Bayesian Global Optimization (BGO) Package

Saul Toscano-Palmerin

Cornell University

Contents

1 Introduction

Bayesian Global optimization (BGO) package is a Bayesian Global Optimization framework written in Python, developed by Saul Toscano-Palmerin. This package implements Stratified Bayesian Optimization (SBO) (Toscano-Palmerin and Frazier, 2016), Knowledge Gradient (KG) (Frazier *et al.*, 2009), Expected Improvement (EI) (Jones *et al.*, 1998) and Probability Improvement (PI) (Brochu *et al.*, 2010). These procedures are usually used on derivative-free black-box global optimization of expensive noisy or noise-free functions. These procedures are widely used because expectations usually satisfied these characteristics: derivatives are unavailable, and we can only approximate them.

2 Brief Description of the BGO Package

The package can be imported by writing:

```python
from BGO.Source import *
```

We then have to create a Bayesian Global optimization object that includes the objective function, what kernel we want to use, and the directory where the results are saved (please see §2.1 for a complete description of the arguments of the constructors).

```python
stratifiedBayesianOptimizationObject=SBO.SBO(**args)
expectedImprovementObject=EI.EI(**args)
knowledgeGradientObject=KG.KG(**args)
```

We can then optimize our objective function by using those objects.

```python
stratifiedBayesianOptimizationObject.SBOAlg(numberIterations,
                                        nRepeat=10,Train=True)
expectedImprovementObject.EIAlg(numberIterations,nRepeat=10,Train=True)
knowledgeGradientObject.KGAlg(numberIterations,nRepeat=10,Train=True)
```

The input of those functions are the number of iterations of the algorithm, nRepeat (int) is the number of different starting points for optimizing the parameters of the kernel, Train (bool) indicates whether or not if we want to train the kernel.

The output is saved in the directory specified in the Bayesian Global optimization object. Six files are created: XHist.txt, hyperparameters.txt, optAngrad.txt, optVOIgrad.txt, optimalSolutions.txt, optimalValues.txt, varHist.txt and yhist.txt (see Table 1).

2.1 Description of the Arguments of the Constructors

The constructors of the Bayesian Global optimization objects have six arguments.

Table 1: Table with the description of the output files.

XHist.txt	Past points.
yhist.txt	Past observations.
varHist.txt	Variances of the past observations.
hyperparameters.txt	Hyperparameters of the kernel.
optAngrad.txt	Gradient of a_n evaluated at its optimum at each stage of the algorithm.
optimalSolutions.txt	Optimum solutions of a_n at each stage of the algorithm.
optimalValues.txt	Evaluations and variances of the objective, G, at the points of optimalSolutions.txt with their variances.
optVOIgrad.txt	Gradients of the VOI evaluated at its optimum at each stage of the algorithm.

2.1.1 SBO

- Objobj: **Objective object**. This object contains:
  - The simulator of $f(x,w,z)$ given (x,w).
  - A function that gives noisy observations of $F(x,w) = E[f(x,w,z)|w]$.
  - A random or deterministic function to choose points from A.
  - A function that simulates w.
  - A function that gives noisy observations of $E[f(x,w,z)]$. This function is only used to see how well we are doing, but it is not necessary.
- miscObj: **Miscellaneous object**. This object contains:
  - A boolean variance that indicates if the code is run in parallel or not.
  - The path where the output is saved.
  - A random seed.
- optObj: **Optimal object** This object contains:
  - Number of starting points for optimizing VOI and a_n.
  - The functions that transform x and w to their domain (e.g., in some cases we want to optimize the function in a discrete space, but we apply our algorithm in a continuous space, and so it is likely that the optimization methods produce an answer outside of our domain).
  - Method used to optimize VOI ("SLSQP" or "OptSteepestDescent").
  - Method used to optimize a_n ("SLSQP" or "OptSteepestDescent").
  - If we want to use "SLSQP", we have to define the constrains of the problem as a dictionary.
- VOIobj: **Value of Information Function (VOI) object** . This object contains:
  - The function that computes $\nabla_{w_{n+1}} B(x_p, n+1)$
  - Number of training points.
  - The dimension of the domain of x.
  - The points of the discretization of the domain as a numpy array.
- statObj: **Statistical object**. This object contains:

- The kernel (Squared Exponential Kernel if not specified.)
- The training data.
- The function that computes $B(x, x', w') = \int \Sigma_0 (x, w, x', w') f(w) dw$.

- dataObj: **Data object**. This object contains:
  - The training points.

2.1.2 KG

- Objobj: **Objective object**. This object contains:
  - The simulator of $f(x, w, z)$ given (x).
  - A function that gives noisy observations of $E[f(x, w, z)]$.
  - a random or deterministic function to choose points from A.
  - A function that simulates w.
  - A function that gives noisy observations of $E[f(x, w, z)]$ with enough observations to have a small variance. This function is only used to see how well we are doing, but it is not necessary.

- miscObj: **Miscellaneous object**. This object contains:
  - A boolean variance that indicates if the code is run in parallel or not.
  - The path where the output is saved.
  - A random seed.

- optObj: **Optimal object** This object contains:
  - Number of starting points for optimizing VOI and a_n.
  - The functions that transform x and w to their domain (e.g., in some cases we want to optimize the function in a discrete space, but we apply our algorithm in a continuous space, and so it is likely that the optimization methods produce an answer outside of our domain).
  - Method used to optimize VOI (“SLSQP” or “OptSteepestDescent”).
  - Method used to optimize a_n (“SLSQP” or “OptSteepestDescent”).
  - If we want to use “SLSQP”, we have to define the constrains of the problem as a dictionary.

- VOIobj: **Value of Information Function (VOI) object** . This object contains:
  - Number of training points.
  - The dimension of the domain of x.
  - The points of the discretization of the domain as a numpy array.

- statObj: **Statistical object**. This object contains:
  - The kernel (Squared Exponential Kernel if not specified.)
  - The training data.

- dataObj: **Data object**. This object contains:
  - The training points.

2.1.3 EI

- Objobj: **Objective object**. This object contains:
  - The simulator of $f(x,w,z)$ given (x).
  - A function that gives noisy observations of $E[f(x,w,z)]$.
  - a random or deterministic function to choose points from A.
  - A function that simulates w.
  - A function that gives noisy observations of $E[f(x,w,z)]$ with enough observations to have a small variance. This function is only used to see how well we are doing, but it is not necessary.
- miscObj: **Miscellaneous object**. This object contains:
  - A boolean variance that indicates if the code is run in parallel or not.
  - The path where the output is saved.
  - A random seed.
- optObj: **Optimal object** This object contains:
  - Number of starting points for optimizing VOI and a_n.
  - The functions that transform x and w to their domain (e.g., in some cases we want to optimize the function in a discrete space, but we apply our algorithm in a continuous space, and so it is likely that the optimization methods produce an answer outside of our domain).
  - Method used to optimize VOI ("SLSQP" or "OptSteepestDescent").
  - Method used to optimize a_n ("SLSQP" or "OptSteepestDescent").
  - If we want to use "SLSQP", we have to define the constrains of the problem as a dictionary.
- VOIobj: **Value of Information Function (VOI) object** . This object contains:
  - Number of training points.
  - The dimension of the domain of x.
- statObj: **Statistical object**. This object contains:
  - The kernel (Squared Exponential Kernel if not specified.)
  - The training data.
- dataObj: **Data object**. This object contains:
  - The training points.

3 Performance Analysis

This section studies the Python code that runs the SBO algorithm on the New York City's Bike (NYCB) problem. We used the *cProfile* module to collect profilling information. The analysis of the code was done in a Dell R820 with four Intel Xeon E5-4650 2.70GHz 8-core processors, and 768GB of RAM.

3.1 Main computations

At iteration n we need to compute:

- The matrix of covariances of the past observations,

$$A_n = \begin{bmatrix} \Sigma_0 (x_1, w_1, x_1, w_1) & \cdots & \Sigma_0 (x_1, w_1, x_n, w_n) \\ \vdots & \ddots & \vdots \\ \Sigma_0 (x_n, w_n, x_1, w_n) & \cdots & \Sigma_0 (x_n, w_n, x_n, w_n) \end{bmatrix} + \mathrm{diag}\left(\sigma^2 (x_1, w_1), \ldots, \sigma^2 (x_n, w_n)\right),$$

  where $\Sigma_0 \left(x_i, w_i, x_j, w_j\right) = \sigma_0^2 \exp(-\alpha_1 \left\|x_i - x_j\right\|^2 - \alpha_2 \left\|w_i - w_j\right\|^2)$. The complexity is $O(n)$.

- The Cholesky decomposition of A_n, $A_n = LL^T$ (we use the function np.linalg.cholesky). The complexity is $O\left(n^3\right)$.

- For each point x in the discretization of A, we have to compute

$$\begin{aligned} B(x, x_n, w_n) &= \int \Sigma_0 (x, w, x_n, w_n) f(w) \, dw && (1) \\ &\approx \sum_{j=1}^{M} \Sigma_0 \left(x, w_j, x', w'\right) f\left(w_j\right) && (2) \end{aligned}$$

  where f is the Poisson density, and $F(w_M) - F(w_1) = 0.95$ where F is the cumulative Poisson distribution. The complexity is $O(m)$.

- Using linalg.solve_triangular, we solve the system $Lz_1 = B^T$ where

$$B = \begin{pmatrix} B(q_1, x_1, w_1) & \cdots & B(q_1, x_n, w_n) \\ \vdots & \ddots & \vdots \\ B(q_m, x_1, w_1) & \cdots & B(q_m, x_n, w_n) \end{pmatrix}$$

  where the discretization of A is $\{q_i\}_{y=1}^{m}$. The complexity is $O\left(mn^2\right)$.

- Using linalg.solve_triangular, we solve the system $Lz_2 = y - \mu_0$, where $y = (y_1, \ldots, y_n)$ are the past outputs of F. The complexity is $O\left(n^2\right)$.

- We compute the vector $a_n = \mu_0 + z_1^T z_2$. This vector is used to compute VOI, and it is the vector of the posterior means of the GP on G, specifically it is $a^n = (a_n(q_i))_{i=1}^{m}$ where

$$a_n(x) = \mathbb{E}_w [\mu_n (x, w)] = \mathbb{E}_w [\mu_0 (x, w)] + [B(x, 1) \ \cdots \ B(x, n)] A_n^{-1} \begin{pmatrix} y_1 - \mu_0 (x_1, w_1) \\ \vdots \\ y_n - \mu_0 (x_n, w_n) \end{pmatrix}$$

  and $B(x, i) = \int \Sigma_0 (x, w, x_i, w_i) f(w) \, dw$. The complexity is $O(mn)$.

- We have to optimize VOI and a_n. We use both scipy.optimize.fmin_slsqp () and a gradient ascent method.

3.1.1 Main Computations for VOI

For $V_n(x_{n+1}, w_{n+1})$,

- We compute $B_N = B(q, n+1) = \int \Sigma_0(x, w, x_{n+1}, w_{n+1}) f(w)\, dw$ for each point q in the discretization of A. The complexity is $O(m)$.

- We have to compute the vector γ,

$$\gamma = \begin{bmatrix} \Sigma_0(x_{n+1}, w_{n+1}, x_1, w_1) \\ \vdots \\ \Sigma_0(x_{n+1}, w_{n+1}, x_n, w_n) \end{bmatrix}.$$

  The complexity is $O(n)$.

- We have to solve $Lz_3 = \gamma$, and compute $z_3 \cdot z_3$. The complexity is $O\left(n^2\right)$.

- We compute the vector $b = \left(B_N - z_1^T z_3\right) / \sqrt{\left(\Sigma_0(x_{n+1}, w_{n+1}, x_{n+1}, w_{n+1}) - z_3 \cdot z_3\right)}$. The complexity is $O(mn)$.

- Using the Algorithm 1 in (Frazier *et al.*, 2009), we can remove all those entries i for which $a_i + b_i z < \max_{k \neq i} a_k + b_k z$ for all z. Then, this algorithm gives us new vectors a' and b' such that

$$V_n(x_{n+1}, w_{n+1}) \approx \sum_{i=1}^{|a'|-1} \left(b'_{i+1} - b'_i\right) f(-|c_i|),$$

  where

$$\begin{aligned} f(z) &:= \varphi(z) + z\Phi(z), \\ c_i &:= \frac{a'_{i+1} - a'_i}{b'_{i+1} - b'_i}, i = 1, \ldots, \left|a'\right| - 1 \end{aligned}$$

  and φ, Φ are the standard normal cdf and pdf, respectively. The complexity is $O(m)$.

For $\nabla V_n(x_{n+1}, w_{n+1})$,

$$\begin{aligned} \nabla V_n(x_{n+1}, w_{n+1}) &= \sum_{i=1}^{|a'|-1} \left(b'_{i+1} - b'_i\right)\left(-\Phi(-|c_i|)\right)\nabla(|c_i|) - \left(\nabla b'_{i+1} - \nabla b'_i\right) f(-|c_i|) \\ &= \sum_{i=1}^{|a'|-1} \left(\nabla b'_{i+1} - \nabla b'_i\right)\left(-\Phi(-|c_i|)|c_i| - f(-|c_i|)\right) \\ &= \sum_{i=1}^{|a'|-1} \left(-\nabla b'_{i+1} + \nabla b'_i\right)\left(\varphi(|c_i|)\right). \end{aligned}$$

and

$$
\begin{aligned}
\nabla b_i^{'} &= \beta_1 \left(\nabla B\left(q_i^{'}, n+1\right) - \nabla\left(\gamma^T\right) A_n^{-1} \begin{bmatrix} B\left(q_i^{'},1\right) \\ \vdots \\ B\left(q_i^{'},n\right) \end{bmatrix} \right) &(3)\\
&\quad -\frac{1}{2}\beta_1^3 \beta_2 \left[\nabla\Sigma_0\left(x_{n+1}, w_{n+1}, x_{n+1}, w_{n+1}\right) - 2\nabla\left(\gamma^T\right) A_n^{-1}\gamma\right] &(4)
\end{aligned}
$$

where

$$
\begin{aligned}
\beta_1 &= \left[\Sigma_0\left(x_{n+1}, w_{n+1}, x_{n+1}, w_{n+1}\right) - \gamma^T A_n^{-1}\gamma\right]^{-1/2} \\
\beta_2 &= B\left(q_i, n+1\right) - \left[B\left(q_i,1\right) \ \cdots \ B\left(q_i,n\right)\right] A_n^{-1}\gamma \\
\nabla\left(\gamma^T\right) &= \left[\nabla\Sigma_0\left(x_{n+1}, w_{n+1}, x_1, w_1\right) \cdots \nabla\Sigma_0\left(x_{n+1}, w_{n+1}, x_n, w_1\right)\right].
\end{aligned}
$$

The complexity is $O\left(m+nm+n^2\right)$.

So, the complexity to compute $V_n(x_{n+1}, w_{n+1})$ and its gradient is $O\left(m+nm+n^2\right)$.

3.1.2 Main Computations for a_n

For $a_n(x)$,

- We have to compute the vector $(B(x,i))_{i=1}^n$. The complexity is $O(n)$.
- We have to solve $Lz_4 = B$. The complexity is $O(n^2)$.
- $a_n(x) = \mu_0 + z_4 \cdot z_2$. The complexity is $O(n)$.

For $\nabla a_n(x)$,

- Compute the gradient of $(B(x,i))_{i=1}^n$, which is equal to

  $$B(x,i)\left(-2.0 \times \alpha_1 \times (x - x_i)\right).$$

  The complexity is $O(n)$.
- We have to solve $Lz_5 = \nabla (B(x,i))_{i=1}^n$. The complexity is $O(n^2)$.
- $\nabla a_n(x) = z_2 \cdot z_5$. The complexity is $O(n)$.

So, the complexity to compute $a_n(x)$ and its gradient is $O\left(n^2\right)$.

3.1.3 Complexity of the Algorithm

Using the results of the previous section, we have that the complexity of every iteration of the algorithm is $O\left(mn+n^3\right)$, where n is the number of the past points and m is the discretization of A (the domain of the points x). The complexity of the algorithm is $O\left(mn^2+n^4\right)$ if it is run during n iterations.

4 Examples

Please go to https://github.com/toscanosaul/BGO/blob/master/CitiBike/citiBike.pdf to see how the library is used on a a realistic problem, using a queuing simulation based on New York City's Citi Bike system, in which system users may remove an available bike from a station at one location within the city, and ride it to a station with an available dock in some other location within the city.

References

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

Frazier, P., Powell, W., and Dayanik, S. (2009). The knowledge-gradient policy for correlated normal beliefs. *INFORMS journal on Computing*, **21**(4), 599–613.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, **13**(4), 455–492.

Toscano-Palmerin, S. and Frazier, P. (2016). Stratified Bayesian Optimization. *arxiv 1602.02338*.