K-means clustering

Kmeans clustering is one of the most popular clustering algorithms and usually the first thing practitioners apply when solving clustering tasks to get an idea of the structure of the dataset. Given a set of observations (x1, x2, ..., xn), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k (≤ n) sets.

Why do we cluster?

- Exploratory data analysis technique used to get an intuition about the structure of the data.
- Summarize data - Represent a large continuous vector with a cluster number.

Algorithm:

- Assume K clusters
- Assume points that are close to other points belong to the same group. Far points are outside of the group
- Randomly initialize cluster centers
- Calculate distance of points to centers
- **Assign** each point to closest cluster center
- **Update**, cluster centers: center of gravity of points in each class.
- Repeat!

Assignment step: Assign each observation to the cluster with the nearest mean: that with the least squared Euclidean distance.

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\},$$

Update step: Recalculate means (centroids) for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j$$

Output:

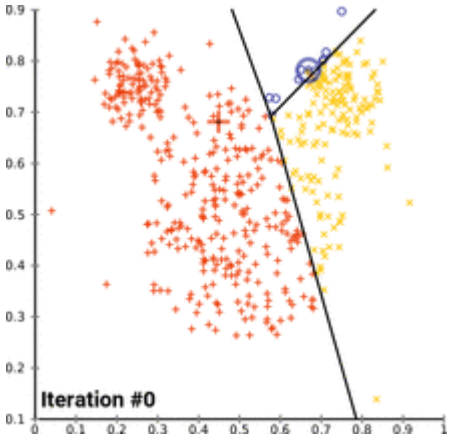
- centroids for each cluster
- cluster id for each point

Design Choices:

- Initialization
    - Randomly select K points as initial cluster centers
    - Number of hyperparrameter K is important

Challenges:

- Optimization
  - Converges to a local minimum
  - May want to perform multiple restarts (re-initialize and try again)

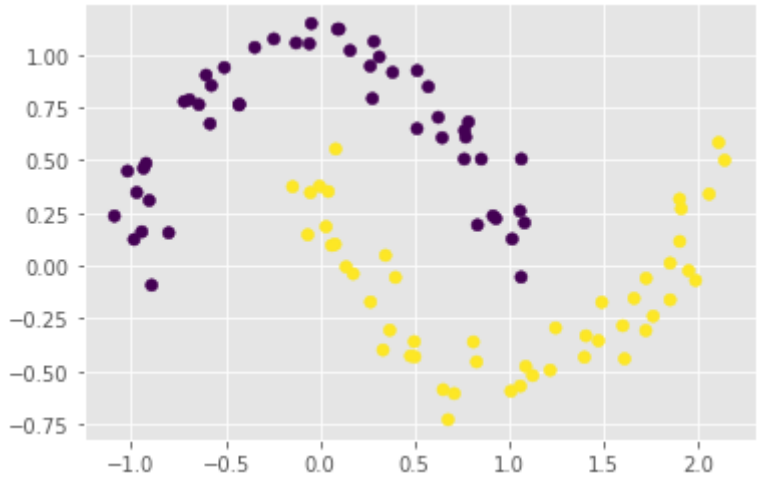



```
from numpy.linalg import norm

#Objecct Oriented Implementation of K-means
class Kmeans:

  #Set some parameters to initialize
  def __init__(self, n_clusters, max_iter=100, tolerance = 1e-3,debug=False,random_
    #nSpecify umber of centroids/clusters K
    self.n_clusters = n_clusters
    #Maximum iterations to run the algorithm
    self.max_iter = max_iter
    #Initialize random number generator
    self.random_state = random_state
    # Tolerance value to indicate that no improvement is made
    self.tol = tolerance
    # enable/disable debug messages
    self.debug = debug

  #Initialize the centroids
  def initialize_centroids(self, X):
    #Initialize centroids by first shuffling the dataset and then randomly
    #selecting K data points for the centroids without replacement
    #np.random.RandomState(self.random_state)
    random_idx = np.random.permutation(X.shape[0])
    #Randomly initialize K cluster centroids
    centroids = np.array(X[random_idx[:self.n_clusters]])
    return centroids

  #recompute a centroid based on the points x_i that belong to the same cluster.
  def compute_centroids(self, X, labels):
    centroids = np.zeros((self.n_clusters, X.shape[1]))
    for k in range(self.n_clusters):
      # average of points in the same cluster
      centroids[k, :] = np.mean(X[labels == k, :], axis=0)
    return centroids
```

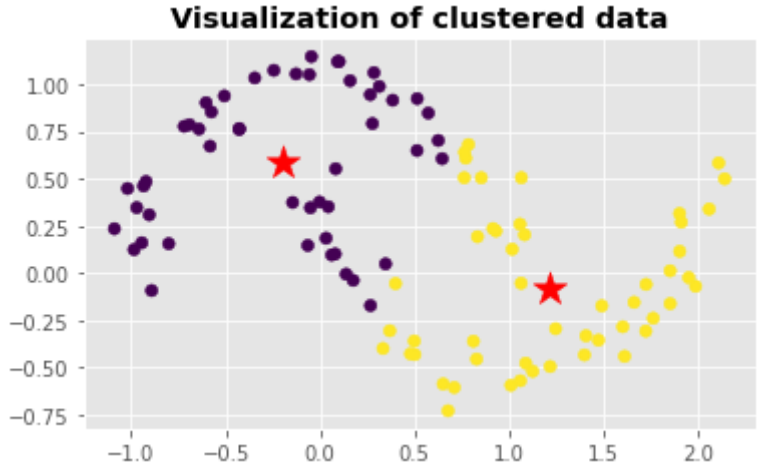
```
#find distance of each point x_i from all the centroids
def compute_distance(self, X, centroids):
  distance = np.zeros((X.shape[0], self.n_clusters))
  for k in range(self.n_clusters):
    row_norm = norm(X - centroids[k, :], axis=1)
    distance[:, k] = np.square(row_norm)
  return distance

#select the centroid with the smaller distance for each point
def find_closest_cluster(self, distance):
  return np.argmin(distance, axis=1)

def compute_sse(self, X, labels, centroids):
  distance = np.zeros(X.shape[0])
  for k in range(self.n_clusters):
    # Average (mean) of points assigned to cluster k
    distance[labels == k] = norm(X[labels == k] - centroids[k], axis=1)
  return np.sum(np.square(distance))

# Main steps of K-Means
def fit(self, X):
  self.centroids = self.initialize_centroids(X)
  if(self.debug):
    print('Initial centroids:')
    print(self.centroids)

  # track lowest error
  min_error = np.Inf

  for i in range(self.max_iter):
    # save previous centroids to compare how they change
    old_centroids = self.centroids

    #Compute the sum of the squared distance between data points and
    #all centroids.
    #find_distance of points x_i
    distance = self.compute_distance(X, old_centroids)
    # Assign each data point to the closest cluster (centroid).
    # find index of cluster centroid closest to x_i
    self.labels = self.find_closest_cluster(distance)
    # Compute the centroids for the clusters by taking the average of the
    # all data points that belong to each cluster.
    # average (mean) of points assigned to cluster k
    self.centroids = self.compute_centroids(X, self.labels)

    #Compute cost function (distortion)
    #distortion, is defined as the sum of the squared distances between each obse
    self.error = (1/X.shape[0])*self.compute_sse(X, self.labels, self.centroids)

    # keep only the best centroids
    if(self.error < min_error):
      min_error = self.error
      best_centroids = self.centroids
```

```
      if(self.debug):
        print(i,": ",self.error)

      # Keep iterating until there is no change to the centroids. i.e assignment
      # of data points to clusters isn't changing.
      # If no significant change to any of the clusters then terminate
      if self.compute_distance(self.centroids, old_centroids).all()< self.tol:
        # Pick clustering solution that gave the lowest distortion
        self.centroids = best_centroids
        break

  def predict(self, X):
    # find disances of x_i from centroids
    distance = self.compute_distance(X, self.centroids)
    #return the index of the closest centroid
    return self.find_closest_cluster(distance)


import matplotlib.pyplot as plt
import numpy as np
from matplotlib import style
import matplotlib.cm as cm

style.use('ggplot')

#1.
#X = np.array([[1, 2],
#              [1.5, 1.8],
#             [5, 8 ],
#             [8, 8],
#              [1, 0.6],
#              [9,11]])
#y=None

from sklearn.datasets import make_blobs,make_moons, load_iris
# generate 2d classification dataset

# 2.
#X, y = make_blobs(n_samples=100, centers=2, n_features=2)

#3.
X, y = make_moons(n_samples=100, noise=0.1)

#4.
#This data sets consists of 3 different types of irises' (Setosa, Versicolour, and
# iris = load_iris()
# X = iris.data[:, :2]  # we only take the first two features.
# y = iris.target

print('Number of samples: ',X.shape[0])
print('Number features: ',X.shape[1])
plt.scatter(X[:, 0], X[:, 1],c=y)
plt.show()
```

```
Number of samples:  100
Number features:  2
```




```
# Run local implementation of kmeans

model = Kmeans(n_clusters=2, max_iter=100,debug=False)
model.fit(X)

# Plot the clustered data
fig, ax = plt.subplots(figsize=(6, 6))
labnames=np.array([i+1 for i in range(model.n_clusters)]).T
plt.scatter(X[:, 0], X[:, 1],c=model.labels)
plt.scatter(model.centroids[:, 0], model.centroids[:, 1], marker='*', s=300, c='r',
plt.title('Visualization of clustered data', fontweight='bold')
ax.set_aspect('equal')
```



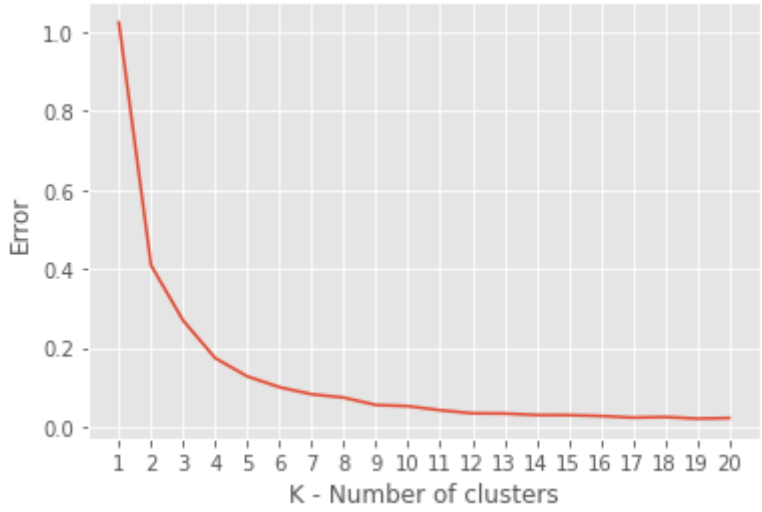
How to choose the number of clusters?

Elbow method: Stop adding clusters when improvement is small.

Sometimes it's still hard to figure out a good number of clusters to use because the curve is monotonically decreasing and may not show any elbow or has an obvious point where the curve starts flattening out.

```
#Finding the best number of clusters
X, y = make_blobs(n_samples=100, centers=3, n_features=2)
X, y = make_moons(n_samples=100, noise=0.1)

err = []
for k in range(1,21):
  me=np.Inf
  for j in range(10):
    model = Kmeans(n_clusters=k)
    model.fit(X)
    if(model.error<me):
      me = model.error
  err.append(me)
plt.plot(err)
plt.xticks(range(20),[r for r in range(1,21)])
plt.xlabel('K - Number of clusters')
plt.ylabel('Error')
print(np.argmin(err))
```

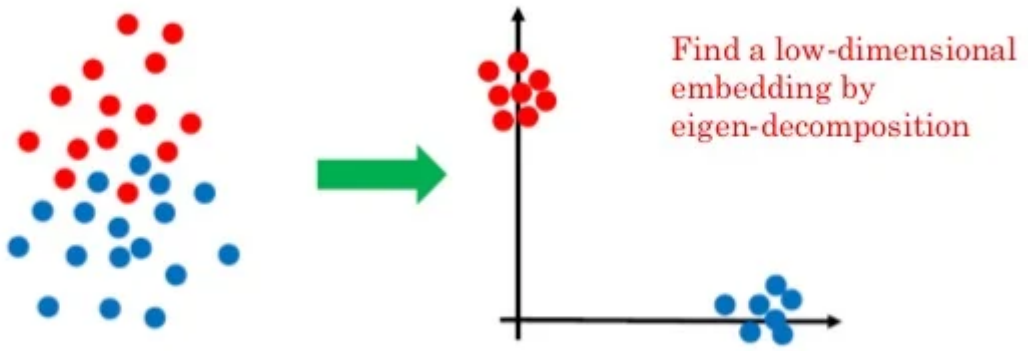
18



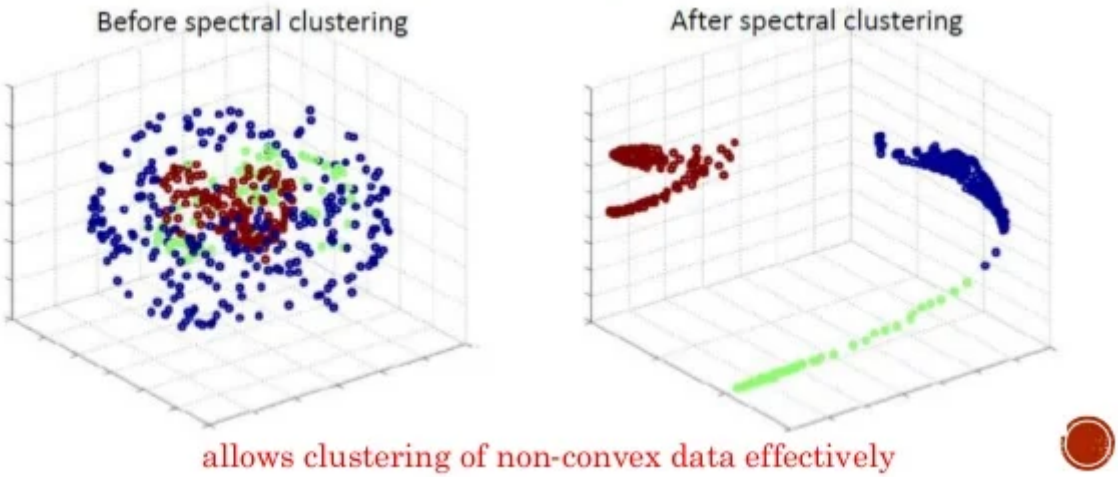
K-Means and Spectral Clustering

K-Means suffers as the geometric shapes of clusters deviates from spherical shapes. The idea is we transform to higher dimensional representation that make the data linearly separable (the same idea that we use in SVMs). Different kinds of algorithms work very well in such scenarios such as SpectralClustering.

Spectral clustering is a technique for clustering based on transformation by dimensionality reduction. In practice Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance when clusters are nested circles on the 2D plane.

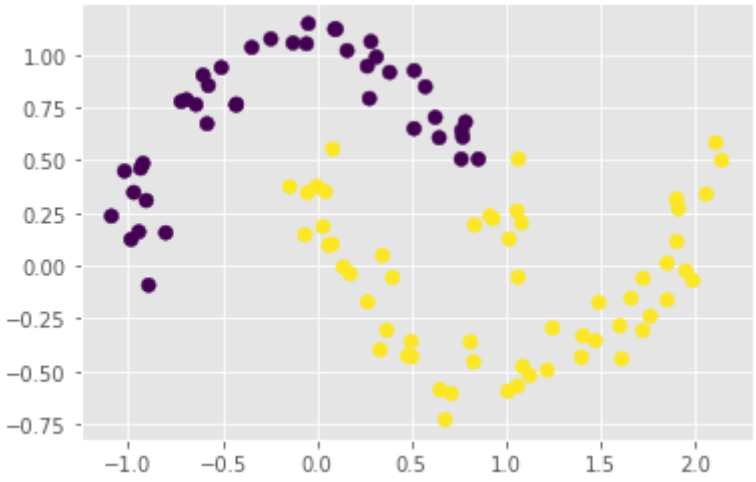


separates data while projecting in the low dimensional space



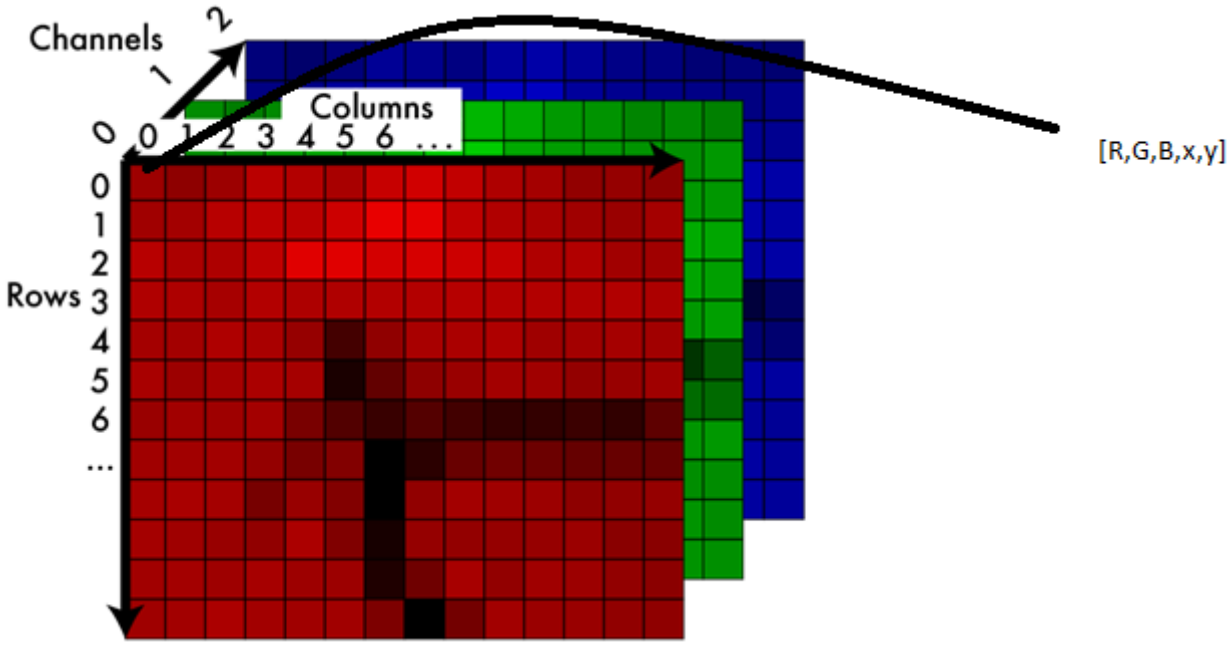
allows clustering of non-convex data effectively

```
from sklearn.cluster import SpectralClustering
model = SpectralClustering(n_clusters=2, affinity='nearest_neighbors',
                           assign_labels='kmeans')
labels = model.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```



K-Means for Image compression

In computer graphics, color quantization is the task of reducing the color palette of an image to a fixed number of colors k. A color is represented as a three dimensional vector [Red, Green, Blue] each ranging from 0...255. So we have a total of 16581375 (255^3) colors. Our goal is to reduce the number of colors to k and represent (compress) the photo using those k colors only. we'll use kmeans algorithm on the image and treat every pixel as a data point. That means reshape the image from height x width x channels to (height * width) x channel, i,e we would have 396 x 396 = 156,816 data points in 3-dimensional space which are the intensity of RGB. If we add also the x,y location we have (height * width) x (channel+2), i,e we would have 396 x 396 = 156,816 data points in 5-dimensional space

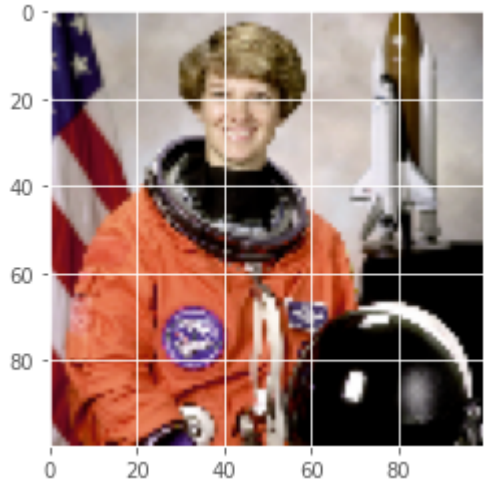


We do this with K-means by finding similar pixels and using the color corresponding to their centroid to represent them.

```
from skimage import data
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from sklearn import preprocessing
import cv2
data.astronaut
image = data.astronaut()
image = cv2.resize(image,(100,100))
plt.imshow(image)

print(image.shape)
imageH,imageW = image.shape[0:2]

#   Initialise data vector with attribute r,g,b,x,y for each pixel
dataVector = np.ndarray(shape=(imageW * imageH, 5), dtype=float)
#   Initialise vector that holds which cluster a pixel is currently in
pixelClusterAppartenance = np.ndarray(shape=(imageW * imageH), dtype=int)
```

```
#   Populate data vector with data from input image
#   dataVector has 5 fields: red, green, blue, x coord, y coord
for y in range(0, imageH):
      for x in range(0, imageW):
        xy = (x, y)
        rgb = image[y,x,:]
        dataVector[x + y * imageW, 0] = rgb[0]
        dataVector[x + y * imageW, 1] = rgb[1]
        dataVector[x + y * imageW, 2] = rgb[2]
        dataVector[x + y * imageW, 3] = x
        dataVector[x + y * imageW, 4] = y

#Features of different types may have different scales.
#For example, pixel coordinates on a 100×100 image vs. RGB color values in the rang
#Problem: Features with larger scales dominate clustering.
#Solution: Scale the features.

#   Standarize the values of our features
# RGB values and dimensions have different scales
dataVector_scaled = preprocessing.normalize(dataVector)

print(dataVector_scaled.shape)

model = Kmeans(n_clusters=10,debug=False)
model.fit(dataVector_scaled)
```

```
(100, 100, 3)
(10000, 5)
```




```
#Find the closest color for each datapoint
import matplotlib.cm as cm
colors = cm.rainbow(np.linspace(0, 1, len(model.centroids)))
for ndv,dV in zip(dataVector_scaled,dataVector):
  item = int(model.predict(ndv[None,:]))
  dV[0] = int(round(model.centroids[item][0] * 255))
  dV[1] = int(round(model.centroids[item][1] * 255))
  dV[2] = int(round(model.centroids[item][2] * 255))

# Replace all the pixels in the original image with the centroid color
for y in range(imageH):
```

```
    for x in range(imageW):
        image[y, x,:]=[int(dataVector[y * imageW + x][0]),
                       int(dataVector[y * imageW + x][1]),
                       int(dataVector[y * imageW + x][2])]

#display the image
plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7f6cadad45d0>
```