Deep Learning Clustering

Saman Paidar Nia

Vienna University of Technology

SPAIDARN@MTU.EDU

_______________ Scientific Report, February 2018 _______________

Availability Codes:

https://github.com/saman-nia/Autoencoder_Clustering

1. Abstract

In this report, we try to optimize an idea which already has been presented under title *Learning Deep Representations for Graph clustering* [1]. The idea is described as follows: “modeling a simple method which embedding the similarity graph by deep autoencoder with sparsity penalty, then runs the K-Means algorithm on the embedding graph to obtain the clustering result”. However, although our model is based on the original idea, but the graph similarity and the loss function and the model training methods are different. More, we compare our results with the two previous results based on the recent papers [1, 2] on the same datasets. We also compare our autoencoder with spectral clustering in terms of their results. Then, we proof that our autoencoder is more efficient and robust than spectral clustering.

Keywords: Deep Learning, Stacked Autoencoder, Graph Similarity, Clustering.

2. Deep Learning Clustering

By employing deep neural network, we are able to solve a few basic problems in accuracy in the traditional machine learning to get the true label. Today, deep learning is widely used in data science such as image classification [11] and it makes artificial intelligence more intelligent. Clustering is one of the most promising methods in machine leaning. Clustering algorithms such as K-Means can be able to have better clustering result if the graph is represented in a smaller dimension. To obtain this important, deep autoencoder also called stacked autoencoder can be used for learning better representations of the data in low dimension [12].

The way it works is that first compute the normalized cosine similarity or normalized correlation similarity of the dataset then passing the original graph similarity through a stacked autoencoder and finally runs k-means algorithm on the embedded graph similarity to obtain labels.

3. Graph notation and similarity graph

3.1. Graph notation

G (V, E) is a graph which *V* is number of vertices (node) and *E* is number of edges. The *E* is a subset of $V \times V$ $(E \subset V \times V)$. The graph edges could have weights, which expression a feature between the nodes [6]. See figure 1 as the structure of the graph.

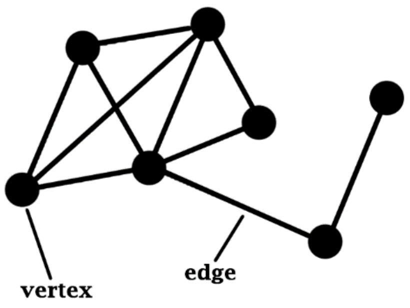

Figure 1: *a graph*

A graph can be directed or undirected:

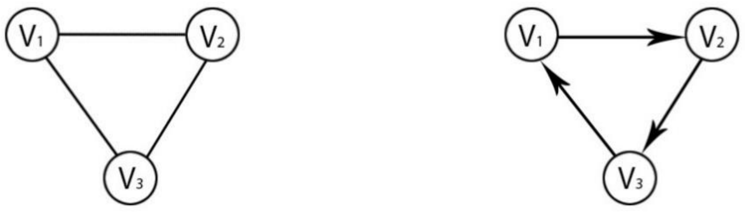

Figure 2: *undirected graph (left) and directed graph (right)*

3.2. Types of similarities graph used

3.2.1 Normalized cosine similarity

Before computing graph similarity, we normalized data. Data can be an array or a sparse matrix.

1. Normalize each non-zero feature of the input vector [3]. All this process called standardization. We normalize the vector before passing it into a similarity computing. After standardization each feature is normally distributed:

$$\hat{\mathrm{v}} = \frac{\vec{\mathrm{v}}}{|\vec{\mathrm{v}}|}$$

2. Compute the cosine distance between vectors *u* and *v*:

$$1 - \frac{\mathrm{u}\,.\,\mathrm{v}}{||\mathrm{u}||_2||\mathrm{v}||_2}$$

Where $||*||_2$ is the 2-norm of its argument *, and *u.v* is the dot product of *u* and *v* [4].

3. Convert the produced cosine distance to a square-form distance matrix:

$$\mathrm{X} = \frac{\mathrm{d} \times \mathrm{d}(-1)}{2}$$

Where cosine distance sized $\frac{d \times d(-1)}{2}$ and *d* is two or more than two dimensions and returns a *d* by *d* distance matrix *X* [4].

4. Compute the random walk of the normalized graph laplacian [6]:

$$\mathrm{L} = \mathrm{D}^{-1}\,\mathrm{L} = \mathrm{I} - \mathrm{D}^{-1}\,\mathrm{W}$$

This includes several steps:

a. Compute the sum of element of array over the zero axis [4]:

```
>>> ([[0, 1], [0, 5]], axis = 0)
([0, 6])
```

b. Extract the diagonal array from similarity matrix of step 4:

$$\boldsymbol{D} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_2 \end{bmatrix} \Rightarrow \boldsymbol{D}^{-1} = \begin{bmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/d_n \end{bmatrix}$$

c. Compute dot product of $D^{-1}L$:

$$\mathrm{L}\,[0]\,[0] \times \mathrm{D}\,[0] + \mathrm{L}\,[1]\,[0] \times \mathrm{L}\,[1] + \ldots + \mathrm{L}\,[\mathrm{n}-1]\,[0] \times \mathrm{D}\,[\mathrm{n}-1]$$

The similarity matrix is row normalized where *L* is the cosine similarity matrix and *D* is the diagonal matrix with the node degrees in the corresponding diagonal elements [1].

5. Compute normalize linearly for each matrix column [5]. After that we have minimum value 0 and maximum value 1:

$$z_i = \frac{x_2 \,.\, \min(x)}{\max(x) - \min(x)}$$

Where $x=(x_1 \ldots x_n)$ and z_i is now our i^{th} normalized data.

3.2.2 Normalized correlation similarity

1. Normalize each non-zero feature of the input vector [3]. All this process called standardization. We normalize the vector before passing it into a similarity computing. After standardization each feature is normally distributed:

$$\hat{v} = \frac{\vec{v}}{|\vec{v}|}$$

1. Compute the correlation distance between vectors *u* and *v*:

$$1 - \frac{(u - \bar{u}) \,.\, (v - \bar{v})}{||(u - \bar{u})||_2||(v - \bar{v})||_2}$$

Where $\bar{v}$ is the mean of the elements of vector *v*, and *x.y* is the dot product of *x* and *y* [4].

2. Convert the produced correlation distance to a square-form distance matrix:

$$X = \frac{d \times d(-1)}{2}$$

Where correlation distance sized $\frac{d \times d(-1)}{2}$ and *d* is two or more than two dimensions and returns a *d* by *d* distance matrix *X* [4].

3. Compute the random walk of the normalized graph laplacian [6]:

$$L = D^{-1} L = I - D^{-1} W$$

This includes several steps:

a. Compute the sum of element of array over the zero axis [4]:

```
>>> ([[0, 1], [0, 5]], axis = 0)
([0, 6])
```

b. Extract the diagonal array from similarity matrix of step 4:

$$\boldsymbol{D} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_2 \end{bmatrix} \Rightarrow \boldsymbol{D}^{-1} = \begin{bmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/d_n \end{bmatrix}$$

c. Compute dot product of $D^{-1}L$:

$$L[0][0] \times D[0] + L[1][0] \times L[1] + \ldots + L[n-1][0] \times D[n-1]$$

The similarity matrix is row normalized where *L* is the correlation similarity matrix and *D* is the diagonal matrix with the node degrees in the corresponding diagonal elements [1].

4. Compute normalize linearly for each matrix column [5]. After that we have minimum value 0 and maximum value 1:

$$z_i = \frac{x_2 \,.\, \min(x)}{\max(x) - \min(x)}$$

Where $x=(x_1 \ldots x_n)$ and z_i is now our i^{th} normalized data.

4. Graph Auto-Encoder

Deep feed-forward neural networks, also called multilayer perceptrons, is about how to train each layer that the layer gradually learned as close as possible to the input. In the unsupervised learning embedding, we need a network model which is able to solve the problem with respect to input aspects [8]. For instance, $y = f(x)$ plots the x to the y for classifier.

A multilayer perceptrons defines a topography for $y = f(x; \theta)$. Then learns θ which is a value in a maximum performance [9]. The autoencoder retained feed-forward neural networks learning method however, the graph contains two parts, the encoder which compresses the input. Then the decoder part reconstructs the input from the embedded data or codes.

After each backpropagation, the cost of the function is reduced, this means that the best feature of the model are derived. When we have the best of features, then we can expect to have a well representation of the input into a low dimensions. This is a non-linear generation with the employee neural networks to obtain more structure [10]. While decreasing the cost function, the K-Means algorithm can also has a better performance on the produced graph for clustering.

See figure 4 as an autoencoder architecture [8]:

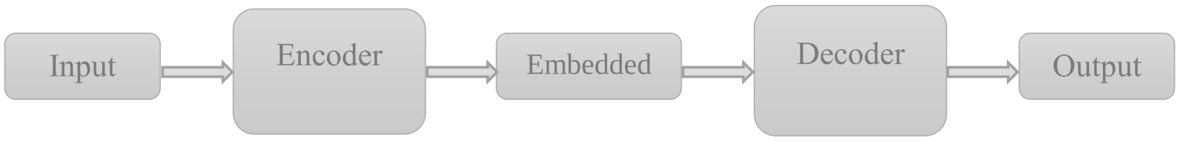

***Figure 4:** the autoencoder architecture*

5. Model description

As much as the array-list is different from the sparse-matrix, the scenario defined for the graph-encoder must also be different for the data input of both. Accordingly, we define two types of graph-encoder for two type of input data. The type of autoencoder network which we used for dimensionality reduction is based on paper *"Reducing the dimensionality of data with neural networks"* [10]. This autoencoder network can be trained at the cost of $O\ (i \cdot n)$, where i represents the training and the training complexity simplifies to $O\ (n)$. Therefore, we use the normalized correlation similarity for the sparse-matrix input and use the normalized cosine similarity for array-list.

For training and optimization of the model, we used the Adam algorithm. Adam uses moving averages of the parameters. Plainly, this authorizes Adam to use a more effective step size. Subsequently the algorithm will converge to this step size without well tuning [14]. The Adam algorithm has a large computation for each parameter in each training step. We also used the moving averages for evaluations thus we would have better results [7].

The parameters include two categories, one constant parameters for both graph-encoder and another specific parameters based on the each graph-encoder.

- Constant parameters:

- Beta 1: The exponential decay rate for the first moment estimates [7]:

$$\beta 1 = 0.9$$

- Beta 2: The exponential decay rate for the second moment estimates [7].

$$\beta 2 = 0.999$$

- Learning rate: The *learning rate* is used which is indifferent to the error gradient. We set the value according to input size:

$$\text{Learning rate} = \frac{\text{input size}}{\beta 1}$$

  Thus, the larger the input, the learning rate gets the smaller values and it lets to the model to obtain lowest cost function with delay. The delay in achieving a best cost function makes the model more likely to improve the distribution of the graph.

- Epsilon: we set the epsilon value to the default value.

$$\text{Epsilon} = 1\text{e-}08$$

  According to the TensorFlow, it might not work well in general, however it works well on our model [7].

- Loss function: we gave the input vector and consequently, we have an output which is the reconstruction of the input. Our function minimize the value by following the formula:

$$|| \text{Input} - \text{Output} || = \sum_{i} (\text{Input}_{i} + \text{Output}_{i})^{2}$$

- Then we pass the similarity matrix through a graph-encoder and Computes sigmoid of x element-wise [7]: $y = 1 / (1 + \exp(-x))$

However there is an interesting point inside the graph and that is a repetitive loop implementation method for both which we illustrate the graph-encoders on table 1 for correlation similarity and 2 for cosine similarity:

Table 1: *graph-encoder for correlation similarity input*

Build deep autoencoder and define all layers and span of variables.
▪ Input is normalized correlation similarity (n×n) created from sparse-matrix. ▪ Ⅎ since the loop is controlled by a while loop, we need a parameter which called stop learning rate to break in the loop when the cost function gained a satisfactory value: Ⅎ = Number of layers ▪ Ꞁ is number of batch data which is: $Ꞁ = \sqrt{\text{size of input}}$ $Ꞁ = \sqrt{Ꞁ}$ ▪ ʞ is number of backpropagation in per epoch which is: $ʞ = \frac{\text{size of input}}{Ꞁ}$

```
For i =1 to Ⅎ
    For j = 1 to ʞ
        Train the autoencoder with Ꞁ number of data along with backpropagation strategy.
        Obtain the cost value.
End
Obtain the code or embedded data after optimization.
Run k-means algorithm on embedded data.
```

For training the model, a mini-batch sample has been randomly selected.

Table 2: graph-encoder for cosine similarity input

```
Build deep autoencoder and define all layers and span of variables.
▪ Input is normalized cosine similarity (n×n) created from array-list.
▪ Ⅎ is number of epoch which is
      $Ⅎ = \frac{\text{total nodes of all layers}}{\text{number of layers}}$
▪ Ꞁ is number of batch data which is:
      $Ꞁ = \sqrt{\text{size of input}}$
      $Ꞁ = \sqrt{Ꞁ}$
▪ ʞ is number of backpropagation in per epoch which is:
      $ʞ = \frac{\text{size of input}}{Ꞁ}$
For j = 1 to Ⅎ
    For i = 1 to ʞ
        Train the autoencoder with Ꞁ number of data along with backpropagation strategy.
        Obtain the cost value.
End
Obtain the code or embedded data after optimization.
Run k-means algorithm on embedded data.
```

6. Experimental evaluation

We experimentally test the models on the several datasets which already have been used by recent papers [1, 2].

6.1. Datasets

6.1.1. **The 20 newsgroups text dataset:** the dataset contains 20 categories of news. The recent papers [1, 2] selected three different groups such as *3NG, 6 NG, 9NG*. For each group, we selected 200 documents randomly. It means for our groups we have 600, 1,200, 1,800 documents [1, 2]. After converting datasets to vector by TF-IDF, we would have sparse-matrix, thus the normalized correlation has a better performance on that.

6.1.2. **Wine dataset:** the dataset contains 3 categories with 178 instances of wines. The data set has 9 features. Both recent paper [1, 2] use the dataset, however the newest one [2] only used the dataset for visualization. Since the dataset is an array-list, thus the normalized cosine similarity has better performance on that.

6.1.3. **Iris dataset:** the dataset contains 3 categories with 150 instances of iris plant. The data set has 9 features [13]. Since the dataset is an array-list, thus the normalized cosine similarity has better performance on that.

The structure of our layers is based on origin paper [1], see table 3:

Table 3: *number of nodes in each layer*

Dataset	Nodes
Wine	178 – 128 – 64
Iris	150 – 128 – 2
3NG	600 – 512 – 256
6NG	1,200 – 1,024 – 512 – 256 - 128
9NG	1,800 – 1,024 – 512 – 256 - 128

See figure 5 to explore autoencoder embedded to two dimensions for the Iris dataset.

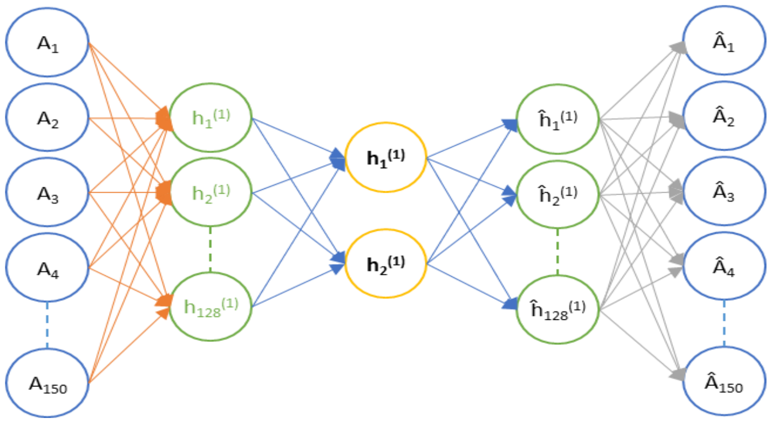

Figure 5: *a dimensionality reduction of the similarity matrix of the Iris dataset used by Hinton and Salakhutdinov, 2006.*

6.2.Results

We show the results of the origin paper [1] as **SAE** which shorted for spars autoencoder and the second paper [2] as **DAE** which shorted for denoising autoencoder.

We also compare our result called **AE** which shorted for autoencoder to both recent result [1, 2]. See table 4:

Table 4: *comparing the NMI scores*

Autoencoder Model	wine	iris	3NG	6NG	9NG
SAE	84.00	*no-result*	81.00	60.00	41.00
DAE	*no-result*	*no-result*	80.44	68.87	59.13
AE	**100**	**100**	**87.83**	**70.09**	**60.64**

Here, we compare the model with normalized spectral density clustering. To show the power of deep learning, we also run the k-means algorithm on the original similarity graph. We use normalized correlation similarity for 20-NG datasets and normalized cosine similarity for wine and iris datasets. See table 5:

Table 5: *comparing the NMI scores of the spectral clustering and AE*

Clustering Algorithms	wine	iris	3NG	6NG	9NG
K-Means	87.21	100	52.95	48.10	47.35
Normalized Spectral Density Clustering	86.62	100	71.14	67.27	53.94
AE	**100**	**100**	**87.83**	**70.09**	**60.64**

7. Visualization of the results

Here, we visualize all the result including the improvement diagram of cost functions and NMI score. See figure 6, 7, 8, 9 for The 20 newsgroups text dataset:

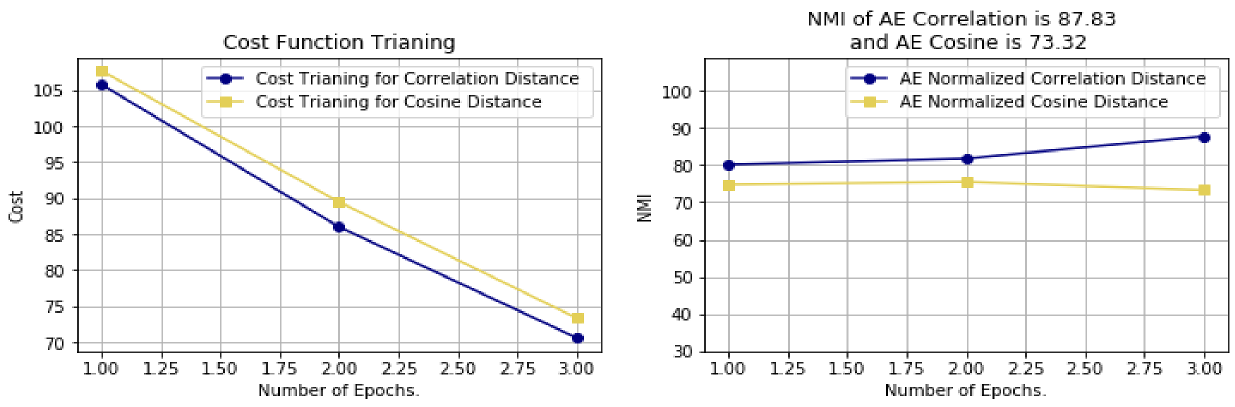

Figure 6: *3NG performances*

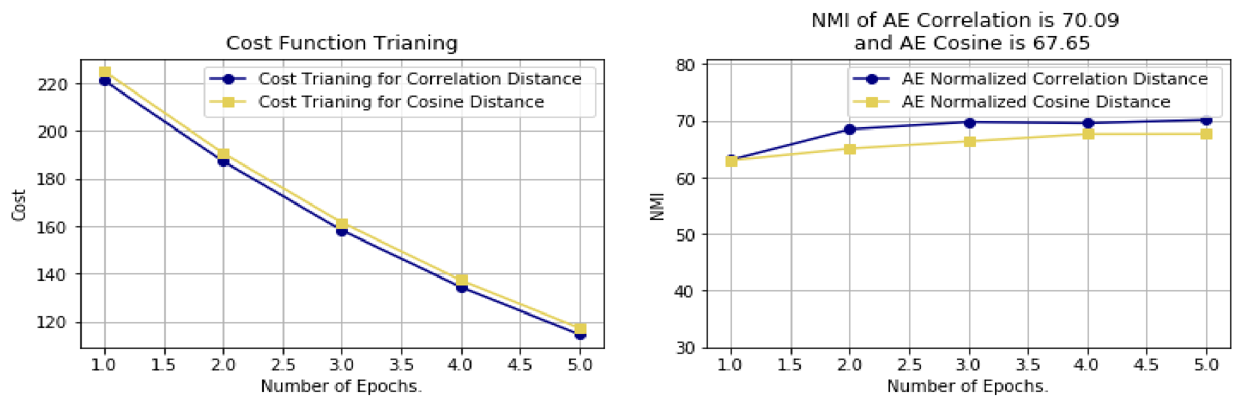

Figure 7: *6NG performances*

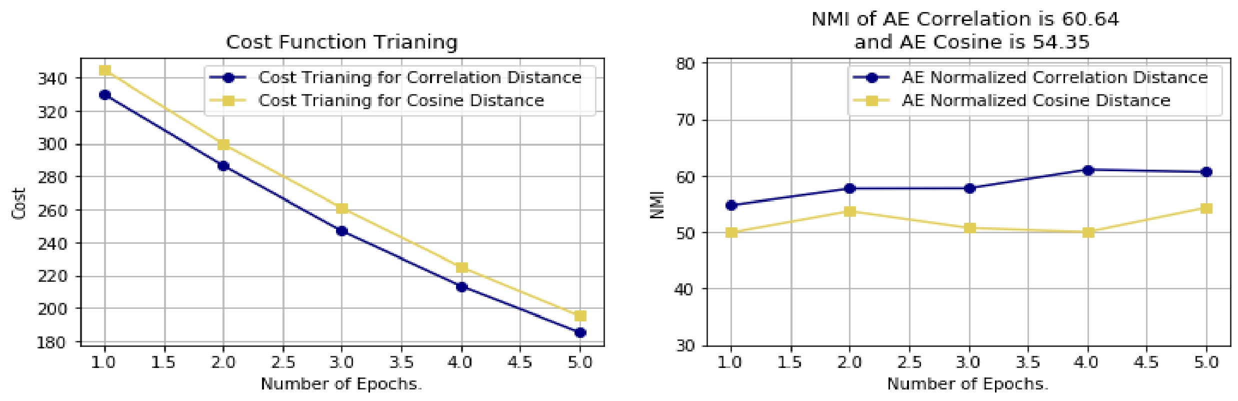

Figure 8: *9NG performances*

As you see, we can have a better performance in normalized correlation similarity for a text dataset. The time of training in the normalized correlation similarity would be less than the normalized cosine similarity, However when we apply the autoencoder on the wine and iris datasets, we have a better performance on the normalized cosine similarity:

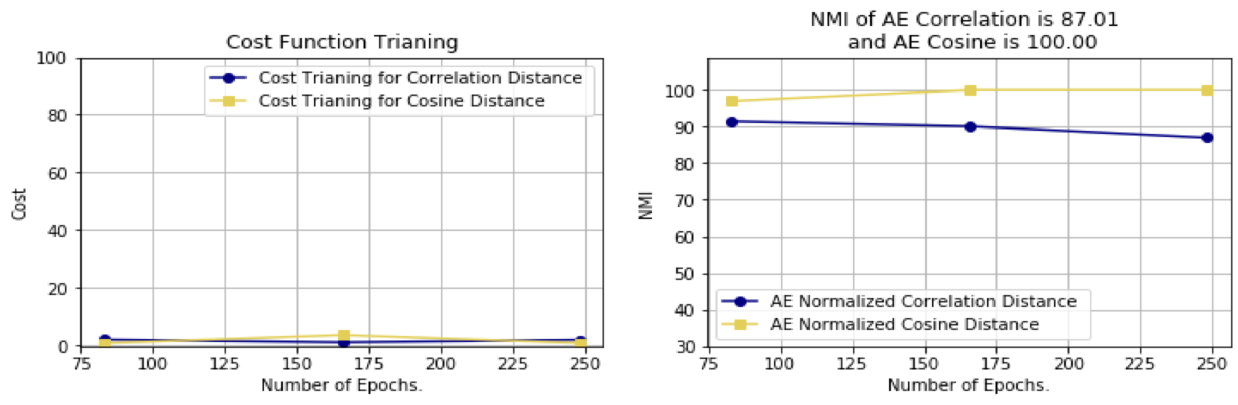

Figure 10: *iris performances*

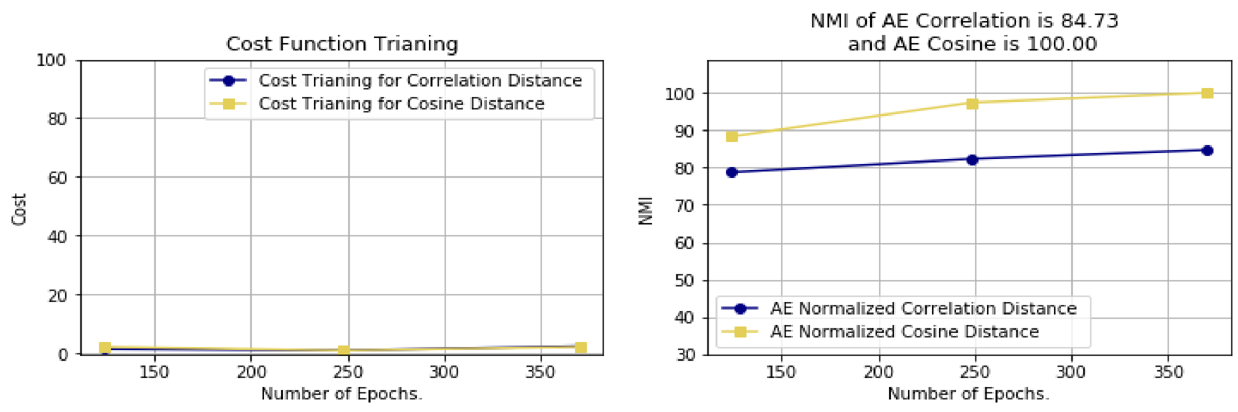

Figure 11: *wine performances*

To show the power of our model, we reduce dimensions of the 3NG dataset to two dimensions, however the result of our model still is better that two recent papers [1, 2]. See table 6.

Table 6: *the structure of 3NG dataset in 2D embedded*

Dataset	Nodes	NMI
3NG	600 – 512 – 256 – 128 – 2	**85.33**

Here we visualize the two dimensions produced for 3NG in figure 11. One with original labels (right) and another with predicted labels by k-means algorithm (left).

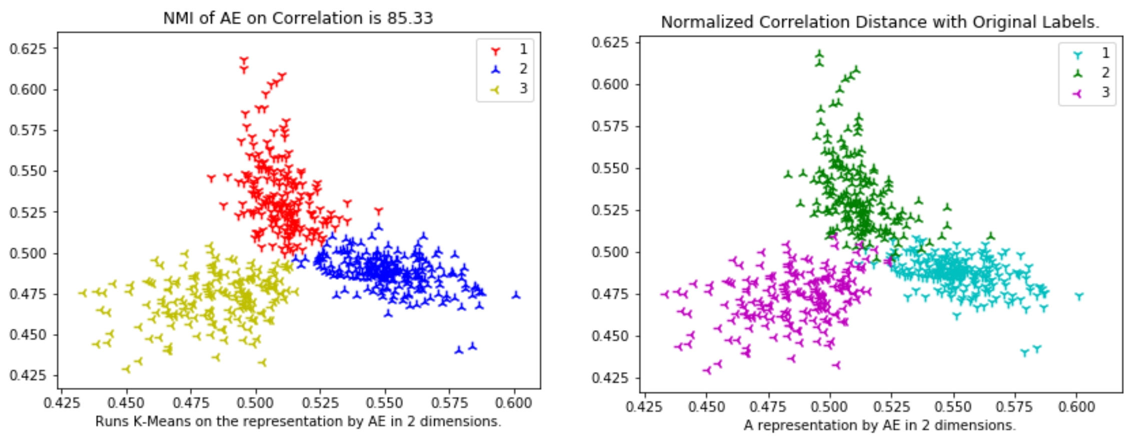

Figure 11: *embedded the 3NG dataset in 2 dimensions.*

Here, we compare both normalized cosine similarity and normalized correlation similarity in two dimensions embedded. See figure 12:

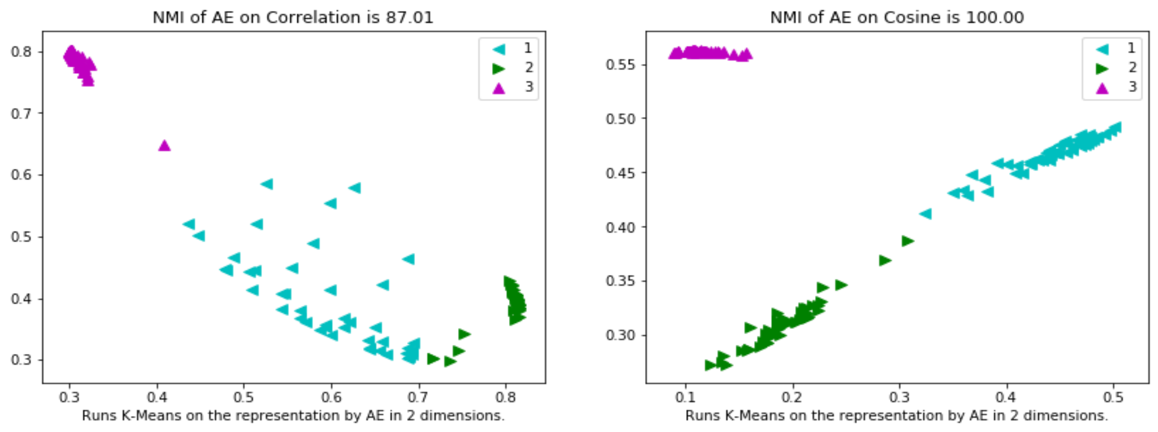

Figure 12: embedded the Iris dataset in 2 dimensions.

Our autoencoder has done a glorious job on the Iris dataset with an input from the normalized cosine similarity.

We also visualize 10 random factors with an average of NMI results. See figure 13:

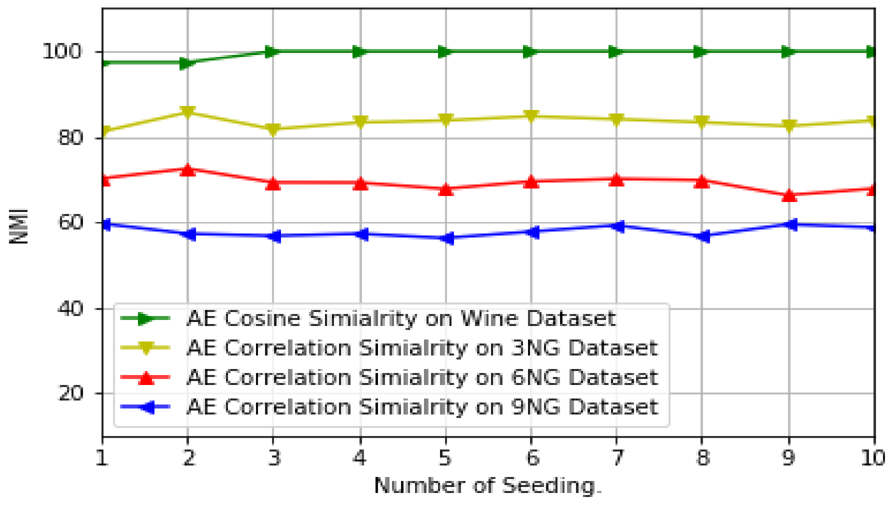

Figure 13: means of random factor.

8. The difference between the proposed model

Although we have tried to continue the two recent paper [1, 2], but there are fundamental differences in the structure of our model and the recent works [1, 2]. The most fundamental difference is in our graph similarity where we used two kind of similarity distance such as normalized cosine similarity and normalized correlation similarity.

The graph similarity present as $D^{-1}L$, where D is diagonal matrix and L is the graph laplacian matrix and then normalized the $D^{-1}L$, whereas they used same $D^{-1}L$, but used only the normalized cosine similarity [1]. The second difference is in the autoencoder training model, where we used a simple autoencoder with a nested loop which outer one for control the number of epochs and inner one for training the autoencoder with backpropagation to optimize the graph similarity.

For training the model we get a batch of input size data which is a random list from the original similarity matrix. The batch size *($\daleth$)* is controlled by the size of input:

$$\daleth = \sqrt{\text{size of input}}$$
$$\daleth = \sqrt{\daleth}$$

Whereas they used a for loop and train whole data set into autoencoder to optimize the graph similarity. In the first paper [1], a sparsity penalty proposed and in the second one [2], a denoising mechanism has been proposed. However, both proposals have not yet been able to provide a default model for all types of input data.

9. Conclusion

In this work, we trained a simple stacked autoencoder and proved that the optimization of graph similarity in our autoencoder is better and more sensible than two recent papers [1, 2]. However, several issues such as the best parameter's value still remain when we have different inputs, especially the difference in structure.

Furthermore, after that we have optimized the structure of our autoencoder, we run the autoencoder more than a hundred times in different processor operating conditions. The result was that, the power of the system and processor at runtime could have a remarkable in the random walk of the model performance. Whatsoever therefore the processor is freer, the model will be optimized better.

The next step is to improve the model and define a standard algorithm based on that for the Python language.

Works Cited

[1] F. Tian, B. Gao, Q. Cui, E. Chen, T. Liu, 2014.

[2] S. Cao, W. Lu, Q, Xu, 2016.

[3] Pedregosa, 2011.

[4] E. Jones, T. Oliphant, P. Peterson and others, 2001.

[5] J. Brownlee, 2016.

[6] Ulrike von Luxburg, 2006.

[7] Tensorflow2015-whitepaper, https://www.tensorflow.org.

[8] N. Bumduma, 2017.

[9] I. Goodfellow, Y. Bengio, A. Courville, 2016.

[10] G. Hinton, Salakhutdinov, 2006.

[11] A. Krizhevsky, L. Sutskever, and G. Hinton 2012.

[12] E. Aljalbout, V. Golkov, Y. Siddiqui, D. Cremers, 2018.

[13] M. Lichman, 2013.

[14] D. P. Kingma, 2015

References

F. Tian, B. Gao, Q. Cui, E. Chen, T. Liu, Learning Deep Representations for Graph Clustering.

E. Aljalbout, V. Golkov, Y. Siddiqui, D. Cremers, Clustering with Deep Learning: Taxonomy and New Methods.

Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

http://www.scipy.org/ , Eric Jones, Travis Oliphant, Pearu Peterson and others, SciPy: Open Source Scientific Tools for Python, 2001.

https://machinelearningmastery.com/normalize-standardize-time-series-data-python/ , Jason Brownlee, 2016

Daixin Wang, Peng Cui, Wenwu Zhu, Tsinghua National Laboratory for Information Science and Technology.

S. Cao, W. Lu, Q, Xu, Deep Neural Networks for Learning Graph Representations, 2016.

N. Bumduma, Fundamentals of Deep Learning, 2017.

I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

Hinton, G.E. and Salakhutdinov, R.R. (2006) Reducing the Dimensionality of Data with Neural

Networks. Science, 313, 504-507. http://dx.doi.org/10.1126/science.1127647

Adam: D. P. Kingma, J. Lei Ba, a Method for Stochastic Optimization, 2015

Tensorflow2015-whitepaper, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, https://www.tensorflow.org.

M. Lichman, Machine Learning Repository, http://archive.ics.uci.edu/ml, 2013.

Asuncion, A., and Newman, D. 2007. Uci machine learning repository.

Bengio, Y.; Lamblin, P.; Popovici, D.; and Larochelle, H. 2007. Greedy layer-wise training of deep networks. Advances in neural information processing systems 19:153.

Poultney, C.; Chopra, S.; Cun, Y. L.; et al. 2006. Efficient learning of sparse representations with an energybased model. In Advances in neural information processing systems.

Satuluri, V., and Parthasarathy, S. 2009. Scalable graph clustering using stochastic flows: applications to community discovery. In Proceedings of the 15th ACM SIGKDD conference, 737–746. ACM.

Smyth, S., and White, S. 2005. A spectral clustering approach to finding communities in graphs. In Proceedings of the 5th SIAM International Conference on Data Mining, 76–84.