11

Cluster Analysis

- "Introduction to Cluster Analysis" on page 11-2
- "Hierarchical Clustering" on page 11-3
- "K-Means Clustering" on page 11-21
- "Gaussian Mixture Models" on page 11-28

Introduction to Cluster Analysis

Cluster analysis, also called *segmentation analysis* or *taxonomy analysis*, creates groups, or *clusters*, of data. Clusters are formed in such a way that objects in the same cluster are very similar and objects in different clusters are very distinct. Measures of similarity depend on the application.

"Hierarchical Clustering" on page 11-3 groups data over a variety of scales by creating a cluster tree or *dendrogram*. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which may be used separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

"K-Means Clustering" on page 11-21 is a partitioning method. The function `kmeans` partitions data into k mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, k-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that k-means clustering is often more suitable than hierarchical clustering for large amounts of data.

"Gaussian Mixture Models" on page 11-28 form clusters by representing the probability density function of observed variables as a mixture of multivariate normal densities. Mixture models of the `gmdistribution` class use an expectation maximization (EM) algorithm to fit data, which assigns posterior probabilities to each component density with respect to each observation. Clusters are assigned by selecting the component that maximizes the posterior probability. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster. Like k-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than k-means clustering when clusters have different sizes and correlation within them.

Hierarchical Clustering

In this section...

"Introduction to Hierarchical Clustering" on page 11-3
"Algorithm Description" on page 11-3
"Similarity Measures" on page 11-4
"Linkages" on page 11-6
"Dendrograms" on page 11-8
"Verifying the Cluster Tree" on page 11-10
"Creating Clusters" on page 11-16

Introduction to Hierarchical Clustering

Hierarchical clustering groups data over a variety of scales by creating a cluster tree or *dendrogram*. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function `clusterdata` supports agglomerative clustering and performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which you can use separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

Algorithm Description

To perform agglomerative hierarchical cluster analysis on a data set using Statistics Toolbox functions, follow this procedure:

1. **Find the similarity or dissimilarity between every pair of objects in the data set.** In this step, you calculate the *distance* between objects using the `pdist` function. The `pdist` function supports many different ways to compute this measurement. See "Similarity Measures" on page 11-4 for more information.

2. **Group the objects into a binary, hierarchical cluster tree.** In this step, you link pairs of objects that are in close proximity using the `linkage`

function. The `linkage` function uses the distance information generated in step 1 to determine the proximity of objects to each other. As objects are paired into binary clusters, the newly formed clusters are grouped into larger clusters until a hierarchical tree is formed. See "Linkages" on page 11-6 for more information.

3 **Determine where to cut the hierarchical tree into clusters.** In this step, you use the `cluster` function to prune branches off the bottom of the hierarchical tree, and assign all the objects below each cut to a single cluster. This creates a partition of the data. The `cluster` function can create these clusters by detecting natural groupings in the hierarchical tree or by cutting off the hierarchical tree at an arbitrary point.

The following sections provide more information about each of these steps.

Note The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

Similarity Measures

You use the `pdist` function to calculate the distance between every pair of objects in a data set. For a data set made up of m objects, there are $m*(m-1)/2$ pairs in the data set. The result of this computation is commonly known as a distance or dissimilarity matrix.

There are many ways to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects; however, you can specify one of several other options. See `pdist` for more information.

Note You can optionally normalize the values in the data set before calculating the distance information. In a real world data set, variables can be measured against different scales. For example, one variable can measure Intelligence Quotient (IQ) test scores and another variable can measure head circumference. These discrepancies can distort the proximity calculations. Using the `zscore` function, you can convert all the values in the data set to use the same proportional scale. See `zscore` for more information.

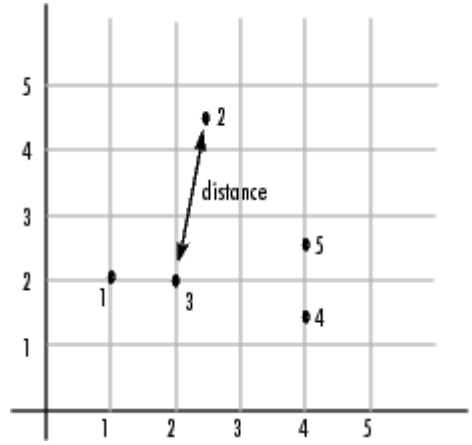
For example, consider a data set, `X`, made up of five objects where each object is a set of *x,y* coordinates.

- **Object 1**: 1, 2
- **Object 2**: 2.5, 4.5
- **Object 3**: 2, 2
- **Object 4**: 4, 1.5
- **Object 5**: 4, 2.5

You can define this data set as a matrix

```
X = [1 2;2.5 4.5;2 2;4 1.5;4 2.5]
```

and pass it to `pdist`. The `pdist` function calculates the distance between object 1 and object 2, object 1 and object 3, and so on until the distances between all the pairs have been calculated. The following figure plots these objects in a graph. The Euclidean distance between object 2 and object 3 is shown to illustrate one interpretation of distance.



Distance Information

The `pdist` function returns this distance information in a vector, `Y`, where each element contains the distance between a pair of objects.

```
Y = pdist(X)
Y =
  Columns 1 through 5
    2.9155    1.0000    3.0414    3.0414    2.5495
  Columns 6 through 10
    3.3541    2.5000    2.0616    2.0616    1.0000
```

To make it easier to see the relationship between the distance information generated by `pdist` and the objects in the original data set, you can reformat the distance vector into a matrix using the `squareform` function. In this matrix, element *i,j* corresponds to the distance between object *i* and object *j* in the original data set. In the following example, element 1,1 represents the distance between object 1 and itself (which is zero). Element 1,2 represents the distance between object 1 and object 2, and so on.

```
squareform(Y)
ans =
         0    2.9155    1.0000    3.0414    3.0414
    2.9155         0    2.5495    3.3541    2.5000
    1.0000    2.5495         0    2.0616    2.0616
    3.0414    3.3541    2.0616         0    1.0000
    3.0414    2.5000    2.0616    1.0000         0
```

Linkages

Once the proximity between objects in the data set has been computed, you can determine how objects in the data set should be grouped into clusters, using the `linkage` function. The `linkage` function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The `linkage` function then links these newly formed clusters to each other and to other objects to create bigger clusters until all the objects in the original data set are linked together in a hierarchical tree.

For example, given the distance vector `Y` generated by `pdist` from the sample data set of *x*- and *y*-coordinates, the `linkage` function generates a hierarchical cluster tree, returning the linkage information in a matrix, `Z`.

```
Z = linkage(Y)
Z =
    4.0000    5.0000    1.0000
```

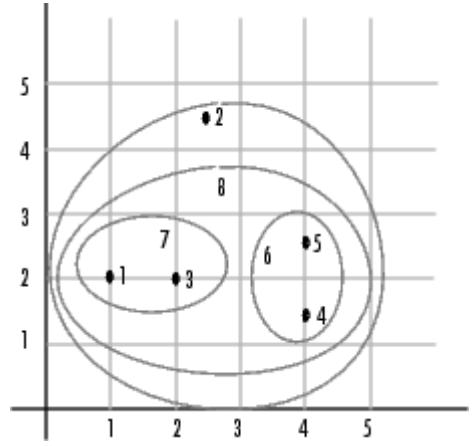
```
    1.0000    3.0000    1.0000
    6.0000    7.0000    2.0616
    2.0000    8.0000    2.5000
```

In this output, each row identifies a link between objects or clusters. The first two columns identify the objects that have been linked. The third column contains the distance between these objects. For the sample data set of x- and y-coordinates, the `linkage` function begins by grouping objects 4 and 5, which have the closest proximity (distance value = 1.0000). The `linkage` function continues by grouping objects 1 and 3, which also have a distance value of 1.0000.

The third row indicates that the `linkage` function grouped objects 6 and 7. If the original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 4 and 5. When the `linkage` function groups two objects into a new cluster, it must assign the cluster a unique index value, starting with the value m+1, where m is the number of objects in the original data set. (Values 1 through m are already used by the original data set.) Similarly, object 7 is the cluster formed by grouping objects 1 and 3.

`linkage` uses distances to determine the order in which it clusters objects. The distance vector `Y` contains the distances between the original objects 1 through 5. But linkage must also be able to determine distances involving clusters that it creates, such as objects 6 and 7. By default, `linkage` uses a method known as single linkage. However, there are a number of different methods available. See the `linkage` reference page for more information.

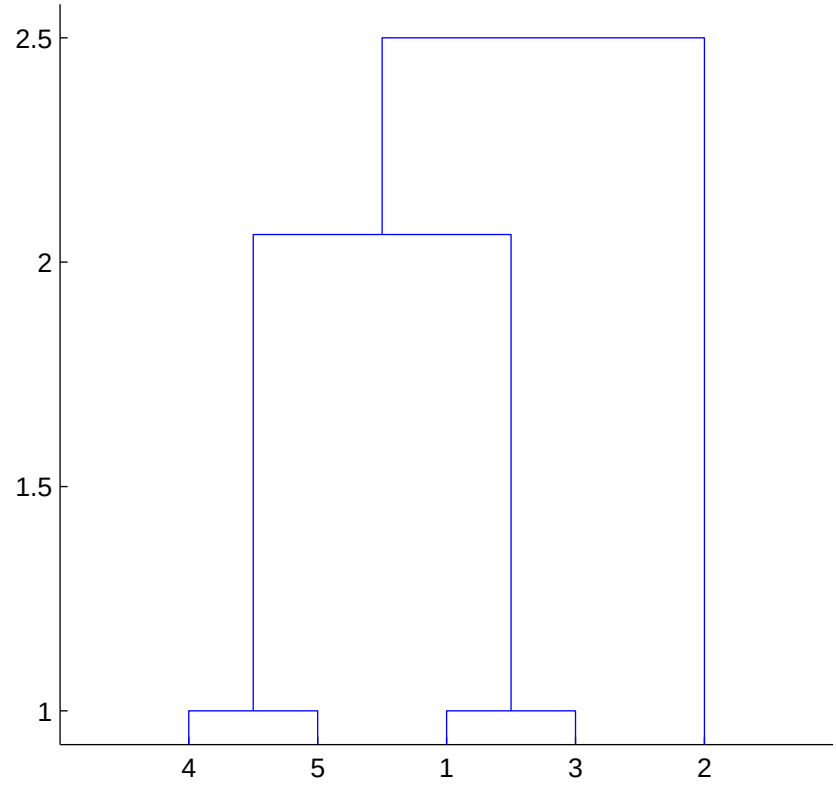
As the final cluster, the `linkage` function grouped object 8, the newly formed cluster made up of objects 6 and 7, with object 2 from the original data set. The following figure graphically illustrates the way `linkage` groups the objects into a hierarchy of clusters.



Dendrograms

The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. The Statistics Toolbox function `dendrogram` plots the tree, as follows:

```
dendrogram(Z)
```



In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing objects 1 and 3 has a height of 1. The link representing the cluster that groups object 2 together with objects 1, 3, 4, and 5, (which are already clustered as object 8) has a height of 2.5. The height represents the distance `linkage` computes between objects 2 and 8. For more information about creating a dendrogram diagram, see the `dendrogram` reference page.

Verifying the Cluster Tree

After linking the objects in a data set into a hierarchical cluster tree, you might want to verify that the distances (that is, heights) in the tree reflect the original distances accurately. In addition, you might want to investigate natural divisions that exist among links between objects. Statistics Toolbox functions are available for both of these tasks, as described in the following sections:

- "Verifying Dissimilarity" on page 11-10
- "Verifying Consistency" on page 11-11

Verifying Dissimilarity

In a hierarchical cluster tree, any two objects in the original data set are eventually linked together at some level. The height of the link represents the distance between the two clusters that contain those two objects. This height is known as the *cophenetic distance* between the two objects. One way to measure how well the cluster tree generated by the `linkage` function reflects your data is to compare the cophenetic distances with the original distance data generated by the `pdist` function. If the clustering is valid, the linking of objects in the cluster tree should have a strong correlation with the distances between objects in the distance vector. The `cophenet` function compares these two sets of values and computes their correlation, returning a value called the *cophenetic correlation coefficient*. The closer the value of the cophenetic correlation coefficient is to 1, the more accurately the clustering solution reflects your data.

You can use the cophenetic correlation coefficient to compare the results of clustering the same data set using different distance calculation methods or clustering algorithms. For example, you can use the `cophenet` function to evaluate the clusters created for the sample data set

```
c = cophenet(Z,Y)
c =
    0.8615
```

where `Z` is the matrix output by the `linkage` function and `Y` is the distance vector output by the `pdist` function.

Execute `pdist` again on the same data set, this time specifying the city block metric. After running the `linkage` function on this new `pdist` output using the average linkage method, call `cophenet` to evaluate the clustering solution.

```
Y = pdist(X,'cityblock');
Z = linkage(Y,'average');
c = cophenet(Z,Y)
c =
    0.9047
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

Verifying Consistency

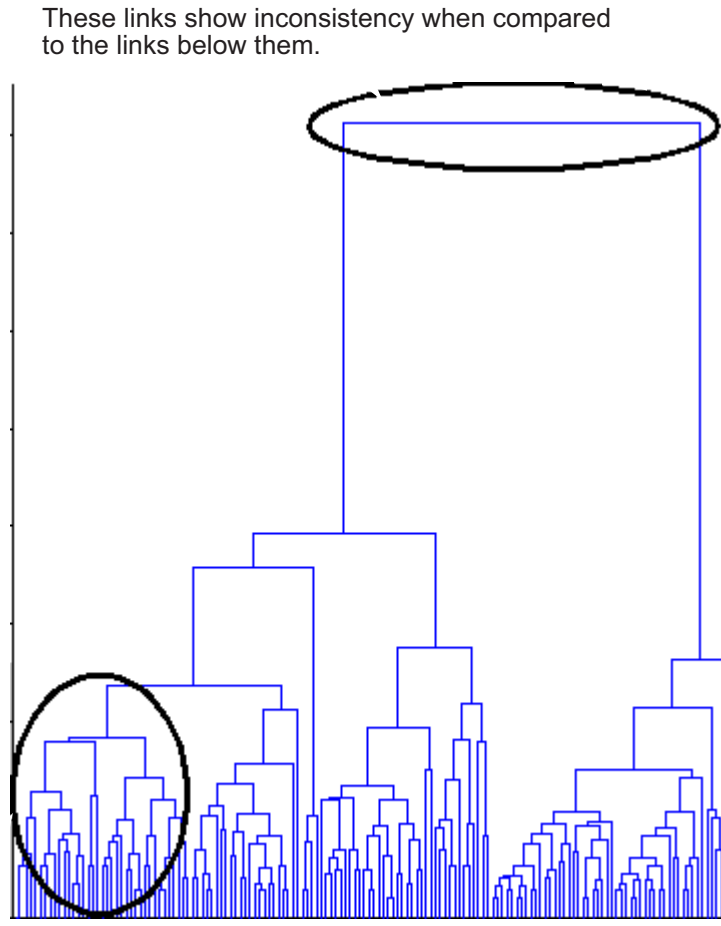
One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree.

A link that is approximately the same height as the links below it indicates that there are no distinct divisions between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of consistency, because the distance between the objects being joined is approximately the same as the distances between the objects they contain.

On the other hand, a link whose height differs noticeably from the height of the links below it indicates that the objects joined at this level in the cluster tree are much farther apart from each other than their components were when they were joined. This link is said to be inconsistent with the links below it.

In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The `cluster` function uses a quantitative measure of inconsistency to determine where to partition your data set into clusters.

The following dendrogram illustrates inconsistent links. Note how the objects in the dendrogram fall into two groups that are connected by links at a much higher level in the tree. These links are inconsistent when compared with the links below them in the hierarchy.



The relative consistency of each link in a hierarchical cluster tree can be quantified and expressed as the *inconsistency coefficient*. This value compares the height of a link in a cluster hierarchy with the average height of links below it. Links that join distinct clusters have a high inconsistency coefficient; links that join indistinct clusters have a low inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link in the cluster tree, use the `inconsistent` function. By default, the `inconsistent`

function compares each link in the cluster hierarchy with adjacent links that are less than two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. You can also specify other depths. The objects at the bottom of the cluster tree, called leaf nodes, that have no further objects below them, have an inconsistency coefficient of zero. Clusters that join two leaves also have a zero inconsistency coefficient.

For example, you can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function in "Linkages" on page 11-6.

```
I = inconsistent(Z)
I =
    1.0000         0    1.0000         0
    1.0000         0    1.0000         0
    1.3539    0.6129    3.0000    1.1547
    2.2808    0.3100    2.0000    0.7071
```

The `inconsistent` function returns data about the links in an (m-1)-by-4 matrix, whose columns are described in the following table.

Column	Description
1	Mean of the heights of all the links included in the calculation
2	Standard deviation of all the links included in the calculation
3	Number of links included in the calculation
4	Inconsistency coefficient

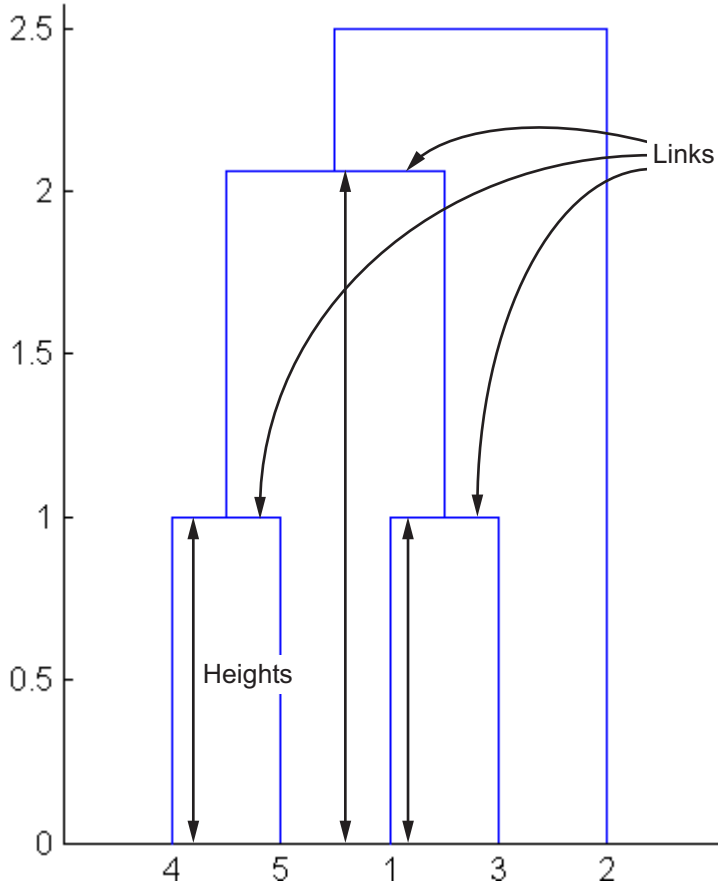
In the sample output, the first row represents the link between objects 4 and 5. This cluster is assigned the index 6 by the `linkage` function. Because both 4 and 5 are leaf nodes, the inconsistency coefficient for the cluster is zero. The second row represents the link between objects 1 and 3, both of which are also leaf nodes. This cluster is assigned the index 7 by the linkage function.

The third row evaluates the link that connects these two clusters, objects 6 and 7. (This new cluster is assigned index 8 in the `linkage` output). Column 3 indicates that three links are considered in the calculation: the link itself and the two links directly below it in the hierarchy. Column 1 represents the mean of the heights of these links. The `inconsistent` function uses the height

information output by the `linkage` function to calculate the mean. Column 2 represents the standard deviation between the links. The last column contains the inconsistency value for these links, 1.1547. It is the difference between the current link height and the mean, normalized by the standard deviation:

```
(2.0616 - 1.3539) / .6129
ans =
    1.1547
```

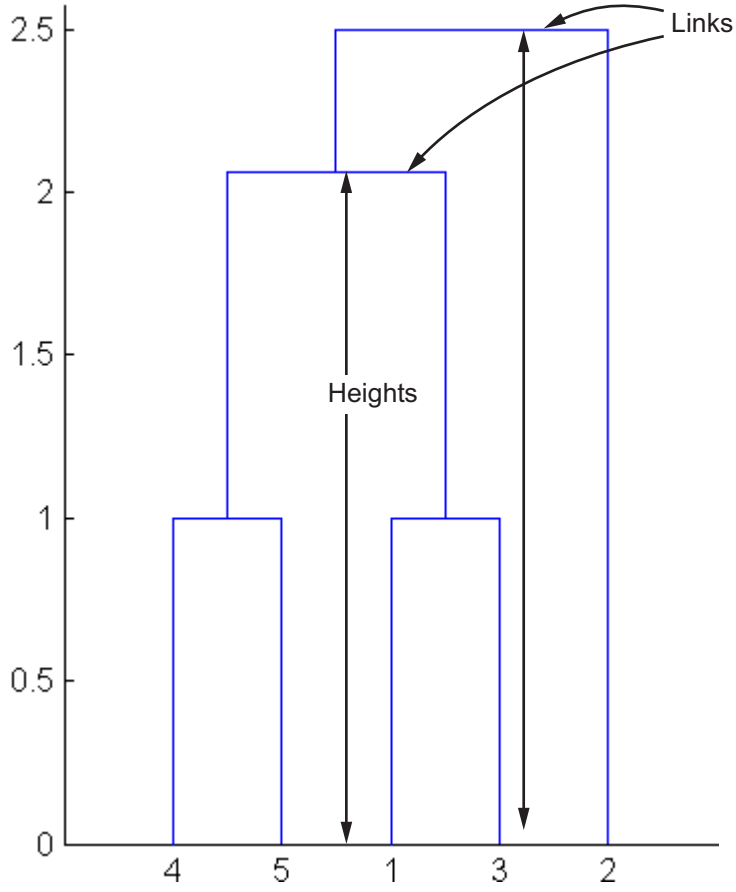
The following figure illustrates the links and heights included in this calculation.



Note In the preceding figure, the lower limit on the *y*-axis is set to `0` to show the heights of the links. To set the lower limit to `0`, select `Axes Properties` from the **Edit** menu, click the **Y Axis** tab, and enter `0` in the field immediately to the right of **Y Limits**.

Row 4 in the output matrix describes the link between object 8 and object 2. Column 3 indicates that two links are included in this calculation: the link itself and the link directly below it in the hierarchy. The inconsistency coefficient for this link is 0.7071.

The following figure illustrates the links and heights included in this calculation.



Creating Clusters

After you create the hierarchical tree of binary clusters, you can prune the tree to partition your data into clusters using the `cluster` function. The `cluster` function lets you create clusters in two ways, as discussed in the following sections:

- "Finding Natural Divisions in Data" on page 11-17
- "Specifying Arbitrary Clusters" on page 11-18

Finding Natural Divisions in Data

The hierarchical cluster tree may naturally divide the data into distinct, well-separated clusters. This can be particularly evident in a dendrogram diagram created from data where groups of objects are densely packed in certain areas and not in others. The inconsistency coefficient of the links in the cluster tree can identify these divisions where the similarities between objects change abruptly. (See "Verifying the Cluster Tree" on page 11-10 for more information about the inconsistency coefficient.) You can use this value to determine where the `cluster` function creates cluster boundaries.

For example, if you use the `cluster` function to group the sample data set into clusters, specifying an inconsistency coefficient threshold of `1.2` as the value of the `cutoff` argument, the `cluster` function groups all the objects in the sample data set into one cluster. In this case, none of the links in the cluster hierarchy had an inconsistency coefficient greater than `1.2`.

```
T = cluster(Z,'cutoff',1.2)
T =
     1
     1
     1
     1
     1
```

The `cluster` function outputs a vector, `T`, that is the same size as the original data set. Each element in this vector contains the number of the cluster into which the corresponding object from the original data set was placed.

If you lower the inconsistency coefficient threshold to `0.8`, the `cluster` function divides the sample data set into three separate clusters.

```
T = cluster(Z,'cutoff',0.8)
T =
    3
    2
    3
    1
    1
```

This output indicates that objects 1 and 3 were placed in cluster 1, objects 4 and 5 were placed in cluster 2, and object 2 was placed in cluster 3.

When clusters are formed in this way, the cutoff value is applied to the inconsistency coefficient. These clusters may, but do not necessarily, correspond to a horizontal slice across the dendrogram at a certain height. If you want clusters corresponding to a horizontal slice of the dendrogram, you can either use the `criterion` option to specify that the cutoff should be based on distance rather than inconsistency, or you can specify the number of clusters directly as described in the following section.

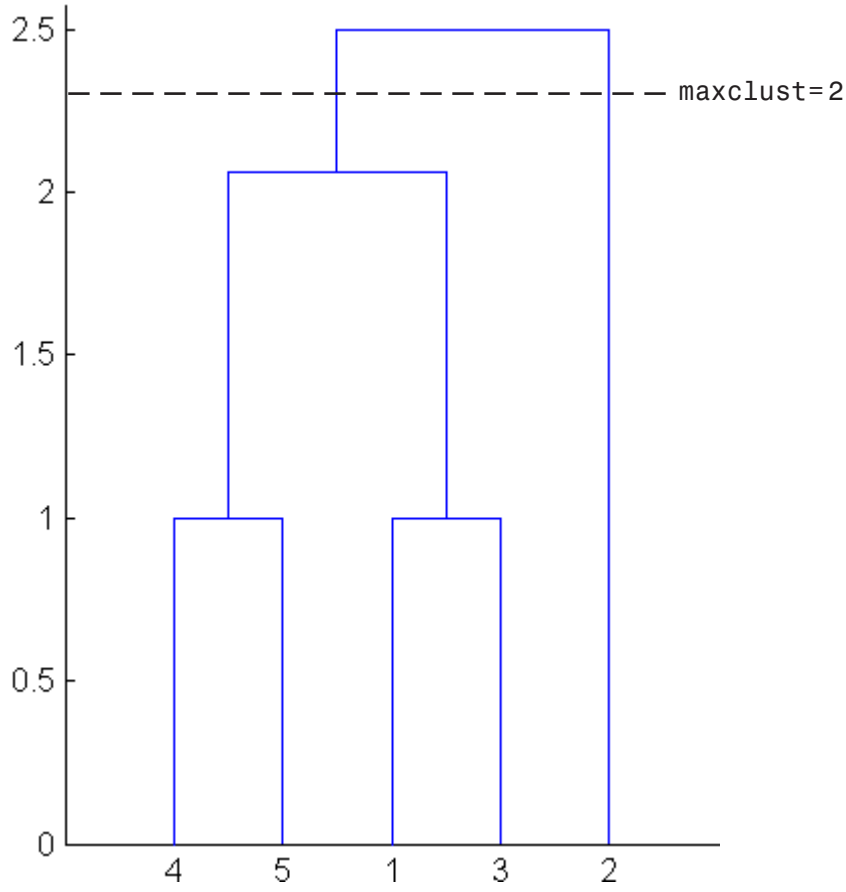
Specifying Arbitrary Clusters

Instead of letting the `cluster` function create clusters determined by the natural divisions in the data set, you can specify the number of clusters you want created.

For example, you can specify that you want the `cluster` function to partition the sample data set into two clusters. In this case, the `cluster` function creates one cluster containing objects 1, 3, 4, and 5 and another cluster containing object 2.

```
T = cluster(Z,'maxclust',2)
T =
     2
     1
     2
     2
     2
```

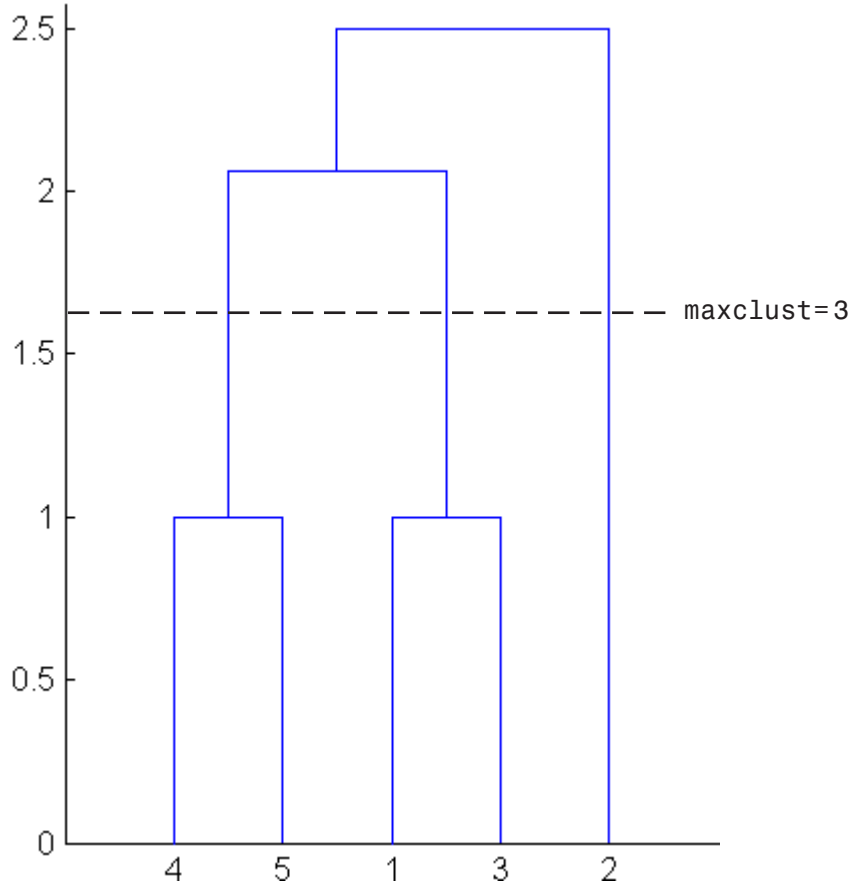
To help you visualize how the `cluster` function determines these clusters, the following figure shows the dendrogram of the hierarchical cluster tree. The horizontal dashed line intersects two lines of the dendrogram, corresponding to setting `'maxclust'` to `2`. These two lines partition the objects into two clusters: the objects below the left-hand line, namely 1, 3, 4, and 5, belong to one cluster, while the object below the right-hand line, namely 2, belongs to the other cluster.



On the other hand, if you set `'maxclust'` to `3`, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

```
T = cluster(Z,'maxclust',3)
T =
     1
     3
     1
     2
     2
```

This time, the `cluster` function cuts off the hierarchy at a lower point, corresponding to the horizontal line that intersects three lines of the dendrogram in the following figure.



K-Means Clustering

In this section...

"Introduction to K-Means Clustering" on page 11-21

"Creating Clusters and Determining Separation" on page 11-22

"Determining the Correct Number of Clusters" on page 11-23

"Avoiding Local Minima" on page 11-26

Introduction to K-Means Clustering

K-means clustering is a partitioning method. The function `kmeans` partitions data into *k* mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, *k*-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that *k*-means clustering is often more suitable than hierarchical clustering for large amounts of data.

`kmeans` treats each observation in your data as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. You can choose from five different distance measures, depending on the kind of data you are clustering.

Each cluster in the partition is defined by its member objects and by its centroid, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize the sum with respect to the measure that you specify.

`kmeans` uses an iterative algorithm that minimizes the sum of distances from each object to its cluster centroid, over all clusters. This algorithm moves objects between clusters until the sum cannot be decreased further. The result is a set of clusters that are as compact and well-separated as possible. You can control the details of the minimization using several optional input parameters to `kmeans`, including ones for the initial values of the cluster centroids, and for the maximum number of iterations.

Creating Clusters and Determining Separation

The following example explores possible clustering in four-dimensional data by analyzing the results of partitioning the points into three, four, and five clusters.

Note Because each part of this example generates random numbers sequentially, i.e., without setting a new state, you must perform all steps in sequence to duplicate the results shown. If you perform the steps out of sequence, the answers will be essentially the same, but the intermediate results, number of iterations, or ordering of the silhouette plots may differ.

First, load some data:

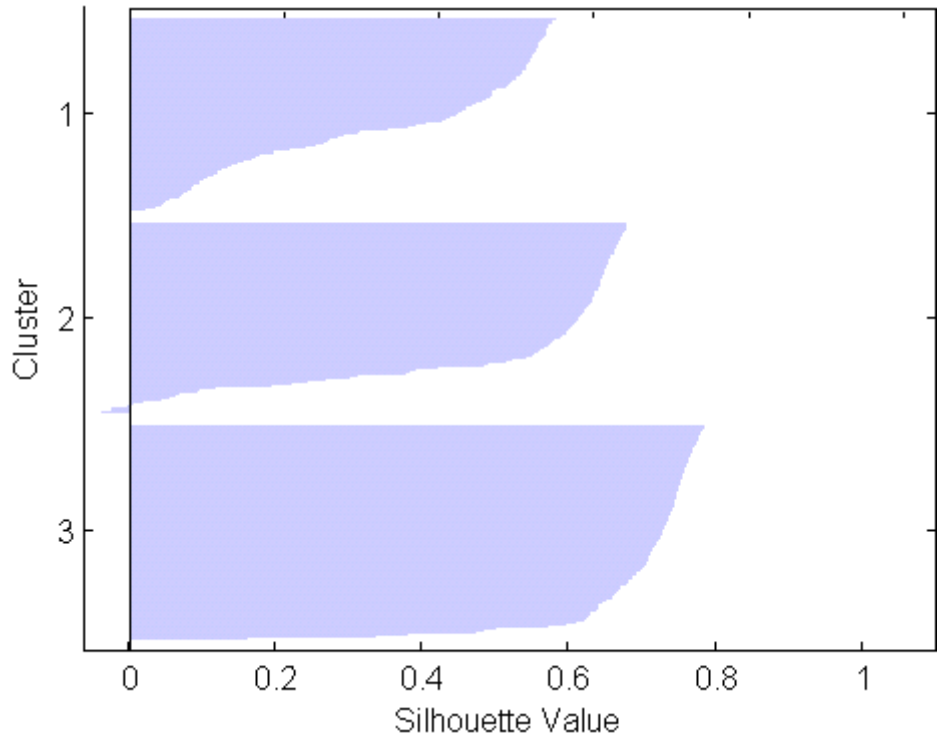
```
load kmeansdata;
size(X)
ans =
   560     4
```

Even though these data are four-dimensional, and cannot be easily visualized, `kmeans` enables you to investigate whether a group structure exists in them. Call `kmeans` with `k`, the desired number of clusters, equal to `3`. For this example, specify the city block distance measure, and use the default starting method of initializing centroids from randomly selected data points:

```
idx3 = kmeans(X,3,'distance','city');
```

To get an idea of how well-separated the resulting clusters are, you can make a silhouette plot using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating points that are very distant from neighboring clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are probably assigned to the wrong cluster. `silhouette` returns these values in its first output:

```
[silh3,h] = silhouette(X,idx3,'city');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
ylabel('Cluster')
```



From the silhouette plot, you can see that most points in the third cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the first cluster contains many points with low silhouette values, and the second contains a few points with negative values, indicating that those two clusters are not well separated.

Determining the Correct Number of Clusters

Increase the number of clusters to see if `kmeans` can find a better grouping of the data. This time, use the optional `'display'` parameter to print information about each iteration:

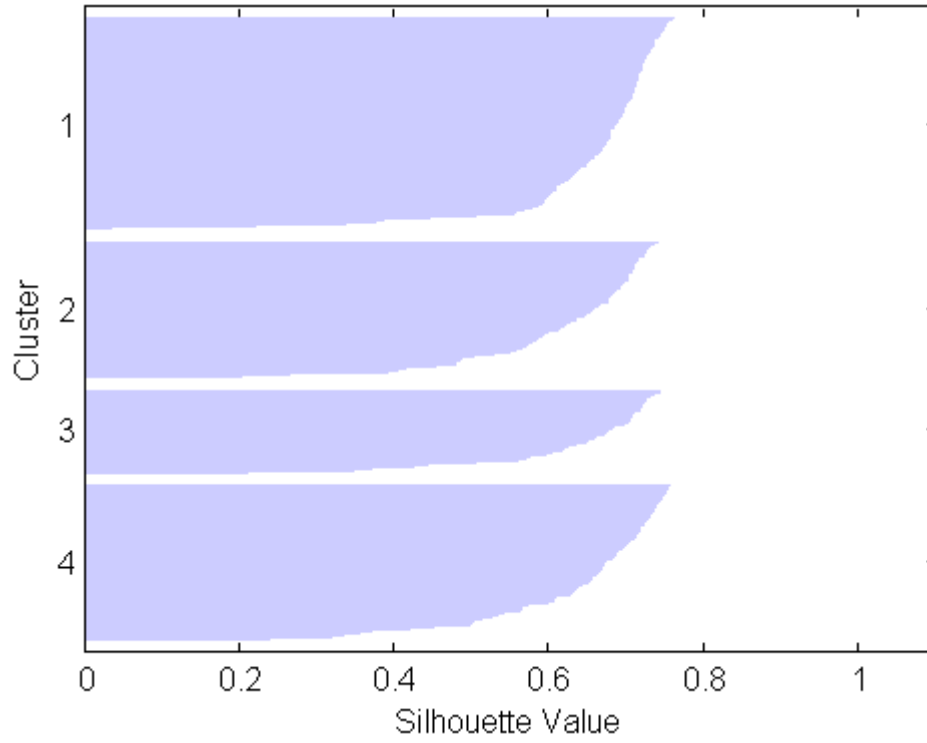
```
idx4 = kmeans(X,4, 'dist','city', 'display','iter');
  iter  phase      num           sum
     1      1      560      2897.56
```

```
     2      1       53      2736.67
     3      1       50      2476.78
     4      1      102      1779.68
     5      1        5       1771.1
     6      2        0       1771.1
6 iterations, total sum of distances = 1771.1
```

Notice that the total sum of distances decreases at each iteration as `kmeans` reassigns points between clusters and recomputes cluster centroids. In this case, the second phase of the algorithm did not make any reassignments, indicating that the first phase reached a minimum after five iterations. In some problems, the first phase might not reach a minimum, but the second phase always will.

A silhouette plot for this solution indicates that these four clusters are better separated than the three in the previous solution:

```
[silh4,h] = silhouette(X,idx4,'city');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
ylabel('Cluster')
```



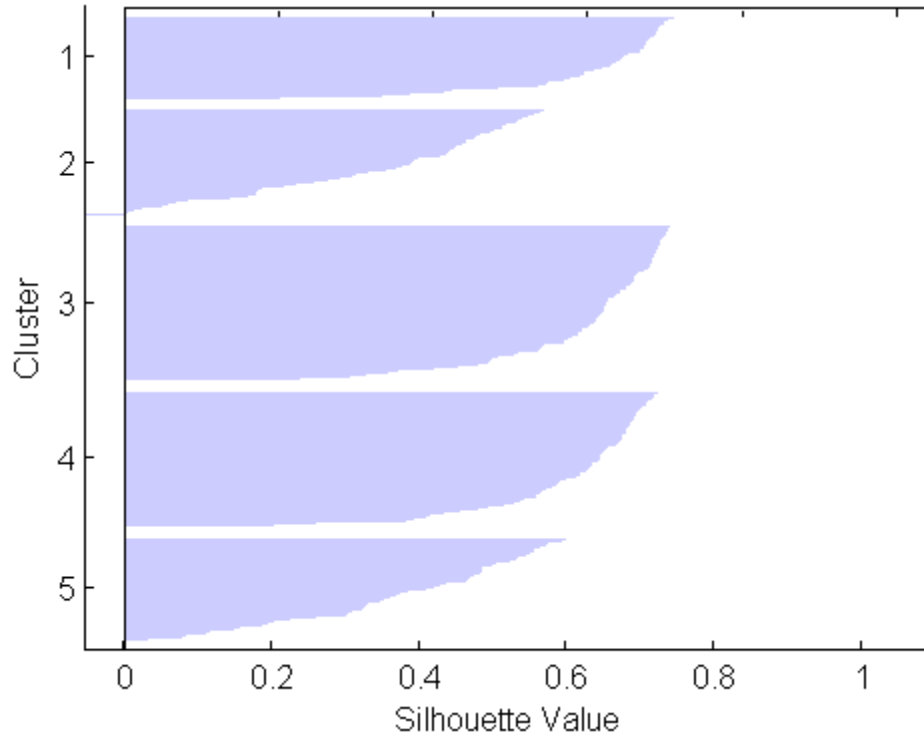
A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases:

```
mean(silh3)
ans =
      0.52594
mean(silh4)
ans =
      0.63997
```

Finally, try clustering the data using five clusters:

```
idx5 = kmeans(X,5,'dist','city','replicates',5);
[silh5,h] = silhouette(X,idx5,'city');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
```

```
ylabel('Cluster')
mean(silh5)
ans =
      0.52657
```



This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for `k`.

Avoiding Local Minima

Like many other types of numerical minimizations, the solution that `kmeans` reaches often depends on the starting points. It is possible for `kmeans` to reach a local minimum, where reassigning any one point to a new cluster would increase the total sum of point-to-centroid distances, but where a

better solution does exist. However, you can use the optional `'replicates'` parameter to overcome that problem.

For four clusters, specify five replicates, and use the `'display'` parameter to print out the final sum of distances for each of the solutions.

```
[idx4,cent4,sumdist] = kmeans(X,4,'dist','city',...
                      'display','final','replicates',5);
17 iterations, total sum of distances = 2303.36
 5 iterations, total sum of distances = 1771.1
 6 iterations, total sum of distances = 1771.1
 5 iterations, total sum of distances = 1771.1
 8 iterations, total sum of distances = 2303.36
```

The output shows that, even for this relatively simple problem, non-global minima do exist. Each of these five replicates began from a different randomly selected set of initial centroids, and `kmeans` found two different local minima. However, the final solution that `kmeans` returns is the one with the lowest total sum of distances, over all replicates.

```
sum(sumdist)
ans =
       1771.1
```

Gaussian Mixture Models

In this section...

"Introduction to Gaussian Mixture Models" on page 11-28

"Clustering with Gaussian Mixtures" on page 11-28

Introduction to Gaussian Mixture Models

Gaussian mixture models are formed by combining multivariate normal density components. For information on individual multivariate normal densities, see "Multivariate Normal Distribution" on page B-58 and related distribution functions listed under "Multivariate Distributions" on page 5-8.

In Statistics Toolbox software, use the `gmdistribution` class to fit data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation.

Gaussian mixture models are often used for data clustering. Clusters are assigned by selecting the component that maximizes the posterior probability. Like *k*-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than *k*-means clustering when clusters have different sizes and correlation within them. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster.

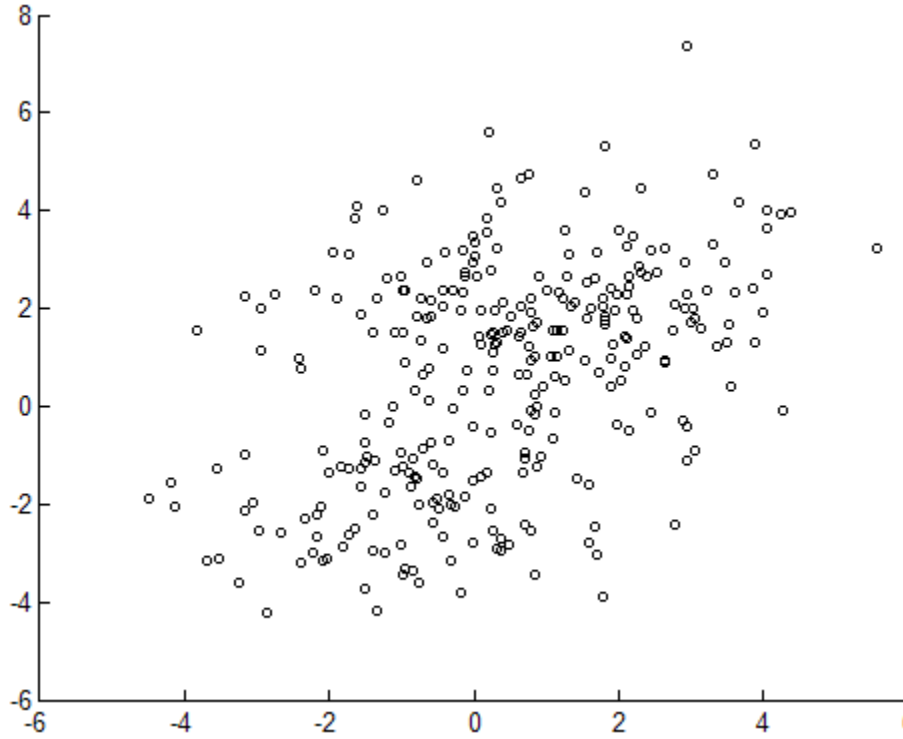
Creation of Gaussian mixture models is described in the "Gaussian Mixture Models" on page 5-99 section of Chapter 5, "Probability Distributions". This section describes their application in cluster analysis.

Clustering with Gaussian Mixtures

Gaussian mixture distributions can be used for clustering data, by realizing that the multivariate normal components of the fitted model can represent clusters.

1 To demonstrate the process, first generate some simulated data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

```
mu1 = [1 2];
sigma1 = [3 .2; .2 2];
mu2 = [-1 -2];
sigma2 = [2 0; 0 1];
X = [mvnrnd(mu1,sigma1,200);mvnrnd(mu2,sigma2,100)];

scatter(X(:,1),X(:,2),10,'ko')
```



2 Fit a two-component Gaussian mixture distribution. Here, you know the correct number of components to use. In practice, with real data, this decision would require comparing models with different numbers of components.

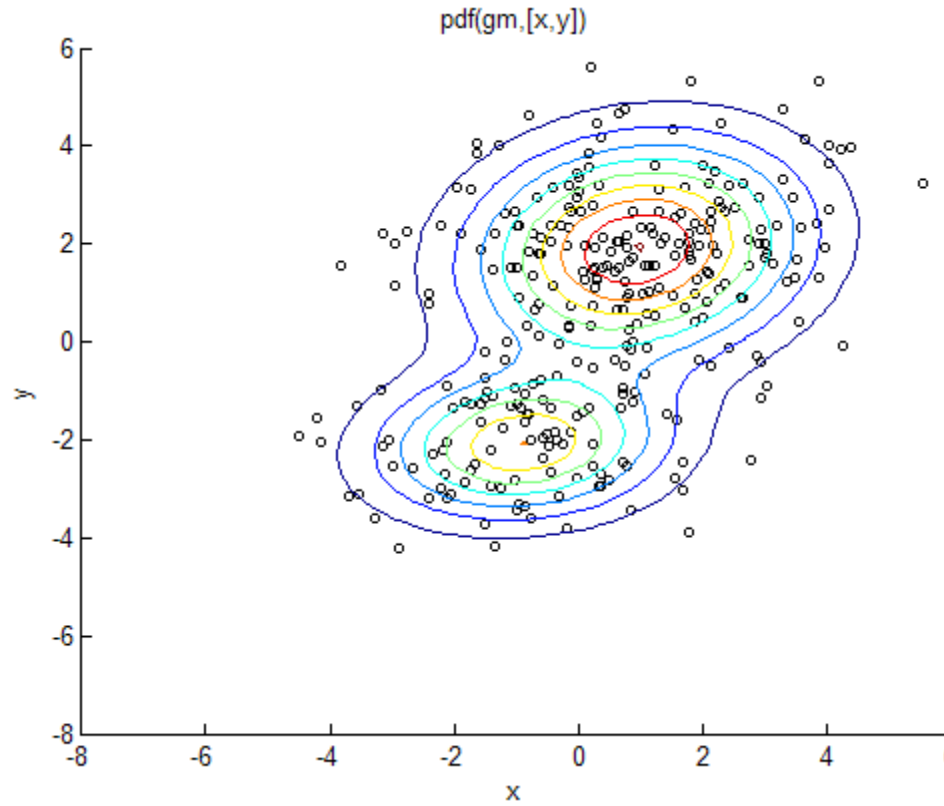
```
options = statset('Display','final');
gm = gmdistribution.fit(X,2,'Options',options);
```

This displays

```
 49 iterations, log-likelihood = -1207.91
```

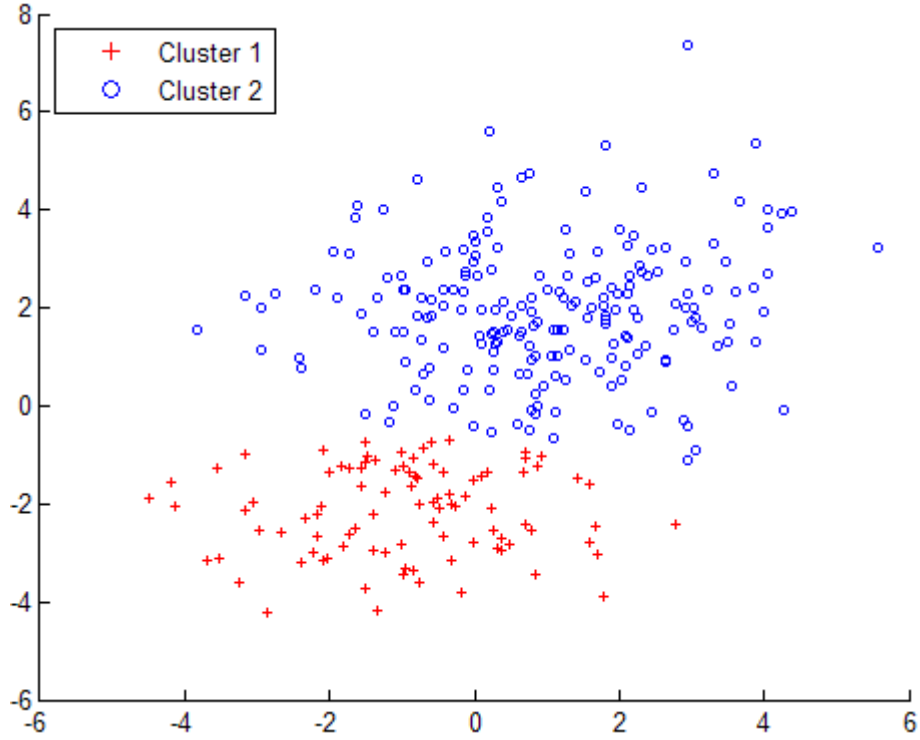
3 Plot the estimated probability density contours for the two-component mixture distribution. The two bivariate normal components overlap, but their peaks are distinct. This suggests that the data could reasonably be divided into two clusters:

```
hold on
ezcontour(@(x,y)pdf(gm,[x y]),[-8 6],[-8 6]);
hold off
```



4 Partition the data into clusters using the `cluster` method for the fitted mixture distribution. The `cluster` method assigns each point to one of the two components in the mixture distribution.

```
idx = cluster(gm,X);
cluster1 = (idx == 1);
cluster2 = (idx == 2);

scatter(X(cluster1,1),X(cluster1,2),10,'r+');
hold on
scatter(X(cluster2,1),X(cluster2,2),10,'bo');
hold off
legend('Cluster 1','Cluster 2','Location','NW')
```

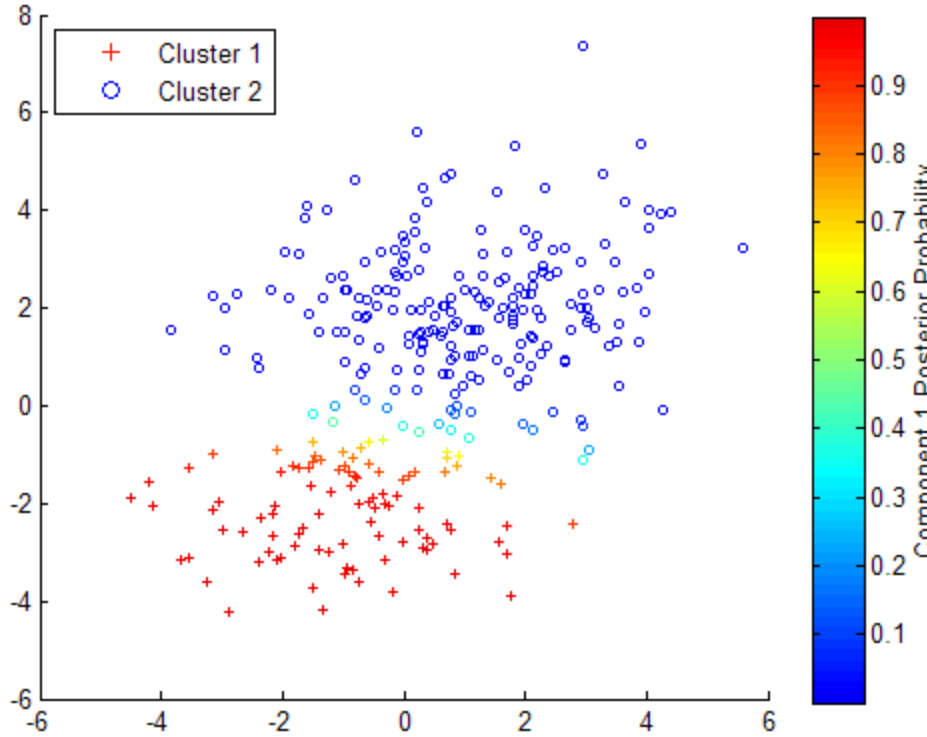


Each cluster corresponds to one of the bivariate normal components in the mixture distribution. `cluster` assigns points to clusters based on the estimated posterior probability that a point came from a component; each point is assigned to the cluster corresponding to the highest posterior probability. The posterior method returns those `posterior` probabilities.

For example, plot the posterior probability of the first component for each point:

```
P = posterior(gm,X);

scatter(X(cluster1,1),X(cluster1,2),10,P(cluster1,1),'+')
hold on
scatter(X(cluster2,1),X(cluster2,2),10,P(cluster2,1),'o')
hold off
legend('Cluster 1','Cluster 2','Location','NW')
clrmap = jet(80); colormap(clrmap(9:72,:))
```

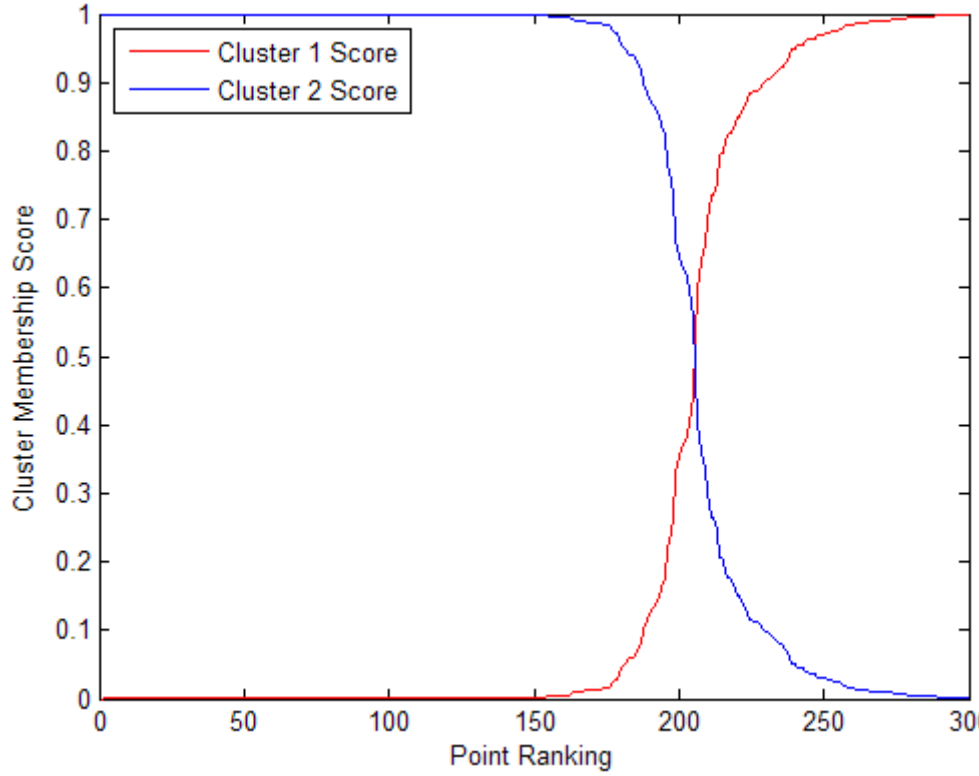
```
ylabel(colorbar,'Component 1 Posterior Probability')
```



Soft Clustering Using Gaussian Mixture Distributions

An alternative to the previous example is to use the posterior probabilities for "soft clustering". Each point is assigned a membership score to each cluster. Membership scores are simply the posterior probabilities, and describe how similar each point is to each cluster's archetype, i.e., the mean of the corresponding component. The points can be ranked by their membership score in a given cluster:

```
[~,order] = sort(P(:,1));
plot(1:size(X,1),P(order,1),'r-',1:size(X,1),P(order,2),'b-');
legend({'Cluster 1 Score' 'Cluster 2 Score'},'location','NW');
ylabel('Cluster Membership Score');
xlabel('Point Ranking');
```



Although a clear separation of the data is hard to see in a scatter plot of the data, plotting the membership scores indicates that the fitted distribution does a good job of separating the data into groups. Very few points have scores close to 0.5.

Soft clustering using a Gaussian mixture distribution is similar to fuzzy K-means clustering, which also assigns each point to each cluster with a membership score. The fuzzy K-means algorithm assumes that clusters are roughly spherical in shape, and all of roughly equal size. This is comparable to a Gaussian mixture distribution with a single covariance matrix that is shared across all components, and is a multiple of the identity matrix. In contrast, `gmdistribution` allows you to specify different covariance options. The default is to estimate a separate, unconstrained covariance matrix for

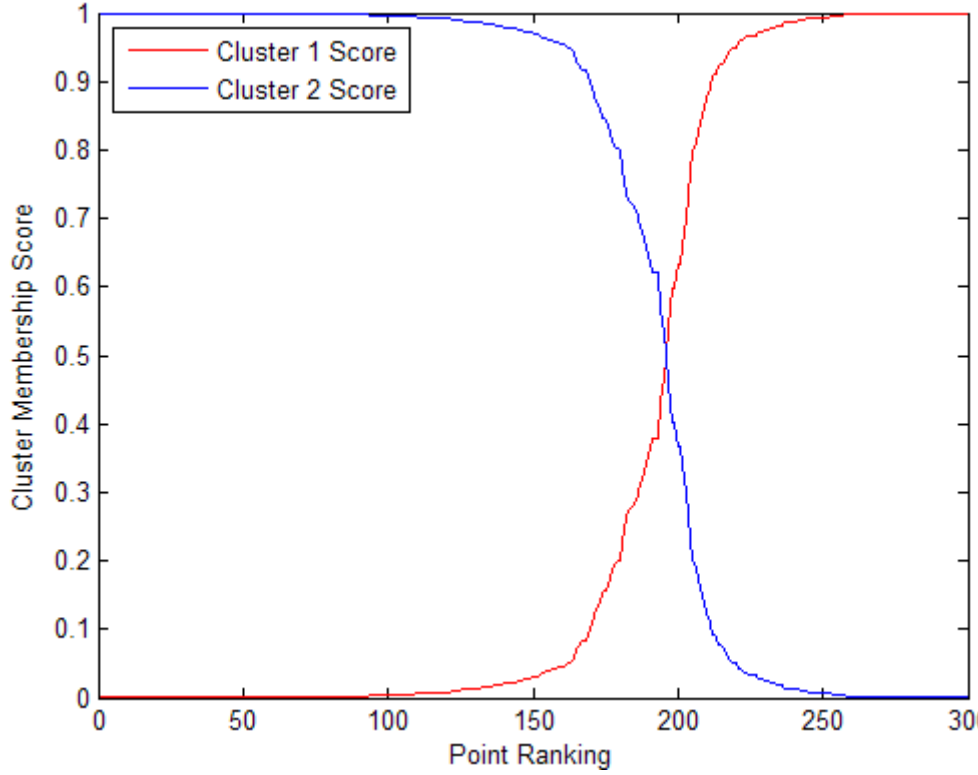
each component. A more restricted option, closer to K-means, would be to estimate a shared, diagonal covariance matrix:

```
gm2 = gmdistribution.fit(X,2,'CovType','Diagonal',...
  'SharedCov',true);
```

This covariance option is similar to fuzzy K-means clustering, but provides more flexibility by allowing unequal variances for different variables.

You can compute the soft cluster membership scores without computing hard cluster assignments, using `posterior`, or as part of hard clustering, as the second output from `cluster`:

```
P2 = posterior(gm2,X); % equivalently [idx,P2] = cluster(gm2,X)
[~,order] = sort(P2(:,1));
plot(1:size(X,1),P2(order,1),'r-',1:size(X,1),P2(order,2),'b-');
legend({'Cluster 1 Score' 'Cluster 2 Score'},'location','NW');
ylabel('Cluster Membership Score');
xlabel('Point Ranking');
```



Assigning New Data to Clusters

In the previous example, fitting the mixture distribution to data using `fit`, and clustering those data using `cluster`, are separate steps. However, the same data are used in both steps. You can also use the `cluster` method to assign new data points to the clusters (mixture components) found in the original data.

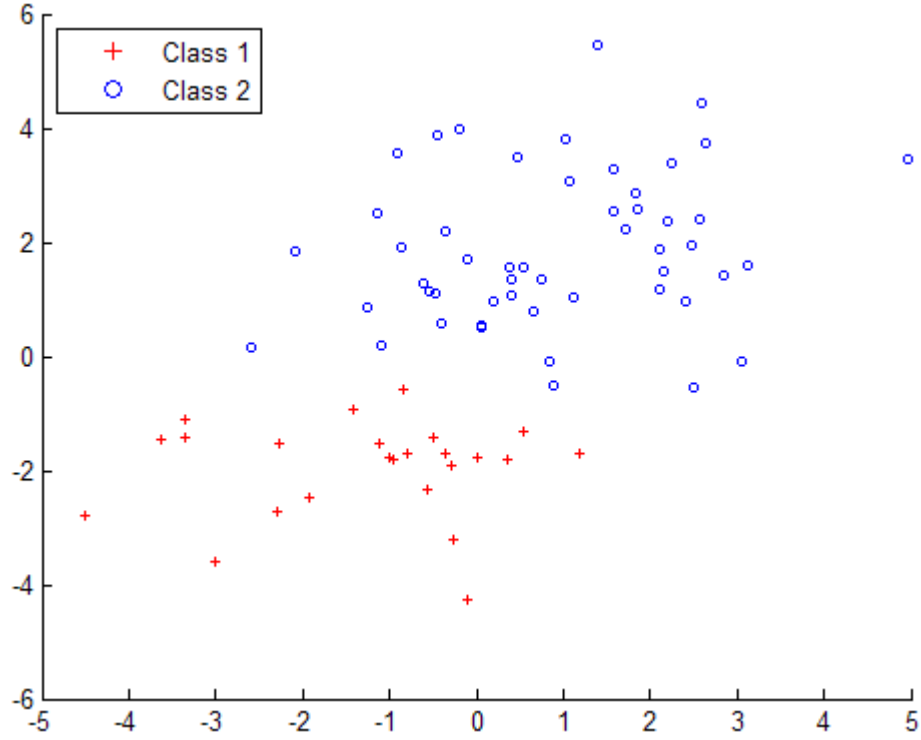
1 Given a data set `X`, first fit a Gaussian mixture distribution. The previous code has already done that.

```
gm

gm =
Gaussian mixture distribution with 2 components in 2 dimensions
```

```
Component 1:
Mixing proportion: 0.312592
Mean:    -0.9082   -2.1109

Component 2:
Mixing proportion: 0.687408
Mean:     0.9532    1.8940
```

2 You can then use `cluster` to assign each point in a new data set, `Y`, to one of the clusters defined for the original data:

```
Y = [mvnrnd(mu1,sigma1,50);mvnrnd(mu2,sigma2,25)];

idx = cluster(gm,Y);
cluster1 = (idx == 1);
cluster2 = (idx == 2);

scatter(Y(cluster1,1),Y(cluster1,2),10,'r+');
hold on
scatter(Y(cluster2,1),Y(cluster2,2),10,'bo');
hold off
legend('Class 1','Class 2','Location','NW')
```



As with the previous example, the posterior probabilities for each point can be treated as membership scores rather than determining "hard" cluster assignments.

For `cluster` to provide meaningful results with new data, `Y` should come from the same population as `X`, the original data used to create the mixture distribution. In particular, the estimated mixing probabilities for the Gaussian mixture distribution fitted to `X` are used when computing the posterior probabilities for `Y`.