Trees and Ensemble Learning Regression and Classification

APM Ch 8

IMLP 2.3.5 2.3.6

Table of Contents

Table of Contents

Tree Based Learning Classification

Imagine you want to distinguish between the following four animals: bears, hawks, penguins, and dolphins. Your goal is to get to the right answer by asking as few if/else questions as possible.

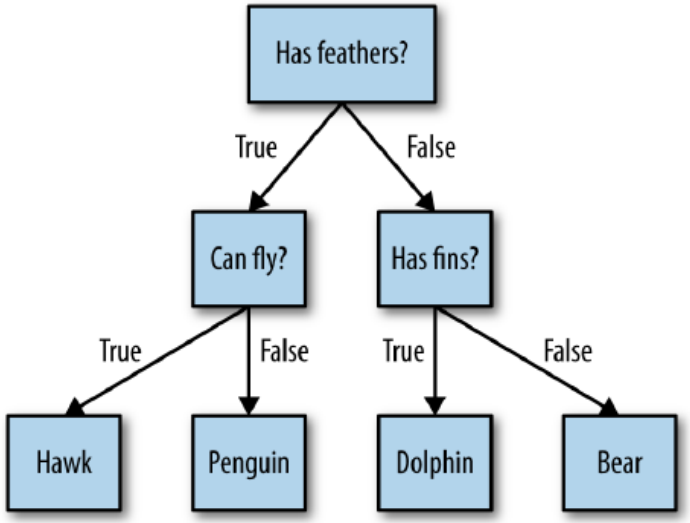


Figure 2-22. A decision tree to distinguish among several animals

Building decision trees

- Usually data does not come in the form of binary yes/no features as in the animal example, but is instead represented as continuous features. The tests that are used on continuous data are of the form “Is feature *i* larger than value *a*?”
- Let us build a decision tree for the following two moons dataset. X[0] will be feature 0 and X[1] will be feature 1. Class 0 is blue and Class 1 is red

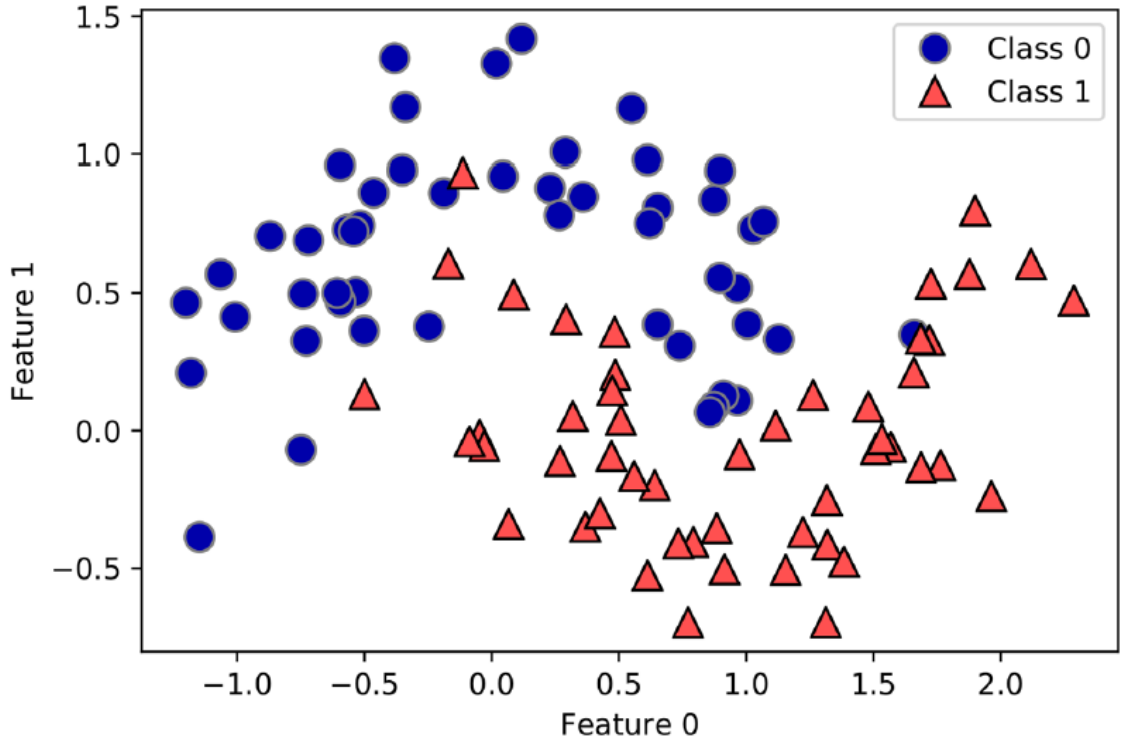


Figure 2-23. Two-moons dataset on which the decision tree will be built

Building decision trees

The split is done by testing whether x[1] <= 0.0596, indicated by a black line. If the test is true, a point is assigned to the left node, which contains 2 points belonging to class 0 and 32 points belonging to class 1. Otherwise the point is assigned to the right node, which contains 48 points belonging to class 0 and 18 points belonging to class 1.

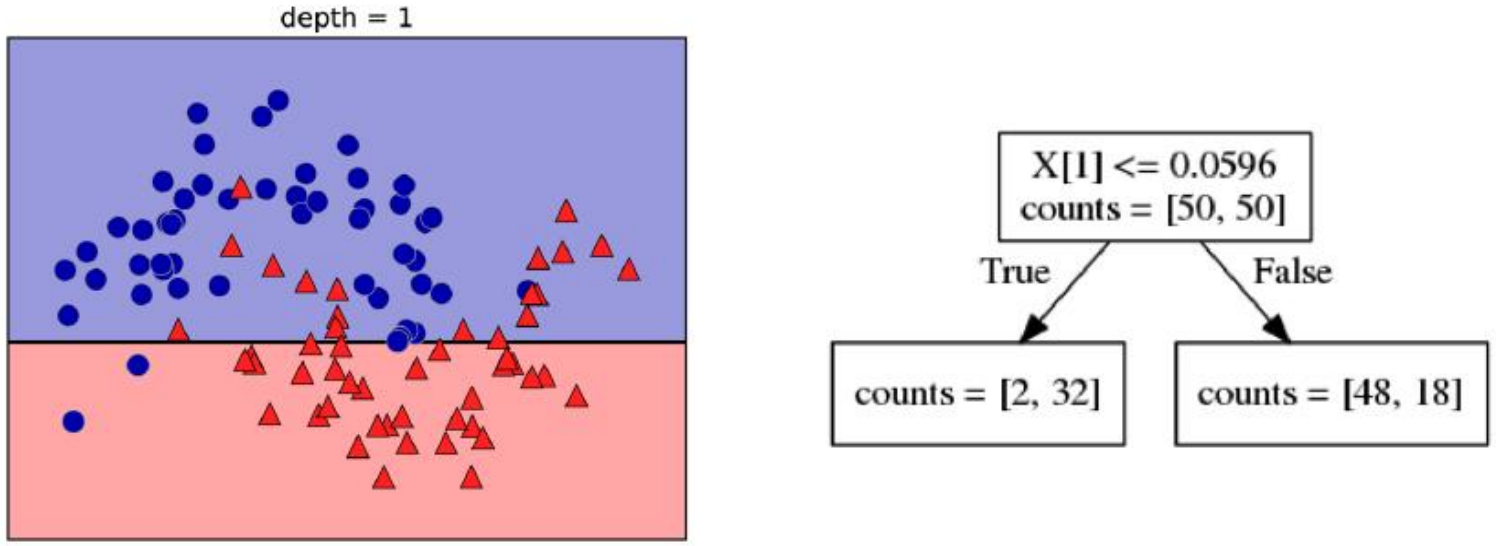


Figure 2-24. Decision boundary of tree with depth 1 (left) and corresponding tree (right)

Therefore one can test X[1] of the sample against .0596 and **if true** it most likely belongs to class 2
But **if it false** then what?

Building decision trees

Even though the first split did a good job of separating the two classes, the bottom region still contains points belonging to class 0, and the top region still contains points belonging to class 1. We can build a more accurate model by repeating the process of looking for the best test in both regions.

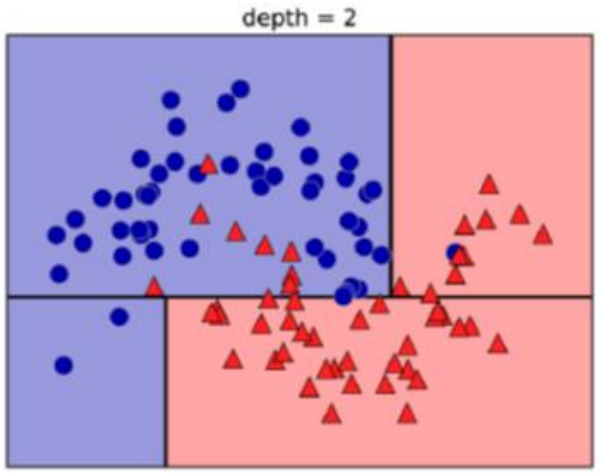
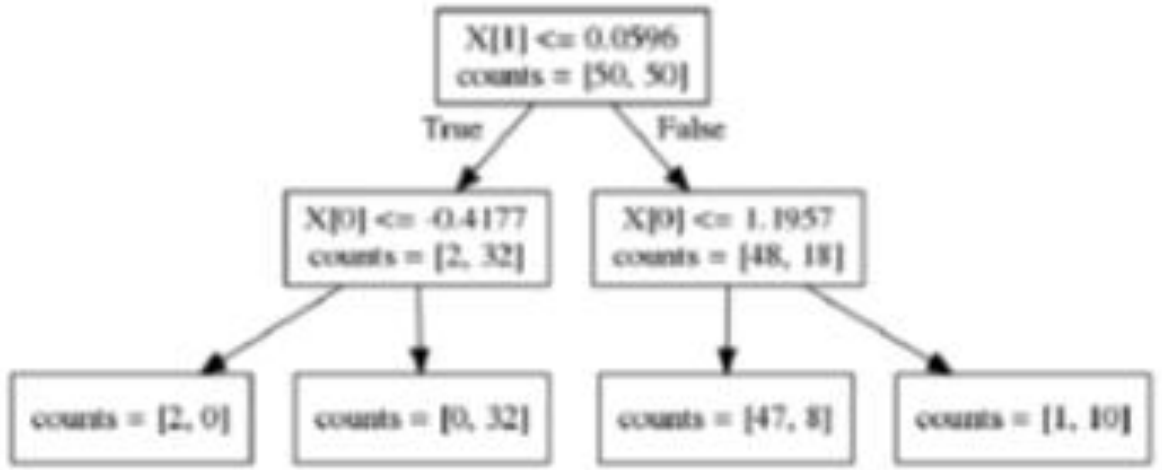


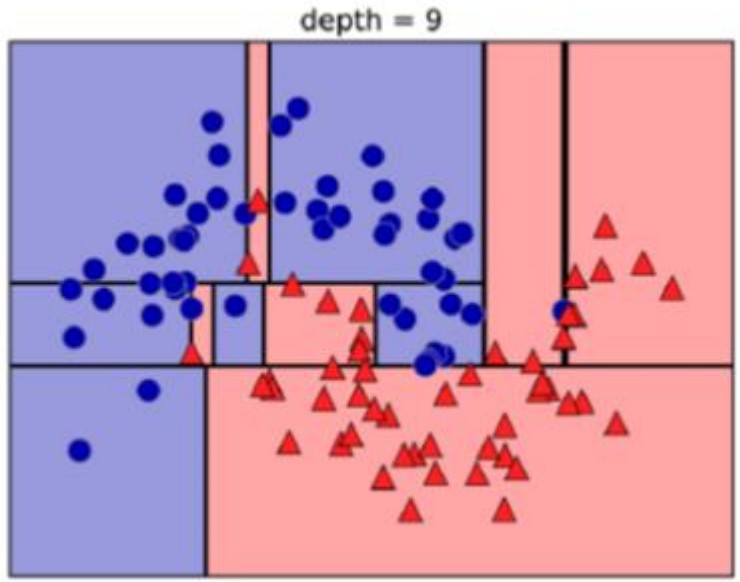
Figure 2-25. Decision boundary of tree with depth 2 tree (right)

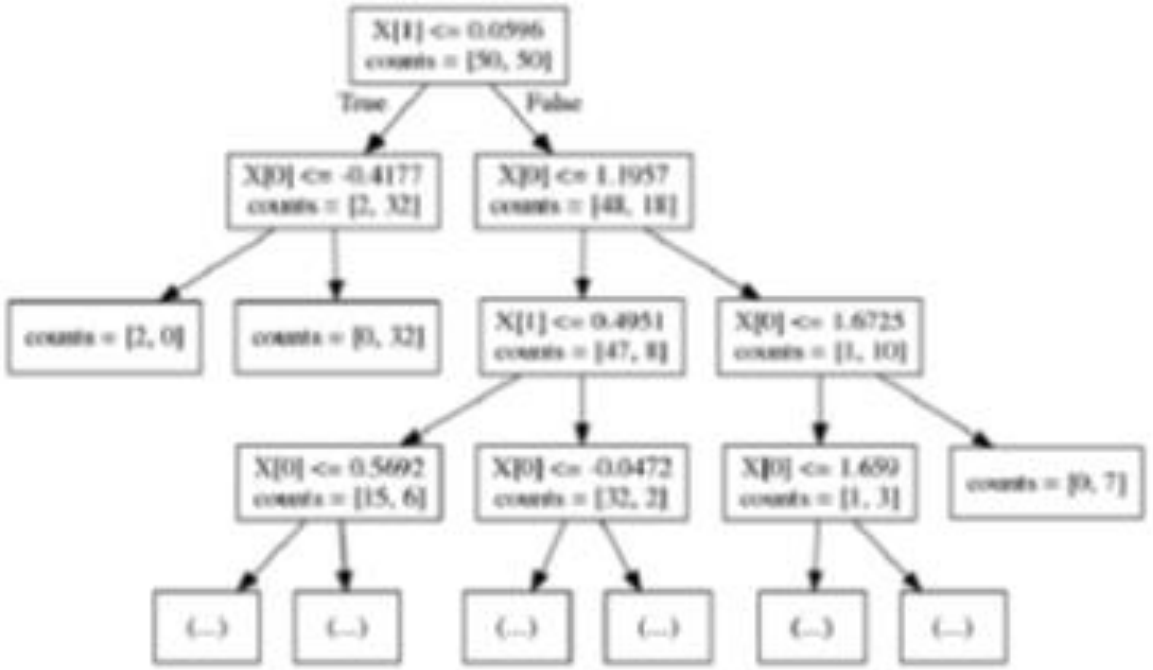


The samples that we split with X[1] are now split with X[2], leading to some paths having only one non-zero class. Once a leaf has only one non-zero the decision path, it is complete. Not all leaves have this property.

Building decision trees

The recursive partitioning of the data is repeated until each region in the partition (each leaf in the decision tree) only contains a single target value (a single class or a single regression value). A leaf of the tree that contains data points that all share the same target value is called *pure*.





Tree Based Learning Regression

Similarly, for regression, we can set the outcome

```
if Predictor A >= 1.7 then
        | if Predictor B >= 202.1 then Outcome = 1.3
        | else Outcome = 5.6
else Outcome = 2.5
```

Tree Based Learning

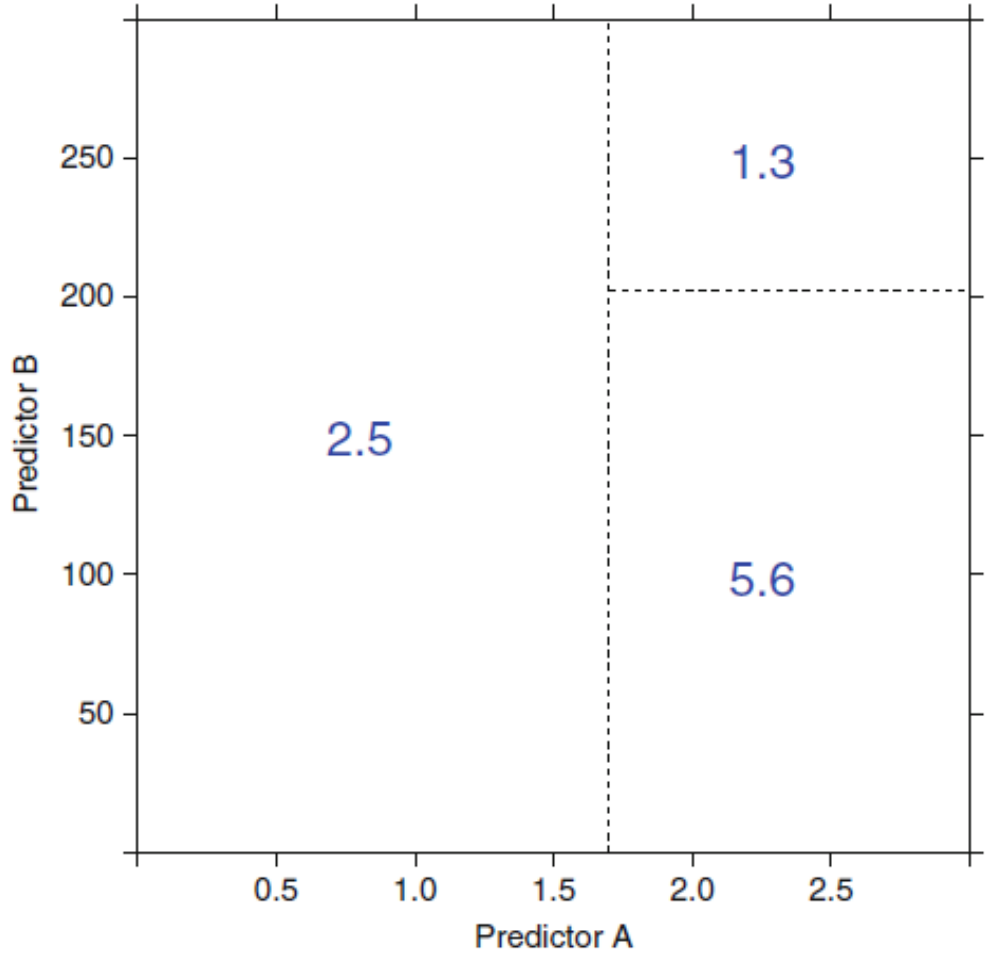


Fig. 8.1: An example of the predicted values within regions defined by a tree-based model

Controlling complexity of decision trees

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly over fit to the training data. The presence of pure leaves mean that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class

There are two common strategies to prevent overfitting:

- stopping the creation of the tree early (also called *pre-pruning*), Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

- building the tree but then removing or collapsing nodes that contain little information (also called *post-pruning* or just *pruning*). (*not implemented in scikit-learn*)

Computing

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
stratify=cancer.target, random_state=42)

tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

Computing with pre-pruning

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,
cancer.target, stratify=cancer.target, random_state=42)


tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test,
y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Computing - Feature importance in trees

Instead of looking at the whole tree, which can be taxing, there are some useful properties that we can derive to summarize the workings of the tree. The most commonly used summary is *feature importance*, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target.” The feature importance's always sum to 1:

```
print(tree.feature_importances_)
```

```
Feature importances:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.01 0.048 0. 0. 0.002 0. 0. 0. 0.
0. 0.727 0.046 0. 0. 0.014 0. 0.018 0.122 0.012 0. ]
```

However, *if a feature has a low value in feature_importance_, it doesn’t mean* that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.

Linear Regression vs Decision Tree (not pruned)

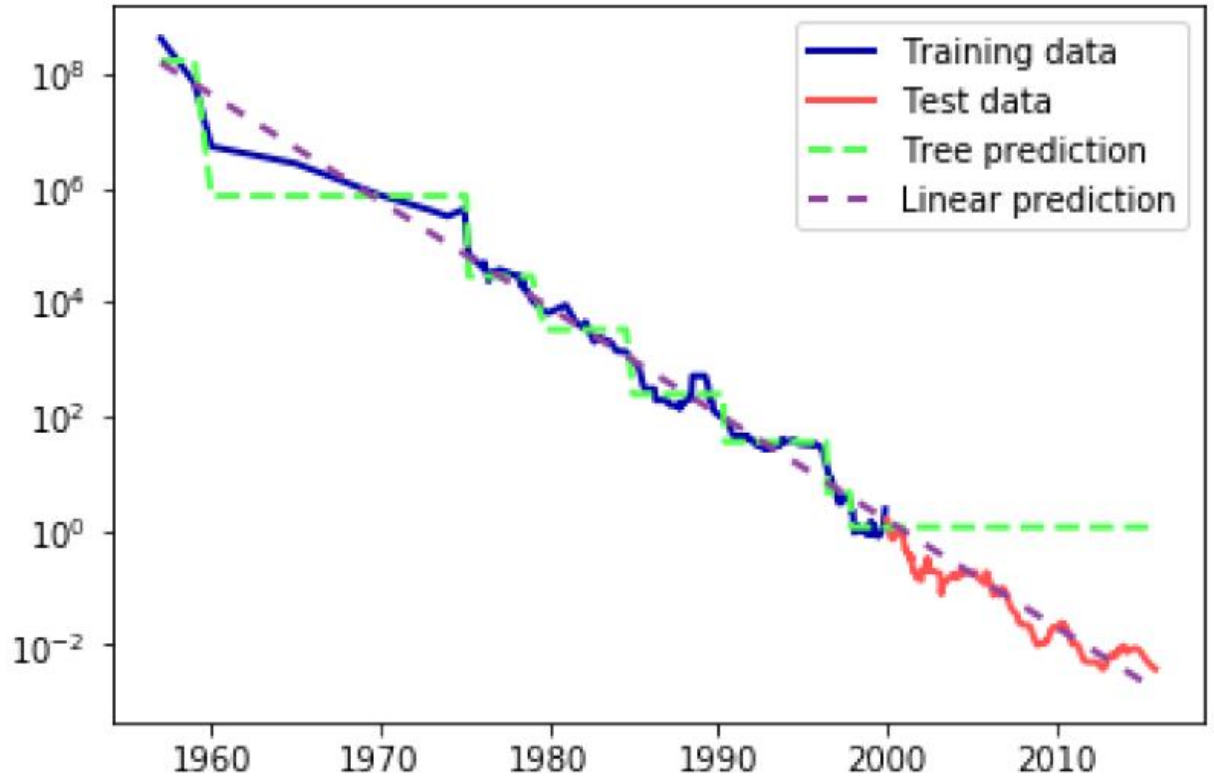


Figure 2-32. Comparison of predictions made by a linear model and predictions made by a regression tree on the RAM price data

Decision trees very well on training data but do very poorly on test data which are “extra polated”

Advantages of Tree Based Learning

- First, they generate a set of conditions that are highly interpretable and are easy to implement.
- Because of the logic of their construction, they can effectively handle many types of predictors (sparse, skewed, continuous, categorical, etc.) without the need to pre-process them.
- In addition, these models do not require the user to specify the form of the predictors' relationship to the response like, for example, a linear regression model requires.(like polynomial kernel etc)
- These models can effectively handle missing data and implicitly conduct feature selection, characteristics that are desirable for many real-life modeling problems.
- The model is not effected any type of scaling of the features.

Disadvantages of Tree Based Learning

(1) model instability (i.e., slight changes in the data can drastically change the structure of the tree or rules and, hence, the interpretation)

(2) Overfitting and less-than-optimal predictive performance.

(3) Poor extrapolation

Constructing Decision trees

- regression trees determine:

- The predictor to split on and value of the split

- The depth or complexity of the tree
- The prediction equation in the terminal nodes

Construction of Regression Trees

For regression, the model begins with the entire data set, S, and searches every distinct value of every predictor to find *the predictor* and *split value* that partitions the data into two groups (S_1 and S_2) such that the overall sums of squares error are minimized:

(predictor to split will be tried one after another and for each predictor, the midpoint of the range of values will be used and then move on to midpoint of each part)

$$\text{SSE} = \sum_{i \in S_1} (y_i - \bar{y}_1)^2 + \sum_{i \in S_2} (y_i - \bar{y}_2)^2,$$

where $\bar{y}_1$ and $\bar{y}_2$ are the averages of the training set outcomes within groups S_1 and S_2, respectively. Then within each of groups S_1 and S_2, this method searches for the predictor and split value that best reduces *SSE*. Because of the recursive splitting nature of regression trees, this method is also known as *recursive partitioning*.

Construction of Regression Trees

Once the full tree has been grown, the tree may be very large and is likely to over-fit the training set. The tree is then *pruned* back to a potentially smaller depth. The processed used is *cost-complexity tuning*. The goal of this process is to find a "right-sized tree" that has the smallest error rate. To do this, we penalize the error rate using the size of the tree:

$$\text{SSE}_{c_p} = \text{SSE} + c_p \times (\#\ \text{Terminal Nodes}),$$

where c_p is called the *complexity parameter*. For a specific value of the complexity parameter, we find the smallest pruned tree that has the lowest penalized error rate.

Construction of Classification Trees

- Two alternative measures, the Gini index (Breiman et al. 1984) and cross entropy, which is also referred to as deviance or information shift the focus from accuracy to purity. For the two-class problem, the Gini index for a given node is defined as

$$p_1 (1 - p_1) + p_2 (1 - p_2) ,$$

where $p1$ and $p2$ are the Class 1 and Class 2 probabilities, respectively. Since this is a two-class problem $p_1+p_2 = 1$. Gini index can be be written as $2p_1p_2$. It is easy to see that the Gini index is minimized when either of the class probabilities is driven towards zero, meaning that the node is pure with respect to one of the classes. Conversely, the Gini index is maximized when $p_1 = p_2$, the case in which the node is least pure.

Table of Contents

Ensemble Trees- Random Forests

- As we just observed, a main drawback of decision trees is that they tend to overfit the training data. Random forests are one way to address this problem. A random forest is essentially a collection of decision trees, where each tree is slightly different from the others.

- The idea behind random forests is that each tree might do a relatively good job of predicting, but will likely overfit on part of the data.

- There are two ways in which the trees in a random forest are randomized: by selecting the data points used to build a tree and by selecting the features in each split test.

Ensemble Trees- Random Forests

- To build a tree, we first take what is called a *bootstrap sample* of our data. That is, from our n_samples data points, we repeatedly *draw an example randomly with replacement* (meaning 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 (say, approximately one third), and some will be repeated.

- To illustrate, let's say we want to create a bootstrap sample of the list ['a', 'b', 'c', 'd']. A possible bootstrap sample would be ['b', 'd', 'd', 'c']. Another possible sample would be ['d', 'a', 'd', 'a'].

- 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 forest are different.

Ensemble Trees- Random Forests

- Next, a decision tree is built based on this newly created dataset.

- Instead of looking for the best test (feature/split point) for each node, in each node the *algorithm randomly selects a subset of the features*, and it looks for the best possible (feature/split) involving one of these features.

- The number of features that are selected is controlled by the max_features parameter.

- This selection of a subset of features is repeated separately in each node, so that each node in a tree can make a decision using a different subset of the features.

Ensemble Trees- Random Forests

- To make a prediction using the random forest, the algorithm first makes a prediction for every tree in the forest. For regression, we can average these results to get our final prediction.
- For classification, a “soft voting” strategy is used. This means each algorithm makes a “soft” prediction, providing a probability for each possible output label. The probabilities predicted by all the trees are averaged, and the class with the highest probability is predicted.

Random Forest Tree Algorithm

```
1  Select the number of models to build, m
2  for i = 1 to m do
3      Generate a bootstrap sample of the original data
4      Train a tree model on this sample
5      for each split do
6          Randomly select k (< P) of the original predictors
7          Select the best predictor among the k predictors and
           partition the data
8      end
9      Use typical tree model stopping criteria to determine when a
       tree is complete (but do not prune)
10 end
```

Algorithm 8.2: Basic Random Forests

As a starting point, m is chosen to be about 1,000 trees. If the cross-validation performance profiles are still improving at 1,000 trees, then incorporate more trees until performance levels off.

Computation

```
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
ax.set_title("Tree {}".format(i))
mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

n_estimators: Number of trees
random_state: Controls both the randomness of the bootstrapping of the samples used when building trees

Computation

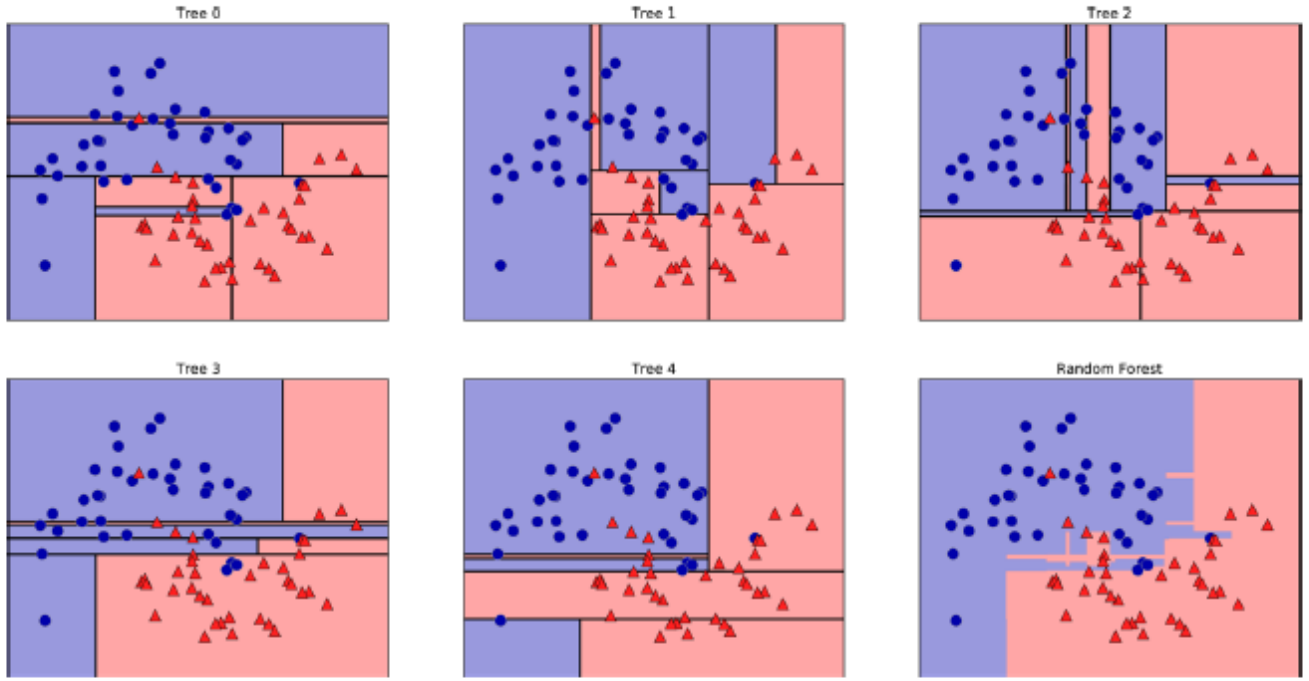


Figure 2-33. Decision boundaries found by five randomized decision trees and the decision boundary obtained by averaging their predicted probabilities

We can clearly see that the decision boundaries learned by the five trees are quite different.
Each of them makes some mistakes, as some of the training points that are plotted here were not actually included in the training sets of the trees, due to the bootstrap sampling.

Computation

```
X_train, X_test, y_train, y_test = train_test_split(
cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(forest.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

- Accuracy on training set: 1.000
- Accuracy on test set: 0.972

Strengths and Weaknesses of Random Forests

- Setting different random states (or not setting the random_state at all) can drastically change the model that is built. The more trees there are in the forest, the more robust it will be against the choice of random state. If you want to have reproducible results, it is important to fix the random_state.

- Random forests don't tend to perform well on very high dimensional, sparse data, such as text data. For this kind of data, linear models might be more appropriate.

- Random forests usually work well even on very large datasets, and training can easily be parallelized over many CPU cores within a powerful computer. However, random forests require more memory and are slower to train and to predict than linear models. If time and memory are important in an application, it might make sense to use a linear model instead.

- The important parameters to adjust are n_estimators, max_features, and possibly pre-pruning options like max_depth. For n_estimators, larger is always better. Averaging more trees will yield a more robust ensemble by reducing overfitting. However, there are diminishing returns, and more trees need more memory and more time to train. A common rule of thumb is to build "as many as you have time/memory for."

Table of Contents

Gradient Boosted Trees

- In contrast to the random forest approach, gradient boosting works by building trees in a serial manner
- Each tree tries to correct the mistakes of the previous one
- By default, there is no randomization in gradient boosted regression trees; instead, strong pre-pruning is used.
- Gradient boosted trees often use very shallow trees, of depth one to five, which makes the model smaller in terms of memory and makes predictions faster.

Gradient Boosted Trees

- The main idea behind gradient boosting is to combine many simple models (in this context known as *weak learners*), like shallow trees.
- Gradient boosted trees are frequently the winning entries in machine learning competitions, and are widely used in industry.
- They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly.

Gradient Boosted Trees

- Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the learning_rate, which controls how strongly each tree tries to correct the mistakes of the previous trees.
- A higher learning rate means each tree can make stronger corrections, allowing for more complex models.
- Adding more trees to the ensemble, which can be accomplished by increasing n_estimators, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

Gradient Boosted Trees Algorithm

1 Select tree depth, D, and number of iterations, K

2 Compute the average response, $\overline{y}$, and use this as the initial predicted value for each sample

3 **for** $k = 1$ *to* K **do**

4 Compute the residual, the difference between the observed value and the *current* predicted value, for each sample

5 Fit a regression tree of depth, D, using the residuals as the response

6 Predict each sample using the regression tree fit in the previous step

7 Update the predicted value of each sample by adding the previous iteration's predicted value to the predicted value generated in the previous step

8 **end**

Algorithm 8.3: Simple Gradient Boosting for Regression

Gradient Boosted 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:{:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

Gradient Boosted Trees

- As both gradient boosting and random forests perform well on similar kinds of data, a common approach is to first try random forests, which work quite robustly.

- If random forests work well but prediction time is at a premium, or it is important to squeeze out the last percentage of accuracy from the machine learning model, moving to gradient boosting often helps.