Name: Daniel Sociu
Group: 407

Documentation PML
Unsupervised Clustering

Data

The dataset used is called Body performance data and is obtained from the kaggle website at https://www.kaggle.com/datasets/andrewmvd/animal-faces. The dataset consists of 16,130 total images of dogs, cats and wildlife, which have to be classified in the same 3 classes. In the validation set there are 500 of each class and the rest of the images are included in the train set, which is fairly balanced. The input photos, which have a resolution of 512x512 and are of extremely high quality, needed to be converted from the image format to an array-like structure using feature extraction so that the chosen clustering models can be applied.

Models

There are two main clustering models I have used for this project, which are DBSCAN and Agglomerative Clustering, both were imported from the sklearn library. Both are used for clustering a given set of vectors by distance, and don't have any ability to learn (they have no weights), therefore the best approach is to find the best parameters which generate some labels for each class given the train set. With the new obtained labels we have to use them to predict the validation dataset and calculate the model accuracy. The main difference in the models' api is that Agglomerative Clustering has the extra option of choosing the number of total desired classes (didn't use that).
DBSCAN works by clustering the points by using neighbors. The points have to be at a maximum of eps distance from each other to be considered neighbors, and a group of neighbors are considered the same class if the size of the cluster is at least min_samples big.
AgglomerativeClustering works by linking classes/clusters together, given a specific distance-threshold, any clusters within the distance will be linked, using one of its linkage algorithms.

Feature extraction

There are two main approaches which transform the input images in a format that can be used with the models described above. Both of them output a single array of features for each image.

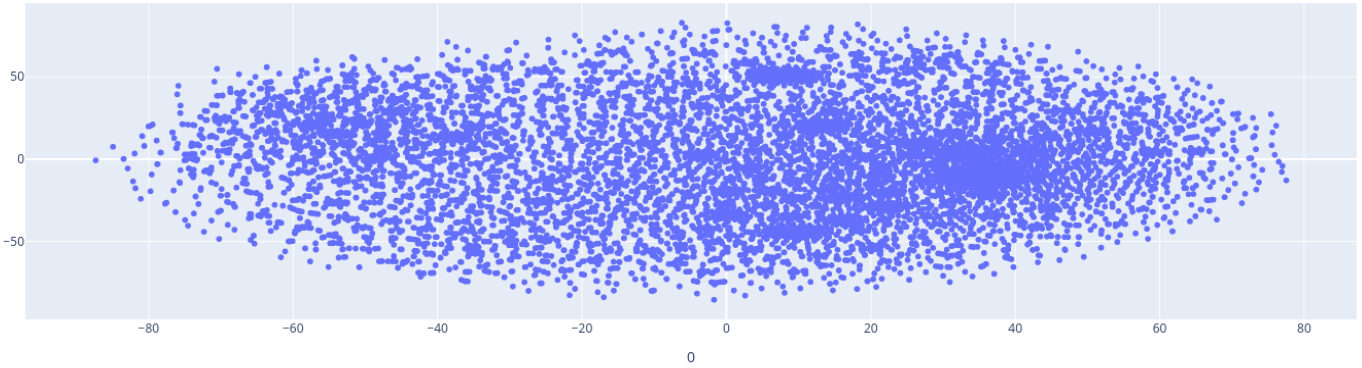
ORB (Oriented FAST and Rotated BRIEF)

The ORB algorithm is a fusion of FAST keypoint detector and BRIEF descriptor with a few improvements and modifications (also it is not patented, therefore free to use). It improves the BRIEF descriptor by adding rotation to the descriptors, the result being called rBRIEF.
For the ORB type feature extractor, we apply the ORB algorithm on each image, extracting a maximum of 100 features. We linearize these descriptors to obtain the image features, which are

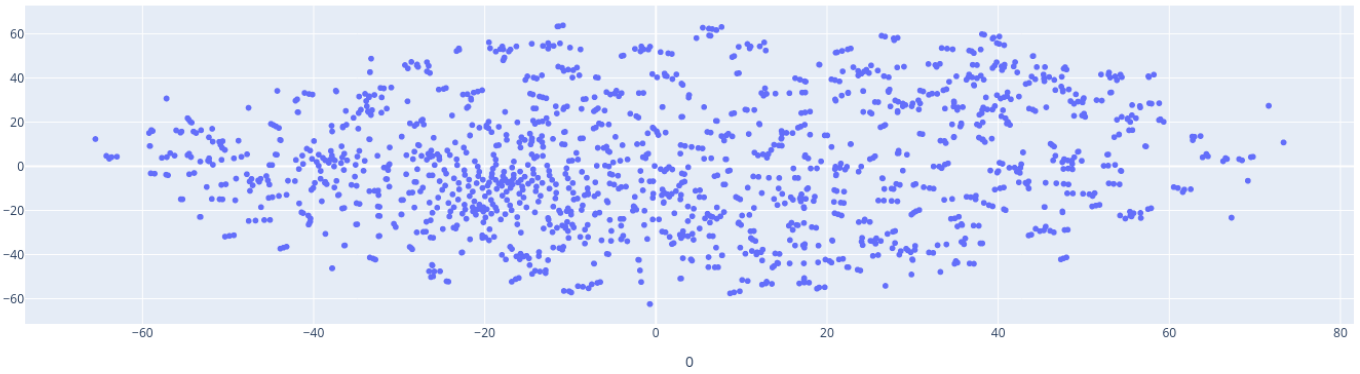
of length 3200 (we pad the ones that didn't meet the minimum length). We scale them to 0-1 and then optionally apply the standard scaler (standard scaler didn't improve the results).

Here I plotted the data with the help of T-SNE which reduces the dimensionality of the data so it can be visualized (initially I used PCA, but it didn't keep the distance between the points well enough)

- Train set:



- Val set:



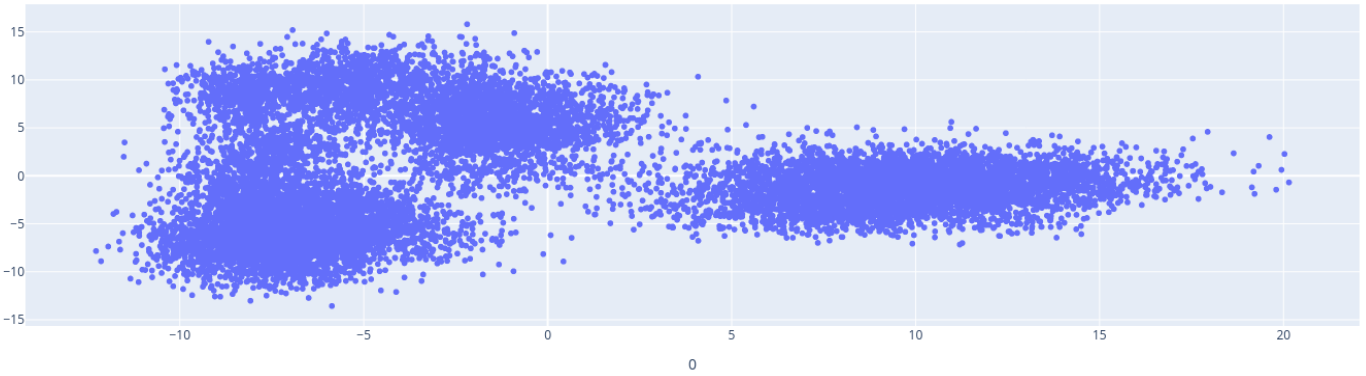
Unfortunately with this feature extraction approach, there couldn't be observed any big three clusters of the data.

Resnet18 feature extractor

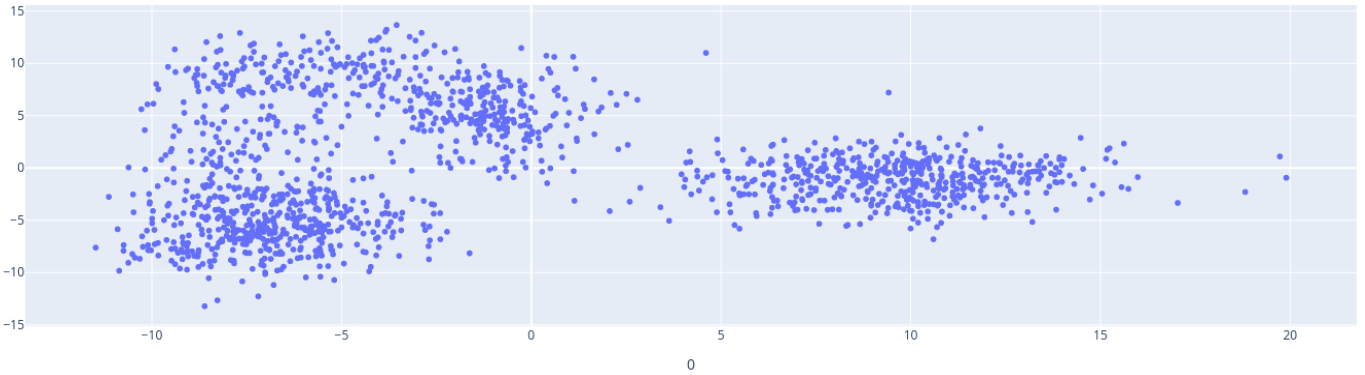
The second approach used is based on the well known resnet architecture, the resnet18 model. I imported the model from the torchvision library, and I loaded the model with the ImageNet1k pretrained weights.
I removed the head or the classifier layer (last layer) of the resnet18 model, which reduced the output from 1000 to 512 values. Therefore after running each image through the model, the output feature vector was used to later on apply the clustering models on them.

This approach turned out to be much better than the previous one, which can be seen in the following plots (here I used the PCA algorithm to reduce the dimensionality of the data for visualization)

- Train set:



- Val set:



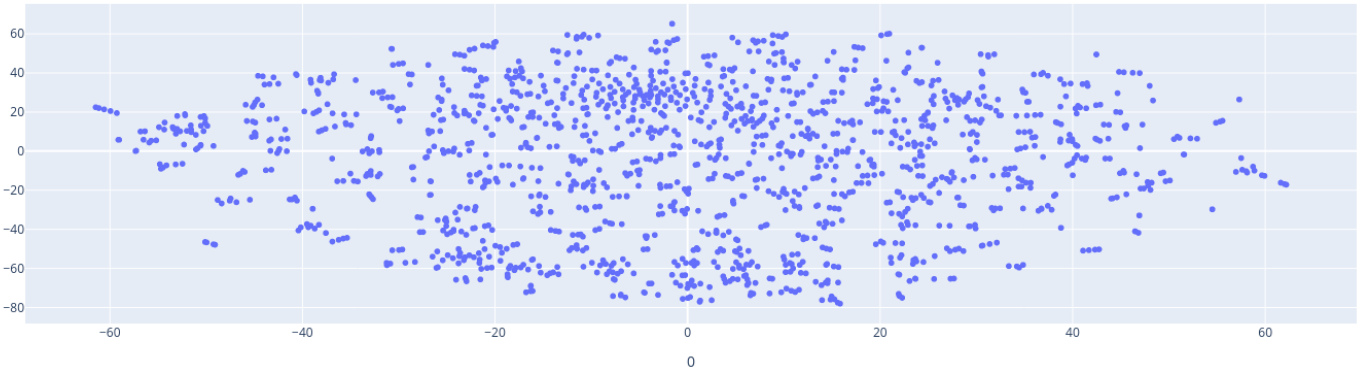
Using this approach the three class clusters can be easily observed.

Training trick applied to data

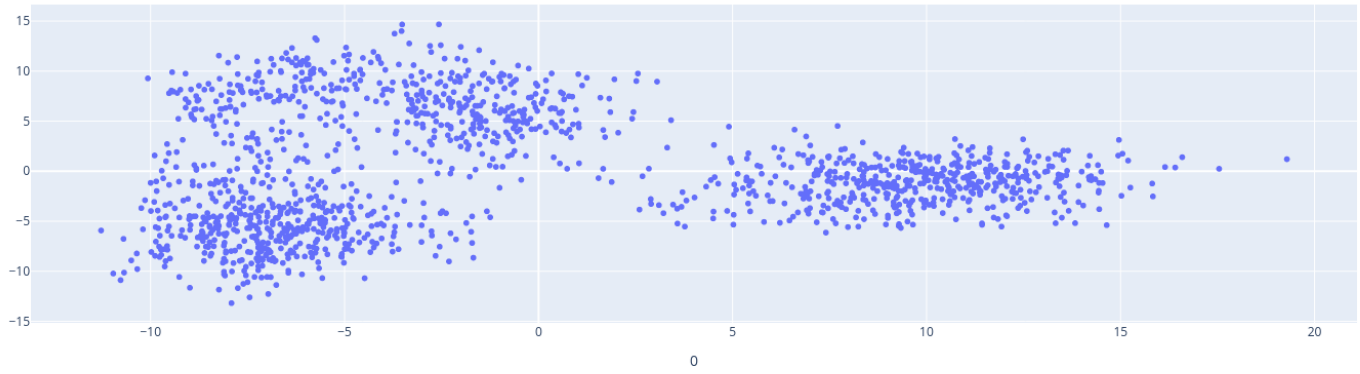
After looking at the graphs, and knowing how the labels are generated, I realized that using more data on the train can generate the wrong labels for the validation dataset therefore I decided to reduce the train dataset to have the same number of samples as val:

NOTE: This sampling was done from the total data, without taking in consideration which class was the data from (basically I didn't use the labels for it, which would invalidate the correctness of approach)

- ORB:



- Resnet:



Training

Once the data was obtained, the training was can be done in two main ways:

1. With grid-search (which finds the best parameters)
    a. First I find the best optimal starting points of distance for both algorithms.
    b. Then I do a grid search (for through parameters) with a step size obtained in an empirical manner.
    c. I uploaded the grid search outputs to github: https://github.com/danielsociu/clusters-comparison/tree/master/outputs
2. Just best models only (with the parameters found by grid-search)

Class matching

Once we fit one of the clustering models on the train set, we will obtain some labels, which correspond to some classes. We have to map these classes to the actual classes, so that we can use these mappings to obtain the validation results. I did this by simply selecting the class with the most matches of the “fake labels” with the real class and assigning that fake label to the class label.
Once this step was done, the calculation of the accuracy became trivial.

Predicting

Having the class matching labels, I predicted the on the validation dataset and used the mapping to map to the proper classes, and anything that wasn’t the proper class I set the prediction to -1.
After that I just used the accuracy_score class from sklearn.metrics.

Random and supervised

The random predictions didn’t use the dataset at all, but only for the calculation of the score. The supervised one, used the same feature extractors to obtain the features of the images, upon which I trained a SVM classifier and then predicted these values. The supervised approach obtained the best score, which was also expected.

Results

The results include the best ones found by the grid search (in the case of unsupervised approaches) and just the default ones for the random and supervised.
The best params are:

- DBSCAN: first is eps, second is min_samples
- Agglomerative: just distance_threshold

Model	**Feature Extractor**	**Train samples**	**Data length**	**Train acc**	**Val acc**	**Best parameters**
Random	-	-	-	-	0.361	seed=42
DBSCAN	ORB	1500	3200	0.35	0.33	22.52, 5
DBSCAN	RESNET	all	512	0.351	0.338	16.31, 5
DBSCAN	RESNET	1500	512	0.462	0.441	13.12, 24
Agglomerative	ORB	1500	3200	0.462	0.482	60.4
Agglomerative	RESNET	all	512	0.589	0.666	343.87
Agglomerative	RESNET	1500	512	0.823	**0.674**	176.16
SVM	ORB	all	3200	0.946	0.672	-
SVM	RESNET	all	512	0.999	**0.998**	-

Conclusion

The DBSCAN performed worse than the Agglomerative model, although both had a similar environment of grid search, and the train data were the same. Even with these feature extractors, it was hard to map the images in a space such that they have good distances so that they can be clustered easily. They had to be more feature engineered such that the distance between the different classes was bigger.
On the other hand, supervised learning can easily learn these features, which means it can increase the distance between the classes by learning the classes and their spaces.

Code

There is only one main script for running all the experiments, which works with terminal arguments. Github: https://github.com/danielsociu/clusters-comparison

```
usage: unsupervised_clustering.py [-h] [--plot] [--verbose]
[--grid-search] [--limit-train] [--standard-scaler] [--type
{DBSCAN,AGLOMERATIVE}] [--feature-type {resnet,orb}]
                                  [--mode
{unsupervised,random,supervised}]
```

```
optional arguments:
  -h, --help            show this help message and exit
  --plot                Whether to plot stuffs
  --verbose             Whether to print extra info
  --grid-search         Whether to use a grid-search strategy
  --limit-train         wether to limit train to same amount of data as
val
  --standard-scaler     whether to scale the orb output
  --type {DBSCAN,AGLOMERATIVE}
                        Type of model to train
  --feature-type {resnet,orb}
                        Type of feature engineering
  --mode {unsupervised,random,supervised}
                        Type of training
```