Topological Data Analysis

Malika Satayeva, Shriya Bhatija, Vincent Lechner, Jinya Sakurai

November 2022

Abstract

This project investigates how well standard tools from Topological Data Analysis (TDA), specifically persistent homology and hierarchical clustering, are able to distinguish the different classes of the ISIC 2018 Challenge training dataset [1, 2]. Persistent homology is applied to investigate on the separability of stable rank signatures in terms of various image pre-processing and embedding (2D, 3D, High-dimensional) combinations that were chosen in a pre-selection procedure incorporating average stable rank separation measures. Consistent signatures with lower uncertainty are suspected to be present at least for the high dimensional embedding. Additionally, for suitable combinations, classifiers in the form of SVM are trained in order to investigate if these signatures may serve as an effective basis for classification, in various kernel variations.

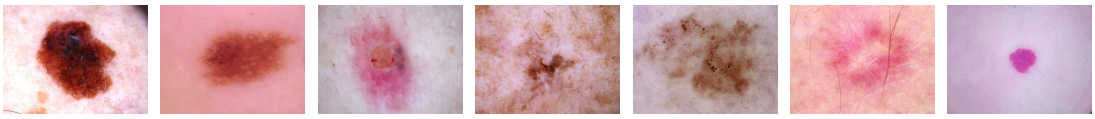
Contents

1 Introduction

Topological data analysis (TDA) is a recent and rapidly expanding field that provides a framework to study datasets by using techniques from algebraic topology. It is built upon the premise that data has shape and that this shape carries relevant information which may be utilized to discover the latent structure of the respective dataset. TDA formalises this idea by introducing homology-based invariant transformations that are applicable to various types of data. Roughly stated, homology is a mathematical theory that arises from observing that different shapes can be distinguished by examining their n-dimensional holes. In the general workflow of TDA, homology is used to extract high-level geometrical information of the data and encode it into a low-level feature space. Having the dataset represented in this new feature space makes it more feasible to analyse and extract information from. This can then be done by using standard machine learning methods.

TDA was introduced to analyse high-dimensional, sparse and noisy data and it has so far shown promising results in that regard. The two main methods that were presented in this course are hierarchical clustering and persistent homology, both of which will be explained further on once the mathematical framework will have been presented.

This project investigates the applicability of persistent homology on the International Skin Imaging Collaboration's (ISIC) 2018 Challenge training dataset[1] [1, 2]. This dataset consists of 10015 dermoscopic images grouped into seven skin disease types: Melanoma (MEL), Melanocytic nevus (NV), Basal cell carcinoma (BCC), Actinic keratosis / Bowen's disease (intraepithelial carcinoma) (AKIEC), Benign keratosis (solar lentigo/seborrheic keratosis/lichen planuslike keratosis) (BKL), Dermatofibroma (DF), and Vascular lesions (VASC). A sample image from each type is displayed in the figure below:



(a) MEL (b) NV (c) BCC (d) AKIEC (e) BKL (f) DF (g) VASC

Figure 1: Sample images of different types of skin lesions in the ISIC 2018 dataset

From this dataset 100 images per class were randomly sampled to form a new training dataset. We aim to investigate as to whether persistent homology in conjunction with an SVM classifier is able to distinguish between the different types of skin lesions. More specifically, we want to to find out whether the different skin lesions have distinct geometrical information that can be captured and distinguished into clusters using persistent homology. This will be done by applying various image pre-processing and embedding ($2D$, $3D$, high-dimensional) combinations to the data samples to further investigate whether prior image modification may enhance class separability.

We conduct our analysis using the Ripser Software[2] [4] and the methods introduced in the course Topological Data Analysis [5]. This report aims to present our workflow and the results obtained. In the next section we will introduce the necessary mathematical concepts needed for our analysis before continuing to describe our pipeline and results. We proceed to interpret and discuss these results and finally close this report with a brief conclusion.

[1] The International Skin Imaging Collaboration holds the largest archive of dermoscopic images that were collected over the recent years from various clinical centers throughout the world. With this collection ISIC sponsors annual challenges in an attempt to improve the diagnosis of skin lesions.

[2] `ripser.py` is a Python package that is able to perform persistent homology and compute stable ranks.

2 Mathematical Framework

This chapter introduces some of the mathematical concepts that were covered in the course Topological Data Anlaysis [5]. We aim to roughly review the relevant notions needed for our analysis. For the full mathematical derivations of these concepts we refer to [5].

Let us fix a set $\mathcal{U}$ that we call *universe*. In the following all sets are considered to be subsets of $\mathcal{U}$. Let X be a subset of $\mathcal{U}$.

Definition 2.1 (Distance, pseudometric and metric)**.** A *distance* on X is a function $d\colon X \times X \to [0, \infty]$ such that for all $x, y, z \in X$ the following conditions hold:

(i) Symmetry: $d(x, y) = d(y, x)$,

(ii) Reflexivity: $d(x, x) = 0$.

Furthermore, a distance is called a *pseudometric* if it satisfies:

(iii) Triangle inequality: $d(x, y) + d(y, z) \leq d(x, z)$.

In addition, a pseudometric is called a *metric* if:

(iv) $d(x, y) = 0$ if and only if $x = y$.

A set X together with a chosen distance/pseudometric/metric d is called a *distance/pseudometric/ metric space* and will be denoted by (X, d).

Definition 2.2 (Gromov-Hausdorff distance)**.** For pseudometric spaces (X, d_X) and (Y, d_Y) the *Gromov-Hausdorff distance* between them is defined as

$$\mathrm{GH}(X, Y) := \inf\{\epsilon \in [0, \infty) \mid (X, d_X) \text{ and } (Y, d_Y) \text{ are } \epsilon\text{-close}\}. \tag{1}$$

Definition 2.3 (Extension)**.** Let $\Delta[X]$ denote the set of all non-empty finite subsets of X. Let $\mathcal{D}(X)$ and $\mathcal{D}(\Delta[X])$ be the sets of all distances on X and $\Delta[X]$, respectively. A function $\phi : \mathcal{D}(X) \to \mathcal{D}(\Delta[X])$ is an *extension* if for all $d \in \mathcal{D}(X)$, $\phi(d)$ restricted to $X \subset \Delta[X]$ is equal to d.

Examples The following distances are extensions from X to $\Delta[X]$. These will become important in our analysis. For $d \in \mathcal{D}(X)$ and $A, B \in \Delta[X]$ we have:

$$\textit{Single linkage} \text{ extension}: \quad sd(A, B) := \min\{d(a, b) \mid a \in A \text{ and } b \in B\} \tag{2}$$

$$\textit{Complete linkage} \text{ extension}: \quad cd(A, B) := \begin{cases} \max\{d(a, b) \mid a \in A \text{ and } b \in B\} & \text{if } A \neq B \\ 0 & \text{if } A = B \end{cases} \tag{3}$$

$$\textit{Average linkage} \text{ extension}: \quad ad(A, B) := \begin{cases} \frac{\sum_{a \in A, b \in B} d(a,b)}{|A||B|} & \text{if } A \neq B \\ 0 & \text{if } A = B \end{cases} \tag{4}$$

$$\textit{Ward linkage} \text{ extension}: \quad wd(A, B) := \frac{|A||B|}{|A| + |B| - 1} l(m_A, m_B), \tag{5}$$

where d is the restriction along $f : X \to \mathbb{R}^n$ of a metric l and moreover

$$m_A = \frac{1}{|A|} \sum_{a \in A} f(a) \text{ and } m_B = \frac{1}{|B|} \sum_{b \in B} f(b).$$

Definition 2.4 (Dendrogram)**.** A *dendrogram* is a sequence of sets $(D_t)_{t \in [0,\infty]}$ together with a sequence of surjective functions $(D_{s<t} : D_s \to D_t)_{s<t \in [0,\infty]}$ such that the following conditions hold:

(i) $D_{r<t} = D_{s<t} \circ D_{r<s}$ for any $r < s < t$ in $[0, \infty)$;

(ii) There is a sequence of real intervals $[0, a_1), [a_1, a_2), ..., [a_{n-1}, a_n), [a_n, \infty)$ such that $D_{s<t}$ is an isomorphism for any $s < t$ where s and t belong to the same interval.

Definition 2.5 (Parametrized tame vector space)**.** A *tame vector space V parametrized* by $[0, \infty)$ is a sequence of finite dimensional vector spaces $(V_t)_{t\in[0,\infty)}$ together with a sequence of linear maps $(V_{s<t} : V_s \to V_t)_{s<t\in[0,\infty)}$ such that the following conditions hold:

(i) $V_{r<t} = V_{s<t} \circ V_{r<s}$ for any $r < s < t$ in $[0, \infty)$;

(ii) There is a sequence of real intervals $[0, a_1), [a_1, a_2), ..., [a_{n-1}, a_n), [a_n, \infty)$ such that $V_{s<t}$ is an isomorphism for any $s < t$ where s and t belong to the same interval.

Theorem 2.6. *Any tame vector space parametrized by* $[0, \infty)$ *is isomorphic to a unqiue direct sum of the form* $\bigoplus_{i=1}^{n}[a_i, b_i)$.

Definition 2.7 (Simplicial Complex)**.** Let K be a finite and non-empty subset of $\mathcal{U}$. K is called a *simplicial complex* if $\sigma \in K$ implies $\tau \in K$ for all $\tau \subset \sigma$.

Definition 2.8 (Vietoris-Rips complex)**.** Let (X, d) be a finite distance space. For $t \in [0, \infty)$ the $Vietoris-Rips\ complex\ at\ scale\ t$ of (X, d), $\mathrm{VR}_t(X, d)$, is the collection of all subsets $\{x_0, \ldots, x_n\} \subset X$ for which it holds $d(x_i, x_j) \leq t$ for any i and j. We denote the time series of $\mathrm{VR}_t(X, d)$ with $\mathrm{VR}_\bullet(X, d)$.

Definition 2.9 (Homology)**.** Let F be a field and let K be a simplicial complex for which we choose an ordering of its vertices: $K_0 = \{v_0 < v_1 < \cdots v_l < \cdots\}$. Denote by $d_i : K_n \to K_{n-1}$ the function that removes the i-th vertex of a given n-dimensional simplex for $n \geq 1$ and $0 \leq i \leq n$. For $n \geq$ let $\delta_n : FK_n \to FK_{n-1}$ be the boundary functions. The n-th *homology* is defined as

$$H_n(K, F) := \ker(\delta_n)/\mathrm{im}(\delta_{n+1}). \tag{6}$$

Remark 2.10. The homology function H_n can be utilized to transform simplicial complexes to tame parameterized vector spaces. For a given input it measures how many more cycles than boundary elements there are. For instance, H_0 indicates the number of connected components while H_1 indicates the number of one-dimensional holes.

Definition 2.11 (Space of piece-wise constant functions (pcf))**.** A function $f : [0, \infty) \to (-\infty, \infty]$ is called $piece-wise\ constant$ if there exists a sequence of intervals $[0, a_1), [a_1, a_2), \ldots, [a_{n-1}, a_n), [a_n, \infty)$ such that f is constant on each of those intervals.

One can equip the set of piece-wise constant functions with metrics. The following two metrics are standard choices:

(i) *Interleaving metric*:

$$d_{\bowtie}(f, g) := \begin{cases} \infty & \text{if } \{v \mid f(x) \geq g(x + v) \text{ and } g(x) \geq f(x + v) \text{ for any } x\} = \emptyset, \\ \inf\{v \mid f(x) \geq g(x + v) \text{ and } g(x) \geq f(x + v) \text{ for any } x\} & \text{else;} \end{cases} \tag{7}$$

(ii) l_p *metric* for $1 \leq p < \infty$:

$$d_p(f, g) := \left(\int_0^\infty |f - g|^p\right)^{\frac{1}{p}}. \tag{8}$$

2.1 Persistent Homology

Persistent homology combines the above mentioned mathematical concepts into is a stream of techniques to transform data samples into piece-wise constant functions. This is done in the following

way:

Given a labelled dataset $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$, each data sample is represented by a finite point cloud equipped with a distance metric. This results in n distance spaces $\mathcal{D}(x_i)$, each one corresponding to one sample.

Once we have the distance spaces $\mathcal{D}(x_i)$ for each sample x_i, one can obtain a tame parametrised vector space via $H_n(\mathrm{VR}_\bullet(-), F)$ by first transforming the distance space $\mathcal{D}(x_i)$ into a simplicial complex $\mathrm{VR}_\bullet(\mathcal{D}(x_i))$ and then applying homology to obtain $H_n(\mathrm{VR}_\bullet(\mathcal{D}(x_i)), F)$.

Then $H_n(\mathrm{VR}_\bullet(\mathcal{D}(x_i)), F)$ is uniquely given by its bar decomposition $V \cong \bigoplus_{i=1}^n [a_i, b_i)$. Further, one can convert tame parameterized vector spaces into piecewise-constant functions using the $\widehat{\mathrm{rank}}$ operator:

$$\widehat{\mathrm{rank}} : V \mapsto \mathrm{pcf}$$

given by $\widehat{\mathrm{rank}}(V_t) = |\{i \mid b_i - a_i \geq t\}|$. Hence, we now have a process of translating distance spaces into piece-wise constant functions.

Having each data sample represented as a piece-wise constant function, standard machine learning methods may be utilized for classification and clustering of the data, see subsection 2.3.

Stability An important (and desired) feature of persistent homology is that this process is stable if the initial space is pseudometric. In particular, if two data samples are similar then their piece-wise constant functions are also going be similar. The following theorem formalizes this property by combining two important results from the course:

Theorem 2.12. *Let (X, d_X) and (Y, d_Y) be two pseudometric spaces. Then it holds*

$$2GH(X, Y) \geq d_{\bowtie}(\widehat{rank} H_n(\, VR_\bullet(X)), \widehat{rank} H_n(\, VR_\bullet(Y))). \tag{9}$$

2.2 Hierarchical Clustering

Similar to persistent homology, the stable rank process enables us to represent finite distance spaces as pcf's. Given our finite distance spaces $\mathcal{D}(x_i)$ and an extension $\phi : \mathcal{D}(x_i) \to \mathcal{D}(\Delta[x_i])$, we can construct a dendrogram $(D_t)_{t\in[0,\infty]}$ by algorithm 1. Here this algorithm that maps d to $(D_t)_{t\in[0,\infty]}$ is called **hierarchical clustering** with respect to ϕ.

Algorithm 1 Calculate a dendrogram, taken and adapted from [5]

Require: X : a finite and non-empty set, d : a distance on X, ϕ : an extension on X
Ensure: $\{a_0, a_1, \ldots, a_i, P_0, P_1, \ldots P_i\}$ forms a dendrogram

$i \leftarrow 0$
$a_0 \leftarrow 0$
$P_0 \leftarrow X/d_0$
$\pi_0 \leftarrow$ the quotient function $X \to X/d_0$
$b_0 \leftarrow$ a distance on P_0 extended by ϕ: $b_0(p_1, p_2) = \phi\, d(p_1, p_2) \quad (p_1, p_2 \in P_0)$
while b_i is **NOT** discrete **do**
    $a_{i+1} \leftarrow \mathrm{sep}(b_i)$
    $d_{i+1} \leftarrow$ the composition shown below:

$$X \times X \xrightarrow{\pi_i \times \pi_i} P_i \times P_i \xrightarrow{(b_i)_{\mathrm{sep}}} [0, \infty],$$

    where $(b_i)_{\mathrm{sep}}$ is the distance on P_i such that:

$$(b_i)_{\mathrm{sep}}(p_1, p_2) = \begin{cases} 0 & \text{if } b_i(p_1, p_2) \leq \mathrm{sep}(b_i), \\ \infty & \text{otherwise} \end{cases}$$

    $P_{i+1} \leftarrow X/d_{i+1}$
    $\pi_{i+1} \leftarrow$ the quotient function $X \to X/d_{i+1}$
    $b_{i+1} \leftarrow$ a distance on P_{i+1} extended by ϕ: $b_{i+1}(p_1, p_2) = \phi\, d(p_1, p_2) \quad (p_1, p_2 \in P_{i+1})$
    $i \leftarrow i + 1$
end while
return $\{a_0, a_1, \ldots, a_i, P_0, P_1, \ldots P_i\}$

The obtained dendrogram $(D_t)_{t\in[0,\infty)}$ is a time series of sets, where at each point in time t the set D_t represents a partition of $\mathcal{D}(x_i)$ induced by a discrete pseudometric at scale t corresponding to a chosen extension. Then one can naturally transform $(D_t)_{t\in[0,\infty)}$ into a tame parameterized vector space $(V)_{t\in[0,\infty)}$ which can further be translated into a pcf, via a process called *stable rank*:

$$\widehat{\mathrm{rank}} : V \mapsto \mathrm{pcf}$$

given by $\widehat{\mathrm{rank}}(V_t) = \dim(V_t)$.

In the following we will connect hierarchical clustering to persistent homology: Note that hierarchical clustering w.r.t. the single linkage extension produces the tame parameterized vector space $(F\pi_0((X, Ld_t))_{t\in[0,\infty)}$. Further, we have the following results from the course [5]:

Lemma 2.13. *Let F be a field. Then it holds $F\pi_0(X, Ld_t) \cong F\pi_0(VR_t(X))$.*

Lemma 2.14. *For any simplicial complex K and field F, it holds $H_0(K, F) \cong F\pi_0(K)$.*

Note that the Lemmata above imply $F\pi_0((X, Ld_t)) \cong H_0(VR_t(X), F)$ for which an equivalent (but for us more useful) statement is given in the below theorem:

Theorem 2.15. *Let (X, d) be a finite distance space. The stable rank process w.r.t the single linkage extension is equivalent to computing the H_0 homology of $VR_\bullet((X, d))$.*

For brevity we will from now refer to hierarchial clustering as persistent homology.

2.3 Support Vector Machine[3]

As indicated, the contents in this section are taken from [3]. Having each data sample converted into a piece-wise constant function with which we can measure the similarity of samples. Leveraging this similarity by support vector machines(SVMs), we try to predict the label of given images. Here, we provide the summary about SVMs.

Consider the binary classification problem given the data $\{(x_i, y_i)\}_{i=1}^n$, where inputs $x_i \in \mathbb{R}^m$ and labels $y_i \in \{-1, 1\}$ for all $i = 1, 2, \ldots, n$. SVMs estimate the label y given the input x, using the linear classifier

$$f(x) = \mathrm{sgn}(w^\top x + b), \tag{10}$$

where $\mathrm{sgn}(\cdot) : \mathbb{R} \to \{-1, 0, 1\}$ returns the sign of the input.

To begin with, we assume that the given data $\{(x_i, y_i)\}_{i=1}^n$ is linearly separable, that is, there exist some $w \in \mathbb{R}^m$ and $b \in \mathbb{R}$, for all $i = 1, 2, \ldots, n$, it holds that

$$y_i(w^\top x_i + b) \geq 0. \tag{11}$$

This means one can draw a hyperplane that divides the data points by class. Here we have innumerable choices of the parameters w and b, so that we need to select the optimal w^* and b^* that are robust to the noises on features. We evaluate the robustness of the classifier by margin, the shortest distance between the hyperplane and feature points $\{x_i\}_{i=1}^n$, and consider the most robust model is the one that maximizes this margin. In light of the fact that the distance r between the point x and the hyperplane $\{x' | w^\top x' + b = 0\}$ is given as

$$r = \frac{w^\top x + b}{||w||},$$

we can formulate the classification problem as the following optimization problem.

$$\max_{w,b} F(w, b) := \frac{1}{||w||} \min_{i=1,\ldots n} [y_i(w^\top x_i + b)]. \tag{12}$$

It is worth noting that for all $\lambda > 0$, it holds that $F(\lambda w, \lambda b) = F(w, b)$, hence we can scale λ as we like. Here, we set λ^* so that

$$\min_{i=1,\ldots n} [y_i((\lambda^* w)^\top x_i + \lambda^* b)] = 1, \tag{13}$$

and the optimization problem 12 can be reduced to

$$\min_{w,b} \frac{1}{2}||w||^2 \quad \text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 \quad \text{for all } i = 1, \ldots, n. \tag{14}$$

Note that we rewrote $w \leftarrow \lambda^* w, b \leftarrow \lambda^* b$, and transformed the maximization of $1/||w||$ to the minimization of $||w||^2/2$ for the simpler calculation.

In the case where the data is not necessarily linearly separable, we can introduce the slack variable $\xi_i \geq 0 \quad (i = 1, \ldots, n)$, and relax the constraint $y_i(w^\top x_i + b) \geq 1 \quad (i = 1, \ldots, n)$ to

$$y_i(w^\top x_i + b) \geq 1 - \xi_i \quad (i = 1, \ldots, n). \tag{15}$$

Building upon the optimization problem 14, we obtain the following objective function:

$$\min_{w,b,\{\xi_i\}} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad \begin{cases} y_i(w^\top x_i + b) \geq 1 - \xi_i & \text{for all } i = 1, \ldots, n, \\ \xi_i \geq 0 & \text{for all } i = 1, \ldots, n, \end{cases} \tag{16}$$

where $C \geq 0$ is a hyper-parameter that balances the minimization of the margin and to what extent you allow the violation of the linear classification.

It is straightforward to extend these SVMs to the non-linear SVMs. Consider the binary classification problem given the data $\{(x_i, y_i)\}_{i=1}^n$, where inputs $x_i \in \mathcal{X}$ and labels $y_i \in \{-1, 1\}$ for all $i = 1, 2, \ldots, n$, and a non-linear feature mapping $\Phi : \mathcal{X} \to \mathcal{H}$ is applied to the input $x \in \mathcal{X}$. The objective function here is given as

$$\min_{w,b,\{\xi_i\}} \frac{1}{2}||w||_{\mathcal{H}}^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t.} \quad \begin{cases} y_i(\langle w, x_i \rangle_{\mathcal{H}} + b) \geq 1 - \xi_i & \text{for all } i = 1, \ldots, n, \\ \xi_i \geq 0 & \text{for all } i = 1, \ldots, n, \end{cases}$$

where $\langle \cdot_1, \cdot_2 \rangle_{\mathcal{H}}$ denotes the inner product in $\mathcal{H}$, and $|| \cdot ||_{\mathcal{H}} = \sqrt{\langle \cdot, \cdot \rangle_{\mathcal{H}}}$.

3 Pre-processing

In this section we will describe how we processed the respective dataset $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$ to make it suitable for our analysis. As explained in section 2, the starting point for the persistent homology pipeline are finite distance spaces. Therefore, our aim is to covert each image sample x_i into a finite distance space, also called *point clouds*. Particularly, our dataset can be represented in the following form

$$\mathcal{I} = \{Z_i \mid i \text{ s.t. } (x_i, y_i) \in \mathcal{T}\}, \tag{17}$$

where Z_i for now is a finite set representing the sample x_i. Note that x_i is a specific image, while y_i represent the label associated with the image class.

From the original dataset we subsampled 100 images for each of the seven classes unless stated otherwise in the experiments, since one of the classes only consists of 115 images to begin with. For the preselection step that will be outlined in a later section, 25 images were used. Now, each class will be represented by an equal number of images and there will not be any bias related to class size.

The original images are of RGB type and have size 600×450 pixels.

In order to transform each of the image samples into a point cloud, we designed a two-step processing integration stream: Firstly, we are going to apply different pre-processing techniques to the original image data, e.g. for noise filtration, size reduction or geometrical feature extraction. Secondly, we are going to use those pre-processed images for point cloud embeddings.

3.1 Image modifications

There are various techniques available for pre-processing image data. For our analysis we are interested in extracting geometrical information from the images, specifically H_0 and H_1 homologies. The following four methods seemed most suitable for our purpose:

Gray-scale: We adopted `PIL.Image.convert("L")` method to grayscale RGB images, which uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000.
```

Topological Image Processing: A recently published work [6] investigates a new image processing technique that we will call $TDA - processing$. Essentially, it constructs a filtration based on sublevel-filtration on a given individual image. [6] Subsequently from the H_0 persistent homology a first segmentation of the region of interest is generated [6]. Finally, topological image modification is applied which includes modified image borders, smoothing and background interpolation as further elaborated in [6] . In summary , this method applies persistent homology to gray-scale images to destruct irrelevant objects to allow for segmentation of the relevant skin lesions. It further enables us to filter out noise such as hair or other skin anomalies in irrelevant regions of the images. [6] The authors additionally provide a code for their processing method that we applied to our dataset in [6], see Figure 2 for an illustration:

Binary Mask: The topologically processed images allow for a better segmentation of the relevant skin lesion, see [6]. As a byproduct of the TDA procedure from [6], a binary mask is created, where each pixel takes on values of either one or zero, where pixels of value 1 correspond to pixels

inside the segmented region, while zero-valued ones are associated to the background. This binary mask may be applied to the gray-scale images as well as to the topologically processed images. The mask may also be applied to the original grey-scale images. The orignal grey scale image as well as the associated TDA-processed image with applied binary mask are shown in Figure 2. This binary mask also allows for cutting down the image to a certain region of interest, a rectangle around the segmentation to reduce the data size.

Edge detection: Edge detection is a useful tool to retrieve topological information from images. Here, we adopted *Canny's edge detection* [7], a seminal method used broadly in the computer vision field. To circumvent the detection of unnecessary background information as edges (e.g. hair) we only apply it to masked images. It transforms masked images to binary images where pixel value 1 shows the pixel is on edge and 0 indicates otherwise. Here, this algorithm has two hyper-parameters: a lower threshold and an upper threshold, which we empirically found the setting as the 10% percentile, and 90% percentile of all pixel values, respectively, leads to better performance. The last step of this pre-processing procedure is to recover intensity information. To this end, the binary mask obtained is applied on the original grey scale values or alternatively to the TDA-processed values. Thus all locations corresponding to edges retain their intensity information.

The following figure visualizes three combinations of pre-processing techniques: The basic grey scale image (c), the TDA-processed image with the binary mask applied (b) and finally the edge detection applied to the grey scale image that was previously masked (d).

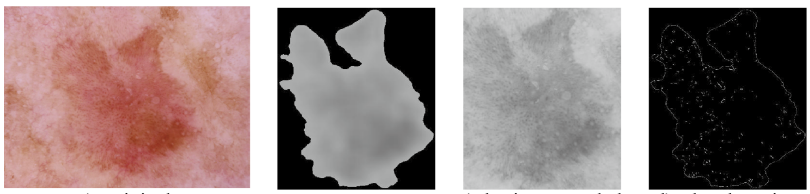


Figure 2: Different pre-processing methods applied to one exemplary image

3.2 Image Embeddings

There are various possibilities for point cloud formations, each extracting different geometrical information from the images. Therefore, each embedding may lead to disparate results when plotting the stable ranks. In the following paragraphs we will describe the embeddings we found intuitive and reasonable enough to consider for the analysis. For computational efficiency reasons these embeddings are applied only after the pre-processed images have been downsized to 56 x 56 pixels if not indicated otherwise.

2D Embedding: From the gray-scale images obtained from the previous step one can form two-dimensional point clouds in the following way:

For each pixel with coordinates (i, j), we calculate the corresponding probability $p_{i,j}$ of being

selected as

$$p_{i,j} = \frac{I_{i,j}}{\sum_{k=1}^{56} I_{i,k}}$$

where $I_{i,j}$ is the gray-scale intensity of a pixel at the position (i, j); $i = \overline{1, 56}$, $j = \overline{1, 56}$.

The reason for column-wise (alternatively we could select row-wise) normalisation is that we aim to increase the expected number of selected points compared to the case when scaling is done by sum of all pixel intensities.

Such that after, by comparing $p_{i,j}$ with a sample $u_{i,j}$ generated from uniform distribution over $[0, 1)$, we either keep it or not:

$$\begin{cases} \text{keep} & p_{i,j} > u_{i,j}, \\ \text{remove} & p_{i,j} \leq u_{i,j}. \end{cases}$$

3D Embedding: A three-dimensional embedding can be formed from the gray-scale images obtained in the image pre-processing step: Each image will be represented by its collection of pixel values (a, b, c) where (a, b) indicates the spatial dimensions and c represents the gray-scale value for the particular pixel. Therefore, each sample x_i is represented by

$$Z_i = \{(a_j, b_j, c_j) \mid 1 \leq j \leq (56 \times 56)\}. \tag{18}$$

The above embeddings translates each image x_i into a set Z_i in $\mathbb{R}^n$, $n = 2, 3$. Since we want to apply stable persistent homology, the point clouds need to be converted into pseudometric spaces. It was decided to equip the point clouds with the standard Euclidean norm on $\mathbb{R}^n$.

Image space embedding: Another natural embedding would be to consider the image to be represented as a point in a higher dimensional space. For this, each pixel corresponds to a coordinate. A given point cloud in this space is made up of individual images. Each point cloud is normalized by the maximum pixel intensity over all images in the cloud. It is clear however, that classification based on a stable rank kernel of the point clouds of the high dimensional space is not possible in a standard way: This embedding would at most consider the signature of a set of images, thus single image classification is less suited for this embedding. Nevertheless, it was considered as an option, as it might yield results on the question of, do sets of given skin lesion types differentiate themselves from each-other with a topological signature? Lastly regarding this embedding, there is an associated question: How to generate the average stable rank, in essence, how many 'samples' do we take of this high dimensional space, how many images should make up a point cloud, as we do not have unlimited data.

Figure 3 briefly depicts the flow chart of our pre-processing pipeline. Note that for the preprocessing steps, not all possible combinations are shown, but just the basic greyscale, the mask applied to the TDA processed image and lastly edge detection applied to the masked basic greyscale.

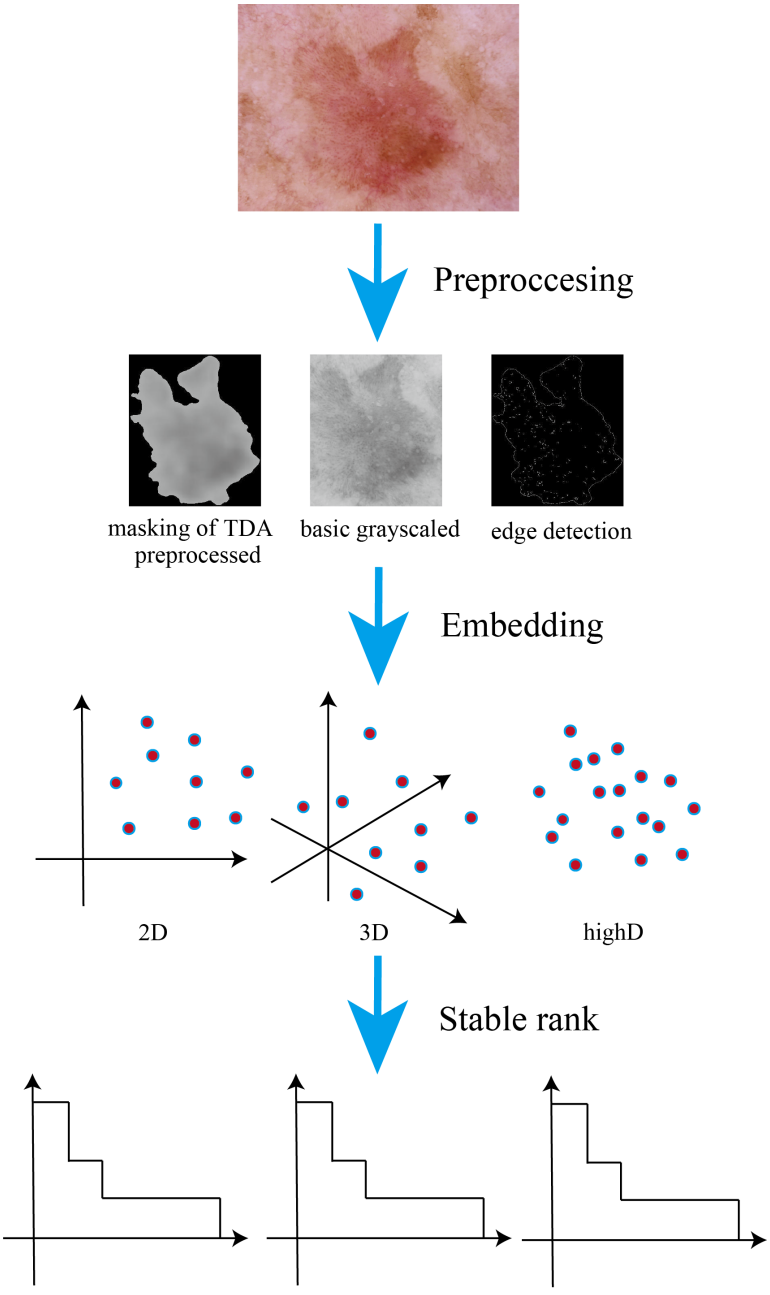


Figure 3: The pipeline of our analysis

4 Methods

We now present the methods that were introduced in order to conduct the analysis. As outlined in section 3, a number of pre-processing steps in combination with various embeddings were considered for potential candidates. However, analysing each combination in detail is infeasible within the scope of this project, hence the decision was made to introduce a pre-selection process.

After the pre-selection process, the most promising pre-processing-embedding combination candidates are investigated in more detail 5 There, an analysis will be presented, that is split into two components: Stablerank analysis and potentially classification with SVM, if applicable. The individual methods of the pre-selection, stablerank analysis and SVM analysis will be further elaborated in this section.

4.1 Pre-selection

We first consider pre-selection. As outlined before, we have in total 18 combinations of pre-processing and embeddings to generate point clouds. The pre-selection procedure has the following structure:

1. Generate consistent data sets with the respective pre-processing steps
2. For each embedding, consider all of the aforementioned data sets
3. Set all evaluation parameters to be consistent between the combination candidates
4. Generate the average H_0 stableranks of each class and for all candidates
5. Obtain a discrepancy score for each candidate

The third, fourth and last point demand more attention. Firstly consider the third: It is clear, the decision was made to accept, that this will not necessarily lead to an optimum in terms of stable rank separation or the ability to classify the respective skin cancer types optimally in the strict sense. This has to do with the fact that settings (e.g. contrast, image resolution, weighting between intensity and spatial information) may be tuned to benefit/hinder the success of a given combination. Thus, for the quantitative analysis of the pre-selection, they are all set to be consistent. The parameters in question are:

- Image resolution: 56×56
- Intensity to spatial information weighting where applicable: 1 to 1
- Contrast enhancement: Not applied

Consider now the fourth and fifth point: For each combination of pre-processing and embedding, we consider the average H_0 stable rank generated for each class, for the linkages *single, average,complete* and *ward*. We consider only the H_0 stable rank for this procedure, as the H_1 stable rank was deemed more unsatisfactory without parameter tuning and required significantly more computation time for certain embeddings. What we would like to obtain, would be a clear separation of the classes, as this may be an indication that there could be a consistent unique signature in form of the stable rank for a given class of skin lesion (as the sample size increases). Although it may be possible to gauge the distinction between average stableranks optically in extreme cases, it may be very difficult and inaccurate= to conduct this for all combinations, not even including the possible combinations with different linkages. Thus, a somewhat heuristic attempt was made to score the separation between average stable ranks. For a given combination of pre-processing, embedding and linkage the following matrix is constructed:

$$D_{ij} = d_{l_1}(f_i, f_j)/k_{ij} \tag{19}$$

In the above, d_{l_1} represents the l_1 distance between piecewise constant functions as introduced in 2. The l_1 distance here is chosen as it gives the normalized score a straightforward geometrical interpretation. Furthermore, $f_i \in F$, where F is defined as the set of average H_0 stableranks, one for each category of skin lesion. Thus in our case $|F| = 7$. Lastly, $k_{i,j}$ is a normalization constant, that is obtained as follows:

$$k_{i,j} = \max(a_{f_i,n}, a_{f_{j,n}}) \times \left|(\max(a_{f_i,n}, a_{f_j,n}) - \min(a_{f_i,n}, a_{f_j,n}))\right| \tag{20}$$

Where $a_{f_s,n}$ denotes the starting value of the ultimate interval of the piecewise constant function in question f_s. Note that here we assume that $lim(f_s) = 0$, within the context of the limit defined in the lecture notes [5] and $a_{f_s,0} = 0$. The last point simply means we do not have to consider the minimum over the left-most starting value. Thus, the interpretation of an entry D_{ij} would be: The absolute discrepancy in terms of the l_1 distance between the stable ranks of two classes i and j, normalized for the maximum possible value attainable between the two. Geometrically speaking, this would correspond to the area taken up by the difference between the two functions, as a fraction of the maximum area attainable. This step is required to make a more objective score, as different linkages and pre-processing steps were found to be on different scales, but the interest lies within the relative separation of the stable ranks within a given combination. Finally, the average over all values in D is taken. It is clear that by symmetry of the l_1 distance, $D_{ij} = D_{ji}$ for all i, j. Additionally $D_{ii} = 0$ as the distance to itself for a given piece-wise constant function is 0. This could be of course taken into account additionally.

4.2 Software

To run our experiments, we made use of several Python libraries such as `TIMprocess`, taken from and described in [8] and `stablerank` as provided in the course SF2956 given by W.Chalóski, associated with the notes [5]. This package in turn partly depends on the python implementation of the persistent homology package called `ripser.py` [4], which is based on *Ripser* originally described in [9]. Lastly, fundamental packages like `numpy`, `scipy`, `matplotlib`, `sklearn` for the SVM classification are incorporated.

`TIMprocess` was applied to get pre-processed gray-scale versions together with segmentation mask of the original images. Further all supplementary functions for distance, distance extension, H_0 and H_1 stable rank calculations and corresponding visualization routines were imported from `stablerank` package.

4.3 Average stable rank generation

In this section we quickly outline the generation of the average stable rank information and the associated uncertainties shown in the plots. The average stable rank generation is carried out as described in 2. Notably, stable rank results for H_0 are calculated not by explicitly constructing a VR complex in `Ripser.py`, but by first constructing a pseudometric distance space, in our case based on the euclidean distance by utilizing `stablerank.Distance(spatial.distance.pdist)` from `stablerank` and utilizing `scipy.spatial`. Subsequently the H_0 stable rank information is obtained by calling the `get_h0sr` method of `stablerank.Distance`.
For the H_1 stable rank informration, we first again define a distance space as before. Then we construct barcodes using the `stablerank.distance` method `get_bc`. Lastly, the barcode information is converted to H_1 stablerank information by utilizing `stablerank.bc_to_sr`. It should be noted that in the calculation of the barcodes we now implicitly generate the information based on VR complexes as the pipeline includes generating the H_1 persistence from `Ripser.py`.

5 Results and Discussion

5.1 Pre-processing method selection

Given sub-optimal procedure described in subsection 4.1, we want to exclude some of the less promising pre-processing and embedding combinations based on average dissimilarity score across 7 classes for each of the 4 linkage functions, so that 3 best performing combinations on most of the 4 linkage functions would be considered for the main pipeline. To decrease the running time of the analysis program, we down-sample an already reduced ISIC dataset, so there are 25 images per class.

Pre-processing	Embeddings 2D				3D				High-D			
	S	*C*	*A*	*W*	*S*	*C*	*A*	*W*	*S*	*C*	*A*	*W*
Direct with mask	0.009	0.006	0.007	0.002	0.004	0.0006	0.0006	0.00008	0.07	0.056	0.063	0.055
Direct no mask	0.021	0.004	0.005	0.001	0.005	0.0011	0.0012	0.0001	0.096	0.0654	0.074	0.063
Edge direct mask	0.016	0.008	0.009	0.002	0.004	0.0011	0.0013	0.0001	0.102	0.095	0.010	0.092
Edge TDA mask	0.007	0.004	0.004	0.001	0.002	0.0006	0.0007	0.0001	0.086	0.086	0.086	0.079
TDA with mask	0.007	0.005	0.006	0.002	0.005	0.0009	0.0008	0.0001	0.078	0.062	0.069	0.061
TDA no mask	0.021	0.005	0.006	0.002	0.01	0.0013	0.0013	0.0001	0.081	0.066	0.074	0.062

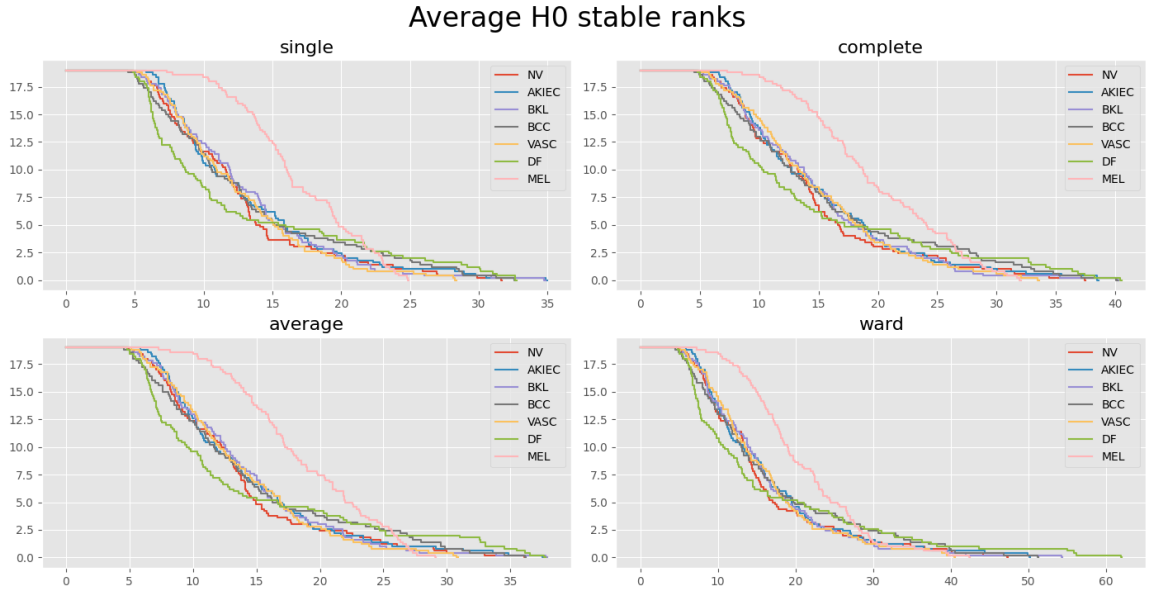
Table 1: Average dissimilarity score across 7 classes

By looking at Table 1, we conclude that edge detection applied on the `PIL` grey-scale convertion of the original image with a corresponding segmentation mask (from `TIMprocess`) on it, on average is better at separating classes for 2D and higher dimensional point clouds. However, the dissimilarity scores themselves are usually below 0.1 indicating that we may not expect even visually distinguishable separation. Note that at this stage it's still a hypothesis, which is to be tested in the next subsection 5.2. Meanwhile, gray-scale image produced by processing original image with topological tools from `TIMprocess` and no background filtering, clearly, gives the best dissimilarity score on $3D$ point clouds for all distance extensions. The downside of the latter pre-processing method is that the dissimilarity score is close to zero which is not what we would expect from a method that is good at extracting distinctions between classes.

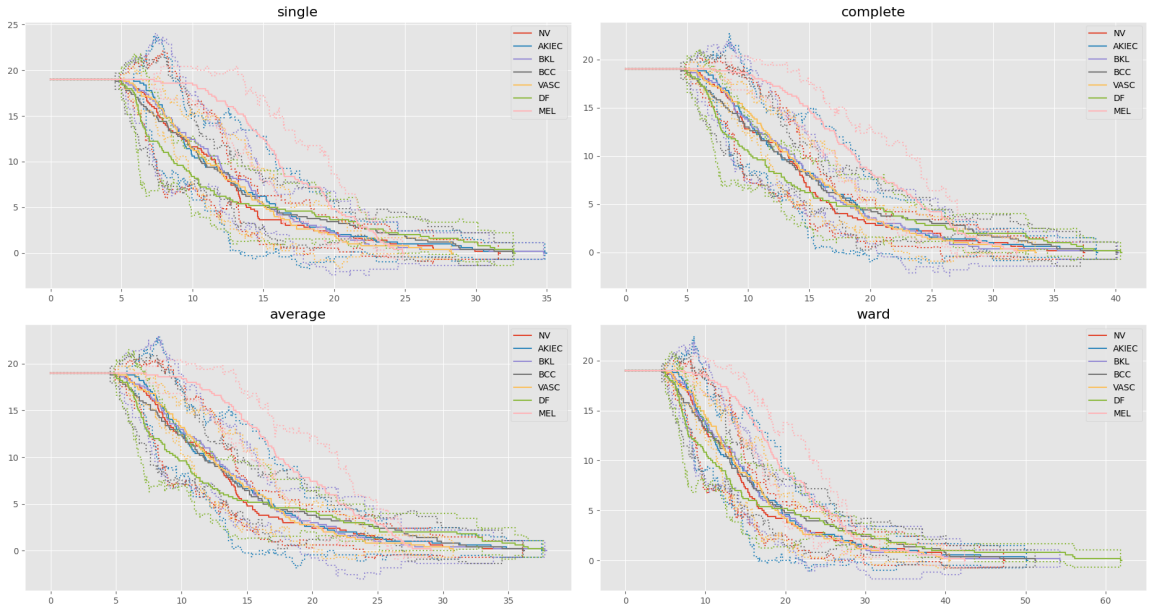
5.2 Stable rank results

High-dimensional Embedding - Edge detection

Here the results are presented and discussed with the high dimensional embedding of the preprocessed data. First consider the H_0 average stable rank results, with the pre-processing step of edge detection applied on the masked standard grey-scale image, without any increase in contrast, as was taken for the pre-selection procedure. Now however, a larger data set is used of 100 images. Furthermore, we consider each image here in higher resolution, 128 $\times$ 128 pixels.



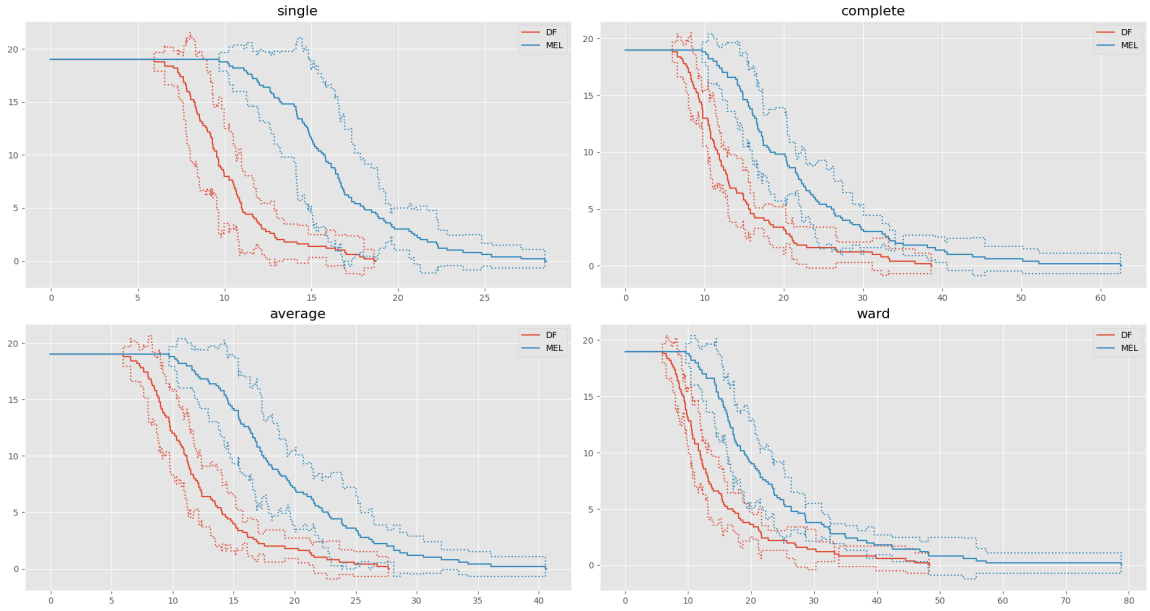
(a) Average H_0 stable ranks for all classes. Per class 100 images were used. The 100 images were split into 5 samples of 20 images each, for which an individual stable rank was calculated to contribute to the average one. No standard deviation is shown for overviews sake.



(a) The same result as the above, $2-\sigma$ bounds added.

Taking a look at the above, we can observe that in terms of average stable rank signatures, we can make out a rather large separation between the class *MEL*, the class *DF* and the rest of the classes. Taking a look however of the standard deviations obtained, we can also observe that the bounds

are rather large. As a consequence, it could be suspected that the edge detection pre-processing amplifies discrepancies that are between *MEL*, *DF* and the rest of the skin lesion types. Taking a look at 2 most distinguishable classes, *MEL* and *DF*, we can plot their signatures individually:



(a) Classes *MEL* and *DF* corresponding to the most seperated classes, shown with an estimated $2-\sigma$ bounds.

Observing the above, we see that the signature in high dimensional space is sufficiently distinguished in order to avoid a large overlap between the 2-σ uncertainty bounds, with the most promising seemingly being the single linkage, from a first observation. It should however be noted that these bounds are determined empirically through the five average stable rank curves per class, thus an additional approximation error here is expected to be present. It is suspected that a there may be a balance that needs to be struck between how many samples of the high dimensional space are required to compute meaningful empirical statistics, as opposed to how detailed should each sample be to approximate a hopefully existing unique signature of the high dimensional set of images of the respective class.

Assume that the high dimensional space of all possible images for a certain lesion class exhibits a certain signature in the H_0 stable rank. Now assume that we have at hand a set of disjoint point clouds, thus no images are shared between them. We would suspect that as the amount of images in each cloud increases, we would in the limit increasingly approximate the global space to a better and better degree. Thus we would suspect that with a sufficient level of detail in each point cloud, so for an increasingly high number of images per cloud, the independently obtained stableranks may converge to the signature in question, as we would approximate the high dimensional region of this specific class to a higher and higher degree. (Assuming a certain notion of continuity).

Thus we also investigate a slightly extended dataset, consisting at 600 images for the classes *NV, MEL, BKL*, 500 images for *BCC*, 300 for *AKIEC* and lastly 100 for *DF* and *VASC*. The different amount of images results from the problem of not all classes having the same amount of images available in the original data set. The aim now is to investigate whether the higher amount of images less diverse stableranks among class point clouds, which could be an indication that indeed

such a signature could exist.

Note that for the question of comparing different class stableranks, the image count per point cloud need to be consistent in comparison of signatures, as otherwise trivially the 0th-homology will start at a different value, corresponding to the amount of points in the respective point clouds generated.

Thus now we generate two disjoint point-clouds for the classes *NV, MEL, BKL* with the most amount of images available and plot them. We conduct this for no contrast added and a very large contrast enhancement by the `PIL` library, that results in an almost binary image.

The result is as follows:

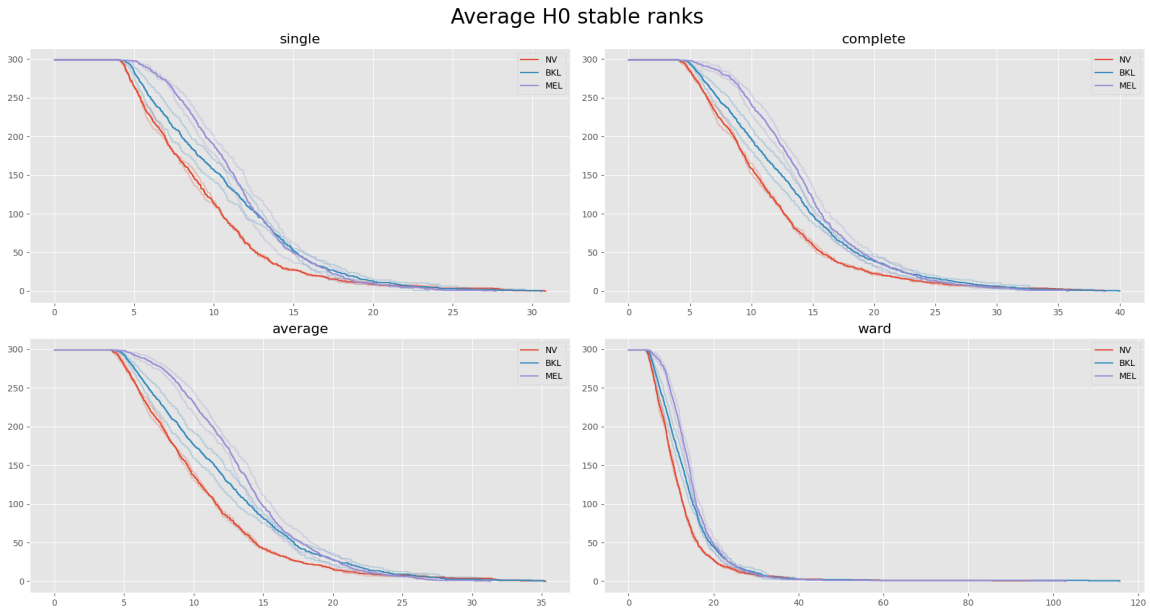


Figure 7: Sample size per point cloud: 300 images, corresponding to only two stable rank calculations per class, indicated in the opaque lines, with the average being the solid line.

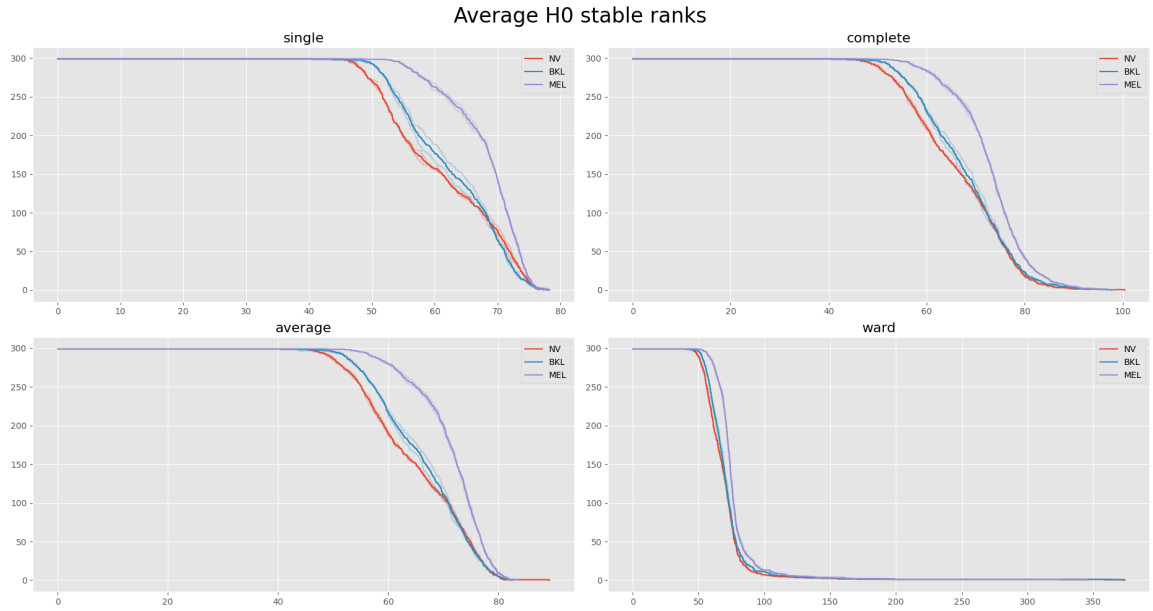


Figure 8: A very large contrast factor of 10^5 was used here, making the image almost binary, leaving only small exceptions for pixels not in $\{0, 255\}$

As can be seen, the two disjoint point clouds per class produce two strongly similar signatures for a given class and especially for a limiting contrast value a big separation is obtained for the melanoma. Furthermore from a first look, it seems that the relative discrepancies between the two stableranks for each class are also reduced with the increase in contrast.

However it is suspected that it is not only the structure and presence of the edge regions alone that contributes to bringing out underlying structural differences between the high dimensional sets: Running the same analysis, this time however on binary images, thus having intensity values between $\{0, 255\}$ only , we obtain the following result:

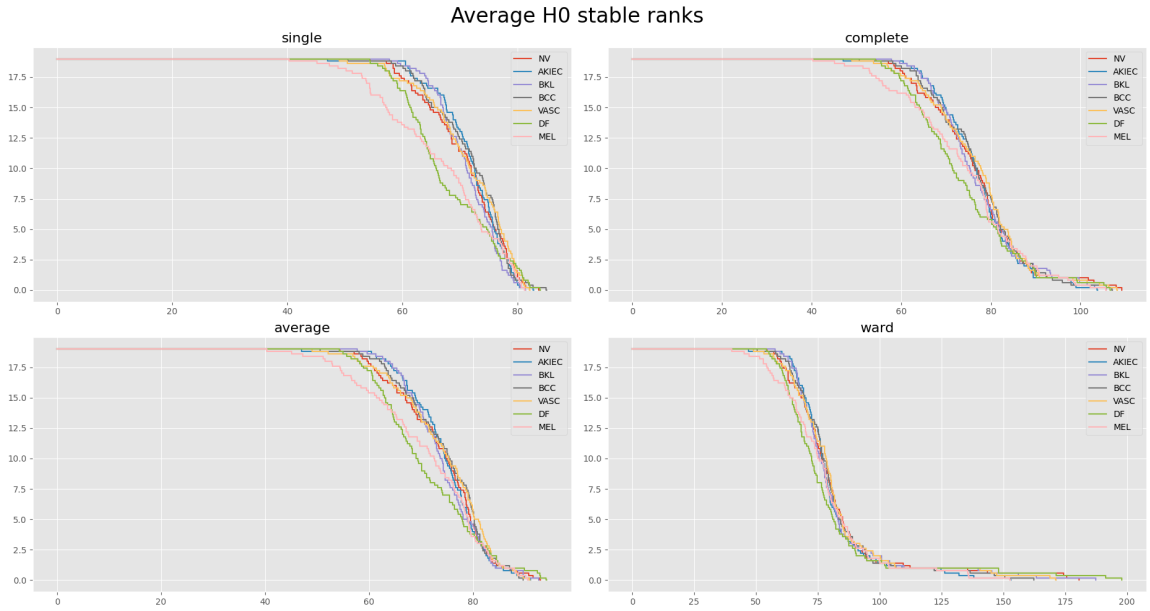
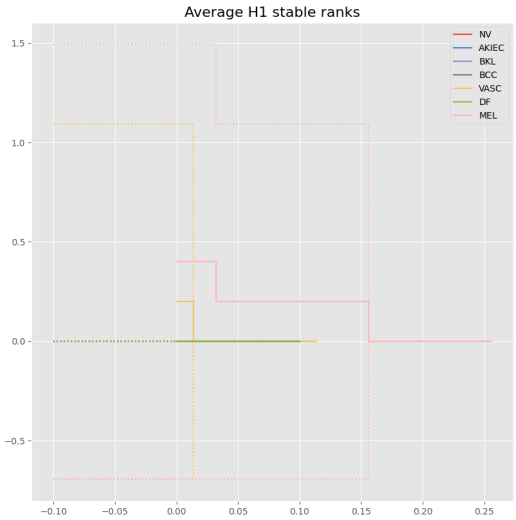


Figure 9: Same experiment under binary image instead of intensity valued edges.

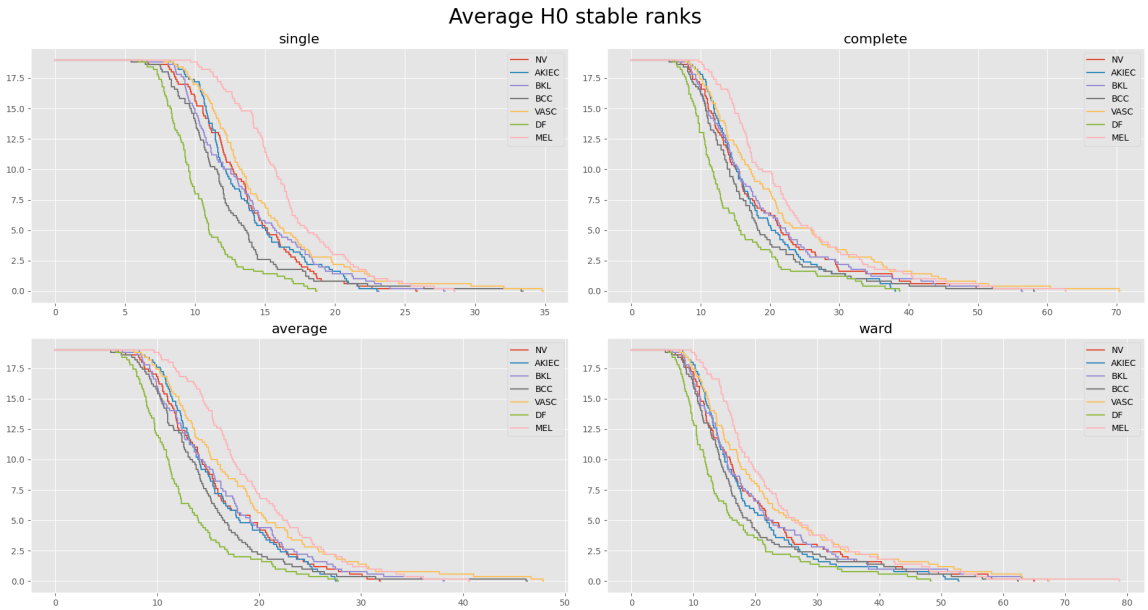
As seen above, qualitatively speaking the topological signature for the *MEL* class has changed drastically, falling more in line with the rest of the classes, to a varying degree of severity depending on the linkage used. The standard deviation is also qualitatively similar, as in the large amount of overlap to the original embedding, thus not shown for brevity's sake.

Lastly, we can consider the H_1 homology between the classes. This is included for completeness for 100 images per class point cloud. However no sufficient indication was found that for the sample sizes available, unique signatures with smaller bounds could be readily obtained with the data set at hand, thus making the first homology in this form less attractive.

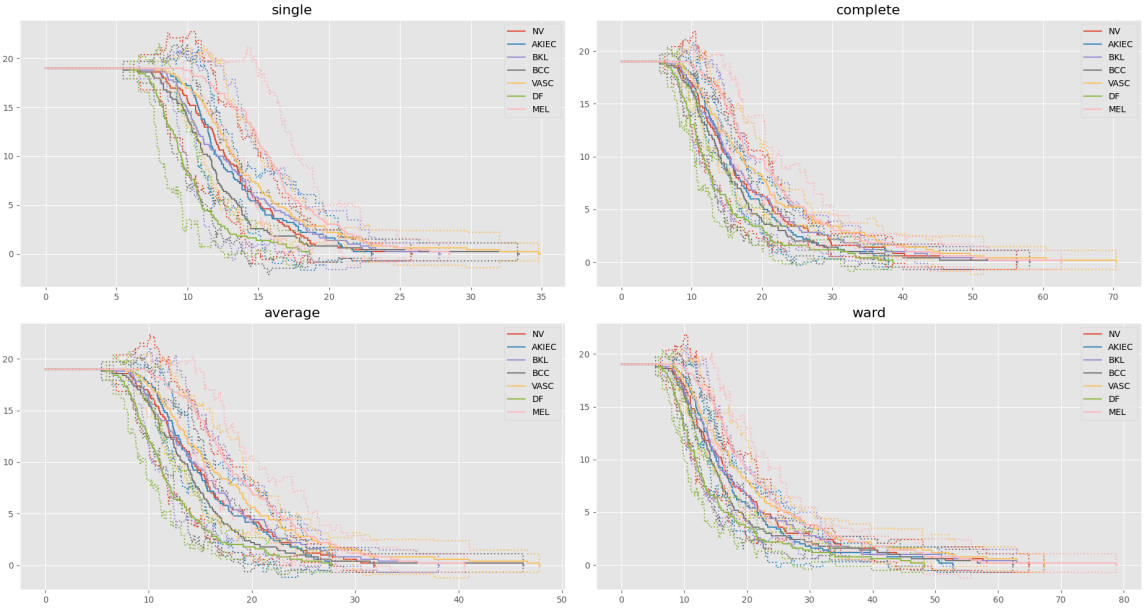


High-dimensional Embedding - Direct greyscales

Based on the previous analysis one may ask if similar behaviour is found if the original images in the grey-scale. Applying the same embedding, we obtain for the data set of 100 images of 'original' (from `PIL` library) grey-scale values per class:

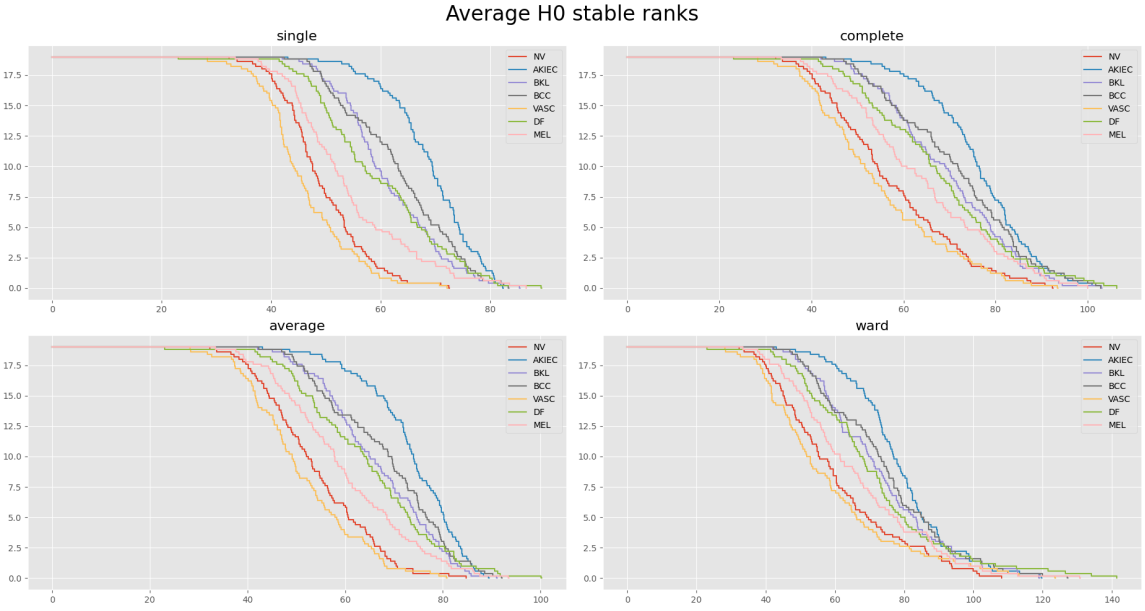


(a) Average H_0 stable ranks for all classes. Per class 100 images were used. The 100 images were split into 5 samples of 20 images each, for which an individual stable rank was calculated to contribute to the average one. No standard deviation is shown for overviews sake.

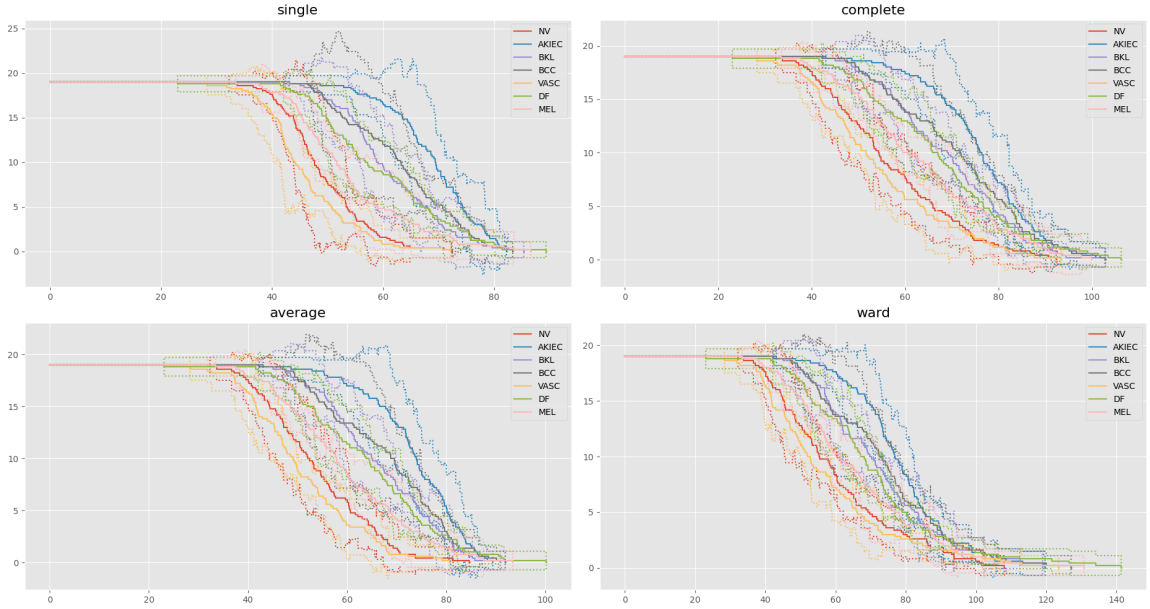


(a) The same result as the above, 2-σ bounds added.

If we increase the contrast again towards a limiting value, we obtain:



(a) Average H_0 stable ranks for all classes. Per class 100 images were used. The 100 images were split into 5 samples of 20 images each, for which an individual stable rank was calculated to contribute to the average one. No standard deviation is shown for overviews sake. The contrast factor was set to 10^5.



(a) The same result as the above, $2 - \sigma$ bounds added.

Although a large degree of overlap between the estimated standard deviations is present, we observe that the original images seem to separate the classes better individually as opposed to the edge detection. We now consider once more the extended data set. And first we split each class of 600 images into two point clouds each and calculate the average between the two resulting stable ranks for each, shown in Figure 15. We then split all of the classes in either two or three point clouds and do the same again, shown in Figure 5.2:

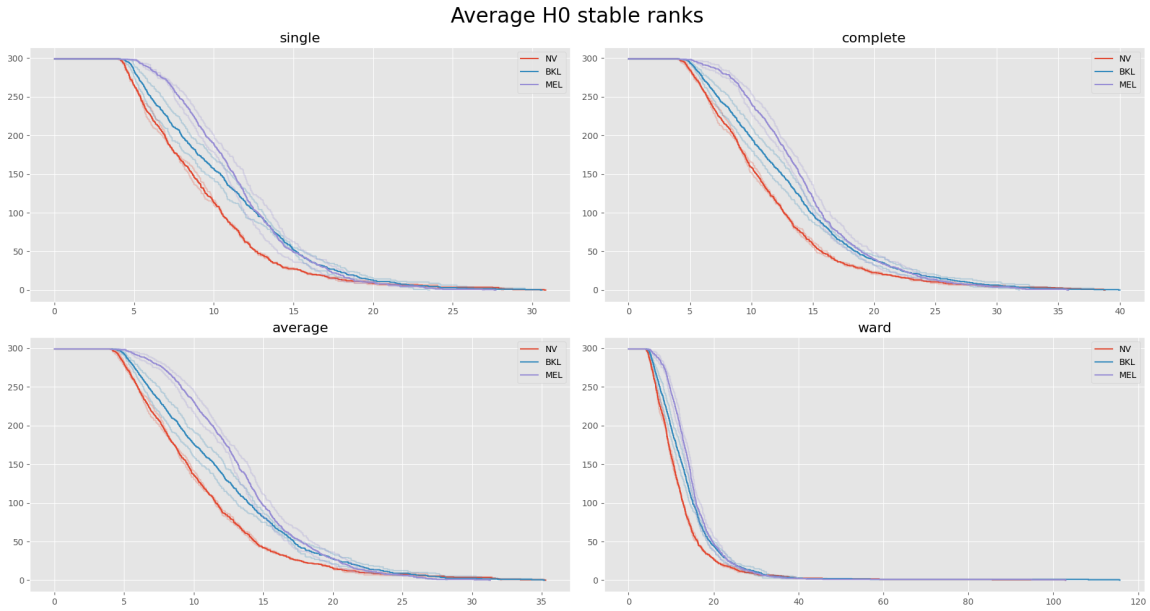


Figure 15: Point clouds of 300 images each for each class of 600 images were used to generate the opaque stable ranks.No contrast was applied.

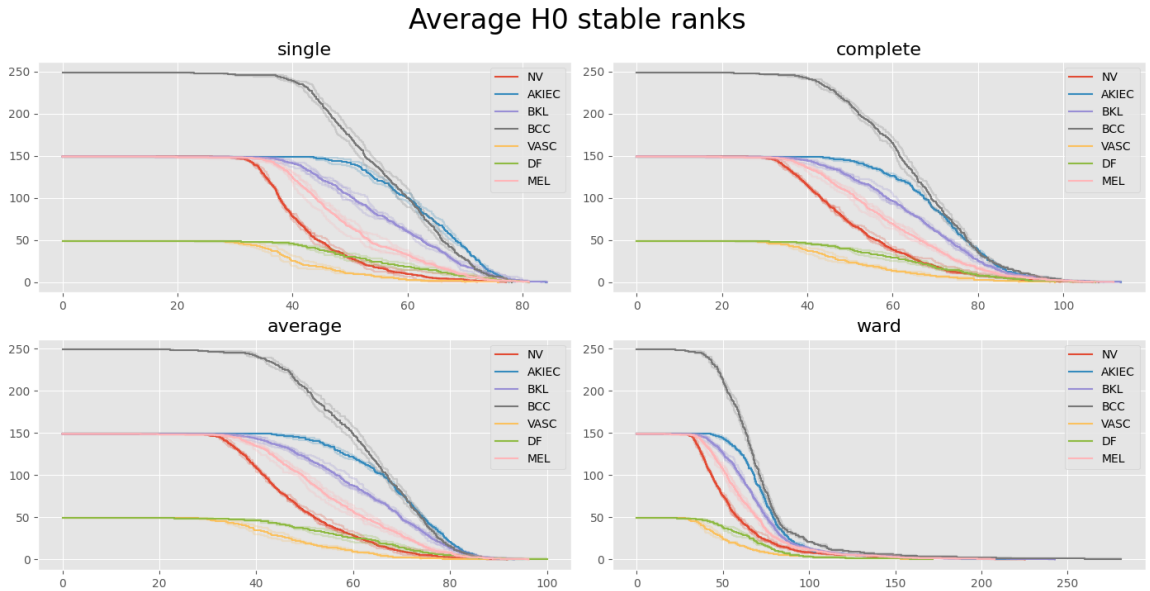


Figure 16: Analysis applied with a limiting contrast factor of 10^5 for two or three pointclouds for each data.

It should be noted that in the above, only signatures with the respective amount of images in the point cloud, i.e. the same starting value, should be compared. Again, the obtained stable ranks of each disjoint point cloud are plotted in the respective colour in opaque. Once more, we see that

there could be an indication that unique signatures may be approached in the limit of high detail per point cloud.

Lastly, again we show the H_1 signatures for completeness:

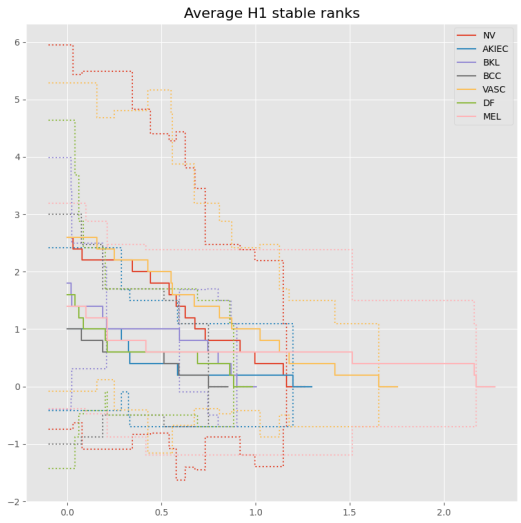


Figure 17: H_1 stable ranks for each class, with added estimated $2-\sigma$ bounds, drawn in dashed.

Again for the H_1 stable ranks we see that the bounds obtained from the sample standard deviation is very large. Although convergence behaviour was somewhat observed for higher image counts per class, the results were again poor and thus again the decision was made to not conduct any further analysis on this approach.

Unfortunately, although these result seem interesting, there is no straightforward way to take the information as obtained through the signatures of the image sets and use it in a more simple classification approach such as SVM, as fundamentally speaking, the comparison here is between the signatures of image sets and not individual images.

3D Embedding

Having analysed certain global properties of the high-dimensional space of image samples, we will now return to the classification task of distinguishing the seven different types of skin lesions. Considering the pre-selection results displayed in Table 1, we will run the persistent homology pipeline after applying the TDA no mask pre-processing and 3D point cloud embedding to the image samples, for an illustration see Figure 18.

(a) original image

(b) TDA no mask preprocessed image

(c) 3D point cloud

Figure 18: Prepoceesing and embedding appiled to 3D point cloud generation

The associated H_0 and H_1 stable ranks are shown in Figure 19 and Figure 19. Firstly, we observe the global behavior of topology through H_0 and H_1 stable rank. It is worth noting that the standard deviation of each stable rank was illustrated to exhibit to what extent we can disentangle each class with TDA tools. Then, we perform a machine learning prediction based on SVMs, whose feature mapping function is the inner product between stable ranks. Figure 19 describes the average of H_0 stable rank amongst different classes, calculated with four different extension functions: single linkage, complete linkage, average linkage, and ward linkage. Solid lines stand for the average of H_0 stable rank, and dashed lines represent standard deviations(2-σ confidence interval) in each class.

It can be seen that the 2-σ confidence intervals for each class overlap each other, indicating that the variance of the H_0 stable rank is too large to successfully separate the classes in terms of the H_0 stable rank. The other observation, coinciding with the outcome of pre-selection, is that, amongst four different linkage functions, single extension function made the biggest difference between classes. This superiority leads better performance of the SVM classification based on the single linkage function we discuss later in subsection 5.3.

This result points out that even though the combination of the TDA no mask pre-processing and the single linkage extension was better at dissimilarity score than other methods, it is not so powerful to distinguish each class through H_0 stable ranks.

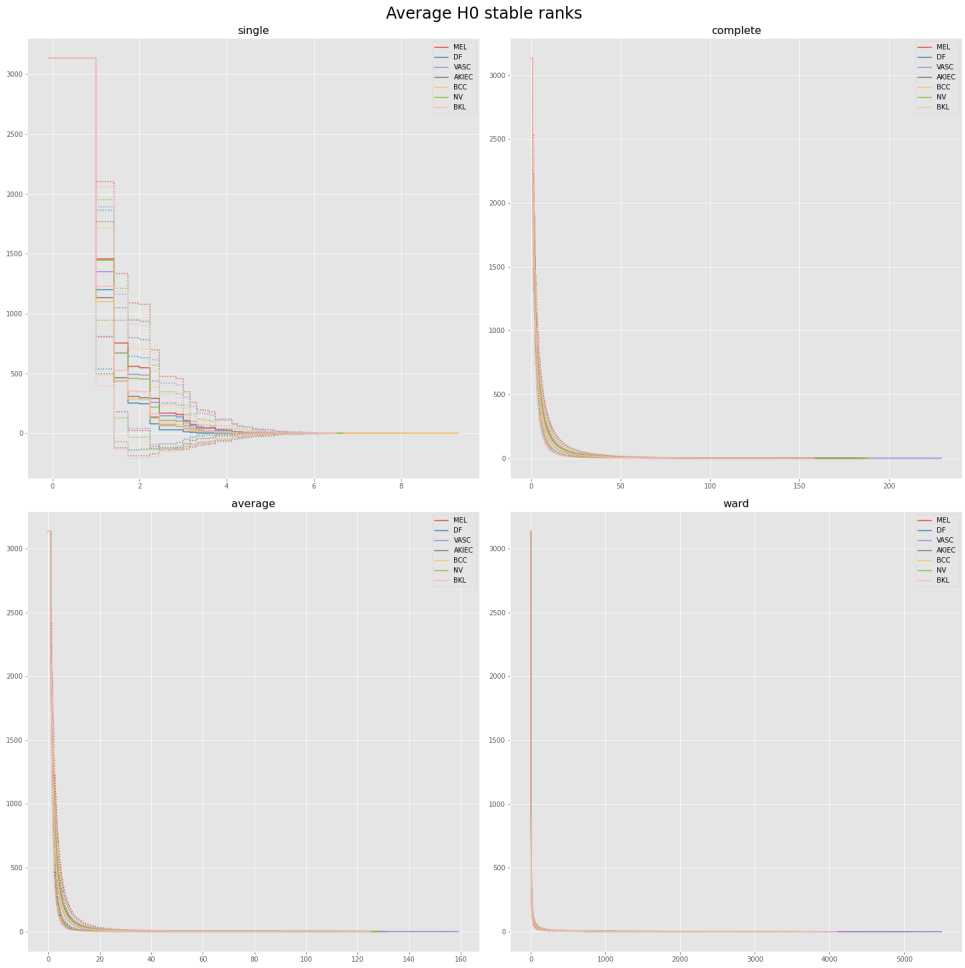


Figure 19: Average H_0 stable ranks of 3D embedding

Figure 20 shows that the average of H_1 stable rank of 3D embedding (solid line), and its corresponding standard deviation (dashed line). As with H_0 stable rank visualization, we observe that classes are indistinguishable because their confidence intervals overlap each other. To summarize, we could not manage to extract beneficial information in terms of clustering classes through the number of connected components (H_0 stable rank), and the number of holes (H_1 stable rank) in this embedding.

These results, furthermore, implies that we need to tune other hyper-parameters we fixed in this experiment, such as the window size of image smoothing, and the weight between pixel coordinates and intensity, which is necessary when mapping images to 3D point clouds.

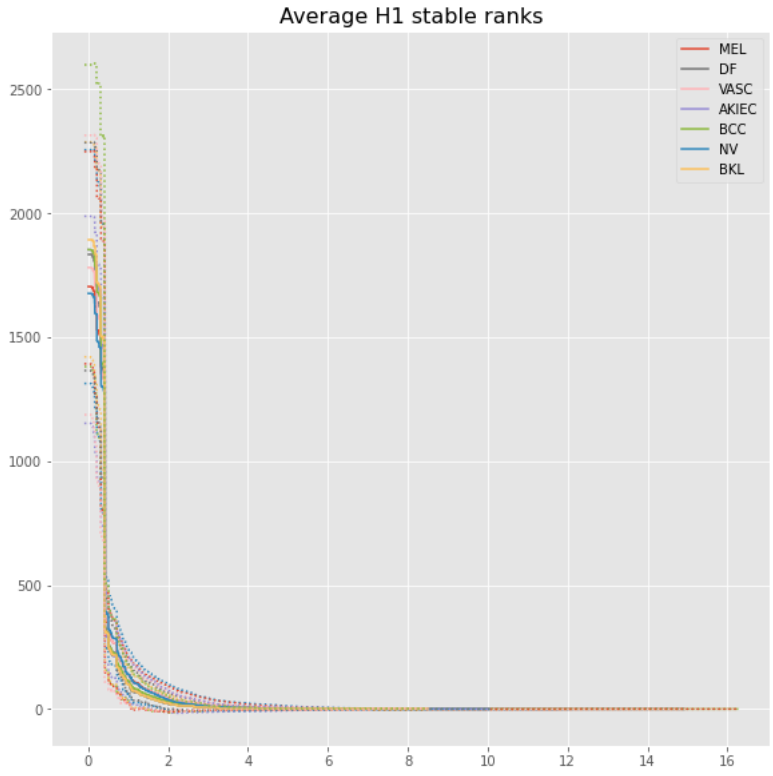


Figure 20: Average H_1 stable ranks of 3D embedding

2D Embedding

In the case of 2−dimensional embedding Table 1 suggests that results may be most promising when pre-processing the images with TDA-methods and applying no binary mask. The average stable ranks and standard deviations obtained are shown in the figures below. We again distinguish between H_0 stable ranks and H_1 stable ranks.

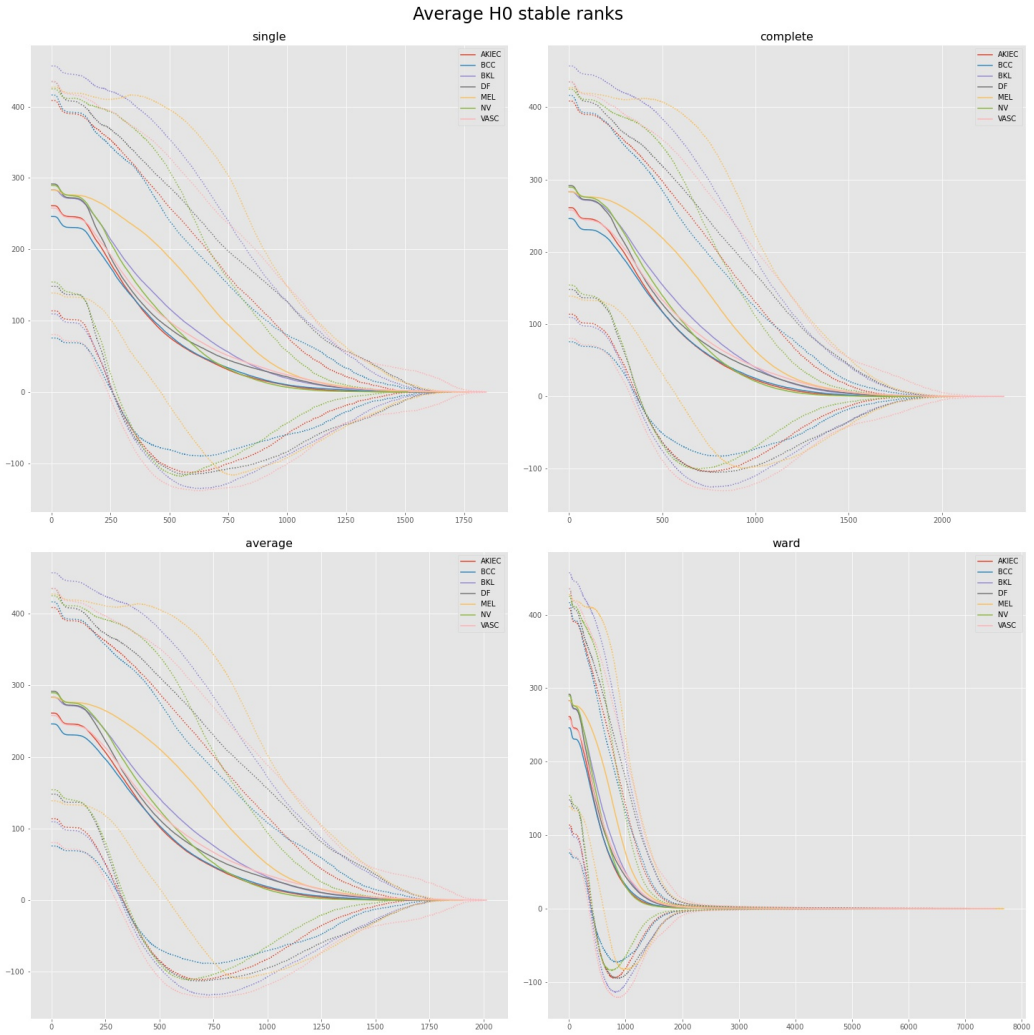


Figure 21: Average H_0 stable ranks of 2D embedding

In Figure 21 above, observe that for each extension the average H_0 stable ranks of the different classes are essentially indistinguishable except for the type *MEL*. Further, we observe that the standard deviations are quite large for each class which raises the concern that the stable ranks vary to a large degree even within each class. All things considered, this indicates that persistent homology was unfortunately not able to capture the distinct geometries of the different classes or that the these types of skin lesions are indiscernible for the H_0 homology in the first place. This observation has also been confirmed with the poor accuracy score of the SVM classifier, see Table 2.

We further observe, that the *average* H_0 stable ranks for *MEL* have a slightly higher curvature suggesting that connected components dissolve more slowly over time. This raises different interpretations of the geometrical structure of type *MEL*: One possibility is that *MEL* may be differentiated from other classes because it has on average more connected components. However, at this point it is difficult to pinpoint the exact reason for the behavior of the average H_0 stable rank of *MEL*. One would have do conduct further studies or call on experts for skin diseases.

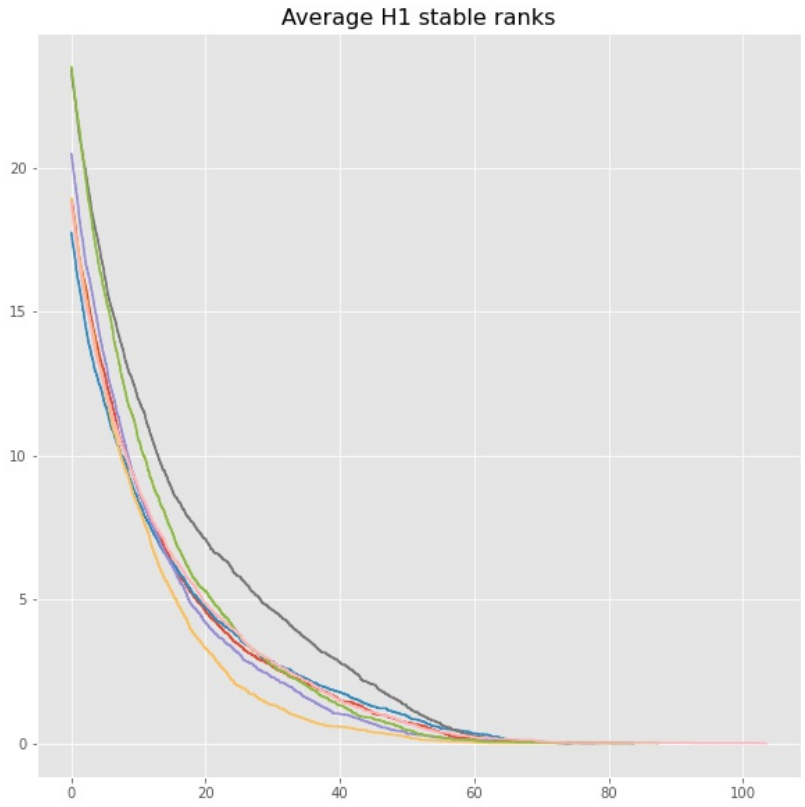


Figure 22: Average H_1 stable ranks of 2D embedding

For the average H_1 stable ranks we make the same conclusion as in the H_0 case: They are mostly indiscernible making it infeasible for the SVM classifier to properly differentiate between the different types of skin lesions. This is again confirmed with a low accuracy score, displayed in Table 2

It is interesting to observe that here type *MEL* has the deepest curvature while type DF has the highest one, exactly reversed to the H_0 case. This might be due to the fact that H_0 and H_1 homologies extract different geometries from the image samples - H_0 is concerned with connected components while H_1 examines one-dimensional holes.

5.3 SVM results

Recalling from subsection 5.2 that there is a low visual discrepancy between the average homology stable ranks of different classes, we assume that the hypothesis of linear separability of classes in L_2 Hilbert space may not be plausible. Nevertheless, we still have to test it by training a kernel-based support vector classifier (SVC) using 4 different kernels: 1) 0-th homology kernel, K_{H_0}, 2) 1-st homology kernel, K_{H_1}, 3) respective sum $K_{H_0} + K_{H_1}$, and 4) element-wise product $K_{H_0} \odot K_{H_1}$. Specifically, K_{H_0} and K_{H_1} are constructed as a standard inner product in L_2 space and with a little refinement we can show that the sum and element-wise product of the two kernels is also a kernel satisfying Mercer's condition. Here, note that we leveraged ward linkage extension function for 2D embedding H_0 homology calculation, and single linkage extension function for 3D embedding, based on stable rank analysis in subsection 5.2.

Additionally, to make our analysis computationally feasible for 3D point clouds, we set the size of the images in the train set to 48×48, and constructed kernels by 25 images per class, instead of

100 images per class.

	K_{H_0}	K_{H_1}	$K_{H_0} + K_{H_1}$	$K_{H_0} \odot K_{H_1}$
2D	0.179	0.166	0.180	0.165
3D	0.240	0.234	0.217	0.263

Table 2: 5-fold cross-validation mean scores

Table 2 shows that the cross-validation scores are generally poor, i.e. the performance of the trained SVC is slightly better than the expected performance of a random guess classifier ($\approx 14\%$ accuracy). Taken together, the poor classification results confirm what we have already concluded from a visual inspection of stable ranks plots (see fig. 19, 20, 21, 22). The best CV-accuracy of 26% was achieved from a K_{H_0} kernel based on stable ranks for single linkage extensions in the case of 3D point cloud embedding, which still results in a poorly generalising SVC. Comparing 0-th and 1-st homology kernels, we see that 0-th ones are slightly more informative for SVC training in both 2D and 3D point clouds settings. In addition, for all kernels considered, the 3D embeddings are superior to the 2D counterparts in terms of classification accuracy. However, our expectations that one of the combinations of these basic kernels could improve the classification results did not come true as also seen in Table 2.

The results of our work cannot be directly compared to [10] that deals with the same diagnosis problem but applies a topological approach relying on cubical complexes and trains SVC based on the kernels constructed from so-called persistence curves and persistence statistics for much bigger size data. As in our case all group members possess general usage computers that do not allow the full analysis of the original ISIC data with 10015 images in total but instead we run the experiments on the train set with a maximum of 700 images (100 images per class).

6 Summary and Conclusion

The aim of the analysis conducted here was to investigate the possibility of significant topological signatures of different types of skin lesions, while making an attempt to classify them based on the information generated from topological data analysis.
For the high dimensional embedding it was found that there is a good indication that the images embedded in high dimensional space show signs of differentiating signatures. For the pre-processing step of edge detection it was found that point clouds generated from the class of Melanomas are distinguished from the rest of the classes. Here potential conclusions could be drawn on how the pre-processing step of edge detection may be related to micro-structural or texture differences between the classes, which should be left to an expert in the field. Furthermore it may be concluded that while utilizing this pre-processing step, the space of melanomas seems to be most distinguished from the space of Dermatofibromas (DF).
For the more general pre-processing step of using the cropped and scaled original images it was found that the classes seem to be exhibited more broadly separated signatures. It was found that under a larger number of images in the point cloud, empirically convergence seems to occur, also for the edge detection pre-processing step. However it is clear that this is merely an indication of such a signature existing and not a proof.

The 2D embedding in conjunction with the TDA pre-processing showed similar results: H_0 as well as H_1 average stable ranks were merely indistinguishable and very high standard deviations could be observed. This suggests that a unique signature per class could not be extracted and one may need to revert to other embeddings and pre-processing techniques.

Coming to the 3D embedding, there is also a problem of highly overlapping regions between stable ranks originating from different classes. Suggesting that switching to this embedding but keeping the same pre-processing setting may not result in more distinguishable geometrical patterns of different classes.

In terms of classification, an effort could be made to investigate the possibility of classifying individual images for the high dimensional embedding. This would require a more detailed look into what mechanisms would allow the determination of the most likely image set in high dimensional space (e.g. how large is the change induced in the topological signature by combining the known sets with the image in question by some adequate measure).
Furthermore, for the 3D embeddings there are parameters to be tuned: the weighting between intensity and distance information, the appropriate contrast of the images as well are all parameters that have not been exploited to their full extent in this project. An interesting investigation could be made into the relationship between e.g. the stable rank separation and the contrast at hand. Furthermore, other methods of transforming to intensity values could be considered, apart from only a linear combination of the RGB values.

References

[1] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. The HAM10000 dataset: A large collection of multi-source dermatoscopic images of common pigmented skin lesions. *CoRR*, abs/1803.10417, 2018.

[2] Noel C. F. Codella, David A. Gutman, M. Emre Celebi, Brian Helba, Michael A. Marchetti, Stephen W. Dusza, Aadi Kalloo, Konstantinos Liopyris, Nabin K. Mishra, Harald Kittler, and Allan Halpern. Skin lesion analysis toward melanoma detection: A challenge at the 2017 international symposium on biomedical imaging (isbi), hosted by the international skin imaging collaboration (ISIC). *CoRR*, abs/1710.05006, 2017.

[3] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[4] Christopher Tralie, Nathaniel Saul, and Rann Bar-On. Ripser.py: A lean persistent homology library for python. *The Journal of Open Source Software*, 3(29):925, Sep 2018.

[5] W. Chachólski. Topological data analysis - sf2956 lecture notes. 2022.

[6] O. Gevaert R. Vandaele, G. A. Nervo. Topological image modification for object detection and topological image processing of skin lesions. 2020.

[7] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[8] Robin Vandaele, Guillaume Nervo, and Olivier Gevaert. Topological image modification for object detection and topological image processing of skin lesions. *Scientific Reports*, 10, 12 2020.

[9] Ulrich Bauer. Ripser: efficient computation of vietoris-rips persistence barcodes. *Journal of Applied and Computational Topology*, 2021.

[10] Yu-Min Chung, Chuan-Shen Hu, Austin Lawson, and Clifford Smyth. Topological approaches to skin disease image analysis. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 100–105. IEEE, 2018.

7 Appendix - Code

This section includes the main code used to conduct the analysis of this project. It should be noted that the code provided here is not the complete work, the decision was made to include the most important files. I.e. The main evaluation scripts for each embedding, as well as the utilities file. An invite to the private repository can be given if desired, alternatively the complete code will be also provided in a `.zip` file together with the report in the final submission. Please note that in the following, **BROKEN** indicates a broken line.

7.1 Genereal Code

`Utilities.py`

The utilities containes a number of functions that were used throughout the different experiments. E.g. the folder generation of preprocessed images, the pre-processing implementation itself etc.

```
import numpy as np
import matplotlib.pyplot as plt
from topimgprocess import TDAimgprocess as TIP # topological image modification and
    BROKENprocessing
import os # os independent path construction
import time # tracking computation time
import random # setting seeds
import matplotlib.pyplot as plt # plotting
import matplotlib.image as mpimg # handling images
import PIL # imaging library
from skimage.segmentation import chan_vese # chan_vese segmentation
from skimage.segmentation import slic # k-means clustering based oversegmentation
from skimage.filters import roberts # Roberts' cross operator for edge detection
from skimage.morphology import convex_hull_image # construct convex hull of a binary
    BROKEN image
import skimage.filters as filters # infer ISODATA thresholds from images
from skimage.segmentation import active_contour # active contour segmentation
import cv2 # binarize images
import ripser as rp
import matplotlib.pyplot as plt
from pathlib import Path
import scipy.sparse as sp
import stablerank.srank as sr
from PIL import Image, ImageEnhance
import scipy.spatial as spatial
import collections
'''
Note: topimgprocess is taken from [1]:
TY  - JOUR
AU  - Vandaele, Robin
AU  - Nervo, Guillaume Adrien
AU  - Gevaert, Olivier
PY  - 2020
DA  - 2020/12/03
TI  - Topological image modification for object detection and topological image
    BROKENprocessing of skin lesions
JO  - Scientific Reports
SP  - 21061
VL  - 10
IS  - 1
AB - /
SN  - 2045-2322
UR  - https://doi.org/10.1038/s41598-020-77933-y
DO  - 10.1038/s41598-020-77933-y
ID  - Vandaele2020
ER  -
```

```
'''


def cut_to_roi(image,binary_mask, buf = 2, mask = False, cut = True):
    '''
    Cuts down image to the region of interest given by a
    binary mask.

    inputs:
    image -  (m , n) array of intensity values
    binary_mask - (m, n) array of binary values
    buffer: buffer in pixels around region of interest
    mask: applies mask to image prior to return

    Output:
    image - (k + 2*buffer, l + 2*buffer)
    '''


    #Returning cut image
    if mask == True:
        image = image * binary_mask
    if cut == False:
        return image

    #Finding indices, i: horizontal, j - vertical
    #regular shape:
    image_shape_j, image_shape_i = image.shape
    index_set_j, index_set_i = np.where(binary_mask == 1.0)
    #Setting maximum boundaries, taking into account possible
    #boundary violations:
    i_max = min(np.max(index_set_i) + buf, image_shape_i - 1)
    i_min = max(np.min(index_set_i) - buf, 0)

    j_max = min(np.max(index_set_j) + buf, image_shape_j - 1)
    j_min = max(np.min(index_set_j) - buf, 0)

    return image[j_min : j_max, i_min : i_max ]



def pre_process(image, window_size = None, border_width = None, sub_sampling_step =
    BROKEN1,mode= 'processed',
                mask = False, cut = True, PIL_compression = False, compression_den =
                    BROKEN 2, plot = False,
                save_folder_path = None, edge_detection = False):
    #Complete pre-processing step for a given image
    '''

    :param image: (M, N) array of intensity values
    :param window_size: Extent of smoothing window for segmentation algorithm in [1]
    :param border_width: Extent of borders for border modification in [1]
    :param sub_sampling_step: Take every step-th value in the array in each
        BROKENdirection. Used to reduce size.
    :param mode: ['processed': Returns full pre-processing from [1], 'modified':
        BROKENReturns intermediate pre-processing step,
    'base': Returns essentially original intensities], for all: mask can be applied
        BROKENor not.
    :param mask: Applies binary mask or not on the final pre-processed image.
    :param cut: Cut to region of interest to save memory.
    :param PIL_compression: Downsampling a given image with PIL. True/False: Built
        BROKENin sub-sampling supplied by PIL-library.
    Carried out after image modification at the end of the pre-processing pipeline
```

```
    BROKENas of now. Standard in PIL now is NEAREST.
:param plot: Enable plotting function to show pre/post pre-processed image
:param compression_den: Compression denominator, the shape will be compressed to
    BROKEN int(original_shape / compression_den)
:param save_folder_path: Path where pre-processed image will be saved to. If
    BROKENNone is given, it will not be saved.
:param edge_detection: Canny algorithm to implement edge detection.
:return: returns (K, L) array of intensity values with pre-processing steps
    BROKENtaken. Note that (K,L)
represents the shape of the image after the region of interest has  been cut out
    BROKEN.

'''
#Setting standard values in case nothing is supplied, from [1]:
if (border_width is None) or (window_size is None):
    shape = image.shape
    #Diagonal length
    diagonal = np.linalg.norm(shape)
if border_width is None:
    border_width = np.ceil(diagonal / 100).astype(int)
if window_size is None:
    window_size = np.ceil(diagonal / 25).astype(int)



#Generating
image_grey = np.asarray(PIL.Image.fromarray(image).convert('L'))
#Noise as introduced in paper, to keep track of
#Note: Base will return the noiseless image, other options will have the
    BROKENprocessed image data, which includes
#A certain amoung of noise, as introduced in the paper, to be able to
    BROKENdistinguish the individual pixels,
#which is required in [1].
#Non_noise data saved.
image_data = np.copy(image_grey)
image_grey = image_grey.astype(float) + 0.01 * np.random.randn(image.shape[0],
    BROKENimage.shape[1])

# Full pre=processing according to [1]
TIP_img = TIP.topological_process_img(image_grey, window_size=window_size,
    BROKENborder_width=border_width)

# Cutting down drastically on size, otherwise ripser has no chance, for
    BROKENintensities
# the processed image is taken for now
#Corresponds to full processed step
intensities = TIP_img['processed'][::sub_sampling_step, ::sub_sampling_step]
#Corresponds to binary mask - thus components
binary_mask = TIP_img['components'][::sub_sampling_step, ::sub_sampling_step]
#Corresponds to intermediate step taken without Backround gradient adjustement.
intermediate_intensity = TIP_img['modified'][::sub_sampling_step, ::
    BROKENsub_sampling_step]

# Working in cut-down version, for region of interest.
if mode == 'processed':
    image_data = intensities
if mode == 'modified':
    image_data = intermediate_intensity
if mode == 'base':
    image_data = image_data
if mode not in ['modified', 'processed', 'base']:
    print('Select suitable mode, mode ' + mode + ' not available.')
    exit(0)
```

```
    image_data = cut_to_roi(image_data, binary_mask, 4,mask = mask, cut = cut )

    #Downsampling with PIL if desired
    if PIL_compression:
        desired_shape = image_data.shape
        desired_shape = desired_shape[0] // compression_den, desired_shape[1] //
            BROKENcompression_den
        #Applying lancos downsampling, standard is NEAREST, for speed.
        image_data = np.asarray((PIL.Image.fromarray(image_data).convert('L')).
            BROKENresize((desired_shape[::-1]), PIL.Image.NEAREST))
    #Canny edge detection
    if edge_detection:
        # define thr1 and thr2, hyperparameters of Canny's edge detection, as q-th
            BROKENpercentile of the image
        thr1 = np.percentile(image_grey, 0.10)
        thr2 = np.percentile(image_grey, 0.90)
        if mode == 'processed' or mode == 'base':
            image_data = cv2.Canny(np.uint8(np.round(image_data)), thr1, thr2) / 255
                BROKEN * image_data
        else:
            print('Edge detection is only applied to the processed images or the
                BROKENoriginal greyscale images.')
            exit(0)
    if plot:
        figures, axis = plt.subplots(1,2)
        axis[0].imshow(image_grey,cmap='gray')
        axis[0].set_title('Before Pre-proc')

        axis[1].imshow(image_data,cmap='gray')
        axis[1].set_title('After Pre-proc')

        plt.show()

    if save_folder_path is not None:
        image = PIL.Image.fromarray(image_data)
        image = image.convert('L')
        image.save(save_folder_path, 'JPEG')


    return image_data


def generate_pre_processed_data_sets(image_count = None,  folder_prefix = ['base', '
    BROKENprocessed'], alternate_suffix = [None,None], window_size = [None, None],
    BROKENborder_width = [None, None], sub_sampling_step = [1,1],mode= ['base','
    BROKENprocessed'],
                mask = [True, True], cut = [True, True], PIL_compression = [True,
                    BROKENTrue], compression_den = [2,2], plot = [False, False],
                                    edge_detection = [False,False],
                                        BROKENoriginal_folder_name = ['
                                        BROKENISIC2018_train','ISIC2018_train']):
    '''
    :param image_count: How many images are selected out of the folder. If all are
        BROKENtaken, [None, None] is to be supplied.
    Else, the folder is subsampled with random images of the folder, according to
        BROKENthe amount. If a dict is passed, is should be of lenght
    n. of classes, with the desired amount of images per class, in the keys of the
        BROKENclass names.

    :param folder_prefix: Prefix in the naming of the folder corresponding to the
        BROKENpre-processing step. Note: If an image exists,
    in the folder already, then the image is added again.

    Note: Following arguments as in pre_process, however now should be supplied in
```

```
    BROKENlist format. E.g. if to different
pre_processings are to be applied, the supply a list of length 2 for each
    BROKENargument.

:param window_size: Extent of smoothing window for segmentation algorithm in [1]
:param border_width: Extent of borders for border modification in [1]
:param sub_sampling_step: Take every step-th value in the array in each
    BROKENdirection. Used to reduce size.
:param mode: ['processed': Returns full pre-processing from [1], 'modified':
    BROKENReturns intermediate pre-processing step,
'base': Returns essentially original intensities], for all: mask can be applied
    BROKENor not.
:param mask: Applies binary mask or not on the final pre-processed image.
:param cut: Cut to region of interest to save memory.
:param PIL_compression: Downsampling a given image with PIL. True/False: Built
    BROKENin sub-sampling supplied by PIL-library.
Carried out after image modification at the end of the pre-processing pipeline
    BROKENas of now. Standard in PIL now is NEAREST.
:param plot: Enable plotting function to show pre/post pre-processed image
:param compression_den: Compression denominator, the shape will be compressed to
    BROKEN int(original_shape / compression_den)
:param alternate_suffix: Gives an alternate suffix instead of the folder origin
    BROKENname.

:return: None, simply used to construct full directory.
'''
print('------Warning: Image folder size is hardcoded to 115------ /n')
variants = len(folder_prefix)

#Iterate over pre_processing variants
#Deal with image counts
for k in range(variants):
    if image_count is not None:
        if type(image_count) == type({}):
            image_indices_dict = {}
            for c in os.listdir('./' + original_folder_name[k] + '/'):
                class_path = './' + original_folder_name[k] + '/' + c
                images_in_class = os.listdir(class_path)
                image_indices_dict[c] = random.sample(range(len(images_in_class)
                    BROKEN), image_count[c])
                #images_per_class = image_count[c]

        # Note: Folder size hardcoded here, 115 images.
        else:
            image_indices = random.sample(range(115), image_count)
            images_per_class = image_count
    if image_count is None:
        images_per_class = 115


    #Generate folder with indicated pre_fix corresponding to the folder in
        BROKENquestion.
    if alternate_suffix[k] is None:
        folder_name = folder_prefix[k] + '_'+ original_folder_name[k]
    else:
        folder_name = folder_prefix[k] + '_'+ alternate_suffix[k]
    Path('./'+folder_name).mkdir(parents=True, exist_ok=True)

    #Go over each class and apply pre_processing:
    classes = os.listdir('./'+original_folder_name[k]+'/')
    for c in classes:
        counter = 0
        Path('./' + folder_name+'/' + c).mkdir(parents=True, exist_ok=True)
        image_names = os.listdir('./' + original_folder_name[k] + '/'+c +'/')
```

```
            if image_count is not None:
                if type(image_count) == type({}):
                    image_names = list(np.asarray(image_names)[image_indices_dict[c
                        BROKEN]])
                    images_per_class = image_count[c]
                else:
                    image_names = list(np.asarray(image_names)[image_indices])

            #Iterating over each image
            for i  in image_names:
                original_image = plt.imread('./' + original_folder_name[k] +'/' + c
                    BROKEN+ '/' + i)
                #Path to be saved in:
                save_path = './' + folder_name + '/' + c + '/'  + i
                if os.path.exists(save_path) == True:
                    save_path = './' + folder_name + '/' + c + '/' + 'copy_' + i

                #Grey scale for now.
                image_grey = np.asarray(PIL.Image.fromarray(original_image).convert(
                    BROKEN'L'))


                result = pre_process(image_grey, window_size=window_size[k],
                    BROKENborder_width=border_width[k],
                                         sub_sampling_step=sub_sampling_step[k],
                                         mode=mode[k], mask=mask[k], cut=cut[k],
                                             BROKENPIL_compression=PIL_compression[k],
                                         compression_den=compression_den[k], plot=plot[k
                                             BROKEN],
                                         save_folder_path=save_path, edge_detection=
                                             BROKENedge_detection[k])
                counter += 1
                print(folder_name,c,i,counter,'of',str(images_per_class) )

    return None


def generate_point_cloud(image_data,weights = (1,1,1),normalize = True):
    '''

    :param image_data: (M, N) array of image data of intensity values of any sort.
    :param weights: absolute factors in front of point cloud entries. E,g, (x,y,z)
        BROKEN-> (w_1 * x, w_2 * x, w_3 * x)
    :return: (M * N, 3) point cloud of entries (i, j, intensity), i-hor, j-vert. In
        BROKENlexicographical ordering.
    '''
    w_1, w_2, w_3 = weights
    desired_shape = image_data.shape
    #Generating grid in lex ordering
    j, i = np.mgrid[0:desired_shape[0], 0:desired_shape[1]]
    j = j.reshape((desired_shape[0] * desired_shape[1], 1))
    i = i.reshape((desired_shape[0] * desired_shape[1], 1))



    # Generating point cloud
    cloud = np.hstack((i * w_1, j * w_2))
    cloud = np.hstack((cloud, w_3 * (np.ravel(image_data).reshape((i.size, 1)))).
        BROKENastype(np.float64)))
    if normalize == True:
       # cloud[:,2] = cloud[:,2] - np.min(cloud[:,2])
        cloud[:,2] =  cloud[:,2] / np.max(cloud[:,2]) * w_3
    return cloud
```

```
def generate_HighD_point_cloud(train_X, train_Y, categories,instance_list = None):
    # print(f"Time to load data: {elapsed_time} seconds ")
    train_cloud = []
    train_index = []

    # {'NV':600, 'AKIEC':300, 'BKL':600, 'BCC':500, 'VASC':100, 'DF':100, 'MEL':600}
    # instance_list = {'NV':100, 'AKIEC':100, 'BKL':100, 'BCC':100, 'VASC':20, 'DF'
        BROKEN:20, 'MEL':100}
    if instance_list is None:
        instance_list = {'NV':20, 'AKIEC':20, 'BKL':20, 'BCC':20, 'VASC':20, 'DF'
            BROKEN:20, 'MEL':20}

    for ind, category in enumerate(categories):
        entity = train_X[train_Y == ind]
        # Generate High D point cloud:
        # instances_per_class = 5
        # images_per_class = entity.shape[0]
        # samples_per_instance = images_per_class // instances_per_class
        # shape = entity.shape
        samples_per_instance = instance_list[category]
        images_per_class = entity.shape[0]
        instances_per_class = images_per_class // samples_per_instance
        shape = entity.shape

        if images_per_class % instances_per_class != 0:
            print('Error, Choose equally divisable data set.')
            exit(0)
        entity_block = entity.reshape(shape[0], shape[1] * shape[2])
        # Reshape the entity:
        entity = entity.reshape((instances_per_class, samples_per_instance, shape[1]
            BROKEN * shape[2]))

        shape = entity.shape
        for k in range(shape[0]):
            train_cloud.append(entity[k].astype(np.float64) / np.max(entity[k]))
            #max_per_image = np.max(entity[k], axis=1)
            #max_per_image = max_per_image.reshape((max_per_image.size, 1)).astype(
                BROKENnp.float64)
            #train_cloud.append(entity[k].astype(np.float64)  / max_per_image)
            train_index.append(ind)


    return train_cloud, train_index


def generate_HighD_H0(train_cloud, cm = 'single'):
    #Generate H0 for a number of point clouds for one category
    train_dist = [sr.Distance(spatial.distance.pdist(HighD, "euclidean")) for HighD
        BROKENin train_cloud]
    avg_per_class_h0sr_list = []

    H0 = [d.get_h0sr(clustering_method= cm, reduced=True) for d in train_dist]
    return H0


#Note: This code was used to preliminarily try to classfiy based on the high
    BROKENdimensional embedding,
#However ultimately it was decided not to implement this section, as it would have
    BROKENrequired a more in depth
#approach, which would not have been possible in the time available. It is still
    BROKENincluded here for completeness.
def generate_average_substitution_score(candidate, comparison_point_cloud,comp_sr =
```

```
    BROKENNone,cm ='single',mode = 'avg'):
    if comp_sr is None:
        comp_sr = generate_HighD_H0([comparison_point_cloud],cm)[0]
    avg_norm_distance = []
    #plt.figure(100)
    avg_comp_cand = []

    for k in range(comparison_point_cloud.shape[0]):

        #comp_sr = generate_HighD_H0([np.delete(comparison_point_cloud,k,axis=0)],cm
            BROKEN)[0]
        #print(k)
        #comp_sr.plot(alpha = 0.2,c = 'r')
        temp_cloud = comparison_point_cloud.copy()
        temp_cloud[k] = candidate


        cand_sr = generate_HighD_H0([temp_cloud],cm)[0]
        avg_comp_cand.append(cand_sr)

        #cand_sr,_ = global_plot_stable_ranks_info(np.asarray(avg_comp_cand))
        #cand_sr.plot(alpha = 0.2, c ='k')
        upper_bound_val = np.maximum(np.max(cand_sr.content[1]), np.max(comp_sr.
            BROKENcontent[1]))
        lower_bound_val = np.minimum(np.min(cand_sr.content[1]), np.min(comp_sr.
            BROKENcontent[1]))
        upper_bound_loc = np.maximum(np.max(cand_sr.content[0]), np.max(comp_sr.
            BROKENcontent[0]))
        lower_bound_loc = np.minimum(np.min(cand_sr.content[0]), np.min(comp_sr.
            BROKENcontent[0]))
        norm_den = np.abs(upper_bound_val - lower_bound_val) * np.abs(
            BROKENupper_bound_loc - lower_bound_loc)
        norm_distance = cand_sr.lp_distance(comp_sr, p=1) / norm_den
        avg_norm_distance.append(norm_distance)

    #return norm_distance
    #plt.show()
    if mode == 'avg':
        score = np.average(np.asarray(avg_norm_distance))
    if mode == 'max':
        score = np.average(np.asarray(avg_norm_distance))
    if mode == 'min':
        score = np.min(np.asarray(avg_norm_distance))
    #print(mode, score)
    return score


def standard_deviation_stable_ranks(pcf_array):
    #Implemented like this, as this allows for the built in
    # functionality of the stablerank package to be used.
    average = np.sum(pcf_array) / pcf_array.size
    std = (np.sum(((pcf_array - average)**2)) / (pcf_array.size - 1))**(0.5)

    return std

def global_plot_stable_ranks_info(pcf_array,plot = False,show = True):
    '''

    :param pcf_array: Array of pcf objects
    :param plot: if True, plots the distribution of pcf around the mean. (2 standard
        BROKEN deviations)
    :param show: if True, directly shows the plot (flushes the matplotlib buffer.)
    :return: plot info, thus average and std
    '''
```

```
    average = np.sum(pcf_array) / pcf_array.size
    std = standard_deviation_stable_ranks(pcf_array)
    if plot == True:
        plt.figure('Distribution')
        average.plot(linestyle='-', color='w')
        upper = average + (std * 2)
        lower = average - (std * 2)
        upper.plot(linestyle='..', color='r')
        lower.plot(linestyle='..', color='r')
        for k in pcf_array:
            k.plot(linestyle='--', color='k', alpha=0.1)
        if show == True:
            plt.show()
    return average, std


def similarity_stable_ranks(pcf_array_1,pcf_array_2,mode = 'l1'):
    print('modes available: "interleaving" or "l1" ')

    avg_1, std_1 =  standard_deviation_stable_ranks(pcf_array_1)
    avg_2, std_2 =  standard_deviation_stable_ranks(pcf_array_2)

    if mode == 'lp':
        dist_avg =  avg_1.lp_distance(avg_2, p=1)
    if mode == 'interleaving':
        dist_avg = il_dist_avg = avg_1.interleaving_distance(avg_2)
    return dist_avg

def similarity_matrix(pcf_array,mode = 'l1',plot = False):
    A = np.zeros((pcf_array.size, pcf_array.size))
    for k in range(pcf_array.size):
        for l in range(pcf_array.size):
            if mode == 'l1':
                A[k,l] = pcf_array[k].lp_distance(pcf_array[l],p=1)

            if mode == 'interleaving':
                A[k, l] = pcf_array[k].interleaving_distance(pcf_array[l], p=1)

            #Normalise
            upper_bound_val = np.maximum(np.max(pcf_array[k].content[1]), np.max(
                BROKENpcf_array[l].content[1]))
            lower_bound_val = np.minimum(np.min(pcf_array[k].content[1]), np.min(
                BROKENpcf_array[l].content[1]))
            upper_bound_loc = np.maximum(np.max(pcf_array[k].content[0]), np.max(
                BROKENpcf_array[l].content[0]))
            lower_bound_loc = np.minimum(np.min(pcf_array[k].content[0]), np.min(
                BROKENpcf_array[l].content[0]))
            norm_den = np.abs(upper_bound_val - lower_bound_val) * np.abs(
                BROKENupper_bound_loc - lower_bound_loc)
            A[k, l] = 1 / norm_den * A[k,l]
    if plot == True:
        plt.matshow(A)
    return A



class ImageLoad(object):

    def __init__(self, PATH='', IMAGE_SIZE = 48, resize=False,enhance_contrast =
        BROKENFalse,
                 CONTRAST_FACTOR = 10.0 ):
        self.PATH = PATH
        self.IMAGE_WIDTH = IMAGE_SIZE
        self.IMAGE_HEIGHT = IMAGE_SIZE
```

```
        self.resize = resize
        self.enhance_contrast = enhance_contrast
        self.contrast_factor = CONTRAST_FACTOR
        self.image_data = []
        self.x_data = []
        self.y_data = []
        self.CATEGORIES = []

        self.list_categories = []

    def get_categories(self):
        for path in os.listdir(self.PATH):
            self.list_categories.append(path)
        print("Labels: ",self.list_categories,'\n')
        return self.list_categories

    def Process_Image(self):

            """:return: X_Data, Y_Data
            """
            self.CATEGORIES = self.get_categories()
            for categories in self.CATEGORIES:
                # Iterate over categories
                train_folder_path = os.path.join(self.PATH, categories)
                class_index = self.CATEGORIES.index(categories)

                for img in os.listdir(train_folder_path):
                    new_path = os.path.join(train_folder_path, img)

                    image_data_temp = Image.open(new_path)
                    if self.resize:
                        image = image_data_temp.resize((self.IMAGE_WIDTH, self.
                            BROKENIMAGE_HEIGHT))

                    else:
                        image = image_data_temp
                        self.IMAGE_WIDTH = image.size[0]
                        self.IMAGE_HEIGHT = image.size[1]

                    if self.enhance_contrast:
                        en = ImageEnhance.Contrast(image)
                        image = en.enhance(self.contrast_factor)

                    self.image_data.append([np.array(image), class_index])
            #random.shuffle(self.image_data)
            data = np.asarray(self.image_data)
            for x in data:
                self.x_data.append(x[0])        # Get X_Data
                self.y_data.append(x[1])        # Get classes
            X_Data = np.asarray(self.x_data)
            Y_Data = np.asarray(self.y_data)

            return X_Data, Y_Data
```

7.2 High Dimensional Embedding

`Main_Highd.py`

This code contains the main code for runnning the experiment of the high dimensional embedding.

```
import os, sys
from PIL import Image, ImageEnhance
import stablerank.srank as sr
```

```
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import scipy.spatial as spatial
import scipy.stats as st
import numpy as np
import random
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import matplotlib.image as mpimg
import topimgprocess.TDAimgprocess as TIP
import utilities as ut
import collections
from functools import reduce
import operator

#from Analysis_homolgy import all_in_class_h0sr as testh0, train_y as truth

from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle
from sklearn.metrics import plot_confusion_matrix, confusion_matrix


import time

import warnings
warnings.filterwarnings("ignore", category=np.VisibleDeprecationWarning)

parent_path = os.getcwd()

class ImageLoad(object):

    def __init__(self, PATH='', IMAGE_SIZE = 48, resize=False,enhance_contrast =
        BROKENFalse,
                 CONTRAST_FACTOR = 10.0 ):
        self.PATH = PATH
        self.IMAGE_WIDTH = IMAGE_SIZE
        self.IMAGE_HEIGHT = IMAGE_SIZE
        self.resize = resize
        self.enhance_contrast = enhance_contrast
        self.contrast_factor = CONTRAST_FACTOR
        self.image_data = []
        self.x_data = []
        self.y_data = []
        self.CATEGORIES = []
        self.image_names = []

        self.list_categories = []

    def get_categories(self):
        for path in os.listdir(self.PATH):
            self.list_categories.append(path)
        print("Labels: ",self.list_categories,'\n')
        return self.list_categories

    def Process_Image(self):

            """:return: X_Data, Y_Data
            """
            self.CATEGORIES = self.get_categories()
            for categories in self.CATEGORIES:
                # Iterate over categories
                train_folder_path = os.path.join(self.PATH, categories)
                class_index = self.CATEGORIES.index(categories)
```

```
                for img in os.listdir(train_folder_path):
                    new_path = os.path.join(train_folder_path, img)
                    self.image_names.append(img)

                    image_data_temp = Image.open(new_path)
                    if self.resize:
                        image = image_data_temp.resize((self.IMAGE_WIDTH, self.
                            BROKENIMAGE_HEIGHT))

                    else:
                        image = image_data_temp
                        self.IMAGE_WIDTH = image.size[0]
                        self.IMAGE_HEIGHT = image.size[1]

                    if self.enhance_contrast:
                        en = ImageEnhance.Contrast(image)
                        image = en.enhance(self.contrast_factor)

                    self.image_data.append([np.array(image), class_index])
            #random.shuffle(self.image_data)
            data = np.asarray(self.image_data)
            for x in data:
                self.x_data.append(x[0])        # Get X_Data
                self.y_data.append(x[1])        # Get classes
            X_Data = np.asarray(self.x_data)
            Y_Data = np.asarray(self.y_data)

            return X_Data, Y_Data

data_v1_path = os.path.join(os.getcwd(), 'Direct_no_mask_ext')


categories = os.listdir(data_v1_path)
start_time = time.time()
I = ImageLoad(PATH = data_v1_path, resize=True,
                            IMAGE_SIZE = 128, enhance_contrast = True,
                               BROKENCONTRAST_FACTOR= 100000.0)
train_X, train_y = I.Process_Image()



#train_X[train_X > 0.0] = 255
elapsed_time = time.time() - start_time
print(f"Time to load data: {elapsed_time} seconds " )
train_cloud = []
train_index = []

#Disable instance list comment for usagfe where desired. Designates the number of
    BROKENimages per point cloud for
#A given class.
#{'NV':600, 'AKIEC':300, 'BKL':600, 'BCC':500, 'VASC':100, 'DF':100, 'MEL':600}
instance_list = {'NV':150, 'AKIEC':150, 'BKL':150, 'BCC':250, 'VASC':50, 'DF':50, '
    BROKENMEL':150}
#instance_list = {'NV':100, 'AKIEC':100, 'BKL':100, 'BCC':100, 'VASC':25, 'DF':25, '
    BROKENMEL':100}
#instance_list = {'NV':20, 'AKIEC':20, 'BKL':20, 'BCC':20, 'VASC':20, 'DF':20, 'MEL'
    BROKEN:20}
#instance_list = {'NV':5, 'AKIEC':5, 'BKL':5, 'BCC':5, 'VASC':5, 'DF':5, 'MEL':5}
for ind, category in enumerate(categories):
    entity = train_X[train_y == ind]
    # Generate High D point cloud:
    #instances_per_class = 5
    #images_per_class = entity.shape[0]
    #samples_per_instance = images_per_class // instances_per_class
```

```
    #shape = entity.shape
    samples_per_instance = instance_list[category]
    images_per_class = entity.shape[0]
    instances_per_class = images_per_class // samples_per_instance
    shape = entity.shape


    if images_per_class % instances_per_class != 0:
        print('Error, Choose equally divisable data set.')
        exit(0)
    entity_block = entity.reshape(shape[0],shape[1] * shape[2])
    #Reshape the entity:
    entity = entity.reshape((instances_per_class,samples_per_instance,shape[1] *
        BROKENshape[2]))


    shape = entity.shape
    for k in range(shape[0]):
            train_cloud.append(entity[k].astype(np.float64) / np.max(entity[k]))
            #max_per_image = np.max(entity[k], axis=1)
            #max_per_image = max_per_image.reshape((max_per_image.size, 1)).astype(
                BROKENnp.float64)
            #train_cloud.append(entity[k].astype(np.float64)  / max_per_image)
            train_index.append(ind)




clustering_methods = ["single", "complete", "average", "ward"]

all_in_class_h0sr = collections.defaultdict(dict)
avg_per_class_h0sr = collections.defaultdict(dict)

train_dist = [sr.Distance(spatial.distance.pdist(HighD, "euclidean")) for HighD in
    BROKENtrain_cloud]
avg_per_class_h0sr_list = []
for cm in clustering_methods:
    for ind, category in enumerate(categories):
        all_in_class_h0sr[cm][category] = [d.get_h0sr(clustering_method=cm,reduced=
            BROKENTrue)  for d in train_dist if train_dist.index(d) in np.where(np.
            BROKENarray(train_index)==ind)[0]]


fig = plt.figure(figsize=(20, 20), constrained_layout=True)
axs = fig.subplots(2, 2)
fig_2 = plt.figure(figsize=(20, 20), constrained_layout=True)
axs_2 = fig_2.subplots(2, 2)



for i in range(axs.size):
    ax = axs[int(i / 2), int(i % 2)]
    ax_2 = axs_2[int(i / 2), int(i % 2)]
    temp_std= []
    temp_avg =[]
    for category in categories:
        if category in categories :#:['MEL', 'DF']:
        #if category in ['MEL', 'BKL','NV']:
            avg, std = ut.global_plot_stable_ranks_info(np.asarray(all_in_class_h0sr
                BROKEN[clustering_methods[i]][category]))
            #temp_std.append(np.average(std.content[1]))
```

```
            temp_avg.append(avg)
            avg.plot(ax = ax, label = category, ls = '-')
            a = avg.plot(ax=ax_2, label=category, ls='-')
            #(avg + (2 * std) ).plot(ax = ax_2, ls =':', c = a[0].get_c())
            #(avg - (2 * std) ).plot(ax = ax_2, ls =':', c = a[0].get_c())

            for k in np.asarray(all_in_class_h0sr[clustering_methods[i]][category]):
                k.plot(ax =ax, c=a[0].get_c(),alpha = 0.25)
        #avg_per_class_h0sr[clustering_methods[i]][category].plot(ax=ax, label=
            BROKENcategory)
    discrepancy_matrix = ut.similarity_matrix(np.asarray(temp_avg), mode='l1')
    print(clustering_methods[i], 'Discrepancy score, avg.:',np.average(
        BROKENdiscrepancy_matrix), 'max.:',np.max(discrepancy_matrix))
    #print(clustering_methods[i],'avg_std:',temp_std)

    ax.set_title(clustering_methods[i], fontsize=16)
    ax.legend()
    ax_2.set_title(clustering_methods[i], fontsize=16)
    ax_2.legend()
fig.suptitle('Average H0 stable ranks', fontsize=24)

plt.show()


#H1 calculations
start_time = time.time()
train_bc = [d.get_bc(maxdim=1) for d in  train_dist]
elapsed_time = time.time() - start_time
print(f"Time to calculate bar code: {elapsed_time} seconds " )
train_h1sr = [sr.bc_to_sr(bar_code, degree = "H1") for bar_code in train_bc]
elapsed_time = time.time() - start_time
print(f"Time to calculate H1 homologies: {elapsed_time} seconds" )



# Need to write a plotting function !!!
avg_per_class_h1sr = collections.defaultdict(dict)
plt.figure(figsize=(8,8), constrained_layout=True)
for ind, category in enumerate(categories):

    buff = np.asarray(train_h1sr)[np.where(np.array(train_index)==ind)[0]]
    avg, std = ut.global_plot_stable_ranks_info(buff)
    a = avg.plot(label=category, ls='-')
    (avg + (2 * std)).plot( ls=':', c=a[0].get_c())
    (avg - (2 * std)).plot( ls=':', c=a[0].get_c())

plt.title('Average H1 stable ranks', fontsize=16)
plt.legend()
plt.show()
```

7.3 3D Embedding

`Main_3D.py`

This is the code used for running the experiments based on the 3D pointcloud embedding.

```
import os, sys
from PIL import Image, ImageEnhance
import stablerank.srank as sr
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import scipy.spatial as spatial
```

```
import scipy.stats as st
import numpy as np
import random
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import matplotlib.image as mpimg
import topimgprocess.TDAimgprocess as TIP
import utilities as ut
import collections
from functools import reduce
import operator



from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle

import time

import warnings
warnings.filterwarnings("ignore", category=np.VisibleDeprecationWarning)

parent_path = os.getcwd()

class ImageLoad(object):

    def __init__(self, PATH='', IMAGE_SIZE = 48, resize=False,enhance_contrast =
        BROKENTrue,
                  CONTRAST_FACTOR = 10.0 ):
        self.PATH = PATH
        self.IMAGE_WIDTH = IMAGE_SIZE
        self.IMAGE_HEIGHT = IMAGE_SIZE
        self.resize = resize
        self.enhance_contrast = enhance_contrast
        self.contrast_factor = CONTRAST_FACTOR
        self.image_data = []
        self.x_data = []
        self.y_data = []
        self.CATEGORIES = []
        self.image_names = []
        self.list_categories = []

    def get_categories(self):
        for path in os.listdir(self.PATH):
            self.list_categories.append(path)
        print("Labels: ",self.list_categories,'\n')
        return self.list_categories

    def Process_Image(self):

            """:return: X_Data, Y_Data
            """
            self.CATEGORIES = self.get_categories()
            for categories in self.CATEGORIES:
                # Iterate over categories
                train_folder_path = os.path.join(self.PATH, categories)
                class_index = self.CATEGORIES.index(categories)

                for img in os.listdir(train_folder_path):
                    new_path = os.path.join(train_folder_path, img)
                    self.image_names.append(img)

                    image_data_temp = Image.open(new_path)
```

```
                    if self.resize:
                        image = image_data_temp.resize((self.IMAGE_WIDTH, self.
                            BROKENIMAGE_HEIGHT))

                    else:
                        image = image_data_temp
                        self.IMAGE_WIDTH = image.size[0]
                        self.IMAGE_HEIGHT = image.size[1]

                    if self.enhance_contrast:
                        en = ImageEnhance.Contrast(image)
                        image = en.enhance(self.contrast_factor)

                    self.image_data.append([np.array(image), class_index])
            #random.shuffle(self.image_data)
            data = np.asarray(self.image_data)
            for x in data:
                self.x_data.append(x[0])        # Get X_Data
                self.y_data.append(x[1])        # Get classes
            X_Data = np.asarray(self.x_data)
            Y_Data = np.asarray(self.y_data)

            return X_Data, Y_Data

data_v1_path = os.path.join(os.getcwd(), 'TDA_mask__25')

#data_v1_path = '../../Images/TDA_Processed_no_cut_full_ISIC2018_train_cd_2'

categories = os.listdir(data_v1_path)
start_time = time.time()
I = ImageLoad(PATH = data_v1_path, resize=True,
                           IMAGE_SIZE = 48, enhance_contrast=False,
                               BROKENCONTRAST_FACTOR= 4.0)
train_X, train_y = I.Process_Image()

elapsed_time = time.time() - start_time
print(f"Time to load data: {elapsed_time} seconds " )
train_cloud = []
for image in train_X:
    train_cloud.append(ut.generate_point_cloud(image,weights=(1,1,1),normalize=
        BROKENFalse))
# For test set (25 images per class)
markers = ['^', 'd', 'x', '+', '*', 'o', 's']
# img_num = 0
fig = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(projection='3d')
#for i in range(len(os.listdir(data_v1_path))):
#    img_num = np.where(train_y == i)[0][0]
#    ax.scatter(train_cloud[img_num][:, 0], train_cloud[img_num][:, 1],
#               train_cloud[img_num][:, 2], marker=markers[i], label=os.listdir(
    BROKENdata_v1_path)[i])

specific_image_index = I.image_names.index('ISIC_0024575.jpg')
ax.scatter(train_cloud[specific_image_index][:, 0], train_cloud[specific_image_index
    BROKEN][:, 1],
               train_cloud[specific_image_index][:, 2], marker=markers[1], label=os.
                   BROKENlistdir(data_v1_path)[1])


#ax.set_xlabel('x-coordinate')
ax.set_zlim(255,0)
#ax.set_ylabel('y-coordinate')
#ax.set_zlabel('intensity')
ax.set_title('3D point clouds')
```

```
ax.legend()
plt.show()
fig.savefig('PointCloudsReport.jpg')



clustering_methods = ["single", "complete", "average", "ward"]


all_in_class_h0sr = collections.defaultdict(dict)
avg_per_calss_h0sr = collections.defaultdict(dict)

train_dist = [sr.Distance(spatial.distance.pdist(fig, "euclidean")) for fig in
    BROKENtrain_cloud]




for cm in clustering_methods:
    for ind, category in enumerate(categories):
        all_in_class_h0sr[cm][category] = [d.get_h0sr(clustering_method=cm)  for d
            BROKENin train_dist if train_dist.index(d) in np.where(train_y==ind)[0]]
        avg_per_calss_h0sr[cm][category] = sum(all_in_class_h0sr[cm][category]) /
            BROKENlen(all_in_class_h0sr[cm][category])
train_h0sr = {}


for cm in clustering_methods:
    train_h0sr[cm] = reduce(operator.concat, list(all_in_class_h0sr[cm].values()))

fig = plt.figure(figsize=(20, 20), constrained_layout=True)
axs = fig.subplots(2, 2)

for i in range(axs.size):
    temp_avg = []
    ax = axs[int(i / 2), int(i % 2)]
    for category in categories:
        avg, std = ut.global_plot_stable_ranks_info(np.asarray(all_in_class_h0sr[
            BROKENclustering_methods[i]][category]))
        temp_avg.append(avg)
        a = avg.plot(ax = ax, label = category, ls = '-')
        (avg + (2 * std) ).plot(ax = ax, ls =':', c = a[0].get_c())
        (avg - (2 * std) ).plot(ax = ax, ls =':', c = a[0].get_c())
        #for k in np.asarray(all_in_class_h0sr[clustering_methods[i]][category]):
        #    k.plot(ax =ax, c='k',alpha = 0.05)
        #avg_per_calss_h0sr[clustering_methods[i]][category].plot(ax=ax, label=
            BROKENcategory)
        discrepancy_matrix = ut.similarity_matrix(np.asarray(temp_avg), mode='l1')
    print(clustering_methods[i], 'Discrepancy score, avg.:',np.average(
        BROKENdiscrepancy_matrix), 'max.:',np.max(discrepancy_matrix))
    ax.set_title(clustering_methods[i], fontsize=16)
    ax.legend()
fig.suptitle('Average H0 stable ranks', fontsize=24)

plt.show()

#SVM for H0

def kernel_from_sr(train_h_sr, clustering_methods):
    h_kernel_train ={}
    for cm in clustering_methods:
        h_kernel_train[cm] = np.asarray([[f.dot(g) for g in train_h_sr[cm]] for f in
            BROKEN train_h_sr[cm]])
    return h_kernel_train
```

```
h0_kernel_train = kernel_from_sr(train_h0sr, clustering_methods)

from sklearn.model_selection import cross_val_score
prediction_h0= {}
cv_score_h0 = {}
for cm in clustering_methods:
    clf_h0 = svm.SVC(kernel='precomputed')
    clf_h0.fit(h0_kernel_train[cm], train_y)
    prediction_h0[cm] = clf_h0.predict(h0_kernel_train[cm])
    cv_score_h0[cm] = cross_val_score(clf_h0, h0_kernel_train[cm], train_y, cv=5)

print("Cross-validation scores")
for cm in clustering_methods:
    print(f"{cm}: {cv_score_h0[cm].mean():.3f}")




#H1 calculations
start_time = time.time()
train_bc = [d.get_bc(maxdim=1) for d in  train_dist]
elapsed_time = time.time() - start_time
print(f"Time to calculate bar code: {elapsed_time} seconds " )
train_h1sr = [sr.bc_to_sr(bar_code, degree = "H1") for bar_code in train_bc]
elapsed_time = time.time() - start_time
print(f"Time to calculate H1 homologies: {elapsed_time} seconds" )


# Need to write a plotting function !!!
# Need to write a plotting function !!!

avg_per_class_h1sr = collections.defaultdict(dict)
fig = plt.figure(figsize=(8,8), constrained_layout=True)
for ind, category in enumerate(categories):

    buff = np.asarray(train_h1sr)[np.where(train_y==ind)[0]]
    avg, std = ut.global_plot_stable_ranks_info(buff)
    a = avg.plot( label=category, ls='-')
    (avg + (2 * std)).plot(ls=':', c =  a[0].get_c())
    (avg - (2 * std)).plot(ls=':', c =  a[0].get_c())

    #avg_per_class_h1sr[category] = sum(buff) / len(buff)
    #avg_per_class_h1sr[category].plot(label = category)
plt.title('Average H1 stable ranks', fontsize=16)
plt.legend()

fig.savefig("h1sr")
#SVM for H1



def kernel_from_h1sr(train_h_sr):
    h_kernel_train = np.asarray([[f.dot(g) for g in train_h_sr] for f in train_h_sr
        BROKEN])
    return h_kernel_train

h1_kernel_train = kernel_from_h1sr(train_h1sr)

clf_h1 = svm.SVC(kernel='precomputed')
clf_h1.fit(h1_kernel_train, train_y)
prediction_h1 = clf_h1.predict(h1_kernel_train)
cv_score_h1 = cross_val_score(clf_h1, h1_kernel_train, train_y, cv=5)
```

```
print("Cross-validation scores")
print(f"{cv_score_h1.mean():.3f}")

clf_h1 = svm.SVC(kernel='precomputed')
clf_h1.fit(h1_kernel_train + h0_kernel_train['single'], train_y)
prediction_h1 = clf_h1.predict(h1_kernel_train + h0_kernel_train['single'])
cv_score_h1 = cross_val_score(clf_h1, h1_kernel_train + h0_kernel_train['single'],
    BROKENtrain_y, cv=5)

print("Cross-validation scores")
print(f"{cv_score_h1.mean():.3f}")
plt.show()

clf_h1 = svm.SVC(kernel='precomputed')
clf_h1.fit(h1_kernel_train * h0_kernel_train['single'], train_y)
prediction_h1 = clf_h1.predict(h1_kernel_train * h0_kernel_train['single'])
cv_score_h1 = cross_val_score(clf_h1, h1_kernel_train * h0_kernel_train['single'],
    BROKENtrain_y, cv=5)

print("Cross-validation scores")
print(f"{cv_score_h1.mean():.3f}")
plt.show()
```

7.4 2D Embedding

main_2D.ipynb

This is the code used for running the experiments based on the 2D pointcloud embedding.

```
import os, sys
from PIL import Image

import stablerank.srank as sr
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import scipy.spatial as spatial
import scipy.stats as st


import numpy as np
import random
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import matplotlib.image as mpimg
get_ipython().run_line_magic('matplotlib', 'inline')
import topimgprocess.TDAimgprocess as TIP
import utilities as ut

import collections
from functools import reduce
import operator


from sklearn.preprocessing import normalize


from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle

import time

import warnings
```

```
warnings.filterwarnings("ignore", category=np.VisibleDeprecationWarning)


# In[17]:


class ImageLoad(object):

    """
    Class for loading data from folders

    """

    def __init__(self, PATH='', IMAGE_SIZE = 48, resize=False,enhance_contrast =
        BROKENFalse,
                 CONTRAST_FACTOR = 2.0 ):
        self.PATH = PATH
        self.IMAGE_WIDTH = IMAGE_SIZE
        self.IMAGE_HEIGHT = IMAGE_SIZE
        self.resize = resize
        self.enhance_contrast = enhance_contrast
        self.contrast_factor = CONTRAST_FACTOR
        self.image_data = []
        self.x_data = []
        self.y_data = []
        self.CATEGORIES = []

        self.list_categories = []

    def get_categories(self):
        for path in os.listdir(self.PATH):
            self.list_categories.append(path)
        #print("Labels: ",self.list_categories,'\n')
        return self.list_categories

    def Process_Image(self):

            """:return: X_Data, Y_Data
            """
            self.CATEGORIES = self.get_categories()
            for categories in self.CATEGORIES:
                # Iterate over categories
                train_folder_path = os.path.join(self.PATH, categories)
                class_index = self.CATEGORIES.index(categories)

                for img in os.listdir(train_folder_path):
                    new_path = os.path.join(train_folder_path, img)

                    image_data_temp = Image.open(new_path)
                    if self.resize:
                        image = image_data_temp.resize((self.IMAGE_WIDTH, self.
                            BROKENIMAGE_HEIGHT))

                    else:
                        image = image_data_temp
                        self.IMAGE_WIDTH = image.size[0]
                        self.IMAGE_HEIGHT = image.size[1]

                    if self.enhance_contrast:
                        en = ImageEnhance.Contrast(image)
                        image = en.enhance(self.contrast_factor)

                    self.image_data.append([np.array(image), class_index])
```

```
            #random.shuffle(self.image_data)
            data = np.asarray(self.image_data)
            for x in data:
                self.x_data.append(x[0])        # Get X_Data
                self.y_data.append(x[1])        # Get classes
            X_Data = np.asarray(self.x_data)
            Y_Data = np.asarray(self.y_data)

            return X_Data, Y_Data


# pre-selection testing: all 6 preprocessing methods for 2D embedding
# data was reduced to 25 images per class

np.random.seed(1)

data_prep_names = ['Direct_mask__25', 'Direct_no_mask__25', 'Edge_direct_mask__25',
    BROKEN'Edge_TDA_mask__25', 'TDA_mask__25',
                    'TDA_no_mask__25']

start_time = time.time()

train_X = {}


# data load: all keys are disabled
for ind, preprocess in enumerate(data_prep_names):
    data_v1_path = os.path.join(os.getcwd(), preprocess)
    categories = os.listdir(data_v1_path)
    if ind == 0:
        buff_X, buff_y = ImageLoad(PATH = data_v1_path, resize=False,
                            enhance_contrast=False).Process_Image()
        train_y = buff_y
    else:
        buff_X, _ = ImageLoad(PATH = data_v1_path, resize=False,
                            enhance_contrast=False).Process_Image()
    train_X[preprocess] = buff_X
elapsed_time = time.time() - start_time
print(f"Time to load data: {elapsed_time} seconds " )


# construction of 2D point clouds
prepr_dict_clouds = {}
for preprocess in data_prep_names:
    train_cloud2D = []
    for image in train_X[preprocess]:
        image_norm = normalize(np.array(image), axis=1, norm='l1')
        x, y  = np.where(image_norm > np.random.rand(image.shape[0], image.shape[1])
            BROKEN)
        train_cloud2D.append(np.hstack((x.reshape(-1,1),y.reshape(-1,1))))
    prepr_dict_clouds[preprocess] = train_cloud2D


# distance matrix calculation
train_dist_preprocess = {}
for preprocess in data_prep_names:
    train_dist_preprocess[preprocess] = [sr.Distance(spatial.distance.pdist(fig, "
        BROKENeuclidean")) for fig in prepr_dict_clouds[preprocess]]



# H0 stable ranks calculations
clustering_methods = ["single", "complete", "average", "ward"]
preprocess_stat_list = []
```

```
for preprocess in data_prep_names:
    all_in_class_h0sr = collections.defaultdict(dict)
    avg_per_class_h0sr = collections.defaultdict(dict)
    preprocess_stat_dict = {}
    train_dist = train_dist_preprocess[preprocess]
    for cm in clustering_methods:
        preprocess_stat_dict[cm] = []
        for ind, category in enumerate(categories):
            all_in_class_h0sr[cm][category] = [d.get_h0sr(clustering_method=cm)  for
                BROKEN d in train_dist if train_dist.index(d) in np.where(train_y==
                BROKENind)[0]]
            avg, _ = ut.global_plot_stable_ranks_info(np.asarray(all_in_class_h0sr[
                BROKENcm][category]))
            preprocess_stat_dict[cm].append(avg)
    preprocess_stat_list.append(preprocess_stat_dict)



# save 3 aggregation statistics: min, average, and max across classes
stat_dict = collections.defaultdict(dict)
for ind, preprocess in enumerate(data_prep_names):
    for cm in clustering_methods:
        stats = []
        results = np.array(ut.similarity_matrix(np.array(preprocess_stat_list[ind][
            BROKENcm])))
        stats.append(np.partition(results.flatten(), 7)[7])
        stats.append(np.mean(results))
        stats.append(np.max(np.max(results)))
        stat_dict[preprocess][cm]  = stats

a = [] # list for storing dissimilarity averages across classes
for preprocess in data_prep_names:
    for cm in clustering_methods:
        a.append(stat_dict[preprocess][cm][1])




# Main experiment


# 100 images per class
preprocess = 'edge_direct_mask_100'
data_v1_path = os.path.join(os.getcwd(), preprocess)
categories = os.listdir(data_v1_path)
train_X, train_y = ImageLoad(PATH = data_v1_path, resize=False,
                             enhance_contrast=False).Process_Image()



# Visualisation of 2D point cloud corresponding to the 1st image from AKIEC class
markers = ['^', 'd', 'x', '+', '*', 'o', 's']
img_num = 0
train_cloud2D = []
image =  train_X[img_num]

image_norm = normalize(np.array(image), axis=1, norm='l1')
x, y  = np.where(image_norm > np.random.rand(image.shape[0], image.shape[1]))
plt.figure(figsize=(10, 10))
plt.scatter(x, y, marker = markers[0], label = categories[0], color = 'coral', s=35)

plt.xlabel('x-coordinate')
plt.ylabel('y-coordinate')
plt.title('2D point cloud')
```

```
plt.legend()
plt.savefig('PointClouds2DReport.jpg')



# 2D point clouds construction
train_cloud2D = []
for image in train_X:
    image_norm = normalize(np.array(image), axis=1, norm='l1')
    x, y  = np.where(image_norm > np.random.rand(image.shape[0], image.shape[1]))
    train_cloud2D.append(np.hstack((x.reshape(-1,1),y.reshape(-1,1))))
train_dist =  [sr.Distance(spatial.distance.pdist(fig, "euclidean")) for fig in
    BROKENtrain_X]



clustering_methods = ["single", "complete", "average", "ward"]
all_in_class_h0sr = collections.defaultdict(dict)
avg_per_class_h0sr = collections.defaultdict(dict)
for cm in clustering_methods:
    for ind, category in enumerate(categories):
        all_in_class_h0sr[cm][category] = [d.get_h0sr(clustering_method=cm)  for d
            BROKENin train_dist if train_dist.index(d) in np.where(train_y==ind)[0]]


train_h0sr = {}
for cm in clustering_methods:
    train_h0sr[cm] = reduce(operator.concat, list(all_in_class_h0sr[cm].values()))



fig = plt.figure(figsize=(20, 20), constrained_layout=True)
axs = fig.subplots(2, 2)
fig_2 = plt.figure(figsize=(20, 20), constrained_layout=True)
axs_2 = fig_2.subplots(2, 2)
for i in range(axs.size):
    ax = axs[int(i / 2), int(i % 2)]
    ax_2 = axs_2[int(i / 2), int(i % 2)]
    for category in categories:
        avg, std = ut.global_plot_stable_ranks_info(np.asarray(all_in_class_h0sr[
            BROKENclustering_methods[i]][category]))
        avg.plot(ax = ax, label = category, ls = '-')
        a = avg.plot(ax=ax_2, label=category, ls='-')
        (avg + (2 * std) ).plot(ax = ax_2, ls =':', c = a[0].get_c())
        (avg - (2 * std) ).plot(ax = ax_2, ls =':', c = a[0].get_c())
    ax.set_title(clustering_methods[i], fontsize=16)
    ax.legend()
    ax_2.set_title(clustering_methods[i], fontsize=16)
    ax_2.legend()
fig_2.suptitle('Average H0 stable ranks', fontsize=24)
fig_2.save('2DAverageH0StableRanks.jpg')

fig = plt.figure(figsize=(30,30), constrained_layout=True)
axs = fig.subplots(2, 2)
for i in range(axs.size):
    ax = axs[int(i/2), int(i%2)]
    for category in categories:
        avg, std = ut.global_plot_stable_ranks_info(np.asarray(all_in_class_h0sr[
            BROKENclustering_methods[i]][category]))
        avg.plot(ax = ax, label = category, ls = '-')

    ax.set_title(clustering_methods[i], fontsize=20)
    ax.legend()
fig.suptitle('Average H0 stable ranks', fontsize=30)
```

```
start_time = time.time()
train_bc = [d.get_bc(maxdim=1) for d in  train_dist]
elapsed_time = time.time() - start_time
print(f"Time to calculate bar code: {elapsed_time} seconds " )
train_h1sr = [sr.bc_to_sr(bar_code, degree = "H1") for bar_code in train_bc]
elapsed_time = time.time() - start_time
print(f"Time to calculate H1 homologies: {elapsed_time} seconds")


avg_per_class_h1sr = collections.defaultdict(dict)
fig = plt.figure(figsize=(8,8), constrained_layout=True)
for ind, category in enumerate(categories):
    buff = np.array(train_h1sr)[np.where(train_y==ind)[0]]
    avg_per_class_h1sr[category] = sum(buff) / len(buff)
    avg_per_class_h1sr[category].plot(label = category)
plt.title('Average H1 stable ranks', fontsize=16)
fig.savefig('2DAverageH1StableRanks.jpg')
plt.legend()


# H0 and H1 homology bar codes
img_num = 0
for i in range(len(categories)):
    plt.figure(figsize = (20,10), constrained_layout=True)
    ax=plt.subplot(1,2,1)
    ax.set_title("H0")
    train_bc[img_num]['H0'].plot()
    ax=plt.subplot(1,2,2)
    ax.set_title("H1")
    train_bc[img_num]['H1'].plot()
    plt.suptitle(categories[i])
    img_num += 25



def kernel_from_sr(train_h_sr, clustering_methods):
    h_kernel_train ={}
    for cm in clustering_methods:
        h_kernel_train[cm] = np.asarray([[f.dot(g) for g in train_h_sr[cm]] for f in
            BROKEN train_h_sr[cm]])
    return h_kernel_train

h0_kernel_train = kernel_from_sr(train_h0sr, clustering_methods)


h1_kernel_train = np.asarray([[f.dot(g) for g in train_h1sr] for f in train_h1sr])
# H0 homology kernel
prediction_h0= {}
cv_score_h0 = {}
confusion_matrix = {}
for cm in clustering_methods:
    clf_h0 = svm.SVC(kernel='precomputed')
    clf_h0.fit(h0_kernel_train[cm], train_y)
    prediction_h0[cm] = clf_h0.predict(h0_kernel_train[cm])
    confusion_matrix[cm] = multilabel_confusion_matrix(train_y, prediction_h0[cm])
    cv_score_h0[cm] = cross_val_score(clf_h0, h0_kernel_train[cm], train_y, cv=5)
print("Cross-validation scores")
for cm in clustering_methods:
    print(f"{cm}: {cv_score_h0[cm].mean():.3f}")
# H1 homology kernel
clf = svm.SVC(kernel='precomputed')
clf.fit(h1_kernel_train, train_y)
prediction_h1 = clf.predict(h1_kernel_train)
```

```
cv_score_h1 = cross_val_score(clf, h1_kernel_train, train_y, cv=5)
cv_score_h1.mean()
# sum of H0 and H1 homology kernels
clf = svm.SVC(kernel='precomputed')
kernel_mixed = h1_kernel_train + h0_kernel_train['ward']
clf.fit(kernel_mixed,train_y)
prediction_mixed = clf.predict(kernel_mixed)
cv_score_mixed = cross_val_score(clf, kernel_mixed, train_y, cv=5)
cv_score_mixed.mean()
# element-wise product of H0 and H1 homology kernels
clf = svm.SVC(kernel='precomputed')
kernel_mixed = np.multiply(h1_kernel_train, h0_kernel_train['ward'])
clf.fit(kernel_mixed , train_y)
prediction_mixed = clf.predict(kernel_mixed)
cv_score_mixed = cross_val_score(clf, kernel_mixed, train_y, cv=5)
cv_score_mixed.mean()
```