Clustering

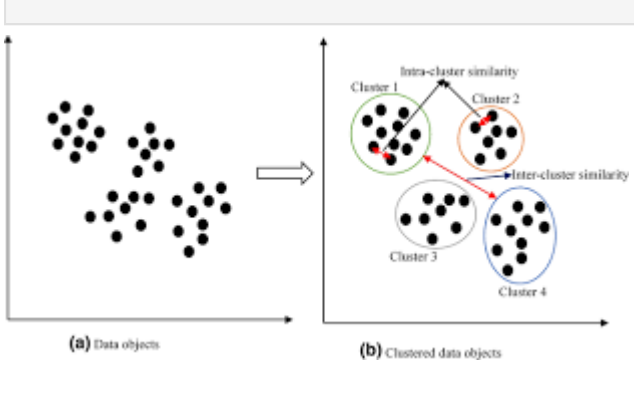
In [7]:
```python
from IPython.display import Image
```

What is Clustering?

Clustering is a type of unsupervised machine learning, which means the actual "y" (target variable) is not given when traning the model. By seperating the datapoints based on their structures or similarities, we're able to create several groups, called **cluster**.

In [8]:
```python
Image("images/cluster_.png")
```

Out[8]:



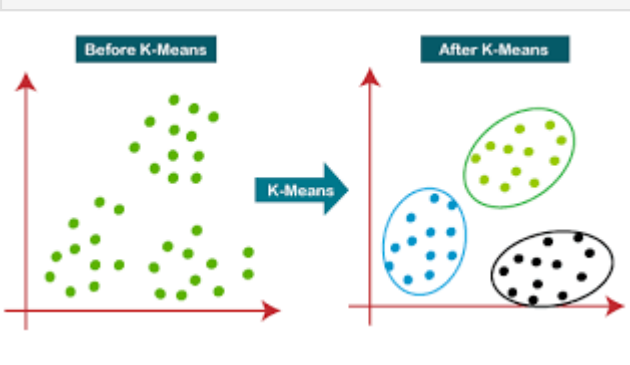
How to implement Clustering in python?

To impletment Clustering, we actually have developed a lot of methods/algorithms. For this project, we will use k-means to achieve our goal. The basic idea is below:

- Step1: Select k unique points from datapoints as initial centroids (k is the number of clusters we want to generate)
- Step2: Put each datapoints into the k clusters based on the minimum distance to each centroids
- Step3: Calculate the mean of each cluster
- Step4: Repeat Step2 and Step3 so that the mean of each cluster = previously chosen centroids

In [9]:
```python
Image("images/cluster.png")
```

Out[9]:



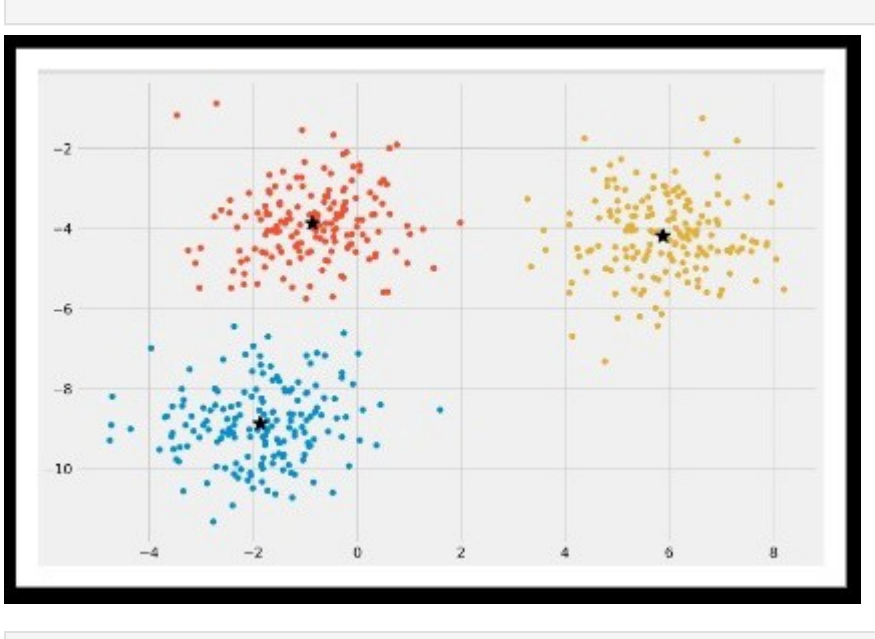
Apply "kmeans++" initialization

When we only use k-means function, we initialize the centroids randomly, which may not generate accurate resulsts when initialization is bad. Thus, in order to improve our model, we can work on a funciton called "select_centroids". This function functions as "kmeans++" in scikit-learn. The basic algorightm is below:

- Step1: Randomly select the first centroid from the data points.
- Step2: For each data point compute its distance from the nearest, previously chosen centroid.
- Step3: Select the next centroid from the data points such that the probability of choosing a point as centroid is directly proportional to its distance from the nearest, previously chosen centroid. (i.e. the point having maximum distance from the nearest centroid is most likely to be selected next as a centroid)
- Step4: Repeat steps 2 and 3 until k centroids have been sampled

In [11]:
```python
Image("images/++.jpeg")
```

Out[11]:



In [187…
```python
%run kmeans
```

Try Our k-means on One-Dimensional Data

Suppose we have several grades, and we want to seperate them into several clusters. We want to seperate theses grades into three groups so that we can have a better understanding of the grades distribution.

In [261…
```python
grades = [92.65, 93.87, 74.06, 86.94,92.26, 94.46, 92.94, 80.65, 92.86, 85.94, 91.79, 95.23, 85.37, 87.85, 87.7
k = 3
grades = np.array(grades).reshape(-1,1)
centroids = select_centroids(grades,3)
centroids,labels = kmeans(grades,k)
print("centroids:", centroids.reshape(1,-1))
print("labels for each x: ", labels)
for j in range(k):
    print("vector assignments: ",grades[labels==j].reshape(1,-1))
```

```
centroids: [[93.23222222 86.762      77.355     ]]
labels for each x:  [0 0 2 1 0 0 0 2 0 1 0 0 1 1 1 0]
vector assignments:  [[92.65 93.87 92.26 94.46 92.94 92.86 91.79 95.23 93.03]]
vector assignments:  [[86.94 85.94 85.37 87.85 87.71]]
vector assignments:  [[74.06 80.65]]
```

In [19]:
```python
Image("images/one_dim.png")
```

Out[19]:

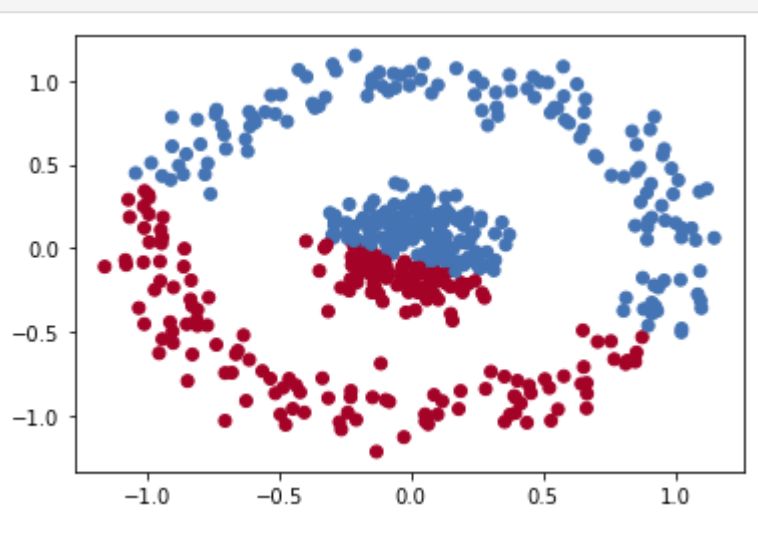
From the graph above, we can see that the "grades" are seperated into three clusters. In a practical perspective, we can use k-means to divide up the grades into ideal clusters.

Try Our k-means on Two-Dimensional Data

Suppose we have a two-dimentional dataset and we want to seperate them into two clusters.

In [329…
```python
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt

X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)
centroids = select_centroids(X,2)
#print(centroids)
centroids, labels = kmeans(X, 2, centroids=centroids)
colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```



From the graph shown above, kmeans performs poorly on disjoint and nested structures because we seperate each point based on their distance. It's only good at seperating datapoints that are clustered togerther. If the datapoints cluster in a more complex pattern, k-means cannot generate good results on that.

Try Our k-means on Multi-Dimensional Data

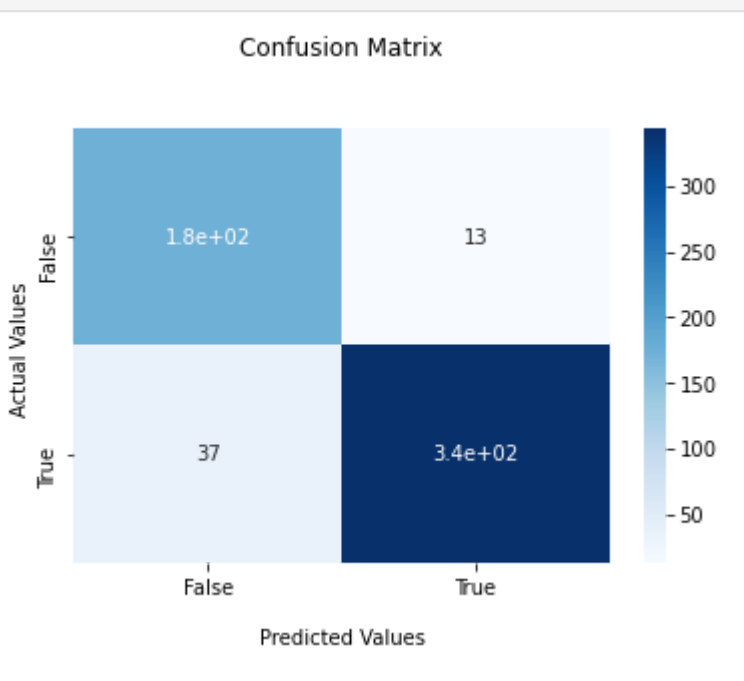
In [327…
```python
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import seaborn as sns
cancer = load_breast_cancer()
X = cancer.data
y = cancer.target #y=0 means cancer, y=1 benigh
sc = StandardScaler()
X = sc.fit_transform(X)

centroids = select_centroids(X, k=2)
centroids, labels = kmeans(X, k=2, centroids = centroids, tolerance= 0.01)

if len(X[labels == 1]) < len(X[labels == 0]):
    labels = [1-i for i in labels]
```

In [14]:
```python
Image("images/confusion_matrix.png")
```

Out[14]:



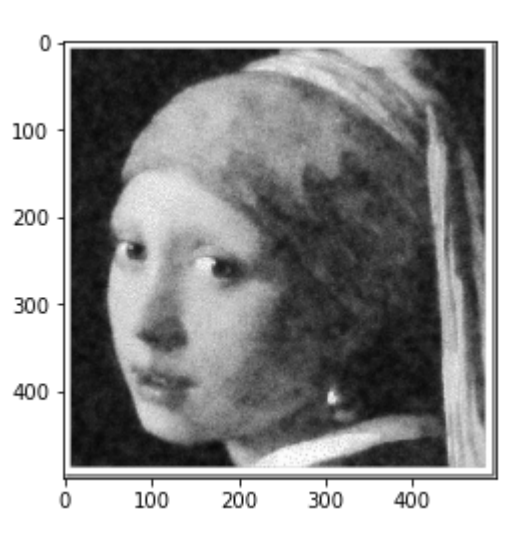
From the confusion matrix above, we can see that k-means is good at clustering binary problem.

Try Our k-means on Image Processing

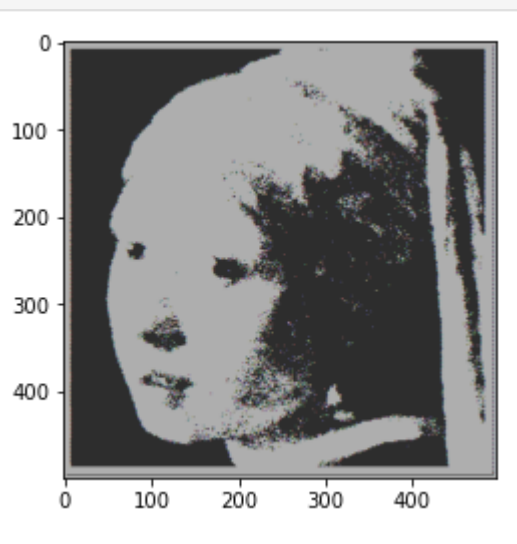
Below is a very famous picture - Girl with a Pearl Earring by Dutch Golden Age painter Johannes Vermeer. K-means can generate some interesting artificial effect of the picture by clustering the color of the image. Rather than use millions of colors, we can usually get away with 256 or even 64 colors by setting different k in our k-means function. Let's start with grey_scaled picture first.

In [297…
```python
import cv2
import warnings
warnings.filterwarnings("ignore")
image = cv2.imread("pearl_grey.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.imshow(image)
```

Out[297… `<matplotlib.image.AxesImage at 0x7f95587a7970>`



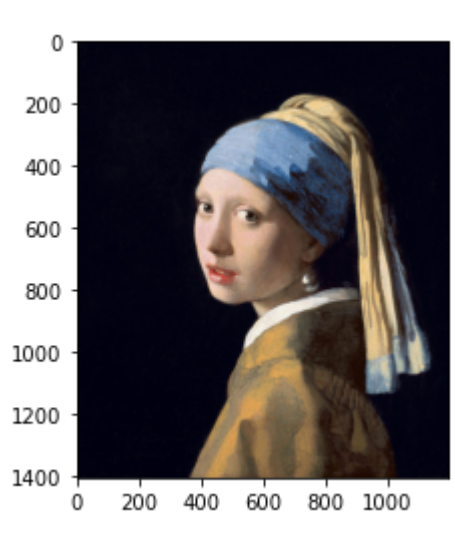
In [298…
```python
pixel_values = image.reshape((-1, 4))
pixel_values = np.float32(pixel_values)
k=4
centroids = select_centroids(pixel_values, k=k)
centroids, labels = kmeans(pixel_values, k=k, centroids=centroids, tolerance=.01)
centroids = centroids.astype(np.uint8)
segmented_image = centroids[labels.flatten()]
segmented_image = segmented_image.reshape(image.shape)
plt.imshow(segmented_image)
plt.show()
```



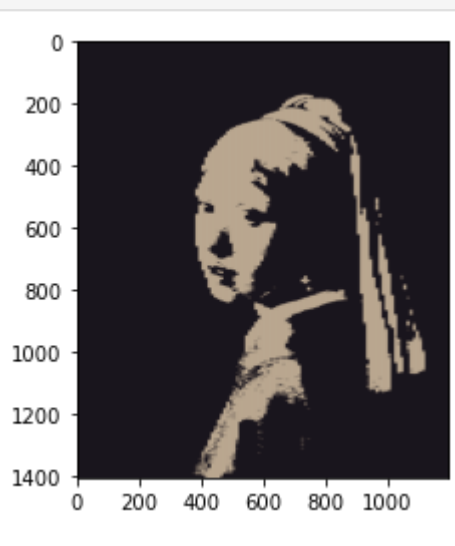
From the graph shown above, we can see the picture got less color, since the original pixels are replaced by the centroids. Original numbers of different colors are reduced to k, which is 4. That results in the change on the picture above.

In [299…
```python
import cv2
image = cv2.imread("pearl.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.imshow(image)
```

Out[299… `<matplotlib.image.AxesImage at 0x7f954c52af40>`



In [302…
```python
pixel_values = image.reshape((-1, 30))
pixel_values = np.float32(pixel_values)
k=30
centroids = select_centroids(pixel_values, k=k)
centroids, labels = kmeans(pixel_values, k=k, centroids=centroids, tolerance=.01)
centroids = centroids.astype(np.uint8)
segmented_image = centroids[labels.flatten()]
segmented_image = segmented_image.reshape(image.shape)
plt.imshow(segmented_image)
plt.show()
```



From the two pictures above, we can also implemenet k-means on colored pictures

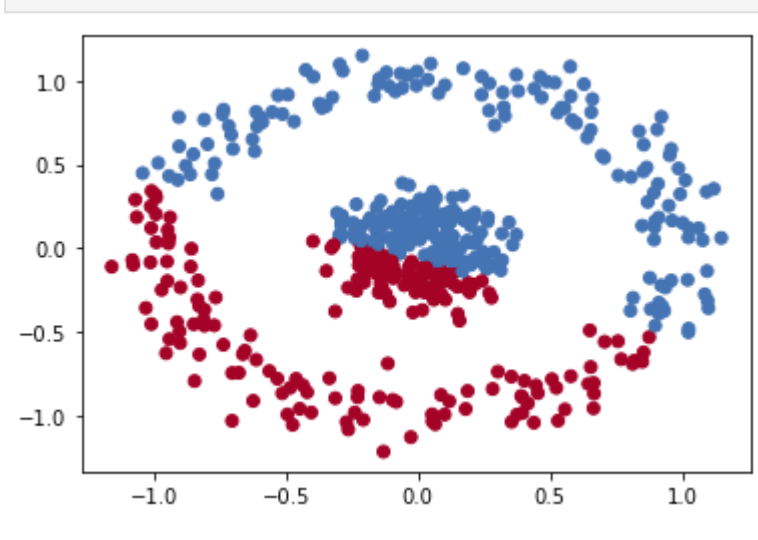
Spectral clustering

As we mentioned above, kmeans performs poorly on disjoint and nested structures because we seperate each point based on their distance. You may wonder, how can we fix it? There's a model in sklearn called **SpectralClustering** , where clusters are grouped by based on the edges connecting them rather than their distances. The basic steps are shown below:

- Take our graph and built an adjacency matrix
- Create the Graph Laplacian by subtracting the adjacency matrix from the degree matrix and calculate the eigenvalues of the Laplacian. The vectors associated with those eigenvalues contain information on how to segment the nodes
- Performe K-Means on those vectors in order to get the labels for the nodes

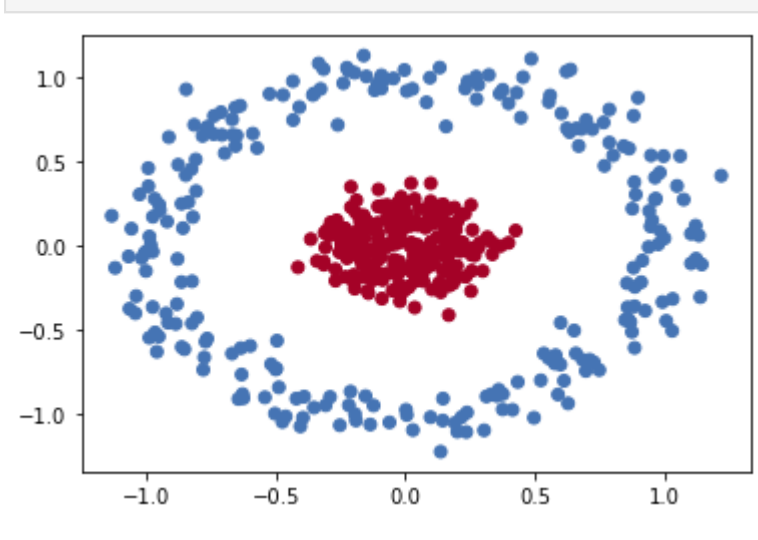
In [15]:
```python
Image("images/pic.png")
```

Out[15]:



In [322…
```python
from sklearn.cluster import SpectralClustering
X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)
cluster = SpectralClustering(n_clusters=2, affinity="nearest_neighbors")
labels = cluster.fit_predict(X)  # pass X not similarity matrix

#print(labels)
colors=np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```



Application

Clustering is a widely-applied technique in the data science industry. Here are some examples:

- **Recommendation engines**: In this method, the clustering method provided an idea of like-minded users. The computation/estimation as data provided by several users is leveraged for improving the performance of collaborative filtering methods. And this can be implemented for rendering recommendations in diverse applications.

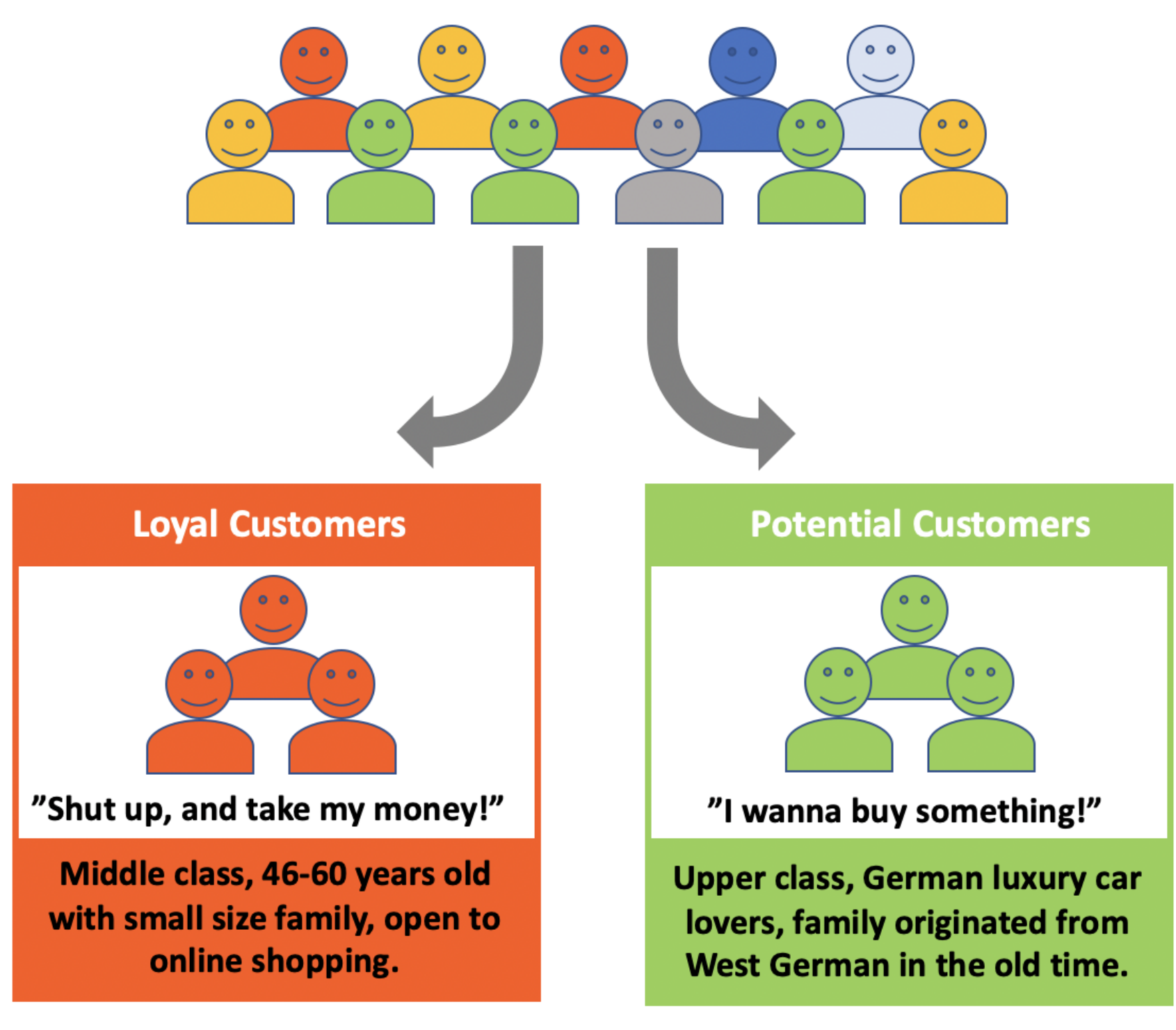
In [16]:
```python
Image("images/rec.jpeg")
```

Out[16]:

- **Market and Customer segmentation**:Clustering methods enable us to segment customers into diverse clusters, depending on which companies can consider novel strategies to apply to their customer base.

In [17]:
```python
Image("images/m.png")
```

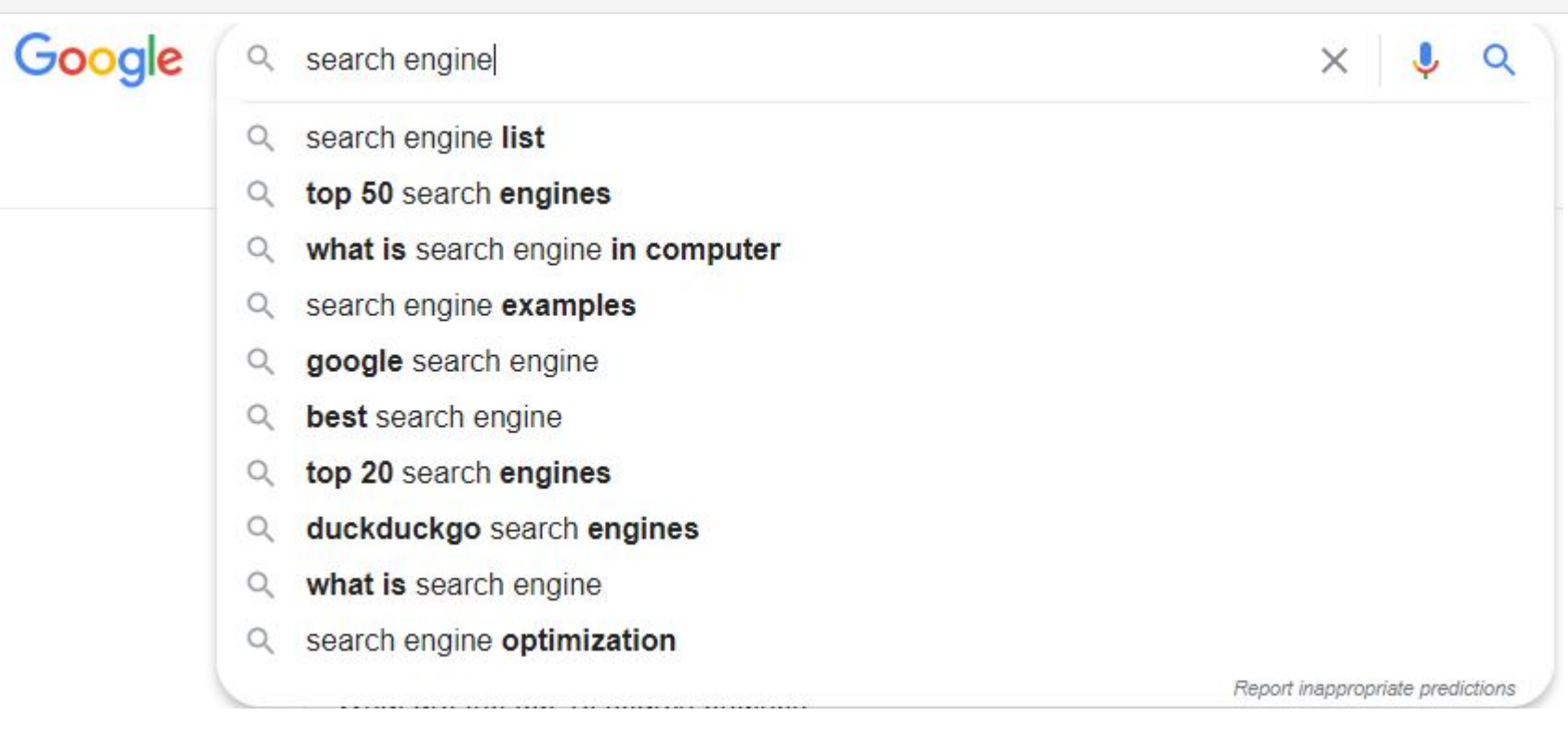
Out[17]:



- **Search Result Clustering**: Depending on the closest similar objects/properties, the data is assigned to a single cluster, giving the plethora sets of similar results of the users. In simple terms, the search engine attempts to group identical objects in one cluster and non-identical objects in another cluster.

In [18]:
```python
Image("images/search.jpg")
```

Out[18]:



Conclusion

Clustering is a very important method in Machine Learning. This report discussed one types of clustering - **k-means** and its initialization - **kmeans++**. K-means is good at group things that are continuous and visually seperated since it divides datapoints based on their distances. This report also mentions Spectral clustering, which group datapoints based on the graph structure.

References

- https://towardsdatascience.com/spectral-clustering-aba2640c0d5b
- https://github.com/parrt/msds689/blob/master/projects/kmeans/kmeans.md#spectral-clustering
- https://www.analyticssteps.com/blogs/5-clustering-methods-and-applications