Jake Huneycutt Follow

May 29, 2018 · 6 min read · Listen

An Introduction to Clustering Algorithms in Python

In data science, we often think about 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.”

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.

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.

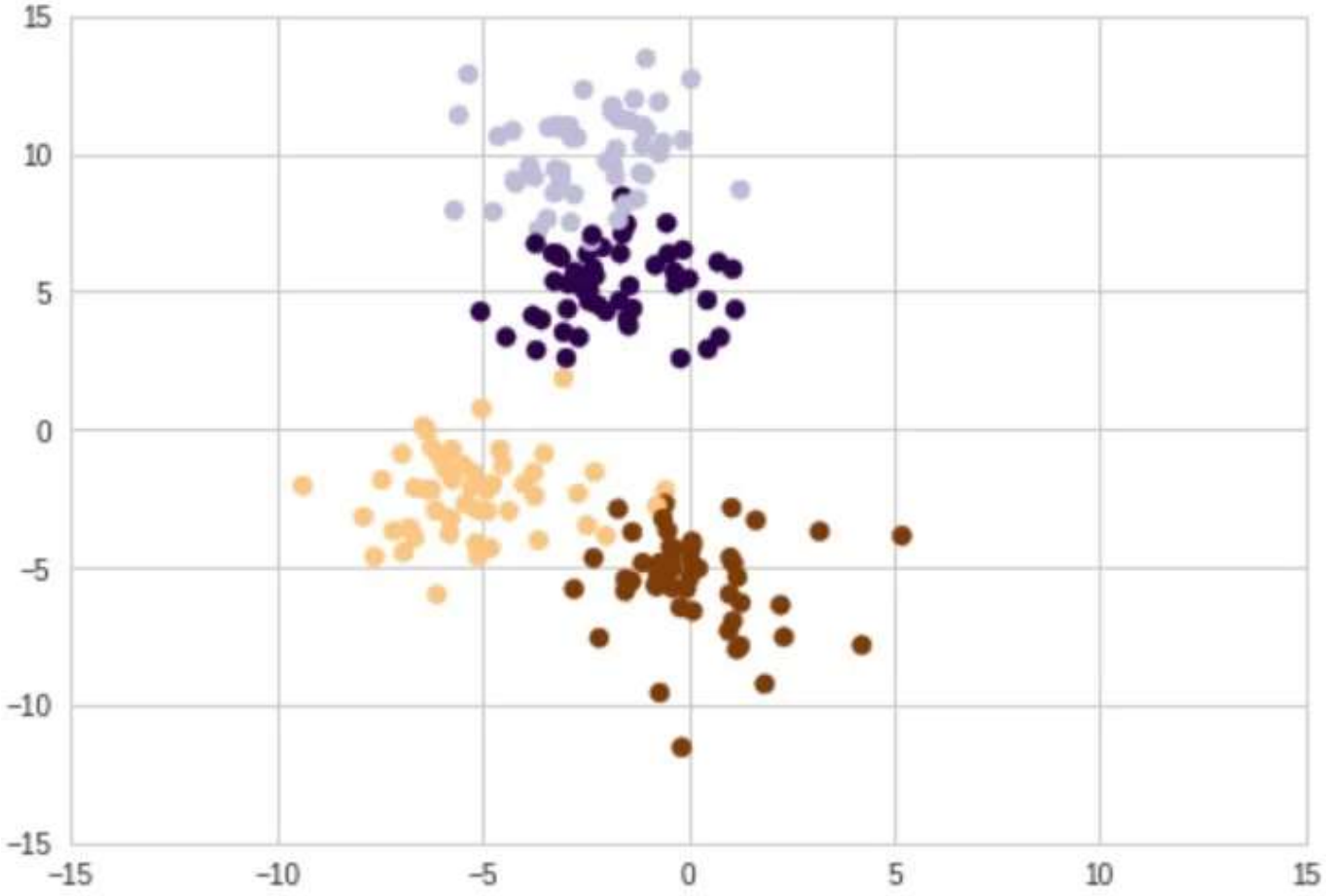
Clustering is one of the most frequently utilized forms of unsupervised learning. In this article, we'll explore two of the most common forms of clustering: k-means and hierarchical.

Understanding the K-Means Clustering Algorithm

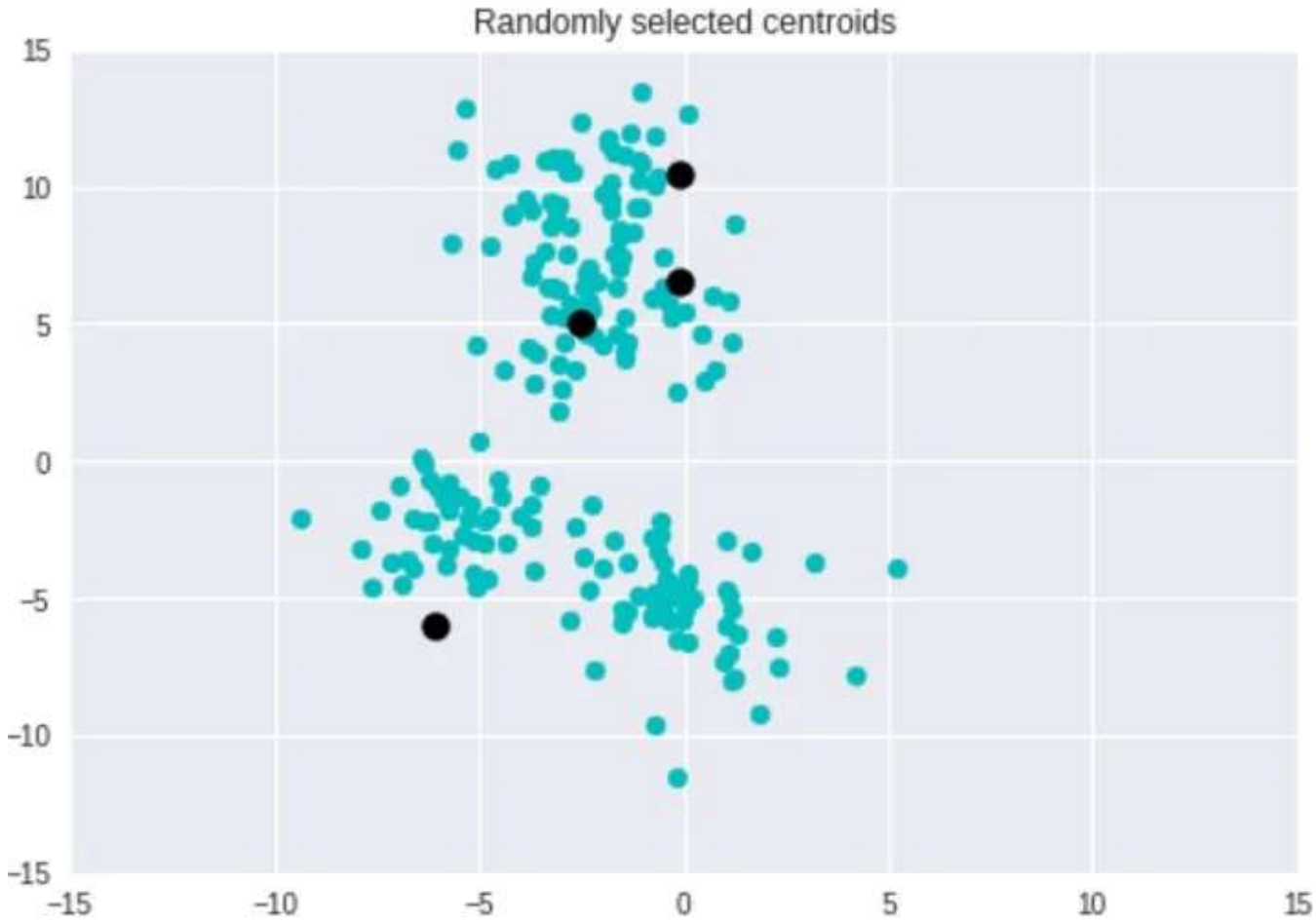
Let's look at how k-means clustering works. First, let me introduce you to my good friend, blobby; i.e. the make_blobs function in Python's sci-kit learn library. We'll create four random clusters using make_blobs to aid in our task.

```
# 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)
```

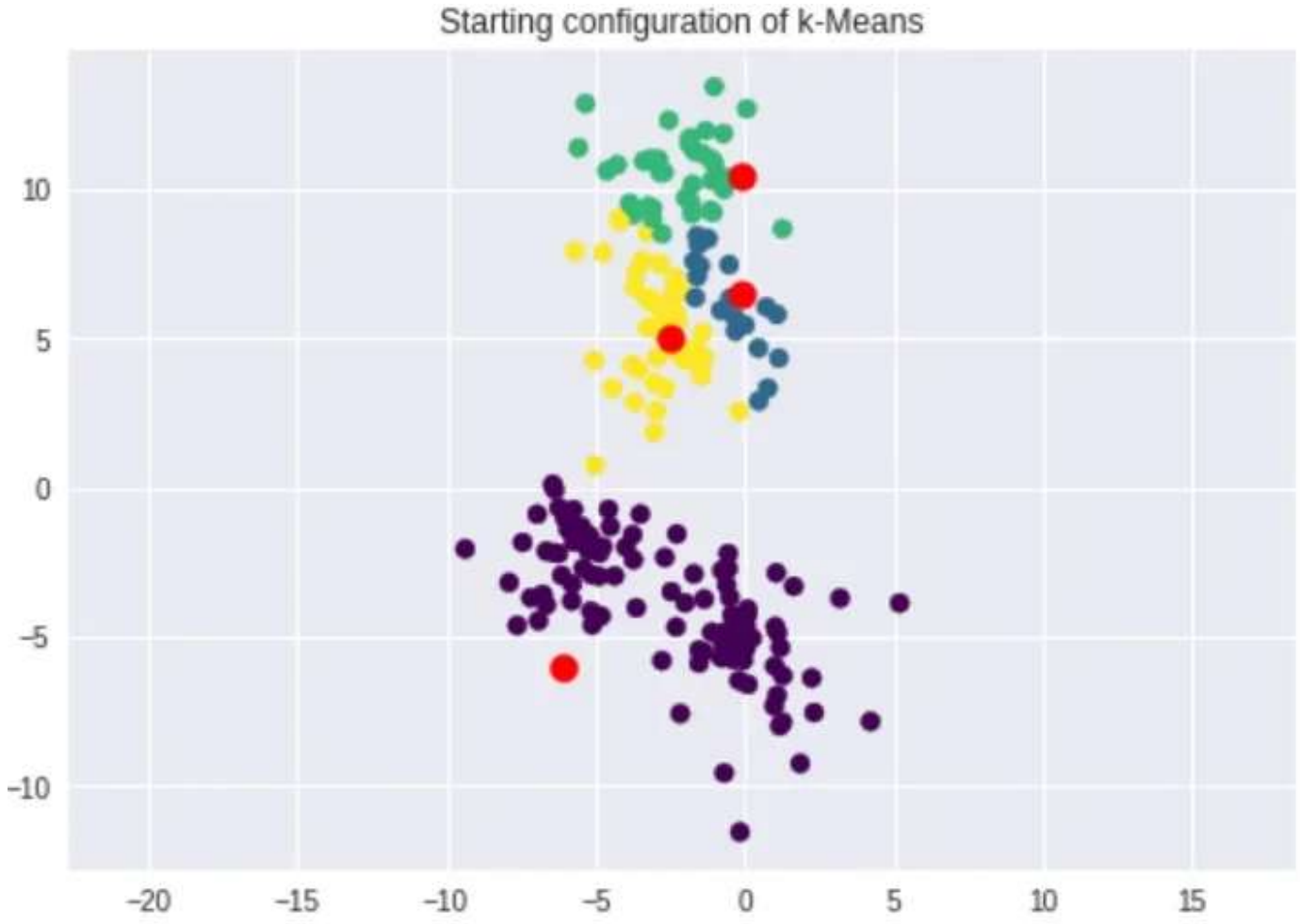
You can see our "blobs" below:



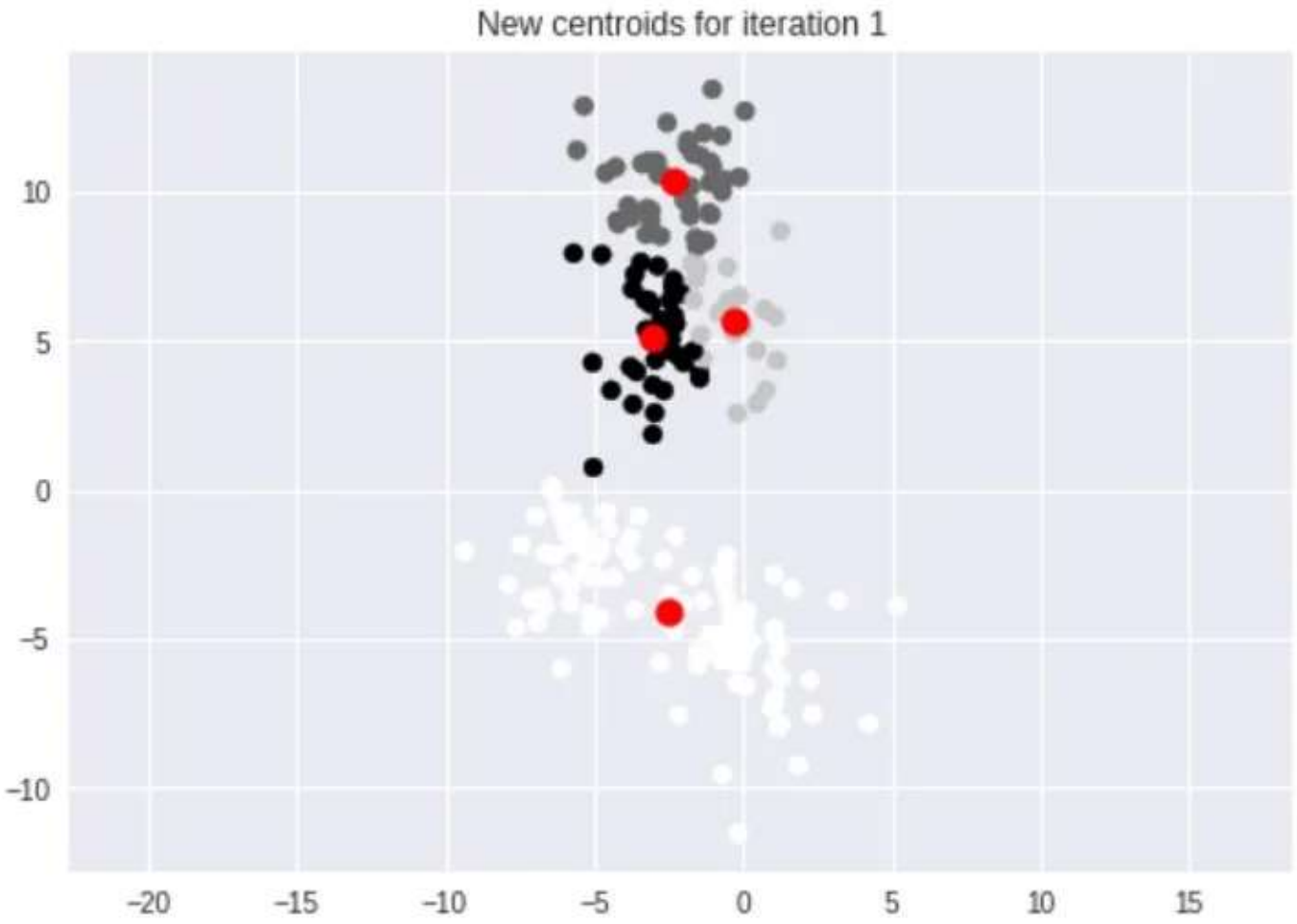
We have four colored clusters, but there is some overlap with the two clusters on top, as well as the two clusters on the bottom. The first step in k-means clustering is to select random centroids. Since our k=4 in this instance, we'll need 4 random centroids. Here is how it looked in my implementation from scratch.



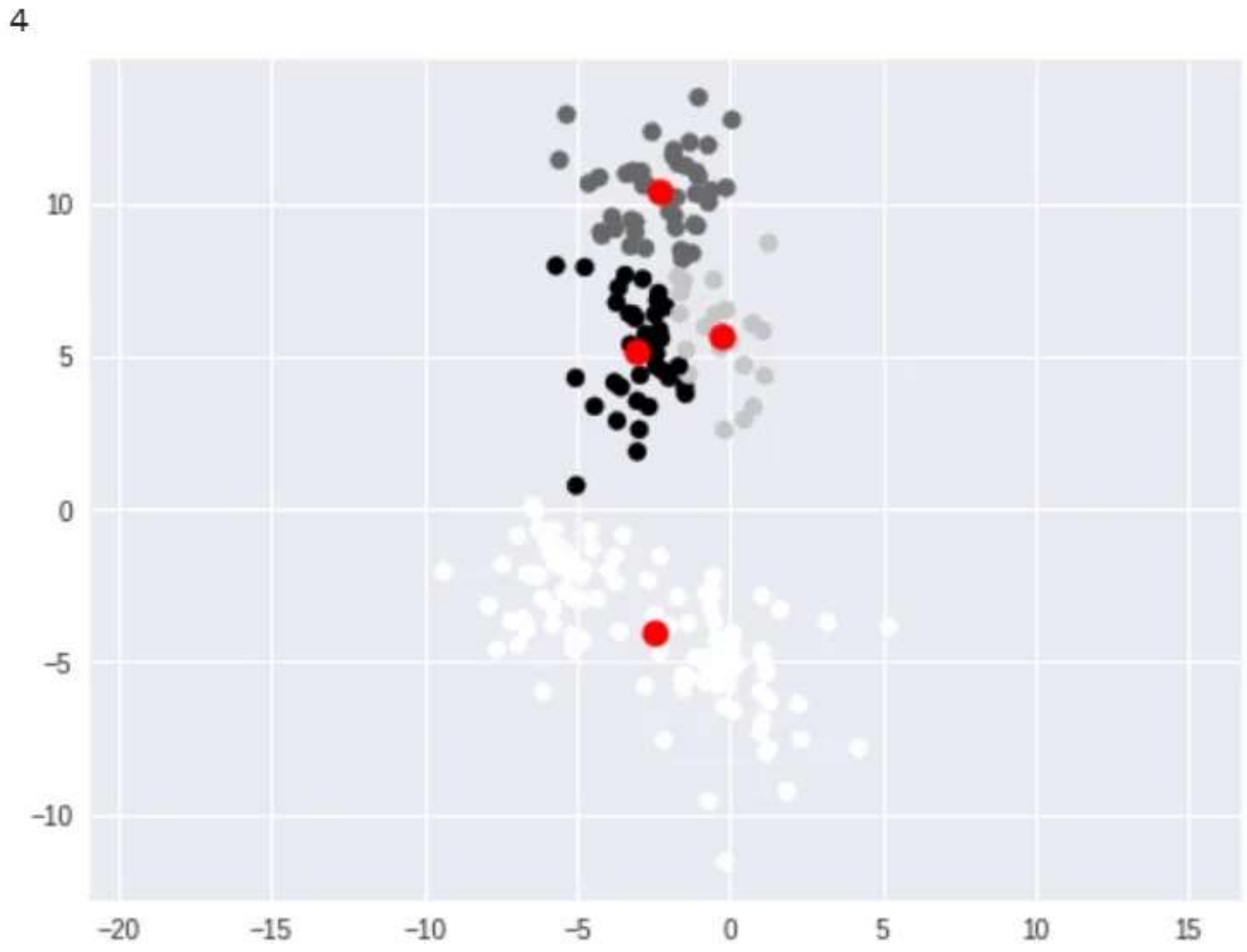
Next, we take each point and find the nearest centroid. There are different ways to measure distance, but I used Euclidean distance, which can be measured using np.linalg.norm in Python.



Now that we have 4 clusters, we find the new centroids of the clusters.



Then we match each point to the closest centroid again, repeating the process, until we can improve the clusters no more. In this case, when the process finished, I ended up with the result below.



Note that these clusters are a bit different than my original clusters. This is the result of the random initialization trap. Essentially, our starting centroids can dictate the location of our clusters in k-mean clustering.

This isn't the result we wanted, but one way to combat this is with the k-means ++ algorithm, which provides better initial seeding in order to find the best clusters. Fortunately, this is automatically done in k-means implementation we'll be using in Python.

Implementing K-Means Clustering in Python

To run k-means in Python, we'll need to import KMeans from sci-kit learn.

```
# import KMeans
from sklearn.cluster import KMeans
```

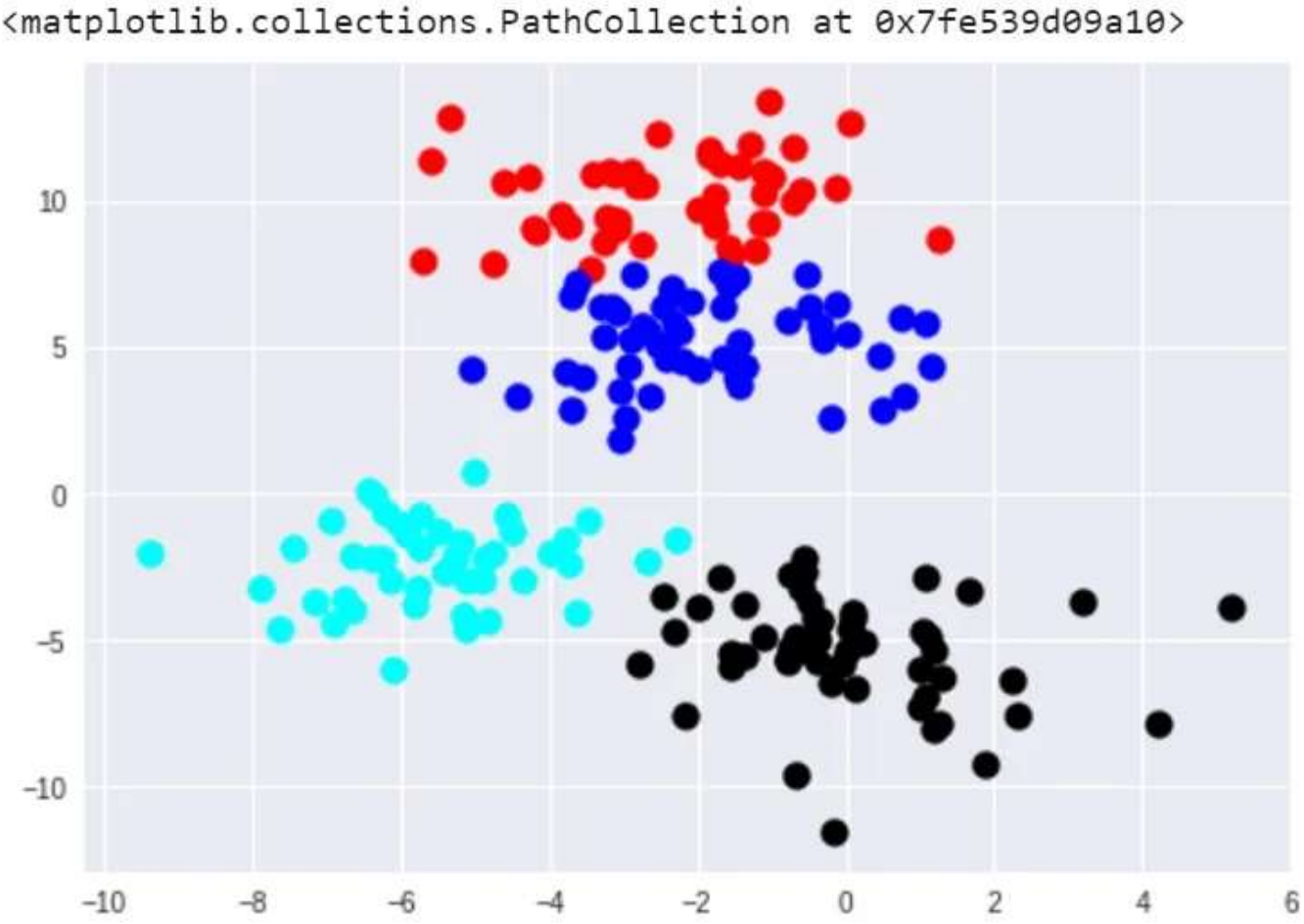
Note that in the documentation, k-means ++ is the default, so we don't need to make any changes in order to run this improved methodology. Now, let's run k-means on our blobs (which were put into a numpy array called 'points').

```
# 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')
```

And voila! 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.

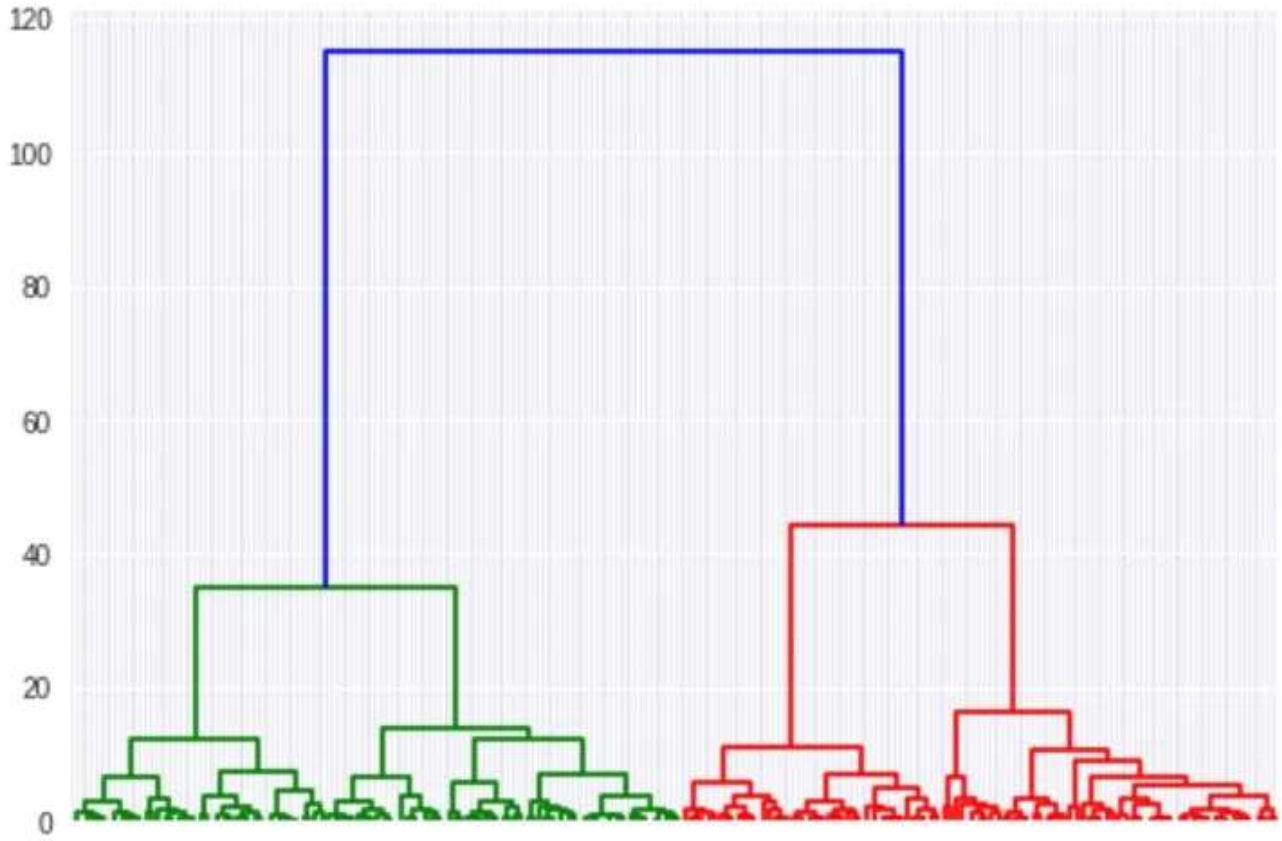


K-means is the most frequently used form of clustering due to its speed and simplicity. Another very common clustering method is hierarchical clustering.

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.

Let's see how agglomerative hierarchical clustering works in Python. First, let's import the necessary libraries from scipy.cluster.hierarchy and sklearn.clustering.

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

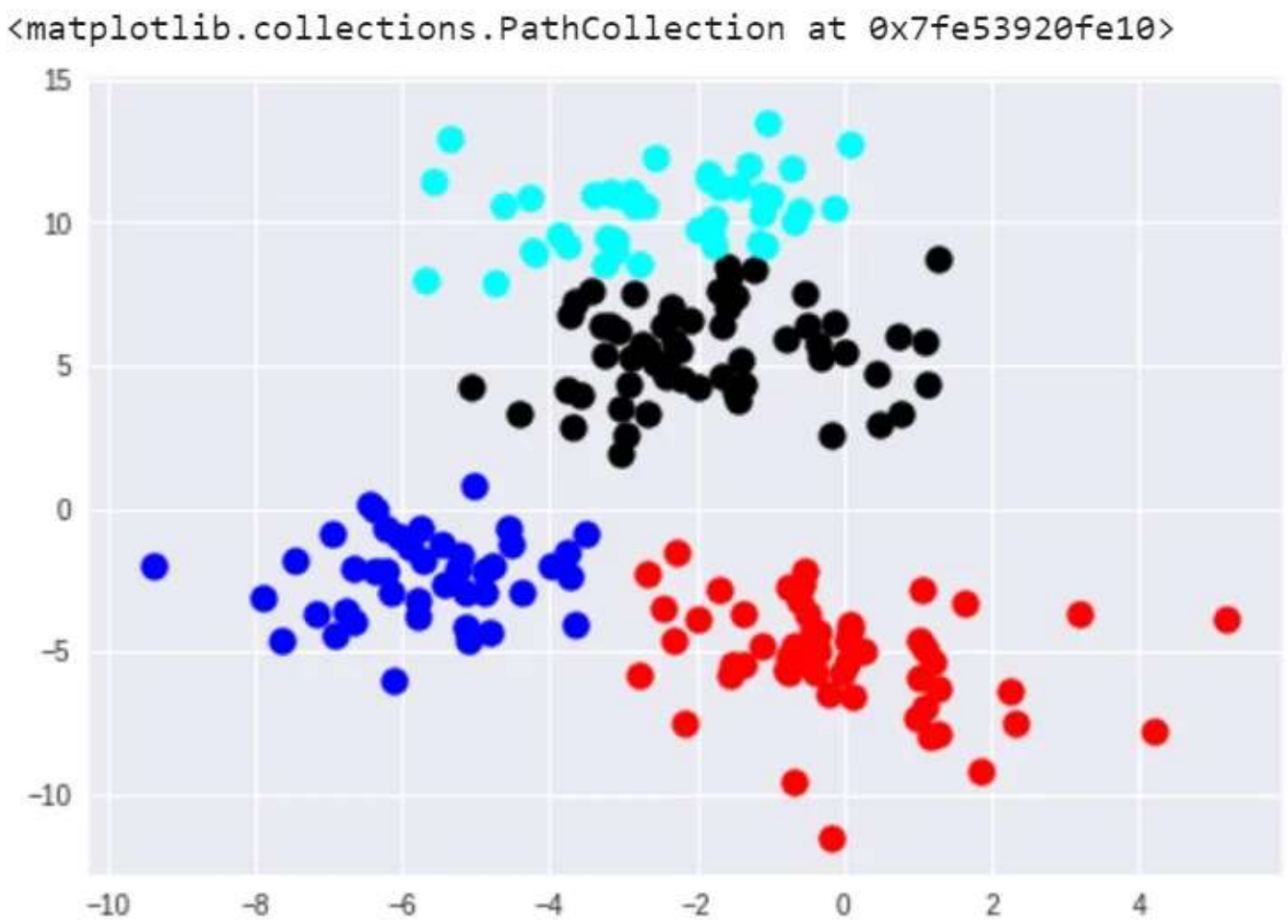
Now, let's create our dendrogram (which I've already shown you above), determine how many clusters we want, and save the data points from those clusters to chart them out.

```
# 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.

Further Reading

If you want to know more about clustering, I highly recommend George Seif's article, "The 5 Clustering Algorithms Data Scientists Need to Know."

Additional Resources

1. G. James, D. Witten, et. al. *Introduction to Statistical Learning*, Chapter 10: Unsupervised Learning, Link (PDF)
2. Andrea Trevino, *Introduction to K-Means Clustering*, Link

3. Kirill Eremenko, *Machine Learning A-Z* (Udemy course), Link

Machine Learning Data Science Python Clustering Unsupervised Learning

---