HyperSpace Documentation

Release 0.2.0

Todd Young

Jan 10, 2019

History:

HyperSpace is a library for distributed Bayesian hyperparameter optimization.

CHAPTER 1

Hyperparameter Optimization

Machine learning (ML) models often contain numerous hyperparameters, free parameters that must be set before the models can be trained. Optimal settings for these hyperparameters are rarely known a priori, though their settings often dictate our algorithms' ability to learn from data. The challenge of hyperparameter tuning for ML algorithms can be attributed to several factors: (1) heterogeneity in the types of hyperparameters, (2) the potentially complex interactions between hyperparameters, and (3) the computational expense inherit to hyperparameter optimization. Though there has been considerable progress in hyperparameter optimization, optimization in this space remains hard.

As noted by Snoek et al in their introduction to Bayesian optimization for machine learning models *[1]*, hyperparameters are often considered nuisances. However, this belies the fact that hyperparameters are design choices made by ML practitioners. When models consist of many hyperparameters, the differences between models presented across the literature could boil down to small changes in model architectures. If it is reasonable to expect that ML practitioners should be able to defend their choice of one model over others, it is also reasonable to expect that their choices for hyperparameter settings be defensible. It could even be the case that hyperparameter optimization could change a seemingly poor model choice into the best performing model for a given problem. We believe that careful study of our models' hyperparameters allows us to make more informed choices when modeling. It pays to know how our models' behavior changes as we vary our hyperparameters.

As the number of model hyperparameters increases, their optimization becomes significantly more challenging as we face a combinatorial increase in potential model configurations. Similarly, there is an increased chance that our models' hyperparameters interact in complex ways. Modeling these interactions in high-dimensional search spaces quickly becomes challenging and can often defy our intuition. Even experts find it difficult to manually configure hyperparameters *[1]*.

While optimizing model performance with respect to hyperparameters is an obvious goal, we posit that hyperparameter optimization techniques must provide insight into how ML models' behavior changes across a range of possible model configurations. We argue that simply finding an optimal setting for our hyperparameters over a given training and validation set is rather uninformative. We may wonder, how much would our models' performance change if we just slightly perturbed our model hyperparameters? Would a small change in our hyperparameters mean a significant change in our models' behavior? After running through some fixed number of optimization iterations, do we believe that a particular setting of our hyperparameters is unique in that it enables our models to generalize well, or could there be several settings of hyperparameters that perform reasonably well? If we found that there are several good settings of hyperparameters, would they have anything in common? We believe these are important questions and that their answers could lead us to a better understanding of the ML models we employ.

In order to take on these challenges, we developed HyperSpace, a parallel Bayesian model based optimization library.

References

CHAPTER 2

Quickstart

HyperSpace works by parallelizing parameter search spaces, running Bayesian model based optimization (SMBO) over each of these spaces in parallel. It was designed to be as minimially invasive as possible so that you do not have to change much of your existing code to get started. It does not mind which machine learning libraries you choose to use. In fact, it does not mind if you are working with machine learning models. You just need an objective function to be minimized.

Here are the basic steps to get you started:

1. Define an objective function.
2. Define a parameter search space.
3. Call one of HyperSpace's minimization functions, passing it the objective function and search space.

2.1 Step 1: Defining the objective function.

Say you have a machine learning model which has three parameters you would like to optimize. Since in machine learning we care about our models' generalization performance, we would like to find the optimal setting of these parameters which minimizes some error on a validation set. Our objective function then is the result of training our model with some settings of our hyperparameters, tested on some validation set not used to train the model. This objective function would look something like this:

```python
def objective(params: list) -> float:
    param0, param1, param2 = params
    # Instantiate your model with new params
    model = Model(param0, param1, param2)
    train_loss = train(model)
    validation_loss = validate(model)
    return validation_loss
```

And this is how we setup our objective functions for HyperSpace! Note that the objective function will always need one argument; this is a list new hyperparameter settings, one for each of your search dimensions. You won't have to worry about setting this list, it is controlled by the HyperSpace minimization functions. You will only need to worry about initially defining the search space.

2.2 Step 2: Defining the parameter search space.

Setting the parameter search space is really easy. You just have to define a list of tuples. Each tuple consists of the lower and upper bounds for each search dimension respectively. Say our example machine learning algorithm's three parameters are integer, real, and categorical valued respectively. We will let *param0* have a lower bound of *5* and and upper bound of *10*, inclusive. Let *param1* be a float that has a lower bound of *0.01* and an upper bound of *1.0*. Finally, let *param2* be categorical with three options: ('cat0', 'cat1', 'cat2'). We would then define our search space like this:

```
params = [(5, 10), (0.01, 1), ('cat0', 'cat1', 'cat2')]
```

And that's all we need to do. Two quick notes before we move on; Note 1: HyperSpace will do a type check for your parameters. if both of your parameters are integer valued (as is the case in *param0*), then HyperSpace will treat that parameter as integer valued. If either of the parameters are real valued, than that search dimension will be treated as real valued. If any of your parameters are strings, or if your tuple consists of more than two values, then the search space will be considered categorical. Note 2; the ordering of the *params* list passed to the objective function by the HyperSpace minimization methods will be the same as you initially defined your search space. So in our example, you will be guaranteed that the *param0* will correspond to the first tuple in the about *params* list, and so on. All that is left to do then is to call one of HyperSpace's minimization methods!

2.3 Step 3: Calling HyperSpace's minimization functions.

HyperSpace has several methods for running Bayesian optimization. There are two major libraries that handle the SMBO methods: Scikit-Optimize and RoBO. There is also a method for running a distributed version of HyperBand. I will let you take a look into the documentation for these various methods. We have tried to make there arguments as similar as possible, though there are some slight differences. For this example, let's use Scikit-Optimize:

```
from hyperspace import hyperdrive


hyperdrive(objective=objective,
           hyperparameters=params,
           results_path='/path/to/save/results',
           checkpoints_path='/path/to/save/checkpoints',
           model="GP",
           n_iterations=100,
           verbose=True,
           random_state=0)
```

HyperSpace here runs a distributed SMBO optimization using a Gaussian process (model='GP') to model our objective function. This will run for 100 iterations (n_iterations), saving a checkpoint after each iteration (saved to the path specified in checkpoints_path). When in verbose mode (verbose=True), the optimization will print the progress of our MPI rank 0. We can make our results reproducible by setting a random state (here random_state=0). There are several other parameters that can be passed to this function, including whether you would like to use latin hypercube sampling for the initial random draws to warm up the SMBO procedure. You can see more of these in the docstrings of the minimization functions.

Just a quick note on the resources needed to run HyperSpace: we designed this library to takle the exponential scaling problem of Bayesian optimization, which states that the number of samples necessary to bound our uncertainty about the optimization scales exponentially with the number of search dimensions. If we have D dimensions, the number of resources required will be 2^D. So, for our example, we need $2^3 = 8$ MPI ranks.

And that is all we need to get running with HyperSpace! If we were to save this example in as a python module called *example.py*, then we would run it using:

```
mpirun -n 8 python3 example.py
```

I hope this quickstart guide is helpful! If you have any questions or comments, let me know on the HyperSpace's GitHub issues!

-Todd.

CHAPTER 3

Minimal Example: Styblinski-Tang

In the quickstart guide, we went over the three main pieces we need to get up and running with HyperSpace. That gave us a general idea of how the library can be used for machine learning models, but it didn't give us something we can run right out of the box. So this time, let's take a look at a complete, minimal example.

For this, we are going to make use of the Styblinski-Tang function, a commonly used benchmark objective function for optimization methods. The Styblinski-Tang function is usually evaluated on the hypercube $x_i \in [-5., 5.]$ for all $i = 1, 2, \ldots, D$ in D dimensions. It's global minimum $f(x^*) = -39.16599 * D$ which can be found at $x^* = (-2.903534, \ldots, -2.903534)$. We are going to use the two dimensional form of the function, which looks like this:

In its two dimensional form, the Styblinski-Tang function has three local minima in addition to its global minimum. Let's see if we can find the global minimum. The following example example contains everything we need to run the optimization. There are several benchmark functions built into HyperSpace in addition to the Styblinski-Tang function. I would encourage you to try out several of the functions, changing their number of search dimensions. This will give you a sense for how HyperSpace, indeed any optimization library, behaves in various dimensions. Without further ado, let's do this:

```python
import argparse
import numpy as np

from hyperspace import hyperdrive
from hyperspace.benchmarks import StyblinskiTang


def main():
    parser = argparse.ArgumentParser(description='Styblinski-Tang Benchmark')
    parser.add_argument('--ndims', type=int, default=2, help='Dimension of Styblinski-
↪Tang')
    parser.add_argument('--results', type=str, help='Path to save the results.')
    args = parser.parse_args()

    stybtang = StyblinskiTange(args.ndims)
    bounds = np.tile((-5., 5.), (args.ndims, 1))
```

(continues on next page)

(continued from previous page)

```
    hyperdrive(objective=stybtang,
               hyperparameters=bounds,
               results_path=args.results,
               checkpoints_path=args.results,
               model="GP",
               n_iterations=50,
               verbose=True,
               random_state=0)


if __name__=='__main__':
    main()
```

Keeping in step with the quickstart guide, we have defined an objective function, *stbtang*. As we mentioned above, this function is typically evaluated on the interval $x_i = [-5., 5.] \in D$, and so is initialized to that by default by HyperSpace. We then define our search bounds using Numpy's tile function, which is a convenient way of creating an *ndims* dimensional array of tuples, each of the bounds from *[-5., 5]*. Then all we need to do is call HyperSpace's *hyperdrive* function, telling it where to save the results (*args.results*).

Just remmber, HyperSpace initializes 2^D optimizations in parallel. Therefore, if if we save this as a module called *styblinskitang.py*, we can run this example use the following:

```
mpirun -n 4 python3 styblinskitang.py --ndims 2 --results </path/to/save/results>
```

Try experimenting with the dimension of the function. This can be done simply by changing the *args.ndims* argument and adjusting the number of MPI ranks according to the 2^D rule. Happy optimizing!

CHAPTER 4

ML Example: Gradient Boosted Trees

In quickstart guide we outlined how to setup hyperparameter optimization for machine learning models. We have also seen a minimal example using the Styblinski-Tang function. Let's take a look at a complete machine learning example using Scikit-Learn.

Here we are going to optimize two hyperparameters for a gradient boosted regression tree model. Keeping it to two hyperparameters will allow us to easily optimize the model on a single machine, since most processors these days have four cores and we can distribute the MPI ranks across those. If you are running a dual core machine, or would like to make the computation a bit lighter, you can remove one of the hyperparameters, in which case you will just have just two MPI ranks.

We will be using the Boston Housing Dataset as it is small and readily available through Scikit-Learn. This dataset consists of 506 samples and 13 attributes, some numeric, some categorical. The label is the median home value for these various Boston houses. Thus, we are looking at a regression problem.

Gradient boosted regression trees have separate hyperparameters. For this example we are going to optimize their *max_depth* and *learning_rate*. Our objective function here is to minimize the negative cross validation score of our model over five folds of the data. The scoring metric used for the cross-validation will be the negative mean absolute error. Alright, let's get straight to it:

```python
import argparse
import numpy as np

from sklearn.datasets import load_boston
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import cross_val_score

from hyperspace import hyperdrive
from hyperspace.kepler import load_results


boston = load_boston()
X, y = boston.data, boston.target
n_features = X.shape[1]
```

(continues on next page)

(continued from previous page)

```python
reg = GradientBoostingRegressor(n_estimators=50, random_state=0)


def objective(params):
    """
    Objective function to be minimized.

    Parameters
    ----------
    * params [list, len(params)=n_hyperparameters]
        Settings of each hyperparameter for a given optimization iteration.
        - Controlled by hyperspaces's hyperdrive function.
        - Order preserved from list passed to hyperdrive's hyperparameters argument.
     """
     max_depth, learning_rate = params

     reg.set_params(max_depth=max_depth,
                    learning_rate=learning_rate)

     return -np.mean(cross_val_score(reg, X, y, cv=5, n_jobs=-1,
                     scoring="neg_mean_absolute_error"))


def main():
    parser = argparse.ArgumentParser(description='ML Example: GBM Regression')
    parser.add_argument('--results', type=str, help='Path to results directory.')
    args = parser.parse_args()

    hparams = [(2, 10),             # max_depth
               (10.0**-2, 10.0**0)] # learning_rate

    hyperdrive(objective=objective,
               hyperparameters=hparams,
               results_path=args.results,
               checkpoints_path=args.results,
               model="GP",
               n_iterations=50,
               verbose=True,
               random_state=0)


 if __name__=='__main__':
     main()
```

If we save this as a module by the name gbm.py, we can run this as:

```
mpirun -n 4 python3 gbm.py --results </path/to/save/results>
```

You might have noticed that we added one piece to this example, the ability to load from previous checkpoints. When we give a path to the *checkpoints_path* argument, HyperSpace will save the its progress at each step of the optimization. If that path contains previously saved checkpoints, HyperSpace will resume the optimization from where it left off.

Check out the other parameters available in Scikit-learn's documentation. See if by including more hyperparameters you can get a better result! And if anyone is interested, we can start up a leaderboard on our GitHub page to see who can get the best score. If you are interested, let me know in the GitHub issues!

CHAPTER 5

Viewing Results

Once we have run HypeSpace, we can then check in to see how the optimization performed. If you recall from previous tutorials, we specified a *results_path* within the *hyperdrive* function. This specified the path where we saved the results from the distributed run. Loading those results is as simple as

```
from my_optimization import objective
from hyperspace.kepler.data_utils import load_results


path = '/results_path'
results = load_results(path, sort=True)
```

Pointing *load_results* to the directory that we saved the HyperSpace results to returns a list of *Scipy.OptimizeResults* objects. By setting *sort=True*, we sort the results by the minimum found in ascending order. Note that we have imported the *objective* function here. This is because the Python pickles the reference to that namespace whenever it saves *Scipy.OptimizeResults* objects. Each element of this list of results contains all of the information gained through the optimization process for the respective distributed ranks. This includes the following:

1. *fun*: the minimum found by the optimization
2. *func_vals*: the function value found at each iteration of the optimization
3. *models*: the specification of the surrogate model at each iteration
4. *random_state*: the random seed
5. *space*: the bounds of the search space for that particular rank
6. *specs*: the specification for the Bayesian SMBO
7. *x*: point found in the domain that returns the minimal *fun* value
8. *x_iters*: the point in domain sampled at each iteration of the optimization

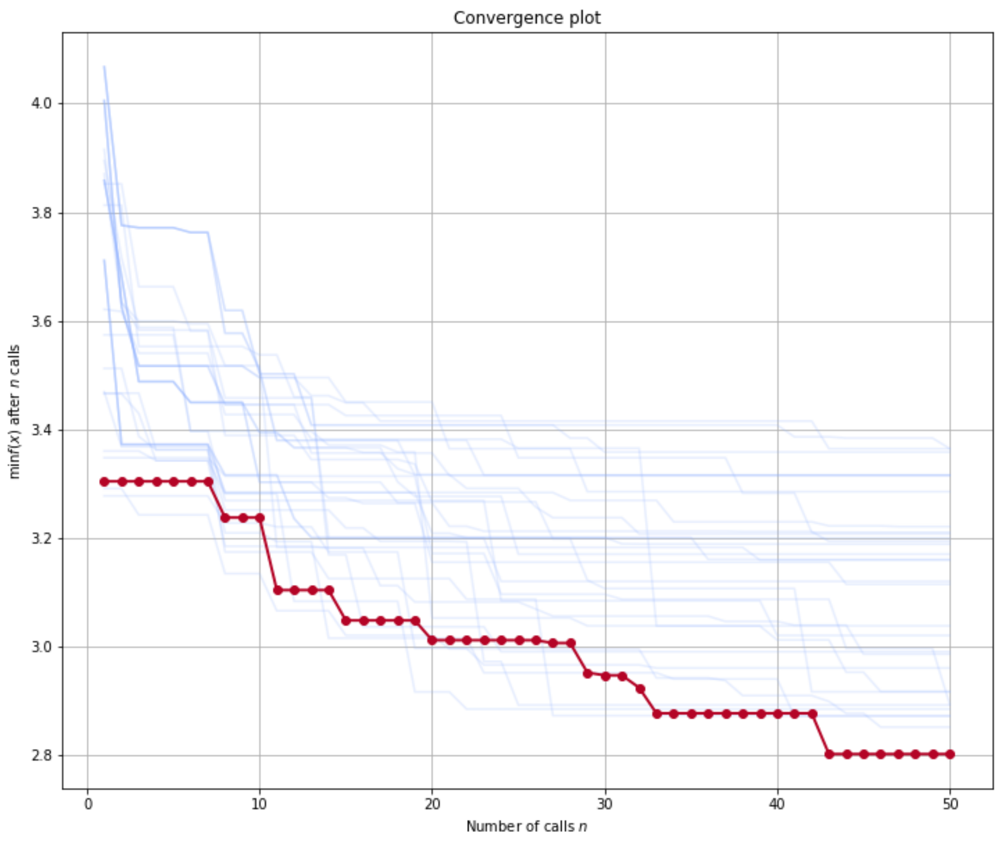
We can then visualize the course of the optimization with the following:

```
from hyperspace.kepler.plots import plot_convergence
```

(continues on next page)

(continued from previous page)

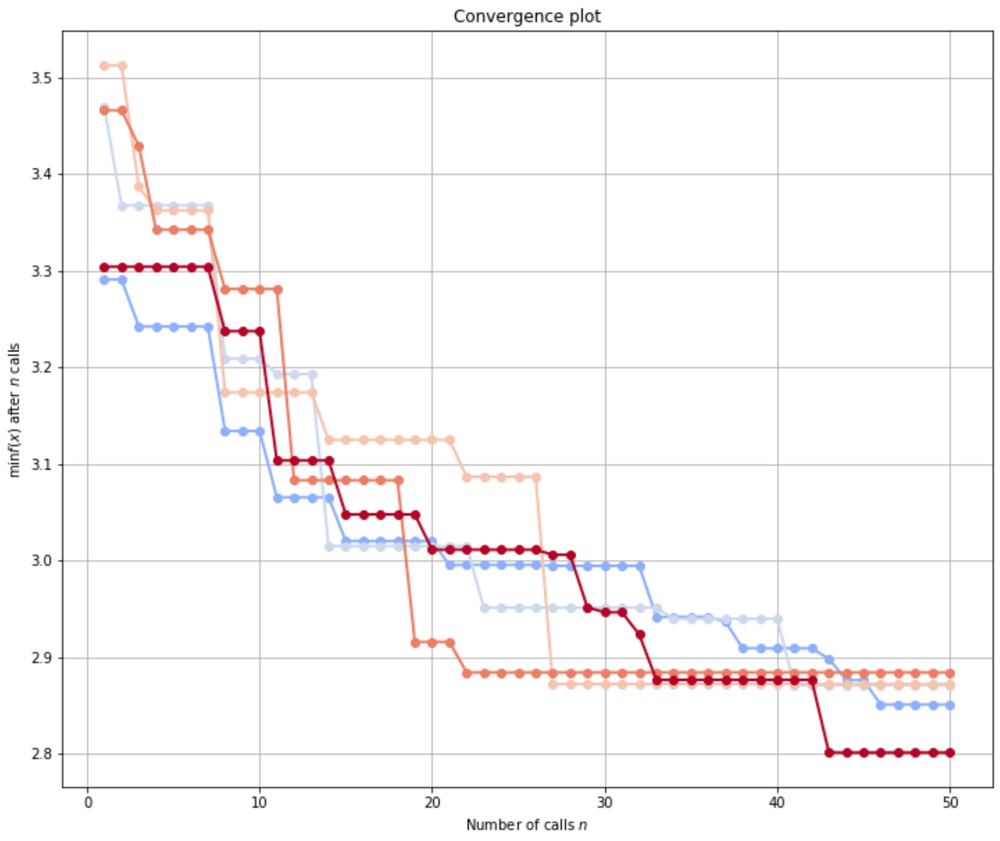
```
best_result = results.pop(0)

_ = plot_convergence(results, best_result)
```



The above figure shows the convergence plot when optimizing five hyperparameters of our regression model from our Gradient Boosted Trees example. The traces show the optimization progress at each rank as we run 50 iterations of Bayesian SMBO in parallel. The red trace shows the best performing rank.

Note that the *plot_convergence* function can accept a list of HyperSpace results (*results* in this case), and it can accept a single rank's result (*best_results* here). If we had many traces from a large scale run, it can sometimes be helpful to look at only a few of the traces at a time. Here is how we can look at the top five results from the above HyperSpace run:

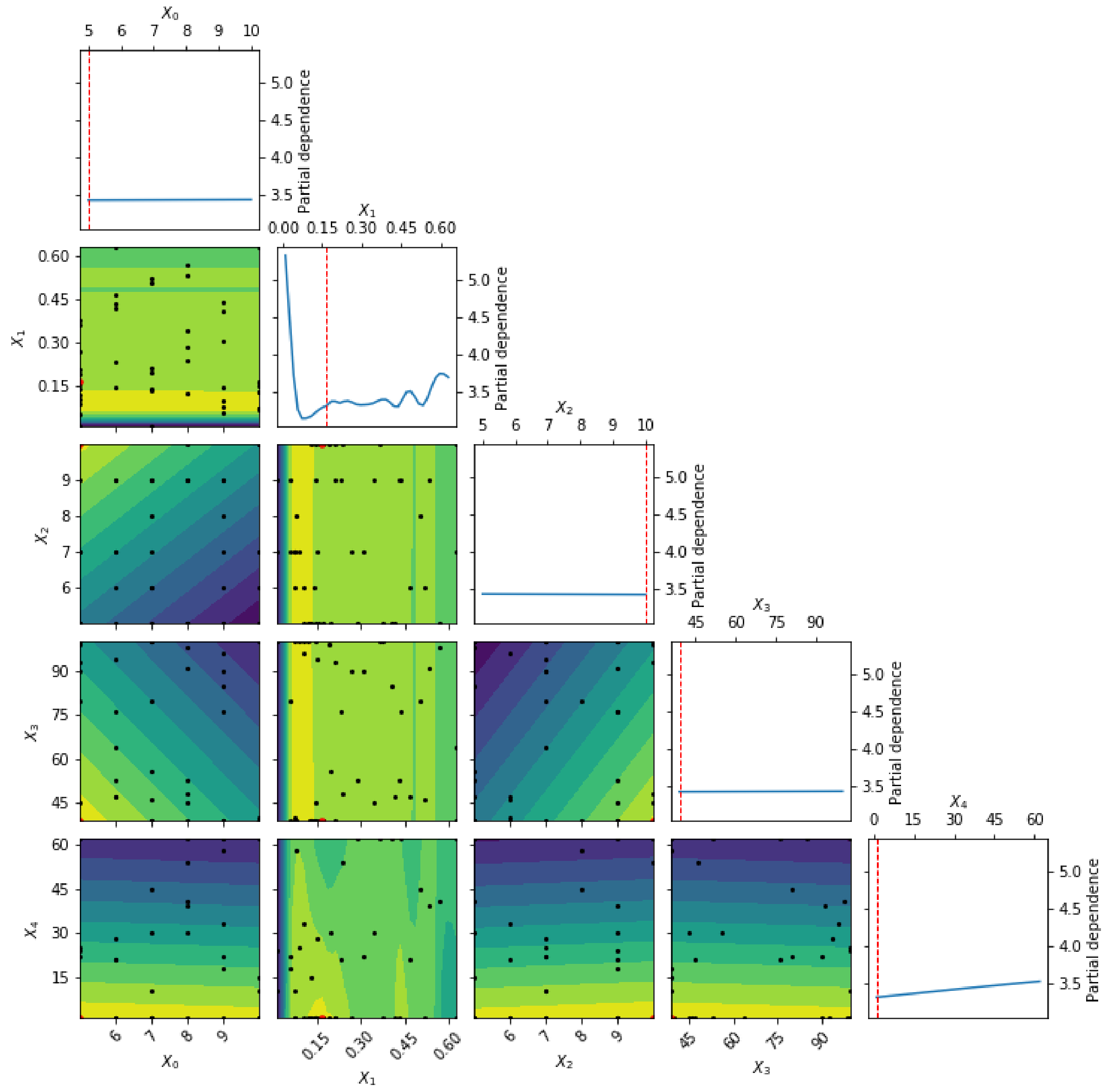
```
_ = plot_convergence(results[0], results[1], results[2], results[3], best_result)
```



By examining the optimization traces, we could then decide to prune some of the search spaces in the future. Since machine learning is always an iterative process, understanding the behavior of our models over various hyperparameter search spaces can inform future model building.

We can also explore the effects of the hyperparameters within each of the search spaces. Since Scikit-Optimize *[2]* is integrated into HyperSpace, we can then explore the partial dependencies between hyperparameters. Originally proposed by Jerome H. Friedman, partial dependence plots were designed to visualize the effects of input variables for gradient boosted machines *[1]*. Here we can use them to understand the partial dependence of our objective function on each of the hyperparameters, holding all others constant, as well as all two-way interactions between hyperparameters, again holding all others constant.

```
from skopt.plots import plot_objective


_ = plot_objective(best_result)
```



The above partial dependence plot shows how our objective function behaves at various settings of our model hyperparameters when considering the best hyperparameter subspace found by HyperSpace. Along the diagonal we have the effect of each hyperparameter, holding all other hyperparameters constant, on the five fold cross validation loss for our gradient boosted regressor. The dotted red line each of those subplots indicates the final setting of the hyperparameter. All off-diagonal plots show the two-way interactions for the hyperparameters, holding all other hyperparameters constant. The black points within these subplots show the points sampled along the two search dimensions. The red point in each of those subplots shows the final setting for the hyperparameters. The countours on the two-way interactions subplots show the loss landscape as viewed by our surrogate Gaussian process. Yellow contours indicate lower regions in the loss, whereas blue regions show higher loss values.

Note that the above partial dependence example shows the partial dependence of our hyperparameters *within the best performing search space* found by HyperSpace. We can also inspect these partial dependencies for all other search spaces used by HyperSpace. By doing so we can explore how a change in the bounds of the search space effects our optimization. We can therefore view where the models are performing well and where they are performing poorly. This is key to better understanding how our models behave. It is not only useful to find when our models perform well, knowing when the models will break down is equally important!

As we iterate through the model building process, it is helpful to visualize the outcomes of the optimization process. This allows us to inject our preferences and intuitions into the process so that we can better steer future directions.

References

CHAPTER 6

Space

6.1 HyperInteger

CHAPTER 7

hyperspace.hyperdrive package

7.1 Submodules

7.2 hyperspace.hyperdrive.hyperdrive module

7.3 Module contents

CHAPTER 8

hyperspace.kepler package

8.1 Submodules

8.2 hyperspace.kepler.data_utils module

8.3 hyperspace.kepler.plots module

8.4 Module contents

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, 2951–2959. 2012.

[1] Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232, 10 2001. URL: https://doi.org/10.1214/aos/1013203451, doi:10.1214/aos/1013203451.

[2] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene-rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. Scikit-optimize/scikit-optimize: v0.5.2. March 2018. doi:10.5281/zenodo.1207017.