Ensembles of Decision Trees

Zhongjian Lin
University of Georgia

September 25, 2024

Ensemble Methods

An ensemble method is an approach that combines many simple "building block" models in order to obtain a single and potentially very powerful model. These simple building block models are sometimes known as weak learners, since they may lead to mediocre predictions on their own.

- Bagging
- Random Forests
- Boosting

Bagging

The Decision Trees suffer from high variance. Bootstrap aggregation, or *bagging*, is a general-purpose procedure for reducing the variance of a statistical learning method. Simple averaging would reduce the variance to the ratio of $1/n$. Thus a natural way to reduce the variance and increase the test set accuracy of a statistical learning method is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions. In other words, we could calculate $\hat{f}_1(x), \hat{f}_2(x), \cdots, \hat{f}_B(x)$ using B separate training sets, and average them in order to obtain a single low-variance statistical learning model, given by

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x).$$

Of course, this is not practical because we generally do not have access to multiple training sets. Instead, we can bootstrap, by taking repeated samples from the (single) training data set. In this approach we generate B different bootstrapped training data sets.

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^*_b(x).$$

Bagging

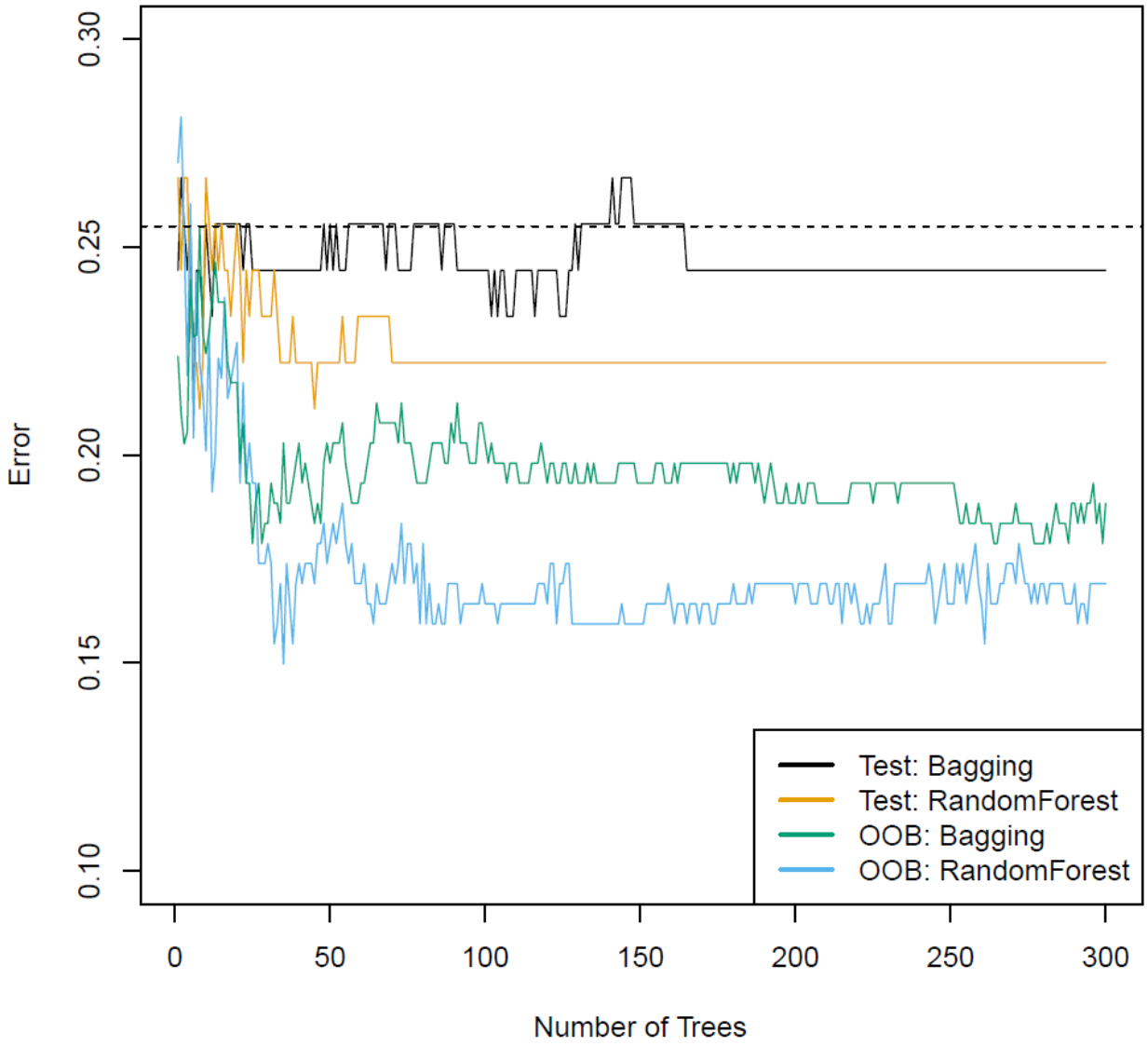
- To apply bagging to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions.
- These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. Bagging has been demonstrated to give impressive improvements in accuracy by combining together hundreds or even thousands of trees into a single procedure.
- For classification, we have a similar procedure. For a given test observation, we can record the class predicted by each of the B trees, and take a majority vote: the overall prediction is the most commonly occurring majority class among the B predictions.

Out-of-Bag Error Estimation

It turns out that there is a very straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation or the validation set approach.

- One can show that on average, each bagged tree makes use of around two-thirds of the observations. The remaining one-third of the observations not used to fit a given bagged tree are referred to as the *out-of-bag* (OOB) observations. We can predict the response for the out-of-bag ith observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the ith observation. In order to obtain a single prediction for the ith observation, we can average these predicted responses (if regression is the goal) or can take a majority vote (if classification is the goal).
- This leads to a single OOB prediction for the ith observation. An OOB prediction can be obtained in this way for each of the n observations, from which the overall OOB MSE (for a regression problem) or classification error (for a classification problem) can be computed. The resulting OOB error is a valid estimate of the test error for the bagged model, since the response for each observation is predicted using only the trees that were not fit using that observation.

Bagging


Error
0.10
0.15
0.20
0.25
0.30
0
50
100
150
200
250
300
Number of Trees
Test: Bagging
Test: RandomForest
OOB: Bagging
OOB: RandomForest

Random Forests

Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees.

- As in bagging, we build a number of decision trees on bootstrapped training samples.
- But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors.
- The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$- that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors.
- Doing this allows us to build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

Random Forests

Random forests is not even allowed to consider a majority of the available predictors. This may sound crazy, but it has a clever rationale.

- Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split. Consequently, all of the bagged trees will look quite similar to each other. This causes problem of "not diversifying well".
- Random forests inject randomness to avoid such strong deterministic trend and to ensure each tree is different: by selecting the data points used to build a tree and by selecting the features in each split test.
- The idea of random forests is that each tree might do a relatively good job of predicting and should also be different from the other trees. Then they will likely overfit on part of the data.

Randomness from Bootstrap and Features Choice

To build a random forest model, you need to decide on the number of trees to build ("n_estimator").

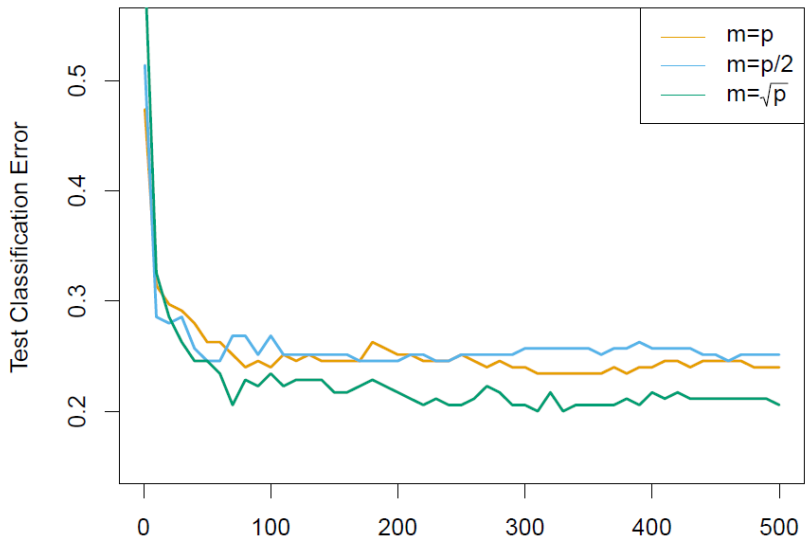
- The first way to inject randomness is by "Bootstrap": From our n_samples data points, we repeatedly draw an example randomly with replacement (i.e. the same sample can be picked multiple times), n_samples times. This will create a dataset that is as big as the original dataset, but some data points will be missing from it, and some will be repeated.
- In each node the algorithm randomly selects a subset of the features, and looks for the best possible test involving one of these features. The amount of features that is selected is controlled by the max_features parameter.

The bootstrap sampling leads to each decision tree in the random forest being built on a slightly different dataset. Because of the selection of features in each node, each split in each tree operates on a different subset of features. Together these two mechanisms ensure that all the trees in the random forests are different.

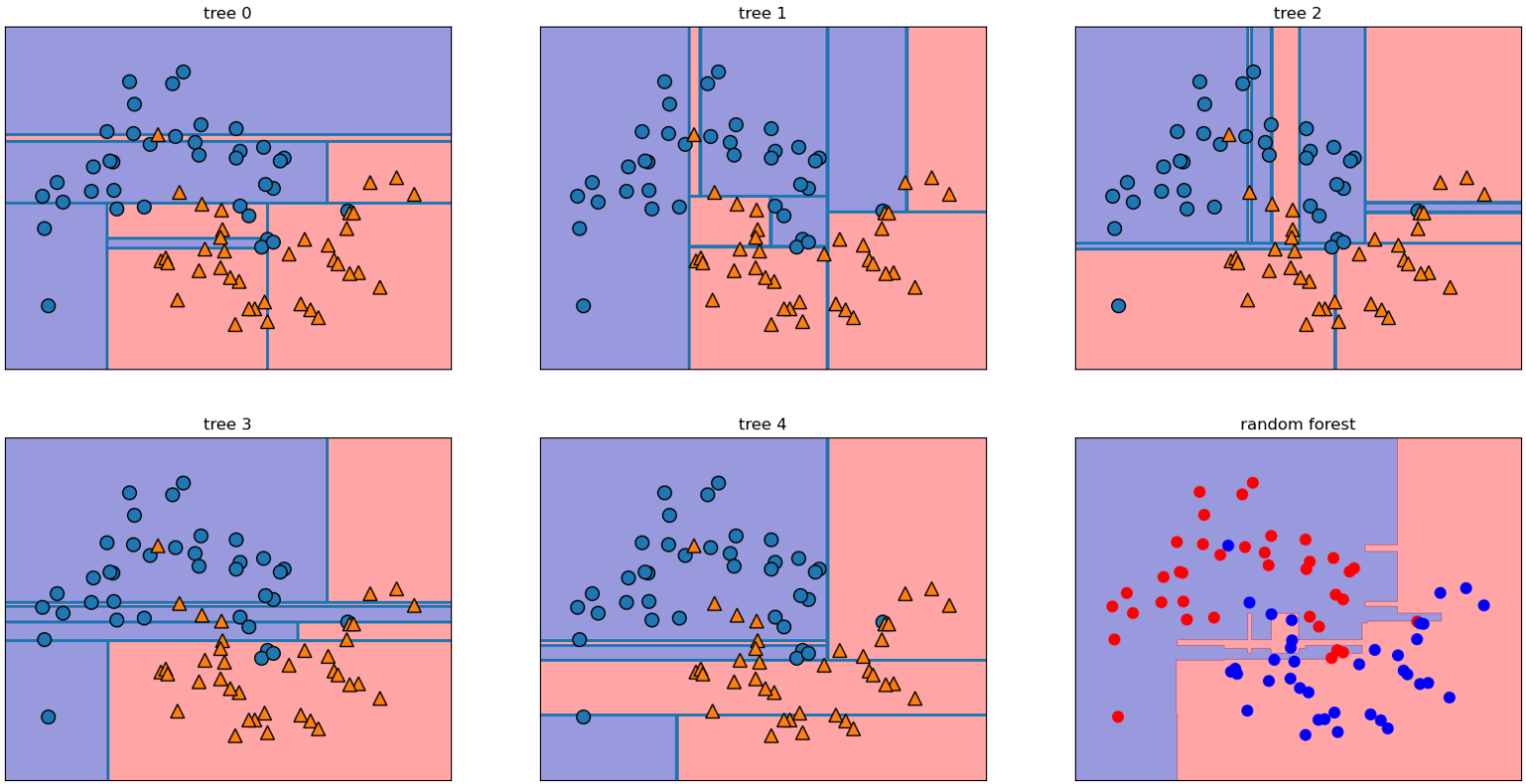
Model Complexity

A critical parameter in this process is max_features (m).

- If we set max_features to n_features, that means that each split can look at all features in the dataset, and no randomness will be injected (*bagging*). If we set max_features to one, that means that the splits have no choice at all on which feature to test, and can only search over different thresholds for the feature that was selected randomly.
- Therefore, a high max_features means that the trees in the random forest will be quite similar, and they will be able to fit the data easily, using the most distinctive features. A low max_features means that the trees in the random forest will be quite different, and that each tree might need to be very deep in order to fit the data well.



Random Forest



The random forest overfit less than any of the trees individually, and provides a much more intuitive decision boundary. In any real application, we would use many more trees (often hundreds or thousands), leading to even smoother boundaries.

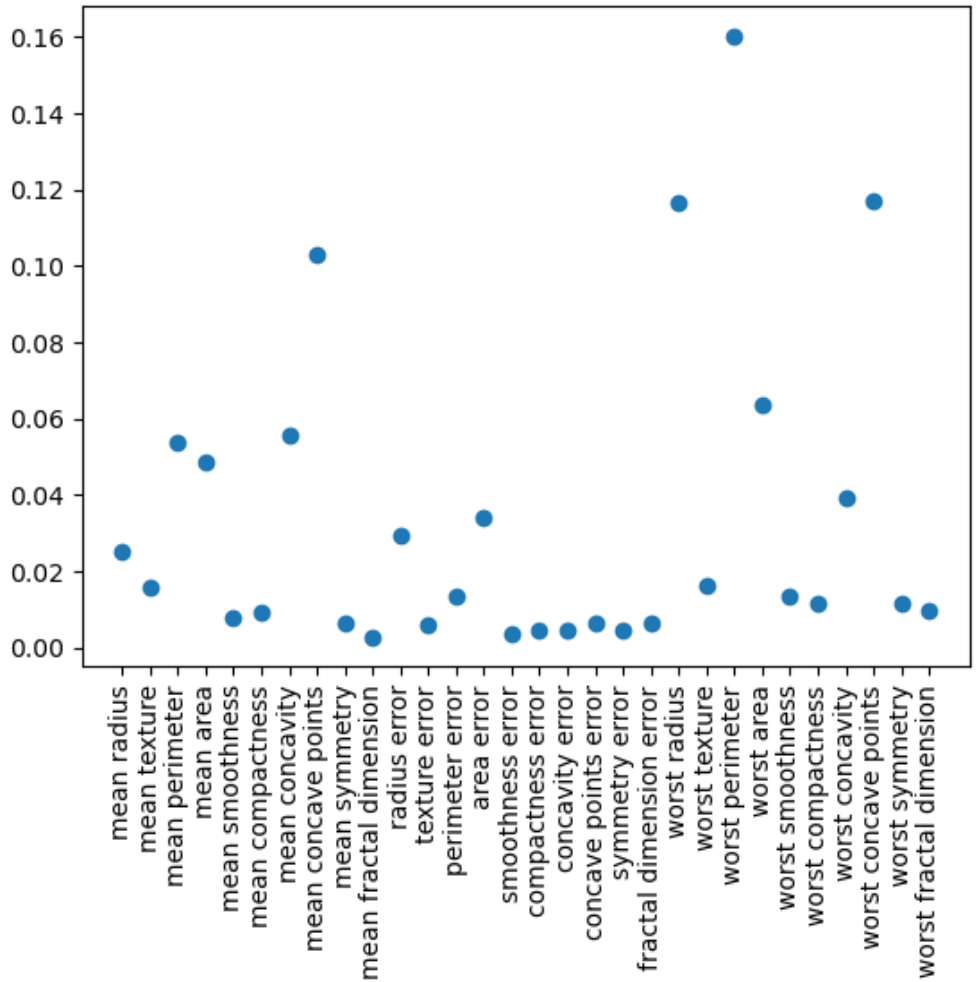
Random Forest on Tumor Classification

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
logisticregression = LogisticRegression().fit(X_train, y_train)
print("training set score: %f" % logisticregression.score(X_train, y_train))
print("test set score: %f" % logisticregression.score(X_test, y_test))
training set score: 0.946009
test set score: 0.951049
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("accuracy on training set: %f" % tree.score(X_train, y_train))
print("accuracy on test set: %f" % tree.score(X_test, y_test))
accuracy on training set: 1.000000
accuracy on test set: 0.881119
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("accuracy on training set: %f" % forest.score(X_train, y_train))
print("accuracy on test set: %f" % forest.score(X_test, y_test))
accuracy on training set: 1.000000
accuracy on test set: 0.972028
```

The random forest gives us an accuracy of 97%, better than the linear models or a single decision tree, without tuning any parameters.

Features Importance

Similarly to the decision tree, the random forest provides feature importances, which are computed by aggregating the feature importances over the trees in the forest. Typically the feature importances provided by the random forest are more reliable than the ones provided by a single tree.



Gradient Boosted Regression Trees

- Gradient boosted regression trees is another ensemble method that combines multiple decision trees to a more powerful model. In contrast to random forests, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one. Gradient boosted trees often use very shallow trees, of depth one to five, often making the model smaller in terms of memory, and making predictions faster.
- The main idea behind gradient boosting is to combine many simple models (in this context known as weak learners), like shallow trees. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance.
- They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameter are set correctly.

Boosting Algorithm

Boosting for Regression Trees

1. Set $\hat{f}(x) = 0$ and residual $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \cdots, B$, repeat:
   (a) Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x).$$
3. Output the boosted model,
$$\hat{f}(x) = \sum_{b=1}^{B} \hat{f}^b(x).$$

Tuning Parameters in Boosting

Boosting has three tuning parameters:

1. The number of trees B. Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
2. The shrinkage parameter λ, a small positive number. This controls the rate at which boosting learns. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Very small λ can require using a very large value of B in order to achieve good performance.
3. The number d of splits in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a stump, consisting of a single split. In this case, the boosted ensemble is fitting an additive model, since each term involves only a single variable. More generally d is the interaction depth, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Gradient Boosted Regression Trees

```
from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split(
 cancer.data, cancer.target, random_state=0)
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
accuracy on training set: 1.000000
accuracy on test set: 0.965035
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
accuracy on training set: 0.990610
accuracy on test set: 0.972028
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
accuracy on training set: 0.988263
accuracy on test set: 0.965035
gbrt = GradientBoostingClassifier(random_state=0, n_estimators=50)
gbrt.fit(X_train, y_train)
print("accuracy on training set: %f" % gbrt.score(X_train, y_train))
print("accuracy on test set: %f" % gbrt.score(X_test, y_test))
accuracy on training set: 1.000000
accuracy on test set: 0.972028
```

Summary of Tree Ensemble Methods

- In bagging, the trees are grown independently on random samples of the observations. Consequently, the trees tend to be quite similar to each other. Thus, bagging can get caught in local optima and can fail to thoroughly explore the model space.
- In random forests, the trees are once again grown independently on random samples of the observations. However, each split on each tree is performed using a random subset of the features, thereby decorrelating the trees, and leading to a more thorough exploration of model space relative to bagging.
- In boosting, we only use the original data, and do not draw any random samples. The trees are grown successively, using a "slow" learning approach: each new tree is fit to the signal that is left over from the earlier trees, and shrunken down before it is used.