hw4-Q1

April 5, 2021

1 1. Unsupervised Learning

[ ]:
```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
```

1.1 1. Generating the data

First, we will generate some data for this problem. Set the number of points $N = 400$, their dimension $D = 2$, and the number of clusters $K = 2$, and generate data from the distribution $p(x|z = k) = \mathcal{N}(\mu_k, \Sigma_k)$. Sample 200 data points for $k = 1$ and 200 for $k = 2$, with

$$\mu_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 6.0 \\ 0.1 \end{bmatrix} \quad \text{and} \quad \Sigma_1 = \Sigma_2 = \begin{bmatrix} 10 & 7 \\ 7 & 10 \end{bmatrix}$$

Here, $N = 400$. Since you generated the data, you already know which sample comes from which class. Run the cell in the IPython notebook to generate the data.

[ ]:
```
# TODO: Run this cell to generate the data
num_samples = 400
cov = np.array([[1., .7], [.7, 1.]]) * 10
mean_1 = [.1, .1]
mean_2 = [6., .1]

x_class1 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1 = np.column_stack((x_class1, np.zeros(num_samples // 2)))
xy_class2 = np.column_stack((x_class2, np.ones(num_samples // 2)))
data_full = np.row_stack([xy_class1, xy_class2])
np.random.shuffle(data_full)
data = data_full[:, :2]
labels = data_full[:, 2]
```

Make a scatter plot of the data points showing the true cluster assignment of each point using different color codes and shape (x for first class and circles for second class):

```
[ ]: # TODO: Make a scatterplot for the data points showing the true cluster␣
      ↪assignments of each point
     # plt.plot(...) # first class, x shape
     # plt.plot(...) # second class, circle shape
     c0 = data[labels==0, :]
     c1 = data[labels==1, :]

     plt.figure()
     plt.plot(c0[:, 0], c0[:, 1], 'x')
     plt.plot(c1[:, 0], c1[:, 1], 'o')
     plt.show()
```

1.2 2. Implement and Run K-Means algorithm

Now, we assume that the true class labels are not known. Implement the k-means algorithm for this problem. Write two functions: `km_assignment_step`, and `km_refitting_step` as given in the lecture (Here, `km_` means k-means). Identify the correct arguments, and the order to run them. Initialize the algorithm with

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix} \ , \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

and run it until convergence. Show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the cost vs. the number of iterations. Report your misclassification error.

```
[ ]: def cost(data, R, Mu):
         N, D = data.shape
         K = Mu.shape[1]
         J = 0
         for k in range(K):
             J += np.dot(np.linalg.norm(data - np.array([Mu[:, k], ] * N),␣
      ↪axis=1)**2, R[:, k])
         return J
```

```
[ ]: # TODO: K-Means Assignment Step
     def km_assignment_step(data, Mu):
         """ Compute K-Means assignment step

         Args:
             data: a NxD matrix for the data points
             Mu: a DxK matrix for the cluster means locations

         Returns:
             R_new: a NxK matrix of responsibilities
         """

         # Fill this in:
         N, D = data.shape
```

```
    K = Mu.shape[1]
    r = np.zeros((N, K))
    for k in range(K):
        err = data - np.expand_dims(Mu[:, k], axis=0)
        r[:, k] = np.sum(err**2, axis=1)
    arg_min = np.argmin(r, axis=1)
    R_new = np.zeros((N, K))
    R_new[list(range(N)), arg_min] = 1
    return R_new
```

[ ]:
```
# TODO: K-means Refitting Step
def km_refitting_step(data, R, Mu):
    """ Compute K-Means refitting step.

    Args:
        data: a NxD matrix for the data points
        R: a NxK matrix of responsibilities
        Mu: a DxK matrix for the cluster means locations

    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """
    N, D = data.shape
    K = Mu.shape[1]
    Mu_new = (data.T @ R) / np.sum(R, axis=0)
    return Mu_new
```

[ ]:
```
# TODO: Run this cell to call the K-means algorithm
N, D = data.shape
K = 2
max_iter = 100
class_init = np.random.binomial(1., .5, size=N)
R = np.vstack([class_init, 1 - class_init]).T

Mu = np.zeros([D, K])
Mu[:, 1] = 1.
R.T.dot(data), np.sum(R, axis=0)

c = []
for it in range(max_iter):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu)
    c.append(cost(data, R, Mu))
    #print(it, cost(data, R, Mu))

class_1 = np.where(R[:, 0])
class_2 = np.where(R[:, 1])
```

```
miss_err = np.sum(R[:, 0]==labels)/N
print('The missclassification error is: %f'  % miss_err)
```

[ ]:
```
# cost vs. the number of iterations
plt.figure()
plt.title('Cost vs. # of iterations')
plt.ylabel('cost')
plt.xlabel('iterations')
plt.plot(np.arange(max_iter), c, 'm-')
plt.show()
```

[ ]:
```
# TODO: Make a scatterplot for the data points showing the K-Means cluster␣
↪assignments of each point
d0 = data[class_1]
d1 = data[class_2]

plt.figure()
plt.plot(d0[:, 0], d0[:, 1], 'x')
plt.plot(d1[:, 0], d1[:, 1], 'o')
plt.show()
```

1.3 3. Implement EM algorithm for Gaussian mixtures

Next, implement the EM algorithm for Gaussian mixtures. Write three functions: `log_likelihood`, `gm_e_step`, and `gm_m_step` as given in the lecture. Identify the correct arguments, and the order to run them. Initialize the algorithm with the same initialization as in Q2.1 for the means, and with $\hat{\Sigma}_1 = \hat{\Sigma}_2 = I$, and $\hat{\pi}_1 = \hat{\pi}_2$ for the covariances.

Run the algorithm until convergence and show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the log-likelihood vs. the number of iterations. Report your misclassification error.

[ ]:
```
def normal_density(x, mu, Sigma):
    return np.exp(-.5 * np.dot(x - mu, np.linalg.solve(Sigma, x - mu))) \
        / np.sqrt(np.linalg.det(2 * np.pi * Sigma))
```

[ ]:
```
def log_likelihood(data, Mu, Sigma, Pi):
    """ Compute log likelihood on the data given the Gaussian Mixture␣
↪Parameters.

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
```

```
        L: a scalar denoting the log likelihood of the data given the Gaussian␣
↪Mixture
    """
    # Fill this in:
    N, D = data.shape
    K = Mu.shape[1]
    L, T = 0., 0.
    for n in range(N):
        T = 0
        for k in range(K):
            T += Pi[k] * normal_density(data[n], Mu[:, k], Sigma[k])
        L += np.log(T)
    return L
```

[ ]:
```
# TODO: Gaussian Mixture Expectation Step
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
        Gamma: a NxK matrix of responsibilities
    """
    # Fill this in:
    N, D = data.shape
    K = Mu.shape[1]
    Gamma = np.zeros((N,K))
    for n in range(N):
        for k in range(K):
            Gamma[n, k] = Pi[k] * normal_density(data[n], Mu[:, k], Sigma[k])
        Gamma[n, :] /= np.sum(Gamma[n, :])
    return Gamma
```

[ ]:
```
# TODO: Gaussian Mixture Maximization Step
def gm_m_step(data, Gamma):
    """ Gaussian Mixture Maximization Step.

    Args:
        data: a NxD matrix for the data points
        Gamma: a NxK matrix of responsibilities

    Returns:
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
```

```
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients
    """
    # Fill this in:
    N, D = data.shape
    K = Gamma.shape[1]
    Nk = np.sum(Gamma, axis=0)
    Mu = np.zeros([D, K])
    Sigma = np.zeros([K, D, D])

    for k in range(K):
        Mu[k] = (data.T @ Gamma)[k] / Nk[k]
        sigmasum = np.zeros([D,D])
        for n in range(N):
            xmmu = data[n:n+1].T - Mu[:,k:k+1]
            sigmasum += Gamma[n,k] * (xmmu @ xmmu.T)
        Sigma[k] = sigmasum / Nk[k]
    Pi = Nk / N
    return Mu, Sigma, Pi
```

[ ]:
```
# TODO: Run this cell to call the Gaussian Mixture EM algorithm
N, D = data.shape
K = 2
Mu = np.zeros([D, K])
Mu[:, 1] = 1.
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200

loglik = []
for it in range(max_iter):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    loglik.append(log_likelihood(data, Mu, Sigma, Pi))
    # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes the␣
 ↪computation longer, but good for debugging

class_1 = np.where(Gamma[:, 0] >= .5)
class_2 = np.where(Gamma[:, 1] >= .5)

miss_err = np.sum(np.float32(Gamma[:, 0] >= .5)==labels)/N
print('The missclassification error is: %f'  % miss_err)
```

[ ]:
```
# cost vs. the number of iterations
plt.figure()
```

```
plt.title('log-likelihood vs. # of iterations')
plt.ylabel('log-likelihood')
plt.xlabel('iterations')
plt.plot(np.arange(max_iter), loglik, 'm-')
plt.show()
```

[ ]:
```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture␣
↪cluster assignments of each point
d0 = data[class_1]
d1 = data[class_2]

plt.figure()
plt.plot(d0[:, 0], d0[:, 1], 'x')
plt.plot(d1[:, 0], d1[:, 1], 'o')
plt.show()
```

1.4 4. Comment on findings + additional experiments

Comment on the results:

- Compare the performance of k-Means and EM based on the resulting cluster assignments.
- Compare the performance of k-Means and EM based on their convergence rate. What is the bottleneck for which method?
- Experiment with 5 different data realizations (generate new data), run your algorithms, and summarize your findings. Does the algorithm performance depend on different realizations of data?

TODO: Your written answer here

- EM algorithm gives a much more accurate classification. The missclassification error is lower and the scatterplot is more similar to the true one.
- K-means converge faster than EM algorithm. The bottleneck of the two algorithms are about 5 for k-means and about 25 for EM algorithm.
- The misclassification error of k-means are always higher than that of EM algorithm. So the performance does not depend on different realization of data.

[ ]: