Automatic Hyperparameter Tuning with Sklearn Using Grid and Random Search

Deep dive into GridSearch and RandomSearch classes of Scikit-learn

Bex T. Mar 5 · 5 min read ⋆

Photo by Helena Lopes on Pexels

What is a hyperparameter?

Today, algorithms that hide a world of math under the hood can be trained with only a few lines of code. Their success depends first on the data trained and then, on what hyperparameters were used by the user. So, what are these hyperparameters?

Hyperparameters are user-defined values like *k* in kNN and *alpha* in Ridge and Lasso regression. They strictly control the fit of the model and this means, for each dataset, there is a unique set of optimal hyperparameters to be found. The most basic way of finding this perfect set would be randomly trying out different values based on gut feeling. However, as you might guess, this method quickly becomes useless when there are many hyperparameters to tune.

Instead, today you will learn about two methods for automatic hyperparameter tuning: Random search and Grid search. Given a set of possible values for all hyperparameters of a model, a Grid search fits a model using every single combination of these hyperparameters. What is more, in each fit, the Grid search uses cross-validation to account for overfitting. After all combinations are tried, the search retains the parameters that resulted in the best score so that you can use them to build your final model.

Random search takes a bit different approach than Grid. Instead of exhaustively trying out every single combination of hyperparameters, which can be computationally expensive and time-consuming, it randomly samples hyperparameters and tries to get closer to the best set.

Fortunately, Scikit-learn provides `GridSearchCV` and `RandomizedSearchCV` classes that make this process a breeze. Today, you will learn all about them!

Prepping the Data

We will be tuning a RandomForestRegressor model on the Iowa housing dataset. I chose Random Forests because it has large enough hyperparameters that make this guide more informative but the process you will be learning can be applied to any model in the Sklearn API. So, let's start:

```
houses_train = pd.read_csv("data/train.csv")
houses_test = pd.read_csv("data/test.csv")

houses_train.head()
```

4701.py hosted with ♡ by GitHub view raw

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	...	MiscVal	MoSold	YrSold	SaleType	SaleCondition	SalePrice
0	1	60	RL	65.0	8450	Pave	...	0	2	2008	WD	Normal	208500
1	2	20	RL	80.0	9600	Pave	...	0	5	2007	WD	Normal	181500
2	3	60	RL	68.0	11250	Pave	...	0	9	2008	WD	Normal	223500
3	4	70	RL	60.0	9550	Pave	...	0	2	2006	WD	Abnorml	140000
4	5	60	RL	84.0	14260	Pave	...	0	12	2008	WD	Normal	250000

5 rows в 81 columns

The target is `SalePrice`. For simplicity, I will choose only numeric features:

```
X = houses_train.select_dtypes(include="number").drop("SalePrice", axis=1)
y = houses_train.SalePrice

X_test = houses_test.select_dtypes(include="number")
```

4702.py hosted with ♡ by GitHub view raw

First, both training and test sets contain missing values. We will use `SimpleImputer` to deal with them:

```
from sklearn.impute import SimpleImputer

# Impute both train and test sets
imputer = SimpleImputer(strategy="mean")
X = imputer.fit_transform(X)
X_test = imputer.fit_transform(X_test)
```

4703.py hosted with ♡ by GitHub view raw

Now, let's fit a base `RandomForestRegressor` with default parameters. As we will use the test set only for final evaluation, I will create a separate validation set using the training data:

```
%%time

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.3)

# Fit a base model
forest = RandomForestRegressor()

_ = forest.fit(X_train, y_train)

>>> print(f"R2 for training set: {forest.score(X_train, y_train)}")
>>> print(f"R2 for validation set: {forest.score(X_valid, y_valid)}\n")

R2 for training set: 0.9785951576271396
R2 for validation set: 0.832622375495487

Wall time: 1.71 s
```

4704.py hosted with ♡ by GitHub view raw

> *Note: The main focus of this article is on how to perform hyperparameter tuning. We won't worry about other topics like overfitting or feature engineering but only narrow down on how to use Random and Grid search so that you can apply automatic hyperparameter tuning in real-life setting.*

We got a 0.83 for R2 on the test set. We fit the regressor only with default parameters which are:

```
>>> forest.get_params()

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

4705.py hosted with ♡ by GitHub view raw

That's a lot of hyperparameters. We won't be tweaking all of them but focus only on the most important ones. Specifically:

- `n_esimators` - number of trees to be used
- `max_feauters` - the number of features to use at each node split
- `max_depth` : the number of leaves on each tree
- `min_samples_split` : the minimum number of samples required to split an internal node
- `min_samples_leaf` : the minimum number of samples in each leaf
- `bootstrap` : method of sampling - with or without replacement.

Both Grid Search and Random Search tries to find the optimal values for each of these hyperparameters. Let's see this in action first with Random Search.

Randomized Search with Sklearn RandomizedSearchCV

Scikit-learn provides `RandomizedSearchCV` class to implement random search. It requires two arguments to set up: an estimator and the set of possible values for hyperparameters called a *parameter grid* or *space*. Let's define this parameter grid for our random forest model:

```
n_estimators = np.arange(100, 2000, step=100)
max_features = ["auto", "sqrt", "log2"]
max_depth = list(np.arange(10, 100, step=10)) + [None]
min_samples_split = np.arange(2, 10, step=2)
min_samples_leaf = [1, 2, 4]
bootstrap = [True, False]

param_grid = {
    "n_estimators": n_estimators,
    "max_features": max_features,
    "max_depth": max_depth,
    "min_samples_split": min_samples_split,
    "min_samples_leaf": min_samples_leaf,
    "bootstrap": bootstrap,
}

>>> param_grid

{'n_estimators': array([ 100,  200,  300,  400,  500,  600,  700,  800,  900, 1000, 1100,
        1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900]),
 'max_features': ['auto', 'sqrt', 'log2'],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, None],
 'min_samples_split': array([2, 4, 6, 8]),
 'min_samples_leaf': [1, 2, 4],
 'bootstrap': [True, False]}
```

4706.py hosted with ♡ by GitHub view raw

This parameter grid dictionary should have hyperparameters as keys in the syntax they appear in the model's documentation. The possible values can be given as an array.

Now, let's finally import `RandomizedSearchCV` from `sklearn.model_selection` and instantiate it:

```
from sklearn.model_selection import RandomizedSearchCV
```

```
2
3  forest = RandomForestRegressor()
4
5  random_cv = RandomizedSearchCV(
6      forest, param_grid, n_iter=100, cv=3, scoring="r2", n_jobs=-1
7  )
```

4707.py hosted with ♡ by GitHub view raw

Apart from the accepted estimator and the parameter grid, it has `n_iter` parameter. It controls how many iterations of random picking of hyperparameter combinations we allow in the search. We set it to 100, so it will randomly sample 100 combinations and return the best score. We are also using 3-fold cross-validation with the coefficient of determination as scoring which is the default. You can pass any other scoring function from `sklearn.metrics.SCORERS.keys()`. Now, let's start the process:

> *Note, since Randomized Search performs cross-validation, we can fit it on the training data as a whole. Because of how CV works, it will create separate sets for training and evaluation. Also, I am setting* `n_jobs` *to -1 to use all cores on my machine.*

```
1  %%time
2
3  _ = random_cv.fit(X, y)
4
5  >>> print("Best params:\n")
6  >>> print(random_cv.best_params_)
7
8  Best params:
9
10 {'n_estimators': 800, 'min_samples_split': 4,
11 'min_samples_leaf': 1, 'max_features': 'sqrt',
12 'max_depth': 20, 'bootstrap': False}
13
14 Wall time: 16min 56s
```

4708.py hosted with ♡ by GitHub view raw

After ~17 minutes of training, the best params found can be accessed with `.best_params_` attribute. We can also see the best score:

```
>>> random_cv.best_score_

0.8690868090696587
```

We got around 87% coefficient of determination which is an improvement of 4% over the base model.

Sklearn GridSearchCV

You should never choose your hyperparameters according to the results of the `RandomSearchCV`. Instead, only use it to narrow down the value range for each hyperparameter so that you can provide a better parameter grid to `GridSearchCV`.

Why not use `GridSearchCV` right from the beginning, you ask? Well, looking at the initial parameter grid:

```
n_iterations = 1

for value in param_grid.values():
    n_iterations *= len(value)

>>> n_iterations

13680
```

4709.py hosted with ♡ by GitHub view raw

There are 13680 possible hyperparam combinations and with a 3-fold CV, the GridSearchCV would have to fit Random Forests 41040 times. Using `RandomizedGridSearchCV`, we got reasonably good scores with just 100 * 3 = 300 fits.

Now, time to create a new grid building on the previous one and feed it to `GridSearchCV`:

```
new_params = {
    "n_estimators": [650, 700, 750, 800, 850, 900, 950, 1000],
    "max_features": ['sqrt'],
    "max_depth": [10, 15, 20, 25, 30],
    "min_samples_split": [2, 4, 6],
    "min_samples_leaf": [1, 2],
    "bootstrap": [False],
}
```

4710.py hosted with ♡ by GitHub view raw

This time we have:

```
n_iterations = 1

for value in new_params.values():
    n_iterations *= len(value)

>>> n_iterations

240
```

4711.py hosted with ♡ by GitHub view raw

240 combinations which is still a lot but we will go with it. Let's import `GridSearchCV` and instantiate it:

```
from sklearn.model_selection import GridSearchCV

forest = RandomForestRegressor()

grid_cv = GridSearchCV(forest, new_params, n_jobs=-1)
```

4712.py hosted with ♡ by GitHub view raw

I didn't have to specify `scoring` and `cv` because we were using the default settings so don't have to specify. Let's fit and wait:

```
%%time

_ = grid_cv.fit(X, y)

```

```
print('Best params:\n')
print(grid_cv.best_params_, '\n')
Best params:

{'bootstrap': False, 'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_sample

Wall time: 35min 18s
```

4713.py hosted with ♡ by GitHub view raw

After 35 minutes, we get the above scores, this time — truly the most optimal scores. Let's see how much they differ from `RandomizedSearchCV`:

```
>>> grid_cv.best_score_

0.8696576413066612
```

Are you surprised? Me too. The difference in results is marginal. However, this might be just a specific case to the given dataset.

When you have computation-heavy models in practice, it is best to get the results of random search and validate them in grid search within a tighter range.

Conclusion

At this point, you might be thinking that all this is great. You got to learn tuning models without even giving a second glance at what actually parameters do and still find their optimal values. But this automation comes at a great cost: it is both computation-heavy and time-consuming.

You might be okay to wait a few minutes for it to finish as we did here. But, our dataset had only 1500 samples. Still, finding the best parameter took us almost an hour if you combine both grid and random searches. Imagine how much you have to wait for large datasets out there.

So, Grid search and Random search for smaller datasets? Hands-down yes! For large datasets, you need to take a different approach. Fortunately, 'the different approach' is already covered by Scikit-learn… again. That's why my next post is going to be on `HalvingGridSearchCV` and `HalvingRandomizedSearchCV`. Stay tuned!