Machine Learning Homework 4

General Instructions

Homework must be submitted electronically on Canvas. Make sure to explain your reasoning or show your derivations. Except for answers that are especially straightforward, you will lose points for unjustified answers, even if they are correct.

General Instructions

You are allowed to work with at most one other student on the homework. With your partner, you will submit only one copy, and you will share the grade that your submission receives. You should set up your partnership on Canvas as a two-person group by joining one of the preset groups named "HW4 Group n" for some number n.

Submit your homework electronically on Canvas. We recommend using LaTeX, especially for the written problems. But you are welcome to use anything as long as it is neat and readable.

For the programming portion, you should only need to modify the Python files. You may modify the iPython notebooks, but you will not be submitting them, so your code must work with our provided iPython notebook.

Relatedly, cite all outside sources of information and ideas.

Written Problems

1. We will derive the expectation-maximization (EM) algorithm using *variational* analysis.

   Let $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ be a set of data vectors. Let $Z = \{z_1, \ldots, z_n\}$ be set of multinomial variables corresponding to which of K Gaussians generated each example. Let the Gaussian parameters be means $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$ and covariance matrices $\{\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K\}$. Let $\Theta = \{\theta_1, \ldots, \theta_K\}$ be multinomial prior probabilities of which Gaussian generates each example.

   Each data point is generated by first sampling a Gaussian index from $p(z|\Theta)$, then sampling from the Gaussian $\mathcal{N}(\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$. The log likelihood of any observations given these mixture model parameters is

   $$L(X, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \Theta) = \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} p(z_i|\Theta)\, \mathcal{N}(x_i|\boldsymbol{\mu}_k, \Sigma_k) \right) = \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \theta_k\, \mathcal{N}(x_i|\boldsymbol{\mu}_k, \Sigma_k) \right) . \tag{1}$$

   Since we will never observe any z variables, these are considered *hidden* or *latent* variables. When computing the likelihood of observed variables, we sum over all possible states of the latent variables, weighted by the probability of those states.

   We start by doing something a little weird. We create an independent distribution q for the latent variables such that

   $$q(z_1, \ldots, z_n) := q(z_1)q(z_2) \ldots q(z_n). \tag{2}$$

   We then rewrite the log likelihood from Equation (1) so that each data point's likelihood is multiplied by the q distribution divided by itself. In other words, we multiply the terms in the innermost summation by $\frac{q(z_i=k)}{q(z_i=k)}$, resulting in the equivalent form of the likelihood:

   $$L(X, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \Theta, q) = \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \theta_k\, \mathcal{N}(x_i|\boldsymbol{\mu}_k, \Sigma_k)\, \underbrace{q(z_i = k)/q(z_i = k)}_{1} \right) . \tag{3}$$

Jensen's inequality guarantees that for any convex function φ and any distribution over random variable X,

$$\varphi\left(\mathbb{E}\left[X\right]\right) \leq \mathbb{E}\left[\varphi(X)\right] \ . \tag{4}$$

We use Jensen's inequality and the fact that $\log$ is a **concave** function (i.e., $-\log$ is a convex function) to form a lower bound on the log likelihood:

$$L(X, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \Theta, q) = \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \theta_k \, \mathcal{N}(x_i | \boldsymbol{\mu}_k, \Sigma_k) q(z_i = k) / q(z_i = k) \right) \tag{5}$$

$$\geq \sum_{i=1}^{n} \sum_{k=1}^{K} q(z_i = k) \log \left(\theta_k \, \mathcal{N}(x_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) / q(z_i = k) \right) \tag{6}$$

$$= \sum_{i=1}^{n} \sum_{k=1}^{K} q(z_i = k) \log \left(\theta_k \, \mathcal{N}(x_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right) - \sum_{i=1}^{n} \sum_{k=1}^{K} q(z_i = k) \log q(z_i = k). \tag{7}$$

(a) (5 points) You now have a lower bound objective function that depends on the Gaussian mixture model parameters $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$, and Θ and the variational distribution q. If we hold q fixed, maximizing the Gaussian mixture model parameters gets us the Gaussian EM updates (the so-called "m-step"). You will prove this fact for one of the mixture parameters. (You are welcome to try the other parameters too for fun.) Show that maximizing the lower bound with respect to the cluster mixture probabilities Θ is exactly the EM update for these variables.

You will need to use a Lagrange multiplier to enforce that $1 = \sum_{k=1}^{K} \theta_k$. Then solve for the settings of the Lagrange multiplier and each parameter θ_k to find the optimum of the constrained optimization. In other words, if the Lagrange multiplier is ζ and the Lagrangian form of the objective function is $\tilde{L}$, then you can find the solution by solving the following equations:

$$\frac{\partial \, \tilde{L}}{\partial \, \theta_k} = 0, \quad \text{and} \quad \frac{\partial \, \tilde{L}}{\partial \, \zeta} = 0. \tag{8}$$

Solve each equation in order and plug in the result into the original Lagrangian objective. Each zero-derivative condition will tell you something about the original objective that allows you to simplify it. You should end up with a final formula for the optimal value of θ_k that is relatively compact; Terms should simplify significantly to result in a simple final expression.

(b) (5 points) This second part is a bit more involved, but follows a similar line of reasoning. Show how to find the q parameters for each data point that maximize the lower bound.

You'll need to use Lagrange multipliers to enforce that $1 = \sum_k q(z_i = k)$ for each i, and you should be able to consider each data point's q distribution independently. Then solve for the settings of the Lagrange multiplier and each parameter $q(z_i = k)$ to find the optimum of the constrained optimization. In other words, if the Lagrange multiplier for the i'th variable is ζ_i and the Lagrangian form of the objective function is $\tilde{L}$, then you can find the solution by solving the following equations:

$$\frac{\partial \, \tilde{L}}{\partial \, q(z_i = k)} = 0, \quad \text{and} \quad \frac{\partial \, \tilde{L}}{\partial \, \zeta_i} = 0, \tag{9}$$

where we abuse notation to refer to the multinomial probability $q(z_i = k)$ as a variable.

You should again end up with a final formula for the optimal value of $q(z_i = k)$ that is relatively compact; Terms should simplify significantly to result in a simple final expression.

2. (5 points) Project proposal. See instructions on project homepage.

Programming Assignment

For this programming assignment, we have provided a lot of starter code. Your tasks will be to complete the code in a few specific places, which will require you to read and understand most of the provided code but will only require you to write a small amount of code yourself.

We have provided synthetic data that was generated with the following procedure:

- First, we generate a two-dimensional random Gaussian mixture model with five Gaussians.
- We then generate n (2000) two-dimensional data points from the mixture model.
- We project the two-dimensional data into 64-dimensional data using a random projection matrix.
- Finally, we add random noise in 64-dimensional space to the projected data.

The resulting data comes from a two-dimensional mixture model, but it is presented using 64-dimensions. You will use PCA and expectation maximization to attempt to fit a mixture model to the data.

The main experiment notebook is `run_synthetic_experiment.ipynb`. It first uses PCA to transform the data into the two most descriptive dimensions. It then runs k-means clustering on the data as a demonstration (k-means is already implemented for you directly in the notebook, but it probably won't work well unless your PCA implementation is correct). Then it runs expectation maximization using different numbers of Gaussians and records the log-likelihood of held-out validation data for each cluster count parameter.

A second experiment notebook called `eigen_and_gmm_faces.ipynb` runs a similar experiment using image data of faces. It uses PCA to reduce the dimensionality of raw-pixel representations of grayscale images. Then it uses a Gaussian mixture model to fit the shape of the reduced-dimension data so that it can generate random faces that look somewhat realistic.

We included a unit test class `test_pca_gmm.py` that runs PCA and Gaussian mixture modeling and runs a few tests. Your implementation should pass these tests in almost all cases. There is some randomness involved in the mixture modeling, so you may have very rare cases where a bad initialization causes a test to fail. But if you consistently fail the tests, then your implementation is probably incorrect.

Since various Python libraries include the functionality to do PCA and Gaussian mixture fitting, you are not allowed to submit code that uses these pre-built functions. You are however allowed to use the built-in numerical linear algebra functions `numpy.linalg.det`, `numpy.linalg.eig`, `numpy.linalg.svd`, or `numpy.linalg.cholesky`. (You don't need all of these, but you may want to use one or two.) You may also use any of the helper functions in `gmm.py`, especially `gaussian_ll`, which computes the log of the Gaussian likelihood for a set of data given a mean and covariance matrix.

1. (5 points) Complete the function `pca` in `pca.py`. The function should take a data matrix as input and return a transformed data matrix and the variance of the transformed data. The dimensions of the transformed data should be ordered such that the earlier indices have the highest variance. Using this ordering, one should be able to truncate to a lower-dimensional space while preserving the highest reconstruction accuracy by using the first few dimensions of the returned data.

2. (6 points) Complete `gmm` and `compute_membership` in `gmm.py`. The `gmm` function should take a data matrix and a number of clusters as input and fit the Gaussians using expectation maximization. And the `compute_membership` function should compute the likelihood for each example that it came from each of the K Gaussians, returning the likelihoods in a matrix with K rows and n columns. The main framework for the algorithm is set up for you already. You need to complete the code for updating the parameters (the Gaussian means and covariances and the cluster priors) and the latent variable probabilities (the cluster assignments).

   One important note for fitting Gaussians is that sometimes we can get degenerate data that creates numerically unstable covariances. To avoid this problem, one fix is to add a small constant to the diagonal of the estimated covariance matrix. The matrix `reg` in `gmm` is set up for you to do this. You can argue that this trick is regularization, or if you're more honest, it's just a reasonable hack to make things work.

3. (4 points) Complete the function `gmm_ll` in `gmm.py`. This function should take a data matrix and Gaussian mixture model parameters as input and output a log likelihood. The formula for the log likelihood is

$$\begin{aligned}
\mathrm{ll}(\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\theta}) &= \log \prod_{i=1}^{n} p(\boldsymbol{x}_i | \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\theta}) \\
&= \log \prod_{i=1}^{n} \sum_{k=1}^{K} \theta_k \, \mathcal{N}(x_i | \boldsymbol{\mu}_k, \Sigma_k) \\
&= \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \theta_k \, \mathcal{N}(x_i | \boldsymbol{\mu}_k, \Sigma_k) \right) \\
&= \sum_{i=1}^{n} \log \left(\sum_{k=1}^{K} \frac{\theta_k}{\sqrt{(2\pi)^d |\Sigma_k|}} \exp \left(-\frac{1}{2} (\boldsymbol{x}_i - \boldsymbol{\mu}_k)^\top \Sigma_k^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu}_k) \right) \right).
\end{aligned} \tag{10}$$

   With some manipulation, you should be able to use the log-sum-exp trick to compute the terms in the summation while avoiding numerical underflow. Notice that the term inside the innermost summation can be written as the $\exp$ of the log density of a Gaussian distribution. The log Gaussian density is returned by the provided function `gaussian_ll`. We provided an implementation of `logsumexp` at the bottom of `gmm.py`.

Once you complete these three pieces, you should be able to run the entirety of both experiment scripts. You should see some structure in the synthetic data points if PCA works correctly to identify the non-noise dimensions of the data, and you should see how using more Gaussians leads to higher likelihood with diminishing returns. And on the face-data experiment, you should see some plausible faces sampled from the learned Gaussian mixture model. (They are not very realistic compared to more modern deep generative models.)