Introducing Random Search

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

What you already know

Very similar to grid search:

- Define an estimator, which hyperparameters to tune and the range of values for each hyperparameter.
- We still set a cross-validation scheme and scoring function

BUT we instead *randomly* select grid squares.

Why does this work?

Bengio & Bergstra (2012):

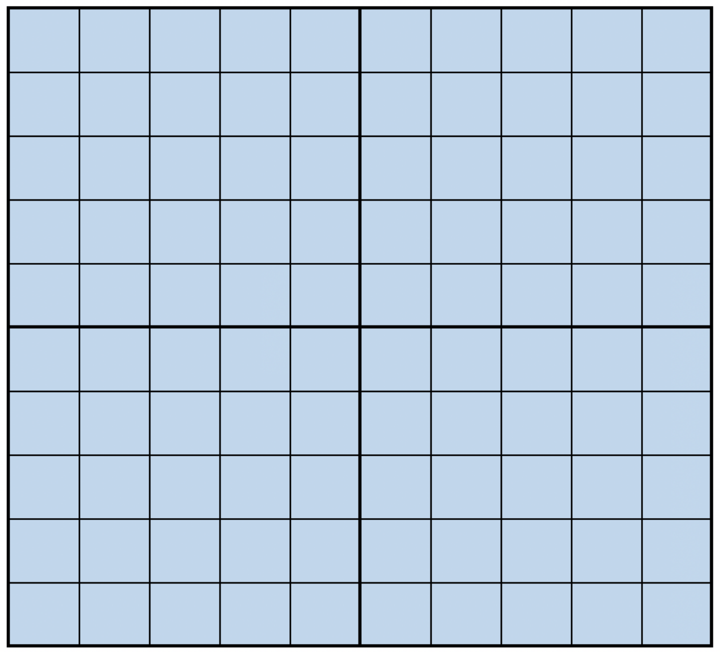
> This paper shows empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid.

Two main reasons:

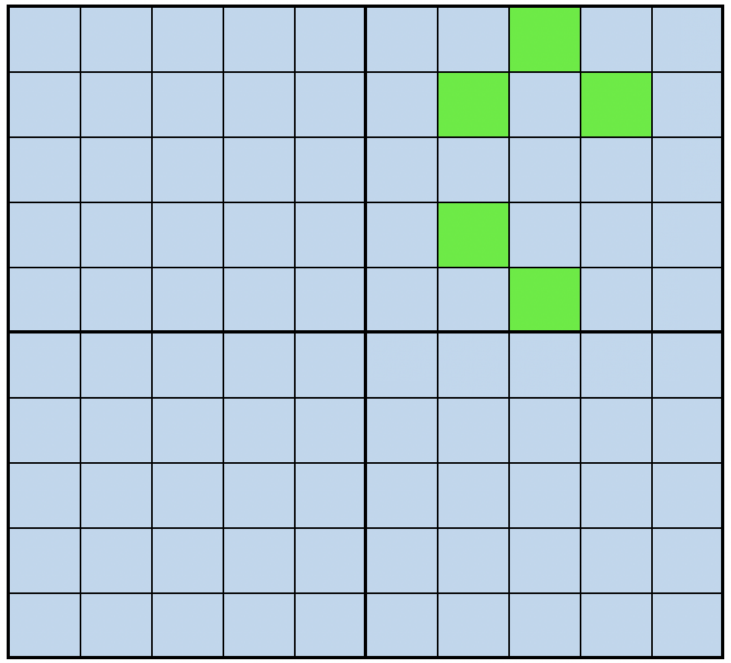
1. Not every hyperparameter is as important
2. A little trick of probability

A probability trick

A grid search:



Our best models:



How many models must we run to have a 95% chance of getting one of the green squares?

A probability Trick

If we randomly select hyperparameter combinations uniformly, let's consider the chance of MISSING every single trial, to show how unlikely that is

- Trial 1 = 0.05 chance of success and (1 - 0.05) of missing
    - Trial 2 = (1-0.05) x (1-0.05) of missing the range
        - Trial 3 = (1-0.05) x (1-0.05) x (1-0.05) of missing again
- In fact, with n trials we have (1-0.05)^n chance that every single trial misses that desired spot.

A probability trick

So how many trials to have a high (95%) chance of getting **in** that region?

- We have (1-0.05)^n chance to miss everything.
- So we must have (1- miss everything) chance to get in there or (1-(1-0.05)^n)
- Solving 1-(1-0.05)^n >= 0.95 gives us **n >= 59**

A probability trick

What does that all mean?

- You are unlikely to keep completely missing the 'good area' for a long time when randomly picking new spots
- A grid search may spend lots of time in a 'bad area' as it covers exhaustively.

Some important notes

Remember:

1. The maximum is still only as good as the grid you set!
2. Remember to fairly compare this to grid search, you need to have the same modeling 'budget'

Creating a random sample of hyperparameters

We can create our own random sample of hyperparameter combinations:

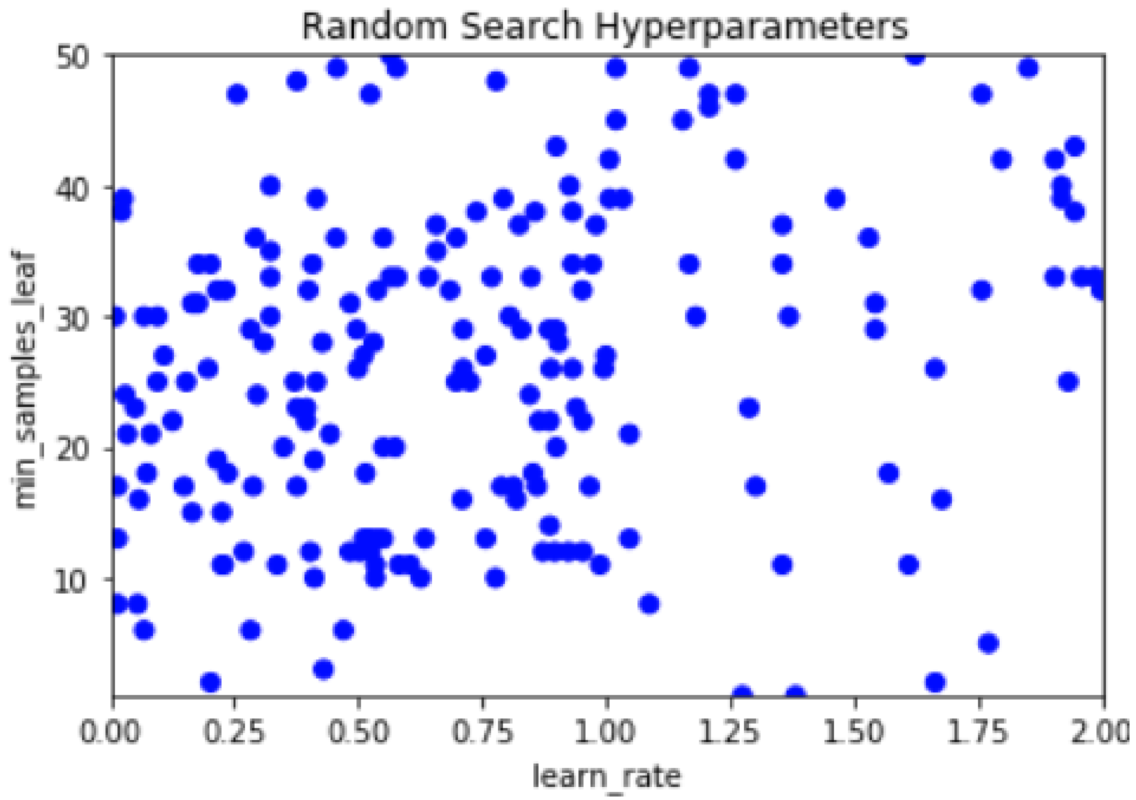
```
# Set some hyperparameter lists
learn_rate_list = np.linspace(0.001,2,150)
min_samples_leaf_list = list(range(1,51))
```

```
# Create list of combinations
from itertools import product
combinations_list = [list(x) for x in
                    product(learn_rate_list, min_samples_leaf_list)]
```

```
# Select 100 models from our larger set
random_combinations_index = np.random.choice(
                            range(0,len(combinations_list)), 100,
                            replace=False)
combinations_random_chosen = [combinations_list[x] for x in
                            random_combinations_index]
```

Visualizing a Random Search

We can also visualize the random search coverage by plotting the hyperparameter choices on an X and Y axis.



Notice how this has a wide range of the scatter but not deep coverage?

Let's practice!

HYPERPARAMETER TUNING IN PYTHON

Random Search in Scikit Learn

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

Comparing to GridSearchCV

We don't need to reinvent the wheel. Let's recall the steps for a Grid Search:

1. Decide an algorithm/estimator
2. Defining which hyperparameters we will tune
3. Defining a range of values for each hyperparameter
4. Setting a cross-validation scheme; and
5. Define a score function
6. Include extra useful information or functions

Comparing to Grid Search

There is only one difference:

- Step 7 = Decide how many samples to take (then sample)

That's it! (mostly)

Comparing Scikit Learn Modules

The modules are similar too:

GridSearchCV:

```
sklearn.model_selection.GridSearchCV(estimator, param_grid,
        scoring=None, fit_params=None,
        n_jobs=None,
        iid='warn',
        refit=True, cv='warn', verbose=0,
        pre_dispatch='2*n_jobs',
        error_score='raise-deprecating',
        return_train_score='warn')
```

RandomizedSearchCV:

```
sklearn.model_selection.RandomizedSearchCV(estimator,
        param_distributions, n_iter=10,
        scoring=None, fit_params=None,
        n_jobs=None, iid='warn', refit=True,
        cv='warn', verbose=0,
        pre_dispatch='2*n_jobs',
        random_state=None,
        error_score='raise-deprecating',
        return_train_score='warn')
```

Key differences

Two key differences:

- `n_iter` which is the number of samples for the random search to take from your grid. In the previous example you did 300.
- `param_distributions` is slightly different from `param_grid` , allowing optional ability to set a distribution for sampling.
    - The default is all combinations have equal chance to be chosen.

Build a RandomizedSearchCV Object

Now we can build a random search object just like the grid search, but with our small change:

```
# Set up the sample space
learn_rate_list = np.linspace(0.001,2,150)
min_samples_leaf_list = list(range(1,51))

# Create the grid
parameter_grid = {
    'learning_rate' : learn_rate_list,
    'min_samples_leaf' : min_samples_leaf_list}

# Define how many samples
number_models = 10
```

Build a RandomizedSearchCV Object

Now we can build the object

```
# Create a random search object
random_GBM_class = RandomizedSearchCV(
    estimator = GradientBoostingClassifier(),
    param_distributions = parameter_grid,
    n_iter = number_models,
    scoring='accuracy',
    n_jobs=4,
    cv = 10,
    refit=True,
    return_train_score = True)
# Fit the object to our data
random_GBM_class.fit(X_train, y_train)
```

Analyze the output

The output is exactly the same!

How do we see what hyperparameter values were chosen?

The `cv_results_` dictionary (in the relevant `param_` columns)!

Extract the lists:

```
rand_x = list(random_GBM_class.cv_results_['param_learning_rate'])
rand_y = list(random_GBM_class.cv_results_['param_min_samples_leaf'])
```

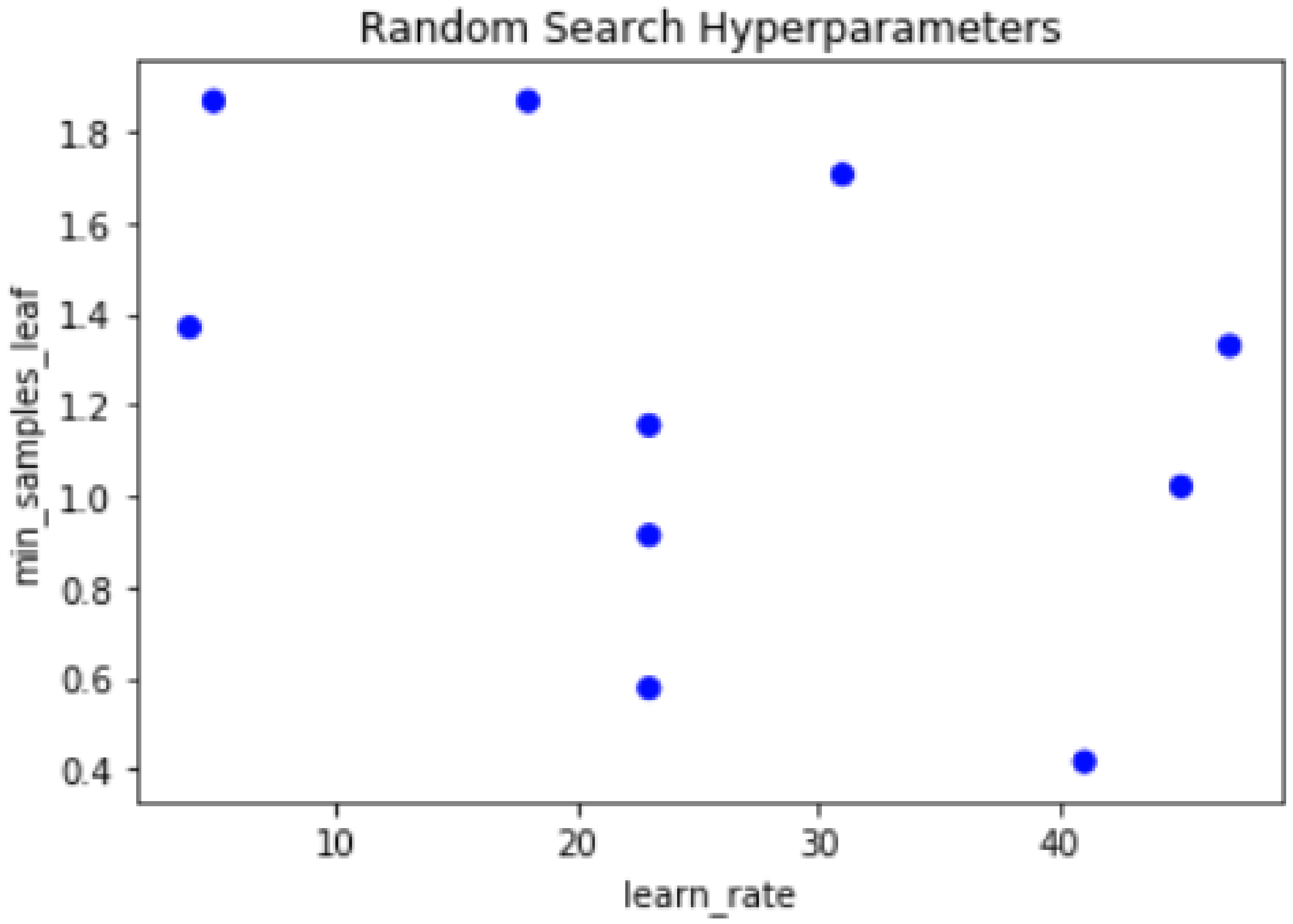
Analyze the output

Build our visualization:

```
# Make sure we set the limits of Y and X appriately
x_lims = [np.min(learn_rate_list), np.max(learn_rate_list)]
y_lims = [np.min(min_samples_leaf_list), np.max(min_samples_leaf_list)]
# Plot grid results
plt.scatter(rand_y, rand_x, c=['blue']*10)
plt.gca().set(xlabel='learn_rate', ylabel='min_samples_leaf',
                title='Random Search Hyperparameters')
plt.show()
```

Analyze the output

A similar graph to before:



Let's practice!

HYPERPARAMETER TUNING IN PYTHON

Comparing Grid and Random Search

HYPERPARAMETER TUNING IN PYTHON

Alex Scriven
Data Scientist

What's the same?

Similarities between Random and Grid Search?

- Both are automated ways of tuning different hyperparameters
- For both you set the grid to sample from (which hyperparameters and values for each)

Remember to think carefully about your grid!

- For both you set a cross-validation scheme and scoring function

What's different?

Grid Search:

- Exhaustively tries all combinations within the sample space
- No Sampling methodology
- More computationally expensive
- Guaranteed to find the best score in the sample space

Random Search:

- Randomly selects a subset of combinations within the sample space (that you must specify)
- Can select a sampling methodology (other than uniform which is default)
- Less computationally expensive
- Not guaranteed to find the best score in the sample space (but likely to find a *good* one *faster*)

Which should I use?

So which one should I use? What are my considerations?

- How much data do you have?
- How many hyperparameters and values do you want to tune?
- How much resources do you have? (Time, computing power)

- More data means random search may be better option.
- More of these means random search may be a better option.
- Less resources means random search may be a better option.

Let's practice!

HYPERPARAMETER TUNING IN PYTHON