Bayesian optimization in machine learning

José Jiménez Luna

January 19, 2017

Contents

Chapter 1

Organization of this work

1.1 Introduction

This Master's Thesis provides an introduction to both Gaussian Processes and Bayesian Optimization. This work 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. The work is written in such a way that it alternates very often between theoretical and practical (in terms of programming) background. This work is written like a programming book or a manual.
- To show the Bayesian Optimization framework works in several real-world machine learning tasks. I do so by selecting several datasets (most of them from biological/physical phenomena), applying such methodology and comparing to other common strategies applied for the same problem.
- Finally, and my personal favourite, to provide the user with usable and easy to use software to apply Bayesian Optimization in their research. This comes in the form of a Python (>3.5) package named pyGPGO. The code can be freely obtained in `https://github.com/hawk31/pyGPGO`. The software is MIT licensed. All the examples and code snippets throughout this manual are based on this software.

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 A. The problem can be formalized as:

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

Most optimization procedures (local based ones such as gradient ascent, for example) assume that the function f is closed-form, that is, can be written in a paper, 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 to find the global minimum of a non-necessarily convex, expensive functions. These methods shine also where there is no closed-form expression to evaluate and does not need any function derivatives.

Now is when the machine learning part of the title comes into play. In machine learning, we are usually interested in minimizing (or maximizing the opposite) a loss function $\mathcal{L}$. These losses can take many forms, for example, 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 that in any case, that these losses are typically defined in a subset of $\mathbb{R}$. We focus on the supervised setting of machine learning, and more specifically regression. 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 the size of the dataset at hand. These machine learning algorithms typically have *hyperparameters*, that is, parameters 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 do an hyperparameter grid search or even randomize it. Since the training of a single model can already take substantial resources in terms of CPU cycles or memory, 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 point $\boldsymbol{x}$ to evaluate according to several criteria.

1.2 How do I read this?

Before we dive directly into the topic at hand, it is mandatory to explain how this manual is intended to be read. Different chapters will cover different material, so if the reader is familiar with a topic in a chapter, for example, the chapter on Gaussian Processes, he can skip directly to the chapter on Bayesian Optimization or to the implementation if he wishes. There is no complicated or unnecessarily advanced material in any of the chapters, as this manual does not aim to be bleeding edge research material.

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 cover the theory behind them both from a weight space view and from a functional view. We will also explain different covariance functions and their role in these models, as well as methods for optimizing their hyperparameters. Finally we will provide the reader with usable code to fit a Gaussian Process in a regression problem. This chapter is heavily based on Carl E. Rasmussen excellent book *Gaussian Processes for Machine Learning*[ref], an more specially in chapters 2, 4 and 5. In fact, if the reader is interested in a more advanced, wider, slowly-paced introduction to Gaussian Processes, this is by far the best resource to go to.

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 algorithms, while simple, are very powerful. The role of several acquisition functions, that is to say, the functions that will propose the next point to evaluate will be discussed, as well as their advantages or disadvantages. The references on this chapter will be more diverse than on the previous chapter, as I will try to summarize several publications. Readers already familiar with Gaussian Processes should jump directly into this chapter.

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 or simulated annealing. Most of these datasets are related to experimental sciences, and some of them are typically used for other benchmarking purposes in other studies.

Chapter 5 has no theoretical content and is the shortest one. It will cover technical explanations of pyGPGO, the software developed alongside this manual. I will also try to provide practical examples on how to use the software.

Chapter 2

Gaussian Process regression

We will focus on regression problems in this chapter. Assume we have some labelled data $D = \{(\boldsymbol{x_i}, y_i) \,|i = 1, \ldots, n\}$, 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 $\boldsymbol{x}$, so that we can make predictions and inference over these. In practice, for simplicity we write that $D = (X, \boldsymbol{y})$, where X is a covariate matrix.

One can interpret a Gaussian Process in several ways, the most used one is the function space view, which is the one we will cover second here and the one we will assume for the rest for the manual. In this view, we consider a Gaussian Process to be 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 first, that might be more appealing to readers familiar with Bayesian linear regression.

2.1 A weight space view for Gaussian Processes

In this section we will try to draw connections between Bayesian linear regression and our introduction to Gaussian Processes, by the use of kernel functions.

2.1.1 Standard Bayesian linear regression

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

$$y = f(\boldsymbol{x}) + \epsilon \tag{2.1}$$

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

$$p(\boldsymbol{y}|X, \boldsymbol{w}) \sim N(X^T \boldsymbol{w}, \sigma_n^2 I) \tag{2.2}$$

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

$$\boldsymbol{w} \sim N(\boldsymbol{0}, \Sigma_p) \tag{2.3}$$

We are interested now on the posterior distribution of w, given both X and y, and assuming the model in Equation 2.1, that is:

$$p(\boldsymbol{w}|\boldsymbol{y}, X) = \frac{p(\boldsymbol{y}|X, \boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{y}|X)} \tag{2.4}$$

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 quite easily that:

$$p(\boldsymbol{w}|X, \boldsymbol{y}) \sim N\left(\frac{1}{\sigma_n^n} A^{-1} X \boldsymbol{y}, A^{-1}\right) \tag{2.5}$$

where $A = \sigma^{-2} X X^T + \Sigma_p^{-1}$. Notice that a very easy 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 proven that:

$$f_*|x_*, X, \boldsymbol{y} \sim N\left(\frac{1}{\sigma_n^2}\boldsymbol{x_*}^T A^{-1} X\boldsymbol{y}, \boldsymbol{x_*}^T A^{-1}\boldsymbol{x_*}\right) \tag{2.6}$$

2.1.2 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. 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 called using the kernel trick. We can do this by the use of 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:

$$f(\boldsymbol{x}) = \phi(\boldsymbol{x})^T \boldsymbol{w} \tag{2.7}$$

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:

$$f_*|x_*, X, \boldsymbol{y} \sim N\left(\frac{1}{\sigma_n^2}\phi(\boldsymbol{x_*})^T A^{-1}\Phi\boldsymbol{y}, \phi(\boldsymbol{x_*})^T A^{-1}\phi(\boldsymbol{x_*})\right) \tag{2.8}$$

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 2.8 can be rewritten as:

$$f_*|x_*, X, \boldsymbol{y} \sim N\left(\boldsymbol{\phi_*}^T\Sigma_p\Phi(K + \sigma_n^2 I)^{-1}\boldsymbol{y}, \boldsymbol{\phi_*}\Sigma_p\boldsymbol{\phi_*} - \boldsymbol{\phi_*}^T\Sigma_p\Phi(K + \sigma_n^2 I)^{-1}\Phi^T\Sigma_p\boldsymbol{\phi_*}\right) \tag{2.9}$$

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:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \psi(\boldsymbol{x})^T\psi(\boldsymbol{x'}) \tag{2.10}$$

where $\Sigma_p^{1/2}$ can be defined by means of a singular value decomposition. Typically, we will replace the original feature vectors by these dot products, *lifting* to a higher space. This will make more sense in the following section.

2.2 A function space view for Gaussian Processes

We formally define a Gaussian Process as a collection of random variables, any finite number of which have a joint Gaussian distribution. This process is totally defined by two functions. The first one is its *mean function*:

$$m(\boldsymbol{x}) = \mathbb{E}\left[f(\boldsymbol{x})\right] \tag{2.11}$$

The second one is its *covariance function*:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \mathbb{E}\left[\left(f(x) - m(x)\right)\left(f(x') - m(x')\right)\right] \tag{2.12}$$

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

$$f(\boldsymbol{x}) \sim \mathrm{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x'})\right) \tag{2.13}$$

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

As seen in the previous section, a Gaussian Process can be viewed as a Bayesian regression model using a particular kernel, that is, for the model $f(\boldsymbol{x}) = \phi(\boldsymbol{x})^T \boldsymbol{w}$ and with the same prior as in equation 2.3 has mean and covariance functions:

$$\mathbb{E}\left[f(\boldsymbol{x})\right] = \phi(\boldsymbol{x})^T \mathbb{E}\left[\boldsymbol{w}\right] = 0 \tag{2.14}$$

$$\mathbb{E}\left[f(\boldsymbol{x}) f(\boldsymbol{x'})\right] = \phi(\boldsymbol{x})^T \mathbb{E}\left[\boldsymbol{w}\boldsymbol{w}^T\right] \phi(\boldsymbol{x'}) = \phi(\boldsymbol{x})^T \Sigma_p \phi(\boldsymbol{x'}) \tag{2.15}$$

It is now a good time to start specifying our first covariance function, the *squared exponential* kernel, defined as:

$$k(x, x') = \exp\left(-\frac{1}{2}|x - x'|^2\right) \tag{2.16}$$

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 $r = |x - x'|$ and therefore the squared exponential kernel becomes:

$$k(x, x') = \exp\left(-\frac{1}{2}r^2\right) \tag{2.17}$$

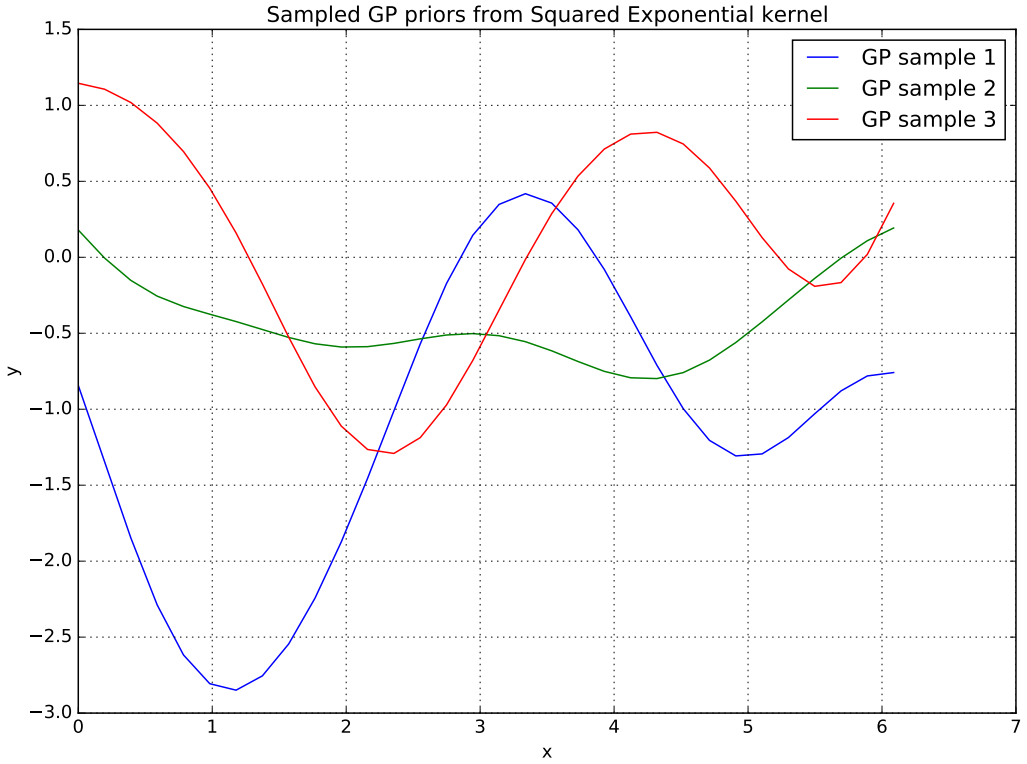
The squared exponential covariance kernel is equivalent to a Bayesian linear model with an infinite number of basis functions. This also implies a distribution over functions. To see this, choose an arbitrary number of points X_* and compute the squared exponential kernel for those. Then just sample from the following multivariate Gaussian:

$$\boldsymbol{f_*} \sim N\left(\boldsymbol{0}, K(X_*, X_*)\right) \tag{2.18}$$

To illustrate this point, we will write a very simple Python script to draw (finite) samples from this function. For the moment, consider evenly spaced samples with a step of $\frac{\pi}{16}$. Some of the code here already uses the pyGPGO implementation of the covariance function, to simplify computation. This code produces Figure 2.1.

```
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')
```

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




```
27          plt.ylabel('y')
28          plt.title('Sampled GP priors from Squared Exponential kernel')
29          plt.grid()
30          plt.legend(loc = 0)
31          plt.show()
```

There's one important concept to explain before we move on. Notice in Figure 2.1 that the drawn functions seem to have a characteristic length-scale. This can be interpreted as the distance you have 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 to another, it is sufficient to consider r/l instead of r in Equation 2.17. This can be thought as an hyperparameter to optimize, but we will return to this in another section.

2.3 Prediction using a Gaussian Process prior

This is probably the most important section in this chapter. We will learn how to incorporate the knowledge of training data $D = \{(\boldsymbol{x_i}, y_i)\,|i = 1, \ldots, n\}$ 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:*

$$\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{bmatrix} \sim N\left(\begin{bmatrix}\boldsymbol{\mu_x}\\ \boldsymbol{\mu_y}\end{bmatrix}, \begin{bmatrix}A & C\\ C^T & B\end{bmatrix}\right) \tag{2.19}$$

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

The reader might already be guessing what we are about to do now in terms of the training output points $\boldsymbol{f}$ and the corresponding testing points $\boldsymbol{f_*}$. According to the prior chosen in 2.18, assume that $\boldsymbol{f}$ and $\boldsymbol{f_*}$ are jointly Gaussian:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f_*} \end{bmatrix} \sim N\left(\boldsymbol{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{2.20}$$

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

$$\boldsymbol{f_*}|\boldsymbol{f} \sim N\left(K(X_*,X)K(X,X)^{-1}\boldsymbol{f}, K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)\right) \tag{2.21}$$

This is pretty much everything basic there is to know about Gaussian Process estimation for regression. Now we have a complete predictive distribution over test values $\boldsymbol{f_*}$, and we can do with it what we please. 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 you query the function there is a i.i.d Gaussian error with mean 0 and variance $\sigma_n^2 > 0$. Assume now the following prior on the noisy observations:

$$Cov(\boldsymbol{y}) = K(X,X) + \sigma_n^2 I \tag{2.22}$$

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

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f_*} \end{bmatrix} \sim N\left(\boldsymbol{0}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{2.23}$$

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

$$\boldsymbol{f_*}|\boldsymbol{f} \sim N(\overline{\boldsymbol{f_*}}, Cov(\boldsymbol{f_*})) \tag{2.24}$$

where now:

$$\overline{\boldsymbol{f_*}} = K(X_*,X)\left(K(X,X) + \sigma_n^2 I\right)^{-1}\boldsymbol{y} \tag{2.25}$$

$$Cov(\boldsymbol{f_*}) = K(X_*,X_*) - K(X_*,X)\left(K(X,X) + \sigma_n^2 I\right)^{-1}K(X,X_*) \tag{2.26}$$

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

$$\boldsymbol{y}|\boldsymbol{f} \sim N(\boldsymbol{f}, \sigma_n^2 I) \tag{2.27}$$

$$\boldsymbol{f}|X \sim N(\boldsymbol{0}, K(X,X)) \tag{2.28}$$

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:

$$p(\boldsymbol{y}|X) = \int p(\boldsymbol{y}|\boldsymbol{f},X)p(\boldsymbol{f}|X)d\boldsymbol{f} \tag{2.29}$$

Using the results from Equations 2.27 and 2.28 we can derive the integral analytically to obtain:

$$\log p(\boldsymbol{y}|X) = -\frac{1}{2}\boldsymbol{y}^T(K + \sigma_n^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi \tag{2.30}$$

We have now all the necessary ingredients to lay down pseudo-code for your own implementation of a Gaussian Process regressor, as presented in Algorithm 1. 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 GPREGRESSOR(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 `GPRegressor` module. A minimal working skeleton of its code is shown below. We do now show the entirety of the module code since most of it is not relevant at this point.

```
import numpy as np
from scipy.linalg import cholesky, solve
from collections import OrderedDict
from scipy.optimize import minimize

class GPRegressor:
        def __init__(self, covfunc, sigma = 0):
                self.covfunc = covfunc
                self.sigma = sigma
        def fit(self, X, y):
                self.X = X
                self.y = y
                self.nsamples = self.X.shape[0]
                self.K = self.covfunc.K(self.X, self.X)
                self.L = cholesky(self.K + self.sigma * np.eye(self.nsamples)).T
                self.alpha = solve(self.L.T, solve(self.L, y))
                self.logp = -.5 * np.dot(self.y, self.alpha) - np.sum(np.log(np.diag(self.L)))
                 - self.nsamples/2 * np.log(2 * np.pi)

        def predict(self, Xstar, return_std = False):
                Xstar = np.atleast_2d(Xstar)
                kstar = self.covfunc.K(self.X, Xstar).T
                fmean = np.dot(kstar, self.alpha)
                v = solve(self.L, kstar.T)
                fcov = self.covfunc.K(Xstar, Xstar) - np.dot(v.T, v)
                if return_std:
                        fcov = np.diag(fcov)
                return fmean, fcov

        def update(self, xnew, ynew):
                y = np.concatenate((self.y, ynew), axis = 0)
                X = np.concatenate((self.X, xnew), axis = 0)
                self.fit(X, y)
```

A few notes on the code, first notice that the `GPRegressor` class only takes as input the covariance function to use k, and the noise level σ_n^2. It is only when you call the `fit` method on the function when you actually fit the model on some data $X, \boldsymbol{y}$. Quantities like the log-marginal likelihood are store within the object. The `predict` method not only can output the variance for a single prediction, but the covariance matrix for m new ones if asked to. For convenience, we also implement an `update` method to update the Gaussian Process parameters when new data is available to us.

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 shown below produces Figure 2.2.

```
import numpy as np
from GPRegressor import GPRegressor
from 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 GPRegressor class
        gp = GPRegressor(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, 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()
```

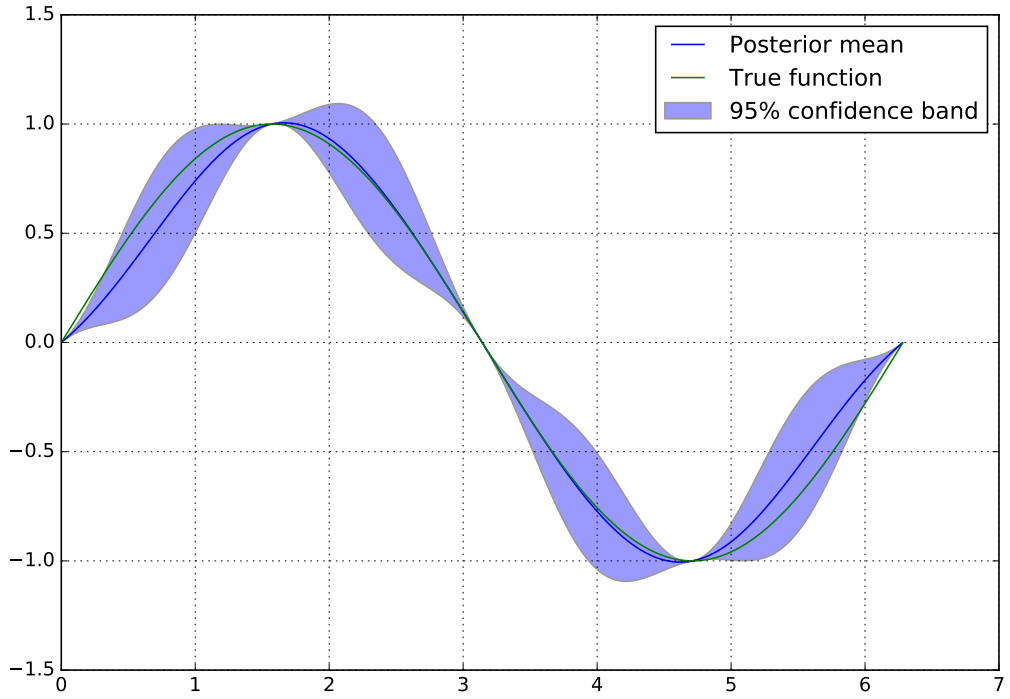
2.4 On covariance functions

A covariance function, 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 any function of $\boldsymbol{x}$ and $\boldsymbol{x'}$ can be considered a covariance function, in general should fulfil properties like the following:

- *Stationarity.* A covariance function is said to be 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 Chapter 4 of Carl E. Rasmussen's book *Gaussian Processes for Machine Learning*. We will not cover this here since it falls beyond the scope of this manual. However, we will start providing examples of the most common covariance functions and

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



their implementation in pyGPGO.

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:

$$k_{SE}(r) = \exp\left(-\frac{r^2}{2l^2}\right) \tag{2.31}$$

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. A simple skeleton of its implementation in pyGPGO is the following:

```
import numpy as np
from scipy.special import gamma, kv
from scipy.spatial.distance import cdist

def l2norm_(X, Xstar):
        return cdist(X, Xstar)

class squaredExponential:
        def __init__(self, l = 1, sigma2f = 1, sigma2n = 0):
                self.l = l
                self.sigma2f = sigma2f
                self.sigma2n = sigma2n

        def K(self, X, Xstar):
                r = l2norm_(X, Xstar)
                return(np.exp(-.5 * r ** 2 / self.l ** 2))
```

The *Matern class* of covariance functions takes the form:

$$k_{\text{Matern}}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{l} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu} r}{l} \right) \tag{2.32}$$

with $\nu, l > 0$ and K_{ν} is a modified Bessel function of the second kind. Notice that if we take limit $\nu \to \infty$ we obtain our beloved squared exponential covariance function. pyGPGO implementation below.

```
class matern:
        def __init__(self, v = 1, l = 1):
                self.v, self.l = v, l

        def K(self, X, Xstar):
                r = l2norm_(x, xstar)
                bessel = kv(self.v, np.sqrt(2 * self.v) * r / self.f)
                f = 2 ** (1 - self.v) / gamma(self.v) *
                        (np.sqrt(2 * self.v) * r / self.l) ** self.v
                res = f * bessel
                res[np.isnan(res)] = 1
                return(res)
```

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

$$k(r) = \exp\left(- \left(\frac{r}{l} \right)^{\gamma} \right) \tag{2.33}$$

for $0 < \gamma \leq 2$. Its implementation in pyGPGO below:

```
class gammaExponential:
        def __init__(self, gamma = 1, l = 1):
                self.gamma = gamma
                self.l = l

        def K(self, X, Xstar):
                r = l2norm_(X, Xstar)
                return(np.exp(-(r / self.l) ** self.gamma))
```

The *Rational Quadratic* covariance function takes the form:

$$k_{RQ}(r) = \left(1 + \frac{r^2}{2\alpha l^2} \right)^{-\alpha} \tag{2.34}$$

with $\alpha, l > 0$. And again its implementation:

```
class rationalQuadratic:
        def __init__(self, alpha = 1, l = 1):
                self.alpha = alpha
                self.l = l

        def K(self, X, Xstar):
                r = l2norm_(X, Xstar)
                return((1 + r**2/(2 * self.alpha * self.l **2))**(-self.alpha))
```

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

$$k_{\text{arcSin}}(\boldsymbol{x}, \boldsymbol{x'}) = \frac{2}{\pi} \sin^{-1} \left(\frac{2\boldsymbol{x}\Sigma\boldsymbol{x'}}{\sqrt{(1 + 2\boldsymbol{x}^T\Sigma\boldsymbol{x})(1 + 2\boldsymbol{x'}^T\Sigma\boldsymbol{x})}} \right) \tag{2.35}$$

And I promise this is the final covariance function implementation in this section.

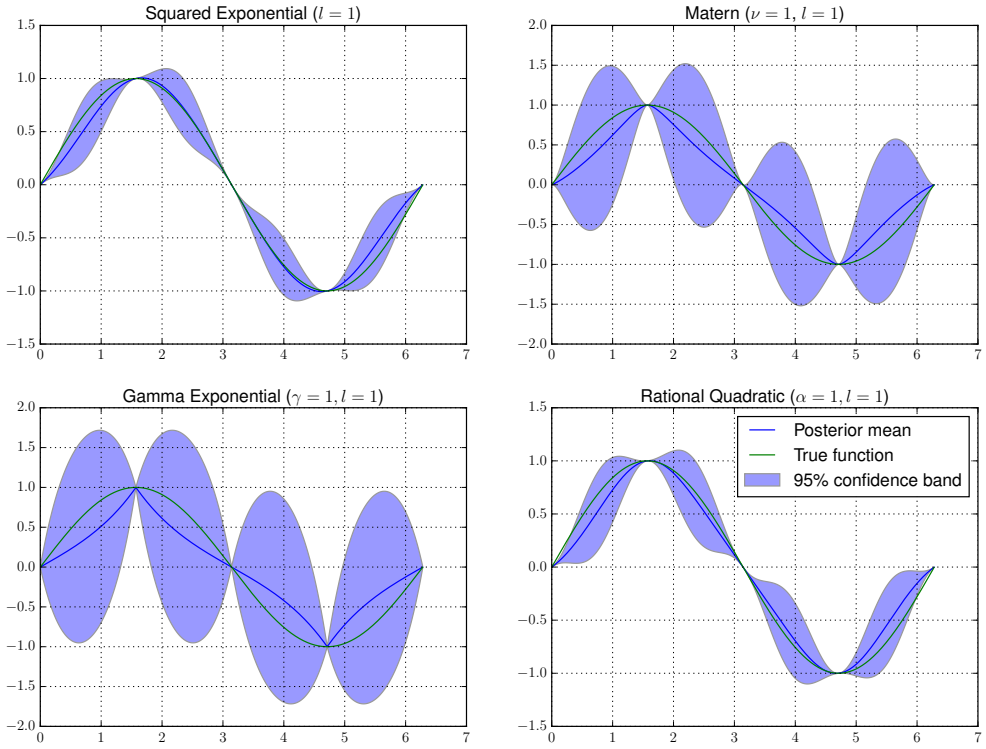
```
class arcSin:
        def __init__(self, n, sigma = None):
                if sigma == None:
                        self.sigma = np.eye(n)
                else:
                        self.sigma = sigma
        def k(self, x, xstar):
                num = 2 * np.dot(np.dot(x[np.newaxis, :], self.sigma), xstar)
                a = 1 + 2 * np.dot(np.dot(x[np.newaxis, :], self.sigma), x)
                b = 1 + 2 * np.dot(np.dot(xstar[np.newaxis, :], self.sigma), xstar)
                res = num / np.sqrt(a * b)
                return(res)
```

2.4.1 A zoo of 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 shown below produces Figure 2.3.

```
import numpy as np
from covfunc import *
from GPRegressor import GPRegressor
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]

        # 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 = GPRegressor(cov)
                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 * ystd, ymean + 1.96 * 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,
```

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




```
                label = '95\% confidence band')
            plt.grid()
            plt.title(titles[i])
    plt.legend(loc = 0)
    plt.show()
```

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 optimize these according to the data they have at hand. Depending on the situation, this optimization may lead to better models, either in terms of accuracy or interpretability. There are several ways to select these hyperparameters, one more analytical, by optimizing the marginal log-likelihood, and simpler ones based on cross-validation.

2.5.1 Type II Maximum Likelihood

This is the Bayesian 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 easily given by:

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

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 space of the numerator in Equation 2.36.

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 2.37 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.

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

where $\frac{\partial K}{\partial\theta_j}$ 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, for example, the complete specification of the `squaredExponential` class is:

```
class squaredExponential:
        def __init__(self, l = 1, sigma2f = 1, sigma2n = 0):
                self.l = l
                self.sigma2f = sigma2f
                self.sigma2n = sigma2n

        def K(self, X, Xstar):
                r = l2norm_(X, Xstar)
                return(np.exp(-.5 * r ** 2 / self.l ** 2))

        def gradK(self, X, Xstar, param = 'l'):
                if param == 'l':
                        r = l2norm_(X, Xstar)
                        num = r**2 * np.exp(-r**2 /(2* self.l**2))
                        den = self.l ** 3
                        l_grad = num / den
                        return(l_grad)
                else:
                        raise ValueError('Param not found')
```

By default, the `GPRegressor` class in pyGPGO does no hyperparameter optimization unless explicitly asked to. If the gradient of the covariance function is available, it will make use of it, if not, it will be estimated in a numerical fashion using the L-BFGS-B algorithm.

Another toy example: Optimizing the characteristic length-scale

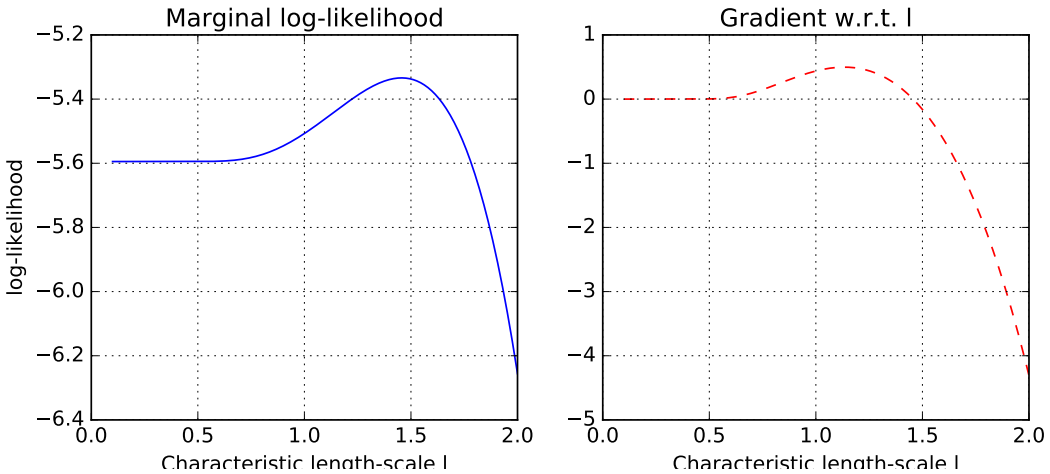
To illustrate the previous point, it may be a good idea to see the behaviour of the marginal log-likelihood and its gradient when we modify the characteristic length scale l in the squared exponential covariance function. The sine function will also serve as playground here. The code below produces Figure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

```
import numpy as np
from GPRegressor import GPRegressor
from 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 = GPRegressor(sexp)
                gp.fit(X, y)
                logp.append(gp.logp)
                grad.append(gradient(gp, sexp))

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

2.5.2 Cross validation

There is a very extensive analytical evaluation of cross validation using Gaussian Process in Rasmussen's book. Here, we will only lay down some very basic ideas related to model selection from a very basic machine learning perspective. The concepts presented here should not be new to the reader, but are more useful if one plans to use Gaussian Processes only as a regression model, not as a surrogate one towards another objective, as we will do in the following chapter.

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 $l = 1.4$.



To evaluate the performance of hyperparameters $\boldsymbol{\theta}$, in $D = (X, \boldsymbol{y})$ one could do the following:

- **Holdout test**. Consider $D = \{D_T, D_V\}$ as a training and validation set from your data and $\boldsymbol{\nu}$ the set of hyperparameter vectors $\boldsymbol{\nu}$ to test. Train your Gaussian Process regressor on D_T with some subset of hyperparameters from $\boldsymbol{\nu}$ and test its performance according to some loss metric $\mathcal{L}$ on D_V. Choose hyperparameters according to the lower loss obtained.
- **k-fold cross validation**. Instead of considering a single test set D_V, partition $D = D_1, \ldots, D_k$. Train your model iteratively on $k - 1$ sets and test on the remaining one. Consider an average of losses for each hyperparameter to test.

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 a new point to evaluate $\boldsymbol{x}$ and some bounds to optimize over. This is normally called a *black box* function. Moreover, function response can be noisy. This is the case when optimizing a loss or fitness function in machine learning, since we only have 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. [refs] It has grabbed the attention of machine learning researchers over simpler model hyperparameter optimization strategies, such as grid search [ref], random search [ref] or simulated annealing [ref]. 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 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 $D_n = \{\boldsymbol{x}_i, y_i, i = 1, \ldots, n\}$.

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 2 provides pseudo-code to implement a basic bayesian optimization module.

Algorithm 2 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** $n = 1, 2, \ldots$ **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.

$$\alpha_{\mathrm{PI}}(\boldsymbol{x}, \mathcal{D}_n) = P(\nu > \tau) = \Phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) \tag{3.4}$$

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 [ref] 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:

$$I(\boldsymbol{x}, \nu, \theta) = (\nu - \tau)\mathbb{I}(\nu > \tau) \tag{3.5}$$

Taking the expectation yields the expected improvement acquisition function:

$$\alpha_{\mathrm{EI}}(\boldsymbol{x}, \mathcal{D}_n) = \mathbb{E}\left[I(\boldsymbol{x}, \nu, \theta)\right] = (\mu_n(\boldsymbol{x}) - \tau)\Phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) + \sigma_n(\boldsymbol{x})\phi\left(\frac{\mu_n(\boldsymbol{x}) - \tau}{\sigma_n(\boldsymbol{x})}\right) \tag{3.6}$$

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 [ref] and proven convergence rates for [ref]. 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.[ref]

3.3.2 Optimistic policies

Optimistic acquisition functions have their origins in the multi-armed bandit setting [ref]. 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) [ref], 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:

$$\alpha_{\text{UCB}}(\boldsymbol{x}, \mathcal{D}_n) = \mu_n(\boldsymbol{x}) + \beta_n \sigma_n(\boldsymbol{x}) \tag{3.7}$$

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. [ref]. 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:

$$\boldsymbol{x}_{n+1} \sim p_*(\boldsymbol{x}|\mathcal{D}_n) \tag{3.8}$$

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 [ref]. We could define formally this acquisition function:

$$\alpha_{\text{TS}}(\boldsymbol{x}, \mathcal{D}_n) = f^{(n)}(\boldsymbol{x}) \tag{3.9}$$

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[ref]. Another new approach is entropy-based [ref]. 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:

$$\alpha_{\text{ES}}(\boldsymbol{x}|\mathcal{D}_n) = H(\boldsymbol{x}^*|\mathcal{D}_n) - \mathbb{E}_{y|\mathcal{D}_n,\boldsymbol{x}}\left[H(\boldsymbol{x}^*|\mathcal{D}_n \cup \{(\boldsymbol{x}, y)\})\right] \tag{3.10}$$

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 [ref] or a space discretization of $\mathcal{A}$. A recent paper [ref] introduced *predictive entropy search* (PES), a method to remove the need for discretization by rewriting Equation 3.10 as:

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

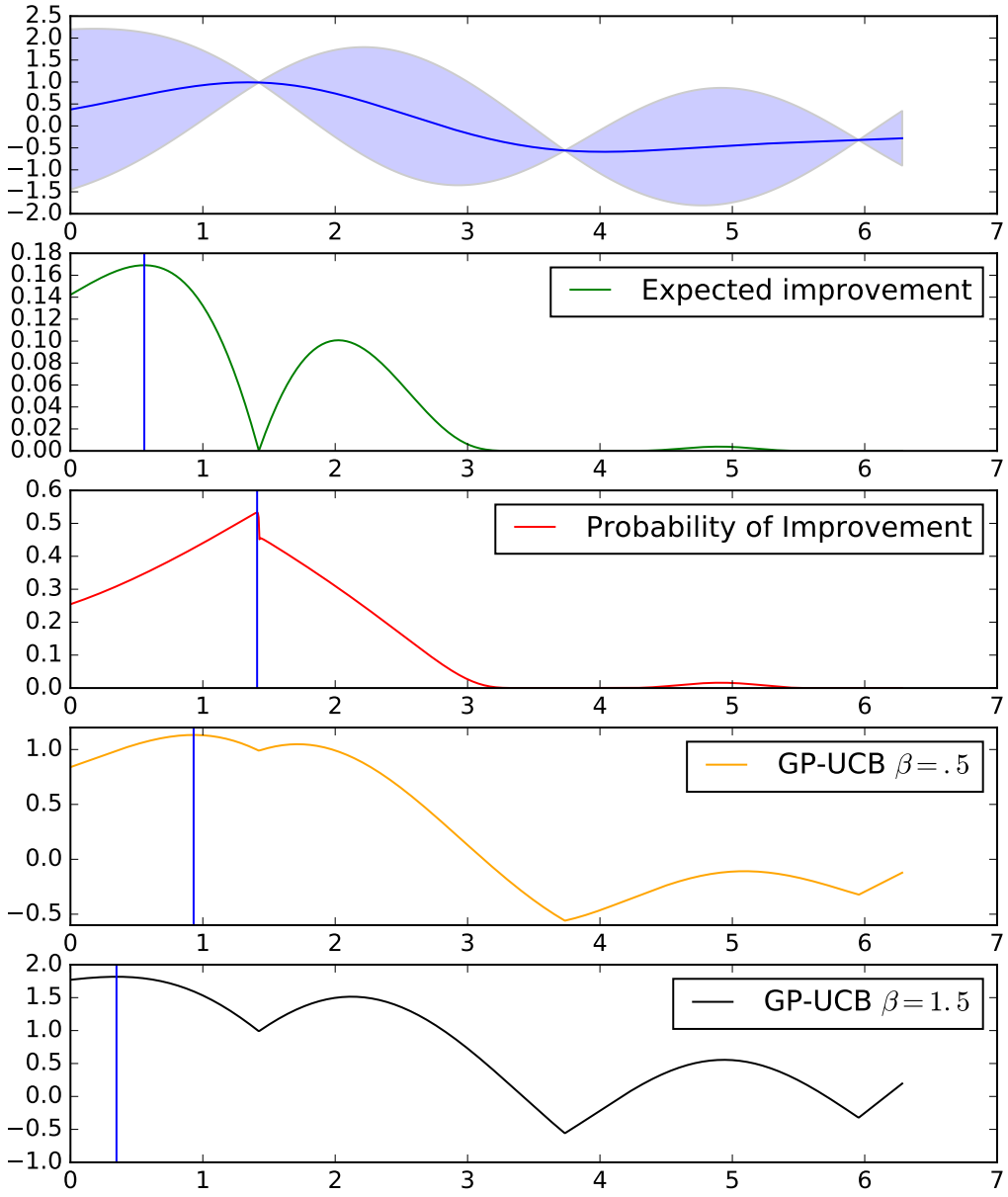
The expectation is approximated in the original paper by Monte Carlo with Thompson samples, with simplifying assumptions. This is arguably the state of the art in acquisition functions, according to the results reported in [ref].

3.3.4 Example: 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 below produces Figure 3.1.

```
import numpy as np
from GPRegressor import GPRegressor
from acquisition import Acquisition
from covfunc import squaredExponential
from 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 = GPRegressor(sexp)
                gpgo = GPGO(gp, acq, f, param)
                gpgo._firstRun(n_eval = 3)
                plotGPGO(gpgo, param, index = index + 2, new = new)
                new = False

        plt.show()
```

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.



Chapter 4

Experiments

Chapter 5

pyGPGO: A simple Python Package for Bayesian Optimization