CSE574 Intro to Machine Learning Project 3

Utkarsh Behre
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
ubehre@buffalo.edu

Abstract

In this project, we were asked to perform cluster analysis using KMeans clustering algorithm, Autoencoder based Kmeans clustering and Gaussian Mixture Model clustering models on the given Fashion-MNIST clothing images. We were expected to train the unsupervised model so that it can identify an image as one of the many clusters. After performing these tasks on the given dataset, herein lies my final report.

1 Introduction

In This project I will be using unsupervised learning, specifically clustering algorithms, to put different images in different clusters. Unsupervised learning deals with the data for which we don't have labels present. This means we can't rely on our normal classification techniques using neural networks. Here, we make use of several clustering algorithms in order to train a model so that data is clustered in such a way that entire data is divided between several clusters. These clusters are basically representation of classes without having actual labels.

Our task is to train the model and come up with clustering in our data so that new points can be predicted to fall under a certain cluster. The centers of these clusters known as centroids are picked during the training. The different algorithms provide variations on how these clusters are decided. The 2 algorithms we will be using in this project are K-Means and Gaussian mixture model. These mainly differ in the way variance of the multiple gaussian distributions are considered. K-means considers equivalent variance while Gaussian mixture model takes into consideration possibilities of having different values of variances for different gaussian distributions. Let's have a detailed look in each of these.

1.1 K-Means Clustering

The K-Means algorithm clusters data by trying to separate samples in n groups of equal variances, minimizing a criterion known as the inertia or within-cluster sum-of-squares. The value of this n needs to be specified for this algorithm beforehand. This algorithm divides a set of samples into disjoint clusters, each described by the mean of the samples in the cluster. The means are commonly called the cluster centroids. These centroids can be any points in the same space as the data and not necessarily have to be the data points. The algorithm aims to choose centroid that minimise the inertia, or within-cluster sum-of-squares criterion:

$$\sum_{i=0}^{n} \min_{\mu_j \epsilon C}(||x_i - \mu_j||^2)$$

1.2 Gaussian Mixture Model (GMM)

Unlike K-Means clustering, where the algorithm just gives a cluster to which a data sample belongs to, GMM gives the probabilities for a data sample to belong to all current clusters. So, 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. This model can also be interpreted as generalized 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. A GaussianMixture.fit method is provided that learns a Gaussian Mixture Model from train data. Given test data, it can assign to each sample the Gaussian it mostly probably belongs to using the GaussianMixture.predict method.

1.3 Auto-Encoder

Autoencoding is a data compression algorithm where the compression and decompression functions are data-specific, lossy, and learned automatically from examples rather than engineered by a human. Autoencoder uses compression and decompression functions which are implemented with neural networks as shown in figure 1 below.

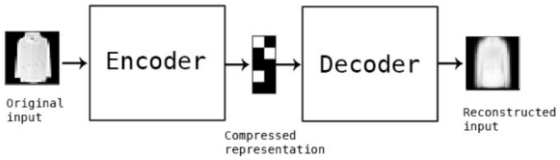


Figure 1: Auto-Encoder general representation

The architecture and approach to building such an auto-encoder is discussed in more details later.

2 Dataset

For training and testing our clusters models in this project, we are provided with Fashion-MNIST dataset. The dataset consists of the images from Zalando's article. All the images are 28x28 pixel sized grayscale images. We have 60,000 images as part of our training set, and 10,000 images as part of our testing set. A pixel value is given between 0 to 255 where higher value means a darker pixel. Each image has 784 pixels available, and a class label value that tells us which class of clothing that image belongs to. A pixel value can be between 0 and 255. There are 10 possible classes to which all the images in the training and testing sets can belong to. These classes are t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. However, we are not going to use the labels during the training of our models. We will only be using these labels to test our models after they have already been trained.

3 Preprocessing

The images are given to us in the form of their pixel values. For all 3 tasks we will be normalizing the data. For the 1st task we need to flatten the 2-dimensional image 28x28 to a 1-dimensional array of 784 pixels. This is also done for the 2nd and 3rd tasks as well but before doing this we make use the data to train our auto-encoder and get the weights involved in all the layers. We then use the same weights found after training to encode our data. This encoded data is then flattened as mentioned before for 2nd and 3rd tasks. For the first task, we are using the entire training data to train k-means model, but we will be dividing the training set into 80:20 ratio for training and validation set for tasks 2nd and 3rd. The testing set will be used at the end when we have trained and finalized our model to test

it on this unseen testing set.

4 Architecture

The code for all the 3 tasks required different approaches. In the 1st task we simply had to use the given input to train k-means model provided by sklearns, while for the 2nd and 3rd tasks we make use Keras library to design an auto-encoder and encode our data that could be used to train the k-means and Gaussian mixture models.

4.1 K-means Clustering Model

We are already given the data from which we know that there can only be 10 classes. For task 1 we simply had to use the KMeans class provided by the sklearns library in sklearn.cluster. We passed the flattened data into the model.fit and once the model was trained, we use the predict to test our model on unseen test data.

4.2 Auto-Encoder based K-Means Clustering Model

For 2nd task we came up with a multi-layered auto-encoder. An autoencoder comprises of an encoder and a decoder and both should be designed in such a way that they mirror each other so that the data is decompressed in the same manner as it was compressed. After few trails as mentioned in the experimentation section the architecture given in figure 2 seemed to be the best given the timeframe we had. Once we trained this auto-encoder. We used same weights for the encoder to encode the data, flattened images from 3-dimensional array to 2-dimensional array and trained our kmeans model.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization_8 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
batch_normalization_9 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 7, 7, 128)         73856
_________________________________________________________________
batch_normalization_10 (Batc (None, 7, 7, 128)         512
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 7, 7, 64)          73792
_________________________________________________________________
batch_normalization_11 (Batc (None, 7, 7, 64)          256
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 14, 14, 32)        18464
_________________________________________________________________
batch_normalization_12 (Batc (None, 14, 14, 32)        128
_________________________________________________________________
up_sampling2d_4 (UpSampling2 (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 28, 28, 1)         289
=================================================================
Total params: 186,497
Trainable params: 185,857
Non-trainable params: 640
```

Figure 2: Auto-Encoder architecture

4.3 Auto-Encoder based Gaussian Mixture Model

In order to compare the both Kmeans and GMM model fairly, we made use of the same auto-encoder model as used in the 2nd task given in figure 2. We got the weights of the different parameters in different layers for all layers and used the same weights for encoder. Then we used the encoded data and flattened the images from 3-dimensional array to 2-dimensional

array.

4.4 Hyperparameters

For the tasks, I had to tune several hyperparameters. These hyperparameters were max_iterations, Relative tolerance (tol), convolution layers, epochs, kernel size, maxpooling, upsampling, batchsize, covariance matrix types. How I tuned these parameters in order to get good models and the results that I had got from them is what I have described in the rest of the report below.

5 Experimentation

I am going to describe what I did for all 3 tasks in their specific sections below. I have however kept a similar pattern of modularizing the code in all the tasks so that the functions can be reused in other tasks and it's easier to understand. The accuracy of a model is found using a method provided by sklearns called **completeness_score** which basically checks the clusters without hard checking the labels, but rather checking if a high frequency of certain type of data samples belong to same cluster as they appeared in the given labels.

5.1 Clustering Images using K-Means Model

Task 1 didn't involve many hyperparameters. I tried the model with different max_iter and tol values. I tried the model with 1, 5, 40, 100, 200,400 and many in between values for max_iter and tried lower tol values like 0.001 and even 0.0001. It seemed to give the best result with max_iter around 200 and tol to be 0.0001 so I kept these as final values. I was able to get about 52.47% accuracy on the testing set at best.

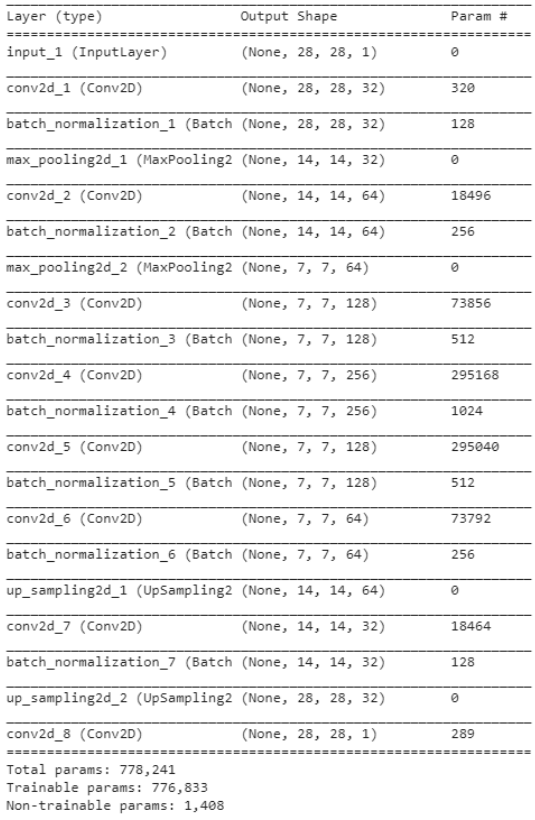
5.2 Auto-Encoder for tasks 2 and 3

For this task I had to come up with an architecture for an auto-encoder first. The layers in encoder and decoder had to be mirrored so that it functions properly. At first I tried with few layers by merely conv with maxpooling and upsampling. This architecture is given in figure 3.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization_17 (Batc (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 28, 28, 1)         289
=================================================================
Total params: 737
Trainable params: 673
Non-trainable params: 64
```

Figure 3: Initial Auto-Encoder architecture

Then I tested by further adding more convolution layers and having batchnormalization on them 1 at a time. A little better architecture that I got is in figure 4.



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
batch_normalization_1 (Batch	(None, 28, 28, 32)	128
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
batch_normalization_2 (Batch	(None, 14, 14, 64)	256
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	73856
batch_normalization_3 (Batch	(None, 7, 7, 128)	512
conv2d_4 (Conv2D)	(None, 7, 7, 256)	295168
batch_normalization_4 (Batch	(None, 7, 7, 256)	1024
conv2d_5 (Conv2D)	(None, 7, 7, 128)	295040
batch_normalization_5 (Batch	(None, 7, 7, 128)	512
conv2d_6 (Conv2D)	(None, 7, 7, 64)	73792
batch_normalization_6 (Batch	(None, 7, 7, 64)	256
up_sampling2d_1 (UpSampling2	(None, 14, 14, 64)	0
conv2d_7 (Conv2D)	(None, 14, 14, 32)	18464
batch_normalization_7 (Batch	(None, 14, 14, 32)	128
up_sampling2d_2 (UpSampling2	(None, 28, 28, 32)	0
conv2d_8 (Conv2D)	(None, 28, 28, 1)	289

Total params: 778,241
Trainable params: 776,833
Non-trainable params: 1,408

Figure 4: Auto-Encoder architecture before testing GMM

The architecture in figure 4 worked well with kmeans but it was crashing when I tried fitting the GMM. This was due to high number of kernels (256). So, in order to keep it fair between kmeans and GMM I ended up removing the conv layer with 256 kernels and I tinkered with more layers in between and so came to the final architecture for auto-encoder as given in figure 5.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization_8 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
batch_normalization_9 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 7, 7, 128)         73856
_________________________________________________________________
batch_normalization_10 (Batc (None, 7, 7, 128)         512
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 7, 7, 64)          73792
_________________________________________________________________
batch_normalization_11 (Batc (None, 7, 7, 64)          256
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 14, 14, 32)        18464
_________________________________________________________________
batch_normalization_12 (Batc (None, 14, 14, 32)        128
_________________________________________________________________
up_sampling2d_4 (UpSampling2 (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 28, 28, 1)         289
=================================================================
Total params: 186,497
Trainable params: 185,857
Non-trainable params: 640
```

Figure 5: Final Auto-Encoder architecture

After trying different batch sizes, the batch size of 480 seemed to work well enough. As per the epochs, trying more than 200 epochs was resulting in negligible decrements in loss so I made 200 epochs as final value for epochs. The training and validation loss graph that I got for our final model is given in figure 8.

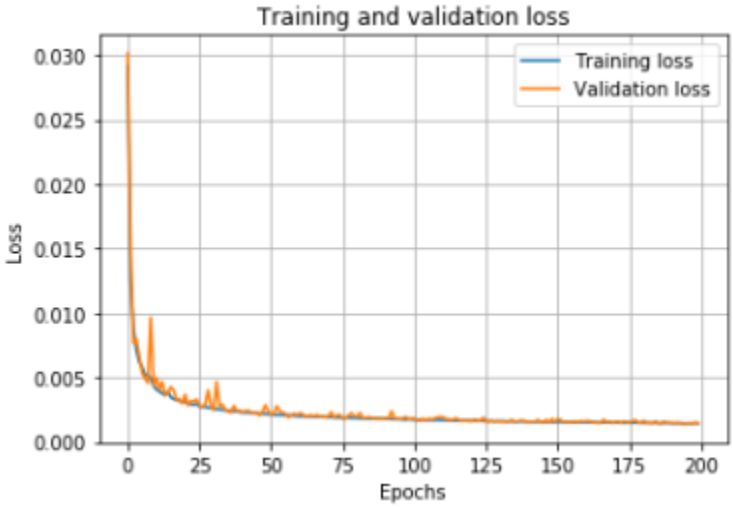


Figure 6: Training vs Validation loss graph for training autoencoder

5.3 Clustering using Auto-Encoder based Kmeans Model

For 2nd task, most of the experimentation done was for getting the right auto-encoder, which is described in previous section. I had to flatten the output of encoder to run into the kmeans. Apart from this everything was same as task 1. For Kmeans I ended up using the same final values of max_iter and tol as used for 1st task. For these hyperparameters I was able to get

about 51.87% accuracy.

5.4 Clustering using Auto-Encoder based Gaussian Mixture Model

For 3rd task, I make use of the same auto-encoder as used in task 2. The major tuning factor for Gaussian mixture model turned out to be which type of covariance matrix to use for our models. I tried all 4 types available to us by sklearns that are spherical, tied, diagonal(diag), and full. The spherical type seemed to be the fastest when it came to time it took to fit the model, while full type seemed to take the longest time to fit the model. However full type seemed to give the best accuracy compared to others. But even results of diag was very close to full type. Tied type was slightly better than spherical.

After many trials, it seemed like even though full gave the best result it wasn't best to use because of the amount of time and resources it took. The model with covariance type of full was not even able to complete in google collab with 25.51 GB ram assigned, when tried on the architecture with 256 kernels conv layer. This changed my choice for covariance matrix type from full to **diag** as final choice. For these final hyperparameters I was able to get about 56.26% accuracy.

5.5 Observations between different models

The very basic model used in 1st task which involved only used Kmeans gave the least accuracy on our test data. However, when I implemented this model with auto-encoder the accuracy did not have much impact. When I finally replaced kmeans with GMM for final task I saw relatively significant improvement in accuracy. The results of the final runs can be seen in the results section.

6 Results

The final accuracies that I got for the 3 tasks are given in the below table 1. The confusion matrix for task1 is given in figure 7, for task 2 is given in figure 8 and for task 3 is given in figure 9. The confusion matrix is not the actual representation as the clusters are different from the labels. The training and validation loss that I got for training the auto-encoder is illustrated in figure 10.

Table 1: Final Results

Task	Accuracy on Testing set
Task1: Kmeans	52.47%
Task 2: Auto-Encoder based Kmeans	51.87%
Task 3: Auto-Encoder based GMM	56.26%

```
[[  5   0   4   3   5   0   0   0 408   2]
 [ 29 890   4 503  27   0  12   0   6   0]
 [587  50  19 277 136   0 189   0   3   0]
 [ 34   9 566  10 627   0 312   0  62   0]
 [ 93  22  61  92  42 649 115  62  84  29]
 [245  29 342 113 159   6 357   0  36   4]
 [  0   0   0   0   0  44   0   2   1 407]
 [  6   0   4   2   4   0  15   0 351   0]
 [  1   0   0   0   0 229   0 791  42  26]
 [  0   0   0   0   0  72   0 145   7 532]]
```

Figure 7: Task 1: Confusion matrix for Kmeans model

```
[[517  41  14 268 115   0 160   0   3   1]
 [  0   0   0   0   0  81   0  39   0 816]
 [153  23 298  64 206   6 261   0  25   5]
 [  5   0   3   1   4   0   2   0 403   1]
 [ 30 814   4 457  24   0  20   0   4   0]
 [  2   0   0   0   0 737   5 174  58  62]
 [  3   0   3   1   4   4  12   2 331   2]
 [  1   0   0   0   0 162   0 785  45 108]
 [252 115 197 201  95  10 278   0  76   3]
 [ 37   7 481   8 552   0 262   0  55   2]]
```

Figure 8: Task 2: Confusion matrix for auto-encoder based Kmeans model

```
[[  0   0   0   0   0 536   0 921   5  95]
 [ 90 107   9 621  67   0  64   0  13   0]
 [ 23   4  23   3  12   4  48   7 437   3]
 [  8   0   7  11  16   0   8   0 450   0]
 [  0   0   0   0   0 346   0  67   2 886]
 [ 60  37  41  58  37 114  58   5  72  15]
 [181   4 275  46 107   0 273   0   0   1]
 [602  18  17 171  92   0 190   0   3   0]
 [ 33   7 628  12 666   0 357   0  18   0]
 [  3 823   0  78   3   0   2   0   0   0]]
```

Figure 9: Task 3: Confusion matrix for auto-encoder based GMM

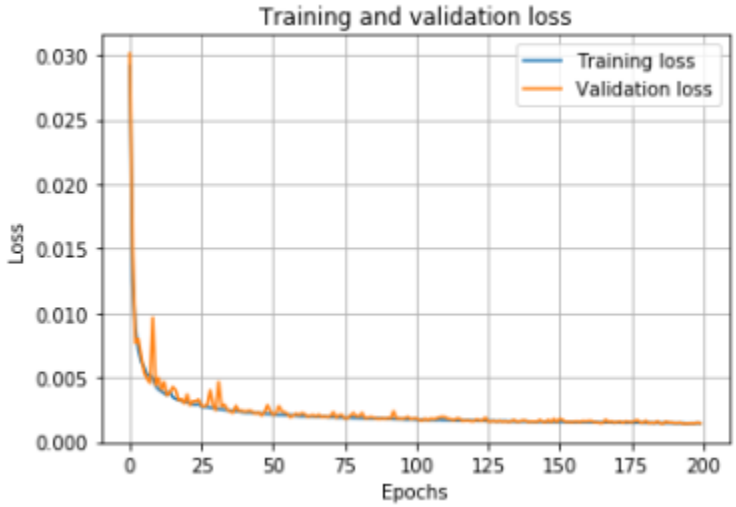


Figure 10: Final Auto-Encoder Training vs Validation loss graph

7 Conclusion

I was able to design the auto-encoders and use it with kmeans and GMM to cluster the given data samples of clothing from Fashion-MNIST dataset and predict which cluster an unknown sample may belong to. I got to a final architecture of autoencoder to use that was good enough to be used with kmeans and GMM to identify an image from an unseen dataset so that it belongs to one of the clusters.

Acknowledgments

I am grateful to Professor Sargur Srihari for helping me understand the concepts of unsupervised learning, different clustering algorithms, and auto-encoders. I also thank Mihir Chauhan for taking efforts in providing clarifications and better understanding of implementation of these concepts in python.

References

[1] Autoencoder as a Classifier (article) - DataCamp

[2] Fashion MNIST - K-means Clustering | Kaggle

[3] KMeans Clustering for Classification - Towards Data Science

[4] Gaussian Mixture Model Selection — scikit-learn 0.21.3 documentation

[4] GitHub - zalandoresearch/fashion-mnist: A MNIST-like fashion product database. Benchmark

[5] https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_selection.html#sphx-glr-auto-examples-mixture-plot-gmm-selection-py

[6] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.completeness_score.html