CP4: Gaussian Mixture Models

Last modified: 2024-04-18 10:57

Status: **Released.**

Due date: Thu. Apr. 25, 2024 by 11:59pm ET (Boston time)

What to turn in:

- ZIP file of source code: https://www.gradescope.com/courses/712231/assignments/4368647/
- PDF report file: https://www.gradescope.com/courses/712231/assignments/4368648/

Your ZIP file should include

- All starter code .py files (with your edits) (in the top-level directory)
- These will be *auto-graded*. Be sure to check gradescope test results for immediate feedback.

Your PDF should include (in order):

- Your full name
- Collaboration statement
- About how many hours did you spend (coding, asking for help, writing it up)?
- Problem 1a, 1b, and 1c

We have included Problem 2 below for your own learning, but it is fully OPTIONAL (not graded)

Questions?: Post to the cp4 topic on the discussion forums.

Jump to: Problem 1 Problem 2 Starter Code Dataset: FashionMNIST

Overview and Background

In this coding practical, we will implement 2 possible algorithms for learning the parameters (weights, means, and standard-deviations) of a Gaussian mixture model:

- Expectation Maximization (Problem 1, required)
- LBFGS Gradient descent (Problem 2, **optional**)

The problems below will try to address several key practical questions:

- How sensitive are methods to initialization?
- How can we effectively select the number of components?
- What kind of structure can GMMs uncover within a dataset?

Starter Code

You can find the starter code and datasets in the course Github repository here:

https://github.com/tufts-ml-courses/cs136-24s-assignments/tree/main/unit4_CP

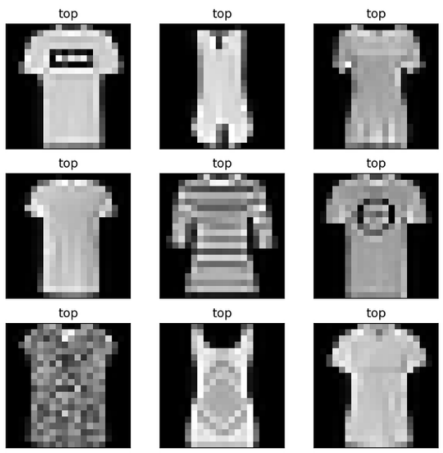
Dataset: Images of Shirts from FashionMNIST

Your goal will be to train mixture models for images of short or long-sleeve shirts ("tops") from the FashionMNIST dataset dataset.

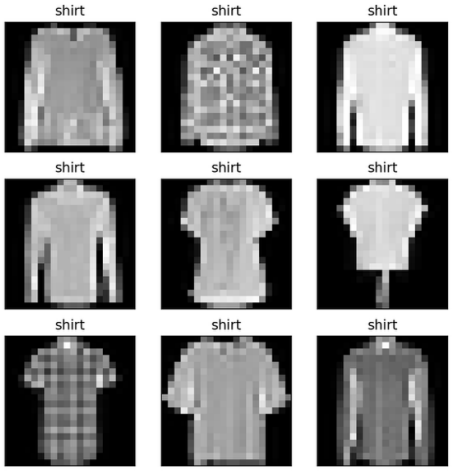
Inside the `data/` folder of the starter code repo, you will find several CSV files:

- tops-20x20flattened_train.csv : Training set of 3000 images (1000 from each category)
- tops-20x20flattened_valid.csv : Validation set of 1500 images (500 from each category)
- tops-20x20flattened_test.csv : Test set of 1500 images (500 from each category)

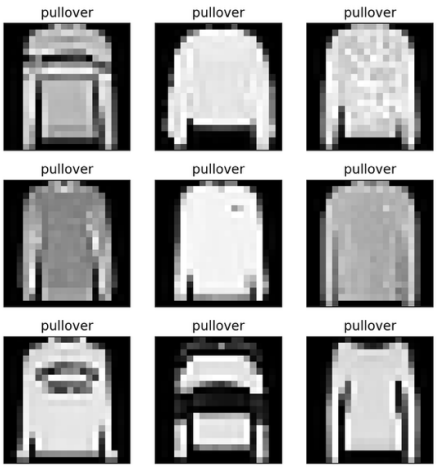
These represent a targeted subset of the FashionMNIST dataset, consisting of 20 pixel x 20 pixel images of 3 categories from Fashion MNIST: "t-shirt", "pullover", and "shirt", illustrated below:



Tops category



Shirts category



Pullover category

To represent each image, we took the 20x20 grayscale image, rescaled each pixel's intensity value to be a float between -1.0 and +1.0, and then "flattened" the image into a feature vector of size 400. Each row of the CSV file represents one "flattened" 20x20 pixel image.

Open question: What is the difference between a "top" and a "shirt"? If anyone has a theory, please post on Piazza.

Probabilistic Model

We are given N data examples of observed feature vectors: $\{x_n\}_{n=1}^N$, with $x_n \in \mathbb{R}^D$.

Our goal for model fitting is to learn a flexible probability distribution over x, $p(x_n)$.

Likelihood model

We model the N observed outputs $x = \{x_n\}_{n=1}^N$ via a K-cluster Gaussian mixture model where we assume all clusters have a diagnoal covariance matrix.

This model has the following likelihood:

$$p(x_{1:N}) = \prod_{n=1}^{N} \text{GMMPDF}(x_n|\pi, \mu, \sigma)$$

$$= \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} \text{NormPDF}(x_{nd}|\mu_{kd}, \sigma_{kd}^2)$$

NB: this is the formulation without latent assignment variables z

We have three kinds of parameters:

- $\pi \in \Delta^K$ are the cluster frequency weights. We define vector $\pi = [\pi_1\ \pi_2\ \pi_3\ \ldots\ \pi_K]$, where each entry is non-negative and the whole vector π sums to one.
- $\mu_k \in \mathbb{R}^D$ is a vector of cluster-specific means.
- $\sigma_k = [\sigma_{k1}\ \sigma_{k2}\ \ldots\ \sigma_{kD}]$ is a vector of cluster-specific standard deviations. Each standard deviation must be positive: $\sigma_{kd} > 0$.

We emphasize that throughout this CP problem set, the variance is controlled by a separate parameter for each data dimension (indexed by d), and thus there is assumed no correlation between distinct dimensions of the vector x_n.

Training Goal: Maximize penalized likelihood

We wish to minimize the following loss function, which can be seen as a penalized maximum likelihood formulation

$$\min_{\substack{\pi \in \Delta^K \\ \mu : \mu_k \in \mathbb{R}^D \\ \sigma : \sigma_k \in \mathbb{R}^D_+}} \text{penalty}(\sigma) - \sum_{n=1}^{N} \log \text{GMMPDF}(x_n | \pi, \mu, \sigma)$$

where the likelihood has been defined above, and the penalty term is designed to avoid the degeneracies of unpenalized maximum likelihood training of GMMs where some clusters can gain "infinite" likelihood by sending $\sigma_k \to 0$.

We will use the following penalty on the standard deviation parameters:

$$\text{penalty}(\sigma) = \sum_{k=1}^{K} \sum_{d=1}^{D} \frac{1}{m^2 \cdot s} \log \sigma_{kd} + \frac{1}{2} \frac{1}{m \cdot s} \sigma_{kd}^{-2}$$

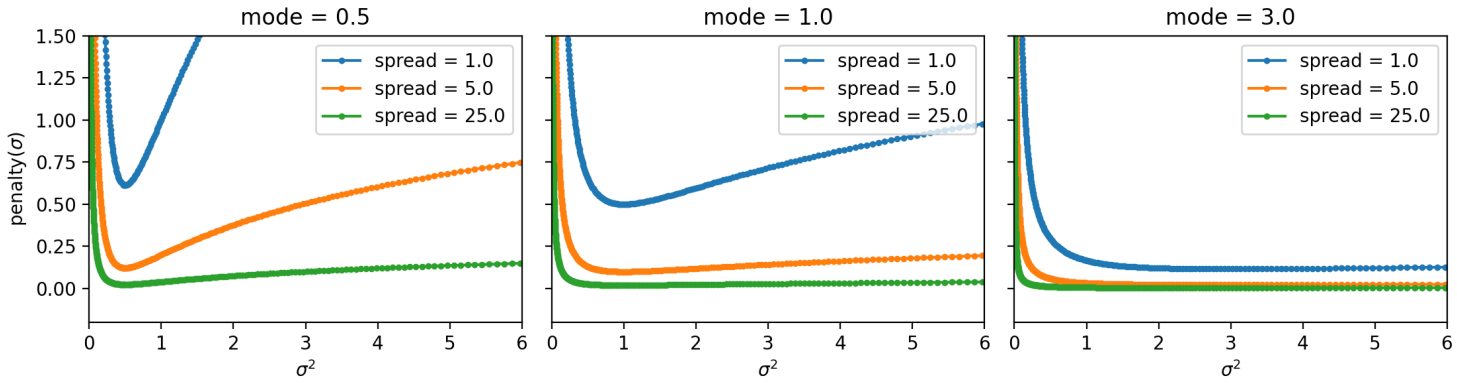
with two parameters: $m > 0$ and $s > 0$.

We will call m the "mode" of the penalty, because if the penalty were the only thing optimized, we would prefer to set our variance to $\sigma^2 = m$.

We will call s the "spread" of the penalty, because larger values make the shape of the function around the mode flatter and flatter (more spread out)

Throughout CP4, we set `m = 25.0` and `s = 100.0`.

See the figure below for an illustration of our penalty function across various m and s values:



Plot of penalty function vs sigma. See `stddev_penalty_function.py` in starter code.

NB: This chosen penalty function can be motivated as arising from a Gamma prior on the variance parameter (and thus we can view our penalized optimization as doing MAP estimation for σ). However, for simplicity we'll focus on the penalty function view.

Pseudocode for Algorithm #1 ("EM"): Expectation Maximization

We will consider the EM algorithm, which is outlined in Bishop's PRML textbook chapter 9 and in our lecture notes from day 19.

The algorithm proceeds as follows:

- Initialize the parameters π, μ, σ at random
- for each iteration $t \in 1, 2, \ldots T$:
  - for each example $n \in 1, 2, \ldots N$:
    - *E step for assignments*: Update assignment probability vector r_n given current parameters π, μ, σ:

$$r_{nk} = \frac{\pi_k \prod_d \text{NormPDF}(x_{nd} | \mu_{kd}, \sigma_{kd})}{\sum_{\ell=1}^K \left(\pi_\ell \prod_d \text{NormPDF}(x_{nd} | \mu_{\ell,d}, \sigma_{\ell,d}) \right)}, \quad \text{for } k \in 1, 2, \ldots K$$

  - for each cluster $k \in 1, 2, \ldots K$:
    - *M step for weights*: Update π_k given x, r: $\pi_k = \frac{\sum_n r_{nk}}{N}$
    - *M step for means*: Update each dim. μ_{kd} given x, r: $\mu_{kd} = \frac{\sum_n r_{nk} x_{nd}}{\sum_n r_{nk}}$
    - *M step for variances*: Update each dim. σ_{kd} given x, r, μ_{kd}: $\sigma_{kd}^2 = \frac{\frac{1}{s \cdot m} + \sum_n r_{nk} (x_{nd} - \mu_{kd})^2}{\frac{1}{s \cdot m^2} + \sum_n r_{nk}}$

NB: We have done the math to incorporate the penalty into the σ update. Recall that m is the mode and s is the spread of the penalty term.

We can show that this algorithm minimizes a principled objective function involving a negative expected log complete likelihood, the negative entropy of $q(z_n | r_n)$, and the penalty term defined above. The first two terms of this objective are worked out for the $D = 1$ case on page 9 of they day19 notes though beware that math is maximization rather than minimization.

Hints for Implementation

Scalability : Avoid for loops as much as possible. Anything you can do with predefined numpy functions that are vectorized (e.g. using `my_sum = np.sum(vec_K)` instead of `for v in vec_K: my_sum += v`), please do so. In our solutions, we have 0 for loops over examples n, and usually within each routine only one for loop over clusters k. Recommended reading: https://realpython.com/numpy-array-programming/.

Numerical stability : You should probably try to use the logsumexp trick to compute the GMMPDF likelihood. The product of many normal PDFs could cause problematic underflow otherwise. Better to compute the sum of log PDFs, and then use the logsumexp to compute the total log probability.

Problem 1: EM for GMMs

In problem 1, we will implement EM coordinate descent for training GMMs, using the penalized ML loss function above.

Tasks for Code Implementation

(All code steps will be evaluated by Autograder)

WRITING CODE 1(i) Implement the `calc_neg_log_lik` method of starter code class `calc_neg_log_lik__np.py`.

WRITING CODE 1(ii) Implement the `estep__calc_r_NK` method of starter code class `GMM_PenalizedMLEstimator_EM`.

WRITING CODE 1(iii) Implement each of 3 the `mstep__update` methods in starter code class `GMM_PenalizedMLEstimator_EM`.

WRITING CODE 1(iv) Implement the remaining TODOs in the `fit` method of starter code class `GMM_PenalizedMLEstimator_EM`.

USING CODE 1(v) Use the provided `train_EM_gmm_with_many_runs.py` script. This considers several values of K: 1, 4, 8, 16. For each value of K, initialize a GMM using the provided `generate_initial_parameters` function with a specified random seed, and train for 20 EM iterations. Repeat for all the 4 different random seeds provided in the starter code (1001, 3001, 4001, 7001).

By design, this has a limited computational budget. Once your code is correct, training all models should take about ~15-30 minutes on a modern laptop (like your instructor's 4-year-old macbook)

- K=4 models typically take ~10 sec to do 20 iters
- K=8 models typically take ~20 sec to do 20 iters

USING CODE 1(vi) Among all EM training runs for each K, select the single run that has the *best validation likelihood* and mark this as the "best" run.

Tasks for Report PDF

1a: FIGURE: EM Validation Likelihood vs. Iteration, across K values. Using your pretrained EM models from 1(v) above, use the "plot_history_for_many_runs.py" script to make a figure where each panel shows a line plot of the validation score (log likelihood per pixel) versus iteration, with a separate line for each random seed.

Caption for 1a: Summarize which model (out of $K = 1, 4, 8, 16$) seems to do best on this data.

1b: FIGURE: Visualization of best EM parameters with K=8 Using the best run with K=8, make a figure showing the visualization of the learned GMM parameters.

Caption for 1b: Provide a caption that interprets this visual to estimate how much of the data appears to consist of (a) long-sleeve, (b) short-sleeve, or (c) no-sleeve shirts.

1c: TABLE: Scores vs K Compute and report the score (log likelihood per pixel) on both **validation** set and **test set**, for the best EM-trained models with K=1, K=4, K=8, and K=16. You'll have to write your own code to load this test set from `data/` and determine the score.

Algorithm #2 ("LBFGS"): Gradient Descent via L-BFGS

All of Problem 2 is optional in Spring '24

We will consider a "gold standard" first-order gradient descent method called "L-BFGS" limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm, which uses first-order gradients computed at every iteration plus intelligent tracking of previous gradient steps to obtain an affordable approximation to the Hessian required for Newton-Raphson second order steps.

Why L-BFGS? Often, its Newton-method-inspired step-size selection is far more effective than just doing steepest descent (e.g. $\theta \leftarrow \theta - \epsilon \nabla_\theta \mathcal{L}(\theta)$) or other first-order methods where you'd need to tune your step size carefully.

Technical Challenge: Transform to unconstrained parameters

For gradient-based methods to work effectively, we cannot just naively take gradient steps of *constrained* parameters like the vector π (which must live in Δ^K) or the scalar standard deviations σ_{kd} (which must live in the positive reals).

Mixture weight transformation : To handle the constrained vector $\pi \in \Delta^K$, we replace it with a deterministic transformation T which produces an unconstrained vector $\rho \in \mathbb{R}^K$ via the following transformation:

$$\rho = T(\pi) = [\log \pi_1 \;\; \log \pi_2 \;\; \log \pi_3 \;\; \ldots \log \pi_K]$$
$$\log \pi = T^{-1}(\rho) = [\rho_1 - \text{logsumexp}(\rho), \;\; \ldots \;\; \rho_K - \text{logsumexp}(\rho)]$$

Recall that the logsumexp function computes $\log \sum_k e^{\rho_k}$ in a numerically stable way.

Throughout, in our code implementation we will work directly with the vector $\log \pi$ (the elementwise application of the logarithm to the length-K vector π).

Variance transformation : To handle the constrained vector $\sigma_k \in \mathbb{R}^D_+$ (the set of vectors with all positive entries), we use the following deterministic transformation U to produce an unconstrained vector $\nu_k \in \mathbb{R}^D$:

$$\sigma_k = U(\nu_k) = [\text{softplus}(\nu_{k1}) \; \text{softplus}(\nu_{k2}) \ldots \text{softplus}(\nu_{kD})]$$
$$\nu_k = U^{-1}(\sigma_k) = [\text{softplus}^{-1}(\sigma_{k1}) \; \text{softplus}^{-1}(\sigma_{k2}) \ldots \text{softplus}^{-1}(\sigma_{kD})]$$

where the softplus function and its inverse are defined as:

$$\text{softplus}(a) = \log(1 + e^a)$$
$$\text{softplus}^{-1}(b) = \log(e^b - 1)$$

We can guarantee that $\text{softplus}(a) > 0$ for any $a \in \mathbb{R}$, as required to produce a valid variance.

Many other transforms *could* have been used. One reason to use the softplus instead of just "exp/log" is that we want small changes in ν (e.g. after a small gradient update) to not produce big changes in π. Softmax has a much more *linear* relationship between ν and σ than the exp would.

Algorithm : Estimating a GMM via L-BFGS

- Initialize the parameters π, μ, σ at random
- Transform to unconstrained parameters: $\rho = T(\pi), \nu = U(\sigma)$.
- Pack all unconstrained parameters into one parameter vector θ, which will have the initial value at iteration 0: $\theta^0 = [\rho, \mu, \nu]$

- Prepare two functions that can take as input the unconstrained vector θ: the loss and the gradient: calc_loss and calc_grad
- Repeat at every iteration $t \in 1, 2, \ldots T$:
    - Compute gradient vector $g^{t-1} \leftarrow \text{calc\_grad}(\theta^{t-1})$
    - Update parameter vector $\theta^t \leftarrow \text{LBFGS\_update}(\theta^{t-1}, g^{t-1})$
- Unpack the final unconstrained params: $\rho, \mu, \nu \leftarrow \theta^T$
- **Return** final constrained parameters: $\pi = T^{-1}(\rho), \mu, \sigma = U^{-1}(\nu)$.

Naturally, you could check for some convergence criteria after every iteration, and terminate early.

Implementation Details for LBFGS

To implement LBFGS, you will need to implement the `calc_loss` function inside `fit`

After that however, your starter code will pretty much do the rest.

Gradients with autograd

Because we are using the `autograd` module, which provides automatic differentiation capability, you do NOT need to implement a `calc_grad` function by hand! Instead, if you have defined `calc_loss` correctly, you can just build a function to compute the gradient by calling: `calc_grad = autograd.grad(calc_loss)`. See the starter code!

If you have never used autograd before, it is pretty easy, just write numpy code as normal and autograd does the rest. You just make sure you use the `ag_np` module, imported like so `import autograd.numpy as ag_np`.

You will need to install this module into your `spr_2024s_env` environment.

For more help on autograd, see:

- The package README
- The package's tutorial TLDR
- The detailed set of "lab" exercises from Tufts COMP 135 (which you should only need the first few exercises from): https://github.com/tufts-ml-courses/comp135-20f-assignments/blob/master/labs/day23_AutogradForGradientDescent.ipynb

LBFGS with scipy.optimize.minimize

You should **not** implement the LBFGS update yourself. You should use `scipy.optimize.minimize`, with the setting `method=l-bfgs-b`. See starter code.

Transformations with starter code

You should also **not** implement the parameter transformation functions either. See starter code and use the transformations there.

Problem 2: LBFGS for GMMs

All of Problem 2 is optional in Spring '24. Recommended for your own learning, but not required for any grade.

In problem 2, we will implement L-BFGS gradient descent for training GMMs, using the penalized ML loss function above.

We will then examine how well this procedure trains models on a limited computational budget.

- Look at likelihoods of heldout data
- Interpret the learned components by inspecting learned weights, means and variances

Tasks for Code Implementation

Nothing required for Spring 2024

WRITING CODE 2(i) Implement the TODOs in the `calc_neg_log_lik` method of `calc_neg_log_lik__ag.py` file.

WRITING CODE 2(ii) Implement the TODOs in the `fit` method of the starter code class `GMM_PenalizedMLEstimator_LBFGS`, which define the overall loss used for gradient descent.

USING CODE 2(iii) Use the provided `train_LBFGS_gmm_with_many_runs.py`. This considers several values of K: 1, 4, 8, 16. For each value of K, initialize a GMM using the provided `generate_initial_parameters` function with a specified random seed, and train for *at most* 30 LBFGS iterations. Repeat for all 4 different random seeds provided in the starter code (1001, 3001, 4001, 7001).

USING CODE 2(iv) Among all training runs for K, select the single run that has the best validation likelihood and mark this as the "best" run.

We hope training all L-BFGS models should in total about ~15-30 minutes on your computer, assuming your code is correct.

Note: If you aren't careful about your code, run time can be much longer. If your code takes significantly longer than this please double check the tips above for fast code. If you still have trouble, please contact instructors asap via Piazza.

Tasks for Report PDF

Nothing required for Spring 2024

For your own learning, we suggest trying to compare LBFGS with EM by making figures/tables that correspond to the parts 1a, 1b, 1c from Problem 1.
