1 Introduction

This is a release of the code used to produce the results in [1], which has since been extended to a full implementation of both Bayesian optimization and constrained Bayesian optimization. Bayesian optimization is a relatively new method for optimizing expensive-to-evaluate blackbox functions, and has seen use in machine learning in the context of hyperparameter optimization. For an excellent overview of Bayesian optimization, see [3] and [4].

2 Bayesian Optimization

While the primary topic of [1] is Bayesian optimization for constrained problems, this toolbox is also a fully-featured implementation of Bayesian optimization for unconstrained problems as well. Both the BO and CBO code are called using the single `bayesopt` function, which has two input arguments and three ouput arguments. Most of the GP inference is performed using the toolbox associated with the book, Gaussian Processes for Machine Learning [2].

```
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

In the following sections, we go over each of these arguments in detail.

2.1 Output Arguments

We first briefly discuss the output arguments, as they are relatively simple. The first output argument, `min_sample` is the parameter setting that minimized the objective function, and `min_value` is the value of the objective function at that point.

2.1.1 Trace

Bayesopt additionally returns a third output argument, a trace. This is a struct that contains three elements when running standard Bayesian Optimization, and four when running with constraints:

```
botrace.samples;     % Matrix of parameter values tested by bayesopt.
botrace.values;      % Objective value for each sample above
botrace.con_values;  % Constraint function values for samples above
botrace.times;       % Time to evaluate objective function on each sample
```

`botrace.samples` is a list of all parameter values run by bayesopt. For example, if 100 iterations of BO are run to optimize a function with 5 parameters, `trace.samples` will be a 100×5 matrix. Similarly, `trace.values` are the objective function values observed by bayesopt when running the parameter settings in `trace.samples.` Thus in this same scenario, `trace.values` would be a 100×1 vector. `trace.times` simply contains the running time of each function evaluation (and therefore would also be a 100×1 vector in our example). Finally, when running with constraints, `trace.con_values` contains a vector of constraint values for each parameter setting run. For example, if the 100 iterations above were run to optimize a function subject to three constraints, `trace.con_values` would be a 100×3 matrix.

Importantly, this trace can be saved to a file either by hand or automatically using the `save trace` option. This allows a run of bayesopt that was halted to be resumed, using the information in a trace file as initial information. Alternatively, if you already have function evaluations for a number of experiments, you can create a trace file manually to run bayesopt starting with these experiments.

2.2 Input Arguments

2.2.1 Objective function F

The first argument to bayesopt, F, is a MATLAB function handle. Broadly, the function handle takes a single argument–a vector of parameters–and returns a single output–the objective function value. As a running example, in [1], one of the simulation functions used was the simple function:

$$\ell(\mathbf{x}) = \sin(x_1) + x_2$$

After setting some options (discussed in the next section), this function could be optimized using bayesopt as follows:

```
% Assume opt was specified above
>> F = @(x) sin(x(1)) + x(2);
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

It's worth noting that while the *function handle* can take only one input (the parameter vector), the function being bound has no such requirement. For example, we might add a constant period to the function above:

$$\ell(\mathbf{x}) = \sin(c \cdot x_1) + x_2$$

We could then bind our function to use a pre-specified value of c as follows:

```
% Assume opt was specified above
>> c = 2.5;
>> F = @(x) sin(c*x(1)) + x(2);
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

This is particularly useful when tuning hyperparameters of machine learning algorithms, as it allows us to pass in datasets. For example, suppose we have a function to train an algorithm on a dataset (X_train,Y_train) and return the validation error on a validation set (X_val,Y_val) using the hyperparams specified by hyper_params:

```
function [validation_error] = train(hyper_params,X_train,Y_train,X_val,Y_val)
```

Then finding the best hyperparameters is as simple as creating a function handle that takes only the hyper parameter values:

```
>> load data.mat; % Get X_train, Y_train, X_val, Y_val
>> F = @(X) train(X,X_train,Y_train,X_val,Y_val);
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

2.2.2 Options

The second input argument to bayesopt is the option struct `opt` that defines a number of important options–some optional, others required–for bayesopt. For your convenience, many of the required arguments can be given default options by using the "defaultopt" function:

```
>> opt = defaultopt;
```

However, we discuss all the options in detail here.

Required Options. We first discuss all required arguments: those arguments that must have values to run bayesopt.

opt.dims - Specifies the number of parameters your objective function takes. In our simulation example, $\ell(\mathbf{x}) = \sin(x_1) + x_2$, this would be set to 2.

opt.mins - The minimum value of each parameter to search for. Together with opt.maxes, this defines a hypercube that bayesopt should look for the optimum in.

opt.maxes - The maximum value of each parameter to search for.

opt.max_iters - Number of iterations of Bayesian Optimization to perform. Defaults to 100.

opt.grid_size - Number of candidate points to densely sample in the hypercube defined by opt.mins and opt.maxes. These points are sampled from a Sobol sequence to cover the space as much as possible. Note that for higher dimensional problems, it may be better to use a smaller grid size, but set `opt.optimize_ei` to true, rather than use a very large grid size. Defaults to 20000.

opt.meanfunc - Which mean function (in GPML) to use for GP inference. Defaults to @meanconst.

opt.covfunc - Which covariance function (in GPML) to use for GP inference. Defaults to @covSEard.

opt.hyp - GP hyperparameters. Defaults to -1, which means to set automatically using MLE.

Note that while it may seem like there are a lot of these parameters, remember that many of these can be set to the default values and will work reasonably well. For example, to optimize our simulation function above, the following is all that is required:

```
>> F = @(x) sin(x(1)) + x(2);
>> opt = defaultopt;    % Sets max_iters, grid_size, meanfunc, covfunc, hyp
>> opt.dims = 2;        % 2 Parameters
>> opt.mins = [0 0];    % Min value of x(1) and x(2)
>> opt.maxes = [6 6];   % Max value of x(1) and x(2)
>> opt.max_iters = 25; % Overwrite default -- won't need 100 iterations.
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

Additional Options. In addition to the above, there are a number of options that are provided either for convenience, or directly modify the optimization.

opt.optimize_ei - If set to 1, this uses the derivative of Expected Improvement (EI) or Expected Constrained Improvement (EIC) to, for each candidate point in the grid, find the nearest EI or EIC peak. Note that, while this is very expensive, it does have several advantages. First, `opt.grid_size` can (and should) be set to a much smaller value, as a dense sampling is no longer as necessary. Second, using this option is highly recommended for higher dimensional search problems, as optimizing EI or EIC directly is more efficient than sampling a high dimensional space densely.

- **Currently, only the covSEard (Squared Exponential covariance with automatic relevance determination) covariance is supported with this option. Support for other covariances will be added in a later release.**
- However, if you so choose, it is easy to extend bayesopt.m to support other covariances. Simply write a matlab function that returns the derivative of the covariance $k(\mathbf{x}, \mathbf{z})$ with respect to a single $\mathbf{z}$ and pass it in as `opt.cov_grad_f`

opt.parallel_jobs - Setting this option to a value greater than 1 will use the method of [3] to run multiple experiments in parallel. To run experiments in parallel, bayesopt makes use of the builtin `parfor` construct in MATLAB. as a result, before running with this option, one of the following two lines should be run (to use 4 cores in this example):

```
>> parpool 4;      % Matlab 2014b and beyond
>> matlabpool 4; % Matlab before 2014b
```

Notes. There are a few notes about the above two options specifically. First, they are currently mutually exclusive, as combining the two is very expensive. Second, while the `parallel_jobs` option, requires a matlabpool, the `optimize_ei` option also benefits greatly from this, as it will then optimize EI on each candidate in parallel.

opt.ei_burnin - Specifies a number of iterations to do without optimizing EI before optimizing EI. Because the optimization of EI and EIC are very expensive, this can speed up optimization, as optimizing EI may not be worth it for a few iterations.

opt.parallel_mc_iters - Number of MCMC samples to obtain for the parallelization strategy of [2]. The higher this is set, the better the parallel experiments will be, but the more expensive choosing them will be.

opt.save_trace - Set to 1 to automatically save a trace (the struct returned by bayesopt) to a file. This trace is saved at the end of each iteration of BO, and can be used to resume a run of BO.

opt.trace_file - a string containing a file name to save the trace to if `save_trace` is set to 1.

opt.resume_trace - Rather than starting a new bayesopt run from scratch, loads the trace from `opt.trace_file` and uses that information for initialization.

opt.grid - Used to specify a custom grid of points to optimize over. This is useful if you want to optimize your function over a discrete set of candidates, rather than an entire hypercube. Note that using this in conjunction with `optimize_ei` will cause the final parameter setting to not be an element of this grid.

opt.do_cbo - Perform constrained optimization if set to 1 (default is 0). Note that all of the above options function for constrained BO. In the specific case of the `optimize_ei` option, EIC is optimized starting from each candidate rather than EI.

opt.lt_const - Row vector of constraint constants for each constraint function. That is, the optimization will be subject to the constraint that the jth constraint function be less than or equal to `opt.lt_const(j)`.

3 Constrained Bayesian Optimization

To use bayesopt to perform constrained optimization requires two simple modifications. First, the option `opt.do_cbo` must be set to 1. Second, the constraint constant must be set in `opt.lt_const`. Finally, the function handle F must return two arguments instead of one. The first argument is still the objective function value as normal. The second output argument is a row vector of constraint function values, one for each constraint function. For example, to solve the following simple simulation constrained problem:

$$\min_{x_1,x_2} \sin(x_1) + x_2 \text{ subject to } \sin(x_1)\sin(x_2) \leq -0.95$$

One could write the following function:

```
function [L,C] = sample_con_func(x)
        L = sin(x(1)) + x(2);
        C = sin(x(1))*sin(x(2));
end
```

And then run the following:

```
>> F = @(x) sample_con_func(x);
>> opt = defaultopt;      % Sets max_iters, grid_size, meanfunc, covfunc, hyp
>> opt.dims = 2;          % 2 Parameters
>> opt.mins = [0 0];      % Min value of x(1) and x(2)
>> opt.maxes = [6 6];     % Max value of x(1) and x(2)
>> opt.max_iters = 25;    % Overwrite default -- won't need 100 iterations.
>> % Constrained BO options.
>> opt.do_cbo = 1;        % Do constrained optimization
>> opt.lt_const = -0.95;  % Constraint constant.
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

Running with multiple inequality constraints is also straightforward. To optimize the same simulation problem with one additional constraint:

$$\min_{x_1,x_2} \sin(x_1) + x_2$$

$$\text{s.t. } \sin(x_1)\sin(x_2) \leq -0.95 \text{ and } x_2^2 \leq 10$$

You modify the function to return both constraint function values:

```
function [L,C] = sample_con_func2(x)
        L = sin(x(1)) + x(2);
        C1 = sin(x(1))*sin(x(2));
        C2 = x(2)^2;
        C = [C1 C2];
end
```

And then specify both constraint constants:

```
>> F = @(x) sample_con_func2(x);
>> opt = defaultopt;       % Sets max_iters, grid_size, meanfunc, covfunc, hyp
>> opt.dims = 2;           % 2 Parameters
>> opt.mins = [0 0];       % Min value of x(1) and x(2)
>> opt.maxes = [6 6];      % Max value of x(1) and x(2)
>> opt.max_iters = 25;     % Overwrite default -- won't need 100 iterations.
>> % Constrained BO options.
>> opt.do_cbo = 1;         % Do constrained optimization
>> opt.lt_const = [-0.95 10]; % Constraint constant.
>> [min_sample,min_value,botrace] = bayesopt(F,opt);
```

4 References

1. Gardner, J., Kusner, M., Weinberger, K., & Cunningham, J. (2014). Bayesian optimization with inequality constraints. In Proceedings of The 31st International Conference on Machine Learning (pg. 937-945).

2. Rasmussen, C. E., & Williams, C. K. I. (2006). Gaussian processes for machine learning.

3. Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In Advances in Neural Information Processing Systems (pg. 2951-2959).

4. Brochu, E., Cora, V. M., & 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.