Unsupervised Machine Learning with K-Means Clustering in Python

Let’s focus on unsupervised learning that deals with unlabeled data, making it a powerful tool for discovering hidden patterns and structures. We'll delve into a powerful technique called K-means clustering using Python.

K-means clustering is a popular unsupervised learning algorithm used in machine learning afor partitioning a dataset into a set of distinct, non-overlapping groups or clusters. The goal of K-means clustering is to group similar data points while keeping dissimilar points in different groups. It achieves this by iteratively dividing the unlabeled dataset into k different clusters in such a way that each data point belongs to only one group that has similar properties.

In this case study we will be using the K-means clustering algorithm on the `mall_customers` data set. This data set is the data of different customers in the mall. It contains five columns namely: 'CustomerID', 'Genre', 'Age', 'Annual Income (k$)', and 'Spending Score'

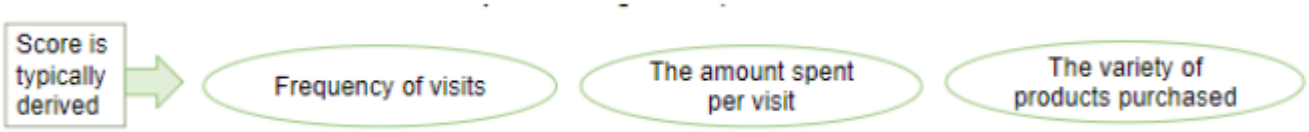
CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-1
1	Male	19	15	39
2	Male	21	15	81
3	Female	20	16	6
4	Female	23	16	77
5	Female	31	17	40
6	Female	22	17	76
7	Female	35	18	6
8	Female	23	18	94
9	Male	64	19	3
10	Female	30	19	72
11	Male	67	19	14
12	Female	35	19	99
13	Female	58	20	15
14	Female	24	20	77
15	Male	37	20	13

We will be using the K-means clustering algorithm on the `mall_customers` data set.

	A	B	C	D	E	F	G
1	Customer	Genre	Age	Annual Income (k$)	Spending Score (1-100)		
2	1	Male	19	15	39		
3	2	Male	21	15	81		
4	3	Female	20	16	6		
5	4	Female	23	16	77		

The Spending Score, ranges from 1 to 100 and it serves as a metric to evaluate a customer's purchasing behavior within the mall

The Spending Score, ranges from 1 to 100 and it serves as a metric to evaluate a customer's purchasing behavior within the mall. A lower spending score indicates that the customer tends to spend less on the goods and services offered by the mall, whereas a higher spending score suggests that the customer is more likely to make significant purchases. This score is typically derived from various factors such as the frequency of visits, the amount spent per visit, and the variety of products purchased.



And based on the Annual Income and Spending Score, we need to group the customers in the dataset into different clusters using K-means. This clustering approach will allow us to identify distinct segments of customers with similar purchasing behaviors, enabling targeted marketing strategies and personalized offerings to enhance customer satisfaction and drive revenue growth.

We will start by opening a new notebook in Google Colab and naming it `K Means Clustering`.

[Open the google collab notebook simultaneously]

You can name it anything that you want. Next, we will move to the files tab on the side panel of Google Colab and upload the provided dataset `customers_data.csv`

[Upload the dataset simultaneously]

Once we have our notebook set up and our dataset uploaded, we'll begin by importing the necessary libraries, such as numpy, matplotlib, pandas, and K-means model from sklearn.cluster module.

Google Collab

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.cluster import KMeans
```

Now, let's import the dataset using pandas and store it in a DataFrame:

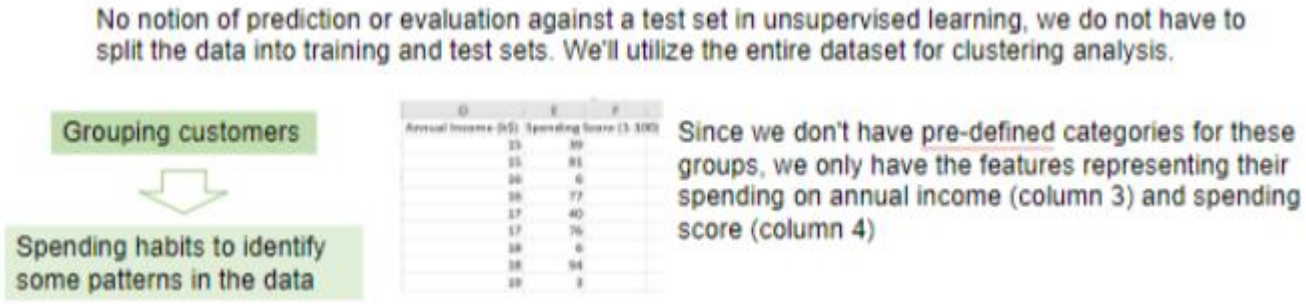
Google Collab

```
dataset = pd.read_csv('Mall_Customers.csv')
```

In unsupervised machine learning like clustering, we don't have a dependent variable to predict, so we only need the features for analysis. The goal is to uncover hidden patterns or structures within the data itself. Therefore, we only have the input features, which we can

represent as a single matrix typically denoted by X. Additionally, since there's no notion of prediction or evaluation against a test set in unsupervised learning, we do not have to split the data into training and test sets. We'll utilize the entire dataset for clustering analysis.

Here, we're interested in grouping customers based on their spending habits to identify some patterns in the data. Since we don't have pre-defined categories for these groups, we only have the features representing their spending on annual income (column 3) and spending score (column 4).



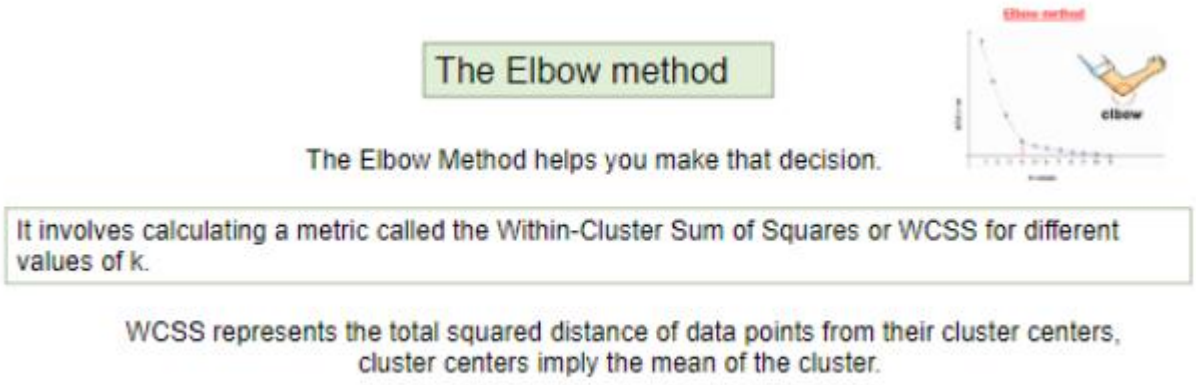
This shows the dataset simultaneously so that the position of annual income and spending score is clear, which is column 3 and 4. So, let's select these two columns from our dataset and store them in a variable named X for further analysis:

Google Collab

X = dataset.iloc[:, [3, 4]].values

This line of code uses pandas' iloc method to select all rows (:) and only the columns at index 3 and 4 ([3, 4]) from our dataset.

Great! We've got our features (X) ready for k-means clustering. Now, an important step before applying the algorithm is to determine the optimal number of clusters (k) because when you use K-means, you have to decide how many clusters you want to divide your data into. This is where the elbow method comes in. The Elbow Method helps you make that decision.



It involves calculating a metric called the Within-Cluster Sum of Squares or WCSS for different values of k. WCSS represents the total squared distance of data points from their cluster centers, cluster centers imply the mean of the cluster.

Here's how the elbow method works:

We'll use a for loop to iterate through a range of potential k values, typically starting from 1 and going up to a chosen maximum (often 10 or more). The maximum value of k should be chosen based on the context of the problem and the characteristics of the data. Sometimes it's appropriate to explore higher values of k, depending on the complexity of the dataset and the desired granularity of clustering.

Inside the loop, for each value of k, we'll execute the k-means algorithm. This will create k clusters in our data.

After running k-means for a specific k, we'll calculate the WCSS for that particular clustering. WCSS is the sum of squared distances between each data point and its assigned cluster centroid. A lower WCSS indicates that the points within a cluster are, on average, closer to their centroid, which suggests tighter clusters.

We typically don't choose a very high value for the maximum k (like close to the number of data points) because with enough clusters, each data point could become its own cluster in other words a cluster would

have only one data point, which would result in a WCSS of zero but wouldn't be a meaningful clustering.

Next, we'll store the calculated WCSS value for each k in a list. After iterating through the loop, we'll have a list of WCSS values corresponding to different numbers of clusters (k). This information will be used to create a visual representation to identify the optimal k.

WCSS metric, will be plotted on the Y-axis and the X-axis will represent the different numbers of clusters ranging from 1 to 10. This iterative process allows us to observe how the WCSS changes as we vary the number of clusters.

Now, to effectively analyze the WCSS for different k values, let's create an empty list to store these WCSS calculations. This list will be populated through a loop that iterates through various k values. We'll name this list as wcss, which stands for Within-Cluster Sum of Squares.

Google Collab

wcss = []

Now that we have the wcss list to store WCSS values, let's implement the loop that iterates through different k options. We'll use a for loop to achieve this. We'll start the for loop with the range from 1 to 11, and as we know the range in python includes lower bound but excludes upper bound. Therefore, we'll write:

Google Collab

for i in range(1, 11):

And now inside this for loop, we'll create a new KMeans object each time through the loop. Since we are iterating from 1 to 11, we'll create 10 different KMeans objects in our case.

Let's create our kmeans object, which we create by calling the KMeans class. We then add parentheses.

Therefore,

Google Collab

kmeans = KMeans()

And now we'll import necessary arguments.

So the first parameter is the number of clusters that the algorithm should identify and we'll call it as n_clusters which starts at 1 and iterates up to 10.

This means that for each iteration of the loop, the algorithm will create a different number of clusters, ranging from 1 to 10. This allows us to compare the algorithm's performance for different cluster numbers and find the optimal number of clusters using the elbow method.

So, each time we create a new KMeans object inside the loop, this n_clusters parameter will take on a different value of i.

Therefore,

Google Collab

kmeans = KMeans(n_clusters = i)

But KMeans can be sensitive to where it initially places its cluster centers. To avoid getting stuck in random initialization trap, you set the init parameter to "k-means++". This is a smart initialization method that strategically chooses centroids to be well-separated in your data, helping the algorithm converge faster and potentially find better cluster assignments.

In other words, it's like giving the algorithm a head start by placing its starting points strategically, which makes it work more efficiently and possibly give better results.

Therefore,

Google Collab

kmeans = KMeans(n_clusters = i, init = 'k-means++')

Additionally, to ensure reproducibility and consistency in our results, we set the random_state parameter to a specific value, such as 42. This ensures that the random initialization of centroids is done in a deterministic manner across different runs of the algorithm.

When you set a value for the random_state parameter, like random_state=42 in the code, you're essentially locking in the starting point for the algorithm's randomness. So, no matter how many times you run the algorithm with the same setup and data, it will always begin from the same starting point. This makes the clustering process predictable and gives you the same results every time you run it.

Therefore,

Google Collab

kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)

After setting up our KMeans object with the appropriate parameters, we're now ready to apply it to our data. The next step is to run the algorithm, essentially training it on our dataset. We're ready to train the KMeans algorithm with 'i' number of clusters.

The function used to train an algorithm is the fit function. So the next step here is to take our KMeans object and call the fit method on it. This fit method will take as an input the matrix of features X.

This process involves the algorithm iteratively updating the cluster centroids to minimize the within-cluster sum of squares (WCSS) and assign each data point to the nearest centroid.

Let's proceed by fitting our KMeans object to the data.

Therefore,

Google Collab

kmeans.fit(X)

Now that we have already trained and run our KMeans algorithm, we need to compute the WCSS (Within-Cluster Sum of Squares) value. We

do this by first taking our wcss list, which is so far initialized as an empty list. Then, we use the append function to add a new value inside the list. The first value we add is the WCSS value computed for 1 cluster, which is actually the sum of the squared distances between all the observation points and the centroid of the single cluster.

Alright, let's append. Since it's a function, we're adding some parenthesis. The way to get the WCSS value is to call an attribute of the KMeans object called inertia_, which will give us exactly that WCSS value. In order to get that attribute, we just need to call our object first, because when we want to get an attribute, we always have to call it from the object, and then we add a dot and then we call the attribute which is here inertia_.

Therefore,

Google Collab

wcss.append(kmeans.inertia_)

This completes one iteration of the loop. As the loop progresses, we'll train the KMeans model with different i values (different numbers of clusters) and keep appending the WCSS for each iteration. This will allow us to plot the elbow method graph in the upcoming steps!

Now, we're ready to plot a simple curve that will visualize the WCSS values for different numbers of clusters from 1 to 10. This curve will depict the WCSS on the y-axis, and the number of clusters used will be on the x-axis.

Therefore,

Google Collab

plt.plot(range(1, 11), wcss)

plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()

plt.plot function from the matplotlib library which is abbreviated as plt is used for creating plots.

range(1, 11) creates a sequence of numbers from 1 to 10 which represents the different numbers of clusters used in the loop.

wcss refers to the list we've been populating with WCSS values throughout the loop iterations.

Next the plt.title adds a title to our graph, labeling it "The Elbow Method".

plt.xlabel('Number of clusters') labels the x-axis of the graph, indicating it represents the number of clusters used. And

plt.ylabel('WCSS') labels the y-axis of the graph, indicating it represents the WCSS values.

Lastly plt.show() displays the graph we just created.

Now let’s run this code cell and plot the elbow method graph.

[Show the graph]

So, after seeing the elbow method graph, how would we choose the optimal number of clusters?

Here we need to identify the "elbow point," which is the point where the rate of decrease in WCSS slows down significantly. This point indicates the optimal number of clusters for our data.

So, after seeing the elbow method graph, how do we choose the optimal number of clusters?

The elbow method helps us visually identify a good stopping point for the number of clusters. Ideally, the curve should have a distinct "elbow" shape. The optimal number of clusters is often chosen at the point where the curve starts to bend sharply. This indicates that adding more

clusters after this point results in diminishing returns, meaning the WCSS reduction is less significant for each additional cluster.

In our example, the curve seems to be flattening out around 5 clusters. Based on the elbow, this suggests that 5 will be the optimal number of clusters for our data.

Now we re-train the KMeans model with the optimal number of clusters.

This involves initializing a new KMeans object with the optimal number of clusters which is five and fitting it to your dataset.

Therefore,

Google Collab

kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)

The rest remains the same as done previously, and just the n_clusters parameter is initialized to 5 instead of using the loop variable i. This ensures that the model is trained with the optimal number of clusters identified through the elbow method. We keep the same way of starting the sorting process (init='k-means++') and the random order we started with (random_state=42) for consistency.

Next, we use the trained KMeans model to predict which cluster each data point belongs to. This is done by calling the fit_predict method on the KMeans object with our dataset X.

Therefore,

Google Collab

y_kmeans = kmeans.fit_predict(X)

This line of code performs two crucial actions in a single step:

First .fit() method trains the KMeans model based on our actual data (X), making sure each cluster has similar data points with similar characteristics. And

After training, .predict(X) assigns a label to each data point. So, for every data point in our dataset, the model assigns a label (or cluster number) indicating which group it belongs to.

The variable y_kmeans stores these cluster labels for each data point.

So, in summary, the trained KMeans model (kmeans) predicts which cluster each data point in dataset X belongs to, and stores the predicted cluster labels in the array y_kmeans.

Now, let's visualize the clusters identified by the KMeans algorithm. Since we have the cluster labels (y_kmeans) for each data point, we can create a scatter plot. This plot will represent each data point as a dot, and we'll use color to differentiate between the clusters based on their labels. On the X-axis, we'll have the annual income, which is our first feature, and on the Y-axis, the spending score.

To accomplish this, we'll create several scatter plots, one for each cluster. This means we'll plot the data points belonging to cluster 1, then cluster 2, and so on, up to cluster 5.

We'll achieve this by utilizing the scatter function from the matplotlib.pyplot module.

Therefore,

Google Collab

plt.scatter()

Since we are using the scatter function to plot each of the five clusters, we'll call this scatter function five times with different inputs. So, now we're starting with cluster number 1.

Google Collab

plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')

The First Argument X[y_kmeans == 0, 0] selects the X-axis coordinates for the data points we want to plot. Remember X here is the matrix of two columns containing the annual income in the first column and spending score in the second column.

[y_kmeans == 0, 0] filters the data points based on their cluster labels (y_kmeans). We're using a condition == 0 to specifically select only those points where the label is 0 (which corresponds to cluster 1).

Next this, 0 tells the function to pick only the first feature which is the annual income from the selected data points. So, it picks the X-coordinate (first feature) for each point in cluster 1.

The Second Argument X[y_kmeans == 0, 1] selects the Y-axis coordinates for the data points we want to plot, similar to the first argument.

In X[y_kmeans == 0, 1] here, 1 tells the function to pick the second feature which is the spending score from the selected data points in cluster 1. So, it picks the Y-coordinate for each point in cluster 1.

Next s = 100 sets the size of the dots representing each data point. Here, s = 100 makes the dots a bit larger for better visibility. You can adjust this value to control the dot size.

Then c = 'red' argument sets the color of the dots for this scatter plot. We're using c = 'red' to represent cluster 1 with red dots. You can choose different colors for other clusters.

Next label = 'Cluster 1' assigns a label to the cluster, which will be used later in the legend (a small box showing the meaning of each color). Here, label = 'Cluster 1' provides a clear identification for the red dots representing cluster 1.

Alright, now let’s do the same for the other clusters. We'll just copy this line of code and paste it right below to plot the second cluster.

Google Collab

Copy paste above line of code and then make the changes as mentioned.

And now, to plot the second cluster, we just have to change:

y_kmeans == 0 to y_kmeans == 1 to select the data points belonging to cluster 2.

Change the color c to differentiate it from the first cluster, let's say 'blue'.

And Update the label to 'Cluster 2' instead of 'Cluster 1' to reflect the change in cluster number.

Google Collab

plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')

now let’s do the same for cluster 3, 4 and 5 as follows.

Google Collab

plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')

plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')

plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')

Now that we've created scatter plots for each individual cluster, let's take a look at the centroids. Remember, centroids are the representative points for each cluster, like the "hearts" or "centers of gravity" that define the core of each group.

To visualize these centroids, we'll use the scatter function again, but this time, we'll provide the coordinates of the centroids themselves. These coordinates are stored in the kmeans.cluster_centers_ attribute of the KMeans model we trained. We'll do this by adding the following line of code:

Google Collab

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 300, c = 'yellow', label = 'Centroids')

here kmeans refers to the KMeans model we've already trained. It likely holds information about the clustering process.

cluster_centers_ is an attribute (a property) of the kmeans model. In the context of KMeans clustering, this attribute specifically stores the coordinates of the centroids.

Next [:, 0] selects all rows (":") and the first column (0) from the "cluster_centers_" attribute. This gives us the x-coordinates of the centroids.

Similar to the previous [:, 1], select all rows and the second column (1), which gives us the y-coordinates of the centroids.

s = 300 sets the size of the marker for each centroid to 300 points. This will make them visually distinct from other data points.

c = 'yellow' sets the color of the centroid markers to yellow. And lastly

label = 'Centroids' assigns a label "Centroids" to this data series, which will be used in the legend if one is created.

In summary, this code snippet extracts the centroids' coordinates from the trained KMeans model and uses them to create a scatter plot with yellow markers sized at 300 points, labeled as "Centroids". This helps visualize where the centroids lie in the data space.

.

Then, we add finishing touches to the plot

[GC]

plt.title('Clusters of customers')

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.legend()

plt.show()

where

plt.title('Clusters of customers'): Sets the title of the plot as 'Clusters of customers'.

plt.xlabel('Annual Income (k$)'): Labels the x-axis as 'Annual Income (k$)' to indicate the data represented on the x-axis.

plt.ylabel('Spending Score (1-100)'): Labels the y-axis as 'Spending Score (1-100)' to indicate the data represented on the y-axis.

plt.legend(): Adds a legend to the plot to distinguish between clusters and centroids.

plt.show(): Displays the plot.

Now let’s execute this code cell in order to visualize the clusters.

So, with these additions, we can clearly observe the five clusters of customers. Now, let's delve into their analysis. Our goal here is to cluster these customers effectively to gain insights into their behavior. By understanding these clusters, we aim to optimize our business strategies by tailoring offers to each group. Alright, let's proceed.

Let's start by examining each cluster. Cluster 4 comprises customers with low annual income and minimal spending at the mall. Moving on to cluster 5, we find customers with high annual income but low spending. Next, cluster 1 consists of customers with low annual income but high spending scores. Cluster 3, on the other hand, includes high-income customers who spend generously at the mall. Lastly,

cluster 2 represents an average group, with customers earning a moderate income and spending moderately at the mall.

Now, armed with this understanding, we can implement targeted marketing strategies to cater to the unique preferences of each cluster. For instance, we can offer discounts and promotions to cluster 1 to encourage their high spending behavior, while focusing on introducing premium products or services to cluster 3 to capitalize on their high-income status. Similarly, we can devise specific incentives for clusters 4 and 5 to increase their spending levels. By tailoring our marketing efforts accordingly, we can maximize engagement and drive profitability across all customer segments.

And this is just a starting point, and further analysis might reveal additional insights within each cluster. By understanding these customer segments, we can craft targeted strategies to strengthen relationships, optimize marketing efforts, and ultimately drive business growth while upholding our moral responsibility to our customers and society.

And that’s all in this part. Thankyou