The Most Comprehensive Guide to K-Means Clustering You'll Ever Need

ALGORITHM CLUSTERING INTERMEDIATE MACHINE LEARNING PYTHON STRUCTURED DATA UNSUPERVISED

Overview

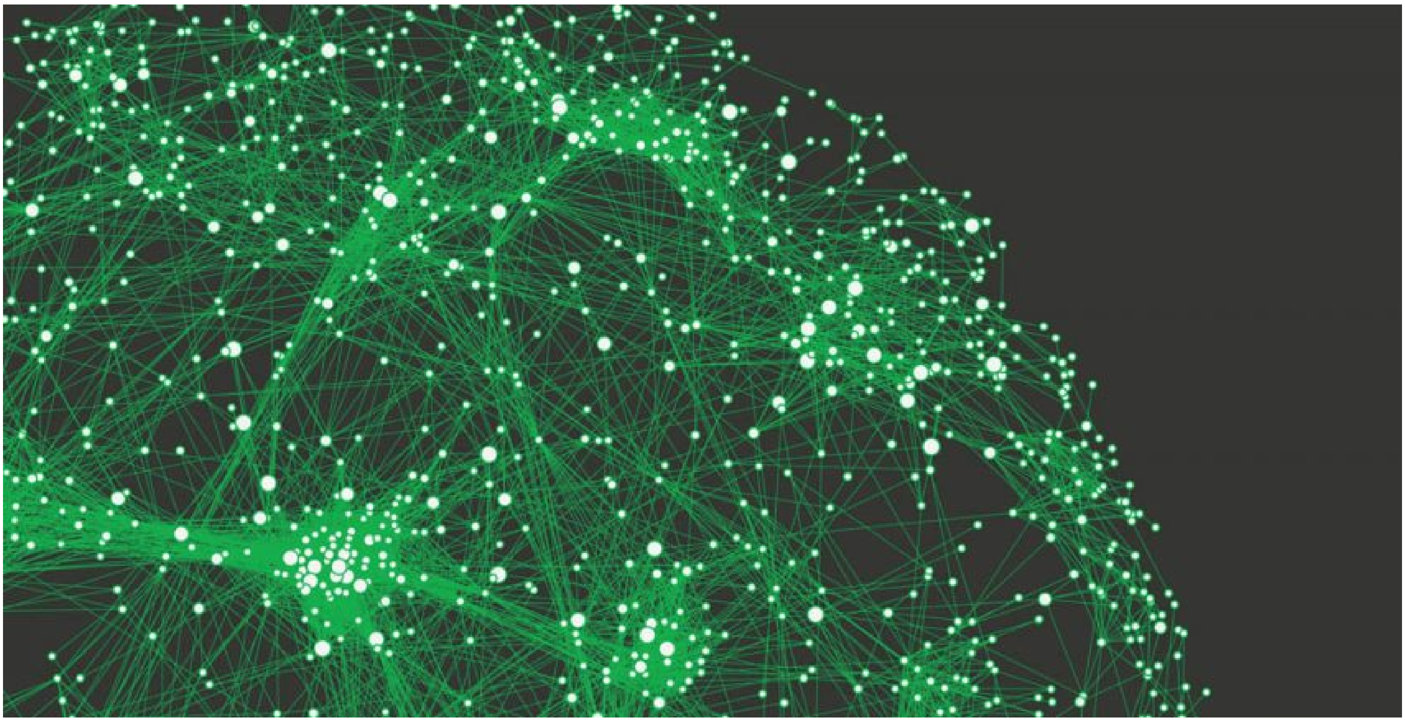
- K-Means Clustering is a simple yet powerful algorithm in data science
- There are a plethora of real-world applications of K-Means Clustering (a few of which we will cover here)
- This comprehensive guide will introduce you to the world of clustering and K-Means Clustering along with an implementation in Python on a real-world dataset

Introduction

I love working on recommendation engines. Whenever I come across any recommendation engine on a website, I can't wait to break it down and understand how it works underneath. It's one of the many great things about being a data scientist!

What truly fascinates me about these systems is how we can group similar items, products, and users together. This grouping, or segmenting, works across industries. And that's what makes the concept of clustering such an important one in data science.

Clustering helps us understand our data in a unique way – by grouping things together into – you guessed it – clusters.



In this article, we will cover k-means clustering and it's components comprehensively. We'll look at clustering, why it matters, its applications and then deep dive into k-means clustering (including how to perform it in Python on a real-world dataset).

And if you want to directly work on the Python code, jump straight here. We have a live coding window where you can build your own k-means clustering algorithm without leaving this article!

Learn more about clustering and other machine learning algorithms (both supervised and unsupervised) in the comprehensive 'Applied Machine Learning' course.

Table of Contents

What is Clustering?

Let's kick things off with a simple example. A bank wants to give credit card offers to its customers. Currently, they look at the details of each customer and based on this information, decide which offer should be given to which customer.

Now, the bank can potentially have millions of customers. Does it make sense to look at the details of each customer separately and then make a decision? Certainly not! It is a manual process and will take a huge amount of time.

So what can the bank do? One option is to segment its customers into different groups. For instance, the bank can group the customers based on their income:

Can you see where I'm going with this? The bank can now make three different strategies or offers, one for each group. Here, instead of creating different strategies for individual customers, they only have to make 3 strategies. This will reduce the effort as well as the time.

The groups I have shown above are known as clusters and the process of creating these groups is known as clustering. Formally, we can say that:

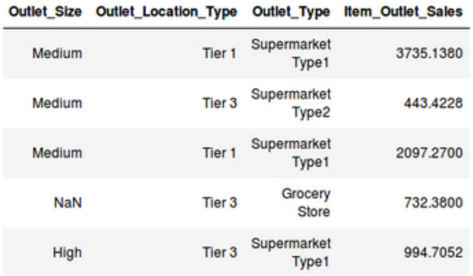
> *Clustering is the process of dividing the entire data into groups (also known as clusters) based on the patterns in the data.*

Can you guess which type of learning problem clustering is? Is it a supervised or unsupervised learning problem?

Think about it for a moment and make use of the example we just saw. Got it? Clustering is an unsupervised learning problem!

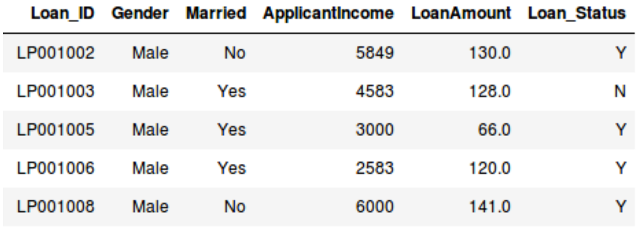
How is Clustering an Unsupervised Learning Problem?

Let's say you are working on a project where you need to predict the sales of a big mart:



Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
Medium	Tier 1	Supermarket Type1	3735.1380
Medium	Tier 3	Supermarket Type2	443.4228
Medium	Tier 1	Supermarket Type1	2097.2700
NaN	Tier 3	Grocery Store	732.3800
High	Tier 3	Supermarket Type1	994.7052

Or, a project where your task is to predict whether a loan will be approved or not:



Loan_ID	Gender	Married	ApplicantIncome	LoanAmount	Loan_Status
LP001002	Male	No	5849	130.0	Y
LP001003	Male	Yes	4583	128.0	N
LP001005	Male	Yes	3000	66.0	Y
LP001006	Male	Yes	2583	120.0	Y
LP001008	Male	No	6000	141.0	Y

We have a fixed target to predict in both of these situations. In the sales prediction problem, we have to predict the *Item_Outlet_Sales* based on *outlet_size, outlet_location_type*, etc. and in the loan approval problem, we have to predict the *Loan_Status* depending on the *Gender, marital status, the income of the customers, etc*.

> So, when we have a target variable to predict based on a given set of predictors or independent variables, such problems are called supervised learning problems.

Now, there might be situations where we do *not* have any target variable to predict.

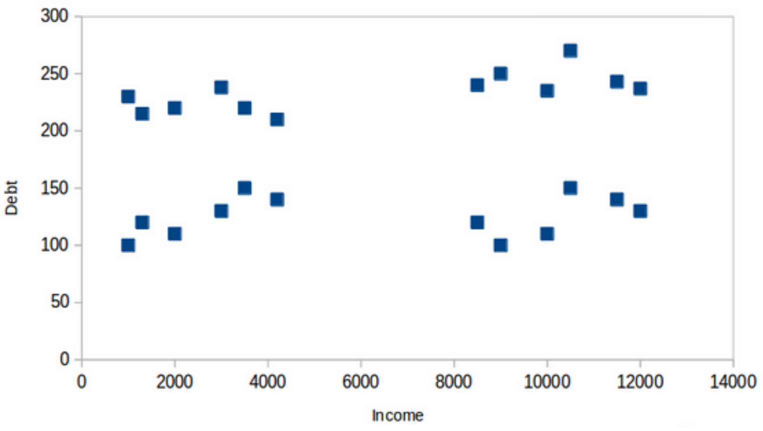
> Such problems, without any fixed target variable, are known as unsupervised learning problems. In these problems, we only have the independent variables and no target/dependent variable.

In clustering, we do not have a target to predict. We look at the data and then try to club similar observations and form different groups. Hence it is an unsupervised learning problem.

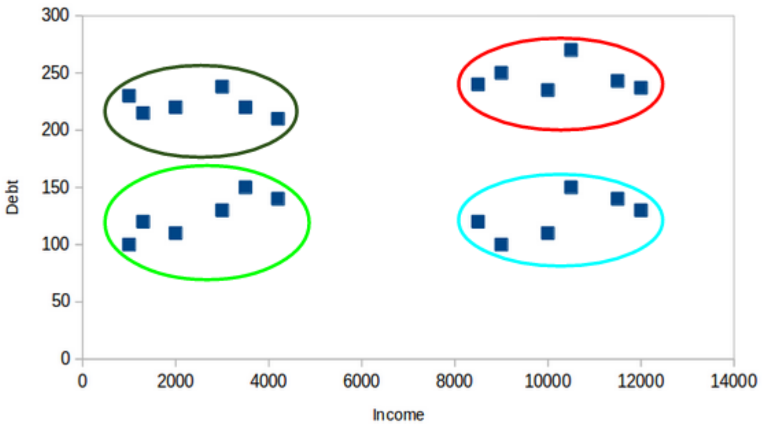
We now know what are clusters and the concept of clustering. Next, let's look at the properties of these clusters which we must consider while forming the clusters.

Properties of Clusters

How about another example? We'll take the same bank as before who wants to segment its customers. For simplicity purposes, let's say the bank only wants to use the income and debt to make the segmentation. They collected the customer data and used a scatter plot to visualize it:



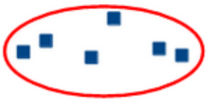
On the X-axis, we have the income of the customer and the y-axis represents the amount of debt. Here, we can clearly visualize that these customers can be segmented into 4 different clusters as shown below:



This is how clustering helps to create segments (clusters) from the data. The bank can further use these clusters to make strategies and offer discounts to its customers. So let's look at the properties of these clusters.

Property 1

All the data points in a cluster should be similar to each other. Let me illustrate it using the above example:

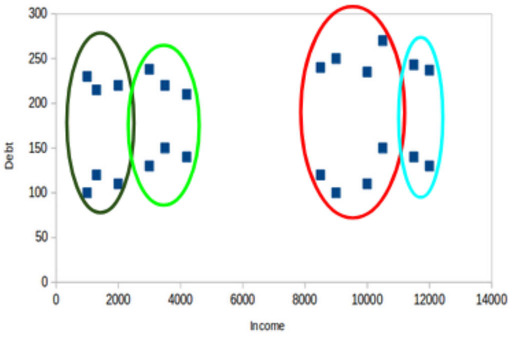


If the customers in a particular cluster are not similar to each other, then their requirements might vary, right? If the bank gives them the same offer, they might not like it and their interest in the bank might reduce. Not ideal.

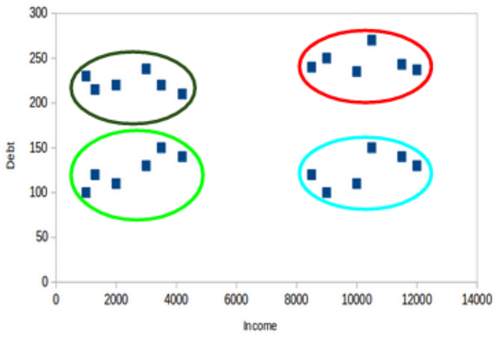
Having similar data points within the same cluster helps the bank to use targeted marketing. You can think of similar examples from your everyday life and think about how clustering will (or already does) impact the business strategy.

Property 2

The data points from different clusters should be as different as possible. This will intuitively make sense if you grasped the above property. Let's again take the same example to understand this property:

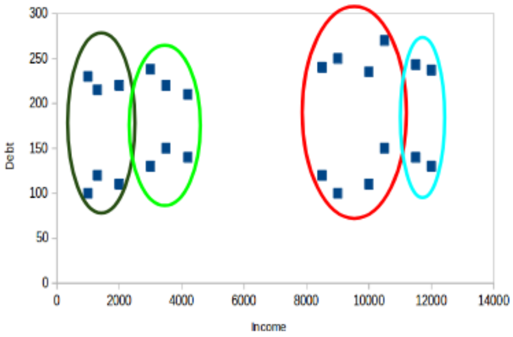


Case - I



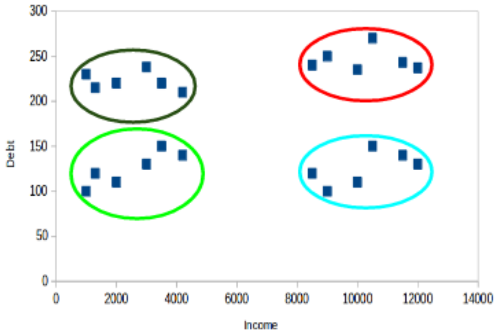
Case - II

Which of these cases do you think will give us the better clusters? If you look at case I:



Case - I

Customers in the red and blue clusters are quite similar to each other. The top four points in the red cluster share similar properties as that of the top two customers in the blue cluster. They have high income and high debt value. Here, we have clustered them differently. Whereas, if you look at case II:



Case - II

Points in the red cluster are completely different from the customers in the blue cluster. All the customers in the red cluster have high income and high debt and customers in the blue cluster have high income and low debt value. Clearly we have a better clustering of customers in this case.

Hence, data points from different clusters should be as different from each other as possible to have more meaningful clusters.

So far, we have understood what clustering is and the different properties of clusters. But why do we even need clustering? Let’s clear this doubt in the next section and look at some applications of clustering.

Applications of Clustering in Real-World Scenarios

Clustering is a widely used technique in the industry. It is actually being used in almost every domain, ranging from banking to recommendation engines, document clustering to image segmentation.

Customer Segmentation

We covered this earlier – one of the most common applications of clustering is customer segmentation. And it isn’t just limited to banking. This strategy is across functions, including telecom, e-commerce, sports, advertising, sales, etc.

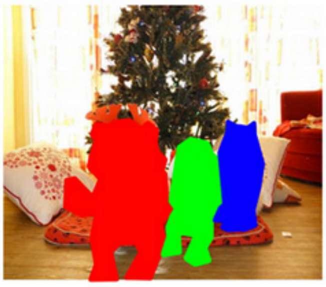
Document Clustering

This is another common application of clustering. Let’s say you have multiple documents and you need to cluster similar documents together. Clustering helps us group these documents such that similar documents are in the same clusters.

Document Clustering

Image Segmentation

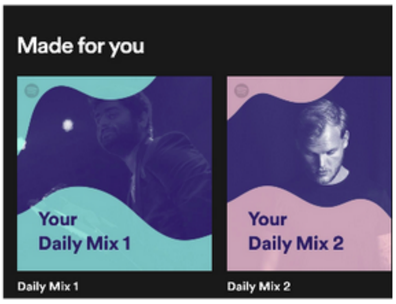
We can also use clustering to perform image segmentation. Here, we try to club similar pixels in the image together. We can apply clustering to create clusters having similar pixels in the same group.



You can refer to this article to see how we can make use of clustering for image segmentation tasks.

Recommendation Engines

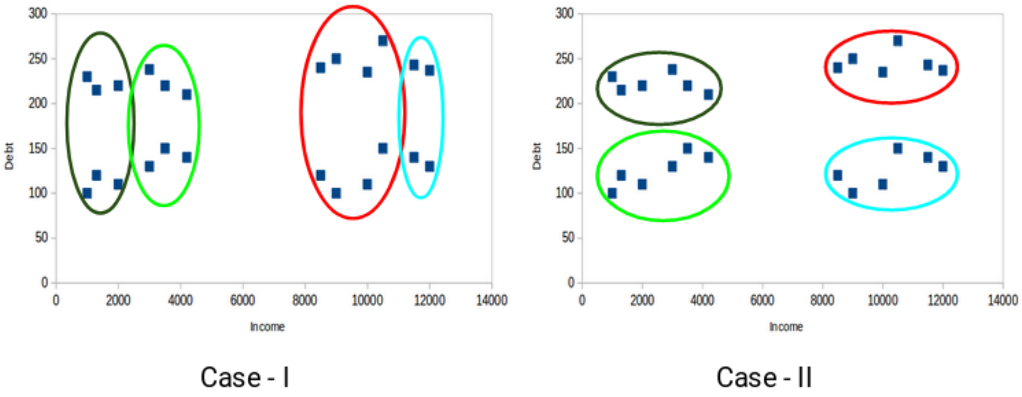
Clustering can also be used in recommendation engines. Let’s say you want to recommend songs to your friends. You can look at the songs liked by that person and then use clustering to find similar songs and finally recommend the most similar songs.



There are many more applications which I'm sure you have already thought of. You can share these applications in the comments section below. Next, let's look at how we can evaluate our clusters.

Understanding the Different Evaluation Metrics for Clustering

The primary aim of clustering is not just to make clusters, but to make good and meaningful ones. We saw this in the below example:



Here, we used only two features and hence it was easy for us to visualize and decide which of these clusters is better.

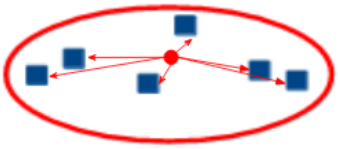
Unfortunately, that's not how real-world scenarios work. We will have a ton of features to work with. Let's take the customer segmentation example again – we will have features like customer's income, occupation, gender, age, and many more. Visualizing all these features together and deciding better and meaningful clusters would not be possible for us.

This is where we can make use of evaluation metrics. Let's discuss a few of them and understand how we can use them to evaluate the quality of our clusters.

Inertia

Recall the first property of clusters we covered above. This is what inertia evaluates. It tells us how far the points within a cluster are. So, **inertia actually calculates the sum of distances of all the points within a cluster from the centroid of that cluster.**

We calculate this for all the clusters and the final inertial value is the sum of all these distances. This distance within the clusters is known as **intracluster distance**. So, inertia gives us the sum of intracluster distances:



Intra cluster distance

Now, what do you think should be the value of inertia for a good cluster? Is a small inertial value good or do we need a larger value? We want the points within the same cluster to be similar to each other, right? Hence, **the distance between them should be as low as possible**.

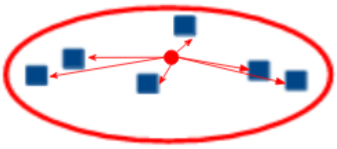
> Keeping this in mind, we can say that the lesser the inertia value, the better our clusters are.

Dunn Index

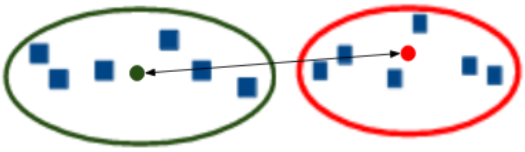
We now know that inertia tries to minimize the intracluster distance. It is trying to make more compact clusters.

Let me put it this way – if the distance between the centroid of a cluster and the points in that cluster is small, it means that the points are closer to each other. So, inertia makes sure that the first property of clusters is satisfied. But it does not care about the second property – that different clusters should be as different from each other as possible.

This is where Dunn index can come into action.



Intra cluster distance



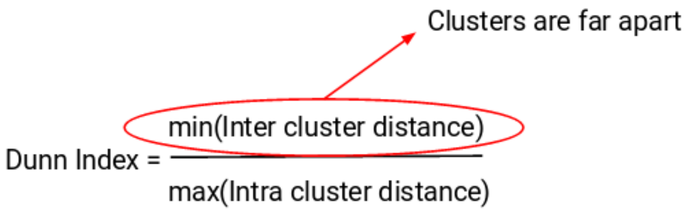
Inter cluster distance

Along with the distance between the centroid and points, **the Dunn index also takes into account the distance between two clusters**. This distance between the centroids of two different clusters is known as **inter-cluster distance**. Let's look at the formula of the Dunn index:

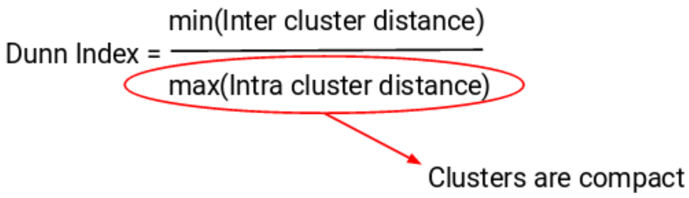
$$\text{Dunn Index} = \frac{\text{min(Inter cluster distance)}}{\text{max(Intra cluster distance)}}$$

> Dunn index is the ratio of the minimum of inter-cluster distances and maximum of intracluster distances.

We want to maximize the Dunn index. The more the value of the Dunn index, the better will be the clusters. Let’s understand the intuition behind Dunn index:



In order to maximize the value of the Dunn index, the numerator should be maximum. Here, we are taking the minimum of the inter-cluster distances. So, the distance between even the closest clusters should be more which will eventually make sure that the clusters are far away from each other.



Also, the denominator should be minimum to maximize the Dunn index. Here, we are taking the maximum of intracluster distances. Again, the intuition is the same here. The maximum distance between the cluster centroids and the points should be minimum which will eventually make sure that the clusters are compact.

Introduction to K-Means Clustering

We have finally arrived at the meat of this article!

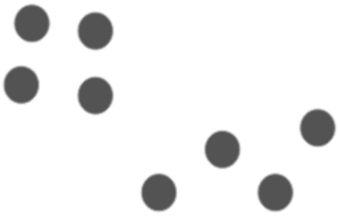
Recall the first property of clusters – it states that the points within a cluster should be similar to each other. So, **our aim here is to minimize the distance between the points within a cluster.**

> There is an algorithm that tries to minimize the distance of the points in a cluster with their centroid – the k-means clustering technique.

K-means is a centroid-based algorithm, or a distance-based algorithm, where we calculate the distances to assign a point to a cluster. In K-Means, each cluster is associated with a centroid.

> ***The main objective of the K-Means algorithm is to minimize the sum of distances between the points and their respective cluster centroid.***

Let’s now take an example to understand how K-Means actually works:



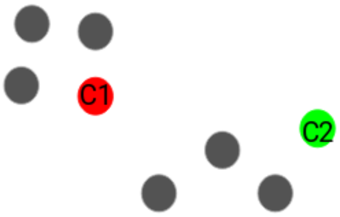
We have these 8 points and we want to apply k-means to create clusters for these points. Here's how we can do it.

Step 1: Choose the number of clusters *k*

The first step in k-means is to pick the number of clusters, k.

Step 2: Select k random points from the data as centroids

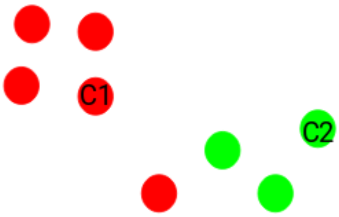
Next, we randomly select the centroid for each cluster. Let's say we want to have 2 clusters, so k is equal to 2 here. We then randomly select the centroid:



Here, the red and green circles represent the centroid for these clusters.

Step 3: Assign all the points to the closest cluster centroid

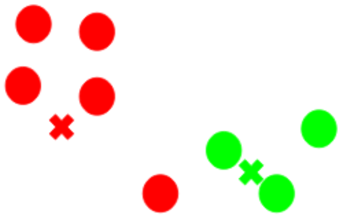
Once we have initialized the centroids, we assign each point to the closest cluster centroid:



Here you can see that the points which are closer to the red point are assigned to the red cluster whereas the points which are closer to the green point are assigned to the green cluster.

Step 4: Recompute the centroids of newly formed clusters

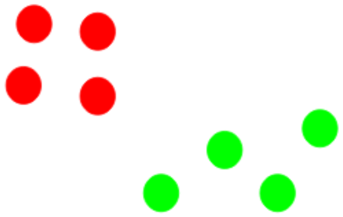
Now, once we have assigned all of the points to either cluster, the next step is to compute the centroids of newly formed clusters:



Here, the red and green crosses are the new centroids.

Step 5: Repeat steps 3 and 4

We then repeat steps 3 and 4:



The step of computing the centroid and assigning all the points to the cluster based on their distance from the centroid is a single iteration. But wait – when should we stop this process? It can't run till eternity, right?

Stopping Criteria for K-Means Clustering

There are essentially three stopping criteria that can be adopted to stop the K-means algorithm:

1. Centroids of newly formed clusters do not change
2. Points remain in the same cluster
3. Maximum number of iterations are reached

We can stop the algorithm if the centroids of newly formed clusters are not changing. Even after multiple iterations, if we are getting the same centroids for all the clusters, we can say that the algorithm is not learning any new pattern and it is a sign to stop the training.

Another clear sign that we should stop the training process if the points remain in the same cluster even after training the algorithm for multiple iterations.

Finally, we can stop the training if the maximum number of iterations is reached. Suppose if we have set the number of iterations as 100. The process will repeat for 100 iterations before stopping.

Implementing K-Means Clustering in Python from Scratch

Time to fire up our Jupyter notebooks (or whichever IDE you use) and get our hands dirty in Python!

We will be working on the loan prediction dataset that you can download here. I encourage you to read more about the dataset and the problem statement here. This will help you visualize what we are working on (and why we are doing this). Two pretty important questions in any data science project.

First, import all the required libraries:

```python
#import libraries
import pandas as pd
import numpy as np
import random as rd
import matplotlib.pyplot as plt
```

view raw

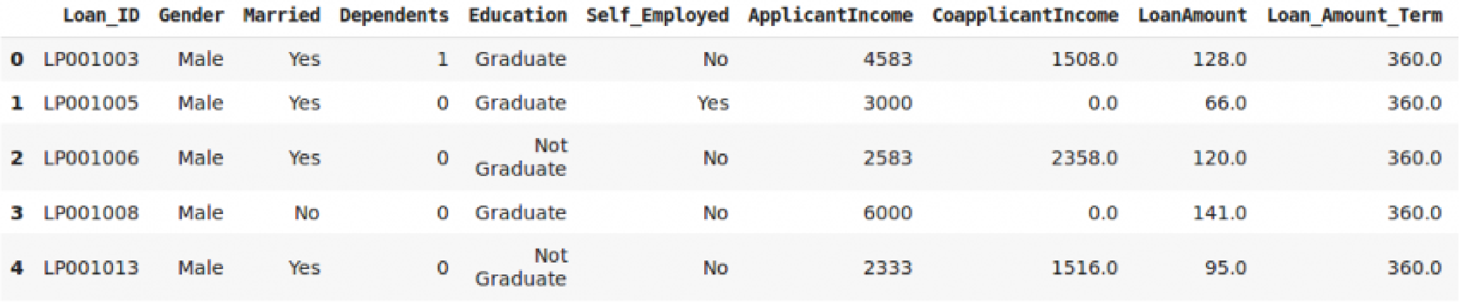
library.py hosted with ❤ by GitHub

Now, we will read the CSV file and look at the first five rows of the data:

```python
data = pd.read_csv('clustering.csv')
data.head()
```

view raw

data.py hosted with ❤ by GitHub



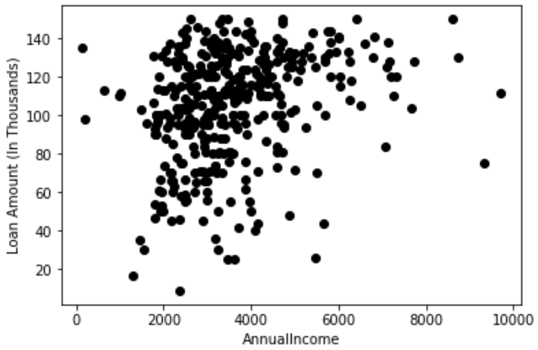
	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
0	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0
1	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0
2	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0
3	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0
4	LP001013	Male	Yes	0	Not Graduate	No	2333	1516.0	95.0	360.0

For this article, we will be taking only two variables from the data – “LoanAmount” and “ApplicantIncome”. This will make it easy to visualize the steps as well. Let’s pick these two variables and visualize the data points:

```python
X = data[["LoanAmount","ApplicantIncome"]]
#Visualise data points
plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')
plt.xlabel('AnnualIncome')
plt.ylabel('Loan Amount (In Thousands)')
plt.show()
```

view raw

visualization.py hosted with ❤ by GitHub

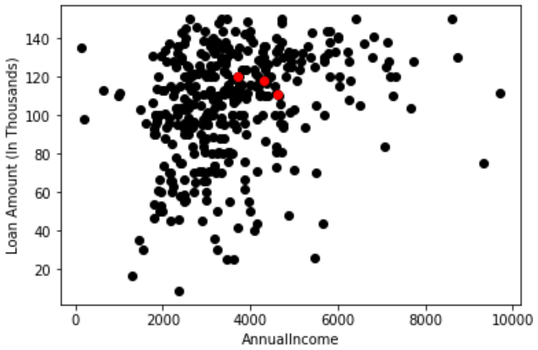


Steps 1 and 2 of K-Means were about choosing the number of clusters (k) and selecting random centroids for each cluster. We will pick 3 clusters and then select random observations from the data as the centroids:

```
# Step 1 and 2 - Choose the number of clusters (k) and select random centroid for each cluster

#number of clusters
K=3

# Select random observation as centroids
Centroids = (X.sample(n=K))
plt.scatter(X["ApplicantIncome"],X["LoanAmount"],c='black')
plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')
plt.xlabel('AnnualIncome')
plt.ylabel('Loan Amount (In Thousands)')
plt.show()
```

view raw

random_initialize.py hosted with ❤ by **GitHub**



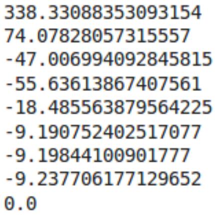
Here, the red dots represent the 3 centroids for each cluster. Note that we have chosen these points randomly and hence every time you run this code, you might get different centroids.

Next, we will define some conditions to implement the K-Means Clustering algorithm. Let's first look at the code:

```
# Step 3 - Assign all the points to the closest cluster centroid
# Step 4 - Recompute centroids of newly formed clusters
# Step 5 - Repeat step 3 and 4

diff = 1
j=0

while(diff!=0):
    XD=X
    i=1
    for index1,row_c in Centroids.iterrows():
        ED=[]
        for index2,row_d in XD.iterrows():
            d1=(row_c["ApplicantIncome"]-row_d["ApplicantIncome"])**2
            d2=(row_c["LoanAmount"]-row_d["LoanAmount"])**2
            d=np.sqrt(d1+d2)
            ED.append(d)
        X[i]=ED
        i=i+1

    C=[]
    for index,row in X.iterrows():
        min_dist=row[1]
        pos=1
        for i in range(K):
            if row[i+1] < min_dist:
                min_dist = row[i+1]
                pos=i+1
        C.append(pos)
    X["Cluster"]=C
    Centroids_new = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]
    if j == 0:
        diff=1
        j=j+1
    else:
        diff = (Centroids_new['LoanAmount'] - Centroids['LoanAmount']).sum() + (Centroids_new['ApplicantIncome'] - Centroids['Ap
        print(diff.sum())
    Centroids = X.groupby(["Cluster"]).mean()[["LoanAmount","ApplicantIncome"]]
```

view raw

kmeans_scratch.py hosted with ❤ by **GitHub**



```
338.33088353093154
74.07828057315557
-47.006994092845815
-55.63613867407561
-18.485563879564225
-9.190752402517077
-9.19844100901777
-9.237706177129652
0.0
```

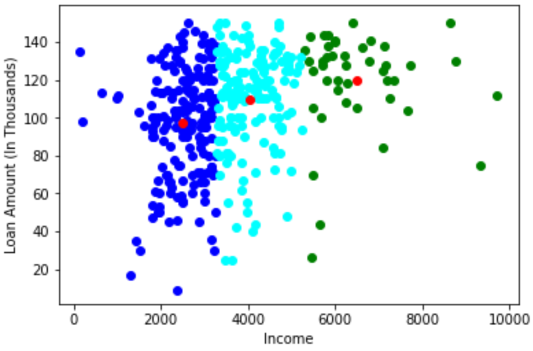
These values might vary every time we run this. Here, we are stopping the training when the centroids are not changing after two iterations. We have initially defined the *diff* as 1 and inside the while loop, we are calculating this *diff* as the difference between the centroids in the previous iteration and the current iteration.

When this difference is 0, we are stopping the training. Let’s now visualize the clusters we have got:

```
color=['blue','green','cyan']
for k in range(K):
    data=X[X["Cluster"]==k+1]
    plt.scatter(data["ApplicantIncome"],data["LoanAmount"],c=color[k])
plt.scatter(Centroids["ApplicantIncome"],Centroids["LoanAmount"],c='red')
plt.xlabel('Income')
plt.ylabel('Loan Amount (In Thousands)')
plt.show()
```

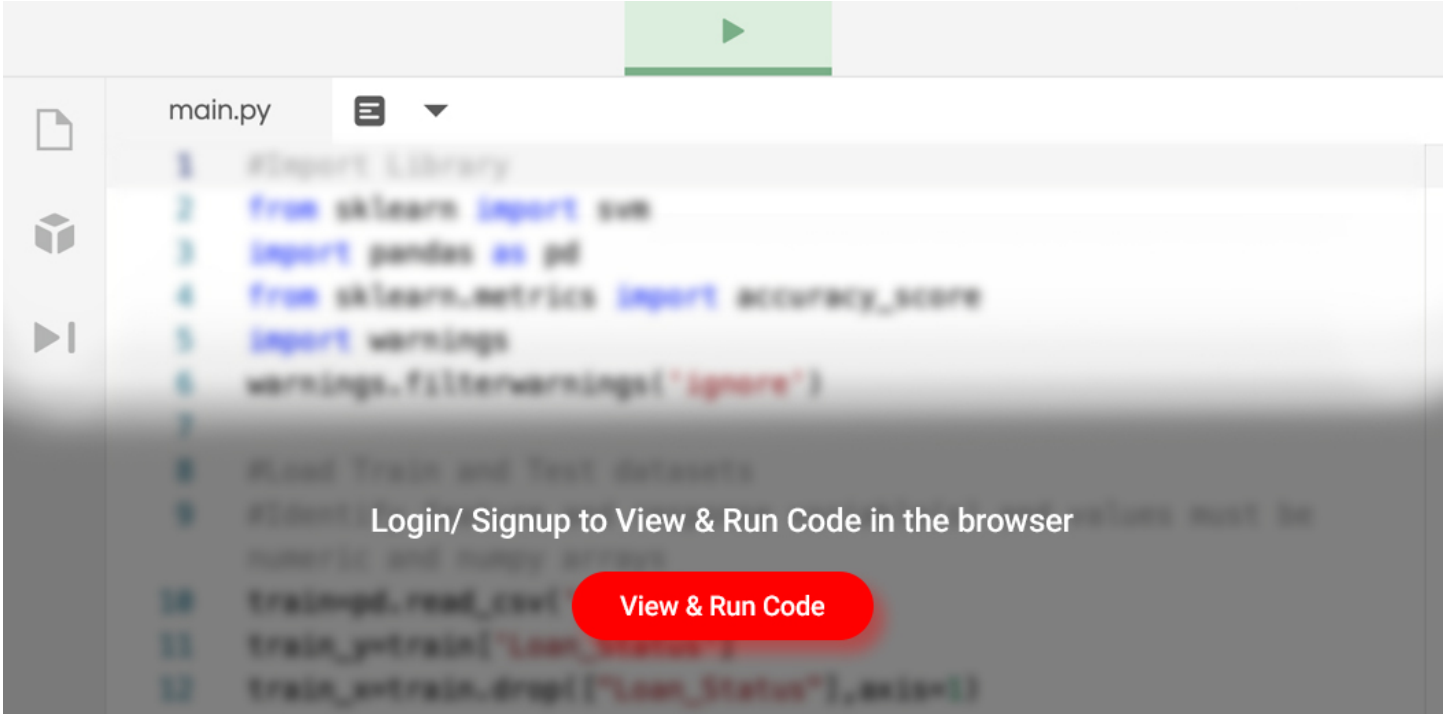
view raw

cluster_visualization.py hosted with ❤ by **GitHub**



Awesome! Here, we can clearly visualize three clusters. The red dots represent the centroid of each cluster. I hope you now have a clear understanding of how K-Means work.

Here is a LIVE CODING window for you to play around with the code and see the results for yourself – without leaving this article! Go ahead and start working on it:

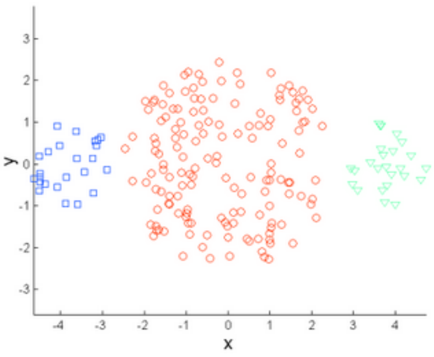


```
#Import Library
from sklearn import svm
import pandas as pd
from sklearn.metrics import accuracy_score
import warnings
warnings.filterwarnings('ignore')

#Load Train and Test datasets
#Identify feature and response variable(s) and values must be
numeric and numpy arrays
train=pd.read_csv(
train_y=train['Loan_Status']
train_x=train.drop(["Loan_Status"],axis=1)
```

However, there are certain situations where this algorithm might not perform as well. Let's look at some challenges which you can face while working with k-means.

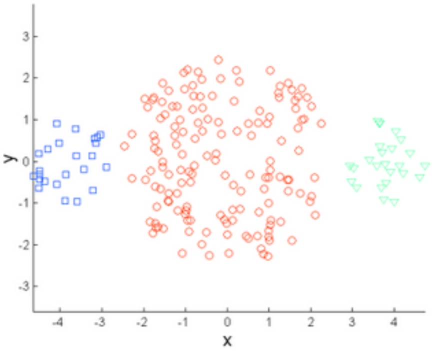
Challenges with the K-Means Clustering Algorithm

One of the common challenges we face while working with K-Means is that the size of clusters is different. Let's say we have the below points:

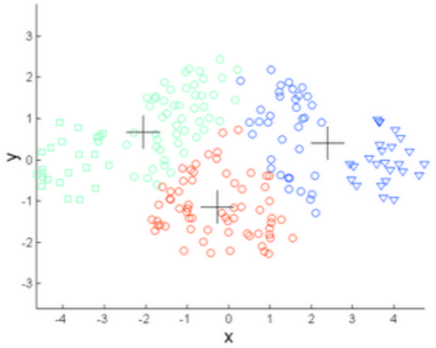


Original Points

The left and the rightmost clusters are of smaller size compared to the central cluster. Now, if we apply k-means clustering on these points, the results will be something like this:

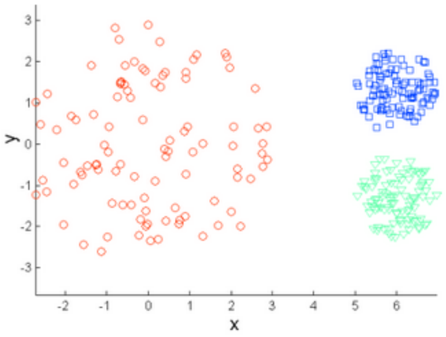


Original Points



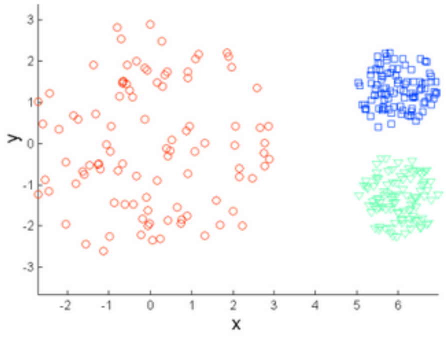
K-means (k = 3)

Another challenge with k-means is when the densities of the original points are different. Let's say these are the original points:

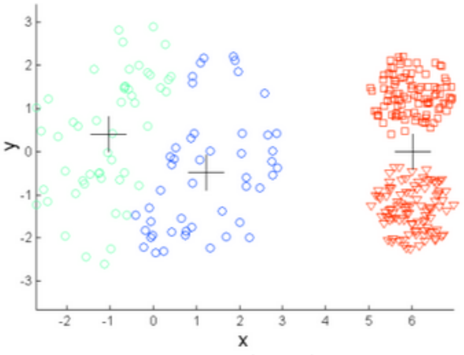


Original Points

Here, the points in the red cluster are spread out whereas the points in the remaining clusters are closely packed together. Now, if we apply k-means on these points, we will get clusters like this:



Original Points



K-means (k = 3)

We can see that the compact points have been assigned to a single cluster. Whereas the points that are spread loosely but were in the same cluster, have been assigned to different clusters. Not ideal so what can we do about this?

One of the solutions is to use a higher number of clusters. So, in all the above scenarios, instead of using 3 clusters, we can have a bigger number. Perhaps setting k=10 might lead to more meaningful clusters.

Remember how we randomly initialize the centroids in k-means clustering? Well, this is also potentially problematic because we might get different clusters every time. So, to solve this problem of random initialization, there is an algorithm called **K-Means++** that can be used to choose the initial values, or the initial cluster centroids, for K-Means.

K-Means++ to Choose Initial Cluster Centroids for K-Means Clustering

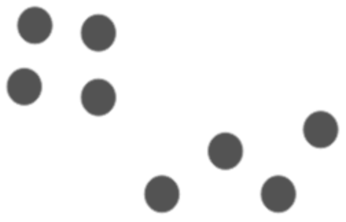
In some cases, if the initialization of clusters is not appropriate, K-Means can result in arbitrarily bad clusters. This is where K-Means++ helps. **It specifies a procedure to initialize the cluster centers before moving forward with the standard k-means clustering algorithm.**

Using the K-Means++ algorithm, we optimize the step where we randomly pick the cluster centroid. We are more likely to find a solution that is competitive to the optimal K-Means solution while using the K-Means++ initialization.

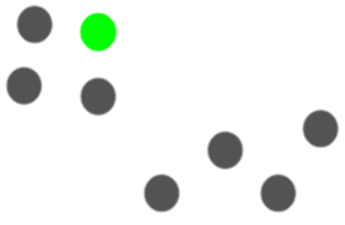
The steps to initialize the centroids using K-Means++ are:

1. The first cluster is chosen uniformly at random from the data points that we want to cluster. This is similar to what we do in K-Means, but instead of randomly picking all the centroids, we just pick one centroid here
2. Next, we compute the distance (D(x)) of each data point (x) from the cluster center that has already been chosen
3. Then, choose the new cluster center from the data points with the probability of x being proportional to (D(x))2
4. We then repeat steps 2 and 3 until *k* clusters have been chosen

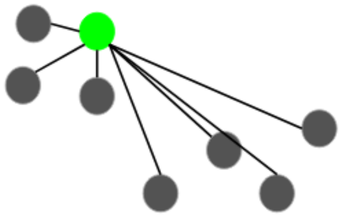
Let's take an example to understand this more clearly. Let's say we have the following points and we want to make 3 clusters here:



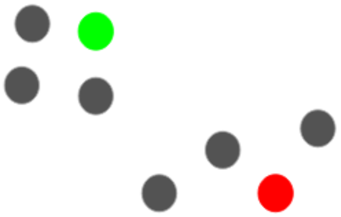
Now, the first step is to randomly pick a data point as a cluster centroid:



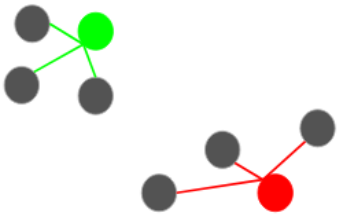
Let's say we pick the green point as the initial centroid. Now, we will calculate the distance (D(x)) of each data point with this centroid:



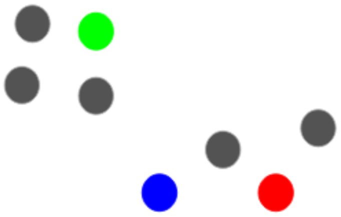
The next centroid will be the one whose squared distance (D(x)2) is the farthest from the current centroid:



In this case, the red point will be selected as the next centroid. Now, to select the last centroid, we will take the distance of each point from its closest centroid and the point having the largest squared distance will be selected as the next centroid:



We will select the last centroid as:



We can continue with the K-Means algorithm after initializing the centroids. Using K-Means++ to initialize the centroids tends to improve the clusters. Although it is computationally costly relative to random initialization, subsequent K-Means often converge more rapidly.

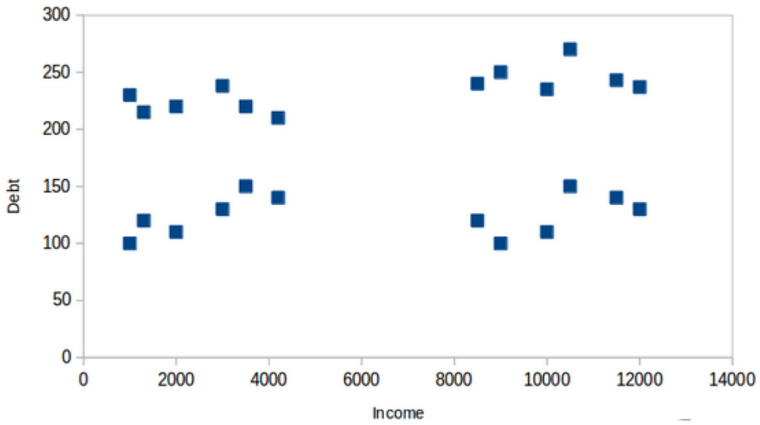
I'm sure there's one question which you've been wondering about since the start of this article – how many clusters should we make? Aka, what should be the optimum number of clusters to have while performing

K-Means?

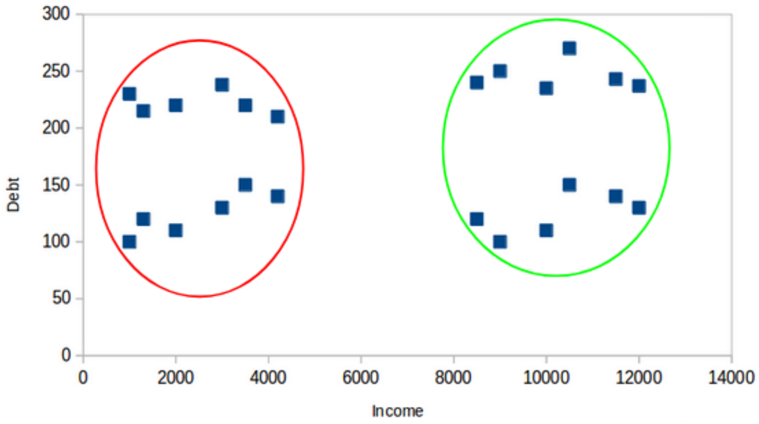
How to Choose the Right Number of Clusters in K-Means Clustering?

One of the most common doubts everyone has while working with K-Means is selecting the right number of clusters.

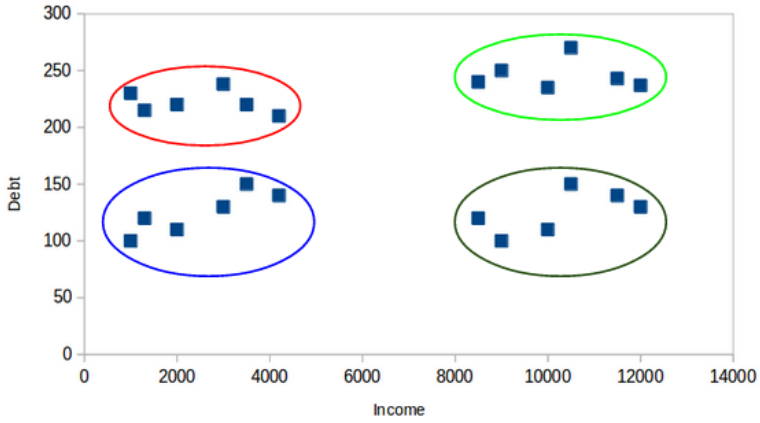
So, let’s look at a technique that will help us choose the right value of clusters for the K-Means algorithm. Let’s take the customer segmentation example which we saw earlier. To recap, the bank wants to segment its customers based on their income and amount of debt:



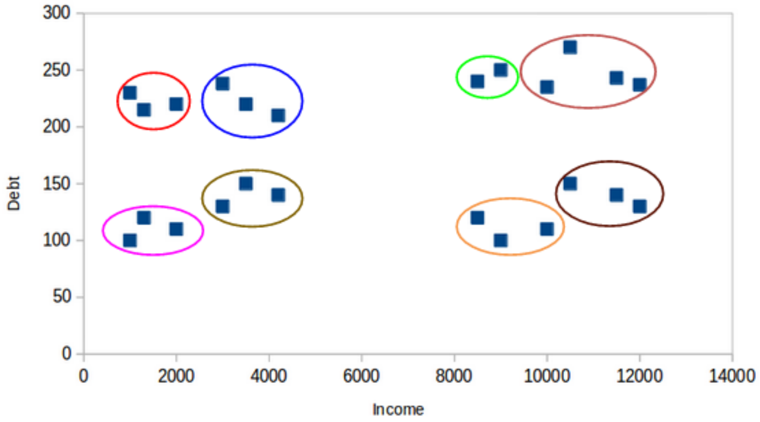
Here, we can have two clusters which will separate the customers as shown below:



All the customers with low income are in one cluster whereas the customers with high income are in the second cluster. We can also have 4 clusters:



Here, one cluster might represent customers who have low income and low debt, other cluster is where customers have high income and high debt, and so on. There can be 8 clusters as well:

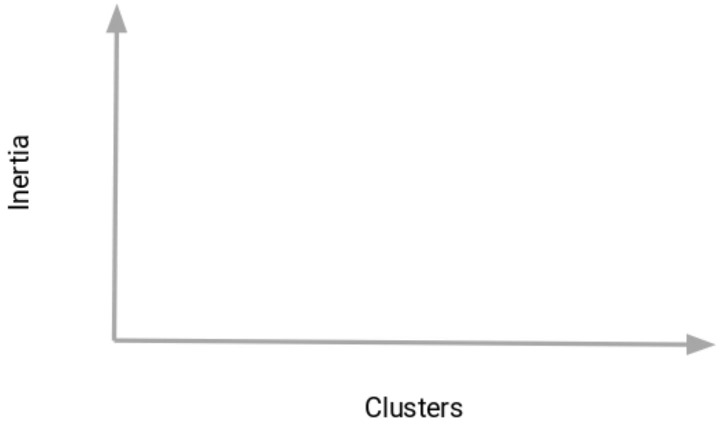


Honestly, we can have any number of clusters. Can you guess what would be the maximum number of possible clusters? One thing which we can do is to assign each point to a separate cluster. Hence, in this case, the number of clusters will be equal to the number of points or observations. So,

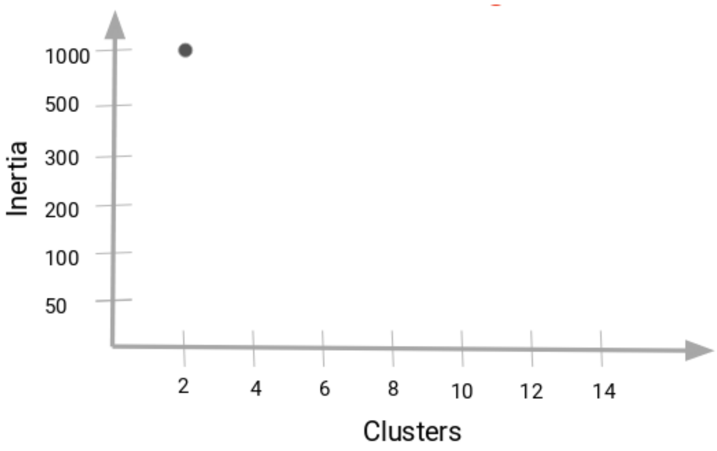
> ***The maximum possible number of clusters will be equal to the number of observations in the dataset.***

But then how can we decide the optimum number of clusters? **One thing we can do is plot a graph, also known as an elbow curve, where the x-axis will represent the number of clusters and the y-axis will be an evaluation metric.** Let’s say inertia for now.

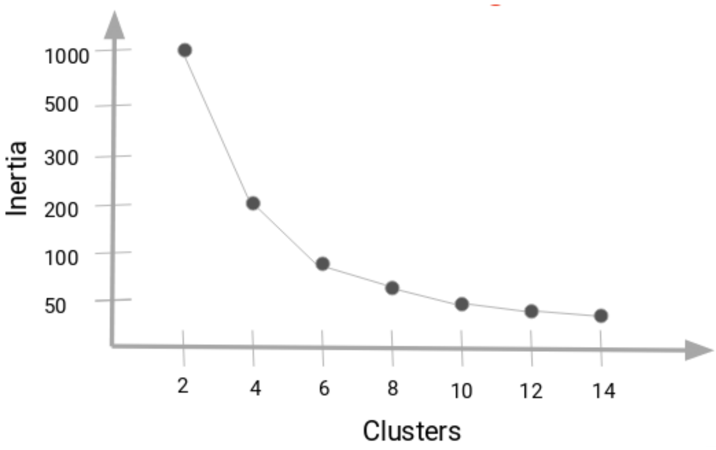
You can choose any other evaluation metric like the Dunn index as well:



Next, we will start with a small cluster value, let's say 2. Train the model using 2 clusters, calculate the inertia for that model, and finally plot it in the above graph. Let's say we got an inertia value of around 1000:



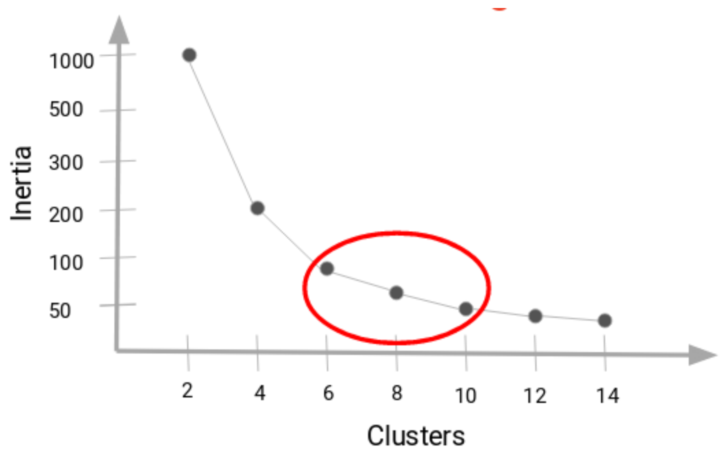
Now, we will increase the number of clusters, train the model again, and plot the inertia value. This is the plot we get:



When we changed the cluster value from 2 to 4, the inertia value reduced very sharply. This decrease in the inertia value reduces and eventually becomes constant as we increase the number of clusters further.

So,

the cluster value where this decrease in inertia value becomes constant can be chosen as the right cluster value for our data.



Here, we can choose any number of clusters between 6 and 10. We can have 7, 8, or even 9 clusters. **You must also look at the computation cost while deciding the number of clusters.** If we increase the number of clusters, the computation cost will also increase. So, if you do not have high computational resources, my advice is to choose a lesser number of clusters.

Let's now implement the K-Means Clustering algorithm in Python. We will also see how to use K-Means++ to initialize the centroids and will also plot this elbow curve to decide what should be the right number of clusters for our dataset.

Implementing K-Means Clustering in Python

We will be working on a wholesale customer segmentation problem. You can download the dataset using this link. The data is hosted on the UCI Machine Learning repository.

The aim of this problem is to segment the clients of a wholesale distributor based on their annual spending on diverse product categories, like milk, grocery, region, etc. So, let's start coding!

We will first import the required libraries:

```
# importing required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.cluster import KMeans
```

view raw

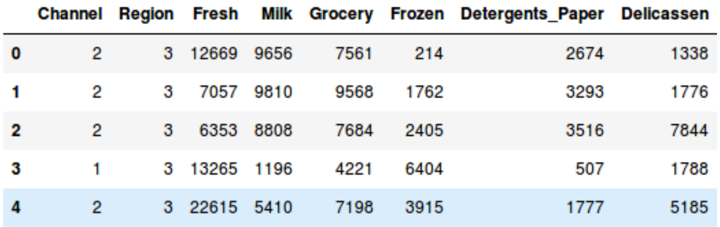
library_2.py hosted with ❤ by **GitHub**

Next, let's read the data and look at the first five rows:

```
# reading the data and looking at the first five rows of the data
data=pd.read_csv("Wholesale customers data.csv")
data.head()
```

view raw

wholesale_data.py hosted with ❤ by **GitHub**



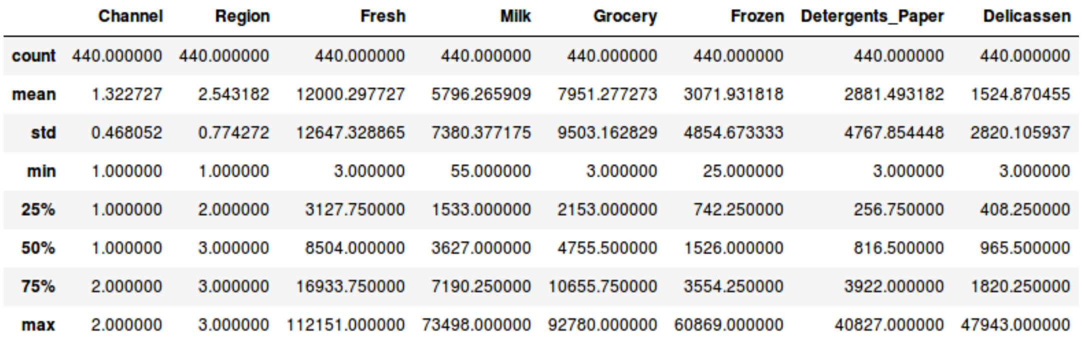
	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185

We have the spending details of customers on different products like Milk, Grocery, Frozen, Detergents, etc. Now, we have to segment the customers based on the provided details. Before doing that, let's pull out some statistics related to the data:

```python
# statistics of the data
data.describe()
```

view raw

data_statistics.py hosted with ❤ by GitHub



	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
count	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000
mean	1.322727	2.543182	12000.297727	5796.265909	7951.277273	3071.931818	2881.493182	1524.870455
std	0.468052	0.774272	12647.328865	7380.377175	9503.162829	4854.673333	4767.854448	2820.105937
min	1.000000	1.000000	3.000000	55.000000	3.000000	25.000000	3.000000	3.000000
25%	1.000000	2.000000	3127.750000	1533.000000	2153.000000	742.250000	256.750000	408.250000
50%	1.000000	3.000000	8504.000000	3627.000000	4755.500000	1526.000000	816.500000	965.500000
75%	2.000000	3.000000	16933.750000	7190.250000	10655.750000	3554.250000	3922.000000	1820.250000
max	2.000000	3.000000	112151.000000	73498.000000	92780.000000	60869.000000	40827.000000	47943.000000

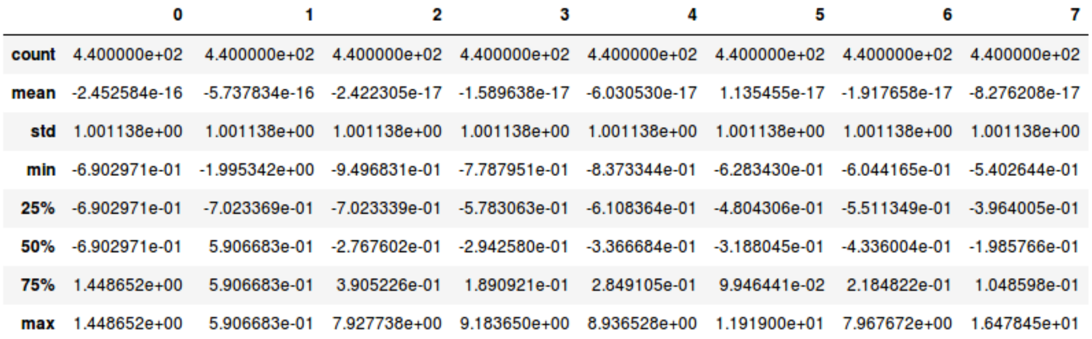
Here, we see that there is a lot of variation in the magnitude of the data. Variables like Channel and Region have low magnitude whereas variables like Fresh, Milk, Grocery, etc. have a higher magnitude.

Since K-Means is a distance-based algorithm, this difference of magnitude can create a problem. So let's first bring all the variables to the same magnitude:

```python
# standardizing the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# statistics of scaled data
pd.DataFrame(data_scaled).describe()
```

view raw

standardize.py hosted with ❤ by GitHub



	0	1	2	3	4	5	6	7
count	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02
mean	-2.452584e-16	-5.737834e-16	-2.422305e-17	-1.589638e-17	-6.030530e-17	1.135455e-17	-1.917658e-17	-8.276208e-17
std	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00
min	-6.902971e-01	-1.995342e+00	-9.496831e-01	-7.787951e-01	-8.373344e-01	-6.283430e-01	-6.044165e-01	-5.402644e-01
25%	-6.902971e-01	-7.023369e-01	-7.023339e-01	-5.783063e-01	-6.108364e-01	-4.804306e-01	-5.511349e-01	-3.964005e-01
50%	-6.902971e-01	5.906683e-01	-2.767602e-01	-2.942580e-01	-3.366684e-01	-3.188045e-01	-4.336004e-01	-1.985766e-01
75%	1.448652e+00	5.906683e-01	3.905226e-01	1.890921e-01	2.849105e-01	9.946441e-02	2.184822e-01	1.048598e-01
max	1.448652e+00	5.906683e-01	7.927738e+00	9.183650e+00	8.936528e+00	1.191900e+01	7.967672e+00	1.647845e+01

The magnitude looks similar now. Next, let’s create a *kmeans* function and fit it on the data:

```python
# defining the kmeans function with initialization as k-means++
kmeans = KMeans(n_clusters=2, init='k-means++')

# fitting the k means algorithm on scaled data
kmeans.fit(data_scaled)
```

view raw

kmeans.py hosted with ❤ by **GitHub**

We have initialized two clusters and pay attention – the initialization is not random here. We have used the k-means++ initialization which generally produces better results as we have discussed in the previous section as well.

Let’s evaluate how well the formed clusters are. To do that, we will calculate the inertia of the clusters:

```python
# inertia on the fitted data
kmeans.inertia_
```

view raw

inertia.py hosted with ❤ by **GitHub**

Output: 2599.38555935614

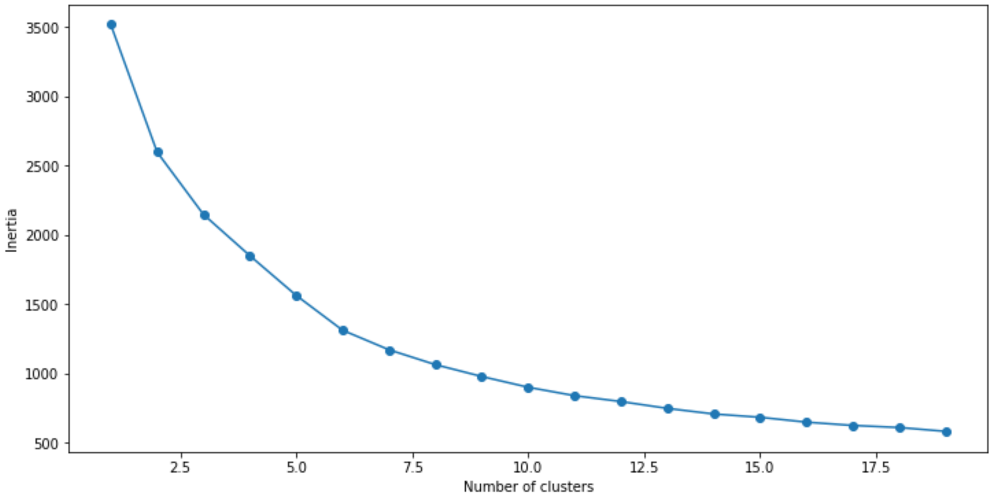
We got an inertia value of almost 2600. Now, let’s see how we can use the elbow curve to determine the optimum number of clusters in Python.

We will first fit multiple k-means models and in each successive model, we will increase the number of clusters. We will store the inertia value of each model and then plot it to visualize the result:

```python
# fitting multiple k-means algorithms and storing the values in an empty list
SSE = []
for cluster in range(1,20):
    kmeans = KMeans(n_jobs = -1, n_clusters = cluster, init='k-means++')
    kmeans.fit(data_scaled)
    SSE.append(kmeans.inertia_)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,20), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

view raw

elbow_curve.py hosted with ❤ by **GitHub**



Can you tell the optimum cluster value from this plot? Looking at the above elbow curve, **we can choose any number of clusters between 5 to 8**. Let's set the number of clusters as 6 and fit the model:

```python
# k means using 5 clusters and k-means++ initialization
kmeans = KMeans(n_jobs = -1, n_clusters = 5, init='k-means++')
kmeans.fit(data_scaled)
pred = kmeans.predict(data_scaled)
```

view raw

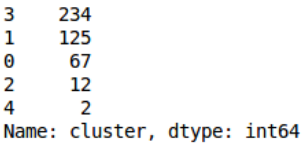
final_kmeans.py hosted with ❤ by **GitHub**

Finally, let's look at the value count of points in each of the above-formed clusters:

```python
frame = pd.DataFrame(data_scaled)
frame['cluster'] = pred
frame['cluster'].value_counts()
```

view raw

cluster_count.py hosted with ❤ by **GitHub**



```
3    234
1    125
0     67
2     12
4      2
Name: cluster, dtype: int64
```

So, there are 234 data points belonging to cluster 4 (index 3), then 125 points in cluster 2 (index 1), and so on. This is how we can implement K-Means Clustering in Python.

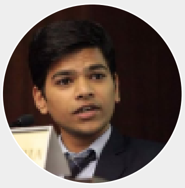
End Notes

In this article, we discussed one of the most famous clustering algorithms – K-Means. We implemented it from scratch and looked at its step-by-step implementation. We looked at the challenges which we might face while working with K-Means and also saw how K-Means++ can be helpful when initializing the cluster centroids.

Finally, we implemented k-means and looked at the elbow curve which helps to find the optimum number of clusters in the K-Means algorithm.

If you have any doubts or feedback, feel free to share them in the comments section below. And make sure you check out the comprehensive '**Applied Machine Learning**' course that takes you from the basics of machine learning to advanced algorithms (including an entire module on deploying your machine learning models!).

Article Url - https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/



Pulkit Sharma

My research interests lies in the field of Machine Learning and Deep Learning. Possess an enthusiasm for learning new skills and technologies.