Summary and Outlook

This chapter introduced a range of unsupervised learning algorithms that can be applied for exploratory data analysis and preprocessing. Having the right representation of the data is often crucial for supervised or unsupervised learning to succeed, and preprocessing and decomposition methods play an important part in data preparation.

Decomposition, manifold learning, and clustering are essential tools to further your understanding of your data, and can be the only ways to make sense of your data in the absence of supervision information. Even in a supervised setting, exploratory tools are important for a better understanding of the properties of the data. Often it is hard to quantify the usefulness of an unsupervised algorithm, though this shouldn't deter you from using them to gather insights from your data. With these methods under your belt, you are now equipped with all the essential learning algorithms that machine learning practitioners use every day.

We encourage you to try clustering and decomposition methods both on two-dimensional toy data and on real-world datasets included in `scikit-learn`, like the `digits`, `iris`, and `cancer` datasets.

Summary of the Estimator Interface

Let's briefly review the API that we introduced in Chapters 2 and 3. All algorithms in `scikit-learn`, whether preprocessing, supervised learning, or unsupervised learning algorithms, are implemented as classes. These classes are called *estimators* in `scikit-learn`. To apply an algorithm, you first have to instantiate an object of the particular class:

In[87]:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

The estimator class contains the algorithm, and also stores the model that is learned from data using the algorithm.

You should set any parameters of the model when constructing the model object. These parameters include regularization, complexity control, number of clusters to find, etc. All estimators have a `fit` method, which is used to build the model. The `fit` method always requires as its first argument the data `X`, represented as a NumPy array or a SciPy sparse matrix, where each row represents a single data point. The data `X` is always assumed to be a NumPy array or SciPy sparse matrix that has continuous (floating-point) entries. Supervised algorithms also require a `y` argument, which is a one-dimensional NumPy array containing target values for regression or classification (i.e., the known output labels or responses).

There are two main ways to apply a learned model in `scikit-learn`. To create a prediction in the form of a new output like `y`, you use the `predict` method. To create a new representation of the input data `X`, you use the `transform` method. Table 3-1 summarizes the use cases of the `predict` and `transform` methods.

Table 3-1. scikit-learn API summary

`estimator.fit(x_train, [y_train])`	
`estimator.predict(X_text)`	`estimator.transform(X_test)`
Classification	Preprocessing
Regression	Dimensionality reduction
Clustering	Feature extraction
	Feature selection

Additionally, all supervised models have a `score(X_test, y_test)` method that allows an evaluation of the model. In Table 3-1, `X_train` and `y_train` refer to the training data and training labels, while `X_test` and `y_test` refer to the test data and test labels (if applicable).

CHAPTER 4
Representing Data and Engineering Features

So far, we've assumed that our data comes in as a two-dimensional array of floating-point numbers, where each column is a *continuous feature* that describes the data points. For many applications, this is not how the data is collected. A particularly common type of feature is the *categorical features*. Also known as *discrete features*, these are usually not numeric. The distinction between categorical features and continuous features is analogous to the distinction between classification and regression, only on the input side rather than the output side. Examples of continuous features that we have seen are pixel brightnesses and size measurements of plant flowers. Examples of categorical features are the brand of a product, the color of a product, or the department (books, clothing, hardware) it is sold in. These are all properties that can describe a product, but they don't vary in a continuous way. A product belongs either in the clothing department or in the books department. There is no middle ground between books and clothing, and no natural order for the different categories (books is not greater or less than clothing, hardware is not between books and clothing, etc.).

Regardless of the types of features your data consists of, how you represent them can have an enormous effect on the performance of machine learning models. We saw in Chapters 2 and 3 that scaling of the data is important. In other words, if you don't rescale your data (say, to unit variance), then it makes a difference whether you represent a measurement in centimeters or inches. We also saw in Chapter 2 that it can be helpful to *augment* your data with additional features, like adding interactions (products) of features or more general polynomials.

The question of how to represent your data best for a particular application is known as *feature engineering*, and it is one of the main tasks of data scientists and machine