**Clustering**

[[Deniz Gunay](https://medium.com/@denizgunay?source=post_page-----49f2894c1c95--------------------------------)](https://medium.com/@denizgunay?source=post_page-----49f2894c1c95--------------------------------)

[Deniz Gunay](https://medium.com/@denizgunay?source=post_page-----49f2894c1c95--------------------------------)

·

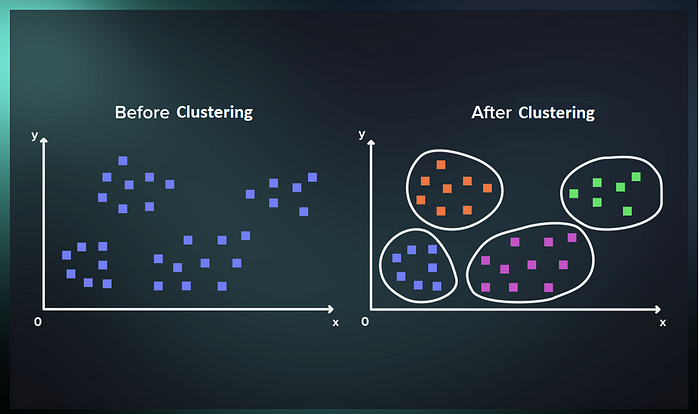
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff08c1e19344b&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40denizgunay%2Fclustering-49f2894c1c95&user=Deniz+Gunay&userId=f08c1e19344b&source=post_page-f08c1e19344b----49f2894c1c95---------------------post_header-----------)

20 min read

·

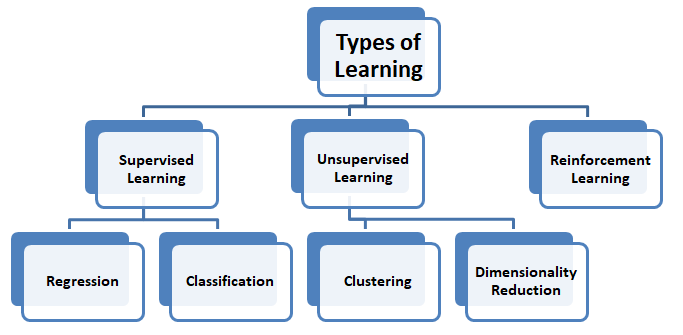
Sep 18, 2023

86



Clustering

In the realm of artificial intelligence and data science, two prominent methodologies reign supreme: supervised learning and unsupervised learning. These approaches serve as the foundation for building intelligent systems and extracting meaningful insights from vast datasets. In this article, we will delve into these fundamental concepts, explore the fascinating world of clustering, and then embark on a journey to understand hierarchical clustering and k-means, two key techniques that play pivotal roles in unsupervised learning.



Types of Learning

**Understanding Supervised and Unsupervised Learning**

Before we delve into the intricacies of clustering algorithms, let’s establish a solid foundation by defining supervised and unsupervised learning.

* **Supervised Learning**: This is a type of machine learning where the model learns from labeled data, making predictions or classifications based on input-output pairs. Essentially, the algorithm is provided with a dataset in which the correct answers or labels are already known, allowing it to map input data to the desired output. Common applications include image recognition, language translation, and spam email classification.
* **Unsupervised Learning**: In contrast, unsupervised learning operates in an environment where the algorithm explores data without labeled outcomes. It seeks to identify patterns, relationships, and structures within the data itself. Unsupervised learning is particularly useful when you want to discover hidden patterns, segment data, or reduce the dimensionality of your dataset.

**Unveiling the Power of Clustering**

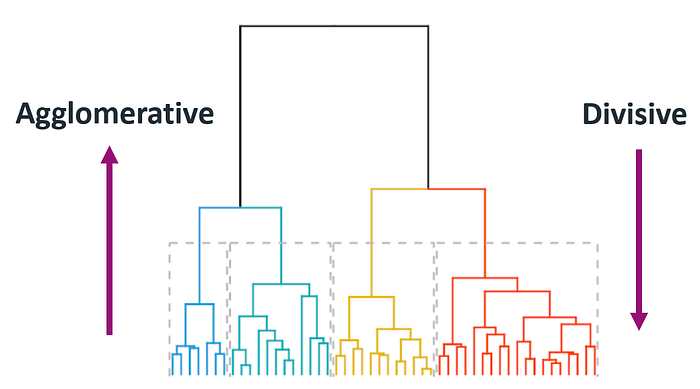
One of the most intriguing applications of unsupervised learning is clustering. Clustering is the process of grouping similar data points together, creating clusters or categories that share common characteristics. This technique is instrumental in various domains, including customer segmentation, image analysis, and anomaly detection. Two primary methods used for clustering are hierarchical clustering and k-means clustering.

**Hierarchical Clustering**

*Before continuing, I recommend you watch the video below:*

Imagine organizing a family tree, with each branch representing progressively finer categories. Hierarchical clustering operates in a similar manner, creating a hierarchical structure of clusters. It starts by treating each data point as its own cluster and then iteratively merges the most similar clusters until all data points are part of a single, large cluster. The result is a tree-like structure, called a **dendrogram**.

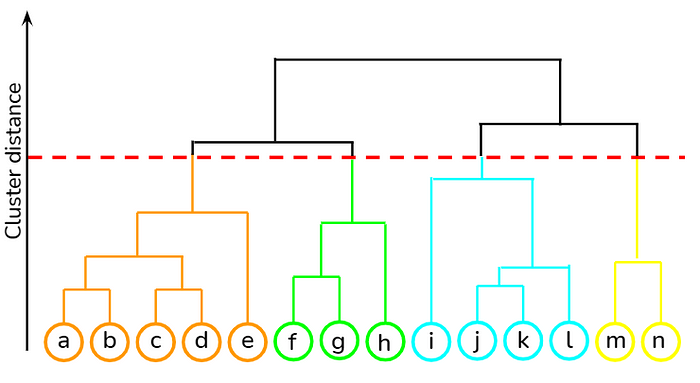
In hierarchical clustering, there are two main approaches: agglomerative and divisive clustering.



Dendrogram

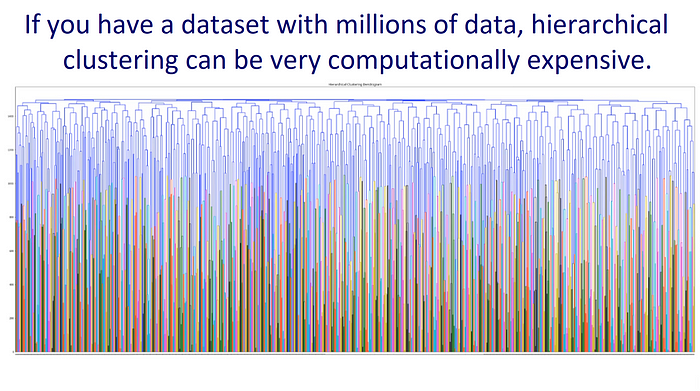
* Agglomerative clustering *starts with individual data points* and merges them into clusters, forming a hierarchy. It’s computationally **efficient**, provides a global view of data structure, offers flexibility in merging criteria, and is **widely used**.
* Divisive clustering *begins with all data points in one cluster* and recursively divides them. It’s computationally more **complex**, explores data structure differently, offers limited merging criteria, and is less commonly used due to its complexity.

**What is a dendrogram in hierarchical clustering?**



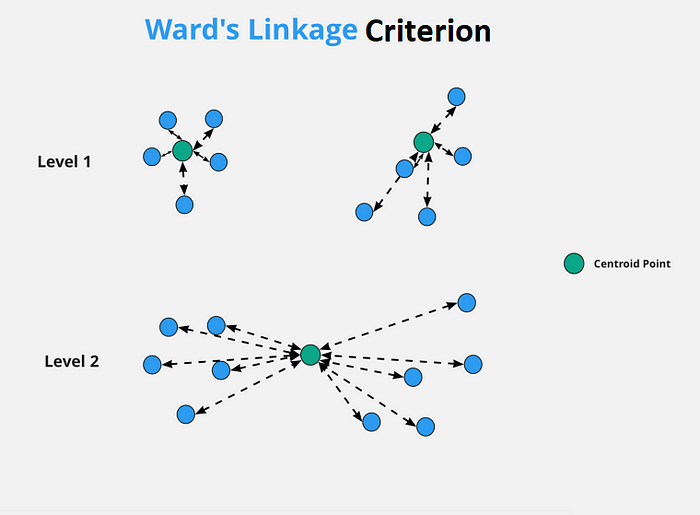
Dendrogram

One of the most intriguing outcomes of the hierarchical clustering is the dendrogram, a graphical representation that reveals the hierarchical relationships between data points or clusters. Dendrograms provide valuable insights, however, creating them can be a **computationally intensive task**, particularly when dealing with large datasets.



When creating a dendrogram in hierarchical clustering, the choice of **linkage criterion** significantly influences the structure and shape of the resulting dendrogram. These linkage criterions determine how clusters are merged at each step of the hierarchy. Now let’s examine the most used linkage criterions:

1. **Ward’s Linkage Criterion**

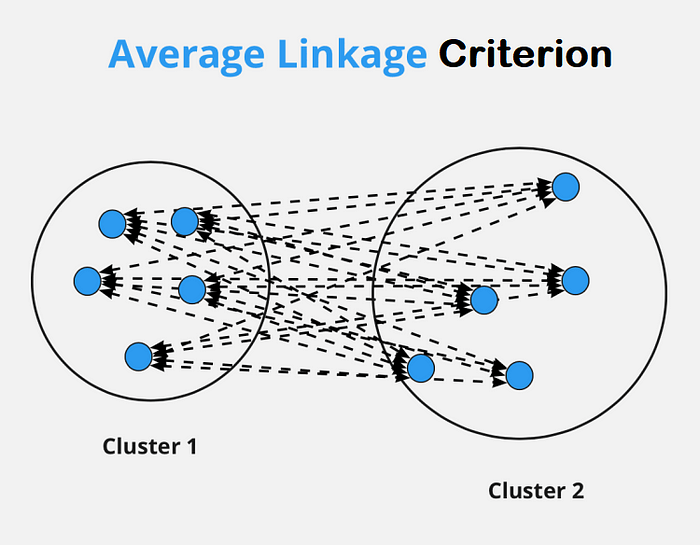


Ward’s Linkage Criterion

The Ward’s linkage criterion minimizes the variance of the distances between data points within clusters. In other words, it seeks to merge clusters that lead to the smallest increase in overall within-cluster variance.

It often results in compact, balanced, and spherical clusters, and tends to merge clusters that are similar not only in their nearest neighbors but also in the internal structure of the clusters, promoting the formation of cohesive and tight clusters.

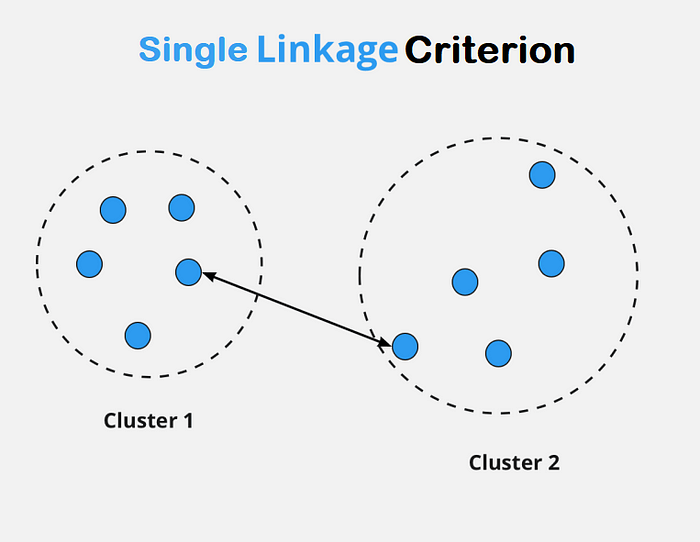
2.**Average Linkage Criterion**



Average Linkage Criterion

The Average linkage criterion calculates the average pairwise distance between all data points in two clusters being considered for merging. The Average linkage can lead to dendrograms with clusters of varying shapes and sizes. It tends to merge clusters based on their overall similarity, but it may not necessarily create clusters with uniform internal structures. This can result in more elongated or irregularly shaped clusters compared to the Ward’s linkage.

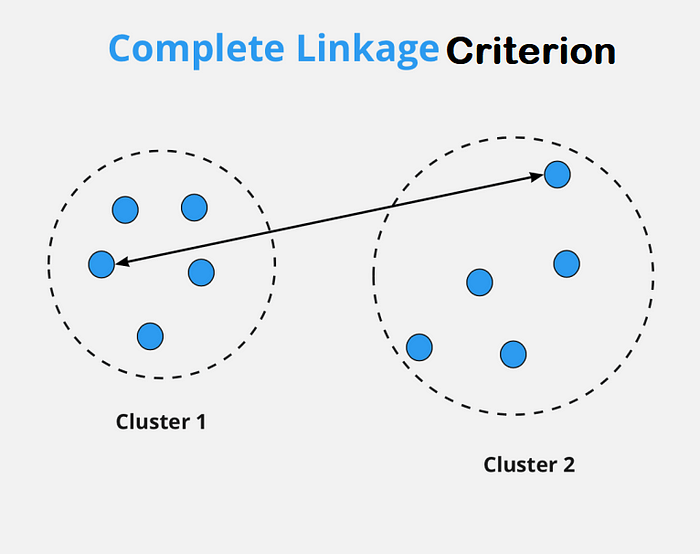
3.**Single Linkage Criterion**



Single Linkage Criterion

Also known as the nearest-neighbor approach. It measures the distance between the closest pair of data points in two clusters. It tends to produce elongated and chain-like clusters.

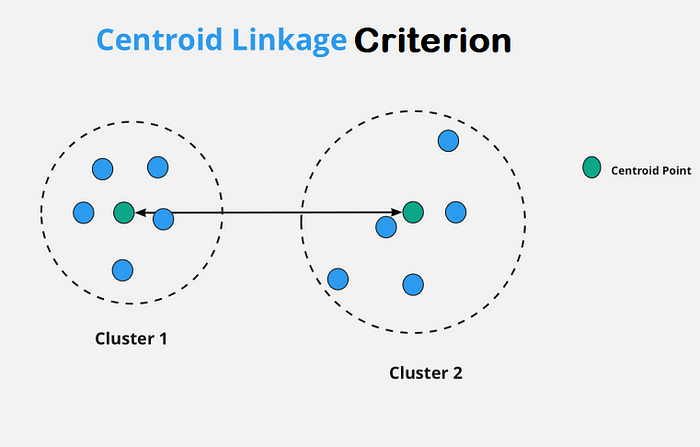
4.**Complete Linkage Criterion**



Complete Linkage Criterion

Also known as the farthest-neighbor approach. It measures the distance between the farthest pair of data points in two clusters. It can result in compact, spherical clusters.

5.**Centroid Linkage Criterion**



Centroid Linkage Criterion

It calculates the distance between the centroids (mean points) of two clusters. This criterion can lead to well-balanced clusters with similar sizes.

6.**Weighted Linkage Criterion:**Similar to average linkage, but it takes into account the sizes of the clusters being merged. It assigns different weights to clusters based on their sizes.

7.**Median Linkage Criterion:**It calculates the distance between the medians of two clusters. The median is the middle point when the data is sorted. It is less sensitive to outliers compared to the centroid linkage.

**Cophenet Correlation — How to decide on linkage criterion?**

*Before continuing, I recommend you watch the video below:*

Cophenet correlation is a measure of the similarity between the pairwise distances of data points in the original data space and the pairwise distances of the data points in the dendrogram created by hierarchical clustering. In simpler terms, it quantifies how faithfully the hierarchical clustering preserves the original distances between data points. A higher Cophenet correlation indicates that the hierarchical clustering effectively captures the pairwise similarities or dissimilarities present in the data.

The Cophenet correlation coefficient ranges between -1 and 1. However, when using libraries like SciPy to calculate the Cophenet correlation coefficient, the result is typically provided as a positive value between 0 and 1. In other words, the value you obtain using SciPy’s Cophenet function represents the strength of the correlation between the pairwise distances in the dendrogram. **Higher values** indicates **better** preservation of original distances and cluster structures.

**Silhouette Score — How to decide the number of clusters?**

Deciding on the right number of clusters is akin to finding the perfect balance between granularity and coherence in data grouping. Too few clusters may oversimplify the data, while too many clusters can result in confusion and redundancy.

In the intricate world of clustering, one of the most formidable challenges is determining the optimal number of clusters. Fortunately, the silhouette score emerges as a guiding beacon, helping us make informed decisions about clustering results.

The silhouette score offers a quantitative measure to gauge the quality of clustering results. It evaluates how well each data point fits within its assigned cluster, taking into account both cohesion (similarity to other points in the same cluster) and separation (dissimilarity from points in other clusters).

The value of the silhouette score is between [-1, 1]

* **0.71–1.0 :**A strong structure has been found
* **0.51–0.70 :**A reasonable structure has been found
* **0.26–0.50 :**The structure is weak and could be artificial.
* **< 0.25 :**No substantial structure has been found

So, if the silhouette score is negative valued, then your clusters suck.

**Coding — Part 1) Hierarchical Clustering**

Now let’s do some coding!

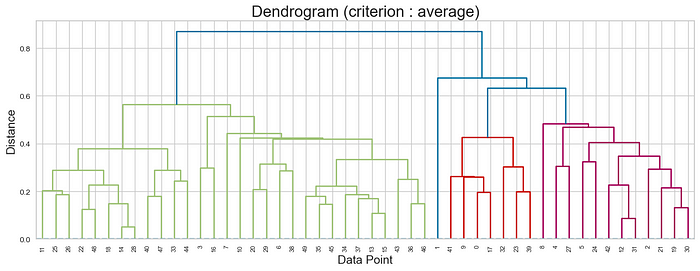
**[USArrests.csv](https://drive.google.com/file/d/1VdZHLfkUjOkvtW5KU1impa9FBorb95mJ/view?usp=sharing&source=post_page-----49f2894c1c95--------------------------------" \t "_blank)**

[Edit description](https://drive.google.com/file/d/1VdZHLfkUjOkvtW5KU1impa9FBorb95mJ/view?usp=sharing&source=post_page-----49f2894c1c95--------------------------------" \t "_blank)

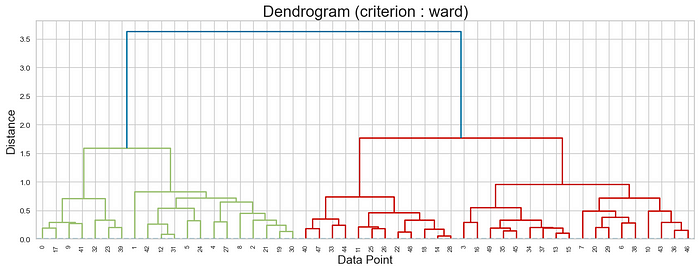
[drive.google.com](https://drive.google.com/file/d/1VdZHLfkUjOkvtW5KU1impa9FBorb95mJ/view?usp=sharing&source=post_page-----49f2894c1c95--------------------------------" \t "_blank)

USArrests dataset contains statistics, in arrest per 100,000 residents for assault, murder and rape in each of the 50 US States in 1973. The percentage of the population living in urban areas is also given.

################################  
# IMPORT  
################################  
  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.cluster import KMeans  
from scipy.spatial.distance import pdist  
from sklearn.metrics import silhouette\_score  
from sklearn.preprocessing import MinMaxScaler  
from yellowbrick.cluster import KElbowVisualizer  
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster, cophenet  
  
  
  
################################  
# FUNCTIONS  
################################  
  
  
def outlier\_thresholds(dataframe, col, q1=.05, q3=.95, decimal=3):  
 quartile1=dataframe[col].quantile(q1)  
 quartile3=dataframe[col].quantile(q3)  
 iqr=quartile3-quartile1  
 low\_limit= round(quartile1 - (iqr\*1.5) , decimal)  
 up\_limit= round(quartile3 + (iqr\*1.5), decimal)  
 return low\_limit , up\_limit  
  
  
  
  
def check\_outlier(dataframe, col\_name, q1=0.05, q3=0.95, lower\_limit = None, upper\_limit = None, print\_num=False):  
 """  
 Check for outliers in a specified column of a DataFrame.  
  
 Parameters:  
 dataframe (DataFrame): The DataFrame containing the data to check for outliers.  
 col\_name (str): The name of the column in the DataFrame to analyze for outliers.  
 q1 (float, optional): The lower quartile percentile (default is 0.05).  
 q3 (float, optional): The upper quartile percentile (default is 0.95).  
 lower\_limit (float, optional): The lower limit threshold for identifying outliers. If not provided,  
 it will be calculated based on the specified percentiles.  
 upper\_limit (float, optional): The upper limit threshold for identifying outliers. If not provided,  
 it will be calculated based on the specified percentiles.  
 print\_num (bool, optional): If True, the function returns the number of outliers and the lower and upper  
 limit thresholds as a tuple (outliers\_count, (lower\_limit, upper\_limit)).  
 Default is False.  
  
 Returns:  
 bool or tuple: If print\_num is False (default), returns True if outliers are detected in the specified  
 column; otherwise, returns False. If print\_num is True, returns a tuple containing  
 the number of outliers and the lower and upper limit thresholds.  
  
 Example:  
 # Check for outliers in the 'Age' column of 'my\_dataframe'  
 is\_outlier = check\_outlier(my\_dataframe, 'Age', q1=0.1, q3=0.9)  
  
 This function checks for outliers in the specified column of a DataFrame based on quartile percentiles.  
 You can customize the percentiles and specify lower and upper limit thresholds. If outliers are found,  
 you can choose to print the number of outliers along with the threshold values.  
 """  
 if (lower\_limit != None) & (upper\_limit != None):  
 low\_limit = lower\_limit  
 up\_limit = upper\_limit  
 else:  
 low\_limit, up\_limit = outlier\_thresholds(dataframe, col\_name, q1, q3)  
  
 if dataframe[(dataframe[col\_name] > up\_limit) | (dataframe[col\_name] < low\_limit)].any(axis=None):  
 if print\_num:  
 return True, dataframe[(dataframe[col\_name] > up\_limit) | (dataframe[col\_name] < low\_limit)].shape[0], (low\_limit,up\_limit)  
 return True  
 else:  
 return False  
  
  
  
  
  
def plot\_dgram(dataframe, criterion="ward", y:float=0, truncate\_mode=None, p:int=30, figsize:tuple=(15,5), standardize:bool=True, interactive:bool=False):  
 """  
 Generate and display a dendrogram plot for hierarchical clustering.  
  
 Parameters:  
 dataframe (DataFrame): The input data to be used for hierarchical clustering.  
 criterion (str, optional): The linkage criterion for hierarchical clustering. Default is "ward".  
 y (float, optional): The threshold distance at which to draw a horizontal line on the dendrogram.  
 Default is 0, indicating no threshold line.  
 truncate\_mode (str, optional): If not None, specifies the method to truncate the dendrogram.  
 Options are "lastp", "level", or None (default).  
 p (int, optional): The number of clusters to be shown when truncate\_mode="lastp".  
 Default is 30.  
 figsize (tuple, optional): The size of the plot figure. Default is (15, 5).  
 standardize (bool, optional): Whether to standardize the input data. Default is True.  
 interactive (bool, optional): If True, enables interactive mode for plotting (Jupyter Notebook).  
 Default is False.  
  
 Returns:  
 None  
  
 Example:  
 plot\_dgram(my\_dataframe, criterion="ward", y=0.5, truncate\_mode="lastp", p=20, figsize=(12, 6), interactive=True)  
   
 Notes:  
 - This function generates a dendrogram plot based on hierarchical clustering of the input data. You can customize  
 the linkage criterion, threshold distance, truncation mode, and other plot settings. If using Jupyter Notebook  
 and setting interactive=True, the plot will be interactive; otherwise, it will be displayed inline.  
 - For the best result, please standardize 'dataframe'.  
 """  
 #you can delete the 'if interactive' block below if you dont use jupiter notebook.  
 if interactive:  
 %matplotlib widget  
   
 #apply scaling to the input data.  
 if standardize:  
 dataframe = MinMaxScaler().fit\_transform(dataframe)  
  
 hc = linkage(dataframe, method = criterion)  
 plt.figure(figsize=figsize)  
 plt.title(f"Dendrogram (criterion : {criterion})", fontsize = 20)  
 plt.ylabel("Distance", fontsize=15)  
 plt.xlabel("Data Point", fontsize=15)  
 dendrogram(hc, truncate\_mode=truncate\_mode, p=p, show\_contracted=True)  
 plt.axhline(y=y, color='b', linestyle='--')  
 plt.show(block=True)  
 #you can delete the line below if you dont use jupiter notebook.  
 %matplotlib inline   
  
  
  
  
  
def cophenet\_scores(dataframe, criterions:list=None, standardize=True):  
 """  
 Calculate and display Cophenet correlation coefficients for hierarchical clustering linkage criterions.  
  
 Parameters:  
 - dataframe (pandas.DataFrame or numpy.ndarray): The input data in the form of a DataFrame or a numpy array.  
 - criterions (list, optional): A list of hierarchical clustering linkage criterions to evaluate. Default is None, which  
 includes ["single", "complete", "average", "weighted", "centroid", "median", "ward"].  
 - standardize (bool, optional): Whether to standardize the input data. Default is True.  
  
 Returns:  
 None  
  
 Notes:  
 - This function calculates the Cophenet correlation coefficients for different hierarchical clustering linkage criterions  
 applied to the input data. It then sorts and displays the results from highest to lowest correlation coefficient.  
 - For the best result, please standardize 'dataframe'.  
  
 """  
 if criterions is None:  
 criterions = ["single", "complete", "average", "weighted", "centroid", "median", "ward"]  
  
 # Create a dictionary to store cophenet correlation coefficients  
 cophenet\_scores = {}  
  
 # Apply scaling to the input data.  
 if standardize:  
 dataframe = MinMaxScaler().fit\_transform(dataframe)  
  
 # Convert the dataframe to a numpy array  
 try:  
 data = dataframe.values  
 except:  
 data = dataframe  
  
 for i, criterion in enumerate(criterions, start=1):  
 # Hierarchical clustering  
 linkage\_matrix = linkage(data, method=criterion)  
   
 # Calculate cophenet correlation coefficient  
 c, coph\_dists = cophenet(linkage\_matrix, pdist(data))  
   
 # Check if the score is NaN, and assign zero in that case  
 if np.isnan(c):  
 c = 0  
   
 # Add results to dictionary  
 cophenet\_scores[criterion] = c  
  
 # Sort the cophenet correlation coefficients from largest to smallest and reverse the order.  
 sorted\_scores = sorted(cophenet\_scores.items(), key=lambda x: x[1], reverse=True)  
   
 # Print results  
 for i, (criterion, score) in enumerate(sorted\_scores, start=1):  
 print(f"{i}. {criterion}: {score:.4f}")  
  
  
  
  
  
def hi\_clustering(dataframe, n\_components:int, criterion='ward', standardize=True):  
 """  
 Perform hierarchical clustering on a dataset.  
  
 Parameters:  
 - dataframe (pandas.DataFrame or numpy.ndarray): The input data in the form of a DataFrame or a numpy array.  
 - n\_components (int): The desired number of clusters.  
 - criterion (str, optional): The linkage criterion for hierarchical clustering. Default is 'ward'.  
 - standardize (bool, optional): Whether to standardize the input data. Default is True.  
  
 Returns:  
 - labels (numpy.ndarray): An array containing cluster labels for each data point.  
   
 Notes:  
 - For the best result, please standardize 'dataframe'.  
  
 """  
  
 if n\_components < 2 or n\_components != int(n\_components):  
 print("'n\_components' can be at least 2 and it should be an integer.")  
 return  
   
 if standardize:  
 dataframe = MinMaxScaler().fit\_transform(dataframe)  
 linkage\_matrix = linkage(dataframe, method=criterion)  
   
 # Algorithmic values  
 threshold = 0.1  
 low\_limit = 0  
  
 # Find the optimal threshold value that ensures n\_components = num\_labels.  
 while True:  
  
 # Check if 'n\_components' is invalid.  
 if (threshold - low\_limit) < 1e-10:  
 print(f"'n\_components' is cannot be {n\_components} when '{criterion}' linkage criterion is used for this dataset. \nPlease look at the dendrogram, and try a valid 'n\_component' value.")  
 return  
   
 # Clusters for the current threshold  
 labels = fcluster(linkage\_matrix, threshold, criterion='distance')  
   
 # Number of clusters for the current threshold  
 num\_labels = len(np.unique(labels))  
   
 if n\_components < num\_labels:  
 # Increase threshold  
 low\_limit = threshold  
 threshold = threshold\*2  
 elif n\_components > num\_labels:  
 # Decrease threshold  
 threshold = (low\_limit + threshold) / 2   
 else:  
 # If n\_components equals num\_labels, break  
 break  
   
 return labels  
  
  
  
  
  
  
  
################################  
# EDA  
################################  
  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
print(df.head())  
'''  
 Murder Assault UrbanPop Rape  
Alabama 13.2 236 58 21.2  
Alaska 10.0 263 48 44.5  
Arizona 8.1 294 80 31.0  
Arkansas 8.8 190 50 19.5  
California 9.0 276 91 40.6  
'''  
  
  
  
  
#No missing value  
print(df.isnull().sum())  
'''  
Murder 0  
Assault 0  
UrbanPop 0  
Rape 0  
dtype: int64  
'''  
  
  
  
  
#No categorical variable  
print(df.info())  
'''  
Index: 50 entries, Alabama to Wyoming  
Data columns (total 4 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Murder 50 non-null float64  
 1 Assault 50 non-null int64   
 2 UrbanPop 50 non-null int64   
 3 Rape 50 non-null float64  
dtypes: float64(2), int64(2)  
memory usage: 2.0+ KB  
'''  
  
  
  
  
#When we apply IQR analysis with 20% and 80%  
#it seems there is no outliers.  
for col in df.columns:  
 print(check\_outlier(df,col,q1=0.20, q3=0.80, print\_num=True))  
'''  
False  
False  
False  
False  
'''  
  
  
  
  
  
  
################################  
# Hierarchical Clustering  
################################  
  
#**NOTE:** You should ALWAYS apply SCALING before clustering  
#since clustering algorithms works by calculating distances.   
#So these distances should be in the same range!  
#Let's plot dendrogram with different criterions such as "average" and "ward"  
plot\_dgram(df, criterion="average", standardize=True)   
# IMAGE IS BELOW (dendrogram\_average.png)  
plot\_dgram(df, criterion="ward", standardize=True)  
# IMAGE IS BELOW (dendrogram\_ward.png)



dendrogram\_average.png



dendrogram\_ward.png

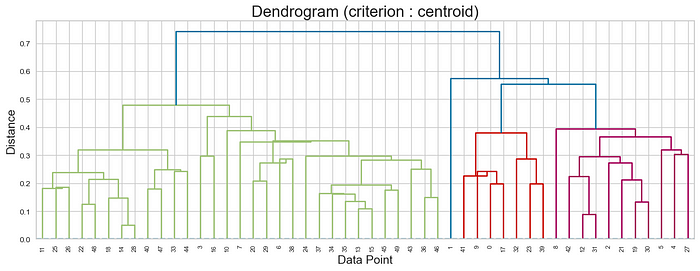
As you can see, the dendrogram varies quite depending on the criterion we use. So, which linkage criterion should we use? The answer lies in the cophnetet correlation coefficient. Let’s calculate it!

cophenet\_scores(df, standardize=True)  
'''  
1. centroid: 0.7336  
2. average: 0.7288  
3. complete: 0.7118  
4. ward: 0.7082  
5. weighted: 0.6911  
6. median: 0.6745  
7. single: 0.5300  
'''

* Centroid: 0.7336​ ​​ ​​​ ​​​ ​​​ ← ​ ​​ ​​​***the best linkage criterion***
* Average: 0.7288
* Ward: 0.7118
* Weighted: 0.6911
* Median: 0.6745
* Single: 0.5300

Thus, we discovered that the best linkage criterion for our dataset is ‘centroid’. We will set the criterion​ ​ ​​parameter to ‘centroid’. Let’s continue!

plot\_dgram(df, criterion="centroid", standardize=True)   
#IMAGE IS BELOW (dendrogram\_centroid.png)



dendrogram\_centroid.png

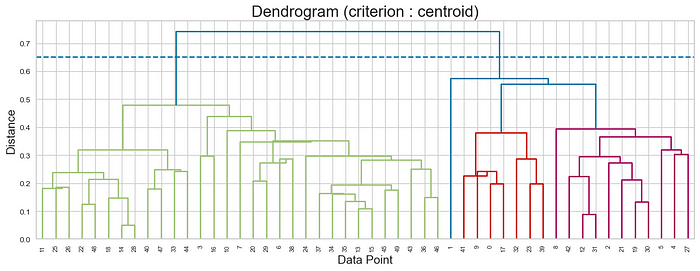
**Can we decide the number of clusters by looking at the dendrogram?**

Yes, we can. However, making a decision just by looking at the dendrogram is a heuristic, so we must validate the possible number of clusters by checking the Silhouette score, after dendrogram analysis.

Now let’s understand how we can analyze the dendrogram. For example, let’s look at the dendrogram plotted with the ‘centroid’ criterion above. First of all, the length of the dendrogram is approximately equal to 0.75 because the 1st horizontal line is at that distance. If the distance between two consecutive horizontal lines is more than approximately 10% of the length of the dendrogram, a dashed line is drawn between those two consecutive horizontal lines, and the number of clusters can be decided according to how many vertical lines are intersected the dashed line.

So, for the above plot, 10% makes 0.75/10 = 0.075  
Now, there is a distance of 0.75–0.57 = 0.18 between the 1st horizontal line and the 2nd horizontal line, so a dashed line can be drawn between the 1st horizontal line and the 2nd horizontal line since 0.18 > 0.075:

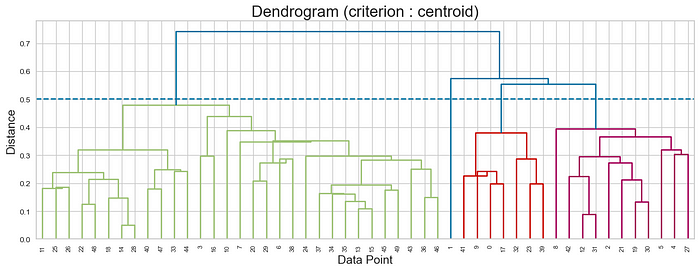
plot\_dgram(df, criterion="centroid", y = 0.65, standardize=True)   
#image is below



Since the dashed line above intersects 2 vertical lines, the number of clusters can be 2.

Alternatively, if we calculate the distance between the 3rd horizontal line and the 4th horizontal line, we find approximately 0.55–0.47 = 0.08  
Since 0.08 > 0.075, a dashed line can be drawn between the 3rd horizontal line and the 4th horizontal line:

plot\_dgram(df, criterion="centroid", y=0.5, standardize=True)   
#image is below



In this case, the dashed line intersected 4 vertical lines. Therefore, the number of clusters can be selected as 4.

Lastly, let’s calculate the distance between 4th horizontal line and 5th horizantal line. It is 0.57–0.43 = 0.04 . Since 0.04 < 0.075 , we cannot draw a dashed line between 4th and 5th line.

Our candidates are n=2 and n=4. Let’s try these number of clusters and then see their Silhouette scores to decide which one is better!

#Hierarchical Clustering(n=2, criterion='centroid')  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
clusters = hi\_clustering(df, n\_components=2, criterion='centroid', standardize=True)  
  
#**NOTE:** 'clusters' is not a dataframe, it is a numpy array.  
#It contains the cluster of each row in the dataset.  
#You can validate the cluster results by looking at   
#the dendrogram above.  
print(clusters)   
'''  
[2 2 2 1 2 2 1 1 2 2 1 1 2 1 1 1 1 2 1 2 1 2 1 2 1 1 1 2 1 1 2 2 2 1 1 1 1  
 1 1 2 1 2 2 1 1 1 1 1 1 1]  
'''  
  
  
#now let's calculate silhouette score.  
silhouette\_avg = silhouette\_score(df, clusters)  
print("Hierarchical Clustering(n=2, criterion='centroid') Silhouette Score:", silhouette\_avg)   
# 0.552

**Hierarchical Clustering (n=2, criterion=’centroid’) Silhouette Score:** 0.552  
Since 0.552 > 0.5, we can say that when n=2, there is a reasonable structure.

Let’s try a different number of cluster, n=4

#Hierarchical Clustering(n=4, criterion='centroid')  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
clusters = hi\_clustering(df, n\_components=4, criterion='centroid', standardize=True)  
  
print(clusters)   
'''  
[2 4 3 1 3 3 1 1 3 2 1 1 3 1 1 1 1 2 1 3 1 3 1 2 1 1 1 3 1 1 3 3 2 1 1 1 1  
 1 1 2 1 2 3 1 1 1 1 1 1 1]  
'''  
  
  
#now let's calculate silhouette score.  
silhouette\_avg = silhouette\_score(df, clusters)  
print("Hierarchical Clustering(n=4, criterion='centroid') Silhouette Score:", silhouette\_avg)   
# 0.272

**Hierarchical Clustering (n=4, criterion=’centroid’) Silhouette Score:** 0.272  
Since 0.272 > 0.25 , a weak structure has been found.

When we compare n=2 and n=4 for hierarchical clustering, we come to the conclusion that n=2 is better.

**NOTE :** *Sometimes, due to the job description, we may need to divide the data into a certain number of clusters. In such cases, even if the Silhouette Score is low, we should divide the dataset into a defined number of clusters.*

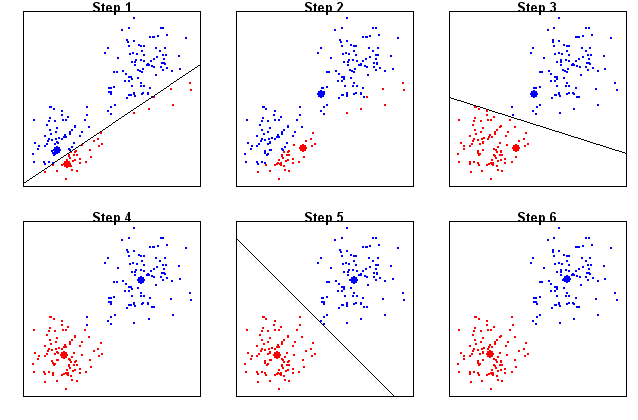
**BONUS:**Let’s try something extra. Please look again at the *dendrogram\_centroid.png* image above and see if it is possible to create 11 clusters. No matter where you draw the dashed line, you will see that it is impossible to create 11 clusters. In cases like this, if you enter an invalid number of cluster (n\_components) into the hi\_clustering() function, the function will give you a warning message. For example:

clusters = hi\_clustering(df, n\_components=11, criterion='centroid', standardize=True)  
'''  
'n\_component' is cannot be 11 when 'centroid' linkage criterion is used for this dataset.   
Please look at the dendrogram, and try a valid 'n\_component' value.  
'''  
  
  
print(clusters)  
# None

**K-Means**

*Before continuing, I recommend you watch the video below:*

While hierarchical clustering builds a hierarchy of clusters, k-means clustering aims to partition data into a predefined number of clusters (k) based on the similarity of data points.



K-Means Algorithm

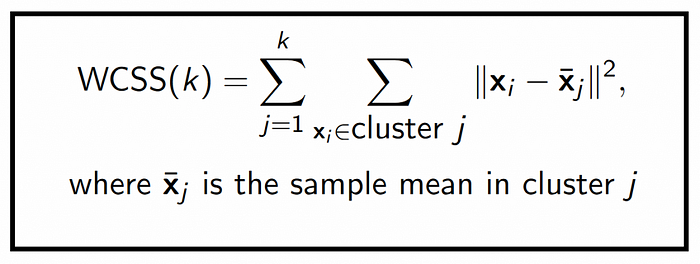
It starts by randomly selecting k cluster centers and assigns each data point to the nearest center. Then, it recalculates the cluster centers as the mean of all data points within each cluster and repeats the assignment and recalculation process until convergence.

The quality of the initial centroids (cluster centers) can significantly impact the algorithm’s convergence and the quality of the final clustering. This is where “*k-means++*” initialization comes into play.  
**K-Means++ :**K-Means++ initialization tends to produce more stable and better-performing results compared to purely random initialization, especially when the dataset is complex or the number of clusters (k) is relatively large.

**random\_state — How can I control the randomness in KMeans?**

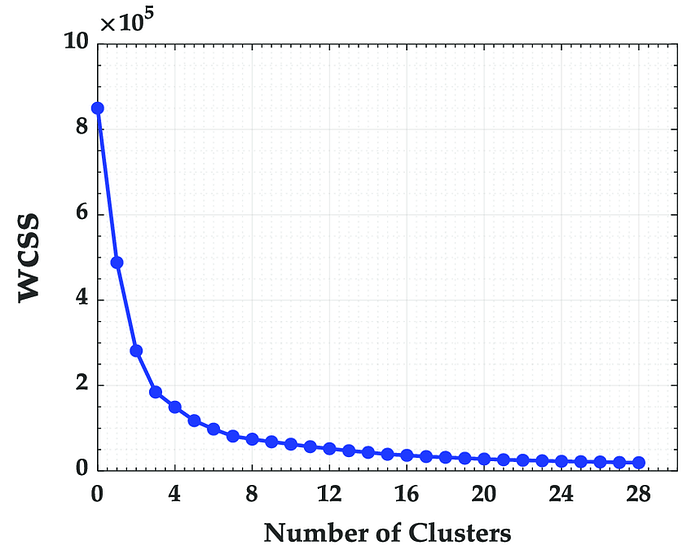
You can use the random\_state parameter to control the randomness in KMeans. Since K-Means involves random initialization of centroids (either with “k-means++” or purely random), setting the random\_state parameter allows you to reproduce the same results across different runs of the algorithm. By default, random\_state is set to None, thus the algorithm use a different random seed each time it runs, resulting in potentially different centroid initializations and, consequently, different clustering outcomes with each run. So, specifying a random\_state is useful when you want to control the randomness, ensure reproducibility of results, compare different clustering approaches, or debug issues in your code.

**Within-Cluster Sum of Squares (WCSS)**



WCSS

WCSS is a metric used to evaluate the performance of a K-Means clustering model. It quantifies the compactness of clusters by measuring the sum of squared distances between data points and their assigned cluster centroids. WCSS tends to decrease as the number of clusters (k) increases because with more clusters, data points are assigned to clusters with centroids that are closer to them. Essentially, smaller clusters can better approximate the data points, resulting in smaller squared distances and lower WCSS values.

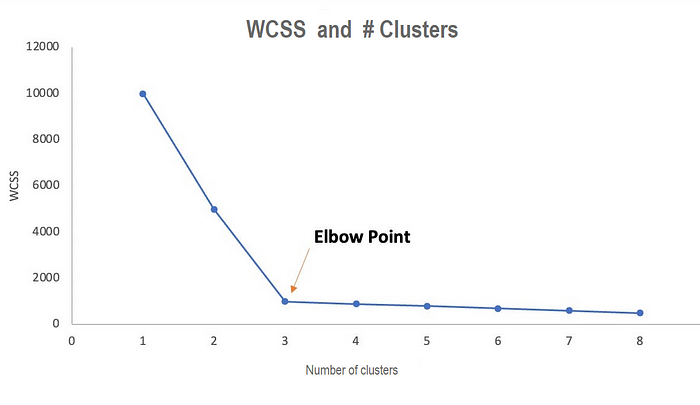


WCSS and number of clusters

As can be seen in the graph above, WCSS approaches zero as the number of clusters increases. In other words, if you create as many clusters as there are data, your WCSS value will be equal to **zero** because each point is a cluster. However, this defeats the purpose of clustering, as it does not provide any meaningful grouping or simplification of the data. Also, it often leads to overfitting. The goal of clustering is to find patterns and structures within data while reducing its complexity. Therefore, our aim here should be to decide the most appropriate number of clusters in the range **[2, number of data]**.

**The Elbow Method**

When using KMeans, we can decide the most appropriate number of clusters with the Elbow method. The Elbow method is a **heuristic** technique used to determine the optimal number of clusters for K-Means. It involves plotting the WCSS values for different values of k (typically ranging from 1 to a maximum number of clusters you want to explore) and looking for the “elbow” point in the graph.



Elbow Method

**NOTE:** *While the Elbow method is a widely used heuristic for determining the number of clusters in K-means clustering, it may not always provide the best results, especially for complex datasets. If you are not in a hurry, most of the time I recommend you to****use hierarchical clustering****instead of Kmeans clustering since utilizing the cophenet correlation coefficient, dendrogram analysis, and silhouette score can provide a more comprehensive and informed approach.*

**Coding — Part 2) KMeans Clustering**

In this part, we will apply KMeans for the n=2 and n=4 (both of them were decided by looking at the dendrogram) and then look at the Silhouette scores. Lastly, we will use elbow method and apply KMeans again with the number of clusters we obtained with the elbow method and look at the Silhouette score.

**NOTE:** *Normally, there is no relationship between Kmeans and dendrogram. In Kmeans, usually we cannot obtain the clusters we obtained with dendrogram analysis. However, in this article, we previously used dendrogram analysis during hierarchical clustering**and decided that number of clusters as n = 2 and n = 4 are good candidates. So, we intuitively think that n = 2 and n = 4 might be also good in Kmeans clustering.*

################################  
# K-Means Clustering  
################################  
  
  
#KMeans(n=2)  
###############  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
df = MinMaxScaler().fit\_transform(df)  
kmeans = KMeans(n\_clusters=2, random\_state=17).fit(df)  
#Default parameters  
kmeans.get\_params()  
'''  
{'algorithm': 'lloyd',  
 'copy\_x': True,  
 'init': 'k-means++',  
 'max\_iter': 300,  
 'n\_clusters': 2,  
 'n\_init': 'warn',  
 'random\_state': 5,  
 'tol': 0.0001,  
 'verbose': 0}  
'''  
  
#see number of clusters  
print(kmeans.n\_clusters) # 2  
  
#see the center of each clusters. remember, our dataset has 4 features.  
#that is, each cluster has a center in each feature.  
print(kmeans.cluster\_centers\_)  
'''  
[[0.24518072 0.23778539 0.53615819 0.22334195]  
 [0.68463855 0.72003425 0.61694915 0.56498708]]  
'''  
  
#this is our predicted clusters for each row.  
print(kmeans.labels\_)  
'''  
[1 1 1 0 1 1 0 0 1 1 0 0 1 0 0 0 0 1 0 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 0 0 0  
 0 0 1 0 1 1 0 0 0 0 0 0 0]  
'''  
  
  
  
  
#Silhouette Score of Kmeans (n=2)  
clusters\_kmeans = kmeans.labels\_  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
silhouette\_avg = silhouette\_score(df, clusters\_kmeans)  
print("Silhouette Score of KMeans (n=2):", silhouette\_avg) # 0.540

**Silhouette Score of KMeans (n=2):** 0.540  
Since 0.54 > 0.5, we can say that when n=2, there is a reasonable structure.

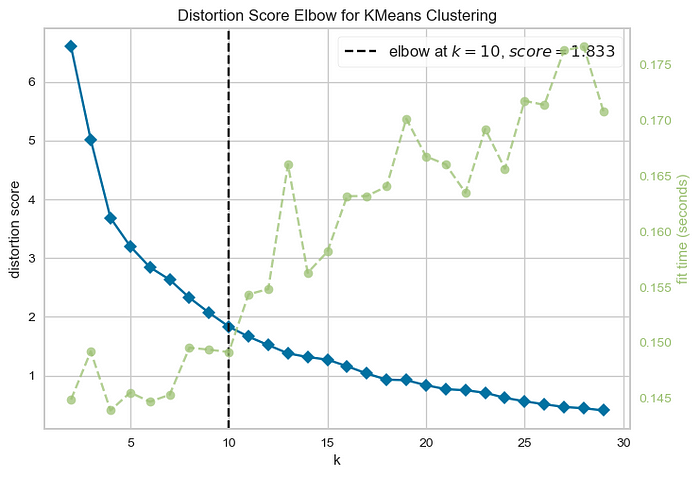
Now, let’s try a different number of cluster, n=4

#KMeans(n=4)  
###############  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
df = MinMaxScaler().fit\_transform(df)  
kmeans = KMeans(n\_clusters=4, random\_state=17).fit(df)  
  
#Silhouette Score of Kmeans (n=4)  
clusters\_kmeans = kmeans.labels\_  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
silhouette\_avg = silhouette\_score(df, clusters\_kmeans)  
print("Silhouette Score of KMeans (n=4):", silhouette\_avg) # 0.233

**Silhouette Score of KMeans (n=4):**0.233  
Since 0.23 < 0.25, no substantial structure has been found.

Lastly, let’s try to find the number of cluster by using Elbow method, and validate it by using Sihouette Score.

#Elbow Method  
####################  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
df = MinMaxScaler().fit\_transform(df)  
kmeans = KMeans(random\_state=17)  
#k=(2,30) because we want to try number of clusters in this range.  
elbow = KElbowVisualizer(kmeans, k=(2, 30))  
elbow.fit(df)  
elbow.show() #IMAGE IS BELOW (elbow.png)



elbow.png

According to the Elbow method, the number of clusters should be 10. But remember, the Elbow method is a **heuristic**, meaning it may not give us the globally optimal result. That’s why we should be skeptical and check the Silhouette Score:

#KMeans (n=10)  
###########################  
  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
df = MinMaxScaler().fit\_transform(df)  
kmeans = KMeans(n\_clusters=10, random\_state=17).fit(df)  
  
#Silhouette Score of Kmeans (n=10)  
clusters\_kmeans = kmeans.labels\_  
df = pd.read\_csv("datasets/USArrests.csv", index\_col=0)  
silhouette\_avg = silhouette\_score(df, clusters\_kmeans)  
print("Silhouette Score of KMeans (n=10):", silhouette\_avg) # -0.057

**Silhouette Score of KMeans (n=10):**— 0.057Since it is a negative value, it seems that no pattern was caught with 10 clusters.

Therefore, we should not rely too much on the number of clusters we obtain from the Elbow method and should definitely look at the Silhouette Score. Typically, it would be safer to make a comment about the number of clusters by looking at the***dendrogram rather than the Elbow method.***

*Thanks for reading…*

[Clustering](https://medium.com/tag/clustering?source=post_page-----49f2894c1c95---------------clustering-----------------)

[K Means](https://medium.com/tag/k-means?source=post_page-----49f2894c1c95---------------k_means-----------------)

[Hierarchical Clustering](https://medium.com/tag/hierarchical-clustering?source=post_page-----49f2894c1c95---------------hierarchical_clustering-----------------)

[Unsupervised Learning](https://medium.com/tag/unsupervised-learning?source=post_page-----49f2894c1c95---------------unsupervised_learning-----------------)

[Machine Learning](https://medium.com/tag/machine-learning?source=post_page-----49f2894c1c95---------------machine_learning-----------------)