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	Frozen	De
count	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	
std	0.468052	0.774272	12647.328865	7380.377175	9503.162829	4854.673333	
min	1.000000	1.000000	3.000000	55.000000	3.000000	25.000000	
25%	1.000000	2.000000	3127.750000	1533.000000	2153.000000	742.250000	
50%	1.000000	3.000000	8504.000000	3627.000000	4755.500000	1526.000000	
75%	2.000000	3.000000	16933.750000	7190.250000	10655.750000	3554.250000	
max	2.000000	3.000000	112151.000000	73498.000000	92780.000000	60869.000000	

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:

```
# 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	5
count	4.400000e+02	4.400000e+02	4.400000e+02	440.000000	4.400000e+02	4.400000e+02
mean	1.614870e-17	3.552714e-16	-3.431598e-17	0.000000	-4.037175e-17	3.633457e-17
std	1.001138e+00	1.001138e+00	1.001138e+00	1.001138	1.001138e+00	1.001138e+00
min	-6.902971e-01	-1.995342e+00	-9.496831e-01	-0.778795	-8.373344e-01	-6.283430e-01
25%	-6.902971e-01	-7.023369e-01	-7.023339e-01	-0.578306	-6.108364e-01	-4.804306e-01
50%	-6.902971e-01	5.906683e-01	-2.767602e-01	-0.294258	-3.366684e-01	-3.188045e-01

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
```

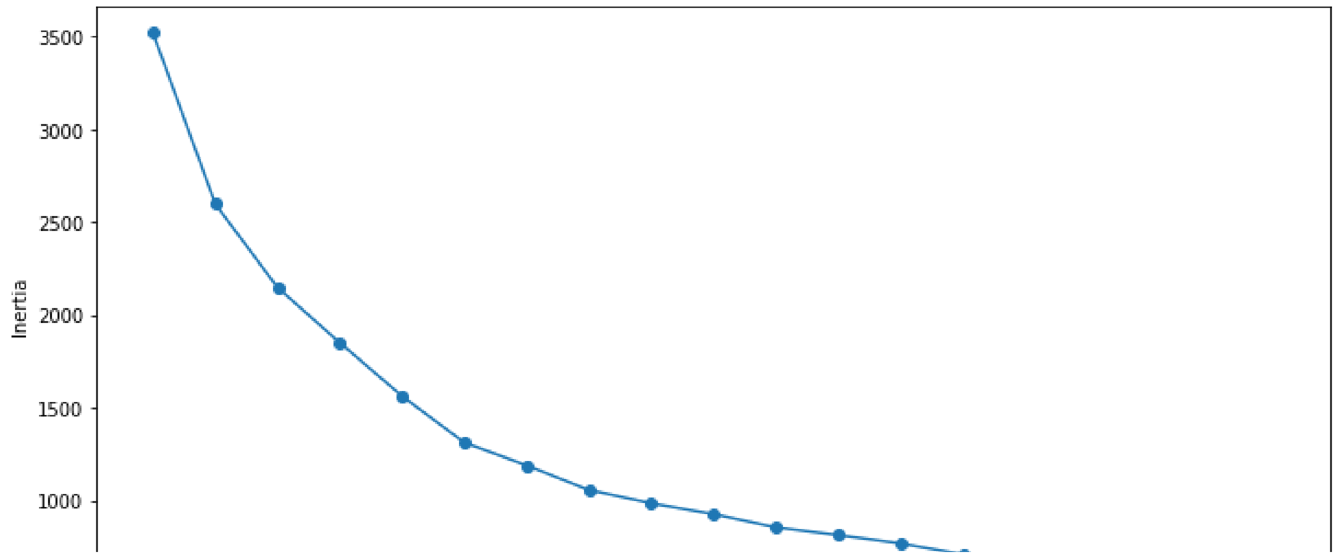
We got an inertia value of almost 2600. Now, let’s see how we can use the elbow curve to determine the optimum number of clusters in Python.

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.

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

```
lst = [2,3,4,10,11,12,20,25,30]
df = pd.DataFrame(lst)
print(df)
```

```
        0
    0   2
    1   3
    2   4
    3  10
    4  11
    5  12
    6  20
    7  25
    8  30
```

```
df.head()
```

```
        0
```

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

```
               0
count   9.000000
mean   13.000000
std     9.987492
min     2.000000
25%     4.000000
50%    11.000000
75%    20.000000
max    30.000000
```

```
# 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
count   9.000000
mean    0.000000
std     1.060660
min    -1.168187
25%    -0.955790
50%    -0.212398
75%     0.743392
max     1.805380
```

```
# 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)
```

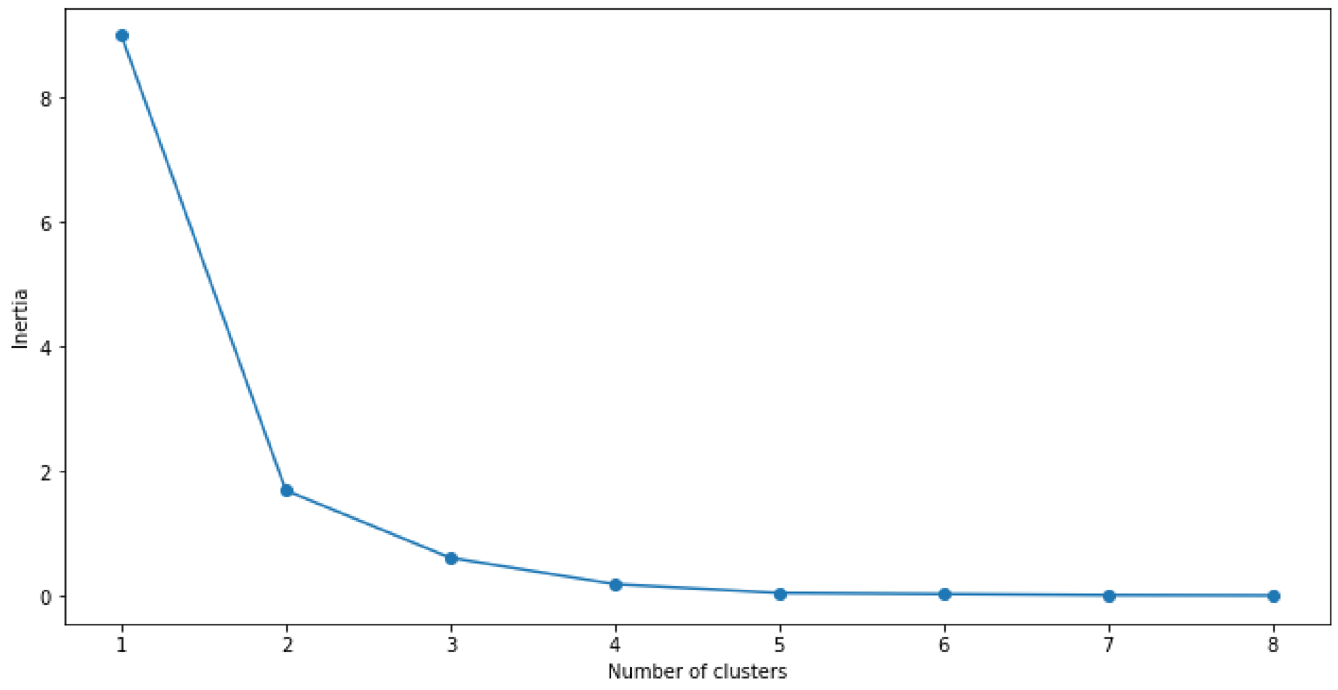
```
KMeans(n_clusters=2)
```

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

```
1.6917293233082706
```

```
# fitting multiple k-means algorithms and storing the values in an empty list
SSE = []
for cluster in range(1,9):
    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(1,9), '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')
```



```
# k means using 5 clusters and k-means++ initialization
kmeans = KMeans( n_clusters = 2, init='k-means++')
kmeans.fit(data_scaled)
pred = kmeans.predict(data_scaled)
```

```
pred

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

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

     1    6
     0    3
     Name: cluster, dtype: int64
```

Double-click (or enter) to edit

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

     Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mour
```

```
blobs = pd.read_csv("/content/drive/My Drive/Colab Notebooks/kmeans_blobs.csv")
```

The data set contains 4 columns with the following information:

1. ID: A unique identifier for the observation
2. x: Attribute corresponding to an x coordinate
3. y: Attribute corresponding to a y coordinate
4. Cluster: An identifier for the cluster the observation belongs to

[ ] ↳ *7 cells hidden*

Step 3 - Calculate distance

We now need to calculate the distance between each of the centroids and the data points. We will assign the data point to the centroid that gives us the minimum error.

Let us create a function to calculate the root of square errors:

```
def rsserr(a,b):
    '''
    Calculate the root of sum of squared errors.
    a and b are numpy arrays
    '''
    return np.square(np.sum((a-b)**2))
```

Let us pick a data point and calculate the error so we can see how this works in practice. We will use point , which is in fact one of the centroids we picked above. As such, we expect that the error for that point and the third centroid is zero.

We therefore would assign that data point to the second centroid. Let’s take a look:

```
for i, centroid in enumerate(range(centroids.shape[0])):
    err = rsserr(centroids.iloc[centroid,:], df.iloc[36,:])
    print('Error for centroid {0}: {1:.2f}'.format(i, err))
```

```
Error for centroid 0: 384.22
Error for centroid 1: 724.64
Error for centroid 2: 0.00
```

Step 4 - Assign centroids

We can use the idea from Step 3 to create a function that helps us assign the data points to corresponding centroids.

We will calculate all the errors associated to each centroid, and then pick the one with the lowest value for assignation:

```
def centroid_assignation(dset, centroids):
    '''
    Given a dataframe `dset` and a set of `centroids`, we assign each
    data point in `dset` to a centroid.
    - dset - pandas dataframe with observations
    - centroids - pa das dataframe with centroids
    '''
    k = centroids.shape[0]
    n = dset.shape[0]
    assignation = []
    assign_errors = []

    for obs in range(n):
        # Estimate error
        all_errors = np.array([])
        for centroid in range(k):
            err = rsserr(centroids.iloc[centroid, :], dset.iloc[obs,:])
            all_errors = np.append(all_errors, err)

        # Get the nearest centroid and the error
        nearest_centroid =  np.where(all_errors==np.amin(all_errors))[0].tolist()[0]
        nearest_centroid_error = np.amin(all_errors)

        # Add values to corresponding lists
        assignation.append(nearest_centroid)
        assign_errors.append(nearest_centroid_error)

    return assignation, assign_errors
```

Let us add some columns to our data containing the centroid assignations and the error incurred.

Furthermore, we can use this to update our scatter plot showing the centroids (denoted with squares) and we colour the observations according to the centroid they have been assigned to:

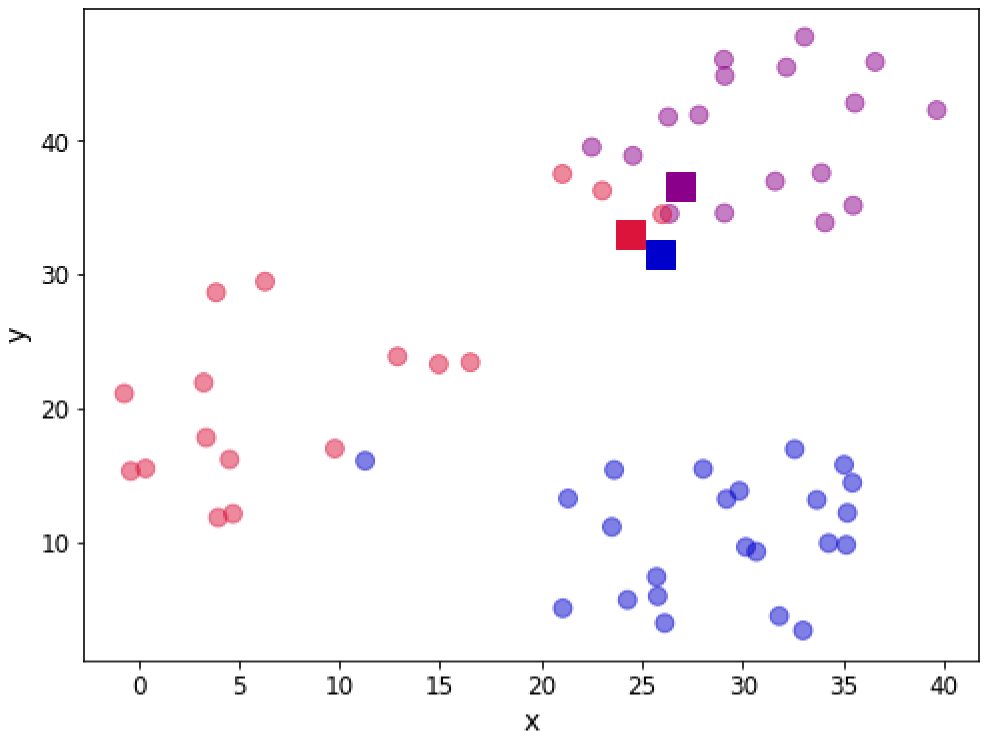
```
df['centroid'], df['error'] = centroid_assignation(df, centroids)
df.head()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
  """Entry point for launching an IPython kernel.
```

	x	y	centroid	error
0	24.412	32.932	0	0.000000
1	35.190	12.189	1	211534.211314
2	26.288	41.718	2	699.601495
3	0.376	15.506	0	776856.744109
4	26.116	3.963	1	576327.599678

```
fig, ax = plt.subplots(figsize=(8, 6))
plt.scatter(df.iloc[:,0], df.iloc[:,1],  marker = 'o',
            c=df['centroid'].astype('category'),
            cmap = customcmap, s=80, alpha=0.5)
plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],
            marker = 's', s=200, c=[0, 1, 2],
            cmap = customcmap)
ax.set_xlabel(r'x', fontsize=14)
ax.set_ylabel(r'y', fontsize=14)
```

```
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



Let us see the total error by adding all the contributions.

We will take a look at this error as a measure of convergence. In other words, if the error does not change, we can assume that the centroids have stabilised their location and we can terminate our iterations.

In practice, we need to be mindful of having found a local minimum (outside the scope of this post).

```
print("The total error is {0:.2f}".format(df['error'].sum()))
```

```
    The total error is 11927659.01
```

Step 5 - Update centroid location

Now that we have a first attempt at defining our clusters, we need to update the position of the k centroids.

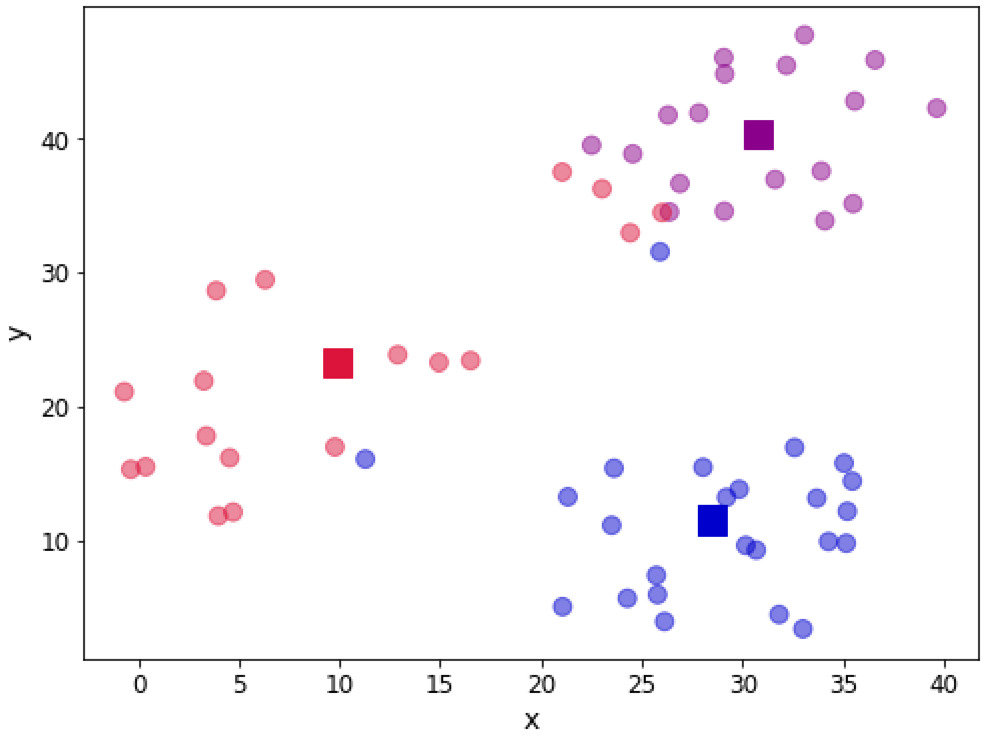
We do this by calculating the mean of the position of the observations assigned to each centroid. Let take a look:

```
centroids = df.groupby('centroid').agg('mean').loc[:, colnames].reset_index(drop = True)
centroids
```

	x	y
0	9.889444	23.242611
1	28.435750	11.546250
2	30.759333	40.311167

We can verify that the position has been updated. Let us look again at our scatter plot:

```
fig, ax = plt.subplots(figsize=(8, 6))
plt.scatter(df.iloc[:,0], df.iloc[:,1],  marker = 'o',
            c=df['centroid'].astype('category'),
            cmap = customcmap, s=80, alpha=0.5)
plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],
            marker = 's', s=200,
            c=[0, 1, 2], cmap = customcmap)
ax.set_xlabel(r'x', fontsize=14)
ax.set_ylabel(r'y', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



Step 6 - Repeat steps 3-5

Now we go back to calculate the distance to each centroid, assign observations and update the centroid location. This calls for a function to encapsulate the loop:

```
def kmeans(dset, k=2, tol=1e-4):
    '''
    K-means implementationd for a
    `dset`:  DataFrame with observations
    `k`: number of clusters, default k=2
    `tol`: tolerance=1E-4
    '''
    # Let us work in a copy, so we don't mess the original
    working_dset = dset.copy()
    # We define some variables to hold the error, the
    # stopping signal and a counter for the iterations
    err = []
    goahead = True
    j = 0

    # Step 2: Initiate clusters by defining centroids
    centroids = initiate_centroids(k, dset)

    while(goahead):
        # Step 3 and 4 - Assign centroids and calculate error
        working_dset['centroid'], j_err = centroid_assignation(working_dset, centroids)
        err.append(sum(j_err))

        # Step 5 - Update centroid position
        centroids = working_dset.groupby('centroid').agg('mean').reset_index(drop = True)

        # Step 6 - Restart the iteration
        if j>0:
            # Is the error less than a tolerance (1E-4)
            if err[j-1]-err[j]<=tol:
                goahead = False
        j+=1

    working_dset['centroid'], j_err = centroid_assignation(working_dset, centroids)
    centroids = working_dset.groupby('centroid').agg('mean').reset_index(drop = True)
    return working_dset['centroid'], j_err, centroids
```

OK, we are now ready to apply our function. We will clean our dataset first and let the algorithm run:

```
np.random.seed(42)
df['centroid'], df['error'], centroids =  kmeans(df[['x','y']], 3)
df.head()
```

	x	y	centroid	error
0	24.412	32.932	2	3767.568743
1	35.190	12.189	1	1399.889001
2	26.288	41.718	2	262.961097
3	0.376	15.506	0	2683.086425
4	26.116	3.963	1	2723.650198

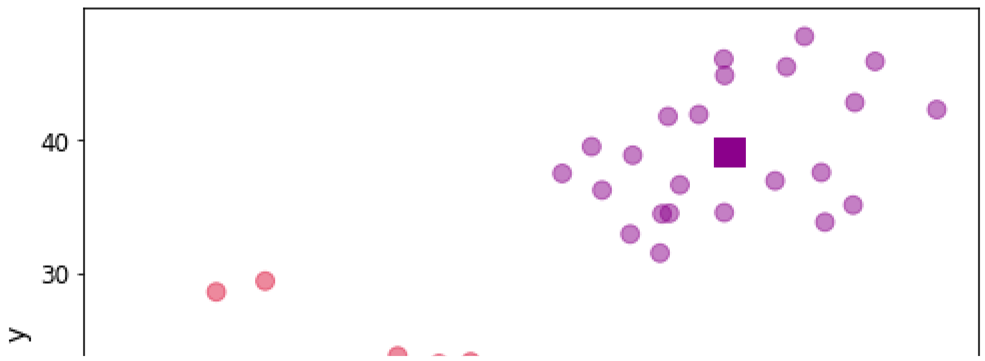
Let us see the location of the final centroids:

```
centroids
```

	x	y
0	6.322867	19.559800
1	29.330864	10.432409
2	29.304957	39.050783

And in a graphical way, let us see our clusters:

```
fig, ax = plt.subplots(figsize=(8, 6))
plt.scatter(df.iloc[:,0], df.iloc[:,1],  marker = 'o',
            c=df['centroid'].astype('category'),
            cmap = customcmap, s=80, alpha=0.5)
plt.scatter(centroids.iloc[:,0], centroids.iloc[:,1],
            marker = 's', s=200, c=[0, 1, 2],
            cmap = customcmap)
ax.set_xlabel(r'x', fontsize=14)
ax.set_ylabel(r'y', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```



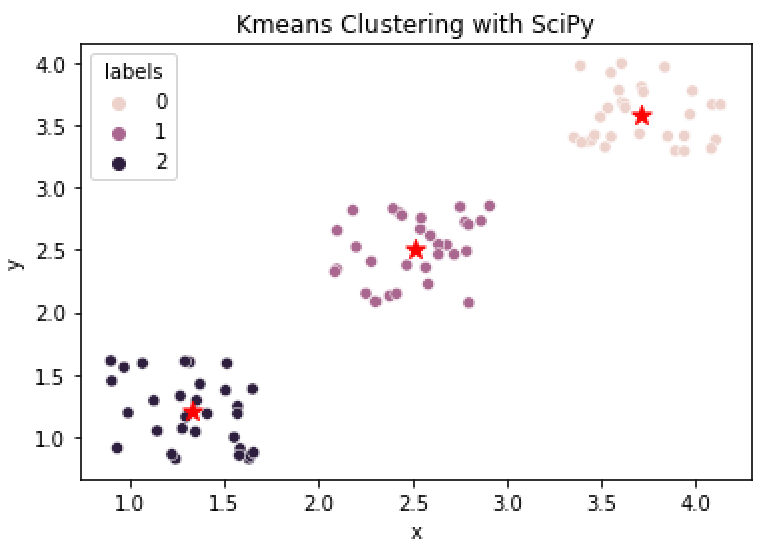
As we can see the three groups have been obtained. In this particular example the data is such that the distinction between the groups is clear. However, we may not be as lucky in every case. So the question about how many groups there are still remains. We can use a screen plot to help us with the error minimisation by looking at running the algorithm with a **sequence k=1,2,3,4** and look for the "elbow" in the plot indicating a good number of clusters to use:

```
err_total = []
n = 10

df_elbow = blobs[['x','y']]

for i in range(n):
    _, my_errs, _ = kmeans(df_elbow, i+1)
    err_total.append(sum(my_errs))
fig, ax = plt.subplots(figsize=(8, 6))
plt.plot(range(1,n+1), err_total, linewidth=3, marker='o')
ax.set_xlabel(r'Number of clusters', fontsize=14)
ax.set_ylabel(r'Total error', fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```

We can now apply the “elbow rule” which is a heuristic to help us determine the number of clusters.

If we think of the line shown above as depicting an arm, then the “elbow” is the point of inflection. In this case the “elbow” is located between 2 and 4 clusters, giving us an indication that choosing 3 is a good fit.

```
import matplotlib.pyplot as plt
from numpy import vstack,array
from numpy.random import rand
import pandas as pd
import seaborn as sns
from scipy.cluster.vq import kmeans,vq,whiten
# Generate data
data = vstack((
    (rand(30,2)+1),(rand(30,2)+2.5),(rand(30,2)+4)
    ))
# standardize the features
data = whiten(data)
# Run KMeans  with 3 clusters
centroids, _ = kmeans(data, 3)
# Predict labels
labels, _ = vq(
    data, centroids
)
# Create DataFrame
df = pd.DataFrame(data, columns=['x','y'])
df['labels'] = labels
# Plot Clusters
sns.scatterplot(
    x='x',
    y='y',
    hue='labels',
    data=df
)
# Plot cluster centers
cs_x = centroids[:,0]
cs_y = centroids[:,1]
plt.scatter(cs_x, cs_y, marker='*', s=100, c='r')
plt.title('Kmeans Clustering with SciPy')
plt.show()
```


Kmeans Clustering with SciPy
labels
0
1
2
y
x
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0