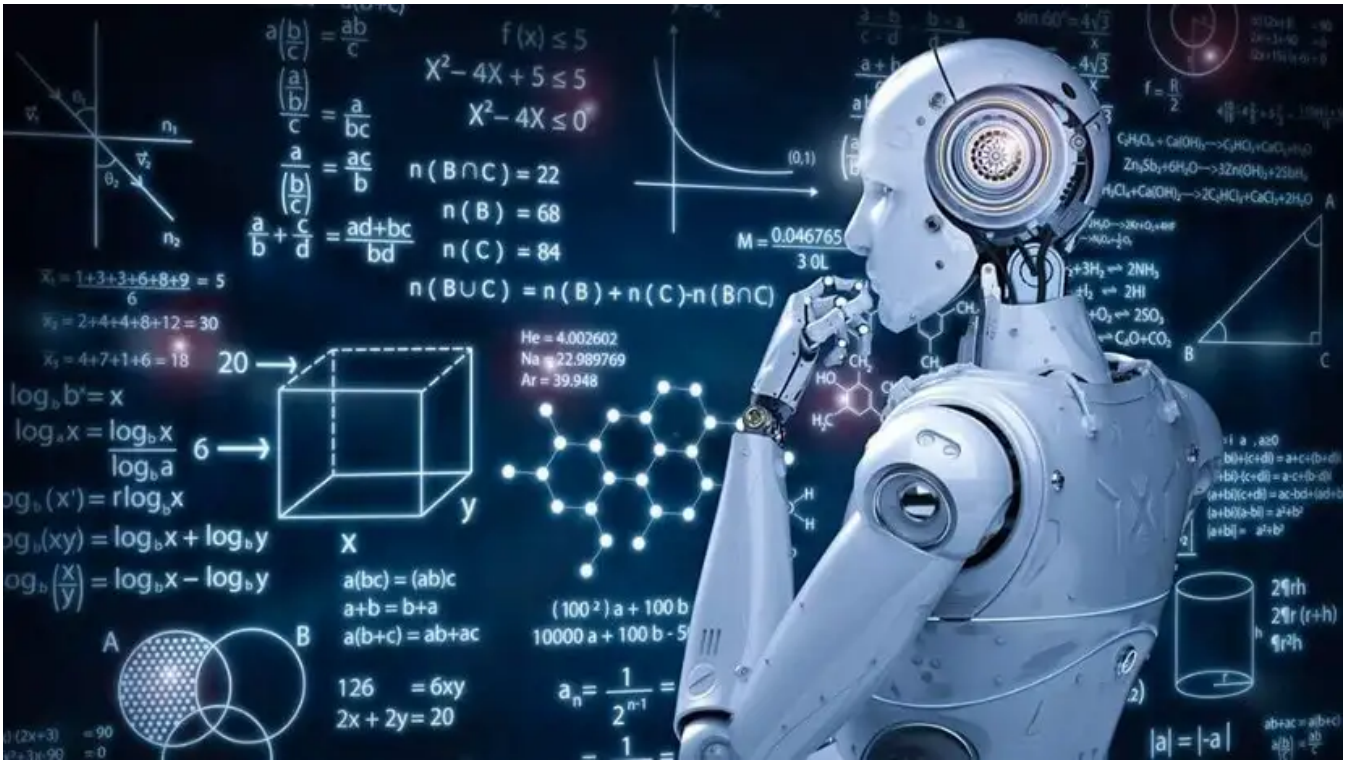
:ware Engineering

Ruan de Bruyn Follow

Nov 15, 2021 · 9 min read · Listen



How to do hyper-parameter tuning for your Python scikit learn models

A frequent obstacle with putting together machine learning models is how to tweak them to be *just* right. ML algorithms can always be tweaked to alter their behaviour. A good example of this is the regularization parameter C in an SVM model. Higher values of C incentivize the model to fit the training data more tightly, but lead to overfitting if you make it too high (the noisier the data, the worse it gets).

Lower values are more toleran[illegible] the SVM favour a simpler decision boundary, which may [illegible] generalize better to data it

hasn't seen before. Of course, there's a catch. If C is too low, your model could underfit, and not perform well on the training set or the test set.

For the uninitiated, these kinds of parameters are called hyper-parameters for machine learning models. Hyper-parameters influence model performance, but are not learned as part of the training process for the model. So, how do we pick the best ones? How do we optimize our models? That's what this post is all about.

Clone the Python project of this blog at https://github.com/rdbruyn/dvt-optimise if you'd like, and let's get into it.

Optimizing with Exhaustive Grid Search

Say we want to optimize an SVM model, with a proper value of C. A naive approach may be to use inspection; manually set a few values of C, and see how it goes. We can try 0.5, 1, and 2. That's not so bad, right? We'll have to train the model 3 times. Not too much of a hassle. That's only one of the hyper-parameters we have available to us, however. There are other ones, like kernel type and tolerance. The kernel dictates what kind of decision boundary the SVM trains, and the tolerance tells the model when it's done training. Let's say we want to optimize these hyper-parameters along with C. We want to try 6 values of C, 4 different kernels, and 2 values of tolerance. That's 6x4x2=48 combinations to try. Somewhat more cumbersome, yes? Fortunately, scikit-learn comes with a way to check model performance with different hyper-parameters, called `GridSearchCV`. It's an exhaustive grid search algorithm, which means that it tries all combinations, and lets you know how it went. In a Python script called `hyperparam_tuning.py`, let's do the following:

```python
from sklearn import svm
from time import time
from pandas import DataFrame
from pathlib import Path

from sklearn.model_selection import GridSearchCV

# make sure that result directory exists before running any of the functions
result_dir = Path.cwd() / 'results'
if not result_dir.exists():
    result_dir.mkdir()


def tune_with_grid_search(x_train, y_train, param_grid):
    svc = svm.SVC()

    start = time()
    gs_results = GridSearchCV(svc, param_grid, cv=5).fit(x_train, y_train)
    duration = time() - start

    results = DataFrame(gs_results.cv_results_)
    results.loc[:, 'mean_test_score'] *= 100
    results.to_csv(result_dir / 'svc_results.csv')

    # take the most relevant columns and sort (for readability)
    results = results.loc[:, ('rank_test_score', 'mean_test_score', 'params')]
    results.sort_values(by='rank_test_score', ascending=True, inplace=True)

    return results, duration
```

hyperparam_tuning_beta.py hosted with ❤ by GitHub view raw

In the code above, we use scikit-learn's GridSearchCV class to do the heavy lifting for us. The `param_grid` argument is a dictionary with all the hyper-parameters we want to try (we'll get to that in a second). The `GridSearchCV` object just tries all the combinations and keeps the results, so we can see which parameters do best. It does this by using crossvalidation. In a nutshell, crossvalidation is a way of *kinda* seeing how well a model performs, without actually involving the test set.

In our case, we're using 5-fold crossvalidation (`cv=5`). So the model is trained 5 times, and the training data set is divided into 5 splits. Every iteration the model trains is on 4/5 splits. Then, the "test score" is taken on the remaining split. After training like this 5 times with the different split combinations, the model is scored by its average accuracy. While you certainly can test model performance on the test

set instead of crossvalidation, this is considered cheating. By doing that, you "leak" knowledge of the test set to the model by using the test set for tuning. Exactly how harmful this is depends on your data, but in principle, you may end up tuning your model so much to get great testing accuracy, that your model overfits on the test data, and ends up not generalizing well (defeating the purpose of having a test set to begin with). By using parts of your training data as a validation set in crossvalidation, you get a rough idea of how well a model performs, without actually exposing it to the test set.

We save the data in a pandas `DataFrame`, and then save it to CSV. There's a lot of different data in there, which you can check out yourself. I filter the columns for the most relevant ones and sort them so the best results show up first. After you find the hyper-parameters that perform best during crossvalidation, you can train the model on the test set, and see how it goes. Let's go ahead and do that in a file called `main.py`:

```python
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

from hyperparam_tuning import tune_with_grid_search

RANDOM_STATE = 35090


def main():
    # use fixed random state for repeatable data set
    X, Y = make_classification(n_samples=3000, random_state=RANDOM_STATE)
    x_train, x_test, y_train, y_test = train_test_split(X, Y, random_state=RANDOM_STATE)

    svc_params = {
        'C': [0.1, 0.5, 1, 2, 5, 10],
        'kernel': ['linear', 'rbf', 'sigmoid', 'poly'],
        'tol': [1e-3, 1e-2]
    }

    gs_results, gs_duration = tune_with_grid_search(x_train, y_train, svc_params)

    print(gs_results.head())

    score = gs_results['mean_test_score'].iloc[0]
    params = gs_results['params'].iloc[0]

    svc = svm.SVC(**params)
    svc.fit(x_train, y_train)
    accuracy = accuracy_score(y_test, svc.predict(x_test))

    print(f'Best score for GridSearchCv is {score:.3f}, took {gs_duration:.2f} seconds')
    print(f'Params: {params}')
    print(f'Corresponding test accuracy: {accuracy * 100:.2f}%\n')


if __name__ == '__main__':
    main()

```

dvt_optimise_main_1.py hosted with ❤ by GitHub view raw

Okay, first things first. We start by making our very own dataset using `make_classification`. It generates 3000 samples of data with 20 features that are clustered in a normal distribution. Check out the documentation for the specifics. All we need to know is, it's a binary classification problem, and we don't have to

worry about scaling or normalizing the data for our models. This is then split into training and test sets, as is tradition. We then feed it our parameter grid, testing 48 different combinations of hyper-parameters. Running this, my terminal output looks as such:

```
~/Documents/DVT blogs/dvt-optimise main !1 ?2
❯ python main.py
   rank_test_score  mean_test_score                                                params
4                1        86.444444  {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.001}
2                2        86.400000      {'C': 0.1, 'kernel': 'rbf', 'tol': 0.001}
3                2        86.400000       {'C': 0.1, 'kernel': 'rbf', 'tol': 0.01}
5                4        86.355556  {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.01}
6                4        86.355556     {'C': 0.1, 'kernel': 'poly', 'tol': 0.001}
```

As mentioned above, there are many more columns in the grid search results, these are the ones I filtered as they are the most relevant. It looks like our best performing model had crossvalidation accuracy of about 86.44%, with hyper-parameters as shown. The second part of the output is below:

```
Best score for GridSearchCv is 86.444, took 17.71 seconds
Params: {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.001}
Corresponding test accuracy: 85.60%
```

So the grid search algorithm took 17.71 seconds to train all the various combinations, and the best one had a test set accuracy of 85.6%, which is close to the crossvalidation accuracy. I'll mention again, crossvalidation accuracy has nothing to do with test set accuracy. A good crossvalidation accuracy is an indicator that a model may have good test set accuracy, but you'll have to see the test set accuracy to confirm this. The two measures are taken on different parts of the dataset.

So. This is all good and well. But maybe a little slow. We trained 48 different models on the full dataset. On my machine, giving the dataset 10 000 samples takes almost 2 minutes to optimize. The time taken to do this ramps up pretty quickly, the more data you have, and the more combinations you have in your hyper-parameter grid. Many datasets for real-world applications could have you training a model for hours or days. Imagine our SVM took an hour to train on such a dataset. Doing that, but times 48, will have you waiting two days to find the best model. Fortunately, an experimental feature from scikit-learn can cut down on training time.

Introducing HalvingGridSearchCV

This relatively new grid search implementation in the scikit-learn library, called *successive halving grid search*, can find optimal parameters fairly quickly, but not quite with the same diligence as `GridSearchCV`. It does this by training all the combinations in your parameter grid but on a small subset of the training data. Once this is done, it only takes the best ones. Then, it retrains those best ones, but with a bit more data. Then it takes the best ones out of those and trains them on more data once again. This is a rinse and repeat process, and only the last iteration uses the entire dataset for crossvalidation. So instead of going all out on every combination, it just kind of roughly checks out the combination to gauge how good it is. The assumption is that if one model is better than another on 100 points of data, surely it must be better when you use 3000 data points. This is not a perfect assumption, but it makes for a decent heuristic. That's why it picks the best combinations at the end of every iteration and then tries again, but with more data. Check out the documentation and this example page to take a closer look. Let's see how it works. In the file we made earlier, `hyper_param_tuning.py`, add the following code:

```
from sklearn import svm
from time import time
from pandas import DataFrame

from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV

# make sure that result directory exists before running any of the functions
result_dir = Path.cwd() / 'results'
if not result_dir.exists():
    result_dir.mkdir()


def tune_with_halving_grid_search(x_train, y_train, param_grid):
    svc = svm.SVC()

    start = time()
    halving_gs_results = HalvingGridSearchCV(
        svc,
        param_grid,
        cv=5,
        factor=3,
        min_resources='exhaust'
    ).fit(x_train, y_train)

    duration = time() - start

    results = DataFrame(halving_gs_results.cv_results_)
    results.loc[:, 'mean_test_score'] *= 100
    results.to_csv(result_dir / 'halving_svc_results.csv')

    # take the most relevant columns and sort (for readability). Remember to sort on the iter columns first, so we see
    # the models with the most training data behind them first.
    results = results.loc[:, ('iter', 'rank_test_score', 'mean_test_score', 'params')]
    results.sort_values(by=['iter', 'rank_test_score'], ascending=[False, True], inplace=True)

    return results, duration
```

Note the `sklearn.experimental import enable_halving_search_cv`. Because this is an experimental feature at the time of writing, you need this to make it work. This is basically the same code as the grid search function. I've added the most important function arguments for the `HalvingGridSearchCV` object. The `factor` argument is the reduction factor, set to 3 (default value). This means that we keep the best third

combinations at every iteration. For the first iteration on our dataset, it should do 48 combinations first, and only use 48/3=16 combinations in the second iteration. The `min_resources` argument dictates how many resources (data points) you start with for each combination in the first iteration. The default value, `'exhaust'`, automatically sets the initial value so that the last iteration uses the whole dataset, which is reasonable default behaviour. If your dataset is particularly big, or you're happy with not using all of it for crossvalidation in the end, you can set the `min_resources` and `max_resources` to suit your needs.

One last thing: looking at the results, be careful not to look for the highest `mean_test_score` to find the best hyper-parameters. This works for the normal exhaustive grid search object because everything is trained with the same data. With the successive halving grid search results, it's possible that one of the candidates got an amazing crossvalidation score on 100 datapoints, but just got lucky, and does poorly on subsequent iterations. It may not even have made it to the last iteration. When you look at these results, only look at the combinations that made it to the last iteration, since those are the best ones, and their crossvalidation score is based on the whole training set. The other entries simply don't give you the full picture.

Let's run both these algorithms side by side, and see what we get. The new `main.py` should look something like this:

```python
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

from hyperparam_tuning import tune_with_grid_search, tune_with_halving_grid_search

RANDOM_STATE = 35090


def main():
    # use fixed random state for repeatable data set
    X, Y = make_classification(n_samples=3000, random_state=RANDOM_STATE)
    x_train, x_test, y_train, y_test = train_test_split(X, Y, random_state=RANDOM_STATE)

    svc_params = {
        'C': [0.1, 0.5, 1, 2, 5, 10],
        'kernel': ['linear', 'rbf', 'sigmoid', 'poly'],
        'tol': [1e-3, 1e-2]
    }

    gs_results, gs_duration = tune_with_grid_search(x_train, y_train, svc_params)
    halving_results, halving_duration = tune_with_halving_grid_search(x_train, y_train, svc_params)

    print(gs_results.head())
    print(halving_results.head())

    score1 = gs_results['mean_test_score'].iloc[0]
    params1 = gs_results['params'].iloc[0]
    score2 = halving_results['mean_test_score'].iloc[0]
    params2 = halving_results['params'].iloc[0]

    svc1 = svm.SVC(**params1)
    svc1.fit(x_train, y_train)
    accuracy1 = accuracy_score(y_test, svc1.predict(x_test))

    svc2 = svm.SVC(**params2)
    svc2.fit(x_train, y_train)
    accuracy2 = accuracy_score(y_test, svc2.predict(x_test))

    print(f'Best score for GridSearchCv is {score1:.3f}, took {gs_duration:.2f} seconds')
    print(f'Params: {params1}')
    print(f'Corresponding test accuracy: {accuracy1 * 100:.2f}%\n')

    print(f'Best score for HalvingGridSearchCv is {score2:.3f}, took {halving_duration:.2f} seconds')
    print(f'Params: {params2}')
```

```
47      print(f'Corresponding test accuracy: {accuracy2 * 100:.2f}%')
48
```

```
~/Documents/DVT blogs/dvt-optimise main !1 ?2
❯ python main.py
   rank_test_score  mean_test_score                                                params
4                1        86.444444  {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.001}
2                2        86.400000      {'C': 0.1, 'kernel': 'rbf', 'tol': 0.001}
3                2        86.400000       {'C': 0.1, 'kernel': 'rbf', 'tol': 0.01}
5                4        86.355556   {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.01}
6                4        86.355556     {'C': 0.1, 'kernel': 'poly', 'tol': 0.001}
    iter  rank_test_score  mean_test_score                                                params
71     3                5        86.339286   {'C': 0.1, 'kernel': 'linear', 'tol': 0.01}
70     3                6        86.294643  {'C': 0.1, 'kernel': 'linear', 'tol': 0.001}
64     2                7        86.174497   {'C': 0.1, 'kernel': 'linear', 'tol': 0.01}
65     2                8        86.040268  {'C': 0.1, 'kernel': 'linear', 'tol': 0.001}
66     2               13        85.637584    {'C': 1, 'kernel': 'sigmoid', 'tol': 0.01}
```

You can see that the best crossvalidation score from the `HalvingGridSearchCV` had a crossvalidation score of 86.34%, very close to the 86.44% of exhaustive grid search. Of real interest is the second part of the data output:

```
Best score for GridSearchCv is 86.444, took 17.71 seconds
Params: {'C': 0.1, 'kernel': 'sigmoid', 'tol': 0.001}
Corresponding test accuracy: 85.60%

Best score for HalvingGridSearchCv is 86.339, took 1.40 seconds
Params: {'C': 0.1, 'kernel': 'linear', 'tol': 0.01}
Corresponding test accuracy: 85.47%
```

The final hyper-parameters are somewhat similar. The exhaustive grid search scored 85.6% accuracy, while the halving grid search got 85.47%. Not bad. Especially if you consider that the halving grid search only took 1.4 seconds. That's more than 12 times faster. For comparison's sake, when running the optimization with 10 000 samples, I get the following output:

```
Best score for GridSearchCv is 94.800, took 103.22 seconds
Params: {'C': 1, 'kernel': 'rbf', 'tol': 0.001}
Corresponding test accuracy: 95.00%

Best score for HalvingGridSearchCv is 94.796, took 10.23 seconds
Params: {'C': 1, 'kernel': 'rbf', 'tol': 0.01}
Corresponding test accuracy: 94.96%
```

Once again, similar crossvalidation scores, similar test set accuracy, similar optimal parameters. About 10 times faster.

Conclusion

We went over some practical examples of how to optimize hyper-parameters for scikit-learn ML models. If you were not familiar with the process before, you should be able to take it from here. It's fairly easy and intuitive to apply it to any of the other estimators in the scikit-learn library. Personally, I really like the successive halving grid search that scikit-learn introduced, and I hope it stays in the API without too many changes. I recommend using it to test out a broad range of hyper-parameters first, to get a feel for which ones tend to work better. You can then use the exhaustive grid search algorithm on a narrower set of parameters if you want to try and squeeze that last bit of accuracy out of your model.

Happy optimizing, my fellow machine learning enthusiasts!

Python Machine Learning Hyperparameter Tuning Scikit Learn