COGS 118B Project 1 Report

Introduction

In this project, our group's ultimate aim is to implement an unsupervised learning algorithm and some methods including a multi-dimensional Bernoulli random variable, KMeans, and a mixture of Bernoulli distributions on our dataset. We are using 'MNIST' as our primary dataset. MNIST is one of the datasets that contains 28 x 28 pixel grayscale images of handwritten single digits between 0 and 9. This dataset is one of the most widely used datasets from the widespread machine learning library, TensorFlow by Google.

Methods

In section 1, we will implement a multi dimensional Bernoulli random variable about the given dataset, Mnist. To fit a Bernoulli r.v., the code will get a Maximum Likelihood Estimation of the parameter for the Bernoulli distribution. First of all, let the theta be the mean of the pixels of the images. To get the j th component of the theta of MLE, we are using a for loop through each pixel across each image, then setting the mean(mu) zero. In the nested for loop, we are getting the sum of the j th pixel value across the images(across the train_set), then saving it as a mean. We divide the mean by the total number of the data at the end of the loop. Once the loop is done, the MLE estimation is visualized by the .reshape method that is imported from numpy. Finally, we are importing matplotlib.pyplot: plt.imshow() creates an image from a two-dimensional numpy array.

In section 2, we are clustering the images by applying the K-means to the Mnist dataset. To implement the cluster of the dataset with K-means, we are writing the functions calcSqDistances, determineRnk, recalcMus, runKMeans. This was heavily influenced by previous K-means implementations done in homework 3 and 4. In the function calcSqDistances(X, Kmus), we store the dataset information into N and D at 'N,D = X.shape'. Then 'K = Kmus.shape[0]' stores the dataset information into K. 'sq_dis = np.zeros((N,K), dtype = np.float32)' stores the shape N by K with zeros as a type float into sq_dis. 'nested for loop' iterates a sequence through the i_th to N_th row and j_th to K_th column. The loop stores the matrix norm. In the function determineRnk(sqDmat), first of all, 'm = np.argmin(sqDmat, axis = 1)' returns the factor of the minimum values of sqDmat that is the input array along the axis = 1 and stores it into m. Then 'np.eye(sqDmat.shape[1])[m]' returns the two dimensional array times the factor m. Function recalcMus(X, Rnk) returns the recalculation of the mean. In the function runKMeans(K), 'rand_inds = np.random.permutation(N)' and 'Kmus = X[rand_inds[0:K], :]'

initialize the cluster centers by randomly permuting points from the data as a sequence. ‘for loop’ iterates the range of maxiters = 1000 and bring the calcSqDistances(X, Kmus) to store it into a sqDmat which will be N by K matrix. The loop also brings the determineRnk function and takes the parameter sqDmat to store it into Rnk. It will be an N by K matrix of binary values. The conditional if which is inside the loop breaks if the cluster centers have converged. Finally, the function runKmeans returns the Kmus and obtains the locations of the clusters K=10 and K=20. The function get_cluster_plot(Kmus, row = 2) obtains how the plot of the result for 10 and 20 clusters is represented differently when it takes the cluster location matrix as an input.

In Section 3, we are implementing the mixture of Bernoulli distributions about K-means. Function train_EM_MOB takes four input arguments that are x as a dataset, pi as an initial mixture probabilities, theta as an initialization of the cluster locations and K=10 as a number of mixtures. According to the EM algorithm for the mixing Bernoulli distributions, we are computing gamma_ik where i ∈ {1, 2, …, N} and k ∈ {1, 2, …, 10} first. First of all, an initial calculation of responsibilities (gamma) is done in the E step. This is done effectively by optimally multiplying the powers of theta and x. With the initial responsibilities obtained, we can then proceed to the M step by summing the responsibilities to calculate a new pi (pi_new) and a new theta (theta_new). Again, it loops and the E step is calculated by calcRespons(x, pi_new, theta_new) and saves it into a variable respons. Once the change in parameters is negligible(if numpy.linalg.norm(change of theta) is smaller than 1e-4), break the condition and update the old theta as the new theta. Finally, the function train_EM_MOB returns updated new pi and new theta.

Results

Finally, we obtain the three different figures from a single Bernoulli (Figure 1), a K-means (Figure 2), and a mixture of Bernoulli (Figure 3). In Figure 1, it is evident that a single Bernoulli performs very poorly in terms of clustering. Its limitation meant that we were essentially taking the mean of each pixel from all images, resulting in a single “cloudy” image without much meaning. In Figure 2 (K=10 and K=20), we can see a much better improvement with K-means as we are now able to assign each image to a particular cluster. With each increase in K-value, we can see a noticeable improvement in the distinction and accuracy between each digit. In Figure 3, we can see similar results to K-means using a Mixture-of-Bernoulli implementation. Like K-means with a cluster count of 10, the digit images are a bit blurry but nonetheless distinctive and recognizable.

Discussion

Through the project our team found that translating mathematical equations and probability theory into code is harder than expected. We learned how to implement a mixture of Bernoulli distributions using Python code. With this implementation, we were also able to see how the Bernoulli distribution differs from the Gaussian distribution both mathematically and code-wise. We also found that the initialization of parameters (theta and pi) indeed have an impact on the final result. With further testing, we could possibly discover an optimal initialization of these parameters (uniform vs. Kmeans initialization). Likewise, with more time and testing, we would also be able to tune certain hyperparameters, such as max iterations and convergence value. Also, we realized that K-means provide a good predictor of cluster quality, compared to a multi-dimensional Bernoulli random variable. Especially on the MNIST dataset, we would like to say K-means is one of the most efficient and intuitive approaches to clustering data.

Some difficulties encountered during the project included translating the equations into code. Particularly, the E-step in the expectation-maximization of the mixture of Bernoulli distribution was difficult because of the various indices and subscripts in the equation. Fortunately, consulting the textbook and discussions were extremely helpful for that task. Another difficulty was optimizing and vectorizing our code. It was intuitively easier to think of the series of addition and multiplication in terms of for loops so having to stray away from nested for loops made it a little more difficult to code. This is especially since none of our group members have had ample experience in writing “optimized” Python code. Luckily, discussion proved to be very helpful and shed light on ways to optimize. Despite the difficulties, we feel the project was educational and helped solidify our understanding of course content.

Figure 1: image for single-Bernoulli

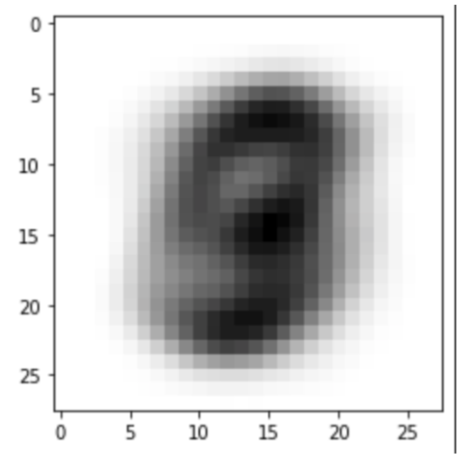


Figure 2 : image for K-means

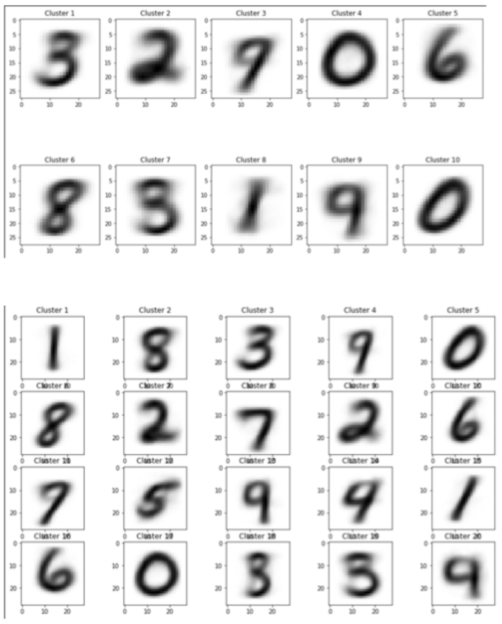
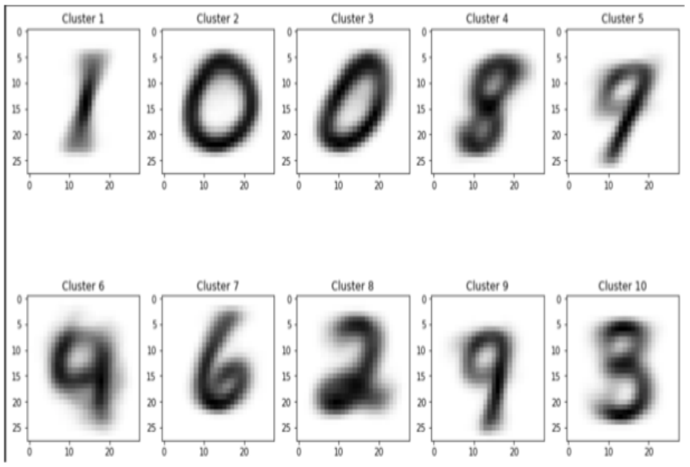


Figure 3 : image for Mixture-of-Bernoulli


Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
Cluster 7
Cluster 8
Cluster 9
Cluster 10