Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because it's not feasible to train other models. However, in lower-dimensional spaces, other models might yield better generalization performance. We will look at some examples in which linear models fail in "Kernelized Support Vector Machines" on page 92.

Method Chaining

The `fit` method of all `scikit-learn` models returns `self`. This allows you to write code like the following, which we've already used extensively in this chapter:

`In[51]:`

```
# instantiate model and fit it in one line
logreg = LogisticRegression().fit(X_train, y_train)
```

Here, we used the return value of `fit` (which is `self`) to assign the trained model to the variable `logreg`. This concatenation of method calls (here `__init__` and then `fit`) is known as *method chaining*. Another common application of method chaining in `scikit-learn` is to `fit` and `predict` in one line:

`In[52]:`

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Finally, you can even do model instantiation, fitting, and predicting in one line:

`In[53]:`

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

This very short variant is not ideal, though. A lot is happening in a single line, which might make the code hard to read. Additionally, the fitted logistic regression model isn't stored in any variable, so we can't inspect it or use it to predict on any other data.

Naive Bayes Classifiers

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear models discussed in the previous section. However, they tend to be even faster in training. The price paid for this efficiency is that naive Bayes models often provide generalization performance that is slightly worse than that of linear classifiers like `LogisticRegression` and `LinearSVC`.

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in `scikit-`

learn: GaussianNB, BernoulliNB, and MultinomialNB. GaussianNB can be applied to any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). BernoulliNB and MultinomialNB are mostly used in text data classification.

The BernoulliNB classifier counts how often every feature of each class is not zero. This is most easily understood with an example:

In[54]:

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Here, we have four data points, with four binary features each. There are two classes, 0 and 1. For class 0 (the first and third data points), the first feature is zero two times and nonzero zero times, the second feature is zero one time and nonzero one time, and so on. These same counts are then calculated for the data points in the second class. Counting the nonzero entries per class in essence looks like this:

In[55]:

```
counts = {}
for label in np.unique(y):
    # iterate over each class
    # count (sum) entries of 1 per feature
    counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

Out[55]:

```
Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

The other two naive Bayes models, MultinomialNB and GaussianNB, are slightly different in what kinds of statistics they compute. MultinomialNB takes into account the average value of each feature for each class, while GaussianNB stores the average value as well as the standard deviation of each feature for each class.

To make a prediction, a data point is compared to the statistics for each of the classes, and the best matching class is predicted. Interestingly, for both MultinomialNB and BernoulliNB, this leads to a prediction formula that is of the same form as in the linear models (see "Linear models for classification" on page 56). Unfortunately, coef_ for the naive Bayes models has a somewhat different meaning than in the linear models, in that coef_ is not the same as *w*.

Strengths, weaknesses, and parameters

`MultinomialNB` and `BernoulliNB` have a single parameter, `alpha`, which controls model complexity. The way `alpha` works is that the algorithm adds to the data `alpha` many virtual data points that have positive values for all the features. This results in a "smoothing" of the statistics. A large `alpha` means more smoothing, resulting in less complex models. The algorithm's performance is relatively robust to the setting of `alpha`, meaning that setting `alpha` is not critical for good performance. However, tuning it usually improves accuracy somewhat.

`GaussianNB` is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text. `MultinomialNB` usually performs better than `BinaryNB`, particularly on datasets with a relatively large number of nonzero features (i.e., large documents).

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data and are relatively robust to the parameters. Naive Bayes models are great baseline models and are often used on very large datasets, where training even a linear model might take too long.

Decision Trees

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.

These questions are similar to the questions you might ask in a game of 20 Questions. 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. You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two. If the answer is "yes," you can ask another question that could help you distinguish between hawks and penguins. For example, you could ask whether the animal can fly. If the animal doesn't have feathers, your possible animal choices are dolphins and bears, and you will need to ask a question to distinguish between these two animals—for example, asking whether the animal has fins.

This series of questions can be expressed as a decision tree, as shown in Figure 2-22.

`In[56]:`

```
mglearn.plots.plot_animal_tree()
```