INF 552 Assignment 2

K-Means and Gaussian Mixture Model (GMM) Clustering

Zongdi Xu(USC ID 5900-5757-70, Part 1&2), Wenkai Xu(USC ID 5417-1457-73, Part 3&4)

Feb 17, 2019

Part 1: Implementation of K-Means

The K-Means algorithm consists two main steps: assigning step and summary step.

(1) Initialization

To begin with, read data records from input file and generate initial cluster centroids using random function.

```
def randomCenteroid(ClusterNum, f):
  x_min = min(f[:, 0])
  x_max = max(f[:, 0])
  y_min = min(f[:, 1])
  y_max = max(f[:, 1])
  p = rand(ClusterNum, 2)
  p = p * np.array([np.ones(ClusterNum) * (x_max - x_min), np.ones(ClusterNum) * (y_max - y_min)]
              ).transpose() + np.array([np.ones(ClusterNum) * x_min, np.ones(ClusterNum) * y_min]).transpose()
  return p

nodes = InitAllPoint()
initCenteroid = randomCenteroid(ClusterNum, nodes)
```

(2) Assigning step

In this step, all data records will be assigned to the known nearest centroids, while distances are measured by Euclid distance.

```
def Distance(a, b):
  return norm(a - b)

def classify(nodes, Centeroid):
  N = len(nodes)
```

```
  M = len(Centeroid)
  idx = [0]*N
  ClusterSize=[0]*M
  for i in range(N):
     dist0 = norm(nodes[i] - Centeroid[0])
     for j in range(1, M):
        dist = norm(nodes[i] - Centeroid[j])
        if dist < dist0:
           dist0 = dist
           idx[i] = j
     ClusterSize[idx[i]]+=1
  return idx,ClusterSize

idx, ClusterSize = classify(nodes, Centeroid)
```

(3) Summary Step

Regenerate every centroid of every cluster.

```
def generate_new_center(nodes, idx, ClusterSize):
  centers = np.zeros((np.size(ClusterSize),2))
  for i in range(np.sum(ClusterSize)):
     centers[idx[i]]+=nodes[i]

  for i in range(np.size(ClusterSize)):
     if ClusterSize[i]>0:
        centers[i]/=ClusterSize[i];
  return centers

NewPoint = generate_new_center(nodes, idx, ClusterSize)
```

(4) Summary Step

The program will continue iteration unless the clustering error converges (determined by a user-defined threshold).

```
while judge_center_difference(Centeroid, NewPoint, deviation) and cnt <
Maximum_iteration:
     Centeroid=NewPoint
     idx, ClusterSize = classify(nodes, Centeroid)
     NewPoint = generate_new_center(nodes, idx, ClusterSize)
```

```
    cnt += 1
```

Part 2: Implementation of GMM

The GMM algorithm can also be concluded as consisting of two major steps: expectation and maximization.

(1)Initialization

To begin with, read data records from input file and generate initial cluster. In this case, every "cluster" is presented as a Gaussian distribution function with parameter phi, mu(mean of the distribution) and sigma(covariance of the distribution).

```
def InitAllPoint():
  res = []
  f = open('clusters.txt', 'r')
  for line in f.readlines():
     res.append([float(x) for x in line.split(',')])

  res = np.array(res)
  x_min = min(res[:, 0])
  x_max = max(res[:, 0])
  y_min = min(res[:, 1])
  y_max = max(res[:, 1])
  return res, x_min, x_max, y_min, y_max


 node, x_min, x_max, y_min, y_max = InitAllPoint()

  nodeNum = len(node)

  nodeDimension = node[0].size

  # Initialization
  phi = [1.0/clusterNum]*clusterNum
  mu = randomCenteroid(clusterNum, node)
  sigma = [rand(nodeDimension, nodeDimension)*min(x_max-x_min, y_max-y_min)
        * np.array([[1, 0], [0, 1]]) for x in range(clusterNum)]
  W = np.zeros((nodeNum, clusterNum))
```

(2) Calculate Expectation

```
# Expectation
    for i in range(nodeNum):
        W[i, :] = [
            phi[j]*multivariate_gaussian(node[i], mu[j], sigma[j]) for j in range(clusterNum)]
        W[i, :] /= np.sum(W[i, :])
```

(3) Maximize the expectation and update the parameters

```
# Maximization
    phi = np.array([np.sum(W[:, j])/nodeNum for j in range(clusterNum)])

    mu = np.dot(W.T, node)
    mu = [mu[j]/np.sum(W[:, j]) for j in range(clusterNum)]

    for j in range(clusterNum):
        sigma[j] = np.zeros((nodeDimension, nodeDimension))
        for i in range(nodeNum):
            x = np.array([node[i]-mu[j]])
            sigma[j] += W[i][j]*x*x.T
        sigma[j] /= np.sum(W[:, j])
```

(4) Determine if the clustering process converges

If certain values of phi and mu compared to that of the last iteration remains actually the same, the clustering process is considered to converge and the iteration will come to an end.

```
for cnt in range(Maximum_iteration):
    phi0, mu0, sigma0, W0 = (phi, mu, sigma, W)

    # Expectation
    for i in range(nodeNum):
        W[i, :] = [
            phi[j]*multivariate_gaussian(node[i], mu[j], sigma[j]) for j in range(clusterNum)]
        W[i, :] /= np.sum(W[i, :])

    # Maximization
```

```
    phi = np.array([np.sum(W[:, j])/nodeNum for j in range(clusterNum)])

    mu = np.dot(W.T, node)
    mu = [mu[j]/np.sum(W[:, j]) for j in range(clusterNum)]

    for j in range(clusterNum):
        sigma[j] = np.zeros((nodeDimension, nodeDimension))
        for i in range(nodeNum):
            x = np.array([node[i]-mu[j]])
            sigma[j] += W[i][j]*x*x.T
        sigma[j] /= np.sum(W[:, j])

    if abs(reduce(lambda x, y: x**2+y**2, phi0)-reduce(lambda x, y: x**2+y**2, phi)) < deviation and
np.dot((np.array(mu)-np.array(mu0)).T, np.array(mu)-np.array(mu0))[0, 0] < deviation:
    # if np.sum(np.abs(W - W0)) < deviation:
        break
```

Part 3: Software Familiarization

(1). Gaussian Mixture Model

We can use mixture.GaussianMixture model from sklearn package. Here is how we use them:

```
from sklearn.mixture import GaussianMixture

GMM = GaussianMixture(n_components=5).fit(X)  # Suppose X is the target data. and we need 5
Guassian distributions

means = GMM.means_     # Here is the means of model
covariances = GMM.covariances_     # Here is the covariances

converge= GMM.converged_      # Check the model whether it is converged.
```

(2) K-means Algorithm

Sklearn library functions that offer good implementations of the K-means Algorithm:

```
from sklearn.cluster import KMeans
```

```
kmeans = KMeans(n_clusters=3).fit(X)   # Suppose X is the given data and we need 3 clusters
kmeans.cluster_centers_                # This is to see the centers of each clusters
kmeans.predict(Y)                      # Predict the closest cluster each sample in X belongs to.
```

Part 4: Application:

K-means:

K-means algorithm is a circular algorithm and it is hard-membership algorithm, which means each data point belongs to only one group. The k-means algorithm is used in a many fields such as **market segmentation, document clustering and image compression, etc.** However, it has some drawbacks: K-Means only detect spherical cluster, which makes it less efficient when applying to a model with complex geometrical shaped data.

Gaussian Mixture Model:

GMM, on the other hand, is a soft-membership algorithm and it can adjust itself to elliptic shape cluster. GMM is sometimes applicated in the field of finance to **predict market bottom or forecast or forecast asset return**.