*7.7 DATA CLUSTERING

In some large data sets, we are interested in discovering groups of items that are similar in some way. For example, suppose we are trying to determine whether a suite of test results is indicative of a malignant tumor. If we have these test results for a large number of patients, together with information about whether each patient has been diagnosed with a malignant tumor, then we can test whether clusters of similar test results correspond uniformly to the same diagnosis. If they do, the clustering is evidence that the tests can be used to test for malignancy, and the clusters give insights into the range of results that correspond to that diagnosis. Because data clustering can result in deep insights like this, it is another fundamental technique used in ***data mining***.

Algorithms that cluster items according to their similarity are easiest to understand initially with two-dimensional data (e.g., longitude and latitude) that can be represented visually. Therefore, before you tackle the tumor diagnosis problem (as an exercise), we will look at a data set containing the geographic locations of vehicular collisions in New York City.[5]

The clustering problem turns out to be very difficult, and there are no known efficient algorithms that solve it exactly. Instead, people use *heuristics*. A heuristic is an algorithm that does not necessarily give the best answer, but tends to work well in practice. Colloquially, you might think of a heuristic as a "good rule of thumb." We will discuss a common clustering heuristic known as ***k-means clustering***. In k-means clustering, the data is partitioned into k clusters, and each cluster has an associated *centroid*, defined to be the mean of the points in the cluster. The heuristic consists of a number of iterations in which each point is (re)assigned to the cluster of the closest, or "most similar," centroid, and then the centroids of each cluster are recomputed based on their potentially new membership. These steps are described in pseudocode below.

Algorithm k-MEANS CLUSTERING

```
Input: a list of data, the number of clusters k, and a number of iterations
1   centroids ← k random items from data
2   repeat iterations times:
3       create k empty clusters
4       repeat for each item in data:
5           assign item to the cluster with the closest centroid
6       recompute the centroid for each cluster
Output: lists of clusters and centroids
```

Defining similarity

Similarity or "closeness" among items in a data set can be defined in a variety of ways; here we will define similarity simply in terms of normal Euclidean distance. If each item in our data set has m numerical attributes, then we can think of these attributes as a point in m-dimensional space. Given two m-dimensional points p and q, we can find the distance

[5] `http://nypd.openscrape.com`

between them using the formula

$$\text{distance}(p,q) = \sqrt{(p[0]-q[0])^2 + (p[1]-q[1])^2 + \cdots + (p[m-1]-q[m-1])^2}.$$

In Python, we can represent each point as a tuple of m numbers. For example, each of the following three tuples might represent six test results ($m = 6$) for one patient:

```
patient1 = (85, 92, 45, 27, 31, 0.0)
patient2 = (85, 64, 59, 32, 23, 0.0)
patient3 = (86, 54, 33, 16, 54, 0.0)
```

To find the distance between `patient1` and `patient2`, we compute

$$\sqrt{(85-85)^2 + (92-64)^2 + (45-59)^2 + (27-32)^2 + (31-23)^2 + (0-0)^2} \approx 32.7$$

and to find the distance between `patient1` and `patient3`, we compute

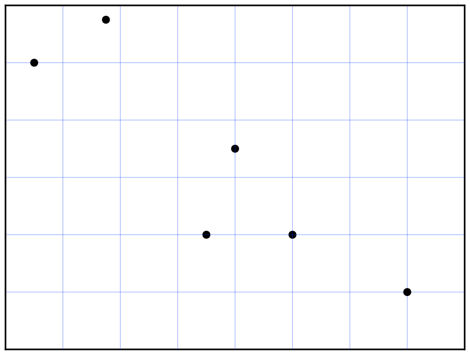
$$\sqrt{(85-86)^2 + (92-54)^2 + (45-33)^2 + (27-16)^2 + (31-54)^2 + (0-0)^2} \approx 47.3.$$

Because the distance between `patient1` and `patient2` is less than the distance between `patient1` and `patient3`, we consider the results for `patient1` to be more similar to the results for `patient2` than to the results for `patient3`.

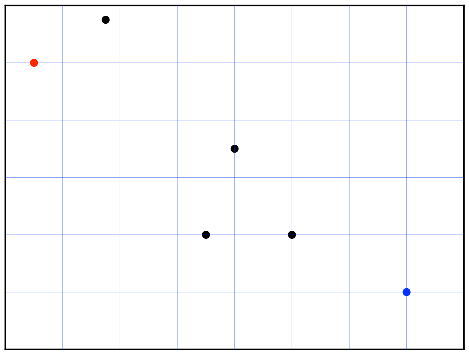
Reflection 1 *What is the distance between* `patient2` *and* `patient3`*? Are the results of* `patient1` *or* `patient3` *more similar to the results of* `patient2`*? Are the results of* `patient1` *or* `patient2` *more similar to the results of* `patient3`*?*

A simple example

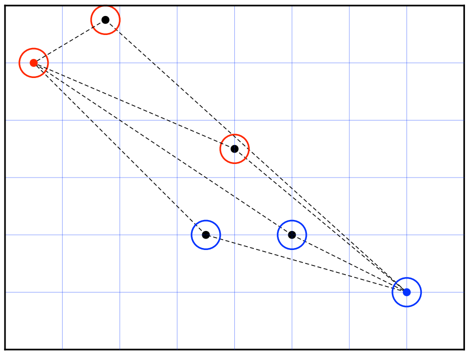
To illustrate in more detail how the heuristic works, suppose we want to group the small set of six points plotted below into $k = 2$ clusters.



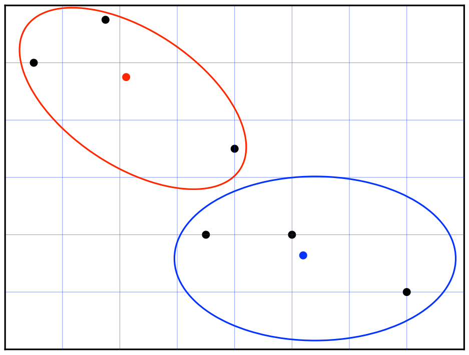
The lower left corner of the rectangle has coordinates (0,0) and each square is one unit on a side. So our six points have coordinates $(0.5, 5)$, $(1.75, 5.75)$, $(3.5, 2)$, $(4, 3.5)$, $(5, 2)$, and $(7, 1)$. We begin by randomly choosing $k = 2$ of our points to be the initial centroids. In the figure below, we chose one centroid to be the red point at (0.5,5) and the other centroid to be the blue point at (7,1).



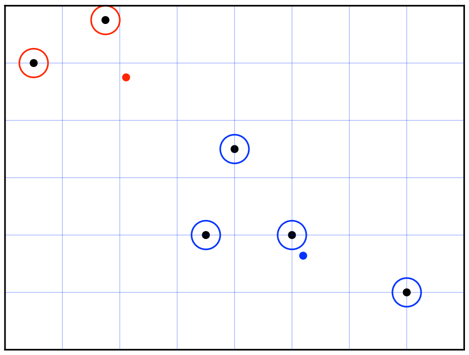
In the first iteration of the heuristic, we compute the distance between every point and each of the two centroids, represented by the dashed line segments below. We assign each point to a cluster associated with the closest centroid. The three points circled in red below are closest to the red centroid and the three points circled in blue are closest to the blue centroid.



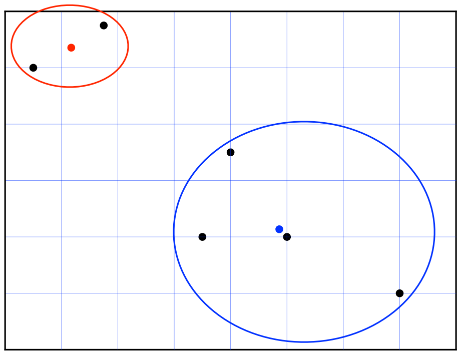
Once every point is assigned to a cluster, we compute new centroids for the clusters, defined to be the mean of all the points in the cluster. The x and y coordinates of the new centroid of the red cluster are $(0.5 + 1.75 + 4)/3 = 25/12 \approx 2.08$ and $(5 + 5.75 + 3.5)/3 = 4.75$. The x and y coordinates of the new centroid of the blue cluster are $(3.5 + 5 + 7)/3 = 31/6 \approx 5.17$ and $(2 + 2 + 1)/3 = 5/3 \approx 1.67$. These are the new red and blue points shown below.



In the second iteration of the heuristic, we once again compute the closest centroid for each point. After this iteration, as illustrated below, the point $(4, 3.5)$ is now grouped with the lower right points because it is closer to the blue centroid than to the red centroid (distance$((2.08, 4.75), (4, 3.5)) \approx 2.29$ and distance$((5.17, 1.67), (4, 3.5)) \approx 2.17$).



Next, we once again compute new centroids. The x and y coordinates of the new centroid of the red cluster are $(0.5+1.75)/2 = 1.125$ and $(5+5.75)/2 = 5.375$. The x and y coordinates of the new centroid of the blue cluster are $(3.5+4+5+7)/4 = 4.875$ and $(1+2+2+3.5)/4 = 2.125$. These new centroids are shown below.



In the third iteration, when we find the closest centroid for each point, we find that nothing changes. Therefore, these clusters are the final ones chosen by the heuristic.

Implementing k-means clustering

When we implement this heuristic in Python, the input list `data` will contain tuples like the points in the previous example:

```
data = [(0.5, 5), (1.75, 5.75), (3.5, 2), (4, 3.5), (5, 2), (7, 1)]
```

We will need k additional lists to hold the items in the k clusters, as well as a list of k centroids. Copying actual data items to these lists in each iteration is inefficient if the items contain many attributes, and ambiguous if multiple items have the same attributes. So we will represent items by their indices in `data` instead. This approach is described in the following more fleshed-out pseudocode.

Algorithm k-MEANS CLUSTERING — MORE DETAIL

```
Input: a list of data, the number of clusters k, and a number of iterations
1  centroids ← a list of k random items from data
2  repeat iterations times:
3      repeat for each index from 0 to k − 1:
4          clusters[index] ← an empty list of indices (of items in data)
5      repeat for each item in data:
6          minimum index ← the index of the centroid closest to item
7          append the index of item to clusters[minimum index]
8      repeat for each index from 0 to k − 1:
9          centroids[index] ← mean of the items in clusters[index]
Output: lists of clusters and centroids
```

To get the initial list of centroids in line 1, we will choose `k` random tuples from `data` by using the `sample` function from the `random` module.

```
centroids = random.sample(data, k)
```

The remainder of the function will consist of a `for` loop that makes `iterations` passes over the list of tuples in `data`. Within the loop, we first create a list of `k` empty lists, one for each cluster (corresponding to lines 3–4 above).

```
clusters = []
for clustIndex in range(k):
    clusters.append([])
```

For example, if `k = 4` then, after this loop, `clusters` will be `[[ ], [ ], [ ], [ ]]`. The first empty list, `clusters[0]`, will be used to hold the indices of the points in the cluster associated with `centroids[0]`; the second empty list, `clusters[1]`, will be used to hold the indices of the points in the cluster associated with `centroids[1]`; and so on. Some care has to be taken to create this list of lists; `clusters = [[]] * k` does not work. Exercise 7.7.6 asks you to investigate this further.

Next, to implement lines 5–7, we iterate over the indices of the points in `data`.

```
for dataIndex in range(len(data)):
    minIndex = 0
    for clustIndex in range(1, k):
        dist = distance(centroids[clustIndex], data[dataIndex])
        if dist < distance(centroids[minIndex], data[dataIndex]):
            minIndex = clustIndex
    clusters[minIndex].append(dataIndex)
```

In each iteration, we find the centroid that is closest to the point `data[dataIndex]`. The inner `for` loop is essentially the same algorithm that we used in Section 7.1 to find the minimum value in a list. The difference here is that we need to find the *index* of the centroid with the minimum distance to `data[dataIndex]`. (We leave writing this `distance` function as Exercise 7.7.1.) We maintain the index of the closest centroid in the variable named `minIndex`. Once we have found the final value of `minIndex`, we append `dataIndex` to the list of indices assigned to the cluster with index `minIndex`.

After we have assigned all of the points to a cluster, to implement lines 8–9, we compute each new centroid with a function named `centroid`. We leave writing this function as Exercise 7.7.2.

```
for clustIndex in range(k):
    centroids[clustIndex] = centroid(clusters[clustIndex], data)
```

The complete function is reproduced below:

```
def kmeans(data, k, iterations):
    """Cluster data into k clusters with the k-means clustering algorithm.

    Parameters:
        data:       a list of points
        k:          the number of desired clusters
        iterations: the number of iterations of the algorithm

    Return value:
        a tuple containing the list of clusters and the list
        of centroids; each cluster is represented by a list
        of indices of the points assigned to that cluster
    """

    centroids = random.sample(data, k)

    for step in range(iterations):
        clusters = []
        for clustIndex in range(k):
            clusters.append([])

        for dataIndex in range(len(data)):
            minIndex = 0
            for clustIndex in range(1, k):
                dist = distance(centroids[clustIndex], data[dataIndex])
                if dist < distance(centroids[minIndex], data[dataIndex]):
                    minIndex = clustIndex
            clusters[minIndex].append(dataIndex)

        for clustIndex in range(k):
            centroids[clustIndex] = centroid(clusters[clustIndex], data)

    return (clusters, centroids)
```

Next we apply this function to a real data set.

Locating bicycle safety programs

Suppose that, in response to an increase in the number of vehicular accidents in New York City involving cyclists, we want to find the best locations to establish a limited number of bicycle safety programs. To do the most good, we would like to centrally locate these programs in areas containing the most cycling injuries. In other words, we want to locate safety programs at the centroids of clusters of accidents.

We can use the k-means clustering heuristic to find these centroids. The data file containing the accident information[6], which is available on the book website, is tab-separated and contains a row for each of 7,642 collisions involving bicycles between August, 2011 and August, 2013 in all five boroughs of New York City. Associated with each collision are 68 attributes that include information about the location, injuries, and whether pedestrians or cyclists were involved. The first few rows of the data file look like this (with triangles representing tab characters):

```
borocode▷precinct▷year▷month▷lon▷lat▷street1▷street2▷collisions▷...
3▷60▷2011▷8▷-73.987424▷40.58539▷CROPSEY AVENUE▷SHORE PARKWAY▷4▷...
3▷60▷2011▷8▷-73.98584▷40.576503▷MERMAID AVENUE▷WEST 19 STREET▷2▷...
3▷60▷2011▷8▷-73.983088▷40.579161▷NEPTUNE AVENUE▷WEST 15 STREET▷2▷...
3▷60▷2011▷8▷-73.986609▷40.574894▷SURF AVENUE▷WEST 20 STREET▷1▷...
3▷61▷2011▷8▷-73.955477▷40.598939▷AVENUE U▷EAST 16 STREET▷2▷...
```

To cluster the data, we will only need a list of accident locations as (longitude, latitude) tuples. To limit the data in our analysis, we will extract the locations of only those accidents that occurred in Manhattan (borocode 1) during 2012. The following function reads this data.

```
def readAccidentData(fileName):
    """Read locations of bicycle accidents in Manhattan.

    Parameter:
        fileName: the name of the data file

    Return value: a list of (longitude, latitude) tuples
    """

    inputFile = open(fileName, 'r', encoding = 'utf-8')
    header = inputFile.readline()
    data = [ ]
    for line in inputFile:
        row = line.split('\t')
        if (int(row[0]) == 1) and (int(row[2]) == 2012): # borough and yr
            data.append((float(row[4]), float(row[5])))  # long, latitude
    inputFile.close()
    return data
```

Once we have the data as a list of tuples, we can call the `kmeans` function to find the clusters and centroids. If we had enough funding for six bicycle safety programs, we would call the function with $k = 6$ as follows:

```
data = readAccidentData('collisions_cyclists.txt')
(clusters, centroids) = kmeans(data, 6, 100)
```

Figure 1, created by the following `plotClusters` function, visualizes the clusters in different colors with centroids represented by stars.

[6]The data file contains the rows of the file at `http://nypd.openscrape.com/collisions.csv.gz` (accessed October 1, 2013) that involved a bicyclist.

```
import matplotlib.pyplot as pyplot

def plotClusters(clusters, data, centroids):
    """Plot clusters and centroids in unique colors.

    Parameters:
        clusters:  a list of k lists of data indices
        data:      a list of points
        centroids: a list of k centroid points

    Return value: None
    """

    colors = ['blue', 'red', 'yellow', 'green', 'purple', 'orange']
    for clustIndex in range(len(clusters)):   # plot clusters of points
        x = []
        y = []
        for dataIndex in clusters[clustIndex]:
            x.append(data[dataIndex][0])
            y.append(data[dataIndex][1])
        pyplot.scatter(x, y, 10, color = colors[clustIndex])

    x = []
    y = []
    for centroid in centroids:    # plot centroids
        x.append(centroid[0])
        y.append(centroid[1])
    pyplot.scatter(x, y, 200, marker = '*', color = 'black')
    pyplot.show()
```

Exercise 7.7.3, asks you to reproduce the plot in Figure 1 by combining the functions in this section. Exercise 7.7.4 asks you to apply k-means clustering to the tumor diagnosis problem that we discussed at the beginning of the chapter.

Exercises

7.7.1* Write a function

`distance(p, q)`

that returns the Euclidean distance between k-dimensional points `p` and `q`, each of which is represented by a tuple with length k.

7.7.2* Write the function

`centroid(cluster, data)`

that is needed by the `kmeans` function. The function should compute the centroid of the given `cluster`, which is a list of indices of points in `data`. Remember that the points in data can be of any length, and the centroid must be a tuple of the same length. It will probably be most convenient to build up the centroid in a list and then convert it to a tuple using the `tuple` function. If the cluster is empty, then return a randomly chosen point from `data` with `random.choice(data)`.

7.7.3* Combine the `distance` and `centroid` functions that you wrote in the previous

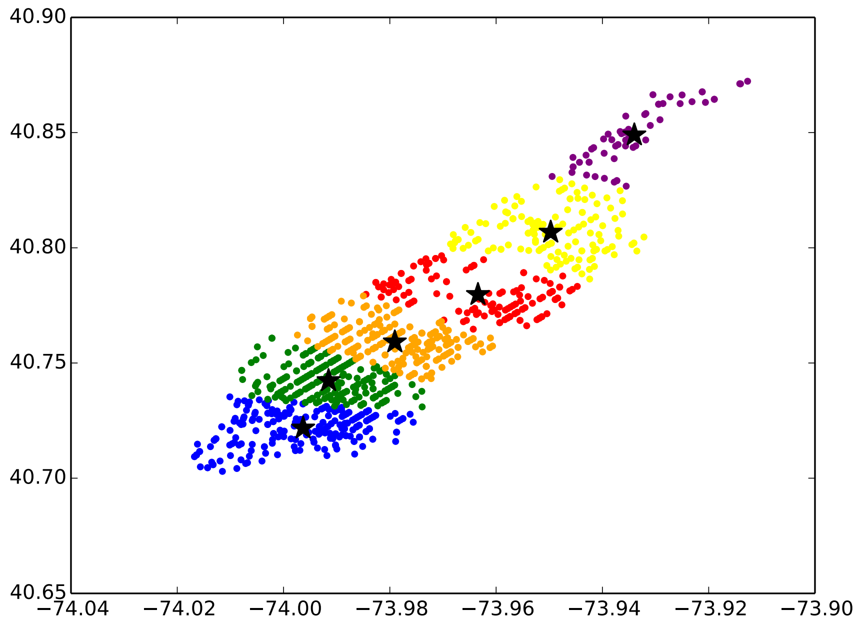


Figure 1 Six clusters of bicycle accidents in Manhattan. Stars represent centroids.

two exercises with the `kmeans`, `readAccidentData`, and `plotClusters` functions to create a complete program that produces the plot in Figure 1. The data file, `collisions_cyclists.txt`, is available on the book website.

7.7.4. In this exercise, you will apply k-means clustering to the tumor diagnosis problem mentioned at the beginning of this section, using a data set of breast cancer biopsy results. This data set, available on the book website, contains nine test results for each of 683 individuals.[7] The first few lines of the comma-separated data set look like this:

```
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
1015425,3,1,1,1,2,2,3,1,1,2
1016277,6,8,8,1,3,4,3,7,1,2
1017023,4,1,1,3,2,1,3,1,1,2
1017122,8,10,10,8,7,10,9,7,1,4
    ⋮
```

The first number in each row is a unique sample number and the last number in each row is either 2 or 4, where 2 means "benign" and 4 means "malignant." Of the 683 samples in the data set, 444 were diagnosed as benign and 239 were diagnosed as malignant. The nine numbers on each line between the sample number and the diagnosis are test result values that have been normalized to

[7]The data set on the book's website is the file from `http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)` with 16 rows containing missing attributes removed.

a 1–10 scale. (There is additional information accompanying the data on the book's website.)

The first step is to write a function

```
readFile(filename)
```

that reads this data and returns two lists, one of test results and the other of diagnoses. The test results will be in a list of 9-element tuples and the diagnosis values will be a list of integers that are in the same order as the test results.

Next, write a program, with the following `main` function, that clusters the data into $k = 2$ groups.

```
def main():
    data, diagnosis = readFile('breast-cancer-wisconsin.csv')
    clusters, centroids = kmeans(data, 2, 10)
    for clustIndex in range(2):
        count = {2: 0, 4: 0}
        for index in clusters[clustIndex]:
            count[diagnosis[index]] = count[diagnosis[index]] + 1
        print('Cluster', clustIndex)
        print('  benign:', count[2], 'malignant:', count[4])
```

If one of these two groups contains predominantly benign samples and the other contains predominantly malignant samples, then this provides evidence that these test results can discriminate between malignant and benign tumors. To complete the program, you will need to incorporate the `distance` and `centroid` functions from Exercises 7.7.1 and 7.7.2, the `kmeans` function, and your `readFile` function. The data file, `breast-cancer-wisconsin.csv`, is available on the book website. Describe the results.

7.7.5. Suppose you are manufacturing t-shirts and need to define three sizes (S, M, and L) based on torso length and chest circumference. You have these measurements from 100 customers (found on the book website). Use k-means clustering to group the customers into three clusters, and use these clusters to define the three t-shirt sizes.

7.7.6. In the `kmeans` function, we needed to create a new list of empty clusters at the beginning of each iteration. To create this list, we did the following:

```
clusters = []
for clustIndex in range(k):
    clusters.append([])
```

Alternatively, we could have created a list of lists using the `*` repetition operator like this:

```
clusters = [[]] * k
```

However, this will not work correctly, as evidenced by the following:

```
>>> clusters = [[]] * 5
>>> clusters
[[], [], [], [], []]
>>> clusters[2].append(4)
>>> clusters
[[4], [4], [4], [4], [4]]
```

What is going on here?

7.7.7. Modify the `kmeans` function so that it does not need to take the number of iterations as a parameter. Instead, the function should iterate until the list of clusters stops changing. In other words, the loop should iterate while the list of clusters generated in an iteration is not the same as the list of clusters from the previous iteration.

Selected Exercise Solutions

7.7.1
```
import math

def distance(p, q):
    m = len(p)
    sumSquares = 0
    for i in range(m):
        sumSquares = sumSquares + (p[i] - q[i]) ** 2
    return math.sqrt(sumSquares)
```

7.7.2
```
import random

def centroid(cluster, data):
    n = len(cluster)
    if n == 0:
        return random.choice(data)
    m = len(data[0])   # dimension of points
    theCentroid = []
    for index in range(m):
        sum = 0
        for dataIndex in cluster:
            point = data[dataIndex]
            sum = sum + point[index]
        theCentroid.append(sum / n)
    return tuple(theCentroid)
```

7.7.3
```
def main():
    data = readAccidentData('collisions_cyclists.txt')
    clusters, centroids = kmeans(data, 6, 100)
    plotClusters(clusters, data, centroids)

main()
```