CSCI 567, Spring 2020
Yan Liu

Homework #3 Programming Assignments

High-level descriptions

The assignment is divided into two part.

Part 1

Dataset We will use **mnist_subset** (images of handwritten digits from 0 to 9). This is the same subset of the full MNIST that we used for Homework 1 and Homework 2. As before, the dataset is stored in a JSON-formated file **mnist_subset.json**. You can access its training, validation, and test splits using the keys 'train', 'valid', and 'test', respectively. For example, suppose we load **mnist_subset.json** to the variable x. Then, $x['train']$ refers to the training set of **mnist_subset**. This set is a list with two elements: $x['train'][0]$ containing the features of size N (samples) $\times D$ (dimension of features), and $x['train'][1]$ containing the corresponding labels of size N.

1. You will be asked to implement the boosting algorithms with decision stump (Sect. 1). Specifically, you will
   - finish implementing the following classes — `boosting`, `decision_stump` and `AdaBoost`. Refer to `boosting.py`, `decision_stump.py` and Sect. 1 for more information.
   - test and debug your code using the provided `boosting_check.py`.
   - run the script `boosting.sh` (which executes `boosting_test.py`). This will output `boosting.json`.
2. You will be asked to implement the decision tree classifier (Sect. 2). Specifically, you will
   - finish implementing the following classes — `DecisionTree`, `TreeNode`. Refer to `decision_tree.py` and Sect. 2 for more information.
   - test and debug your code using the provided `decision_tree_check.py`.
   - run the script `decision_tree.sh` (which executes `decision_tree_test.py`). This will output `decision_tree.json`.

As in the previous homework, you are not responsible for loading/pre-processing data; we have done that for you (e.g., we have pre-processed the data so it has two classes: digits 0–4 and digits 5–9.). For specific instructions, please refer to text in Sect 1 and the instructions `boosting.py`.

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**.

Problem 1 Boosting

In the lectures, we discussed boosting algorithms, which construct a strong (binary) classifier based on iteratively adding one weak (binary) classifier into it. A weak classifier is learned to (approximately) maximize the weighted training accuracy at the corresponding iteration, and the weight of each training example is updated after every iteration.

In this question, you will implement decision stumps as weak classifiers and AdaBoost. For simplicity, instead of implementing an actual base algorithm that learns decision stumps, we fix a finite set of decision stumps $\mathcal{H}$ in advance and pick the one with smallest weighted error at each round of boosting.

1.1 General boosting framework

A boosting algorithm sequentially learns β_t and $h_t \in \mathcal{H}$ for $t = 0, \ldots, T-1$ and finally constructs a strong classifier as $H(\mathbf{x}) = \text{sign}\left[\sum_{t=0}^{T-1} \beta_t h_t(\mathbf{x})\right]$.

Please read the class "Boosting" defined in `boosting.py`, and then complete the class by implementing the "TODO" part(s) as indicated in `boosting.py`.

1.2 Decision stump

A decision stump $h \in \mathcal{H}$ is a classifier characterized by a triplet $(s \in \{+1, -1\}, b \in \mathbb{R}, d \in \{0, 1, \cdots, D-1\})$ such that

$$h_{(s,b,d)}(\mathbf{x}) = \begin{cases} s, & \text{if } x_d > b, \\ -s, & \text{otherwise.} \end{cases} \tag{1}$$

That is, each decision stump function only looks at a single dimension x_d of the input vector $\mathbf{x}$, and check whether x_d is larger than b. Then s decides which label to predict if $x_d > b$.

Please first read `classifier.py` and `decision_stump.py`, and then complete the class "DecisionStump" by implementing the "TODO" part(s) as indicated in `decision_stump.py`.

1.3 AdaBoost

AdaBoost is a powerful and popular boosting method, summarized in Algorithm 1.

Algorithm 1 The AdaBoost algorithm

```
Require: $\mathcal{H}$: A set of classifiers, where $h \in \mathcal{H}$ takes a $D$-dimensional vector as input and outputs either $+1$
    or $-1$, a training set $\{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{+1, -1\})\}_{n=0}^{N-1}$, the total number of iterations T.
Ensure: Learn $H(\mathbf{x}) = \text{sign}\left[\sum_{t=0}^{T-1} \beta_t h_t(\mathbf{x})\right]$, where $h_t \in \mathcal{H}$ and $\beta_t \in \mathbb{R}$.
1: Initialization $D_0(n) = \frac{1}{N}, \forall n \in \{0, 1, \cdots, N-1\}$.
2: for $t = 0, 1, \cdots, T-1$ do
3:     find $h_t =_{h \in \mathcal{H}} \sum_n D_t(n) \mathbb{I}[y_n \neq h(\mathbf{x}_n)]$
4:     compute $\epsilon_t = \sum_n D_t(n) \mathbb{I}[y_n \neq h_t(\mathbf{x}_n)]$
5:     compute $\beta_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$
6:     compute $D_{t+1}(n) = \begin{cases} D_t(n)\exp(-\beta_t), & \text{if } y_n = h_t(\mathbf{x}_n) \\ D_t(n)\exp(\beta_t), & \text{otherwise} \end{cases}, \forall n \in \{0, 1, \cdots, N-1\}$
7:     normalize $D_{t+1}(n) \leftarrow \frac{D_{t+1}(n)}{\sum_{n'} D_{t+1}(n')}, \forall n \in \{0, 1, \cdots, N-1\}$
8: end for
```

Please read the class "AdaBoost" defined in `boosting.py`, and then complete the class by implementing the "TODO" part(s) as indicated in `boosting.py`.

1.4 Hint

You can match the results you are getting. It will be close to the one below.
AdaBoost testing accuracies: [1.0, 0.95, 0.95]

Problem 2 Decision Tree

In this question, you will implement a simple decision tree algorithm. For simplicity, only discrete features are considered here.

2.1 Conditional Entropy Recall that the quality of a split can be measured by the conditional entropy. Please read `decision_tree.py` and complete the function "conditional_entropy" (where we use base 2 for logarithm).

2.2 Tree construction The building of a decision tree involves growing and pruning. For simplicity, in this programming set you are only asked to grow a tree.

As discussed a tree can be constructed by recursively splitting the nodes if needed, as summarized in Algorithm 2 (whose interface is slightly different from the pseudocode discussed in the class). Please read `decision_tree.py` and complete the function "split".

Algorithm 2 The recursive procedure of decision tree algorithm

```
1: function TreeNode.split(self)
2:     find the best attribute to split the node
3:     split the node into child_nodes
4:     for child_node in child_nodes do
5:         if child_node.splittable then          ▷ See Lecture Slides for the three "non-splittable" cases
6:             child_node.split()
7:         end if
8:     end for
9: end function
```

2.3 Hint You can match the results you are getting. It will be close to the one below.
train_accu 0.80833, test_accu 0.8

Part 2

High Level Description

Tasks In this task 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

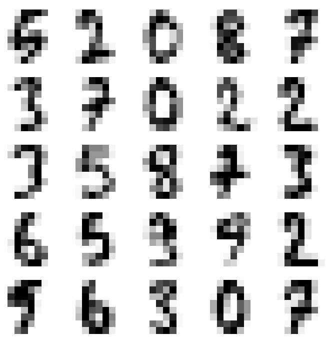
- Implement K-means clustering algorithm to identify clusters in a two-dimensional toy-dataset.
- Implement image compression using K-means clustering algorithm.
- Implement Gaussian Mixture Models to learn a generative model and generate samples from a mixture distribution.

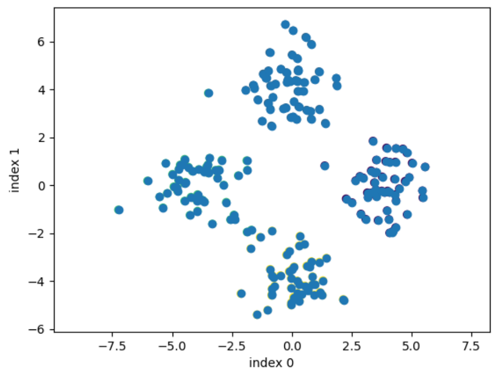
Running the code We have provided two scripts to run the code. Run `kmeans.sh` after you finish implementation of k-means clustering, classification and compression. Run `gmm.sh` after you finish implementing Gaussian Mixture Models.

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

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**.

Problem 3 K-means Clustering

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

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

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

$$\hat{r}_{ik} = \begin{cases} 1 & k =_{k'} \|\mu_{k'} - \mathbf{x}_i\|_2^2 \\ 0 & \text{Otherwise.} \end{cases} \tag{3}$$

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

$$\hat{\mu}_k = \frac{\sum_{i=1}^{N} r_{ik} \mathbf{x}_i}{\sum_{i=1}^{N} r_{ik}} \tag{4}$$

What the K-means algorithm does is simply to alternate between these two steps. See Algorithm 3 for the pseudocode.

Algorithm 3 K-means clustering algorithm

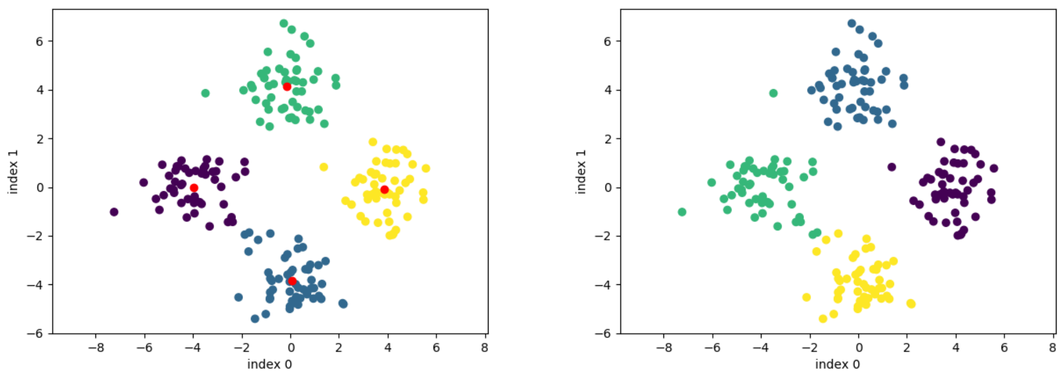
```
1: Inputs:
       An array of size $N \times D$ denoting the training set, x
       Maximum number of iterations, max_iter
       Number of clusters, K
       Error tolerance, e
2: Outputs:
       Array of size $K \times D$ of means, $\{\mu_k\}_{k=1}^K$
       Membership vector R of size N, where $R[i] \in [K]$ is the index of the cluster that
       example $i$ belongs to.
3: 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. $10^{10}$)
4: repeat
5:     Compute membership $r_{ik}$ using eq. 3
6:     Compute distortion measure $J_{new}$ using eq. 2
7:     if $|J - J_{new}| \leq e$ then
         STOP
8:     end if
9:     Set $J := J_{new}$
10:    Compute means $\mu_k$ using eq. 4
11: until maximum iteration is reached
```

3.1 Implementing K-means clustering algorithm Implement Algorithm 3 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

3.2 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`.

Problem 4 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 from Lecture (we change the notation ω_k to π_k):

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i\,;_k\,,_k)}{\sum_k \pi_k \mathcal{N}(\mathbf{x}_i\,;_k\,,_k)} \tag{5}$$

$$N_k = \sum_{i=1}^{N} \gamma_{ik} \tag{6}$$

$$_k = \frac{\sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i}{N_k} \tag{7}$$

$$_k = \frac{\sum_{i=1}^{N} \gamma_{ik}(\mathbf{x}_i - _k)(\mathbf{x}_i - _k)^T}{N_k} \tag{8}$$

$$\pi_k = \frac{N_k}{N} \tag{9}$$

Algorithm 4 provides a more detailed pseudocode. Also recall the incomplete log-likelihood is

$$\sum_{n=1}^{N} \ln p(\mathbf{x}_n) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i\,;_k\,,_k) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \frac{\pi_k}{\sqrt{(2\pi)^D |_k|}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - _k)_k^{T\,-1}(\mathbf{x}_i - _k)\right) \tag{10}$$

4.1 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.

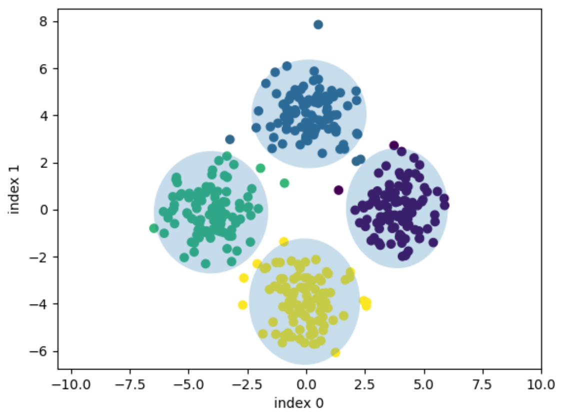


Figure 3: Gaussian Mixture model on toy dataset

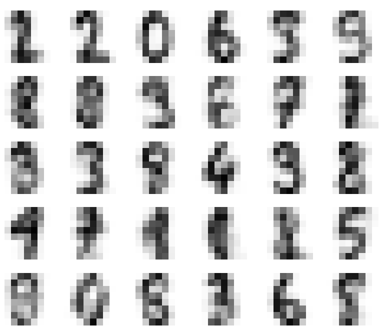
Algorithm 4 EM algorithm for estimating GMM parameters

```
1: 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
2: Outputs:
       Array of size $K \times D$ of means, $\{\mu_k\}_{k=1}^K$
       Variance matrix $\Sigma_k$ of size $K \times D \times D$
       A vector of size $K$ denoting the mixture weights, pi_k
3: 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. (8) and Eq. (9) where $\gamma_{ik}$ is the binary membership found by K-means.
4: Compute the log-likelihood $l$ using Eq. (10)
5: repeat
6:     E Step: Compute responsibilities using Eq. (5)
7:     M Step:
         Estimate means using Eq. (7)
         Estimate variance using Eq. (8)
         Estimate mixture weight using Eq. (9)
8:     Compute new log-likelihood $l_{new}$
9:     if $|l - l_{new}| \leq e$ then
         STOP
10:    end if
11:    Set $l := l_{new}$
12: until maximum iteration is reached
```

4.2 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

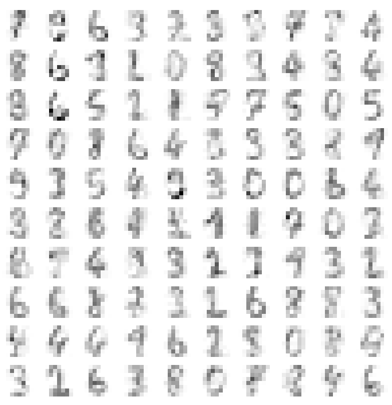

(b) Random digits sample generated from GMM

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.