- *Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances*[9] between the instances.
- *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).
- *Linear Discriminant Analysis* (LDA) is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

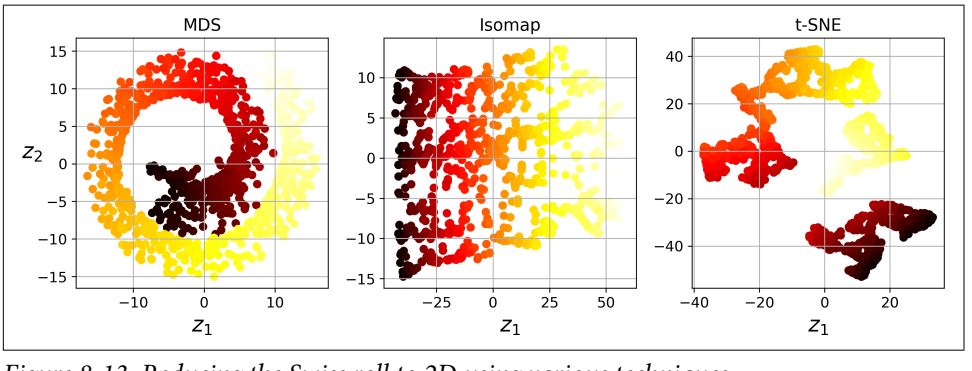

Figure 8-13. Reducing the Swiss roll to 2D using various techniques

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?

9 The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

6. In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA, or Kernel PCA?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next evaluate the classifier on the test set: how does it compare to the previous classifier?
10. Use t-SNE to reduce the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can write colored digits at the location of each instance, or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms such as PCA, LLE, or MDS and compare the resulting visualizations.

Solutions to these exercises are available in ???.

CHAPTER 9
Unsupervised Learning Techniques

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 9 in the final release of the book.

Although most of the applications of Machine Learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is actually unlabeled: we have the input features **X**, but we do not have the labels **y**. Yann LeCun famously said that "if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake". In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

For example, say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks. But wait, there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as "defective" or "normal". This will generally require human experts to sit down and manually go through all the pictures. This is a long, costly and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier's performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch. Wouldn't it

be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

In Chapter 8, we looked at the most common unsupervised learning task: dimensionality reduction. In this chapter, we will look at a few more unsupervised learning tasks and algorithms:

- *Clustering*: the goal is to group similar instances together into *clusters*. This is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.
- *Anomaly detection*: the objective is to learn what "normal" data looks like, and use this to detect abnormal instances, such as defective items on a production line or a new trend in a time series.
- *Density estimation*: this is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset. This is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Ready for some cake? We will start with clustering, using K-Means and DBSCAN, and then we will discuss Gaussian mixture models and see how they can be used for density estimation, clustering, and anomaly detection.

Clustering

As you enjoy a hike in the mountains, you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not perfectly identical, yet they are sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don't need an expert to identify groups of similar-looking objects. This is called *clustering*: it is the task of identifying similar instances and assigning them to *clusters*, i.e., groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, this is an unsupervised task. Consider Figure 9-1: on the left is the iris dataset (introduced in Chapter 4), where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as Logistic Regression, SVMs or Random Forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the top left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the lower right cluster is actually composed of two distinct sub-clusters. That said, the dataset actually has two additional features (sepal length and width), not

represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).

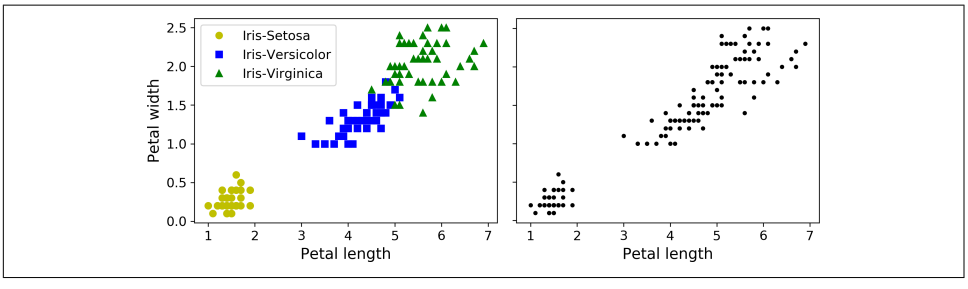

Figure 9-1. Classification (left) versus clustering (right)

Clustering is used in a wide variety of applications, including:

- For customer segmentation: you can cluster your customers based on their purchases, their activity on your website, and so on. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, this can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.
- For data analysis: when analyzing a new dataset, it is often useful to first discover clusters of similar instances, as it is often easier to analyze clusters separately.
- As a dimensionality reduction technique: once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instance's feature vector $\mathbf{x}$ can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k dimensional. This is typically much lower dimensional than the original feature vector, but it can preserve enough information for further processing.
- For *anomaly detection* (also called *outlier detection*): any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second, and so on. Anomaly detection is particularly useful in detecting defects in manufacturing, or for *fraud detection*.
- For semi-supervised learning: if you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This can greatly increase the amount of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

- For search engines: for example, some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database: similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is to find this image's cluster using the trained clustering model, and you can then simply return all the images from this cluster.
- To segment an image: by clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to reduce the number of different colors in the image considerably. This technique is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. For example, some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms: K-Means and DBSCAN, and we will show some of their applications, such as non-linear dimensionality reduction, semi-supervised learning and anomaly detection.

K-Means

Consider the unlabeled dataset represented in Figure 9-2: you can clearly see 5 blobs of instances. The K-Means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at the Bell Labs in 1957 as a technique for pulse-code modulation, but it was only published outside of the company in 1982, in a paper titled "Least square quantization in PCM".[1] By then, in 1965, Edward W. Forgy had published virtually the same algorithm, so K-Means is sometimes referred to as Lloyd-Forgy.

1 "Least square quantization in PCM," Stuart P. Lloyd. (1982).

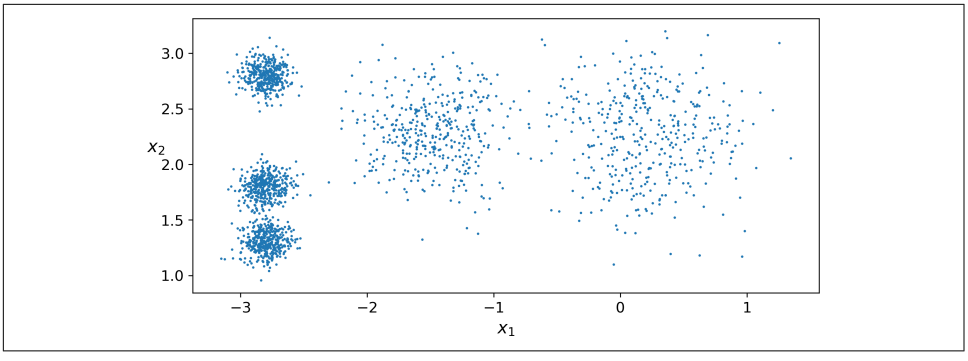

Figure 9-2. An unlabeled dataset composed of five blobs of instances

Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters k that the algorithm must find. In this example, it is pretty obvious from looking at the data that k should be set to 5, but in general it is not that easy. We will discuss this shortly.

Each instance was assigned to one of the 5 clusters. In the context of clustering, an instance's *label* is the index of the cluster that this instance gets assigned to by the algorithm: this is not to be confused with the class labels in classification (remember that clustering is an unsupervised learning task). The `KMeans` instance preserves a copy of the labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the 5 centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

Of course, you can easily assign new instances to the cluster whose centroid is closest:

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

If you plot the cluster's decision boundaries, you get a Voronoi tessellation (see Figure 9-3, where each centroid is represented with an X):

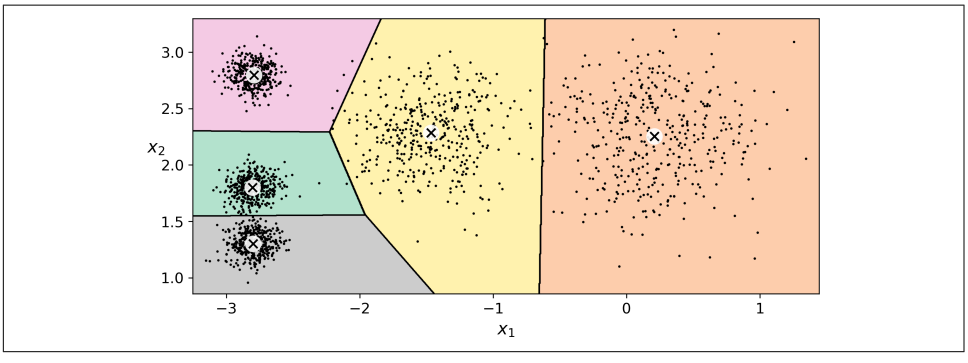

Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled (especially near the boundary between the top left cluster and the central cluster). Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters since all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to just give each instance a score per cluster: this is called *soft clustering*. For example, the score can be the distance between the instance and the centroid, or conversely it can be a similarity score (or affinity) such as the Gaussian Radial Basis Function (introduced in Chapter 5). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

In this example, the first instance in `X_new` is located at a distance of 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid and 2.87 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a k-dimensional dataset: this can be a very efficient non-linear dimensionality reduction technique.

The K-Means Algorithm

So how does the algorithm work? Well it is really quite simple. Suppose you were given the centroids: you could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate all the centroids by computing the mean of the instances for each cluster. But you are given neither the labels nor the centroids, so how can you proceed? Well, just start by placing the centroids randomly (e.g., by picking *k* instances at random and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small), it will not oscillate forever[2]. You can see the algorithm in action in Figure 9-4: the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, in just 3 iterations the algorithm has reached a clustering that seems close to optimal.

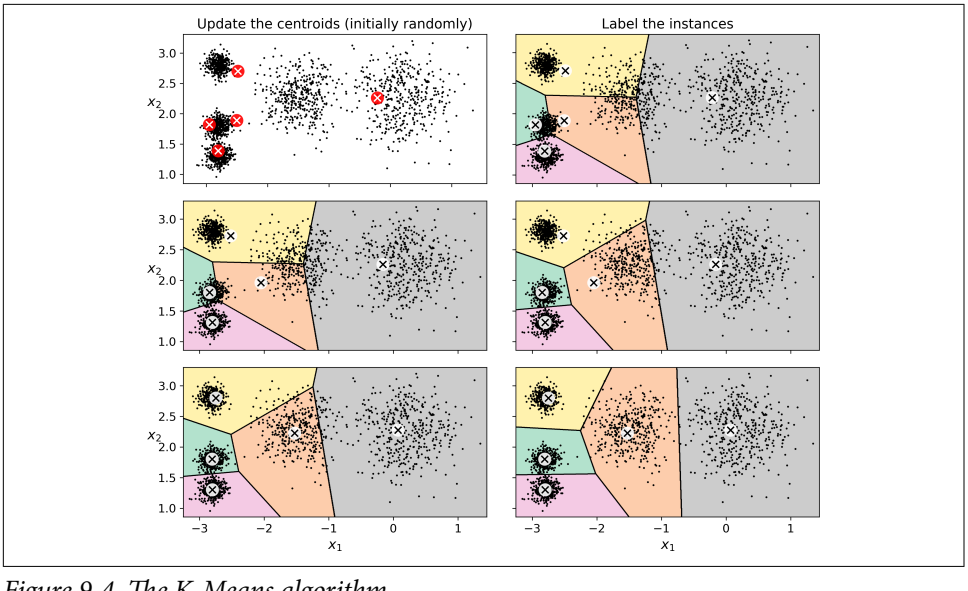

Figure 9-4. The K-Means algorithm

2 This can be proven by pointing out that the mean squared distance between the instances and their closest centroid can only go down at each step.

The computational complexity of the algorithm is generally linear with regards to the number of instances m, the number of clusters k and the number of dimensions n. However, this is only true when the data has a clustering structure. If it does not, then in the worst case scenario the complexity can increase exponentially with the number of instances. In practice, however, this rarely happens, and K-Means is generally one of the fastest clustering algorithms.

Unfortunately, although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): this depends on the centroid initialization. For example, Figure 9-5 shows two sub-optimal solutions that the algorithm can converge to if you are not lucky with the random initialization step:

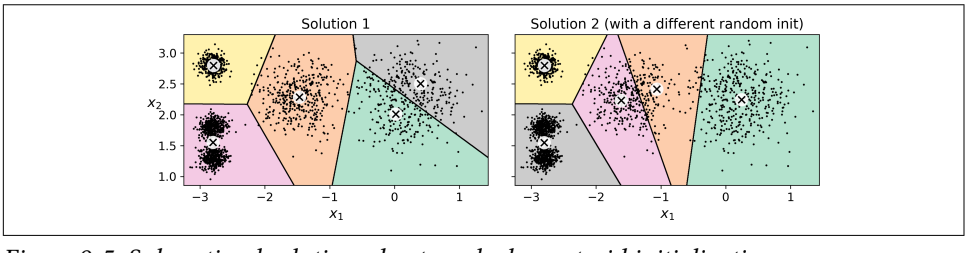

Figure 9-5. Sub-optimal solutions due to unlucky centroid initializations

Let's look at a few ways you can mitigate this risk by improving the centroid initialization.

Centroid Initialization Methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids, and set `n_init` to 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. This is controlled by the `n_init` hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier actually runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? Well of course it uses a performance metric! It is called the model's *inertia*: this is the mean squared distance between each instance and its closest centroid. It is roughly equal to 223.3 for the model on the left of Figure 9-5, 237.5 for the model on the right of Figure 9-5, and 211.6 for the model in Figure 9-3. The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia: in this example, the model in Figure 9-3 will be selected (unless we are very unlucky with `n_init` consecutive random initiali-