CSEN 140L - Lab 8

Nolan Anderson

Lab 8 High-Level Objectives

- Conduct K-Means Clustering on the MNIST dataset
- Complete the tasks on the lab assignment PDF a single Jupyter notebook (.ipynb) file
- Use only the **NumPy** and **Matplotlib** libraries for your implementation
    - No machine learning libraries (except for the data retrieval part given by professor)
- Demo to me and submit your notebook to Camino before leaving the lab section

The Data Explained

- The MNIST dataset consists of 28 x 28 grayscale images of handwritten digits (0 – 9).
    - Since the images are 28 x 28, each image has 784 pixels.
        - Each pixel counts as a feature.
    - Because the dataset is quite large, we will just be using the first 2000 samples.
    - As such, your subset of the data should be (2000 x 784).

Clustering and K-Means

- Clustering is unsupervised machine learning algorithm.
  - Because of this, we don't have access to the true labels during the training.
- K-Means is an example of a clustering algorithm.
  - K-Means has a hyperparameter K, the number of clusters.
    - In this case, since we know there are 10 types of digits, we require you to set $K = 10$, but strictly speaking the value of K is your choice when clustering.

K-Means Algorithm

- Initialize K centroids
  - In our case, we are going to randomly select *K* samples from our dataset to be our initial centroids.
- Optimize the clusters
  - Calculate the distance of all samples to the K centroids
  - Assign samples to the closest centroids
  - Calculate a new centroid for each of the K clusters
  - Repeat the above optimization steps until convergence
    - Convergence can be defined in terms of cluster assignments (i.e. no changes in cluster assignments) or centroids (i.e. no changes in the centroids)

Euclidean Distance

- When we refer to distance in the K-Means Algorithm, we are talking about euclidean distance:

$$\sum_{1}^{N}[(X_i - c)^2]$$

  - **X_i** is the ith sample in the dataset (N total)
  - **c** is a centroid

K-Means Loss Function

- For clustering, we can use the sum of squared error (SSE) as our loss function.
- For each cluster, SSE is computed as follows:

$$SSE = \sum_{1}^{N} (X_i - \bar{X})^2$$

  - **X_i** = ith sample of the cluster
  - **X_bar** = the cluster's centroid

- The total SSE for your clustering results is simply the sum of the SSE values from each cluster.

General Code Structure

- def my_kmeans(X, K, M):
    - // outermost loop cycles over different initializations
    - for m in range(M):
        - centroids = randomly selected each time
        - // actual kmeans loop below
        - while (not converged):
            - for k in range(K):
                - compute distance between samples X and centroid[k]
            - assign observations to clusters (use argmin)
            - for k in range(K):
                - recompute centroid[k] by taking the mean of each cluster
            - Check convergence and stop loop if converged

Tips

- Use a fixed random seed in your implementation to allow for repeatability
    - See the next slide for a suggestion of how to do this
- Use the *plot_centroids* function provided in the Files to plot your final centroids to see if they are reasonable
    - I have provided two sample centroids in that file so you can see what the plot is supposed to look like
- Don’t forget that you are required to use **K** = 10 and **M** = 15

Helpful NumPy Functions

- rng = np.random.default_rng(*SEED*)
  - Allows you to instantiate a random number generator with a fixed seed (*SEED*)
  - rng has a method *choice* that allows you to randomly select a subset of a given set (useful for initial centroid selection)
- np.argmin()
  - Useful to find cluster assignments
- np.where()
  - Useful to select samples belonging to a particular cluster
- np.mean()
  - Useful for recomputing the centroids