Chapter 7: Ensemble Learning and Random Forests

Warren Alphonso

March 2019

A group of predictors is called an *ensemble*. Thus, aggregating multiple predictors is called *Ensemble Learning*. We can train a group of Decision Tree classifiers on different random subsets of a training set. To make predictions, we just obtain the prediction of all individual trees and then predict the class that gets the most votes. This ensemble of Decision Trees is called a *Random Forest*.

1 Voting Classifiers

Even if each classifier is a *weak learner* (it does only slightly better than random guessing), the ensemble created can still be a *strong learner* (achieving high accuracy), providing there are sufficient weak learners and they are sufficiently diverse. This is due to the law of large numbers. This only occurs if all classifiers are perfectly independent and make uncorrelated errors, which is not the case since they are trained on the same data.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="liblinear", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma="auto", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

voting_clf.fit(X_train, Y_train)

from sklearn.metrics import accuracy_score
```

```
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

If all classifiers are able to estimate class probabilities (ie they have a `predict_proba()` method), then we can predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. We just need to replace `voting="hard"` with `voting="soft"`. If we do this, the above ensemble achieves over 91% accuracy.

2 Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, like we did in the previous section. Another way is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When sampling is performed **with** replacement, this method is called *bagging*. When performed **without** replacement, it is called *pasting*. It is important to note that predictors can all be trained in parallel via different CPU cores so bagging and pasting are popular since they scale very well.

3 Random Forests

The Random Forest algorithm introduces extra randomness when growing trees: instead of searching or the very best feature when splitting a node, it searches for the best feature among a random subset of features. This results in greater tree diversity which trades a higher bias for a lower variance.

3.1 Extra-Trees

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting. It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds. A forest of such extremely random trees is called an *Extremely Randomized Trees* ensemble. It is hard to tell in advance whether a `RandomForestClassifier` will perform better than an `ExtraTreesClassifier`. The only way to know is to try both and compare them using cross-validation.

3.2 Feature Importance

A great quality of Random Forest is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average. We can access this result using the `feature_importances_ variable`.

```
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)

sepal length (cm) 0.11249225099876374
sepal width (cm) 0.023119288282510326
petal length (cm) 0.44103046436395765
petal width (cm) 0.4233579963547681
```

4 Boosting

Boosting refers to any Ensemble method that can combine several weak learners into a strong learner. The idea is to train predictors sequentially, each trying to correct its predecessor.

4.1 AdaBoost

AdaBoost (short for Adaptive Boosting) makes each new predictor pay more attention to the training instances that its predecessor underfitted.
To build an AdaBoost classifier, a first base classifier is trained and used to make predictions on the training set. The relative weight of misclassified training instances is then increased. A second classifier is trained using the updated weights and again it makes predictions on the training set, weights are updated, and so on. Each instance weight $w^{(i)}$ is initially set to $\frac{1}{m}$. A first predictor is trained and its weighted error rate r_1 is computed on the training set. The predictor's weight a_j is then computed. The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be near zero. If it is worse than random, its weight will be negative. Next, the instance weights are updated. Then, all the instance weights are normalized. Finally, a new predictor is trained using the updated weights and the whole process is repeated.
To make predictions, AdaBoost simply computes the predictions of all the predictors and weights them using the predictor weights a_j.
Scikit-Learn actually uses a multiclass version of AdaBoost called *SAMME* (which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function).

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```

4.2 Gradient Boosting

Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. Instead of tweaking the instance weights, this method tries to fit the new predictors to the *residual errors* made by the previous predictor. This is called *Gradient Tree Boosting* or *Gradient Boosted Regression Trees*.
First, let's fit a `DecisionTreeRegressor` to the training set. Now, train a second `DecisionTreeRegressor` on the residual errors made by the first predictor. Then, train a third regressor on the residual errors made by the second predictor. Now, we have an ensemble containing three trees. It can make predictions on new instances simply by adding up the predictions of all the trees.
A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class. It has hyperparameters to control the growth of the Decision Trees (`max_depth, min_samples_leaf,` and so on) as well as hyperparameters to control the ensemble training, such as number of trees (`n_estimators`).

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0, random_s
gbrt.fit(X, y)
```

The `learning_rate` hyperparameter scales the contribution of each tree. Setting it to a low value such as `0.1` means we will need more trees in the ensemble to fit the training data but the predictions will usually generalize better.
In order to find the optimal number of trees, we can use early stopping. To implement this, use `staged_predict()`. It returns an iterator over the predictions made by the ensemble at each stage of training.

5 Stacking

The final Ensemble method we will discuss is *stacking* (short for stacked generalization). Instead of trivial functions (such as hard voting) to aggregate the predictions of all predictors, we can instead train a model to perform this aggregation. The final predictor is called a *blender* or *meta learner* and takes these predictions as inputs to make the final prediction.
To train the blender, the training set is split in two subsets. The first subset is used to train the predictors in the first layer. Then, the first layer predictors are used to make predictions on the second set since the predictors never saw

these instances during training. For each instance, there are several predicted values (based on how many predictors we have). We now create a new training set using these predicted values as input features and keeping the target values. The blender is trained on this new training set.
It is actually possible to train several different blenders this way (one using Linear Regression, another using Random Forest Regression, etc) and then we get another layer of blenders. Now, the trick is to split the training set into **three** subsets: the first one is used to train the first layer, the second one is for the second layer, and the third is for the final blender.