BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits

Ruben Martinez-Cantin  
Centro Universitario de la Defensa  
Zaragoza, 50090, Spain

Editor: Balázs Kégl

Abstract
BayesOpt is a library with state-of-the-art Bayesian optimization methods to solve nonlinear optimization, stochastic bandits or sequential experimental design problems. Bayesian optimization characterized for being sample efficient as it builds a posterior distribution to capture the evidence and prior knowledge of the target function. Built in standard C++, the library is extremely efficient while being portable and flexible. It includes a common interface for C, C++, Python, Matlab and Octave.

Keywords: Bayesian optimization, efficient global optimization, sequential model-based optimization, sequential experimental design, Gaussian processes

1. Introduction
Bayesian optimization (Mockus, 1989; Brochu et al., 2010) is a special case of nonlinear optimization where the algorithm decides which point to explore next based on the analysis of a distribution over functions $P(f)$, for example a Gaussian process or other surrogate model. The decision is taken based on a certain criterion $C(\cdot)$ called acquisition function. Bayesian optimization has the advantage of having a memory of all the observations, encoded in the posterior of the surrogate model $P(f|D)$ (see Figure 1). Usually, this posterior distribution is sequentially updated using a nonparametric model. In this setup, each observation improves the knowledge of the function in all the input space, thanks to the spatial correlation (kernel) of the model. Consequently, it requires a lower number of iterations compared to other nonlinear optimization algorithms. However, updating the posterior distribution and maximizing the acquisition function increases the cost per sample. Thus, Bayesian optimization is normally used to optimize expensive target functions $f(\cdot)$, which can be multimodal or without closed form. The quality of the prior and posterior information about the surrogate model is of paramount importance for Bayesian optimization, because it can reduce considerably the number of evaluations to achieve the same performance.

2. BayesOpt Library
BayesOpt uses a surrogate model of the form: $f(x) = \phi(x)^T w + \epsilon(x)$, where we have $\epsilon(x) \sim \mathcal{TP}(0, \sigma^2_s (K(\theta) + \sigma^2_n I))$. Here, $\mathcal{TP}(\cdot)$ means a Student-t process or a mixture of Student-t processes, with the Gaussian process as a special case. This model can be considered as a linear regression model $\phi(x)^T w$ with heteroscedastic perturbation $\epsilon(x)$, as a nonparametric
Input: target \( f(\cdot) \), priors \( P(f), P(\theta) \), criterion \( C(\cdot) \), budget \( N \)  
Output: \( x^* \)

Build a data set \( D \) of points \( X = x_1 \ldots x_l \) and its response \( y = y_1 \ldots y_l \) using an initial design.

While \( i < N \):
- Update the distribution with all data available, for example, \( P(f|D) \propto \int P(D|f,\theta)P(f)P(\theta)\,d\theta \)
- Select the point \( x_i \) which maximizes the criterion: \( x_i = \text{arg max} \, C(x|P(f|D)) \). Observe \( y_i = f(x_i) \)
- Augment the data with the new point and response: \( D \leftarrow D \cup \{x_i, y_i\} \) \( i \leftarrow i + 1 \)

Figure 1: General algorithm for Bayesian optimization.

process with nonzero mean function or as a semiparametric model. The library allows to define priors on \( w, \sigma^2 \) and \( \theta \). The marginal posterior \( P(f|D) \) can be computed in closed form, except for the kernel parameters \( \theta \). BayesOpt allows to use different approximations based on empirical Bayes (Santner et al. 2003) or MCMC (Snoek et al., 2012) on \( P(\theta|D) \).

2.1 Implementation

Efficiency has been one of the main objectives during development. We found evidence that updating \( \theta \) every iteration might be unnecessary or even counterproductive (Bull, 2011). For empirical Bayes (ML or MAP of \( \theta \)), we found that a combination of global and local derivative-free methods such as DIRECT (Jones et al., 1993) and BOBYQA (Powell, 2009) marginally outperforms gradient-based method, in CPU time, for optimizing \( \theta \) by avoiding the overhead of computing the marginal likelihood derivative.

One of the most critical components, in terms of computational cost, is the computation of the inverse of the kernel matrix \( K^{-1} \) (Rasmussen and Nickisch, 2010). We compared different numerical solutions and we found that the Cholesky decomposition method outperforms any other method in terms of performance and numerical stability. Furthermore, we can exploit the structure of the Bayesian optimization algorithm in two ways. First, points arrive sequentially. Thus, we can do incremental computations of matrices and vectors, except when the distribution of the kernel parameters \( P(\theta|D) \) is updated. For example, at each iteration, we know that only \( n \) new elements will appear in the correlation matrix \( K \), that is, the correlation of the new point with each of the existing points. The rest of the matrix remains invariant. Thus, instead of computing the whole Cholesky decomposition of \( K \), being \( O(n^3) \), we just add a new row of elements to the Cholesky decomposition, which is \( O(n^2) \). Second, finding the optimal decision at each iteration \( x_i \) requires multiple queries of the acquisition function from the same posterior \( C(x|P(f|D)) \) (see Figure 1). Thus, we can precompute some terms of the posterior and criterion functions that are independent of the query point \( x \). To our knowledge, this is the first software for Gaussian processes/Bayesian optimization that exploits the idea of precomputing terms for multiple queries.

Table 1 compares CPU time (single thread) and accuracy of two different configurations of BayesOpt with respect to other open source libraries. SMAC (Hutter et al., 2011), HyperOpt (Bergstra et al., 2011) and Spearmint (Snoek et al., 2012) used the HPOlib (Eggesperger et al., 2013) timing system, based on runsolver. DiceOptim (Roustant et al., 2012) used R timing system (\texttt{proc.time}). For BayesOpt, standard \texttt{ctime} was used.

Another main objective has been flexibility. The user can easily select among different algorithms, hyperpriors, kernels or mean functions. Currently, the library supports contin-
BayesOpt: A Bayesian Optimization Library

<table>
<thead>
<tr>
<th></th>
<th>Branin (2D)</th>
<th>Camelback (2D)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gap 50 samp.</td>
<td>Gap 200 samp.</td>
</tr>
<tr>
<td>SMAC</td>
<td>0.19444 (0.195)</td>
<td>0.06780 (0.059)</td>
</tr>
<tr>
<td>HyperOpt</td>
<td>0.069499 (0.414)</td>
<td>0.07507 (0.059)</td>
</tr>
<tr>
<td>Spearmint</td>
<td>1.48953 (1.468)</td>
<td><strong>0.00000</strong> (0.000)</td>
</tr>
<tr>
<td>DiceOptim</td>
<td><strong>0.00004</strong> (0.000)</td>
<td>0.00003 (0.000)</td>
</tr>
<tr>
<td>BayesOpt1</td>
<td>1.16844 (1.745)</td>
<td><strong>0.00000</strong> (0.000)</td>
</tr>
<tr>
<td>BayesOpt2</td>
<td>0.04742 (0.116)</td>
<td><strong>0.00000</strong> (0.000)</td>
</tr>
</tbody>
</table>

Table 1: Mean (and standard deviation) of the optimization gap, $f(x_{\text{best}}) - f(x_{\text{opt}})$, and time (in seconds) for 10 runs for different number of samples (including initial design) to illustrate the convergence of each method. ID represent the number of samples of the initial design for each algorithm and problem.

BayesOpt is a library for Bayesian optimization that is highly compatible in many platforms and setups. It has been designed to be highly compatible in many platforms and setups. It has been tested and compiled in different operating systems (Linux, Mac OS, Windows), with

2.2 Compatibility

BayesOpt has been designed to be highly compatible in many platforms and setups. It has been tested and compiled in different operating systems (Linux, Mac OS, Windows), with
different compilers (Visual Studio, GCC, Clang, MinGW). The core of the library is written in C++, however, it provides interfaces for C, Python and Matlab/Octave.

2.3 Using the Library

There is a common API implemented for several languages and programming paradigms. Before running the optimization we need to follow two simple steps:

2.3.1 Target Function Definition

Defining the function that we want to optimize can be achieved in two ways. We can directly send the function (or a pointer) to the optimizer based on a function template. For example, in C/C++:

```cpp
double my_function (unsigned int n_query, const double *query, double *gradient, void *func_data);
```

The gradient has been included for future compatibility. Python, Matlab and Octave interfaces define a similar template function.

For a more object-oriented approach, we can inherit the abstract module and define the virtual methods. Using this approach, we can also include nonlinear constraints in the checkReachability method. This is available for C++ and Python. For example, in C++:

```cpp
class MyOptimization : public bayesopt::ContinuousModel {
public:
    MyOptimization(size_t dim, bopt::params param) : ContinuousModel(dim, param) {}
    double evaluateSample(const boost::numeric::ublas::vector<double> &query) {
        // My function here
    }
    bool checkReachability(const boost::numeric::ublas::vector<double> &query) {
        // My constraints here
    }
};
```

2.3.2 BayesOpt Parameters

The parameters are defined in the bopt_params struct or a dictionary in Python. The details of each parameter can be found in the included documentation. The user can define expressions to combine different functions (kernels, criteria, etc.). All the parameters have a default value, so it is not necessary to define all of them. For example, in Matlab:

```matlab
par.surr_name = 'sStudentTProcessNIG'; % Surrogate model and hyperpriors
% We combine Expected Improvement, Lower Confidence Bound and Thompson sampling
par.crit_name = 'cHedge(cEI, cLCB, cThompsonSampling)';
% Sum of kernels
par.kernel_name = 'kSum(kMaternISO3, kRQISO)';
par.kernel_hp_mean = [1, 1]; par.kernel_hp_std = [5, 5]; % Hyperprior on kernel
par.l_type = 'L_MCMC'; % Method for learning the kernel parameters
par.sc_type = 'SC_MAP'; % Score function for learning the kernel parameters
par.n_iterations = 200; % Number of iterations <=> Budget
par.epsilon = 0.1; % Add an epsilon-greedy step for better exploration
```

Acknowledgments

We would like to thank Luis Montesano for his insightful comments. This work has been supported by Spanish government grant DPI2011-25892 and CUD grant CUD2013-5.
References


