Theoretical foundations of Machine Learning

TP 4

Decision Tree & Random Forest

To visualize the decision trees that will be built in this lab session, you need to install the `graphviz`, `python-graphviz`, and `pydotplus` libraries.[1] This can be done by launching Anaconda Navigator and navigating to the Environments page.
We are working with a real estate dataset. The goal is to predict property prices based on environmental features. This is therefore a regression problem. For more details, one can refer to the dataset documentation.

```
import numpy as np
from sklearn.datasets import fetch_california_housing
data = fetch_california_housing()
X = data.data
y = data.target

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=17)
```

The argument `random_state=17` sets the seed used by the random number generator within the `train_test_split` function. This ensures that the shuffling of the data is reproducible across runs. It does not affect all randomness globally, but only the randomness internal to that function. Using a fixed `random_state` makes it easier to compare different models by ensuring that they are trained and tested on the same data splits.
A decision tree can be trained for this regression problem as follows:

```
from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor(random_state=0).fit(X_train, y_train)
```

By default, the impurity criterion used for regression is the Mean Squared Error (MSE). For each node, the split is selected randomly among those that minimize the criterion. However, for each decision tree trained in this lab, we will enforce deterministic behavior using the `random_state=0` argument.
You can access the depth and the number of nodes of the tree `dt` using `dt.tree_.max_depth` and `dt.tree_.node_count`, respectively.

Question 1. Define a function `tree_summary` that takes a trained decision tree as an argument and displays its training score, test score, depth, and number of nodes. Apply this function to the tree `dt` constructed above. Comment on the results.

You can limit the depth of the tree using the `max_depth` argument.

Question 2. Train a decision tree, named `dt2`, with its depth limited to 10. Observe its scores and characteristics. Comment on the results.
You can visualize a decision tree `dt3` with maximal depth 3 as follows:

```
from sklearn.externals.six import StringIO
from IPython.display import Image
```

[1] You can also try installing `graphviz` by typing `pip install graphviz` and `conda install graphviz` in a terminal (Mac or Linux).

```
from sklearn.tree import export_graphviz
from pydotplus import graph_from_dot_data

foo = StringIO()
export_graphviz(dt3, out_file=foo, impurity=False)
graph = graph_from_dot_data(foo.getvalue())
Image(graph.create_png())
```

Question 3. Which explanatory variables are involved in the decision tree `dt3`?

Question 4. Using 10-fold cross-validation, choose the best value for the maximum depth of the decision tree.

The following questions aim to guide the construction of a predictor obtained by aggregating several decision trees trained on different training samples. More precisely, let $S = \{(X_i, Y_i)\}_{i\in[m]}$ be the original training sample, and let n be the number of decision trees we want to aggregate.
For each $k \in [n]$, a sample of size m is drawn:

$$S^{(k)} = \{(X_i^{(k)}, Y_i^{(k)})\}_{i\in[m]}$$

where each example $(X_i^{(k)}, Y_i^{(k)})$ is sampled uniformly and independently from the original dataset S (this is thus a sampling with replacement).
A decision tree $\hat{f}^{(k)}$ is then trained on each sample $S^{(k)}$, and the final predictor $\hat{f}$ is obtained by averaging:

$$\forall x \in \mathcal{X},\ \hat{f}(x) = \frac{1}{n}\sum_{k=1}^{n} \hat{f}^{(k)}(x)$$

Question 5. Define a function `r2_score` that takes two arrays `y_true` and `y_predict` as arguments, containing respectively the true labels and the predicted labels for a given sample. The function should return the corresponding R^2 score.

```
from sklearn.utils import resample
```

Question 6. Using the function `resample`, draw (with replacement) a new sample `(X_train_, y_train_)` of the same size from the training sample `(X_train, y_train)`. Since sampling is with replacement, some examples may appear multiple times in the new sample, while others may not appear at all.
Train a decision tree on this new sample (with `random_state=0` and default values for the other parameters), and display its R^2 score using the `r2_score` function. Verify that it gives the same result as the predictor's built-in `.score` method.

Question 7. Build 5 predictors in the same way as in the previous question, resampling `(X_train, y_train)` each time. Compute the predictions of each predictor on the test set.

Question 8. Consider the predictor defined as the average of the 5 predictors built in the previous question. Compute its predictions on the test set and calculate the corresponding R^2 score.

Question 9. Define a function `tree_aggregation` that takes an integer argument `n_trees`, trains `n_trees` decision trees as previously described, and displays the R^2 score of the aggregated predictor on the test set. Run this function with different values for `n_trees` and try to achieve a better score.

The procedure that involves building a new training sample using uniform sampling with replacement is called *bootstrap*. The process of aggregating several predictors trained on bootstrap samples is called *bagging* (short for bootstrap aggregating). In the special case of decision trees, bagging is also known as a *random forest*.
Scikit-learn provides a built-in implementation of the random forest algorithm, which can be used as follows:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=5).fit(X_train, y_train)
```

Question 10. Compare the results obtained using the `tree_aggregation` function with those from the built-in Scikit-learn algorithm.