In-Depth: Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes; see "In Depth: Naive Bayes Classification" on page 382) and a powerful discriminative classifier (support vector machines; see "In-Depth: Support Vector Machines" on page 405). Here we'll take a look at motivating another powerful algorithm—a non-parametric algorithm called *random forests*. Random forests are an example of an *ensemble* method, a method that relies on aggregating the results of an ensemble of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts; that is, a majority vote among a number of estimators can end up being better than any of the individual estimators doing the voting! We will see examples of this in the following sections. We begin with the standard imports:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Motivating Random Forests: Decision Trees

Random forests are an example of an *ensemble learner* built on decision trees. For this reason we'll start by discussing decision trees themselves.

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero in on the classification. For example, if you wanted to build a decision tree to classify an animal you come across while on a hike, you might construct the one shown in Figure 5-67.

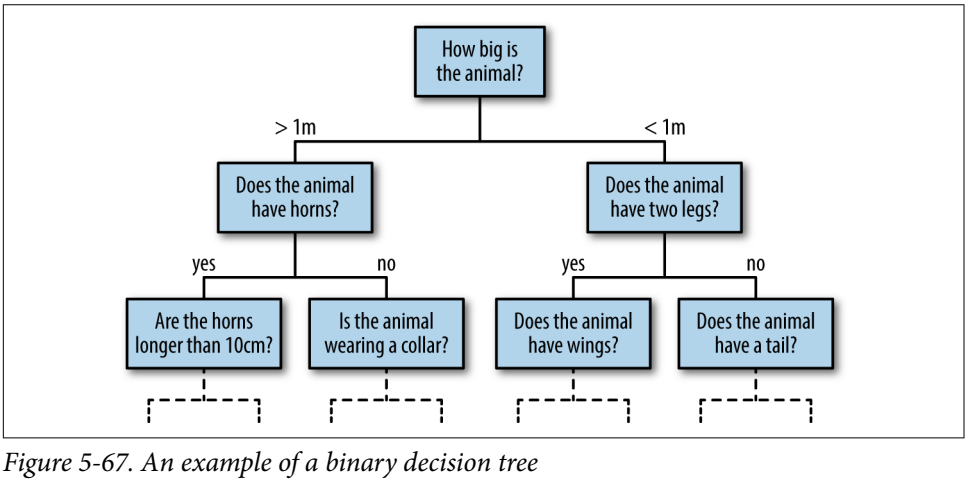

Figure 5-67. An example of a binary decision tree

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes. The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data; that is, each node in the tree splits the data into two groups using a cutoff value within one of the features. Let's now take a look at an example.

Creating a decision tree

Consider the following two-dimensional data, which has one of four class labels (Figure 5-68):

```
In[2]: from sklearn.datasets import make_blobs

       X, y = make_blobs(n_samples=300, centers=4,
                         random_state=0, cluster_std=1.0)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

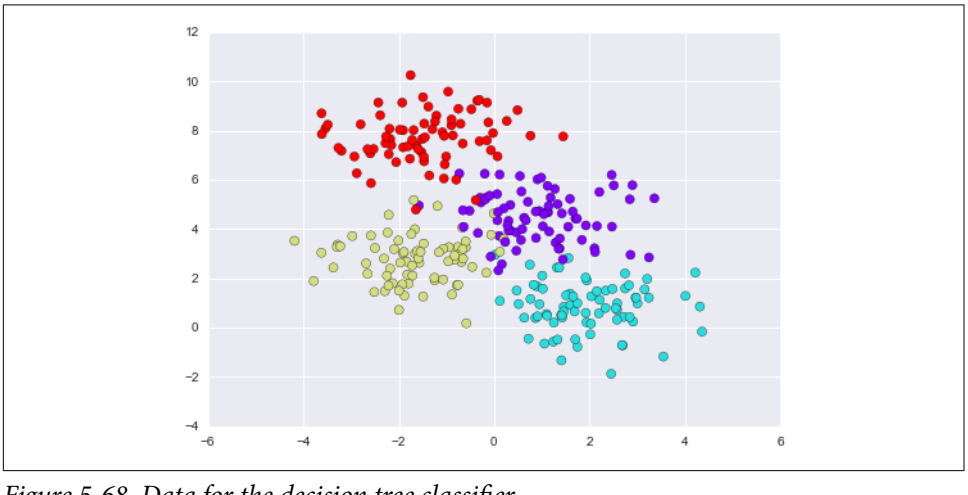

Figure 5-68. Data for the decision tree classifier

A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it. Figure 5-69 presents a visualization of the first four levels of a decision tree classifier for this data.

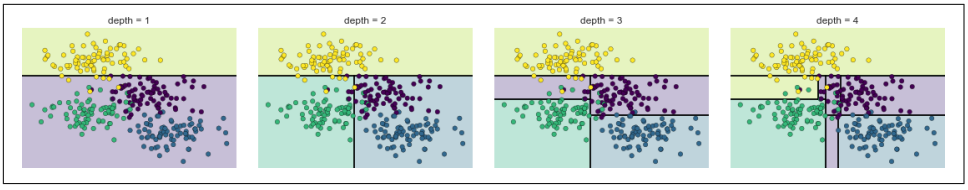

Figure 5-69. Visualization of how the decision tree splits the data

Notice that after the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch. Except for nodes that contain all of one color, at each level *every* region is again split along one of the two features.

This process of fitting a decision tree to our data can be done in Scikit-Learn with the `DecisionTreeClassifier` estimator:

```
In[3]: from sklearn.tree import DecisionTreeClassifier
       tree = DecisionTreeClassifier().fit(X, y)
```

Let's write a quick utility function to help us visualize the output of the classifier:

```
In[4]: def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
           ax = ax or plt.gca()

           # Plot the training points
           ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
                      clim=(y.min(), y.max()), zorder=3)
           ax.axis('tight')
           ax.axis('off')
           xlim = ax.get_xlim()
           ylim = ax.get_ylim()

           # fit the estimator
           model.fit(X, y)
           xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                                np.linspace(*ylim, num=200))
           Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

           # Create a color plot with the results
           n_classes = len(np.unique(y))
           contours = ax.contourf(xx, yy, Z, alpha=0.3,
                                  levels=np.arange(n_classes + 1) - 0.5,
                                  cmap=cmap, clim=(y.min(), y.max()),
                                  zorder=1)

           ax.set(xlim=xlim, ylim=ylim)
```

Now we can examine what the decision tree classification looks like (Figure 5-70):

```
In[5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

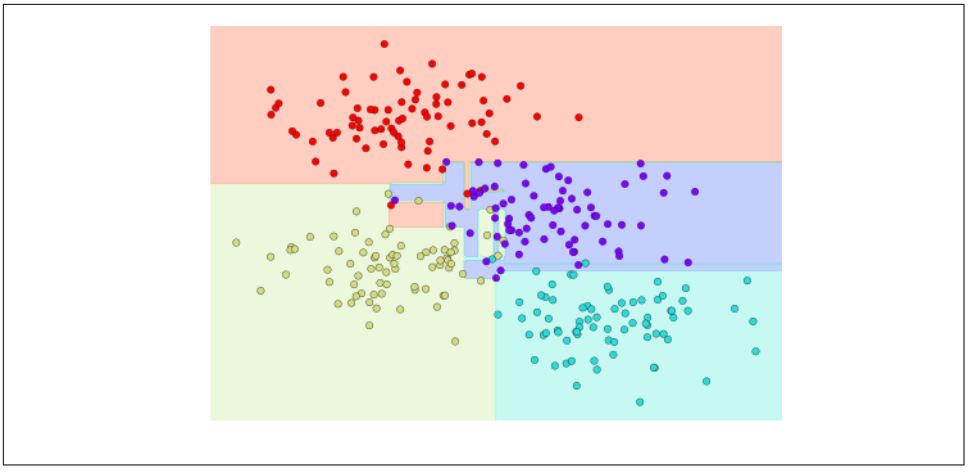

Figure 5-70. Visualization of a decision tree classification

If you're running this notebook live, you can use the helpers script included in the online appendix to bring up an interactive visualization of the decision tree building process (Figure 5-71):

```
In[6]: # helpers_05_08 is found in the online appendix
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.plot_tree_interactive(X, y);
```

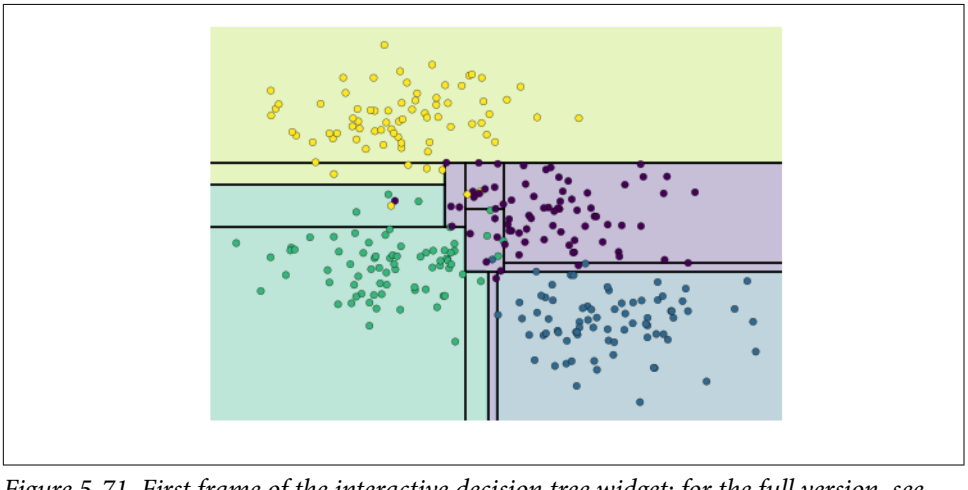

Figure 5-71. First frame of the interactive decision tree widget; for the full version, see the online appendix

Notice that as the depth increases, we tend to get very strangely shaped classification regions; for example, at a depth of five, there is a tall and skinny purple region

between the yellow and blue regions. It's clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise properties of the data. That is, this decision tree, even at only five levels deep, is clearly overfitting our data.

Decision trees and overfitting

Such overfitting turns out to be a general property of decision trees; it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions they are drawn from. Another way to see this overfitting is to look at models trained on different subsets of the data—for example, in Figure 5-72 we train two different trees, each on half of the original data.

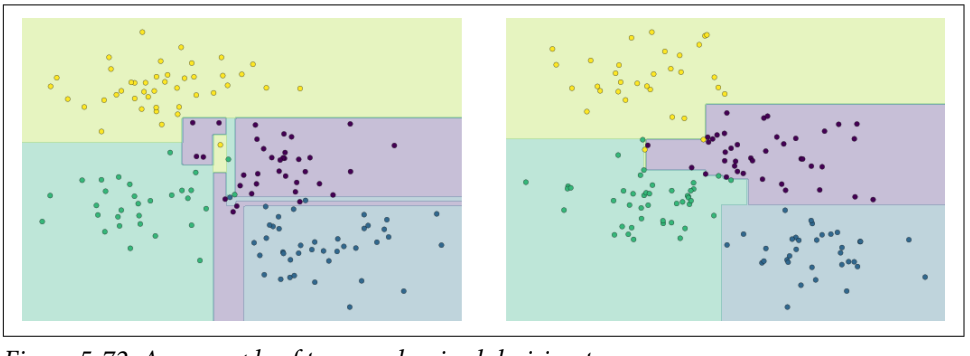

Figure 5-72. An example of two randomized decision trees

It is clear that in some places, the two trees produce consistent results (e.g., in the four corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from *both* of these trees, we might come up with a better result!

If you are running this notebook live, the following function will allow you to interactively display the fits of trees trained on a random subset of the data (Figure 5-73):

```
In[7]: # helpers_05_08 is found in the online appendix
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.randomized_tree_interactive(X, y)
```

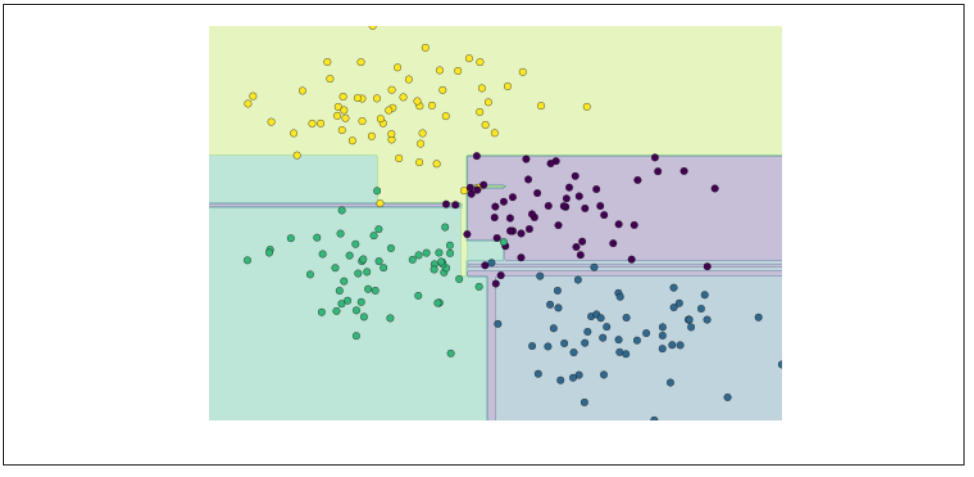

Figure 5-73. First frame of the interactive randomized decision tree widget; for the full version, see the online appendix

Just as using information from two trees improves our results, we might expect that using information from many trees would improve our results even further.

Ensembles of Estimators: Random Forests

This notion—that multiple overfitting estimators can be combined to reduce the effect of this overfitting—is what underlies an ensemble method called *bagging*. Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which overfits the data, and averages the results to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

We can do this type of bagging classification manually using Scikit-Learn's `Bagging Classifier` meta-estimator as shown here (Figure 5-74):

```
In[8]: from sklearn.tree import DecisionTreeClassifier
       from sklearn.ensemble import BaggingClassifier

       tree = DecisionTreeClassifier()
       bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                               random_state=1)

       bag.fit(X, y)
       visualize_classifier(bag, X, y)
```