We will be working on a wholesale customer segmentation problem. we can download the dataset using this link.

link text

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
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
     Mounted at /content/drive
```

```
df = pd.read_csv("/content/drive/My Drive/Colab Notebooks/Wholesale customers data.csv")
```

```
df
```

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338

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

df.head()
```

	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:

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

	Channel	Region	Fresh	Milk	Grocery	Froz
count	440.000000	440.000000	440.000000	440.000000	440.000000	440.0000
mean	1.322727	2.543182	12000.297727	5796.265909	7951.277273	3071.9318
std	0.468052	0.774272	12647.328865	7380.377175	9503.162829	4854.6733
min	1.000000	1.000000	3.000000	55.000000	3.000000	25.0000
25%	1.000000	2.000000	3127.750000	1533.000000	2153.000000	742.2500
50%	1.000000	3.000000	8504.000000	3627.000000	4755.500000	1526.0000
75%	2.000000	3.000000	16933.750000	7190.250000	10655.750000	3554.2500
max	2.000000	3.000000	112151.000000	73498.000000	92780.000000	60869.0000

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:

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

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

	0	1	2	3	4	
count	4.400000e+02	4.400000e+02	4.400000e+02	440.000000	4.400000e+02	4.400000
mean	1.614870e-17	3.552714e-16	-3.431598e-17	0.000000	-4.037175e-17	3.633457
std	1.001138e+00	1.001138e+00	1.001138e+00	1.001138	1.001138e+00	1.001138
min	-6.902971e-01	-1.995342e+00	-9.496831e-01	-0.778795	-8.373344e-01	-6.2834
25%	-6.902971e-01	-7.023369e-01	-7.023339e-01	-0.578306	-6.108364e-01	-4.8043
50%	-6.902971e-01	5.906683e-01	-2.767602e-01	-0.294258	-3.366684e-01	-3.1880

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

```
# 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)
```

```
KMeans(n_clusters=2)
```

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:

Inertia measures how well a dataset was clustered by K-Means.

It is calculated by measuring the distance between each data point and its centroid, squaring this distance, and summing these squares across one cluster.

A good model is one with low inertia AND a low number of clusters (K).

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

```
2599.3873849123092
```

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.

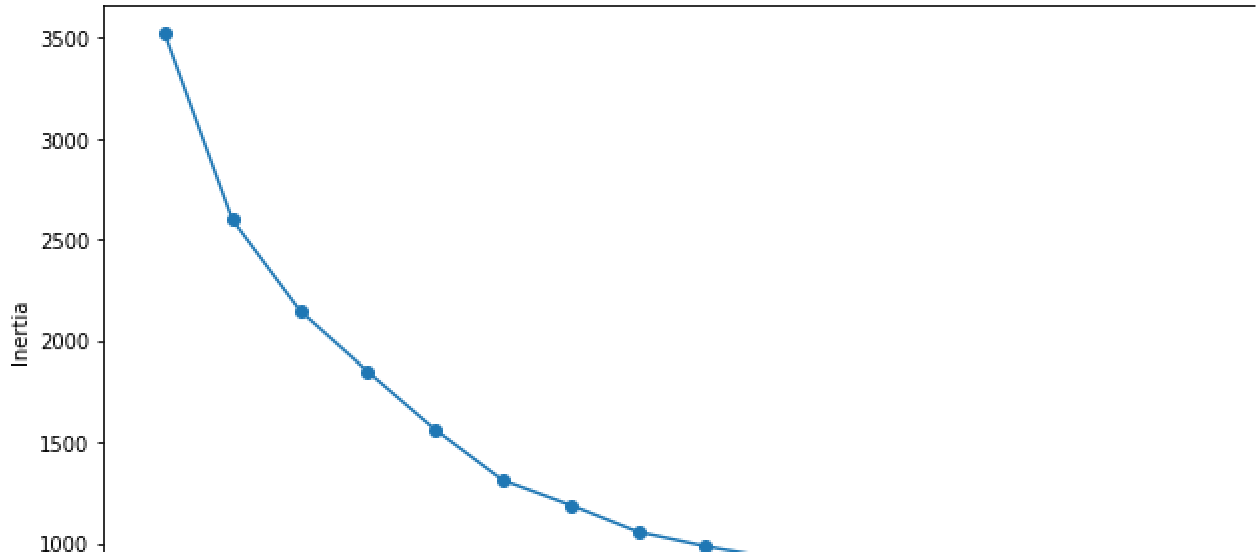
We will first fit multiple k-means models and in each successive model, we will increase the number of clusters. We will store the inertia value of each model and then plot it to visualize the result:

The **Elbow** Method is one of the most popular methods to determine this optimal value of k.

```
# fitting multiple k-means algorithms and storing the values in an empty list
SSE = []
for cluster in range(1,20):
    kmeans = KMeans( 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')
```

```
Text(0, 0.5, 'Inertia')
```



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:

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

```
pred
```

```
array([4, 4, 4, 1, 4, 4, 4, 4, 1, 4, 4, 4, 4, 4, 4, 1, 4, 1, 4, 1, 4, 1,
       1, 0, 4, 4, 1, 1, 4, 1, 1, 1, 1, 1, 1, 4, 1, 4, 4, 1, 1, 1, 4, 4,
       4, 4, 4, 2, 4, 4, 1, 1, 4, 4, 1, 1, 2, 4, 1, 1, 4, 2, 4, 4, 1, 2,
       1, 4, 1, 1, 1, 0, 1, 4, 4, 1, 1, 4, 1, 1, 1, 4, 4, 1, 4, 2, 2, 0,
       1, 1, 1, 1, 2, 1, 4, 1, 4, 1, 1, 1, 4, 4, 4, 1, 1, 1, 4, 4, 4, 4,
       1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 4, 4, 1, 4, 4, 4, 1, 1, 4, 4, 4, 4, 1, 1, 1, 4, 4, 1, 4, 1, 4,
       1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 4, 4, 1, 1, 1, 4, 1, 1, 3, 4,
       3, 3, 4, 4, 3, 3, 3, 4, 3, 3, 3, 4, 3, 2, 3, 3, 4, 3, 4, 3, 4, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 3, 3, 4, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       4, 3, 4, 3, 4, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 4, 1, 4, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 4, 1, 4, 1, 4, 4, 1, 4, 4, 4, 4, 4, 4, 4, 1,
       1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
       1, 4, 1, 2, 1, 4, 1, 1, 1, 1, 4, 4, 1, 4, 1, 1, 4, 4, 1, 4, 1, 4,
       1, 4, 1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1, 1, 4, 1, 1, 4,
       1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 4, 1, 1, 1, 1, 1, 1, 4, 4, 1,
       4, 1, 1, 4, 1, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1],
      dtype=int32)
```

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

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

```
1    235
4    126
3     63
2     10
0      6
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.