ensemble_hyperparameters

May 29, 2021

1 Hyperparameter tuning

In the previous section, we did not discuss the parameters of random forest and gradient-boosting. However, there are a couple of things to keep in mind when setting these.

This notebook gives crucial information regarding how to set the hyperparameters of both random forest and gradient boosting decision tree models.

Caution!

For the sake of clarity, no cross-validation will be used to estimate the testing error. We are only showing the effect of the parameters on the validation set of what should be the inner cross-validation.

1.1 Random forest

The main parameter to tune for random forest is the `n_estimators` parameter. In general, the more trees in the forest, the better the statistical performance will be. However, it will slow down the fitting and prediction time. The goal is to balance computing time and statistical performance when setting the number of estimators when putting such learner in production.

The `max_depth` parameter could also be tuned. Sometimes, there is no need to have fully grown trees. However, be aware that with random forest, trees are generally deep since we are seeking to overfit the learners on the bootstrap samples because this will be mitigated by combining them. Assembling underfitted trees (i.e. shallow trees) might also lead to an underfitted forest.

[ ]:
```python
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

data, target = fetch_california_housing(return_X_y=True, as_frame=True)
target *= 100  # rescale the target in k$
data_train, data_test, target_train, target_test = train_test_split(
    data, target, random_state=0)
```

[ ]:
```python
import pandas as pd
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor

param_grid = {
    "n_estimators": [10, 20, 30],
    "max_depth": [3, 5, None],
}
grid_search = GridSearchCV(
    RandomForestRegressor(n_jobs=-1), param_grid=param_grid,
    scoring="neg_mean_absolute_error", n_jobs=-1,
)
grid_search.fit(data_train, target_train)

columns = [f"param_{name}" for name in param_grid.keys()]
columns += ["mean_test_score", "rank_test_score"]
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results["mean_test_score"] = -cv_results["mean_test_score"]
cv_results[columns].sort_values(by="rank_test_score")
```

We can observe that in our grid-search, the largest `max_depth` together with the largest `n_estimators` led to the best statistical performance.

1.2 Gradient-boosting decision trees

For gradient-boosting, parameters are coupled, so we cannot set the parameters one after the other anymore. The important parameters are `n_estimators`, `max_depth`, and `learning_rate`.

Let's first discuss the `max_depth` parameter. We saw in the section on gradient-boosting that the algorithm fits the error of the previous tree in the ensemble. Thus, fitting fully grown trees will be detrimental. Indeed, the first tree of the ensemble would perfectly fit (overfit) the data and thus no subsequent tree would be required, since there would be no residuals. Therefore, the tree used in gradient-boosting should have a low depth, typically between 3 to 8 levels. Having very weak learners at each step will help reducing overfitting.

With this consideration in mind, the deeper the trees, the faster the residuals will be corrected and less learners are required. Therefore, `n_estimators` should be increased if `max_depth` is lower.

Finally, we have overlooked the impact of the `learning_rate` parameter until now. When fitting the residuals, we would like the tree to try to correct all possible errors or only a fraction of them. The learning-rate allows you to control this behaviour. A small learning-rate value would only correct the residuals of very few samples. If a large learning-rate is set (e.g., 1), we would fit the residuals of all samples. So, with a very low learning-rate, we will need more estimators to correct the overall error. However, a too large learning-rate tends to obtain an overfitted ensemble, similar to having a too large tree depth.

[ ]:
```
from sklearn.ensemble import GradientBoostingRegressor

param_grid = {
    "n_estimators": [10, 30, 50],
    "max_depth": [3, 5, None],
    "learning_rate": [0.1, 1],
}
grid_search = GridSearchCV(
    GradientBoostingRegressor(), param_grid=param_grid,
    scoring="neg_mean_absolute_error", n_jobs=-1
)
grid_search.fit(data_train, target_train)

columns = [f"param_{name}" for name in param_grid.keys()]
columns += ["mean_test_score", "rank_test_score"]
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results["mean_test_score"] = -cv_results["mean_test_score"]
cv_results[columns].sort_values(by="rank_test_score")
```

Caution!

Here, we tune the n_estimators but be aware that using early-stopping as in the previous exercise will be better.