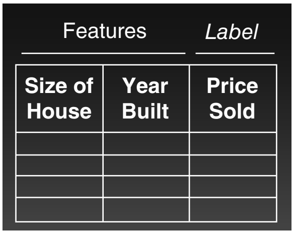
CHAPTER

10

Unsupervised Learning—Clustering Using K-Means

What Is Unsupervised Learning?

So far, all of the machine learning algorithms that you have seen are supervised learning. That is, the datasets have all been labeled, classified, or categorized. Datasets that have been labeled are known as *labeled data*, while datasets that have not been labeled are known as *unlabeled data*. Figure 10.1 shows an example of labeled data.

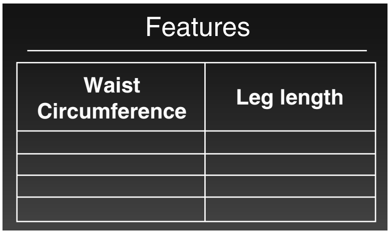


Features		*Label*
Size of House	Year Built	Price Sold

Figure 10.1: Labeled data

Based on the size of the house and the year in which it was built, you have the price at which the house was sold. The selling price of the house is the *label*, and your machine learning model can be trained to give the estimated worth of the house based on its size and the year in which it was built.

Unlabeled data, on the other hand, is data without label(s). For example, Figure 10.2 shows a dataset containing a group of people's waist circumference and corresponding leg length. Given this set of data, you can try to cluster them into groups based on the waist circumference and leg length, and from there you can figure out the average dimension in each group. This would be useful for clothing manufacturers to tailor different sizes of clothing to fit its customers.



Features	
Waist Circumference	Leg length

Figure 10.2: Unlabeled data

Unsupervised Learning Using K-Means

Since there is no label in unlabeled data, it is thus of interest to us that we are able to find patterns in that unlabeled data. This technique of finding patterns in unlabeled data is known as *clustering*. The main aim of clustering is to segregate groups with similar traits and assign them into groups (commonly known as *clusters*).

One of the common algorithms used for clustering is the K-Means algorithm. K-Means clustering is a type of unsupervised learning:

- Used when you have unlabeled data
- The goal is to find groups in data, with the number of groups represented by K

The goal of K-Means clustering is to achieve the following:

- K centroids representing the center of the clusters
- Labels for the training data

In the next section, you will learn how clustering using K-Means works.

How Clustering in K-Means Works

Let's walk through a simple example so that you can see how clustering using K-Means works. Suppose you have a series of unlabeled points, as shown in Figure 10.3.

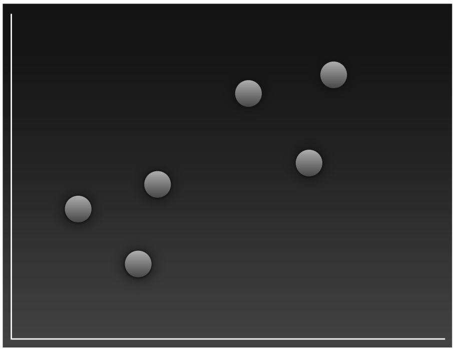


Figure 10.3: A set of unlabeled data points

Your job is to cluster all of these points into distinct groups so that you can discover a pattern among them. Suppose you want to separate them into two groups (that is, K=2). The end result would look like Figure 10.4.

Figure 10.4: Clustering the points into two distinct clusters

First, you will randomly put K number of centroids on the graph. In Figure 10.5, since K equals 2, we will randomly put two centroids on the graph: C_0 and C_1. For each point on the graph, measure the distance between itself and each of the centroids. As shown in the figure, the distance (represented by d_0) between a and C_0 is shorter than the distance (represented by d_1) between a and C_1. Hence, a is now classified as cluster 0. Likewise, for point b, the distance between itself and C_1 is shorter than the distance between itself and C_0. Hence, point b is classified as cluster 1. You repeat this process for all the points in the graph.

After the first round, the points would be clustered, as shown in Figure 10.6.

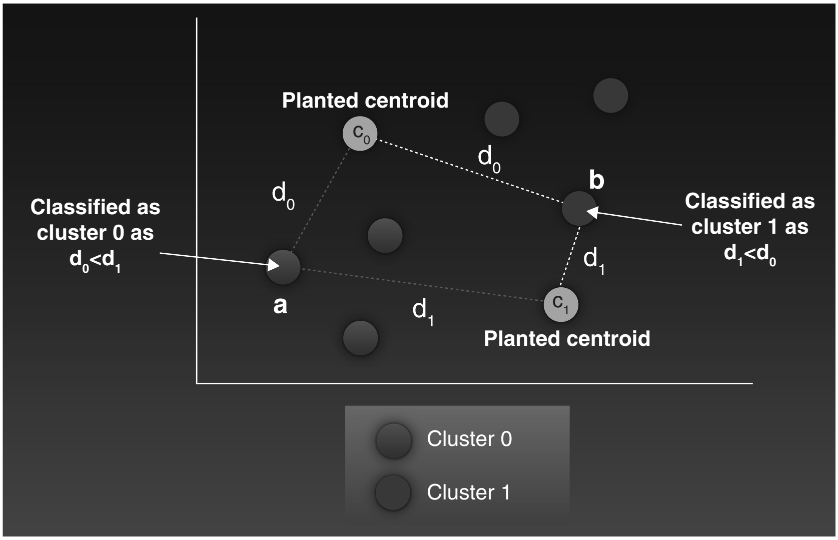


Figure 10.5: Measuring the distance of each point with respect to each centroid and finding the shortest distance

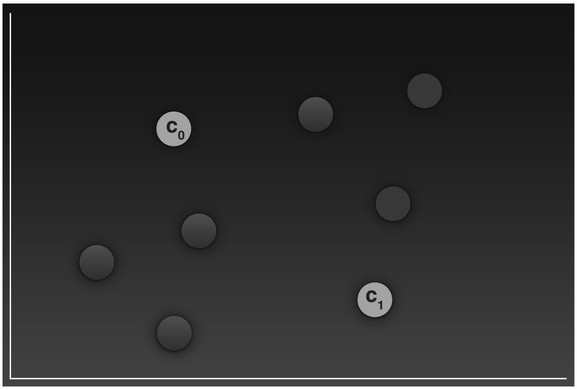


Figure 10.6: Groupings of the points after the first round of clustering

Now take the average of all of the points in each cluster and reposition the centroids using the newly calculated average. Figure 10.7 shows the new positions of the two centroids.

You now measure the distance between each of the old centroids and the new centroids (see Figure 10.8). If the distance is 0, that means that the centroid did not change position and hence the centroid is found. You repeat the entire process until all the centroids do not change position anymore.

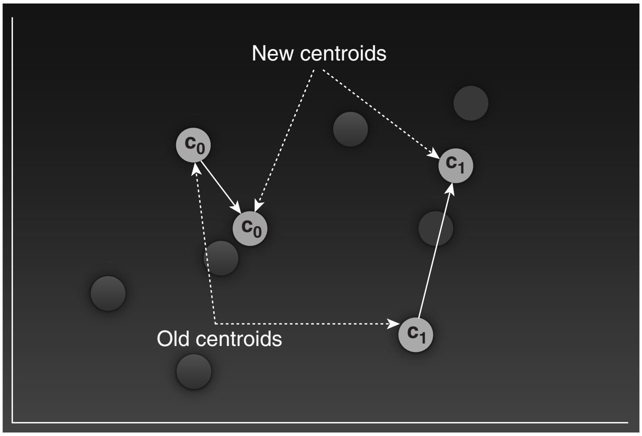


Figure 10.7: Repositioning the centroids by taking the average of all the points in each cluster

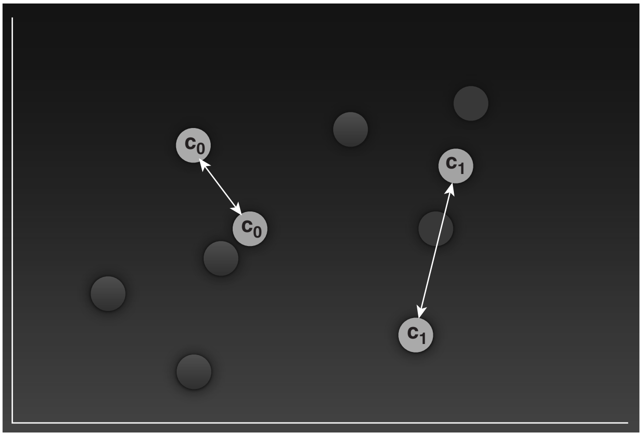


Figure 10.8: Measuring the distance between each centroid; if the distance is 0, the centroid is found

Implementing K-Means in Python

Now that you have a clear picture of how K-Means works, it is useful to implement this using Python. You will first implement K-Means using Python, and then see how you can use Scikit-learn's implementation of K-Means in the next section.

Suppose you have a file named `kmeans.csv` with the following content:

```
x,y
1,1
2,2
```

```
2,3
1,4
3,3
6,7
7,8
6,8
7,6
6,9
2,5
7,8
8,9
6,7
7,8
3,1
8,4
8,6
8,9
```

Let's first import all of the necessary libraries:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Then load the CSV file into a Pandas dataframe, and plot a scatter plot showing the points:

```
df = pd.read_csv("kmeans.csv")
plt.scatter(df['x'],df['y'], c='r', s=18)
```

Figure 10.9 shows the scatter plot with the points.

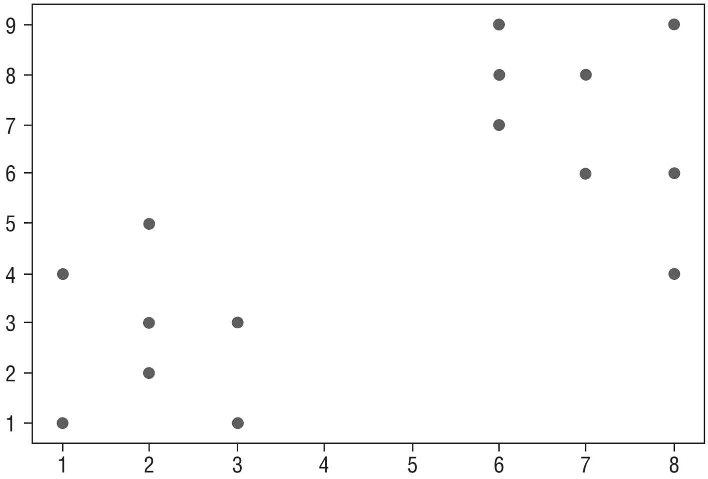


Figure 10.9: The scatter plot showing all the points

You can now generate some random centroids. You also need to decide on the value of K. Let's assume K to be 3 for now. You will learn how to determine the optimal K later in this chapter. The following code snippet generates three random centroids and marks them on the scatter plot:

```
#---let k assume a value---
k = 3

#---create a matrix containing all points---
X = np.array(list(zip(df['x'],df['y'])))

#---generate k random points (centroids)---
Cx = np.random.randint(np.min(X[:,0]), np.max(X[:,0]), size = k)
Cy = np.random.randint(np.min(X[:,1]), np.max(X[:,1]), size = k)

#---represent the k centroids as a matrix---
C = np.array(list(zip(Cx, Cy)), dtype=np.float64)
print(C)

#---plot the orginal points as well as the k centroids---
plt.scatter(df['x'], df['y'], c='r', s=8)
plt.scatter(Cx, Cy, marker='*', c='g', s=160)
plt.xlabel("x")
plt.ylabel("y")
```

Figure 10.10 shows the points, as well as the centroids on the scatter plot.

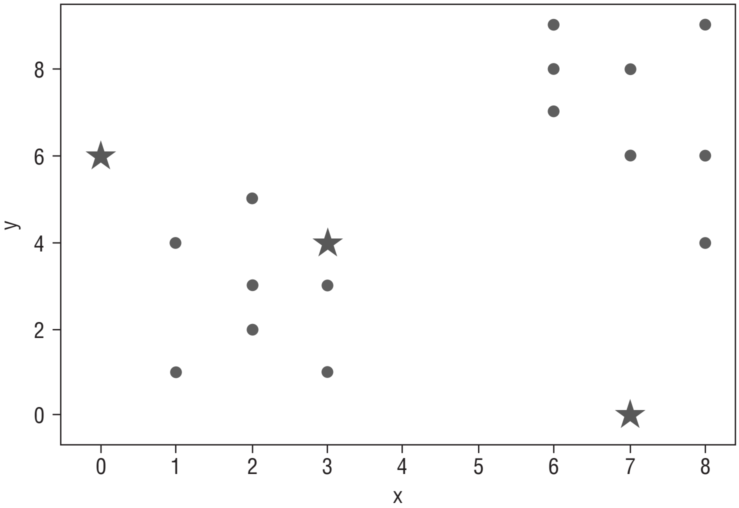


Figure 10.10: The scatter plot with the points and the three random centroids

Now comes the real meat of the program. The following code snippet implements the K-Means algorithm that we discussed earlier in the "How Clustering in K-Means Works" section:

```
from copy import deepcopy

#---to calculate the distance between two points---
def euclidean_distance(a, b, ax=1):
    return np.linalg.norm(a - b, axis=ax)

#---create a matrix of 0 with same dimension as C (centroids)---
C_prev = np.zeros(C.shape)

#---to store the cluster each point belongs to---
clusters = np.zeros(len(X))

#---C is the random centroids and C_prev is all 0s---
#---measure the distance between the centroids and C_prev---
distance_differences = euclidean_distance(C, C_prev)

#---loop as long as there is still a difference in
# distance between the previous and current centroids---
while distance_differences.any() != 0:
    #---assign each value to its closest cluster---
    for i in range(len(X)):
        distances = euclidean_distance(X[i], C)

        #---returns the indices of the minimum values along an axis---
        cluster = np.argmin(distances)
        clusters[i] = cluster

    #---store the prev centroids---
    C_prev = deepcopy(C)

    #---find the new centroids by taking the average value---
    for i in range(k):  #---k is the number of clusters---
        #---take all the points in cluster i---
        points = [X[j] for j in range(len(X)) if clusters[j] == i]
        if len(points) != 0:
            C[i] = np.mean(points, axis=0)

    #---find the distances between the old centroids and the new
centroids---
    distance_differences = euclidean_distance(C, C_prev)

#---plot the scatter plot---
colors = ['b','r','y','g','c','m']
for i in range(k):
    points = np.array([X[j] for j in range(len(X)) if clusters[j] == i])
    if len(points) > 0:
        plt.scatter(points[:, 0], points[:, 1], s=10, c=colors[i])
    else:
        # this means that one of the clusters has no points
        print("Plesae regenerate your centroids again.")

    plt.scatter(points[:, 0], points[:, 1], s=10, c=colors[i])
    plt.scatter(C[:, 0], C[:, 1], marker='*', s=100, c='black')
```

With the preceding code snippet, the centroids would now be computed and displayed on the scatter plot, as shown in Figure 10.11.

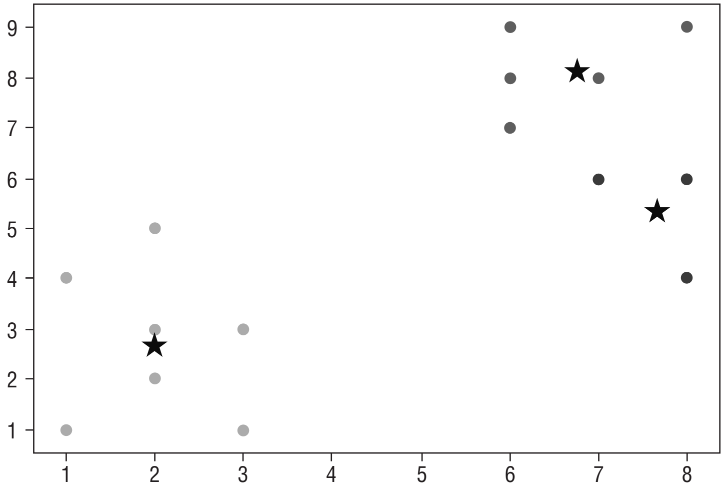


Figure 10.11: The scatter plot showing the clustering of the points as well as the new-found centroids

TIP **Due to the locations of the points, it is possible that the centroids you obtained may not be identical to the one shown in Figure 10.11.**

Also, there may be cases where after the clustering, there are no points belonging to a particular centroid. In this case, you have to regenerate the centroid and perform the clustering again.

You can now also print out the clusters to which each point belongs:

```
for i, cluster in enumerate(clusters):
    print("Point " + str(X[i]),
          "Cluster " + str(int(cluster)))
```

You should be able to see the following output:

```
Point [1 1] Cluster 2
Point [2 2] Cluster 2
Point [2 3] Cluster 2
Point [1 4] Cluster 2
Point [3 3] Cluster 2
Point [6 7] Cluster 1
Point [7 8] Cluster 1
Point [6 8] Cluster 1
Point [7 6] Cluster 0
Point [6 9] Cluster 1
Point [2 5] Cluster 2
Point [7 8] Cluster 1
Point [8 9] Cluster 1
Point [6 7] Cluster 1
Point [7 8] Cluster 1
Point [3 1] Cluster 2
Point [8 4] Cluster 0
Point [8 6] Cluster 0
Point [8 9] Cluster 1
```

TIP The cluster numbers that you will see may not be the same as the ones shown in the preceding code.

More importantly, you want to know the location of each centroid. You can do so via printing out the value of `C`:

```
print(C)
'''
[[ 7.66666667  5.33333333]
 [ 6.77777778  8.11111111]
 [ 2.          2.71428571]]
'''
```

Using K-Means in Scikit-learn

Rather than implementing your own K-Means algorithm, you can use the `KMeans` class in Scikit-learn to do clustering. Using the same dataset that you used in the previous section, the following code snippet creates an instance of the `KMeans` class with a cluster size of 3:

```
#---using sci-kit-learn---
from sklearn.cluster import KMeans
k=3
kmeans = KMeans(n_clusters=k)
```

You can now train the model using the `fit()` function:

```
kmeans = kmeans.fit(X)
```

To assign a label to all of the points, use the `predict()` function:

```
labels = kmeans.predict(X)
```

To get the centroids, use the `cluster_centers` property:

```
centroids = kmeans.cluster_centers_
```

Let's print the clusters label and centroids and see what you got:

```
print(labels)
print(centroids)
```

You should see the following:

```
[1 1 1 1 1 0 0 0 2 0 1 0 0 0 0 1 2 2 0]
[[ 6.77777778  8.11111111]
 [ 2.          2.71428571]
 [ 7.66666667  5.33333333]]
```

TIP **Due to the locations of the points, it is possible that the centroids you obtained may not be identical to the one shown here in the text.**

Let's now plot the points and centroids on a scatter plot:

```
#---map the labels to colors---
c = ['b','r','y','g','c','m']
colors = [c[i] for i in labels]

plt.scatter(df['x'],df['y'], c=colors, s=18)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=100, c='black')
```

Figure 10.12 shows the result.

Using the model that you have just trained, you can use it to predict the cluster to which a point will belong using the `predict()` function:

```
#---making predictions---
cluster = kmeans.predict([[3,4]])[0]
print(c[cluster])  # r

cluster = kmeans.predict([[7,5]])[0]
print(c[cluster])  # y
```

The preceding statements print the cluster in which a point is located using its color: `r` for red and `y` for yellow.

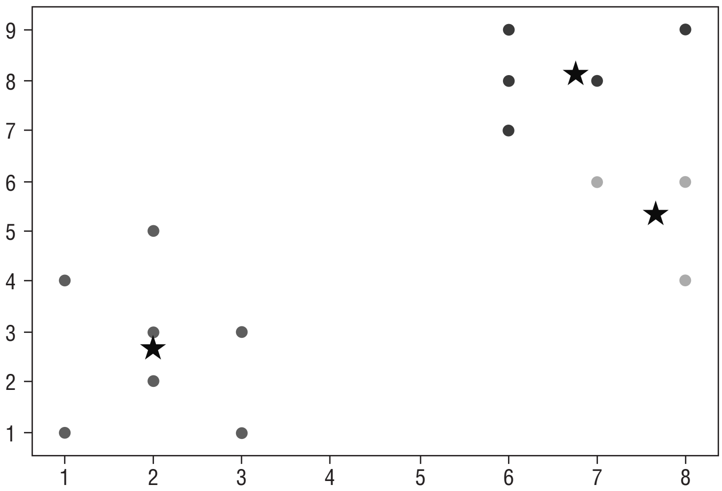


Figure 10.12: Using the KMeans class in Scikit-learn to do the clustering

TIP **You may get different colors for the predicted points, which is perfectly fine.**

Evaluating Cluster Size Using the Silhouette Coefficient

So far, we have been setting K to a fixed value of 3. How do you ensure that the value of K that you have set is the optimal number for the number of clusters? With a small dataset, it is easy to deduce the value of K by visual inspection; however, with a large dataset, it will be a more challenging task. Also, regardless of the dataset size, you will need a scientific way to prove that the value of K you have selected is the optimal one. To do that, you will use the Silhouette Coefficient.

The *Silhouette Coefficient* is a measure of the quality of clustering that you have achieved. It measures cluster cohesion, which is the space between clusters. The range of values for the Silhouette Coefficient is between –1 and 1.

The Silhouette Coefficient formula is given as:

$$1-(a / b)$$

where:

- `a` is the average distance of a point to all other points in the same cluster; if `a` is small, cluster cohesion is good, as all of the points are close together
- `b` is the *lowest* average distance of a point to all other points in the closest cluster; if `b` is large, cluster separation is good, as the nearest cluster is far apart

If `a` is small and `b` is large, the Silhouette Coefficient is high. The value of k that yields the highest Silhouette Coefficient is known as the *optimal K*.

Calculating the Silhouette Coefficient

Let's walk through an example of how to calculate the Silhouette Coefficient of a point. Consider the seven points and the clusters (`K=3`) to which they belong, as shown in Figure 10.13.

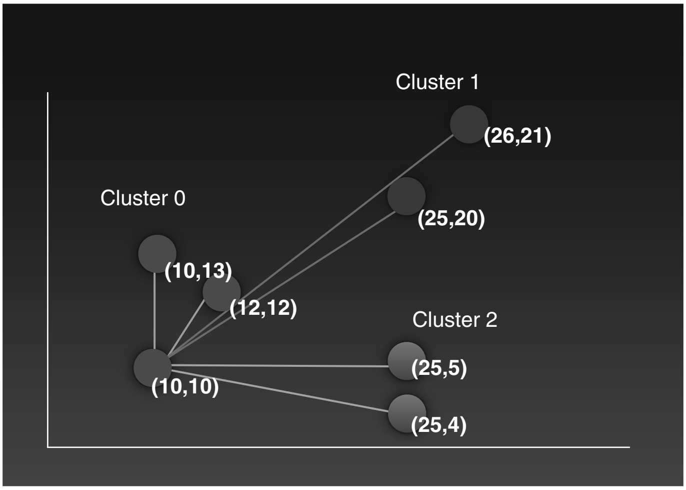


Figure 10.13: The set of points and their positions

Let's calculate the Silhouette Coefficient of a particular point and walk through the math. Consider the point (10,10) in cluster 0:

- Calculate its average distance to all other points in the same cluster:
  - (10,10) – (12,12) = √8 = 2.828
  - (10,10) – (10,13) = √9 = 3
  - Average: (2.828 + 3.0) / 2 = 2.914
- Calculate its average distance to all other points in cluster 1:
  - (10,10) – (25,20) = √325 = 18.028
  - (10,10) – (26,21) = √377 = 19.416
  - Average: (18.028 + 19.416) / 2 = 18.722
- Calculate its average distance to all other points in cluster 2:
  - (10,10) – (25,5) = √250 = 15.811
  - (10,10) – (25,4) = √261= 16.155
  - Average: (15.811 + 16.156) / 2 = 15.983
- Minimum average distance from (10,10) to all the points in cluster 1 and 2 is `min(18.722,15.983)` = 15.983

Therefore, the Silhouette Coefficient of point (10,10) is 1 – (a/b) = 1 – (2.914/15.983) = 0.817681—and this is just for one point in the dataset. You need to calculate the Silhouette Coefficients of the other six points in the dataset. Fortunately, Scikit-learn contains the `metrics` module that automates this process.

Using the `kmean.csv` example that you used earlier in this chapter, the following code snippet calculates the Silhouette Coefficient of all of the 19 points in the dataset and prints out the average of the Silhouette Coefficient:

```
from sklearn import metrics

silhouette_samples = metrics.silhouette_samples(X, kmeans.labels_)
print(silhouette_samples)

print("Average of Silhouette Coefficients for k =", k)
print("============================================")
print("Silhouette mean:", silhouette_samples.mean())
```

You should see the following results:

```
[ 0.67534567  0.73722797  0.73455072  0.66254937  0.6323039   0.33332111
  0.63792468  0.58821402  0.29141777  0.59137721  0.50802377  0.63792468
  0.52511161  0.33332111  0.63792468  0.60168807  0.51664787  0.42831295
  0.52511161]

Average of Silhouette Coefficients for k = 3
============================================
Silhouette mean: 0.55780519852
```

In the preceding statements, you used the `metrics.silhouette_samples()` function to get an array of Silhouette Coefficients for the 19 points. You then called the `mean()` function on the array to get the average Silhouette Coefficient. If you are just interested in the average Silhouette coefficient and not the Silhouette Coefficient for the individual points, you can simply call the `metrics.silhouette_score()` function, like this:

```
print("Silhouette mean:", metrics.silhouette_score(X, kmeans.labels_))
# Silhouette mean: 0.55780519852
```

Finding the Optimal K

Now that you have seen how to calculate the mean Silhouette Coefficient for a dataset with K clusters, what you want to do next is to find the optimal K that gives you the highest average Silhouette Coefficient. You can start with a cluster size of 2, up to the cluster size of one less than the size of the dataset. The following code snippet does just that:

```
silhouette_avgs = []
min_k = 2

#---try k from 2 to maximum number of labels---
for k in range(min_k, len(X)):
    kmean = KMeans(n_clusters=k).fit(X)
    score = metrics.silhouette_score(X, kmean.labels_)
    print("Silhouette Coefficients for k =", k, "is", score)
    silhouette_avgs.append(score)

f, ax = plt.subplots(figsize=(7, 5))
ax.plot(range(min_k, len(X)), silhouette_avgs)

plt.xlabel("Number of clusters")
plt.ylabel("Silhouette Coefficients")

#---the optimal k is the one with the highest average silhouette---
Optimal_K = silhouette_avgs.index(max(silhouette_avgs)) + min_k
print("Optimal K is ", Optimal_K)
```

The code snippet will print out something similar to the following:

```
Silhouette Coefficients for k = 2 is 0.689711206994
Silhouette Coefficients for k = 3 is 0.55780519852
Silhouette Coefficients for k = 4 is 0.443038181464
Silhouette Coefficients for k = 5 is 0.442424857695
Silhouette Coefficients for k = 6 is 0.408647742839
Silhouette Coefficients for k = 7 is 0.393618055172
Silhouette Coefficients for k = 8 is 0.459039364508
Silhouette Coefficients for k = 9 is 0.447750636074
Silhouette Coefficients for k = 10 is 0.512411340842
Silhouette Coefficients for k = 11 is 0.469556467119
Silhouette Coefficients for k = 12 is 0.440983139813
Silhouette Coefficients for k = 13 is 0.425567707244
Silhouette Coefficients for k = 14 is 0.383836485201
Silhouette Coefficients for k = 15 is 0.368421052632
Silhouette Coefficients for k = 16 is 0.368421052632
Silhouette Coefficients for k = 17 is 0.368421052632
Silhouette Coefficients for k = 18 is 0.368421052632
Optimal K is  2
```

As you can see from the output, the optimal K is 2. Figure 10.14 shows the chart of the Silhouette Coefficients plotted against the number of clusters (k).

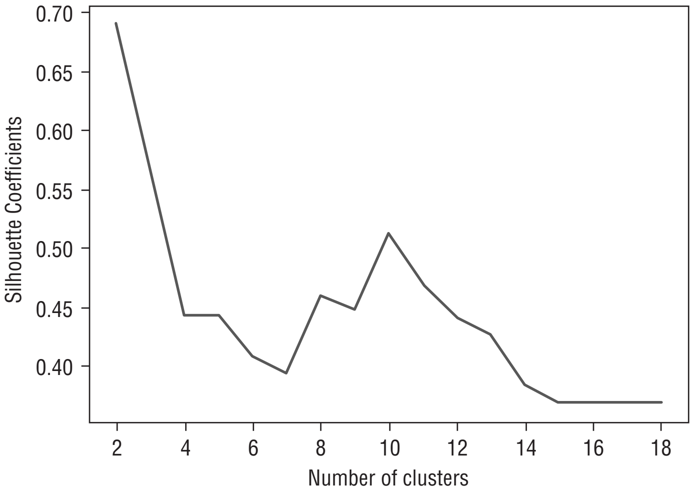


Figure 10.14: The chart showing the various values of K and their corresponding Silhouette Coefficients

Using K-Means to Solve Real-Life Problems

Suppose you are a clothing designer, and you have been tasked with designing a new series of Bermuda shorts. One of the design problems is that you need to come up with a series of sizes so that it can fit most people. Essentially, you need to have a series of sizes of people with different:

- Waist Circumference
- Upper Leg Length

So, how do you find the right combination of sizes? This is where the K-Means algorithm comes in handy. The first thing you need to do is to get ahold of a dataset containing the measurements of a group of people (of a certain age range). Using this dataset, you can apply the K-Means algorithm to group these people into clusters based on the specific measurement of their body parts. Once the clusters are found, you would now have a very clear picture of the sizes for which you need to design.

For the dataset, you can use the Body Measurement dataset from `https://data.world/rhoyt/body-measurements`. This dataset has 27 columns and 9338 rows. Among the 27 columns, two columns are what you need:

BMXWAIST: Waist Circumference (cm)

BMXLEG: Upper Leg Length (cm)

For this example, assume that the dataset has been saved locally with the filename `BMX_G.csv`.

Importing the Data

First, import the data into a Pandas dataframe:

```
%matplotlib inline
import numpy as np
import pandas as pd

df = pd.read_csv("BMX_G.csv")
```

Examine its shape, and you should see 9338 rows and 27 columns:

```
print(df.shape)
# (9338, 27)
```

Cleaning the Data

The dataset contains a number of missing values, so it is important to clean the data. To see how many empty fields each column contains, use the following statement:

```
df.isnull().sum()
```

You should see the following:

```
Unnamed: 0        0
seqn              0
bmdstats          0
bmxwt            95
bmiwt          8959
bmxrecum       8259
bmirecum       9307
bmxhead        9102
bmihead        9338
bmxht           723
bmiht          9070
bmxbmi          736
bmdbmic        5983
bmxleg         2383
bmileg         8984
bmxarml         512
bmiarml        8969
bmxarmc         512
bmiarmc        8965
bmxwaist       1134
bmiwaist       8882
bmxsad1        2543
bmxsad2        2543
bmxsad3        8940
```

```
bmxsad4        8940
bmdavsad       2543
bmdsadcm       8853
dtype: int64
```

Observe that the column `bmxleg` has 2383 missing values and `bmxwaist` has 1134 missing values, so you would need to remove them as follows:

```
df = df.dropna(subset=['bmxleg','bmxwaist'])  # remove rows with NaNs
print(df.shape)
# (6899, 27)
```

After removing the `bmxleg` and `bmxwaist` columns with missing values, there are now 6899 rows remaining.

Plotting the Scatter Plot

With the data cleaned, let's plot a scatter plot showing the distribution in upper leg length and waist circumference:

```
import matplotlib.pyplot as plt

plt.scatter(df['bmxleg'],df['bmxwaist'], c='r', s=2)
plt.xlabel("Upper leg Length (cm)")
plt.ylabel("Waist Circumference (cm)")
```

Figure 10.15 shows the scatter plot.

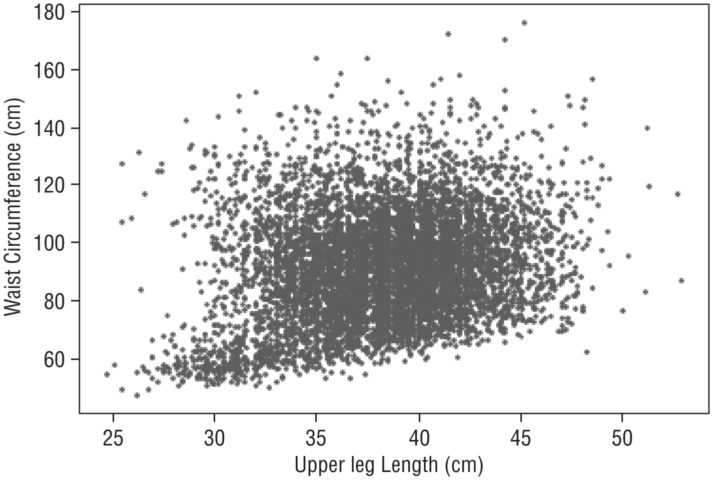


Figure 10.15: The scatter plot showing the distribution of waist circumference and upper leg length

Clustering Using K-Means

Assume that you want to create two sizes of Bermuda shorts. In this case, you would like to cluster the points into two clusters; that is, K=2. Again, we can use Scikit-learn's KMeans class for this purpose:

```
#---using sci-kit-learn---
from sklearn.cluster import KMeans

k = 2
X = np.array(list(zip(df['bmxleg'],df['bmxwaist'])))

kmeans = KMeans(n_clusters=k)
kmeans = kmeans.fit(X)
labels = kmeans.predict(X)
centroids = kmeans.cluster_centers_

#---map the labels to colors---
c = ['b','r','y','g','c','m']
colors = [c[i] for i in labels]

plt.scatter(df['bmxleg'],df['bmxwaist'], c=colors, s=2)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=100, c='black')
```

Figure 10.16 shows the points separated into two clusters, red and blue, together with the two centroids.

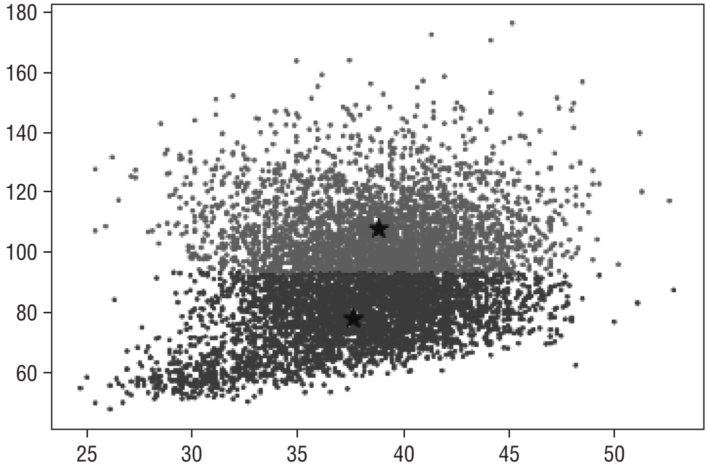


Figure 10.16: Clustering the points into two clusters

For you, the most important information is the value of the two centroids:

```
print(centroids)
```

You should get the following:

```
[[  37.65663043   77.84326087]
 [  38.81870146  107.9195713 ]]
```

This means that you can now design your Bermuda shorts with the following dimensions:

- Waist 77.8 cm, upper leg length 37.7 cm
- Waist 107.9 cm, upper leg length 38.8 cm

Finding the Optimal Size Classes

Before deciding on the actual different sizes to make, you wanted to see if the `K=2` is the optimal one, hence you try out different values of K from 2 to 10 and look for the optimal K:

```
from sklearn import metrics

silhouette_avgs = []
min_k = 2

#---try k from 2 to maximum number of labels---
for k in range(min_k, 10):
    kmean = KMeans(n_clusters=k).fit(X)
    score = metrics.silhouette_score(X, kmean.labels_)
    print("Silhouette Coefficients for k =", k, "is", score)
    silhouette_avgs.append(score)

#---the optimal k is the one with the highest average silhouette---
Optimal_K = silhouette_avgs.index(max(silhouette_avgs)) + min_k
print("Optimal K is", Optimal_K)
```

The results are as shown here:

```
Silhouette Coefficients for k = 2 is 0.516551581494
Silhouette Coefficients for k = 3 is 0.472269050688
Silhouette Coefficients for k = 4 is 0.436102446644
Silhouette Coefficients for k = 5 is 0.418064636123
Silhouette Coefficients for k = 6 is 0.392927895139
Silhouette Coefficients for k = 7 is 0.378340717032
Silhouette Coefficients for k = 8 is 0.360716292593
Silhouette Coefficients for k = 9 is 0.341592231958
Optimal K is 2
```

The result confirms that the optimal K is 2. That is, you should have two different sizes for the Bermuda shorts that you are designing.

However, the company wanted you to have more sizes so that it can accommodate a wider range of customers. In particular, the company feels that four sizes would be a better decision. To do so, you just need to run the `KMeans` code snippet that you saw in the "Clustering Using K-Means" section and set `k =4`.

You should now see the clusters as shown in Figure 10.17.

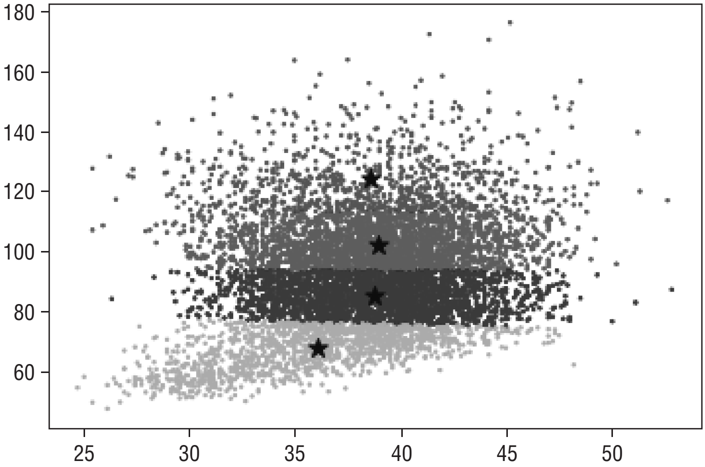


Figure 10.17: Clustering the points into four clusters

The centroids locations are as follows:

```
[[  38.73004292   85.05450644]
 [  38.8849217   102.17011186]
 [  36.04064872   67.30131125]
 [  38.60124294  124.07853107]]
```

This means that you can now design your Bermuda shorts with the following dimensions:

- Waist 67.3 cm, upper leg length 36.0 cm
- Waist 85.1 cm, upper leg length 38.7 cm
- Waist 102.2 cm, upper leg length 38.9 cm
- Waist 124.1 cm, upper leg length 38.6 cm

Summary

In this chapter, you learned about unsupervised learning. Unsupervised learning is a type of machine learning technique that allows you to find patterns in data. In unsupervised learning, the data that is used by the algorithm (for example, K-Means, as discussed in this chapter) is not labeled, and your role is to discover its hidden structures and assign labels to them.