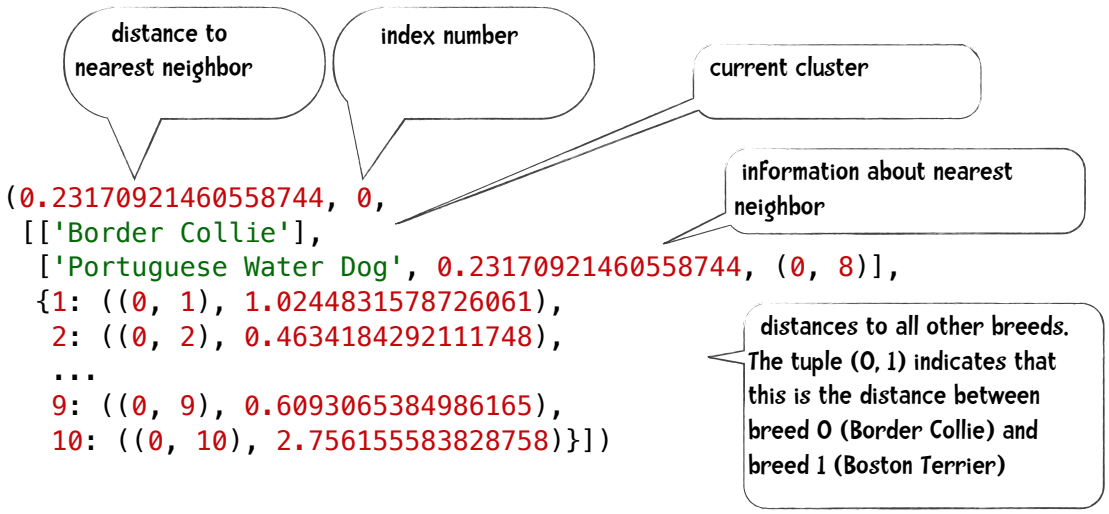
In our case of hierarchical clustering, We use the distance between breeds as the primary priority. To resolve ties we will use an index number. The first element we put on the queue will have an index of 0, the second element an index of 1, the third , 2, and so on. Our complete entry we add to the queue will be of the form:



We initialize the priority queue by placing on the queue, an entry like this for each breed.

Repeat the following until there is only one cluster.

We get two items from the queue, merge them into one cluster and put that entry on the queue. In our dog breed example, we get the entry for Border Collie and the entry for Portuguese Water Dog. We create the queue

```
['Border Collie', 'Portuguese Water Dog']
```

Next we compute the distance of this new cluster to all the other dog breeds except those in the new cluster. We do this by merging the distance dictionaries of the two initial clusters in the following way. Let's call the distance dictionary of the first item we get from the queue distanceDict1, the distance dictionary of the second item we get from the queue distanceDict2, and the distance dictionary we are constructing for the new cluster newDistanceDict.

```
Initialize newDistanceDict to an empty dictionary
for each key, value pair in distanceDict1:
  if there is an entry in distanceDict2 with that key:
     if the distance for that entry in distanceDict1 is
        shorter than that in distanceDict2:
           place the distanceDict1 entry in newDistanceDict
       else:
           place the distanceDict1 entry in newDistanceDict
```

key	value in the Border Collie Distance List	value in the Portuguese Water Dog Distance List	value in the Distance List for the new cluster
0	–	((0, 8), 0.2317092146055)	–
1	((0, 1), 1.02448315787260)	((1, 8), 1.25503395239308)	((0, 1), 1.02448315787260)
2	((0, 2), 0.46341842921117)	((2, 8), 0.69512764381676)	(0, 2), 0.46341842921117)
3	((0, 3), 2.52128307411504)	((3, 8), 2.3065500082408)	((3, 8), 2.3065500082408)
4	((0, 4), 2.41700998092941)	((4, 8), 2.643745991701)	((0, 4), 2.41700998092941)
5	((0, 5), 1.31725590972761)	((5, 8), 1.088215707936)	((5, 8), 1.088215707936)
6	((0, 6), 0.90660838225252)	((6, 8), 0.684696194462)	((6, 8), 0.684696194462)
7	((0, 7), 3.98523295438990)	((7, 8), 3.765829069545)	((7, 8), 3.765829069545)
8	((0, 8), 0.23170921460558)	–	–
9	((0, 9), 0.60930653849861)	((8, 9), 0.566225873458)	((8, 9), 0.566225873458)
10	((0, 10), 2.7561555838287)	((8, 10), 2.980333906137)	((0, 10), 2.7561555838287)

The complete entry that will be placed on the queue as a result of merging the Border Collie and the Portuguese Water Dog will be

```
(0.4634184292111748, 11, [('Border Collie', 'Portuguese Water Dog'),
 [2, 0.4634184292111748, (0, 2)],
 {1: ((0, 1), 1.0244831578726061), 2: ((0, 2), 0.4634184292111748),
  3: ((3, 8), 2.306550008240866), 4: ((0, 4), 2.4170099809294157),
  5: ((5, 8), 1.0882157079364436), 6: ((6, 8), 0.6846961944627522),
  7: ((7, 8), 3.7658290695451373), 9: ((8, 9), 0.5662258734585477),
  10: ((0, 10), 2.756155583828758)}])
```

Code It

Can you implement the algorithm presented above in Python?
To help you in this task, there is a Python file on the book's website, hierarchicalClustererTemplate.py (http://guidetodatamining.com/guide/pg2dm-python/ch8/hierarchicalClustererTemplate.py) that gives you a starting point. You need to:

1. Finish the init method.
   For each entry in the data:
   1. compute the Euclidean Distance from that entry to all other entries and create a Python Dictionary as described above.
   2. Find the nearest neighbor
   3. Place the info for this entry on the queue.

2. Write a cluster method. This method should repeatedly:
   1. retrieve the top 2 entries on the queue
   2. merge them
   3. place the new cluster on the queue

   until there is only one cluster on the queue.

Code It - solution

> Remember:
> This is only my solution and not necessarily the best solution. You might have come up with a better one!

```
from queue import PriorityQueue
import math


"""
Example code for hierarchical clustering
"""

def getMedian(alist):
    """get median value of list alist"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2


def normalizeColumn(column):
    """Normalize column using Modified Standard Score"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result

class hClusterer:
    """ this clusterer assumes that the first column of the data is a label
    not used in the clustering. The other columns contain numeric data"""

    def __init__(self, filename):
        file = open(filename)
        self.data = {}
        self.counter = 0
        self.queue = PriorityQueue()
        lines = file.readlines()
```

```
file.close()
header = lines[0].split(',')
self.cols = len(header)
self.data = [[] for i in range(len(header))]
for line in lines[1:]:
    cells = line.split(',')
    toggle = 0
    for cell in range(self.cols):
        if toggle == 0:
           self.data[cell].append(cells[cell])
           toggle = 1
        else:
            self.data[cell].append(float(cells[cell]))
# now normalize number columns (that is, skip the first column)
for i in range(1, self.cols):
        self.data[i] = normalizeColumn(self.data[i])

###
###  I have read in the data and normalized the
###  columns. Now for each element i in the data, I am going to
###     1. compute the Euclidean Distance from element i to all the
###        other elements.  This data will be placed in neighbors,
###        which is a Python dictionary. Let's say i = 1, and I am
###        computing the distance to the neighbor j and let's say j
###        is 2. The neighbors dictionary for i will look like
###        {2: ((1,2), 1.23),  3: ((1, 3), 2.3)... }
###
###     2. find the closest neighbor
###
###     3. place the element on a priority queue, called simply queue,
###        based on the distance to the nearest neighbor (and a counter
###        used to break ties.



# now push distances on queue
rows = len(self.data[0])

for i in range(rows):
    minDistance = 99999
    nearestNeighbor = 0
    neighbors = {}
    for j in range(rows):
        if i != j:
            dist = self.distance(i, j)
            if i < j:
                pair = (i,j)
            else:
                pair = (j,i)
            neighbors[j] = (pair, dist)
```

```
                if dist < minDistance:
                    minDistance = dist
                    nearestNeighbor = j
                    nearestNum = j
            # create nearest Pair
            if i < nearestNeighbor:
                nearestPair = (i, nearestNeighbor)
            else:
                nearestPair = (nearestNeighbor, i)

            # put instance on priority queue
            self.queue.put((minDistance, self.counter,
                            [[self.data[0][i]], nearestPair, neighbors]))
            self.counter += 1


    def distance(self, i, j):
        sumSquares = 0
        for k in range(1, self.cols):
            sumSquares += (self.data[k][i] - self.data[k][j])**2
        return math.sqrt(sumSquares)


    def cluster(self):
         done = False
         while not done:
             topOne = self.queue.get()
             nearestPair = topOne[2][1]
             if not self.queue.empty():
                 nextOne = self.queue.get()
                 nearPair = nextOne[2][1]
                 tmp = []
                 ##
                 ##  I have just popped two elements off the queue,
                 ##  topOne and nextOne. I need to check whether nextOne
                 ##  is topOne's nearest neighbor and vice versa.
                 ##  If not, I will pop another element off the queue
                 ##  until I find topOne's nearest neighbor. That is what
                 ##  this while loop does.
                 ##

                 while nearPair != nearestPair:
                     tmp.append((nextOne[0], self.counter, nextOne[2]))
                     self.counter += 1
                     nextOne = self.queue.get()
                     nearPair = nextOne[2][1]
                 ##
                 ## this for loop pushes the elements I popped off in the
                 ## above while loop.
                 ##
```

```
for item in tmp:
    self.queue.put(item)

if len(topOne[2][0]) == 1:
   item1 = topOne[2][0][0]
else:
    item1 = topOne[2][0]
if len(nextOne[2][0]) == 1:
   item2 = nextOne[2][0][0]
else:
    item2 = nextOne[2][0]
##  curCluster is, perhaps obviously, the new cluster
##  which combines cluster item1 with cluster item2.
curCluster = (item1, item2)

## Now I am doing two things. First, finding the nearest
## neighbor to this new cluster. Second, building a new
## neighbors list by merging the neighbors lists of item1
## and item2. If the distance between item1 and element 23
## is 2 and the distance betweeen item2 and element 23 is 4
## the distance between element 23 and the new cluster will
## be 2 (i.e., the shortest distance).
##

minDistance = 99999
nearestPair = ()
nearestNeighbor = ''
merged = {}
nNeighbors = nextOne[2][2]
for (key, value) in topOne[2][2].items():
   if key in nNeighbors:
       if nNeighbors[key][1] < value[1]:
            dist =  nNeighbors[key]
       else:
           dist = value
       if dist[1] < minDistance:
            minDistance =  dist[1]
            nearestPair = dist[0]
            nearestNeighbor = key
       merged[key] = dist

if merged == {}:
   return curCluster
else:
   self.queue.put( (minDistance, self.counter,
                    [curCluster, nearestPair, merged]))
   self.counter += 1
```

```
def printDendrogram(T, sep=3):
    """Print dendrogram of a binary tree.  Each tree node is represented by a
    length-2 tuple. printDendrogram is written and provided by David Eppstein
    2002. Accessed on 14 April 2014:
    http://code.activestate.com/recipes/139422-dendrogram-drawing/ """

    def isPair(T):
        return type(T) == tuple and len(T) == 2

    def maxHeight(T):
        if isPair(T):
            h = max(maxHeight(T[0]), maxHeight(T[1]))
        else:
            h = len(str(T))
        return h + sep

    activeLevels = {}

    def traverse(T, h, isFirst):
        if isPair(T):
            traverse(T[0], h-sep, 1)
            s = [' ']*(h-sep)
            s.append('|')
        else:
            s = list(str(T))
            s.append(' ')

        while len(s) < h:
            s.append('-')

        if (isFirst >= 0):
            s.append('+')
            if isFirst:
                activeLevels[h] = 1
            else:
                del activeLevels[h]

        A = list(activeLevels)
        A.sort()
        for L in A:
            if len(s) < L:
                while len(s) < L:
                    s.append(' ')
                s.append('|')

        print (''.join(s))

        if isPair(T):
```

```
            traverse(T[1], h-sep, 0)

    traverse(T, maxHeight(T), -1)
```

```
filename = '//Users/raz/Dropbox/guide/pg2dm-python/ch8/dogs.csv'
 n
hg = hClusterer(filename)
cluster = hg.cluster()
printDendrogram(cluster)
```

When I run this code I get the following results:

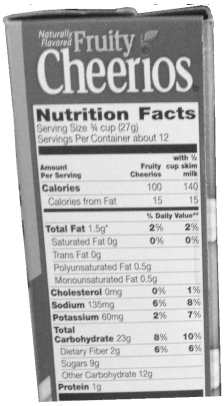
which match the results we computed by hand. That's encouraging.

you try!

Breakfast Cereals

On the book's website, there is a file containing nutritional information about 77 breakfast cereals including

- cereal name
- calories per serving
- protein (in grams)
- fat (in grams)
- sodium (in mg)
- fiber (grams)
- carbohydrates (grams)
- sugars (grams)
- potassium (mg)
- vitamins (% of RDA)



Can you perform hierarchical clustering of this data?

Which cereal is most similar to Trix?

To Muesli Raisins & Almonds?

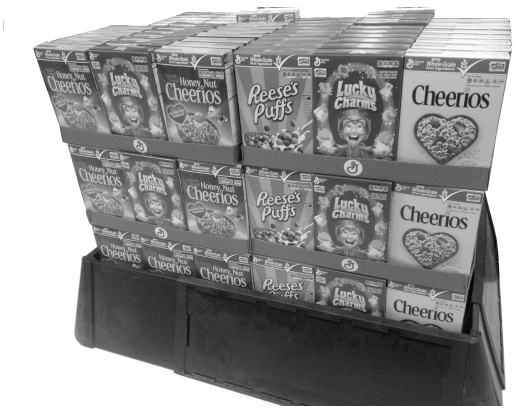
This data set is from Carnegie Mellon University:
http://lib.stat.cmu.edu/DASL/DataFiles/Cereals.html

you try - results

To run the clusterer on this dataset we only needed to change the filename from dogs.csv to cereal.csv. Here is an abbreviated version of the results:

```
Mueslix Crispy Blend ------------------------------------------------------------------------+
                                                                                             |--+
Muesli Raisins & Almonds ---------------------------------------------------------------+    |
                                                                                        |--+
Muesli Peaches & Pecans ----------------------------------------------------------------+

...

Lucky Charms ----------+
                       |--+
Fruity Pebbles --+     |
                 |--+  |
Trix ------------+  |  |
                    |--+
Cocoa Puffs -----+  |
                 |--+
Count Chocula ---+
```

Trix, is most similar to Fruity Pebbles. (I recommend you confirm this by running out right now and buying a box of each.) Perhaps not surprisingly, Muesli Raisins & Almonds is closest to Muesli Peaches & Pecans.



That's it for hierarchical clustering!

That was pretty easy!

Introducing ...

k-means clustering

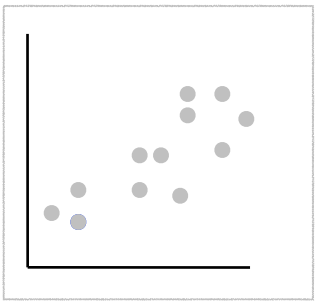
With k-means clustering we specify how many clusters to make. This is the 'k'. If we want to make 2 groups k = 2, if we want to make 100, k=100.

k-means clustering is The Most Popular clustering algorithm!

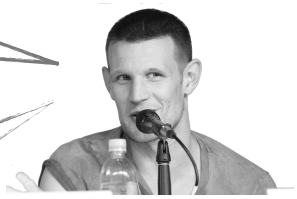
K-means is cool!

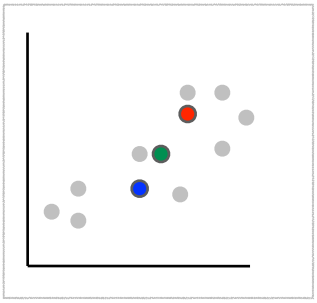
The algorithm is over 50 years old! It was first proposed by Dr. Stuart Lloyd of Bell Labs in 1957.

Here is what you need to know about k-means



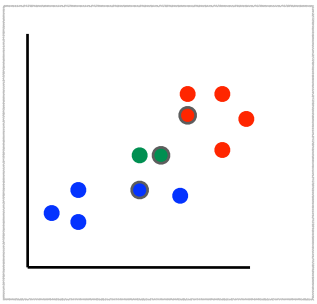
Here are some instances we want to cluster into 3 groups (k=3). Suppose they are dog breeds as mentioned earlier and the dimensions are height and weight.



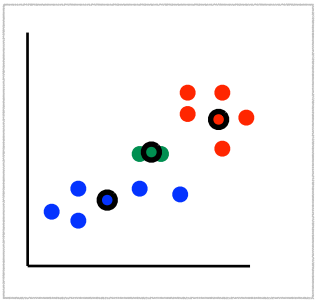


Because k=3, we pick 3 random points as the initial centroids of each cluster ('initial centroid' means the initial center or mean of the cluster).

Right then. We've indicated these initial centroids as red, green, and blue circles.



Okay. Next, we are going to assign each instance to the nearest centroid. The points assigned to each centroid are a cluster. So we have created k initial clusters!!



Now, for each cluster, we compute the mean (average) point of that cluster. This will be our updated centroid.

And repeat (assign each instance to the centroid & recompute centroids) until the centroids don't move much or we have reached some maximum number of iterations.

The basic k-means algorithm is:

```
1. select k random instances to be the
   initial centroids
2. REPEAT
3. assign each instance to the nearest
   centroid. (forming k clusters)
4. update centroids by computing mean
   of each cluster
5. UNTIL centroids don't change (much).
```

Let's go through an example. Consider the following points (x and y coordinates):

```
(1, 2)
(1, 4)
(2, 2)
(2, 3)
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

Say we want to cluster these into 2 groups.

step 1 of above algorithm: select k random instances to be initial centroids.

Suppose we randomly select (1, 4) as centroid 1 and (4, 2) as centroid 2.

step 3: assign each instance to the nearest centroid

To assign each instance to the nearest centroid we can use any of the distance measures we have previously discussed. To keep things simple, for this example let's use Manhattan Distance.

point	distance from centroid 1 (1, 4)	distance from centroid 2 (4, 2)
(1, 2)	2	3
(1,4)	0	5
(2, 2)	3	2
(2, 3)	2	3
(4, 2)	5	0
(4, 4)	3	2
(5, 1)	7	2
(5, 3)	5	2

Based on these distances we assign the points to the following clusters:

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 3)
```

```
CLUSTER 2
(2, 2)
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

step 4: update centroids

We compute the new centroids by computing the mean of each cluster. The mean x coordinate of cluster 1 is:

(1 + 1 + 2) / 3 = 4/3 = 1.33

and the mean y is

(2 + 4 + 3) / 3 = 9/3 = 3

So the new cluster 1 centroid is (1.33, 3).

The new centroid for cluster 2 is (4, 2.4)

step 5: until centroids don't change

The old centroids were (1, 4) and (4, 2) and the new ones are (1.33, 3) and (4, 2.4). The centroids changed so we repeat.

step 3: assign each instance to the nearest centroid

Again we compute Manhattan Distance.

point	distance from centroid 1 (1.33, 3)	distance from centroid 2 (4, 2.4)
(1, 2)	1.33	3.4
(1, 4)	1.33	4.6
(2, 2)	1.67	2.4
(2, 3)	0.67	2.6
(4, 2)	3.67	0.4
(4, 4)	3.67	1.6
(5, 1)	5.67	2.4
(5, 3)	3.67	1.6

and based on these distances assign the points to clusters:

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)
```

```
CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

step 4: update centroids

We compute the new centroids by computing the mean of each cluster.

Cluster 1 centroid: (1.5, 2.75)

Cluster 2 centroid: (4.5, 2.5)

step 5: until centroids don't change

The centroids changed so we repeat.

step 3: assign each instance to the nearest centroid

Again we compute Manhattan Distance.

point	distance from centroid 1 (1.5, 2.75)	distance from centroid 2 (4.5, 2.5)
(1, 2)	1.25	4.0
(1, 4)	1.75	5.0
(2, 2)	1.25	3.0
(2, 3)	0.75	3.0
(4, 2)	3.25	1.0
(4, 4)	3.75	2.0
(5, 1)	5.25	2.0
(5, 3)	3.75	1.0

and based on these distances assign the points to clusters:

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)
```

```
CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

step 4: update centroids

We compute the new centroids by computing the mean of each cluster.

Cluster 1 centroid: (1.5, 2.75)

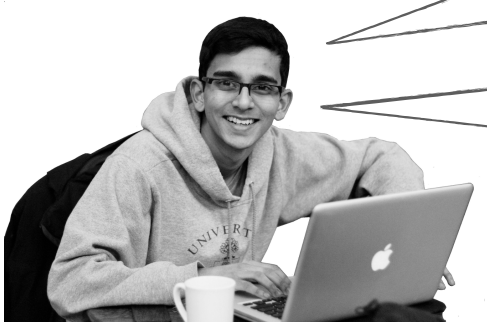
Cluster 2 centroid: (4.5, 2.5)

step 5: until centroids don't change

The updated centroids are identical to the previous ones so the algorithm converged on a solution and we can stop. The final clusters are

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)
```

```
CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```



We stop when the centroids don't change. This is the same condition as saying no point are shifting from one cluster to another. This is what we mean when we say the algorithm 'converges'.

During the execution of the algorithm, the centroids shift from their initial position to some final position. The vast majority of this shift occurs during the first few iterations. Often, the centroids barely move during the final iterations.

This means that the k-means algorithm produces good clusters early on and later iterations are likely to produce only minor refinements.

Because of this behavior of the algorithm, we can dramatically reduce its execution time by relaxing our criteria of "no points are shifting from one cluster to another" to "fewer than 1% of the points are shifting from one cluster to another."
This is a common approach!

K-means is simple!

For you computer science geeks:

K-means is an instance of the Expectation Maximization (EM) Algorithm, which is an iterative method that alternates between two phases. We start with an initial estimate of some parameter. In the K-means case we start with an estimate of the centroids. In the expectation (E) phase, we use this estimate to place points into their **expected** cluster. In the Maximization (M) phase we use these expected values to adjust the estimate of the centroids. If you are interested in learning more about the EM algorithm the wikipedia page http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm is a good place to start.

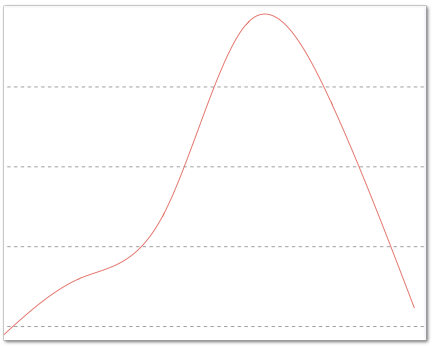
Hill Climbing

I would like to briefly interrupt our discussion of K-means clustering to talk about hill climbing algorithms. Suppose our goal is to reach the peak of some mountain and we come up with the following algorithm:

```
start at some random location on the mountain.
REPEAT
   take a step in the direction that will take you higher.
UNTIL there is no direction that will take you higher.
```

This seems like a reasonable algorithm.

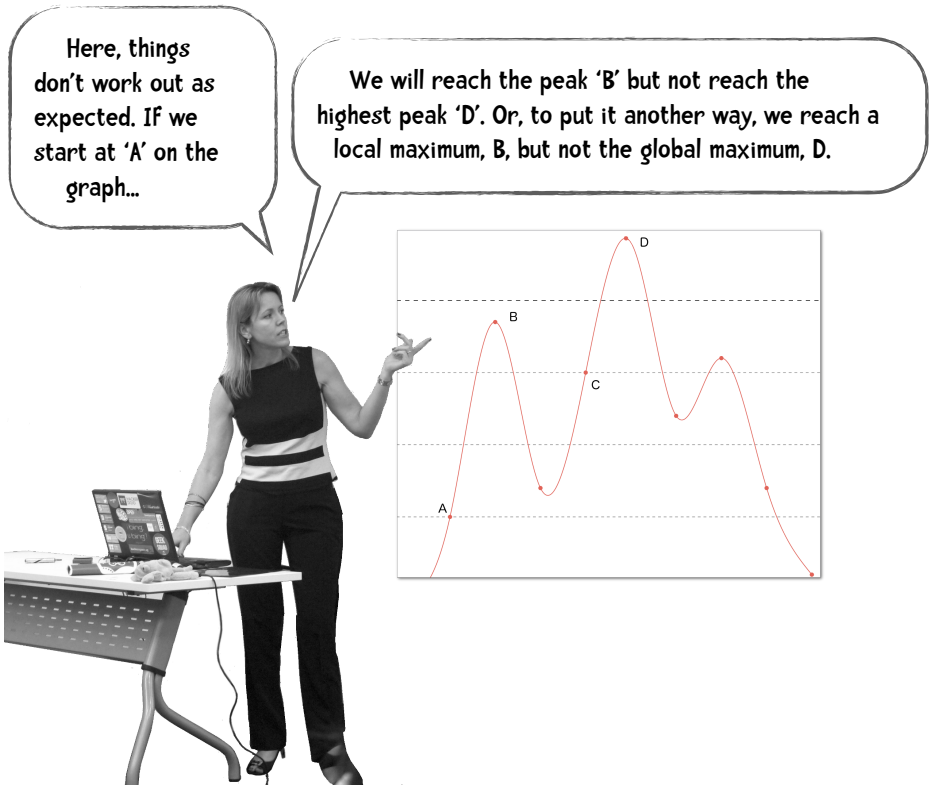
Consider using it with the mountain shown here ⇨



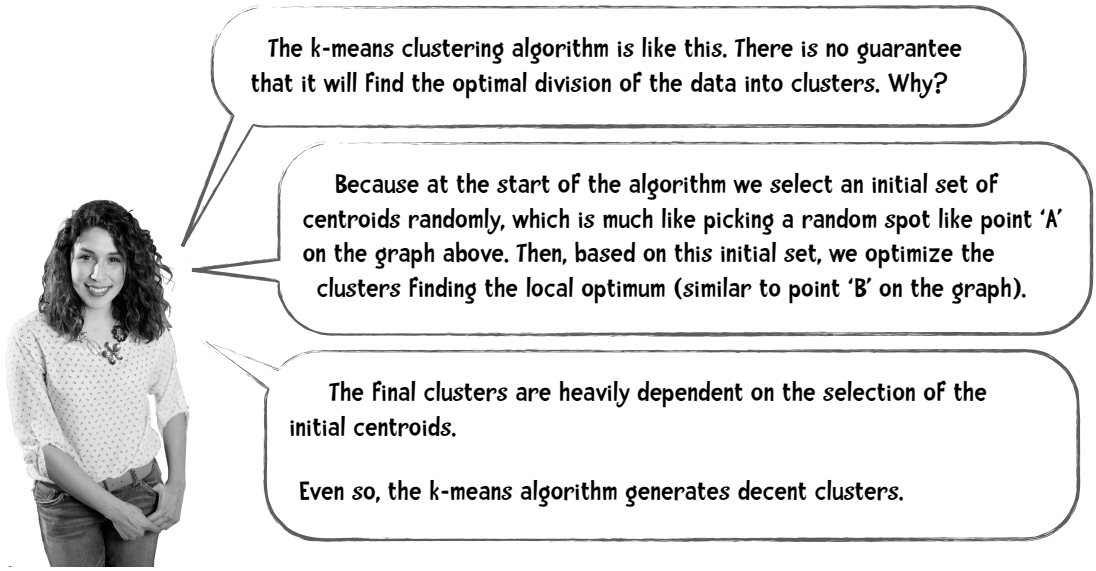
You can see that regardless of where we are plopped down on the mountain, we will reach the peak if we follow the algorithm.

And if we think of this as a graph, we will reach the peak value regardless of where we start on the graph.

Now let's consider using the algorithm with the graph on the following page



Thus, this simple version of the hill-climbing algorithm is not guaranteed to reach the optimal solution.



SSE or Scatter

To determine the quality of a set of clusters we can use the **sum of the squared error** (SSE). This is also called **scatter**. Here is how to compute it: for each point we will square the distance from that point to its centroid, then add all those squared distances together. More formally,

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} dist(c_i, x)^2$$

Let's dissect that. In the first summation sign we are iterating over the clusters. So initially i equals cluster 1, then i equals cluster 2, up to i equals cluster k. The next summation sign iterates over the points in that cluster—something like, for each point x in cluster i. *Dist* is whatever distance formula we are using (for example, Manhattan, or Euclidean). So we compute the distance between that point, x, and the centroid for the cluster c_i, square that distance and add it to our total.

Let's say we run our k-means algorithm twice on the same data and for each run we pick a different set of random initial centroids. Is the set of clusters that were computed during the first run worse or better than the set computed during the second run? To answer that question we compute the SSE for both sets of clusters. The set with the smaller SSE is the better of the two.

Time to start coding!

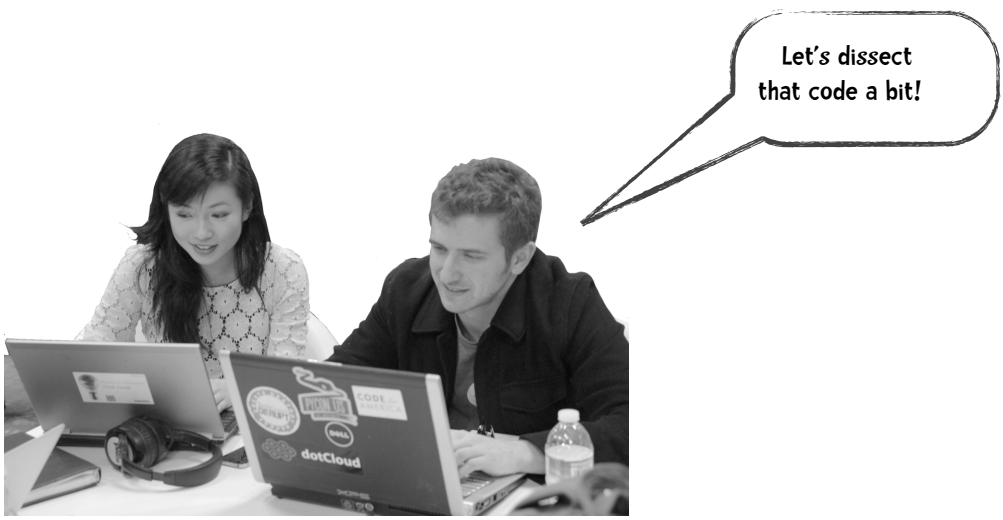
Here's the code for basic k-means

```
import math
import random


def getMedian(alist):
    """get median of list"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2


def normalizeColumn(column):
    """normalize the values of a column using Modified Standard Score
    that is (each value - median) / (absolute standard deviation)"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result


class kClusterer:
    """ Implementation of kMeans Clustering
    This clusterer assumes that the first column of the data is a label
    not used in the clustering. The other columns contain numeric data
    """

    def __init__(self, filename, k):
        """ k is the number of clusters to make
        This init method:
           1. reads the data from the file named filename
           2. stores that data by column in self.data
           3. normalizes the data using Modified Standard Score
```

```
   4. randomly selects the initial centroids
   5. assigns points to clusters associated with those centroids
"""
file = open(filename)
self.data = {}
self.k = k
self.counter = 0
self.iterationNumber = 0
# used to keep track of % of points that change cluster membership
# in an iteration
self.pointsChanged = 0
# Sum of Squared Error
self.sse = 0
#
# read data from file
#
lines = file.readlines()
file.close()
header = lines[0].split(',')
self.cols = len(header)
self.data = [[] for i in range(len(header))]
# we are storing the data by column.
# For example, self.data[0] is the data from column 0.
# self.data[0][10] is the column 0 value of item 10.
for line in lines[1:]:
    cells = line.split(',')
    toggle = 0
    for cell in range(self.cols):
        if toggle == 0:
           self.data[cell].append(cells[cell])
           toggle = 1
        else:
            self.data[cell].append(float(cells[cell]))

self.datasize = len(self.data[1])
self.memberOf = [-1 for x in range(len(self.data[1]))]
#
# now normalize number columns
#
for i in range(1, self.cols):
        self.data[i] = normalizeColumn(self.data[i])

# select random centroids from existing points
random.seed()
self.centroids = [[self.data[i][r]  for i in range(1, len(self.data))]
                  for r in random.sample(range(len(self.data[0])),
                                         self.k)]
self.assignPointsToCluster()
```

```
    def updateCentroids(self):
        """Using the points in the clusters, determine the centroid
        (mean point) of each cluster"""
        members = [self.memberOf.count(i) for i in range(len(self.centroids))]
        self.centroids = [[sum([self.data[k][i]
                                for i in range(len(self.data[0]))
                                if self.memberOf[i] == centroid])/members[centroid]
                           for k in range(1, len(self.data))]
                          for centroid in range(len(self.centroids))]


    def assignPointToCluster(self, i):
        """ assign point to cluster based on distance from centroids"""
        min = 999999
        clusterNum = -1
        for centroid in range(self.k):
            dist = self.euclideanDistance(i, centroid)
            if dist < min:
                min = dist
                clusterNum = centroid
        # here is where I will keep track of changing points
        if clusterNum != self.memberOf[i]:
            self.pointsChanged += 1
        # add square of distance to running sum of squared error
        self.sse += min**2
        return clusterNum


    def assignPointsToCluster(self):
        """ assign each data point to a cluster"""
        self.pointsChanged = 0
        self.sse = 0
        self.memberOf = [self.assignPointToCluster(i)
                         for i in range(len(self.data[1]))]


    def euclideanDistance(self, i, j):
        """ compute distance of point i from centroid j"""
        sumSquares = 0
        for k in range(1, self.cols):
            sumSquares += (self.data[k][i] - self.centroids[j][k-1])**2
        return math.sqrt(sumSquares)

    def kCluster(self):
        """the method that actually performs the clustering
        As you can see this method repeatedly
            updates the centroids by computing the mean point of each cluster
            re-assign the points to clusters based on these new centroids
```

```
        until the number of points that change cluster membership
        is less than 1%.
        """
        done = False

        while not done:
            self.iterationNumber += 1
            self.updateCentroids()
            self.assignPointsToCluster()
            #
            # we are done if fewer than 1% of the points change clusters
            #
            if float(self.pointsChanged) / len(self.memberOf) <  0.01:
                done = True
        print("Final SSE: %f" % self.sse)

    def showMembers(self):
        """Display the results"""
        for centroid in range(len(self.centroids)):
             print ("\n\nClass %i\n========" % centroid)
             for name in [self.data[0][i]  for i in range(len(self.data[0]))
                          if self.memberOf[i] == centroid]:
                 print (name)

##
## RUN THE K-MEANS CLUSTERER ON THE DOG DATA USING K = 3
###
km = kClusterer('dogs2.csv', 3)
km.kCluster()
km.showMembers()
```



As with our code for the hierarchical clusterer, we are storing the data by column. Consider our dog breed data. If we represent the data in spreadsheet form, it would likely look like this (the height and weight are normalized):

breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
German Shepherd	0.9016	0.81481
...	...	...

And if we were to transfer this data to Python we would likely make a list that looks like the following:

```
data = [ data for the Border Collie,
         data for the Boston Terrier,
         ... ]
```

So to fully specify the data format:

```
data = [ ['Border Collie',  0, -0.1455],
         ['Boston Terrier', -0.7213, -0.873],
         ... ]
```

So we are storing the data by row. This seems like the common sense approach and the one we have been using throughout the book. Alternatively, we can store the data column first:

```
data = [ column 1 data,
         column 2 data,
         column 3 data ]
```

So for our dog example:

```
data = [ [‘Border Collie’, ‘Boston Terrier’, ‘Brittany Spaniel’, ...],
         [ 0, –0.7213, –0.3607, ...],
         [–0.1455, –0.7213, –0.4365, ...],
         ... ]
```

This is what we did for the hierarchical clusterer and what we are doing here for k-means. The benefit of this approach is that it makes implementing many of the math functions easier. We can see this in the first two procedures in the code above, `getMedian` and `normalizeColumn`. Because we stored the data by column, these procedures take simple lists as arguments.

```
>>> normalizeColumn([8, 6, 4, 2])
[1.5, 0.5, –0.5, –1.5]
```

The constructor method, `__init__` takes as arguments, the filename of the data file and k, the number of clusters to construct. It reads the data from the file and stores the data by column. It normalizes the data using the `normalizeColumn` procedure, which implements the Modified Standard Score method. Finally, it selects *k* elements from this data as the initial centroids and assigns each point to a cluster depending on that point’s distance to the initial centroids. It does this assignment using the method `assignPointsToCluster`.

The method, kCluster actually performs the clustering by repeatedly calling updateCentroids, which computes the mean of each cluster and `assignPointsToCluster` until fewer than 1% of the points change clusters. The method showMembers simply displays the results.

Running the code on the dog breed data yields the following results:

```
Final SSE: 5.243159


Class 0
========
Bullmastiff
Great Dane
```

```
Class 1
========
Boston Terrier
Chihuahua
Yorkshire Terrier


Class 2
========
Border Collie
Brittany Spaniel
German Shepherd
Golden Retriever
Portuguese Water Dog
Standard Poodle
```

Wow! For this small dataset the clusterer does extremely well.

You try

How well does the kmeans clusterer work with the cereal dataset with k = 4

- Do the sweet cereals cluster together (Cap'n'Crunch, Cocoa Puffs, Froot Loops, Lucky Charms?
- Do the bran cereals cluster together (100% Bran, All-Bran, All-Bran with Extra Fiber, Bran Chex?
- What does Cheerios cluster with?

Try the clusterer with the auto mpg dataset with different values for k=8?
Does this follow your expectations of how these cars should be grouped?

You try - my results

How well does the kmeans clusterer work with the cereal dataset with k = 4.

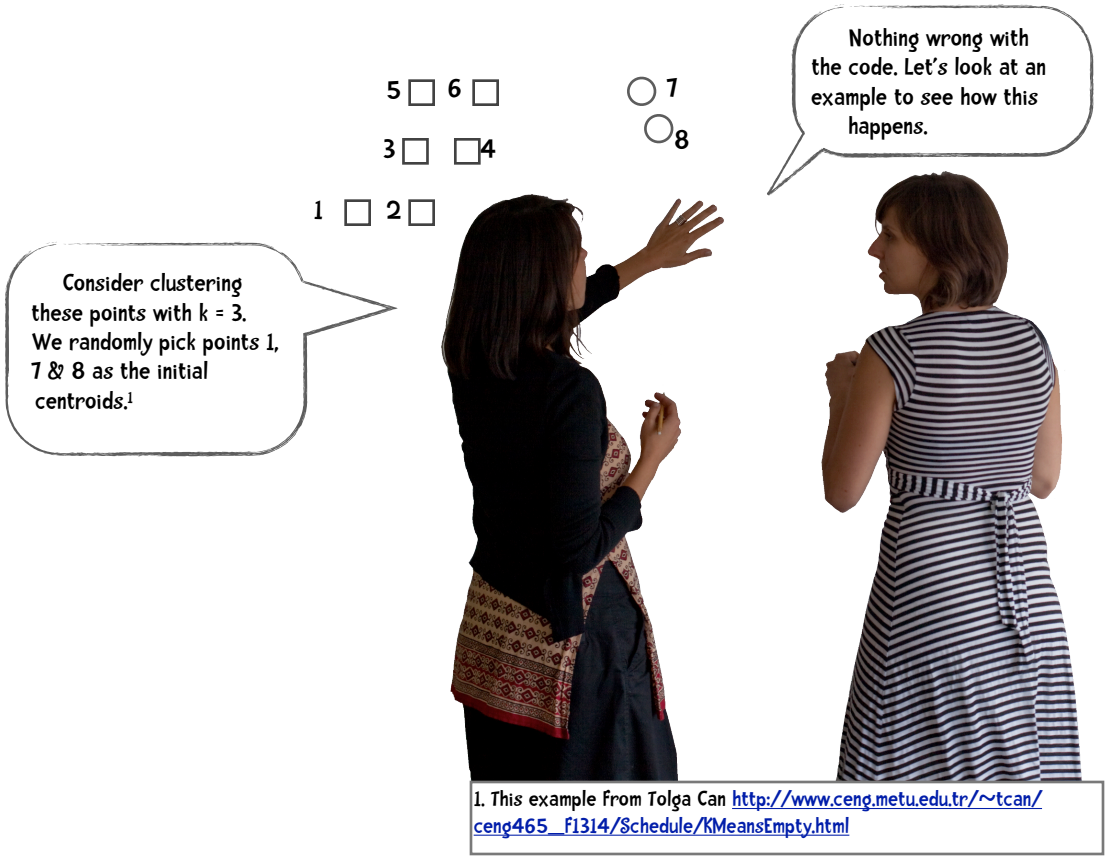
Your results may vary from mine but here is what I found out.

- Do the sweet cereals cluster together (Cap'n'Crunch, Cocoa Puffs, Froot Loops, Lucky Charms?
  Yes, all these sweet cereals (plus Count Chocula, Fruity Pebbles, and others) are in the same sweet cluster.

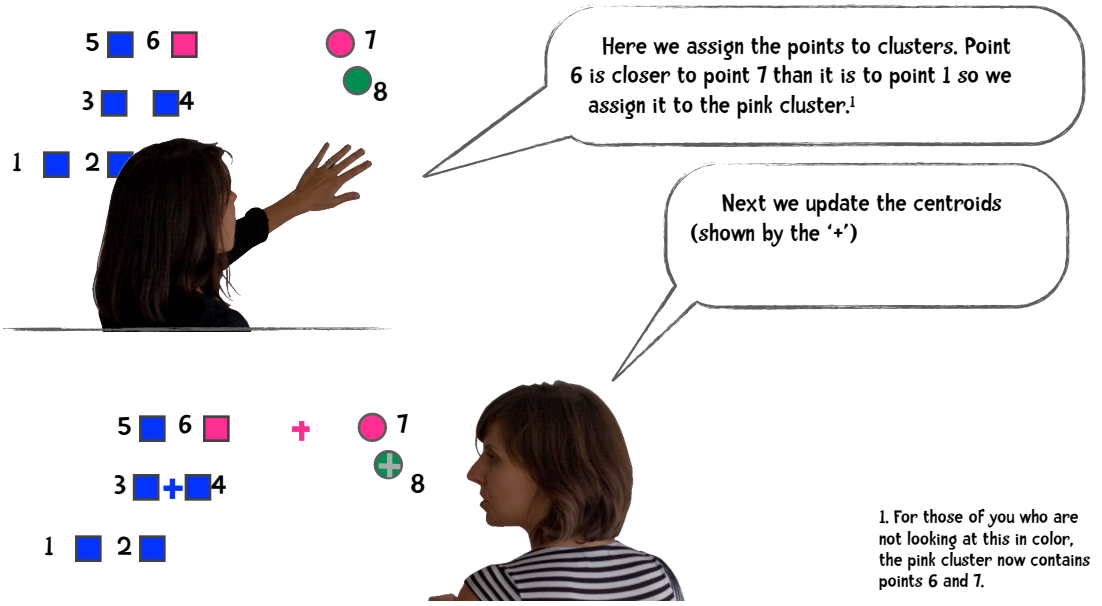
- Do the bran cereals cluster together (100% Bran, All-Bran, All-Bran with Extra Fiber, Bran Chex?
  Again, yes! Included in this cluster are also Raisin Bran and Fruitful Bran.

- What does Cheerios cluster with?
  Cheerios always seems to be in the same cluster as Special K

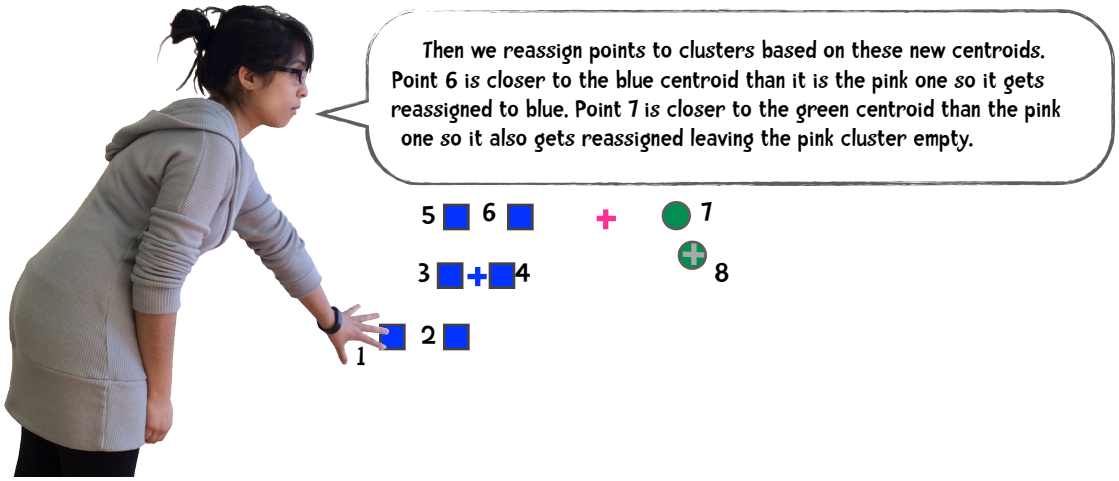
Try the clusterer with the auto mpg dataset with different values for k=8?
Does this follow your expectations of how these cars should be grouped?
The clusterer seems to do a reasonable job on this dataset but on rare occasions you will notice one or more of the clusters are empty.



1. This example from Tolga Can http://www.ceng.metu.edu.tr/~tcan/ceng465_f1314/Schedule/KMeansEmpty.html



1. For those of you who are not looking at this in color, the pink cluster now contains points 6 and 7.



In sum, just because we specify how many groups to make does not mean that the k-means clusterer will produce that many non-empty groups. This may be a good thing. Just looking at the data above, it appears to be naturally clustered into two groups and our attempt to cluster the data into three failed. Suppose we have 1,000 instances we would like to cluster into 10 groups and when we run the clusterer two of the groups are empty. This result may indicate something about the underlying structure of the data. Perhaps the data does not naturally divide into ten groups and we can explore other groupings (trying to cluster into eight groups, for example).

On the other hand, sometimes when we specify 10 clusters we actually want 10 non-empty clusters. If that is the case, we need to alter the algorithm so it detects an empty cluster. Once one is detected the algorithm changes that cluster's centroid to a different point. One possibility is to change it to the instance that is furthest from its corresponding centroid. (In the example above, once we detect the pink cluster is empty, we re-assign the pink centroid to point 1, since point 1 is the furthest point to its corresponding centroid. That is, I compute the distances from

1 to its centroid
2 to its centroid
3 to its centroid
4 to its centroid
5 to its centroid
6 to its centroid
7 to its centroid
8 to its centroid

and pick the point that is furthest from its centroid as the new centroid of the empty cluster.

(sigh) Wouldn't it be dreamy if we could make k-means faster and more accurate.

With a simple change to k-means we can! The new algorithm is called **k-means++**

Even the name makes it sound newer, better, faster, and more accurate —a turbocharged k-means!

k-means++

In the previous section we examined the *k*-means algorithm in its original form as it was developed in the late 50s. As we have seen, it is easy to implement and performs well. It is still the most widely used clustering algorithm on the planet. But it is not without its flaws. A major weakness in *k*-means is in the first step where it **randomly** picks *k* of the datapoints to be the initial centroids. As you can probably tell by my bolding and embiggening the word 'random', it is the random part that is the problem. Because it is random, sometimes the initial centroids are a great pick and lead to near optimal clustering. Other times the initial centroids are a reasonable pick and lead to good clustering. But sometimes—again, because we pick randomly—sometimes the initial centroids are poor leading to non-optimal clustering. The *k*-means++ algorithm fixes this defect by changing the way we pick the initial centroids. Everything else about *k*-means remains the same.

embiggen: verb. To make larger, to make the size increase.

k-means++ -- selecting the initial set of centroids

1. Initially, the set of initial centroids is empty.
2. Select the first centroid randomly from the data points as before.
3. Until we have k initial centroids:
   a. Compute the distance, D, between each datapoint (dp) and its closest centroid. This distance is D(dp).
   b. In a probability proportional to D(dp) select one datapoint at random to be a new centroid and add it to the set of centroids.
   c. REPEAT

Let's dissect the meaning of "In a probability proportional to D(dp) select one datapoint to be a new centroid." To do this, I will present a simple example. Suppose we are in the middle of this process. We have already selected two initial centroids and are in the process of selecting another one. So we are on step 3a of the above algorithm. Let's say we have 5 remaining centroids and their distances to the 2 centroids (c_1 and c_2) are as follows:

	Dc1	Dc2
dp1	5	7
dp2	9	8
dp3	2	5
dp4	3	7
dp5	5	2

Dc1 means "distance to centroid 1 and Dc2 means "distance to centroid 2." dp1 represents datapoint 1.

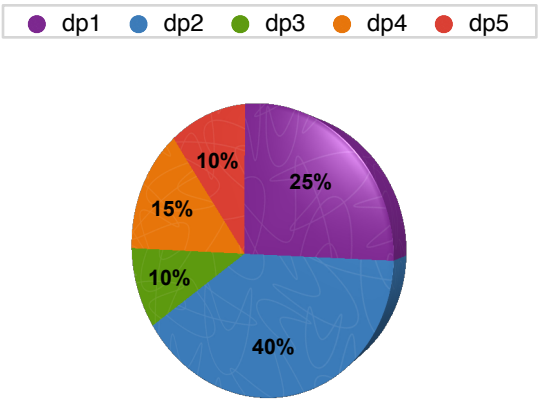
Step 3a says we pick the closest distance so we get:

	closest
dp1	5
dp2	8
dp3	2
dp4	3
dp5	2

Now we are going to convert those numbers to a decimals whose sum equals 1 (I'll call this the weight). To do that we sum the original numbers. In this case the sum equals 20. Now we divide each number by the sum. The result is shown here

	weight
dp1	0.25
dp2	0.40
dp3	0.10
dp4	0.15
dp5	0.10
sum	1.00

I like to think of this as a roulette wheel that looks like this:



We are going to spin a ball on that wheel, see where it lands, and pick that as the new centroid. This is what we mean by "In a probability proportional to D(dp) select one datapoint to be a new centroid."

Let us rough out this idea in Python. Say we have a list tuples containing a datapoint and its weight

```
data = [("dp1", 0.25), ("dp2", 0.4), ("dp3", 0.1),
        ("dp4", 0.15), ("dp5", 0.1)]
```

The function roulette will now select a datapoint in a probability proportional to its weight:

```
import random
random.seed()

def roulette(datalist):
      i = 0
      soFar = datalist[0][1]
      ball = random.random()
      while soFar < ball:
         i += 1
         soFar += datalist[i][1]
      return datalist[i][0]
```

If the function did pick with this proportion, we would predict that if we picked 100 times, 25 of them would be dp1; 40 of them would be dp2; 10 of them dp3; 15 dp4; and 10, dp5. Let's see if that is true:

```
import collections
results = collections.defaultdict(int)
for i in range(100):
      results[roulette(data)] += 1
print results
```

```
{'dp5': 11, 'dp4': 15, 'dp3': 10, 'dp2': 38, 'dp1': 26}
```

Great! Our function does return datapoints in roughly the correct proportion.

The idea in k-means++ clustering is that, while we still pick the initial centroids randomly, we prefer centroids that are far away from one another.

Code It

Can you implement k-means++ in Python?

Again, the only difference between our previous implementation of k-means and this code is in how we select the initial centroids. Make a copy of our original k-means code and modify it. Our original code created the initial centroids in this line:

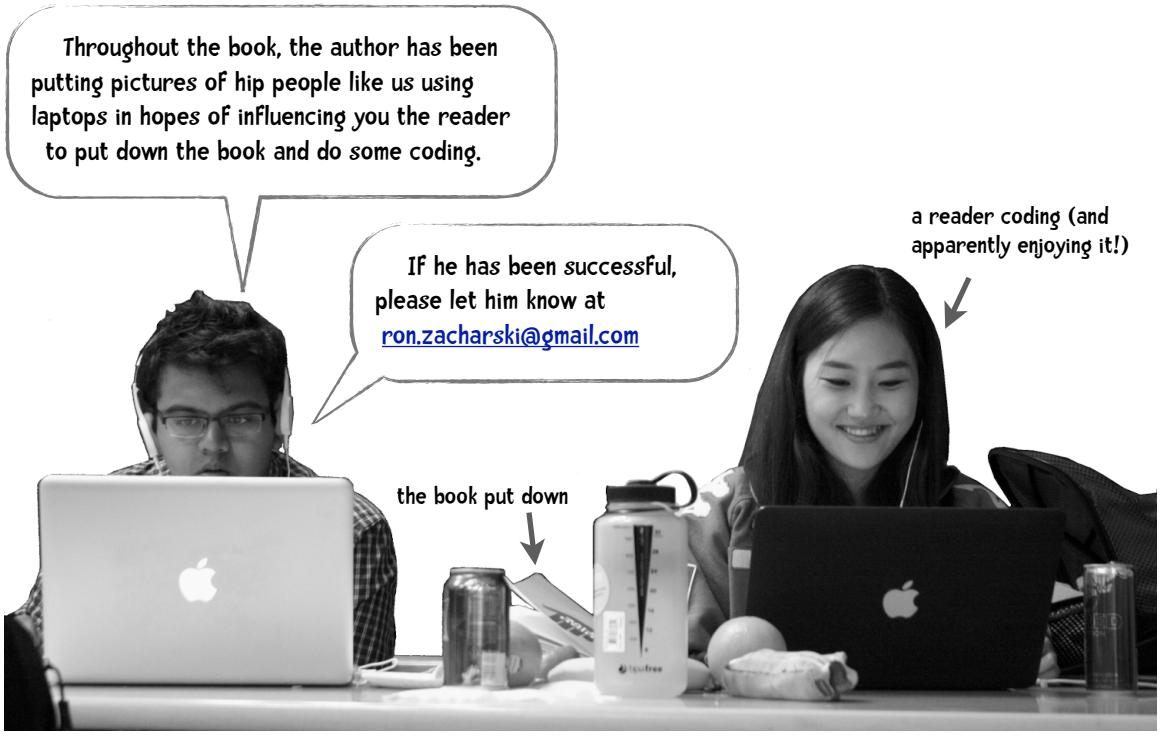
```
self.centroids = [[self.data[i][r]  for i in range(1, len(self.data))]
                  for r in random.sample(range(len(self.data[0])),
                                         self.k)]
```

Let us replace that line with:

```
self.selectInitialCentroids()
```

Your job is to write that method!

Good luck!



Code It -solution

Here is my version of selectInitialCentroids:

```
def distanceToClosestCentroid(self, point, centroidList):
    result = self.eDistance(point, centroidList[0])
    for centroid in centroidList[1:]:
        distance = self.eDistance(point, centroid)
        if distance < result:
            result = distance
    return result

def selectInitialCentroids(self):
    """implement the k-means++ method of selecting
    the set of initial centroids"""
    centroids = []
    total = 0
    # first step is to select a random first centroid
    current = random.choice(range(len(self.data[0])))
    centroids.append(current)
    # loop to select the rest of the centroids, one at a time
    for i in range(0, self.k - 1):
        # for every point in the data find its distance to
        # the closest centroid
        weights = [self.distanceToClosestCentroid(x, centroids)
                   for x in range(len(self.data[0]))]
        total = sum(weights)
        # instead of raw distances, convert so sum of weight = 1
        weights = [x / total for x in weights]
        #
        # now roll virtual die
        num = random.random()
        total = 0
        x = -1
        # the roulette wheel simulation
        while total < num:
            x += 1
            total += weights[x]
        centroids.append(x)
    self.centroids = [[self.data[i][r] for i in range(1, len(self.data))]
                      for r in centroids]
```

The Python code for the entire k-means++ classifier is on the book's website: http://guidetodatamining.com

Summary

Clustering is all about discovery. However, the simple examples we have been using in this chapter may obscure this fundamental idea. After all, we know how to cluster breakfast cereals without a computer's help—sugary cereals, healthy cereals. And we know how to cluster car models—a Ford F150 goes in the truck category, a Mazda Miata in the sports car category, and a Honda Civic in the fuel efficient category. But consider a task where discovery IS important.

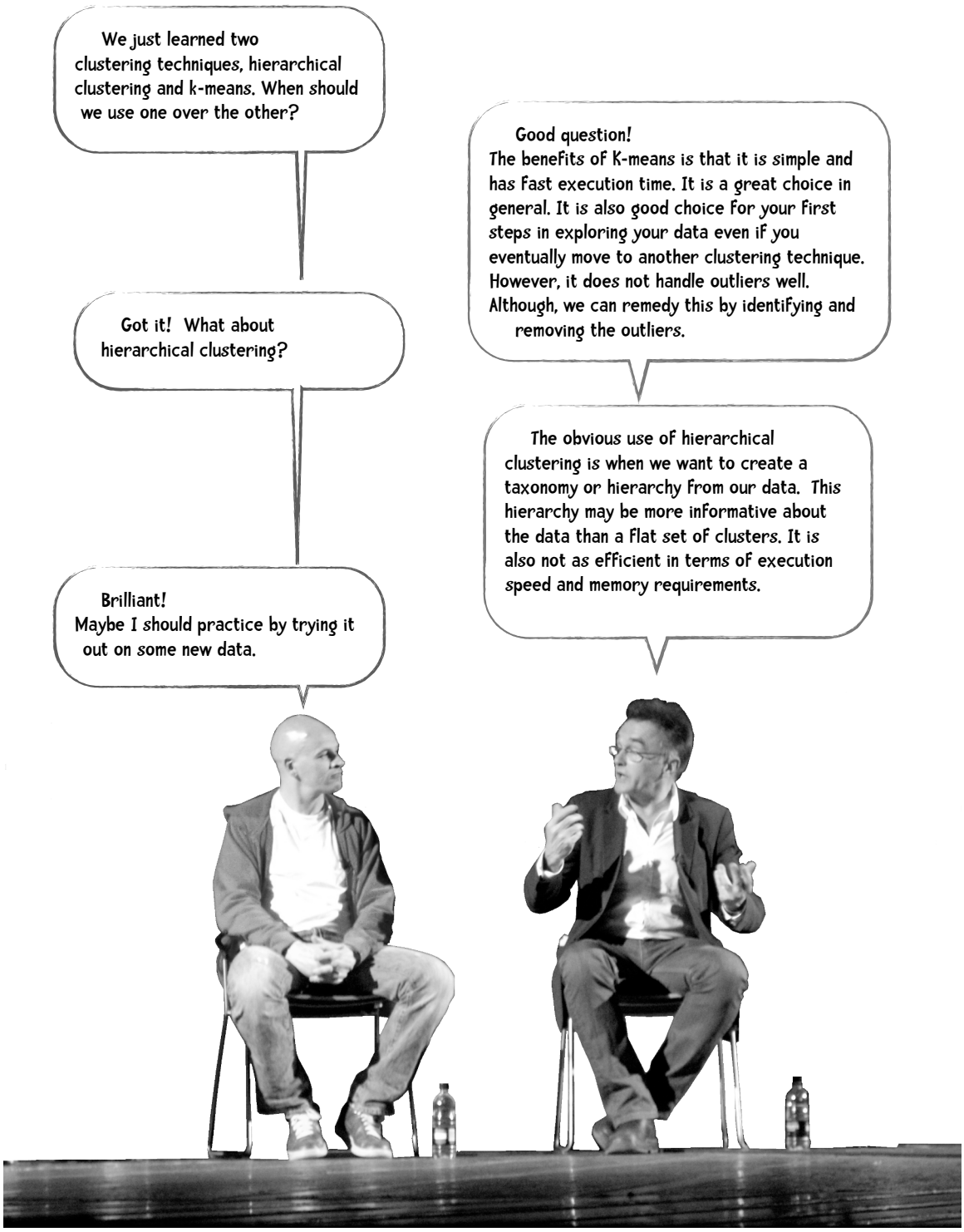
When we do a web search we are presented with a long list of results. For example, when I just did a Google search on "carbon sequestration" I get over 2.8 million results. A number of researchers have examined the benefits of clustering these results. Instead of that long list of carbon sequestration results we might also see categories like "carbon sequestration in freshwater wetlands" and "carbon sequestration in forests."

Josh Gotbaum's team conducted extensive interviews with 3,000 people asking them questions about their values. Using these interviews they clustered the people into five groups. When they examined the clusters they gave them the descriptions:

1. extending opportunity to others
2. working within a community
3. achieving independence
4. focusing on family
5. defending righteousness

They then crafted targeted campaign ads to each group.

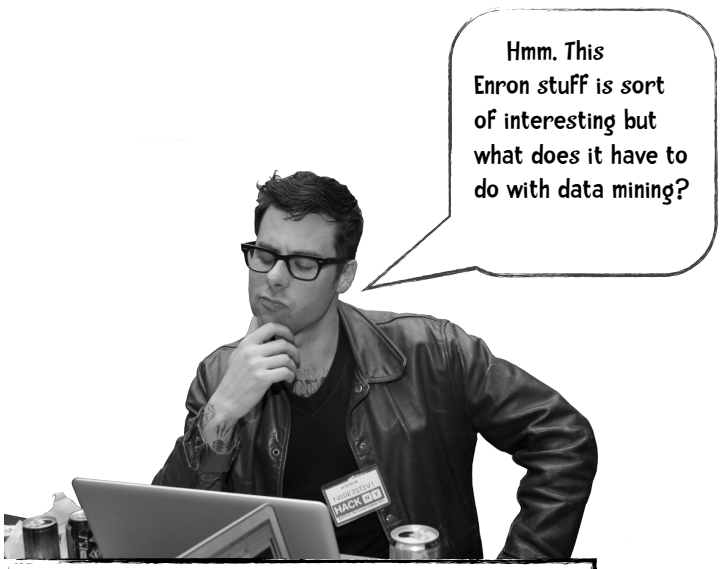
from The Numerati by Stephen Baker


We just learned two clustering techniques, hierarchical clustering and k-means. When should we use one over the other?
Good question! The benefits of K-means is that it is simple and has fast execution time. It is a great choice in general. It is also good choice for your first steps in exploring your data even if you eventually move to another clustering technique. However, it does not handle outliers well. Although, we can remedy this by identifying and removing the outliers.
Got it! What about hierarchical clustering?
The obvious use of hierarchical clustering is when we want to create a taxonomy or hierarchy from our data. This hierarchy may be more informative about the data than a flat set of clusters. It is also not as efficient in terms of execution speed and memory requirements.
Brilliant! Maybe I should practice by trying it out on some new data.

Enron

Perhaps you remember Enron and the Enron Scandal. In its heyday Enron was a mega-huge energy company with revenues over $100 billion and over 20,000 employees (Microsoft's revenue then was only $22 billion). Due to systemic sleaziness and corruption including creating an artificial energy shortage that resulted in electricity blackouts in California, Enron went bankrupt and a bunch of people went to jail. For a documentary about this see Enron: The Smartest Guys in the Room, which is available for streaming from Netflix and Amazon Prime.

Now you might be thinking "Hey, this Enron stuff is sort of interesting but what does it have to do with data mining?"



For more information on the Enron database see the Wikipedia entry: http://en.wikipedia.org/wiki/Enron_Corpus
and
https://www.cs.cmu.edu/~./enron/

This database is an amazing resource for reearchers in a wide variety of areas.

We are going to try to cluster a small part of the Enron corpus. For our simple test corpus, I have extracted the information of who sent email to whom and represented it in table form as shown here:

	Kay	Chris	Sara	Tana	Steven	Mark
Kay	0	53	37	6	0	12
Chris	53	0	1	0	2	0
Sara	37	1	0	1144	0	962
Tana	6	0	1144	0	0	1201
Steven	0	0	2	0	0	0
Mark	12	0	962	1201	0	0

In the dataset provided on our website, I've extracted this information for 90 individuals.

Suppose I am interested in clustering people to discover the relationships among these individuals.

Link analysis

There is an entire subfield of data mining called link analysis devoted to this type of problem (evaluating relationships among entities) and there are specialized algorithms devoted to this task.

You try

Can you perform hierarchical clustering on the Enron email dataset?

You can download the data from our website. (http://www.guidetodatamining.com. You may need to alter the code slightly to better match the problem.

Good luck!

You try - solution

In the dataset provided on our website, I've extracted this information for 90 individuals.

We are clustering the people based on similarity of email correspondence. If most of my email correspondence is with Ann, Ben and Clara, and most of yours is with these people as well, that provides evidence that we are in the same group. The idea is something like this:

between ->	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	127	25	119	5	1	6
your emails	172	35	123	7	3	5

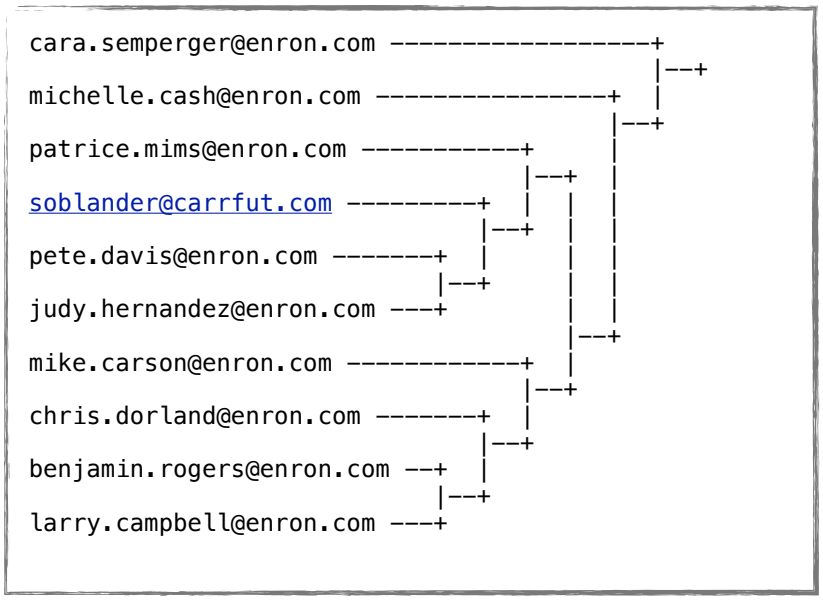
Because our rows are similar, we cluster together. A problem arises when we add in our columns:

between ->	me	you	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	2	190	127	25	119	5	1	6
your emails	190	3	172	35	123	7	3	5

In looking at the 'me' column, you corresponded with me 190 times but I only sent myself email twice. The 'you' column is similar. Now when we compare our rows they don't look so similar. Before I included the 'me' and 'you' columns the Euclidean distance was 46 and after I included them it was 269! To avoid this problem when I compute the Euclidean distance between two people I eliminate the columns for those two people. This required a slight change to the distance formula:

```
def distance(self, i, j):
    #enron specific distance formula
    sumSquares = 0
    for k in range(1, self.cols):
        if  (k != i) and (k != j) :
            sumSquares += (self.data[k][i] - self.data[k][j])**2
    return math.sqrt(sumSquares)
```

Here is a subtree of the results:



I also performed k-means++ on the data, with k = 8. Here are some of the groups it discovered:

```
Class 5
========
chris.germany@enron.com
scott.neal@enron.com
marie.heard@enron.com
leslie.hansen@enron.com
mike.carson@enron.com

Class 6
========
sara.shackleton@enron.com
mark.taylor@enron.com
susan.scott@enron.com

Class 7
========
tana.jones@enron.com
louise.kitchen@enron.com
mike.grigsby@enron.com
david.forster@enron.com
m.presto@enron.com
```

These results are interesting. Class 5 contains a number of traders. Chris Germany and Leslie Hansen are traders. Scott Neal is a vice president of trading. Marie Heard is a lawyer. Mike Carson is a manager of South East trading. The members of Class 7 are also interesting. All I know about Tana Jones is that she is an 'executive'. Louise Kitchen is President of online trading. Mike Grigsby was Vice President of Natural Gas. David Forster was a Vice President of trading. Kevin Presto (m.presto) was also a Vice President and a senior trader.