Lecture 7.3: Expectation Maximization

By courtesy of Siwei Causevic (https://towardsdatascience.com/implement-expectation-maximization-em-algorithm-in-python-from-scratch-f1278d1b9137).

1. Expectation maximization

EM is an iterative algorithm to find the maximum likelihood of data when there are latent variables. The algorithm iterates between performing an expectation (E) step, which updates the posterior distribution and computes the log-likelihood using the current estimate for the parameters, and a maximization (M) step, which computes parameters by maximizing the expected log-likelihood from the E step. The parameter-estimates from M step are then used in the next E step.

1.1 Mathematical deduction

We define the known variables as x, and the unknown label as z. We make two assumptions: the prior distribution $p(z)$ is binomial and $p(x|z)$ in each cluster is a Gaussian (i.e., we are dealing with a GMM with two components, $K = 2$).

$$p(z;\phi) = \phi^{1\{z=1\}}(1-\phi)^{1\{z=0\}}$$

$$p\left(x \mid z=0; \mu_0, \Sigma_0\right) = \mathcal{N}\left(\mu_0, \Sigma_0\right)$$

$$p\left(x \mid z=1; \mu_1, \Sigma_1\right) = \mathcal{N}\left(\mu_1, \Sigma_1\right)$$

All parameters are randomly initialized. For simplicity, we use θ to represent all parameters in the following equations.

$$\theta := \phi, \mu_0, \mu_1, \Sigma_0, \Sigma_1$$

At the expectation (E) step, we calculate the posteriors:

$$\begin{aligned} q\left(z^{(i)}=1\right) &:= p\left(z^{(i)}=1 \mid x^{(i)}; \theta\right) \\ &= \frac{p\left(x^{(i)} \mid z^{(i)}=1; \theta\right) p\left(z^{(i)}=1; \theta\right)}{\sum_{z^{(i)} \in \{0,1\}} p\left(x^{(i)} \mid z^{(i)}; \theta\right) p\left(z^{(i)}; \theta\right)} \end{aligned}$$

$$\begin{aligned} q\left(z^{(i)}=0\right) &:= p\left(z^{(i)}=0 \mid x^{(i)}; \theta\right) \\ &= \frac{p\left(x^{(i)} \mid z^{(i)}=0; \theta\right) p\left(z^{(i)}=0; \theta\right)}{\sum_{z^{(i)} \in \{0,1\}} p\left(x^{(i)} \mid z^{(i)}; \theta\right) p\left(z^{(i)}; \theta\right)} \end{aligned}$$

At the maximization (M) step, we find the maximizers of the log-likelihood and use them to update θ. Notice that the summation inside the logarithm makes the computation hard. To move the summation out of the logarithm, we use Jensen's inequality to find the lower bound.

$$
\begin{aligned}
\sum_{i=1}^{M} \log \sum_{z^{(i)} \in\{0,1\}} p\left(x^{(i)}, z^{(i)} ; \theta\right) &= \sum_{i=1}^{M} \log \sum_{z^{(i)}} q\left(z^{(i)}\right) \frac{p\left(x^{(i)}, z^{(i)} ; \theta\right)}{q\left(z^{(i)}\right)} \\
&= \sum_{i=1}^{M} \log \mathbb{E}_{z^{(i)} \sim q}\left[\frac{p\left(x^{(i)}, z^{(i)} ; \theta\right)}{q\left(z^{(i)}\right)}\right] \\
&\geq \sum_{i=1}^{M} \mathbb{E}_{z^{(i)} \sim q} \log \left[\frac{p\left(x^{(i)}, z^{(i)} ; \theta\right)}{q\left(z^{(i)}\right)}\right] \\
&= \sum_{i=1}^{M} \sum_{z^{(i)}} q\left(z^{(i)}\right) \log p\left(x^{(i)}, z^{(i)} ; \theta\right) \\
&\quad - \sum_{i=1}^{M} \sum_{z^{(i)}} q\left(z^{(i)}\right) \log q\left(z^{(i)}\right) = \ell(\theta)
\end{aligned}
$$

The equality holds when $q(z^{(i)}) = p(z^{(i)} | x^{(i)}; \theta)$.

$$\theta = \operatorname{argmax} \ell(\theta)$$

Luckily, there are closed-form solutions for the maximizers in GMM.

$$\phi = \frac{\sum_{i=1}^{M} q\left(z^{(i)}=1 \mid x^{(i)}\right)}{M}$$

$$\mu_0 = \frac{\sum_{i=1}^{M} x^{(i)} q\left(z^{(i)}=0 \mid x^{(i)}\right)}{\sum_{i=1}^{M} q\left(z^{(i)}=0 \mid x^{(i)}\right)}$$

$$\mu_1 = \frac{\sum_{i=1}^{M} x^{(i)} q\left(z^{(i)}=1 \mid x^{(i)}\right)}{\sum_{i=1}^{M} q\left(z^{(i)}=1 \mid x^{(i)}\right)}$$

$$\Sigma_0 = \frac{\sum_{i=1}^{M} q\left(z^{(i)}=0 \mid x^{(i)}\right)\left(x^{(i)}-\mu_0\right)\left(x^{(i)}-\mu_0\right)^T}{\sum_{i=1}^{M} q\left(z^{(i)}=0 \mid x^{(i)}\right)}$$

$$\Sigma_1 = \frac{\sum_{i=1}^{M} q\left(z^{(i)}=1 \mid x^{(i)}\right)\left(x^{(i)}-\mu_1\right)\left(x^{(i)}-\mu_1\right)^T}{\sum_{i=1}^{M} q\left(z^{(i)}=1 \mid x^{(i)}\right)}$$

We use these updated parameters in the next iteration of E step, get the new posteriors and run M-step. What the EM algorithm does is to repeat these two steps until the expected log-likelihood converges.

1.2 Implement EM from scratch

There are many packages including scikit-learn that offer high-level APIs to train GMMs with EM. In this section, we will implement the algorithm from scratch.

In [1]:
```
#Importing required libraries
import numpy as np
import pandas as pd
from scipy import stats
from scipy.special import logsumexp
from sklearn.mixture import GaussianMixture
from matplotlib import pyplot as plt
import time
# from google.colab import drive
# drive.mount('/content/drive')
# %cd /content/drive/MyDrive/Colab_Notebooks/lecture\ 7
```

When companies launch a new product, they usually want to find out the target customers. If they have data on customers' purchasing history and shopping preferences (as input features), they can utilize them to infer what types of customers are more likely to purchase the new product (i.e., infer the latent binary vairable z). There are many models to solve this typical unsupervised learning problem and the Gaussian Mixture Model (GMM) is one of them.

Using the known personal data, we have engineered two features x_1, x_2 represented by a matrix x, and our goal is to forecast whether each customer will like the product $(z = 1)$ or not $(z = 0)$.

In [2]:
```
data_unlabeled = pd.read_csv("./unlabeled.csv")
x_unlabeled = data_unlabeled[["x1", "x2"]].values
x_unlabeled.shape
```

Out[2]:
```
(1000, 2)
```

In some cases, we have a small amount of labeled data. For example, we might know some customers' preferences from surveys. Considering the potential customer base is huge, the amount of labeled data we have is insufficient for supervised learning, yet we can use such small amount of paired data to initialize the parameters.

In [3]:
```
data_labeled = pd.read_csv("./labeled.csv")
x_labeled = data_labeled[["x1", "x2"]].values
z_labeled = data_labeled["y"].values
```

First we initialize all the unknown parameters. In `initialize_params()` , we learn the initial parameters from the labeled data by direct parameter estimates:

$$\phi' = \frac{\sum_{i=1}^{m} 1\left\{z^{(i)} = 1\right\}}{n}$$

$$\mu_0' = \frac{\sum_{i=1}^{m} x^{(i)} 1\left\{z^{(i)} = 0\right\}}{\sum_{i=1}^{m} 1\left\{z^{(i)} = 0\right\}}$$

$$\mu_1' = \frac{\sum_{i=1}^{m} x^{(i)} 1\left\{z^{(i)} = 1\right\}}{\sum_{i=1}^{m} 1\left\{z^{(i)} = 1\right\}}$$

$$\Sigma_0' = \frac{\sum_{i=1}^{m} 1\left\{z^{(i)}=0\right\}\left(x^{(i)}-\mu_0'\right)\left(x^{(i)}-\mu_0'\right)^T}{\sum_{i=1}^{m} 1\left\{z^{(i)}=0\right\}}$$

$$\Sigma_1' = \frac{\sum_{i=1}^{m} 1\left\{z^{(i)}=1\right\}\left(x^{(i)}-\mu_1'\right)\left(x^{(i)}-\mu_1'\right)^T}{\sum_{i=1}^{m} 1\left\{z^{(i)}=1\right\}}$$

These estimated parameters are used in the first E step.

In [4]:
```
def initialize_params(x_labeled, z_labeled):
    n = x_labeled.shape[0]
    phi = x_labeled[z_labeled == 1].shape[0] / n
    mu0 = np.sum(x_labeled[z_labeled == 0], axis=0) / x_labeled[z_labeled == 0].s
    mu1 = np.sum(x_labeled[z_labeled == 1], axis=0) / x_labeled[z_labeled == 1].s
    sigma0 = np.cov(x_labeled[z_labeled == 0].T, bias= True)
    sigma1 = np.cov(x_labeled[z_labeled == 1].T, bias=True)
    return {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sigma
```

Then we pass the initialized parameters to `e_step()` and calculate the posteriors $q(z=1|x)$ and $q(z=0|x)$ for every data point as well as the expected log-likelihood which we will maximize in the M step.

In [5]:
```
def e_step(x, params):

    xxx = np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]).pdf(
            stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)])
    print(xxx)
    log_p_z_x = np.log([1-params["phi"], params["phi"]])[np.newaxis, ...] + \
                np.log([stats.multivariate_normal(params["mu0"], params["sigma0"]
                stats.multivariate_normal(params["mu1"], params["sigma1"]).pdf(x)
    log_p_z_x_norm = logsumexp(log_p_z_x, axis=1)

    return log_p_z_x_norm, np.exp(log_p_z_x - log_p_z_x_norm[..., np.newaxis])
```

In `m_step()` , the parameters are updated using the closed-form solutions in Eqs. (7) ~ (11). Note that if there weren't closed-form solutions, we would need to solve the optimization problem using gradient ascent and find the parameter estimates.

In [6]:
```
def m_step(x, params):
    total_count = x.shape[0]
    _, heuristics = e_step(x, params)
    heuristic0 = heuristics[:, 0]
    heuristic1 = heuristics[:, 1]
    sum_heuristic1 = np.sum(heuristic1)
    sum_heuristic0 = np.sum(heuristic0)
    phi = (sum_heuristic1/total_count)
    mu0 = (heuristic0[..., np.newaxis].T.dot(x)/sum_heuristic0).flatten()
    mu1 = (heuristic1[..., np.newaxis].T.dot(x)/sum_heuristic1).flatten()
    diff0 = x - mu0
    sigma0 = diff0.T.dot(diff0 * heuristic0[..., np.newaxis]) / sum_heuristic0
    diff1 = x - mu1
    sigma1 = diff1.T.dot(diff1 * heuristic1[..., np.newaxis]) / sum_heuristic1
    params = {'phi': phi, 'mu0': mu0, 'mu1': mu1, 'sigma0': sigma0, 'sigma1': sig
    return params
```

Now we can repeat running the two steps until the expected log-likelihood converges. `run_em()` returns the predicted labels, the posteriors and expected log-likelihood from all training steps.

In [7]:
```
def get_avg_log_likelihood(x, params):
    loglikelihood, _ = e_step(x, params)
    return np.mean(loglikelihood)


def run_em(x, params):
    avg_loglikelihoods = []
    while True:
        avg_loglikelihood = get_avg_log_likelihood(x, params)
        avg_loglikelihoods.append(avg_loglikelihood)
        if len(avg_loglikelihoods) > 2 and abs(avg_loglikelihoods[-1] - avg_logli
            break
        params = m_step(x_unlabeled, params)
    print("phi: %s\nmu_0: %s\nmu_1: %s\nsigma_0: %s\nsigma_1: %s"
               % (params['phi'], params['mu0'], params['mu1'], params['sigma0'],
    _, posterior = e_step(x_unlabeled, params)
    forecasts = np.argmax(posterior, axis=1)
    return forecasts, posterior, avg_loglikelihoods
```

Running the EM algorithm, we see the expected log-likelihood converges in a few steps.

In [8]:

```
init_params = initialize_params(x_labeled, z_labeled)
our_forecasts, our_posterior, our_loglikelihoods = run_em(x_unlabeled, init_param
print("total steps: ", len(our_loglikelihoods))
plt.plot(our_loglikelihoods)
plt.title("log likelihoods")
print("forecasts: ", our_forecasts)
```

```
[[ -3.14186983 -28.41583981  -5.7623618  ...  -1.11787968  -2.73648487
   -5.08890809]
 [ -1.90561911  -5.69982309  -1.02301499 ...  -7.17287343  -1.94850116
  -19.66996914]]
[[ -3.14186983 -28.41583981  -5.7623618  ...  -1.11787968  -2.73648487
   -5.08890809]
 [ -1.90561911  -5.69982309  -1.02301499 ...  -7.17287343  -1.94850116
  -19.66996914]]
[[ -3.04655787 -25.20731056  -5.28481836 ...  -1.07642696  -2.64423481
   -4.16236159]
 [ -1.96399612  -5.18585497  -1.13118636 ...  -6.95035737  -2.06554275
  -18.80811691]]
[[ -3.04655787 -25.20731056  -5.28481836 ...  -1.07642696  -2.64423481
   -4.16236159]
 [ -1.96399612  -5.18585497  -1.13118636 ...  -6.95035737  -2.06554275
  -18.80811691]]
[[ -3.08267887 -25.37369184  -5.31928026 ...  -1.06063824  -2.66719184
   -4.14904424]
 [ -1.99274352  -5.12577696  -1.14336282 ...  -6.9688029   -2.09820229
```

To verify our implementation, we compare our forecasts with forecasts from the scikit-learn API. To build the model in scikit-learn, we simply call the GaussianMixture API and fit the model with our unlabeled data. Don't forget to pass the estimated parameters to the model so it has the same initialization as our implementation. `GMM_sklearn()` returns the forecasts and posteriors from scikit-learn.

In [9]:

```
def GMM_sklearn(x, weights=None, means=None, covariances=None):
    model = GaussianMixture(n_components=2,
                            covariance_type='full',
                            tol=0.01,
                            max_iter=1000,
                            weights_init=weights,
                            means_init=means,
                            precisions_init=covariances)
    model.fit(x)
    print("\nscikit learn:\nphi: %s\nmu_0: %s\nmu_1: %s\nsigma_0: %s\nsigma_1: %s
                  % (model.weights_[1], model.means_[0, :], model.means_[1, :], mode
    return model.predict(x), model.predict_proba(x)[:,1]
```

```
In [10]: init_params = initialize_params(x_labeled, z_labeled)
         weights = [1 - init_params["phi"], init_params["phi"]]
         means = [init_params["mu0"], init_params["mu1"]]
         covariances = [init_params["sigma0"], init_params["sigma1"]]
         sklearn_forecasts, posterior_sklearn = GMM_sklearn(x_unlabeled, weights, means, c
         print("forecasts: ", sklearn_forecasts)
```

```
scikit learn:
phi: 0.59647894226803
mu_0: [-1.06169376 -1.0563389 ]
mu_1: [0.96408565 0.98206315]
sigma_0: [[0.35027155 0.29629092]
 [0.29629092 0.73083581]]
sigma_1: [[0.74510804 0.16156928]
 [0.16156928 0.32021029]]
forecasts:  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0
 1 1 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1
 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0
 0 0 0 1 1 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0
 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 1 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0
 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 1 0 0 1
 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0
 1 1 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1
 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0
 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 0 0 1
 1 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1
 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0
 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1 0 0
 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1
 0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 1 0 0 1 1 1 0 1 0 0 1
 0]
```

```
In [11]: output_df = pd.DataFrame({'our_forecasts': our_forecasts,
                                   'our_posterior': our_posterior[:, 1],
                                   'sklearn_forecasts': sklearn_forecasts,
                                   'posterior_sklearn': posterior_sklearn})
         print("\n%s%% of forecasts matched." % (output_df[output_df["our_forecasts"] == c
```

```
99.4% of forecasts matched.
```

Comparing the results, we see that the learned parameters from both models are very close (99.4% forecasts matched). In case you are curious, the minor difference is mostly caused by parameter regularization and numeric precision in matrix calculation.