**Clustering Algorithms:**

How to use data to make predictions on new data points? This is called “supervised learning.” Sometimes, however, rather than ‘making predictions’, we instead want to categorize data into buckets. This is termed “unsupervised learning.”

**Example of Supervised Learning**

To illustrate the difference, let’s say we’re at a major pizza chain and we’ve been tasked with creating a feature in the order management software that will predict delivery times for customers. In order to achieve this, we are given a dataset that has delivery times, distances traveled, day of week, time of day, staff on hand, and volume of sales for several deliveries in the past. From this data, we can make predictions on future delivery times. This is a good example of supervised learning.

**Example of Unsupervised Learning**

Now, let’s say the pizza chain wants to send out targeted coupons to customers. It wants to segment its customers into 4 groups: large families, small families, singles, and college students. We are given prior ordering data (e.g. size of order, price, frequency, etc) and we’re tasked with putting each customer into one of the four buckets. This would be an example of “unsupervised learning” since we’re not making predictions; we’re merely categorizing the customers into groups.

Clustering is one of the most frequently utilized forms of unsupervised learning. In this notes, we’ll explore two of the most common forms of clustering: k-means and hierarchical.

**Understanding the K-Means Clustering Algorithm**

We’ll create four random clusters using make\_blobs to aid in our task.

|  |
| --- |
| # import statements from sklearn.datasets import make\_blobs import numpy as np import matplotlib.pyplot as plt  # create blobs data = make\_blobs(n\_samples=200, n\_features=2, centers=4, cluster\_std=1.6, random\_state=50)  # create np array for data points points = data[0]  # create scatter plot plt.scatter(data[0][:,0], data[0][:,1], c=data[1], cmap='viridis') plt.xlim(-15,15) plt.ylim(-15,15)  plt.show() |

**Implementing K-Means Clustering in Python**

# import KMeans  
from sklearn.cluster import KMeans

# create kmeans object  
kmeans = KMeans(n\_clusters=4)

# fit kmeans object to data  
kmeans.fit(points)

# print location of clusters learned by kmeans object  
print(kmeans.cluster\_centers\_)

# save new clusters for chart  
y\_km = kmeans.fit\_predict(points)

**Now, we can see the results by running the following code in matplotlib.**

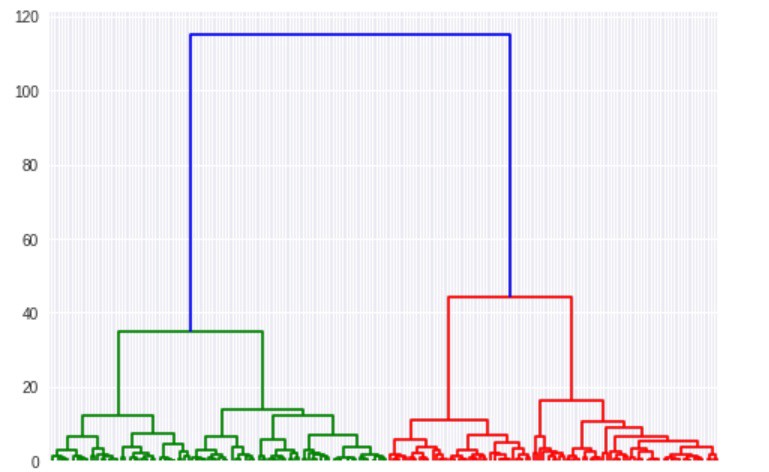
plt.scatter(points[y\_km ==0,0], points[y\_km == 0,1], s=100, c='red')  
plt.scatter(points[y\_km ==1,0], points[y\_km == 1,1], s=100, c='black')  
plt.scatter(points[y\_km ==2,0], points[y\_km == 2,1], s=100, c='blue')  
plt.scatter(points[y\_km ==3,0], points[y\_km == 3,1], s=100, c='cyan')

We have our 4 clusters. Note that the k-means++ algorithm did a better job than the plain ole’ k-means I ran in the example, as it nearly perfectly captured the boundaries of the initial clusters we created.

**Implementing Agglomerative Hierarchical Clustering**

Agglomerative hierarchical clustering differs from k-means in a key way. Rather than choosing a number of clusters and starting out with random centroids, we instead begin with every point in our dataset as a “cluster.” Then we find the two closest points and combine them into a cluster. Then, we find the next closest points, and those become a cluster. We repeat the process until we only have one big giant cluster.

Along the way, we create what’s called a dendrogram. This is our “history.” You can see the dendrogram for our data points below to get a sense of what’s happening.



The dendrogram plots out each cluster and the distance. We can use the dendrogram to find the clusters for any number we chose. In the dendrogram above, it’s easy to see the starting points for the first cluster (blue), the second cluster (red), and the third cluster (green). Only the first 3 are color-coded here, but if you look over at the red side of the dendrogram, you can spot the starting point for the 4th cluster as well. The dendrogram runs all the way until every point is its own individual cluster.

# import hierarchical clustering libraries  
import scipy.cluster.hierarchy as sch  
from sklearn.cluster import AgglomerativeClustering

Now, let’s create our dendrogram (which I’ve already shown you above), determine how many clusters we want, and save the data points from those clusters to chart them out.

# create dendrogram  
dendrogram = sch.dendrogram(sch.linkage(points, method='ward'))

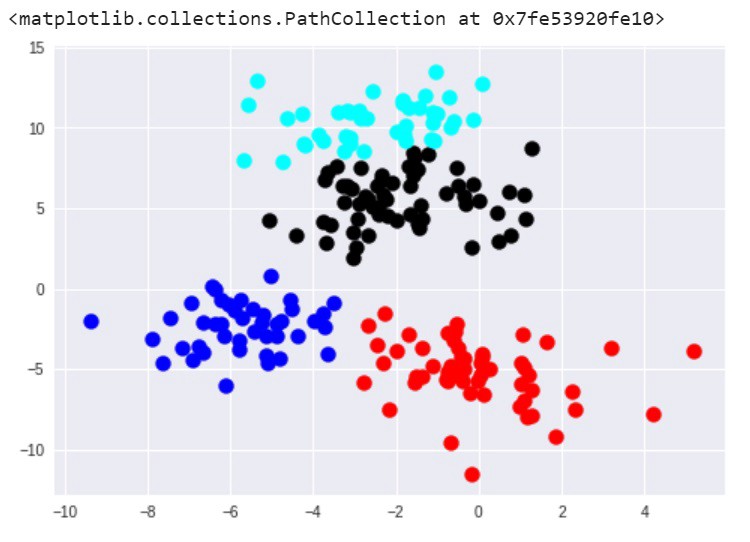
# create clusters  
hc = AgglomerativeClustering(n\_clusters=4, affinity = 'euclidean', linkage = 'ward')

# save clusters for chart  
y\_hc = hc.fit\_predict(points)

Now, we’ll do as we did with the k-means algorithm and see our clusters using matplotlib.

plt.scatter(points[y\_hc ==0,0], points[y\_hc == 0,1], s=100, c='red')  
plt.scatter(points[y\_hc==1,0], points[y\_hc == 1,1], s=100, c='black')  
plt.scatter(points[y\_hc ==2,0], points[y\_hc == 2,1], s=100, c='blue')  
plt.scatter(points[y\_hc ==3,0], points[y\_hc == 3,1], s=100, c='cyan')

Here are the results:



In this instance, the results between k-means and hierarchical clustering were pretty similar. This is not always the case, however. In general, the advantage of agglomerative hierarchical clustering is that it tends to produce more accurate results. The downside is that hierarchical clustering is more difficult to implement and more time/resource consuming than k-means.