ensemble_hyperparameters

February 29, 2024

1 Hyperparameter tuning

In the previous section, we did not discuss the hyperparameters of random forest and histogram gradient-boosting. This notebook gives crucial information regarding how to set them.

Caution!

For the sake of clarity, no nested cross-validation is used to estimate the variability of the testing error. We are only showing the effect of the parameters on the validation set.

We start by loading the california housing dataset.

[1]:
```
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
)
```

1.1 Random forest

The main parameter to select in random forest is the `n_estimators` parameter. In general, the more trees in the forest, the better the generalization performance would be. However, adding trees slows down the fitting and prediction time. The goal is to balance computing time and generalization performance when setting the number of estimators. Here, we fix `n_estimators=100`, which is already the default value.

Caution!

Tuning the n_estimators for random forests generally result in a waste of computer power. We just need to ensure that it is large enough so that doubling its value does not lead to a significant improvement of the validation error.

Instead, we can tune the hyperparameter `max_features`, which controls the size of the random subset of features to consider when looking for the best split when growing the trees: smaller values for `max_features` lead to more random trees with hopefully more uncorrelated prediction errors. However if `max_features` is too small, predictions can be too random, even after averaging with the trees in the ensemble.

If `max_features` is set to `None`, then this is equivalent to setting `max_features=n_features` which means that the only source of randomness in the random forest is the bagging procedure.

```
[2]: print(f"In this case, n_features={len(data.columns)}")
```

```
In this case, n_features=8
```

We can also tune the different parameters that control the depth of each tree in the forest. Two parameters are important for this: `max_depth` and `max_leaf_nodes`. They differ in the way they control the tree structure. Indeed, `max_depth` enforces growing symmetric trees, while `max_leaf_nodes` does not impose such constraint. If `max_leaf_nodes=None` then the number of leaf nodes is unlimited.

The hyperparameter `min_samples_leaf` controls the minimum number of samples required to be at a leaf node. This means that a split point (at any depth) is only done if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. A small value for `min_samples_leaf` means that some samples can become isolated when a tree is deep, promoting overfitting. A large value would prevent deep trees, which can lead to underfitting.

Be aware that with random forest, trees are expected to be deep since we are seeking to overfit each tree on each bootstrap sample. Overfitting is mitigated when combining the trees altogether, whereas assembling underfitted trees (i.e. shallow trees) might also lead to an underfitted forest.

```
[3]: import pandas as pd
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor

param_distributions = {
    "max_features": [1, 2, 3, 5, None],
    "max_leaf_nodes": [10, 100, 1000, None],
    "min_samples_leaf": [1, 2, 5, 10, 20, 50, 100],
}
search_cv = RandomizedSearchCV(
    RandomForestRegressor(n_jobs=2),
    param_distributions=param_distributions,
    scoring="neg_mean_absolute_error",
    n_iter=10,
    random_state=0,
    n_jobs=2,
)
search_cv.fit(data_train, target_train)

columns = [f"param_{name}" for name in param_distributions.keys()]
columns += ["mean_test_error", "std_test_error"]
cv_results = pd.DataFrame(search_cv.cv_results_)
cv_results["mean_test_error"] = -cv_results["mean_test_score"]
cv_results["std_test_error"] = cv_results["std_test_score"]
cv_results[columns].sort_values(by="mean_test_error")
```

```
[3]:   param_max_features param_max_leaf_nodes param_min_samples_leaf  \
3                      2                 None                      2
0                      2                 1000                     10
7                   None                 None                     20
4                      5                  100                      2
8                   None                  100                     10
6                   None                 1000                     50
9                      1                  100                      2
2                      1                  100                      1
5                      1                 None                    100
1                      3                   10                     10

   mean_test_error  std_test_error
3        33.994792        0.246031
0        36.833724        0.573227
7        37.321119        0.399929
4        40.175336        0.575806
8        40.367495        0.545907
6        40.865090        0.476823
9        49.851861        0.852632
2        50.013578        0.577053
5        54.847401        0.830265
1        54.950441        0.718516
```

We can observe in our search that we are required to have a large number of `max_leaf_nodes` and thus deep trees. This parameter seems particularly impactful with respect to the other tuning parameters, but large values of `min_samples_leaf` seem to reduce the performance of the model.

In practice, more iterations of random search would be necessary to precisely assert the role of each parameters. Using `n_iter=10` is good enough to quickly inspect the hyperparameter combinations that yield models that work well enough without spending too much computational resources. Feel free to try more interations on your own.

Once the `RandomizedSearchCV` has found the best set of hyperparameters, it uses them to refit the model using the full training set. To estimate the generalization performance of the best model it suffices to call `.score` on the unseen data.

```
[4]: error = -search_cv.score(data_test, target_test)
     print(
         f"On average, our random forest regressor makes an error of {error:.2f} k$"
     )
```

```
On average, our random forest regressor makes an error of 33.77 k$
```

1.2 Histogram gradient-boosting decision trees

For gradient-boosting, hyperparameters are coupled, so we cannot set them one after the other anymore. The important hyperparameters are `max_iter`, `learning_rate`, and `max_depth` or `max_leaf_nodes` (as previously discussed random forest).

Let's first discuss `max_iter` which, similarly to the `n_estimators` hyperparameter in random forests, controls the number of trees in the estimator. The difference is that the actual number of trees trained by the model is not entirely set by the user, but depends also on the stopping criteria: the number of trees can be lower than `max_iter` if adding a new tree does not improve the model enough. We will give more details on this in the next exercise.

The depth of the trees is controlled by `max_depth` (or `max_leaf_nodes`). We saw in the section on gradient-boosting that boosting algorithms fit the error of the previous tree in the ensemble. Thus, fitting fully grown trees would 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, or few leaves ($2^3 = 8$ to $2^8 = 256$). Having very weak learners at each step helps reducing overfitting.

With this consideration in mind, the deeper the trees, the faster the residuals are corrected and then less learners are required. Therefore, it can be beneficial to increase `max_iter` if `max_depth` is low.

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 would need more estimators to correct the overall error. However, a too large learning-rate tends to obtain an overfitted ensemble, similar to having very deep trees.

[5]:
```
from scipy.stats import loguniform
from sklearn.ensemble import HistGradientBoostingRegressor

param_distributions = {
    "max_iter": [3, 10, 30, 100, 300, 1000],
    "max_leaf_nodes": [2, 5, 10, 20, 50, 100],
    "learning_rate": loguniform(0.01, 1),
}
search_cv = RandomizedSearchCV(
    HistGradientBoostingRegressor(),
    param_distributions=param_distributions,
    scoring="neg_mean_absolute_error",
    n_iter=20,
    random_state=0,
    n_jobs=2,
)
search_cv.fit(data_train, target_train)

columns = [f"param_{name}" for name in param_distributions.keys()]
columns += ["mean_test_error", "std_test_error"]
cv_results = pd.DataFrame(search_cv.cv_results_)
cv_results["mean_test_error"] = -cv_results["mean_test_score"]
cv_results["std_test_error"] = cv_results["std_test_score"]
```

```python
cv_results[columns].sort_values(by="mean_test_error")
```

```
[5]:    param_max_iter param_max_leaf_nodes param_learning_rate  mean_test_error  \
    14             300                  100             0.01864        31.035077
    6              300                   20            0.047293        31.840754
    13             300                   10            0.297739        32.624231
    2               30                   50            0.176656        32.647931
    9              100                   20            0.083745        33.090565
    19             100                   10            0.215543        33.169722
    12             100                   20            0.067503        33.553517
    16             300                    5             0.05929        35.920223
    1              100                    5            0.160519        36.213915
    0             1000                    2            0.125207        40.837972
    7             1000                    2            0.054511        42.188889
    8                3                    5            0.906226        49.827555
    18              10                    5            0.248463        50.335463
    5               10                  100            0.061034        61.595766
    17               3                    5            0.079415        81.420508
    4               10                    2              0.0351        82.546361
    15               3                   50            0.019923        87.655659
    3                3                    2            0.039361        87.790307
    11               3                   10            0.019351        88.391757
    10               3                    5             0.01724        88.874222

        std_test_error
    14        0.468271
    6         0.215393
    13        1.139313
    2         0.554608
    9         0.436939
    19        0.344800
    12        0.586867
    16        0.503477
    1         0.606962
    0         0.389736
    7         0.652427
    8         0.726456
    18        0.738581
    5         0.666544
    17        0.956599
    4         1.010775
    15        1.123802
    3         1.086382
    11        1.073165
    10        1.063774
```

Caution!

Here, we tune max_iter but be aware that it is better to set max_iter to a fixed, large enough value and use parameters linked to early_stopping as we will do in Exercise M6.04.

In this search, we observe that for the best ranked models, having a smaller `learning_rate`, requires more trees or a larger number of leaves for each tree. However, it is particularly difficult to draw more detailed conclusions since the best value of each hyperparameter depends on the other hyperparameter values.

We can now estimate the generalization performance of the best model using the test set.

[6]:
```python
error = -search_cv.score(data_test, target_test)
print(f"On average, our HGBT regressor makes an error of {error:.2f} k$")
```

```
On average, our HGBT regressor makes an error of 30.50 k$
```

The mean test score in the held-out test set is slightly better than the score of the best model. The reason is that the final model is refitted on the whole training set and therefore, on more data than the cross-validated models of the grid search procedure.

We summarize these details in the following table:

Bagging & Random Forests	**Boosting**
fit trees **independently**	fit trees **sequentially**
each **deep tree overfits**	each **shallow tree underfits**
averaging the tree predictions **reduces overfitting**	sequentially adding trees **reduces underfitting**
generalization improves with the number of trees	too many trees may cause overfitting
does not have a `learning_rate` parameter	fitting the residuals is controlled by the `learning_rate`

[ ]:
```python
```

[ ]:
```python
```