O'REILLY

From Data Science
*from Scratch*
by Joel Grus

# K-means and hierarchical clustering with Python

LESSON

# K-means and hierarchical clustering with Python

**MATERIALS OR DOWNLOADS NEEDED IN ADVANCE**

Download this lesson's code from GitHub. You'll find this lesson's code in Chapter 19, and you'll need the `linear_algebra functions` from Chapter 4.

*Where we such clusters had*

*As made us nobly wild, not mad*

—Robert Herrick

Some algorithms are examples of what's known as supervised learning, in that they start with a set of *labeled* data and use that as the basis for making predictions about new, unlabeled data. Clustering, however, is an example of unsupervised learning, in which we work with completely unlabeled data (or in which our data has labels but we ignore them).

## The Idea

Whenever you look at some source of data, it's likely that the data will somehow form *clusters*. A data set showing where millionaires live probably has clusters in places like Beverly Hills and Manhattan. A data set showing how many hours people work each week probably has a cluster around 40 (and if it's taken from a state with laws mandating special benefits for people who work at least 20 hours a week, it probably has another cluster right around 19). A data set of demographics of registered voters likely forms a variety of clusters (e.g., “soccer moms,” “bored retirees,” “unemployed millennials”) that pollsters and political consultants likely consider relevant.

There is generally no “correct” clustering. An alternative clustering scheme might group some of the “unemployed millenials” with “grad students,” others with “parents’ basement dwellers.” Neither scheme is necessarily more correct—instead, each is likely more optimal with respect to its own “how good are the clusters?” metric.

Furthermore, the clusters won’t label themselves. You’ll have to do that by looking at the data underlying each one.

## The Model

For us, each `input` will be a vector in *d*-dimensional space (which, as usual, we will represent as a list of numbers). Our goal will be to identify clusters of similar inputs and (sometimes) to find a representative value for each cluster.

For example, each input could be (a numeric vector that somehow represents) the title of a blog post, in which case the goal might be to find clusters of similar posts, perhaps in order to understand what our users are blogging about. Or imagine that we have a picture containing thousands of `(red, green, blue)` colors and that we need to screen-print a 10-color version of it. Clustering can help us choose 10 colors that will minimize the total “color error.”

One of the simplest clustering methods is *k-means*, in which the number of clusters *k* is chosen in advance, after which the goal is to partition the inputs into sets $S_1, \ldots, S_k$ in a way that minimizes the total sum of squared distances from each point to the mean of its assigned cluster.

There are a lot of ways to assign *n* points to *k* clusters, which means that finding an optimal clustering is a very hard problem. We'll settle for an iterative algorithm that usually finds a good clustering:

1. Start with a set of *k-means*, which are points in *d*-dimensional space.
2. Assign each point to the mean to which it is closest.
3. If no point's assignment has changed, stop and keep the clusters.
4. If some point's assignment has changed, recompute the means and return to step 2.

Using the `vector_mean` function, it's pretty simple to create a class that does this:

```
class KMeans:
    """performs k-means clustering"""

    def __init__(self, k):
        self.k = k          # number of clusters
        self.means = None   # means of clusters

    def classify(self, input):
        """return the index of the cluster closest to the
input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input,
self.means[i]))
```

```
    def train(self, inputs):
        # choose k random points as the initial means
        self.means = random.sample(inputs, self.k)
        assignments = None

        while True:
            # Find new assignments
            new_assignments = map(self.classify, inputs)

            # If no assignments have changed, we're done.
            if assignments == new_assignments:
                return

            # Otherwise keep the new assignments,
            assignments = new_assignments

            # And compute new means based on the new
assignments
            for i in range(self.k):
                # find all the points assigned to cluster i
                i_points = [p for p, a in zip(inputs,
assignments) if a == i]

                # make sure i_points is not empty so don't
divide by 0
                if i_points:
                    self.means[i] = vector_mean(i_points)
```

Let’s take a look at how this works.

## Example: Meetups

To celebrate DataSciencester’s growth, your VP of User Rewards wants to organize several in-person meetups for your hometown users, complete with beer, pizza, and DataSciencester t-shirts. You know the locations of all your local users (Figure 1-1), and she’d like you to choose meetup locations that make it convenient for everyone to attend.

Depending on how you look at it, you probably see two or three clusters. (It’s easy to do visually because the data is only two-dimensional. With more dimensions, it would be a lot harder to eyeball.)

Imagine first that she has enough budget for three meetups. You go to your computer and try this:

```
random.seed(0)          # so you get the same results as me
clusterer = KMeans(3)
clusterer.train(inputs)
print clusterer.means
```

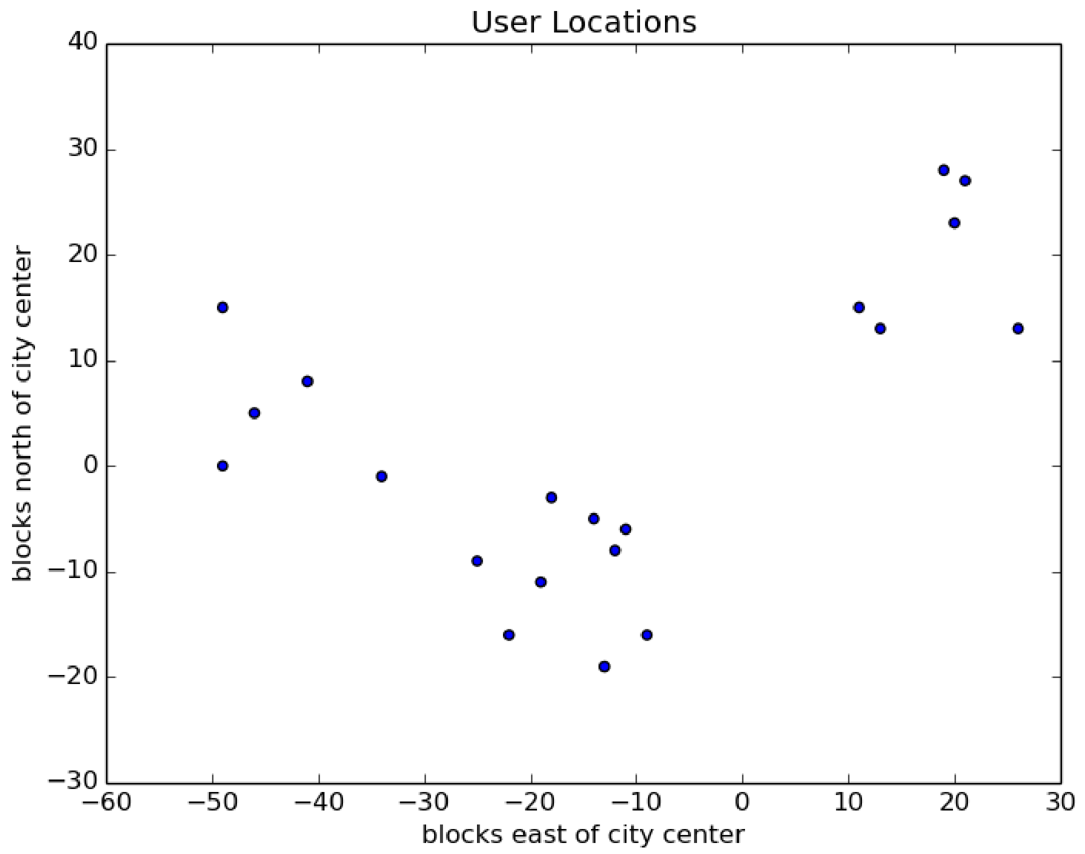

*Figure 1-1. The locations of your hometown users*

You find three clusters centered at [-44,5], [-16,-10], and [18, 20], and you look for meetup venues near those locations (Figure 1-2).

You show it to the VP, who informs you that now she only has enough budget for *two* meetups.

“No problem,” you say:

```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

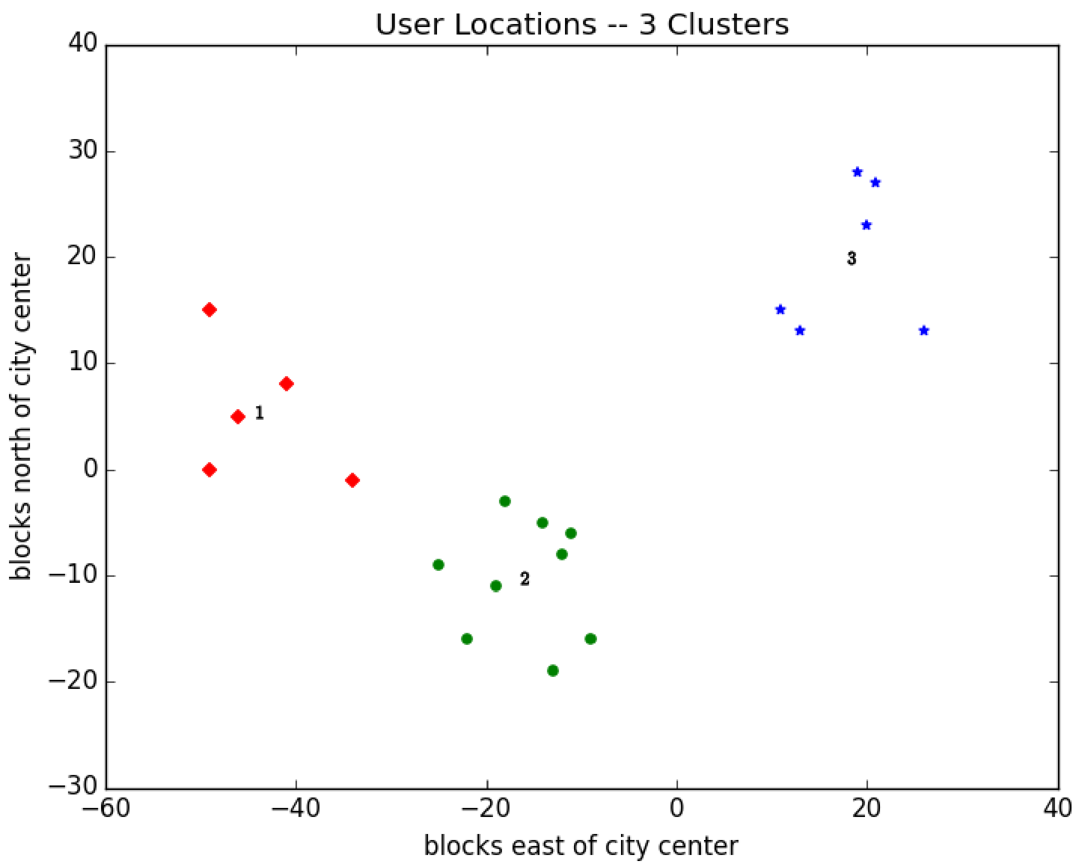

*Figure 1-2. User locations grouped into three clusters*

As shown in Figure 1-3, one meetup should still be near [18, 20], but now the other should be near [-26, -5].

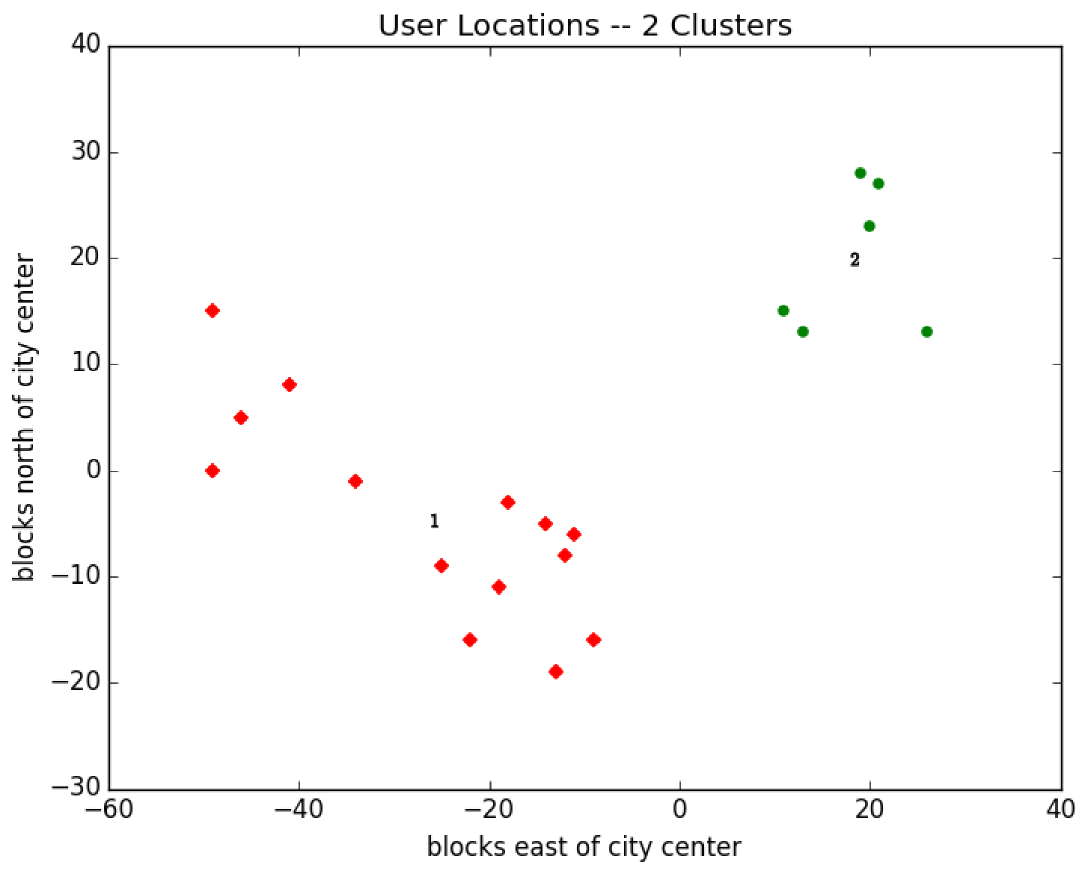

*Figure 1-3. User locations grouped into two clusters*

## Choosing k

In the previous example, the choice of *k* was driven by factors outside of our control. In general, this won't be the case. There is a wide variety of ways to choose a *k*. One that's reasonably easy to understand involves plotting the sum of squared errors (between each point and the mean of its cluster) as a function of *k* and looking at where the graph "bends":

```
def squared_clustering_errors(inputs, k):
    """finds the total squared error from k-means clustering
the inputs"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs,
assignments))

# now plot from 1 up to len(inputs) clusters

ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("total squared error")
plt.title("Total Error vs. # of Clusters")
plt.show()
```

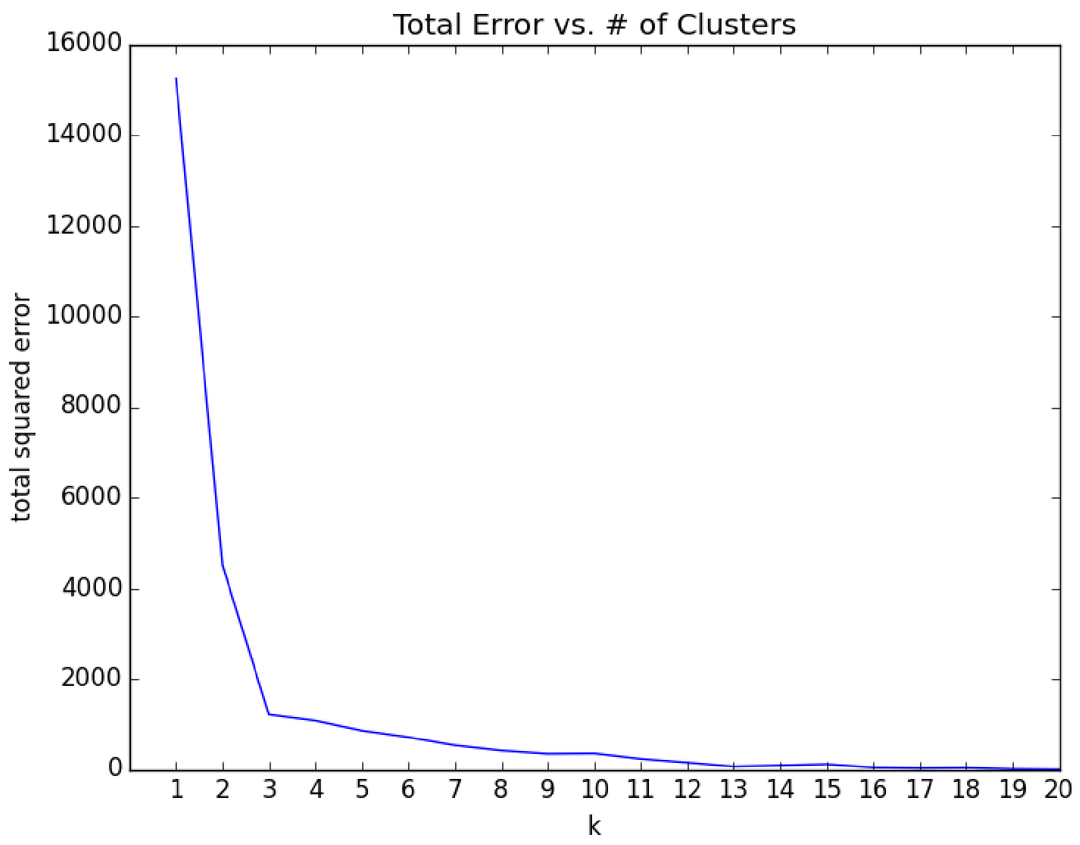

*Figure 1-4. Choosing a k*

Looking at Figure 1-4, this method agrees with our original eyeballing that 3 is the “right” number of clusters.

## Example: Clustering Colors

The VP of Swag has designed attractive DataSciencester stickers that he’d like you to hand out at meetups. Unfortunately, your sticker printer can print at most five colors per sticker. And since the VP of Art is on sabbatical, the VP of Swag asks if there’s some way you can take his design and modify it so that it only contains five colors.

Computer images can be represented as two-dimensional array of pixels, where each pixel is itself a three-dimensional vector `(red,` `green, blue)` indicating its color.

Creating a five-color version of the image then entails:

1. Choosing five colors
2. Assigning one of those colors to each pixel

It turns out this is a great task for k-means clustering, which can partition the pixels into five clusters in red-green-blue space. If we then recolor the pixels in each cluster to the mean color, we're done.

To start with, we'll need a way to load an image into Python. It turns out we can do this with `matplotlib`:

```
path_to_png_file = r"C:\images\image.png"   # wherever your
image is
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

Behind the scenes `img` is a NumPy array, but for our purposes, we can treat it as a list of lists of lists.

`img[i][j]` is the pixel in the *i*th row and *j*th column, and each pixel is a list `[red, green, blue]` of numbers between 0 and 1 indicating the color of that pixel:

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

In particular, we can get a flattened list of all the pixels as:

```
pixels = [pixel for row in img for pixel in row]
```

and then feed them to our clusterer:

```
clusterer = KMeans(5)
clusterer.train(pixels)   # this might take a while
```

Once it finishes, we just construct a new image with the same format:

```
def recolor(pixel):
    cluster = clusterer.classify(pixel)        # index of
the closest cluster
    return clusterer.means[cluster]            # mean of the
closest cluster

new_img = [[recolor(pixel) for pixel in row]   # recolor
this row of pixels
           for row in img]                     # for each
row in the image
```

and display it, using `plt.imshow()`:

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

It is difficult to show color results in a black-and-white book, but Figure 1-5 shows grayscale versions of a full-color picture and the output of using this process to reduce it to five colors:

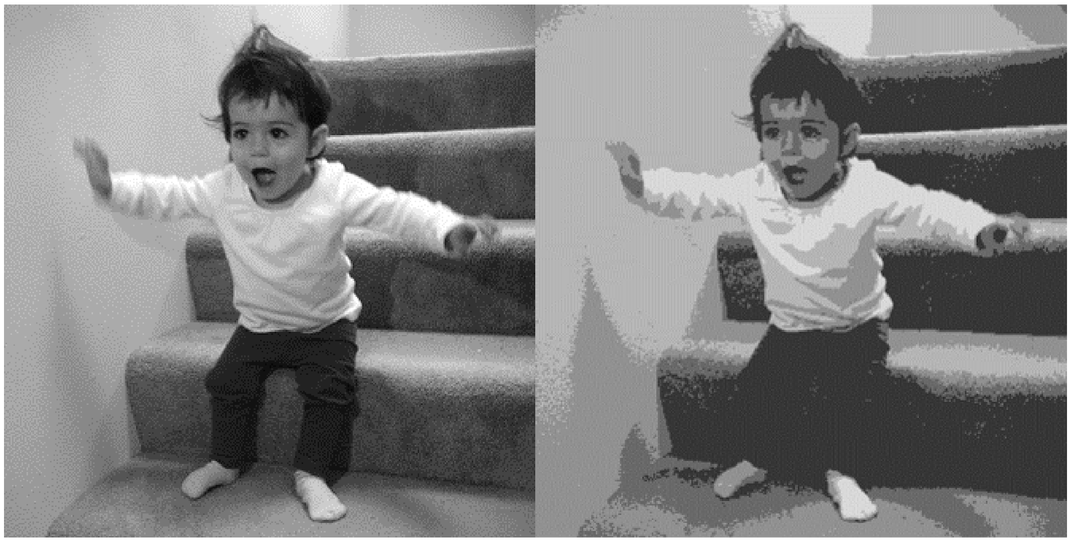

*Figure 1-5. Original picture and its 5-means decoloring*

## Bottom-up Hierarchical Clustering

An alternative approach to clustering is to “grow” clusters from the bottom up. We can do this in the following way:

1. Make each input its own cluster of one.
2. As long as there are multiple clusters remaining, find the two closest clusters and merge them.

At the end, we’ll have one giant cluster containing all the inputs. If we keep track of the merge order, we can recreate any number of clusters by unmerging. For example, if we want three clusters, we can just undo the last two merges.

We’ll use a really simple representation of clusters. Our values will live in *leaf* clusters, which we will represent as 1-tuples:

```
leaf1 = ([10, 20],)   # to make a 1-tuple you need the
trailing comma
```

```
leaf2 = ([30, -15],)  # otherwise Python treats the
parentheses as parentheses
```

We’ll use these to grow *merged* clusters, which we will represent as 2-tuples (merge order, children):

```
merged = (1, [leaf1, leaf2])
```

We’ll talk about merge order in a bit, but first let’s create a few helper functions:

```
def is_leaf(cluster):
    """a cluster is a leaf if it has length 1"""
    return len(cluster) == 1

def get_children(cluster):
    """returns the two children of this cluster if it's a
merged cluster;
    raises an exception if this is a leaf cluster"""
    if is_leaf(cluster):
        raise TypeError("a leaf cluster has no children")
    else:
        return cluster[1]

def get_values(cluster):
    """returns the value in this cluster (if it's a leaf
cluster)
    or all the values in the leaf clusters below it (if it's
not)"""
    if is_leaf(cluster):
        return cluster      # is already a 1-tuple
containing value
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

In order to merge the closest clusters, we need some notion of the distance between clusters. We'll use the *minimum* distance between elements of the two clusters, which merges the two clusters that are closest to touching (but will sometimes produce large chain-like clusters that aren't very tight). If we wanted tight spherical clusters, we might use the *maximum* distance instead, as it merges the two clusters that fit in the smallest ball. Both are common choices, as is the *average* distance:

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """compute all the pairwise distances between cluster1
and cluster2
    and apply _distance_agg_ to the resulting list"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in
get_values(cluster2)])
```

We'll use the merge order slot to track the order in which we did the merging. Smaller numbers will represent *later* merges. This means when we want to unmerge clusters, we do so from lowest merge order to highest. Since leaf clusters were never merged (which means we never want to unmerge them), we'll assign them infinity:

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0]  # merge_order is first element of
2-tuple
```

Now we're ready to create the clustering algorithm:

```
def bottom_up_cluster(inputs, distance_agg=min):
    # start with every input a leaf cluster / 1-tuple
    clusters = [(input,) for input in inputs]

    # as long as we have more than one cluster left...
    while len(clusters) > 1:
        # find the two closest clusters
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                      key=lambda (x, y): cluster_distance(x,
y, distance_agg))

        # remove them from the list of clusters
        clusters = [c for c in clusters if c != c1 and c !=
c2]

        # merge them, using merge_order = # of clusters left
        merged_cluster = (len(clusters), [c1, c2])

        # and add their merge
        clusters.append(merged_cluster)

    # when there's only one cluster left, return it
    return clusters[0]
```

Its use is very simple:

```
base_cluster = bottom_up_cluster(inputs)
```

This produces a cluster whose ugly representation is:

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                     ([20, 23],)]),
               ([26, 13],)]),
          (16, [([11, 15],),
                ([13, 13],)])]),
     (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                               ([-46, 5],)]),
                         ([-41, 8],)]),
                    ([-49, 15],)]),
               ([-34, -1],)]),
          (6, [(7, [(8, [(10, [([-22, -16],),
                               ([-19, -11],)]),
                         ([-25, -9],)]),
                    (13, [(15, [(17, [([-11, -6],),
                                      ([-12, -8],)]),
                                ([-14, -5],)]),
                          ([-18, -3],)])]),
               (12, [([-13, -19],),
                     ([-9, -16],)])])])])])
```

For every merged cluster, I lined up its children vertically. If we say "cluster 0" for the cluster with merge order 0, you can interpret this as:

- Cluster 0 is the merger of cluster 1 and cluster 2.
- Cluster 1 is the merger of cluster 3 and cluster 16.
- Cluster 16 is the merger of the leaf `[11, 15]` and the leaf `[13, 13]`.
- And so on…

Since we had 20 inputs, it took 19 merges to get to this one cluster. The first merge created cluster 18 by combining the leaves `[19, 28]` and `[21, 27]`. And the last merge created cluster 0.

Generally, though, we don't want to be squinting at nasty text representations like this. (Although it could be an interesting exercise to create a user-friendlier visualization of the cluster hierarchy.) Instead let's write a function that generates any number of clusters by performing the appropriate number of unmerges:

```
def generate_clusters(base_cluster, num_clusters):
    # start with a list with just the base cluster
    clusters = [base_cluster]

    # as long as we don't have enough clusters yet...
    while len(clusters) < num_clusters:
        # choose the last-merged of our clusters
        next_cluster = min(clusters, key=get_merge_order)
        # remove it from the list
        clusters = [c for c in clusters if c !=
next_cluster]
        # and add its children to the list (i.e., unmerge
it)
        clusters.extend(get_children(next_cluster))

    # once we have enough clusters...
    return clusters
```

So, for example, if we want to generate three clusters, we can just do:

```
three_clusters = [get_values(cluster)
                  for cluster in
generate_clusters(base_cluster, 3)]
```

which we can easily plot:

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # magic unzipping trick
    plt.scatter(xs, ys, color=color, marker=marker)

    # put a number at the mean of the cluster
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')

plt.title("User Locations -- 3 Bottom-Up Clusters, Min")
plt.xlabel("blocks east of city center")
plt.ylabel("blocks north of city center")
plt.show()
```

This gives very different results than k-means did, as shown in Figure 1-6.

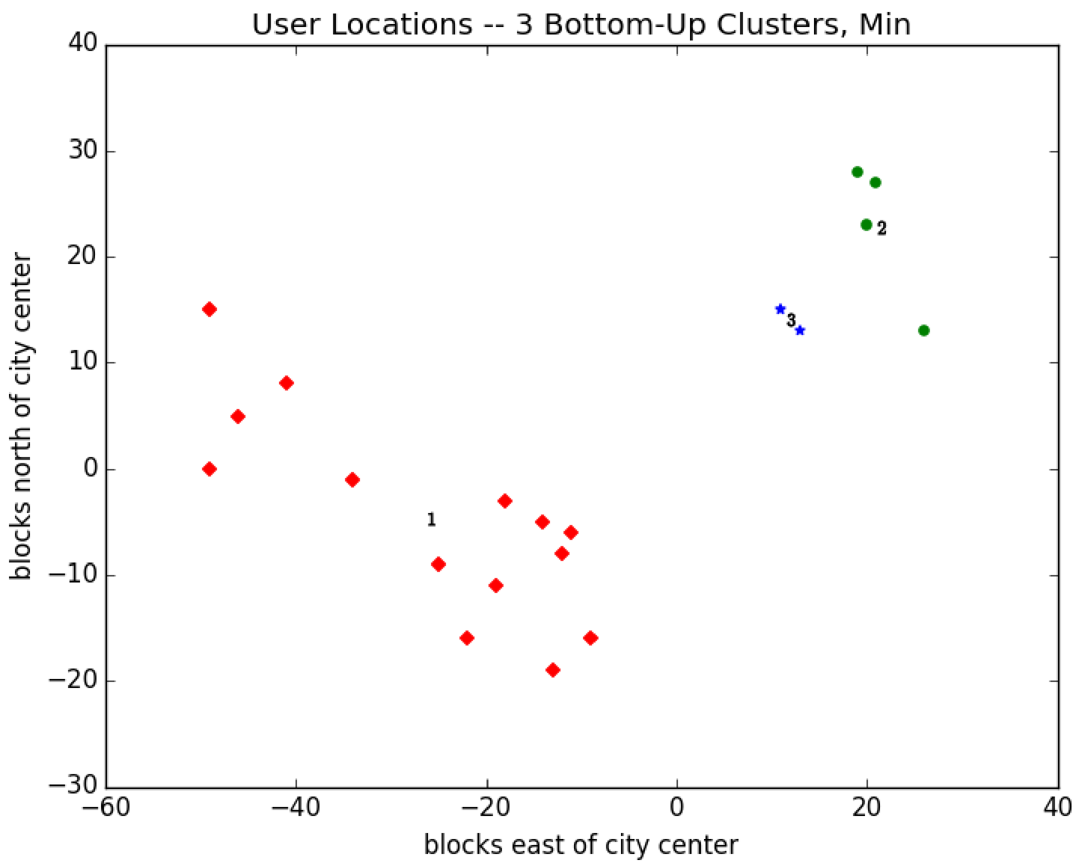

*Figure 1-6. Three bottom-up clusters using min distance*

As we mentioned above, this is because using `min` in `cluster_distance` tends to give chain-like clusters. If we instead use `max` (which gives tight clusters) it looks the same as the 3-means result (Figure 1-7).

NOTE

The `bottom_up_clustering` implementation above is relatively simple, but it’s also shockingly inefficient. In particular, it recomputes the distance between each pair of inputs at every step. A more efficient implementation might precompute the distances between each pair of inputs and then perform a lookup inside cluster_distance. A *really* efficient implementation would likely also remember the `cluster_distance`s from the previous step.

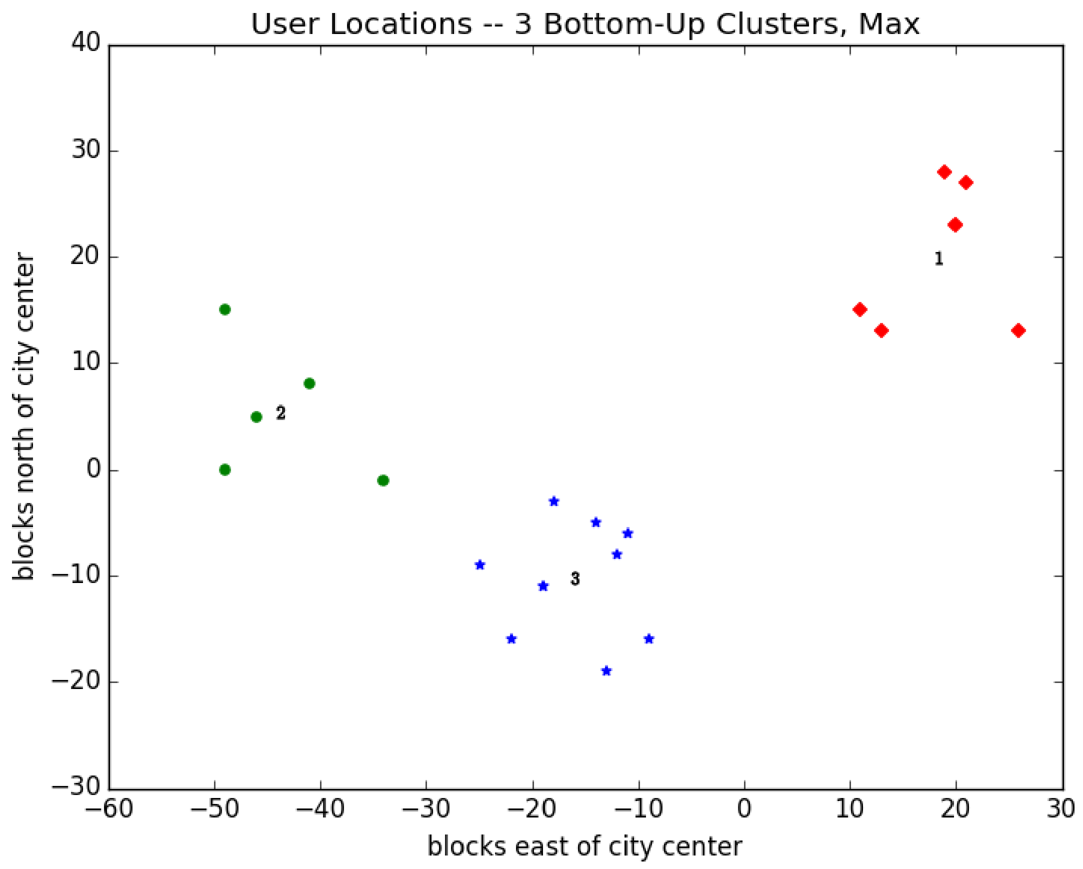

*Figure 1-7. Three bottom-up clusters using max distance*

## For Further Exploration

- `scikit-learn` has an entire module `sklearn.cluster` that contains several clustering algorithms including `KMeans` and the

`Ward` hierarchical clustering algorithm (which uses a different criterion for merging clusters than ours did).

- SciPy has two clustering models `scipy.cluster.vq` (which does k-means) and `scipy.cluster.hierarchy` (which has a variety of hierarchical clustering algorithms).