ensemble_random_forest

May 29, 2021

1 Random forests

In this notebook, we will present the random forest models and show the differences with the bagging ensembles.

Random forests are a popular model in machine learning. They are a modification of the bagging algorithm. In bagging, any classifier or regressor can be used. In random forests, the base classifier or regressor is always a decision tree.

Random forests have another particularity: when training a tree, the search for the best split is done only on a subset of the original features taken at random. The random subsets are different for each split node. The goal is to inject additional randomization into the learning procedure to try to decorrelate the prediction errors of the individual trees.

Therefore, random forests are using **randomization on both axes of the data matrix**:

- by **bootstrapping samples** for **each tree** in the forest;
- randomly selecting a **subset of features** at **each node** of the tree.

1.1 A look at random forests

We will illustrate the usage of a random forest classifier on the adult census dataset.

[1]:
```
import pandas as pd

adult_census = pd.read_csv("../datasets/adult-census.csv")
target_name = "class"
data = adult_census.drop(columns=[target_name, "education-num"])
target = adult_census[target_name]
```

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

%%[markdown]

The adult census contains some categorical data and we encode the categorical features using an `OrdinalEncoder` since tree-based models can work very efficiently with such a naive representation of categorical variables.

Since there are rare categories in this dataset we need to specifically encode unknown categories at prediction time in order to be able to use cross-validation. Otherwise some rare categories could

only be present on the validation side of the cross-validation split and the `OrdinalEncoder` would raise an error when calling the its `transform` method with the data points of the validation set.

[2]:
```
from sklearn.preprocessing import OrdinalEncoder
from sklearn.compose import make_column_transformer, make_column_selector

categorical_encoder = OrdinalEncoder(
    handle_unknown="use_encoded_value", unknown_value=-1
)
preprocessor = make_column_transformer(
    (categorical_encoder, make_column_selector(dtype_include=object)),
    remainder="passthrough"
)
```

We will first give a simple example where we will train a single decision tree classifier and check its statistical performance via cross-validation.

[3]:
```
from sklearn.pipeline import make_pipeline
from sklearn.tree import DecisionTreeClassifier

tree = make_pipeline(preprocessor, DecisionTreeClassifier(random_state=0))
```

[4]:
```
from sklearn.model_selection import cross_val_score

scores_tree = cross_val_score(tree, data, target)

print(f"Decision tree classifier: "
      f"{scores_tree.mean():.3f} +/- {scores_tree.std():.3f}")
```

```
Decision tree classifier: 0.820 +/- 0.006
```

Similarly to what was done in the previous notebook, we construct a `BaggingClassifier` with a decision tree classifier as base model. In addition, we need to specify how many models do we want to combine. Note that we also need to preprocess the data and thus use a scikit-learn pipeline.

[5]:
```
from sklearn.ensemble import BaggingClassifier

bagged_trees = make_pipeline(
    preprocessor,
    BaggingClassifier(
        base_estimator=DecisionTreeClassifier(random_state=0),
        n_estimators=50, n_jobs=2, random_state=0,
    )
)
```

[6]:
```
scores_bagged_trees = cross_val_score(bagged_trees, data, target)

print(f"Bagged decision tree classifier: "
      f"{scores_bagged_trees.mean():.3f} +/- {scores_bagged_trees.std():.3f}")
```

```
Bagged decision tree classifier: 0.846 +/- 0.005
```

Note that the generalization performance of the bagged trees is already much better than the performance of a single tree.

Now, we will use a random forest. You will observe that we do not need to specify any `base_estimator` because the estimator is forced to be a decision tree. Thus, we just specify the desired number of trees in the forest.

[7]:
```
from sklearn.ensemble import RandomForestClassifier

random_forest = make_pipeline(
    preprocessor,
    RandomForestClassifier(n_estimators=50, n_jobs=2, random_state=0)
)
```

[8]:
```
scores_random_forest = cross_val_score(random_forest, data, target)

print(f"Random forest classifier: "
      f"{scores_random_forest.mean():.3f} +/- "
      f"{scores_random_forest.std():.3f}")
```

```
Random forest classifier: 0.851 +/- 0.004
```

It seems that the random forest is performing slightly better than the bagged trees possibly due to the randomized selection of the features which decorrelates the prediction errors of individual trees and as a consequence make the averaging step more efficient at reducing overfitting.

1.2 Details about default hyperparameters

For random forests, it is possible to control the amount of randomness for each split by setting the value of `max_features` hyperparameter:

- `max_feature=0.5` means that 50% of the features are considered at each split;
- `max_features=1.0` means that all features are considered at each split which effectively disables feature subsampling.

By default, `RandomForestRegressor` disables feature subsampling while `RandomForestClassifier` uses `max_features=np.sqrt(n_features)`. These default values reflect good practices given in the scientific literature.

However, `max_features` is one of the hyperparameters to consider when tuning a random forest: - too much randomness in the trees can lead to underfitted base models and can be detrimental for the ensemble as a whole, - too few randomness in the trees leads to more correlation of the prediction errors and as a result reduce the benefits of the averaging step in terms of overfitting control.

[ ]:
```
```