Bayesian Optimization: bayes_opt or hyperopt

BEGINNER DATA SCIENCE MACHINE LEARNING PYTHON

This article was published as a part of the Data Science Blogathon.

Introduction

If you have experience in Machine Learning, specifically supervised learning, you should have known that hyper parameter-tuning is an important process to improve model accuracy. This process tunes hyperparameters in a Machine Learning algorithm.

As we have known, every algorithm requires input parameters from observed data and hyperparameters are not from the observed data. Hyperparameters are different for each algorithm. If we do not tune the hyperparameters, the default hyperparameters are used.

There are many ways to do hyper parameter-tuning. This article will later focus on Bayesian Optimization as this is my favorite. There are 2 packages that I usually use for Bayesian Optimization. They are "bayes_opt" and "hyperopt" (Distributed Asynchronous Hyper-parameter Optimization). We will simply compare the two in terms of the time to run, accuracy, and output.

But before that, we will discuss some basic knowledge of hyperparameter-tuning.

Hyperparameter-tuning

Hyperparameter-tuning is the process of searching the most accurate hyperparameters for a dataset with a Machine Learning algorithm. To do this, we fit and evaluate the model by changing the hyperparameters one by one repeatedly until we find the best accuracy.

The search methods can be uninformed search and informed search. Uninformed search tries sets of hyperparameters repeatedly and independently. Each search does not inform or suggest the other searches. Examples of uninformed search are GridSearchCV and RandomizedSearchCV.

Hyper Parameter tuning Using GridSearchCV

Now, I want to perform hyperparameter-tuning on GradientBoostingClassifier. The dataset is from Kaggle competition. The hyperparameters to tune are "max_depth", "max_features", "learning_rate", "n_estimators", and "subsample".

Note that as mentioned above, these hyperparameters are only for GradientBoostingClassifier, not for the other algorithms. The accuracy metric is the accuracy score. I will run 5 fold cross-validation.

Below is the code for GridSearchCV. We can see that the value options for each hyperparameter are set in the "param_grid".

For example, the GridSearchCV will try to run with n_estimators of 80, 100, and so on until 150. To know how many times the GridSearchCV will run, just multiply the number of value options in each hyperparameter with one another. It will be 8 x 3 x 3 x 5 x 3 = 1080. And for each of the 1080 GridSearchCV, there will be 5 fold cross-validation. That makes 1080 x 5 = 5400 models should be built to find which is the best.

```
# Load packages from scipy.stats import uniform from sklearn.model_selection import cross_val_score from sklearn.ensemble import GradientBoostingClassifier from sklearn.model_selection import GridSearchCV # GridSearchCV param_grid = {'max_depth':[3,4,5,6,7,8,9,10], 'max_features':[0.8,0.9,1], 'learning_rate': [0.01,0.1,1], 'n_estimators':[80,100,120,140,150], 'subsample': [0.8,0.9,1]} grid = GridSearchCV(estimator=GradientBoostingClassifier(), param_grid=param_grid, scoring=acc_score, cv=5) grid.fit(X_train.iloc[1:100,], y_train.iloc[1:100,])
```

Disadvantages

The disadvantage of this method is that we can miss good hyperparameter values not set in the beginning. For instance, we do not set an option for the max_features to be 0.85 or the learning_rate to be 0.05. We do not know if that combination can give better accuracy. To overcome this, we can try RandomizedSearchCV.

Hyper Parameter-Tuning Using RandomizedSearchCV

Below is the code for that. Examine that the code sets a range of possible values for each hyperparameter. For example, the learning_rate can have any values from 0.01 to 1 distributed uniformly.

```
from sklearn.model_selection import RandomizedSearchCV # RandomizedSearhCV param_rand = {'max_depth':uniform(3,10), 'max_features':uniform(0.8,1), 'learning_rate':uniform(0.01,1), 'n_estimators':uniform(80,150), 'subsample':uniform(0.8,1)} rand = RandomizedSearchCV(estimator=GradientBoostingClassifier(), param_distributions=param_rand, scoring=acc_score, cv=5) rand.fit(X_train.iloc[1:100,], y_train.iloc[1:100,])
```

Problem With Uninformed Search

The problem with uninformed search is that it takes relatively a long time to build all the models. Informed search can solve this problem. In informed search, the previous models with a certain set of hyperparameter values can inform the later model which hyperparameter values better to select.

One of the methods to do this is coarse-to-fine. This involves running GridSearchCV or RandomizedSearchCV more than once. Each time, the hyperparameter value range is more specific.

For example, we start RandomizedSearchCV with learning_rate ranging from 0.01 to 1. Then, we find out that high accuracy models have their learning_rate around 0.1 to 0.3. Hence, we can run again GridSearchCV focusing on the learning_rate between 0.1 and 0.3. This process can continue until a satisfactory result is achieved. The first trial is coarse because the value range is large, from 0.01 to 1. The later trial is fine as the value range is focused on 0.1 to 0.3.

The drawback of the coarse-to-fine method is that we need to run the code repeatedly and observe the value range of hyperparameters-tuning. You might be thinking if there is a way to automate this. Yes, that is why my favorite is Bayesian Optimization.

Baysian Optimization

Bayesian Optimization also runs models many times with different sets of hyperparameter values, but it evaluates the past model information to select hyperparameter values to build the newer model. This is said to spend less time to reach the highest accuracy model than the previously discussed methods.

bayes_opt

As mentioned in the beginning, there are two packages in python that I usually use for Bayesian Optimization. The first one is bayes_opt. Here is the code to run it.

```
from bayes_opt import BayesianOptimization # Gradient Boosting Machine def gbm_cl_bo(max_depth, max_features,
learning_rate, n_estimators, subsample): params_gbm = {} params_gbm['max_depth'] = round(max_depth)
params_gbm['max_features'] = max_features params_gbm['learning_rate'] = learning_rate
params_gbm['n_estimators'] = round(n_estimators) params_gbm['subsample'] = subsample scores =
cross_val_score(GradientBoostingClassifier(random_state=123, **params_gbm), X_train, y_train,
scoring=acc_score, cv=5).mean() score = scores.mean() return score # Run Bayesian Optimization start =
time.time() params_gbm ={ 'max_depth':(3, 10), 'max_features':(0.8, 1), 'learning_rate':(0.01, 1),
'n_estimators':(80, 150), 'subsample': (0.8, 1) } gbm_bo = BayesianOptimization(gbm_cl_bo, params_gbm,
random_state=111) gbm_bo.maximize(init_points=20, n_iter=4) print('It takes %s minutes' % ((time.time() -
start)/60))
```

output:

```
| iter | target | learni... | max_depth | max_fe... | n_esti... | subsample | -------------------------------
------------------------------------------------------- | 1 | 0.7647 | 0.616 | 4.183 | 0.8872 | 133.8 | 0.8591
| | 2 | 0.7711 | 0.1577 | 3.157 | 0.884 | 96.71 | 0.8675 | | 3 | 0.7502 | 0.9908 | 4.664 | 0.8162 | 126.9 |
0.9242 | | 4 | 0.7681 | 0.2815 | 6.264 | 0.8237 | 85.18 | 0.9802 | | 5 | 0.7107 | 0.796 | 8.884 | 0.963 |
149.4 | 0.9155 | | 6 | 0.7442 | 0.8156 | 5.949 | 0.8055 | 111.8 | 0.8211 | | 7 | 0.7286 | 0.819 | 7.884 |
0.9131 | 99.2 | 0.9997 | | 8 | 0.7687 | 0.1467 | 7.308 | 0.897 | 108.4 | 0.9456 | | 9 | 0.7628 | 0.3296 |
5.804 | 0.8638 | 146.3 | 0.9837 | | 10 | 0.7668 | 0.8157 | 3.239 | 0.9887 | 146.5 | 0.9613 | | 11 | 0.7199 |
0.4865 | 9.767 | 0.8834 | 102.3 | 0.8033 | | 12 | 0.7708 | 0.0478 | 3.372 | 0.8256 | 82.34 | 0.8453 | | 13 |
0.7679 | 0.5485 | 4.25 | 0.8359 | 90.47 | 0.9366 | | 14 | 0.7409 | 0.4743 | 8.378 | 0.9338 | 110.9 | 0.919 |
| 15 | 0.7216 | 0.467 | 9.743 | 0.8296 | 143.5 | 0.8996 | | 16 | 0.7306 | 0.5966 | 7.793 | 0.8355 | 140.5 |
0.8964 | | 17 | 0.772 | 0.07865 | 5.553 | 0.8723 | 113.0 | 0.8359 | | 18 | 0.7589 | 0.1835 | 9.644 | 0.9311 |
89.45 | 0.9856 | | 19 | 0.7662 | 0.8434 | 3.369 | 0.8407 | 141.1 | 0.9348 | | 20 | 0.7566 | 0.3043 | 8.141 |
0.9237 | 94.73 | 0.9604 | | 21 | 0.7683 | 0.02841 | 9.546 | 0.9055 | 140.5 | 0.8805 | | 22 | 0.7717 | 0.05919
| 4.285 | 0.8093 | 92.7 | 0.9528 | | 23 | 0.7676 | 0.1946 | 7.351 | 0.9804 | 108.3 | 0.929 | | 24 | 0.7602 |
0.7131 | 5.307 | 0.8428 | 91.74 | 0.9193 |
======================================================================================= It takes
20.90080655813217 minutes
```

```
params_gbm = gbm_bo.max['params'] params_gbm['max_depth'] = round(params_gbm['max_depth'])
params_gbm['n_estimators'] = round(params_gbm['n_estimators']) params_gbm
```

Output:

```
{'learning_rate': 0.07864837617488214, 'max_depth': 6, 'max_features': 0.8723008386644597, 'n_estimators':
113, 'subsample': 0.8358969695415375}
```

The package bayes_opt takes 20 minutes to build 24 models. The best accuracy is 0.772.

hyperopt

Another package is hyperopt. Here is the code.

```
from hyperopt import hp, fmin, tpe # Run Bayesian Optimization from hyperopt start = time.time() space_lr = {'max_depth': hp.randint('max_depth', 3, 10), 'max_features': hp.uniform('max_features', 0.8, 1), 'learning_rate': hp.uniform('learning_rate',0.01, 1), 'n_estimators': hp.randint('n_estimators', 80,150), 'subsample': hp.uniform('subsample',0.8, 1)} def gbm_cl_bo2(params): params = {'max_depth': params['max_depth'], 'max_features': params['max_features'], 'learning_rate': params['learning_rate'], 'n_estimators': params['n_estimators'], 'subsample': params['subsample']} gbm_bo2 = GradientBoostingClassifier(random_state=111, **params) best_score = cross_val_score(gbm_bo2, X_train, y_train, scoring=acc_score, cv=5).mean() return 1 - best_score gbm_best_param = fmin(fn=gbm_cl_bo2, space=space_lr, max_evals=24, rstate=np.random.RandomState(42), algo=tpe.suggest) print('It takes %s minutes' % ((time.time() - start)/60))
```

Output:

```
100%|██████████| 24/24 [19:53<00:00, 49.74s/trial, best loss: 0.22769091027055077] It takes 19.897333371639252 minutes
```

```
gbm_best_param
```

Output:

```
{'learning_rate': 0.03516615427790515, 'max_depth': 6, 'max_features': 0.8920776081423815, 'n_estimators': 148, 'subsample': 0.9981549036976672}
```

The package hyperopt takes 19.9 minutes to run 24 models. The best loss is 0.228. It means that the best accuracy is 1 – 0.228 = 0.772. The duration to run bayes_opt and hyperopt is almost the same.

The accuracy is also almost the same although the results of the best hyperparameters are different. But, there is another difference. The package bayes_opt shows the process of tuning the values of the hyperparameters. We can see which values are used for each iteration. The package hyperopt only shows one line of the progress bar, best loss, and duration.

In my opinion, I prefer bayes_opt because, in reality, we may feel the tuning process takes too long time and just want to terminate the process. After stopping the processing, we just want to take the best hyperparameter-tuning result. We can do that with bayes_opt, but not with hyperopt.

There are still other ways of automatic hyperparameter-tuning. Not only the hyperparameter-tuning, but choosing the Machine Learning algorithms also can be automated. I will discuss that next time. The above code is available here.

About Author

Connect with me here.

The media shown in this article on Top Machine Learning Libraries in Julia are not owned by Analytics Vidhya and is used at the Author's discretion.

Article Url - https://www.analyticsvidhya.com/blog/2021/05/bayesian-optimization-bayes_opt-or-hyperopt/

Rendyk