# CS 6170 - Computational Topology
# Project 2
## Main Report

Yash Gangrade (u1143811), MS in Computing 23rd March 2019

## Contents



## List of Figures

# 1 Computation and Comparison of Barcodes

## 1.1 Selection of dataset

Here, we had to chose 10 images from 8 classes so in total of 80 synthetic images. I am using the following list of images. The list of chosen classes are, Apple, Car, Cellular Phone, Children, Cup, Fork, Hammer, and Pencil. We chose the first 10 images from each of these datasets.

## 1.2 Computation of Point cloud and Sampling

I am using the same method as used in Project 1 to compute the boundary of the image and storing it as the point cloud. The bottleneck distances and wassertian distance computation takes a lot of time and system also goes out of memory if we use the original image barcodes. Therefore, to get rid of the issues, we sample 200 points from the boundary and then store it as point cloud. I have tried other numbers as well like 150, 250 etc. but 200 is an optimal number of points where you get a good boundary filtration.

## 1.3 Computation of Bar codes

Once we have the point clouds, next step is to compute the barcodes from the PointCloud Data. Here we also have to separate the bar codes in form of Dimension 0 and Dimension 1. To compute the barcode, I am calling the ripser command using the subprocess library. Next, we just write a parser to separate and store dimension 0 and dimension 1 features. The results can be found in OutputDataP1_Barcode_Sampled Folder.

## 1.4 Comparison of the barcodes

Till this point, we have the barcodes, next step is to compare these different barcodes. Two methods that we are gonna use are the bottleneck and wassertian distance. To compute these distances, we are using in built functions from sklearn_tda library. I was using the persim.bottleneck distance earlier but that is computationally more expensive than the sklearn_tda library. After getting the distances, we are gonna project it to a 2D plane by using Dimensionality reduction techniques namely MDS and TSNE. Please find the resulting plots from TSNE and MDS in the subsequent pages.

Note: Please follow the comments in the code to get the understanding of the works.

### 1.5 MDS and TSNE results for Bottleneck and Wassertian Distances

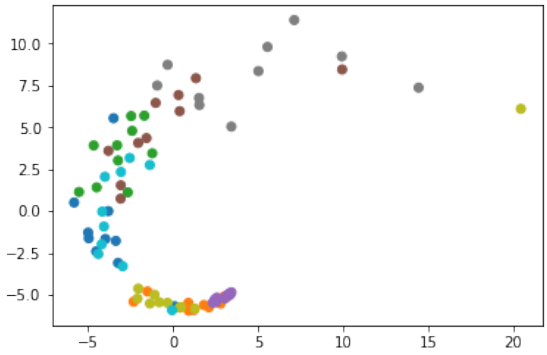


Figure 1: MDS applied to Dimension 0 Bottleneck Distances

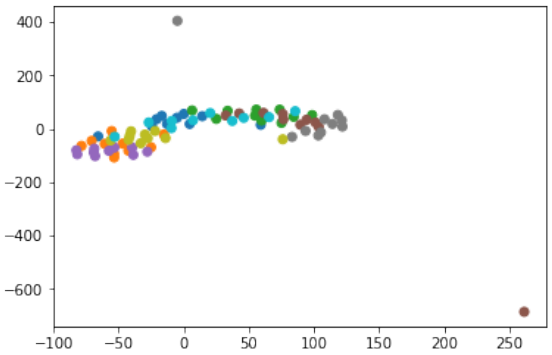


Figure 2: TSNE applied to Dimension 0 Bottleneck Distances

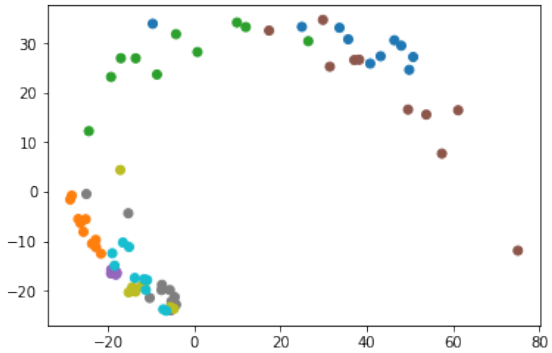


Figure 3: MDS applied to Dimension 1 Bottleneck Distances

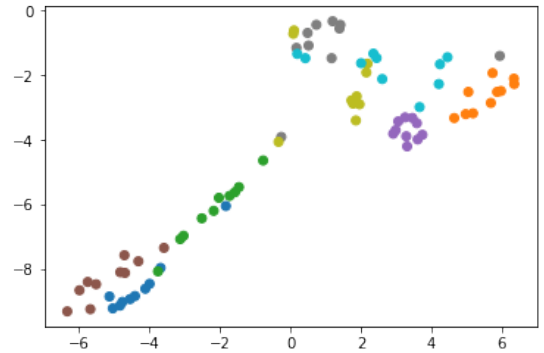


Figure 4: TSNE applied to Dimension 1 Bottleneck Distances

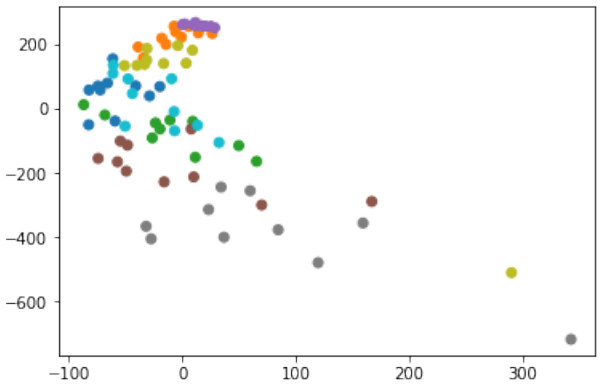


Figure 5: MDS applied to Dimension 0 Wassertian Distances

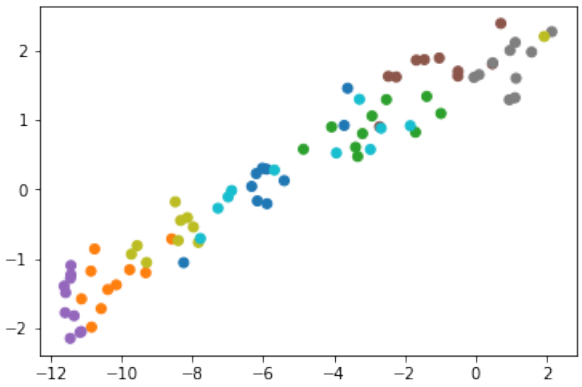


Figure 6: TSNE applied to Dimension 0 Wassertian Distances

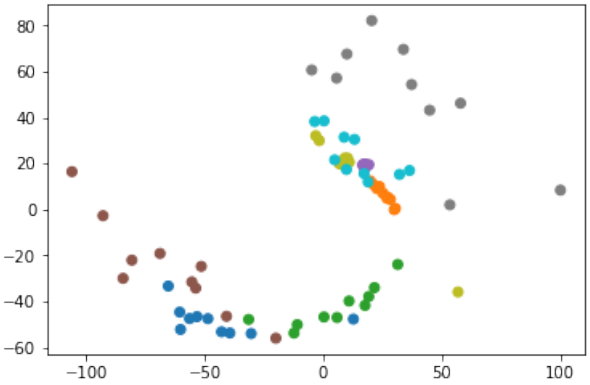


Figure 7: MDS applied to Dimension 1 Wassertian Distances

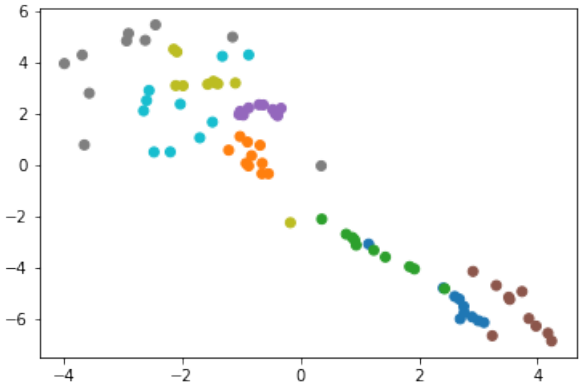


Figure 8: TSNE applied to Dimension 1 Wassertian Distances

### 1.6 Raw images MDS and TSNE results

Here, we are passing the raw images to MDS and TSNE functions. But for that, we need to process the images a bit. First thing is to resize all the 80 images to same size. To get the resizing coordinates, I first found out the maximum dimension in X and Y dimension of the image. In my dataset, it's 720 px so we used an online tool to resize all the images to 730 * 730 px (extra 10 px to make sure no points are at the boundary). This resizing requires the use of padding by black pixels or 0 values. If we use white pixels for padding, it would result in extra dimension 0 and 1 features.

Then, we read and store all the 80 images into a numpy array (named 'imgArray'). This is an array of arrays and it has dimensions of (80, 730, 730). Then, we flatten out all the images in this array by using the method or row by row concatenation discussed in the class. Finally, we call the MDS and TSNE functions over the flattened array and plot the results on a 2D plane.

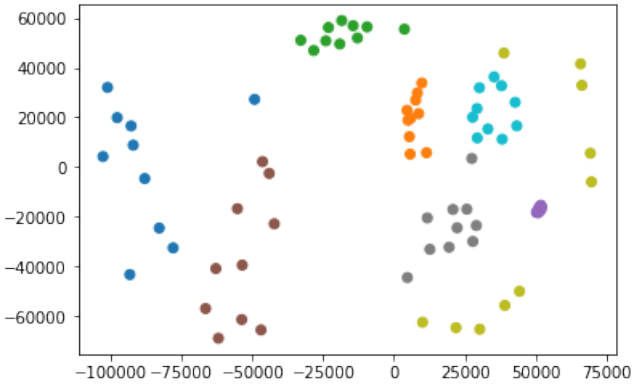


Figure 9: MDS applied to raw or slightly processed image

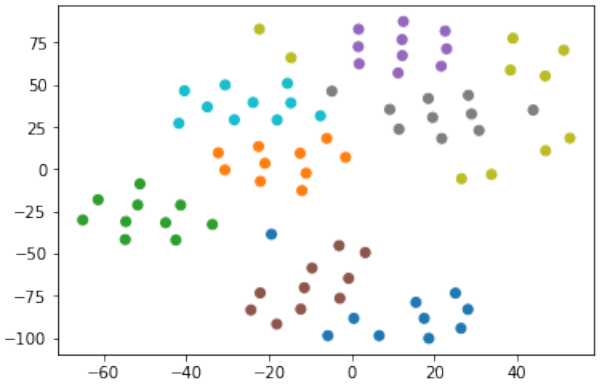


Figure 10: TSNE applied to raw or slightly processed image

**Comparison with previous results:** We get 8 different plots in Part 1.1 and 1.2 for dimension 0 and 1 respectively. From all the plots of MDS and TSNE, we can observe that the TSNE produces a bit better results than MDS in terms of clustering of the points. Also, we can observe that the output plots from the raw images are also slightly better than the plots we get from bottleneck and wassertian distance. One more thing to be observed is for dimension 0 and MDS, both wassertian distance and bottleneck distances produces plots which are not clustered so well compared to TSNE.

# 2 TDA + ML

## 2.1 Creating Labels for 4 and 8 classes respectively

Initialize labels for the input data points. In this case it will be 1 label each for the 80 images. For the 8 classes, we will have 1 label for each image classes. Eg. apple will have label 0, car will have label 1 and so on. For the 4 classes case, we are simply allocating a single label to the combination of two image classes. For example, the "apple" and "car " have same labels. The assignment of equal labels to two classes is finalized by looking at the persistence diagram of the images and see which classes are the closest ones and assign them the same label

## 2.2 Using SVM to classify the 80 images into 4 and 8 classes

Here we are using the flattened image array to get the train, test, training labels, testing labels after splitting the original data by 80% training and 20% testing ratio. For each of the classes, first 8 images will be considered a part of the training set and the rest 2 images will be added to the training set. Then, we are using the linear SVC kernel to fit and calculate the accuracy of the prediction. We get very high accuracy of prediction, nearly about 93.75% most of the times. It sometimes fluctuates to 80% or 100% depending on the random train_test_split we have.

## 2.3 Using Persistence Scale Space Kernels

To perform the classification using Persistence Scale Space Kernels, we use the scikitlearn_tda library. There's an inbuilt function called PersistenceScaleSpaceKernel(). We first split the training and testing data from the persistence diagram array i.e. dgmArr1 and dgmArr0. We then use SVC with 'precomputed' options to compute the fit and score of the Scale Space Kernel. We run this code first for 8 classes/labels and for both 0 and 1 dimensional features. We then perform similar runs for 4 classes/labels. Accracy of predictions are reported with each of the running cell.

We get good accuracies for both the runs with 8 and 4 labels respectively. For dimension 1 in both the cases, we get fluctuating accuracies centered around 62.5% with mostly on high side being 81%. Similarly for dimension 0, we get relatively lower accuracies centered around 43.75%.

## 2.4 Persistence Image Classification

Similar to the previous case, here also we perform the experiments with 8 and 4 classes/labels resp. First, we compute the persistence images from the persistence diagrams array i.e dgmArr0 or dgmArr1 using the PersImage function implemented in persim. Once, we have the persistence diagram array, we split the training and testing data from the persistence diagram array. After that, we use linear SVC kernel to perform classification.

We get good accuracies for both the runs with 8 labels respectively. In the case of 8 labels, for dimension 1 in both the cases, we get fluctuating accuracies centered around 56.25% with high being 93.75% and low being around 43%. Similarly for dimension 0, we get relatively lower accuracies centered around 37.5%. In the case of 4 labels, for dimension 1, we get varying accuracies on each run centred around 37.5% and for dimension 0, we get varying accuracies centered around 25%.

### 2.5 Summarized Results for Part 2

The table below summarizes all the accuracies. But please note, these are just an snapshot at a point of time, real accuracies fluctuates with every run since the testing and training data is changed on every run. Also, for the first image classification there is no role of dimension 0 and 1 but to maintain the integrity of the table, values are duplicated for dimension 0 and 1 in a commensurate manner.

| | Accuracy ( 8 Labels) - Dim 0 | Accuracy ( 4 Labels) - Dim 0 | Accuracy ( 8 Labels) - Dim 1 | Accuracy ( 4 Labels) - Dim 1 |
|---|---|---|---|---|
| Image Classification (Linear Kernel) | 93.75% | 100% | 93.75% | 100% |
| Persistence Space Scale Kernel | 43.75% | 43.75% | 68.75% | 81.25% |
| Persistence Image Classification | 37.5% | 25% | 56.25% | 37.5% |

## 3 Appendix

### 3.1 Folder Description

- **Full Data** -> Full original MPEG-7 Data
- **InputDataP1** -> contains the 80 input image files which is being used for the processing
- **InputDataP1_resize** -> contains the resized 80 input image files (each image is now 730*730 px)
- **OutputDataP1_PointCloud_NonSampled** -> Point cloud from the original image. No sampling is done here. I am not using it in this code but just included it as a reference.
- **OutputDataP1_PointCloud_Sampled** -> Due to memory and timing issues, we sample 200 points from the data as we discussed earlier. I am using this data to generate barcodes.
- **OutputP1_Barcode** -> Barcodes or Persistence Diagrams for the original image. Not using it in the code.
- **OutputDataP1_Barcode_Sampled** -> Barcodes or Persistence Diagrams for the sampled image.
- **OutputDataP1_Distance** -> Contains the bottleneck and wassertian distance matrices for dimension 0 and 1 resp. It also contains the flattened image array of size 80 * 532900.
- **Figures** -> contains the plots from this ipynb

### 3.2 Python Libraries used

Following libraries were used in their latest versions: numpy, matplotlib, scipy, subprocess, persim, ripser, tadasets, tqdm, sklearn - MDS, sklearn TSNE, PIL, Counter from collections, sklearn_tda, sklearn svm, sklearn model selection