Assigned: 23 September

Homework #3

EE 541: Fall 2024

Due: Friday, 04 October at 22:00. Submission instructions will follow separately on brightspace.

1. Unsupervised clustering algorithms are an efficient means to identify groups of related objects within large populations. Implement the following two clustering algorithms and apply them to the data in `cluster.txt`. The file contains data as: `x`, `y`, `class`. If necessary, use a regular expression to remove lines that are empty or that are invalid data. You may safely ignore any line that fails the regular expression.

   (a) Use K-Means clustering with 3-clusters to label each (x, y) pair as `Head`, `Ear_Right`, or `Ear_Left`. You may use any standard `NumPy` or `SciPy` packages or experiment with your own implementation.

   Produce a scatter plot marking each (x, y) pair as either BLUE (class = `Head`), RED (class = `Ear_Left`) or GREEN (class = `Ear_Right`). Compare the K-means predicted labels to the true label and generate a confusion matrix showing the respective accuracies.

   (b) Gaussian Mixture Models (GMM) are a common method to cluster data from multi-modal probability densities. Expectation maximization (EM) is an iterative procedure to compute (locally) optimal GMM parameters – GMM cluster means μ_k, covariances Σ_k, and mixing weights w_k. EM consists of two-steps. The **E**[xpectation]-step uses the mixture parameters to update estimates of hidden variables. The *true* but unknown class is an example of a hidden variable. The **M**[aximization]-step then uses the new hidden variable estimates to update the mixture parameter estimates. This back-and forth update provably increases the likelihood function and the estimate eventually converges to a local likelihood maximum. The GMM update equations follow.

   E-Step: use current mixture parameters estimates to calculate membership probabilities (*a.k.a.,* the hidden variables) for each sample, $\gamma_k(x_n)$,

$$\gamma_k(x_n) = \frac{w_k f(x_n; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} w_j f(x_n; \mu_j, \Sigma_j)}$$

   M-step: use new $\gamma_k(x_n)$ to update mixture parameter estimates,

$$\mu_k = \frac{\sum_{n=1}^{N} \gamma_k(x_n)\, x_n}{\sum_{n=1}^{N} \gamma_k(x_n)}$$

$$\Sigma_k = \frac{\sum_{n=1}^{N} \gamma_k(x_n)\,(x_n - \mu_k)\,(x_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma_k(x_n)}$$

$$w_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_k(x_n)$$

for $k \in \{1, \ldots, K\}$ where $K \in \mathbb{Z}^+$ is the (predefined) number of mixture components, x_n for $n \in \{1, \ldots, N\}$ are the data samples, and $f(x; \mu_k, \Sigma_k)$ is the d-dimensional jointly Gaussian pdf:

$$f(x; \mu_k, \Sigma_k) = \frac{\exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)}{\sqrt{(2\pi)^d |\Sigma_k|}}.$$

- Implement Expectation Maximization and use it to estimate mixture parameters for a 3-component GMM for the `cluster.csv` dataset. DO NOT use an EM implementation from `NumPy`, `SciPy`, or any other package. You must implement and use the above EM equations.

- Initialize $\gamma_k(x_n)$ for each sample using the K-means labels from part (a) as "one-hot" membership probabilities (*i.e.,* initialize one of the probabilities as "1" and all others are "0"). Then compute initial μ_k and Σ_k for each mixture component.

- Run EM until it has sufficiently converged. Use either the negative log-likelihood

$$\ell = \sum_{n=1}^{N} \log\left(\sum_{k=1}^{K} w_k f(x_n; \mu_k, \Sigma_k)\right) \tag{1}$$

  as a convergence metric or monitor the class assignments until there are only small changes. Be aware that some points may "flip-flop" even when fully converged. Assign each datapoint to the mixture component with the largest membership probability. Produce a Blue-Red-Green scatter plot as in part (a) and generate a confusion matrix showing the respective classification accuracies.

- Generate figures showing the class assignments during the first four iterations.

- Comment on the difference between the clustering result in (a) and (b). Describe any obvious difference between the plots and indicate which performs better.

2. The hd5 format can store multiple data objects in a single file each keyed by object name – *e.g.,* you can store a numpy float array called regressor and a numpy integer array called labels in the same file. Hd5 also allows fast non-sequential access to objects without scanning the entire file. This means you can efficiently access objects and data such as `x[idxs]` with non-consecutive indexes *e.g.,* `idxs = [2, 234, 512]`. This random-access property is useful when extracting a random subset from a larger training database.

   In this problem you will create an hd5 file containing a numpy array of binary random sequences that you generate yourself. Follow these steps:

   (1) Run the provided template python file (`random_binary_collection.py`). The script is set to `DEBUG` mode by default.

   (2) Experiment with the assert statements to trap errors and understand what they are doing by

using the shape method on numpy arrays, etc.

(3) Set the `DEBUG` flag to `False`. Manually create 10 binary sequences each with length 50. It is important that you do this by hand, *i.e.*, , **do not use a coin, computer, or random number generator**.

(4) Verify that your hd5 file was written properly by checking that it can be read-back.

(5) Submit your hd5 file as directed.

3. The MNIST dataset of handwritten digits is one of the earliest and most used datasets to benchmark machine learning classifiers. Each datapoint contains 784 input features – the pixel values from a 28×28 image – and belongs to one of 10 output classes – represented by the numbers 0-9.

   In this problem you will use numpy to program the feed-forward phase for a deep multilayer perceptron (MLP) neural network. We provide you with a pre-trained MLP using the MNIST dataset. The MLP has an input layer with 784-neurons, 2 hidden layers of 200 and 100 neurons, and a 10-neuron output layer. The model assumes ReLU activation for the hidden layers and softmax function in the output layer.

   (a) Extract the weights and biases of the pre-trained network from `mnist_network_params.hdf5`. The file has 6 keys corresponding to: `W1`, `b1`, `W2`, `b2`, `W3`, `b3`. Verify the dimension of each numpy array with the shape property.

   (b) The file `mnist_testdata.hdf5` contains 10,000 images. `xdata` holds pixel intensities and `ydata` contains the corresponding class labels. Extract these. Note: each image vector is 784-dimensions and the label is 10-dimensional with one-hot encoded, *i.e.,* label `[0,0,0,1,0,0,0,0,0,0]` means the image is class "3".

   (c) Write functions to calculate ReLU and softmax:

   - $\mathrm{ReLU}\,(x) = \max(0, x)$.
   - $\mathrm{Softmax}\,(x) = \left[\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}}\right]$.

     The softmax function takes a vector of size n and returns another vector of size n that you can interpret as a probability distribution. For example: `Softmax([0; 1; 2]) = [0:09; 0:24; 0:67]` so you can conclude that the 3rd element is the most likely outcome.

   (d) Use numpy to create an MLP to classify 784-dimensional images into the target 10-dimensional output. Use ReLU activation for the two hidden layers and softmax the output layer. Format and write your results to file `result.json` according to:

```
data = [
  {"index": XXX, "activations": [YYY,..., YYY], "classification": ZZZ},
  ...
]

import json
with open("result.json", "w") as f:
  f.write(json.dumps(data))
```

Output `index` and `activation` as integers (not string) and include only the 10 final-layer output activations as numeric floats.

(e) Compare your prediction with the (true) `ydata` label. Count the classification as correct if the position of the maximum element in your prediction matches with the position of the 1 in `ydata`. Tally the number of correctly classified images from the whole set of 10,000. [hint: 9790 correct].

(f) Identify and investigate several datapoints that your MLP classified correctly and several it classified incorrectly. Inspect them visually. Is the correct class obvious to you in the incorrect cases? Use `matplotlib` to visualize:

```
import matplotlib.pyplot as plt
plt.imshow(xdata[i].reshape(28,28))
plt.show()
# the index i selects which image to visualize
# xdata[i] is a 784x1 numpy array
```