Chapter 7

Clustering

Clustering is the process of examining a collection of "points," and grouping the points into "clusters" according to some distance measure. The goal is that points in the same cluster have a small distance from one another, while points in different clusters are at a large distance from one another. A suggestion of what clusters might look like was seen in Fig. 1.1. However, there the intent was that there were three clusters around three different road intersections, but two of the clusters blended into one another because they were not sufficiently separated.

Our goal in this chapter is to offer methods for discovering clusters in data. We are particularly interested in situations where the data is very large, and/or where the space either is high-dimensional, or the space is not Euclidean at all. We shall therefore discuss several algorithms that assume the data does not fit in main memory. However, we begin with the basics: the two general approaches to clustering and the methods for dealing with clusters in a non-Euclidean space.

7.1 Introduction to Clustering Techniques

We begin by reviewing the notions of distance measures and spaces. The two major approaches to clustering – hierarchical and point-assignment – are defined. We then turn to a discussion of the "curse of dimensionality," which makes clustering in high-dimensional spaces difficult, but also, as we shall see, enables some simplifications if used correctly in a clustering algorithm.

7.1.1 Points, Spaces, and Distances

A dataset suitable for clustering is a collection of *points*, which are objects belonging to some *space*. In its most general sense, a space is just a universal set of points, from which the points in the dataset are drawn. However, we should be mindful of the common case of a Euclidean space (see Section 3.5.2),

which has a number of important properties useful for clustering. In particular, a Euclidean space's points are vectors of real numbers. The length of the vector is the number of dimensions of the space. The components of the vector are commonly called *coordinates* of the represented points.

All spaces for which we can perform a clustering have a distance measure, giving a distance between any two points in the space. We introduced distances in Section 3.5. The common Euclidean distance (square root of the sums of the squares of the differences between the coordinates of the points in each dimension) serves for all Euclidean spaces, although we also mentioned some other options for distance measures in Euclidean spaces, including the Manhattan distance (sum of the magnitudes of the differences in each dimension) and the L_∞-distance (maximum magnitude of the difference in any dimension).

Figure 7.1: Heights and weights of dogs taken from three varieties

Example 7.1: Classical applications of clustering often involve low-dimensional Euclidean spaces. For example, Fig. 7.1 shows height and weight measurements of dogs of several varieties. Without knowing which dog is of which variety, we can see just by looking at the diagram that the dogs fall into three clusters, and those clusters happen to correspond to three varieties. With small amounts of data, any clustering algorithm will establish the correct clusters, and simply plotting the points and "eyeballing" the plot will suffice as well. □

However, modern clustering problems are not so simple. They may involve Euclidean spaces of very high dimension or spaces that are not Euclidean at all. For example, it is challenging to cluster documents by their topic, based on the occurrence of common, unusual words in the documents. It is challenging to cluster moviegoers by the type or types of movies they like.

We also considered in Section 3.5 distance measures for non-Euclidean spaces. These include the Jaccard distance, cosine distance, Hamming distance,

and edit distance. Recall that the requirements for a function on pairs of points to be a distance measure are that

1. Distances are always nonnegative, and only the distance between a point and itself is 0.
2. Distance is symmetric; it doesn't matter in which order you consider the points when computing their distance.
3. Distance measures obey the triangle inequality; the distance from x to y to z is never less than the distance going from x to z directly.

7.1.2 Clustering Strategies

We can divide (cluster!) clustering algorithms into two groups that follow two fundamentally different strategies.

1. *Hierarchical* or *agglomerative* algorithms start with each point in its own cluster. Clusters are combined based on their "closeness," using one of many possible definitions of "close." Combination stops when further combination leads to clusters that are undesirable for one of several reasons. For example, we may stop when we have a predetermined number of clusters, or we may use a measure of compactness for clusters, and refuse to construct a cluster by combining two smaller clusters if the resulting cluster has points that are spread out over too large a region.
2. The other class of algorithms involve *point assignment*. Points are considered in some order, and each one is assigned to the cluster into which it best fits. This process is normally preceded by a short phase in which initial clusters are estimated. Variations allow occasional combining or splitting of clusters, or may allow points to be unassigned if they are *outliers* (points too far from any of the current clusters).

Algorithms for clustering can also be distinguished by:

(a) Whether the algorithm assumes a Euclidean space, or whether the algorithm works for an arbitrary distance measure. We shall see that a key distinction is that in a Euclidean space it is possible to summarize a collection of points by their *centroid* – the average of the points. In a non-Euclidean space, there is no notion of a centroid, and we are forced to develop another way to summarize clusters.

(b) Whether the algorithm assumes that the data is small enough to fit in main memory, or whether data must reside in secondary memory, primarily. Algorithms for large amounts of data often must take shortcuts, since it is infeasible to look at all pairs of points, for example. It is also necessary to summarize clusters in main memory, since we cannot hold all the points of all the clusters in main memory at the same time.

7.1.3 The Curse of Dimensionality

High-dimensional Euclidean spaces have a number of unintuitive properties that are sometimes referred to as the "curse of dimensionality." Non-Euclidean spaces usually share these anomalies as well. One manifestation of the "curse" is that in high dimensions, almost all pairs of points are equally far away from one another. Another manifestation is that almost any two vectors are almost orthogonal. We shall explore each of these in turn.

The Distribution of Distances in a High-Dimensional Space

Let us consider a d-dimensional Euclidean space. Suppose we choose n random points in the unit cube, i.e., points $[x_1, x_2, \ldots, x_d]$, where each x_i is in the range 0 to 1. If $d = 1$, we are placing random points on a line of length 1. We expect that some pairs of points will be very close, e.g., consecutive points on the line. We also expect that some points will be very far away – those at or near opposite ends of the line. The average distance between a pair of points is $1/3$.[1]

Suppose that d is very large. The Euclidean distance between two random points $[x_1, x_2, \ldots, x_d]$ and $[y_1, y_2, \ldots, y_d]$ is

$$\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$$

Here, each x_i and y_i is a random variable chosen uniformly in the range 0 to 1. Since d is large, we can expect that for some i, $|x_i - y_i|$ will be close to 1. That puts a lower bound of 1 on the distance between almost any two random points. In fact, a more careful argument can put a stronger lower bound on the distance between all but a vanishingly small fraction of the pairs of points. However, the maximum distance between two points is $\sqrt{d}$, and one can argue that all but a vanishingly small fraction of the pairs do not have a distance close to this upper limit. In fact, almost all points will have a distance close to the average distance.

If there are essentially no pairs of points that are close, it is hard to build clusters at all. There is little justification for grouping one pair of points and not another. Of course, the data may not be random, and there may be useful clusters, even in very high-dimensional spaces. However, the argument about random data suggests that it will be hard to find these clusters among so many pairs that are all at approximately the same distance.

Angles Between Vectors

Suppose again that we have three random points A, B, and C in a d-dimensional space, where d is large. Here, we do not assume points are in the unit cube;

[1] You can prove this fact by evaluating a double integral, but we shall not do the math here, as it is not central to the discussion.

they can be anywhere in the space. What is angle ABC? We may assume that A is the point $[x_1, x_2, \ldots, x_d]$ and C is the point $[y_1, y_2, \ldots, y_d]$, while B is the origin. Recall from Section 3.5.4 that the cosine of the angle ABC is the dot product of A and C divided by the product of the lengths of the vectors A and C. That is, the cosine is

$$\frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2}\sqrt{\sum_{i=1}^{d} y_i^2}}$$

As d grows, the denominator grows linearly in d, but the numerator is a sum of random values, which are as likely to be positive as negative. Thus, the expected value of the numerator is 0, and as d grows, its standard deviation grows only as $\sqrt{d}$. Thus, for large d, the cosine of the angle between any two vectors is almost certain to be close to 0, which means the angle is close to 90 degrees.

An important consequence of random vectors being orthogonal is that if we have three random points A, B, and C, and we know the distance from A to B is d_1, while the distance from B to C is d_2, we can assume the distance from A to C is approximately $\sqrt{d_1^2 + d_2^2}$. That rule does not hold, even approximately, if the number of dimensions is small. As an extreme case, if $d = 1$, then the distance from A to C would necessarily be $d_1 + d_2$ if A and C were on opposite sides of B, or $|d_1 - d_2|$ if they were on the same side.

7.1.4 Exercises for Section 7.1

! **Exercise 7.1.1:** Prove that if you choose two points uniformly and independently on a line of length 1, then the expected distance between the points is 1/3.

!! **Exercise 7.1.2:** If you choose two points uniformly in the unit square, what is their expected Euclidean distance?

! **Exercise 7.1.3:** Suppose we have a d-dimensional Euclidean space. Consider vectors whose components are only $+1$ or -1 in each dimension. Note that each vector has length $\sqrt{d}$, so the product of their lengths (denominator in the formula for the cosine of the angle between them) is d. If we chose each component independently, and a component is as likely to be $+1$ as -1, what is the distribution of the value of the numerator of the formula (i.e., the sum of the products of the corresponding components from each vector)? What can you say about the expected value of the cosine of the angle between the vectors, as d grows large?

7.2 Hierarchical Clustering

We begin by considering hierarchical clustering in a Euclidean space. This algorithm can only be used for relatively small datasets, but even so, there

are some efficiencies we can make by careful implementation. When the space is non-Euclidean, there are additional problems associated with hierarchical clustering. We therefore consider "clustroids" and the way we can represent a cluster when there is no centroid or average point in a cluster.

7.2.1 Hierarchical Clustering in a Euclidean Space

Any hierarchical clustering algorithm works as follows. We begin with every point in its own cluster. As time goes on, larger clusters will be constructed by combining two smaller clusters, and we have to decide in advance:

1. How will clusters be represented?
2. How will we choose which two clusters to merge?
3. When will we stop combining clusters?

Once we have answers to these questions, the algorithm can be described succinctly as:

```
WHILE it is not time to stop DO
    pick the best two clusters to merge;
    combine those two clusters into one cluster;
END;
```

To begin, we shall assume the space is Euclidean. That allows us to represent a cluster by its centroid or average of the points in the cluster. Note that in a cluster of one point, that point is the centroid, so we can initialize the clusters straightforwardly. We can then use the merging rule that the distance between any two clusters is the Euclidean distance between their centroids, and we should pick the two clusters at the shortest distance. Other ways to define intercluster distance are possible, and we can also pick the best pair of clusters on a basis other than their distance. We shall discuss some options in Section 7.2.3.

Example 7.2: Let us see how the basic hierarchical clustering would work on the data of Fig. 7.2. These points live in a 2-dimensional Euclidean space, and each point is named by its (x, y) coordinates. Initially, each point is in a cluster by itself and is the centroid of that cluster. Among all the pairs of points, there are two pairs that are closest: (10,5) and (11,4) or (11,4) and (12,3). Each is at distance $\sqrt{2}$. Let us break ties arbitrarily and decide to combine (11,4) with (12,3). The result is shown in Fig. 7.3, including the centroid of the new cluster, which is at (11.5, 3.5).

You might think that (10,5) gets combined with the new cluster next, since it is so close to (11,4). But our distance rule requires us to compare only cluster centroids, and the distance from (10,5) to the centroid of the new cluster is $1.5\sqrt{2}$, which is slightly greater than 2. Thus, now the two closest clusters are

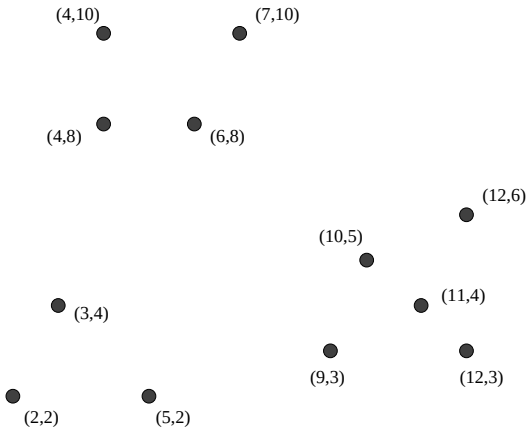

Figure 7.2: Twelve points to be clustered hierarchically

those of the points (4,8) and (4,10). We combine them into one cluster with centroid (4,9).

At this point, the two closest centroids are (10,5) and (11.5, 3.5), so we combine these two clusters. The result is a cluster of three points (10,5), (11,4), and (12,3). The centroid of this cluster is (11,4), which happens to be one of the points of the cluster, but that situation is coincidental. The state of the clusters is shown in Fig. 7.4.

Now, there are several pairs of centroids that are at distance $\sqrt{5}$, and these are the closest centroids. We show in Fig. 7.5 the result of picking three of these:

1. (6,8) is combined with the cluster of two elements having centroid (4,9).
2. (2,2) is combined with (3,4).
3. (9,3) is combined with the cluster of three elements having centroid (11,4).

We can proceed to combine clusters further. We shall discuss alternative stopping rules next. □

There are several approaches we might use to stopping the clustering process.

1. We could be told, or have a belief, about how many clusters there are in the data. For example, if we are told that the data about dogs is taken from Chihuahuas, Dachshunds, and Beagles, then we know to stop when there are three clusters left.
2. We could stop combining when at some point the best combination of existing clusters produces a cluster that is inadequate. We shall discuss

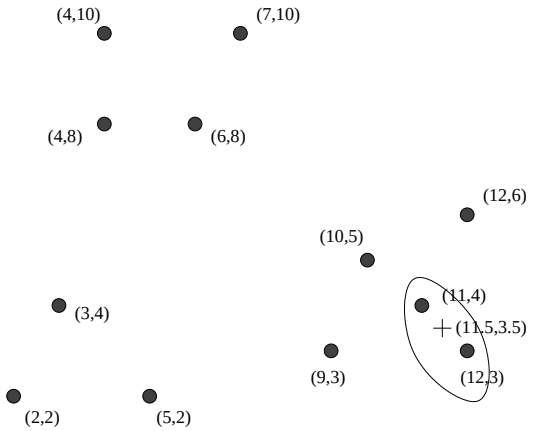

Figure 7.3: Combining the first two points into a cluster

various tests for the adequacy of a cluster in Section 7.2.3, but for an example, we could insist that any cluster have an average distance between the centroid and its points no greater than some limit. This approach is only sensible if we have a reason to believe that no cluster extends over too much of the space.

3. We could continue clustering until there is only one cluster. However, it is meaningless to return a single cluster consisting of all the points. Rather, we return the tree representing the way in which all the points were combined. This form of answer makes good sense in some applications, such as one in which the points are genomes of different species, and the distance measure reflects the difference in the genome.[2] Then, the tree represents the evolution of these species, that is, the likely order in which two species branched from a common ancestor.

Example 7.3: If we complete the clustering of the data of Fig. 7.2, the tree describing how clusters were grouped is the tree shown in Fig. 7.6. □

7.2.2 Efficiency of Hierarchical Clustering

The basic algorithm for hierarchical clustering is not very efficient. At each step, we must compute the distances between each pair of clusters, in order to find the best merger. The initial step takes $O(n^2)$ time, but subsequent steps take time proportional to $(n-1)^2, (n-2)^2, \ldots$. The sum of squares up to n is $O(n^3)$, so this algorithm is cubic. Thus, it cannot be run except for fairly small numbers of points.

[2]This space would not be Euclidean, of course, but the principles regarding hierarchical clustering carry over, with some modifications, to non-Euclidean clustering.

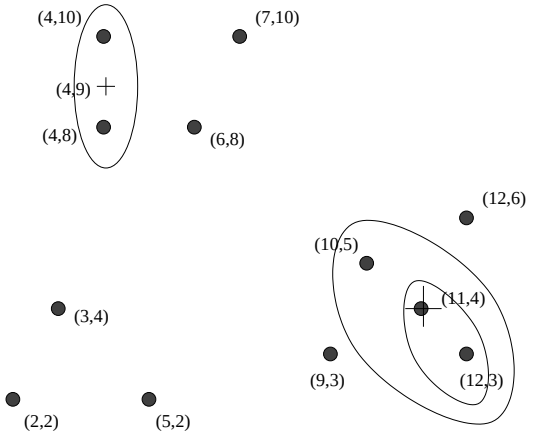

Figure 7.4: Clustering after two additional steps

However, there is a somewhat more efficient implementation of which we should be aware.

1. We start, as we must, by computing the distances between all pairs of points, and this step is $O(n^2)$.

2. Form the pairs and their distances into a priority queue, so we can always find the smallest distance in one step. This operation is also $O(n^2)$.

3. When we decide to merge two clusters C and D, we remove all entries in the priority queue involving one of these two clusters; that requires work $O(n \log n)$ since there are at most $2n$ deletions to be performed, and priority-queue deletion can be performed in $O(\log n)$ time.

4. We then compute all the distances between the new cluster and the remaining clusters. This work is also $O(n \log n)$, as there are at most n entries to be inserted into the priority queue, and insertion into a priority queue can also be done in $O(\log n)$ time.

Since the last two steps are executed at most n times, and the first two steps are executed only once, the overall running time of this algorithm is $O(n^2 \log n)$. That is better than $O(n^3)$, but it still puts a strong limit on how large n can be before it becomes infeasible to use this clustering approach.

7.2.3 Alternative Rules for Controlling Hierarchical Clustering

We have seen one rule for picking the best clusters to merge: find the pair with the smallest distance between their centroids. Some other options are:

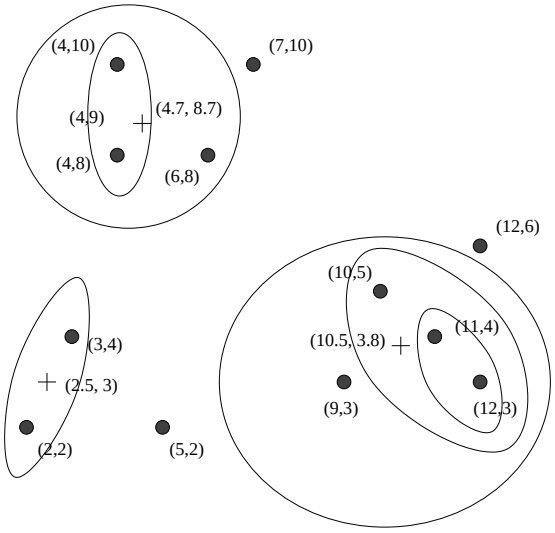

Figure 7.5: Three more steps of the hierarchical clustering

1. Take the distance between two clusters to be the minimum of the distances between any two points, one chosen from each cluster. For example, in Fig. 7.3 we would next chose to cluster the point (10,5) with the cluster of two points, since (10,5) has distance $\sqrt{2}$, and no other pair of unclustered points is that close. Note that in Example 7.2, we did make this combination eventually, but not until we had combined another pair of points. In general, it is possible that this rule will result in an entirely different clustering from that obtained using the distance-of-centroids rule.

2. Take the distance between two clusters to be the average distance of all pairs of points, one from each cluster.

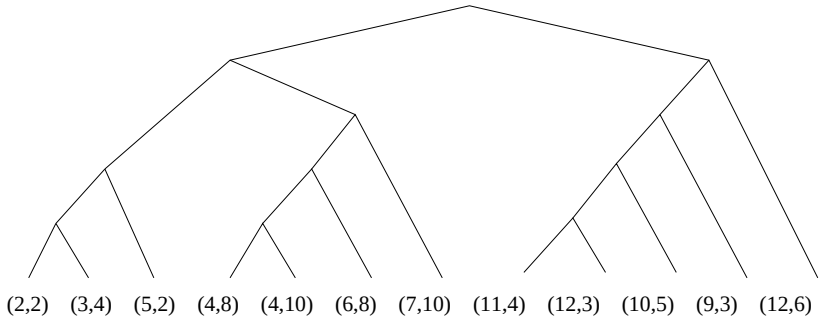

Figure 7.6: Tree showing the complete grouping of the points of Fig. 7.2

3. The *radius* of a cluster is the maximum distance between all the points and the centroid. Combine the two clusters whose resulting cluster has the lowest radius. A slight modification is to combine the clusters whose result has the lowest average distance between a point and the centroid. Another modification is to use the sum of the squares of the distances between the points and the centroid. In some algorithms, we shall find these variant definitions of "radius" referred to as "the radius."

4. The *diameter* of a cluster is the maximum distance between any two points of the cluster. Note that the radius and diameter of a cluster are not related directly, as they are in a circle, but there is a tendency for them to be proportional. We may choose to merge those clusters whose resulting cluster has the smallest diameter. Variants of this rule, analogous to the rule for radius, are possible.

Example 7.4: Let us consider the cluster consisting of the five points at the right of Fig. 7.2. The centroid of these five points is (10.8, 4.2). There is a tie for the two furthest points from the centroid: (9,3) and (12,6), both at distance $\sqrt{4.68} = 2.16$. Thus, the radius is 2.16. For the diameter, we find the two points in the cluster having the greatest distance. These are again (9,3) and (12,6). Their distance is $\sqrt{18} = 4.24$, so that is the diameter. Notice that the diameter is not exactly twice the radius, although it is close in this case. The reason is that the centroid is not on the line between (9,3) and (12,6). □

We also have options in determining when to stop the merging process. We already mentioned "stop when we have k clusters" for some predetermined k. Here are some other options.

1. Stop if the diameter of the cluster that results from the best merger exceeds a threshold. We can also base this rule on the radius, or on any of the variants of the radius mentioned above.

2. Stop if the *density* of the cluster that results from the best merger is below some threshold. The density can be defined in many different ways. Roughly, it should be the number of cluster points per unit volume of the cluster. That ratio can be estimated by the number of points divided by some power of the diameter or radius of the cluster. The correct power could be the number of dimensions of the space. Sometimes, 1 or 2 is chosen as the power, regardless of the number of dimensions.

3. Stop when there is evidence that the next pair of clusters to be combined yields a bad cluster. For example, we could track the average diameter of all the current clusters. As long as we are combining points that truly belong in a cluster, this average will rise gradually. However, if we combine two clusters that really don't deserve to be combined, then the average diameter will take a sudden jump.

Example 7.5: Let us reconsider Fig. 7.2. It has three natural clusters. We computed the diameter of the largest – the five points at the right – in Example 7.4; it is 4.24. The diameter of the 3-node cluster at the lower left is 3, the distance between (2,2) and (5,2). The diameter of the 4-node cluster at the upper left is $\sqrt{13} = 3.61$. The average diameter, 3.62, was reached starting from 0 after nine mergers, so the rise is evidently slow: about 0.4 per merger.

If we are forced to merge two of these natural clusters, the best we can do is merge the two at the left. The diameter of this cluster is $\sqrt{89} = 9.43$; that is the distance between the two points (2,2) and (7,10). Now, the average of the diameters is $(9.43 + 4.24)/2 = 6.84$. This average has jumped almost as much in one step as in all nine previous steps. That comparison indicates that the last merger was inadvisable, and we should roll it back and stop. □

7.2.4 Hierarchical Clustering in Non-Euclidean Spaces

When the space is non-Euclidean, we need to use some distance measure that is computed from points, such as Jaccard, cosine, or edit distance. That is, we cannot base distances on "location" of points. The algorithm of Section 7.2.1 requires distances between points to be computed, but presumably we have a way to compute those distances. A problem arises when we need to represent a cluster, because we cannot replace a collection of points by their centroid.

Example 7.6: The problem arises for any of the non-Euclidean distances we have discussed, but to be concrete, suppose we are using edit distance, and we decide to merge the strings `abcd` and `aecdb`. These have edit distance 3 and might well be merged. However, there is no string that represents their average, or that could be thought of as lying naturally between them. We could take one of the strings that we might pass through when transforming one string to the other by single insertions or deletions, such as `aebcd`, but there are many such options. Moreover, when clusters are formed from more than two strings, the notion of "on the path between" stops making sense. □

Given that we cannot combine points in a cluster when the space is non-Euclidean, our only choice is to pick one of the points of the cluster itself to represent the cluster. Ideally, this point is close to all the points of the cluster, so it in some sense lies in the "center." We call the representative point the *clustroid*. We can select the clustroid in various ways, each designed to, in some sense, minimize the distances between the clustroid and the other points in the cluster. Common choices include selecting as the clustroid the point that minimizes:

1. The sum of the distances to the other points in the cluster.
2. The maximum distance to another point in the cluster.
3. The sum of the squares of the distances to the other points in the cluster.

Example 7.7: Suppose we are using edit distance, and a cluster consists of the four points `abcd`, `aecdb`, `abecb`, and `ecdab`. Their distances are found in the following table:

	`ecdab`	`abecb`	`aecdb`
`abcd`	5	3	3
`aecdb`	2	2	
`abecb`	4		

If we apply the three criteria for being the centroid to each of the four points of the cluster, we find:

Point	Sum	Max	Sum-Sq
`abcd`	11	5	43
`aecdb`	7	3	17
`abecb`	9	4	29
`ecdab`	11	5	45

We can see from these measurements that whichever of the three criteria we choose, `aecdb` will be selected as the clustroid. In general, different criteria could yield different clustroids. □

The options for measuring the distance between clusters that were outlined in Section 7.2.3 can be applied in a non-Euclidean setting, provided we use the clustroid in place of the centroid. For example, we can merge the two clusters whose clustroids are closest. We could also use the average or minimum distance between all pairs of points from the clusters.

Other suggested criteria involved measuring the density of a cluster, based on the radius or diameter. Both these notions make sense in the non-Euclidean environment. The diameter is still the maximum distance between any two points in the cluster. The radius can be defined using the clustroid in place of the centroid. Moreover, it makes sense to use the same sort of evaluation for the radius as we used to select the clustroid in the first place. For example, if we take the clustroid to be the point with the smallest sum of squares of distances to the other nodes, then define the radius to be that sum of squares (or its square root).

Finally, Section 7.2.3 also discussed criteria for stopping the merging of clusters. None of these criteria made direct use of the centroid, except through the notion of radius, and we have already observed that "radius" makes good sense in non-Euclidean spaces. Thus, there is no substantial change in the options for stopping criteria when we move from Euclidean to non-Euclidean spaces.

7.2.5 Exercises for Section 7.2

Exercise 7.2.1: Perform a hierarchical clustering of the one-dimensional set of points 1, 4, 9, 16, 25, 36, 49, 64, 81, assuming clusters are represented by

their centroid (average), and at each step the clusters with the closest centroids are merged.

Exercise 7.2.2: How would the clustering of Example 7.2 change if we used for the distance between two clusters:

(a) The minimum of the distances between any two points, one from each cluster.

(b) The average of the distances between pairs of points, one from each of the two clusters.

Exercise 7.2.3: Repeat the clustering of Example 7.2 if we choose to merge the two clusters whose resulting cluster has:

(a) The smallest radius.

(b) The smallest diameter.

Exercise 7.2.4: Compute the density of each of the three clusters in Fig. 7.2, if "density" is defined to be the number of points divided by

(a) The square of the radius.

(b) The diameter (not squared).

What are the densities, according to (a) and (b), of the clusters that result from the merger of any two of these three clusters. Does the difference in densities suggest the clusters should or should not be merged?

Exercise 7.2.5: We can select clustroids for clusters, even if the space is Euclidean. Consider the three natural clusters in Fig. 7.2, and compute the clustroids of each, assuming the criterion for selecting the clustroid is the point with the minimum sum of distances to the other point in the cluster.

! **Exercise 7.2.6:** Consider the space of strings with edit distance as the distance measure. Give an example of a set of strings such that if we choose the clustroid by minimizing the sum of the distances to the other points we get one point as the clustroid, but if we choose the clustroid by minimizing the maximum distance to the other points, another point becomes the clustroid.

7.3 K-means Algorithms

In this section we begin the study of point-assignment algorithms. The best known family of clustering algorithms of this type is called k-means. They assume a Euclidean space, and they also assume the number of clusters, k, is known in advance. It is, however, possible to deduce k by trial and error. After an introduction to the family of k-means algorithms, we shall focus on a particular algorithm, called BFR after its authors, that enables us to execute k-means on data that is too large to fit in main memory.

7.3.1 K-Means Basics

A k-means algorithm is outlined in Fig. 7.7. There are several ways to select the initial k points that represent the clusters, and we shall discuss them in Section 7.3.2. The heart of the algorithm is the for-loop, in which we consider each point other than the k selected points and assign it to the closest cluster, where "closest" means closest to the centroid of the cluster. Note that the centroid of a cluster can migrate as points are assigned to it. However, since only points near the cluster are likely to be assigned, the centroid tends not to move too much.

```
Initially choose k points that are likely to be in
    different clusters;
Make these points the centroids of their clusters;
FOR each remaining point p DO
    find the centroid to which p is closest;
    Add p to the cluster of that centroid;
    Adjust the centroid of that cluster to account for p;
END;
```

Figure 7.7: Outline of k-means algorithms

An optional step at the end is to fix the centroids of the clusters and to reassign each point, including the k initial points, to the k clusters. Usually, a point p will be assigned to the same cluster in which it was placed on the first pass. However, there are cases where the centroid of p's original cluster moved quite far from p after p was placed there, and p is assigned to a different cluster on the second pass. In fact, even some of the original k points could wind up being reassigned. As these examples are unusual, we shall not dwell on the subject.

7.3.2 Initializing Clusters for K-Means

We want to pick points that have a good chance of lying in different clusters. There are two approaches.

1. Pick points that are as far away from one another as possible.
2. Cluster a sample of the data, perhaps hierarchically, so there are k clusters. Pick a point from each cluster, perhaps that point closest to the centroid of the cluster.

The second approach requires little elaboration. For the first approach, there are variations. One good choice is:

```
Pick the first point at random;
```

```
WHILE there are fewer than k points DO
    Add the point whose minimum distance from the selected
      points is as large as possible;
END;
```

Example 7.8: Let us consider the twelve points of Fig. 7.2, which we reproduce here as Fig. 7.8. In the worst case, our initial choice of a point is near the center, say (6,8). The furthest point from (6,8) is (12,3), so that point is chosen next.

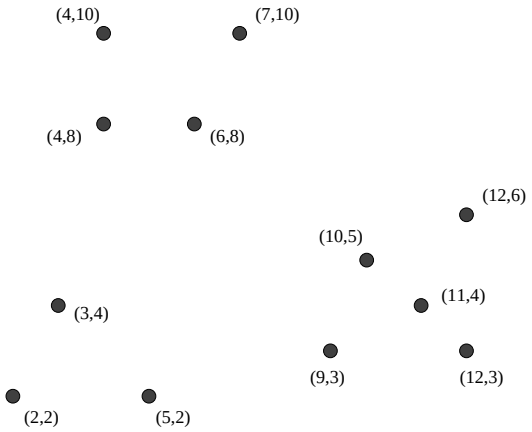

Figure 7.8: Repeat of Fig. 7.2

Among the remaining ten points, the one whose minimum distance to either (6,8) or (12,3) is a maximum is (2,2). That point has distance $\sqrt{52} = 7.21$ from (6,8) and distance $\sqrt{101} = 10.05$ to (12,3); thus its "score" is 7.21. You can check easily that any other point is less than distance 7.21 from at least one of (6,8) and (12,3). Our selection of three starting points is thus (6,8), (12,3), and (2,2). Notice that these three belong to different clusters.

Had we started with a different point, say (10,5), we would get a different set of three initial points. In this case, the starting points would be (10,5), (2,2), and (4,10). Again, these points belong to the three different clusters. □

7.3.3 Picking the Right Value of k

We may not know the correct value of k to use in a k-means clustering. However, if we can measure the quality of the clustering for various values of k, we can usually guess what the right value of k is. Recall the discussion in Section 7.2.3, especially Example 7.5, where we observed that if we take a measure of appropriateness for clusters, such as average radius or diameter, that value will grow slowly, as long as the number of clusters we assume remains at or above the true number of clusters. However, as soon as we try to form fewer

clusters than there really are, the measure will rise precipitously. The idea is expressed by the diagram of Fig. 7.9.

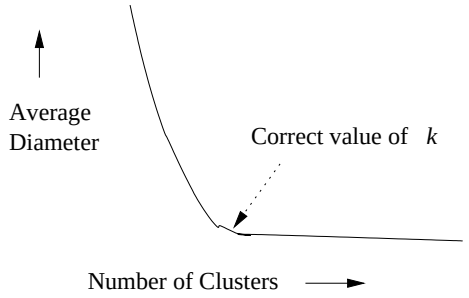

Figure 7.9: Average diameter or another measure of diffuseness rises quickly as soon as the number of clusters falls below the true number present in the data

If we have no idea what the correct value of k is, we can find a good value in a number of clustering operations that grows only logarithmically with the true number. Begin by running the k-means algorithm for $k = 1, 2, 4, 8, \ldots$. Eventually, you will find two values v and $2v$ between which there is very little decrease in the average diameter, or whatever measure of cluster cohesion you are using. We may conclude that the value of k that is justified by the data lies between $v/2$ and v. If you use a binary search (discussed below) in that range, you can find the best value for k in another $\log_2 v$ clustering operations, for a total of $2\log_2 v$ clusterings. Since the true value of k is at least $v/2$, we have used a number of clusterings that is logarithmic in k.

Since the notion of "not much change" is imprecise, we cannot say exactly how much change is too much. However, the binary search can be conducted as follows, assuming the notion of "not much change" is made precise by some formula. We know that there is too much change between $v/2$ and v, or else we would not have gone on to run a clustering for $2v$ clusters. Suppose at some point we have narrowed the range of k to between x and y. Let $z = (x + y)/2$. Run a clustering with z as the target number of clusters. If there is not too much change between z and y, then the true value of k lies between x and z. So recursively narrow that range to find the correct value of k. On the other hand, if there is too much change between z and y, then use binary search in the range between z and y instead.

7.3.4 The Algorithm of Bradley, Fayyad, and Reina

This algorithm, which we shall refer to as *BFR* after its authors, is a variant of k-means that is designed to cluster data in a high-dimensional Euclidean space. It makes a very strong assumption about the shape of clusters: they must be normally distributed about a centroid. The mean and standard deviation for a cluster may differ for different dimensions, but the dimensions must be

independent. For instance, in two dimensions a cluster may be cigar-shaped, but the cigar must not be rotated off of the axes. Figure 7.10 makes the point.

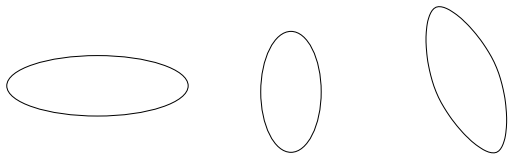

Figure 7.10: The clusters in data for which the BFR algorithm may be used can have standard deviations that differ along different axes, but the axes of the cluster must align with the axes of the space

The BFR Algorithm begins by selecting k points, using one of the methods discussed in Section 7.3.2. Then, the points of the data file are read in chunks. These might be chunks from a distributed file system or a conventional file might be partitioned into chunks of the appropriate size. Each chunk must consist of few enough points that they can be processed in main memory. Also stored in main memory are summaries of the k clusters and some other data, so the entire memory is not available to store a chunk. The main-memory data other than the chunk from the input consists of three types of objects:

1. *The Discard Set*: These are simple summaries of the clusters themselves. We shall address the form of cluster summarization shortly. Note that the cluster summaries are not "discarded"; they are in fact essential. However, the points that the summary represents *are* discarded and have no representation in main memory other than through this summary.

2. *The Compressed Set*: These are summaries, similar to the cluster summaries, but for sets of points that have been found close to one another, but not close to any cluster. The points represented by the compressed set are also discarded, in the sense that they do not appear explicitly in main memory. We call the represented sets of points *miniclusters*.

3. *The Retained Set*: Certain points can neither be assigned to a cluster nor are they sufficiently close to any other points that we can represent them by a compressed set. These points are held in main memory exactly as they appear in the input file.

The picture in Fig. 7.11 suggests how the points processed so far are represented.

The discard and compressed sets are represented by $2d + 1$ values, if the data is d-dimensional. These numbers are:

(a) The number of points represented, N.

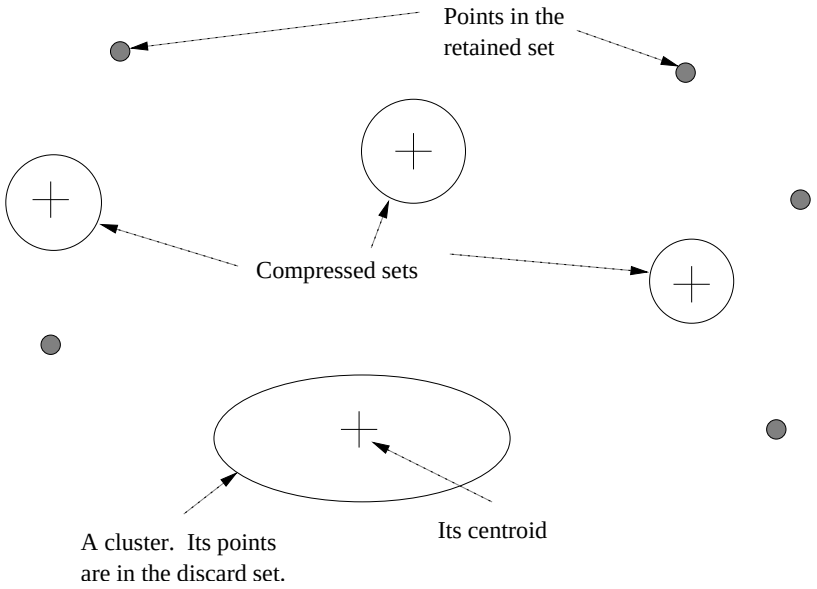

Figure 7.11: Points in the discard, compressed, and retained sets

(b) The sum of the components of all the points in each dimension. This data is a vector SUM of length d, and the component in the ith dimension is SUM_i.

(c) The sum of the squares of the components of all the points in each dimension. This data is a vector SUMSQ of length d, and its component in the ith dimension is SUMSQ_i.

Our real goal is to represent a set of points by their count, their centroid and the standard deviation in each dimension. However, these $2d+1$ values give us those statistics. N is the count. The centroid's coordinate in the ith dimension is the SUM_i/N, that is the sum in that dimension divided by the number of points. The variance in the ith dimension is $\text{SUMSQ}_i/N - (\text{SUM}_i/N)^2$. We can compute the standard deviation in each dimension, since it is the square root of the variance.

Example 7.9: Suppose a cluster consists of the points $(5,1)$, $(6,-2)$, and $(7,0)$. Then $N = 3$, SUM $= [18,-1]$, and SUMSQ $= [110,5]$. The centroid is SUM$/N$, or $[6,-1/3]$. The variance in the first dimension is $110/3 - (18/3)^2 = 0.667$, so the standard deviation is $\sqrt{0.667} = 0.816$. In the second dimension, the variance is $5/3 - (-1/3)^2 = 1.56$, so the standard deviation is 1.25. □

7.3.5 Processing Data in the BFR Algorithm

We shall now outline what happens when we process a chunk of points.

Benefits of the N, SUM, SUMSQ Representation

There is a significant advantage to representing sets of points as it is done in the BFR Algorithm, rather than by storing N, the centroid, and the standard deviation in each dimension. Consider what we need to do when we add a new point to a cluster. N is increased by 1, of course. But we can also add the vector representing the location of the point to SUM to get the new SUM, and we can add the squares of the components of the vector to SUMSQ to get the new SUMSQ. Had we used the centroid in place of SUM, then we could not adjust the centroid to account for the new point without doing some calculation involving N, and the recomputation of the standard deviations would be far more complex as well. Similarly, if we want to combine two sets, we just add corresponding values of N, SUM, and SUMSQ, while if we used the centroid and standard deviations as a representation, the calculation would be far more complex.

1. First, all points that are sufficiently close to the centroid of a cluster are added to that cluster. As described in the box on benefits, it is simple to add the information about the point to the N, SUM, and SUMSQ that represent the cluster. We then discard the point. The question of what "sufficiently close" means will be addressed shortly.

2. For the points that are not sufficiently close to any centroid, we cluster them, along with the points in the retained set. Any main-memory clustering algorithm can be used, such as the hierarchical methods discussed in Section 7.2. We must use some criterion for deciding when it is reasonable to combine two points into a cluster or two clusters into one. Section 7.2.3 covered the ways we might make this decision. Clusters of more than one point are summarized and added to the compressed set. Singleton clusters become the retained set of points.

3. We now have miniclusters derived from our attempt to cluster new points and the old retained set, and we have the miniclusters from the old compressed set. Although none of these miniclusters can be merged with one of the k clusters, they might merge with one another. The criterion for merger may again be chosen according to the discussion in Section 7.2.3. Note that the form of representation for compressed sets (N, SUM, and SUMSQ) makes it easy to compute statistics such as the variance for the combination of two miniclusters that we consider merging.

4. Points that are assigned to a cluster or a minicluster, i.e., those that are not in the retained set, are written out, with their assignment, to secondary memory.

Finally, if this is the last chunk of input data, we need to do something with the compressed and retained sets. We can treat them as outliers, and never cluster them at all. Or, we can assign each point in the retained set to the cluster of the nearest centroid. We can combine each minicluster with the cluster whose centroid is closest to the centroid of the minicluster.

An important decision that must be examined is how we decide whether a new point p is close enough to one of the k clusters that it makes sense to add p to the cluster. Two approaches have been suggested.

(a) Add p to a cluster if it not only has the centroid closest to p, but it is very unlikely that, after all the points have been processed, some other cluster centroid will be found to be nearer to p. This decision is a complex statistical calculation. It must assume that points are ordered randomly, and that we know how many points will be processed in the future. Its advantage is that if we find one centroid to be significantly closer to p than any other, we can add p to that cluster and be done with it, even if p is very far from all centroids.

(b) We can measure the probability that, if p belongs to a cluster, it would be found as far as it is from the centroid of that cluster. This calculation makes use of the fact that we believe each cluster to consist of normally distributed points with the axes of the distribution aligned with the axes of the space. It allows us to make the calculation through the *Mahalanobis distance* of the point, which we shall describe next.

The Mahalanobis distance is essentially the distance between a point and the centroid of a cluster, normalized by the standard deviation of the cluster in each dimension. Since the BFR Algorithm assumes the axes of the cluster align with the axes of the space, the computation of Mahalanobis distance is especially simple. Let $p = [p_1, p_2, \ldots, p_d]$ be a point and $c = [c_1, c_2, \ldots, c_d]$ the centroid of a cluster. Let σ_i be the standard deviation of points in the cluster in the ith dimension. Then the Mahalanobis distance between p and c is

$$\sqrt{\sum_{i=1}^{d} \left(\frac{p_i - c_i}{\sigma_i}\right)^2}$$

That is, we normalize the difference between p and c in the ith dimension by dividing by the standard deviation of the cluster in that dimension. The rest of the formula combines the normalized distances in each dimension in the normal way for a Euclidean space.

To assign point p to a cluster, we compute the Mahalanobis distance between p and each of the cluster centroids. We choose that cluster whose centroid has the least Mahalanobis distance, and we add p to that cluster provided the Mahalanobis distance is less than a threshold. For instance, suppose we pick four as the threshold. If data is normally distributed, then the probability of

a value as far as four standard deviations from the mean is less than one in a million. Thus, if the points in the cluster are really normally distributed, then the probability that we will fail to include a point that truly belongs is less than 10^{-6}. And such a point is likely to be assigned to that cluster eventually anyway, as long as it does not wind up closer to some other centroid as centroids migrate in response to points added to their cluster.

7.3.6 Exercises for Section 7.3

Exercise 7.3.1: For the points of Fig. 7.8, if we select three starting points using the method of Section 7.3.2, and the first point we choose is (3,4), which other points are selected.

!! **Exercise 7.3.2:** Prove that no matter what point we start with in Fig. 7.8, if we select three starting points by the method of Section 7.3.2 we obtain points in each of the three clusters. *Hint*: You could solve this exhaustively by begining with each of the twelve points in turn. However, a more generally applicable solution is to consider the diameters of the three clusters and also consider the *minimum intercluster distance*, that is, the minimum distance between two points chosen from two different clusters. Can you prove a general theorem based on these two parameters of a set of points?

! **Exercise 7.3.3:** Give an example of a dataset and a selection of k initial centroids such that when the points are reassigned to their nearest centroid at the end, at least one of the initial k points is reassigned to a different cluster.

Exercise 7.3.4: For the three clusters of Fig. 7.8:

(a) Compute the representation of the cluster as in the BFR Algorithm. That is, compute N, SUM, and SUMSQ.

(b) Compute the variance and standard deviation of each cluster in each of the two dimensions.

Exercise 7.3.5: Suppose a cluster of three-dimensional points has standard deviations of 2, 3, and 5, in the three dimensions, in that order. Compute the Mahalanobis distance between the origin $(0, 0, 0)$ and the point $(1, -3, 4)$.

7.4 The CURE Algorithm

We now turn to another large-scale-clustering algorithm in the point-assignment class. This algorithm, called *CURE* (Clustering Using REpresentatives), assumes a Euclidean space. However, it does not assume anything about the shape of clusters; they need not be normally distributed, and can even have strange bends, S-shapes, or even rings. Instead of representing clusters by their centroid, it uses a collection of representative points, as the name implies.

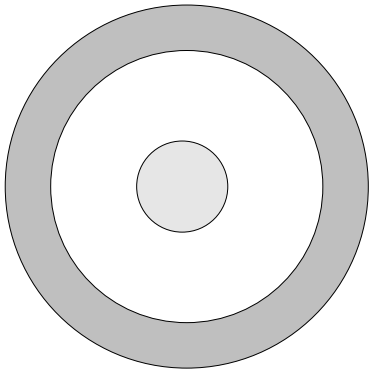

Figure 7.12: Two clusters, one surrounding the other

Example 7.10 : Figure 7.12 is an illustration of two clusters. The inner cluster is an ordinary circle, while the second is a ring around the circle. This arrangement is not completely pathological. A creature from another galaxy might look at our solar system and observe that the objects cluster into an inner circle (the planets) and an outer ring (the Kuyper belt), with little in between. □

7.4.1 Initialization in CURE

We begin the CURE algorithm by:

1. Take a small sample of the data and cluster it in main memory. In principle, any clustering method could be used, but as CURE is designed to handle oddly shaped clusters, it is often advisable to use a hierarchical method in which clusters are merged when they have a close pair of points. This issue is discussed in more detail in Example 7.11 below.

2. Select a small set of points from each cluster to be *representative points*. These points should be chosen to be as far from one another as possible, using the method described in Section 7.3.2.

3. Move each of the representative points a fixed fraction of the distance between its location and the centroid of its cluster. Perhaps 20% is a good fraction to choose. Note that this step requires a Euclidean space, since otherwise, there might not be any notion of a line between two points.

Example 7.11 : We could use a hierarchical clustering algorithm on a sample of the data from Fig. 7.12. If we took as the distance between clusters the shortest distance between any pair of points, one from each cluster, then we would correctly find the two clusters. That is, pieces of the ring would stick

together, and pieces of the inner circle would stick together, but pieces of ring would always be far away from the pieces of the circle. Note that if we used the rule that the distance between clusters was the distance between their centroids, then we might not get the intuitively correct result. The reason is that the centroids of both clusters are in the center of the diagram.

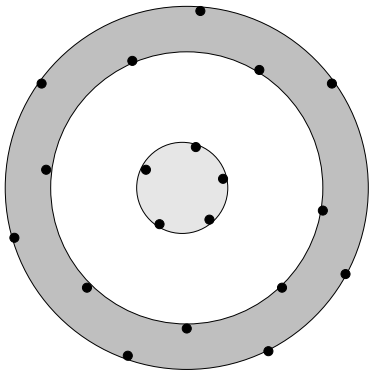

Figure 7.13: Select representative points from each cluster, as far from one another as possible

For the second step, we pick the representative points. If the sample from which the clusters are constructed is large enough, we can count on a cluster's sample points at greatest distance from one another lying on the boundary of the cluster. Figure 7.13 suggests what our initial selection of sample points might look like.

Finally, we move the representative points a fixed fraction of the distance from their true location toward the centroid of the cluster. Note that in Fig. 7.13 both clusters have their centroid in the same place: the center of the inner circle. Thus, the representative points from the circle move inside the cluster, as was intended. Points on the outer edge of the ring also move into their cluster, but points on the ring's inner edge move outside the cluster. The final locations of the representative points from Fig. 7.13 are suggested by Fig. 7.14. □

7.4.2 Completion of the CURE Algorithm

The next phase of CURE is to merge two clusters if they have a pair of representative points, one from each cluster, that are sufficiently close. The user may pick the distance that defines "close." This merging step can repeat, until there are no more sufficiently close clusters.

Example 7.12: The situation of Fig. 7.14 serves as a useful illustration. There is some argument that the ring and circle should really be merged, because their centroids are the same. For instance, if the gap between the ring and circle were

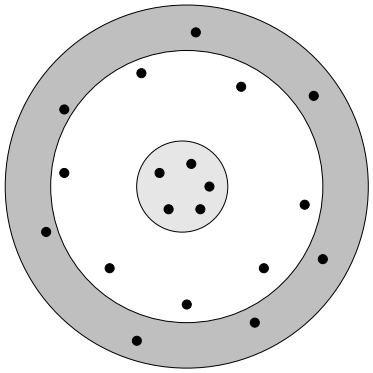

Figure 7.14: Moving the representative points 20% of the distance to the cluster's centroid

much smaller, it might well be argued that combining the points of the ring and circle into a single cluster reflected the true state of affairs. For instance, the rings of Saturn have narrow gaps between them, but it is reasonable to visualize the rings as a single object, rather than several concentric objects. In the case of Fig. 7.14 the choice of

1. The fraction of the distance to the centroid that we move the representative points and
2. The choice of how far apart representative points of two clusters need to be to avoid merger

together determine whether we regard Fig. 7.12 as one cluster or two. □

The last step of CURE is point assignment. Each point p is brought from secondary storage and compared with the representative points. We assign p to the cluster of the representative point that is closest to p.

Example 7.13: In our running example, points within the ring will surely be closer to one of the ring's representative points than to any representative point of the circle. Likewise, points within the circle will surely be closest to a representative point of the circle. An outlier – a point not within the ring or the circle – will be assigned to the ring if it is outside the ring. If the outlier is between the ring and the circle, it will be assigned to one or the other, somewhat favoring the ring because its representative points have been moved toward the circle. □

7.4.3 Exercises for Section 7.4

Exercise 7.4.1: Consider two clusters that are a circle and a surrounding ring, as in the running example of this section. Suppose:

i. The radius of the circle is c.

ii. The inner and outer circles forming the ring have radii i and o, respectively.

iii. All representative points for the two clusters are on the boundaries of the clusters.

iv. Representative points are moved 20% of the distance from their initial position toward the centroid of their cluster.

v. Clusters are merged if, after repositioning, there are representative points from the two clusters at distance d or less.

In terms of d, c, i, and o, under what circumstances will the ring and circle be merged into a single cluster?

7.5 Clustering in Non-Euclidean Spaces

We shall next consider an algorithm that handles non-main-memory data, but does not require a Euclidean space. The algorithm, which we shall refer to as GRGPF for its authors (V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French), takes ideas from both hierarchical and point-assignment approaches. Like CURE, it represents clusters by sample points in main memory. However, it also tries to organize the clusters hierarchically, in a tree, so a new point can be assigned to the appropriate cluster by passing it down the tree. Leaves of the tree hold summaries of some clusters, and interior nodes hold subsets of the information describing the clusters reachable through that node. An attempt is made to group clusters by their distance from one another, so the clusters at a leaf are close, and the clusters reachable from one interior node are relatively close as well.

7.5.1 Representing Clusters in the GRGPF Algorithm

As we assign points to clusters, the clusters can grow large. Most of the points in a cluster are stored on disk, and are not used in guiding the assignment of points, although they can be retrieved. The representation of a cluster in main memory consists of several *features*. Before listing these features, if p is any point in a cluster, let ROWSUM(p) be the sum of the squares of the distances from p to each of the other points in the cluster. Note that, although we are not in a Euclidean space, there is some distance measure d that applies to points, or else it is not possible to cluster points at all. The following features form the *representation* of a cluster.

1. N, the number of points in the cluster.

2. The clustroid of the cluster, which is defined specifically to be the point in the cluster that minimizes the sum of the squares of the distances to the other points; that is, the clustroid is the point in the cluster with the smallest ROWSUM.

3. The rowsum of the clustroid of the cluster.

4. For some chosen constant k, the k points of the cluster that are closest to the clustroid, and their rowsums. These points are part of the representation in case the addition of points to the cluster causes the clustroid to change. The assumption is made that the new clustroid would be one of these k points near the old clustroid.

5. The k points of the cluster that are furthest from the clustroid and their rowsums. These points are part of the representation so that we can consider whether two clusters are close enough to merge. The assumption is made that if two clusters are close, then a pair of points distant from their respective clustroids would be close.

7.5.2 Initializing the Cluster Tree

The clusters are organized into a tree, and the nodes of the tree may be very large, perhaps disk blocks or pages, as would be the case for a B-tree or R-tree, which the cluster-representing tree resembles. Each leaf of the tree holds as many cluster representations as can fit. Note that a cluster representation has a size that does not depend on the number of points in the cluster.

An interior node of the cluster tree holds a sample of the clustroids of the clusters represented by each of its subtrees, along with pointers to the roots of those subtrees. The samples are of fixed size, so the number of children that an interior node may have is independent of its level. Notice that as we go up the tree, the probability that a given cluster's clustroid is part of the sample diminishes.

We initialize the cluster tree by taking a main-memory sample of the dataset and clustering it hierarchically. The result of this clustering is a tree T, but T is not exactly the tree used by the GRGPF Algorithm. Rather, we select from T certain of its nodes that represent clusters of approximately some desired size n. These are the initial clusters for the GRGPF Algorithm, and we place their representations at the leaf of the cluster-representing tree. We then group clusters with a common ancestor in T into interior nodes of the cluster-representing tree, so in some sense, clusters descended from one interior node are as close as possible. In some cases, rebalancing of the cluster-representing tree will be necessary. This process is similar to the reorganization of a B-tree, and we shall not examine this issue in detail.

7.5.3 Adding Points in the GRGPF Algorithm

We now read points from secondary storage and insert each one into the nearest cluster. We start at the root, and look at the samples of clustroids for each of the children of the root. Whichever child has the clustroid closest to the new point p is the node we examine next. When we reach any node in the tree, we look at the sample clustroids for its children and go next to the child with the clustroid closest to p. Note that some of the sample clustroids at a node may have been seen at a higher level, but each level provides more detail about the clusters lying below, so we see many new sample clustroids each time we go a level down the tree.

Finally, we reach a leaf. This leaf has the cluster features for each cluster represented by that leaf, and we pick the cluster whose clustroid is closest to p. We adjust the representation of this cluster to account for the new node p. In particular, we:

1. Add 1 to N.
2. Add the square of the distance between p and each of the nodes q mentioned in the representation to ROWSUM(q). These points q include the clustroid, the k nearest points, and the k furthest points.

We also estimate the rowsum of p, in case p needs to be part of the representation (e.g., it turns out to be one of the k points closest to the clustroid). Note we cannot compute ROWSUM(p) exactly, without going to disk and retrieving all the points of the cluster. The estimate we use is

$$\text{ROWSUM}(p) = \text{ROWSUM}(c) + Nd^2(p, c)$$

where $d(p, c)$ is the distance between p and the clustroid c. Note that N and ROWSUM(c) in this formula are the values of these features before they were adjusted to account for the addition of p.

We might well wonder why this estimate works. In Section 7.1.3 we discussed the "curse of dimensionality," in particular the observation that in a high-dimensional Euclidean space, almost all angles are right angles. Of course the assumption of the GRGPF Algorithm is that the space might not be Euclidean, but typically a non-Euclidean space also suffers from the curse of dimensionality, in that it behaves in many ways like a high-dimensional Euclidean space. If we assume that the angle between p, c, and another point q in the cluster is a right angle, then the Pythagorean theorem tell us that

$$d^2(p, q) = d^2(p, c) + d^2(c, q)$$

If we sum over all q other than c, and then add $d^2(p, c)$ to ROWSUM(p) to account for the fact that the clustroid is one of the points in the cluster, we derive ROWSUM(p) = ROWSUM(c) + $Nd^2(p, c)$.

Now, we must see if the new point p is one of the k closest or furthest points from the clustroid, and if so, p and its rowsum become a cluster feature,

replacing one of the other features – whichever is no longer one of the k closest or furthest. We also need to consider whether the rowsum for one of the k closest points q is now less than ROWSUM(c). That situation could happen if p were closer to one of these points than to the current clustroid. If so, we swap the roles of c and q. Eventually, it is possible that the true clustroid will no longer be one of the original k closest points. We have no way of knowing, since we do not see the other points of the cluster in main memory. However, they are all stored on disk, and can be brought into main memory periodically for a recomputation of the cluster features.

7.5.4 Splitting and Merging Clusters

The GRGPF Algorithm assumes that there is a limit on the radius that a cluster may have. The particular definition used for the radius is $\sqrt{\text{ROWSUM}(c)/N}$, where c is the clustroid of the cluster and N the number of points in the cluster. That is, the radius is the square root of the average square of the distance from the clustroid of the points in the cluster. If a cluster's radius grows too large, it is split into two. The points of that cluster are brought into main memory, and divided into two clusters to minimize the rowsums. The cluster features for both clusters are computed.

As a result, the leaf of the split cluster now has one more cluster to represent. We should manage the cluster tree like a B-tree, so usually, there will be room in a leaf to add one more cluster. However, if not, then the leaf must be split into two leaves. To implement the split, we must add another pointer and more sample clustroids at the parent node. Again, there may be extra space, but if not, then this node too must be split, and we do so to minimize the squares of the distances between the sample clustroids assigned to different nodes. As in a B-tree, this splitting can ripple all the way up to the root, which can then be split if needed.

The worst thing that can happen is that the cluster-representing tree is now too large to fit in main memory. There is only one thing to do: we make it smaller by raising the limit on how large the radius of a cluster can be, and we consider merging pairs of clusters. It is normally sufficient to consider clusters that are "nearby," in the sense that their representatives are at the same leaf or at leaves with a common parent. However, in principle, we can consider merging any two clusters C_1 and C_2 into one cluster C.

To merge clusters, we assume that the clustroid of C will be one of the points that are as far as possible from the clustroid of C_1 or the clustroid of C_2. Suppose we want to compute the rowsum in C for the point p, which is one of the k points in C_1 that are as far as possible from the centroid of C_1. We use the curse-of-dimensionality argument that says all angles are approximately right angles, to justify the following formula.

$$\text{ROWSUM}_C(p) = \text{ROWSUM}_{C_1}(p) + N_{C_2}\big(d^2(p, c_1) + d^2(c_1, c_2)\big) + \text{ROWSUM}_{C_2}(c_2)$$

In the above, we subscript N and ROWSUM by the cluster to which that feature

refers. We use c_1 and c_2 for the clustroids of C_1 and C_2, respectively.

In detail, we compute the sum of the squares of the distances from p to all the nodes in the combined cluster C by beginning with $\text{ROWSUM}_{C_1}(p)$ to get the terms for the points in the same cluster as p. For the N_{C_2} points q in C_2, we consider the path from p to the clustroid of C_1, then to the clustroid of C_2, and finally to q. We assume there is a right angle between the legs from p to c_1 and c_1 to c_2, and another right angle between the shortest path from from p to c_2 and the leg from c_2 to q. We then use the Pythagorean theorem to justify computing the square of the length of the path to each q as the sum of the squares of the three legs.

We must then finish computing the features for the merged cluster. We need to consider all the points in the merged cluster for which we know the rowsum. These are, the centroids of the two clusters, the k points closest to the clustroids for each cluster, and the k points furthest from the clustroids for each cluster, with the exception of the point that was chosen as the new clustroid. We can compute the distances from the new clustroid for each of these $4k + 1$ points. We select the k with the smallest distances as the "close" points and the k with the largest distances as the "far" points. We can then compute the rowsums for the chosen points, using the same formulas above that we used to compute the rowsums for the candidate clustroids.

7.5.5 Exercises for Section 7.5

Exercise 7.5.1: Using the cluster representation of Section 7.5.1, represent the twelve points of Fig. 7.8 as a single cluster. Use parameter $k = 2$ as the number of close and distant points to be included in the representation. *Hint*: Since the distance is Euclidean, we can get the square of the distance between two points by taking the sum of the squares of the differences along the x- and y-axes.

Exercise 7.5.2: Compute the radius, in the sense used by the GRGPF Algorithm (square root of the average square of the distance from the clustroid) for the cluster that is the five points in the lower right of Fig. 7.8. Note that (11,4) is the clustroid.

7.6 Clustering for Streams and Parallelism

In this section, we shall consider briefly how one might cluster a stream. The model we have in mind is one where there is a sliding window (recall Section 4.1.3) of N points, and we can ask for the centroids or clustroids of the best clusters formed from the last m of these points, for any $m \leq N$. We also study a similar approach to clustering a large, fixed set of points using map-reduce on a computing cluster (no pun intended). This section provides only a rough outline to suggest the possibilities, which depend on our assumptions about how clusters evolve in a stream.

7.6.1 The Stream-Computing Model

We assume that each stream element is a point in some space. The sliding window consists of the most recent N points. Our goal is to precluster subsets of the points in the stream, so that we may quickly answer queries of the form "what are the clusters of the last m points?" for any $m \leq N$. There are many variants of this query, depending on what we assume about what constitutes a cluster. For instance, we may use a k-means approach, where we are really asking that the last m points be partitioned into exactly k clusters. Or, we may allow the number of clusters to vary, but use one of the criteria in Section 7.2.3 or 7.2.4 to determine when to stop merging clusters into larger clusters.

We make no restriction regarding the space in which the points of the stream live. It may be a Euclidean space, in which case the answer to the query is the centroids of the selected clusters. The space may be non-Euclidean, in which case the answer is the clustroids of the selected clusters, where any of the definitions for "clustroid" may be used (see Section 7.2.4).

The problem is considerably easier if we assume that all stream elements are chosen with statistics that do not vary along the stream. Then, a sample of the stream is good enough to estimate the clusters, and we can in effect ignore the stream after a while. However, the stream model normally assumes that the statistics of the stream elements varies with time. For example, the centroids of the clusters may migrate slowly as time goes on, or clusters may expand, contract, divide, or merge.

7.6.2 A Stream-Clustering Algorithm

In this section, we shall present a greatly simplified version of an algorithm referred to as BDMO (for the authors, B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan). The true version of the algorithm involves much more complex structures, which are designed to provide performance guarantees in the worst case.

The BDMO Algorithm builds on the methodology for counting ones in a stream that was described in Section 4.6. Here are the key similarities and differences:

- Like that algorithm, the points of the stream are partitioned into, and summarized by, buckets whose sizes are a power of two. Here, the *size* of a bucket is the number of points it represents, rather than the number of stream elements that are 1.

- As before, the sizes of buckets obey the restriction that there are one or two of each size, up to some limit. However, we do not assume that the sequence of allowable bucket sizes starts with 1. Rather, they are required only to form a sequence where each size is twice the previous size, e.g., $3, 6, 12, 24, \ldots$.

- Bucket sizes are again restrained to be nondecreasing as we go back in time. As in Section 4.6, we can conclude that there will be $O(\log N)$ buckets.

- The *contents* of a bucket consists of:

  1. The size of the bucket.
  2. The timestamp of the bucket, that is, the most recent point that contributes to the bucket. As in Section 4.6, timestamps can be recorded modulo N.
  3. A collection of records that represent the clusters into which the points of that bucket have been partitioned. These records contain:
     (a) The number of points in the cluster.
     (b) The centroid or clustroid of the cluster.
     (c) Any other parameters necessary to enable us to merge clusters and maintain approximations to the full set of parameters for the merged cluster. We shall give some examples when we discuss the merger process in Section 7.6.4.

7.6.3 Initializing Buckets

Our smallest bucket size will be p, a power of 2. Thus, every p stream elements, we create a new bucket, with the most recent p points. The timestamp for this bucket is the timestamp of the most recent point in the bucket. We may leave each point in a cluster by itself, or we may perform a clustering of these points according to whatever clustering strategy we have chosen. For instance, if we choose a k-means algorithm, then (assuming $k < p$) we cluster the points into k clusters by some algorithm.

Whatever method we use to cluster initially, we assume it is possible to compute the centroids or clustroids for the clusters and count the points in each cluster. This information becomes part of the record for each cluster. We also compute whatever other parameters for the clusters will be needed in the merging process.

7.6.4 Merging Buckets

Following the strategy from Section 4.6, whenever we create a new bucket, we need to review the sequence of buckets. First, if some bucket has a timestamp that is more than N time units prior to the current time, then nothing of that bucket is in the window, and we may drop it from the list. Second, we may have created three buckets of size p, in which case we must merge the oldest two of the three. The merger may create two buckets of size $2p$, in which case we may have to merge buckets of increasing sizes, recursively, just as in Section 4.6.

To merge two consecutive buckets, we need to do several things:

1. The size of the bucket is twice the sizes of the two buckets being merged.

2. The timestamp for the merged bucket is the timestamp of the more recent of the two consecutive buckets.

3. We must consider whether to merge clusters, and if so, we need to compute the parameters of the merged clusters. We shall elaborate on this part of the algorithm by considering several examples of criteria for merging and ways to estimate the needed parameters.

Example 7.14: Perhaps the simplest case is where we are using a k-means approach in a Euclidean space. We represent clusters by the count of their points and their centroids. Each bucket has exactly k clusters, so we can pick $p = k$, or we can pick p larger than k and cluster the p points into k clusters when we create a bucket initially as in Section 7.6.3. We must find the best matching between the k clusters of the first bucket and the k clusters of the second. Here, "best" means the matching that minimizes the sum of the distances between the centroids of the matched clusters.

Note that we do not consider merging two clusters from the same bucket, because our assumption is that clusters do not evolve too much between consecutive buckets. Thus, we would expect to find in each of two adjacent buckets a representation of each of the k "true" clusters that exist in the stream.

When we decide to merge two clusters, one from each bucket, the number of points in the merged cluster is surely the sum of the numbers of points in the two clusters. The centroid of the merged cluster is the weighted average of the centroids of the two clusters, where the weighting is by the numbers of points in the clusters. That is, if the two clusters have n_1 and n_2 points, respectively, and have centroids $\mathbf{c}_1$ and $\mathbf{c}_2$ (the latter are d-dimensional vectors for some d), then the combined cluster has $n = n_1 + n_2$ points and has centroid

$$\mathbf{c} = \frac{n_1\mathbf{c}_1 + n_2\mathbf{c}_2}{n_1 + n_2}$$

□

Example 7.15: The method of Example 7.14 suffices when the clusters are changing very slowly. Suppose we might expect the cluster centroids to migrate sufficiently quickly that when matching the centroids from two consecutive buckets, we might be faced with an ambiguous situation, where it is not clear which of two clusters best matches a given cluster from the other bucket. One way to protect against such a situation is to create more than k clusters in each bucket, even if we know that, when we query (see Section 7.6.5), we shall have to merge into exactly k clusters. For example, we might choose p to be much larger than k, and, when we merge, only merge clusters when the result is sufficiently coherent according to one of the criteria outlined in Section 7.2.3. Or, we could use a hierarchical strategy, and make the best merges, so as to maintain $p > k$ clusters in each bucket.

Suppose, to be specific, that we want to put a limit on the sum of the distances between all the points of a cluster and its centroid. Then in addition to the count of points and the centroid of a cluster, we can include an estimate of this sum in the record for a cluster. When we initialize a bucket, we can compute the sum exactly. But as we merge clusters, this parameter becomes an estimate only. Suppose we merge two clusters, and want to compute the sum of distances for the merged cluster. Use the notation for centroids and counts in Example 7.14, and in addition, let s_1 and s_2 be the sums for the two clusters. Then we may estimate the radius of the merged cluster to be

$$n_1|\mathbf{c}_1 - \mathbf{c}| + n_2|\mathbf{c}_2 - \mathbf{c}| + s_1 + s_2$$

That is, we estimate the distance between any point x and the new centroid $\mathbf{c}$ to be the distance of that point to its old centroid (these distances sum to $s_1 + s_2$, the last two terms in the above expression) plus the distance from the old centroid to the new (these distances sum to the first two terms of the above expression). Note that this estimate is an upper bound, by the triangle inequality.

An alternative is to replace the sum of distances by the sum of the squares of the distances from the points to the centroid. If these sums for the two clusters are t_1 and t_2, respectively, then we can produce an estimate for the same sum in the new cluster as

$$n_1|\mathbf{c}_1 - \mathbf{c}|^2 + n_2|\mathbf{c}_2 - \mathbf{c}|^2 + t_1 + t_2$$

This estimate is close to correct if the space is high-dimensional, by the "curse of dimensionality." □

Example 7.16: Our third example will assume a non-Euclidean space and no constraint on the number of clusters. We shall borrow several of the techniques from the GRGPF Algorithm of Section 7.5. Specifically, we represent clusters by their clustroid and rowsum (sum of the squares of the distances from each node of the cluster to its clustroid). We include in the record for a cluster information about a set of points at maximum distance from the clustroid, including their distances from the clustroid and their rowsums. Recall that their purpose is to suggest a clustroid when this cluster is merged with another.

When we merge buckets, we may choose one of many ways to decide which clusters to merge. For example, we may consider pairs of clusters in order of the distance between their clustroids. We may also choose to merge clusters when we consider them, provided the sum of their rowsums is below a certain limit. Alternatively, we may perform the merge if the sum of rowsums divided by the number of points in the clusters is below a limit. Any of the other strategies discussed for deciding when to merge clusters may be used as well, provided we arrange to maintain the data (e.g., cluster diameter) necessary to make decisions.

We then must pick a new clustroid, from among the points most distant from the clustroids of the two merged clusters. We can compute rowsums for

each of these candidate clustroids using the formulas given in Section 7.5.4. We also follow the strategy given in that section to pick a subset of the distant points from each cluster to be the set of distant points for the merged cluster, and to compute the new rowsum and distance-to-clustroid for each. □

7.6.5 Answering Queries

Recall that we assume a query is a request for the clusters of the most recent m points in the stream, where $m \leq N$. Because of the strategy we have adopted of combining buckets as we go back in time, we may not be able to find a set of buckets that covers exactly the last m points. However, if we choose the smallest set of buckets that cover the last m points, we shall include in these buckets no more than the last $2m$ points. We shall produce, as answer to the query, the centroids or clustroids of all the points in the selected buckets. In order for the result to be a good approximation to the clusters for exactly the last m points, we must assume that the points between $2m$ and $m+1$ will not have radically different statistics from the most recent m points. However, if the statistics vary too rapidly, recall from Section 4.6.6 that a more complex bucketing scheme can guarantee that we can find buckets to cover at most the last $m(1+\epsilon)$ points, for any $\epsilon > 0$.

Having selected the desired buckets, we pool all their clusters. We then use some methodology for deciding which clusters to merge. Examples 7.14 and 7.16 are illustrations of two approaches to this merger. For instance, if we are required to produce exactly k clusters, then we can merge the clusters with the closest centroids until we are left with only k clusters, as in Example 7.14. Or we can make a decision whether or not to merge clusters in various ways, as we sampled in Example 7.16.

7.6.6 Clustering in a Parallel Environment

Now, let us briefly consider the use of parallelism available in a computing cluster.[3] We assume we are given a very large collection of points, and we wish to exploit parallelism to compute the centroids of their clusters. The simplest approach is to use a map-reduce strategy, but in most cases we are constrained to use a single Reduce task.

Begin by creating many Map tasks. Each task is assigned a subset of the points. The Map function's job is to cluster the points it is given. Its output is a set of key-value pairs with a fixed key 1, and a value that is the description of one cluster. This description can be any of the possibilities suggested in Section 7.6.2, such as the centroid, count, and diameter of the cluster.

Since all key-value pairs have the same key, there can be only one Reduce task. This task gets descriptions of the clusters produced by each of the Map

[3]Do not forget that the term "cluster" has two completely different meanings in this section.

tasks, and must merge them appropriately. We may use the discussion in Section 7.6.4 as representative of the various strategies we might use to produce the final clustering, which is the output of the Reduce task.

7.6.7 Exercises for Section 7.6

Exercise 7.6.1: Execute the BDMO Algorithm with $p = 3$ on the following 1-dimensional, Euclidean data:

$$1, 45, 80, 24, 56, 71, 17, 40, 66, 32, 48, 96, 9, 41, 75, 11, 58, 93, 28, 39, 77$$

The clustering algorithms is k-means with $k = 3$. Only the centroid of a cluster, along with its count, is needed to represent a cluster.

Exercise 7.6.2: Using your clusters from Exercise 7.6.1, produce the best centroids in response to a query asking for a clustering of the last 10 points.

7.7 Summary of Chapter 7

- *Clustering*: Clusters are often a useful summary of data that is in the form of points in some space. To cluster points, we need a distance measure on that space. Ideally, points in the same cluster have small distances between them, while points in different clusters have large distances between them.
- *Clustering Algorithms*: Clustering algorithms generally have one of two forms. Hierarchical clustering algorithms begin with all points in a cluster of their own, and nearby clusters are merged iteratively. Point-assignment clustering algorithms consider points in turn and assign them to the cluster in which they best fit.
- *The Curse of Dimensionality*: Points in high-dimensional Euclidean spaces, as well as points in non-Euclidean spaces often behave unintuitively. Two unexpected properties of these spaces are that random points are almost always at about the same distance, and random vectors are almost always orthogonal.
- *Centroids and Clustroids*: In a Euclidean space, the members of a cluster can be averaged, and this average is called the centroid. In non-Euclidean spaces, there is no guarantee that points have an "average," so we are forced to use one of the members of the cluster as a representative or typical element of the cluster. That representative is called the clustroid.
- *Choosing the Clustroid*: There are many ways we can define a typical point of a cluster in a non-Euclidean space. For example, we could choose the point with the smallest sum of distances to the other points, the smallest sum of the squares of those distances, or the smallest maximum distance to any other point in the cluster.

- *Radius and Diameter*: Whether or not the space is Euclidean, we can define the radius of a cluster to be the maximum distance from the centroid or clustroid to any point in that cluster. We can define the diameter of the cluster to be the maximum distance between any two points in the cluster. Alternative definitions, especially of the radius, are also known, for example, average distance from the centroid to the other points.

- *Hierarchical Clustering*: This family of algorithms has many variations, which differ primarily in two areas. First, we may chose in various ways which two clusters to merge next. Second, we may decide when to stop the merge process in various ways.

- *Picking Clusters to Merge*: One strategy for deciding on the best pair of clusters to merge in a hierarchical clustering is to pick the clusters with the closest centroids or clustroids. Another approach is to pick the pair of clusters with the closest points, one from each cluster. A third approach is to use the average distance between points from the two clusters.

- *Stopping the Merger Process*: A hierarchical clustering can proceed until there are a fixed number of clusters left. Alternatively, we could merge until it is impossible to find a pair of clusters whose merger is sufficiently compact, e.g., the merged cluster has a radius or diameter below some threshold. Another approach involves merging as long as the resulting cluster has a sufficiently high "density," which can be defined in various ways, but is the number of points divided by some measure of the size of the cluster, e.g., the radius.

- *K-Means Algorithms*: This family of algorithms is of the point-assignment type and assumes a Euclidean space. It is assumed that there are exactly k clusters for some known k. After picking k initial cluster centroids, the points are considered one at a time and assigned to the closest centroid. The centroid of a cluster can migrate during point assignment, and an optional last step is to reassign all the points, while holding the centroids fixed at their final values obtained during the first pass.

- *Initializing K-Means Algorithms*: One way to find k initial centroids is to pick a random point, and then choose $k-1$ additional points, each as far away as possible from the previously chosen points. An alternative is to start with a small sample of points and use a hierarchical clustering to merge them into k clusters.

- *Picking K in a K-Means Algorithm*: If the number of clusters is unknown, we can use a binary-search technique, trying a k-means clustering with different values of k. We search for the largest value of k for which a decrease below k clusters results in a radically higher average diameter of the clusters. This search can be carried out in a number of clustering operations that is logarithmic in the true value of k.

- *The BFR Algorithm*: This algorithm is a version of k-means designed to handle data that is too large to fit in main memory. It assumes clusters are normally distributed about the axes.

- *Representing Clusters in BFR*: Points are read from disk one chunk at a time. Clusters are represented in main memory by the count of the number of points, the vector sum of all the points, and the vector formed by summing the squares of the components of the points in each dimension. Other collection of points, too far from a cluster centroid to be included in a cluster, are represented as "miniclusters" in the same way as the k clusters, while still other points, which are not near any other point will be represented as themselves and called "retained" points.

- *Processing Points in BFR*: Most of the points in a main-memory load will be assigned to a nearby cluster and the parameters for that cluster will be adjusted to account for the new points. Unassigned points can be formed into new miniclusters, and these miniclusters can be merged with previously discovered miniclusters or retained points. After the last memory load, the miniclusters and retained points can be merged to their nearest cluster or kept as outliers.

- *The CURE Algorithm*: This algorithm is of the point-assignment type. It is designed for a Euclidean space, but clusters can have any shape. It handles data that is too large to fit in main memory.

- *Representing Clusters in CURE*: The algorithm begins by clustering a small sample of points. It then selects representative points for each cluster, by picking points in the cluster that are as far away from each other as possible. The goal is to find representative points on the fringes of the cluster. However, the representative points are then moved a fraction of the way toward the centroid of the cluster, so they lie somewhat in the interior of the cluster.

- *Processing Points in CURE*: After creating representative points for each cluster, the entire set of points can be read from disk and assigned to a cluster. We assign a given point to the cluster of the representative point that is closest to the given point.

- *The GRGPF Algorithm*: This algorithm is of the point-assignment type. It handles data that is too big to fit in main memory, and it does not assume a Euclidean space.

- *Representing Clusters in GRGPF*: A cluster is represented by the count of points in the cluster, the clustroid, a set of points nearest the clustroid and a set of points furthest from the clustroid. The nearby points allow us to change the clustroid if the cluster evolves, and the distant points allow for merging clusters efficiently in appropriate circumstances. For each of these points, we also record the rowsum, that is the square root

of the sum of the squares of the distances from that point to all the other points of the cluster.

- *Tree Organization of Clusters in GRGPF*: Cluster representations are organized into a tree structure like a B-tree, where nodes of the tree are typically disk blocks and contain information about many clusters. The leaves hold the representation of as many clusters as possible, while interior nodes hold a sample of the clustroids of the clusters at their descendant leaves. We organize the tree so that the clusters whose representatives are in any subtree are as close as possible.

- *Processing Points in GRGPF*: After initializing clusters from a sample of points, we insert each point into the cluster with the nearest clustroid. Because of the tree structure, we can start at the root and choose to visit the child with the sample clustroid nearest to the given point. Following this rule down one path in the tree leads us to a leaf, where we insert the point into the cluster with the nearest clustroid on that leaf.

- *Clustering Streams*: A generalization of the DGIM Algorithm (for counting 1's in the sliding window of a stream) can be used to cluster points that are part of a slowly evolving stream. The BDMO Algorithm uses buckets similar to those of DGIM, with allowable bucket sizes forming a sequence where each size is twice the previous size.

- *Representation of Buckets in BDMO*: The size of a bucket is the number of points it represents. The bucket itself holds only a representation of the clusters of these points, not the points themselves. A cluster representation includes a count of the number of points, the centroid or clustroid, and other information that is needed for merging clusters according to some selected strategy.

- *Merging Buckets in BDMO*: When buckets must be merged, we find the best matching of clusters, one from each of the buckets, and merge them in pairs. If the stream evolves slowly, then we expect consecutive buckets to have almost the same cluster centroids, so this matching makes sense.

- *Answering Queries in BDMO*: A query is a length of a suffix of the sliding window. We take all the clusters in all the buckets that are at least partially within that suffix and merge them using some strategy. The resulting clusters are the answer to the query.

- *Clustering Using Map-Reduce*: We can divide the data into chunks and cluster each chunk in parallel, using a Map task. The clusters from each Map task can be further clustered in a single Reduce task.

7.8 References for Chapter 7

The ancestral study of clustering for large-scale data is the BIRCH Algorithm of [6]. The BFR Algorithm is from [2]. The CURE Algorithm is found in [5].

The paper on the GRGPF Algorithm is [3]. The necessary background regarding B-trees and R-trees can be found in [4]. The study of clustering on streams is taken from [1].

1. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan, "Maintaining variance and k-medians over data stream windows," *Proc. ACM Symp. on Principles of Database Systems*, pp. 234–243, 2003.

2. P.S. Bradley, U.M. Fayyad, and C. Reina, "Scaling clustering algorithms to large databases," *Proc. Knowledge Discovery and Data Mining*, pp. 9–15, 1998.

3. V. Ganti, R. Ramakrishnan, J. Gehrke, A.L. Powell, and J.C. French:, "Clustering large datasets in arbitrary metric spaces," *Proc. Intl. Conf. on Data Engineering*, pp. 502–511, 1999.

4. H. Garcia-Molina, J.D. Ullman, and J. Widom, *Database Systems: The Complete Book* Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2009.

5. S. Guha, R. Rastogi, and K. Shim, "CURE: An efficient clustering algorithm for large databases," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 73–84, 1998.

6. T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: an efficient data clustering method for very large databases," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 103–114, 1996.