. Once we have sampled said point, we re-fit our Gaussian Process to update our posterior with the new information gathered and proceed the same way until convergence. The mentioned acquisition functions are both heuristic and myopic, in the sense that they define some behaviour given the posterior and only take the information available at a single step of the optimization. Typically, these functions trade-off exploration and exploitation of the target function, and their optima is close to where the posterior variance of the Gaussian Process is large (exploration) or where its posterior mean is high (exploitation). We will choose the next sampled point to evaluate by maximizing these acquisition functions. Algorithm 6 provides pseudo-code to implement a basic bayesian optimization module.

Algorithm 6 Bayesian optimization framework.

1: Sample a small number of points $\boldsymbol{x} \in \mathcal{A}$. Evaluate $f(\boldsymbol{x})$ to get $\mathcal{D}_n$
2: **for** $n = 1, 2, \dots$ **do**
3: Fit a GP regression model on $\mathcal{D}_n$
4: $\boldsymbol{x}_{n+1} \leftarrow \arg\max_{\boldsymbol{x}} \alpha(\boldsymbol{x}, \mathcal{D}_n)$
5: Evaluate $f(\boldsymbol{x}_{n+1}) = y_{n+1}$
6: Augment data $\mathcal{D}_{n+1} = \{\mathcal{D}_n, (\boldsymbol{x}_{n+1}, y_{n+1})\}$
7: **end for**

3.3 On acquisition functions

Thus far we have described the statistical model behind the optimization framework. The next natural step to ask is how we can define acquisition functions depending on its behaviour or the function we wish to maximize. In pyGPGO, the most common acquisition functions are implemented under the `Acquisition` class in the `acquisition` module. We can classify most of them in three main groups: improvement-based, optimistic, and information-based policies. We will start by analysing each of them:

3.3.1 Improvement-based policies

These acquisition functions' behaviour is to favour points that are in some way likely to improve upon the best observed value so far τ. Since any finite sample of a Gaussian Process is a multivariate Gaussian distribution, the most straightforward idea is to use an estimation of the *probability of improvement* of point evaluation ν w.r.t. τ.

$$\alpha_{\text{PI}}(\boldsymbol{x}, \mathcal{D}_n) = P\left(\nu > \tau\right) = \Phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) \tag{3.5}$$

where Φ denotes the standard normal cumulative density function and $\mu_n(\boldsymbol{x})$ and $\sigma_n(\boldsymbol{x})$ are the posterior mean and standard deviation of the fitted Gaussian Process at step n. In a sense, what this acquisition function is doing is just accumulating the posterior probability mass above τ at $\boldsymbol{x}$. The associated utility function is just an indicator of improvement $I(\boldsymbol{x}, \nu, \theta) = \mathbb{I}(\nu > \tau)$. While this is a very natural acquisition function to use, it has been shown [17] that it behaves greedily if the best τ is not known.

Another very popular acquisition function is called *expected improvement*. This incorporates the amount of improvement over τ by weighing the probability of improvement over the difference $\nu - \tau$. Formally:

$$I(\boldsymbol{x}, \nu, \theta) = (\nu - \tau)\mathbb{I}(\nu > \tau) \tag{3.6}$$

Taking the expectation yields the expected improvement acquisition function:

$$\alpha_{\mathrm{EI}}(\boldsymbol{x}, \mathcal{D}_n) = \mathbb{E}\left[I(\boldsymbol{x}, \nu, \theta)\right] = (\mu_n(\boldsymbol{x}) - \tau)\Phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) + \sigma_n(\boldsymbol{x})\phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) \tag{3.7}$$

where ϕ is in this case the standard normal density function. This acquisition function is by far the most used, since it has been empirically studied [17] and proven convergence rates for [8]. We have assumed that τ is the best observed value so far during the optimization procedure, but theoretical convergence is only guaranteed when τ is the best value f can take in $\mathcal{A}$. During practical research, however, this does not seem to be a concern [32].

3.3.2 Optimistic policies

Optimistic acquisition functions have their origins in the multi-armed bandit setting [20]. These policies behave optimistically in the face of uncertainty, as a way to tradeoff exploration and exploitation. The most popular of methods in this class is the *Gaussian process upper confidence bound* (GP-UCB) [34], with provable regret bounds. It works by taking a quantile of the posterior process, and since it is Gaussian, we can derive the result analytically:

$$\alpha_{\mathrm{UCB}}(\boldsymbol{x}, \mathcal{D}_n) = \mu_n(\boldsymbol{x}) + \beta_n\sigma_n(\boldsymbol{x}) \tag{3.8}$$

where β_n controls the quantile we may be interested in. Theoretically motivated by the multi-armed bandits, there are guidelines to select and schedule β_n dynamically. Notice that if we choose $\beta = \beta_n$, to be one value or another, we will be encouraging the algorithm to exploit frequently by choosing points with high posterior mean (small β) or to explore frantically by choosing points with high posterior variance (high β).

3.3.3 Information-based policies

These are a newer class of methods that consider the posterior distribution over an unknown minimizer $\boldsymbol{x}^*$. One of the most popular policies in this categories is again motivated by the multi-armed bandit problem, Thompson sampling [18]. This very old strategy consists in randomly sampling rewards from the posterior distribution and picking the highest one. It is a randomized acquisition function in the sense:

$$\boldsymbol{x}_{n+1} \sim p_*(\boldsymbol{x}|\mathcal{D}_n) \tag{3.9}$$

This method, however, is not as simple to implement as the previously discussed one. It is not entirely clear how to sample in the continuous space of the Gaussian Process. There have been studies that solve this issue by using techniques like spectral sampling [26]. We could define formally this acquisition function:

$$\alpha_{\mathrm{TS}}(\boldsymbol{x}, \mathcal{D}_n) = f^{(n)}(\boldsymbol{x}) \tag{3.10}$$

where $f^{(n)} \sim GP\left(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})\right)$ by spectral sampling. It has been shown, however, that this method tends to perform greedily on high-dimensional spaces [12]. Another new approach is entropy-based [37]. They aim to reduce the uncertainty in location $\boldsymbol{x}^*$ by choosing points likely to reduce the entropy in $p(\boldsymbol{x}|\mathcal{D}_n)$. The acquisition function can be defined as:

$$\alpha_{\mathrm{ES}}(\boldsymbol{x}|\mathcal{D}_n) = H(\boldsymbol{x}^*|\mathcal{D}_n) - \mathbb{E}_{y|\mathcal{D}_n,\boldsymbol{x}}\left[H(\boldsymbol{x}^*|\mathcal{D}_n \cup \{(\boldsymbol{x}, y)\})\right] \tag{3.11}$$

where H notes the differential entropy function of the posterior distribution. As with Thompson sampling, the function is not tractable in continuous spaces. Several studies have been done approximating this quantity, either by using simple Monte Carlo sampling [37] or a space discretization of $\mathcal{A}$ [12]. A recent paper [13] introduced *predictive entropy search* (PES), a method to remove the need for discretization by rewriting Equation 3.11 as:

$$\alpha_{\text{PES}}(\boldsymbol{x}, \mathcal{D}_n) = H(y|\mathcal{D}_n, \boldsymbol{x}) - \mathbb{E}_{\boldsymbol{x}^*|\mathcal{D}_n}\left[H(y|\mathcal{D}_n, \boldsymbol{x}, \boldsymbol{x}^*)\right] \tag{3.12}$$

The expectation is approximated in the original paper by Monte Carlo with Thompson samples, with simplifying assumptions. This is arguably the current state of the art in acquisition functions, according to the results reported in [13].

3.3.4 Acquisition function portfolios

In a *no free lunch* fashion, it can be shown that no acquisition function will outperform the others in every single problem. In fact, it has been proven [14] that the acquisition function to provide optimal performance can change even in different points of the optimization procedure. It is natural, therefore, to consider an ensemble of acquisition functions and act upon it. In general, this implies optimizing all of these functions at each optimization step and then choosing among candidate points using a meta-criteria. This higher order criteria can be seen as a second level acquisition function.

Earlier approaches rely on modifications of the Hedge algorithm [1], again inspired by the multi-armed bandit problem. It is basically based on measuring past performance of points proposed by the different acquisition functions to predict future performance (or gain), via another objective function. However, this strategy tends to undervalue exploration, which also provides valuable information on the target. Another more recent approach [29] is called *Entropy Search Portfolio*, that considers candidates by weighing the gain of information towards the optimum. Formally it is defined as:

$$\alpha_{\text{ESP}}(\boldsymbol{x}, \mathcal{D}_n) = -\mathbb{E}_{y|\mathcal{D}_n, \boldsymbol{x}}\left[H\left[\boldsymbol{x}_*|\mathcal{D}_n \cup \{(\boldsymbol{x}, y)\}\right]\right] \tag{3.13}$$

and then we try to maximize over the candidates provided by the k based acquisition functions $\boldsymbol{x}_{1:K,n}$.

$$\boldsymbol{x}_n = \underset{\boldsymbol{x}_{1:K,n}}{\arg\max}\, \alpha_{\text{ESP}}(\boldsymbol{x}|\mathcal{D}_n) \tag{3.14}$$

In other words, this method chooses the candidate that is expected to reduce the most the entropy about the minimizer $\boldsymbol{x}_*$.

3.3.5 Visualizing the behaviour of an acquisition function

To demonstrate the behaviour of different acquisition functions on a step of Bayesian optimization, we will create a small script with our sine function example. This will help us understand visually the trade-off between exploration and exploitation in each case. The code provided in Appendix **??** produces Figure 3.1.

3.3.6 Why does Bayesian Optimization work?

In this small section, we will consider very briefly why the Bayesian Optimization procedure is efficiently making use of the information provided by the mean and variance posterior distribution of the GP. The explanation is mostly visual by means of Figure 3.2. The plot shows a Gaussian Process regression model, as well as its confidence interval, comprised from a lower confidence bound $L = \mu(\boldsymbol{x}) - q\sigma(\boldsymbol{x})$ and an upper confidence bound $U = \mu(\boldsymbol{x}) + q\sigma(\boldsymbol{x})$ for a quantile $q > 0$.

In practice, when we are interested in doing hyperparameter search in machine-learning, the algorithms are very blunt, in the sense that the most common strategies explore the entire space, either exhaustively or randomly. The acquisition functions defined in this chapter before use information from the Gaussian Process prior to explore efficiently the space. The specifics of each acquisition function have already been discussed, specially their exploration/exploitation balance, but they all share some common logic.

Figure 3.1: Acquisition function behaviour for Expected Improvement, Probability of Improvement, GP-UCB ($\beta = .5$) and GP-UCB($\beta = 1.5$) in the sine function example.

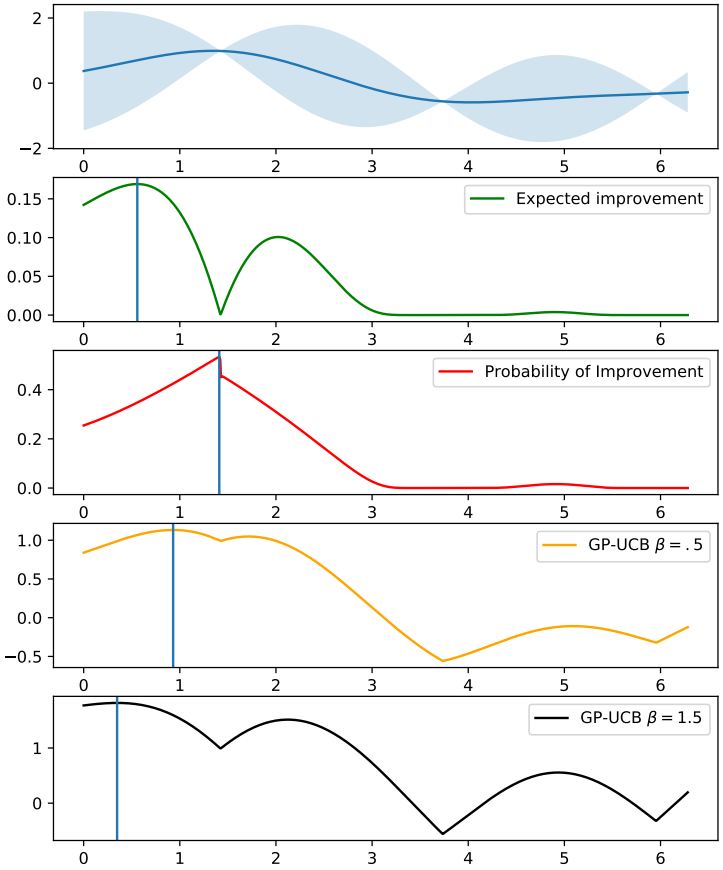
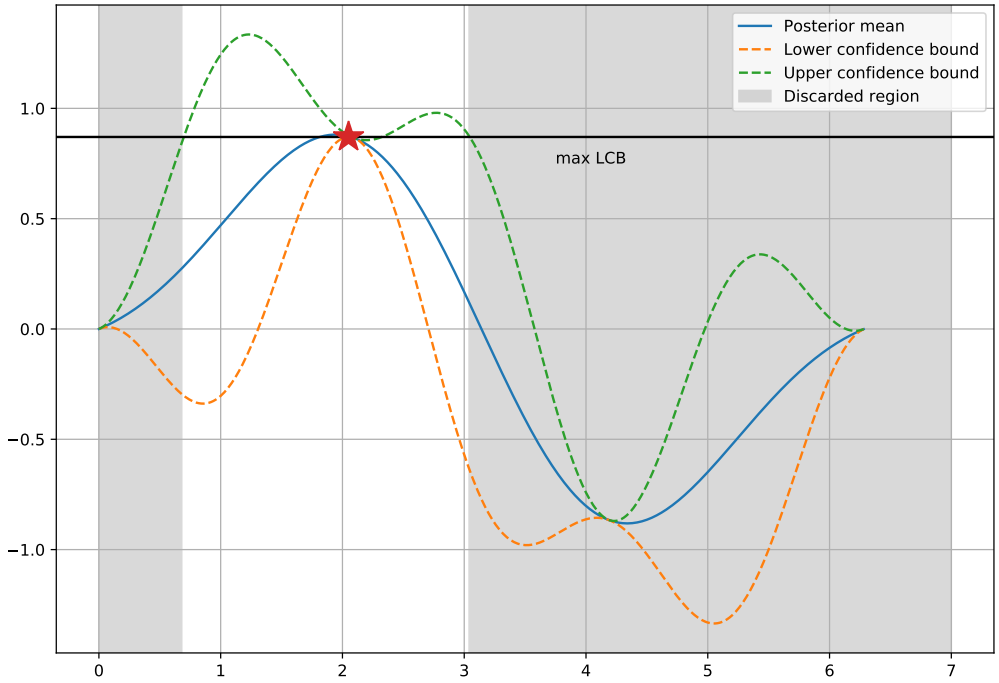


Figure 3.2: A visual explanation on why Bayesian optimization is efficient at exploring the space. It ignores all the input space where the UCB is lower than the point with maximum LCB.



Bayesian Optimization is efficient because it ignores all the space where the predicted upper confidence bound is lower than the maximum value of the lower confidence bound. It only uses the space that fulfills this criteria, according to the strategy selected by the chosen acquisition function. This remains a very reasonable assumption throughout the whole optimization procedure. In practice, the framework resembles a branch-and-bound type algorithm, but probabilistically. The procedure assumes that the true function will lie, with high probability $1 - \delta$ within both lower and upper confidence bounds respectively. However, there is always the probability δ of the function not fulfilling this criteria, in which case the fit surrogate Gaussian Process will be updated with further evaluations. The code for generating this particular representation can be consulted in Appendix A.7.

While plenty of theoretical properties are known for bandit algorithms in general, only some have been established for Bayesian Optimization recently. In particular, for the Gaussian Process surrogate some consistency proofs exist [23] in the one dimensional case and in the multidimensional by the use of partitioning [16, 36]. Finite sample bounds have been provided recently for the GP-UCB acquisition function [33], while this was only proven for the fixed hyperparameter setting. However, despite the recent interest in theoretical properties of this framework, the gap between these and practice is still large [32].

3.4 Role of GP hyperparameters in optimization

We already considered the role of hyperparameters in Gaussian Process regression in section 2.5. However, one may wonder how the estimation of this parameters, one way or another may affect the optimization procedure. So far, we have assumed that during the optimization procedure, parameters θ were given.

Here we will consider two ways of handling hyperparameters during the optimization procedure, the one we presented, type II maximum likelihood estimation and approximate marginalization. For the moment, consider a generic acquisition function $\alpha : \mathcal{X} \times \Theta \to \mathbb{R}$, where $\theta \in \Theta$ are our Gaussian Process hyperparameters. Naturally, one wishes to marginalize the uncertainty caused by θ with the following expression:

$$\alpha(\boldsymbol{x}) = \mathbb{E}_{\theta|\mathcal{D}_n}\left[\alpha(\boldsymbol{x}, \theta)\right] = \int_{\Theta} \alpha(\boldsymbol{x}|\theta) p(\theta|\mathcal{D}_n) d\theta. \tag{3.15}$$

The simplest way to do this is what we saw in 2.36, to optimize the marginal log-likelihood to obtain MAP estimates $\hat{\theta}_{\text{MAP}}$. Then, in each step of the optimization procedure, we simply maximize:

$$\hat{\alpha}(\boldsymbol{x}) = \alpha(\boldsymbol{x}, \hat{\theta}) \tag{3.16}$$

That is, we optimize the acquisition function defined by the *optimal* hyperparameters for the Gaussian Process determined in each step. Again, optimizing the marginal log-likelihood is a problem of its own, but it is common to use quasi-Newton methods such as L-BFGS-B methods.

A more Bayesian approach is to incorporate the uncertainty of θ into our model, since it may have an important role in guiding exploration. Point estimates are in a sense *winners* that may not capture the complexity of the response surface. The second approach we will be considering here, will be therefore to marginalize out hyperparameters using Markov Chain Monte Carlo (MCMC) sampling techniques. In practice, we will need to average M samples $\left\{\theta_n^{(i)}\right\}_{i=1}^{M}$ from the posterior distribution $p(\theta|\mathcal{D}_n)$:

$$\mathbb{E}_{\theta|\mathcal{D}_n}\left[\alpha(\boldsymbol{x}, \theta)\right] \approx \frac{1}{M} \sum_{i=1}^{M} \alpha(\boldsymbol{x}, \theta_n^{(i)}) \tag{3.17}$$

Since it is not possible to have an analytical expression for the posterior distribution $p(\theta|\mathcal{D}_n)$, it is common to use MCMC techniques like Hamiltonian Monte Carlo [24] to produce a sequence of samples whose stationary distribution is the posterior we are looking for. Once M valid samples are obtained, they are evaluated in the acquisition function and averaged. pyGPGO implements these *integrated* acquisition functions using the pyMC3 software [9]. A plot of sampled Gaussian Process associated with its posterior sampled $p(\theta|\mathcal{D}_n)$ can be seen in Figure 3.3. Its associated script can be checked in Appendix A.6.

Quadrature methods can be used instead of MCMC techniques, yielding a weighted mixture:

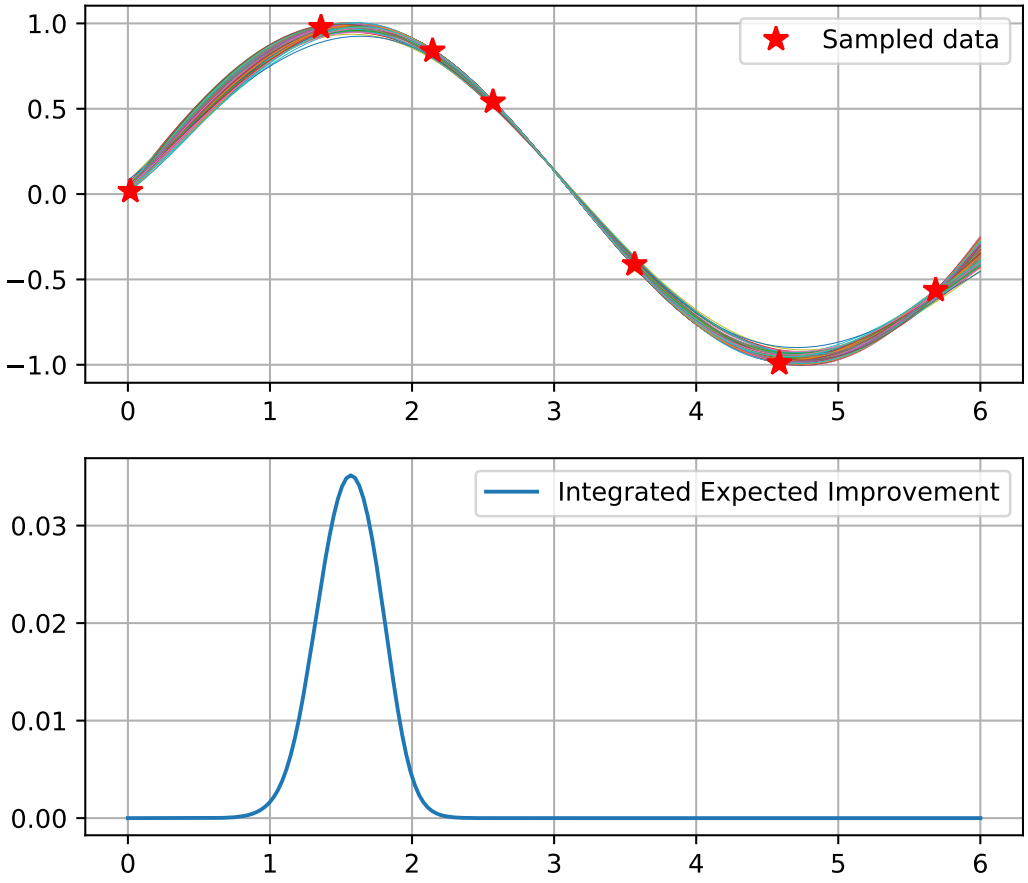
$$\mathbb{E}_{\theta|\mathcal{D}_n}\left[\alpha(\boldsymbol{x}, \theta)\right] \approx \sum_{i=1}^{M} \omega_i \alpha(\boldsymbol{x}, \theta_n^{(i)}) \tag{3.18}$$

However, and to finish this section, in the problems that we will tackle in the experiments, it is usually a bad idea to estimate kernel hyperparameters. Estimating these hyperparameters with few function evaluations is a very challenging task, and can lead to disastrous results, as proven in [3, 8]. The marginal log-likelihood surface can easily fall into traps or be very flat, as seen by the (not cherry-picked) example in Figure 2.4. Even the more advanced MCMC or quadrature methods still suffer from this problem [38].

3.5 Optimizing the acquisition function

We have presented many acquisition functions in this chapter and provided a simple example to demonstrate basic functionality. However, so far, we have assumed that the acquisition function can be easily optimized. This is however, a problem of its own. The reader may be thinking that we have, in fact, changed one optimization problem (the one where we are interested in optimizing f) for another! (in which we now have to optimize α). This is technically true, but bear in mind that while f is very expensive to evaluate, α is very cheap, and it is reasonable to spend a bit more computational effort in evaluating α if it implies having to evaluate f less.

Figure 3.3: Means of 200 posterior predictive distributions, taken from associated GPs to each posterior sample $p(\theta|\mathcal{D}_n)$ in the MCMC procedure. The integrated acquisition functions better take into account the uncertainty of hyperparameters, which leads to less peaky functions.



Maximizing α, however, is not an easy task. The acquisition function is often multi-modal and therefore non-convex, as it can be see again in Figure 3.1. Theoretical convergence, furthermore, is only guaranteed when the optimal point $\boldsymbol{x}^*$ in the acquisition function is found [36]. At the end of the day, we encounter yet another global optimization problem that needs to be solved. From a practical point of view, there are many approaches the community has taken to solve this problem, from discretization [32] to adaptive-grids [2]. If gradient information is available (rarely the case), a multi-start gradient ascent approach can be taken [22]. Evolutionary approaches like CMA-ES can also be used [11]. pyGPGO, in the `GPGO` module uses by default a multi-start quasi-Newton method (L-BFGS-B) to optimize the acquisition function, which in practice seems to work reasonably well.

Other methods have been proposed as alternatives to Bayesian Optimization in this aspect. [19, 7] sequentially build space-partitioning trees by splitting leaves with high function values or upper confidence bounds. This is called *Simultaneous Optimistic Optimization* (SOO) [38]. Though these algorithms do not require any auxiliary information (smoothness) or optimization, it has been shown that they do not perform quite as competitively as the Bayesian optimization framework, especially when prior knowledge is available.

3.6 Computational costs

Remember from chapter 2 that Gaussian Process regression has analytical expressions for the mean and variance of the posterior process. However, this exact inference is $O(n^3)$, where n is the number of training samples. This is caused by the inversion of K. pyGPGO uses a Cholesky decomposition that once computed, can reduce the cost of predicting to $O(n^2)$. If during the search procedure, however, we change K, for example, due to hyperparameter optimization, this $O(n^3)$ order is unavoidable for the traditional framework. Several approaches have been explored in the literature for approximating the output of the analytical solution. Mainly, there are two types of solutions, those that try to sparsify the process and those that use another type of surrogate model, such as Random Forests.

3.6.1 Approximations to the analytical GP. Alternative surrogates.

One of the first solutions is the *Sparse pseudo-input Gaussian Processes* (SPGP) [31]. This is a straightforward approach to model large n using $m < n$ pseudo-inputs to reduce the rank of the covariance matrix to m. This method forces the interaction between the $\boldsymbol{x}_{1:n}$ data points and test points $\boldsymbol{x}_*$ inducing m pseudo-inputs, achieving an approximate posterior in $O(nm^2 + m^3)$. Let $\boldsymbol{f}$ and $\boldsymbol{f}_*$ denote two sets of latent function (them being our training and testing points respectively). The assumption is that $\boldsymbol{f} and \boldsymbol{f}_*$ are independent given a third set of variables $\boldsymbol{u}$, that is:

$$p\left(\boldsymbol{f}_*, \boldsymbol{f}\right) = \int p(\boldsymbol{f}_*, \boldsymbol{f}, \boldsymbol{u})\, d\boldsymbol{u} \approx \int q(\boldsymbol{f}_*|\boldsymbol{u}) q(\boldsymbol{f}|\boldsymbol{u}) p(\boldsymbol{u})\, d\boldsymbol{u} = q(\boldsymbol{f}, \boldsymbol{f}_*) \tag{3.19}$$

where $\boldsymbol{u}$ is a vector representing the function values at the pseudo-inputs. Different pseudo-input Gaussian Process pseudo-input approximations specify their own form of $q(\boldsymbol{f}|\boldsymbol{u})$ and $q(\boldsymbol{f_*}|\boldsymbol{u})$ training and test conditionals [25]. How to choose the locations of these pseudo-inputs is another problem, is is usually done by maximizing the marginal log-likelihood of the SPGP [27]. Another approach uses variational inference [35] to marginalize the pseudo-inputs to maximize the fidelity to the original Gaussian Process. It has been noted, however, that the computational savings of the pseudo-input methods impact heavily variance estimates. In the Bayesian optimization framework, this variance of the posterior predictive distribution is heavily used to guide exploration, so this behavior is undesirable.

Another approach is named *Sparse spectrum Gaussian Processes* (SSGP). Using the ideas of pseudo-input methods, these methods apply the same concept in kernel spectral space [21]. From basic spectral analysis, Bochner's theorem states that a stationary kernel $k(\boldsymbol{x}, \boldsymbol{x}_*)$ defines a positive finite Fourier spectrum $s(\boldsymbol{\omega})$:

$$k(\boldsymbol{x}) = \frac{1}{(2\pi)^d} \int e^{-i\boldsymbol{\omega}^T \boldsymbol{x}} s(\boldsymbol{\omega}) d\boldsymbol{\omega} \tag{3.20}$$

We can normalize this spectrum to make it a valid probability density function, such as $p(\boldsymbol{\omega}) = \frac{s(\boldsymbol{\omega})}{\nu}$. Now evaluating the kernel is the same as the expectation of the Fourier basis with respect to $p(\boldsymbol{\omega})$:

$$k(\boldsymbol{x}, \boldsymbol{x}_*) = \nu \mathbb{E}_{\boldsymbol{\omega}} \left[e^{-i\boldsymbol{w}^T (\boldsymbol{x} - \boldsymbol{x}_*)} \right] \tag{3.21}$$

Monte Carlo techniques can be used to approximate this expectation using m samples from the spectral density, so that:

$$k(\boldsymbol{x}, \boldsymbol{x}_*) \approx \frac{\nu}{m} \sum_{i=1}^{m} e^{-i\boldsymbol{\omega}_{(i)}^T \boldsymbol{x}} e^{-i\boldsymbol{\omega}_{(i)}^T \boldsymbol{x}_*} \tag{3.22}$$

where $\boldsymbol{\omega}_{(i)} \sim s(\boldsymbol{\omega})/\nu$. The computational cost in this approach can be reduced to $O(nm^2 + m^3)$. It has been noted for this approximation method that whereas the uncertainty estimates are smoother than with the pseudo-input methods, observations away from the observed values exhibit irregular variance estimates. This again is undesirable behavior in the Bayesian Optimization framework.

As an alternative, several authors [15] have suggested using different surrogate models in the Bayesian Optimization framework. Special attention has been drawn towards the Random Forest regression model, in the context of sequential model-based algorithm configuration. Random Forests [5] are a very popular choice in the machine-learning community with very successful results in practice. They are an ensemble bagging tree-based method. Random Forests allow for training using sub-samples of data, giving it the ability to scale to large evaluation budgets, where exact analytical Gaussian Process regression would be infeasible in practice. The exploration strategy requires an uncertainty estimation for prediction at test points to apply the Bayesian optimization framework. The empirical variance in the predictions across trees was proposed as a substitute. This heuristic has been shown to work in practice [15].

Random Forests are known to provide very good estimates when doing *interpolation* of seen data (or in its neighbourhood), they are very poor *extrapolators*. On points far from training data, the predictions across all trees in the model are very similar, providing poor predictions and more importantly, providing extremely confident (but erroneous) variance estimates. While Gaussian Processes are also terrible extrapolators, they produce reliable variance estimations far from training data, yielding sometimes better estimates for exploration and exploitation in the framework. An example of a Random Forest with variance estimates on the sine data is provided in Figure 3.4.

Other authors [28] have suggested the use of the t-Student process instead, claiming that it naturally models heavy tailed behaviour. Similarly to our Gaussian Process theory, we could assume that $f(\boldsymbol{x}) \sim \mathcal{TP}(\nu, m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}))$. Since the conditional distribution of a multivariate Student-t is also multivariate Student-t, we can marginalize new observations $\boldsymbol{f}^*$ on $\boldsymbol{f}$ like we did in Section 2.3. That is, it can be proven that:

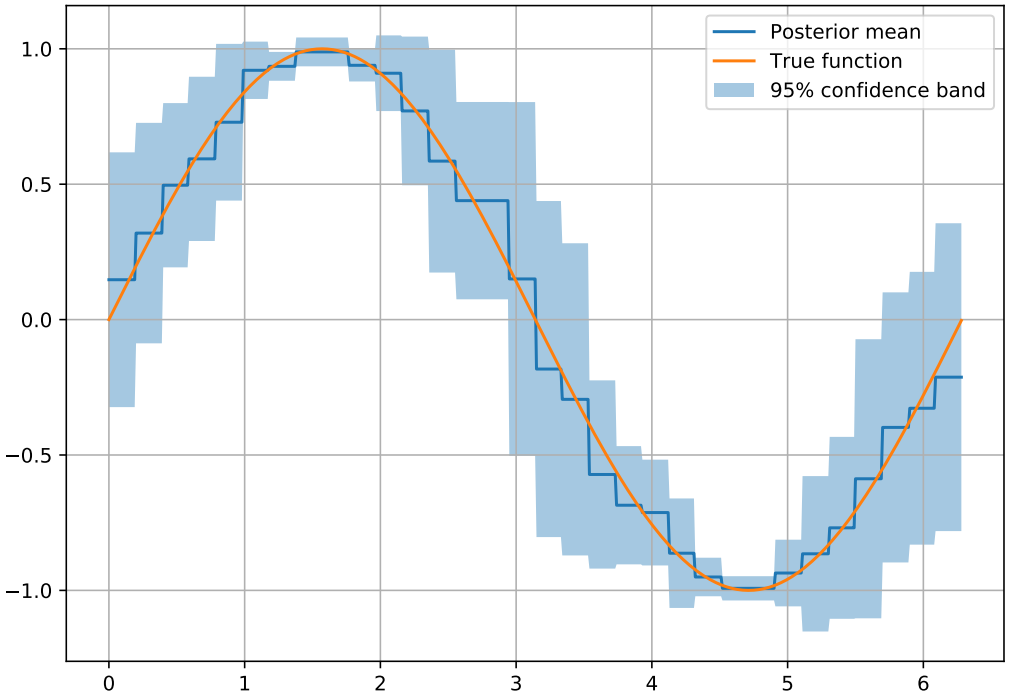
$$p(\boldsymbol{f}^* | \boldsymbol{f}) = \frac{p(\boldsymbol{f}^*, \boldsymbol{f})}{p(\boldsymbol{f})} \propto \left(1 + \frac{\beta_2}{\beta_1 + \nu - 2} \right)^{-\frac{\nu + n}{2}} \tag{3.23}$$

which turns out to be the kernel of another multivariate Student-t, that is:

$$\boldsymbol{f}^* | \boldsymbol{f} \sim \mathcal{T} \left(\nu + n_1, \boldsymbol{\phi}_2, \frac{\nu + \beta_1 - 2}{\nu + n_1 - 2} \tilde{K}_{**} \right) \tag{3.24}$$

where n_1 and n are the training and total sample numbers, ν are the degrees of freedom of the Student-t Process, $\beta_1 = \boldsymbol{y}^T K^{-1} \boldsymbol{y}$, $\boldsymbol{\phi}_2 = K_*^T K^{-1} \boldsymbol{y}$ and $\tilde{K}_{**} = K_{**} - K_*^T K^{-1} K_*$. Notice that the predictive mean is the same as the Gaussian Process case, while the predictive covariance now explicitly depends on the observations.

Figure 3.4: A Random Forest with variance estimates over trees trained on the sine function



3.6.2 Parallelization

The Bayesian Optimization framework is inherently a sequential decision problem, where each candidate point to sample is selected after fitting a Gaussian Process to the currently available data. However, and given the parallel nature of current CPU architectures, significant speed-ups in clock time can be achieved if they are made good use of. That implies evaluating the acquisition function in parallel. While not entirely parallel in nature, some authors have proposed approaches based on the imputation of yet-to-run evaluations. Given $\mathcal{D}_n = \{(\boldsymbol{x}_n, y_n)\}$ known data and $\mathcal{D}_p = \{\boldsymbol{x}_p\}$ remaining to evaluate data, an idea is to impute the latter, $\hat{\mathcal{D}}_p = \{(\boldsymbol{x}_p, \hat{y}_p)\}$, and then use the typical Bayesian optimization framework on the augmented data $\mathcal{D}_n \cup \hat{\mathcal{D}}_p$.

Simple strategies have been proposed over time. The *constant liar* strategy proposes $\hat{y}_p = c, \forall p$, where $c \in \mathbb{R}$ is a predefined constant. Other strategies like the *kriging believer* uses the Gaussian Process posterior predictive mean instead: $\hat{y}_p = \mu_n(\boldsymbol{x}_p)$. More complex approaches have been proposed instead, for example [32] proposed the use of s fantasies sampled from each unfinished experiment (out of a total of J) from the full GP posterior predictive distribution. Then they are averaged in a Monte Carlo fashion:

$$\alpha(x, \mathcal{D}_n, \mathcal{D}_p) = \int_{\mathbb{R}^J} \alpha(x, \mathcal{D}_n \cup \hat{\mathcal{D}}_p) P(\hat{y_{1:J}}, \mathcal{D}_n) dy_{p,1:J} \tag{3.25}$$

$$\approx \frac{1}{S} \sum_{i=1}^{S} \alpha(x, \mathcal{D}_n \cup \hat{\mathcal{D}}_p^{(s)}) \tag{3.26}$$

and $\hat{\mathcal{D}}_p^{(s)} \sim P(\hat{y}_{1:J}, \mathcal{D}_n)$. It has demonstrated empirically that this approach works reasonably well when α is chosen to be the Expected Improvement acquisition function. Other attempts [10] have also been made using GP-UCB. Note that while these approaches are valid, they are not parallel per se. A true parallel approach to Bayesian optimization would propose simultaneously a set of candidates, or would use the information of posterior Gaussian Processes computed in another thread to make the sequential decision.

3.7 Step-by-step examples

We have explored the basics of Gaussian Process regression and Bayesian Optimization by now. To provide a more thorough understanding on the logic behind Algorithm 6, we will provide two different examples in this section. While not very complex, they serve illustrative purposes on how the optimization framework works. We hope this section will clear any practical consideration on how the procedure selects the next point to sample, graphically.

3.7.1 Optimizing the sine function

Again, it is no surprise for the reader that we choose the sine function as our first example to provide an intuition on how the Bayesian Optimization framework works step-by-step. We will be optimizing our function within the boundaries $x \in [0, 2\pi]$, where we know an optima is exactly at $x^* = \pi/2$. This example is particularly interesting to see how the next point is chosen balancing exploration and exploitation of the acquisition function. Now, as in most practical cases throughout this thesis, we will use the Expected Improvement acquisition function. The step-by-step optimization procedure for 6 complete epochs can be checked in Figure 3.5. The code to produce these Figures can be checked in Appendix A.8.

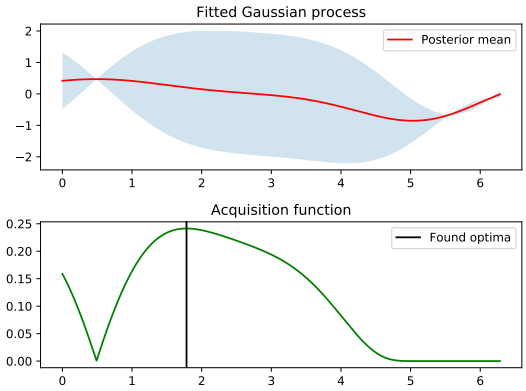
It can be appreciated that the chosen acquisition function leverages achieves a very reasonable compromise between mean and variance in the first stages of optimization, but exploits heavily in steps 3 and 4. It tries to explore again in step 5, and comes back to exploit at the end of the procedure.

3.7.2 Optimizing the Rastrigin function

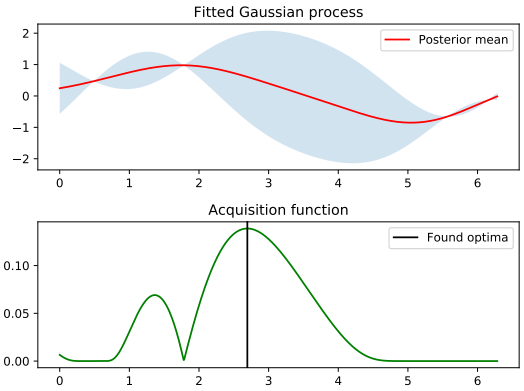
In this example, we will try to optimize the two-dimensional Rastrigin function:

Figure 3.5: Six complete optimization epochs in the Bayesian Optimization framework for the target sine function in $x \in [0, 2\pi]$.

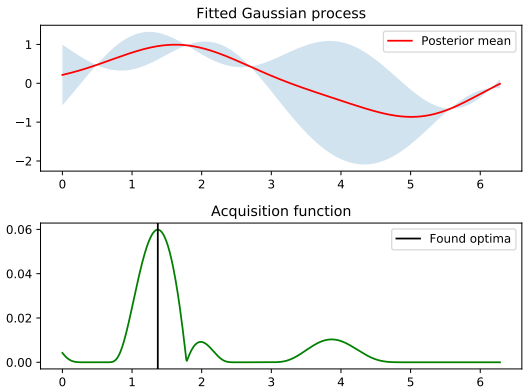
(a) Step 1



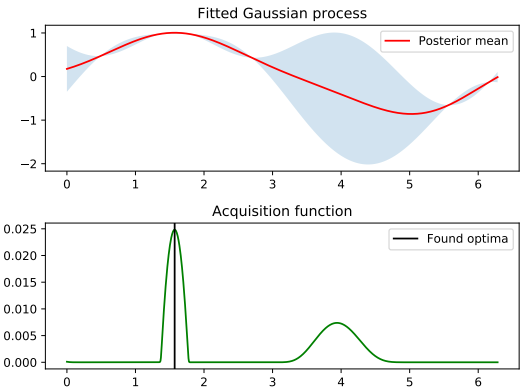
(b) Step 2



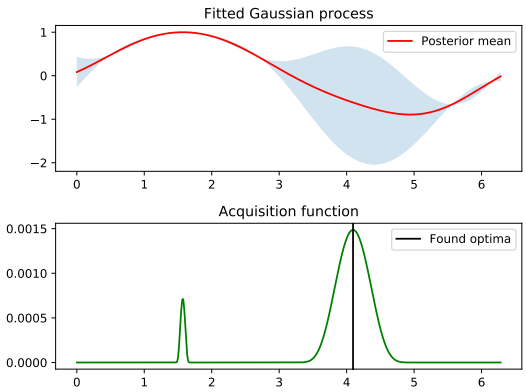
(c) Step 3



(d) Step 4



(e) Step 5



(f) Step 6

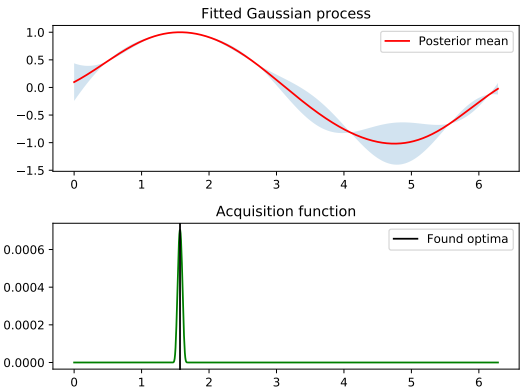
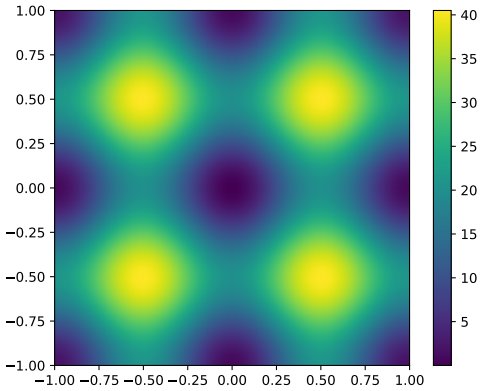


Figure 3.6: A 2D representation of the Rastrigin function for $x, y \in [-1, 1]$.



$$f(x, y|a, b) = (a - x)^2 + b(y - x^2)^2 \tag{3.27}$$

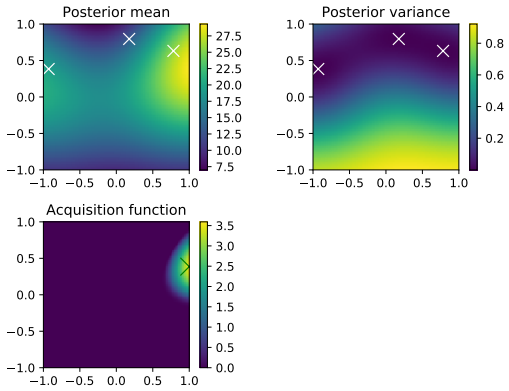
For given hyperparameters $a = 1$, $b = 100$ and in a bounding box defined by $x, y \in [-1, 1]$. A biplot of this function can be checked in Figure 3.6. This particular version of the Rastrigin function has four different local optima, therefore it is multimodal. We would expect our GPGO algorithm to converge to one of these optima, while still exploring the space at earlier stages of optimization.

Given the low-dimensional space we are optimizing, we will plot the posterior mean and variance of the Gaussian Process, as well as the acquisition function. Again, the Expected Improvement acquisition function is used. The complete step-by-step optimization procedure for 6 epochs can be seen in Figure 3.7. For a more visually appealing explanation of the same procedure, there is a video in the associated GitHub repository of this thesis. The code to generate these figures is available in Appendix A.9.

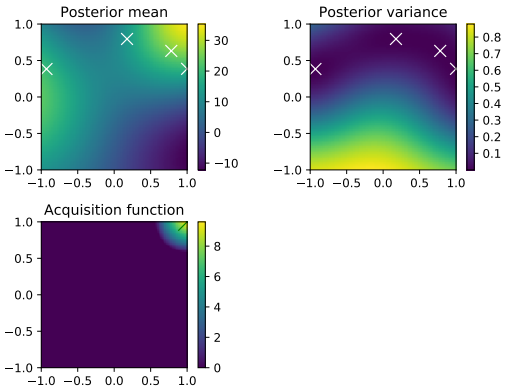
It can be seen that during the previous steps before fitting a Gaussian Process, that a random evaluation is reasonably close to one of the optima. This causes the acquisition function to exploit that area in the first step. In the second step, however, it starts exploring the area that is close to a second local optima, mostly because it is a large-variance area. It exploits in the third step, and keeps exploring the rest of the space during the following stages, therefore efficiently balancing exploitation and exploration as intended in the optimization procedure.

Figure 3.7: Six complete optimization epochs in the Bayesian Optimization framework for the target Rastrigin 2D function.

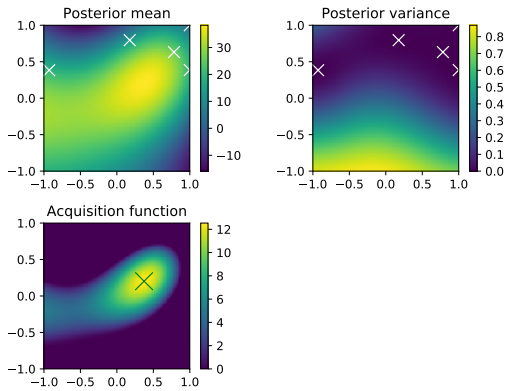
(a) Step 1



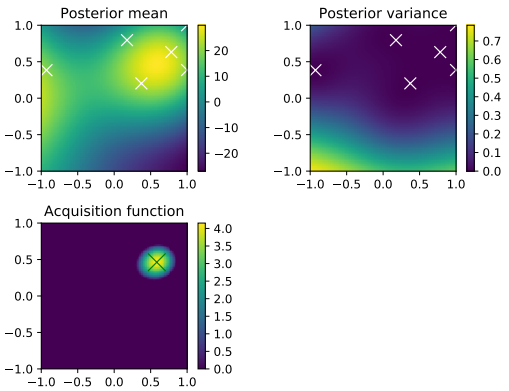
(b) Step 2



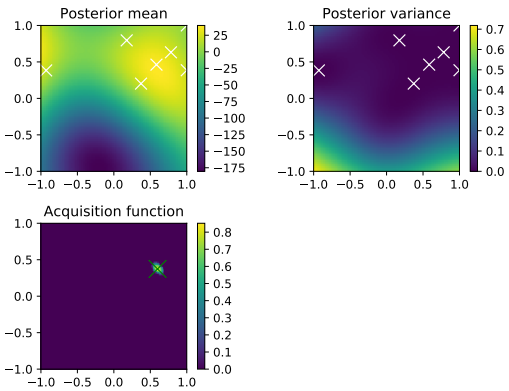
(c) Step 3



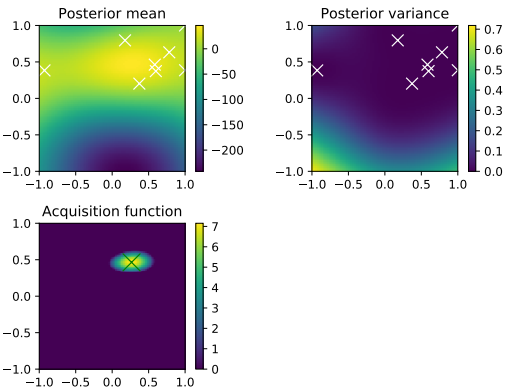
(d) Step 4



(e) Step 5



(f) Step 6



Bibliography

[1] P. Auer et al. "Gambling in a rigged casino: The adversarial multi-armed bandit problem". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. Vol. 68. 68. 1995, pp. 322–331. ISBN: 0-8186-7183-1. DOI: 10.1109/SFCS.1995.492488.

[2] Rémi Bardenet and Balázs Kégl. "Surrogating the surrogate: accelerating Gaussian-process-based global optimization with a mixture cross-entropy algorithm". In: *Proceedings of the 27th International Conference on Machine Learning* (2010), pp. 55–62.

[3] Romain Benassi, Julien Bect, and Emmanuel Vazquez. "Robust Gaussian process-based global optimization using a fully Bayesian expected improvement criterion". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6683 LNCS. 2011, pp. 176–190. ISBN: 9783642255656. DOI: 10.1007/978-3-642-25566-3_13.

[4] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. ISSN: 1532-4435.

[5] Leo Breiman. "Random forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 08856125. DOI: 10.1023/A:1010933404324. arXiv: /dx.doi.org/10.1023{\%}2FA{\%}3A1010933404324 [http:].

[6] 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". In: *ArXiv* (2010), p. 49. DOI: 1012.2599. arXiv: 1012.2599.

[7] Sébastien Bubeck et al. "X-armed bandits". In: *Multi-Armed Bandits* (2010), pp. 1–38. ISSN: 15324435. arXiv: arXiv:1001.4475v2. URL: http://arxiv.org/abs/1001.4475.

[8] Adam D Bull. "Convergence Rates of Efficient Global Optimization Algorithms". In: *Journal of Machine Learning Research* 12 (2011), pp. 2879–2904. ISSN: 1532-4435. arXiv: arXiv:1101.3501v3.

[9] Peadar Coyle. "Probabilistic Programming and PyMC3". In: *PROC. OF THE 8th EUR. CONF. ON PYTHON IN SCIENCE* (2015). arXiv: 1607.0379.

[10] Thomas Desautels, Andreas Krause, and Joel Burdick. "Parallelizing exploration-exploitation tradeoffs with gaussian process bandit optimization". In: *Active Learning* 15 (2012), pp. 3873–3923. arXiv: 1206.6402. URL: http://arxiv.org/abs/1206.6402.

[11] Nikolaus Hansen and Andreas Ostermeier. "Completely Derandomized Self-Adaptation in Evolution Strategies". In: *Evolutionary Computation* 9.2 (2001), pp. 159–195. ISSN: 1063-6560. DOI: 10.1162/106365601750190398. URL: http://www.mitpressjournals.org/doi/abs/10.1162/106365601750190398.

[12] Phillipp Hennig and Christian J Schuler. "Entropy Search for Information-Efficient Global Optimization". In: *Machine Learning Research* 13.1999 (2012), pp. 1809–1837. ISSN: 15324435. DOI: http://dx.doi.org/10.1063/1.1699114. arXiv: 1112.1217.

[13] José Miguel Hernández-Lobato, Matthew W Hoffman, and Zoubin Ghahramani. "Predictive Entropy Search for Efficient Global Optimization of Black-box Functions". In: *Advances in Neural Information Processing Systems 28* (2014), pp. 1–9. ISSN: 10495258. arXiv: arXiv:1406.2541v1. URL: https://jmhldotorg.files.wordpress.com/2014/10/pes-final.pdf.

[14] Matthew Hoffman, Eric Brochu, and Nando De Freitas. "Portfolio Allocation for Bayesian Optimization". In: *Conference on Uncertainty in Artificial Intelligence* (2011), pp. 327–336. arXiv: arXiv:1009.5419v1.

[15] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6683 LNCS. 2011, pp. 507–523. ISBN: 9783642255656. DOI: 10.1007/978-3-642-25566-3_40.

[16] A Ilinskas. "Global Optimization Based on a Statistical Model and Simplicial Partitioning". In: 1221.02 (2002).

[17] D. R. Jones. "A Taxonomy of Global Optimization Methods Based on Response Surfaces". In: *Journal of Global Optimization* 21 (2001), pp. 345–383. ISSN: 09255001. DOI: 10.1023/A:1012771025575. URL: http://www.ingentaconnect.com/content/klu/jogo/2001/00000021/00000004/00360694.

[18] Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. "Thompson sampling: An asymptotically optimal finite-time analysis". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7568 LNAI. 2012, pp. 199–213. ISBN: 9783642341052. DOI: 10.1007/978-3-642-34106-9_18. arXiv: arXiv:1205.4217v2.

[19] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *Proceedings of ECML* (2006), pp. 282–203. ISSN: 03029743. DOI: 10.1007/11871842. URL: http://link.springer.com/chapter/10.1007/11871842%7B%5C_%7D29.

[20] T. L. Lai and Herbert Robbins. "Asymptotically efficient adaptive allocation rules". In: *Advances in Applied Mathematics* 6.1 (1985), pp. 4–22. ISSN: 10902074. DOI: 10.1016/0196-8858(85)90002-8.

[21] M Lázaro-Gredilla et al. "Sparse Spectrum Gaussian Process Regression". In: *Journal of Machine Learning Research* 11 (2010), pp. 1865–1881. ISSN: 1532-4435. DOI: 10.1137/10080991X. arXiv: arXiv:1304.6949v1. URL: http://www.jmlr.org/papers/volume11/lazaro-gredilla10a/lazaro-gredilla10a.pdf.

[22] Daniel James Lizotte, Russell Greiner, and Dale Schuurmans. "An experimental methodology for response surface optimization methods". In: *Journal of Global Optimization* 53.4 (2012), pp. 699–736. ISSN: 09255001. DOI: 10.1007/s10898-011-9732-z.

[23] Marco Locatelli. "Bayesian algorithms for one-dimensional global optimization". In: *J. Global Optim.* 10.1 (1997), pp. 57–76. ISSN: 0925-5001. DOI: Doi10.1023/A:1008294716304.

[24] Radford M. Neal. "MCMC using Hamiltonian dynamics". In: *Handbook of Markov Chain Monte Carlo* (2011), pp. 113–162. ISSN: ¡null¿. DOI: doi:10.1201/b10905-6. arXiv: 1206.1901.

[25] Joaquin Quiñonero-candela, Carl Edward Rasmussen, and Ralf Herbrich. "A unifying view of sparse approximate Gaussian process regression". In: *Journal of Machine Learning Research* 6 (2005), pp. 1935–1959. ISSN: 1533-7928. URL: http://jmlr.org/papers/volume6/quinonero-candela05a/quinonero-candela05a.pdf.

[26] Ali Rahimi and Ben Recht. "Random features for large-scale kernel machines". In: *Advances in neural information ...* 1 (2007), pp. 1–8. ISSN: 0033-6599. DOI: 10.1.1.145.8736. URL: http://machinelearning.wustl.edu/mlpapers/paper%7B%5C_%7Dfiles/NIPS2007%7B%5C_%7D833.pdf.

[27] M Seeger, Cki Williams, and Nd Lawrence. "Fast forward selection to speed up sparse Gaussian process regression". In: *Workshop on AI and Statistics* 9 (2003), p. 2003. URL: http://ipg.epfl.ch/%7B~%7Dseeger/lapmalmainweb/papers/aistats03-final.pdf.

[28] Amar Shah, Andrew Gordon Wilson, and Zoubin Ghahramani. "Bayesian Optimization using Student-t Processes". In: 2 (2014), pp. 1–5.

[29] Bobak Shahriari et al. "An Entropy Search Portfolio for Bayesian Optimization". In: *arXiv preprint arXiv:1406.4625* (2014), p. 10. arXiv: 1406.4625. URL: http://arxiv.org/abs/1406.4625.

[30] Bobak Shahriari et al. *Taking the human out of the loop: A review of Bayesian optimization*. 2016. DOI: 10.1109/JPROC.2015.2494218. arXiv: arXiv:1011.1669v3.

[31] Edward Snelson and Zoubin Ghahramani. "Sparse Gaussian Processes using Pseudo-inputs". In: *Advances in Neural Information Processing Systems 18* (2006), pp. 1257–1264. ISSN: 1049-5258. DOI: 10.1.1.60.2209. arXiv: 1402.1389. URL: http://papers.nips.cc/paper/2857-sparse-gaussian-processes-using-pseudo-inputs.pdf.

[32] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Adv. Neural Inf. Process. Syst. 25* (2012), pp. 1–9. ISSN: 10495258. DOI: 2012arXiv1206.2944S. arXiv: arXiv:1206.2944v2.

[33] Niranjan Srinivas et al. "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design". In: (2009). ISSN: 00189448. DOI: 10.1109/TIT.2011.2182033. arXiv: 0912.3995. URL: http://arxiv.org/abs/0912.3995%7B%5C%%7D0Ahttp://dx.doi.org/10.1109/TIT.2011.2182033.

[34] Niranjan Srinivas et al. "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design". In: *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)* (2010), pp. 1015–1022. ISSN: 00189448. DOI: 10.1109/TIT.2011.2182033. arXiv: 0912.3995. URL: http://arxiv.org/abs/0912.3995.

[35] Michalis Titsias. "Variational Learning of Inducing Variables in Sparse Gaussian Processes". In: *Aistats* 5 (2009), pp. 567–574. ISSN: 15324435. URL: http://eprints.pascal-network.org/archive/00006353/.

[36] Emmanuel Vazquez and Julien Bect. "Convergence properties of the expected improvement algorithm with fixed mean and covariance functions". In: *Journal of Statistical Planning and Inference* 140.11 (2010), pp. 3088–3095. ISSN: 03783758. DOI: 10.1016/j.jspi.2010.04.018. arXiv: 0712.3744.

[37] Julien Villemonteix, Emmanuel Vazquez, and Eric Walter. "An informational approach to the global optimization of expensive-to-evaluate functions". In: *Journal of Global Optimization* 44.4 (2009), pp. 509–534. ISSN: 09255001. DOI: 10.1007/s10898-008-9354-2. arXiv: 0611143 [cs].

[38] Ziyu Wang et al. "Bayesian Multi-Scale Optimistic Optimization". In: *AISTATS* 33 (2014), p. 15. ISSN: 15337928. arXiv: 1402.7005. URL: http://arxiv.org/abs/1402.7005.

Chapter 4

Experiments

In this chapter we will expose all the results using all implemented code for Gaussian Process regression and Bayesian Optimization in pyGPGO. First, we start by providing some benchmarking rules, how performance shall be evaluated across different models and datasets. In general, the models presented here hopefully span enough diversity: support vector machines , neural networks and bagging/boosting methods are among these candidates. Datasets come from very different research areas, spanning from biophysics to medicine. Some datasets will span an entire section, explaining the rationale behind the problem, while results for others will be briefly discussed. We hope to show here that Bayesian Optimization works reasonably well for hyperparameter optimization of machine-learning models, and can outperform other well known strategies in the majority of cases.

4.1 Benchmarking rules

In this section we first present other strategies for machine-learning hyperparameter optimization, followed by an explanation of the rules used for benchmarking as well as some theoretical background of the models explored here.

4.1.1 Other strategies for hyper-parameter optimization

Hyper-parameter optimization is usually the most tedious part in fitting a machine-learning algorithm. In practice, we are interested in models whose loss, evaluated on an independent part of data is as low as possible. Different hyper-parameter choices lead to different losses, therefore finding the optimal set is of importance. In reality, we will never know that the point found is the global optimum, but from a practical point of view, we are only interested in finding the model that works best in a production setting.

In the machine-learning community, hyper-parameter optimization is often overlooked, and in fact, some of the most famous models (e.g. Random Forests) have been shown to be somewhat insensitive to hyper-parameter optimization [49]. Two common strategies for finding *better* hyper-parameters are presented here: grid search and random search. These two methods are used in different occasions: in particular grid search is typically used when the search space is not too large and the model to fit is not expensive to evaluate. Random search is more popular when it becomes impractical to explore the search. More details of these two common strategies are presented below.

- **Grid search**. Also known as parameter sweep. This is simply an exhaustive search in a user specified subset of parameter space. Search bounds have to be set manually. When multiple hyper-parameters have to be optimized over their sets, grid search considers the Cartesian product of all of them. This poses a problem when the number of hyper-parameters to optimize grows, also known as the curse of dimensionality. While suffering from this problem, grid search is still widely used for small to mid-sized problems, where function evaluations are not very expensive. Also, it is embarrassingly parallel: function evaluations can be distributed over in a simple way.

- **Random search**. Grid search is exhaustive since it considers all possible evaluations over a Cartesian product over parameter sets. Randomized search in hyper-parameter space is also a very popular method for the task at hand. In particular, this method has been shown to work considerably better in high-dimensional spaces than grid search [6]. There is also evidence that often sometimes some hyper-parameters do not affect the loss significantly.

4.1.2 Evaluation metrics

Throughout all the experiments, we will be minimizing loss functions, in some form or the other. Every problem presented here is formulated either as a regression or classification task, therefore it is of importance to define early on what type of loss to use in each problem. For binary classification problems, we will use the logarithmic loss, also known as binary cross-entropy, defined by:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{n} = \sum_i \left(y_i \log(\hat{y_i}) + (1 - y_i) \log(1 - \hat{y_i})\right) \tag{4.1}$$

The log-loss is very popular in the machine-learning community [7]. This particular metric does not only take into account whether a classifier makes the right decision given a threshold c, (like the 0/1 loss would), but also the confidence in predictions $\hat{\boldsymbol{y}}$. It is also very natural since it is just the negative log-likelihood of a Bernoulli random variable. The use of the logarithm both punishes erroneous extremely confident positive or negative predictions.

We generalize the previous metric for multi-class classification problems. The following expression is typically called categorical cross-entropy:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{p}}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \log(\hat{p}_{ij}) \tag{4.2}$$

where $\hat{p}_{ij}$ is the predicted probability of a sample i belonging to class j, and m is the number of classes considered. For continuous regression problems, the loss that we use is the typical mean squared error, defined by:

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_i \left(y_i - \hat{y_i}\right)^2 \tag{4.3}$$

We believe that the losses proposed here are very natural choices in both classification and regression problems.

While we have discussed the metrics to evaluate in each predictive problem, we still need to define how these losses will be evaluated in each step. To evaluate the performance of a given hyper-parameter optimization procedure, we have to balance both the performance in terms of loss and the number of evaluations necessary in order to get to a satisfactory solution. In practice, this is exactly how we will benchmark the different strategies, through a plot where the x-axis represent the number of function evaluations, and the y-axis the best log-loss found.

Evaluating the loss function itself can lead to multiple different values depending on the test values. One could choose an approach where this loss is evaluated on a single holdout test. This would lead to noisy estimates, however. We choose the more stable approach of performing a shuffled $k = 5$ cross-validation scheme to obtain a more reliable loss estimate. In practice, this means that we fit 5 models with the same architecture to different train/test splits and average the loss results in each. A total of $n = 53$ functions evaluations will be allowed for each strategy in all datasets. This accounts for the fact that the Bayesian Optimization needs at least 3 function evaluations to fit a surrogate Gaussian Process regressor in the beggining.

4.1.3 Bayesian optimization setup

For all the tests performed with the Bayesian Optimization scheme, we choose the standard squared exponential kernel with a starting default characteristic length-scale $l = 1$, signal variance $\sigma_f^2 = 1$ and noise variance $\sigma_n^2 = 0$. These three hyper-parameters will be continuously adapted using a Type-II Maximum Likelihood approach using gradients during the optimization process, as detailed in Section 2.5.1. Different acquisition functions will be tested in each problem: Expected Improvement, GP-UCB ($\beta = .5$) and GP-UCB ($\beta = 1.5$). All features in all datasets are scaled by default to have zero mean and unit variance. The random seed is fixed for all experiments for reproducibility.

4.1.4 Machine-learning models used

Since the shape of our objective function depends on both the dataset and the predictor, we try to span as many different types of machine-learning models as possible to provide the most extensive evaluation as possible. Except for rare occasions where we could not fit a model to a particular dataset for numerical conditions, all models are evaluated in all datasets with the same number of parameters and bounds to optimize over. We consider the following models: Support Vector Machines (SVM) with radial basis function kernel, K-nearest neighbors (KNN), Neural Networks with a single hidden layer (MLP) and Gradient Boosting Machines (GBM). We briefly detail how these work now.

Support Vector Machines

A SVM model [10] uses the concept of hyperplanes in high or infinite dimensional space in order for classification or regression purposes. In particular, a good classification model is the one that places an hyperplane that achieves maximum distance to training points of any class. Intuitively, the larger this margin, the lower the generalization error of the model. For classification, the model can be defined via optimization. Assume $\boldsymbol{x}_i \in \mathbb{R}^p$, and $y_i \in \{0, 1\}$, then the problem is to minimize:

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\epsilon}} \ & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{n}\epsilon_i \\ s.t. \ & y_i(\boldsymbol{w}^T\phi(\boldsymbol{x}_i) + b) \geq 1 - \epsilon_i \\ & \epsilon_i \geq 0, \ \ i = 1, \ldots, n \end{aligned} \tag{4.4}$$

In practice it makes more sense to minimize its dual:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \ & \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ s.t. \ & \boldsymbol{y}^T\boldsymbol{\alpha} = 0 \\ & 0 \leq \alpha_i \leq C, \ \ , i = 1, \ldots, n \end{aligned} \tag{4.5}$$

where $\boldsymbol{e}$ is the unit vector, C is an hyperparameter controlling an upper bound, Q is a $n \times n$ semidefinite positive matrix defined by $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and K is our defined kernel function. Finally, the decision function is defined as:

$$d(\boldsymbol{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i K(\boldsymbol{x_i}, \boldsymbol{x}) + \rho\right) \tag{4.6}$$

For regression problems we now consider $y_i \in \mathbb{R}$ and we try to minimize:

$$\begin{aligned} \min_{\boldsymbol{w},b,\gamma,\gamma^*} \ & \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{n}(\gamma_i + \gamma_i^*) \\ s.t. \ & y_i - \boldsymbol{w}^T\phi(\boldsymbol{x}_i) - b \leq \epsilon + \gamma_i \\ & \boldsymbol{w}^T\phi(\boldsymbol{x}_i) + b - y_i \leq \epsilon + \gamma_i^* \\ & \gamma_i, \gamma_i^* \geq 0 \ \ i = 1, \ldots, n \end{aligned} \tag{4.7}$$

Table 4.1: Parameters to be optimized for all SVM models in the benchmark.

Parameter	Type	Bounds
C	$\mathbb{R}^+$	$[10^{-5}, 10^5]$ (log-scaled)
γ	$\mathbb{R}^+$	$[10^{-5}, 10^5]$ (log-scaled)

Table 4.2: Parameters to be optimized for all KNN models in the benchmark.

Parameter	Type	Bounds
k	Integer	$\{10, \ldots, 50\}$

Likewise, we normally minimize its dual:

$$\begin{aligned} \min_{\boldsymbol{\alpha},\boldsymbol{\alpha}^*} \ & \frac{1}{2}(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)Q(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*) + \epsilon \boldsymbol{e}^T(\boldsymbol{\alpha}+\boldsymbol{\alpha}^*) - \boldsymbol{y}^T(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*) \\ s.t. \ & \boldsymbol{e}^T(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*) = 0 \\ & 0 \leq \alpha_i, \alpha_i^* \leq C, \ \ i = 1, \ldots, n \end{aligned} \tag{4.8}$$

Our decision function now becomes:

$$g(\boldsymbol{x}) = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(\boldsymbol{x_i}, \boldsymbol{x}) + \rho \tag{4.9}$$

For all the testing involved in the following sections, we will use `scikit-learn` implementation of Support Vector Machines, which is in turn based on `LibSVM` [9]. We will optimize over two hyperparameters, C, the penalty parameter in the error term of Equations 4.5 and 4.8, and γ, an radial basis function hyperparameter controlling the smoothness of the decision function. They will be optimized on the range defined by Table 4.1.

K-nearest neighbors

In contrast to other strategies presented here, K-nearest neighbors does not approach learning by constructing a generalizable internal model, but simply stores training instances. Classification for an example is then performed using a majority vote of its closest points in distance. For the case of regression, we take the average of mentioned points target instead. Since computing a whole distance matrix for all examples is computationally expensive ($\mathcal{O}(dn^2)$ for n samples and d dimensions), several alternatives have been proposed. KDTree [26] is arguably one of the most popular ones. Intuitively, it works the following way: if we know points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are far in space, and we know point $\boldsymbol{x}_k$ is close to $\boldsymbol{x}_j$, then we know $\boldsymbol{x}_i$ and $\boldsymbol{x}_k$ must be far in space without *explicitly* having to compute their distance. It can be proven that this can reduce the computational complexity to $\mathcal{O}(dn \log n)$. For all benchmarking run in this work, we optimize only parameter k, the number of neighbors to consider, over a range specified in Table 4.2.

Gradient Boosting Machines

Boosting [44] is a machine-learning technique for simultaneously reducing the bias and variance of a classifier or regressor. It is based on the concept of ensembles, that is, a set of weak models, such as trees that are combined in a smart way to produce a strong model. In particular, Gradient Boosting Machines [15] are a particular instance of models using this boosting principle. It builds its internal model considering tree models in a stage-wise fashion and generalizes them by optimizing a given differentiable loss function. Assuming training data $\{\boldsymbol{x}_i, y_i\} \quad i = 1, \ldots, n$, the algorithm works by approximating a function $\hat{F}(\boldsymbol{x})$ to an original $F(\boldsymbol{x})$, which minimizes the expected value of some loss function $\mathcal{L}(\boldsymbol{y}, F(\boldsymbol{x}))$, that is:

Table 4.3: Parameters to be optimized for all GBM models in the benchmark.

Parameter	Type	Bounds
`learning_rate`	$\mathbb{R}^+$	$\left[10^{-5}, 10^{-2}\right]$
`n_estimators`	Integer	$\{10, \ldots, 100\}$
`max_depth`	Integer	$\{2, \ldots, 100\}$
`min_samples_split`	Integer	$\{2, \ldots, 100\}$

$$\hat{F}(\boldsymbol{x}) = \arg\min_{F} \mathbb{E}_{\boldsymbol{x},y} \left[\mathcal{L}(\boldsymbol{y}, F(\boldsymbol{x})\right] \tag{4.10}$$

Gradient boosting machine defines F to be a weighted sum of weak learners h_i from some class $\mathcal{H}$:

$$F(\boldsymbol{x}) = \sum_{i=1}^{n} \gamma_i h_i(\boldsymbol{x}) + c \tag{4.11}$$

We start by some constant approximation F_0 and expand it iteratively:

$$\begin{aligned} F_0(\boldsymbol{x}) &= \arg\min_{\gamma} \sum_{i=1}^{n} \mathcal{L}(y_i, \gamma) \\ F_m(\boldsymbol{x}) &= F_{m-1}(\boldsymbol{x}) + \arg\min_{f \in \mathcal{H}} \sum_{i=1}^{n} \mathcal{L}(y_i, F_{m-1} + f(\boldsymbol{x}_i)) \end{aligned} \tag{4.12}$$

The problem comes when trying to optimize and arbitrary f for any loss $\mathcal{L}$, so instead, we use gradient descent to minimize:

$$\begin{aligned} F_m(\boldsymbol{x}) &= F_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_{F_{m-1}} \mathcal{L}(y_i, F_{m-1}(x_i)) \\ \gamma_m &= \arg\min_{\gamma} \sum_{i=1}^{n} \mathcal{L}\left(y_i, F_{m-1}(\boldsymbol{x}_i) - \gamma \frac{\partial \mathcal{L}\left(y_i, F_{m-1}(\boldsymbol{x}_i)\right)}{\partial F_{m-1}(\boldsymbol{x}_i)}\right) \end{aligned} \tag{4.13}$$

γ is then chosen by some univariate optimization algorithm, such as line search. The most popular variant of Gradient Boosting Machines is Gradient Boosting Trees, where we choose f to be some tree classifier/regressor, such as CART 4.5 [32]. For the benchmarks considered in this work, we use `scikit-learn`'s GBM implementation, and try to optimize the hyperparameters defined by Table 4.3. The `learning_rate` parameter shrinks the contribution of each tree, `n_estimators` is the number of weak trees to fit, `max_depth` is the maximum depth of each weak tree, and `min_samples_split` is the minimum required number of samples to split a node in each tree.

Multilayer perceptron

Neural networks are a popular model now, specially with the rise of Deep Learning [31] over the last years. In this work, we will only consider neural network models with a single hidden layer, or Multilayer Perceptron (MLP) models. The output of one layer, given some input $\boldsymbol{x} \in \mathbb{R}^p$ is a function $f : \mathbb{R}^p \to \mathbb{R}^q$. Since a MLP contains only one hidden layer, the output of the whole model for either regression or binary classification can be written as:

$$f(\boldsymbol{x}) = G\left(b^{(2)} + W^{(2)}\left(s\left(b^{(1)} + W^{(1)}\boldsymbol{x}\right)\right)\right), \tag{4.14}$$

where $W^{(1)}$ and $W^{(2)}$ is a matrix of learned weights of size $p \times q$ and $q \times 1$ respectively, $b^{(1)}$ and $b^{(2)}$ are bias vectors and both G and s are non-linear differentiable functions. In practice, s will be a rectified linear unit function, that is:

Table 4.4: Parameters to be optimized for all MLP models in the benchmark.

Parameter	Type	Bounds
`hidden_layer_size`	Integer	$[5, 50]$
`alpha`	$\mathbb{R}^+$	$[0, 0.9]$

$$s(\boldsymbol{x}) = \max(0, \boldsymbol{x}), \tag{4.15}$$

and $G(\boldsymbol{x})$ will be the logistic function for binary classification:

$$G(\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{x})} \tag{4.16}$$

For regression, no non-linearity is applied. For c multiple classes, $W^{(2)}$ changes size to $q \times c$, and typically chooses G to be the softmax function:

$$G(\boldsymbol{x})_j = \frac{\exp(\boldsymbol{x}_j)}{\sum_{i=1}^{c} \exp(\boldsymbol{x}_i)}, \quad j = 1, \ldots, c \tag{4.17}$$

Matrices $W^{(1)}$, $W^{(2)}$ and biases $b^{(1)}$, $b^{(2)}$ are trained using backpropagation [43] through the use of stochastic gradient descent (SGD) [8]. For this particular piece of work, we use the Multilayer Perceptron implementation of `scikit-learn`, and optimize over hyperparameters detailed in Table 4.4. In particular, we consider the number of hidden units in the hidden layer and α, a parameter controlling L_2 regularization on the learned weights.

4.2 The binding affinity dataset

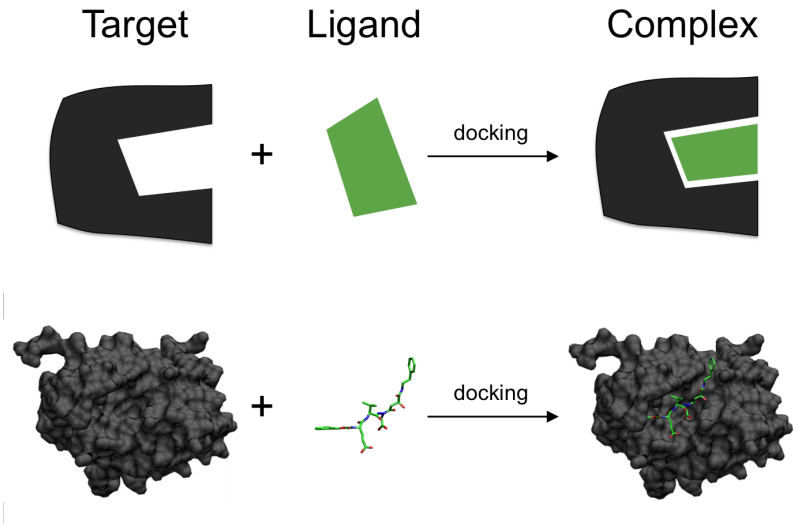
We start our benchmarking with one of the most popular problems in computational chemistry, protein-ligand binding affinity prediction. Summarizing, this is defined as a supervised learning problem where two interacting molecules intervene, a target protein and a small drug-like molecule binding to the former.

4.2.1 Description of the problem

Docking procedures (see Figure 4.1) in structural biology typically work as follows: first, by generating a large number of poses of the ligand (a small drug-like molecule compared to the host protein). A pose encompasses position, orientation and conformation of said ligand. Once enough poses are generated, a *scoring function* is in charge of re-ranking these, that is, its job is to find the correct pose amongst the generated. Correct poses have more *strength* to the binding site of the target than incorrect ones, and this is typically quantified by means of dissociation (K_d), inhibition constant (K_i) or free energy. These quantities are real valued, and can normally range from 10^{-9} to 10^6 kcal/mol in studies. To account for this very large range of affinities, one typically defines a target variable as $y = -\log_{10} K_{d,i}$ and considers it as a classical regression problem.

While many accurate and reliable algorithms exist for pose generation, the main drawback in docking studies continues to be the scoring function itself [30]. This, therefore continues to be an important open problem in computational chemistry. Over the years, plenty of scoring functions have been developed, for which the most common be classified into empirical [16, 27], force-field based [20] or knowledge-based [39, 18]. These *classical* scoring functions do not fully account for certain physical processes that are important for molecular recognition, therefore limiting their ability to rank for some particular protein-ligand binding pairs. Furthermore, each scoring function assumes a particular functional relationship between some variables characterizing the protein-ligand binding complex and their corresponding binding affinity. For example, typical scoring functions can take the form of a weighted sum of physico-chemical properties (as in the case of a Linear Regression (LR) or Partial Least Squares (PLS) [1]). In general, these scoring functions are evaluated from a regression perspective, reporting metrics such as mean squared

Figure 4.1: Small illustration of the docking procedure. Ligand poses generating by a docking program are ranked according to a given scoring function, hopefully resulting in a valid complex.



error (MSE) or Pearson's Correlation Coefficient.

There is, however, an inherent drawback to these classical approaches: assuming a rigid functional between complex descriptors and its affinity clearly limits a predictor's accuracy. It is also known that this restriction for a scoring function constitutes an additional source of error [5]. As an alternative, non-parametric machine-learning scoring functions have been proposed. These machine learning functions, by not assuming a given structure between the complex and its affinity can capture implicit, hard to model directly relationships between them. This fact has sprung several machine-learning based scoring functions, such as the case of RF-Score [3], ID-score [33], NN-score [14], SFC-Score [53] among many others. Protein-ligand descriptors are computed for these, and examples of those are paired atom-type atom counts, one-dimensional fingerprints computed by RDKit [29], ionic interactions or hydrogen-bonds. Interestingly, it has been shown that in general, more specific/complex descriptors do not necessarily lead to lower errors [4].

4.2.2 Description of the dataset

Benchmarking protein-ligand scoring functions is fairly standardized nowadays. Several benchmarking datasets have been developed over the last years, such as the CSAR activity challenge [13] or the PDB-bind database [51], which we will use here. They generally provide an unambiguous and reproducible way to compare scoring function on exactly the same test set, extracted from the Protein Data Bank in a sensible way. In particular PDBbind (v.2015) database defines several self-contained sets for use: the general set, which contains all available affinity information for 14260 protein-ligand pairs (including Ki, Kd and IC_{50}). Out of this, the refined set composed of 3706 protein-ligand pairs is extracted according to several quality criteria: in terms of resolution (< 3Å) and experimental conditions. Finally, out of the refined set, a core set composed of 195 diverse enough protein-ligand pairs is extracted for benchmarking purposes. In general, since the core set is a subset of the refined one, researchers train on the set difference between the two, so as to avoid overfitting problems.

We will use the refined set of proteins here, since we would like to test according to the protocol defined in Section 4.1.2, that is using a $k = 5$ cross-validation scheme, and 195 pairs is not enough to get reliable enough results. We perform some filtering first, by avoiding protein-ligand pairs for which only IC_{50} kinetic information is available, since this value largely depends on experimental conditions. We treat K_i and K_d indifferently, as in common in research, for all the comparisons drawn here. The final set is comprised of $n = 3623$ structurally unique protein-ligand pairs.

Regarding descriptor computations, we recreate the ones used by RF-Score. They are simply paired atom-type counts between the protein and its ligand. The following atom types were considered for both protein and ligand:

$$\{P_j\}_{j=1}^{9} = \{C, N, O, F, P, S, Cl, Br, I\}$$
$$\{L_i\}_{i=1}^{9} = \{C, N, O, F, P, S, Cl, Br, I\}$$

And the descriptors computed can be expressed as:

$$\boldsymbol{x}\left(Z(P_j), Z(L_i)\right) = \sum_{k=1}^{K_j} \sum_{l=1}^{L_i} \Theta(d_{\text{cutoff}} - d_{kl}) \tag{4.18}$$

where d_{jk} is the distance between protein atom k of type j and ligand atom l of type i. Kj is the total number of atoms of type j from the protein and L_i the total number of atoms of type i in the ligand. Z is a function returning the atomic number of an element and Θ is a heavystep function returning 1 for distances below d_{cutoff} and 0 otherwise. This computation would result on a Cartesian product of 81 different features, 39 of which resulted in redundant zeroes across the entire dataset, so they were removed for a final set of 42 features. All molecular computations detailed here were performed using the HTMD Python package [12] for atomic manipulation. The complete dataset is available in the GitHub repository of this thesis.

4.2.3 Experiments

We detail here results for all the models and parameter ranges considered in the protocol. Figures 4.7 - 4.9 show protocol evaluations for the binding affinity dataset. It can be seen that all models benefit from the hyper-parameter optimization procedure detailed in this master's thesis, since all Bayesian Optimization strategies achieve a better mean squared error in much fewer function evaluations. Interestingly, the Bayesian Optimization approach tends to perform similarly regardless of the chosen acquisition function. In fact, for both K-nearest neighbors and the Gradient Boosting Machine, no difference can be observed between them.

Figure 4.2: SVM results for the binding affinity dataset.

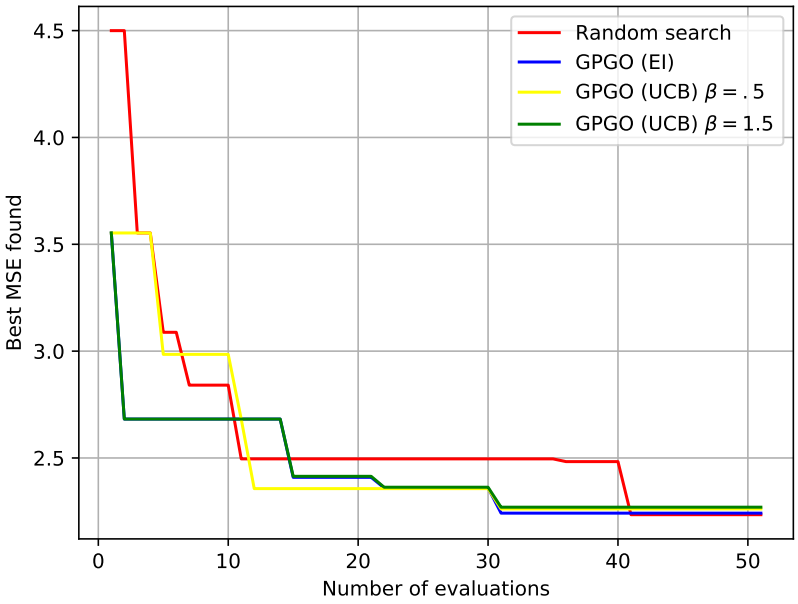


Figure 4.3: K-nearest neighbors results for the binding affinity dataset.

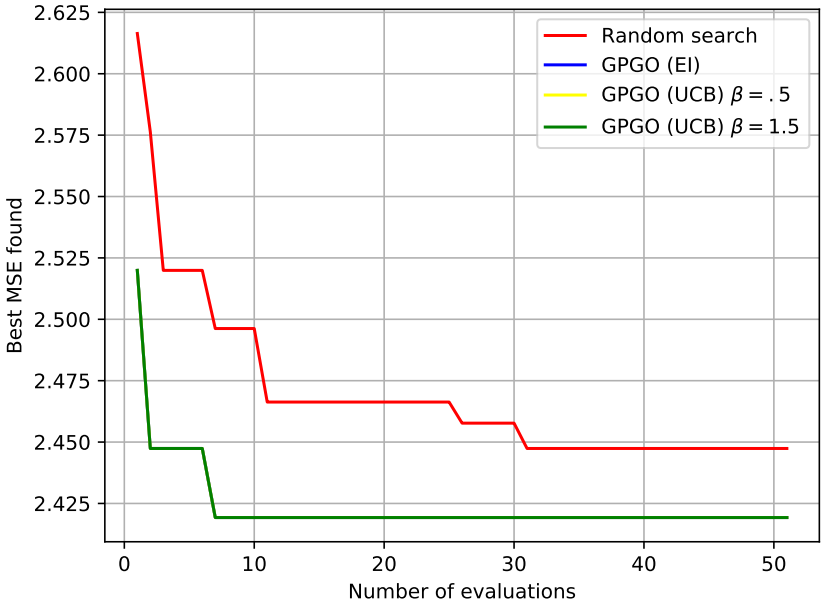


Figure 4.4: Gradient Boosting Machine results for the binding affinity dataset.

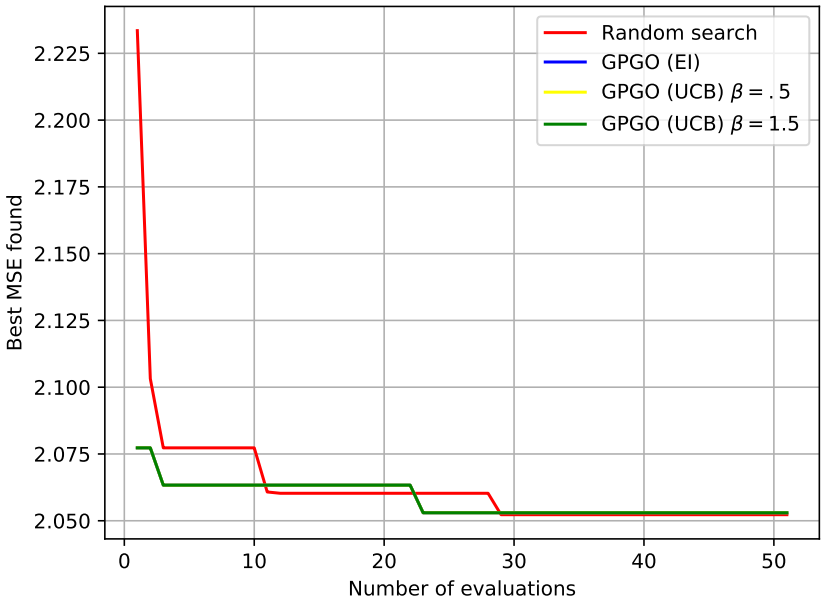


Figure 4.5: MLP results for the binding affinity dataset.

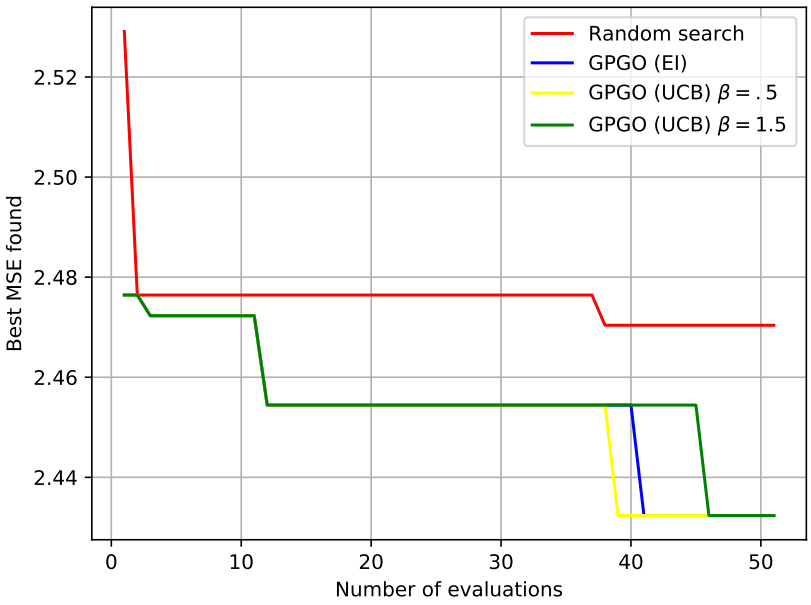
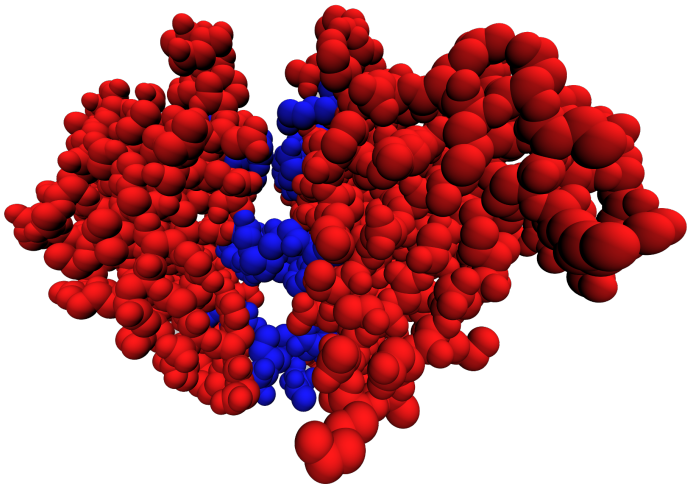


Figure 4.6: An interface between two proteins. Residues drawn in blue are defined as interficial according to heavy-atom distance.



4.3 The protein-protein interface prediction dataset

Our second round of experiments focuses on yet another problem in computational chemistry, protein-protein interface prediction. In this particular instance, we focus on determining whether a particular set of residues can be considered part of an interface with another protein. Machine-learning methods can be applied on this particular instance. We briefly describe the problem in more detail now.

4.3.1 Description of the problem

Proteins are the main catalytic agents, transporters, signal transmitters on cells. In practice, proteins do not function on their own, they interact with other molecules to carry out their role. Particularly, proteins can interact with other proteins, and for instance, alterations on the interface (see Figure 4.6) can lead to disease. Hence, over the last years, protein-protein interfaces have become targets for rational drug-design [25]. But a previous step for structural based drug design is identifying the interface itself. Many biochemical or biophysical experimental techniques are able to identify protein-protein interfaces, such as X-ray crystallography [45] or nuclear magnetic resonance (NMR) [17]. However, while these methods are extremely accurate, carrying them out in practice is expensive, labor-intensive and very time-consuming. This is the main reason why computational approaches arose, they provide inexpensive ways of predicting these interfaces.

Computational methods for predicting protein-protein interfaces can be mainly classified in three different groups: (1) data-driven or knowledge-based methods, where the quality of the predictions heavily relies on existing experimental data, either by the use of homology models [24, 52] or statistical models; (2) methods that rely on protein-protein docking [48], that is, methods using physical and geometrical properties to search putative conformations with low free energy or (3), co-evolution based methods that work with the assumption that interface residues are preserved during co-evolution, typically identifying these using multiple sequence alignments [19]. In the work presented here, we focus on (1). All methods have strength and weaknesses, and can be combined in different ways to make more reliable predictions.

Among the data-driven approaches, we can either distinguish among sequence based methods, those that require only the one-dimensional sequence of a target protein, and structural based methods, which use the full three-dimensional structure of a protein. The latter methods, while typically do not dispose as much data compared to the former, offer several advantages: rather than identifying every single residue

on a protein, they can filter their search to the ones on the surface using quantities such as the Relative Solvent Accessible Area [46].

In practice, machine-learning models are applied to structure-based methods. The problem of predicting interfaces is defined as a binary classification task, where we classify each residue on the surface as either interficial or not. Typically a neighbourhood around each residue is considered, which is typically called a *surface patch*, from which some properties $\boldsymbol{x}$ (descriptors) are extracted. Some recent structure-based machine learning methods are SPPIDER [42], PINUP [34], ProMate [41] or PIER [28]. There is also the problem of defining what an interface is, specially when benchmarking different methods: the most common one defines a residue as interficial if any heavy-atom is within k Å distance of any heavy-atom of the interacting protein [2]. Other definitions consider van der Waals surface distance [24] or delta accessible surface area change upon complexation [23].

4.3.2 Description of the dataset

For the work described here, we use the PIFACE [11] database. PIFACE is a non-redundant database extracted from the the PDB. It also provides clusters of *unique* extracted interfaces, in terms of structural similarity metrics. In total, it comprises 22604 non-redundant representative interfaces, from which a random sample of 2261 is selected for the work detailed here. While there are other more common databases used for protein-protein interface prediction, such as the protein-protein docking benchmark (DB4) [21], we choose PIFACE mostly for (1) the number of annotated interfaces is significantly larger, (2) it provides structural clusters that help prevent overfitting issues when training classification models and (3) it is very clearly organized following PDB standards. Notice that while 2261 interfaces are selected, our samples are composed of residues, and depending on the size of the interacting proteins, the number of contacting ones can result to be quite large.

Special care has to be taken with the unbalanced dataset problem [37]. Since only a few residues of a protein are considered part of the interface (positive-cases), most of the residues are marked non-interficial (negative-cases). In practice, this poses a problem since most classifiers would have an unfair tendency to classify most of the instances as the majority class, therefore rendering every conclusions useless. There are many strategies to circumvent this problem, the most simple one being balancing both classes by undersampling the majority class. There is also theoretical evidence on why this approach is preferable to others [50].

In terms of the descriptors used, we used a simplified version of the ones detailed in [22] for another problem of similar characteristics: protein binding site prediction. In summary, we treat protein structures from a computer vision perspective, as if they were 3D images. Coordinates of this 3D image are defined to span the bounding box of the protein plus a buffer of 8Å to account for pockets located close to its edges. The 3D image is then discretized into a grid of 1x1x1Å^3 sized voxels. For each of said voxels, a compendium of atomic-based pharmacophoric properties is defined. Voxel occupancies are defined with respect to the atoms in the protein depending on their excluded volume and other seven atom properties: hydrophobic, aromatic, hydrogen bond acceptor or donor, positive or negative ionizable and metallic. These are called *channels*, to draw a comparison to computer vision, where an image can be represented with three different color arrays: red, green and blue. The AutoDock 4 [40] atom types found in Table 4.5 were used with the rules of Table 4.6 to assign each atom to a specific channel. Non-protein atoms are filtered out of the calculation. Atom occupancies were calculated by taking the simplest approximation for the pair correlation function defined by

$$g(r) = \exp\left(-\beta V(r)\right), \tag{4.19}$$

where $V(r) = \epsilon(r_{\text{vdw}}/r)^{12}$ is the repulsive component of a Lennard-Jones potential and r_{vdw} is the Van der Waals atom radius. For simplicity we take the same ϵ for every atom type and such that $\beta\epsilon = 1$. The single atom occupancy estimate is therefore given by

$$n(r) = 1 - \exp\left(-(r_{\text{vdw}}/r)^{12}\right). \tag{4.20}$$

Table 4.5: AutoDock 4 atom types

Element	Description
C	Non H-bonding Aliphatic Carbon
A	Non H-bonding Aromatic Carbon
NA	Acceptor 1 H-bond Nitrogen
NS	Acceptor S Spherical Nitrogen
OA	Acceptor 2 H-bonds Oxygen
OS	Acceptor S Spherical Oxygen
SA	Acceptor 2 H-bonds Sulphur
HD	Donor 1 H-bond Hydrogen
HS	Donor S Spherical Hydrogen
MG	Non H-bonding Magnesium
ZN	Non H-bonding Zinc
MN	Non H-bonding Manganese
CA	Non H-bonding Calcium
FE	Non H-bonding Iron

Finally, the occupancy for each property of each voxel is calculated as the maximum of the contribution of all atoms belonging to that channel at its center. We provide a pseudo-code algorithm for the calculation of these descriptors in Algorithm 7. For user convenience, these descriptors are implemented and available in the HTMD [12] Python package for molecular manipulation and computing. In the work described in [22], the main model is a 3D-convolutional neural-network, since these descriptors are three dimensional (times 8 property channels) themselves. To simplify analyses and use the same models as with the other datasets, we flatten said channel descriptors to two-dimensions using the average for each channel. In practice, these represent the average prevalence of a particular property in a particular neighbourhood of the protein. The computed dataset is available in the accompanying GitHub repository of this work.

Table 4.6: Property - atom type (AutoDock 4) correspondence used for Deepsite's 3D descriptor computation

Property	Rule
Hydrophobic	atom type C or A
Aromatic	atom type A
Hydrogen bond acceptor	atom type NA or NS or OA or OS or SA
Hydrogen bond donor	atom type HD or HS with O or N partner
Positive ionizable	atom with positive charge
Negative ionizable	atom with negative charge
Metal	atom type MG or ZN or MN or CA or FE
Excluded volume	all atom types

4.3.3 Experiments

We repeat the same experiments as with the previous dataset. In all cases, Bayesian Optimization routines achieve better function evaluations in significantly less iterations than random search. Of particular interest is again how robust these methods are regardless of the acquisition function choice. Given the evidence presented here, and the size of some datasets present in computational chemistry problems (as well as in other areas), I believe Bayesian Optimization should be the default strategy to optimize hyperparameters of machine-learning models.

Algorithm 7 Descriptor pseudo-code computation.

function OCCUPANCY(atomCoords, centerCoords, radii, channels)
  for each atom A in system **do**
    $\boldsymbol{a} \leftarrow \text{atomCoords}_A$
    $\boldsymbol{h} \leftarrow \text{channels}_A$
    $\boldsymbol{r}_{\text{vdw}} \leftarrow \text{radii}_A$
    for each center $\boldsymbol{c}$ in centerCoords **do**
      $r \leftarrow L_2\text{Dist}(\boldsymbol{c}, \boldsymbol{a})$
      $x \leftarrow \frac{r_{\text{vdw}}}{r}$
      $n \leftarrow 1 - \exp(-x^{12})$
      for each channel p in $\boldsymbol{h}$ **do**
        $O_{\boldsymbol{c},p} \leftarrow \max\{n, O_{\boldsymbol{c},p}\}$
      end for
    end for
  end for
end function

Figure 4.7: SVM results for the protein interface dataset.

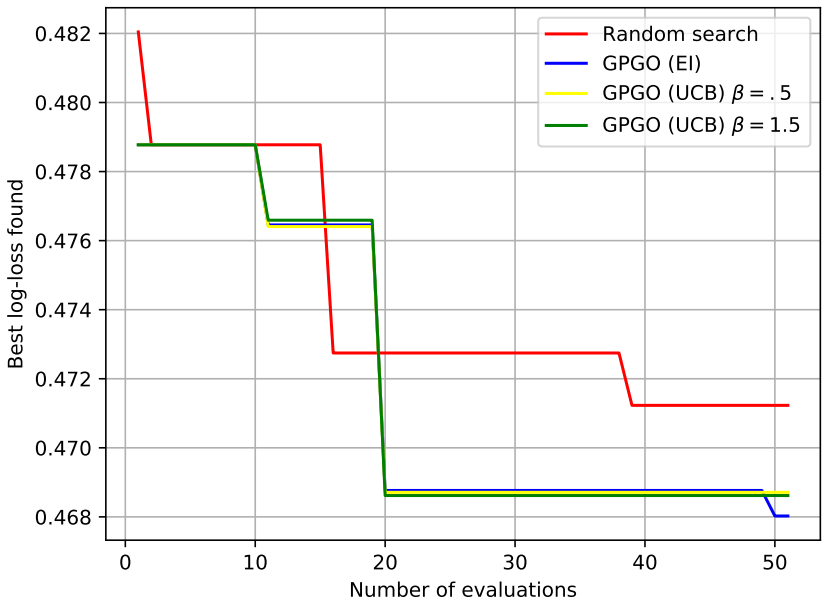


Figure 4.8: K-nearest neighbors results for the protein interface dataset.

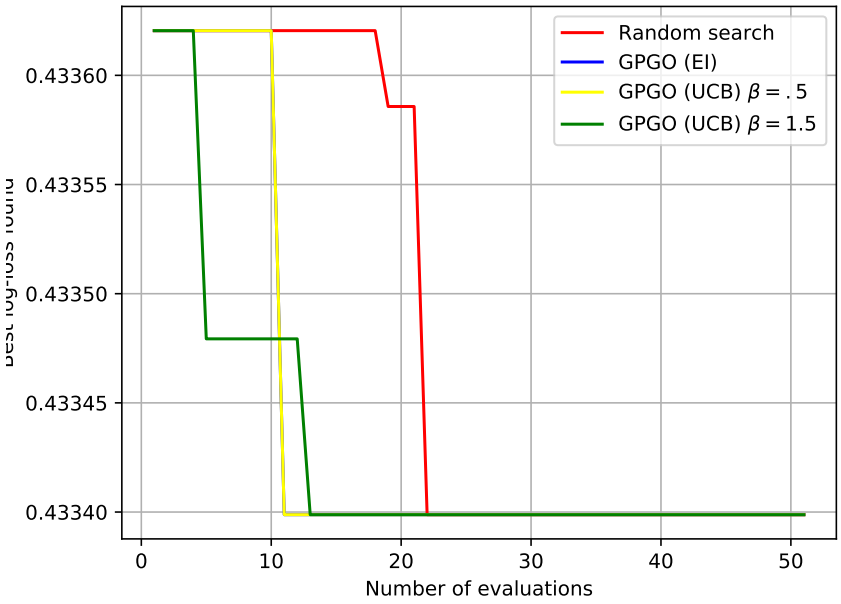


Figure 4.9: Gradient Boosting Machine results for the protein interface dataset.

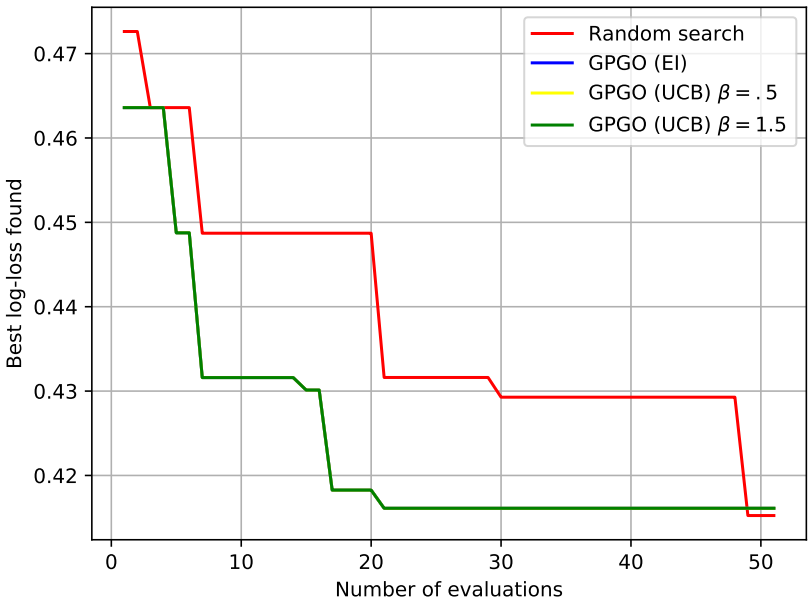
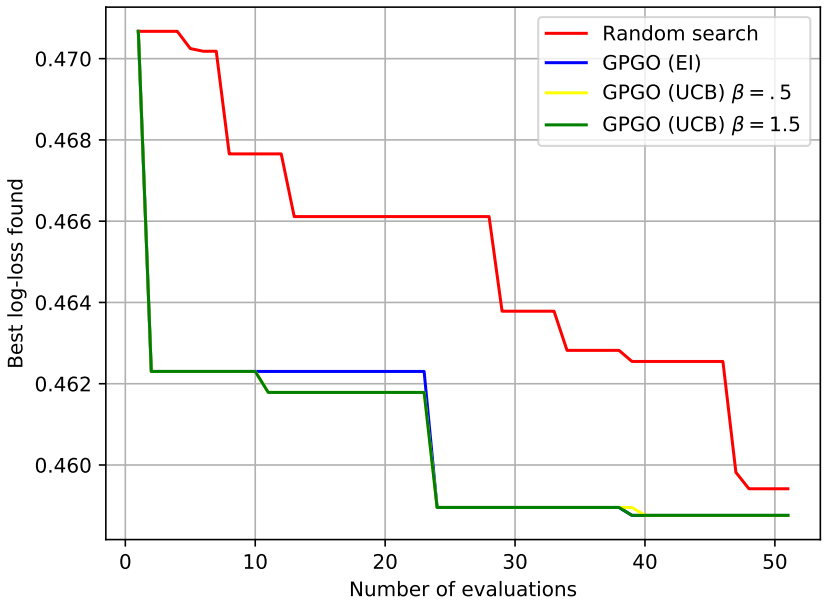


Figure 4.10: MLP results for the protein interface dataset.



4.4 Other datasets

In this section, we detail more results for the same models considered throughout the chapter. We aim to further demonstrate with other examples that Bayesian Optimization is a good default strategy for optimizing machine-learning hyperparameters. The datasets shown here span areas different from the ones considered in the more detailed previous examples.

4.4.1 The breast cancer dataset

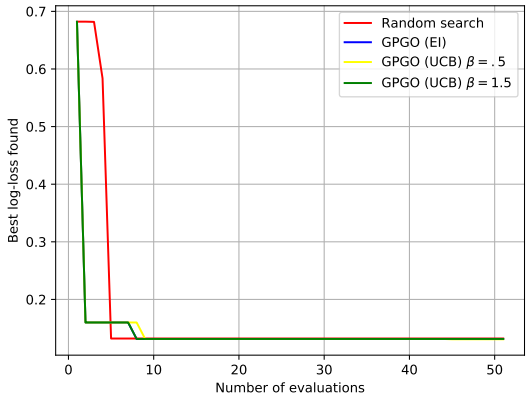
This breast cancer data comes from a study by the University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia in 1986. It was originally used in [38]. It is available as open data in the UCI Machine Learning repository [35]. A small descriptions of the predictors used for this binary classification problem can be found in Table 4.7. Results can be found on Figure 4.12.

Table 4.7: Description of the breast cancer dataset

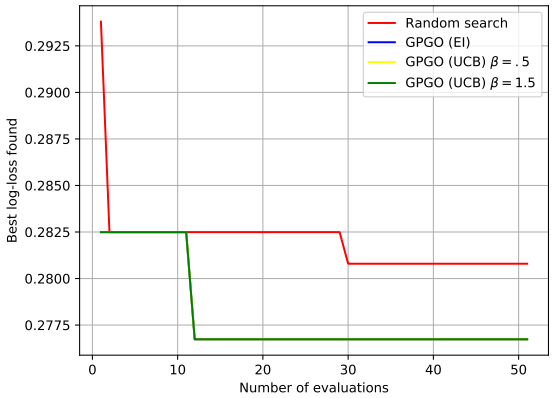
Variable	**Values**
Target	no-recurrence-events, recurrence-events
age	10-19, 20-29, . . . , 90-99.
menopause	less than 40, greater or equal 40, pre-menopausic
tumor size	0-4, 5-9, . . . , 55-59
invasive nodes	0-2, 3-5, . . . , 36-39
node-caps	yes, no
degree malignity	1, 2, 3
breast	left, right
location	left-up, left-low, right-up, right-low, central
irradiation	yes, no

Figure 4.11: Benchmarking results for the breast cancer dataset.

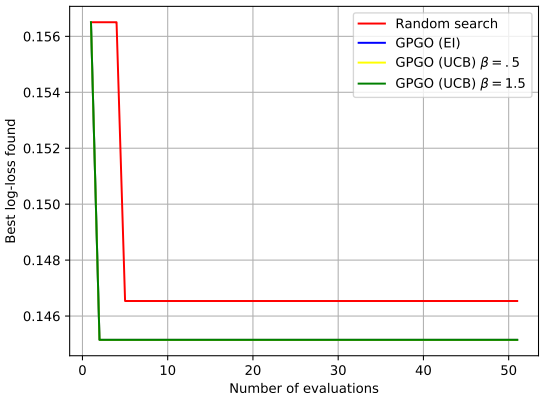
(a) SVM



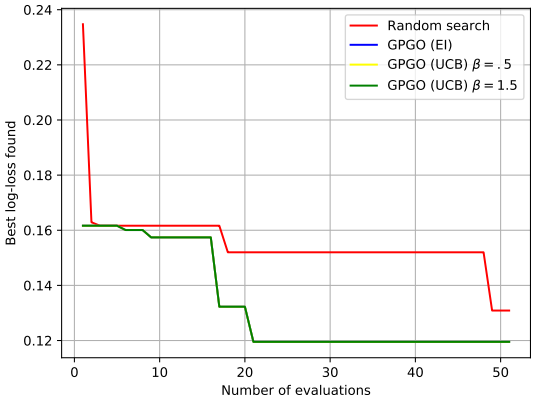
(b) KNN



(c) MLP



(d) GBM



4.4.2 The LSVT voice rehabilitation dataset

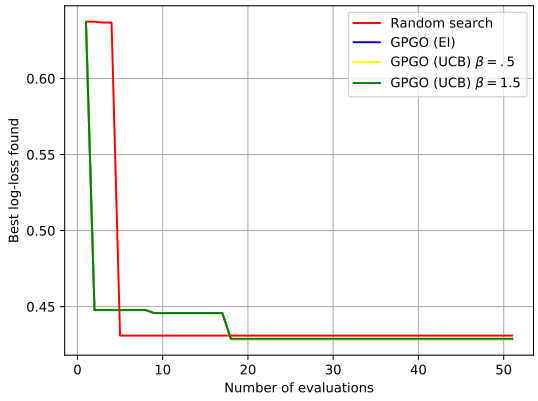
Dataset originally used in [47], in order to demonstrate that it was possible to replicate expert assessment in Parkinson's disease speech rehabilitation using machine-learning techniques. Also available in the UCI machine-repository. Each attribute (feature) corresponds to the application of a speech signal processing algorithm which aims to characterise objectively the signal. These algorithms include standard perturbation analysis methods, wavelet-based features, fundamental frequency-based features, and tools used to mine non-linear time-series. We refer the reader to the original paper for more information on the features.

4.4.3 The Parkinson's disease dataset

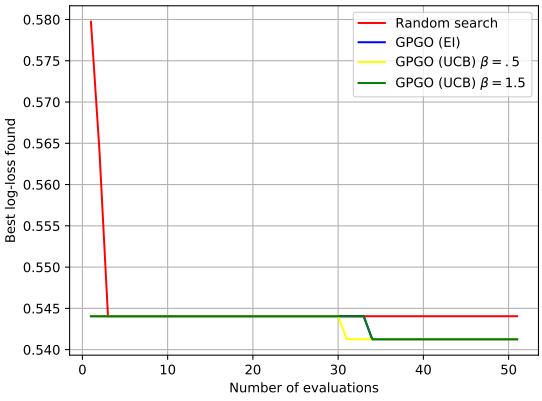
The dataset was created by Max Little of the University of Oxford, in collaboration with the National Centre for Voice and Speech, Denver, Colorado, who recorded the speech signals. The original study [36] published the feature extraction methods for general voice disorders. This dataset is composed of a range of biomedical voice measurements from 31 people, 23 with Parkinson's disease. Each feature in the table is a particular voice measure, and each row corresponds one of 195 voice recording from these individuals. The main aim of the data is to discriminate healthy people from those with the disease. Features are described in Table 4.8. Results can be checked in Figure 4.13.

Figure 4.12: Benchmarking results for the LSVT Voice Rehabilitation dataset.

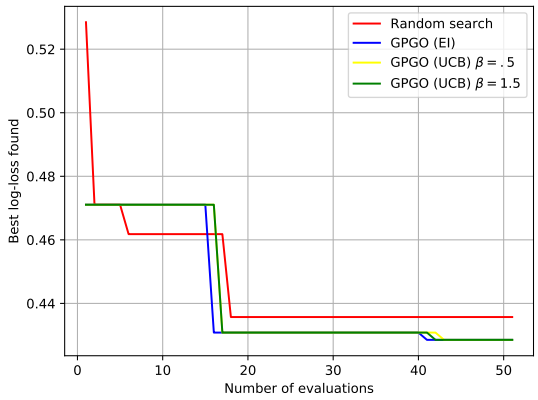
(a) SVM



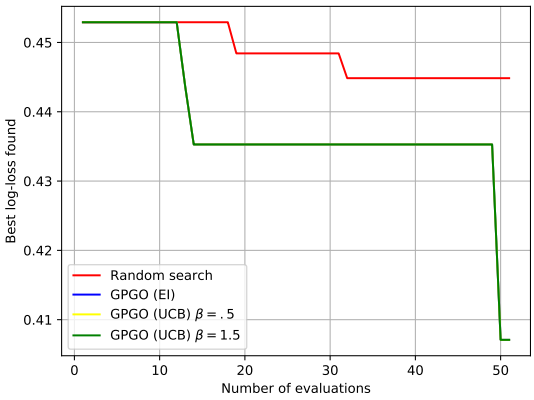
(b) KNN



(c) MLP



(d) GBM



4.5 Discussion

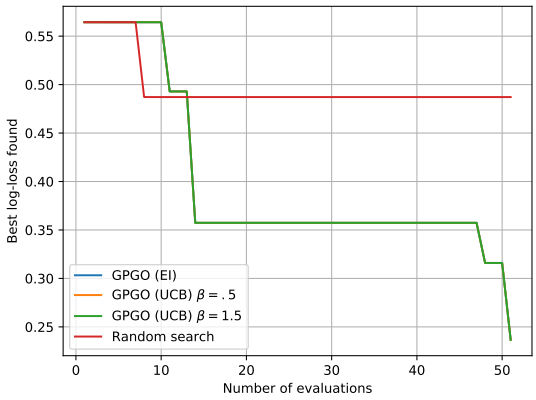
In these experiments, we have shown that Bayesian Optimization is substantially more efficient than random search when optimizing the parameters of a machine-learning model. This can be seen by the fact that we have undergone the same evaluation over datasets and machine-learning models of diverse enough nature. However, random search is embarrassingly parallel over threads, and therefore can potentially outperform the techniques discussed here when given extra computational power. Further work needs to be done in parallelizing Bayesian Optimization strategies. This means sharing evaluation information over threads, so that in each iteration we can fit the most precise surrogate model possible.

Table 4.8: Features used in the Parkinson's disease dataset

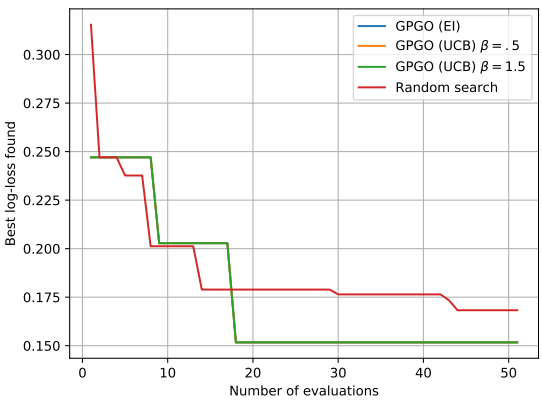
Variable	Values
Target	Health status of the subject
MDVP:Fo (Hz)	Average vocal fundamental frequency
MDVP:Fhi (Hz)	Maximum vocal fundamental frequency
MDVP:Flo (Hz)	Minimum vocal fundamental frequency
MDVP:Jitter (%), MDVP:Jitter (Abs), MDVP:RAP, MDVP:PPQ, Jitter:DDP	Several measures of variation in fundamental frequency
MDVP:Shimmer, MDVP:Shimmer (dB), Shimmer:APQ3, Shimmer	Several measures of variation in amplitude
NHR, HNR	Two measures of ratio of noise to tonal components in the voice
RPDE, D2	Two nonlinear dynamical complexity measures
DFA	Signal fractal scaling exponent
spread1, spread2, PPE	Three nonlinear measures of fundamental frequency variation

Figure 4.13: Benchmarking results for the Parkinson's disease dataset.

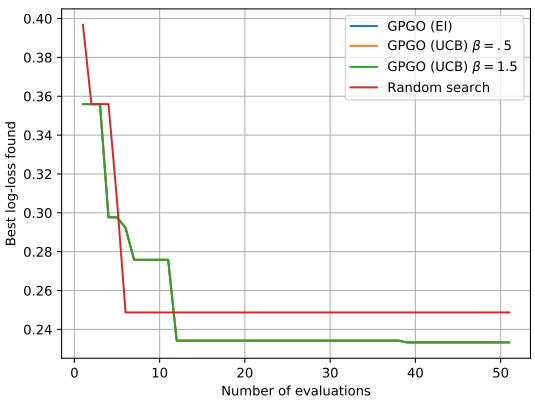
(a) SVM



(b) MLP



(c) GBM



Bibliography

[1] Hervé Abdi. "Partial Least Squares (PLS) Regression". In: *Encyclopedia for research methods for the social sciences*. 2003, pp. 792–795. ISBN: 9781412950589. DOI: http://dx.doi.org/10.4135/9781412950589.n690.

[2] Fayyaz ul Amir Afsar Minhas, Brian J. Geiss, and Asa Ben-Hur. "PAIRpred: Partner-specific prediction of interacting residues from sequence and structure". In: *Proteins: Structure, Function and Bioinformatics* 82.7 (2014), pp. 1142–1155. ISSN: 10970134. DOI: 10.1002/prot.24479.

[3] Pedro J. Ballester and John B O Mitchell. "A machine learning approach to predicting protein-ligand binding affinity with applications to molecular docking". In: *Bioinformatics* 26.9 (2010), pp. 1169–1175. ISSN: 13674803. DOI: 10.1093/bioinformatics/btq112. arXiv: 0-387-31073-8.

[4] Pedro J. Ballester, Adrian Schreyer, and Tom L. Blundell. "Does a more precise chemical description of protein-ligand complexes lead to more accurate prediction of binding affinity?" In: *Journal of Chemical Information and Modeling* 54.3 (2014), pp. 944–955. ISSN: 15205142. DOI: 10.1021/ci500091r.

[5] Bernhard Baum et al. "Non-additivity of functional group contributions in protein-ligand binding: A comprehensive study by crystallography and isothermal titration calorimetry". In: *Journal of Molecular Biology* 397.4 (2010), pp. 1042–1054. ISSN: 00222836. DOI: 10.1016/j.jmb.2010.02.007.

[6] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. ISSN: 1532-4435.

[7] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Vol. 4. 4. 2006, p. 738. ISBN: 9780387310732. DOI: 10.1117/1.2819119. arXiv: 0-387-31073-8. URL: http://www.library.wisc.edu/selectedtocs/bg0137.pdf.

[8] Leon Bottou. "Stochastic Learning". In: *Learning* (2003), p. 22. ISSN: 00335533. DOI: 10.1007/978-3-540-28650-9_7. URL: http://www.cs.nyu.edu/courses/fall10/G22.2965-001/stocgraddescent.pdf.

[9] Chih-chung Chang and Chih-jen Lin. "LIBSVM : A Library for Support Vector Machines". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2 (2011), pp. 1–39. ISSN: 21576904. DOI: 10.1145/1961189.1961199. arXiv: 0-387-31073-8.

[10] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (1995), pp. 273–297. ISSN: 15730565. DOI: 10.1023/A:1022627411411. arXiv: arXiv:1011.1669v3.

[11] Engin Cukuroglu et al. "Non-redundant unique interface structures as templates for modeling protein interactions". In: *PLoS ONE* 9.1 (2014). ISSN: 19326203. DOI: 10.1371/journal.pone.0086738.

[12] S. Doerr et al. "HTMD: High-Throughput Molecular Dynamics for Molecular Discovery". In: *Journal of Chemical Theory and Computation* 12.4 (2016), pp. 1845–1852. ISSN: 15499626. DOI: 10.1021/acs.jctc.6b00049.

[13] James B. Dunbar et al. "CSAR data set release 2012: Ligands, affinities, complexes, and docking decoys". In: *Journal of Chemical Information and Modeling* 53.8 (2013), pp. 1842–1852. ISSN: 15499596. DOI: 10.1021/ci4000486.

[14] Jacob D. Durrant and J. Andrew McCammon. "NNScore 2.0: A neural-network receptor-ligand scoring function". In: *Journal of Chemical Information and Modeling* 51.11 (2011), pp. 2897–2903. ISSN: 15499596. DOI: 10.1021/ci2003889.

[15] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine". In: *Annals of Statistics* 29.5 (2001), pp. 1189–1232. ISSN: 00905364. DOI: DOI10.1214/aos/1013203451. arXiv: arXiv:1011.1669v3.

[16] Richard A. Friesner et al. "Glide: A New Approach for Rapid, Accurate Docking and Scoring. 1. Method and Assessment of Docking Accuracy". In: *Journal of Medicinal Chemistry* 47.7 (2004), pp. 1739–1749. ISSN: 00222623. DOI: 10.1021/jm0306430. arXiv: arXiv:1011.1669v3.

[17] Christoph Gobl et al. "NMR approaches for structural analysis of multidomain proteins and complexes in solution". In: *Progress in Nuclear Magnetic Resonance Spectroscopy* 80 (2014), pp. 26–63. ISSN: 00796565. DOI: 10.1016/j.pnmrs.2014.05.003.

[18] H Gohlke, M Hendlich, and G Klebe. "Knowledge-based scoring function to predict protein-ligand interactions". In: *Journal of Molecular Biology* 295.2 (2000), pp. 337–356. ISSN: 0022-2836. DOI: 10.1006/jmbi.1999.3371\rS0022-2836(99)93371-5[pii].

[19] Thomas A. Hopf et al. "Sequence co-evolution gives 3D contacts and structures of protein complexes". In: *eLife* 3 (2014). ISSN: 2050084X. DOI: 10.7554/eLife.03430. arXiv: 1405.0929.

[20] Niu Huang et al. "Molecular mechanics methods for predicting protein-ligand binding." In: *Physical chemistry chemical physics : PCCP* 8.44 (2006), pp. 5166–5177. ISSN: 1463-9076. DOI: 10.1039/b608269f.

[21] Howook Hwang et al. "Protein-protein docking benchmark version 4.0". In: *Proteins: Structure, Function and Bioinformatics* 78.15 (2010), pp. 3111–3114. ISSN: 08873585. DOI: 10.1002/prot.22830. arXiv: NIHMS150003.

[22] José Jiménez et al. "DeepSite: protein binding site predictor based on 3D-convolutional neural networks". In: *[submitted]* (2017).

[23] Susan Jones and Janet M. Thornton. "Analysis of protein-protein interaction sites using surface patches." In: *Journal of Molecular Biology* 272.1 (1997), pp. 121–132. ISSN: 00222836. DOI: 10.1006/jmbi.1997.1234. URL: http://www.ncbi.nlm.nih.gov/pubmed/9299342$%5Cbackslash$nhttp://linkinghub.elsevier.com/retrieve/pii/S0022283697912341.

[24] Rafael a Jordan et al. "Predicting protein-protein interface residues using local surface structural similarity". In: *BMC Bioinformatics* 13.1 (2012), p. 41. ISSN: 1471-2105. DOI: 10.1186/1471-2105-13-41. URL: http://www.biomedcentral.com/1471-2105/13/41.

[25] Harry Jubb, Tom L. Blundell, and David B. Ascher. *Flexibility and small pockets at protein-protein interfaces: New insights into druggability*. 2015. DOI: 10.1016/j.pbiomolbio.2015.01.009.

[26] Matthew B. Kennel. "KDTREE 2: Fortran 95 and C++ software to efficiently search for near neighbors in a multi-dimensional Euclidean space". In: *arXiv preprint arXiv:0408067* (2004). arXiv: 0408067 [physics]. URL: http://arxiv.org/abs/physics/0408067.

[27] André Krammer et al. "LigScore: A novel scoring function for predicting binding affinities". In: *Journal of Molecular Graphics and Modelling* 23.5 (2005), pp. 395–407. ISSN: 10933263. DOI: 10.1016/j.jmgm.2004.11.007.

[28] Irina Kufareva et al. "PIER: Protein interface recognition for structural proteomics". In: *Proteins: Structure, Function and Genetics* 67.2 (2007), pp. 400–417. ISSN: 08873585. DOI: 10.1002/prot.21233. arXiv: 0605018 [q-bio].

[29] Paul Labute. "A widely applicable set of descriptors". In: *Journal of Molecular Graphics and Modelling* 18.4-5 (2000), pp. 464–477. ISSN: 10933263. DOI: 10.1016/S1093-3263(00)00068-1.

[30] Andrew R. Leach, Brian K. Shoichet, and Catherine E. Peishoff. *Prediction of protein-ligand interactions. Docking and scoring: Successes and gaps*. 2006. DOI: 10.1021/jm060999m.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539. arXiv: arXiv:1312.6184v5. URL: http://dx.doi.org/10.1038/nature14539.

[32] Roger J Lewis, D Ph, and West Carson Street. "An Introduction to Classification and Regression Tree (CART) Analysis". In: *2000 Annual Meeting of the Society for Academic Emergency Medicine* 310 (2000), 14p. DOI: 10.1.1.95.4103. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.4103%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[33] Guo Bo Li et al. "ID-score: A new empirical scoring function based on a comprehensive set of descriptors related to protein-ligand interactions". In: *Journal of Chemical Information and Modeling* 53.3 (2013), pp. 592–600. ISSN: 15499596. DOI: 10.1021/ci300493w.

[34] Shide Liang et al. "Protein binding site prediction using an empirical scoring function". In: *Nucleic Acids Research* 34.13 (2006), pp. 3698–3707. ISSN: 03051048. DOI: 10.1093/nar/gkl454.

[35] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: http://archive.ics.uci.edu/ml.

[36] Max A Little et al. "Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection." In: *Biomedical engineering online* 6 (2007), p. 23. ISSN: 1475-925X. DOI: 10.1186/1475-925X-6-23. arXiv: 0707.0086.

[37] Rushi Longadge, S Snehlata Dongre, and Latesh Malik. "Class imbalance problem in data mining: review". In: *International Journal of Computer Science and Network* 2.1 (2013), pp. 83–87. ISSN: 2277-5420. DOI: 10.1109/SIU.2013.6531574. arXiv: 1305.1707.

[38] Ryszard S. Michalski, I. Mozetic, and J. Hong. "The AQ15 Inductive Learning System: an Overview and Experiments". In: *Proceedings of IMAL 1986*. 1986, p. 36.

[39] W. T M Mooij and Marcel L. Verdonk. "General and targeted statistical potentials for protein-ligand interactions". In: *Proteins: Structure, Function and Genetics* 61.2 (2005), pp. 272–287. ISSN: 08873585. DOI: 10.1002/prot.20588.

[40] Garrett M. Morris et al. "Software news and updates AutoDock4 and AutoDockTools4: Automated docking with selective receptor flexibility". In: *Journal of Computational Chemistry* 30.16 (2009), pp. 2785–2791. ISSN: 01928651. DOI: 10.1002/jcc.21256. arXiv: NIHMS150003.

[41] Hani Neuvirth, Ran Raz, and Gideon Schreiber. "ProMate: A structure based prediction program to identify the location of protein-protein binding sites". In: *Journal of Molecular Biology* 338.1 (2004), pp. 181–199. ISSN: 00222836. DOI: 10.1016/j.jmb.2004.02.040.

[42] Aleksey Porollo and Jarosław Meller. "Prediction-based fingerprints of protein-protein interactions". In: *Proteins: Structure, Function and Genetics*. Vol. 66. 3. 2007, pp. 630–645. ISBN: 0887-3585. DOI: 10.1002/prot.21248. arXiv: 0605018 [q-bio].

[43] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 0028-0836. DOI: 10.1038/323533a0. arXiv: arXiv:1011.1669v3.

[44] Robert E. Schapire. "A brief introduction to boosting". In: *IJCAI International Joint Conference on Artificial Intelligence*. Vol. 2. 1999, pp. 1401–1406. ISBN: 3540440119. DOI: citeulike-article-id:765005. arXiv: arXiv:1508.01136v1.

[45] Yigong Shi. *A glimpse of structural biology through X-ray crystallography*. 2014. DOI: 10.1016/j.cell.2014.10.051.

[46] Matthew Z. Tien et al. "Maximum allowed solvent accessibilites of residues in proteins". In: *PLoS ONE* 8.11 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0080635. arXiv: arXiv:1211.4251v3.

[47] Athanasios Tsanas et al. "Objective automatic assessment of rehabilitative speech treatment in Parkinson's disease". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 22.1 (2014), pp. 181–190. ISSN: 15344320. DOI: 10.1109/TNSRE.2013.2293575.

[48] Ilya A. Vakser. *Protein-protein docking: From interaction to interactome*. 2014. DOI: 10.1016/j.bpj.2014.08.033.

[49] A. Verikas, A. Gelzinis, and M. Bacauskiene. "Mining data with random forests: A survey and results of new tests". In: *Pattern Recognition* 44.2 (2011), pp. 330–349. ISSN: 00313203. DOI: 10.1016/j.patcog.2010.08.011.

[50] Byron C. Wallace et al. “Class imbalance, redux”. In: *Proceedings - IEEE International Conference on Data Mining, ICDM*. 2011, pp. 754–763. ISBN: 9780769544083. DOI: 10.1109/ICDM.2011.33.

[51] Renxiao Wang et al. “The PDBbind database: Collection of binding affinities for protein-ligand complexes with known three-dimensional structures”. In: *Journal of Medicinal Chemistry* 47.12 (2004), pp. 2977–2980. ISSN: 00222623. DOI: 10.1021/jm0305801.

[52] Li C Xue, Drena Dobbs, and Vasant Honavar. “HomPPI: a class of sequence homology based protein-protein interface prediction methods.” In: *BMC bioinformatics* 12 (2011), p. 244. ISSN: 1471-2105. DOI: 10.1186/1471-2105-12-244.

[53] David Zilian and Christoph a Sotri. “SFCscoreRF: A Random Forest-Based Scoring Function for Improved A ffi nity Prediction of Protein - Ligand Complexes”. In: *Journal of chemical information and modeling* 53 (2013), pp. 1923–1933. ISSN: 1549-960X. DOI: 10.1021/ci400120b.

Chapter 5

pyGPGO: Bayesian Optimization for Python

In this chapter, we will explain the functionalities behind pyGPGO, the Bayesian Optimization software developed alongside this Master's thesis. pyGPGO aims to be minimalistic, easy to understand, modular, complete in functionality and up-to-date with latest research. pyGPGO is under the MIT License, which means it can be used for both academic and commercial purposes, while providing no warranty for the user. pyGPGO is completely open-source, users can contribute to the package in any way the see fit, either by reporting bugs, filling out missing documentation, providing new functionality or improve the existing one.

All of pyGPGO's code can be accessed through its associated GitHub repository, in `https://github.com/hawk31/pyGPGO`. Extensive documentation for all the code in the package is available in both HTML and PDF format at ReadTheDocs `http://readthedocs.org/projects/pygpgo/`. For cleanness I have decided not to include neither in them in the present document.

In summary, pyGPGO is a Python package to perform Bayesian Optimization with minimal effort from the user. In this final chapter we start by providing an overview of the functionality provided, this is by no means exhaustive and users are encouraged to check the documentation for in-detail explanations. We then explain the logic behind the implemented modules: this is done to provide some basic guidance for the user on where to look for intended functionality. Installation details are described next.

Several real-world examples using the present software are then discussed, this section aims to help users understand the logic behind the software in a practical way. Different examples from the ones considered here can be found either in the Appendix of this thesis or in the GitHub repository of the package. Finally, we compare pyGPGO with other existing software and describe several future goals and improvements for the package.

5.1 Features

The idea behind pyGPGO stems from the fact that there are many possible choices in the Bayesian Optimization framework, to name a few:

- Choice of surrogate model.
- Covariance function to use
- Acquisition behaviour
- Hyperparameter treatment
- ...

In general, these many choices motivate a modular design, in an object-oriented way, so that users can experiment with different architectural choices. Most of the available software focuses on a particular implementation of the Bayesian optimization algorithm. pyGPGO on the other hand is completely modular and provides extensive functionality. In particular, my software features:

- A completely modular and customizable design. Easy to setup and minimal dependencies.
- A wide range of surrogate models: Gaussian Processes, Student-t Processes, Random Forests, Extra Random Forests and Gradient Boosting Machines.
- Most of the usual covariance functions, as well as its derivatives: squared exponential, Matérn, γ-exponential, rational quadratic, exponential sine, and dot product.
- Many acquisition function behaviours: probability of improvement, expected improvement, upper confidence bound and entropy-based, as well as their integrated versions.
- Type II Maximum-Likelihood of covariance function hyperparameters.
- MCMC sampling for full-Bayesian treatment of hyperparameters (via `pyMC3`).

To the best of my knowledge, pyGPGO is one of the most complete and simple packages for Bayesian Optimization implemented.

5.2 Package logistics

All functionality is divided in different *modules*, each performing a very specific task. We briefly describe these here. Again, the user is encourage to check the provided documentation for more detailed explanations on how to use the package.

- The `covfunc` module contains all the code related to covariance function calculations.
- The `surrogates` module implements classes corresponding to different surrogate models
- The `Acquisition` module lets the user specify different acquisition function strategies.
- The `GPGO` module implements the main Bayesian Optimization procedure.

Apart from this functionality, there is plenty of additional material in the GitHub repository of this package:

- A folder named `examples`, with all the coding examples laid down throughout the course of this master's thesis.
- Another folder named `testing` with all the code used regarding benchmarking and testing of the datasets used in Chapter 4 of this work.
- Yet another folder named `datasets` with all the datasets tested in Chapter 4, in .csv format.
- Finally, a folder named `mthesis_text`, containing all the text work presented here.

5.3 Installation

pyGPGO comes in the form of a Python package. Only Python versions equal or higher than 3.5 are currently supported. In principle, pyGPGO should work with Windows, OSX and Linux, though only the latter has been tested. Most Unix based systems already come with Python distribution installed. To check whether it on a bash/cmd terminal:

```
1 python --version
```

If Python is not installed on the system, we highly recommend installing the Python distribution Anaconda [1]. Along with the distribution, they also provide most common packages for numeric/scientific operations. In particular, should the user choose to install the Anaconda distribution all dependencies needed by pyGPGO are covered. After downloading the executable corresponding to your particular system configuration from their web page, it suffices to do (for UNIX-based systems):

```
bash Anaconda3-x.x.x-Linux-x86_64.sh
```

Follow the instructions to install it on a local path in your system. In particular, the Anaconda installer will ask if the user wants to prepend to the system `PATH` the route to the Python binaries. Windows users have a graphical installer available in the Anaconda website.

If the user has a working Python environment in the system and does not want to use the one provided by Anaconda, some dependencies must be fulfilled in order for pyGPGO to work. In particular, both `numpy` and `joblib` need to be installed. Typically, one should use the packages available in the Python Package Index (PyPI) and install them using the Python Package manager `pip`. From a bash terminal:

```
pip install --upgrade numpy joblib
```

It is recommended to install pyGPGO's development version, which can be retrieved from the associated GitHub repository:

```
pip install git+https://github.com/hawk31/pyGPGO
```

Optionally, for MCMC inference of hyperparameters, we depend on pyMC3, whose installation is currently a bit more challenging (as of Apr 2017). It should suffice to do:

```
git clone https://github.com/pymc-devs/pymc3.git
cd pymc3/
python setup.py install
```

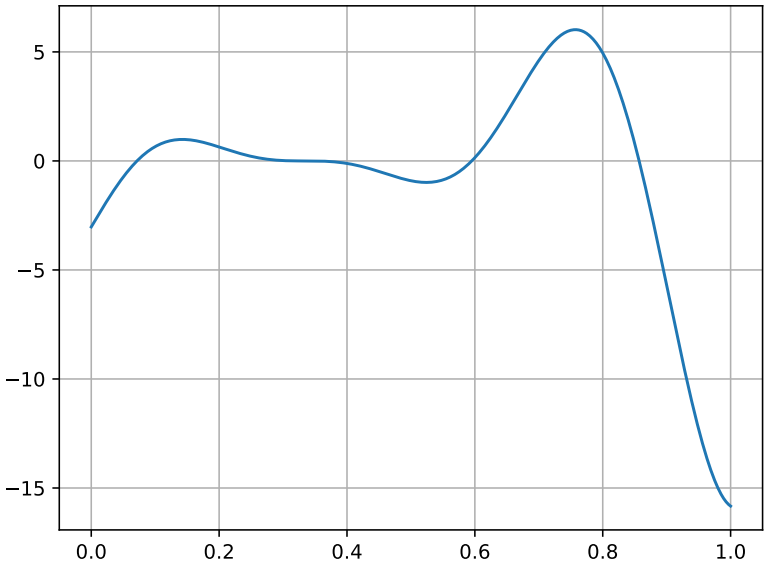
5.3.1 A minimal example

Here we go through a minimal example to show how to use pyGPGO in its most simple form. We comment line by line using the IPython console.

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: from pyGPGO.covfunc import squaredExponential
   ...: from pyGPGO.surrogates.GaussianProcess import GaussianProcess
   ...: from pyGPGO.acquisition import Acquisition
   ...: from pyGPGO.GPGO import GPGO
```

After loading `numpy` and `matplotlib`, we start loading all the needed modules for our example: we will use the `squaredExponential` covariance function, a `GaussianProcess` regressor, an `Acquisition` function and `GPGO`, the class for Bayesian Optimization. We fix a random seed, and define the function we are about to optimize, a plot of which is available in Figure 5.1.

Figure 5.1: Example function for optimization with pyGPGO.




```
In [2]:     np.random.seed(20)
   ...:     def f(x):
   ...:         return -((6*x-2)**2*np.sin(12*x-4))
```

We now instantiate our covariance function, our regressor and our acquisition:

```
In [3]:     sexp = squaredExponential()
   ...:     gp = GaussianProcess(sexp)
   ...:     acq = Acquisition(mode = 'ExpectedImprovement')
```

We define our the parameters to optimize over now, as a dictionary. Note that function `f` takes as parameter `x`. It is a continuous variable, where $x \in [0, 1]$. If we had other variables, we simply add them to the dictionary with its type and bounds.

```
In [4]:     params = {'x': ('cont', (0, 1))}
```

We now instantiate our `GPGO` class, passing all previous created objects:

```
In [5]:         gpgo = GPGO(gp, acq, f, params)
```

We finally optimize for a number of iterations:

```
In [6]: gpgo.run(max_iter = 10)
```

Check the result just by calling:

```
In [7]: gpgo.getResult()
Out [8]:         (OrderedDict([('x', 0.76321944301549549)]), 6.0013872547078249)
```

After 10 iterations, the best value for `x` our optimizer has found is 0.7632, with a function value of 6.001.

5.4 Examples

In this section, we will detail several examples on how to use pyGPGO for real world tasks. They will provide extra details on how much functionality the package exposes to the end user and may serve as a blueprint for other tasks.

5.4.1 Gaussian Process regression using the `GaussianProcess` module.

While pyGPGO is mainly a Bayesian Optimization package, it also exposes a very competent class for performing Gaussian Process regression. Here we provide a simple example on how to perform Gaussian Process Regression on noisy synthetic data. The script below produces Figure 5.3.

```
import numpy as np
import matplotlib.pyplot as plt
from pyGPGO.GPGO import GPGO
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential


if __name__ == '__main__':
    rng = np.random.RandomState(0)
    X = rng.uniform(0, 5, 20)[:, np.newaxis]
    y = 0.5 * np.sin(3 * X[:, 0]) + rng.normal(0, 0.5, X.shape[0])

    sexp = squaredExponential()
    gp = GPRegressor(sexp, optimize = True, usegrads = True)
    gp.fit(X, y)

    X_ = np.linspace(0, 5, 100)
    y_mean, y_var = gp.predict(X_[:, np.newaxis], return_std=True)
    y_std = np.sqrt(y_var)
    plt.plot(X_, y_mean, 'k', lw=2, zorder=9, label = 'Posterior mean')
    plt.fill_between(X_, y_mean - 1.64 * y_std,
                     y_mean + 1.64 * y_std,
                     alpha=0.4, color='blue')
    plt.plot(X_, 0.5*np.sin(3*X_), 'r', lw=2, zorder=9, label = 'Original function')
    plt.scatter(X[:, 0], y, c='r', s=50, zorder=10)
    plt.legend(loc = 0)
    params = gp.getcovparams()
    plt.title('Optimal params | \$l\$={}, \$\sigma_n^2\$
    ={},\sigma_f^2={}'.format(np.round(params['l'],3),
              np.round(params['sigmaf'], 3), np.round(params['sigman'], 3)))
    plt.tight_layout()
    plt.show()
```

A few notes on the code, notice that we add two extra parameters to the `surrogates.GaussianProcess` module: `optimize=True, usegrads=True` indicates to the instance that it should perform Type II maximum likelihood estimation of the `squaredExponential` instance hyper-parameters using gradient information if available. By default all hyper-parameters are optimized, in this case, $\left\{l, \sigma_n^2, \sigma_f^2\right\}$ are optimized. If we want only a subset of them to be optimized or none at all, we can specify so in the corresponding covariance function instance.

5.4.2 MCMC inference over hyperparameters using the `GaussianProcessMCMC` module

The presented previous approach optimizes the marginal likelihood, in an empirical Bayes fashion in order to estimate hyperparemeters. As an alternative, pyGPGO also supports MCMC inference over hyperparameters via pyMC3, another Python package for Bayesian Inference. At the moment, both the NUTS [3] (No-U-Turn Sampler), a Hamiltonian Monte Carlo approach is available, as well as Variational Inference using ADVI [4] (Automatic Differentiation Variational Inference). The following example uses our module to fit a GP to some data and perform inference using the sampled posterior distributions. The code presented below generates Figure 5.4.

```
import numpy as np
import matplotlib.pyplot as plt
from pyGPGO.surrogates.GaussianProcessMCMC import GaussianProcessMCMC
from pyGPGO.covfunc import squaredExponential


if __name__ == '__main__':
    np.random.seed(1337)
    sexp = squaredExponential()
    gp = GaussianProcessMCMC(sexp, niter=2000, init='MAP')

    X = np.linspace(0, 6, 7)[:, None]
    y = np.sin(X).flatten()
    gp.fit(X, y)
    gp.posteriorPlot()
```

5.4.3 Using the `GPGO` module for global optimization.

In this example, we will try to optimize a variant of the Franke function [2] given by:

$$
\begin{aligned}
f(x,y) = & \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \\
& \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}\right) + \\
& \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \\
& \frac{1}{4}\exp\left(-(9x-4)^2 - (9y-7)^2\right)
\end{aligned}
\tag{5.1}
$$

A contour plot of said function can be checked in Figure 5.2. Optimum is around $\{x = 0.2, y = 0.2\}$. We use the Matérn ($\nu = 3/2$) covariance function and a simple Gaussian Process disregarding hyperparameter treatment.

```python
import numpy as np
from pyGPGO.covfunc import matern32
from pyGPGO.acquisition import Acquisition
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.GPGO import GPGO

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm

def f(x, y):
    # Franke's function (https://www.mathworks.com/help/curvefit/franke.html)
    one = 0.75 * np.exp(-(9*x-2)**2/4 - (9*y - 2)**2/4)
    two = 0.75 * np.exp(-(9*x+1)**2/49 - (9*y + 1)/10)
    three = 0.5 * np.exp(-(9*x - 7)**2/4 - (9*y -3)**2/4)
    four = 0.25 * np.exp(-(9*x -4)**2 - (9*y-7)**2)
    return one + two + three - four


def plotFranke():
    x = np.linspace(0, 1, num=1000)
    y = np.linspace(0, 1, num=1000)
    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    fig = plt.figure()
    ax = fig.gca(projection='3d')

    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                           linewidth=0)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()

if __name__ == '__main__':
    plotFranke()

    cov = matern32()
    gp = GaussianProcess(cov)
    acq = Acquisition(mode='ExpectedImprovement')
    param = {'x': ('cont', [0, 1]),
             'y': ('cont', [0, 1])}

    np.random.seed(1337)
    gpgo = GPGO(gp, acq, f, param)
    gpgo.run(max_iter=10)
```

5.4.4 Optimizing parameters of a machine-learning model using the GPGO module.

While pyGPGO can optimize any function the user specifies, the main topic of this main thesis was the application of these algorithms to optimize the hyperparameters of machine-learning algorithms. We provide a very simple way of how to do so using `scikit-learn`, arguably the most complete machine-learning package for Python. The example generates synthetic data and tries to optimize a Support

Vector Machine classifier parameters (C, γ), using cross-validation. A plot of the generated data can be checked in Figure 5.5.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.datasets import make_moons
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score


from pyGPGO.GPGO import GPGO
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential


def evaluateModel(C, gamma):
    clf = SVC(C=C, gamma=gamma)
    return np.average(cross_val_score(clf, X, y))


if __name__ == '__main__':
    np.random.seed(20)
    X, y = make_moons(n_samples = 200, noise = 0.3)

    cm_bright = ListedColormap(['#fc4349', '#6dbcdb'])

    fig = plt.figure()
    plt.scatter(X[:, 0], X[:, 1], c = y, cmap = cm_bright)
    plt.show()

    sexp = squaredExponential()
    gp = GaussianProcess(sexp, optimize = True, usegrads = True)
    acq = Acquisition(mode = 'UCB', beta = 1.5)

    params = {'C':      ('cont', (1e-4, 1e4)),
              'gamma':  ('cont', (1e-4, 10))
             }

    gpgo = GPGO(gp, acq, evaluateModel, params)
    gpgo.run(max_iter = 50)
    gpgo.getResult()
```

5.5 Comparison with existing software

pyGPGO is, to the best of my knowledge, one of the most complete packages in Python for performing Bayesian optimization. It implements plenty of functionality, e.g different surrogate regressors, several strategies for hyperparameter treatment while being completely modular and open-source. A complete comparison of features against other popular packages of the Python eco-system can be checked in Table 5.1.

Figure 5.2: Franke's function contour plot.

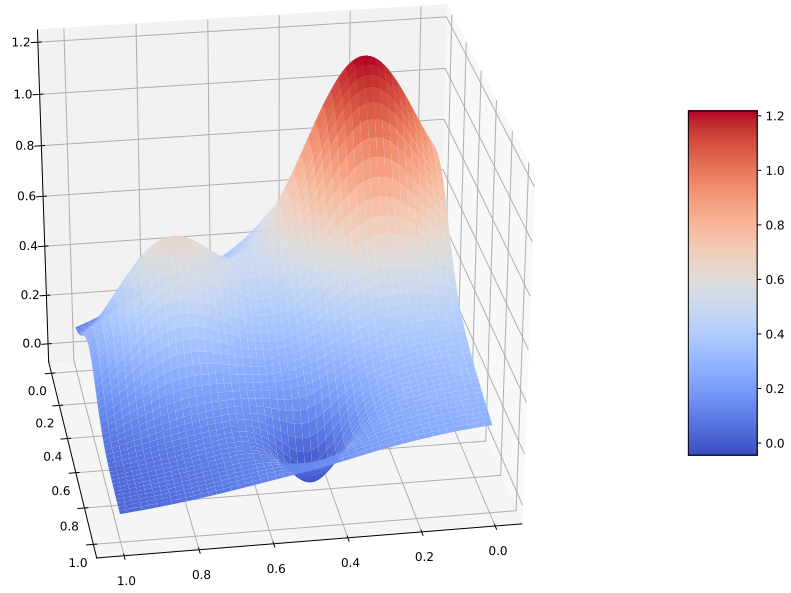


Figure 5.3: Gaussian Process regression of noisy inputs with pyGPGO.

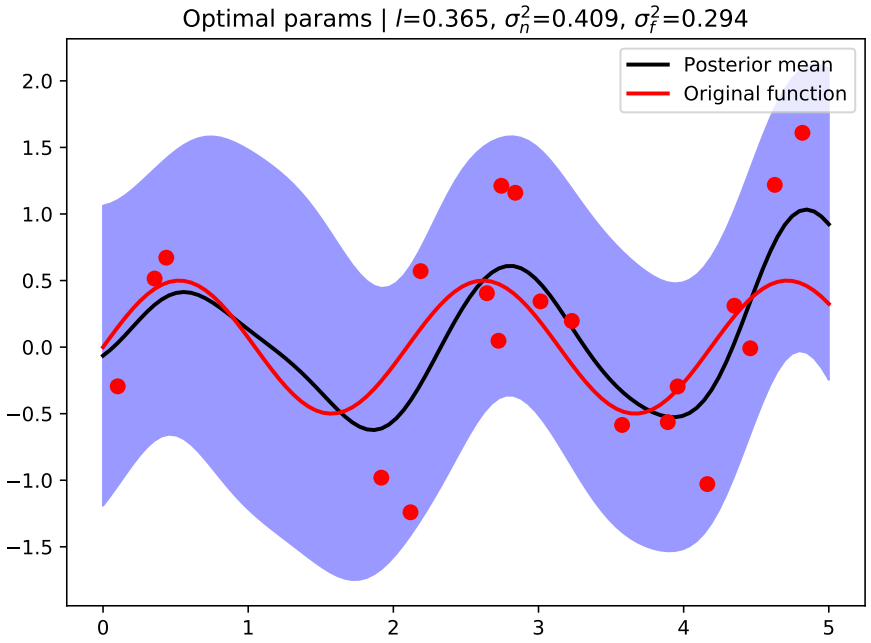


Figure 5.4: Posterior sampled distributions of hyperparameters using MCMC.

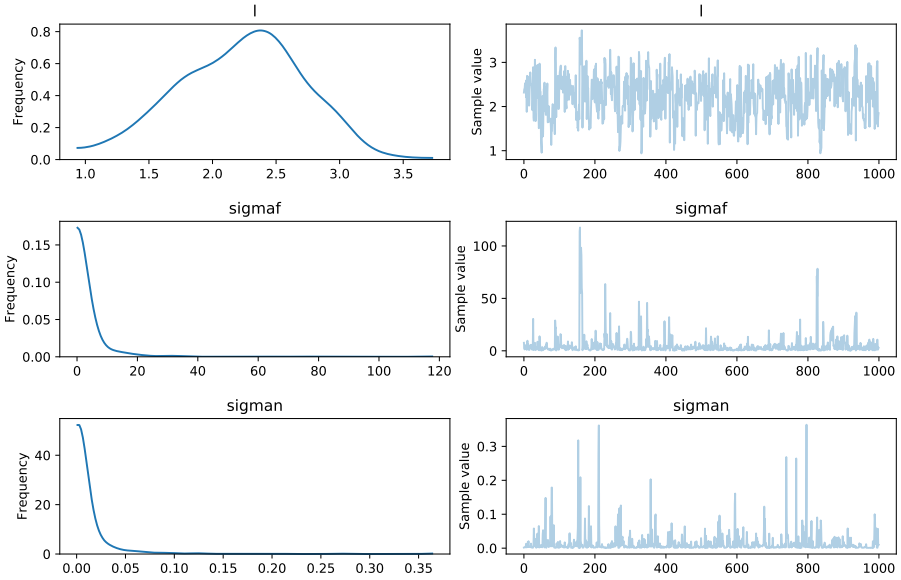
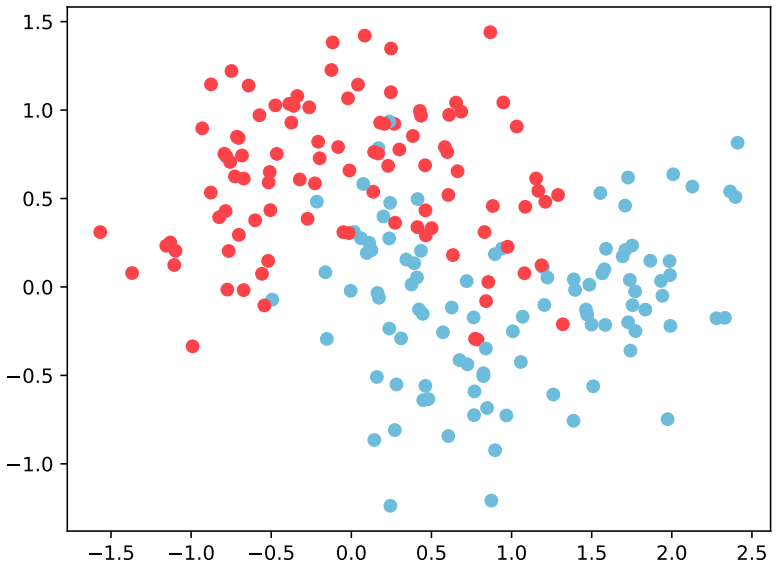


Figure 5.5: Synthetic data generated for our `sklearn` optimization example.



5.6 Future work

pyGPGO, while a very complete software for Bayesian Optimization still remains in development. In particular, there are several improvements that will be tackled on:

- Improvements over covariance functions: support for complex kernel structures, such as linear combinations of such, with automatic gradient computation.
- Support for more diverse acquisition functions: predictive entropy search, hedge or Thompson-like.
- A wider choice of surrogate models: Sparse-input or warped Gaussian Processes.
- GPU support for MCMC sampling.
- Overall speed improvements and code optimization.

In conclusion, pyGPGO, like any other piece of software, is never finished work. Being open-sourced, anyone can either contribute to the code base available through its GitHub repository or download the project and modify it in any way they see fit. pyGPGO's MIT License also allows for it to be used in mostly any kind of software, either academic or commercial.

Table 5.1: Comparison of pyGPGO features against other software packages for Bayesian Optimization in the Python environment. (as of Apr 2017)

	pyGPGO	Spearmint	fmfn/BayesianOptimization	pyBO	MOE	GPyOpt	scikit-optimize
GP implementation	Native	Native	via scikit-learn[a]	via Reggie[b]	Native	via GPy[c]	via scikit-learn
Modular	Yes	No	No	No	No	Experimental	No
Surrogates	{GP, tSP, RF, ET, GBM}	{GP}	{GP}	{GP}	{GP}	{GP, RF, WGP}	{GP, RF, GBM}
Type II ML optimization	Yes	No	No	No	Yes	Yes	Yes
MCMC inference	Yes (via pyMC3)[d]	Yes	No	Yes	No	Yes	No
Choice of MCMC sampler	Yes (via pyMC3)	Yes	No	Yes	No	No	No
Acquisition functions	{PI, EI, UCB, Entropy}	{EI}	{PI, EI, UCB}	{PI, EI, UCB, Th}	{EI}	{PI, EI, UCB}	{PI, EI, UCB}
Integrated acquisition functions	Yes	Yes	No	Yes	No	Yes	No
Parallel acquisition function opt.	Yes	No	No	No	No	No	No
License	MIT	Academic	MIT	BSD-2	Apache	BSD-3	BSD
Last update	-	Apr 2016	Mar 2017	Sept 2015	Apr 2016	Apr 2017	Apr 2017
Python version	>3.5	2.7	2/3	2/3	2.7	2/3	2/3

[a] http://scikit-learn.org/stable/
[b] https://github.com/mwhoffman/reggie
[c] https://github.com/SheffieldML/GPy
[d] https://github.com/pymc-devs/pymc3

Bibliography

[1] Inc Continuum Analytics. *Anaconda: Continuum Analytics*. 2016. URL: https://www.continuum.io/why-anaconda.

[2] Richard Franke. *A critical comparison of some methods for interpolation of scattered data*. Tech. rep. 1980, p. 379. URL: http://oai.dtic.mil/oai/oai?verb=getRecord%7B%5C&%7DmetadataPrefix=html%7B%5C&%7Didentifier=ADA081688.

[3] Md Hoffman and Andrew Gelman. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo". In: *Journal of Machine Learning Research* 15 (2014), p. 30. ISSN: 15337928. arXiv: 1111.4246.

[4] Alp Kucukelbir et al. "Automatic Differentiation Variational Inference". In: *Arxiv* (2016), pp. 1–38. ISSN: 15337928. DOI: 10.3847/0004-637X/819/1/50. arXiv: 1603.00788. URL: http://arxiv.org/abs/1603.00788.

Appendices

Appendix A

Examples code

A.1 drawGP.py

```
import numpy as np
from numpy.random import multivariate_normal
from covfunc import squaredExponential
import matplotlib.pyplot as plt

if __name__ == '__main__':
        np.random.seed(93)
        # Equally spaced values of Xstar
        Xstar = np.arange(0, 2 * np.pi, step = np.pi/16)
        Xstar = np.array([np.atleast_2d(x) for x in Xstar])[:, 0]
        sexp = squaredExponential()
        # By default assume mean 0
        m = np.zeros(Xstar.shape[0])
        # Compute squared-exponential matrix
        K = sexp.K(Xstar, Xstar)

        n_samples = 3
        # Draw samples from multivariate normal
        samples = multivariate_normal(m, K, size = n_samples)

        # Plot values
        x = Xstar.flatten()
        plt.figure()
        for i in range(n_samples):
                plt.plot(x, samples[i], label = 'GP sample {}'.format(i + 1))
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('Sampled GP priors from Squared Exponential kernel')
        plt.grid()
        plt.legend(loc = 0)
        plt.show()
```

A.2 sineGP.py

```
import numpy as np
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.covfunc import squaredExponential
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Build synthetic data (sine function)
    x = np.arange(0, 2 * np.pi + 0.01, step=np.pi / 2)
    y = np.sin(x)
    X = np.array([np.atleast_2d(u) for u in x])[:, 0]

    # Specify covariance function
    sexp = squaredExponential()

    # Instantiate GaussianProcess class
    gp = GaussianProcess(sexp)
    # Fit the model to the data
    gp.fit(X, y)

    # Predict on new data
    xstar = np.arange(0, 2 * np.pi, step=0.01)
    Xstar = np.array([np.atleast_2d(u) for u in xstar])[:, 0]
    ymean, ystd = gp.predict(Xstar, return_std=True)

    # Confidence interval bounds
    lower, upper = ymean - 1.96 * np.sqrt(ystd), ymean + 1.96 * np.sqrt(ystd)

    # Plot values
    plt.figure()
    plt.plot(xstar, ymean, label='Posterior mean')
    plt.plot(xstar, np.sin(xstar), label='True function')
    plt.fill_between(xstar, lower, upper, alpha=0.4, label='95% confidence band')
    plt.grid()
    plt.legend(loc=0)
    plt.show()
```

A.3 covzoo.py

```
from pyGPGO.covfunc import *
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Build synthetic data (sine function)
    x = np.arange(0, 2 * np.pi + 0.01, step=np.pi / 2.05)
    y = np.sin(x)
    X = np.array([np.atleast_2d(u) for u in x])[:, 0]

    # Covariance functions to loop over
    covfuncs = [squaredExponential(), matern(), gammaExponential(), rationalQuadratic()]
    titles = [r'Squared Exponential ($l = 1$)', r'Matern ($\nu = 1$, $l = 1$)',
              r'Gamma Exponential ($\gamma = 1, l = 1$)', r'Rational Quadratic ($\alpha = 1, l = 1$)
```

```
    plt.figure()
    #plt.rc('text', usetex=True)
    for i, cov in enumerate(covfuncs):
        gp = GaussianProcess(cov, optimize=True, usegrads=False)
        gp.fit(X, y)
        xstar = np.arange(0, 2 * np.pi, step=0.01)
        Xstar = np.array([np.atleast_2d(u) for u in xstar])[:, 0]
        ymean, ystd = gp.predict(Xstar, return_std=True)

        lower, upper = ymean - 1.96 * np.sqrt(ystd), ymean + 1.96 * np.sqrt(ystd)
        plt.subplot(2, 2, i + 1)
        plt.plot(xstar, ymean, label='Posterior mean')
        plt.plot(xstar, np.sin(xstar), label='True function')
        plt.fill_between(xstar, lower, upper, alpha=0.4, label='95\% confidence band')
        plt.grid()
        plt.title(titles[i])
    plt.legend(loc=0)
    plt.show()
```

A.4 hyperopt.py

```
import numpy as np
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.covfunc import squaredExponential
import matplotlib.pyplot as plt


def gradient(gp, sexp):
    alpha = gp.alpha
    K = gp.K
    gradK = sexp.gradK(gp.X, gp.X, 'l')
    inner = np.dot(np.atleast_2d(alpha).T, np.atleast_2d(alpha)) - np.linalg.inv(K)
    return (.5 * np.trace(np.dot(inner, gradK)))


if __name__ == '__main__':
    x = np.arange(0, 2 * np.pi + 0.01, step=np.pi / 2)
    X = np.array([np.atleast_2d(u) for u in x])[:, 0]
    y = np.sin(x)

    logp = []
    grad = []
    length_scales = np.linspace(0.1, 2, 1000)

    for l in length_scales:
        sexp = squaredExponential(l=l)
        gp = GaussianProcess(sexp)
        gp.fit(X, y)
        logp.append(gp.logp)
        grad.append(gradient(gp, sexp))

    plt.figure()
```

```
    plt.subplot(1, 2, 1)
    plt.plot(length_scales, logp)
    plt.title('Marginal log-likelihood')
    plt.xlabel('Characteristic length-scale l')
    plt.ylabel('log-likelihood')
    plt.grid()
    plt.subplot(1, 2, 2)
    plt.plot(length_scales, grad, '--', color='red')
    plt.title('Gradient w.r.t. l')
    plt.xlabel('Characteristic length-scale l')
    plt.grid()
    plt.show()
```

A.5 acqzoo.py

```
import numpy as np
import matplotlib.pyplot as plt
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential
from pyGPGO.GPGO import GPGO


def plotGPGO(gpgo, param, index, new=True):
    param_value = list(param.values())[0][1]
    x_test = np.linspace(param_value[0], param_value[1], 1000).reshape((1000, 1))
    y_hat, y_var = gpgo.GP.predict(x_test, return_std=True)
    std = np.sqrt(y_var)
    l, u = y_hat - 1.96 * std, y_hat + 1.96 * std
    if new:
        plt.figure()
        plt.subplot(5, 1, 1)
        plt.fill_between(x_test.flatten(), l, u, alpha=0.2)
        plt.plot(x_test.flatten(), y_hat)
    plt.subplot(5, 1, index)
    a = np.array([-gpgo._acqWrapper(np.atleast_1d(x)) for x in x_test]).flatten()
    plt.plot(x_test, a, color=colors[index - 2], label=acq_titles[index - 2])
    gpgo._optimizeAcq(method='L-BFGS-B', n_start=1000)
    plt.axvline(x=gpgo.best)
    plt.legend(loc=0)


if __name__ == '__main__':
    def f(x):
        return (np.sin(x))

    acq_1 = Acquisition(mode='ExpectedImprovement')
    acq_2 = Acquisition(mode='ProbabilityImprovement')
    acq_3 = Acquisition(mode='UCB', beta=0.5)
    acq_4 = Acquisition(mode='UCB', beta=1.5)
    acq_list = [acq_1, acq_2, acq_3, acq_4]
    sexp = squaredExponential()
```

```
    param = {'x': ('cont', [0, 2 * np.pi])}
    new = True
    colors = ['green', 'red', 'orange', 'black']
    acq_titles = [r'Expected improvement', r'Probability of Improvement', r'GP-UCB $\beta = .5$',
                  r'GP-UCB $\beta = 1.5$']

    for index, acq in enumerate(acq_list):
        np.random.seed(200)
        gp = GaussianProcess(sexp)
        gpgo = GPGO(gp, acq, f, param)
        gpgo._firstRun(n_eval=3)
        plotGPGO(gpgo, param, index=index + 2, new=new)
        new = False

    plt.show()
```

A.6 integratedacq.py

```
import numpy as np
import matplotlib.pyplot as plt
from pyGPGO.surrogates.GaussianProcessMCMC import GaussianProcessMCMC
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential
from pyGPGO.GPGO import GPGO


if __name__ == '__main__':
    sexp = squaredExponential()
    gp = GaussianProcessMCMC(sexp)

    def f(x):
        return np.sin(x)

    np.random.seed(200)
    param = {'x': ('cont', [0, 6])}
    acq = Acquisition(mode='IntegratedExpectedImprovement')
    gpgo = GPGO(gp, acq, f, param)
    gpgo._firstRun(n_eval=7)

    plt.figure()
    plt.subplot(2, 1, 1)

    Z = np.linspace(0, 6, 100)[:, None]
    post_mean, post_var = gpgo.GP.predict(Z, return_std=True, nsamples=200)
    for i in range(200):
        plt.plot(Z.flatten(), post_mean[i], linewidth=0.4)

    plt.plot(gpgo.GP.X.flatten(), gpgo.GP.y, '*', label='Sampled data', markersize = 10, color='red'
    plt.grid()
    plt.legend()

    xtest = np.linspace(0, 6, 200)[:, np.newaxis]
```

```
    a = [-gpgo._acqWrapper(np.atleast_2d(x)) for x in xtest]
    plt.subplot(2, 1, 2)
    plt.plot(xtest, a, label = 'Integrated Expected Improvement')
    plt.grid()
    plt.legend()
```

A.7 bayoptwork.py

```
import numpy as np
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.covfunc import squaredExponential
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Build synthetic data (sine function)
    x = np.arange(0, 2 * np.pi + 0.01, step=np.pi / 1.5)
    y = np.sin(x)
    X = np.array([np.atleast_2d(u) for u in x])[:, 0]

    # Specify covariance function
    sexp = squaredExponential()
    # Instantiate GPRegressor class
    gp = GaussianProcess(sexp)
    # Fit the model to the data
    gp.fit(X, y)

    # Predict on new data
    xstar = np.arange(0, 2 * np.pi, step=0.01)
    Xstar = np.array([np.atleast_2d(u) for u in xstar])[:, 0]
    ymean, ystd = gp.predict(Xstar, return_std=True)

    # Confidence interval bounds
    lower, upper = ymean - 1.96 * ystd, ymean + 1.96 * ystd

    # Plot values
    plt.figure()
    plt.plot(xstar, ymean, label='Posterior mean')
    plt.plot(xstar, lower, '--', label='Lower confidence bound')
    plt.plot(xstar, upper, '--', label='Upper confidence bound')
    plt.axhline(y=np.max(lower), color='black')
    plt.axvspan(0, .68, color='grey', alpha=0.3)
    plt.plot(xstar[np.argmax(lower)], np.max(lower), '*', markersize=20)
    plt.axvspan(3.04, 7, color='grey', alpha=0.3, label='Discarded region')
    plt.text(3.75, 0.75, 'max LCB')
    plt.grid()
    plt.legend(loc=0)
    plt.show()
```

A.8 sineopt.py

```
import numpy as np
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.covfunc import squaredExponential
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Build synthetic data (sine function)
    x = np.arange(0, 2 * np.pi + 0.01, step=np.pi / 2)
    y = np.sin(x)
    X = np.array([np.atleast_2d(u) for u in x])[:, 0]

    # Specify covariance function
    sexp = squaredExponential()

    # Instantiate GaussianProcess class
    gp = GaussianProcess(sexp)
    # Fit the model to the data
    gp.fit(X, y)

    # Predict on new data
    xstar = np.arange(0, 2 * np.pi, step=0.01)
    Xstar = np.array([np.atleast_2d(u) for u in xstar])[:, 0]
    ymean, ystd = gp.predict(Xstar, return_std=True)

    # Confidence interval bounds
    lower, upper = ymean - 1.96 * np.sqrt(ystd), ymean + 1.96 * np.sqrt(ystd)

    # Plot values
    plt.figure()
    plt.plot(xstar, ymean, label='Posterior mean')
    plt.plot(xstar, np.sin(xstar), label='True function')
    plt.fill_between(xstar, lower, upper, alpha=0.4, label='95% confidence band')
    plt.grid()
    plt.legend(loc=0)
    plt.show()
```

A.9 rastriginopt.py

```
import os
from collections import OrderedDict

import numpy as np
import matplotlib.pyplot as plt

from pyGPGO.GPGO import GPGO
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential


def rastrigin(x, y, A=10):
    return (2 * A + (x ** 2 - A * np.cos(2 * np.pi * x)) + (y ** 2 - A * np.cos(2 *
    ↪  np.pi * y)))
```

```
15
16
17 def plot_f(x_values, y_values, f):
18     z = np.zeros((len(x_values), len(y_values)))
19     for i in range(len(x_values)):
20         for j in range(len(y_values)):
21             z[i, j] = f(x_values[i], y_values[j])
22     plt.imshow(z.T, origin='lower', extent=[np.min(x_values), np.max(x_values),
       ↪ np.min(y_values), np.max(y_values)])
23     plt.colorbar()
24     plt.show()
25     plt.savefig(os.path.join(os.getcwd(),
       ↪ 'mthesis_text/figures/chapter3/rosen/rosen.pdf'))
26
27
28 def plot2dgpgo(gpgo):
29     tested_X = gpgo.GP.X
30     n = 100
31     r_x, r_y = gpgo.parameter_range[0], gpgo.parameter_range[1]
32     x_test = np.linspace(r_x[0], r_x[1], n)
33     y_test = np.linspace(r_y[0], r_y[1], n)
34     z_hat = np.empty((len(x_test), len(y_test)))
35     z_var = np.empty((len(x_test), len(y_test)))
36     ac = np.empty((len(x_test), len(y_test)))
37     for i in range(len(x_test)):
38         for j in range(len(y_test)):
39             res = gpgo.GP.predict([x_test[i], y_test[j]])
40             z_hat[i, j] = res[0]
41             z_var[i, j] = res[1][0]
42             ac[i, j] = -gpgo._acqWrapper(np.atleast_1d([x_test[i], y_test[j]]))
43     fig = plt.figure()
44     a = fig.add_subplot(2, 2, 1)
45     a.set_title('Posterior mean')
46     plt.imshow(z_hat.T, origin='lower', extent=[r_x[0], r_x[1], r_y[0], r_y[1]])
47     plt.colorbar()
48     plt.plot(tested_X[:, 0], tested_X[:, 1], 'wx', markersize=10)
49     a = fig.add_subplot(2, 2, 2)
50     a.set_title('Posterior variance')
51     plt.imshow(z_var.T, origin='lower', extent=[r_x[0], r_x[1], r_y[0], r_y[1]])
52     plt.plot(tested_X[:, 0], tested_X[:, 1], 'wx', markersize=10)
53     plt.colorbar()
54     a = fig.add_subplot(2, 2, 3)
55     a.set_title('Acquisition function')
56     plt.imshow(ac.T, origin='lower', extent=[r_x[0], r_x[1], r_y[0], r_y[1]])
57     plt.colorbar()
58     gpgo._optimizeAcq(method='L-BFGS-B', n_start=500)
59     plt.plot(gpgo.best[0], gpgo.best[1], 'gx', markersize=15)
60     plt.tight_layout()
61     plt.savefig(os.path.join(os.getcwd(),
       ↪ 'mthesis_text/figures/chapter3/rosen/{}.pdf'.format(item)))
62     plt.show()
63
64
65 if __name__ == '__main__':
66     x = np.linspace(-1, 1, 1000)
```

```
    y = np.linspace(-1, 1, 1000)
    plot_f(x, y, rastrigin)

    np.random.seed(20)
    sexp = squaredExponential()
    gp = GaussianProcess(sexp)
    acq = Acquisition(mode='ExpectedImprovement')

    param = OrderedDict()
    param['x'] = ('cont', [-1, 1])
    param['y'] = ('cont', [-1, 1])

    gpgo = GPGO(gp, acq, rastrigin, param, n_jobs=-1)
    gpgo._firstRun()

    for item in range(7):
        plot2dgpgo(gpgo)
        gpgo.updateGP()
```

Appendix B

Testing code

B.1 utils.py

```
import os

import numpy as np
import pandas as pd
from sklearn.metrics import log_loss, mean_squared_error
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler

from pyGPGO.GPGO import GPGO
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.acquisition import Acquisition
from pyGPGO.covfunc import squaredExponential


class loss:
    def __init__(self, model, X, y, method='holdout', problem='binary'):
        self.model = model
        self.X = X
        self.y = y
        self.method = method
        self.problem = problem
        sc = StandardScaler()
        self.X = sc.fit_transform(self.X)
        if self.problem == 'binary':
            self.loss = log_loss
        elif self.problem == 'cont':
            self.loss = mean_squared_error
        else:
            self.loss = log_loss

    def evaluateLoss(self, **param):
        if self.method == 'holdout':
            X_train, X_test, y_train, y_test = train_test_split(self.X, self.y,
             ↪  random_state=93)
            clf = self.model.__class__(**param, problem=self.problem).eval()
            clf.fit(X_train, y_train)
            if self.problem == 'binary':
```

```
37                 yhat = clf.predict_proba(X_test)[:, 1]
38             elif self.problem == 'cont':
39                 yhat = clf.predict(X_test)
40             else:
41                 yhat = clf.predict_proba(X_test)
42             return (- self.loss(y_test, yhat))
43         elif self.method == '5fold':
44             kf = KFold(n_splits=5, shuffle=False)
45             losses = []
46             for train_index, test_index in kf.split(self.X):
47                 X_train, X_test = self.X[train_index], self.X[test_index]
48                 y_train, y_test = self.y[train_index], self.y[test_index]
49                 clf = self.model.__class__(**param, problem=self.problem).eval()
50                 clf.fit(X_train, y_train)
51                 if self.problem == 'binary':
52                     yhat = clf.predict_proba(X_test)[:, 1]
53                 elif self.problem == 'cont':
54                     yhat = clf.predict(X_test)
55                 else:
56                     yhat = clf.predict_proba(X_test)
57                 losses.append(- self.loss(y_test, yhat))
58             return (np.average(losses))
59
60
61 def cumMax(history):
62     n = len(history)
63     res = np.empty((n,))
64     for i in range(n):
65         res[i] = np.max(history[:(i + 1)])
66     return (res)
67
68
69 def build(csv_path, target_index, header=None):
70     data = pd.read_csv(csv_path, header=header)
71     data = data.as_matrix()
72     y = data[:, target_index]
73     X = np.delete(data, obj=np.array([target_index]), axis=1)
74     return X, y
75
76
77 def evaluateDataset(csv_path, target_index, problem, model, parameter_dict,
   ↪ method='holdout', seed=20, max_iter=50):
78     print('Now evaluating {}...'.format(csv_path))
79     X, y = build(csv_path, target_index)
80
81     wrapper = loss(model, X, y, method=method, problem=problem)
82
83     print('Evaluating EI')
84     np.random.seed(seed)
85     sexp = squaredExponential()
86     gp = GaussianProcess(sexp, optimize=True, usegrads=True)
87     acq_ei = Acquisition(mode='ExpectedImprovement')
88     gpgo_ei = GPGO(gp, acq_ei, wrapper.evaluateLoss, parameter_dict, n_jobs=1)
89     gpgo_ei.run(max_iter=max_iter)
90
```

```
91      # Also add UCB, beta = 0.5, beta = 1.5
92      print('Evaluating UCB beta = 0.5')
93      np.random.seed(seed)
94      sexp = squaredExponential()
95      gp = GaussianProcess(sexp, optimize=True, usegrads=True)
96      acq_ucb = Acquisition(mode='UCB', beta=0.5)
97      gpgo_ucb = GPGO(gp, acq_ucb, wrapper.evaluateLoss, parameter_dict, n_jobs=1)
98      gpgo_ucb.run(max_iter=max_iter)
99
100     print('Evaluating UCB beta = 1.5')
101     np.random.seed(seed)
102     sexp = squaredExponential()
103     gp = GaussianProcess(sexp, optimize=True, usegrads=True)
104     acq_ucb2 = Acquisition(mode='UCB', beta=1.5)
105     gpgo_ucb2 = GPGO(gp, acq_ucb2, wrapper.evaluateLoss, parameter_dict, n_jobs=1)
106     gpgo_ucb2.run(max_iter=max_iter)
107
108     print('Evaluating random')
109     np.random.seed(seed)
110     r = evaluateRandom(gpgo_ei, wrapper.evaluateLoss, n_eval=max_iter + 1)
111     r = cumMax(r)
112
113     return np.array(gpgo_ei.history), np.array(gpgo_ucb.history),
        ↪  np.array(gpgo_ucb2.history), r
114
115
116 def plotRes(gpgoei_history, gpgoucb_history, gpgoucb2_history, random, datasetname,
    ↪  model, problem):
117     import matplotlib
118     import matplotlib.pyplot as plt
119     x = np.arange(1, len(random) + 1)
120     fig = plt.figure()
121     plt.plot(x, -random, label='Random search', color='red')
122     plt.plot(x, -gpgoei_history, label='GPGO (EI)', color='blue')
123     plt.plot(x, -gpgoucb_history, label=r'GPGO (UCB) $\beta=.5$', color='yellow')
124     plt.plot(x, -gpgoucb2_history, label=r'GPGO (UCB) $\beta=1.5$', color='green')
125     plt.grid()
126     plt.legend(loc=0)
127     plt.xlabel('Number of evaluations')
128     if problem == 'binary':
129         plt.ylabel('Best log-loss found')
130     else:
131         plt.ylabel('Best MSE found')
132     datasetname = datasetname.split('.')[0]
133     plt.savefig(os.path.join(os.path.abspath('.'),
        ↪  'testing/results/{}/{}.pdf'.format(model.name, datasetname)))
134     plt.close(fig)
135     return None
136
137
138 def evaluateRandom(gpgo, loss, n_eval=20):
139     res = []
140     for i in range(n_eval):
141         param = gpgo._sampleParam()
142         l = loss(**param)
```

```
143         res.append(l)
144         print('Param {}, loss: {}'.format(param, l))
145     return (res)
```

B.2 modaux.py

```
1  from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,
   ↪  AdaBoostClassifier, AdaBoostRegressor, \
2      GradientBoostingClassifier, GradientBoostingRegressor
3  from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
4  from sklearn.neural_network import MLPClassifier, MLPRegressor
5  from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
6  from sklearn.svm import SVC, SVR
7  from collections import OrderedDict
8
9  d_rf = OrderedDict()
10 d_rf['n_estimators'] = ('int', (10, 50))
11 d_rf['min_samples_split'] = ('cont', (0.1, 0.5))
12 d_rf['max_features'] = ('cont', (0.1, 0.5))
13
14 d_knn = OrderedDict()
15 d_knn['n_neighbors'] = ('int', (10, 50))
16
17 d_mlp = OrderedDict()
18 d_mlp['hidden_layer_size'] = ('int', (5, 50))
19 d_mlp['alpha'] = ('cont', (1e-5, 0.9))
20
21
22 d_svm = OrderedDict()
23 d_svm['C'] = ('cont', (-4, 5))
24 d_svm['gamma'] = ('cont', (-4, 5))
25
26 d_tree = OrderedDict()
27 d_tree['max_features'] = ('cont', (0.1, 0.99))
28 d_tree['max_depth'] = ('int', (4, 30))
29 d_tree['min_samples_split'] = ('cont', (0.1, 0.99))
30
31 d_ada = OrderedDict()
32 d_ada['n_estimators'] = ('int', (5, 200))
33 d_ada['learning_rate'] = ('cont', (1e-5, 1))
34
35 d_gbm = OrderedDict()
36 d_gbm['learning_rate'] = ('cont', (10e-5, 1e-1))
37 d_gbm['n_estimators'] = ('int', (10, 100))
38 d_gbm['max_depth'] = ('int', (2, 100))
39 d_gbm['min_samples_split'] = ('int', (2, 100))
40
41
42
43 class Tree:
44     def __init__(self, problem='binary', max_features=0.5, max_depth=1,
       ↪  min_samples_split=2):
```

```
45          self.problem = problem
46          self.max_features = max_features
47          self.max_depth = int(max_depth)
48          self.min_samples_split = min_samples_split
49          self.name = 'Tree'
50
51      def eval(self):
52          if self.problem == 'binary':
53              mod = DecisionTreeClassifier(max_features=self.max_features,
                ↪  max_depth=self.max_depth, min_samples_split=self.min_samples_split,
                ↪  random_state=20)
54          else:
55              mod = DecisionTreeRegressor(max_features=self.max_features,
                ↪  max_depth=self.max_depth, min_samples_split=self.min_samples_split,
                ↪  random_state=20)
56          return mod
57
58
59  class Ada:
60      def __init__(self, problem='binary', n_estimators=50, learning_rate=1):
61          self.problem = problem
62          self.n_estimators = int(n_estimators)
63          self.learning_rate = learning_rate
64          self.name = 'Ada'
65
66      def eval(self):
67          if self.problem == 'binary':
68              mod = AdaBoostClassifier(n_estimators=self.n_estimators,
                ↪  learning_rate=self.learning_rate, random_state=20)
69          else:
70              mod = AdaBoostRegressor(n_estimators=self.n_estimators,
                ↪  learning_rate=self.learning_rate, random_state=20)
71          return mod
72
73
74  class GBM:
75      def __init__(self, problem='binary', learning_rate=0.1, n_estimators=100,
        ↪  max_depth=3, min_samples_split=2,
76                   min_samples_leaf=1, min_weight_fraction_leaf=0.0, subsample=1.0,
                     ↪  max_features=1.0):
77          self.problem = problem
78          self.learning_rate = learning_rate
79          self.n_estimators = int(n_estimators)
80          self.max_depth = int(max_depth)
81          self.min_samples_split = int(min_samples_split)
82          self.min_samples_leaf = int(min_samples_leaf)
83          self.min_weight_fraction_leaf = min_weight_fraction_leaf
84          self.subsample = subsample
85          self.max_features = max_features
86          self.name = 'GBM'
87
88      def eval(self):
89          if self.problem == 'binary':
```

```
90              mod = GradientBoostingClassifier(learning_rate=self.learning_rate,
                ↪  n_estimators=self.n_estimators, max_depth=self.max_depth,
                ↪  min_samples_split=self.min_samples_split,
                ↪  min_samples_leaf=self.min_samples_leaf,
                ↪  min_weight_fraction_leaf=self.min_weight_fraction_leaf,
                ↪  subsample=self.subsample, max_features=self.max_features,
                ↪  random_state=20)
91          else:
92              mod = GradientBoostingRegressor(learning_rate=self.learning_rate,
                ↪  n_estimators=self.n_estimators, max_depth=self.max_depth,
                ↪  min_samples_split=self.min_samples_split,
                ↪  min_samples_leaf=self.min_samples_leaf,
                ↪  min_weight_fraction_leaf=self.min_weight_fraction_leaf,
                ↪  subsample=self.subsample, max_features=self.max_features,
                ↪  random_state=20)
93          return mod
94
95
96  class SVM:
97      def __init__(self, problem='binary', C=0, gamma=0, kernel='rbf'):
98          self.problem = problem
99          self.C = 10**C
100         self.gamma = 10**gamma
101         self.kernel = kernel
102         self.name = 'SVM'
103
104     def eval(self):
105         if self.problem == 'binary':
106             mod = SVC(kernel=self.kernel, C=self.C, gamma=self.gamma,
                    ↪  probability=True, random_state=20)
107         else:
108             mod = SVR(kernel=self.kernel, C=self.C, gamma=self.gamma)
109         return mod
110
111
112 class RF:
113     def __init__(self, problem='binary', n_estimators=10, max_features=0.5,
114                  min_samples_split=0.3, min_samples_leaf=0.2):
115         self.problem = problem
116         self.n_estimators = int(n_estimators)
117         self.max_features = max_features
118         self.min_samples_split = min_samples_split
119         self.min_samples_leaf = min_samples_leaf
120         self.name = 'RF'
121
122     def eval(self):
123         if self.problem == 'binary':
124             mod = RandomForestClassifier(n_estimators=self.n_estimators,
125                                          max_features=self.max_features,
126                                          min_samples_split=self.min_samples_split,
127                                          min_samples_leaf=self.min_samples_leaf,
128                                          n_jobs=-1,
129                                          random_state=20)
130         else:
131             mod = RandomForestRegressor(n_estimators=self.n_estimators,
```

```
                                        max_features=self.max_features,
                                        min_samples_split=self.min_samples_split,
                                        min_samples_leaf=self.min_samples_leaf,
                                        n_jobs=-1,
                                        random_state=20)
        return mod


class KNN:
    def __init__(self, problem='binary', n_neighbors=5, leaf_size=30):
        self.problem = problem
        self.n_neighbors = int(n_neighbors)
        self.leaf_size = int(leaf_size)
        self.name = 'KNN'

    def eval(self):
        if self.problem == 'binary':
            mod = KNeighborsClassifier(n_neighbors=self.n_neighbors,
                                       leaf_size=self.leaf_size)
        else:
            mod = KNeighborsRegressor(n_neighbors=self.n_neighbors,
                                      leaf_size=self.leaf_size)
        return mod


class MLP:
    def __init__(self, problem='binary', hidden_layer_size=100, alpha=10e-4,
                 learning_rate_init=10e-4, beta_1=0.9, beta_2=0.999):
        self.problem = problem
        self.hidden_layer_sizes = (int(hidden_layer_size),)
        self.alpha = alpha
        self.learning_rate_init = learning_rate_init
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.name = 'MLP'

    def eval(self):
        if self.problem == 'binary':
            mod = MLPClassifier(hidden_layer_sizes=self.hidden_layer_sizes,
             ↪  alpha=self.alpha, learning_rate_init=self.learning_rate_init,
             ↪  beta_1=self.beta_1, beta_2=self.beta_2, random_state=20)
        else:
            mod = MLPRegressor(hidden_layer_sizes=self.hidden_layer_sizes,
             ↪  alpha=self.alpha, learning_rate_init=self.learning_rate_init,
             ↪  beta_1=self.beta_1, beta_2=self.beta_2, random_state=20)
        return mod
```

B.3 testing.py

```
from testing.utils import evaluateDataset, plotRes
from testing.modaux import *
import os
```

```
import numpy as np

if __name__ == '__main__':
    models = [SVM(), MLP(), GBM(), KNN()]
    params = [d_svm, d_mlp, d_gbm, d_knn]

    path = os.path.join(os.getcwd(), 'datasets')
    datasets = ['aff.csv', 'pinter.csv', 'breast_cancer.csv', 'indian_liver.csv',
    ↪  'parkinsons.csv',
                'lsvt.csv', 'pima-indians-diabetes.csv']
    problems = ['cont', 'binary', 'binary', 'binary', 'binary', 'binary', 'binary']
    targets = [0, 0, 0, 10, 16, 0, 8]

    for model, parameter_dict in zip(models, params):
        print('Evaluating model {}'.format(model.name))
        for dataset, target, problem in zip(datasets, targets, problems):
            model.problem = problem
            np.random.seed(93)
            print(np.random.randn(1))
            try:
                g, g2, g3, r = evaluateDataset(os.path.join(path, dataset),
                ↪  target_index=target, model=model, parameter_dict=parameter_dict,
                ↪  method='5fold', seed=20, max_iter=50, problem=problem)
                plotRes(g, g2, g3, r, dataset, model, problem=problem)
                print(np.random.randn(1))
            except Exception as e:
                print(e)
                continue
```