Agglomerative Clustering

Agglomerative clustering refers to a collection of clustering algorithms that all build upon the same principles: the algorithm starts by declaring each point its own cluster, and then merges the two most similar clusters until some stopping criterion is satisfied. The stopping criterion implemented in `scikit-learn` is the number of clusters, so similar clusters are merged until only the specified number of clusters are left. There are several *linkage* criteria that specify how exactly the "most similar cluster" is measured. This measure is always defined between two existing clusters.

The following three choices are implemented in `scikit-learn`:

`ward`
: The default choice, `ward` picks the two clusters to merge such that the variance within all clusters increases the least. This often leads to clusters that are relatively equally sized.

`average`
: `average` linkage merges the two clusters that have the smallest average distance between all their points.

`complete`
: `complete` linkage (also known as maximum linkage) merges the two clusters that have the smallest maximum distance between their points.

`ward` works on most datasets, and we will use it in our examples. If the clusters have very dissimilar numbers of members (if one is much bigger than all the others, for example), `average` or `complete` might work better.

The following plot (Figure 3-33) illustrates the progression of agglomerative clustering on a two-dimensional dataset, looking for three clusters:

`In[61]:`

```
mglearn.plots.plot_agglomerative_algorithm()
```

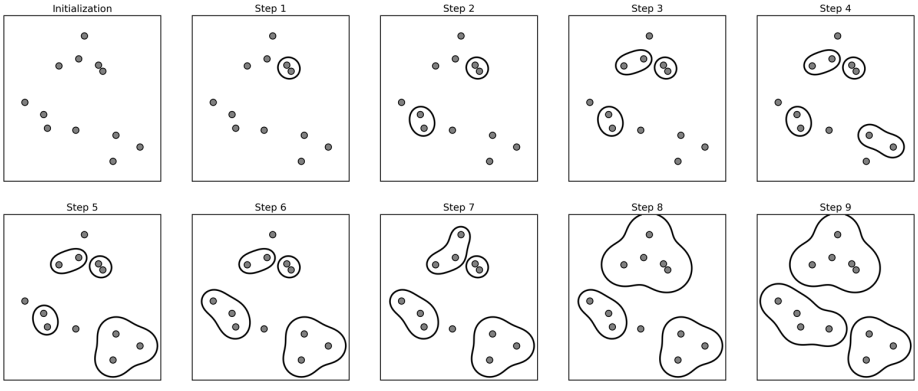

Figure 3-33. Agglomerative clustering iteratively joins the two closest clusters

Initially, each point is its own cluster. Then, in each step, the two clusters that are closest are merged. In the first four steps, two single-point clusters are picked and these are joined into two-point clusters. In step 5, one of the two-point clusters is extended to a third point, and so on. In step 9, there are only three clusters remaining. As we specified that we are looking for three clusters, the algorithm then stops.

Let's have a look at how agglomerative clustering performs on the simple three-cluster data we used here. Because of the way the algorithm works, agglomerative clustering cannot make predictions for new data points. Therefore, `AgglomerativeClustering` has no `predict` method. To build the model and get the cluster memberships on the training set, use the `fit_predict` method instead.[5] The result is shown in Figure 3-34:

In[62]:

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

5 We could also use the `labels_` attribute, as we did for *k*-means.

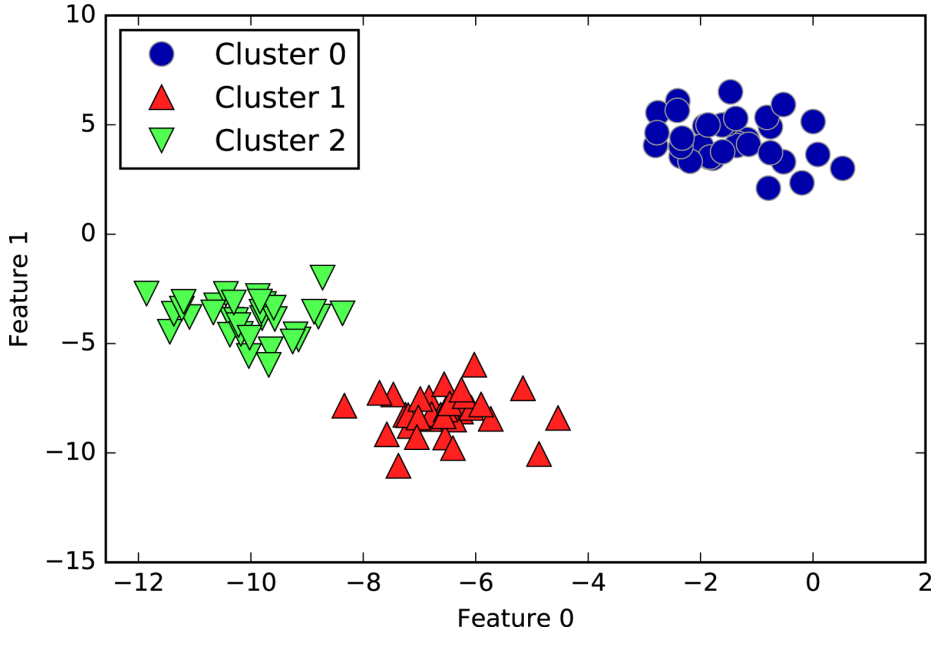

Figure 3-34. Cluster assignment using agglomerative clustering with three clusters

As expected, the algorithm recovers the clustering perfectly. While the `scikit-learn` implementation of agglomerative clustering requires you to specify the number of clusters you want the algorithm to find, agglomerative clustering methods provide some help with choosing the right number, which we will discuss next.

Hierarchical clustering and dendrograms

Agglomerative clustering produces what is known as a *hierarchical clustering*. The clustering proceeds iteratively, and every point makes a journey from being a single point cluster to belonging to some final cluster. Each intermediate step provides a clustering of the data (with a different number of clusters). It is sometimes helpful to look at all possible clusterings jointly. The next example (Figure 3-35) shows an overlay of all the possible clusterings shown in Figure 3-33, providing some insight into how each cluster breaks up into smaller clusters:

`In[63]:`

```
mglearn.plots.plot_agglomerative()
```

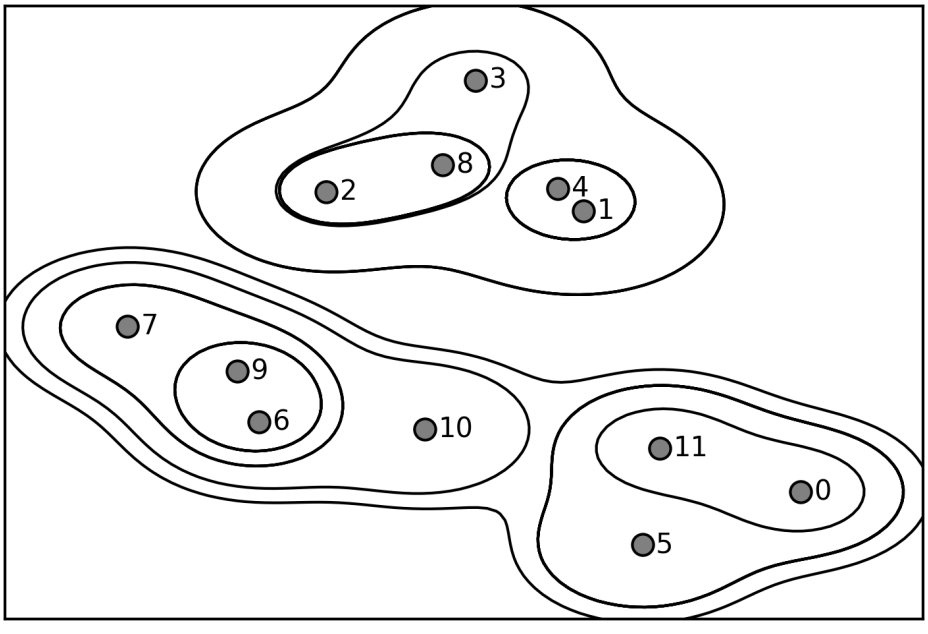

Figure 3-35. Hierarchical cluster assignment (shown as lines) generated with agglomerative clustering, with numbered data points (cf. Figure 3-36)

While this visualization provides a very detailed view of the hierarchical clustering, it relies on the two-dimensional nature of the data and therefore cannot be used on datasets that have more than two features. There is, however, another tool to visualize hierarchical clustering, called a *dendrogram*, that can handle multidimensional datasets.

Unfortunately, `scikit-learn` currently does not have the functionality to draw dendrograms. However, you can generate them easily using SciPy. The SciPy clustering algorithms have a slightly different interface to the `scikit-learn` clustering algorithms. SciPy provides a function that takes a data array `X` and computes a *linkage array*, which encodes hierarchical cluster similarities. We can then feed this linkage array into the scipy `dendrogram` function to plot the dendrogram (Figure 3-36):

In[64]:

```
# Import the dendrogram function and the ward clustering function from SciPy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# Apply the ward clustering to the data array X
# The SciPy ward function returns an array that specifies the distances
# bridged when performing agglomerative clustering
linkage_array = ward(X)
```

```
# Now we plot the dendrogram for the linkage_array containing the distances
# between clusters
dendrogram(linkage_array)

# Mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

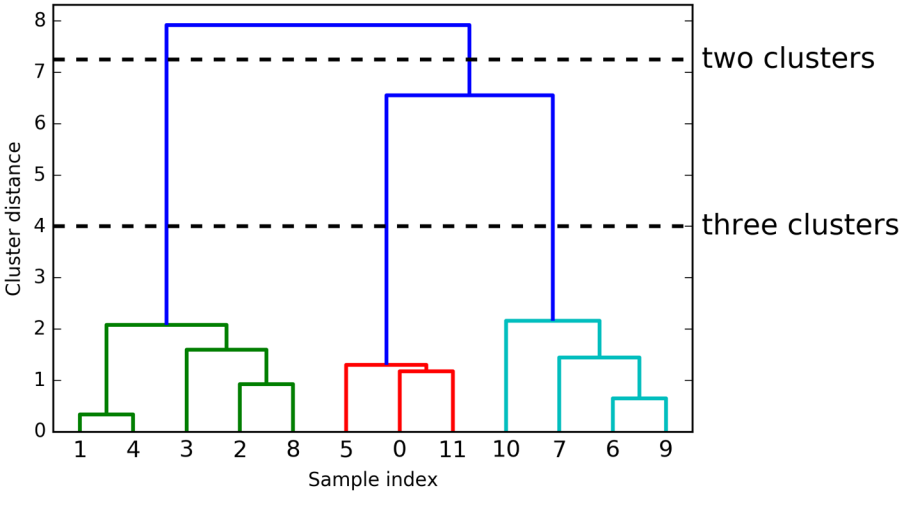

Figure 3-36. Dendrogram of the clustering shown in Figure 3-35 with lines indicating splits into two and three clusters

The dendrogram shows data points as points on the bottom (numbered from 0 to 11). Then, a tree is plotted with these points (representing single-point clusters) as the leaves, and a new node parent is added for each two clusters that are joined.

Reading from bottom to top, the data points 1 and 4 are joined first (as you could see in Figure 3-33). Next, points 6 and 9 are joined into a cluster, and so on. At the top level, there are two branches, one consisting of points 11, 0, 5, 10, 7, 6, and 9, and the other consisting of points 1, 4, 3, 2, and 8. These correspond to the two largest clusters in the lefthand side of the plot.

The y-axis in the dendrogram doesn't just specify when in the agglomerative algorithm two clusters get merged. The length of each branch also shows how far apart the merged clusters are. The longest branches in this dendrogram are the three lines that are marked by the dashed line labeled "three clusters." That these are the longest branches indicates that going from three to two clusters meant merging some very far-apart points. We see this again at the top of the chart, where merging the two remaining clusters into a single cluster again bridges a relatively large distance.

Unfortunately, agglomerative clustering still fails at separating complex shapes like the `two_moons` dataset. But the same is not true for the next algorithm we will look at, DBSCAN.

DBSCAN

Another very useful clustering algorithm is DBSCAN (which stands for "density-based spatial clustering of applications with noise"). The main benefits of DBSCAN are that it does not require the user to set the number of clusters *a priori*, it can capture clusters of complex shapes, and it can identify points that are not part of any cluster. DBSCAN is somewhat slower than agglomerative clustering and *k*-means, but still scales to relatively large datasets.

DBSCAN works by identifying points that are in "crowded" regions of the feature space, where many data points are close together. These regions are referred to as *dense* regions in feature space. The idea behind DBSCAN is that clusters form dense regions of data, separated by regions that are relatively empty.

Points that are within a dense region are called *core samples* (or core points), and they are defined as follows. There are two parameters in DBSCAN: `min_samples` and `eps`. If there are at least `min_samples` many data points within a distance of `eps` to a given data point, that data point is classified as a core sample. Core samples that are closer to each other than the distance `eps` are put into the same cluster by DBSCAN.

The algorithm works by picking an arbitrary point to start with. It then finds all points with distance `eps` or less from that point. If there are less than `min_samples` points within distance `eps` of the starting point, this point is labeled as *noise*, meaning that it doesn't belong to any cluster. If there are more than `min_samples` points within a distance of `eps`, the point is labeled a core sample and assigned a new cluster label. Then, all neighbors (within `eps`) of the point are visited. If they have not been assigned a cluster yet, they are assigned the new cluster label that was just created. If they are core samples, their neighbors are visited in turn, and so on. The cluster grows until there are no more core samples within distance `eps` of the cluster. Then another point that hasn't yet been visited is picked, and the same procedure is repeated.