Automated Hyperparameter Method:

HyperOpt is an open-source python package that uses an algorithm called Tree-based Parzen Esimtors (TPE) to select model hyperparameters which optimize a user-defined objective function. By simply defining the functional form and bounds of each hyperparameter, TPE thoroughly yet efficiently searches through complex hyperspace to reach optimums.

TPE is a sequential algorithm that leverages bayesian updating and follows the below sequence.

1. Train a model with several sets of randomly-selected hyperparameters, returning objective function values.
2. Split our observed objective function values into "good" and "bad" groups, according to some threshold gamma (γ).
3. Calculate the "promisingness" score, which is just $P(x|good) / P(x|bad)$.
4. Determine the hyperparameters that maximize promisingness via mixture models.
5. Fit our model using the hyperparameters from step 4.
6. Repeat steps 2–5 until a stopping criteria.

Our Goal

We want to produce really good models, but do so in an efficient and ideally hands-off manner.

Our goal is to understand how to leverage algorithms to automate the model tuning process.

To help us think about that goal, let's use an analogy: we're pirates looking for buried treasure. It's also important to note that we're very efficient pirates who are looking to minimize our time searching for the buried treasure. So, how should we minimize time spent searching? The answer is **use a map!**

In figure 1, we have a fictitious map that shows where our treasure is located. After lots of climbing and digging, it wouldn't be too hard to reach that treasure because we know exactly where it's located.

But what happens when we don't have a map?

When tasked with tuning a model, we unfortunately are not given a map. Our terrain, which corresponds to the hyperparmeter search space, is unknown. Furthermore, the location of our treasure, which corresponds to the optimal set of hyperparameters, is also unknown.

With that setup, let's talk about some potential ways to efficiently explore this space and find some treasure!

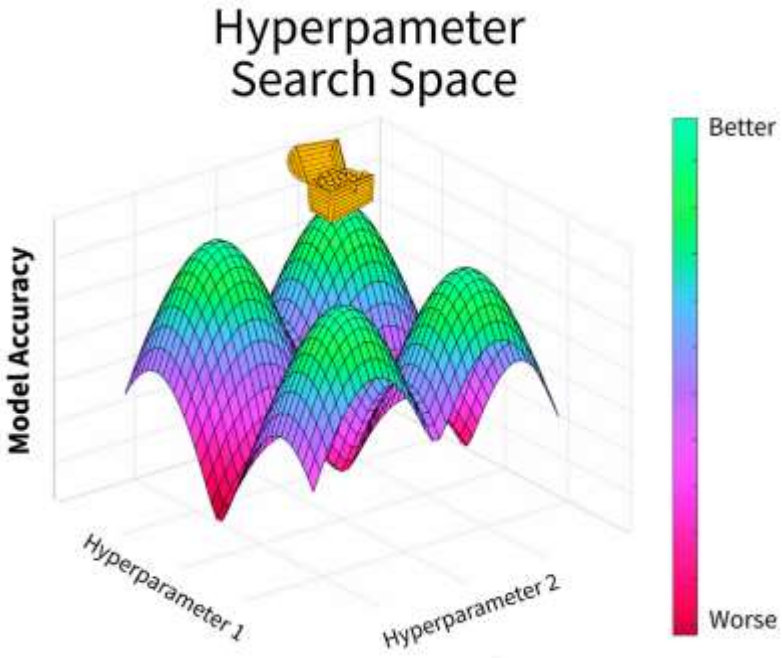

Figure 1: example 3D hyperparameter search space. The location of the treasure chest is a global optimum. Image by author.

1.2 — Potential Solutions

The original method for model tuning is "manual" — manually testing many different configurations and see which hyperparameter combination produces the best model. While informative, this process is inefficient. There must be a better way…

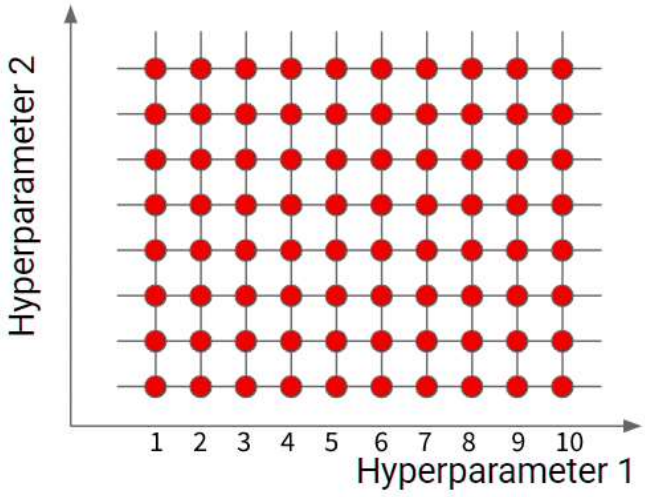

Figure 2: example of grid search layout.

1.2.1 — Grid Search (worst)

Our first optimization algorithm is grid search. Grid search iteratively tests all possible combinations of hyperparameters within a user-specified grid.

Figure 2: example of grid search layout.

For instance, in figure 2, wherever you see a red dot is where we will retrain and evaluate our model. This framework is inefficient because it **reuses bad hyperparameters**. For instance, if hyperparameter 2 has little impact on our objective function, we will still test all combinations of its values, thereby increasing the required number of iterations by 10x (in this example).In summary, **grid search is subject to the curse of dimensionality and recalculates information between evaluations, but is still used widely.**

1.2.2 — Random Search (good)

Our second algorithm is random search. Random search tries random values within a user-specified grid. Unlike with grid search, we aren't relegated to testing every possible combination of hyperparameters, which increases efficiency.

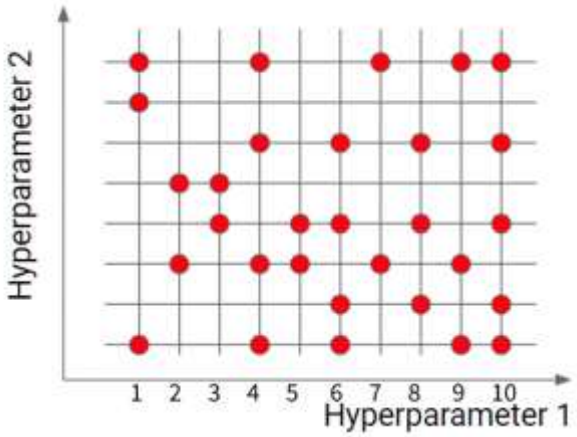

Figure 3: example of random search.

Here's a cool fact: random search will find (on average) a top 5% hyperparameter configuration within 60 iterations. That said, as with grid search you must transform your search space to reflect the functional form of each hyperparam.

Random search is a good baseline for hyperparameter optimization.

1.2.3 — Bayesian Optimization (better)

Our third candidate is our first Sequential Model-Based Optimization (SMBO) algorithm. The key conceptual difference from the prior techniques is we **iteratively use prior runs to determine future points of exploration.**

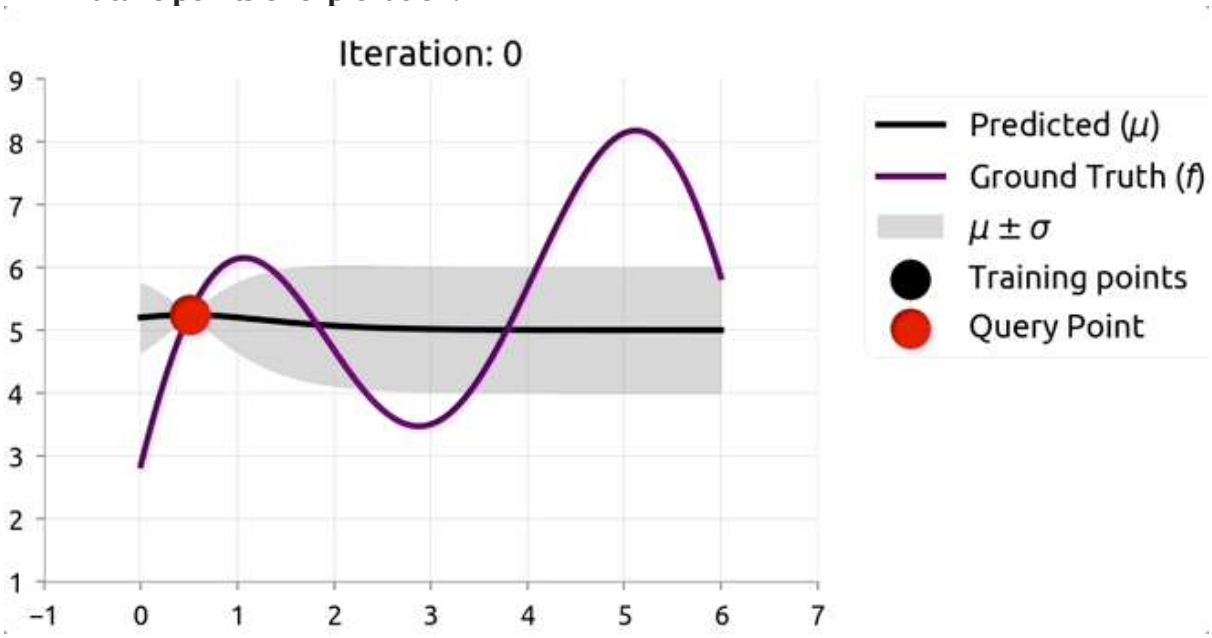

Figure 4: bayesian optimization example

Bayesian hyperparameter optimization looks to develop a probabilistic distribution of our hyperparameter search space. From there, it uses an acquisition function, such as expected improvement, to transform our hyperspace to be more "searchable." Finally, it uses an optimization algorithm, such as stochastic gradient descent, to find a the hyperparameters that maximize our acquisition function. Those hyperparameters are used to fit our model and the process is repeated until convergence.

Bayesian optimization typically outperforms random search, however it has some core limitations such as requiring numeric hyperparameters.

1.2.4 — Tree-Based Parzen Estimators (best)

Finally, let's talk about the star of the show: Tree-based Parzen Estimators (TPE). TPE is another SMBO algorithm that typically outperforms basic bayesian optimization, but the main selling point is it handles complex hyperparameter relationships via a tree structure.

```
{
    'type': 'svm',
    'C': hp.lognormal('svm_C', 0, 1),
    'kernel': hp.choice('svm_kernel', [
        {'ktype': 'linear'},
        {'ktype': 'RBF', 'width': hp.lognormal('svm_rbf_width', 0, 1)},
        ]),
},
```
Figure 5: example of hierarchical structure for TPE

Let's use figure 5 to understand this **tree structure**. Here we're training a Support Vector Machine (SVM) classifier. We will test two kernels: `linear` and RBF. A `linear` kernel does not take a width parameter

but RBF does, so by using a nested dictionary we're able to encode this structure and thereby limit the search space.

TPE also support categorical variables which traditional Bayesian optimization does not.

In general, TPE is a really robust and efficient hyperparameter optimization solution.

How does Tree-based Parzen Estimators (TPE) work?

Now that we have a general understanding of some popular hyperparameter optimization algorithms, let's do a deep dive into how TPE works.

Returning to our analogy, we're pirates looking for buried treasure **but don't have a map**. Our captain needs the treasure ASAP, so we need to dig in strategic locations that have a high probability of having treasure, using prior digs to determine the location of future digs.

1 — Initialization

To start, we **define the constraints on our space**. As noted above, our hyperparameters often have a functional form, max/min values, and hierarchical relation to other hyperparameters. Using our knowledge about our ML algorithms and our data, we can define our search space.

Next, we must **define our objective function**, which is used to evaluate how "good" our hyperparameter combination is. Some examples include classic ML loss functions, such as RMSE or AUC.

Great! Now that we have a bounded search space and a way to measure success, we're ready to start searching…

2 — Iterative Bayesian Optimization

Bayesian optimization is a sequential algorithm that finds points in hyperspace with a high probability of being "successful" according to an objective function. TPE leverages bayesian optimization but uses some clever tricks to improve performance and handle search space complexity…

2.1 The Conceptual Setup

The first trick is modeling *P(x|y)* instead of *P(y|x)*…

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

Figure 6: conditional probability that TPE looks to solve.

Bayesian optimization typically looks to model *P(y|x)*, which is the probability of an objective function value (*y*), given hyperparameters (*x*). TPE does the opposite — it looks to model *P(x|y)*, which is the probability of the hyperparameters (*x*), given the objective function value (*y*).

In short, TPE tries to find the best objective function values, then determine the associated hyperparameters.

With that very important setup, let's get into the actual algorithm.

2.2--- Split our Data into "Good" and "Bad" Groups

Remember, our goal is to find the best hyperparameter values according to some objective function. So, how can we leverage *P(x|y)* to do that?

First, TPE splits our observed data points into two groups: **good**, denoted *g(x)*, and **bad**, denoted *l(x)*. The cutoff between good and bad is determined by a user-defined parameter gamma (γ), which corresponds to the objective function percentile that splits our observations (*y**).

So, with γ = 0.5, our objective function value that splits our observations (*y**) will be the median of our observed points.

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

Bad Objective Function Vals

Good Objective Function Vals

Figure 7: breakdown of p(x|y) into two sets. Image by author.

As shown in figure 7, we can formalize *p(x|y)* using the above framework. And, to roll with the pirate analogy…

Pirate Perspective: looking at the places we've already explored, l(x) lists places with very little treasure and g(x) lists places with lots of treasure.

2.3— Calculate the "Promisingness" Score

Second, TPE defines how we should evaluate an unobserved hyperparameter combination via the **"promisingness" score**.

$$P = \frac{g(x)}{l(x)} = \frac{P(x|good)}{P(x|bad)}$$

Figure 8: promisingness score definition. Image by author.

Figure 8 defines our promisingness score (P), which is just a ratio with the following components…

- **Numerator**: the probability of observing a set of hyperparameters (x), given the corresponding objective function value is "**good**."
- **Denominator**: the probability of observing a set of hyperparameters (x), given the corresponding objective function value is "**bad**."

The bigger the "promisingness" value, the more likely that our hyperparameters *x* will produce a "good" objective function.

Pirate Perspective: promisingness shows how likely a given location in our terrain will be to have lots of treasure.

2.4— Create a Probability Density Estimates

Third, TPE looks to evaluate the "promisingness" score via mixture of models. **The idea of mixture models is to take multiple probability distributions and put them together using a linear combination.** These combined probability distributions are then used to develop probability density estimates.

Generally, the mixture modeling process is…

1. **Define the distribution type of our points.** In our case, if our variable is categorical we use a re-weighted categorical distribution and if its numeric we use a gaussian (i.e. normal) or uniform distribution.
2. **Iterate over each point and insert a distribution at that point.**
3. **Sum the mass of all distributions to get a probability density estimate.**

Note that this process is run individually for both sets $l(x)$ and $g(x)$.

Let's walk through an example in figure 9…

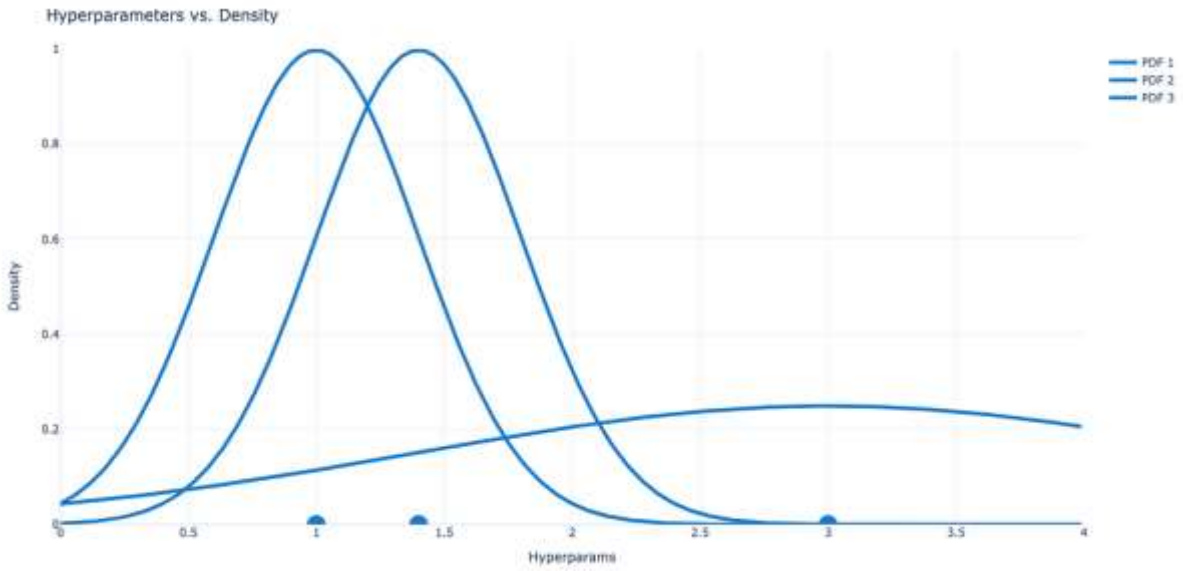

Figure 9: example of truncated gaussian distributions fit to 3 hyperparameter observations. Image by author.

For each observation (blue dots on the x-axis), we create a normal distribution ~N(μ, σ), where…

- **μ (mu)** is the mean of our normal distribution. It's value is the location of our point along the x-axis.
- **σ (sigma)** is the standard deviation of our normal distribution. Its value is the distance to the closest neighboring point**.**

If points are close together, the standard deviation will be small and thereby the distribution will be very tall and conversely, if points are spread apart, the distribution will be flat (figure 10)…

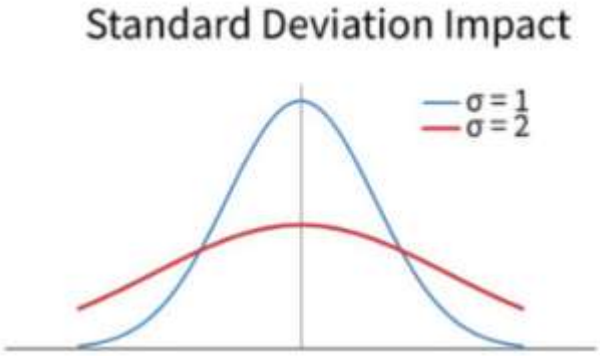

Figure 10: example of the impact of standard deviation on the shape of a normal distribution.

Pirate Perspective: NA — pirates aren't great with mixture models.

Another quick aside before moving on: if you're reading the literature you'll notice that TPE uses "truncated" gaussians, which simply means the gaussians are bounded by the range we specify in our hyperparameter configuration instead of extending to +/- infinity.

2.5 — Determining the Next Point to Explore!

Let's bring these pieces together. So far, we've 1) acquired objective function observations, 2) defined our "promisingness" formula, and 3) created a probability density estimate via mixture models based on prior values. We have all the pieces to evaluate a given point!

Our first step is to create an average probability density function (PDF) for both *g(x)* and *l(x)*.

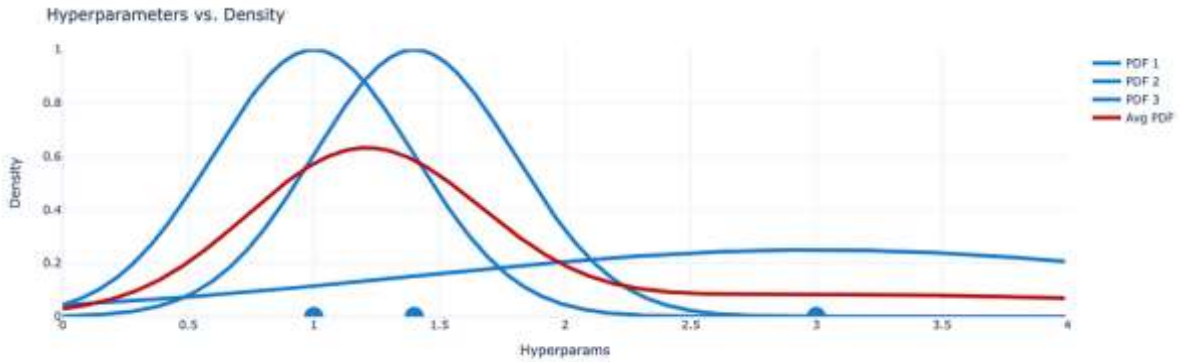

Figure 11: overlay of the average probability density given 3 observed points.

An example process is shown in figure 11 — the red line is our average PDF and is simply the sum of all PDFs divided by the number of PDFs.

Using the average PDF, we can get the probability of any hyperparameter value (*x*) being in *g(x)* or *l(x)*.

For instance, let's say observed values in figure 11 belong to the "good" set, *g(x)*. Based on our average PDF, it's unlikely that a hyperparameter value of 3.9 or 0.05 belong to the "good" set. Conversely, a hyperparameter value of ~1.2 seems to be very likely to belong to the "good" set.

Now this is just one half of the picture. We apply the same methodology for the "bad" set, *l(x)*. Since we're looking to maximize *g(x) / l(x)*, **promising points should be located where *g(x)* is high and *l(x)* is low.**

Pretty cool, right?

With these probability distributions, we can sample from our tree-structured hyperparameters and find the set of hyperparameters that maximize "promisingness" and are thereby worth exploring.

Pirate Perspective: the next location we dig is the location that maximizes the (probability of having lots of treasure) / (probability of having little treasure).

Summary

Hyperparameter tuning is a necessary part of the ML model lifecycle, but is time consuming. Sequential Model-Based Optimization (SMBO) algorithms excel at searching complex hyperspaces for optimums, and they can be applied to hyperparameter tuning. Tree-based Parzen Estimators (TPE) is a very efficient SMBO and outperforms both Bayesian Optimization and Random Search.

TPE repeats the below steps until a stopping criteria:

1. Divide observed points into “good” and “bad” sets, according to some hyperparameter, gamma.
2. Fit a mixture model to both the “good” and “bad” set to develop an average probability density estimate.
3. Select the point that maximizes the “promisingness” score, which leverages step 2 to estimate the probability of being in the “good” and “bad” sets.

Finally, we have a really cool code snippet that shows how to parallelize HyperOpt via SparkTrials. It also logs all of our iterations to MLflow.