**K-means clustering: algorithm, initialization, and convergence.**

**What is K-means Clustering?**

[Unsupervised Machine Learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/)is the process of teaching a computer to use unlabeled, unclassified data and enabling the algorithm to operate on that data without supervision. Without any previous data training, the machine’s job in this case is to organize unsorted data according to parallels, patterns, and variations.

K means clustering, assigns data points to one of the K clusters depending on their distance from the center of the clusters. It starts by randomly assigning the clusters centroid in the space. Then each data point assign to one of the cluster based on its distance from centroid of the cluster. After assigning each point to one of the cluster, new cluster centroids are assigned. This process runs iteratively until it finds good cluster. In the analysis we assume that number of cluster is given in advanced and we have to put points in one of the group.

In some cases, K is not clearly defined, and we have to think about the optimal number of K. K Means clustering performs best data is well separated. When data points overlapped this clustering is not suitable. K Means is faster as compare to other clustering technique. It provides strong coupling between the data points. K Means cluster do not provide clear information regarding the quality of clusters. Different initial assignment of cluster centroid may lead to different clusters. Also, K Means algorithm is sensitive to noise. It maymhave stuck in local minima.

**What is the objective of k-means clustering?**

The goal of [clustering](https://www.geeksforgeeks.org/clustering-in-machine-learning/) is to divide the population or[set](https://www.geeksforgeeks.org/set-in-cpp-stl/) of data points into a number of groups so that the data points within each group are more[comparable](https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/) to one another and different from the data points within the other groups. It is essentially a grouping of things based on how similar and different they are to one another.

**How k-means clustering works?**

We are given a data set of items, with certain features, and values for these features (like a vector). The task is to categorize those items into groups. To achieve this, we will use the K-means algorithm, an unsupervised learning algorithm. ‘K’ in the name of the algorithm represents the number of groups/clusters we want to classify our items into.

(It will help if you think of items as points in an n-dimensional space). The algorithm will categorize the items into k groups or clusters of similarity. To calculate that similarity, we will use the Euclidean distance as a measurement.

The algorithm works as follows:

1. First, we randomly initialize k points, called means or cluster centroids.
2. We categorize each item to its closest mean, and we update the mean’s coordinates, which are the averages of the items categorized in that cluster so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

The “points” mentioned above are called means because they are the mean values of the items categorized in them. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature *x,* the items have values in [0,3], we will initialize the means with values for *x* at [0,3]).

The above algorithm in pseudocode is as follows:

Initialize k means with random values  
--> For a given number of iterations:  
   
 --> Iterate through items:  
   
 --> Find the mean closest to the item by calculating   
 the euclidean distance of the item with each of the means  
   
 --> Assign item to mean  
   
 --> Update mean by shifting it to the average of the items in that cluster

**Implementation of K-Means Clustering in Python**

**Example 1**

**Import the necessary Libraries**

We are importing[Numpy](https://www.geeksforgeeks.org/numpy-in-python-set-1-introduction/) for statistical computations,[Matplotlib](https://www.geeksforgeeks.org/matplotlib-tutorial/) to plot the[graph,](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) and make\_blobs from sklearn.datasets.

* Python3

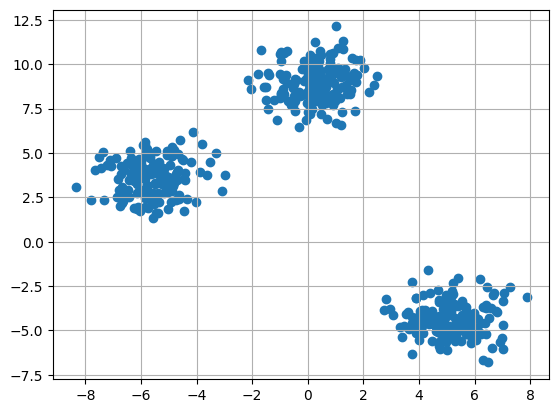
|  |
| --- |
| **import** numpy as np  **import** matplotlib.pyplot as plt  **from** sklearn.datasets **import** make\_blobs |

**Create the custom dataset with make\_blobs and plot it**

* Python3

|  |
| --- |
| X,y **=** make\_blobs(n\_samples **=** 500,n\_features **=** 2,centers **=** 3,random\_state **=** 23)    fig **=** plt.figure(0)  plt.grid(True)  plt.scatter(X[:,0],X[:,1])  plt.show() |

**Output**:



*Clustering dataset*

**Initialize the random centroids**

The code initializes three clusters for K-means clustering. It sets a random seed and generates random cluster centers within a specified range, and creates an empty[list](https://www.geeksforgeeks.org/list-cpp-stl/) of points for each cluster.

* Python3

|  |
| --- |
| k **=** 3    clusters **=** {}  np.random.seed(23)    **for** idx **in** range(k):      center **=** 2**\***(2**\***np.random.random((X.shape[1],))**-**1)      points **=** []      cluster **=** {          'center' : center,          'points' : []      }        clusters[idx] **=** cluster    clusters |

**Output:**

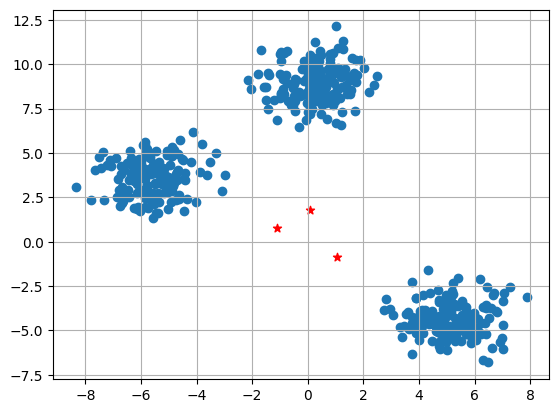
{0: {'center': array([0.06919154, 1.78785042]), 'points': []},  
 1: {'center': array([ 1.06183904, -0.87041662]), 'points': []},  
 2: {'center': array([-1.11581855, 0.74488834]), 'points': []}}

**Plot the random initialize center with data points**

* Python3

|  |
| --- |
| plt.scatter(X[:,0],X[:,1])  plt.grid(True)  **for** i **in** clusters:      center **=** clusters[i]['center']      plt.scatter(center[0],center[1],marker **=** '\*',c **=** 'red')  plt.show() |

**Output**:



*Data points with random center*

The plot displays a scatter plot of data points (X[:,0], X[:,1]) with grid lines. It also marks the initial cluster centers (red stars) generated for K-means clustering.

**Define Euclidean distance**

* Python3

|  |
| --- |
| **def** distance(p1,p2):  **return** np.sqrt(np.sum((p1**-**p2)**\*\***2)) |

**Create the function to Assign and Update the cluster center**

The E-step assigns data points to the nearest cluster center, and the M-step updates cluster centers based on the mean of assigned points in K-means clustering.

* Python3

|  |
| --- |
| #Implementing E step  **def** assign\_clusters(X, clusters):  **for** idx **in** range(X.shape[0]):          dist **=** []            curr\_x **=** X[idx]    **for** i **in** range(k):              dis **=** distance(curr\_x,clusters[i]['center'])              dist.append(dis)          curr\_cluster **=** np.argmin(dist)          clusters[curr\_cluster]['points'].append(curr\_x)  **return** clusters    #Implementing the M-Step  **def** update\_clusters(X, clusters):  **for** i **in** range(k):          points **=** np.array(clusters[i]['points'])  **if** points.shape[0] > 0:              new\_center **=** points.mean(axis **=**0)              clusters[i]['center'] **=** new\_center                clusters[i]['points'] **=** []  **return** clusters |

**Step 7: Create the function to Predict the cluster for the datapoints**

* Python3

|  |
| --- |
| **def** pred\_cluster(X, clusters):      pred **=** []  **for** i **in** range(X.shape[0]):          dist **=** []  **for** j **in** range(k):              dist.append(distance(X[i],clusters[j]['center']))          pred.append(np.argmin(dist))  **return** pred |

**Assign, Update, and predict the cluster center**

* Python3

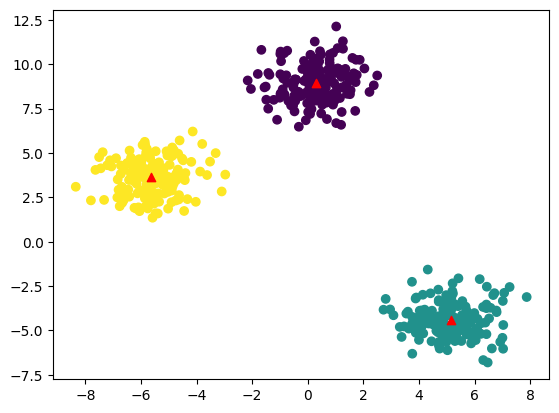
|  |
| --- |
| clusters **=** assign\_clusters(X,clusters)  clusters **=** update\_clusters(X,clusters)  pred **=** pred\_cluster(X,clusters) |

**Plot the data points with their predicted cluster center**

* Python3

|  |
| --- |
| plt.scatter(X[:,0],X[:,1],c **=** pred)  **for** i **in** clusters:      center **=** clusters[i]['center']      plt.scatter(center[0],center[1],marker **=** '^',c **=** 'red')  plt.show() |

**Output**:



*K-means Clustering*

The plot shows data points colored by their predicted clusters. The red markers represent the updated cluster centers after the E-M steps in the K-means clustering algorithm.

**Example 2**

**Import the necessary libraries**

* Python3

|  |
| --- |
| **import** pandas as pd  **import** numpy as np  **import** seaborn as sns  **import** matplotlib.pyplot as plt  **import** matplotlib.cm as cm  **from** sklearn.datasets **import** load\_iris  **from** sklearn.cluster **import** KMeans |

**Load the Dataset**

* Python3

|  |
| --- |
| X, y **=** load\_iris(return\_X\_y**=**True) |

**Elbow Method**

Finding the ideal number of groups to divide the data into is a basic stage in any unsupervised algorithm. One of the most common techniques for figuring out this ideal value of k is the elbow approach.

* Python3

|  |
| --- |
| #Find optimum number of cluster  sse **=** [] #SUM OF SQUARED ERROR  **for** k **in** range(1,11):      km **=** KMeans(n\_clusters**=**k, random\_state**=**2)      km.fit(X)      sse.append(km.inertia\_) |

**Plot the Elbow graph to find the optimum number of cluster**

* Python3

|  |
| --- |
| sns.set\_style("whitegrid")  g**=**sns.lineplot(x**=**range(1,11), y**=**sse)    g.set(xlabel **=**"Number of cluster (k)",        ylabel **=** "Sum Squared Error",        title **=**'Elbow Method')    plt.show() |

**Output:**

A graph with a line

Description automatically generated

*Elbow Method*

From the above graph, we can observe that at k=2 and k=3 elbow-like situation. So, we are considering K=3

**Build the Kmeans clustering model**

* Python3

|  |
| --- |
| kmeans **=** KMeans(n\_clusters **=** 3, random\_state **=** 2)  kmeans.fit(X) |

**Output:**

KMeans  
KMeans(n\_clusters=3, random\_state=2)

**Find the cluster center**

* Python3

|  |
| --- |
| kmeans.cluster\_centers\_ |

**Output:**

array([[5.006 , 3.428 , 1.462 , 0.246 ],  
 [5.9016129 , 2.7483871 , 4.39354839, 1.43387097],  
 [6.85 , 3.07368421, 5.74210526, 2.07105263]])

**Predict the cluster group:**

* Python3

|  |
| --- |
| pred **=** kmeans.fit\_predict(X)  pred |

**Output:**

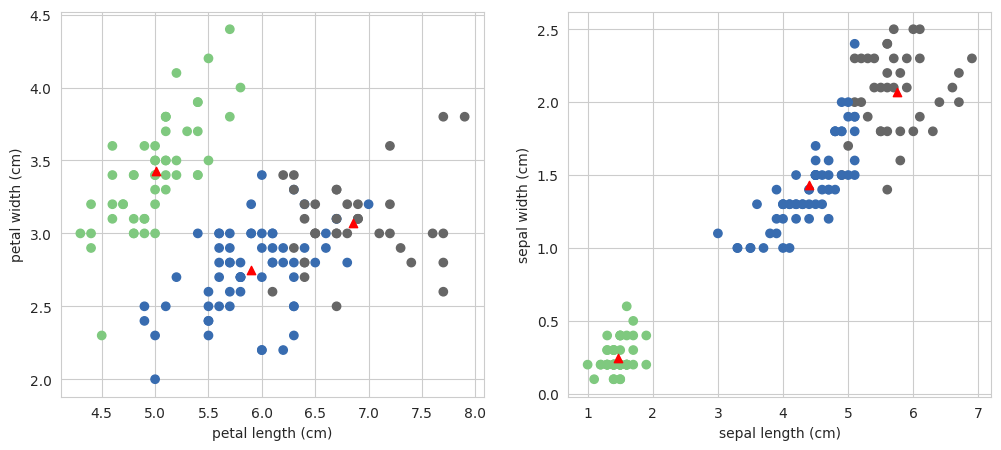
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 0, 0, 0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2,  
 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2,  
 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1], dtype=int32)

**Plot the cluster center with data points**

* Python3

|  |
| --- |
| plt.figure(figsize**=**(12,5))  plt.subplot(1,2,1)  plt.scatter(X[:,0],X[:,1],c **=** pred, cmap**=**cm.Accent)  plt.grid(True)  **for** center **in** kmeans.cluster\_centers\_:      center **=** center[:2]      plt.scatter(center[0],center[1],marker **=** '^',c **=** 'red')  plt.xlabel("petal length (cm)")  plt.ylabel("petal width (cm)")    plt.subplot(1,2,2)  plt.scatter(X[:,2],X[:,3],c **=** pred, cmap**=**cm.Accent)  plt.grid(True)  **for** center **in** kmeans.cluster\_centers\_:      center **=** center[2:4]      plt.scatter(center[0],center[1],marker **=** '^',c **=** 'red')  plt.xlabel("sepal length (cm)")  plt.ylabel("sepal width (cm)")  plt.show() |

**Output:**



*K-means clustering*

The subplot on the left display petal length vs. petal width with data points colored by clusters, and red markers indicate K-means cluster centers. The subplot on the right show sepal length vs. sepal width similarly.

**Hierarchical clustering: agglomerative and divisive approaches.**

Hierarchical clustering is a popular [unsupervised machine learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/) technique used to group similar data points into clusters based on their similarity or dissimilarity. It is called “hierarchical” because it creates a tree-like hierarchy of clusters, where each node represents a cluster that can be further divided into smaller sub-clusters.

There are two types of [hierarchical clustering](https://www.geeksforgeeks.org/hierarchical-clustering-in-data-mining/) techniques:

1. Agglomerative and
2. Divisive clustering

**Agglomerative Clustering**

[Agglomerative clustering](https://www.geeksforgeeks.org/agglomerative-methods-in-machine-learning/) is a type of hierarchical clustering algorithm that merges the most similar pairs of data points or clusters, building a hierarchy of clusters until all the data points belong to a single cluster. It starts with each data point as its own cluster and then iteratively merges the most similar pairs of clusters until all data points belong to a single cluster

**Divisive Clustering**

Divisive Clustering is the technique that starts with all data points in a single cluster and recursively splits the clusters into smaller sub-clusters based on their dissimilarity. It is also known as, “top-down” clustering. It starts with all data points in a single cluster, and then recursively splits the clusters into smaller sub-clusters based on their dissimilarity.

Unlike agglomerative clustering, which starts with each data point as its own cluster and iteratively merges the most similar pairs of clusters, divisive clustering is a “divide and conquer” approach that breaks a large cluster into smaller sub-clusters

**Example 1:**

Here is a short example of agglomerative clustering using randomly generated data in Python –

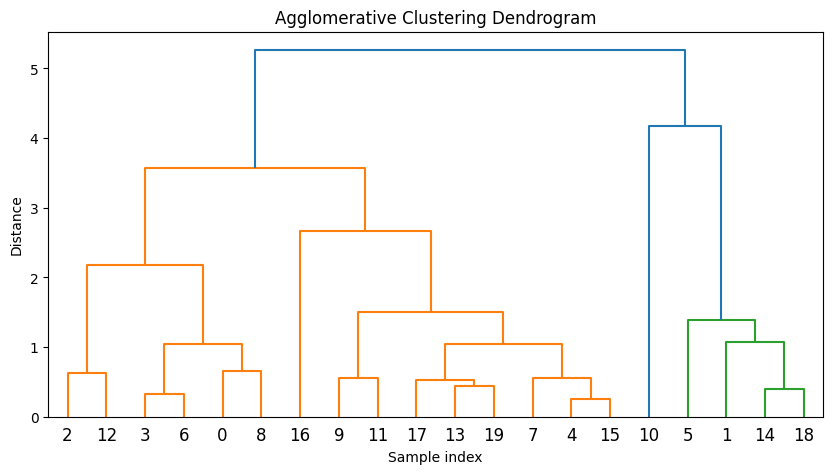
In this example, we first create a random dataset with 50 samples and two features using NumPy’s randn function. Then, we use the linkage function from SciPy’s cluster.hierarchy module to perform hierarchical clustering using complete linkage method. The resulting linkage matrix Z contains information about the cluster merging process.

Finally, we plot the dendrogram using the dendrogram function from the same module. The dendrogram shows how the clusters were merged and at what distance, starting from individual samples at the bottom and ending with a single cluster at the top.

* Python

|  |
| --- |
| # Import the necessary libraries  **import** numpy as np  **import** matplotlib.pyplot as plt  **from** scipy.cluster.hierarchy **import** dendrogram, linkage    # Create a random dataset with two features and 50 samples  np.random.seed(0)  x **=** np.random.randn(20, 2)    # Perform hierarchical clustering using caomplete linkage  z **=** linkage(x, method**=**'complete')    # Plot the dendrogram  plt.figure(figsize**=**(10, 5))  plt.title('Agglomerative Clustering Dendrogram')  plt.xlabel('Sample index')  plt.ylabel('Distance')  dendrogram(z)  plt.show() |

**Output:**



*Agglomerative Clustering*

Note that there are other methods of linkage that can be used, such as single, average, and ward. The method used will affect the resulting dendrogram and clustering.

**Example 2:**

Here is a short example of agglomerative clustering using randomly generated data in Python –

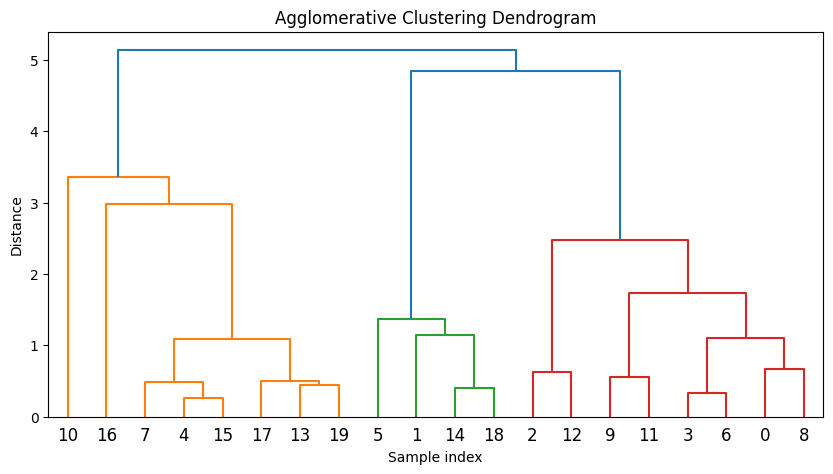
In this example, we first generate a sample dataset with 10 data points and 2 features using NumPy. We then perform agglomerative clustering using the linkage function from SciPy, which takes the data matrix as input along with the clustering method (ward in this case) and distance metric (euclidean in this case). The output of linkage is a linkage matrix that represents the hierarchical clustering structure.

We then plot the dendrogram using the dendrogram function from SciPy, which takes the linkage matrix as input along with the plotting axis (ax). We set the color threshold to 0 to display all clusters in the dendrogram.

* Python

|  |
| --- |
| # Import the necessary libraries  **from** scipy.cluster.hierarchy **import** dendrogram, linkage  **import** numpy as np  **import** matplotlib.pyplot as plt    # Generate sample data  np.random.seed(0)  X **=** np.random.randn(20, 2)    # Perform divisive clustering  Z **=** linkage(X, method**=**'ward', metric**=**'euclidean')    # Plot dendrogram  # Plot the dendrogram  plt.figure(figsize**=**(10, 5))  plt.title('Agglomerative Clustering Dendrogram')  plt.xlabel('Sample index')  plt.ylabel('Distance')  dendrogram(Z)  plt.show() |

**Output:**



*Agglomerative clustering*

The resulting plot will show the hierarchical clustering structure with each data point as a leaf node in the dendrogram and the clusters at different levels of the hierarchy.

**Difference between agglomerative clustering and Divisive clustering :**

| **S.No.** | **Parameters** | **Agglomerative Clustering** | **Divisive Clustering** |
| --- | --- | --- | --- |
| 1. | **Category** | Bottom-up approach | Top-down approach |
| 2. | **Approach** | each data point starts in its own cluster, and the algorithm recursively merges the closest pairs of clusters until a single cluster containing all the data points is obtained. | all data points start in a single cluster, and the algorithm recursively splits the cluster into smaller sub-clusters until each data point is in its own cluster. |
| 3. | **Complexity    level** | Agglomerative clustering is generally more computationally expensive, especially for large datasets as this approach requires the calculation of all pairwise distances between data points, which can be computationally expensive. | Comparatively less expensive as divisive clustering only requires the calculation of distances between sub-clusters, which can reduce the computational burden. |
| 4. | **Outliers** | Agglomerative clustering can handle outliers better than divisive clustering since outliers can be absorbed into larger clusters | divisive clustering may create sub-clusters around outliers, leading to suboptimal clustering results. |
| 5. | **Interpretability** | Agglomerative clustering tends to produce more interpretable results since the dendrogram shows the merging process of the clusters, and the user can choose the number of clusters based on the desired level of granularity. | divisive clustering can be more difficult to interpret since the dendrogram shows the splitting process of the clusters, and the user must choose a stopping criterion to determine the number of clusters. |
| 6. | **Implementation** | Scikit-learn provides multiple linkage methods for agglomerative clustering, such as “ward,” “complete,” “average,” and “single,” | divisive clustering is not currently implemented in Scikit-learn. |
| 7. | **Example** | Here are some of the applications in which Agglomerative Clustering is used :  Image segmentation, Customer segmentation, Social network analysis, Document clustering, Genetics, genomics, etc., and many more. | Here are some of the applications in which Divisive Clustering is used :  Market segmentation, Anomaly detection, Biological classification, Natural language processing, etc. |

**Density-based clustering: DBSCAN, OPTICS.**