Lab 3: Expectation Maximisation

Patrick de Kok

October 12, 2012

In this lab assignment, the datasets of both `banana.mat` and `spiral.mat` have been analyzed.

1 Gaussian Distribution

In this exercise, I have trained a classifier, based on a single Gaussian distribution for each class of both datasets.

For testing, I use the last 30% of both datasets. Both datasets have not been shuffled. The train and test sets of `banana.mat` and `spiral.mat` have been plotted in Figure 1 and Figure 2.

The training of the classifiers has been done as follows. For both classes $C_{\mathtt{A}}$ and $C_{\mathtt{B}}$, the mean $\boldsymbol{\mu} = [\boldsymbol{\mu}_{\mathtt{A}}, \boldsymbol{\mu}_{\mathtt{B}}]$ and covariances $\boldsymbol{\Sigma} = [\boldsymbol{\Sigma}_{\mathtt{A}}, \boldsymbol{\Sigma}_{\mathtt{B}}]$ are computed. For this assignment, I have assumed that the frequencies of $C_{\mathtt{A}}$ and $C_{\mathtt{B}}$ are representative of the real distribution. Therefore, $p(C_k)$ is equal to the number of data points of class C_k, divided by the total number of data points. I then compute the most likely class $C_{ML}(\mathbf{y})$ for a data point $\mathbf{y}$:

$$\begin{aligned}
C_{ML}(\mathbf{x}) &= \arg\max_{C_k} p(\mathbf{x}|C_k) \\
&= \arg\max_{C_k} \frac{p(C_k|\mathbf{x})p(C_k)}{\sum_j p(\mathbf{x}|C_j)p(C_j)} \\
&= \arg\max_{C_k} p(C_k|\mathbf{x})p(C_k) \\
&= \arg\max_{C_k} \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)p(C_k)
\end{aligned}$$

In MATLAB, the probability density function $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is computed by `mvnpdf(x, MU, SIGMA)`.

This computation for `banana.mat` results in the confusion matrix and error rate:

$$\begin{aligned}
CM(C_{ML}, \texttt{banana.mat}) &= \begin{bmatrix} 273 & 41 \\ 27 & 259 \end{bmatrix} \\
err(C_{ML}, \texttt{banana.mat}) &= 0.1133
\end{aligned}$$

Figure 1: A plot of the dataset `banana.mat`. The red "+"'s and magenta "o"'s represent the train and test data points from class A. The blue "×"'s and cyan "∗"'s represent the train and test data points from class B.

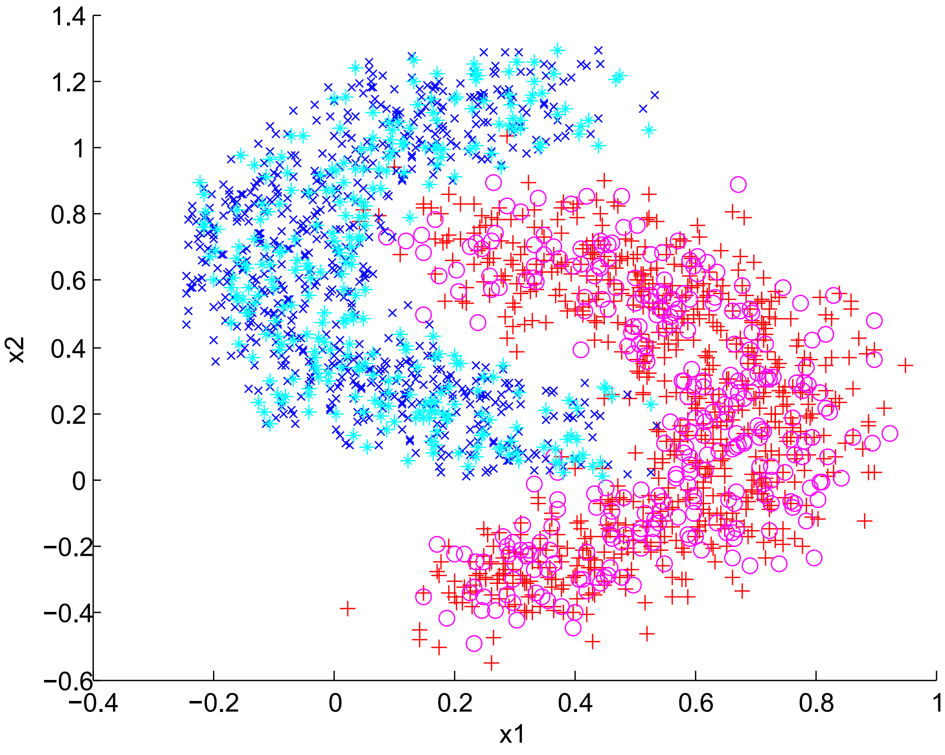
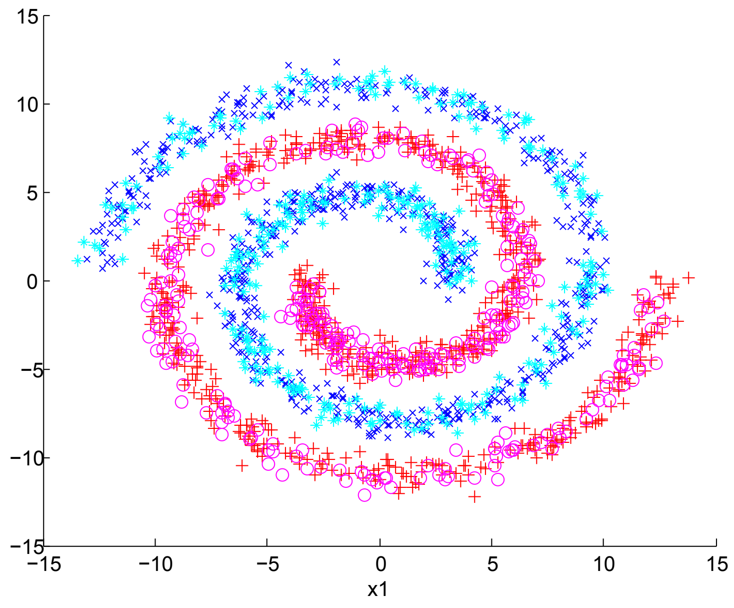


Figure 2: A plot of the dataset `spiral.mat`. The red "+"'s and magenta "o"'s represent the train and test data points from class A. The blue "×"'s and cyan "∗"'s represent the train and test data points from class B.



The corresponding values for `spiral.mat` are:

$$CM(C_{ML}, \texttt{spiral.mat}) = \begin{bmatrix} 219 & 110 \\ 81 & 190 \end{bmatrix}$$

$$err(C_{ML}, \texttt{banana.mat}) = 0.3183$$

2 Mixtures of Gaussians

In the second exercise, a classifier will be trained to recognize the classes of the `banana.mat` and `spiral.mat` datasets, based on the EM algorithm.

The iterative loop of the EM algorithm has been implemented already, as well as a visualization module, which is called in every iteration. To make everything work, I have implemented the following three MATLAB functions.

- `MOG = init_mog(X, C)`. Given the data and the desired number of Gaussians, compute the mean `MU`, covariance `SIGMA` and mixing coefficient `PI` for each Gaussian. For N data points, each Gaussian mixture is fitted through $N_k = N/\texttt{C}$ points, and the mean and covariance is computed for this set. `PI` is set to N_k. This is stored in `MOG`, a `C`-by-1 cell array of structures containing only these three keys. For the `k`th element of `MOG`, `MU`, `SIGMA` and PI in the software correspond with the mathematical concepts of $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ and π_k.

- `[Q LL] = mog_E_step(X, MOG)`. This implements the expectation step of the EM algorithm. For each data point $\mathtt{x} = \mathtt{X}(\mathtt{n},:)$ and the kth mixture with mean $\boldsymbol{\mu}_k$, covariance $\boldsymbol{\Sigma}_k$ and mixing coefficient π_k, the responsibility $\pi_k \mathcal{N}(\mathtt{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is divided by the sum of all responsibilities of the kth mixture, and stored in $\mathtt{Q}(\mathtt{n},\mathtt{k})$, a N-by-C matrix. This corresponds with equation 9.13 of [1]:

$$\begin{aligned}\mathtt{Q}(\mathtt{n},\mathtt{k}) = \gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) &= \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{C} p(z_j = 1)p(\mathbf{x}|z_j = 1)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{C} \pi_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}\end{aligned}$$

  However, in my implementation, only one iterator is used; MATLAB can compute the Gaussian distribution for a N-by-D matrix, given a 1-by-D mean and D-by-D covariance matrix. Another speedup is computing the denominator d together with the numerator, and only dividing each `Q(n, k)` when the complete denominator is known.

  LL is the log-likelihood of the dataset under the mixture model. This is computed corresponding to equation 9.14 of [1]:

$$\mathtt{LL} = \ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{C} \pi_k \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

  Because the innermost sum is the same as d, the implementation of this formula only takes marginally more time when Q is already computed.

- `MOG = mog_M_step(X, Q, MOG)`. This function updates the mean MU, covariance SIGMA and mixing coefficient PI in each cell of MOG. For each kth element, this is done according to equations 9.17, 9.19 and 9.22 of [1]:

$$\begin{aligned}\boldsymbol{\mu}_k &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \\ \boldsymbol{\Sigma}_k &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \\ \pi_k &= \frac{N_k}{N}\end{aligned}$$

  where N is the number of data points in X, $\mathbf{x}_n$ corresponds with the nth row of X and N_k is defined by equation 9.18:

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

such that $N = \sum_{k=1}^{\mathtt{C}} N_k$.

There are some things that do not follow straightforward from these definitions to the implementation.

Note that $\mathtt{Q(n,k)}$ corresponds with $\gamma(z_{nk})$. N_k is implemented as a 1-by-$\mathtt{C}$ row vector, generated by summing over all rows of $\mathtt{Q}$.

Because the mean and data points are stored as row vectors, the transpose operator is placed on the first difference vector in the computation of $\mathbf{\Sigma}_k$:

$$\mathbf{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \mathtt{Q(n,k)}(\mathtt{x}_n - \boldsymbol{\mu}_k)^T (\mathtt{x}_n - \boldsymbol{\mu}_k)$$

$\mathbf{\Sigma}_k$ is only updated if it is non-singular. In the code, this is defined as $\mathtt{cond}(\mathbf{\Sigma}_k) < 10^{10}$. If $\mathbf{\Sigma}_k$ is singular, the Gaussian has a really small area (almost no area) with high probabilities, so that almost no data point will be covered by it. This is hinted at in the exercise.

Finding the most likely class for the data is done in a similar fashion as for the previous exercise:

$$\begin{aligned}
C_{ML}(\mathbf{x}) &= \arg\max_{C_k} p(\mathbf{x}|C_k) \\
&= \arg\max_{C_k} p(C_k) \sum_j p(\mathbf{x}_n|z_j, C_k) p(z_j|C_k) \\
&= \arg\max_{C_k} p(C_k) \sum_j \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{jk}, \mathbf{\Sigma}_{jk}) p(z_j|C_k) \\
&= \arg\max_{C_k} p(C_k) \sum_j \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_{jk}, \mathbf{\Sigma}_{jk}) \pi_{jk}
\end{aligned}$$

I have tested the test data of both datasets for different models involving 1 through 40 Gaussian mixtures. The error rate of the test set of the `banana.mat` dataset is plotted against the number of Gaussian mixtures in Figure 3. Similar data is plotted in Figure 4 for the `spiral.mat` data set.

For the `banana.mat` dataset, the lowest error rate is 0.0117, which occurred with 10, 18 and 26 Gaussian mixtures. According to Occam's razor, one should always prefer the simplest hypothesis, I would advise using $\mathtt{C} = 10$.

I find the same number of Gaussians for the `spiral.mat` dataset. The lowest error rate is 0.0017, for $\mathtt{C} \in \{10, 16\}$. With the same argumentation, I advise 10 Gaussians.

Figure 3: The number of Gaussians plotted against the error rate for the `banana.mat` dataset.

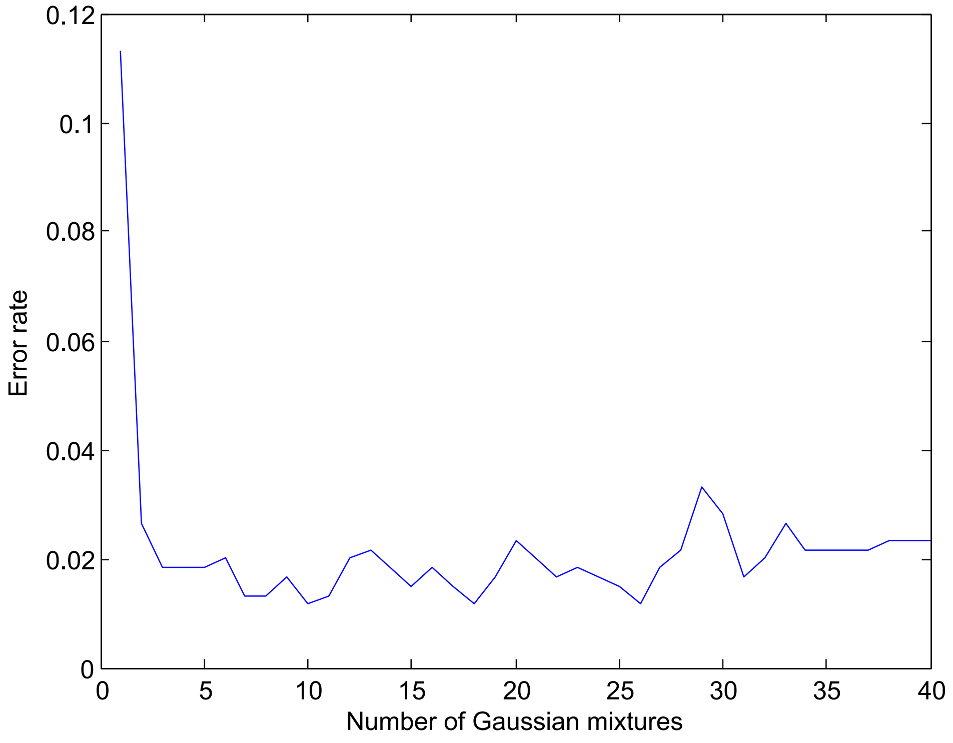
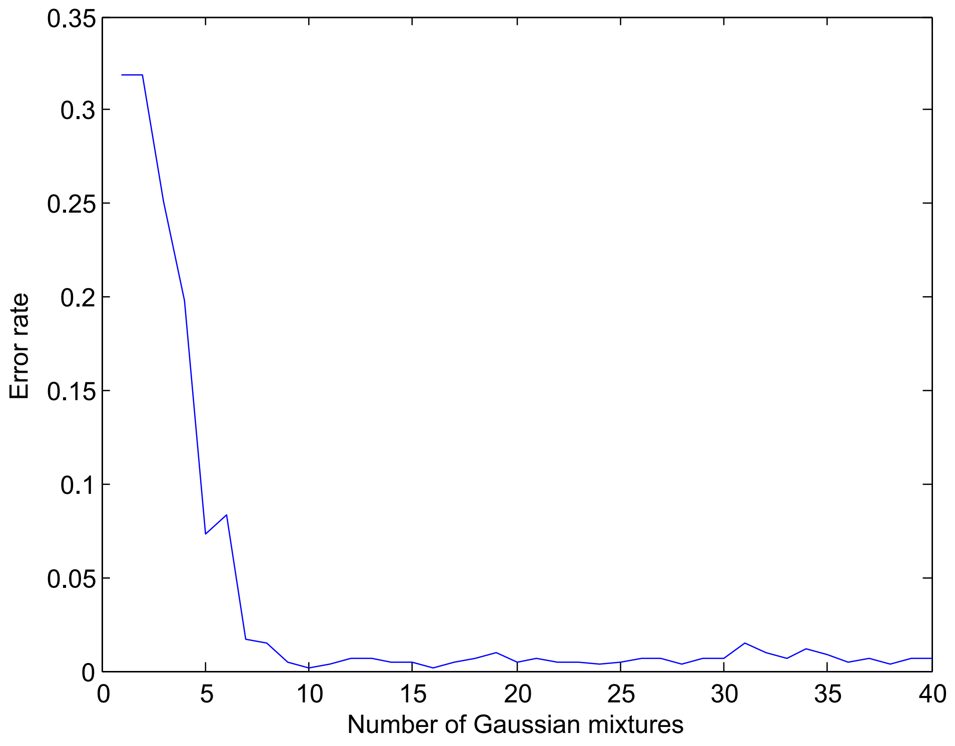


Figure 4: The number of Gaussians plotted against the error rate for the `spiral.mat` dataset.



3 Log-Probabilities

Because of underflow errors occurring when multiplying floating point numbers, one might like to work with logaritms of probabilities, instead of directly with probabilities. Multiplication of two logarithms $\ln(a)\ln(b)$ is the same as the logarithm of a sum $\ln(a+b)$. To do this, we need to compute that new value.

For two N-dimensional vectors x, y, determine per dimension the maximum and minimum element, and store these two vector as *high* and *low*. Then the logarithm of the sum is computed through $high + \ln(1 + \exp(low - high))$ [1]. The function is implemented in such a way that it computes the sum by adding the ith row by the intermediate result. By doing it in this way, `logsumexp` accepts a general matrix as input, and computes the sum along the rows. The implementation computes the correct sum. For example `logsumexp([-1000, -1001]) = -999.6867`.

The expectation step should be slightly modified. Where we used before

$$\mathtt{Q(n,k)} = \frac{\pi_k \mathtt{mvnpdf}(\mathbf{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{C} \pi_j \mathtt{mvnpdf}(\mathbf{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

we now compute:

$$\mathtt{Q(n,k)} = \exp\left(\ln \pi_k \mathtt{lmvnpdf}(\mathbf{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) - \sum_{j=1}^{C} \pi_j \mathtt{lmvnpdf}(\mathbf{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)$$

This did not change the results for both datasets, both for the best number of Gaussian mixtures for classification, as the error rates.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[1]Note that in the implementation, the scalar 1 must be replaced by a N-dimensional vector with all 1s.