RoBo Documentation

Release 0.1

José Licón, Joel Kaiser

August 06, 2015

Contents

1 Installation **3**
- 1.1 Dependencies . . . 3
- 1.2 How to install RoBO . . . 3

2 Basic Usage **5**
- 2.1 RoBO in a few lines of code . . . 5
- 2.2 Bayesian optimization with RoBO . . . 5

3 Modules **11**
- 3.1 Task . . . 11
- 3.2 Models . . . 12
- 3.3 Acquisition functions . . . 12
- 3.4 Maximizers . . . 13
- 3.5 Solver . . . 13

4 Advanced **15**
- 4.1 REMBO . . . 15
- 4.2 Bayesian optimization with MCMC sampling of the GP's hyperparameters . . . 15

5 Indices and tables **17**

Contents:

CHAPTER 1

Installation

1.1 Dependencies

- numpy >= 1.7
- scipy >= 0.12
- GPy==0.6.1
- emcee==2.1.0
- matplolib >= 1.3
- direct
- cma

1.2 How to install RoBO

You can install RoBO by cloning the repository and executing the setup script:

```
git clone https://github.com/automl/RoBO
cd RoBO/
for req in $(cat requirements.txt); do pip install $req; done
python setup.py install
```

CHAPTER 2

Basic Usage

2.1 RoBO in a few lines of code

RoBO offers a simple interface such that you can use it as a optimizer for black box function without knowing what's going on inside. In order to do that you first have to define the objective function and the bounds of the configuration space:

```
import numpy as np
from robo.fmin import fmin

def objective_function(x):
    return  np.sin(3 * x) * 4 * (x - 1) * (x + 2)

X_lower = np.array([0])
X_upper = np.array([6])
```

The you can start RoBO with the following command and it will return the best configuration / function value it found:

```
x_best, fval = fmin(objective_function, X_lower, X_upper)
```

2.2 Bayesian optimization with RoBO

RoBO is a flexible modular framework for Bayesian optimization. It distinguishes between different components that are necessary for Bayesian optimization and treats all of those components as modules which allows us to easily switch between different modules and add new-modules:

- *Task*: This module contains the necessary information that RoBO needs to optimize the objective function (for example an interface for the objective function the input bounds and the dimensionality of the objective function)
- *Models*: This is the regression method to model the current believe of the objective function
- *Acquisition functions*: This module represents the acquisition function which acts as a surrogate that determines which configuration will be evaluated in the next step.
- *Maximizers* This module is used to optimize the acquisition function to pick the next configuration

2.2.1 Defining an objective function

RoBo can optimize any function $X \rightarrow Y$ with X as an $N \times D$ numpy array and Y as an $N \times K$ numpy array. Thereby N is the number of points you want to evaluate at, D is the dimension of the input X and K the number of output

dimensions (mostly $K = 1$). In order to optimize any function you have to define a task object that implements the interface BaseTask. This class should contain the objective function and the bounds of the input space.

```
import numpy as np

    from robo.task.base_task import BaseTask

class ExampleTask(BaseTask):

        def __init__(self):
            self.X_lower = np.array([0])
            self.X_upper = np.array([6])
            self.n_dims = 1

        def objective_function(self, x):
            return np.sin(3 * x) * 4 * (x - 1) * (x + 2)

    task = ExampleTask()
```

2.2.2 Building a model

The first step to optimize this objective function is to define a model that captures the current believe of potential functions. The probably most used method in Bayesian optimization for modeling the objective function are Gaussian processes. RoBO uses the well-known GPy library as implementation for Gaussian processes. The following code snippet shows how to use a GPy model via RoBO:

```
import GPy

from robo.models.GPyModel import GPyModel

kernel = GPy.kern.Matern52(input_dim=task_ndims)
model = GPyModel(kernel, optimize=True, noise_variance = 1e-4, num_restarts=10)
```

RoBO offers a wrapper interface GPyModel to access the Gaussian processes in GPy. We have to specify a kernel from GPy library as covariance function when we initialize the model. For further details on those kernels visit GPy. We can either use fix kernel hyperparameter or optimize them by optimizing the marginal likelihood. This is achieved by setting the optimize flag to True.

2.2.3 Creating the Acquisition Function

After we defined a model we can define an acquisition function as a surrogate function that is used to pick the next point to evaluate. RoBO offers the following acquisition functions in the acquisition package:

In order to use an acquisition function (in this case Expected Improvement) you have to pass it the models as well as the bounds of the input space:

```
from robo.acquisition.EI import EI
from robo.recommendation.incumbent import compute_incumbent

acquisition_func = EI(model, X_upper=task.X_upper, X_lower=task.X_lower, compute_incumbent=compute_i
```

Expected Improvement as well as Probability of Improvement need as additional input the current best configuration (i.e. incumbent). There are different ways to determine the incumbent. You can easily plug in any method by giving Expected Improvement a function handle (via compute_incumbent). This function is supposed to return a

configuration and expects the model as input. In the case of EI and PI you additionally have to specify the parameter "par" which controls the balance between exploration and exploitation of the acquisition function.

2.2.4 Maximizing the acquisition function

The last component is the maximizer which will be used to optimize the acquisition function in order to get a new configuration to evaluate. RoBO offers different ways to optimize the acquisition functions such as:

- grid search
- DIRECT
- CMA-ES
- stochastic local search

Here we will use a simple grid search to determine the configuration with the highest acquisition value:

```
from robo.maximizers.grid_search import GridSearch

maximizer = GridSearch(acquisition_func, task.X_lower, task.X_upper)
```

2.2.5 Putting it all together

Now we have all the ingredients to optimize our objective function. We can put all the above described components in the BayesianOptimization class

```
from robo.solver.bayesian_optimization import BayesianOptimization

bo = BayesianOptimization(acquisition_fkt=acquisition_func,
                          model=model,
                          maximize_fkt=maximizer,
                          task=task)
```

Afterwards we can run it by:

```
bo.run(num_iterations=10)
```

2.2.6 Saving output

You can save RoBO's output by passing the parameters 'save_dir' and 'num_save'. The first parameter 'save_dir' specifies where the results will be saved and the second parameter 'num_save' after how many iterations the output should be saved.

```
bo = BayesianOptimization(acquisition_fkt=acquisition_func,
                          model=model,
                          maximize_fkt=maximizer,
                          task=task)
                          save_dir="path_to_directory",
                          num_save=1)
```

RoBO will save then the following information:

- X: The configuration it evaluated so far
- y: Their corresponding function values
- incumbent: The best configuration it found so far

- incumbent_value: Its function value
- time_function: The time each function evaluation took
- optimizer_overhead: The time RoBO needed to pick a new configuration

2.2.7 Implementing the Bayesian optimization loop

This example illustrates how you can implement the main Bayesian optimization loop by yourself:

```
import GPy
import matplotlib.pyplot as plt
import numpy as np

from robo.models.GPyModel import GPyModel
from robo.acquisition.EI import EI
from robo.maximizers.grid_search import GridSearch
from robo.recommendation.incumbent import compute_incumbent
from robo.task.base_task import BaseTask


# The optimization function that we want to optimize. It gets a numpy array with shape (N,D) where N
class ExampleTask(BaseTask):
    def __init__(self):
        X_lower = np.array([0])
        X_upper = np.array([6])
        super(ExampleTask, self).__init__(X_lower, X_upper)

    def objective_function(self, x):
        return np.sin(3 * x) * 4 * (x - 1) * (x + 2)

task = ExampleTask()

# Defining the method to model the objective function
kernel = GPy.kern.Matern52(input_dim=task.n_dims)
model = GPyModel(kernel, optimize=True, noise_variance=1e-4, num_restarts=10)

# The acquisition function that we optimize in order to pick a new x
acquisition_func = EI(model, X_upper=task.X_upper, X_lower=task.X_lower, compute_incumbent=compute_in


# Set the method that we will use to optimize the acquisition function
maximizer = GridSearch(acquisition_func, task.X_lower, task.X_upper)


# Draw one random point and evaluate it to initialize BO
X = np.array([np.random.uniform(task.X_lower, task.X_upper, task.n_dims)])
Y = task.evaluate(X)

# This is the main Bayesian optimization loop
for i in xrange(10):
    # Fit the model on the data we observed so far
    model.train(X, Y)

    # Update the acquisition function model with the retrained model
    acquisition_func.update(model)

    # Optimize the acquisition function to obtain a new point
```

```
    new_x = maximizer.maximize()

    # Evaluate the point and add the new observation to our set of previous seen points
    new_y = task.objective_function(np.array(new_x))
    X = np.append(X, new_x, axis=0)
    Y = np.append(Y, new_y, axis=0)
```

CHAPTER 3

Modules

3.1 Task

In order to optimize any function, RoBO expects a task object that is derived from the BaseTask class. If you want to optimize your own objective function you need to derive from this base class and implement at least the objective_function(self, x): method as well as the self.X_lower and self.X_upper. However you can add any additional information here. For example the well-known synthetic benchmark function Branin would look like:

```
import numpy as np

from robo.task.base_task import BaseTask


class Branin(BaseTask):

    def __init__(self):
        self.X_lower = np.array([-5, 0])
        self.X_upper = np.array([10, 15])
        self.n_dims = 2
        self.opt = np.array([[-np.pi, 12.275], [np.pi, 2.275], [9.42478, 2.475]])
        self.fopt = 0.397887

    def objective_function(self, x):
        y = (x[:, 1] - (5.1 / (4 * np.pi ** 2)) * x[:, 0] ** 2 + 5 * x[:, 0] / np.pi - 6) ** 2
        y += 10 * (1 - 1 / (8 * np.pi)) * np.cos(x[:, 0]) + 10

        return y[:, np.newaxis]
```

In this case we can also set the known global optimas and the best function value. This allows to plot the distance between the best found function value and the global optimum. However, of course for real world benchmark we do not have this information so you can just drop them.

Note that the method objective_function(self, x) expects a 2 dimensional numpy array and also returns a two dimension numpy array. Furthermore bounds are also specified as numpy arrays:

```
self.X_lower = np.array([-5, 0])
self.X_upper = np.array([10, 15])
```

3.2 Models

The model class contains the regression model that is used to model the objective function. To use any kind of regression model in RoBO it has to implement the interface from them BaseModel class. Also each model has its own hyperparameters (for instance the type of kernel for GaussianProcesses). Here is an example how to use GPs in RoBO:

```
import GPy
from robo.models.GPyModel import GPyModel

kernel = GPy.kern.Matern52(input_dim=task.n_dims)
model = GPyModel(kernel, optimize=True, noise_variance=1e-4, num_restarts=10)
model.train(X, Y)
mean, var = model.predict(X_test)
```

3.3 Acquisition functions

The role of an acquisition function in Bayesian optimization is to compute how useful it is to evaluate a candidate x. In each iteration RoBO maximizes the acquisition function in order to pick a new configuration which will be then evaluated. The following acquisition functions are currently implemented in RoBO and each of them has its own properties.

- Expected Improvement
- Log Expected Improvement
- Probability of Improvement
- Entropy
- EntropyMC
- Upper Confidence Bound

Each acquisition function expects at least a model and a the input bounds of the task as input, for example:

```
acquisition_func = EI(model, X_upper=task.X_upper, X_lower=task.X_lower)
```

Furthermore, every acquisition functions has its own individual parameters that control its computations. To compute now the for a specific x its acquisition value you can call. The input point x has to be a $1 \times D$ numpy array:

```
val = acquisition_func(x)
```

If you marginalize over the hyperparameter of a Gaussian Process via the GPyMCMC module this command will compute the sum over the acquisition value computed based on every single GP

Some acquisition functions allow to compute gradient, you can compute them by:

```
val, grad = acquisition_func(x, derivative=True)
```

If you updated your model with new data you also have to update you acquisition function by:

```
acquisition_func.update(model)
```

3.4 Maximizers

The role of the maximizers is to optimize the acquisition function in order to find a new configuration which will be evaluated in the next iteration. All maximizer have to implement the BaseMaximizer interface. Ever maximizer has its own parameter (see here for more information) but all expect at least an acquisition function object as well as the bounds of the input space:

```
maximizer = CMAES(acquisition_func, task.X_lower, task.X_upper)
```

Afterwards you can easily optimize the acquisition function by:

```
x_new = maximizer.maximize()
```

3.5 Solver

The solver module represents the actual Bayesian optimizer. The standard module is BayesianOptimization which implements the vanilla BO procedure.

```
bo = BayesianOptimization(acquisition_fkt=acquisition_func,
                  model=model,
                  maximize_fkt=maximizer,
                  task=task,
                  save_dir=os.path.join(save_dir, acq_method + "_" + max_method, "run_" + str(run_id)
                  num_save=1)

bo.run(num_iterations)
```

If you just want to perform one single iteration based on some given data to get a new configuration you can call:

```
new_x = bo.choose_next(X, Y)
```

It also offers functions to save the output and measure the time of each function evaluation and the optimization overhead. If you develop a new BO strategy it might be a good idea to derive from this class and uses those functionalities to be compatible with RoBO's tools.

CHAPTER 4

Advanced

4.1 REMBO

Random EMbedding Bayesian Optimization (REMBO) tackles the problem of high dimensional input spaces with low effective dimensionality. It creates a random matrix to perform a random projection from a high dimensional space into a smaller embedded subspace (rembo-paper). If you want to use REMBO for you objective function you just have to derive from the REMBO task an call its __init__() function in the constructor:

```
class BraninInBillionDims(REMBO):
    def __init__(self):
        self.b = Branin()
        X_lower = np.concatenate((self.b.X_lower, np.zeros([999998])))
        X_upper = np.concatenate((self.b.X_upper, np.ones([999998])))
        super(BraninInBillionDims, self).__init__(X_lower, X_upper, d=2)

    def objective_function(self, x):
        return self.b.objective_function(x[:, :2])
```

Afterwards you can simply optimize your task such as any other task. It will then automatically perform Bayesian optimization in the lower embedded subspace to find a new configuration. To evaluate a configuration it will be transformed back to the original space.

```
task = BraninInBillionDims()
kernel = GPy.kern.Matern52(input_dim=task.n_dims)
model = GPyModel(kernel, optimize=True, noise_variance=1e-3, num_restarts=10)
acquisition_func = EI(model, task.X_lower, task.X_upper, compute_incumbent)
maximizer = CMAES(acquisition_func, task.X_lower, task.X_upper)
bo = BayesianOptimization(acquisition_fkt=acquisition_func,
                          model=model,
                          maximize_fkt=maximizer,
                          task=task)

bo.run(500)
```

4.2 Bayesian optimization with MCMC sampling of the GP's hyperparameters

So far we optimized the GPy's hyperparameter by maximizing the marginal loglikelihood. If you want to marginalise over hyperparameter you can use the GPyModelMCMC module:

```
kernel = GPy.kern.Matern52(input_dim=branin.n_dims)
model = GPyModelMCMC(kernel, burnin=20, chain_length=100, n_hypers=10)
```

It used the HMC method implemented in GPy to sample the marginal loglikelihood. Afterwards you can simply plug it into your acquisition functions

```
acquisition_func = EI(model, X_upper=branin.X_upper, X_lower=branin.X_lower, compute_incumbent=comput

maximizer = Direct(acquisition_func, branin.X_lower, branin.X_upper)
bo = BayesianOptimization(acquisition_fkt=acquisition_func,
                  model=model,
                  maximize_fkt=maximizer,
                  task=task)

bo.run(10)
```

RoBO will then compute an marginalised acquistion value by computing the acquisition value based on each single GP and sum over all of them.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search