PRML Assignment3

1. Data

1.1 Design

Like assignment1, 2-D gaussian distribution is suitable to construct the dataset. Each cluster obeys a 2-D gaussian distribution with specific mu and sigma. But unlike assignment1, it is only the data that is used for training.

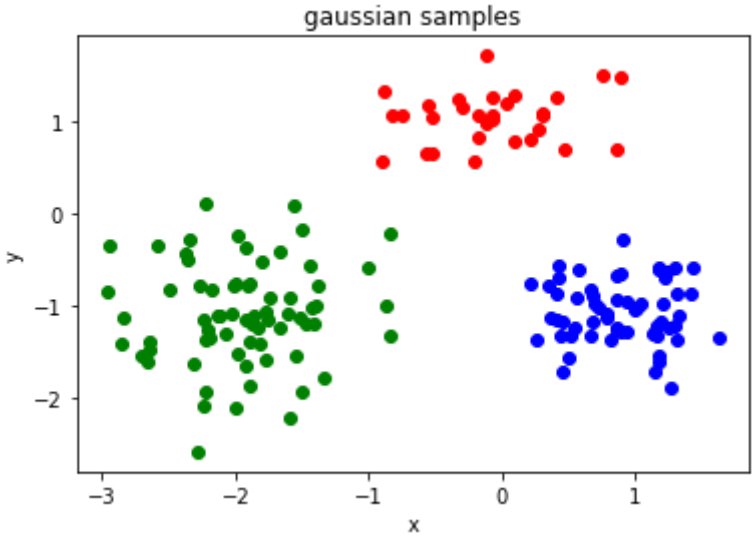
1.2 Preprocessing

First, generate three clusters of gaussian distributed samples with specific expectation, covariance matrix and number(gen_sample(mu,cov,n)). Create dataset based on the gaussian distributions generated (gen_dataset(k,mus,covs,ns,filename)). Then, write back the data and labels to the file and store it. Data and labels can be loaded from the file as arrays, with shape of (n,2) and (n,) respectively(load_dataset(filename)). Only data is used during the training process. Labels just serve to test the accuracy of the model.

1.3 Visualization

(mus=[[0,1],[-2,-1],[1,-1]], covs= [[[.3,0],[0,.1]],[[.2,0],[0,.3]], [[.1,0],[0,.1]]] ns=[30,70,60])

As is pointed in the figure, each cluster is marked with a unique color.



2. Model

- E-step: fix parameters μ, σ, calculate posterior distribution p($z^{(n)}|x^{(n)}$)

  $N(x^{(n)};\mu_k,\sigma_k) = \frac{1}{\sqrt{2\pi}\sigma_k} exp(-\frac{(x-\mu_k)^2}{2\sigma_k^2})$

```
def phi(Y, mu_k, cov_k):
    norm = multivariate_normal(mean=mu_k, cov=cov_k)
    return norm.pdf(Y)
for k in range(K):
    prob[:, k] = phi(Y, mu[k], cov[k])
```

$$\gamma_{nk} = \frac{\pi_k N(x^{(n)};\mu_k,\sigma_k)}{\sum_{k=1}^{K} \pi_k N(x^{(n)};\mu_k,\sigma_k)}$$

γ_{nk} defines the posterior probability for $x^{(n)}$ to belong to the k^{th} Gaussian distribution.

```
for k in range(K):
    gamma[:, k] = self.alpha[k] * prob[:, k]
for i in range(N):
    gamma[i, :] /= np.sum(gamma[i, :])
```

- M-step: find a group of parameters to maximize ELBO.

$$N_k = \sum_{n=1}^{N} \gamma_{nk}$$

$$\pi_k = \frac{N_k}{N}$$

$$\mu_k = \frac{1}{k} \sum_{n=1}^{N} \gamma_{nk} x^{(n)}$$

$$\sigma_k = \frac{1}{k} \sum_{n=1}^{N} \gamma_{nk} (x^{(n)} - \mu_k)^2$$

```
for k in range(K):
    Nk = np.sum(gamma[:, k])
    alpha[k] = Nk / N
    mu[k, :] = np.sum(np.multiply(x, gamma[:, k]), axis=0) / Nk
    cov_k = (x - mu[k]).T * np.multiply((x - mu[k]), gamma[:, k]) / Nk
    cov.append(cov_k)
```

- Calculate Log-margin distribution: whether it converges.

$$L(\theta) = \sum_{i=1}^{m} log \sum_{z^{(i)}} \pi_i(z^{(i)}) P(x^{(i)}, z^{(i)}|\theta) \quad s.t. \sum_{z} \pi_i(z^{(i)}) = 1$$

```
for k in range(self.K):
    P[:,k] = prob(self.X, self.mu[k], self.cov[k])
self.loss.append(np.sum(np.log(P.dot(self.alpha))))
```

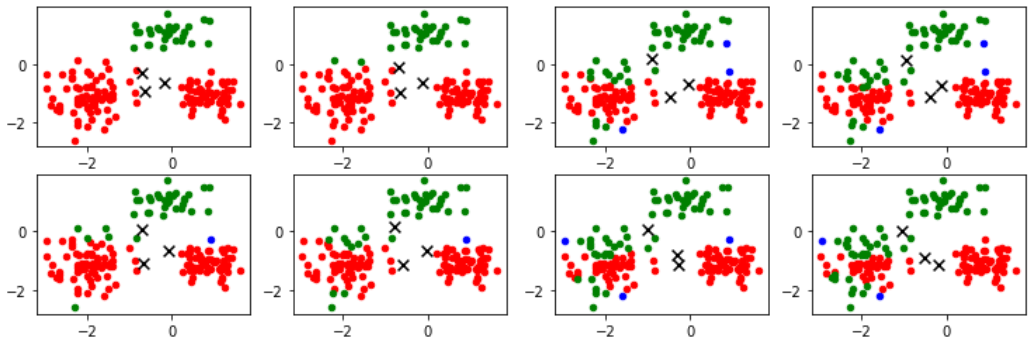
3. Analysis

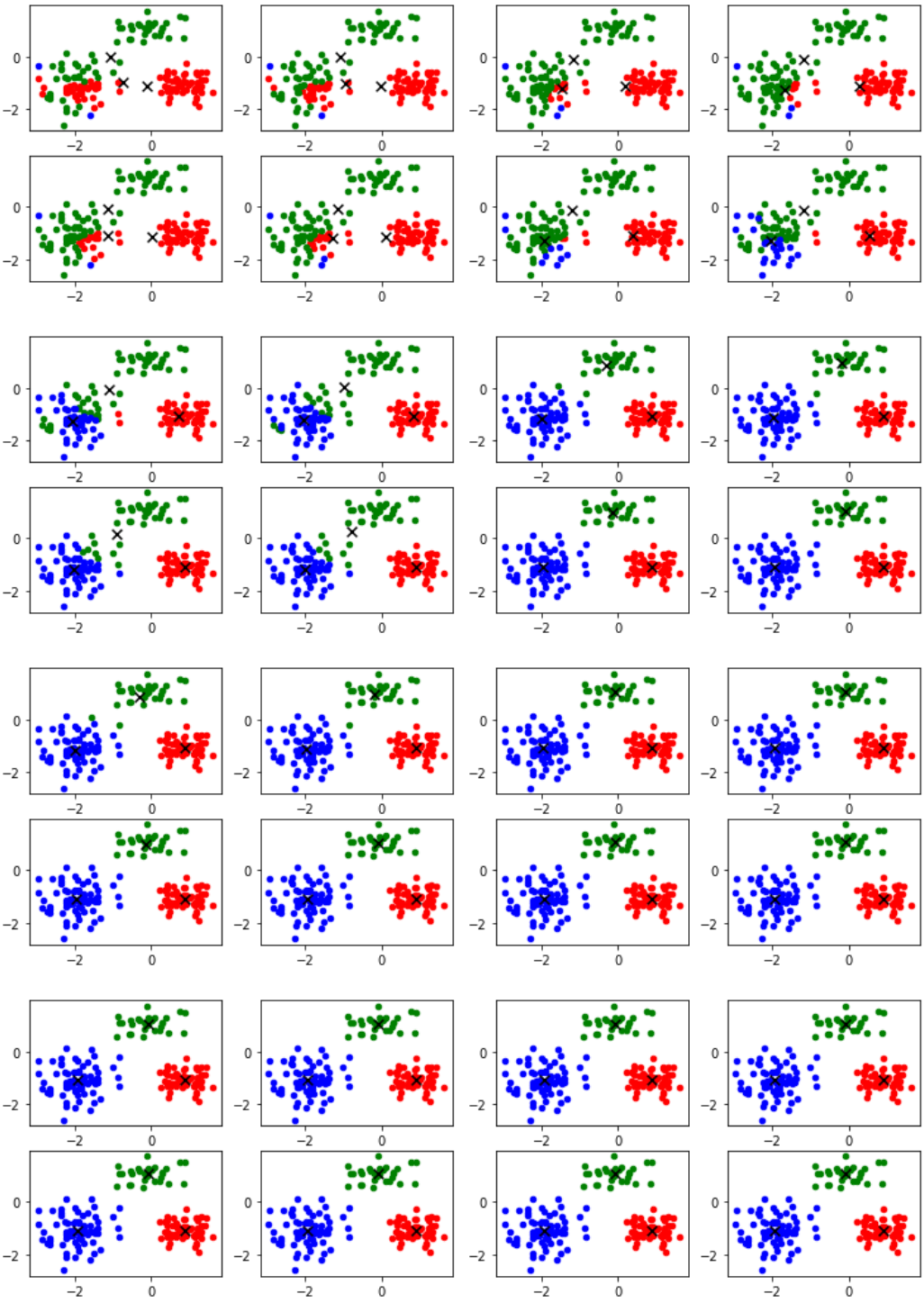
3.1 Training

An training example:

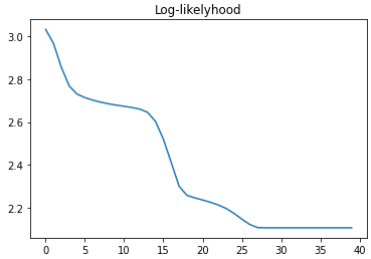
(mus=[[0,1],[-2,-1],[1,-1]], covs= [[[.3,0],[0,.1]],[[.2,0],[0,.3]], [[.1,0],[0,.1]]] ns=[30,70,60],iteration=40)

Visualization of training process:





Log-likelihood



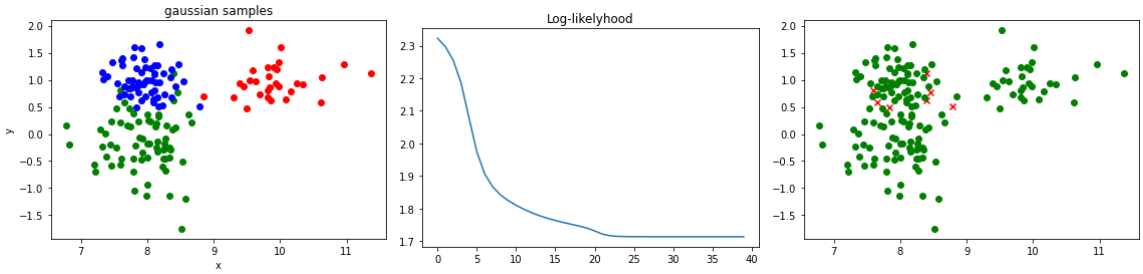
Judging from the loss curve, the model converges at nearly the 26th iteration. This also coincides with the fact that from the 26th iteration, the center of each cluster tends to remain stable.

Accuracy

The model performs well on the 3-cluster dataset, whose accuracy reaches 100%.

3.2 Separating capacity

(mus=[[10,1],[8,0],[8,1]], covs= [[[.3,0],[0,.1]],[[.2,0],[0,.3]], [[.1,0],[0,.1]]] ns=[30,70,60],iteration=40)



The accuracy decreases to 95.625% when the overlaps between clusters get larger. As is shown in the right picture above, the red cross represents the samples going to the wrong clusters. The miss-clustered samples falls in the area near the close boundary. However, the loss function converges more quickly. **That implies the initial value is one factor contributing to the speed to converge despite of the separating capacity of the dataset.**

3.3 Scale Data

If data is scaled, order of magnitude is uniformed for each dim. Thus, such data preprocessing keeps the influence on the parameters inserted by each dim the same. In one specific experiment, the accuracy rises from 88.75% to 90% after scaling. However, in another independent experiment on the same dataset, the accuracy drops from 91.25% to 88.75%. I guess scaling data often works when the order of magnitude differs dramatically among clusters and the distance between clusters is long. Under this circumstances, scaling data to the close order of magnitude accelerates the speed to converge and avoids the uneven influence on the parameters inserted by different dims. **In my dataset, there is no significant difference in the order of magnitude, thus this method doesn't work well.**

3.4 Evaluation

Since it is a clustering problem, each cluster may be assigned a different label compared to their initial assigned one. Thus, I employ a trick to evaluate the model. I give each cluster a label according to the predicted result. Then, I plot the result to compare it with the initial distribution and change the labels for the predicted result basing on that order. If we cannot tell them apart directly, calculating the distance between the predicted and initial mu to find the closet one may help. After adjusting the labels, this code can be used to evaluate performance:

```
wrong=[]
right=[]
for k in range(3):
    for i in range(clusters[k].shape[0]):
        if clusters[k][i] not in dataset[k]:
            wrong.append(clusters[k][i])
        else:
            right.append(clusters[k][i])
wrong=np.asarray(wrong)
right=np.asarray(right)
plt.scatter(wrong[:, 0], wrong[:, 1], marker='x',color='r')
plt.scatter(right[:, 0], right[:, 1], marker='o',color='g')
plt.show()
accuracy = right.shape[0]/(right.shape[0]+wrong.shape[0])
print(accuracy)
```

4. Discussion

4.1 Initialization

(mus=[[10,1],[8,0],[8,1]], covs= [[[.3,0],[0,.1]],[[.2,0],[0,.3]], [[.1,0],[0,.1]]] ns=[30,70,60],iteration=40)

4.1.1 Importance

EM algorithm is sensitive to the determination of initial value to iteration. Thus, it matters a lot when it comes to choosing the initialization method.

4.1.2 Random initialization

If the random initialization process is employed, the iterations it takes to converge is random as is shown by the previous experiment. It is likely to converge to local optimum . And as shown in the pictures during the training process, in the beginning iterations, the model tend to assume that the data only falls in two clusters according to the random initial values.

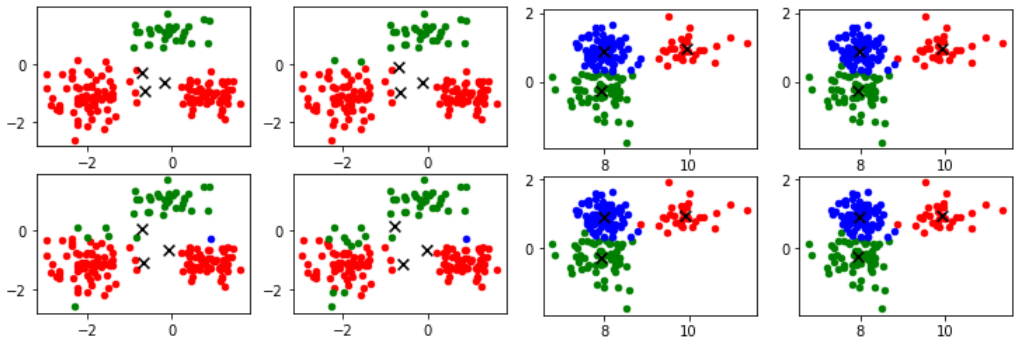
4.1.3 K-means++ initialization

Another method for initialization is K-means++. It is accordant with EM algorithm for parameter estimation of finite mixture model. Hence we can get a rough estimate of parameters from K-means++ as the initial input of the EM algorithm. K-means++ can boil down the following steps:

1. **Randomly select the first** centroid from the data points.
2. For each data point compute its distance from the nearest, previously chosen centroid.
3. **Select the next centroid 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)
4. Repeat steps 2 and 3 until k centroids have been sampled

4.1.4 Comparison

The accuracy rises from 95.625% to 96.25%, and the extreme prediction in the previous training doesn't occur.



(The first four iterations)

	c1	c2	c3	accuracy
real	(10,1)	(8,0)	(8,1)	
random	(9.91619479, 0.97290325)	(7.94555335, -0.12076131)	(7.9570914 , 0.98164505)	95.625%
k-means++,EM	(9.91457466, 0.9722658)	(7.9375902 , -0.16917106)	(7.96418163, 0.95794234)	96.25%

4.1.6 Drawback

The first center point is randomly selected. Thus, it cannot guarantee the initialization always makes sense. But it still outperforms the random-initialized model since **the probability for NAN in parameters to occur decreases.**

4.2 K-means and GMM

4.2.1 Link

The formula deduction are similar for the two, thus, they should bear some similarities. First, they can both be employed in clustering problems. The core idea of the two are quite similar. K-means first calculate the distance between the sample and the center of each cluster and assign it to the cluster with the minimum distance. Then, the center of each cluster is re-calculated according to the new partition. GMM calculates the probability a sample obeys certain distribution. Then,it re-calculates the mu and cov for each distribution based on the update. It can be seen that **both K-means and GMM boil down to two steps and the two steps closely depend on each other.** Moreover, they both **need an assigned value k**. The choice of K has great influence on the performance of model. Furthermore, they **only converge to the local optimum value** because they are not convex optimization.

4.2.2 Difference

K-means is a hard assignment while mixtures of Gaussians is a soft assignment. While K-means directly assign a sample to a cluster with the max probability and update the parameters based on the new partition, GMM saves posterior probability for $x^{(n)}$ to belong to the k^{th} Gaussian distribution in a matrix and utilize it to update the parameters. The difference between argmax and softmax can be used as an analogy. And since GMM is a soft assignment, **it can be used in not only clustering problems but also probability density estimation**. GMM can be also used to **generate new samples**. Thus, it is able to solve the maximum likelihood estimate when the observation data is incomplete.

5. Demo

```
python source.py
```

6. Reference

[1] 邱锡鹏，《神经网络与深度学习》https://nndl.github.io/

[2] 王鑫，《基于高斯混合模型的k均值初始化EM算法的研究》，2012

[3] Ye Li, Yiyan Chen, Research on Initialization on EM Algorithm Based on Gaussian Mixture Model, 2018