Introduction to Bayesian Optimization with Hyperopt

September 27, 2018

1 Introduction: Bayesian Optimization using Hyperopt

In this notebook we will walk through the basics of Bayesian Model-Based Optimization using Hyperopt to find the minimum of a function. After we are familiar with the basic concepts, we can use these powerful methods to solve many problems, inlcuding hyperparamter optimization of machine learning models.

1.1 Bayesian Model-Based Optimization Primer

There are four parts to an optimization problem:

1. Objective function: what we want to minimize
2. Domain space: values of the parameters over which to minimize the objective
3. Hyperparameter optimization function: constructs the surrogate function and chooses next values to evaluate
4. Trials: score, parameter pairs recorded each time we evaluate the objective function

Evaluating the objective function is the expensive part of optimization, so ideally we want to limit calls to this function. One way we can limit calls is by choosing the next values to try in the objective function based on the past results. Bayesian optimization differs from random or grid search by doing exactly this: rather than just selecting from a grid **uninformed** by past objective function evaluations, Bayesian methods take into account the previous results to try more promising values. They work by constructing a probability model of the objective function (called a surrogate function) $p(\text{score}|\text{parameters}$ which is much easier to optimize than the actual objective function.

After each evaluation of the objective function, the algorithm updates the probability model (usually given as $p(y|x)$ incorporating the new results. Sequential Model-Based Optimization (SMBO) methods are a formalization of Bayesian optimization that update the probability model sequentially: every evaluation of the objective function with a set of values updates the model with the idea that eventually the model will come to represent the true objective function. This is an application of Bayesian Reasoning. The algorithm forms an initial idea of the objective function and updates it with each new piece of evidence.

The next values to try in the objective function are selected by the algorithm optmizing the probability model (surrogate function) usually with a criteria known as Expected Improvement. Finding the values that will yield the greatest expected improvement in the surrogate function is much cheaper than evaluating the objective function itself. By choosing the next values based on a model rather than randomly, the hope is that the algorithm with converge to the true best values

much quicker. The overall goal is to evaluate the objective function fewer times by spending a little more time choosing the next values. Overall, Bayesian Optimization and SMBO methods:

- Converge to a lower score of the objective function than random search
- Require far less time to find the optimum of the objective function

So, we get both faster optimization and a better result. These are both two desirable outcomes especially when we are working with hyperparameter tuning of machine learning models!

1.1.1 Sequential Model Based Optimization using the Tree Parzen Estimator

SMBO methods differ in part 3, the algorithm used to construct the probability model (also called the Surrogate Function). Several options are for the surrogate function are:

- Gaussian Processes
- Tree-structured Parzen Estimator
- Random Forest Regression

Hyperopt implements the Tree-strucuted Parzen Estimator. The details of this algorithm are not that complicated, but we will not discuss them in this notebook (for details refer to [this article.]) We don't need to worry about implementing the algorithm because Hyperopt takes care of that for us. We just have to make sure we have properly defined the **objective function** and the **domain of values to search over**. Then, we select the algorithm and let it run. Hyperopt will automatically keep track of past results to inform the model (part 4.) but we can also use a custom object to get more information about the optimization procedure.

1.2 Hyperopt

Hyperopt is an open-source Python library for Bayesian optimization that implements SMBO using the Tree-structured Parzen Estimator. There are a number of libraries available for Bayesian optimization and Hyperopt differs in that it is the only one to currently offer the Tree Parzen Estimator. Other libraries use a Gaussian Process or a Random Forest regression for the surrogate function (probability model).

In this notebook, we will implement both random search (Hyperopt has a method for this) as well as the Tree Parzen Estimator, a Sequential Model-Based Optimization method. We will use a simple problem that will allow us to learn the basics as well as a number of techniques that will be very helpful when we get to more complex use cases. With all that out of the way, let's get started with Hyperopt!

Resources For more details on Bayesian Model Based Optimization, here are some good articles:

- Algorithms for Hyper-Parameter Optimization Link
- Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures Link
- Bayesian Optimization Primer Link
- Taking the Human Out of the Loop: A Review of Bayesian Optimization Link

```
In [7]: # Good old pandas and numpy
        import pandas as pd
        import numpy as np

        # Unfortunately I'm still using matplotlib for graphs
        import matplotlib.pyplot as plt
        import seaborn as sns
```

2 Objective

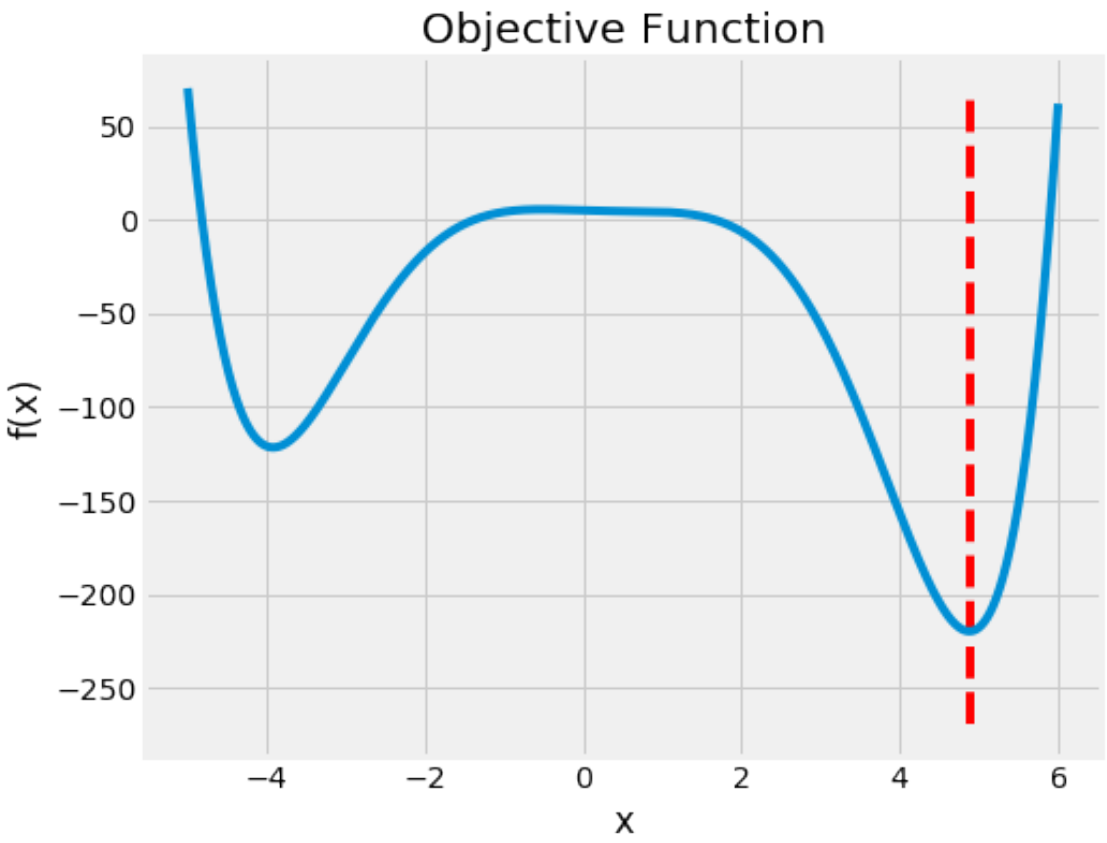
For our objective function, we will use a simple polynomial function with the goal being to find the minimu value. This function has one global minimum over the range we define it as well as one local minimum.

When we define the objective function, we must make sure it returns a single real-value number to *minimize*. If we use a metric such as accuracy, then we would have to return the *negative* of accuracy to tell our model to find a better accuracy! We can also return a dictionary (we will see this later) where one of the keys must be "loss". Here we will just return the output of the function $f(x)$

```
In [8]: def objective(x):
            """Objective function to minimize"""

            # Create the polynomial object
            f = np.poly1d([1, -2, -28, 28, 12, -26, 100])

            # Return the value of the polynomial
            return f(x) * 0.05
```

```
In [9]: # Space over which to evluate the function is -5 to 6
        x = np.linspace(-5, 6, 10000)
        y = objective(x)

        miny = min(y)
        minx = x[np.argmin(y)]

        # Visualize the function
        plt.figure(figsize = (8, 6))
        plt.style.use('fivethirtyeight')
        plt.title('Objective Function'); plt.xlabel('x'); plt.ylabel('f(x)')
        plt.vlines(minx, min(y)- 50, max(y), linestyles = '--', colors = 'r')
        plt.plot(x, y);

        # Print out the minimum of the function and value
        print('Minimum of %0.4f occurs at %0.4f' % (miny, minx))
```

```
Minimum of -219.8012 occurs at 4.8779
```



3 Domain

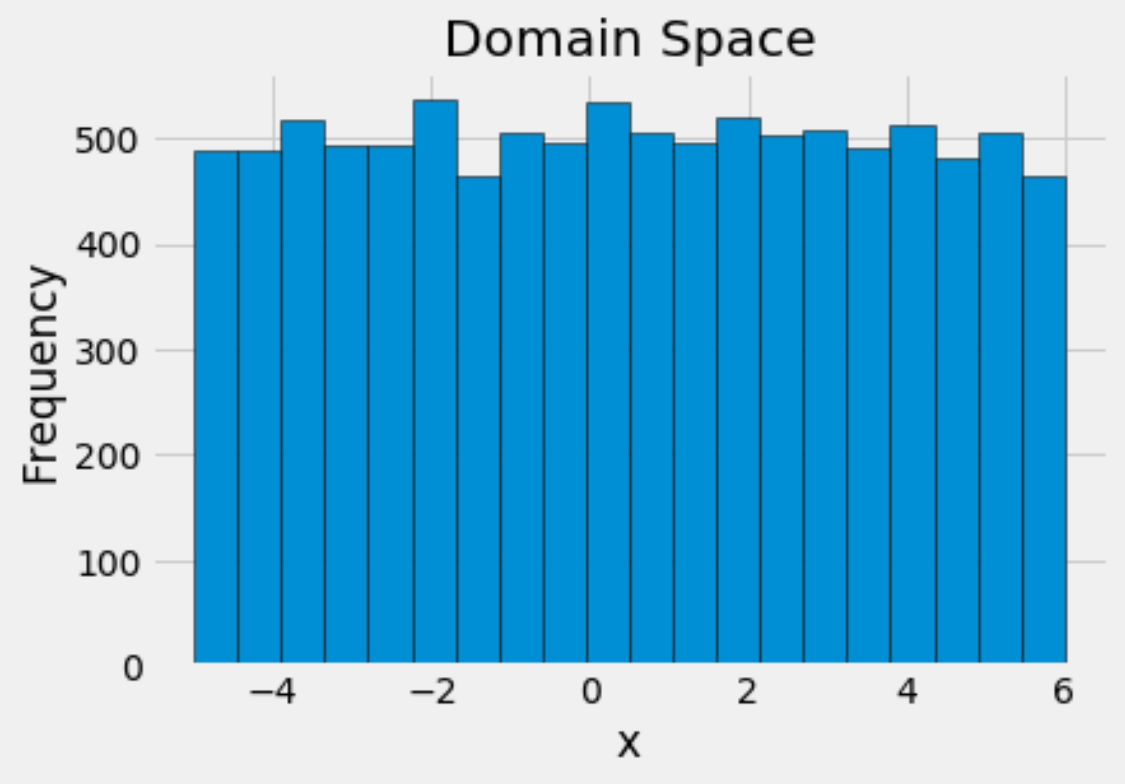
The domain is the values of x over which we evaluate the function. First we can use a uniform distribution over the space our function is defined.

```
In [10]: from hyperopt import hp

         # Create the domain space
         space = hp.uniform('x', -5, 6)
```

We can draw samples from the space using a Hyperopt utility. This is useful for visualizing a distribution.

```
In [11]: from hyperopt.pyll.stochastic import sample


         samples = []

         # Sample 10000 values from the range
         for _ in range(10000):
```

```
    samples.append(sample(space))


# Histogram of the values
plt.hist(samples, bins = 20, edgecolor = 'black');
plt.xlabel('x'); plt.ylabel('Frequency'); plt.title('Domain Space');
```



When running, our algorithm will sample values from this distribution, initially at random as it explores the domain space, but then over time, it will "focus" on the most promising values. Therefore, the algorithm should more values around 4.9, the minimum of the function. We can compare this to random search which should try values evenly from the entire distribution.

4 Hyperparameter Optimization Algorithm

There are two choices for a hyperparameter optimization algorithm in Hyperopt: random and Tree Parzen Estimator. We can use both and compare the results. Using the `suggest` algorithm in these families automatically configures the algorithm for us.

```
In [12]: from hyperopt import rand, tpe

         # Create the algorithms
         tpe_algo = tpe.suggest
         rand_algo = rand.suggest
```

5 History

Storing the history is as simple as making a `Trials` object that we pass into the function call. This is not strictly necessary, but it gives us information that we can use to understand what the algorithm is doing.

```
In [13]: from hyperopt import Trials

         # Create two trials objects
         tpe_trials = Trials()
         rand_trials = Trials()
```

6 Run the Optimization

Now that all four parts are in place, we are ready to minimize! Let's do 2000 runs of the minimization with both the random algorithm and the Tree Parzen Estimator algorithm.

The `fmin` function takes in exactly the four parts specified above as well as the maximum number of evaluations to run. We will also set a `rstate` for reproducible results across multiple runs.

```
In [14]: from hyperopt import fmin

         # Run 2000 evals with the tpe algorithm
         tpe_best = fmin(fn=objective, space=space, algo=tpe_algo, trials=tpe_trials,
                         max_evals=2000, rstate= np.random.RandomState(50))

         print(tpe_best)

         # Run 2000 evals with the random algorithm
         rand_best = fmin(fn=objective, space=space, algo=rand_algo, trials=rand_trials,
                          max_evals=2000, rstate= np.random.RandomState(50))
```

```
{'x': 4.878481851906148}
```

```
In [15]: # Print out information about losses
         print('Minimum loss attained with TPE:    {:.4f}'.format(tpe_trials.best_trial['result
         print('Minimum loss attained with random: {:.4f}'.format(rand_trials.best_trial['resul
         print('Actual minimum of f(x):            {:.4f}'.format(miny))

         # Print out information about number of trials
         print('\nNumber of trials needed to attain minimum with TPE:    {}'.format(tpe_trials
         print('Number of trials needed to attain minimum with random: {}'.format(rand_trials.

         # Print out information about value of x
         print('\nBest value of x from TPE:    {:.4f}'.format(tpe_best['x']))
         print('Best value of x from random: {:.4f}'.format(rand_best['x']))
         print('Actual best value of x:      {:.4f}'.format(minx))
```

```
Minimum loss attained with TPE:    -219.8012
Minimum loss attained with random: -219.8012
Actual minimum of f(x):            -219.8012

Number of trials needed to attain minimum with TPE:    655
Number of trials needed to attain minimum with random: 235

Best value of x from TPE:    4.8785
Best value of x from random: 4.8776
Actual best value of x:      4.8779
```

The Tree Parzen estimator and random search found the exact same minimum of the function to 4 decimal places. Sometimes even random search gets lucky (especially when we run it for so many iterations. However, we can see that TPE found the minimum in about half the number of iterations. After finding this minimum, we could have stopped running the algorith!

Let's see if there is any difference in time between the two methods. We can use the built in Jupyter magic command for timing (running the function 3 times).

```
In [16]: %%timeit -n 3
         # Run 2000 evals with the tpe algorithm
         best = fmin(fn=objective, space=space, algo=tpe_algo, max_evals=200)

831 ms ś 12.9 ms per loop (mean ś std. dev. of 7 runs, 3 loops each)


In [17]: %%timeit -n 3

         # Run 2000 evals with the random algorithm
         best = fmin(fn=objective, space=space, algo=rand_algo, max_evals=200)

167 ms ś 15.6 ms per loop (mean ś std. dev. of 7 runs, 3 loops each)
```

As a point of interest, the random algorithm ran about 5 times faster than the tpe algorithm. This shows that the TPE method is taking more time to propose the next set of parameters while the random method is just choosing from the space well, randomly. The extra time to choose the next parmaeters is made up for by choosing better parameters that should let us make fewer overall calls to the objective function (which is the most expensive part of optimization). Here we ran the same number of totals calls, but this was probably not necessary because the Tree Parzen Estimator will converge on the optimum quickly without the need for more iterations.

7 Results

We see that both models returned values very close to the optimal. To see how they differ in the search procedure, we can take a look at the trials objects.

```
In [18]: tpe_results = pd.DataFrame({'loss': [x['loss'] for x in tpe_trials.results], 'iterati
                                    'x': tpe_trials.idxs_vals[1]['x']})
```

```
tpe_results.head()
```

Out[18]:

	iteration	loss	x
0	0	36.210073	5.957885
1	1	-202.384052	4.470885
2	2	-75.519449	3.218963
3	3	5.543552	-0.515859
4	4	35.078011	-4.916832

Extracting these results was a little work. We could have formatted the objective function to return more useful information (in a little bit we'll how to do this).

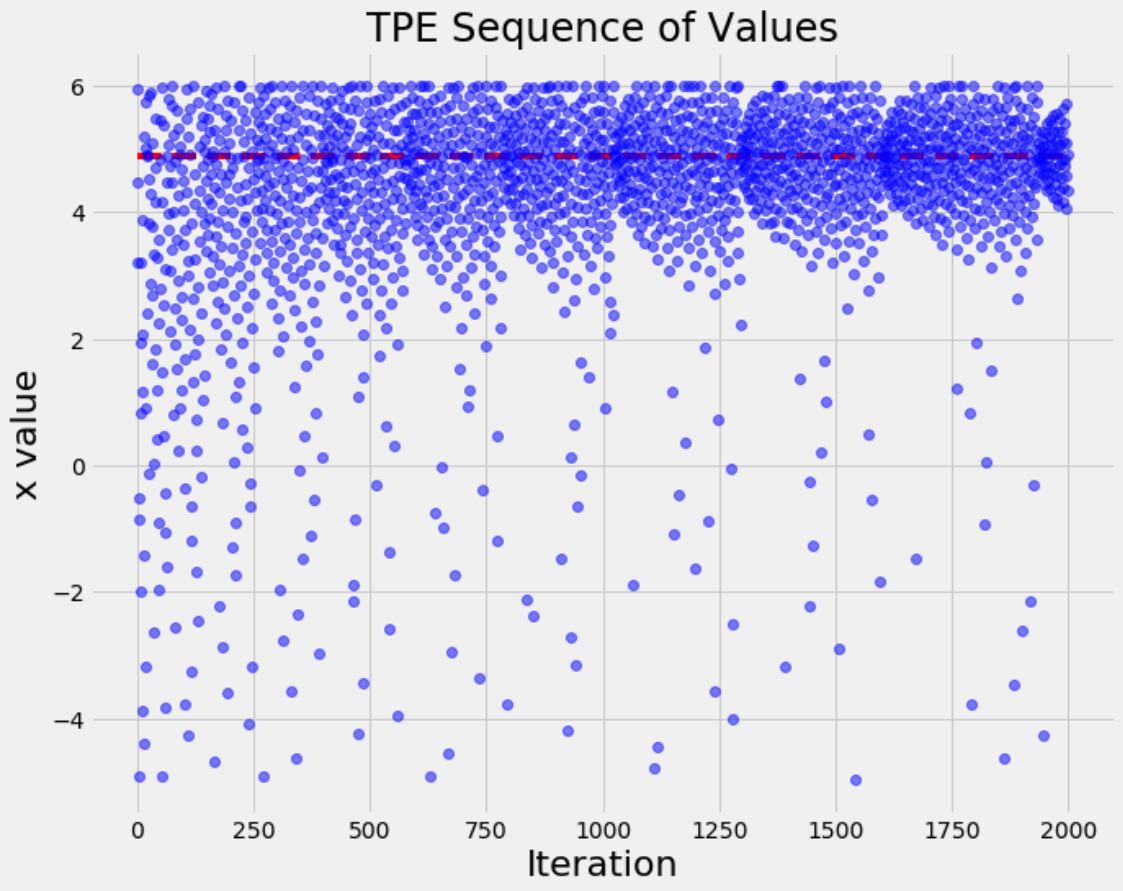
First we can plot the values that were evaluated over time. As the algorithm progresses, these should tend to cluster around the actual best value (near 4.9)

```
In [19]: tpe_results['rolling_average_x'] = tpe_results['x'].rolling(50).mean().fillna(method =
         tpe_results['rolling_average_loss'] = tpe_results['loss'].rolling(50).mean().fillna(me
         tpe_results.head()
```

Out[19]:

	iteration	loss	x	rolling_average_x	rolling_average_loss
0	0	36.210073	5.957885	2.105103	-74.393795
1	1	-202.384052	4.470885	2.105103	-74.393795
2	2	-75.519449	3.218963	2.105103	-74.393795
3	3	5.543552	-0.515859	2.105103	-74.393795
4	4	35.078011	-4.916832	2.105103	-74.393795

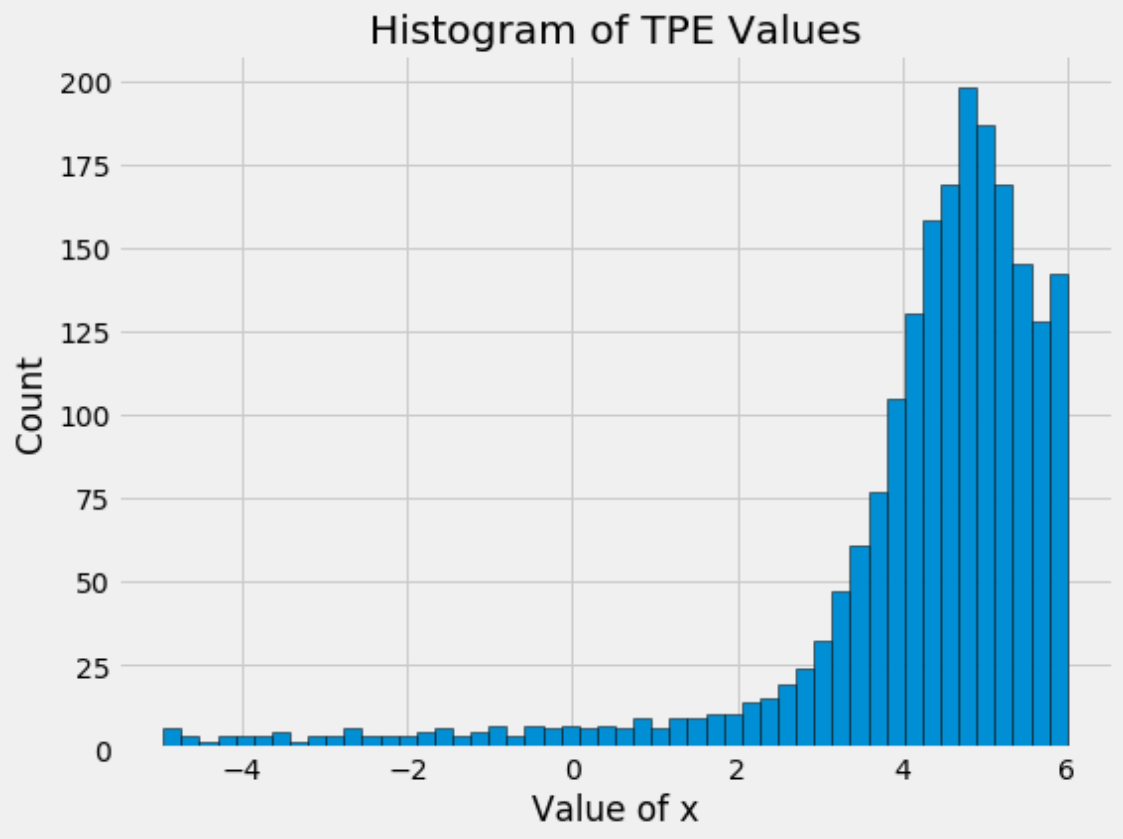
```
In [20]: plt.figure(figsize = (10, 8))
         plt.plot(tpe_results['iteration'], tpe_results['x'],  'bo', alpha = 0.5);
         plt.xlabel('Iteration', size = 22); plt.ylabel('x value', size = 22); plt.title('TPE S
         plt.hlines(minx, 0, 2000, linestyles = '--', colors = 'r');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-3.jpg",dpi=300)
```



We can see that over time, the algorithm tended to try values closer to 4.9. The local minimum around -4 likely threw off the algorithm initially, but the points tend to cluster around the actual minimum as the algorithm progresses.

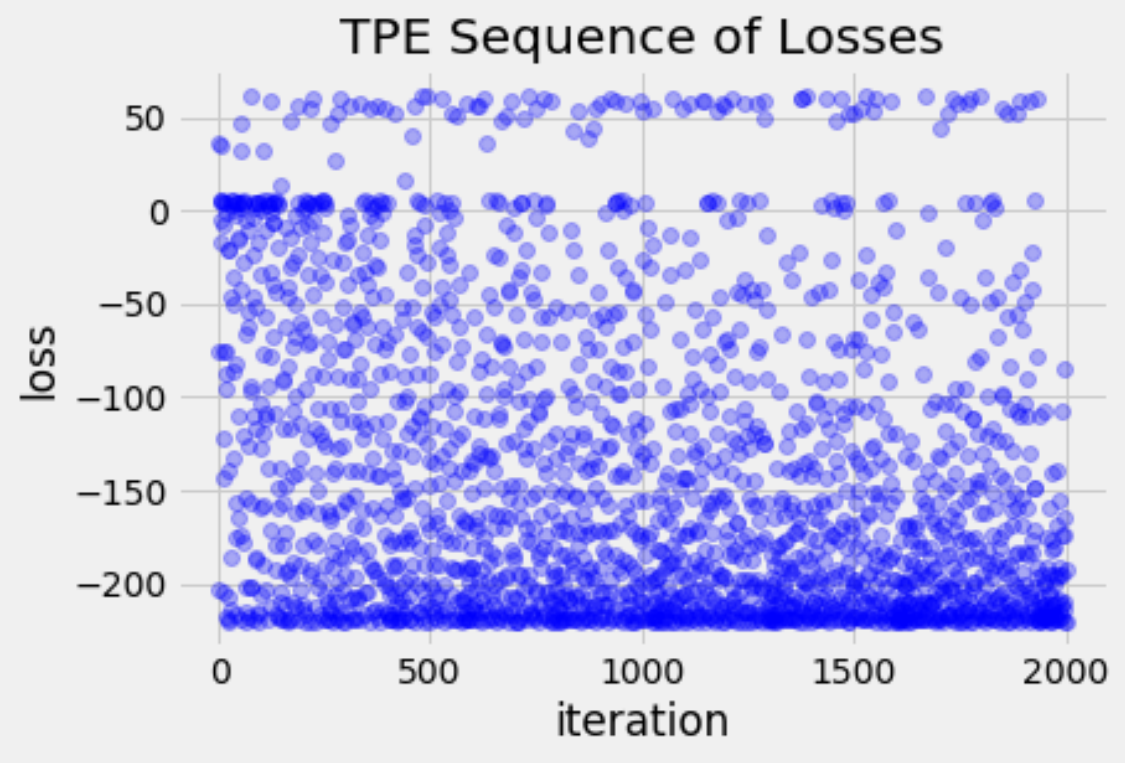
We can also plot the histogram to see the distribution of values tried.

```
In [21]: plt.figure(figsize = (8, 6))
         plt.hist(tpe_results['x'], bins = 50, edgecolor = 'k');
         plt.title('Histogram of TPE Values'); plt.xlabel('Value of x'); plt.ylabel('Count');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-4.jpg",dpi=300)
```



Sure enough, the algorithm tried many values closer to 4.9 that anywhere else! This clearly shows the benefits of choosing the next values based on the past values: more evaluations of promising values.

```
In [22]: # Sort with best loss first
         tpe_results = tpe_results.sort_values('loss', ascending = True).reset_index()

         plt.plot(tpe_results['iteration'], tpe_results['loss'], 'bo', alpha = 0.3);
         plt.xlabel('iteration'); plt.ylabel('loss'); plt.title('TPE Sequence of Losses');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-5.jpg",dpi=300)

         print('Best Loss of {:.4f} occured at iteration {}'.format(tpe_results['loss'][0], tp

Best Loss of -219.8012 occured at iteration 655
```



7.0.1 Random Results

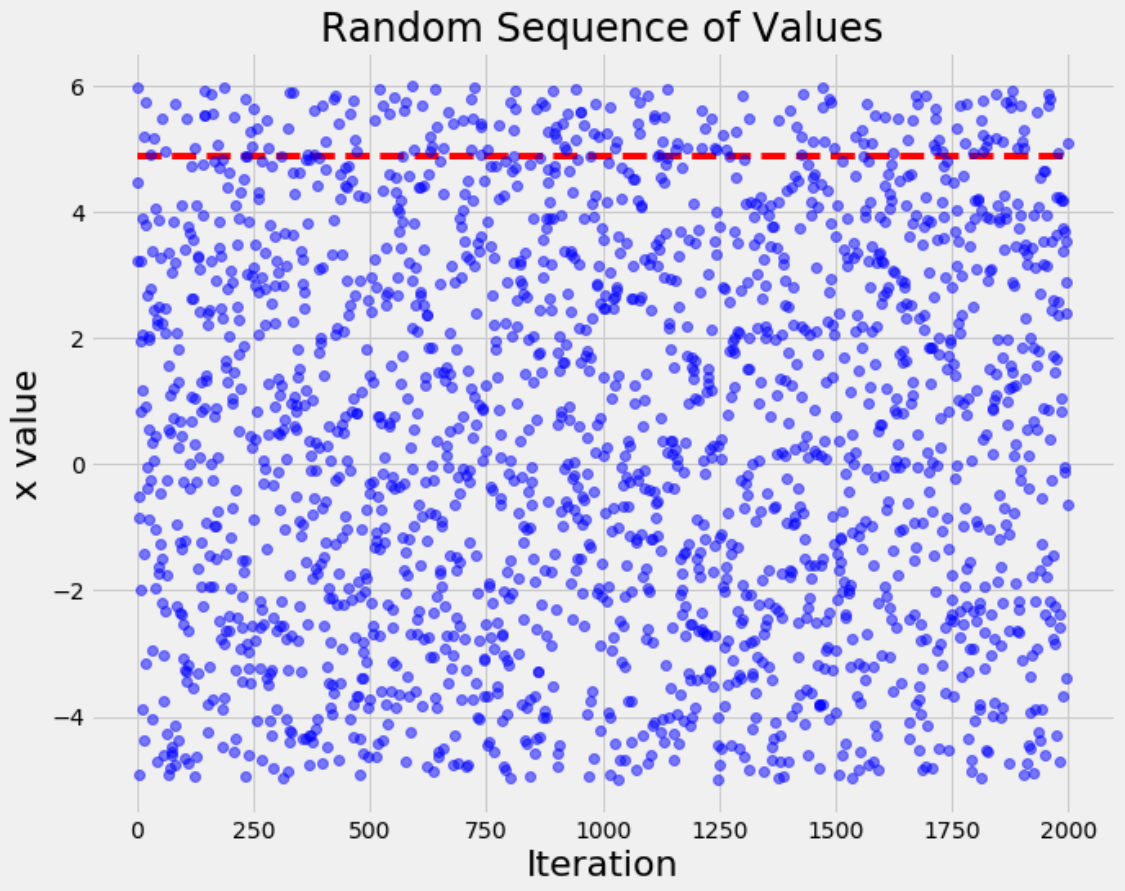
We should contrast the TPE results with those from random search. Here we do not expect to see any trend in values evaluated over time because the values are selected randomly.

```
In [23]: rand_results = pd.DataFrame({'loss': [x['loss'] for x in rand_trials.results], 'iterat
                                      'x': rand_trials.idxs_vals[1]['x']})

         rand_results.head()
```

Out[23]:

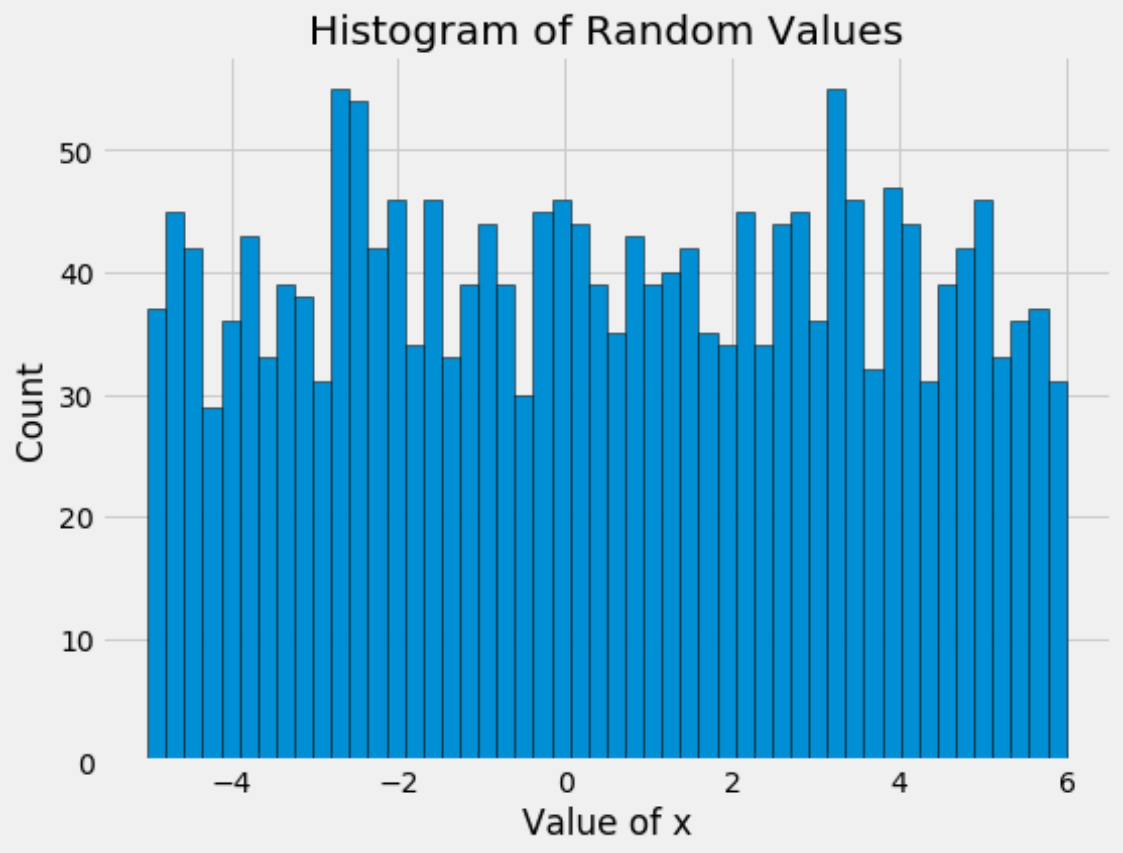
	iteration	loss	x
0	0	36.210073	5.957885
1	1	-202.384052	4.470885
2	2	-75.519449	3.218963
3	3	5.543552	-0.515859
4	4	35.078011	-4.916832

```
In [24]: plt.figure(figsize = (10, 8))
         plt.plot(rand_results['iteration'], rand_results['x'],  'bo', alpha = 0.5);
         plt.xlabel('Iteration', size = 22); plt.ylabel('x value', size = 22); plt.title('Rando
         plt.hlines(minx, 0, 2000, linestyles = '--', colors = 'r');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-6.jpg",dpi=300)
```



Clearly, the random algorithm is just "randomly" choosing the next set of values to try! There is no logical order here.

```
In [25]: # Sort with best loss first
         rand_results = rand_results.sort_values('loss', ascending = True).reset_index()

         plt.figure(figsize = (8, 6))
         plt.hist(rand_results['x'], bins = 50, edgecolor = 'k');
         plt.title('Histogram of Random Values'); plt.xlabel('Value of x'); plt.ylabel('Count')
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-7.jpg",dpi=300)

         # Print information
         print('Best Loss of {:.4f} occured at iteration {}'.format(rand_results['loss'][0], ra

Best Loss of -219.8012 occured at iteration 235
```



Again, there is no discernable clustering because this method is choosing the next values randomly. In this case, it happended across the minimum of the function because it was a relatively simple problem and we used many iterations. This would not likely be the case when we are optimizing more complex functions.

8 Slightly Advanced Concepts

These probably should not be called advanced but smarter concepts because they will make our jobs easier. We will cover two upgrades we can make:

- Smarter domain space over which to search
- Return more useful information from the objective function

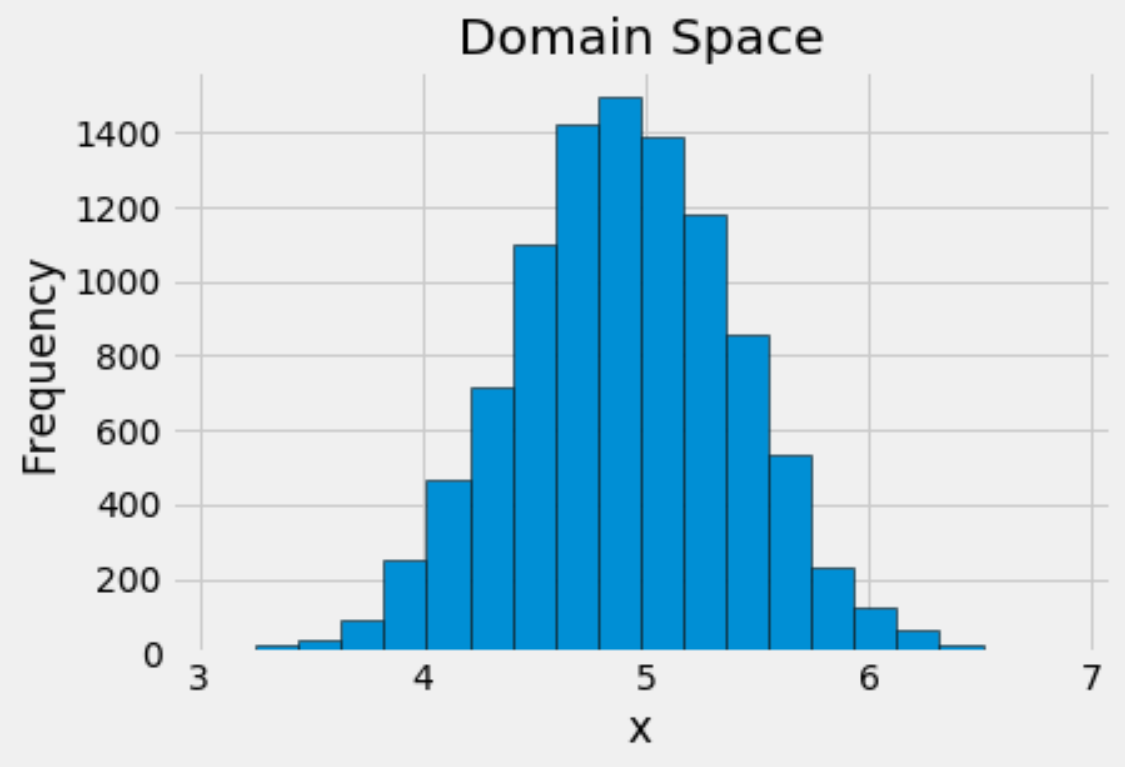
These will be helpful when it comes time for hyperparameter optimization. Master the details on the easy problems so you can use the tools more effectively on the difficult problems.

8.0.1 Better Domain Space

In this problem, we can cheat because we know where the minimum is and therefore can define a region of higher probability around this value of x. In more complicated problems, we don't have a graph to show us the minimum, but we can still use experience and knowledge to inform our choice of a domain space.

Here we will make a normally distributed domain space around the value where the minimum of the objective function occurs, around 4.9. This is simple to do in Hyperopt.

```
In [26]: # Normally distributed space
         space = hp.normal('x', 4.9, 0.5)

In [27]: samples = []

         # Sample 10000 values from the range
         for _ in range(10000):
             samples.append(sample(space))


         # Histogram of the values
         plt.hist(samples, bins = 20, edgecolor = 'black');
         plt.xlabel('x'); plt.ylabel('Frequency'); plt.title('Domain Space');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-8.jpg",dpi=300)
```



Much closer to the true value! This would help both the random search and the TPE find the minimum quicker (for random search it would help because we are concentrating the possible values around the optimum).

8.1 More Useful Trials Object

Another modification to make is to return more useful information from the objective function. We do this using a dictionary with any information we want included. The only requirements are that the dictionary must contain a single real-valued metric to minimize stored under a `"loss"` key and whether the function sucessfully ran, stored under a `"status"` key. Here we make the modifications to the objective to store the value of x as well as the time to evaluate.

```
In [28]: from hyperopt import STATUS_OK
         from timeit import default_timer as timer

In [29]: def objective(x):
             """Objective function to minimize with smarter return values"""

             # Create the polynomial object
             f = np.poly1d([1, -2, -28, 28, 12, -26, 100])

             # Evaluate the function
             start = timer()
             loss = f(x) * 0.05
             end = timer()

             # Calculate time to evaluate
             time_elapsed = end - start

             results = {'loss': loss, 'status': STATUS_OK, 'x': x, 'time': time_elapsed}

             # Return dictionary
             return results
```

We run the algorithm the same as before. We'll create a new `Trials` object to save new results.

```
In [30]: # New trials object
         from hyperopt import Trials
         trials = Trials()

         # Run 2000 evals with the tpe algorithm
         best = fmin(fn=objective, space=space, algo=tpe_algo, trials=trials,
                     max_evals=2000, rstate= np.random.RandomState(120))
```

This time our trials object will have all of our information in the `results` attribute.

```
In [31]: results = trials.results
         results[:2]

Out[31]: [{'loss': -189.3682041842684,
           'status': 'ok',
           'time': 0.00010575333860174396,
           'x': 5.312379584994148},
```

```
 {'loss': -219.325099632915,
  'status': 'ok',
  'time': 7.76416916181688e-05,
  'x': 4.8166084516702705}]
```

This time we have more information from the results. We can extract the results into a dataframe for inspect and plotting if we like.

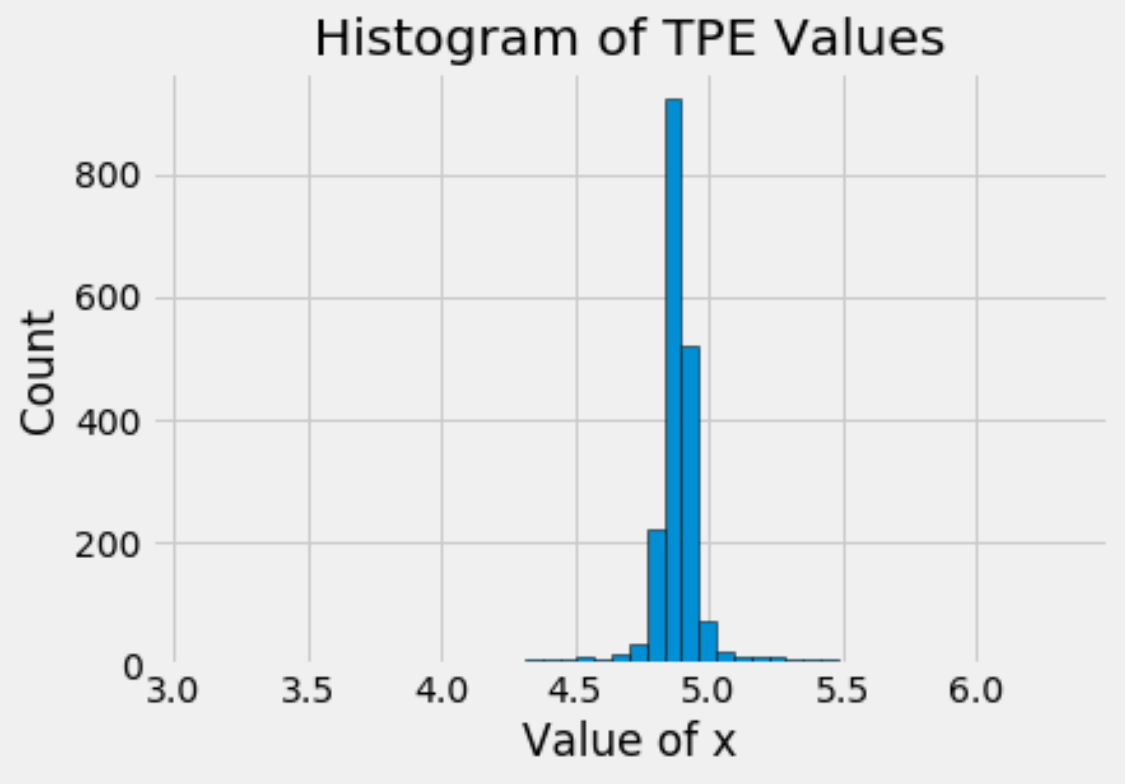
```
In [32]: # Results into a dataframe
         results_df = pd.DataFrame({'time': [x['time'] for x in results],
                                    'loss': [x['loss'] for x in results],
                                    'x': [x['x'] for x in results],
                                     'iteration': list(range(len(results)))})

         # Sort with lowest loss on top
         results_df = results_df.sort_values('loss', ascending = True)
         results_df.head()
```

Out[32]:

	iteration	loss	time	x
956	956	-219.801204	0.000048	4.878152
1316	1316	-219.801204	0.000049	4.878111
402	402	-219.801204	0.000040	4.878189
914	914	-219.801203	0.000046	4.878064
1954	1954	-219.801203	0.000052	4.878222

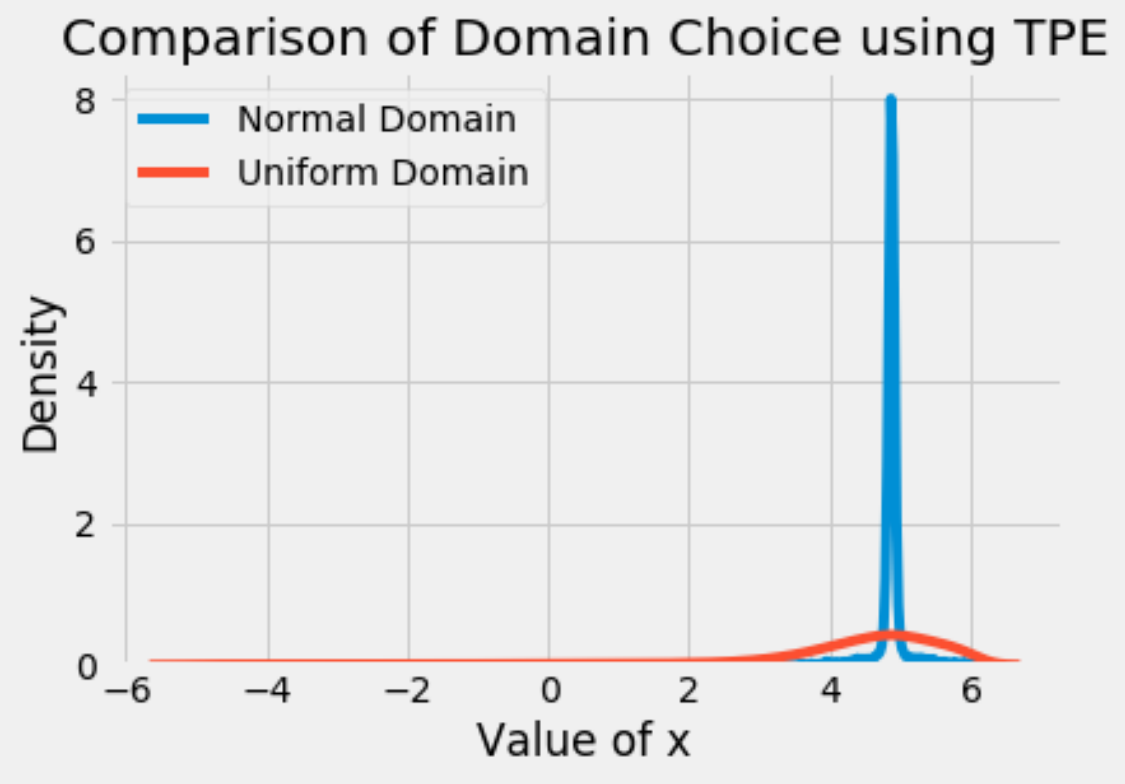
This was a simple optimization problem, so the lowest score obtained does not really differ from that with the uniform distribution. It took around the same number of iterations to converge. We can compare the distribution though to see if better values were tried.

```
In [33]: plt.hist(results_df['x'], bins = 50, edgecolor = 'k');
         plt.title('Histogram of TPE Values'); plt.xlabel('Value of x'); plt.ylabel('Count');
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-9.jpg",dpi=300)
```



Indeed the values of x evaluated cluster much closer to the optimum! The algorithm spend much more time around the best value than searching the domain space. We can compare this distribution to that attained with the TPE algorithm on the uniform domain using a Kernel Density Estimate Plot.

```
In [34]: sns.kdeplot(results_df['x'], label = 'Normal Domain')
         sns.kdeplot(tpe_results['x'], label = 'Uniform Domain')
         plt.legend(); plt.xlabel('Value of x'); plt.ylabel('Density'); plt.title('Comparison
         plt.savefig("C:\\Users\\RY\\Desktop\\figure-10.jpg",dpi=300)
```




```
In [35]: print('Lowest Value of the Objective Function = {:.4f} at x = {:.4f} found in {:.0f} i
                                                                                    results_
                                                                                    results_
```

```
Lowest Value of the Objective Function = -219.8012 at x = 4.8782 found in 956 iterations.
```

Well that definitely shows the value of choosing a good prior! When it comes to hyperparameter tuning, we often do not know ahead of time what values to concentrate around but we can try to use past experience or the work of others to inform our search spaces.

One-Line Optimization

```
In [36]: # Just because you can do it in one line doesn't mean you should!
         best = fmin(fn = lambda x: np.poly1d([1, -2, -28, 28, 12, -26, 100])(x) * 0.05,
                     space = hp.normal('x', 4.9, 0.5), algo=tpe.suggest,
                     max_evals = 2000)

         best
```

```
Out[36]: {'x': 4.878147069585637}
```

9 Conclusions

In this notebook, we saw a basic implementation of Bayesian Model-Based optimization using Hyperopt. This requires four parts:

1. Objective: what we want to minimize
2. Domain: values of the parameters over which to minimize the objective
3. Hyperparameter optimization function: how the surrogate function is built and the next values are proposed
4. Trials consisting of score, parameters pairs

The differences between random search and and Sequential Model-Based Optimization is clear: random search is *uninformed* and therefore requires more trials to minimize the objective function. The Tree Parzen Estimator, an algorithm used for SMBO, spends more time choosing the next values, but overall requires fewer evaluations of the objective function because it is able to *reason* about the next values to evaluate. Over many iterations, SMBO algorithms concentrate the search around the most promising values, yielding:

- Lower scores on the objective function
- Faster optimization

Bayesian model-based optimiziation means construction a probability model $p(y|x)$ of the objective function and updating this model as more information is collected. As the number of evaluations increases, the model (also called a surrogate function) becomes a more accurate resprresentation of the objective function and the algorithm spends more time evaluating promising values.

This notebook showed a the basic implementation of Bayesian Model-Based optimization using Hyperopt, but already we can see how it has significant advantages over random or grid search based methods. In future notebooks we will explore using Bayesian model-based optimization on more complex problems, namely hyperparameter optimization of machine learning models.