ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ＆ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

MACHINE LEARNING

Image Similarity with NN

Authors:

Vasiliki Koumarela	VasiaKoum	- 1115201600074
Charalampos Katimertzis	chariskms	- 1115201600062

Parts

Part 1 : Reduce-Dimensions-Bottleneck-Autoencoder

Part 2 : LSH-and-TrueN-Approximation-factor

Part 3: EMD-metric

Part 4 : Clustering-and-Classification

Introduction

Requirements

- g++ compiler
- Python 3.8
- Keras
- Numpy
- Matplotlib
- Tensorflow
- Pandas
- Sklearn

Files

- ❖ Part1: reduce.py

- Part1: functions.py
- Part2: search.cpp
- Part2: lshAlgorithms.cpp
- Part2: dataset.cpp
- Part2: hash.cpp
- Part2: metrics.cpp
- Part3: linear.py
- Part3: search.cpp
- Part3: Algorithms.cpp
- Part3: dataset.cpp
- Part3: metrics.cpp
- Part4: centroids.cpp
- Part4: cluster.cpp
- Part4: classification.py
- Part4: functions.py

Metrics and times were measured at Google Colab with specs: CPU x2 @ 2.2GHz, RAM:13 GB, GPU: Nvidia Tesla K80 (.py programs)

Part 1:

Reduce-Dimensions-Bottleneck-Autoencoder

The model was trained with a subset of 60k of black and white images of digits. The encoder for one layer uses **Conv2D**, **BatchNormalization** and **MaxPooling2D** (for the first 2 layers). **Bottleneck(flatten_layer,**

embedding_layer, dense_layer, reshape_layer) is used as an intermediate part between encoder and decoder and it reduces the original dimensions into smaller ones. The decoder uses Conv2D, BatchNormalization and **UpSampling** (for the last 2 layers). The program is using the mirrored architecture, that is symmetric layers in the encoder & decoder, which is more efficient. **Mean_Squared_error** (for loss → `mean_of_square(y_true - y_pred)`) & **RMSprop** (for optimizer → maintaining a moving average of the square of gradients, dividing the gradient by the root of this average) were used to compile the model. Also, for each hidden layer, **REctified Linear Unit** activation function is used except from the last layer in the decoder that **Sigmoid** activation function is called.

The reducer is executed by:

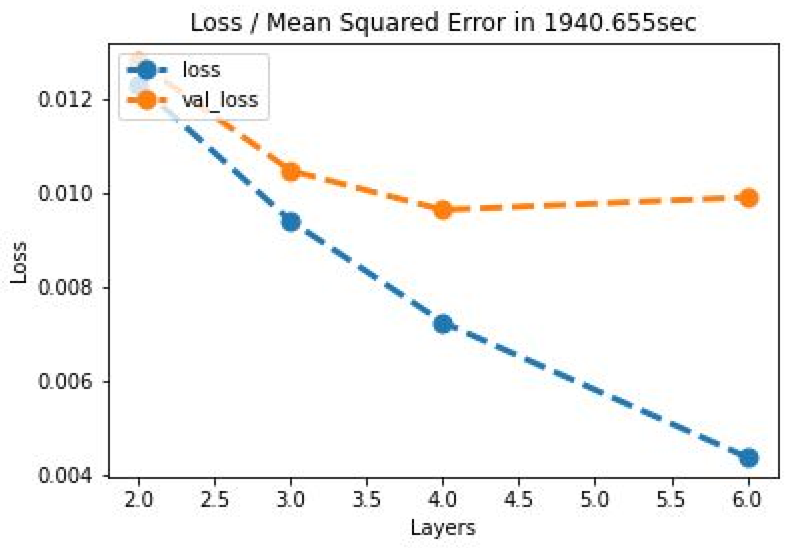
```
python reduce.py -d ../Datasets/train-images-idx3-ubyte -q
../Datasets/t10k-images-idx3-ubyte -od outdata -oq outquery
```
.

The user choices are: 1) Execution with different hyperparameters, 2) Plot of the error-graphs, 3) Save of the existing model, 4) Exit. If the user chooses (1), then the user choices are: 1) Layers, 2) Filter_Size, 3) Filters/Layer, 4) Epochs, 5) Batch_Size, 6)Latent_Vector. For the graph plots, the program shows a graph with the losses & the changed hyperparameter. The user, executing the program once, can change multiple hyperparameters and the graph plots will include all the changes of the hyperparameters, keeping the initial hyperparameters constant and each time the selected hyperparameter will be changed. The output from this program is the reduced dimensions (2 bytes each pixel) and it is used as input in Part2 and Part4.

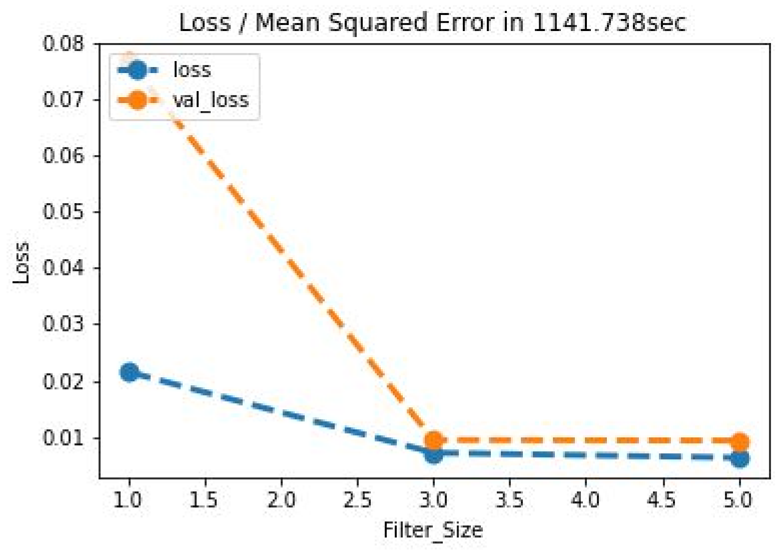
1. EXPERIMENTS

We executed the reduce.py with different values for hyperparameters and these are the results:

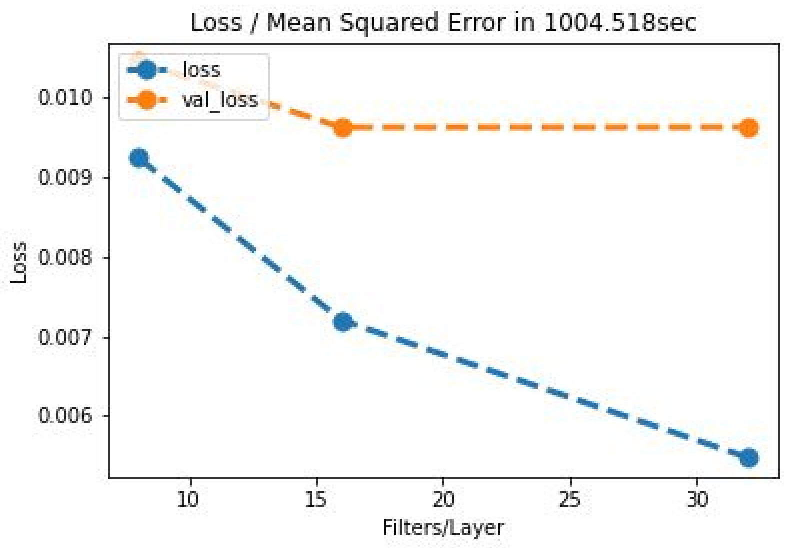
- **Lx_FS3_FL16_E100_B40_LV10**



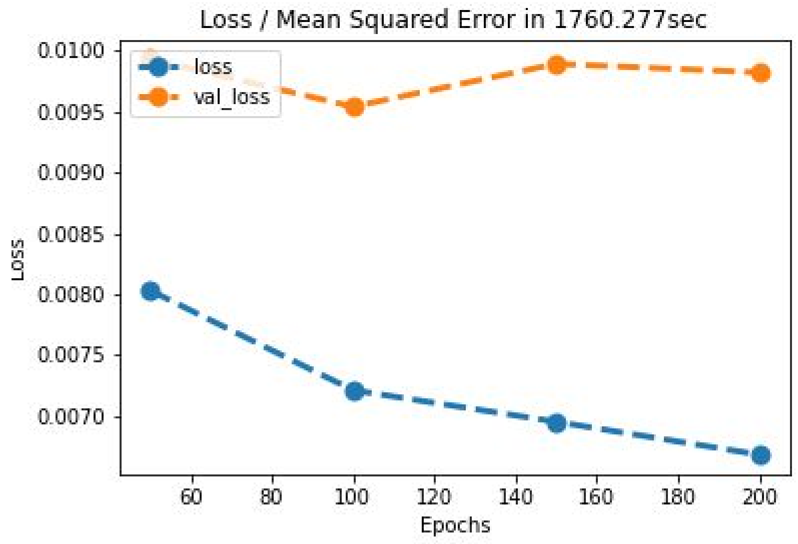
- **L4_FSx_FL16_E100_B40_LV10**



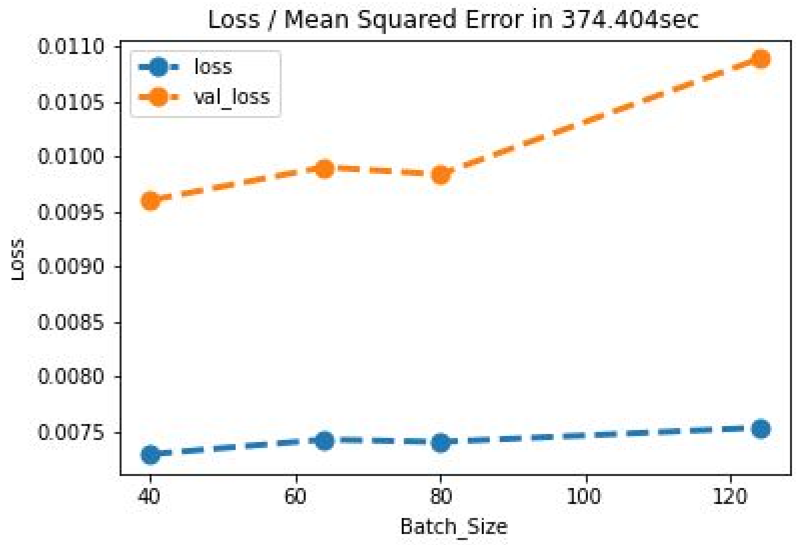
- **L4_FS3_FLx_E100_B40_LV10**



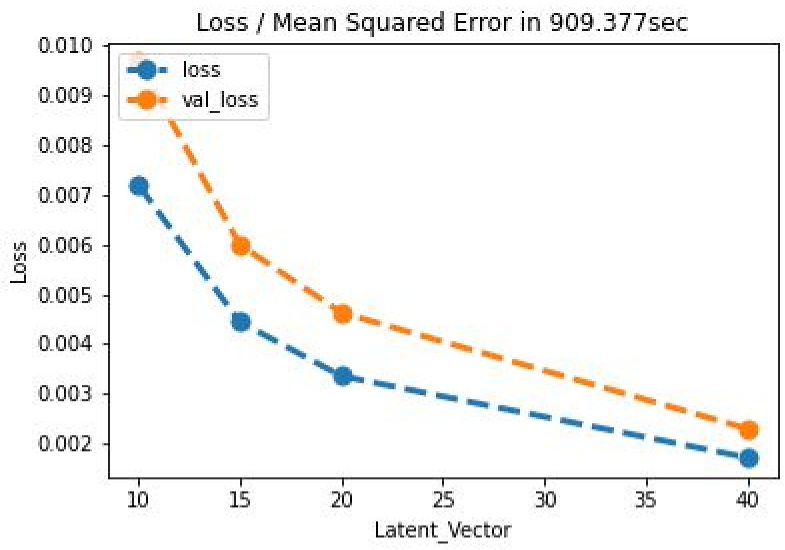
- **L4_FS3_FL16_Ex_B40_LV10**



- **L4_FS3_FL16_E100_Bx_LV10**



- **L4_FS3_FL16_E100_B40_LVx**



Losses using activation function (ReLU, Softmax) in Bottleneck part of autoencoder:

LAYERS	4		4		4	
FILTER_SIZE	3		3		3	
FILTERS/LAYER	32		32		32	
EPOCHS	30		30		30	
BATCH_SIZE	124		124		124	
LATENT_SIZE	**10**		**15**		**30**	
	los	*v-los*	*los*	*v-los*	*los*	*v-los*
RELU & SOFTMAX	0.0088	0.0129	0.0059	0.0108	0.0035	0.0055
NO RELU & SOFTMAX	0.0077	0.0112	0.0050	0.0076	0.0027	0.0046

We use min-max feature scaling type to restrict the range of values in the dataset between any arbitrary points a and b:

$$X' = \frac{(b-a)(X-X_{min})}{X_{max}-X_{min}} + a$$

With ReLU & Softmax, X_{min} is: 0.0 and X_{max} is approximate: 2135.762. On the other hand, without the use of activation function, X_{min} and X_{max} range is between -4433.5776 and 3345.7158. We want to normalize the values between 0 and 25500. So without activation function the range of numbers is larger and the normalization is more accurate.

2. EXPLANATION OF RESULTS

From the experiments we observe that the best values for the hyperparameters are: Layers=4, Filter_Size=3, Filters/Layer=16, Epochs=100, Batch_Size=40 or 80, Latent_Vector $\in$[20,40]. After selecting the values for the hyperparameters, we observe that when layers are increased, the loss plot falls. On the other hand, validation_loss for layer=4 has the minimum loss. Also, for Filter_Size, the best choice is 3 because this value has the minimum val_loss. Besides that, 3 is usually the one chosen because we have symmetry with the previous layer pixel and the output, so we will not have to account for distortions across the layers which happen when using an even sized kernel. The Filters/Layer catch few of some simple features of images (edges, color tone, etc) and the next layers are trying to obtain more complex features based on simple ones. From the error-graphs, the best value for Filters/Layer is 16 because in this value, the val_loss is low. For Epochs and Batch_Size, as we increase the epochs, the losses are decreasing and respectively as we decrease the Batch_Size, the values for losses are decreasing, as well. From loss_values.csv, we observe that the Layers and Filters/Layer affect negatively training time and the deeper the CNN, the higher are the loss values.

Part 2:

LSH-and-TrueN-Approximation-factor

1. EXPERIMENTS

For the experiments we use **10k** queries with **60k** images from our dataset.

Original Dimensions of image 28x28 | Reduced Dimension of image 1x10

```
Query: 0
Nearest neighbor Reduced: 27059
Nearest neighbor LSH: 38620
Nearest neighbor True: 53843
distanceReduced: 6539
distanceLSH: 6731
distanceTrue: 5244

Query: 1
Nearest neighbor Reduced: 23468
Nearest neighbor LSH: 27535
Nearest neighbor True: 28882
distanceReduced: 14365
distanceLSH: 14040
distanceTrue: 10378

 ↓↓↓
 ↓↓↓

 ↓↓↓



Query: 9998
Nearest neighbor Reduced: 1311
Nearest neighbor LSH: 36407
Nearest neighbor True: 36407
distanceReduced: 12134
distanceLSH: 12057
distanceTrue: 12057

Query: 9999
Nearest neighbor Reduced: 22424
Nearest neighbor LSH: 22424
Nearest neighbor True: 22424
distanceReduced: 9603
distanceLSH: 9603
distanceTrue: 9603

Average time by query:
tReduced: 1.925 ms
tLSH: 102.869 ms
tTrue: 135.338 ms
Results:
Approximation Factor LSH: 1.09287
Approximation Factor Reduced: 1.38428
```

2. EXPLANATION OF RESULTS

We understand that the smallest dimensions of the images have a huge positive effect on time and a small negative effect on accuracy.

Part 3:

EMD-metric

1. EXPERIMENTS

For the experiments we use **10** queries with **100** images from our dataset.

Different cluster dimensions for EMD metric:

	CLUSTER SIZE: 4X4	CLUSTER SIZE: 7X7	CLUSTER SIZE: 14X14
MANHATTAN	Accuracy: **47%** Time: 0.21 sec	Accuracy: **47%** Time: 0.21 sec	Accuracy: **47%** Time: 0.21 sec
EMD	Accuracy: **50%** Time: 133.87 sec	Accuracy: **41%** Time: 62.4 sec	Accuracy: **26.9%** Time: 55.0 sec

2. EXPLANATION OF RESULTS

We understand that EMD is the most time consuming metric but we can achieve a good accuracy as we shrink the clusters (better resolution). MANHATTAN is by far the best choice because we have good time for accuracy.

Part 4:

Clustering-and-Classification

For this part we executed the classification.py with optimized autoencoder model **L4_FS3_FL32_E100_B40.h5**. The accuracy from classification with: **Layers 4.0 Fc_units 32.0 Epochs 100.0 Batch_Size 64.0** is: (t10k-images-idx3-ubyte for testing images)

```
Test loss:  0.0016728265909478068 Test accuracy:  0.9914000034332275
Found  9914  correct labels
Found  86  incorrect labels
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       980
           1       1.00      1.00      1.00      1135
           2       0.99      0.99      0.99      1032
           3       0.99      1.00      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.99      0.99      0.99       892
           6       1.00      0.98      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

The output of this program has the following format:

```
CLUSTER-1 {size:<int>, image_numberA, ..., image_numberX}
...
CLUSTER-K {size:<int>, image_numberK, ..., image_numberZ}
```

Then the output file is given as input for cluster.cpp.It is executed as:

```
./cluster -i <input file new space> -d <input file original space> -n
<classes from NN as clusters file> -c <configuration file> -o <output
file>
```

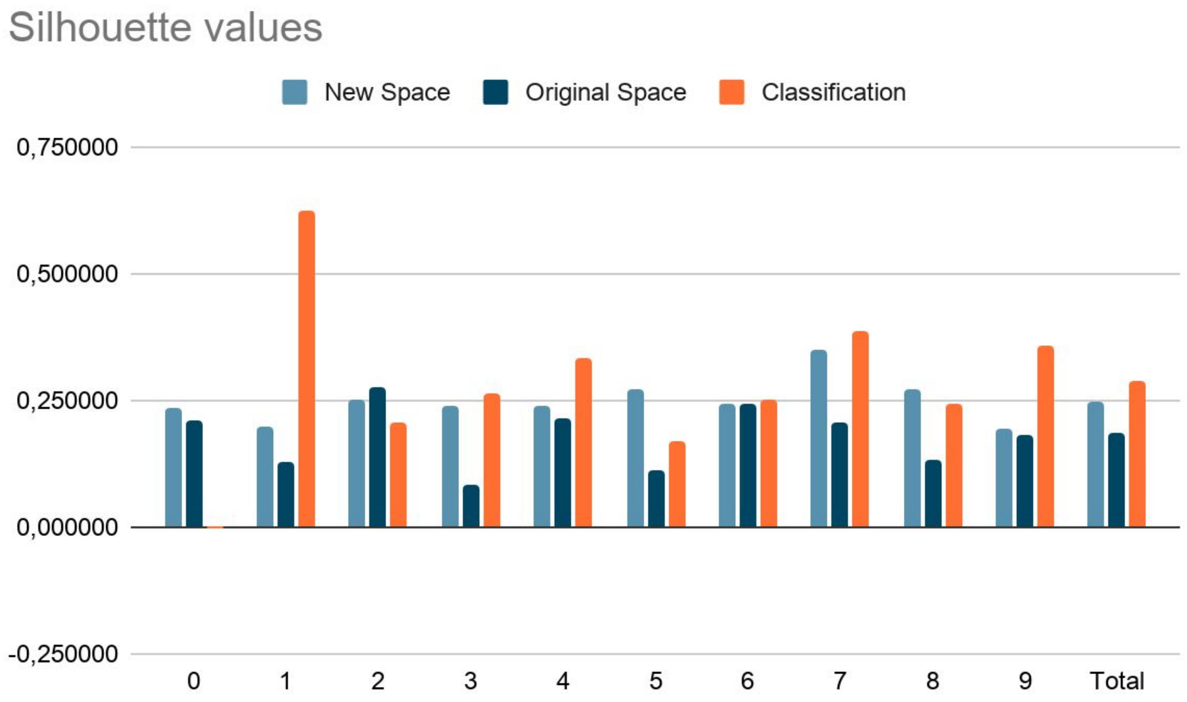
In cluster.cpp, we group S1 k-medians of the images of the input set in the new space and S2 of the input set in the original space. We use the `<classes from NN as clusters file>` to categorize the images of the input set and clustering S3 based on it. Then, in cluster.cpp (from exercise 1) we compare the three formulations in terms of silhouette and its evaluation target function in the initial space (k ~ 10) with Manhattan metric. In the end, the user has the option to save a file with format: <IMAGE_INDEX> <LABEL_FOUND> <CORRECT_LABEL>, which contains the found and the correct labels of each clustering. The user can execute *python labelgraph.py [label_file]* for label-graph.

1. EXPERIMENTS

From output file (executed for 10k items) of cluster.cpp we get the value of objective function (Sum of the minimum distance between Centroids and Query-images)

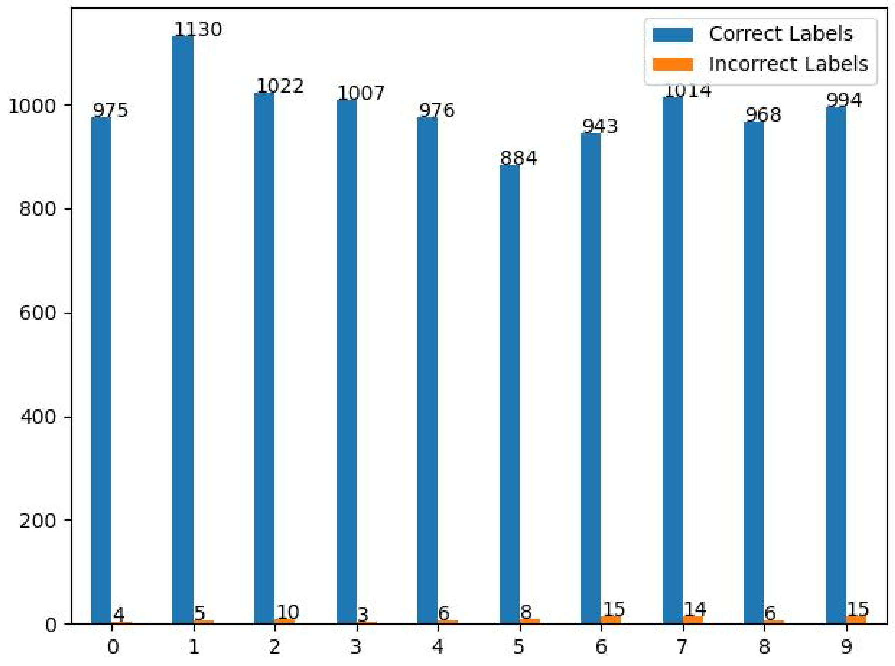
NEW SPACE	ORIGINAL SPACE	CLASSES AS CLUSTERS
235.852.000	**190.824.000**	**193.958.000**

Also, we get and visualize the Silhouette values for C1(New Space), C2(Original Space), C3(Classification)



Finally, we executed *python labelgraph.py [label_file]* for classification results in order to check the correctness of labels.

- **CLASSIFICATION:**



2. EXPLANATION OF RESULTS

From the experiments we observe that for the objective function , S2(Original Space) has the best results. On the other hand, S3(Classification) has better values for the silhouette method (a measure of how similar an object is to its own cluster compared to other clusters), followed by S1(New Space) and S2. The S1 clustering has very low dimensions(1x10) compared to the S2 dimensions(28x28)and this negatively affects the finding of the "correct" centroid resulting in a much worse value objective function(images are at much more distance as accuracy has been lost during compression). We cannot compare the results of label correctness with those of Silhouette and Objective function, because classification clustering is a result from Neural Network with 99% accuracy while the calculation of Silhouette and Objective function are based on distance of the vectors. For this reason, S1 and S3 have many incorrect labels. Eg. S1 confuses 8 with 0 and 5 or 6 with 3.

```
INDEX     LABELS     CORRECT-LABELS
110        0          8
127        0          5
128        0          8
...
3550       3          6
3558       3          5
3622       3          6
```

In conclusion, S3(Classification) has the best clustering results but it is the most time consuming of the rest and when we use a wide range of dataset, it can exceed 1 hour of execution.