Project 2 PML – Unsupervised Learning

Author: Marinescu Alexandru

Group: 407

Table of Contents:

1. Introduction

This is the documentation for using **unsupervised learning** methods on a classification problem.

The purpose isn't to replace supervised methods, but to gain additional insights about the data and separate them in clusters close to the original scenario, thus giving better representation of data.

For this project, a couple of rules stand in place:

the dataset must have at least 1000 data samples
if features are tabular, it must have at least 100 features (this does not apply to image, text, audio)
must not be a popular dataset (MNIST, CIFAR, Iris, Movie Reviews, etc.)
dataset must have labels for interpretability / comparison with supervised method
at most 10 students can choose the same model combination
at most 20 students can choose a certain model

It's important to also mention these here, to give full context of the project's setup. Thus, **HDBSCAN** and **Agglomerative Clustering** were the methods of choice for unsupervised learning. Along with these, there will also be a comparison with *random choice* and some supervised methods for baseline. For the latter, the metric choice is important, because unsupervised learning methods, don't classify the data into novel classes, as mentioned in the start. For this, an interpretation is needed.

And **Disaster_Classification_Dataset** is the dataset of choice and you can find it on *kaggle* datasets: https://www.kaggle.com/datasets/sayankr007/disaster-classification-dataset, a public and simple one

2. Related Work

In [1] ***Comparative Review of Supervised vs. Unsupervised Learning in Cloud Security Applications*** a comparative analysis of supervised and unsupervised learning methods is done.

This comes in the context of cloud security, for textual and file data, which is different from this project. Still, the paper showcases the strengths and weaknesses of each of the approaches given the topic. While the *unsupervised methods* are better for exploratory data analysis and can detect outliers (anomalies and unusual patterns), they note that the results can be difficult to interpret and data preprocessing is essential, as we'll see in this project.

While trying to find models pre-trained using unsupervised learning methods for feature extraction, I stumbled on [2] ***A Simple Framework for Contrastive Learning of Visual Representations*** a paper that presents SimCLR and compares it to state-of-the-art encoders. This is a pipeline for contrastive learning of visual representation, which bases on 4 major components:

- a stochastic data augmentation module that gives 2 correlated samples from same data
- a neural network base encoder that extracts representation vectors from data
- a neural network projection head (maps data to space of contrastive loss function)
- a contrastive loss function

As opposed to this, data won't be augmented in this project (it will be explained in another section). But this gives a very good insight of what to expect for good data representation.

3. Method

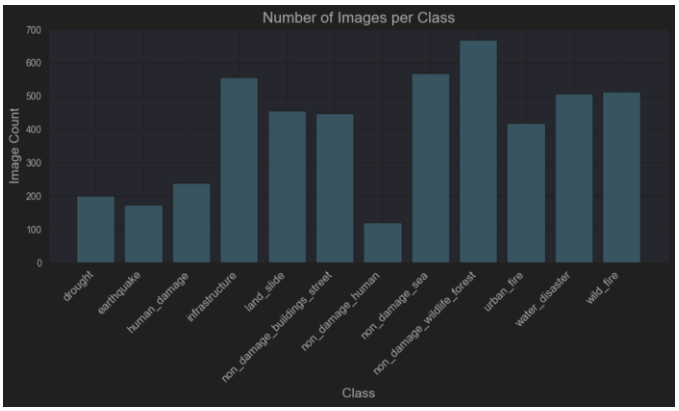
Dataset

As mentioned in the introduction section, the chosen dataset is **Disaster_Classification_Dataset.**

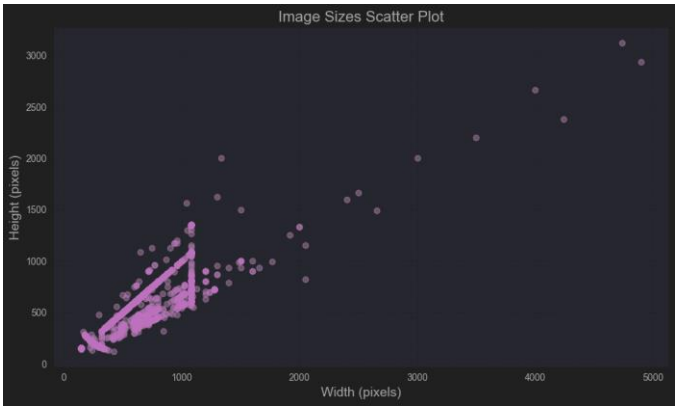
It's a relatively small size image dataset of 12 class disasters: ['drought', 'earthquake', 'land_slide', 'human_damage', 'infrastructure', 'water_disaster', 'non_damage_buildings_street', 'wild_fire',

'non_damage_human', 'non_damage_sea', 'non_damage_wildlife_forest', 'urban_fire'].

From this I started to explore the data. First thing that was noticed is that the set is **imbalanced**. This is the sample distribution per class:



Lowest is 120 samples and highest is 670 samples. Next, I also observed that image size varied quite a lot:



By taking an average of all image sizes we obtain a *common image size* of (433, 382). However, I went if an official common image size of **(200, 200)**, because [3] kernels and pooling operations are easier work on square matrices, but also because I didn't have the local computational power to load larger data onto the GPU (and didn't want to use the CPU, or cloud solutions). We'll skip the details, since this is also discussed in the *limitations* section

Common Preprocessing

It didn't take too long to realize that the given data was already augmented. A quick look from a folder:

These 2 photos are coming from 'earthquake' class folder. There are more like this scattered through the dataset, so there is no need to further augment the data and create needless correlation.

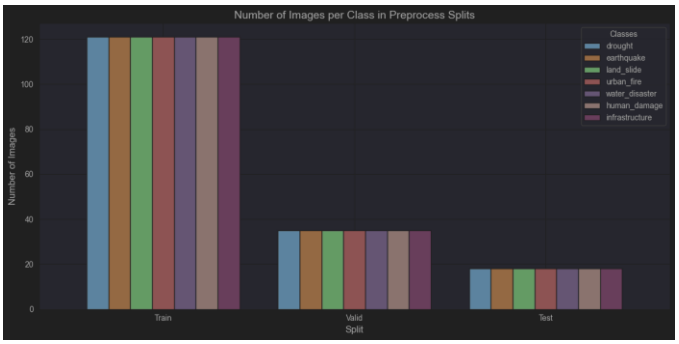
Second is to decide how many classes to keep. The dataset is quite small, and the data isn't balanced, or of high quality, so the chances to obtain good results are low if we keep all classes. However, we have to abide to the project's rules regarding total size and sample size.

I excluded the `non_damage_human` class, and the next one with the fewest samples was `earthquake` with *174* images. I wanted to eventually balance the classes, thus use a *common image count* for all, so I'm left with 1000 / 174 ~ 5.74 -> a minimum of 6 classes. I chose the following 7 classes:

- "drought"
- "earthquake"
- "land_slide"
- "urban_fire"
- "water_disaster"
- "human_damage"
- "infrastructure"

Besides this, as mentioned above, classes were balanced to the minimum between them, which is 174 images / class. Images have been resized to a common size of (200, 200) as mentioned before.

Afterwards, all data was split in training, validation and testing sets with a split of 0.7, 0.2 and 0.1 for each set. This data will be used by all methods.



Supervised Scenario

Neural Network

Using a ***CNN*** seems an obvious choice to start with in a multiclass image classification problem. Data was prepared using *image_dataset_from_directory* function from **keras** API in batches of 32.

Afterwards, pixel values were rescaled by 1/255 for **Normalization** (layer was applied to all datasets).

Many possible architectures were tried. However, because of time issues, hardware limitations and the fact that this is supposed to be a baseline for unsupervised methods, this is the end result:

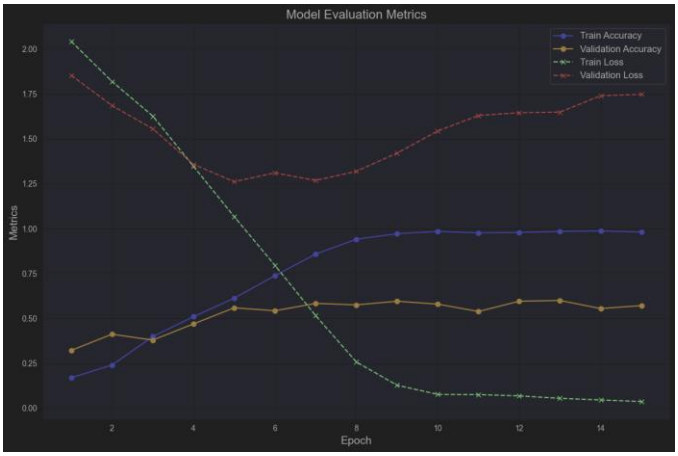
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 200, 200, 32)	416
max_pooling2d (MaxPooling2D)	(None, 100, 100, 32)	0
conv2d_1 (Conv2D)	(None, 100, 100, 32)	4128
max_pooling2d_1 (MaxPooling2D)	(None, 50, 50, 32)	0
lstm (LSTM)	(None, 256)	82183168
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 128)	32896
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 7)	903

Stacking more *convolution* layers was the hope of obtaining more visual features (embeddings) from data. But more than 2 got into overfitting.

That's when *LSTM* (Long-Short Term Memory) layer was added. Results have improved and while disaster images might not have explicit temporal information, the LSTM layer processes sequential relationships in extracted features, potentially improving class separation.

After averaging the results in a single vector, the general technique of using a *Dense* layer at the end proved useful (in the hope of generalizing more and reduce overfitting, *Dropout* was introduced).

ADAM was the choice for optimizer and for the loss function *sparse_categorical_crossentropy* was used



The above results are obtained after 18 epochs with a batch size of 32. You can observe that the accuracy for validation data slowly converges to 0.6, which isn't particularly great. That being said, the results on testing are in the Evaluation section.

Random Forest

After that, I wanted to also attempt some machine learning methods, in the hope of improving the baseline. **Random Forest** was first, but before, I had to load the data. It's essentially the same preprocessing as done for *CNNs* but with a custom function instead of using the keras API: load into sets, normalization and label encoding.

The implementation from **sklearn** was used (*RandomForestRegressor*) and *GridSearchCV* method from **sklearn** was utilized to obtain the best parameters. This will try all possible combinations of given hyperparameters and does cross-validation for performance between each

combination. For this specific search, the following setup was used:

- RandomForestRegressor as the model
- 'n_estimators': [50, 100, 200]
- 'max_depth': [None, 10, 20, 30]
- 'min_samples_split': [2, 5, 10]
- 'min_samples_leaf': [1, 2, 4]
- 'accuracy' as scoring
- the vectorized training data features for fit

Best parameters were: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 5, 'n_estimators': 100}, with an accuracy of 0.4629.

Support Vector Classifier

There was also the expectancy that defining hyperplanes for each class margin would do better than CNNs, which would require more data. In theory, this would benefit better from the smaller dataset.

For the **sklearn** implementation *SVC, GridSearchCV* method was used to find the best combination, in:

- SVC as the model
- 'C': [0.1, 1, 10, 100]
- 'kernel': ['linear', 'rbf']
- 'gamma': ['scale', 'auto', 0.01, 0.001]
- 'accuracy' as scoring
- the vectorized training data features for fit

Best parameters were: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}, with an accuracy of 0.4581.

Unsupervised Scenario

Since initial data representation was so important, for the sake of exploration, 3 methods were chosen for preprocessing:

Neural Network Encoder

This was my first time using autoencoders with TF, so I had to [3] search a little. Compression ideas have existed before, but the whole point here is to compress and reconstruct data by learning some parameters based on a loss function (L1, or L2 loss like *mse*). However, it's not going to be a classical neural network, but just a pipeline that user other structures (TF layers in this case) to change data.

An **Encoder** is used for feature extraction and dimensionality reduction into a meaningful representation (embeddings). Architecture used:

Layer (type)	Output Shape	Param #
Input_layer_1 (InputLayer)	(None, 200, 200, 3)	0
conv2d_3 (Conv2D)	(None, 200, 200, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 100, 100, 32)	0
conv2d_4 (Conv2D)	(None, 100, 100, 64)	18496
max_pooling2d_3 (MaxPooling2D)	(None, 50, 50, 64)	0
flatten_1 (Flatten)	(None, 160000)	0
dense_2 (Dense)	(None, 256)	40960256

Autoencoder is formed starting with the above encoder in order to learn and decode. Attached to the above is:

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 160000)	41120000
reshape_1 (Reshape)	(None, 50, 50, 64)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 50, 50, 64)	36928
up_sampling2d_2 (UpSampling2D)	(None, 100, 100, 64)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 100, 100, 32)	18464
up_sampling2d_3 (UpSampling2D)	(None, 200, 200, 32)	0
conv2d_5 (Conv2D)	(None, 200, 200, 3)	867

Some other options were attempted, but not too many here. Final shape of training embeddings is (1092, 256), so a 256 length vector per sample.

Principal Component Analysis

PCA (Principal Component Analysis) is a statistical technique used for dimensionality reduction while retaining as much variability (information) as possible in the dataset. It transforms the original features into a smaller set of uncorrelated variables called **principal components** (eigenvectors of eigenvalues of the larger correlation matrix of data)

1. Data has to be centered
2. We calculate the covariance matrix of data
3. Compute eigenvalues and eigenvectors
4. We sort eigenvectors by corresponding eigenvalues in descending order
5. Have to project the dataset onto a vector space defined by eigenvectors basis

The implementation from **sklearn** was used (*PCA*), with 256 components to mimic the other earlier method (later I found out that less components can provide better results -> probably curse of dimensionality). Features shape is still (1092, 256).

Pre-trained model (SIMCLR)

As mentioned in the *Related Work* section, **SimCLR (Simple Contrastive Learning of Representations)** is an unsupervised learning framework designed for learning representations of data, images to be precise. This works within the project requirements

However, I couldn't load it using my local setup, so I started with a **ResNet50** backbone and tried to do something similar (I understand if this is no longer within project requirements):

Layer (type)	Output Shape	Param #
input_layer_13 (InputLayer)	(None, 200, 200, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23587712
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_13 (Dense)	(None, 512)	1049088
activation_6 (Activation)	(None, 512)	0
dense_14 (Dense)	(None, 256)	131328
activation_7 (Activation)	(None, 256)	0
dense_15 (Dense)	(None, 128)	32896

This time, embeddings dimensions were lowered to 128, since earlier attempts on 256 weren't that successful. The embeddings shape is (1092, 128).

HDBSCAN

HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) extends *DBSCAN* which is a clustering algorithm, by adding hierarchical capabilities.

Whereas DBSCAN groups features that are closely packed together (density metric), HDBSCAN builds a hierarchy of clusters, merges based on a density metric relative to each other and selects the most stable configuration as final output.

The implementation from **hdbscan** was used (*hdbscan.HDBSCAN*). Using 2 repetitive structures, we found the best options for 'min_cluster_size` ([2, 3, 5, 10, 20, 50]) and 'min_samples' ([1, 2, 5, 10, 15]) parameters in accordance to *silhouette* measure. It provides a measure of how similar each point is to its own cluster compared to other clusters. The score for a point *i* is calculated by:

$$s_i = \frac{b_i - a_i}{\max(b_i, a_i)} \; where \begin{matrix} a_i - \; same\; cluster\; distance \\ b_i - nearest\; clusster\; distance \end{matrix}$$

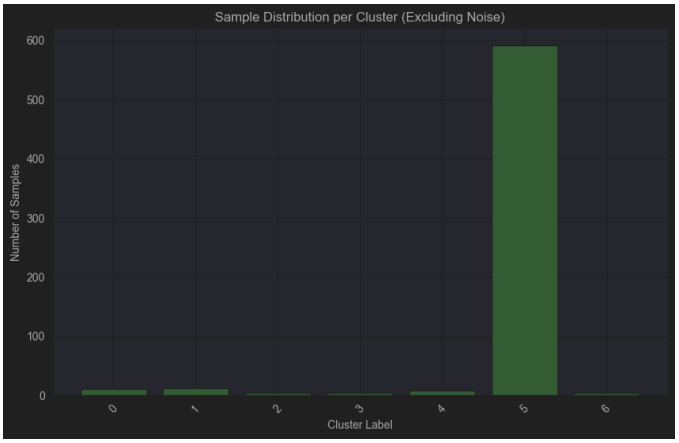
And the actual silhouette score is an average sum of the score of all points, which ranges from **-1 to 1**.

- closer to 1 means points are well clustered
- closer to 0 means points are on decision boundary
- closer to -1 means points are misclassified

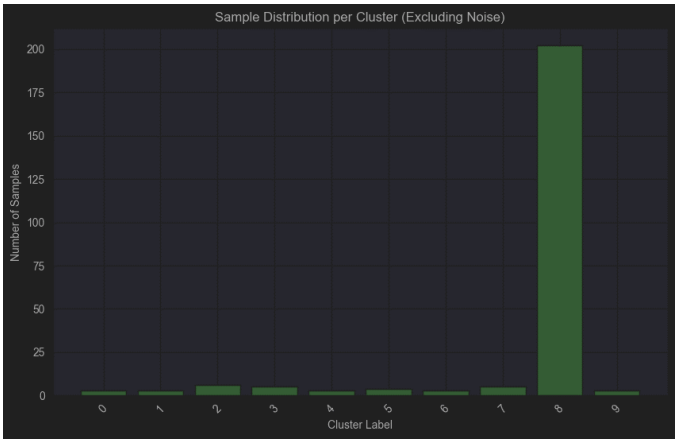
Unfortunately, most likely because of data representation and sample count in general, all methods had the best score for 2 clusters. In opposition to this, I chose to keep the parameters that give a cluster count closer to the true value (7)

Preprocessing Method	min cluster size	min samples	silhouette score
Custom CNN	5	2	-0.003389
PCA	3	2	-0.168647
Custom Pretrained SIMCLR	5	1	-0.153076

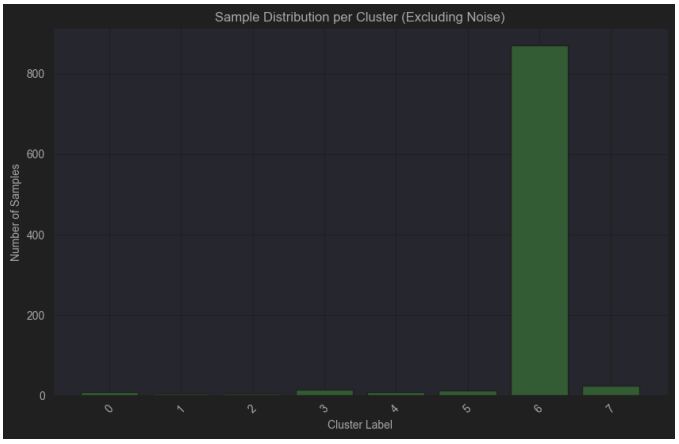
HDBSCAN + Custom CNN Encoder Embeddings



HDBSCAN + PCA Embeddings



HDBSCAN + Custom SIMCLR Embeddings



These are training data results. For evaluation on testing dataset, check the Evaluation section below

Agglomerative Clustering

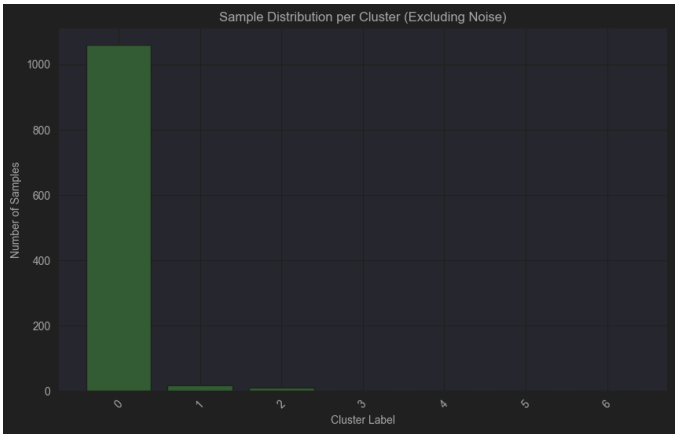
Agglomerative clustering is a type of hierarchical clustering that builds a hierarchy of clusters in a *bottom-up approach*. Initially, each data point is treated as its own cluster, and the algorithm iteratively merges the closest clusters until all points are grouped into a single cluster or a desired number of clusters is achieved.

The implementation from **sklearn** was used (*AgglomerativeClustering*). In order to not be totally biased, using 2 repetitive structures, we found the best options for ‘n_clusters` ([2, 3, 4, 5, 6, 7, 8]) and ‘linkage (['ward', 'complete', 'average', 'single']) parameters in accordance to *silhouette* measure.

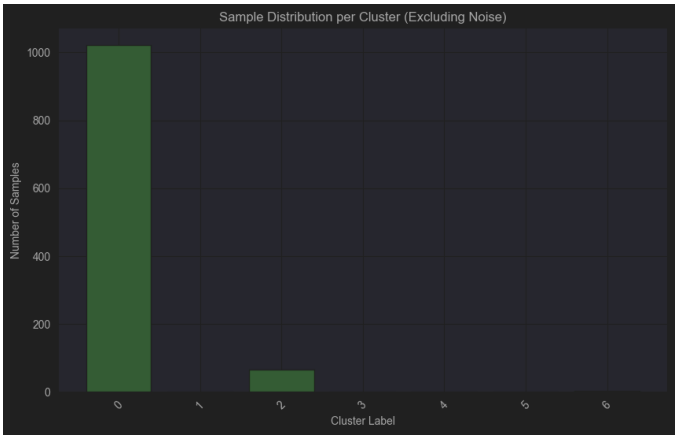
Just like the previous case, you will observe the tendency to organize data in just 2 clusters, so for fair comparison, I kept the parameters for the true value of cluster count (7).

Preprocessing Method	n_clusters	linkage	silhouette score
Custom CNN	7	average	0.379417
PCA	7	average	0.152544
Custom Pretrained SIMCLR	7	average	0.403244

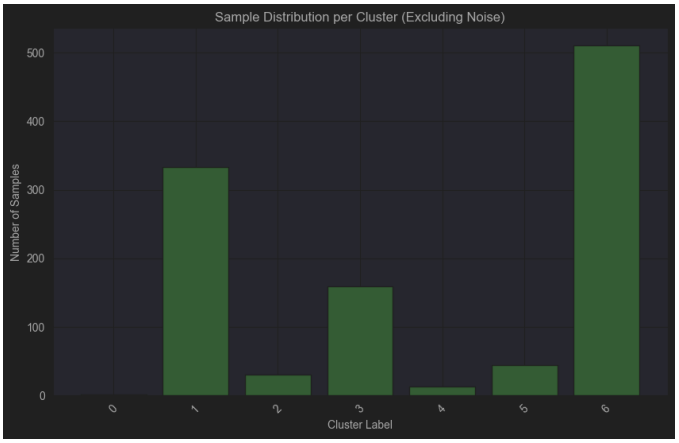
Agglomerative clustering + Custom CNN Encoder Embeddings



Agglomerative clustering + PCA Embeddings



Agglomerative clustering + Custom SIMCLR Embeddings



These are training data results. For evaluation on testing dataset, check the Evaluation section below

Random Choice

This has been added as a project requirement for comparison with the other 2 unsupervised learning methods. Basically, for each training sample, a random label integer (0, to 6) is associated.

The comparison will be done with the rest on the testing data.

Evaluation

For the supervised methods, we have the following accuracy metrics on testing dataset:

Method	Accuracy
CNN	0.6507
Random Forest	0.5079
Support Vector Classifier	0.4921

For the unsupervised methods, I also wanted to add some evaluation metrics that involved the true labels as well for the testing data. Thus, besides *silhouette score*, **adjusted rand index (ARI)** and **normalized mutual information (NMI)** were added

This was a last minute addition, and I'm not totally sure what they mean, besides the fact that ARI should measure the similarity between the true labels and clustering and NMI quantifies how much information is between clusters and true labels.

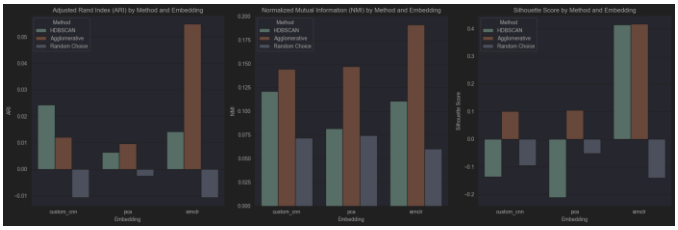
ARI score also ranges from -1 to 1 and:

- 1 means clustering is identical to labels
- 0 means clustering is as good as random
- -1 or closer indicates worse than random

NMI score ranges from 0 to 1 and:

- 1 means clustering is identical to labels
- 0 means no mutual info between clustering and true labels

These are the results for the testing dataset. From a technical point of view, there was no need to take the random values 3 times for each method of embedding, but it was easier to put it all together and the results don't vary as much to cause issues



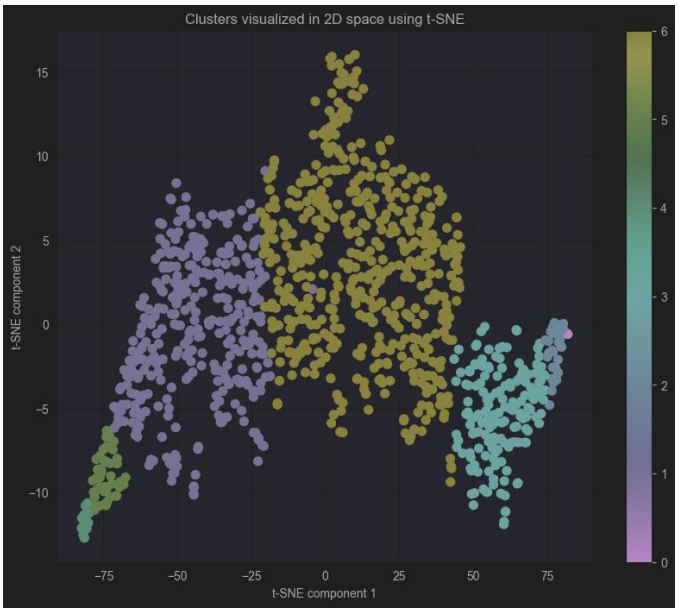
Method	Embedding	ARI	NMI	Silhouette
HDBSCAN	custom cnn	0.024270	0.120839	-0.137064
Agglomerative	custom cnn	0.012007	0.144264	0.099823
Random Choice	custom cnn	-0.010642	0.071637	-0.095237
HDBSCAN	pca	0.006430	0.081609	-0.211244
Agglomerative	pca	0.009610	0.147191	0.104716
Random Choice	pca	-0.002548	0.074286	-0.051876
HDBSCAN	simclr	0.014127	0.110588	0.413573
Agglomerative	simclr	0.054797	0.191327	0.415850
Random Choice	simclr	-0.010662	0.060332	-0.140808

4. Conclusion

Evaluation Results

Technically, the best unsupervised scenario was obtained using **Agglomerative Clustering** on data embeddings obtained from the **SIMCLR** inspired encoder. This is better than any results of *random choice*, or *HDBSCAN* and it can also be seen on the bar plots in the previous section.

I tried to reduce the dimensions of the feature to 2D in order to plot a representation of the clustering obtained via the methods, and the above was the most concise. Training data result:

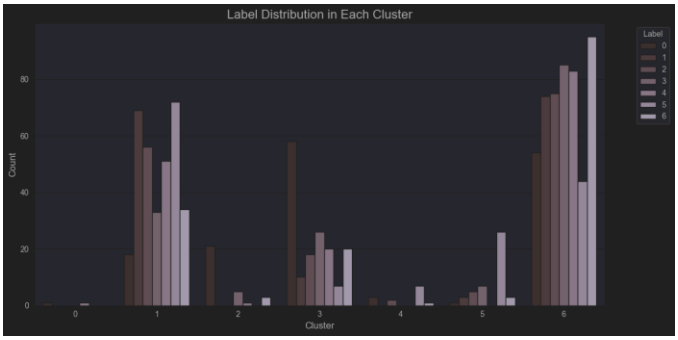


That being said, results aren't good in general. Not only this, but data was balanced and so should the cluster size be, but this isn't the case anywhere.

I believe that **supervised** methods did better in the context of multiclass disaster image classification problem. I'm saying this, because I tried to get an interpretation for clusters given by the best case, by majority in the clusters and this is the result:

{0: 4, 1: 5, 2: 0, 3: 0, 4: 5, 5: 5, 6: 6}

This is for the training data and this isn't the whole story, because of distribution of labels per cluster:



This indicates that an interpretation is difficult to be obtained and even the one given by *majority per cluster* method isn't reliable.

Most likely, better results could have been obtained by having better structured embeddings.

But, unsupervised methods excel at feature extraction and could be used in other scenarios, such as *augmenting data*, *finding outliers*, etc. Obviously, solving classification problems is also very beneficial since it could help in automation of labelling unknown data.

Limitations

One of the limitations was the dataset itself, in the sense that sample size was a bit small (and already augmented), but more importantly, classes were unbalanced and images varied in resolution quite a lot. I enjoyed it as a challenge, but for a real case scenario, a better dataset is needed.

Even now, I'm not entirely sure if I did something wrong in the embeddings section of the project, or if the data just wasn't rich enough. I think it's a combination of both, leaning more towards myself.

Lastly was the fact that I wanted to do everything locally and got limited by hardware. For example, this is why I couldn't use a higher common image size, because GPU didn't have enough memory to load everything in (or had to lower the batch size to 8). Grid search also took some time with my CPU

In the end: it was a nice learning experience.

5. Bilbiography

[1]. Kharbanda, N. S. (2024). Comparative review of supervised vs. unsupervised learning in cloud security applications. *International Research Journal of Engineering and Technology (IRJET)*, *8*(1), 186

[2]. Chen, T., Kornblith, S., Norouzi, M., & Hinton, G. (2020). A simple framework for contrastive learning of visual representations. *Proceedings of the 37th International Conference on Machine Learning (ICML)*, *PMLR*, 119, 1597–1607.

[3]. mrgrhn. (2021, January 24). *AutoEncoders with TensorFlow*. Analytics Vidhya. https://medium.com/analytics-vidhya/autoencoders-with-tensorflow-2f0a7315d161

[4]. Delua, J. (2021, March 12). *Supervised versus unsupervised learning: What's the difference?*. IBM. Retrieved from https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning

[5]. Berba, P. (2020, July 8). *A gentle introduction to HDBSCAN and density-based clustering*. Towards Data Science. Retrieved from https://towardsdatascience.com/a-gentle-introduction-to-hdbscan-and-density-based-clustering-5fd79329c1e8

[6]. Benhur, S., & Whitfield, B. (2024, October 16). *Hierarchical clustering: Agglomerative and divisive clustering.* Built In. Retrieved from https://builtin.com/machine-learning/agglomerative-clustering

[7]. Ionescu, R. (2024–2025). *Practical Machine Learning*. Faculty of Mathematics and Computer Science, University of Bucharest., All courses, https://practical-ml-fmi.github.io/ML/