San Diego State University

Math 693 A - Project Report

Bayesian Optimization for Machine Learning

Pradeep Singh
MS Student, CSRC
San Diego State University

1 Abstract

This project explores Bayesian optimization techniques for hyperparameter tuning in machine learning algorithms and compares it with different methods like: manual search, grid search, random search. Goal of this project is twofold: 1) To study how bayesian optimization can be used in hyperparameter tuning in order to improve the current methods, and 2) Comprehensive analysis of hyperparameter optimization algorithms in Machine Learning.

2 Introduction

A machine learning model is nothing but a mathematical formula with a number of parameters that need to be learned from the data, this is the crux of machine learning. This is done by fitting a model to the data, also known as model training by optimizing a loss function. In other words, by training a model/ optimizing a loss function with existing data, we are able to learn model parameters.

Learning the best possible (hyper)parameters are very crucial as they define the performance of your model. In order to have good performance many optimization methods has been developed which optimize the hyperparameter of machine learning models.

The most common methods are manual search, grid search, random search, etc. These methods are somewhat inefficient because they do not choose the next parameter to evaluate based on previous results, as a result, they often spend a significant amount of time evaluating "bad" parameter values. This process of learning the best possible values for (hyper)parameters is called hyperparameter tuning or model optimization and it is one of the toughest challenge in machine learning. Entire branches of machine learning and deep learning theory have been dedicated to the optimization of models.

Lately, people from bayesian world have been experimenting with how different bayesian methods can be used in machine learning? A lot of progress has been made with some impressive results, like, Bayesian Optimization, etc. This project explores different bayesian methods for machine learning and compares them with traditional methods.

3 Hyperparameter

A hyperparameter is a special kind of parameters that cannot be directly learned from the regular training process. These parameters express "higher-level" properties of the model such as its complexity or how fast it should learn. They are usually fixed before the actual training process begins. Some examples of hyperparameters are: number of neural network layers, learning rate, depth of a tree, number of clusters in k-means clustering, etc.

4 Hyperparameters Tuning Algorithms

There are bunch of algorithms that are used to fine tune machine learning models. Some of the most used ones are:

4.1 Grid Search

The simplest algorithms that you can use for hyperparameter optimization is a Grid Search. The idea is simple and straightforward. You just need to define a grids of different parameter values, train model for all possible parameter combinations in those grids and select the best one. This method is a good choice only when model can train quickly, which is not the case for many models, especially for neural networks.

Let's assume we need to optimize 5 parameters and for every parameter we will try 10 different values. Therefore, we need to make 100,000 evaluations. Assuming that network trains 10 minutes on average we will have finished hyperparameter tuning in almost 2 years. This is crazy. Typically, network trains much longer and we need to tune more hyperparameters, which means that it can take forever to run grid search for typical neural network.

4.2 Random Search

The idea is similar to Grid Search, but instead of trying all possible combinations we will just use randomly selected subset of the parameters. Instead of trying to check 100,000 samples we can check only 1,000 of parameters. Now, it should take a week to run hyperparameter optimization instead of 2 years. Although, it is one of the best method out there, this could still take a lot of time.

4.3 Manual Search/ Hand-tuning

The straightforward way to tune hyperparameters is based on human experience. Humans manually pick and choose different values for different parameters and select the best one. Humans have ability to learn from previous experience and mistake, which they use to evaluate their choices for parameters when tuning their models. They follow a trial and error process with various configurations of hyperparameters. This process of experimenting with hyperparameters is heuristic and different people with different experience might come up with different settings and the process is not easily reproducible.

4.4 Bayesian Optimization

This methods is inspired by bayesian world, where you encode your prior belief in the model and then as you observed more and more data you change your belief. It resemble the method that we have described above in the Hand-tuning section. It uses a set of previously evaluated parameters and resulting performance (accuracy) to make an assumption about unobserved parameters. I'll describe this more in later section.

5 Optimization

Suppose we have a function $f : X \to \mathbb{R}$ that we wish to minimize on some domain $X \subseteq \chi$. That is, we wish to find,

$$x^* = \arg\min_{x \epsilon X} f(x)$$

This problem is typically called (global) optimization and has been the subject of decades of study. A common approach to optimization problems is to make some assumptions about f. For example, when the objective function f is known to be convex and the domain X is also convex, the problem is known as convex optimization.

If an exact functional form for f is not available, i.e. f behaves as a "black box", What can we do? Such a problem is called black box optimization problems. Evaluation of the such functions are restricted to sampling at a point x and getting a possibly noisy response.

If f is cheap to evaluate we could sample at many points e.g. via grid search, random search or numeric gradient estimation. However, if function evaluation is expensive e.g. tuning hyperparameters of a deep neural network then it is important to minimize the number of samples drawn from the black box function f.

This is the domain where Bayesian optimization techniques are most useful. They attempt to find the global optimum in a minimum number of steps. Bayesian optimization incorporates prior belief about f and updates the prior with samples drawn from f to get a posterior that better approximates f. The model used for approximating the objective function is called surrogate model. It also uses an acquisition function that directs sampling to areas where an improvement over the current best observation is likely.

6 Bayesian Optimization

Bayesian optimization is a derivative-free optimization method and falls in a class of optimization algorithms called *sequential model-based optimization (SMBO)* algorithms [E Brochu, 2010]. These algorithms use previous observations of the loss function, to determine the next (optimal) point to sample function for.

These algorithms keep track of past evaluation results which it use to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function. This model is called a "surrogate" for the objective function and is represented as:

$$p(y|x) \quad \text{OR} \quad p(score|hyperparameters)$$

The surrogate is much easier to optimize than the objective function. Bayesian methods work by finding the next set of hyperparameters to evaluate on the actual objective function by selecting hyperparameters that perform best on the surrogate function. The

aim of Bayesian reasoning is to become "less wrong" with more data which these approaches do by continually updating the surrogate probability model after each evaluation of the objective function.

In other words:

1. Build a surrogate probability model of the objective function.
2. Find the hyperparameters that perform best on the surrogate
3. Apply these hyperparameters to the true objective function.
4. Update the surrogate model incorporating the new results.
5. Repeat steps 2–4 until max iterations or time is reached.

There are two major choice that must be made in Bayesian Optimization. First, one must select a prior over functions that will express assumptions about the function being optimized. Second, we must choose an acquisition function, which is used to construct a utility function from the model posterior, allowing us to determine the next point to evaluate.

7 Surrogate Model

The surrogate model is the probability representation of the objective function built using previous evaluations. There are two most popular forms of the surrogate function, Gaussian Processes and Tree-structured Parzen Estimator [JS Bergstra, 2011].

7.1 Gaussian Processes

A GP is the generalization of a Gaussian distribution to a distribution over functions, instead of random variables. Just as a Gaussian distribution is completely specified by its mean and variance, a GP is completely specified by its mean function $m(x)$, and covariance function $k(x, x^)$.

For a set of data points $x_{1:n} = x_1, \ldots, x_n$, we assume that the values of the loss function $f_{1:n} = f(x_1), \ldots, f(x_n)$ can be described by a multivariate Gaussian distribution

$$f_{1:n} \sim \mathcal{N}(m(\mathbf{x}_{1:n}), \mathbf{K}),$$

where $m(\mathbf{x}_{1:n}) = [m(\mathbf{x}_1), \ldots, m(\mathbf{x}_n)]^T$ and the $n*n$ kernel matrix K has entries given by

$$[K]_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$$

We can think of a GP as a function that, instead of returning a scalar $f(x)$, returns the mean and variance of a normal distribution over the possible values of f at x.

A GP is a popular probability model, because it induces a posterior distribution over the loss function that is analytically tractable. This allows us to update our beliefs of what f looks like, after we have computed the loss for a new set of hyperparameters.

7.2 Tree-structured Parzen Estimator

The methods of SMBO differ in how they construct the surrogate model $p(y|x)$. The Tree-structured Parzen Estimator builds a model by applying Bayes rule. Instead of directly representing $p(y|x)$, it instead uses:

$$p(y|x) = \frac{p(x|y) * p(y)}{p(y)}$$

$p(x|y)$, which is the probability of the hyperparameters given the score on the objective function, in turn is expressed:

$$p(x|y) = \begin{cases} (l(x)) & \text{if } y < y^* \\ (g(x)) & \text{if } y \geq y^* \end{cases}$$

where $y < y^*$ represents a lower value of the objective function than the threshold. The explanation of this equation is that we make two different distributions for the hyperparameters: one where the value of the objective function is less than the threshold, $l(x)$, and one where the value of the objective function is greater than the threshold, $g(x)$.

Intuitively, it seems that we want to draw values of x from $l(x)$ and not from $g(x)$ because this distribution is based only on values of x that yielded lower scores than the threshold. It turns out this is exactly what the math says as well! With Bayes Rule, and a few substitutions, the expected improvement equation (which we are trying to maximize) becomes:

8 Acquisition Function

Acquisition functions propose sampling points in the search space. They trade off exploitation and exploration. Exploitation means sampling where the surrogate model predicts a high objective and exploration means sampling at locations where the prediction uncertainty is high. Both correspond to high acquisition function values and the goal is to maximize the acquisition function to determine the next sampling point.

There are multiple acquisition functions that has been interpreted in bayesian framework. Few of the acquisition functions are: probability of improvement, expected improvement, bayesian expected losses, upper confidence bounds (UCB), Thompson sampling and mixtures of these. Out of all of these, expected improvement seems to be popular one:

$$EI(x) = \mathbb{E}[max\{0, f(x) - f(\hat{x})\}$$

where x is the current optimal set of hyperparameters. Maximising this quantity will give us the point that, in expectation, improves upon f the most.

The nice thing about the expected improvement is, that we can actually compute this expectation under the GP model, by using integration by parts.

$$EI(x) = \begin{cases} (\mu(x)f(\hat{x}))\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$$

$$Z = \frac{\mu(x) - \hat{f(x)}}{\sigma(x)}$$

where $\Phi(z)$, and $\phi(z)$, are the cumulative distribution and probability density function of the (multivariate) standard normal distribution.

This closed form solution gives us some insight into what sort of values will result in a higher expected improvement. From the above, we can derive that;

1. EI is high when the (posterior) expected value of the loss $\mu(x)$ is higher than the current best value $f(\hat{x})$; or
2. EI is high when the uncertainty $\sigma(x)$ around the point x is high.

Intuitively, this makes sense. If we maximize the expected improvement, we will either sample from points for which we expect a higher value of f, or points in a region of f we haven't explored yet $\sigma(x)$ is high). In other words, it trades off exploitation versus exploration.

9 Algorithm

After all this hard work, we are finally able to combine all the pieces together, and formulate the Bayesian optimization algorithm:

1. Given observed values $f(x)$, update the posterior expectation of f using the GP model.
2. Find x_{new} that maximises the EI: $x_{new} = \arg\max EI(x)$.
3. Compute the value of f for the point x_{new}.

This procedure is either repeated for a pre-specified number of iterations, or until convergence.

When looking at the second step, you may notice that we still have to maximize another function, the acquisition function. The nice thing here, is that the acquisition function is a lot easier to optimize than the original loss f. In case of the EI acquisition, we can even compute the derivatives analytically, and use a gradient-based solver to maximise the function.

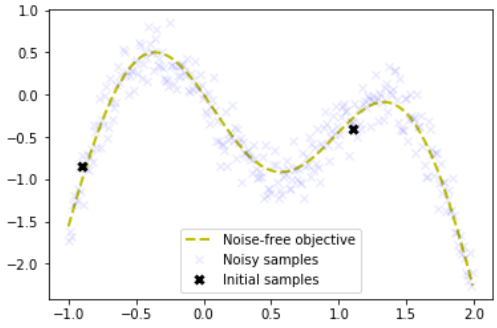
10 Implementation

In the following section, we will implement a simple objective function and it's bayesian optimization in two ways: 1) From scratch using NumPy, SciPy and scikit-learn for Gaussian process, 2) Using GPyOpt and Scikit-optimize, both of which are bayesian optimization libraries.

We treat our function as black box (even though we have it's analytical expression) and iteratively approximate it with a Gaussian process during Bayesian optimization. We also assume that samples drawn from the objective function are noisy and the noise level is given by the noise variable. Optimization is done within given bounds. We also assume that there exist two initial samples in X_{init} and Y_{init}. We will be using expected improvement described in section 8 as our acquisition function. Goal is to find the global optimum in a small number of steps.

The following plot shows the noise-free objective function, the amount of noise by plotting a large number of samples and the two initial samples.

Figure 1: Plotting Objective function



10.1 Implementation from scratch

We use NumPy and SciPy to implement acquisition function (expected improvement) and it's optimization. And, scikit-learn for the gausiaan processes as our surrogate model. Now, we have all components needed to run Bayesian optimization with the algorithm outlined above. The Gaussian process in the following example is configured with a Matérn kernel which is a generalization of the squared exponential kernel or RBF kernel. The known noise level is configured with the alpha parameter.

We run bayesian optimization 10 iterations. In each iteration, a row with two plots is produced. The left plot shows the noise-free objective function, the surrogate function which is the GP posterior predictive mean, the 95% confidence interval of the mean and the noisy samples obtained from the objective function so far. The right plot shows the acquisition function. The vertical dashed line in both plots shows the proposed sampling

point for the next iteration which corresponds to the maximum of the acquisition function.

Figure 2: Plotting Objective function

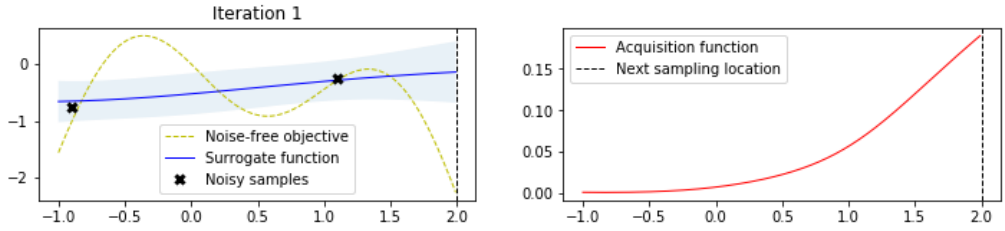


Figure 3: Plotting Objective function

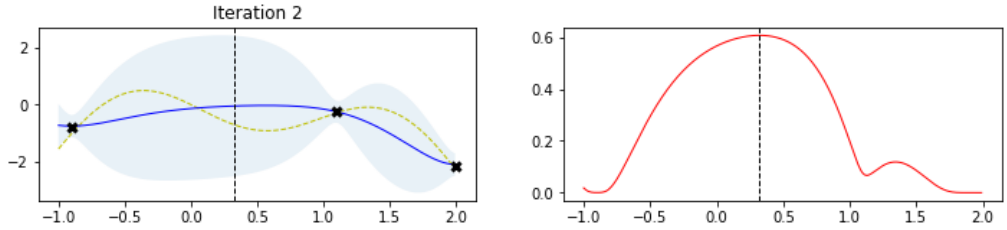


Figure 4: Plotting Objective function

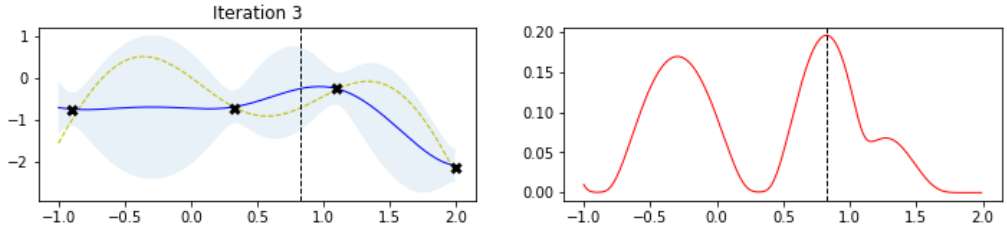


Figure 5: Plotting Objective function

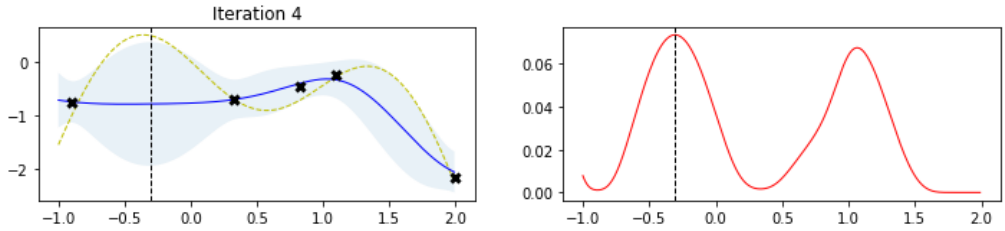


Figure 6: Plotting Objective function

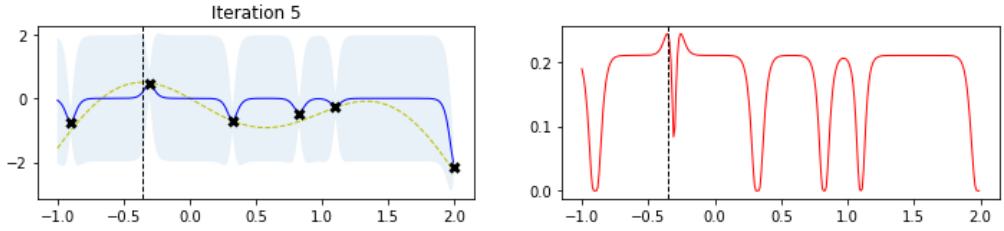


Figure 7: Plotting Objective function

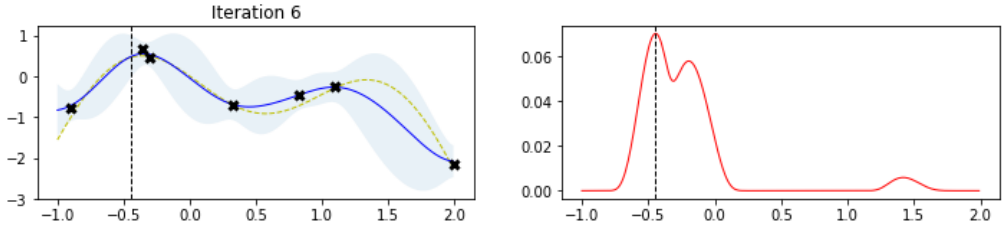


Figure 8: Plotting Objective function

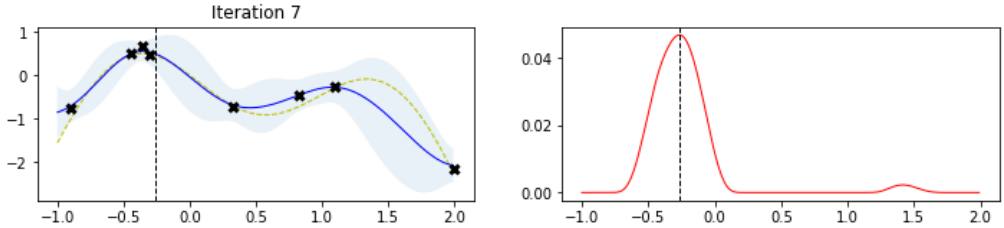


Figure 9: Plotting Objective function

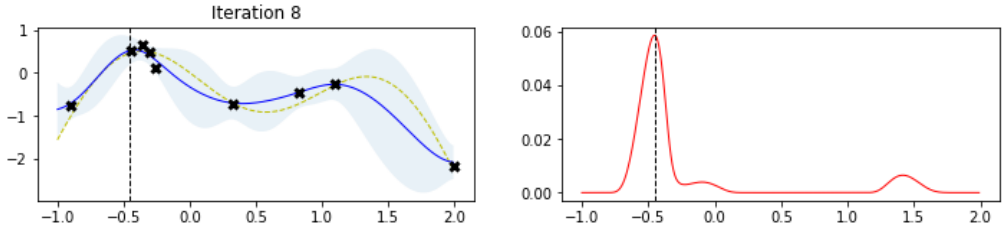


Figure 10: Plotting Objective function

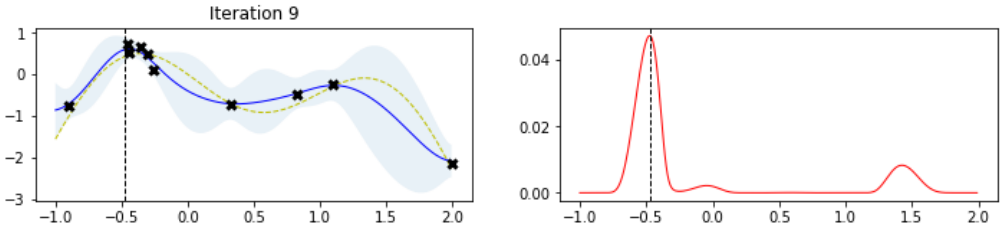
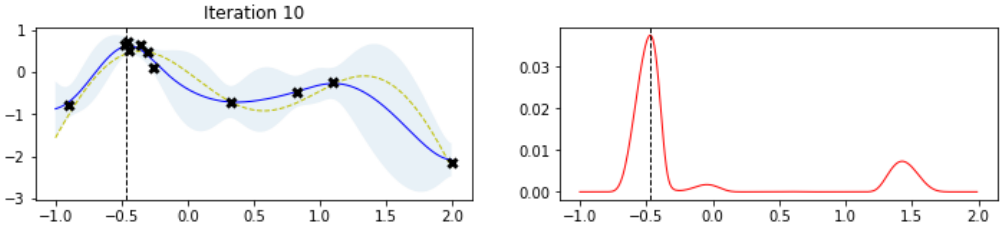


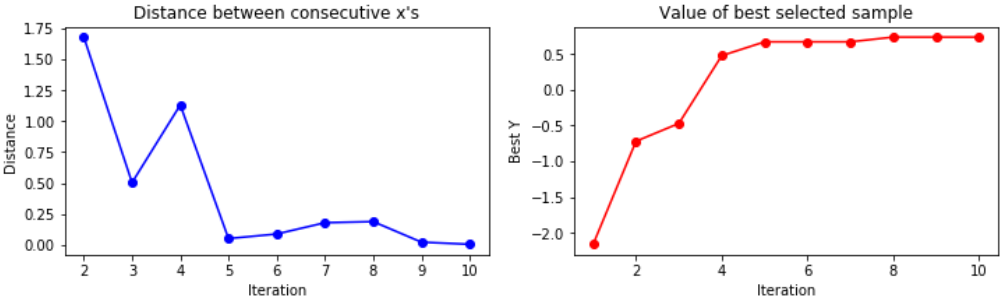
Figure 11: Plotting Objective function



Note how the two initial samples initially drive search into the direction of the local maximum on the right side but exploration allows the algorithm to escape from that local optimum and find the global optimum on the left side. Also note how sampling point proposals often fall within regions of high uncertainty (exploration) and are not only driven by the highest surrogate function values (exploitation).

Below, you can see a convergence plot. You can see how many iterations are needed to find a maximum and if the sampling point proposals stay around that maximum i.e. converge to small proposal differences between consecutive steps.

Figure 12: Convergence Plot



10.2 Implementation using GpyOpt and Scikit-optimize

There are numerous Bayesian optimization libraries out there, GPflow, GPyOpt, Scikit-optimize, BayesOpt, etc. We will stick with GPyOpt and Scikit-optimize.

10.2.1 Scikit-optimize

Scikit-optimize is a library for sequential model-based optimization that is based on scikit-learn. It also supports Bayesian optimization using Gaussian processes. The API is designed around minimization, hence, we have to provide negative objective function values. The results obtained here slightly differ from previous results because of non-deterministic optimization behavior and different noisy samples drawn from the objective function.

Figure 13: Plotting Objective function

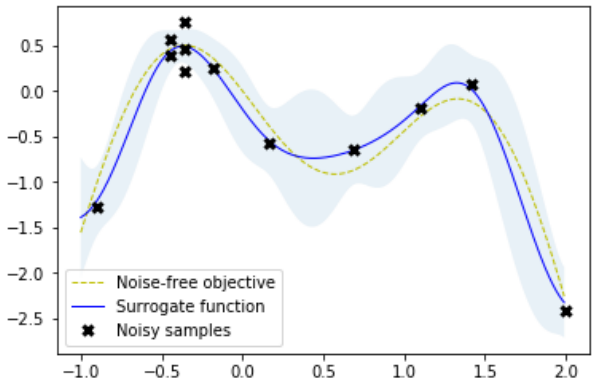
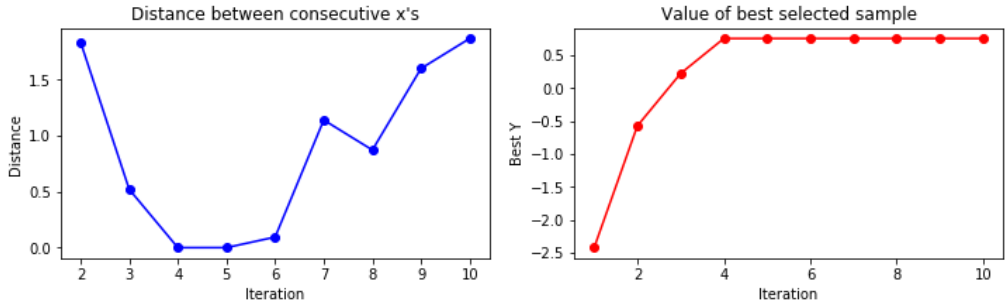


Figure 14: Convergence Plot with Scikit-optimize



10.2.2 GpyOpt

GpyOpt is a bayesian optimization library based on GPy. The abstraction level of the API is comparable to that of scikit-optimize. The Bayesian Optimization API provides

a maximize parameter to configure whether the objective function shall be maximized or minimized (default). Also, the built-in plot_acquisition and plot_convergence methods display the minimization result in any case. Again, the results obtained here slightly differ from previous results because of non-deterministic optimization behavior and different noisy samples drawn from the objective function.

Figure 15: Plotting Objective function

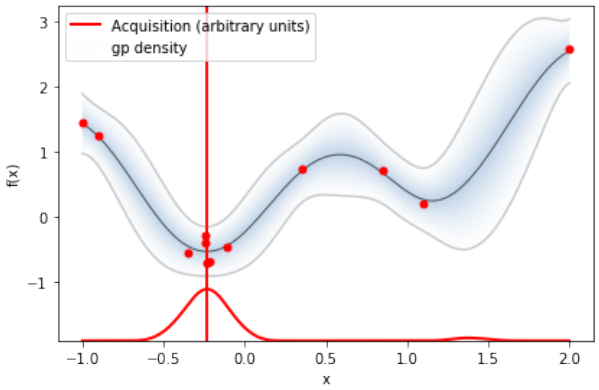
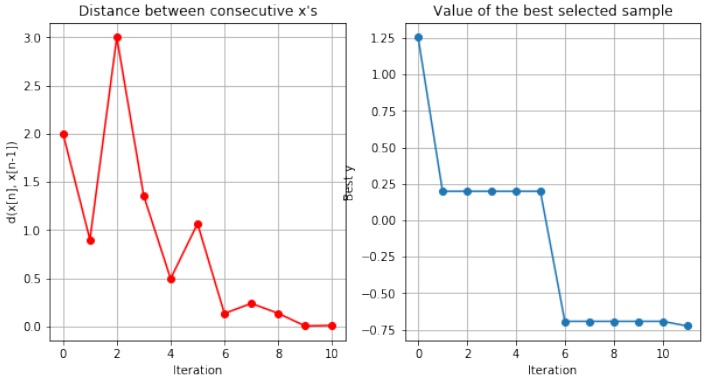


Figure 16: Convergence Plot with GPyOpt



11 Tuning XGBoost Hyperparameters

Now, having seen bayesian optimization in previous sections, let's use it to optimize the hyperparameters of an XGBRegressor (XGBoost) with GPyOpt. We will compare bayesian optimization performance compares to random search. XGBRegressor is part of XGBoost, a flexible and scalable gradient boosting library. XGBRegressor implements the scikit-learn estimator API and can be applied to regression problems. We perform regression on a small toy dataset that is part of scikit-learn.

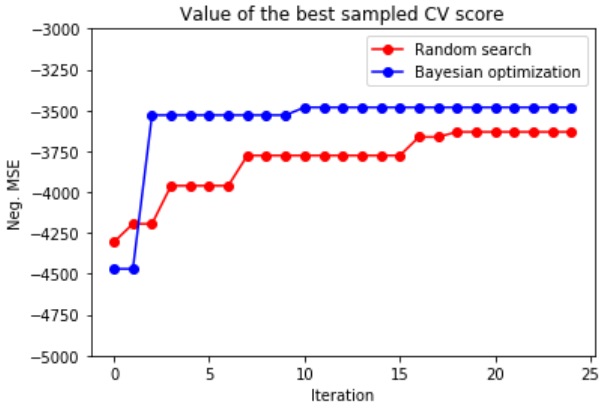
11.1 Random Search

For hyperparameter tuning with random search, we use *RandomSearchCV* of scikit-learn and compute a cross-validation score for each randomly selected point in hyperparameter space.

11.2 Bayesian Optimization

To tune hyperparameters with Bayesian optimization we implement an objective function cv_score that takes hyperparameters as input and returns a cross-validation score. Here, we assume that cross-validation at a given point in hyperparameter space is deterministic and therefore set the exact_feval parameter of BayesianOptimization to True. Depending on model fitting and cross-validation details this might not be the case but we ignore that here.

Figure 17: Plotting Objective function



12 Result

13 Conclusion

On average, Bayesian optimization finds a better optimium in a smaller number of steps than random search and beats the baseline in almost every run. This trend becomes even more prominent in higher-dimensional search spaces. Here, the search space is 5-dimensional which is rather low to substantially profit from Bayesian optimization. One advantage of random search is that it is trivial to parallelize. Parallelization of Bayesian optimization is much harder and subject to research (see [4], for example).

References

1. J. Mockus (2013), Bayesian approach to global optimization: theory and applications. Kluwer Academic.

2. E. Brochu (2010), A Tutorial on Bayesian Optimization of Expensive Cost Functions. arXiv:1012.2599.

3. J. Bergstra (2011), Algorithms for hyper-parameter optimization. Advances in Neural Information Processing Systems, 2011.

4. J. Bergstra (2012), Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research 13 (2012) 281-305

5. Peter I. Frazier (2018), A Tutorial on Bayesian Optimization. arXiv:1807.02811

6. J. Snoek (2012) Practical Bayesian Optimization of Machine Learning Algorithms. Advances in Neural Information Processing Systems, 2012.

7. B. Letham (2018), Efficient tuning of online systems using Bayesian optimization. Facebook AI Research - Bayesian Analysis 2018.

8. R. Jenatton (2017), Bayesian Optimization with Tree-structured Dependencies. PMLR 70:1655-1664, 2017.

9. T. Huijskens (2018), Bayesian optimization with scikit-learn.