6
Clustering

So far, we have considered ML models which require labeled data in order to learn. However, there is a large class of models which can learn from *unlabeled* data. From this chapter, we will begin to introduce models from this modeling paradigm, called *unsupervised learning*. In this chapter, we focus on one application of unsupervised learning, called *clustering algorithm*.

6.1 Unsupervised Learning

Unsupervised learning is a branch of machine learning which only uses unlabeled data. Examples of unlabeled data include a text corpus containing the works of William Shakespeare (Chapter 8) or a set of unlabeled images (Chapter 7). Some key goals in this setting include:

- *Learn the structure of data:* It is possible to learn if the data consists of clusters, or if it can be represented in a lower dimension.
- *Learn the probability distribution of data:* By learning the probability distribution where the training data came from, it is possible to generate synthetic data which is "similar" to real data.
- *Learn a representation for data:* We can learn a representation that is useful in solving other tasks later. With this new representation, for example, we can reduce the need for labeled examples for classification.

6.2 Clustering

Clustering is one of the main tasks in unsupervised learning. It is the process of detecting *clusters* in the dataset. Often the membership of a cluster can replace the role of a label in the training dataset. In general, clusters reveal a lot of information about the underlying structure of the data.

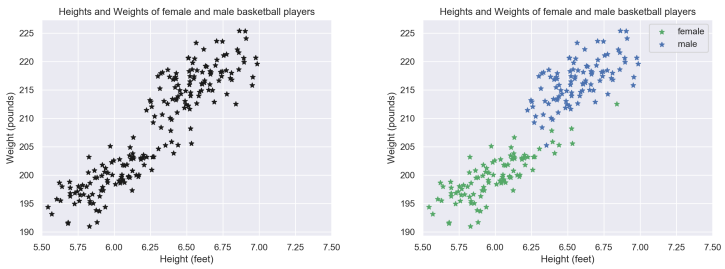

Figure 6.1: Height vs weight scatter plot of basketball players. In the plot on the right, the points in green and blue respectively correspond to female and male players.

In Figure 6.1, we see a scatter plot of measurements of height and weight of basketball players. If you look at the plot on the left, it is easy to conclude that there is a usual linear relationship between the height and the weight of the athletes. However, upon further inspection, it seems like there are two clusters of the data points, separated around the middle of the plot. In fact, this is indeed the case! If we label the dataset with the additional information of whether the data point is from a male or female athlete, the plot on the right shows something more than just the linear relationship. In practice, however, we do not always have access to this additional label. Instead, one uses clustering algorithms to find natural clusterings of the data. This raises the question of what a "clustering" is, in the first place.

Technically, any partition of the dataset $\mathcal{D}$ into k subsets $C_1, C_2, \ldots, C_k$ can be called a clustering. [1] That is,

$$\bigcup_{i=1}^{k} C_i = \mathcal{D} \quad \text{and} \quad \bigcap_{i=1}^{k} C_i = \emptyset$$

But we intuitively understand that not all partitions are a natural clustering of the dataset; our goal therefore will be to define what a "good" clustering is.

[1] Here k, the number of clusters may be given as part of the problem, or k may have to be decided upon after looking at the dataset. We'll revisit this soon.

6.2.1 *Some Attempts to Define a "Good" Cluster*

The sample data in Figure 6.1 suggests that our vision system has evolved to spot natural clusterings in two or three dimensional data. To do machine learning, however, we need a more precise definition in $\mathbb{R}^d$: specifically, for any partition of the dataset into clusters, we try to quantify the "goodness" of the clusters.

Definition 6.2.1 (Cluster: Attempt 1)**.** *A "good" **cluster** is a subset of points which are closer to each other than to all other points in the dataset.*

But this definition does not apply to the clusters in Figure 6.1. The points in the middle of the plot are far away from the points on the top right corner or the bottom left corner. So whichever cluster we assign the middle points to, they will be farther away from some

points in their assigned cluster than to some of the points on the other cluster. Ok, so that did not work. Consider the following definition.

Definition 6.2.2 (Cluster: Attempt 2)**.** *A "good" **cluster** is a subset of points which are closer to the **mean** of their own cluster than to the mean of other clusters.*

Here Mean and Variance are defined as follows:

Definition 6.2.3 (Mean and Variance of Clusters)**.** *Let C_i be one of the clusters for a dataset $\mathcal{D}$. Let $m_i = |C_i|$ denote the **cluster size**. The **mean** of the cluster C_i is*

$$\vec{\mathbf{y}}_i = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \vec{\mathbf{x}}$$

*and the **variance** within the cluster C_i is*

$$\sigma_i^2 = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$$

You may notice that Definition 6.2.2 appears to be using circular reasoning: it defines clusters using the mean of the clusters, but the mean can only be calculated once the clusters have been defined. [2]

[2] Such circular reasoning occurs in most natural formulations of clustering. Look at the Wikipedia page on clustering for some other formulations.

6.3 k-Means Clustering

In this section, we present a particular partition of the dataset called the *k-means clustering*. Given k, the desired number of clusters, the k-means clustering partitions $\mathcal{D}$ into k clusters $C_1, C_2, \ldots, C_k$ so as to minimize the cost function:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2 \tag{6.1}$$

This can be seen as minimizing the average of the individual *cost* of the k clusters, where cost of C_i is $\sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$. [3] This idea is similar in spirit to our earlier attempt in Definition 6.2.2 — we want the distance of each data point to the mean of the cluster to be small. But this method is able to circumvent the problem of circular reasoning.

[3] Notice that each cluster cost is the cluster size times the variance.

The process of finding the optimal solution for (6.1) is called the *k-means clustering problem*.

6.3.1 k-Means Algorithm

Somewhat confusingly, the most famous algorithm that is used to solve the *k-means clustering problem* is also called *k-means*. It is technically a *heuristic*, meaning it makes intuitive sense but it is not

guaranteed to find the optimum solution.[4] The following is the k-means algorithm. It is given some *initial* clustering (we discuss some choices for initialization below) and we repeat the following iteration until we can no longer improve the cost function:

[4] There is extensive research on finding near-optimal solutions to k-means. The problem is known to be NP-complete, so we believe that an algorithm that is guaranteed to produce the optimum solution on *all instances* must require exponential time.

Maintain clusters $C_1, C_2, \ldots, C_k$
For each cluster C_i, find the mean $\vec{\mathbf{y}}_i$
Initialize new clusters $C'_i \leftarrow \emptyset$
for $\vec{\mathbf{x}} \in \mathcal{D}$ **do**
  $i_x = \arg\min_i \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2$
  $C'_{i_x} \leftarrow C'_{i_x} \cup \{\vec{\mathbf{x}}\}$
end for
Update clusters $C_i \leftarrow C'_i$

At each iteration, we find the mean of each current cluster. Then for each data point, we assign it to the cluster whose mean is the closest to the point, without updating the mean of the clusters. In case there are multiple cluster means that the point is closest to, we apply the tie-breaker rule that the point gets assigned to the current cluster if it is among the closest ones; otherwise, it will be randomly assigned to one of them. Once we have assigned all points to the new clusters, we update the current set of clusters, thereby updating the mean of the clusters as well. We repeat this process until there is no point that is mis-assigned.

6.3.2 *Why Does k-Means Algorithm Terminate in Finite time?*

The k-means algorithm is actually quite akin to Gradient Descent, in the sense that the iterations are trying to improve the cost.

Lemma 6.3.1. *Given a set of points $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \ldots, \vec{\mathbf{x}}_m$, their mean $\vec{\mathbf{y}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$ is the point that minimizes the average squared distance to the points.*

Proof. For any vector $\vec{\mathbf{z}}$, let $C(\vec{\mathbf{z}})$ denote the sum of squared distance to the set of points. That is,

$$\begin{aligned} C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} \|\vec{\mathbf{z}} - \vec{\mathbf{x}}_i\|_2^2 &= \sum_{i=1}^{m} ((\vec{\mathbf{z}} - \vec{\mathbf{x}}_i) \cdot (\vec{\mathbf{z}} - \vec{\mathbf{x}}_i)) \\ &= \sum_{i=1}^{m} (\vec{\mathbf{z}} \cdot \vec{\mathbf{z}} - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \vec{\mathbf{x}}_i \cdot \vec{\mathbf{x}}_i) \\ &= \sum_{i=1}^{m} (\|\vec{\mathbf{z}}\|_2^2 - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \|\vec{\mathbf{x}}_i\|_2^2) \end{aligned}$$

To find the optimal $\vec{\mathbf{z}}$, we set the gradient ∇C to 0

$$\nabla C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} (2\vec{\mathbf{z}} - 2\vec{\mathbf{x}}_i) = 0$$

which yields the solution

$$\vec{\mathbf{z}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$$

□

We are ready to prove the main result.

Theorem 6.3.2. *Each iteration of the k-means Algorithm 6.3.1, possibly except for the last iteration before termination, strictly decreases the total cluster cost (6.1).*

Proof. We follow the same notation as in Algorithm 6.3.1. The total cost at the end of one iteration is:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2$$

where $\vec{\mathbf{y}}_i'$ is the mean of the newly defined cluster C_i'. Notice that each of the cluster cost $\sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2$ is the sum of the squared distance between a set of points $\vec{\mathbf{x}} \in C_i'$ and their mean. By Lemma 6.3.1, this sum is smaller than the sum of squared distance between the same set of points to any other point. In particular, we can compare with $\vec{\mathbf{y}}_i$, the mean of C_i before the update. That is,

$$\sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2 \leq \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$$

for any $1 \leq i \leq k$. If we sum over all clusters, we see that

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i' \right\|_2^2 \leq \sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$$

Now notice that the summand $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2$ in the right hand side of the inequality is the squared distance between the point $\vec{\mathbf{x}}$ and the mean $\vec{\mathbf{y}}_i$ (before update) of the cluster C_i' that $\vec{\mathbf{x}}$ is newly assigned to. In other words, we can rewrite this term as $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2$ and instead sum over all points $\vec{\mathbf{x}}$ in the dataset. That is,

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2^2 = \sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2$$

Finally, recall that the index i_x was defined as $i_x = \arg\min_i \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_i \right\|_2$. In particular, if j was the index of the cluster that a data point $\vec{\mathbf{x}}$ originally belonged to, then $\left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2 \leq \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_j \right\|_2^2$. Therefore, we have the following inequality,

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x} \right\|_2^2 \leq \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \left\| \vec{\mathbf{x}} - \vec{\mathbf{y}}_j \right\|_2^2$$

The equality holds in the inequality above if and only if when $\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 = \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$ for each point $\vec{\mathbf{x}}$, which means that the original cluster C_j was one of the closest clusters to $\vec{\mathbf{x}}$. By the tie-breaker rule, i_x would have been set to j. This is exactly the case when the algorithm terminates immediately after this iteration since no point is reassigned to a different cluster. In all other cases, we have a strict inequality:

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 < \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$$

Notice that the right hand side of the inequality is the total cost at the beginning of the iteration. □

Now we are ready to prove that the k-means algorithm is guaranteed to terminate in finite time. Since each iteration strictly reduces the cost, we conclude that the current clustering (*i.e.*, partition) will never be considered again, except at the last iteration when the algorithm terminates. Since there is only a finite number of possible partitions of the dataset $\mathcal{D}$, the algorithm must terminate in finite time.

6.3.3 *k-Means Algorithm and Digit Classification*

You might be familiar with the MNIST hand-written digits dataset. Here, each image, which depicts some digit between 0 and 9, is represented as a an 8×8 matrix of pixels and each pixel can take on a different luminosity value from 0 to 15.

We can apply k-means clustering to differentiate between images depicting the digit "1" and the digit "0." After running the model with $k = 2$ on 360 images of the two digits, we achieve the clusters in Figure 6.2. [5] Note the presence of two colored regions: a point is colored red if a hypothetical held-out data point at that location would get assigned a "0;" otherwise it is colored blue. This assignment is based on which cluster center is closer.

[5] This 2D visualization of the clusters is achieved through a technique called low dimensional representation, which is covered in Chapter 7.

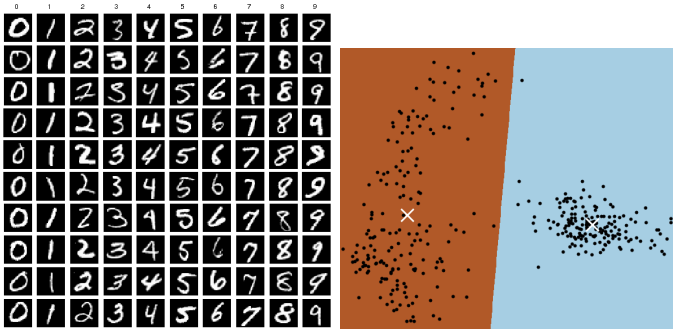

Figure 6.2: Sample images from the MNIST dataset (left) and 2D visualization of the k-means clusters differentiating between the digits "1" and "0" (right). Only two images were misclassified!

This example also shows that clustering into two clusters can be turned into a technique for binary classification — use training data to come up with two clusters; at test time, compute a ± 1 label for each data point according to which of the two cluster centers it is closer to.

6.3.4 *Implementation Detail: How to Pick the Initial Clustering*

The choice of initial clusters greatly influences the quality of the solution found by the k-means algorithm. The most naive method is to pick k data points randomly to serve as the initial cluster centers and create k clusters by assigning each data point to the closest cluster center. However, this approach can be problematic. Suppose there exists some "ground truth" clustering of the dataset. By picking the initial clusters randomly, we may end up splitting one of these ground truth clusters (*e.g.*, two initial centers are drawn from within the same ground truth cluster), and the final clustering ends up being very sub-optimal. Thus one tries to select the initial clustering more intelligently. For instance the popular k-means++ initialization procedure [6] is the following:

[6] It was invented by Arthur and Vassilvitskii in 2007.

1. Choose one center uniformly at random among all data points.
2. For each data point $\vec{\mathbf{x}}$ compute $D(\vec{\mathbf{x}})$, the distance between $\vec{\mathbf{x}}$ and the nearest center which has already been chosen.
3. Choose a new data point at random as a new center, where a point $\vec{\mathbf{x}}$ is chosen with probability proportional to $D(\vec{\mathbf{x}})^2$.
4. Repeat steps 2 and 3 until k centers have been chosen.

In COS 324, we will not expect you to understand why this is a good initialization procedure, but you may be expected to be able to implement this or similar procedures in code.

6.3.5 *Implementation Detail: Choice of k*

Above we assumed that the number of clusters k is given, but in practice you have to choose the appropriate number of clusters k.

Example 6.3.3. *Is there a value of k that guarantees an optimum cost of 0? Yes! Just set $k = n$ (i.e., each point is its own cluster). Of course, this is useless from a modeling standpoint!*

Problem 6.3.4. *Argue that the optimum cost for $k + 1$ clusters is no more than the optimum cost for k clusters.*

Note that Problem 6.3.4 only concerns the optimum cost, which as we discussed may not be attained by the k-means algorithm. Nevertheless, it does suggest that we can try various values of k and see when the cost is low enough to be acceptable.

A frequent heuristic is the *elbow method*: create a plot of the number of clusters vs. the final value of the cost as in Figure 6.3 and look for an "elbow" where the objective tapers off. Note that if the dataset is too complicated for a simple Euclidean distance cost, the data might not be easy to cluster "nicely" meaning there is no "elbow" shown on the plot.

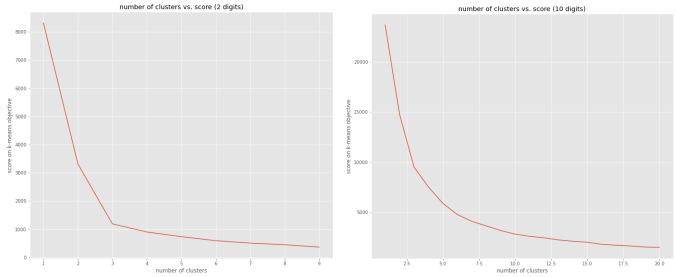

Figure 6.3: Two graphs of number of clusters vs. final value of cost. There is a distinct elbow on the left, but not on the right.

6.4 *Clustering in Programming*

In this section, we briefly discuss how to implement k-means algorithm for digit classification in Python. As usual, we use the *numpy* package to speed up computation and the *matplotlib* package for visualization. Additionally, we use the *sklearn* package to help perform the clustering.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA

# prepare dataset
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...

# define functions
def initialize_cluster_mean(X, k):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # returns Y: array of shape (k, d), each row is the center of a cluster

def assign_cluster(X, Y)
    # X: array of shape (n, d), each row is a d-dimensional data point
    # Y: array of shape (k, d), each row is the center of a cluster
    # returns loss, the sum of squared distance from each point to its
                                        assigned cluster
```

```
    # returns C: array of shape (n), each value is the index of the closest
                                        cluster

def update_cluster_mean(X, k, C):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # C: array of shape (n), each value is the index of the closest cluster
    # returns Y: array of shape (k, d), each row is the center of a cluster

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y

def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()

# run k-means algorithm and plot the result
loss, C, Y = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

We start by importing outside packages.

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
```

The *load_digits()* method loads the MNIST digits dataset, with around 180 data points per digit. The *scale()* method linearly scales each of the data points such that the mean is 0 and variance is 1. The *PCA()* method helps visualize the MNIST digits data points, which are 64-dimensional, in the Cartesian plane (*i.e.*, $\mathbb{R}^2$). See the next Chapter 7 for details on this process.

Next we prepare the dataset by calling the *load_digits()* method.

```
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...
```

Notice that we discard the target array y because we are performing clustering, a type of unsupervised learning. If we were to perform supervised learning instead, we would need to make use of y.

Then we define the functions necessary for the k-means algorithm.

```
def initialize_cluster_mean(X, k):
    return Y

def assign_cluster(X, Y)
    return loss, C

def update_cluster_mean(X, k, C):
    return Y

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y
```

In practice, it is common to limit the number of cluster update iterations (*i.e.*, the parameter *max_iters*) and specify the smallest amount of loss change allowed for one iteration (*i.e.*, the constant ϵ). By terminating the algorithm once either one of the conditions is reached, we can get an approximate solution within a reasonable amount of time.

Next, take a look at the helper function used to plot the result of the k-means algorithm.

```
def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()
```

The *cycle()* method from the *itertools* package lets you iterate through an array indefinitely, with the index wrapping around back to the start, once it reaches the end of the array.

Now, consider the *for* loop section in the helper function above. We first use *Boolean* conditions to concisely generate a new array.

```
idx = (C == i)
```

This generates a *Boolean* array with the same length as C, where each entry is either *True/False* based on whether the corresponding entry in C is equal to i. The following code is equivalent.

```
idx = np.zeros(C.size)
for j in range(C.size):
```

```
    idx[j] = (C[j] == i)
```

We then use a technique called *Boolean masking* to extract a particular subset of rows of X.

```
X[idx, 0]
```

Notice that in place of a list of indices of rows to extract, we are indexing with the *Boolean* array we just defined. The code will extract only the rows where the *Boolean* value is *True*. For example, if the value of *idx* is *[True, False, True]*, then the code above is equivalent to

```
X[[0, 2], 0]
```

Finally, we make use of the helper functions we defined earlier to run the k-means algorithm and plot results.

```
_, C, _ = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

The first line of this code snippet shows how we can use the _ symbol to selectively disregard individual return values of a function call. The second line of code uses the *PCA()* method to transform the 64-dimensional data *X_train* into 2-dimensional data so that we can visualize it with the *scatter_plot()* method. We will learn the details of this process in the next Chapter 7.