Clustering analysis of MNIST Fashion Dataset

Priya Rao
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
prao4@buffalo.edu

Abstract

This paper describes a clustering analysis performed by means of both K-Means – without the use of an autoencoder and with the usage of an autoencoder – and Gaussian Mixture Model (GMM). The corpus dataset being used is that of the MNIST fashion dataset and the number of target vectors will be unknown since this experiment is being treated as an unsupervised prototype.

1 Introduction

K-means clustering in simple terms is a method of partitioning the n data into a certain number of k clusters wherein each data belongs to the cluster with the nearest distance from the centroid of that cluster. Although this computation is NP-hard, there are algorithms to achieve convergence to a local optimum. In this experiment we look at mainly two methods of clustering algorithms – K-Means and GMM. The primary difference between the two is that the former tends to find clusters of a certain comparable spatial extent while the latter tends to follow the expectation-maximization approach which allows the clusters to take up different shapes.

The algorithm has a loose relationship to the k-nearest neighbor classifier, a popular machine learning technique for classification that is often confused with k-means due to the name. Applying the 1-nearest neighbor classifier to the cluster centers obtained by k-means classifies new data into the existing clusters. This is known as nearest centroid classifier or Rocchio algorithm. First part of this experiment deals with the naïve k-means clustering while the second part deals with the usage of autoencoder to enhance the performance and minimize losses from the naïve k-means. The third part of this experiment explores GMM alongside the constructed autoencoder.

1.1 Part 1 – K-Means

In this clustering method, given the training corpus, the goal is to partition the data such that it is grouped into a few cohesive clusters. Since this is an unsupervised learning model, we are given only the x feature vectors. And from these 'k' centroids are to be predicted. The first step would be to initialize these cluster centroids randomly. The next step would be to adjust the centroid until the centroids no longer change their positions. This algorithm also aims to minimize the sum of squared errors.

$$J(V) = \sum_{i=1}^{c} \sum_{j=1}^{c_i} \left(\|x_i - v_j\| \right)^2$$

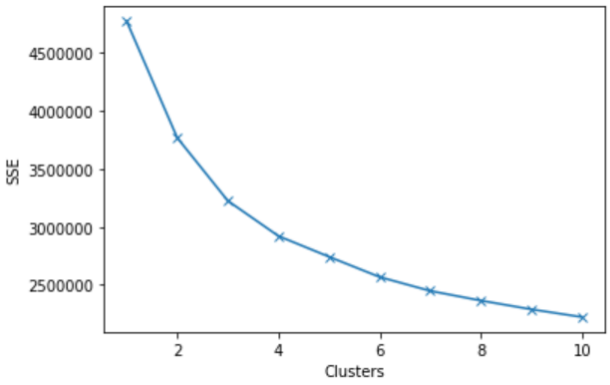
where,

'$||x_i - v_j||$' is the Euclidean distance between x_i and v_j.

'c_i' is the number of data points in i^{th} cluster.

'c' is the number of cluster centers.

When using K-means, one of the things to be done is to make sure the optimal number of clusters is chosen. Too little and data with significant differences will be grouped together. Too many clusters and it would contribute to overfitting the data. To mitigate this, we use the *elbow method* which is a common technique used for this task. It involves estimating the model using various numbers of clusters and calculating the negative of the *within-cluster sum of squares* for each number of clusters chosen using the score method from *sklearn*. For the MNIST fashion data set that we have, if it was run for 10 clusters, we obtain the graph as follows:



Here it would seem that 5 or 7 could be the elbow point since the curve is linear at and after that point. But since we are given as part of the problem statement that 10 clusters are to be considered, this method would not be necessary.

1.2 Part 2 – K-Means with AutoEncoder

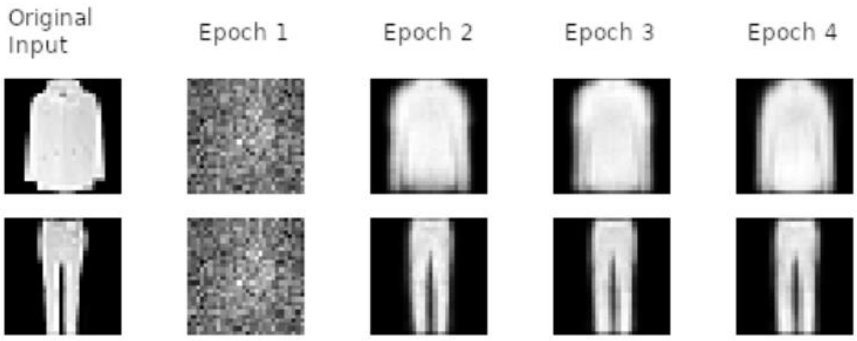
An autoencoder is an unsupervised ANN that learns how it can efficiently compress and encode a given data and also how to reconstruct this data back from the reduced output of the encoding. As such, it effectively learns to discard the noise in the data thereby reducing the data dimensions. An autoencoder is comprised of two parts – an encoder and a decoder. The encoder compresses the input into a latent-space representation given as:

$$h = f(x)$$

where x is the input image and h is the latent space representation post application of this function. The decoder on the other hand, tries to reconstruct the original image from the output generated by the encoder (latent space representation). This can be depicted as:

$$r = g(h)$$

where h is the latent representation from the encoder and r is the reconstructed output. The goal of an autoencoder is to achieve a highest possible similarity of this reconstructed output to the input image.
An example of this for the given fashion MNIST dataset is given below for 4 epochs:



As visible, as the number of epochs grow the autoencoder learns to distinguish the noise from the actual image and the output is seemingly closer to the original input while simultaneously minimizing the losses.

Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder but are also trained to make the new representation have various nice properties. Different kinds of autoencoders aim to achieve different kinds of properties. The four different types of autoencoders are:

- Vanilla autoencoder: In its simplest form, the autoencoder is a three layers net, i.e. a neural net with one hidden layer.

- Multilayer autoencoder: If one hidden layer is not enough, we can obviously extend the autoencoder to more hidden layers.

- Convolutional autoencoder: Autoencoders can be used with convolutions instead of Fully-connected layers since the principle is the same but using images (3D vectors) instead of flattened 1D vectors. The input image is down sampled to give a latent representation of smaller dimensions and force the autoencoder to learn a compressed version of the images.

- Regularized autoencoder: There are other ways we can constraint the reconstruction of an autoencoder than to impose a hidden layer of smaller dimension than the input. Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

1.3 Part 3 – GMM with AutoEncoder

A *Gaussian Mixture* is a function that is comprised of several Gaussians, each identified by $k \in \{1,\ldots, K\}$, where K is the number of clusters of our dataset. Each Gaussian k in the mixture is comprised of the following parameters:

- A mean μ that defines its center.

- A covariance Σ that defines its width. This would be equivalent to the dimensions of an ellipsoid in a multivariate scenario.

- A mixing probability π that defines how big or small the Gaussian function will be.

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the

covariance structure of the data as well as the centers of the latent Gaussians. The GaussianMixture object implements the expectation-maximization (EM) algorithm for fitting mixture-of-Gaussian models. It can also draw confidence ellipsoids for multivariate models and compute the Bayesian Information Criterion to assess the number of clusters in the data.

In EM, model parameters are randomly initialized. Then we alternate between (E) assigning values to hidden variables, based on parameters and (M) computing parameters based on fully observed data.

E-Step

Coming up with values to hidden variables, based on parameters. In order to choose the best values for the class variable based on the features of a given piece of data in the given corpus, we can see that for each data-point, that centroid is chosen which it is closest to, by method of calculating the Euclidean distance, and is assigned that centroid's label.

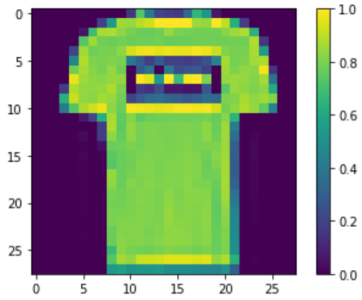
M-Step

Coming up with parameters, based on full assignments. In order to choose the best parameter values based on the features of a given piece of data in the given corpus, we can see that this can be done by taking the mean of all the data-points which were labeled as c.

2 Dataset Definition

The dataset used for this experiment was the Fashion-MNIST is a dataset of Zalando's article images, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each
pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The below figure shows one row of the training dataset:

```
plt.figure()
plt.imshow(train_images[1])
plt.colorbar()
plt.grid(False)
plt.show()
```



The dataset was split as training, validation and testing as Training_Data [6000x28x28], Validation_Data [5000x28x28] and Test_Data[5000x28x28]. All three methods employ the same data split in order to achieve an even training of all three models and draw definitive conclusions for the predictions and accuracy. The data was fetched using the util_mnist_reader.py file provided for all three parts of this experiment.

3 Preprocessing

As part of data preprocessing, there were two crucial phases involved in all three parts of this experiment:

Phase 1: Data extraction

The data is extracted from the given Fashion-MNIST training and testing database as 60000 for test data and 5000 for validation data and 5000 for testing data:

```
from google.colab import drive
drive.mount('/content/drive')

X_train, y_train = load_mnist('/content/drive/My Drive/Colab Notebooks/data/fashion', kind='train')
X_data, y_data = load_mnist('/content/drive/My Drive/Colab Notebooks/data/fashion', kind='t10k')
```

The shuffle flag is set to False to have a uniform set of data for all three methods involved in this experiment. The input data was then reshaped if needed.

Phase 2: Data normalization

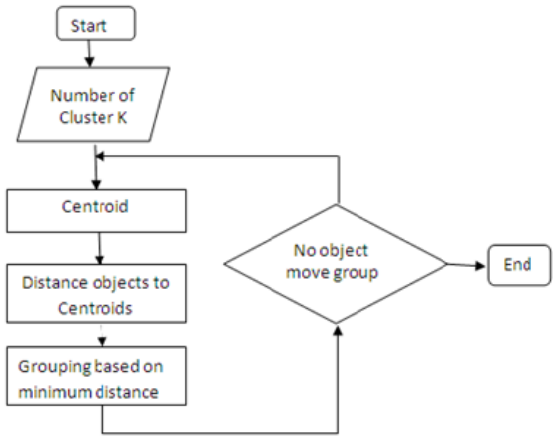
Each of the input vectors were normalized by dividing by 255 since the value of a pixel can range from 0 to 255.

For K-Means (part 1) alone however, an additional step that is being performed is that the dataset taken is a combined one, i.e., the Training and the Testing data are concatenated to obtain the maximum accuracy possible.

4 Architecture

4.1 Part 1 – K-Means

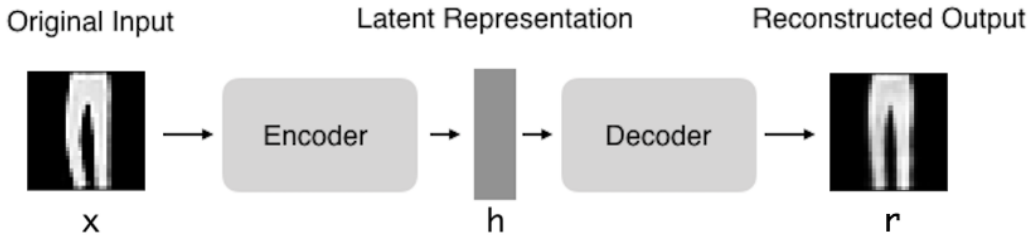
The K-Means architecture simply follows the below flowchart. The steps are listed below:



1. Choose a value of k, number of clusters to be formed.
2. Randomly select k data points from the data set as the initial cluster centroids/centers
3. For each datapoint:
    a. Compute the distance between the datapoint and the cluster centroid
    b. Assign the datapoint to the closest centroid
4. For each cluster calculate the new mean based on the datapoints in the cluster.
5. Repeat 3 & 4 steps until mean of the clusters stops changing or maximum number of iterations reached.

4.2 Part 2 – K-Means with AutoEncoder

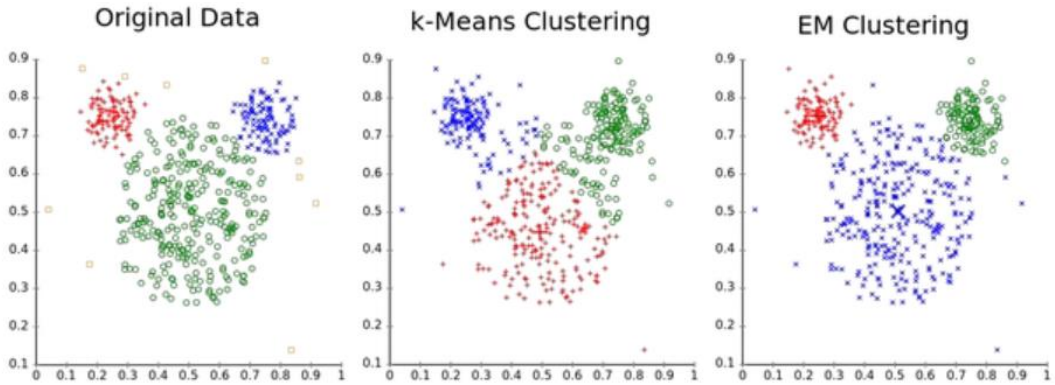
Autoencoder is a data compression algorithm where there are two major parts, encoder, and decoder. The encoder's job is to compress the input data to lower dimensional features. For example, one sample of the 28x28 MNIST image has 784 pixels in total, the encoder we built can compress it to an array with only ten floating point numbers also known as the features of an image. The decoder part, on the other hand, takes the compressed features as input and reconstruct an image as close to the original image as possible. Autoencoder is unsupervised learning algorithm in nature since during training it takes only the images themselves and not need labels. An example of this architecture is demonstrated below where the original image is sent to the encoder producing its latent representation while the decoder in turn performs the opposite by reconstructing the image from the latent representation.



Architecture of an Autoencoder

4.3 Part 3 – GMM with AutoEncoder

GMM works similar to K-Means in the sense that for a given clustering problem, it would group together the closest data points. But the primary difference between K-Means and GMM method of clustering is in K-Means, it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster as shown below in the second picture. An issue would arise when the data wouldn't cluster in a circular form but instead would require a "soft" grouping. This is where GMM's EM algorithm renders a better result since it would allow for such accomodations.



The E-M algorithm takes into account two steps:

E-step: compute

$$\mathrm{E}_{z|x,\theta^{(t)}}\left[\log\left(p(\mathbf{x},\mathbf{z}\mid\theta)\right)\right]=\sum_{\mathbf{z}}\log\left(p(\mathbf{x},\mathbf{z}\mid\theta)\right)p\left(\mathbf{z}\mid\mathbf{x},\theta^{(t)}\right)$$

M-step: solve

$$\theta^{(t+1)}=\underset{\theta}{\operatorname{argmax}}\sum_{\mathbf{z}}\log\left(p(\mathbf{x},\mathbf{z}\mid\theta)\right)p\left(\mathbf{z}\mid\mathbf{x},\theta^{(t)}\right)$$

5 Results

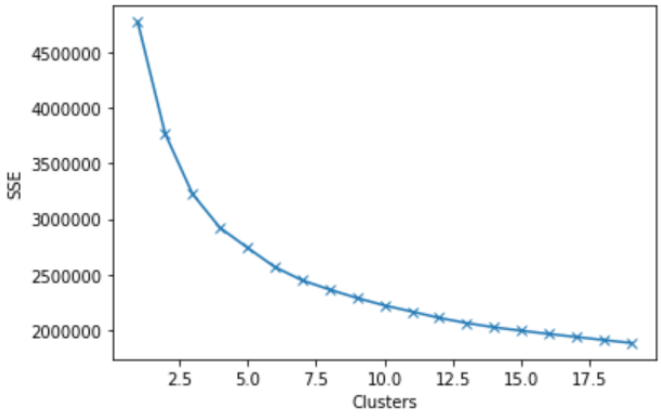
The results of the experiment were drawn based on tuning of the hyperparameters as well as using different regularization techniques and activation functions. The loss function was plotted against the epoch for both the validation dataset and the training dataset. The metrics for this experiment were calculated using the confusion matrix and the *CoClust* accuracy.

The confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. It not only allows the visualization of the performance of an algorithm but also allows for easy identification of confusion between classes e.g. one class is commonly mislabeled as the other. Most performance measures are computed from the confusion matrix. The name stems from the fact that it is easy to identify if the system is confusing two classes. The diagonal represents the true positives achieved by the model.

In this experiment the *coclust.evaluation.external* module is being used which basically provides functions to evaluate clustering or co-clustering results with external information such as the true labeling of the clusters. The method accuracy gives the best accuracy for clustering or co-clustering results.

5.1 Part 1 – K-Means

Upon plotting the loss versus clusters of the naïve K-Means model, there is an accuracy of 53.98% percent achieved. However, the iterative solution to first demonstrating the elbow point deduction from 10 clusters (mentioned as part of the introduction) produced 5 or 7 clusters as an optimal. However, upon increasing the number of clusters as shown below for 20 clusters, we can see that around 8-10 shows a linearity in the data, which means these would be the ideal number of clusters in order to categorize the data.



Sum of Squared Errors (kmeans.inertia_) versus number of clusters

The SSE was calculated using the kmeans.inertia_ which gives the sum of squared distances of samples to their closest cluster center.

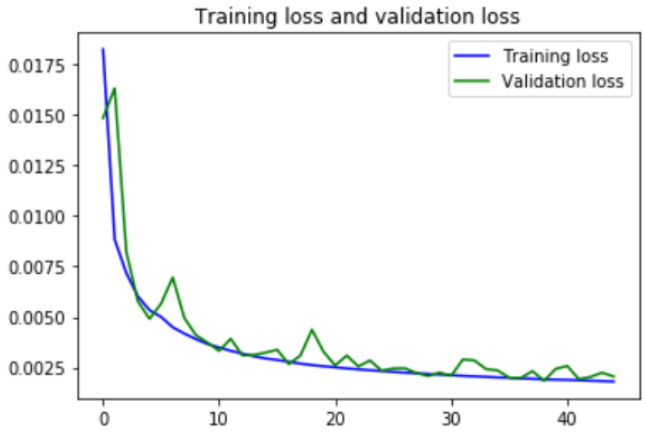
```
SSE for clusters =  10 is 2233496.1077513914
Accuracy for K-Means :  0.5398571428571428
```

The accuracy obtained is explained by the following confusion matrix:

```
0.5398571428571428
[[4001  199  208   23   47  508 1977    3   34    0]
 [ 285 6298   30    2   55  148  182    0    0    0]
 [  98   12 2168   23 2178  473 2020    1   27    0]
 [1987 3636   20    5   66  487  787    0   11    1]
 [ 817  173 1096   15 3561  230 1077    0   31    0]
 [   3    1    0   13    0 4680   58 1672   12  561]
 [1224   61 1071   56 1404  729 2428    6   19    2]
 [   0    0    0    6    0  724    1 5929    0  340]
 [  24   24  346 2508   45  496  379  333 2837    8]
 [   8    3    2    7   11  235   69  777    5 5883]]
```

5.2 Part 2 – K-Means with AutoEncoder

The AutoEncoder being used in this experiment is that of a Convolutional AutoEncoder and utilizes the ReLU and sigmoid activation functions which have been described as part of the architecture. The hyper parameters here would be the number of iterations as well as the number of clusters, but since we are already given the number of clusters as part of the experiment this would not be explored.



Training losses and validation losses versus the number of epochs

```
[[284   0   6  51   5   0 137   0  10   0]
 [  0 429   2  54   7   0  26   0   1   0]
 [ 13   0 164   3 130   0 159   0  10   0]
 [  8  18   0 334  10   0 129   0   1   0]
 [  1   1 125  47 229   0  73   0   3   0]
 [  0   0   0   0   0 121  31 261   1 101]
 [ 62   0  87  48  89   0 206   0  26   0]
 [  0   0   0   0   0   1   0 408   1  90]
 [  0   0   8   5   5   0  93   6 356   1]
 [  0   0   2   0   1 211   0  11   1 297]]
```

Confusion Matrix for K-Means with AutoEncoder

An accuracy of 56.56% was achieved using 45 epochs as the hyperparameter setting.

Obtaining the accuracy of the KMeans + AutoEncoder Model using CoClust

```
[ ]  print ("CoClust Accuracy: ")
     accuracy(y_test, clustered_training_set)
```

```
CoClust Accuracy:
/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_assig
  DeprecationWarning)
0.5656
```

5.3 Part 3 – GMM with AutoEncoder

The confusion matrix generated for the GMM + AutoEncoder model is as follows:

```
[[285   1   0  54   4   1 139   0   9   0]
 [  0 433   0  48   7   1  29   0   1   0]
 [ 25   0   0   6 273   0 162   0  13   0]
 [  3  80   0 274   7   0 135   0   1   0]
 [  2   1   0  63 331   0  78   0   4   0]
 [  0   0  27   0   0 368   6  87   1  26]
 [ 58   2   0  52 171   2 208   0  25   0]
 [  0   0   1   0   0  41   0 380   1  77]
 [  1   0   0   3  10  17  87   4 352   0]
 [  0   0 182   0   1  19   1  12   1 307]]
```

The accuracy for this model obtained was 58.76% which is higher than both models showing that this model fares better in case of this dataset which further confirms the clustering done as part of this model.

Obtain the accuracy of the GMM + AutoEncoder Model using CoClust

```
[ ]  accuracy(y_test,labels_gmm)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/linear_
  DeprecationWarning)
0.5876
```

6 Conclusion

Through the three models, altering the number of nodes, hidden layers, activation functions being used, regularization techniques, epochs and the learning rate, alters the results giving rise to different accuracies and predictions. In the case of part 1 of this experiment, an accuracy of 53.98% was achieved within 100 epochs. In the case of part 2 of this experiment, where the autoencoder was included an accuracy of 56.56% within 45 epochs. In the final case of part 3, an accuracy of 58.76% was achieved in the GMM model within roughly 300 epochs.

References

[1] Wikipedia – K-Means Clustering https://en.wikipedia.org/wiki/K-means_clustering

[2] Christopher M. Bishop ©2006 Pattern Recognition and Machine Learning. Springer-Verlag Berlin, Heidelberg

[3] Medium – AutoEncoder https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726

[4] KMeans [Stanford Lecture Notes] – https://stanford.edu/~cpiech/cs221/handouts/kmeans.html

[5] Towards Data Science - https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f

[6] Programmer Sought – http://www.programmersought.com/article/911154478/

[7] Geeks for geeks – https://www.geeksforgeeks.org/confusion-matrix-machine-learning/

[8] CoClust – https://coclust.readthedocs.io/en/v0.2.1/api/evaluation.html

[9] Wikipedia – Expectation Maximization Gaussian Mixture – https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm#Gaussian_mixture

[10] https://stats.stackexchange.com/questions/212358/em-algorithm-e-step

[11] https://www.kaggle.com/s00100624/digit-image-clustering-via-autoencoder-kmeans/notebook

[12] https://smorbieu.gitlab.io/accuracy-from-classification-to-clustering-evaluation/

[13] https://www.dlology.com/blog/how-to-do-unsupervised-clustering-with-keras/