CSCI 567, Fall'18
Haipeng Luo

Homework #4 Programming Assignment

Due: 11:59 pm, November 4, 2018

General instructions

Your repository will have now a directory "P4/". Please do not change the name of this repository or the names of any files we have added to it. Please perform a `git pull` to retrieve these files. You will find within it: `baboon.tiff`, `data_loader.py`, `gmm.py`, `gmm.sh`, `gmmTest.py`, `kmeans.py`, `kmeans.sh`, `kmeansTest.py`, `utils.py`. Depending on your environment you may need to install the python library named, "pillow", which is used by matplotlib to process some of the images needed for this assignment. You do not need to import pillow in the files we provide, it only needs to be installed in your environment. You can install it by running 'pip install pillow' in your command line.

High Level Description

0.1 Tasks In this assignment you are asked to implement K-means clustering to identify main clusters in the data, use the discovered centroid of cluster for classification, and implement Gaussian Mixture Models to learn a generative model of the data. Specifically, you will

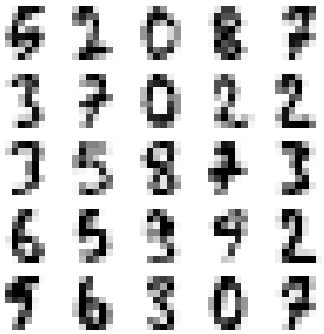
- Implement K-means clustering algorithm to identify clusters in a two-dimensional toy-dataset.
- Implement image compression using K-means clustering algorithm.
- Implement classification using the centroids identified by clustering.
- Implement Gaussian Mixture Models to learn a generative model and generate samples from a mixture distribution.

0.2 Running the code We have provided two scripts to run the code. Run `kmeans.sh` after you finish implementation of k-means clustering, classification and compression. Run `gmm.sh` after you finish implementing Gaussian Mixture Models.

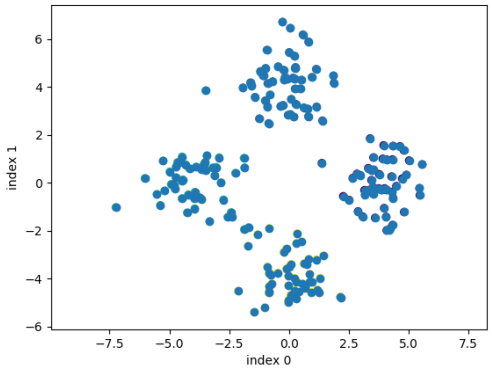
0.3 Dataset Through out the assignment we will use two datasets (See fig. 1) — Toy Dataset and Digits Dataset (you do not need to download).

Toy Dataset is a two-dimensional dataset generated from 4 Gaussian distributions. We will use this dataset to visualize the results of our algorithm in two dimensions.

We will use digits dataset from sklearn [1] to test K-means based classifier and generate digits using Gaussian Mixture model. Each data point is a 8×8 image of a digit. This is similar to MNIST but less complex.



(a) Digits



(b) 2-D Toy Dataset

Figure 1: Datasets

0.4 Cautions Please **DO NOT** import packages that are not listed in the provided code. Follow the instructions in each section strictly to code up your solutions. **Do not change the output format**. **Do not modify the code unless we instruct you to do so**. A homework solution that does not match the provided setup, such as format, name, initializations, etc., **will not** be graded. It is your responsibility to **make sure that your code runs with the provided commands and scripts on the VM**. Finally, make sure that you **git add, commit, and push all the required files**, including your code and generated output files.

0.5 Final submission After you have solved problem 1 and 2, execute `bash kmeans.sh` command and `bash gmm.sh` command. Git add, commit and push `plots` and `results` folder and all the *.py files.

Problem 1 K-means Clustering

Recall that for a dataset $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^D$, the K-means distortion objective is

$$J(\{\mu_k\}, \{r_{ik}\}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \|\mu_k - \mathbf{x}_i\|_2^2 \tag{1}$$

where $\mu_1, \ldots, \mu_K$ are centroids of the K clusters and $r_{ik} \in \{0, 1\}$ represents whether example i belongs to cluster k.

Clearly, fixing the centroids and minimizing J over the assignment give

$$\hat{r}_{ik} = \begin{cases} 1 & k = \arg\min_{k'} \|\mu_{k'} - \mathbf{x}_i\|_2^2 \\ 0 & \text{Otherwise.} \end{cases} \tag{2}$$

On the other hand, fixing the assignment and minimizing J over the centroids give

$$\hat{\mu}_k = \frac{\sum_{i=1}^{N} r_{ik} \mathbf{x}_i}{\sum_{i=1}^{N} r_{ik}} \tag{3}$$

What the K-means algorithm does is simply to alternate between these two steps. See Algorithm 1 for the pseudocode.

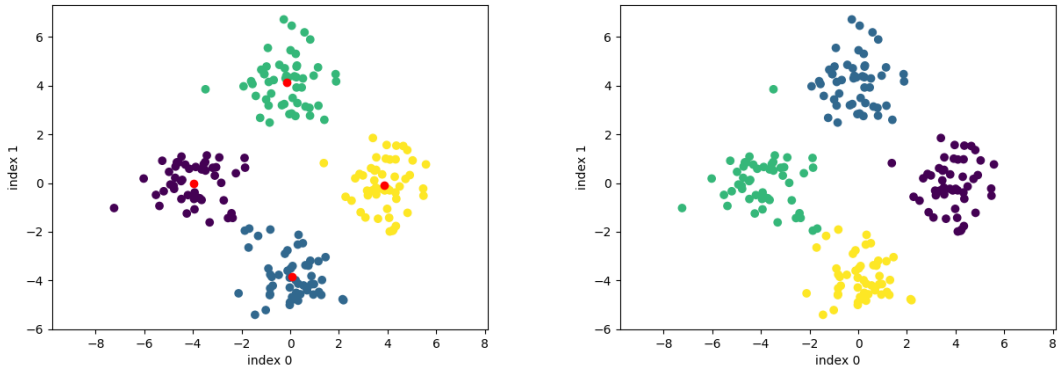
Algorithm 1 K-means clustering algorithm

Inputs:
An array of size $N \times D$ denoting the training set, `x`
Maximum number of iterations, `max_iter`
Number of clusters, `K`
Error tolerance, `e`
Outputs:
Array of size $K \times D$ of means, $\{\mu_k\}_{k=1}^{K}$
Membership vector `R` of size N, where $R[i] \in [K]$ is the index of the cluster that example i belongs to.
Initialize:
Set means $\{\mu_k\}_{k=1}^{K}$ to be `K` points selected from `x` uniformly at random (with replacement), and J to be a large number (e.g. 10^{10})
repeat
    Compute membership r_{ik} using eq. 2
    Compute distortion measure J_{new} using eq. 1
    if $|J - J_{new}| \leq e$ **then**
        STOP
    end if
    Set $J := J_{new}$
    Compute means μ_k using eq. 3
until maximum iteration is reached

1.1 Implementing K-means clustering algorithm Implement Algorithm 1 by filling out the TODO parts in class `KMeans` of file `kmeans.py`. Note the following:

- Use `numpy.random.choice` for the initialization step.
- If at some iteration, there exists a cluster k with no points assigned to it, then do not update the centroid of this cluster for this round.
- While assigning a sample to a cluster, if there's a tie (i.e. the sample is equidistant from two centroids), you should choose the one with smaller index (like what *numpy.argmin* does).

After you complete the implementation, execute `bash kmeans.sh` command to run k-means on toy dataset. You should be able to see three images generated in `plots` folder. In particular, you can see `toy_dataset_predicted_labels.png` and `toy_dataset_real_labels.png` and compare the clusters identified by the algorithm against the real clusters. Your implementation should be able to recover the correct clusters sufficiently well. Representative images are shown in fig. 2. Red dots are cluster centroids. Note that color coding of recovered clusters may not match that of correct clusters. This is due to mis-match in ordering of retrieved clusters and correct clusters (which is fine).



(a) Predicted Clusters (b) Real Clusters

Figure 2: Clustering on toy dataset

1.2 Image compression with K-means In the next part, we will look at lossy image compression as an application of clustering. The idea is simply to treat each pixel of an image as a point $\mathbf{x}_i$, then perform K-means algorithm to cluster these points, and finally replace each pixel with its centroid.

What you need to implement is to compress an image with K centroids given. Specifically, complete the function `transform_image` in the file `kmeansTest.py`.

After your implementation, execute `bash kmeans.sh` again and you should be able to see an image `baboon_compressed.png` in the `plots` folder. You can see that this image is distorted as compared to the original `baboon.tiff`.

1.3 Classification with k-means Another application of clustering is to obtain a faster version of the nearest neighbor algorithm. Recall that nearest neighbor evaluates the distance of a test sample from every training point to predict its class, which can be very slow. Instead, we can compress the entire training set to just the K centroids, where each centroid is now labeled as the majority class of the corresponding cluster. After this compression the prediction time of nearest neighbor is reduced from $O(N)$ to just $O(K)$ (see Algorithm 2 for the pseudocode).

Algorithm 2 Classification with K-means clustering

1: **Inputs:**
   Training Data : {X,Y}
   Parameters for running K-means clustering
2: **Training:**
   Run K-means clustering to find centroids and membership (reuse your code from Problem 1.1)
   Label each centroid with majority voting from its members. i.e. $\arg\max_c \sum_i r_{ik} \mathbb{I}\{y_i = c\}$
3: **Prediction:**
   Predict the same label as the nearest centroid (that is, 1-NN on centroids).

Note: 1) break ties in the same way as in previous problems; 2) if some centroid doesn't contain any point, set the label of this centroid as 0.

Complete the `fit` and `predict` function in `KMeansClassifier` in file `kmeans.py`. Once completed, run `kmeans.sh` to evaluate the classifier on a test set. For comparison, the script will also print accuracy of a logistic classifier and a nearest neighbor classifier. (Note: a naive K-means classifier may not do well but it can be an effective unsupervised method in a classification pipeline [2].)

Problem 2 Gaussian Mixture Model

Next you will implement Gaussian Mixture Model (GMM) for clustering and also generate data after learning the model. Recall the key steps of EM algorithm for learning GMMs on Slide 52 of Lec 8 (we change the notation ω_k to π_k):

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i ; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_k \pi_k \mathcal{N}(\mathbf{x}_i ; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \tag{4}$$

$$N_k = \sum_{i=1}^{N} \gamma_{ik} \tag{5}$$

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i}{N_k} \tag{6}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T}{N_k} \tag{7}$$

$$\pi_k = \frac{N_k}{N} \tag{8}$$

Algorithm 3 provides a more detailed pseudocode. Also recall the incomplete log-likelihood is

$$\sum_{n=1}^{N} \ln p(\mathbf{x}_n) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i ; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \frac{\pi_k}{\sqrt{(2\pi)^D |\boldsymbol{\Sigma}_k|}} \exp\left(-\frac{1}{2} (\mathbf{x}_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k) \right) \tag{9}$$

2.1 Implementing EM Implement EM algorithm (class `Gaussian_pdf`, function `fit` and function `compute_log_likelihood`) in file `gmm.py` to estimate mixture model parameters. Please note the following:

- For K-means initialization, the inputs of K-means are the same as those of EM's.
- When computing the density of a Gaussian with covariance matrix Σ, use $\Sigma' = \Sigma + 10^{-3} I$ **when** Σ is not invertible (in case it's still not invertible, keep adding $10^{-3} I$ until it is invertible).

After implementation execute `bash gmm.sh` command to estimate mixture parameters for toy dataset. You should see a Gaussian fitted to each cluster in the data. A representative image is shown in fig. 3. We evaluate both initialization methods and you should observe that initialization with K-means usually converges faster.

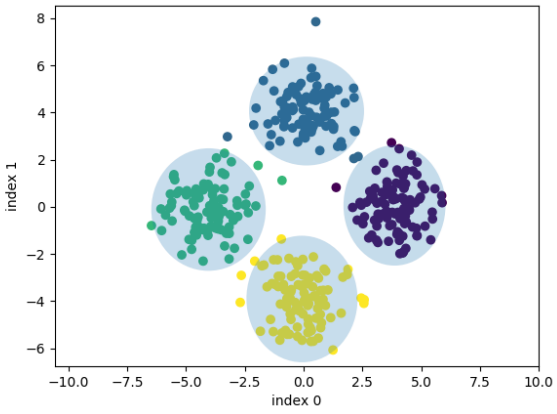


Figure 3: Gaussian Mixture model on toy dataset

Algorithm 3 EM algorithm for estimating GMM parameters

Inputs:
An array of size $N \times D$ denoting the training set, `x`
Maximum number of iterations, `max_iter`
Number of clusters, `K`
Error tolerance, `e`
Init method — K-means or random
Outputs:
Array of size $K \times D$ of means, $\{\mu_k\}_{k=1}^{K}$
Variance matrix Σ_k of size $K \times D \times D$
A vector of size K denoting the mixture weights, `pi_k`
Initialize:

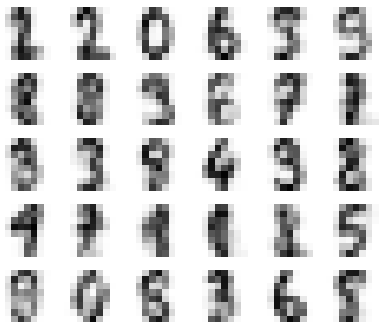
- For "random" init method: initialize means uniformly at random from [0,1) for each dimension (use `numpy.random.rand`), initialize variance to be identity matrix for each component, initialize mixture weight to be uniform.
- For "K-means" init method: run K-means, initialize means as the centroids found by K-means, and initialize variance and mixture weight according to Eq. (7) and Eq. (8) where γ_{ik} is the binary membership found by K-means.

Compute the log-likelihood l using Eq. (9)
repeat
  E Step: Compute responsibilities using Eq. (4)
  M Step:
    Estimate means using Eq. (6)
    Estimate variance using Eq. (7)
    Estimate mixture weight using Eq. (8)
  Compute new log-likelihood l_{new}
  if $|l - l_{new}| \leq e$ **then**
    STOP
  end if
  Set $l := l_{new}$
until maximum iteration is reached

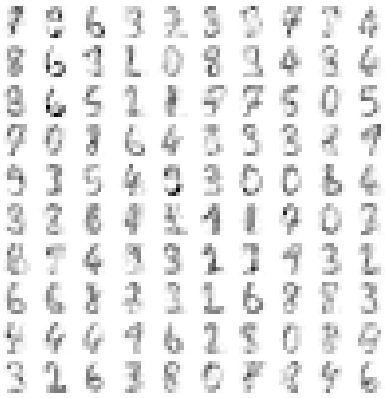
2.2 Implementing sampling We also fit a GMM with K = 30 using the digits dataset. An advantage of GMM compared to K-means is that we can sample from the learned distribution to generate new synthetic examples which look similar to the actual data.

To do this, implement `sample` function in `gmm.py` which uses `self.means`, `self.variances` and `self.pi_k` to generate digits. Recall that sampling from a GMM is a two step process: 1) first sample a component k according to the mixture weight; 2) then sample from a Gaussian distribution with mean μ_k and variance Σ_k. Use `numpy.random` for these sampling steps.

After implementation, execute `bash gmm.sh` again. This should produce visualization of means μ_k and some generated samples for the learned GMM. Representative images are shown in fig. 4.



(a) Means of GMM learnt on digits



(b) Random digits sample generated from GMM

Figure 4: Results on digits dataset

References

[1] sklearn.datasets.digits http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html#sklearn.datasets.load_digits

[2] Coates, A., & Ng, A. Y. (2012). Learning feature representations with k-means. In Neural networks: Tricks of the trade (pp. 561-580). Springer, Berlin, Heidelberg.