Universitat
de les Illes Balears

FINAL DEGREE REPORT

AN INTRODUCTION TO TOPOLOGICAL DATA ANALYSIS AND ITS COMPUTATIONAL APPLICATIONS

Bernat Jordà Carbonell

Grau de Matematiques

Escola Politècnica Superior

Any acadèmic 2022-23

AN INTRODUCTION TO TOPOLOGICAL DATA ANALYSIS AND ITS COMPUTATIONAL APPLICATIONS

Bernat Jordà Carbonell

Final Degree Report

Escola Politècnica Superior

Universitat de les Illes Balears

Any acadèmic 2022-23

Paraules clau del treball: Topological Data Analysis, Persistent Homology, Mapper,Algebraic Topology, Data

Tutor: Gabriel Riera Roca

Moltíssimes gràcies a tots aquells que m'han ajudat a fer aquest camí: A en Delfí, qui hem va ensenyar que m'agradaven les matemàtiques, a tots els professors que m'han format durant aquesta etapa, i a tots aquells que m'han donat suport en la meva vida privada.

Journey before destination.

CONTENTS

ACRÒNIMS

TDA Topological Data Analysis

VR Vietoris-Rips

ABSTRACT

Topological Data Analysis (TDA) is a new and promising field that uses advanced concepts in algebraic topology to extract topological and geometric information about data sets. It is especially useful on really high-dimensional datasets where classic data analysis tool fail to recover all the meaningful information.

The two main methods in this new branch are *Mapper* and *Persistence Homology*. We will introduce both of them, starting with an introduction into the theoretical framework of algebraic topology that makes the methods possible and ensures they are trustworthy, explaining their properties. Then we will describe how those properties are used in *persistence homology* and the general uses that it has.

Finally we will review the state of the art and how different packages use and implement the methods just before explaining the *Mapper* algorithm, and showcase some practical computational examples, ending with a well-known application of *Mapper* to a cancer dataset that resulted in the discovery of a new type of cancer.

CHAPTER 1

INTRODUCTION

Modern Data Analysis has been changing a lot in recent years mostly due to one main factor: The dramatic increase in available data. This whole situation is the result of a sum of technical advances and will only become more extreme as the years pass by. Developing new efficient data processing methods is becoming a necessity as the more used once start to become obsolete. This task focuses around making sense to the flood of data we are suffering.

The main approaches to solve this problem are clustering algorithms and dimensionality reduction procedures. All of these techniques, which aim to describe the shape of the data, are nowadays omnipresent. Although these methods are usually still very useful, they only provide crude descriptions of data's shape. It's clear that, as powerful as these tools are, they are also evidently limited. Furthermore, data nowadays not only has a challenge in the size by itself, its complexity is another problem. This includes high-dimensionality, being incomplete, and having a lot of noise.

Trying to create new tools that are give us information in the more complex and difficult cases where the old ones fall short, some authors have been approaching data analysis in a more geometrically way. One of those approaches is TDA: Topological Data Analysis, the main subject of this thesis.

TDA tries to apply topology techniques to extract topological information from the data. These techniques create a general framework that provides dimensionality reduction and robustness to noise in a manner that is independent of the particular metric. All of this makes it a specially useful approach in those high-dimensional, noisy and complex data sets that are so troublesome for classic tools. This is due to its topological nature: TDA combines algebraic topology and other tools from pure mathematics for a more rigorous study of shape.

One of the two most prominent TDA algorithms to date is *Persistence Homology*. This algorithm focuses on finding the topological features of the manifold that models

the population just from the discrete subset of data that is a sample. On the other hand, the other prominent algorithm in TDA is *Mapper*, that centers around how to represent the data structure being topologically and geometrically accurate.

These two algorithms share a really similar theoretical background: Both of them use simplical complexes as the back bone of their operation among other similarities. Consequently, their theoretical background is quite similar and overlaps a lot. Moreover, the two algorithms complement each other greatly: even if they start with really similar techniques, on search for raw quantitative topological information that is usually hard to visualize while the other is wholly centered around qualitative information and visualization. As a result of all this, it is generally a good idea to work with both algorithms and most TDA packages implement both algorithms together.

In line with this general trend, these two algorithms will be the general object of study in this project. We will start in Chapter 2 properly explaining simplicial complexes and chain groups, their topology features and how to work with them. We will follow with Chapter 3, explaining how we will create complexes from discrete data and how we will use a multitude of simplical complexes and chains generated from the same sample of discrete data to find the persistence of its topological features. Moreover, we will end the chapter showing how to represent and interpret the result of *Persistence Homology*. We will end up with Chapter 4, explaining the different implementations of the algorithm in different packages, talking about the state of the art and illustrating some examples. It will be in that same chapter where we will glide over the *Mapper* algorithm, explaining it and showing it in action.

1.1 Persistent homology

One of the main selling points of TDA and the first of the two algorithms that we will discuss is *persistent homology*. The mathematical concept behind the algorithm is found in the fact that it is possible to find topological invariants of a geometric object barely from a discrete sample (Niyogi-Smale-Weinberger theorem [1]). From this notion come into being the idea of finding some topological knowledge behind the geometric structures that data sets sample. Since holes are the main invariant in algebraic topology, the first method that comes to mind is to try find "holes" in a data set structure.

Analogously to how in classical statistics we infer information about the population by examining a sample, persistence homology extracts information about the topology of the manifold that generated a discrete set of points. Obviously, the manifold that generates a sample is the same manifold that models the population the sample comes from.

However, just like in classical inference, the topological information extracted comes bounded to a certain likelihood. To cope with that, instead of extracting the general topological information of the sample using a specific scale, we examine the topological features at different scales and see in how many scales each feature appears. Clearly, the features that are present at the biggest range of scales are the most relevant.

The main problem is that achieving knowledge about the different topological

features and their respective persistences through different scales in a discrete data set is not trivial, at all. First we get the value of the distance between each pair of points of the set. Afterwards, for one of those values, we generate a graph-like structure (a simplicial complex) with a node for each point and a vertex between each pair of nodes at a distance less than the given value. Finally we will find the topological features of the simplicial complex generated using techniques seen in chapter 2.

We will use homology groups to extract the topological information of the complexes, but we will have to generate a complex and repeat the rest of the procedure for every single distance value extracted. Then, once we have all those complexes and their respective topological information, we will link the topological features that are the same across different complexes. Since all the complexes are generated from the same set of points, it is trivial to link the single elements between different complexes through the original point that generated them. Consequently, finding the connection between two topological features in different complexes is not hard when you know that those features were generated by the same elements.

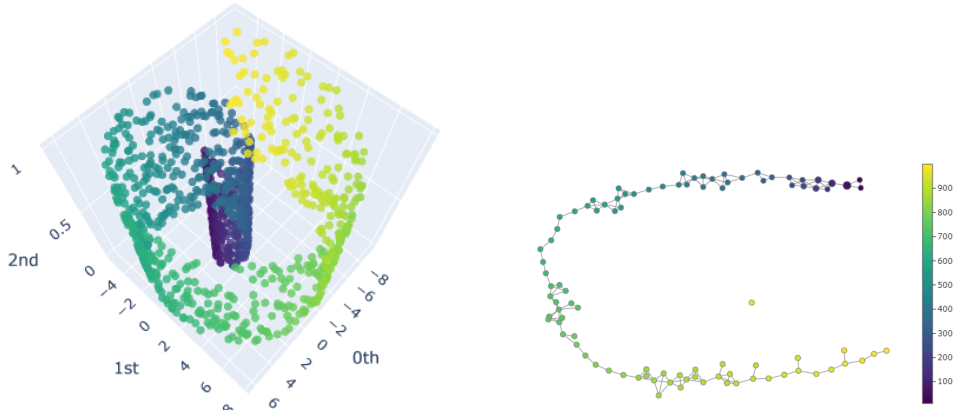
All of this, and the theoretical framework that makes it all work will be properly explained in chapter 2. In that chapter the more obvious problems will be solved and we will see that there are non-evident problems that arise from some of the other solutions. We will put examples as we go through the different stages for a better comprehension and once everything is clearly explained we will end up with some computation examples to see how different TDA packages treat Persistent Homology.

1.2 Mapper

Persistence Homology is a powerful technique that lets us extract quantitative topological features of the space of the data, but usually it is quite hard to interpret the information that provides us. On the contrary, the other method that we will study is *Mapper* and was developed with the purpose of obtaining simplification, visualization and qualitative analysis of high dimensional data [2] . As usual in Data analysis, the combination of qualitative analysis to better understand the evidence and quantitative data to prove or disprove the resulting hypothesis is a great pair.

The idea behind *Mapper* consists in, roughly speaking, applying clustering to sections of the data to put together groups of points into just one node and connect the clusters from overlaying sections if they share any points. With that it converts a discrete set of data into a simplicial complex. Going into a little bit more details, the algorithm consists of 5 steps:

- Applying a function to all the objects in the dataset to map them to a lower-dimensional space.
- Construct an overlapping cover of the image of the function.
- Applying a clustering algorithm to the preimage of each region of the cover.



(a) Point cloud (b) *Mapper*

Figure 1.1: As you can see, the Mapper algorithm preserves the topological structure of the spiral being just a curved plane

- Create a graph with a node for each cluster generated in all intervals and and edge between each two intervals that share at least on point (this is why it is important to have an overlapping cover).
- Plot the resulting graph to visualize.

One of the most attractive features of *Mapper* is its modularity and flexibility. We can change each step individually and we can use any function, cover and/or clustering algorithm we want. This permits reducing the dimension of the sample through the function chosen and picking the most appropriate cover and clustering algorithm depending on the particular data, permitting a really customized analysis.

But the most important property of the algorithm is that the algorithm is great at preserving topological and geometric information at specific resolution. The resolution is mostly dependent of the cover chosen. As an example to understand what *Mapper* can do, we apply the algorithm with the most basic options to a dataset that has a three-dimensional spiral shape in Figure 1.1. We will further look into this in Chapter 4.

CHAPTER 2

SIMPLICIAL HOMOLOGY

The Persistence homology and Mapper algorithms work in a way that if directly presented might not be intuitive. Therefore, to fully understand these algorithms and the mathematical reason of why do they work we need a theoretical framework. This is why the pure theoretic idea with functors and categories instead of matrices and numerical operations will be presented first. Gliding over the more important concepts and the theories that make the tools presented add up. In this chapter we are going to review the theoretical foundations needed to understand the core features of the work. It is meant as a refresh for the experienced reader but it also works as a brief introduction of concepts for the novice reader into Algebraic Topology.

This chapter starts with an outline of the algorithm before going into the definitions and applications of simplicial objects. It continues with the definition of chain objects, more precisely chain complexes. Then it proceeds with the homology of chain complexes to tie all the concepts together in a formal and practical way that let us accomplish the main objective of TDA: find geometric invariants, more precisely, holes.

The homology of chain complexes will lead us to homology groups or vector spaces, which are the usual way in algebraic topology to identify holes. We will better understand the different loops in a surface and how they behave under continuous deformations or homotopy. With that we will find a way of knowing the number of n-dimensional holes in the surface (and what they really are). Finally, we will end the chapter proving some fundamental properties of the just mentioned objects, most importantly, the functorality of homology.

With all of this, an understanding of the mathematical background that proves the output information to be relevant and truthful is properly achieved. Once the geometric and topological intuition of the algorithm is settled, we will then explain how each part of the process is implemented and used in applications.

It is worth mentioning that, while not utterly necessary, it might prove useful to have

some basic notions of category theory, mostly of the functor and category concepts as well as the manifold concept from topology. We will define and explain other related concepts but not in detail: most proofs will be left in the references or as common literature. Otherwise, the cohesion of the project would suffer and we wouldn't be able to focus on our main topic. We also expect an advanced understanding of abstract algebra and topology: Topological spaces, homomorphisms and so on. We will use those and other concepts usually without providing a definition or explanation. Even then, we will give some specific algebraic topology background. Otherwise, the comprehension and study of the text would not be friendly to the reader.

For some readers, knowing the purpose of the introduced concepts might result in a easier understanding of those concepts. Consequently, we will provide a basic, rough, simplified outline of TDA. Hopefully this will improve the assimilation of the concepts and the use of them later on.

Persitence homology outline

To properly grasp the geometric aspect of Persistence Homology we have to think of our data as a sample of a manifold in an Euclidean space. The raw data that we work with is then a set of points in a metric space with a certain distance within each two points. The process is as follows:

1. First the data is cleaned and processed, and then we choose a proper distance function between the elements of our data. This is important because we might work with genomics, time series or even images, where the distance between two given samples is not trivial.

2. Then, we extract information about the distance between our points. If our data are point clouds or time series we generate a distance matrix and if we are working with a graph a weighted adjacency matrix.

3. Once we have the matrix we choose some arbitrary value ϵ and then we mark each pair of points closer than ϵ from each other. We represent that information with a simplicial complex: Each point from our data is a 0-simplex in the complex, and if two points x, y are at distance $d(x, y) \leq \epsilon$ then the 1-simplex generated by those two points is also part of the complex.

4. If multiple points are joint pairwise we mark the subset conformed by those points, adding the simplex they generate to the complex. If three nodes are joint, they form a 2-simplex, four a 3-simplex and so on. Note that a 2-simplex is a triangle and 3-simplex is a tetrahedron.

5. After all simplices are added, we end up with a complex and we start to look at the different closed paths in the complex. We then look for closed paths that contain a hole: As in classic topology, think about the complex as a topological space, and the closed path as a topological object that can get stretched and twisted. If we can't transform our closed path into a point through those transformations, it means that the path contains a hole.

6. We do that not only with 1 dimensional paths but also with surfaces (composed of 2-simplices) and higher dimensional equivalents. Later on we will see that for really high dimensional the computational cost is quite rough and usually it does not give a relevant amount of topological information. Therefore, it is quite usual to see the algorithm applied only to low dimensions.

7. Once we have found all the non-collapsible structures at different dimensions of our complex in our given ϵ, we do the same with other values of ϵ. We will actually do that for each different value in the distance matrix: We have then found the holes that different thickness between points generate.

8. We finally have to identify the holes in different values of ϵ that are the same holes, as in they can be topologically transformed into each other. We then mark at which value of ϵ do they appear and at which value do they collapse. Then we make a graphic marking birth and death of different holes (see persistence barcode and persistence diagram in Figure 2.1).

2.1 Simplicial, abstract and ordered complexes

The most fundamental objective of TDA is to study and get some knowledge about the shape of the data. To do so, some algorithms represent the data as a graph, taking into account the distances between close data points. We will also need a way to identify closed loops in the graph. We will later see (in Section 3.1) that our algorithm will only use n-dimensional triangles called simplices. This means that the way that we will generate our graphs they will not have whole squares or cubes, but plenty of tetrahedrons instead.

To accomplish that, this section will properly introduce the simplices as the geometric object that generalizes a triangle. Later on it will go into the combinatorial equivalent, abstract simplices and how algebraic operations are defined on them. That will require giving a proper order to the later objects for an easier manipulation.

2.1.1 Simplices

First, we will explain a generalization of the triangle: the simplex. Since a simplex can have any arbitrary dimension, the usual wording is n-simplex or n-simplices, where n is the dimension. More precisely, a n-simplex is the convex hull of its $n+1$ vertices that do not lie in a hyperplane of dimension less than n (see Figure 2.2 for a better understanding).

The mathematical expression of a n-simplex is: Given $n+1$ points $p_0,\dots,p_n \in \mathbb{R}^N$ such that $\{p_0-p_1,\dots,p_0-p_n\}$ are linearly independent, the n-simplex S_n that they determine is

$$S_n = \left\{ \theta_0 p_0 + \cdots + \theta_n p_n \;\middle|\; \sum_{i=0}^{n} \theta_i = 1 \text{ and } \theta_i \geq 0 \text{ for } i = 0,\dots,n \right\}.$$

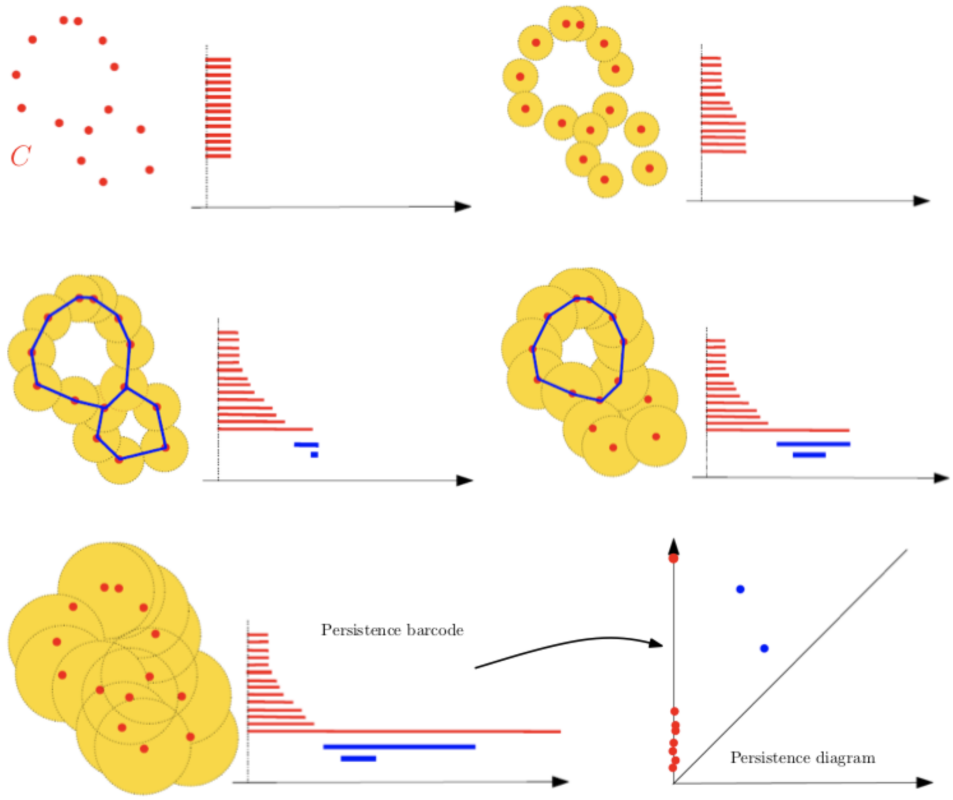


Figure 2.1: The different values of ϵ represented by different radii. Marking if two points are close enough by adding the simplex that they generate. If a set of simplices generate a closed path that can not be transformed into a point we also mark it (see the blue paths in middle row). All of this information is represented into a *persistence barcode* (the barcode on the right of each picture) where a red bar starts whit each point and collapses when two points are close enough (the two points merge). This is also done for holes or non-collapsible paths: The blue bar starts at the lowest value of ϵ where the path is formed and it ends at the lowest value of ϵ where the path is collapsible into a point and is therefore not a hole anymore. As we can see in the middle row, right, one of the closed paths is now collapsible into a point and is accordingly not represented as a hole anymore.

We will refer to this set as the n-simplex they generate and we will use the notation $[p_0,\dots p_n]$ to determine that simplex. It's important to remark that this is the convex hull of the set of points $\{p_0,\dots p_n\}$.

A *face* of a n-simplex or i-face is an i-simplex, $i < n$ that is determined by a subset of the n-simplex generators. The most important case are the $(n-1)$-faces of any simplex $[p_0,\dots p_n]$, defined as $[p_0,\dots,p_{i-1},p_{i+1},\dots,p_n]$ for some $0 \leq i \leq n$. Basically, a face is the simplicial equivalent of a subset and it is useful for notation.

The *interior* of a simplex S will be denoted as $\text{int}(S)$ and it is the subset of points where $\theta_i > 0 \; \forall 0 < i < n$:

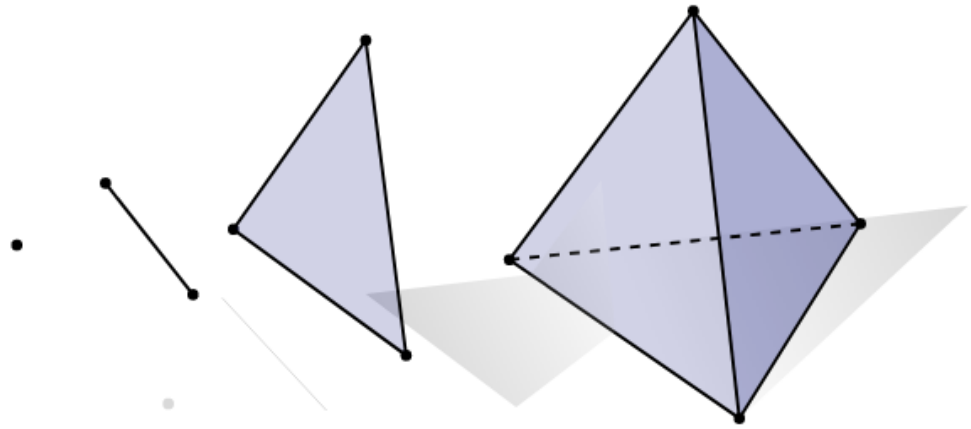


Figure 2.2: From left to right, 0-simplex, 1-simplex, 2-simplex and 3-simplex. Each figure is solid, not hollow.

$$\text{int}(S_n) = \left\{ \theta_0 p_0 + \cdots + \theta_n p_n \;\middle|\; \sum_{i=0}^{n} \theta_i = 1 \text{ and } \theta_i > 0 \text{ for } i = 0, \ldots, n \right\}$$

See how the only difference between the interior and the simplex is that $\theta_i \neq 0$. Therefore, the faces of the simplex are not included (trivially, if you set $\theta_i = 0$ you get the i-th face of the simplex.

2.1.2 Simplicial complex

Now that we have defined what a simplex is we can start working on sets and unions of simplices to get to the specific concept of graph that we are seeking. A simplicial complex X is a set of simplices that satisfy:

- Every face of a simplex in X is also in X (see Figure 2.3 B).
- The non-empty intersections of 2 simplices in X is a face of both (see Figure 2.3 C and D).

2.1.3 Abstract simplicial complexes

Now that we have properly explained the geometric aspect of simplicial complexes we have the need to make an algebraic generalization that our software is capable of working with. If we get rid of the associated geometry of the objects and focus on the combinatorial aspect we get an abstract simplicial complex.

An abstract simplicial complex (we will call it just complex or abstract complex) is a pair of sets (P, X) where the elements of X (simplices) are subsets of P (points) such that:

- $\forall p \in P,\ \{p\} \in X$: all points of the complex are represented as 0-simplices.

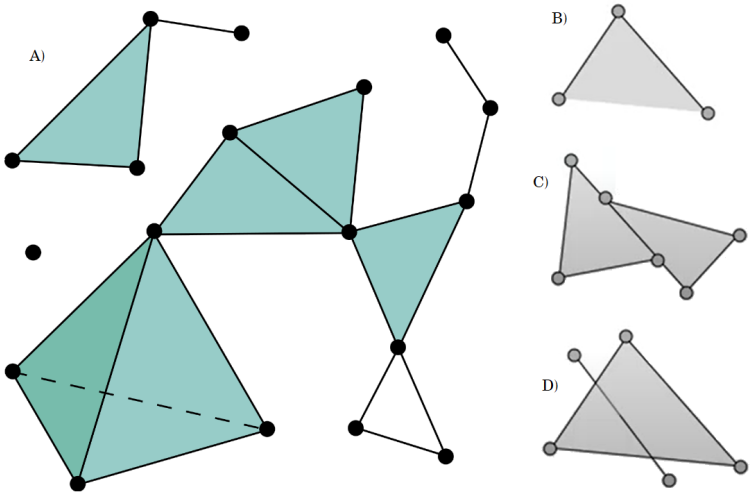
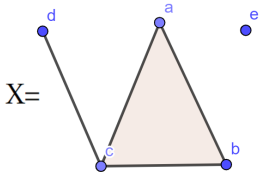


Figure 2.3: A) is a simplicial complex. B) doesn't include one of the faces of the 2-simplex. C) has a union between simplices that is a partial edge. D) makes a non face intersection

- If $x \in X$ and $y \subsetneq x$ then $y \in X$: all proper subsets of a simplex (called faces) are also simplexes in the complex.

Since all elements of P are present in X as singletons, we will denote the abstract simplicial complex (P, X) by X (some authors even define abstract complexes without the use of a set of points P). We will also make some abuse of terminology, calling the elements of P and the singletons of X both points and vertices. Soon we will see that every simplicial complex has an abstract simplicial complex associated (and the other way around). The dimension of any simplex $x \in X$ is defined by the cardinality of x minus 1: $\dim(X) = |x| - 1$.

With this we have properly defined an algebraic representation of complexes that is going to be a lot more manageable from a computational perspective. It is trivial to construct an abstract simplicial complex given a geometrical one (see Figure 2.4). We can also make a geometric realization of a given finite abstract simplicial complex pretty easily making a geometrical simplex (the proof can be found in [3].



$$\rightsquigarrow \quad X_{ab} = \left\{ \begin{array}{l} \{a\},\{b\},\{c\},\{d\},\{e\},\{a,b\}, \\ \{a,c\},\{b,c\},\{c,d\},\{a,b,c\} \end{array} \right\}$$

Figure 2.4: The simplicial complex X has an associated abstract complex in X_{ab}. You can see that every simplex in X is represented as a set in X_{ab}

Ordered and directed simplicial complexes

Some of the calculations in our algorithms have the need of some robust order to the vertices and simplices. Given any abstract simplicial complex we can use a total order over the set of vertices P to generate an *ordered simplicial complex* (also referred as ordered complexes). This way an n-simplex will be referred to using its ordered vertices by $[p_0, \ldots, p_n]$. If you see again Figure 2.4 with the alphabetical order the sets in X_{ab} are already ordered.

Using the order in the vertices we can order all sets that represent each simplex. With that, all simplices will be tuples with the mentioned order restricted to its elements. The resulting set of tuples is a *directed simplicial complex* (also referred as directed complexes). Every tuple represents an ordered simplex and is denoted with a properly modified terminology analogous to how we refer simplices in ordered complexes. It is worth clarifying that we will work with negative tuples, for example: $-(a,b,c) = (a,c,b)$

Finally, it should be noted that different authors use different notations and definitions: some only consider simplicial complexes with order, incorporating order as a a feature of the base definition. On the contrary, other authors find the need of total order too restrictive and use a partial order that, restricted to any simplex, is a total order. Both definition will suffice our needs but we went with the more generalist one.

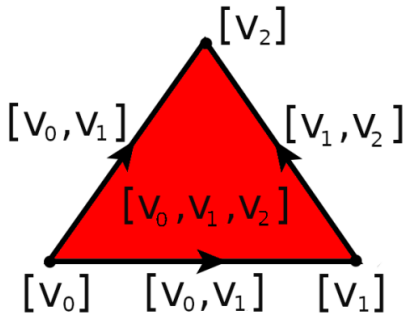


Figure 2.5: We can see the directed simplicial complex $X = [v_0, v_1, v_2]$ and the direction of the simplices that compose it's faces.

Directed simplicial complexes will be the main structure in our work. Now that we have properly defined them we may also look into the functions and relations that we will use on them. therefore, we introduce useful definitions

Simplicial map A simplicial map $f : X \to Y$ between (possibly ordered) complexes is defined as a map between their respective sets of vertices with the property that the images of the vertices of a simplex in X always span a simplex in Y. In other words, if $\{x_0, \ldots, x_n\} \subset X$ span a simplex in X, then $\{f(x_0), \ldots, f(x_n)\}$ spans a simplex in Y. In ordered complexes order need to be preserved.

Subcomplex A complex X (simplicial, abstract or ordered) is a subcomplex of another complex Y if every simplex of X is a simplex of Y.

Boundary We have defined the boundary of a simplex in a geometric way. In an abstract case, the boundary of a n-simplex is the set of $(n-1)$-faces of that simplex. For an abstract simplex $S_n = \{p_0, \ldots, p_n\}$, its boundary would be:

$$\mathrm{Bd}\,(S_n) = \left\{\{p_0, \ldots, \hat{p_i}, \ldots, p_n\} : 0 \leq i \leq n\right\}.$$

where $\hat{p_i}$ means that we delete the p_i element.

The algorithms that are used in TDA will operate on abstract directed simplicial complexes (more about this in Viteoris-Rips section 3.1). Even though this is the case, it is by far easier to understand and visualize the process and the demonstrations with a geometrical reference. Therefore, we will usually talk and work with the geometrical simplicial complexes, explaining and developing theorems and corollaries for topological spaces.

Therefore, from now we will assume all complexes are built using a set of points with a given total order and that between ordered complexes, simplical maps must preserve the order: $v \leq w \implies f(v) \leq f(w)$.

2.2 Chain groups and complexes

To properly define the topological characteristics that are holes, we are going to use abstract algebra. In this section some key concepts will be introduced: chains and boundary maps. To properly define and find holes we need some tools to discern non collapsible paths in a simplicial complex. Chains and boundary maps will help us do that properly in sec:2.3. Similarly to simplices themselves, these are a very important piece in the construction of the future objects. Therefore, some examples and visualization of them will be provided.

n-chain groups: Let X be an ordered simplicial complex. The vector space of n-chains (traditionally called n-chain group) with $\Bbbk$-coefficients, $C_n(X;\Bbbk)$, is the $\Bbbk$-vector space with basis the set of oriented n-simplices of X . Therefore, the elements of $C_n(X;\Bbbk)$ are finite formal linear combinations $\sum_{i=1}^{S} c_i\sigma_i$ where $c_i \in \Bbbk$ and σ_i is an oriented n-simplex of X. When $\Bbbk$ can be deduced from context, we shall omit it altogether and simply denote n-chain vector spaces by $C_n(X)$.

Boundary map The boundary map is the linear transformation $\partial_n : C_n(X;\Bbbk) \to C_{n-1}(X;\Bbbk)$ that maps a n-simplex into the alternating sum of its faces. Let $S_n = [p_0, \ldots, p_n]$ be a n-simplex, then the image of $\partial_n(S_n)$ would be:

$$\partial_n\left([p_0, \ldots, p_n]\right) = \sum_{i=0}^{n} (-1)^i [p_0, \ldots, \hat{p_i}, \ldots, p_n]$$

Here is where the need of order in our complexes become evident: we can't properly produce an alternating sum without order in the elements.

Some examples:

- $\partial_0([p_0]) = [] = 0$
- $\partial_1([p_0, p_1]) = [p_1] - [p_0]$
- $\partial_2([p_0, p_1, p_2]) = [p_1, p_2] - [p_0, p_2] + [p_0, p_1]$
- $\partial_3([p_0, p_1, p_2, p_3]) = [p_1, p_2, p_3] - [p_0, p_2, p_3] + [p_0, p_1, p_3] - [p_0, p_1, p_2]$

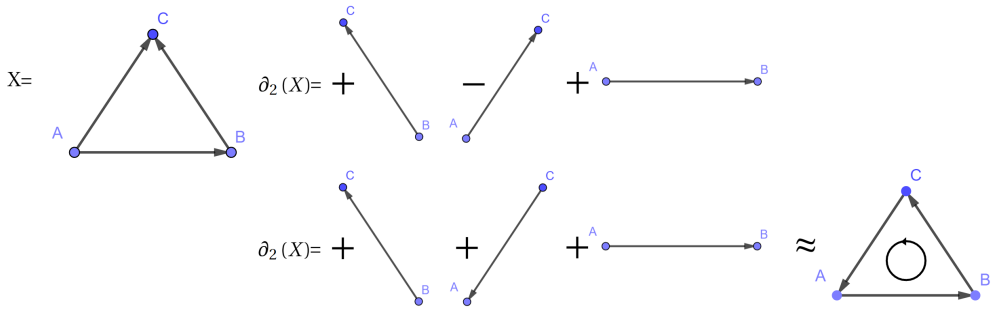


Figure 2.6: In the image we can see how the simplex $X = [A, B, C]$ (with a classical alphabetical order) has, roughly speaking, the orientation of its faces fixed by the boundary map. It is important to remark that the image of the boundary map is the alternate sum of those simplices but, in a general case, that sum is not equal to the complex on the right. That reinterpretation is just put there so the viewer can see how the boundary map aligns in a way the direction of the faces.

The geometric interpretation is that the image of the boundary map applied to a simplex is the alternating sum of the faces that form the boundary of that simplex. We will abuse notation and say boundary of a simplex to refer to the image of the boundary map of said simplex if it has a direction. The boundary map is an algebraic way to encode the geometric boundary of a simplex. Therefore, all the information from the geometric boundary can be extracted with it. The use of an alternating sum is to properly align the direction of the different faces (see Figure 2.6). This is done to preserve the sense of orientation in which faces are connected in higher dimensional simplices.

2.2.1 Chain Complexes

A set $\{(C_i, \partial_i)\}$ of $\Bbbk$-vector spaces C_i and homomorphisms $\partial_i : C_i \to C_{i-1}$

$$\dots \longrightarrow C_{n+1} \xrightarrow{\partial_{n+1}} C_n \xrightarrow{\partial_n} C_{n-1} \longrightarrow \dots$$

is called a chain complex if it satisfies $\partial_{n+1} \circ \partial_n = 0$ for all n. We will call elements in the kernel of ∂_n n-cycles, and elements in the image of ∂_{n+1} n-boundaries. It is essential to realize that, due to the condition that the composition of consecutive ∂_i maps is zero, $\operatorname{im}(\partial_{n+1}) \subseteq \ker(\partial_n)$.

We will now prove that chain groups $C_i(X, \Bbbk)$ and boundary maps ∂_i defined in the last section generate a chain complex. To do that we only have to prove that the

composition of boundary maps cancels:

$$\begin{aligned}
&\partial_{n-1}\circ\partial_n\left([p_0,\dots,p_n]\right)\\
&\quad=\partial_{n-1}\left(\partial_n\left([p_0,\dots,p_n]\right)\right)\\
&\quad=\partial_{n-1}\left(\sum_{i=0}^{n}(-1)^i[p_0,\dots,\hat{p}_i,\dots,p_n]\right)\\
&\quad=\sum_{i=0}^{n}(-1)^i\partial_{n-1}([p_0,\dots,\hat{p}_i,\dots,p_n])\\
&\quad=\sum_{i=0}^{n}(-1)^i\left(\sum_{j=0}^{i-1}(-1)^j\left([p_0,\dots,\hat{p}_j,\dots,\hat{p}_i,\dots,p_n]\right)+\sum_{j=i}^{n-1}(-1)^j\left([p_0,\dots,\hat{p}_i,\dots,\widehat{p_{j+1}},\dots,p_n]\right)\right)
\end{aligned}$$

For any given $0\le a<b\le n$, we will have the term $j=a$, $i=b$ in the left summand and the term $i=a$, $j+1=b$ on the right summand. Therefore creating

$$\begin{aligned}
(-1)^a(-1)^b[p_0,\dots,\hat{p}_a,\dots,\hat{p}_b,\dots,p_n]+(-1)^a(-1)^{b-1}[p_0,\dots,\hat{p}_a,\dots,\hat{p}_b,\dots,p_n]\\
=\left((-1)^{a+b}+(-1)^{a+b-1}\right)[p_0,\dots,\hat{p}_a,\dots,\hat{p}_b,\dots,p_n]\\
=0
\end{aligned}$$

This happens for every two elements of the original simplex, therefore $\partial_{n-1}\circ\partial_n\left([p_0,\dots,p_n]\right)=0$. Since this holds for all n, it happens for all boundary maps: The composition of two consecutive boundary maps equals 0.

We just proved that a sequence of chain groups with a sequence of boundary maps generate a chain complex, a *simplicial chain complex with* $\Bbbk$ *coefficients* $\{(C_i(X,\Bbbk),\partial_i)\}$. Notice how, in this chain, boundaries are just the image of the boundary map and cycles are linear combinations with boundary is 0. It also makes sense that all boundaries are cycles, therefore they cancel out.

Now that that we have properly fleshed out a simplicial complex-associated chain complex, it's easier to discern what boundaries and cycles are. An example similar to the demonstration done of boundary map composition cancelation:

$$\begin{aligned}
\partial_1\circ\partial_2\left([p_0,p_1,p_2]\right)=\partial_1\left(\partial_2\left([p_0,p_1,p_2]\right)\right)=\partial_1\left([p_1,p_2]-[p_0,p_2]+[p_0,p_1]\right)=\\
\partial_1\left([p_1,p_2]\right)-\partial_1\left([p_0,p_2]\right)+\partial_1\left([p_0,p_1]\right)=(p_2-p_1)-(p_2-p_0)+(p_1-p_0)=0
\end{aligned}$$

From here it is obvious that $[p_1,p_2]-[p_0,p_2]+[p_0,p_1]$ is a boundary (the boundary of $[p_0,p_1,p_2]$, to be precise). Changing direction and signs, it might be easier to see the closed loop of oriented n-simplices:

$$\partial_1\left([p_0,p_1]+[p_1,p_2]+[p_2,p_0]\right)=\left([p_1]-[p_0]\right)+\left([p_2]-[p_1]\right)+\left([p_0]-[p_2]\right)=0$$

With this it should be easy to understand that every closed loop will become 0 if transformed by the boundary map. We have already discussed that the boundary map of a simplex results in its faces properly oriented thanks to the alternated sum: It makes the faces align in a way that the boundary map will generate a loop. It is noteworthy that since we are talking about chain groups, the elements of those vector spaces can be called chains.

2.3 Homology of chain complexes

In this section we will go over the usual ways in algebraic topology to identify holes, which are homology groups or vector spaces. To do that, we will better understand the different loops in a surface and how they behave with homotopy or continuous deformation. Observe that boundaries represent loops that can be continuously deformed into a point (via the higher dimensional simplex they come from) while cycles represent any loop. If a cycle can not be transformed into any boundary, it is non-collapsible and, consequently, contains a hole. With that in mind, we will learn how to transform simplicial complexes into a vector space that is a quotient of n-cycles modulo n-boundaries. With that we will find a way of knowing the number of n-dimensional holes in the surface (and what they really are). Finally, we will end the section with some light upon the computational process that replicate this process.

We now may define the homology associated to a complex through the simplicial chain complex. Let X be a simplicial complex and $(C(X;\Bbbk),\partial)$ its simplicial chain complex. The nth *homology group* with $\Bbbk$-coeffients $H_n(X;\Bbbk)$, is the quotient vector space of n-cycles over n-boundaries. That is, $H_n(X;\Bbbk) := \ker(\partial_n)/\mathrm{im}(\partial_{n+1})$. This will always be well-defined since, thanks to the composition condition of chain complexes, $\mathrm{im}(\partial_{n+1}) \subseteq \ker(\partial_n)$.

The main idea is to take equivalence classes where boundaries are equivalent to 0. Once that's the case, two diferent chains are equivalent (or homologous) if the difference is a boundary. $H_n(X)$ being the nth homology group of X, its elements are called homology classes. Each homology class is an equivalence class of cycles that in turn are a linear combination of simplices of the original complex. If the homology class is not equivalent to a boundary it means that those cycles include a region that is not a boundary but yet, a cycle. Therefore we are talking of an equivalence class of chains that enclose a region that is not entirely part of X: each homology class encloses a hole in X.

To better understand different homology groups and holes, look at the torus X with a radius from the center of the hole to the center of the torus of 2 and a radius of the tube of 1 . The smallest contractible space that contains the torus Y:

$$X = \left\{(x,y,z)|\left(2-\sqrt{x^2+y^2}\right)^2 + z^2 = 1\right\},$$
$$Y = \left\{(x,y,z)|\left(2-\sqrt{x^2+y^2}\right)^2 + z^2 \leq 1\right\} \bigcup \left\{(x,y,0)|x^2+y^2 \leq 1\right\},$$

The circle $L_1 = \{(x,y,0)|x^2+y^2 < 1\}$ is a subset of Y but its intersection with the given torus is null. Since the boundary of L_1 in Y is the circumference $\{(x,y,0)|x^2+y^2 = 1\}$ with dimension 1, we can say that L_1 is a 1-dimensional hole of the torus X. Similarly, the interior of the torus $L_2 = \left\{(x,y,z)|\left(2-\sqrt{x^2+y^2}\right)^2 + z^2 \leq 1\right\}$ is a subset of $Y \setminus X$ with a 2-dimensional boundary (the torus itself) and therefore is a 2-dimensional hole.

To sum all it up:

- $H_n(X) = \ker(\partial_n)/\mathrm{im}(\partial_{n+1})$ nth homology groups are the quotient group of n-cycles modulo n-boundaries.
- $H_0(X)$ describes the path-connected components of X.
- $H_1(X)$ describes the one dimensional holes of X (the most common term of holes in casual talking).
- H_n describes the n-dimensional geometric features that can be seen as the number of holes with n-dimensional boundary.

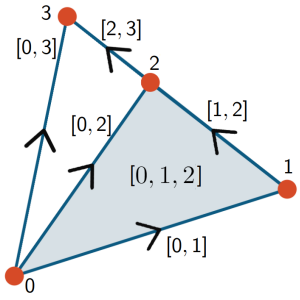


Figure 2.7: The complex X described in the example. It's easy to see that it has one connected component and one 1-dimensional hole in the empty region bounded by 0,2 and 3

We will explain a practical example to complete this section. Let X be the directed simplicial complex $\{[0], [1], [2], [3], [0,1], [0,2], [0,3], [1,2], [2,3], [0,1,2]\}$ (see image fig: 2.7 for better understanding). First we have to calculate the chain groups.

- $C_0(X;\mathbb{Q}) = \mathbb{Q}\{[0], [1], [2], [3]\} \cong \mathbb{Q} \oplus \mathbb{Q} \oplus \mathbb{Q} \oplus \mathbb{Q} = \mathbb{Q}^4$
- $C_1(X;\mathbb{Q}) = \mathbb{Q}\{[0,1], [0,2], [0,3], [1,2], [2,3]\} \cong \mathbb{Q}^5$
- $C_2(X;\mathbb{Q}) = \mathbb{Q}\{[0,1,2]\} \cong \mathbb{Q}$
- $C_k(X;\mathbb{Q}) = 0, \quad \forall k \geq 3$

Next we will find the boundary maps ∂_n:

- $\partial_3 : 0 \to C_2$. Obviously, $\mathrm{img}(\partial_3) = 0$
- $\partial_2 : C_2 \to C_1$. There is only one generator in C_2: $[0,1,2]$. All other elements of C_2 are multiples of it: $k \cdot [0,1,2] (k \in \mathbb{Q})$. Since $\partial_2([0,1,2]) \neq 0$, $\ker(\partial_2) = 0$ and $\mathrm{img}(\partial_2) = \langle [0,1] - [0,2] + [1,2] \rangle$. Therefore $\dim \mathrm{img}(\partial_2) = 1$.
- $\partial_1 : C_1 \to C_0$. Since all cycles have to start and finish in the same point, it is easy to verify that the only two cycles in C_1 are those generated by $[0,1] - [0,2] + [1,2]$ and $[0,2] - [0,3] + [2,3]$ concluding that $\ker(\partial_1) = \langle [0,1] - [0,2] + [1,2], [0,2] - [0,3] + [2,3] \rangle \cong \mathbb{Q}^2$. On the other hand, $\mathrm{img}(\partial_1)$ is a little bit harder. Any element of C_1 will result in a linear combination of $a([1] - [0]) + b([2] - [0]) + c([3] - [0]) + d([2] - [1]) + e([3] - [2])$. The trick resides in the fact that those five generators are not

linearly independent. To visualize it we will use a matrix where each column represents a 1-simplex and each row a 0-simplex. This is the matrix that represent the linear transformation ∂_1:

$$\begin{pmatrix} -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

It's just a question of linear algebra to realize that the rank is 3. This is due to the fact that $\partial_1([1,2]) = \partial_1([0,2]) - \partial_1([0,1])$ and $\partial_1([2,3]) = \partial_1([0,3]) - \partial_1([0,2])$. Therefore, $\text{img}(\partial_1) = \langle [1]-[0], [2]-[0], [3]-[0]]\rangle$, with dimension 3.

- $\partial_0 : C_0 \to 0$. Finally, it is trivial to see that $\ker(\partial_0) = \langle [0], [1], [2], [3]\rangle = C_0 \cong \mathbb{Q}^4$

With this we can calculate the homology groups:

$$\begin{aligned} H_0 &= \ker(\partial_0) \big/ \text{im}(\partial_1) \\ &= \langle [0], [1], [2], [3]\rangle \big/ \langle [1]-[0], [2]-[0], [3]-[0]]\rangle \\ &= \big\langle \big[[0] \big] \big\rangle \cong \mathbb{Q} \\ H_1 &= \ker(\partial_1) \big/ \text{im}(\partial_2) \\ &= \langle [0,1]-[0,2]+[1,2], [0,2]-[0,3]+[2,3]\rangle \big/ \langle [0,1]-[0,2]+[1,2]\rangle \\ &= \big\langle \big[[0,2]-[0,3]+[2,3] \big] \big\rangle \cong \mathbb{Q} \\ H_2 &= \ker(\partial_2)/\text{im}(\partial_3) = (0)/(0) \cong 0 \end{aligned}$$

The information that this gives us about X is not trivial at first. Since the homology groups are quotients, the geometric interpretation is that X represents a topological space with one path-connected component and a one dimensional hole. Remember that homology groups are cycles modulo boundaries, quotients of vector spaces and they inherit the structure of a vector space. In our case every $H_n(X)$ is a $\mathbb{Q}$-vector space but it depends on the field that we chose when defining the chain. The dimension of the n-th homology group determines the number of n-dimensional holes in X, which are represented by the generator classes of $H_n(X)$.

Now that we have properly understood how we extract information through homology, we have understood the theoretical process that happens with a given ordered complex. However, once applying Persistence Homology, a big amount of different ordered complexes will be generated and the process that we exemplified will be applied to all of those complexes. Therefore, we need a fast way of computing homology groups (or the relevant information that we can extract from them). As usual, most of the computation operations will be done using matrices as the one that we have used. See Figure 2.8 to better understand the computational aspect that will be used on the algorithm. Representing each boundary map as a matrix, calculating their respective rank and kernel and then, since it's a quotient of those, the homology group will be $H_n \cong \mathbb{Q}^{\text{kernel}_n - \text{rank}_{n+1}}$ (this does not work in general but it does work for vector spaces).

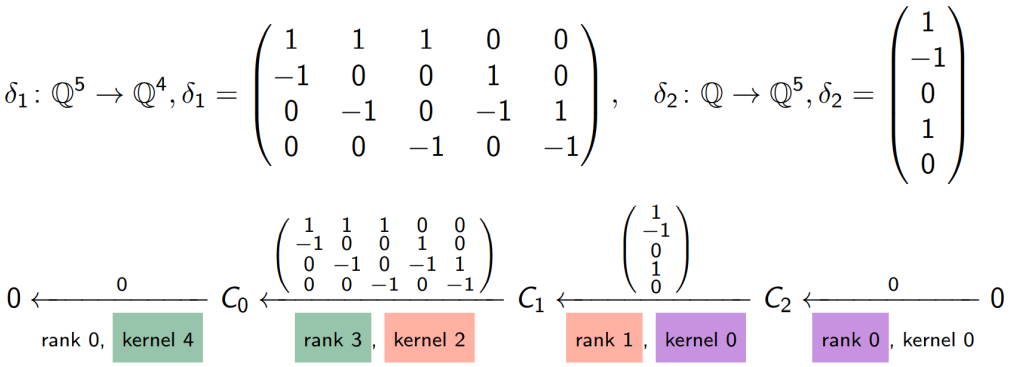


Figure 2.8: In each step, the boundary map is represented as a matrix. With each column an element of the domain and each row an element of the codomain of the boundary map. The reader can verify how the image of $[0,1,2]$ is $+[0,1]-[0,2]+0\cdot[0,3]+[1,2]+0\cdot[2,3]$. See how it has been colorized, H_0 depends on the green values, H_1 on the pink values and so on.

2.4 Functorality of homology

The matrix example before was calculated simplicial homology. It is worth mentioning that simplicial homology and singular homology are equivalent in their results (as do many other homologies [4]). In any way, the important fact is that we have only calculated the homology for a complex. We still don't know how to apply that to a database. As we discussed in the outline we will transform the data into a family of complexes and then calculate the homology groups of each one. We require a way to relate the different complexes and their respective homology groups to properly understand how holes appear and die.

To do so, we need to dive into how homology transforms the category of simplicial complexes and simplicial maps into the category of vector spaces with linear transformations. Moreover, we will introduce a way of mapping between chain complexes and make sure that it works with all previous concepts and is well defined. The section will end with a more general definition of chain group that permits a wider use of TDA.

2.4.1 Homology as a functor

One thing that we are going to show is that homology groups are homotopy invariants. To better use and understand that, it is very useful to see the construction of each homology group as a functor $H_n(-,\Bbbk) : \mathrm{Simp} \to \mathrm{Vect}_\Bbbk$. With Simp we refer to the category of non empty ordered simplicial complexes and order-preserving simplicial maps. The functor maps simplicial complexes into vector spaces over a field $\Bbbk$ that represent their n-th homology group, and simplicial maps into $\Bbbk$-linear maps that match each deformation into its corresponding operation of chains. To be precise, let S, T be simplicial complexes, a simplicial map $f : S \to T$ determines an induced group homomorphism of homology groups $H_n(f) : H_n(S;\mathbb{Q}) \to H_n(T;\mathbb{Q})$ for each $n > 0$ (usually called f_* for short)

It is relevant to note that is the functorality of homology groups what made the algorithm so flexible and powerful (its a topic on its own and we will not further elaborate). If we also see the chain complex as a functor $C_\bullet(-) : \mathrm{Simp} \to \mathrm{Ch}(\mathrm{Vect}_{\Bbbk})$, where a simplicial complex X is mapped into the simplicial chain complex $C_\bullet(X, \Bbbk)$. Simplicial maps, on the other hand, will induce chain maps, the relation between chain complexes that we have been looking for:

Given $(A_\bullet, \partial^A_\bullet)$ and $(B_\bullet, \partial^B_\bullet)$ $\Bbbk$-VS chain complexes, a *chain map* $f : A_\bullet \to B_\bullet$ is a sequence of linear transformations $f_n : A_n \to B_n$ for each n that commutes with the boundary operators of the two chains. That is $\partial^B_n \circ f_n = f_{n-1} \circ \partial^A_n$ for all n, such that it makes the following diagram commute:

$$\begin{array}{ccccccccc}
\dots & \xrightarrow{\partial^A_{n+2}} & A_{n+1} & \xrightarrow{\partial^A_{n+1}} & A_n & \xrightarrow{\partial^A_n} & A_{n-1} & \xrightarrow{\partial^A_{n-1}} & \dots \\
 & & \downarrow{\scriptstyle f_{n+1}} & & \downarrow{\scriptstyle f_n} & & \downarrow{\scriptstyle f_{n-1}} & & \\
\dots & \xrightarrow{\partial^B_{n+2}} & B_{n+1} & \xrightarrow{\partial^B_{n+1}} & B_n & \xrightarrow{\partial^B_n} & B_{n-1} & \xrightarrow{\partial^B_{n-1}} & \dots
\end{array}$$

To better understand how a simplical map induces a chain map, let $f : X \to Y$ be a simplicial map between complexes, then for all $n \in \mathbb{N}$, f induces a linear transformation $f_n : C_n(X) \to C_n(Y)$ that for any given n-simplex $x \in X$, $x = [p_0, \dots, p_n]$, then $f(x) = [f(p_0), \dots, f(p_n)]$ if it spans a n-simplex in Y, or 0 otherwise. From there, it is trivial to prove that this sequence of restricted linear transformations span a chain map.

Is worth noting that, similarly to how a simplicial map between complexes induces a chain map between the associated simplicial chain complexes, a map between chain complexes induces a linear transformation of vector spaces between the nth homology groups of the respective chains. Remember that homology groups are quotients of cycles ($Z_n(X)$), modulo boundaries ($B_n(X)$). With that and the fact that chain maps and boundary operators commute in mind, lets proceed with the proof:

as in $f\partial = \partial f$ lead us to prove that the image of a boundary is a boundary and the image of a cycle is a cycle (in chain maps).

If an element x is a cycle, its boundary is 0. Using the commutation property, the boundary of the image of x is then also a boundary

$$\left.\begin{array}{l} x \in Z_n(X) \leftrightarrow \partial_n(x) = 0 \\ \partial_n(f_n(x)) = f_{n-1}(\partial_n(x)) \end{array}\right\} \to \partial_n(f_n(x)) = f_{n-1}(0) = 0 \to f_n(x) \in Z_n(Y)$$

In a similar way, if an element x is a boundary, it is part of the image of ∂_{n+1}. Substitute x and apply the commutation property

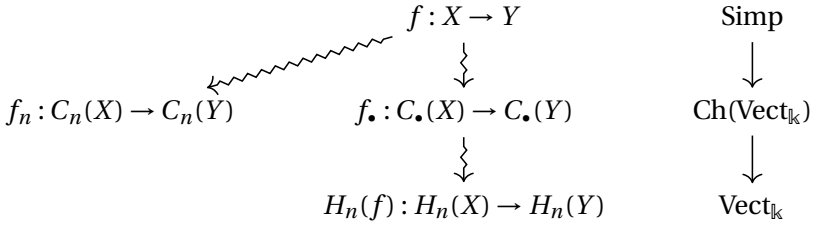
$$\begin{aligned} x \in B_n(X) &\to \exists u \in C_{k+1}(X) : \partial_{n+1}(u) = x \\ &\to f_n(x) = f_n(\partial_{n+1}(u)) = \partial_{n+1}(f_{n+1}(u)) \\ &\to f_n(x) \in B_n(Y) \end{aligned}$$

Therefore, we can restrict f_n to cycles and to boundaries. Therefore, given any chain

map $f_\bullet : A_\bullet \to B_\bullet$ induces a map in homology that is composed by the linear operators:

$$H_n(f) : H_n(A) \to H_n(B)$$
$$H_n(f)([x]) = [f_n(x)]$$

Thanks to this, we can prove that the homology is



And, as we discussed before, H_n defines a functor $H_n : \mathrm{Ch}_\bullet(\Bbbk) \to \mathrm{Vect}_{\Bbbk}$ that goes from the category of chain complexes $\mathrm{Ch}_\bullet(\Bbbk)$ with chain groups $\Bbbk$-vector spaces and chain maps to the category of $\Bbbk$-vector spaces with linear transformations. It is remarkable that we will do some abuse of notation by giving the same name to two different functors. The reason is that we construct the H_n functors in a way that $H_n(X) = H_n(C_\bullet(X))$. Some authors use to different functors with the same notation, some authors describe is as only one functor but use the same notation for the two cases. What is consistent between most authors is that they use the same notation H_n for the two functors.

From here the functorality of homology from this construction only require to proof that $H(\mathrm{id}) = \mathrm{id}$, $H(f \circ g) = H(f) \circ H(g)$, which are rather trivial proofs. All induced maps for their respective homology groups will be isomorphism. The same is true for the homeomorphism of the asociated topological spaces. From here we will se that homeomorphisms between topological spaces induce isomorphism between chain complexes and therefore isomorphism between homotopy groups in all levels.

2.4.2 Chain homotopy

The robustness and flexibility of all of this is thanks to homology functors being homotopy invariant on its own. To see this we will generalize the general concept of homotopy into chain complexes, see the relation that it have with the complex that generate the chain and its uses.

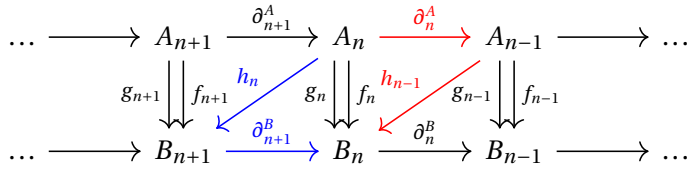
We will start remembering the classical concept of homotopy between two continuous maps: $f, g : A \to B$ are homotopic if exists a third continuous map $h : A \times [0,1] \to B$ such that

$$\begin{cases} h(x,0) = f(x) \\ h(x,1) = g(x) \end{cases}$$

And h is what we call an homotopy: a continuous deformation between the two maps.

In the same vein, we have the need of a similar concept to be able to say that two chain maps are homotopic. In this case, we say that two chain maps $f, g : A_\bullet \to B_\bullet$ are chain-homotopic if there exists $h : A_n \to B_{n+1}$ for each n such that $f_n - g_n = \partial^B_{n+1} \circ h_n - h_{n-1} \circ \partial^A_n$. The definition is analogue of the notion of homotopy of maps of spaces. For

reference, in a diagram we marked in different color the two compositions that are subtracted.



It's not intuitive at first why call f and g chain homotopic. The reason is that if we take two abstract simplicial complexes whose geometric realizations are homotopically equivalent, the associated chain complexes are homotopically equivalent. It can be derived from considering the chain complex of $A \times [0,1]$ and B and the homotopy $h : A \times [0,1] \to B$. The real reason that chain homotopy is important is because, if f and g are chain homotopic, then

$$\partial_n \circ (f_n - g_n) = \partial_n \circ \partial_{n+1} \circ h_n - \partial_n \circ h_{n-1} \circ \partial_n = 0 - \partial_n \circ h_{n-1} \circ \partial_n$$

f_n and g_n differ only by boundary. Therefore they induce the same homology. That is a lot more intuitive and similar to classic homotopy and continue deformations of maps. From this we get that if a simplicial map $f : X \to Y$ is an homotopy equivalence, then f induces an isomorphism on homology. This confirms that we have a well defined functor:

$$H_n : \text{Simp} \to \text{Vect}_{\mathbb{k}}$$

Remark that, as we mentioned in the end of the last subsection, this proofs that $|X| \simeq |Y| \implies H_n(X) \cong H_n(Y)$. This is the last piece needed to proof the so much praised functorality of the method. Moreover, it makes it so homotopy is functorial and can be seen as a functor Ho, even though this is a fact that we will not use directly:

$$\text{Ho}(\text{Simp}) \to \text{Ho}(\text{Ch}(\text{Vect}_{\mathbb{k}})) \to \text{Vect}_{\mathbb{k}}$$

With a wider version of the definition of n-chains the same constructions that we made for $\text{Vect}_{\mathbb{k}}$ can also be done for R-modules, covering vector spaces, abelian groups and more:

Chain group: for any simplicial complex X, given a fixed n and a ring R, the n-chain group $C_n(X,R)$ is the free R-module with basis the set of oriented n-simplices. The elements of $C_n(X,R)$ are called n-chains and are linear combinations of the oriented n-simplices of X.

Remark that for R-modules we could have an abelian group with coefficients in $\mathbb{Z}$, which we have used as $\mathbb{Q}$-vector spaces in some examples to get easier quotients. Any other ring could work, $\mathbb{Z}$, for example, results in quotients like $\mathbb{Z}/2\mathbb{Z}$, and has been used for computational aspect of TDA. This is the general version with R-modules but is important that it can operate with Abelian groups and $\mathbb{k}$-vector spaces. We have to remark this because, even though those are R-modules, the internal operations are not the same and functors, unlike other transformations, care about objects, not sets. This means that internal operations are relevant this time.

The important aspect to realize from this last section is that homology can be applied in immense amount of ways: with vector spaces, abelian groups or any other R-module, giving a lot of freedom and allowing TDA to be applied even in edge cases. The functorality that we seen is also a great trait . Functoriality means that continuous maps of space give rise to maps of path components. The fact that all our constructions are functorial transformation allows us to search for generators of H_n through different values of ε.

CHAPTER 3

TOPOLOGICAL DATA ANALYSIS AND PERSISTENCE HOMOLOGY

One of the main points of topological data analysis is that we find a way of retrieving topological features out of a discrete set of points. In the last chapter we have revised and explained a lot of tools that use complexes and chains to specifically do that. But there is a crucial question that we have not answered yet: How do we transform our sets of points into complexes to properly use those tools? How do we generate complexes from data?

We first look at the data as a discrete set of points in a metric space with each data entry represented by a point of the set. We want to then get some topology notion from that set and the first problem is that, as expected, the natural topology of the point cloud is trivial and offers next to no information. There is a clever solution to this problem, which consists in transforming each point into a ball of a certain radius ϵ and then look at unions of balls. Instead of a discrete space, we now have a manifold.

In this chapter we will go over two different ways to transform a set of balls of a given radius ϵ into a complex. We will also see how to transform any collection of open sets into a complex and a way to infer topological information from samples of a manifold. All of this will be applied repeatedly to generate a sequence of complexes and all their respective homology groups. Finally, we will end up the chapter with methods to extract information from this sequence of chains and the maps between them, and to treat that information.

3.1 Vietoris-Rips and Čech complexes

Converting a discrete set of points P into the union of balls of radius ϵ is a direct and practical way to generate a non trivial homology group. However, finding the topological information of high dimensional manifolds is still pretty hard that way.

Thus TDA uses simplicial complexes to circumvent that problem: Instead of creating a manifold by converting the set of points into the union of balls, it generates complexes whose geometric realization is comparable to the union of those balls to get some topological info of the data. In essence, we will try to combine the two ideas in one algorithm to generate accurate simplices.

3.1.1 Čech and Nerve complex

Let X be a discrete set of points in a given metric space (M, d_M). Given a fixed $\epsilon > 0$, the *Čech complex* of X, $\check{C}_\epsilon(X, d_M)$ is the simplicial complex constructed as follows:

- Take the points of X as the vertex set (the P set) of the complex.
- Then, if a subset of points $\{x_0, x_1, \ldots, x_k\} \subset X$ satisfies $\bigcap_i B_\epsilon(x_i) \neq \emptyset$ then the k-simplex $[x_0, x_1, \ldots, x_k]$ is part of the complex.

The constructed object should be the complex:

$$\check{C}_\epsilon(X, d_M) = \left\{ [x_0, \ldots, x_n] \;\middle|\; \bigcap_{i=0}^{n} B_\epsilon(x_i) \neq \emptyset \right\}$$

It's easy to see that the Čech complex is the construction method seen in points 3 and 4 of the chapter 2 outline. It is a complex representation of the manifold generated by the union of balls as we will properly see later in [1] thus conserving all the properties that we covered in the chapter 2 and solving our main issue with discrete data sets.

Since we will work a lot with discrete sets of points in a given metric space, we will from now on abuse notation and call (X, d_X) the combination of a discrete set of points X in a given metric space (M, d_M). We will do this because we don't care about M at all, only about the given distance d_M (which we will call d_X from now on). Some authors work exclusively with finite metric spaces (making the elements of X the only elements of their metric space, which, in practice, is identical to what we work with). This will make even more sense in the next sections.

The Čech complex as described transforms a set of discrete points into an abstract simplicial complex. Nevertheless, it is worth mentioning that it is a special case of a more generic standard construction in algebraic topology that generates a simplicial complex from a family of open sets: the *nerve* or *nerve complex*. To be more precise, the Čech Complex is the nerve of a family of balls.

Let $\{U_i\}_{i \in I}$ be a family of open sets, the *nerve* $N(\{U_i\}_{i \in I})$ is the abstract simplicial complex generated by taking a set of points $\{i_0, i_1, \ldots, i_n\} \subset I$,if it satisfies $\bigcap_n U_j \neq \emptyset$ then the n-simplex $[i_0, i_1, \ldots, i_n]$ is part of the complex. Observe that there will be a 0-simplex for each element of I that will serve as the vertices. In short:

$$N(\{U_i\}_{i \in I}) = \left([i_0, \ldots, i_n] \middle| \bigcap_{j=0}^{n} U_j \neq \emptyset \right)$$

There are two main reasons why we introduced the nerve complex

- The most important reason to properly introduce the nerve is that, as a generalization of the Čech complex, it can be used as a way to generate complexes starting from any family of sets that are not balls. This is a big part of the Mapper algorithm seen in subsection: 4.3 and the main reason Mapper and persistent homology are related

- We introduce the nerve here because of the *nerve lemma*, a classical result that can be seen in [5]. This lemma proves that there is a homeomorphism between the Čech complex and the union of balls that generates it.

Even with all of this we still have a problem, computing whether the intersection of the balls is empty to generate the complex is computationally heavy in higher dimensions and we want to be able to generalize all these concepts into non-Euclidean spaces. To adjust this we need to find a new construction that is computationally more manageable but still retains the great properties of the Čech complex.

3.1.2 Vietoris-Rips complex and functorality

To solve the problem we will look back at the main idea of the Čech complex and come by a condition that is similar to the intersection of balls but results in easier ways of computing it. The main property of the intersection of balls is that it assures a certain distance between the points that generate them but it gets more complicated when there is more than 2 balls. If we do not use balls altogether and we boil it down to just the distance between the different points we end up with the *Vietoris-Rips (VR)*:

Let (X, d_X) be a discrete set of points with a defined distance. Given a fixed $\epsilon > 0$ the *Vietoris-Rips complex* of X, $\mathrm{VR}_\epsilon(X, d_X)$ is the simplicial complex constructed as follows:

- Take the points of X as the vertex set of the complex.

- Then, if a subset of points $\{x_0, x_1, \ldots, x_k\} \subset X$ satisfies $\forall 0 \leq i, j \leq k; d_X(x_i, x_j) \leq 2\epsilon$ then the k-simplex $[x_0, x_1, \ldots, x_k]$ is part of the complex.

The constructed object should be the complex:

$$\mathrm{VR}_\epsilon(X, d_X) = \left\{ [x_0, \ldots, x_n] \;\middle|\; \forall\, i, j \;\; d(x_i, x_j) \leq 2\epsilon \right\}$$

Just looking at the thought process that got us to define the VR complex, it is easy to see the similarities between it and the Čech complex. Nevertheless, the two constructions are sometimes different as can be seen in Figure 3.1.

The main problem that VR complexes have is that there is no result analogous to the Nerve lemma for the Čech complex. However, there is a close relationship between the two of them and we will see in the end of subsection: 3.1.3 that the topological information that we seek can still be retrieved using VR. It is worth mentioning that, since the distance between the centers of two balls is less than the sum of their radii and the triangle inequality of any distance function, it is trivial to prove that:

$$\check{C}_\epsilon(X, d_X) \subseteq \mathrm{VR}_\epsilon(X, d_X) \subseteq \check{C}_{2\epsilon}(X, d_X)$$

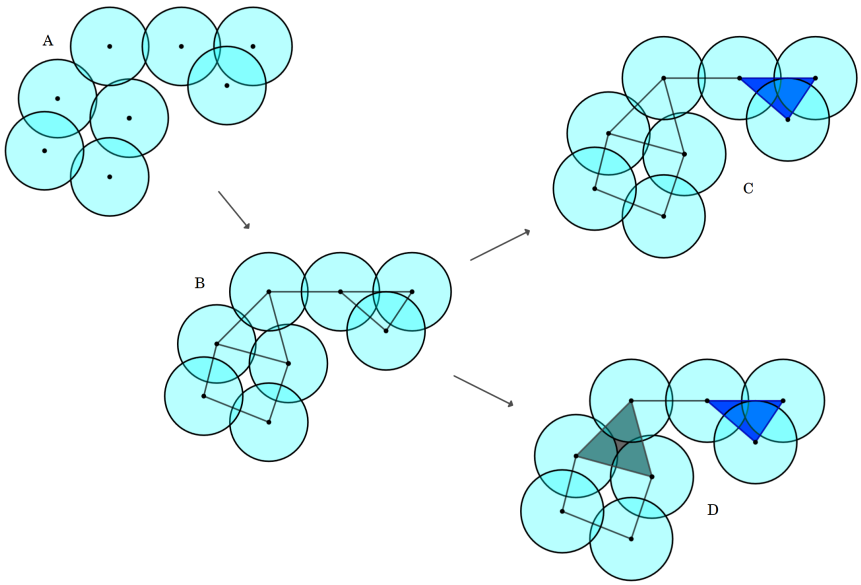


Figure 3.1: As we can see in A, the construction of the Vietoris-Rips and the Čech complex start taking the points and the balls centered on those balls of radius ϵ. For oblivious reasons the 1-simplices would be identical in the two complexes (B) but, at higher dimensions, we can see how groups of points at pairwise distance less than 2ϵ of each other can exist with void intersection: in C the Čech complex has only one 2-simplex. On the other hand, in D we can observe how the Vietoris-Rips complex of the same discrete set generator has an additional 2-simplex (the black one)

Due to some results that we will see, the VR complex succeeds at retrieving information, doing a better job than the Čech complex thanks to its easier computation. Therefore, we will focus mainly on the VR complex from now on; furthermore, the properties of the Čech complex are analogous to the ones that we will discuss about VR.

The most important of the properties of both constructions is functorality, both in the chosen amplitude ϵ and the generator set of points X. By construction, is quite trivial that for a given $\epsilon_1 < \epsilon_2$ and a pair (X, d_X), there is an inclusion $\mathrm{VR}_{\epsilon_1}(X, d_X) \subseteq \mathrm{VR}_{\epsilon_2}(X, d_X)$. Moreover, that inclusion induces a simplicial map $\mathrm{VR}_{\epsilon_1}(X, d_X) \to \mathrm{VR}_{\epsilon_2}(X, d_X)$. Furthermore, let f be a Lipschitz continuous map from (M, d_M) to (N, d_N), and let Y be the image of X (for our interest we can just see $f : (X, d_X) \to (Y, d_Y)$)

$$f : X \to Y \quad \text{such that} \quad \exists k \geq 0 : \forall x_1, x_2 \in X,\ d_Y(f(x_1), f(x_2)) \leq k d_X(x_1, x_2)$$

Then, because we now know an upper limit in the distance between the Y points we know that an induced simplicial map

$$f : \mathrm{VR}_{\epsilon}(X, d_X) \to \mathrm{VR}_{k\epsilon}(Y, d_Y)$$

exists for any given ϵ. Every set of points from X that stem a simplex in $\mathrm{VR}_{\epsilon}(X, d_X)$ due to being at a pairwise distance less than ϵ will be at a pairwise distance less than

$k\epsilon$ in (Y, d_Y), generating a simplex in $\mathrm{VR}_{k\epsilon}(Y, d_Y)$. The two functorial aspects of the construction arise from this two points, the Vietoris-Rips construction specifies:

- The functor $\mathrm{VR}_\epsilon(-)$: FMLips $\to$ Simp, where FMLips is the category of finite metric spaces and Lipschitz maps with constant 1.
- The functor $\mathrm{VR}_{(-)}(X, d_X) : \mathbb{R} \to$ Simp, where $\mathbb{R}$ is considered as a category with set of objects $\mathbb{R}$ and arrows $\varepsilon_1 \to \varepsilon_2$ whenever $\varepsilon_1 \leq \varepsilon_2$.

The first one, given a fixed ϵ, transforms the category of sets of points with metric maps (LC 1) to the category of simplicial complexes with simplicial maps. The second one, given a fixed pair (X, d_X), transforms each possible $\epsilon \in \mathbb{R}$ into the simplicial complex generated by $\mathrm{VR}_\epsilon(X, d_X)$.

The implications here is that if we change ϵ around with a given data set (X, d_X), a map between the associated complexes will exist. Furthermore, the same is true if we change the data set respecting the original distances (we could add points to the original X), generating a map from the original data set to the extended one.

3.1.3 Niyogi-Smale-Weinberger

With what we have seen, we can already get some information about the manifold that the union of balls generate. But, similar to classical statistics, we want to infer information from the population through a sample. To be more specific to our case, we assume that the data is sampled from a population that is a manifold with a Riemmanian structure. Consequently, we want to infer topological data of the population that conforms our sample but, when is that possible? As in classical statistics, a lot of problems surrounding sampling arise: when is the sample big enough? when is the sample representative? how do the properties of the original manifold related to all of this?

As a result, the size of this topic might be too big for the introductory nature of this project. Therefore, we will mention some results and bring additional lectures for the interested reader but we will no dive into it. One of the main results is the *Niyogi-Smale-Weinberger theorem* [1] that states:

Theorem (Niyogi-Smale-Weinberger)**.** *Let X be a set of points drawn from M a compact submanifold of $\mathbb{R}^n$. Under certain conditions, there is a homotopy equivalence between the union of balls that the points generate and the manifold*

$$\bigcup_{x \in X} B_\epsilon(x) \simeq M$$

The conditions that need to be met depend on the number of points taken, the size of the manifold and the minimum radius at which the manifold self-intersects. The most important result is that, in particular, the homology groups coincide. With this it is proven that, under certain conditions, the Čech complex of a sample has the same homology group than the original manifold.

This result provides a vital consistency check: we can accurately recover topological invariants from a topological space using finite sets of samples. The theorem only

proves it for the Čech complex but conveniently there is an analogous result for the VR complex [6]. However, a problem arises: We don't have a reasonable way to choose the values of ϵ that better recover the topological information of the originating space. Moreover, noise and sampling variation can cause instability in the VR complex and its homology.

Furthermore, even when we know the most optimal ϵ, the amount of sampling needed to discern the topological features is too big and is usually impossible to know which value of ϵ we need to chose. In other words, getting just one sample would require an unreasonable amount of points and a value of ϵ that we will usually not have any way of guessing. All of this in combination with the possibility of different scales in the same manifold makes it so that choosing a single value for ϵ is unresaonable.

To solve this problem we resort to the functorality of both constructions that we discussed at the end of the subsection 3.1.2. Thanks to the possibility of mapping between the results of different values of ϵ, we can look at different scales and an array of values of ϵ and map them to observe the homological features that remain stable through a fan of values. The features that persist at a bigger fan of scales usually are the most relevant features.

This is the persistence part of the algorithm. From an array of values we look at how different invariants persist through scale. Before considering how to choose the different values of ϵ, we have to use the functorial aspect of $\mathrm{VR}_{(-)}(X, d_X)$ to find a proper way to compare the various $H_n\left(\mathrm{VR}_{\epsilon_i}(X, d_X)\right)$ to track the behavior of homology invariants through scale. We already discussed the (inclusion) simplicial maps that go from the VR complex generated by ϵ to the VR complex generated by $\epsilon \leq \epsilon'$:

$$\mathrm{VR}_{\epsilon}(X, d_X) \hookrightarrow \mathrm{VR}_{\epsilon'}(X, d_X)$$

We can successfully chain the maps for an increasing sequence of ϵ_i getting a sequence of simplicial maps

$$\cdots \hookrightarrow \mathrm{VR}_{\epsilon_{i-1}}(X, d_X) \hookrightarrow \mathrm{VR}_{\epsilon_i}(X, d_X) \hookrightarrow \mathrm{VR}_{\epsilon_{i+1}}(X, d_X) \to \ldots$$

Thanks to the functorial aspect of homology, for any fixed n we can apply the functor $H_n : \mathrm{Simp} \to \mathrm{Vect}_{\Bbbk}$ to both parts of the simplicial map and to an entire sequence:

$$\cdots \to H_n\left(\mathrm{VR}_{\epsilon_{i-1}}(X, d_X)\right) \to H_n\left(\mathrm{VR}_{\epsilon_i}(X, d_X)\right) \to H_n\left(\mathrm{VR}_{\epsilon_{i+1}}(X, d_X)\right) \to \ldots$$

We can then always compose the aforementioned functor $\mathrm{VR}_{(-)}(X, d_X) : \mathbb{R} \to \mathrm{Simp}$ with the homology group functor $H_n : \mathrm{Simp} \to \mathrm{Vect}_{\Bbbk}$ to get

$$H_n\left(\mathrm{VR}_{(-)}(X, d_X)\right) : \mathbb{R} \to \mathrm{Vect}_{\Bbbk},$$

that maps any $\epsilon \in \mathbb{R}$ to the vector space that represents the nth homology group of the Vietoris-Rips simplicial complex of the finite set X with a given distance function d_X.

3.2 Persistent homology

This functor is the tool that fixes one of the biggest challenges of Persistent Homology. We wanted to track how homology evolved as the value of ϵ changed and how persistent the different homological features were in this range of scales. We already had ways to get those homological features in any value of ϵ, but we didn't have any way to compare the different homology groups generated.

Before explaining how to tie the different homology, we will sum it all up: with our given set of points X, we choose a distance function d_X, our set of scale values $E = \{\epsilon_i\}$, and the field $\Bbbk$, we want to transform a metric space into a grid of vector spaces before continuing with homology:

$$(X, d_X) \xrightarrow{\text{simplices}} \{\text{VR}_{\epsilon_i}(X, d_X)\}_{\epsilon_i \in E} \xrightarrow{\text{chains}} \{\{C_n(\text{VR}_{\epsilon_i}(X, d_X); \Bbbk)\}_{n \in \mathbb{Z}^+}\}_{\epsilon_i \in E} \to \dots$$

For practical reasons, from now on, we will write $\text{VR}_\epsilon(X)$ when d_X remains unchanged as we have done for homology and chain groups when the field $\Bbbk$ was obvious or given and unchanged.

So, after choosing the distance function d_X we express our data as a finite metric space (X, d_X). Afterwards we choose a set of distance values $E = \{\epsilon_i\}$ and we transform our metric space into a set of complexes through the usage of Vietoris-Rips. Finally, we transform each of those complexes into chain complexes with chain maps between them. Suppose that we have a fixed distance function d_X and a field $\Bbbk$:

$$\dots \xhookrightarrow{f_{\epsilon_{i-2},\bullet}} C_\bullet(\text{VR}_{\epsilon_i - 1}(X)) \xhookrightarrow{f_{\epsilon_i - 1,\bullet}} C_\bullet(\text{VR}_{\epsilon_i}(X)) \xhookrightarrow{f_{\epsilon_i,\bullet}} C_\bullet(\text{VR}_{\epsilon_{i+1}}(X)) \xhookrightarrow{f_{\epsilon_{i+1},\bullet}} \dots$$

Where the $f_{\epsilon_i,\bullet}$ chain maps are induced by the inclusion simplicial maps $f_{\epsilon_i} : \text{VR}_{\epsilon_i}(X) \to \text{VR}_{\epsilon_{i+1}}(X)$ that send each element x of one complex into itself. Remember that there is an inclusion $\text{VR}_{\epsilon_i}(X) \subset \text{VR}_{\epsilon_{i+1}}(X)$. We will refer to $f_{\epsilon_i,\bullet}$ as inclusion chain maps.

To better visualize what is going on, we will expand each chain into its different chain groups to see the grid of vector spaces. This is going to be useful to later on visualize the homology groups and the linear transformations induced by the chain maps at figure: 3.2

3.2.1 Birth and death of topological features

Now we will apply homology to the grid, getting homology groups and the linear transformations between homology groups induced by the chain maps. From using the homology functor, we can go back to the definition of homology as the quotient of cycles module boundary. Remember that $H_n(X) = Z_n^i(X)/B_n^i(X)$, where $B_n^i(X)$ are the boundaries of $C_n(\text{VR}_{\epsilon_i}(X))$, $B_n^i = \text{im}(\partial_{n+1}^i)$ and $Z_n^i(X)$ are the cycles $Z_n^i(X) = \ker(\partial_n^i)$. We now apply the homology functors to the grid, visualizing the different homology groups as quotients of the different chain groups. And we also picture the linear transformations between the homology groups induced by the inclusion between

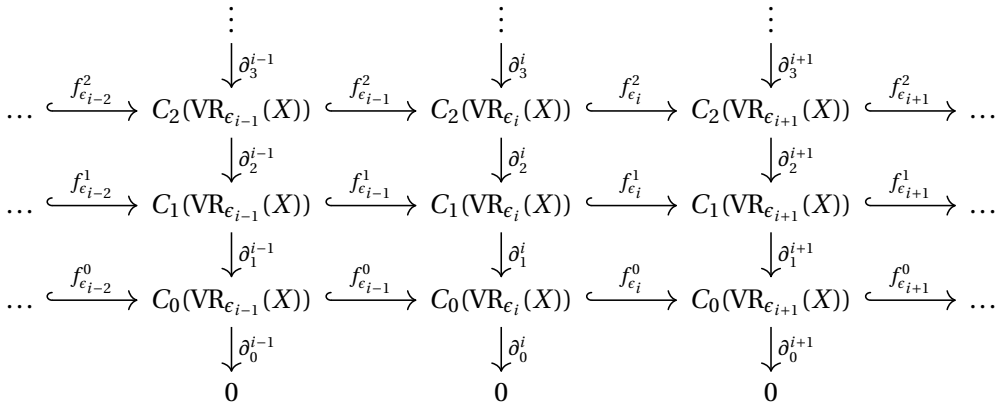


Figure 3.2: Every column represents the chain complex of a given simplicial complex generated with Vietoris-Rips with a set value ϵ_i. Also, in each row there are the $\mathbb{k}$-vector spaces with basis sets of oriented n-simplices. Since $f_{\epsilon_i,\bullet}$ are induced by inclusions, $f^n_{\epsilon_i}$ will send any linearly independent elements of $C_n(\mathrm{VR}_{\epsilon_i}(X))$ into linearly independent elements of $C_n(\mathrm{VR}_{\epsilon_{i+1}}(X))$. Therefore, $f^n_{\epsilon_i}$ is also an inclusion no matter the values of i and n.

chain complexes. Boundary maps tags are not included for visibility, but the gray grid in the background is the same that we just visualized. The diagram helps visualize how, with algebraic topology, TDA converts a set of points X into sequences of homology groups.

We see that, as discussed before, the inclusion function between chain groups (f_{ϵ_i}) of different chains induces a linear transformation between the homology groups. Remark that f functions are inclusions but $H_n\left(f\right)$ are not necessarily. This is because cycles and boundaries change at different paces: some element $x \in C_n(\mathrm{VR}_{\epsilon_{i-1}}(X))$ might be a cycle that is not a boundary in $C_n(\mathrm{VR}_{\epsilon_{i-1}}(X))$, while $f(x) \in C_n(\mathrm{VR}_{\epsilon_i}(X))$ is a boundary in $C_n(\mathrm{VR}_{\epsilon_i}(X))$. The informal geometric interpretation of the argument is that, as the radius of the balls ϵ increases, some balls that formed a hole, might touch each other, generating a new simplex. The cycle that generated the hole is now a boundary, "filling" the hole and making the hole *die* at high enough values of ϵ. See figure: 2.1.

Before continuing, we will simplify the notation used for the functions of the type $H_n\left(f_{\epsilon_i}\right)$. Since they are always between $H_n\left(\mathrm{VR}_{\epsilon_i}(X)\right)$ and $H_n\left(\mathrm{VR}_{\epsilon_{i+1}}(X)\right)$, we can do some abuse of notation to ease the reading and it will still be easy to discern the different functions by looking at where are they applied. Therefore, we will refer to $H_n\left(f_{\epsilon_i}\right)$ as γ_i from now on.

We will call the elements of $H_n\left(\mathrm{VR}_\epsilon(X)\right)$ n-dimensional features at scale ϵ. Roughly speaking, these elements represent n-dimensional holes (section: 2.3) in the geometric realization of $\mathrm{VR}_\epsilon(X)$. These induced functions allow us to compare the homology groups and get information about how the topological features change at different scales which is really useful. Moreover, they are used to know the smallest and biggest

$\vdots$
$\vdots$
$\vdots$
$\dots$
$f^2_{\epsilon_{i-2}}$
$C_2(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$f^2_{\epsilon_{i-1}}$
$C_2(\mathrm{VR}_{\epsilon_i}(X))$
$f^2_{\epsilon_i}$
$C_2(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$f^2_{\epsilon_{i+1}}$
$\dots$
$z_2^{i-1}/_{B_2^{i-1}}$
$z_2^i/_{B_2^i}$
$z_2^{i+1}/_{B_2^{i+1}}$
$H_2(f_{\epsilon_{i-2}})$
$H_2(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$H_2(f_{\epsilon_{i-1}})$
$H_2(\mathrm{VR}_{\epsilon_i}(X))$
$H_2(f_{\epsilon_i})$
$H_2(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$H_2(f_{\epsilon_{i+1}})$
$f^1_{\epsilon_{i-2}}$
$C_1(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$f^1_{\epsilon_{i-1}}$
$C_1(\mathrm{VR}_{\epsilon_i}(X))$
$f^1_{\epsilon_i}$
$C_1(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$f^1_{\epsilon_{i+1}}$
$z_1^{i-1}/_{B_1^{i-1}}$
$z_1^i/_{B_1^i}$
$z_1^{i+1}/_{B_1^{i+1}}$
$H_1(f_{\epsilon_{i-2}})$
$H_1(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$H_1(f_{\epsilon_{i-1}})$
$H_1(\mathrm{VR}_{\epsilon_i}(X))$
$H_1(f_{\epsilon_i})$
$H_1(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$H_1(f_{\epsilon_{i+1}})$
$f^0_{\epsilon_{i-2}}$
$C_0(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$f^0_{\epsilon_{i-1}}$
$C_0(\mathrm{VR}_{\epsilon_i}(X))$
$f^0_{\epsilon_i}$
$C_0(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$f^0_{\epsilon_{i+1}}$
$z_0^{i-1}/_{B_0^{i-1}}$
$z_0^i/_{B_0^i}$
$z_0^{i+1}/_{B_0^{i+1}}$
$H_0(f_{\epsilon_{i-2}})$
$H_0(\mathrm{VR}_{\epsilon_{i-1}}(X))$
$H_0(f_{\epsilon_{i-1}})$
$H_0(\mathrm{VR}_{\epsilon_i}(X))$
$H_0(f_{\epsilon_i})$
$H_0(\mathrm{VR}_{\epsilon_{i+1}}(X))$
$H_0(f_{\epsilon_{i+1}})$
0
0
0

scales where any given feature exists:

- Given a non zero element $h \in H_n\left(\mathrm{VR}_{\epsilon_i}(X)\right)$ if $\nexists g : \gamma_{i-1}(g) = h$, it is a hole in $\mathrm{VR}_{\epsilon_i}(X, d_X)$ but not in $\mathrm{VR}_{\epsilon_{i-1}}(X)$, we say that h *is born* at i
- Given an element $h \in H_n\left(\mathrm{VR}_{\epsilon_{j-1}}(X)\right)$, if $\gamma_j(h) = 0$ or if there exists another element h' that was born earlier and $\gamma_j(h) = \gamma_j(h')$, we say that h *dies* at j. In case two different elements h, h' satisfy $\gamma_j(h) = \gamma_j(h')$ and are born at the exact same type, we arbitrarily chose one of them to die.[1]

The class h represents an n-dimensional feature in Viteoris-Rips complexes at values from $[\epsilon_i, \epsilon_j)$: For values $\epsilon' < \epsilon_i$, the elements of X that generate h don't get to generate a cycle in $\mathrm{VR}_{\epsilon'}(X)$. And for values $\hat{\epsilon} \geq \epsilon_j$, those same elements generate a boundary in $\mathrm{VR}_{\hat{\epsilon}}(X)$.

This last part is the reason why the functoriality of $H_n\left(\mathrm{VR}_{(-)}(X)\right)$ is so important. The functor, with the induced transformations, provides a way to compare homologies of the complexes of different ϵ values. Furthermore, these sequences are the central piece of persistent homology, as the information that we extract with this tool is the birth and death of topological features. With those two values we know how persistent a feature is and so, we can now search for the most persistent features.

3.3 Persistence modules and homology

Now we will introduce a more general notation for what the diagrams in the last section represent. It will help better understand that the sequence of Vietoris-Rips complexes is a particular case of the generalized case of filtration. We go over how the sequences of n-chain groups are persistent modules that, alongside homology, generate persistent simplicial homology. Moreover, this new concept improves the definition of birth and death of the topological features. At the end of the section all comes together in the creation of a multiset that will clearly display all the data in a persistence diagram.

A *filtered complex* is a collection of simplicial complexes $\{X_s\}_{s\in\mathbb{R}}$ such that for each $s \leq t$, X_s is a subcomplex of X_t. It is important to note that the functor $\mathrm{VR}_{(-)}(X)$ is a filtered system and the sequence of simplicial complexes $\{\mathrm{VR}_\epsilon(X)\}_{\epsilon\in\mathbb{R}}$ is a filtered complex.

A *persistence module* is a functor from $\mathbb{R}$ as a partially ordered set to a category of modules (vector spaces in our case). Essentially, it is a collection of vector spaces $V(x)$ with linear maps $f_{xy} : V(x) \to V(y)$ for each pair $x \leq y$. These are called *structure maps* and must satisfy $\forall r \leq s \leq t$, $f_{st} \circ f_{rs} = f_{rt}$ and all but finitely many structure maps must be isomorphisms. It is important to note that the functor $C_n(\mathrm{VR}_{(-)}(X))$ is a persistence module, a collection of vector spaces for each real number ϵ, $C_n(\mathrm{VR}_\epsilon(X))$ and the linear maps the inclusions previously described $f^n_{ij} : C_n\left(\mathrm{VR}_{\epsilon_i}(X)\right) \to C_n\left(\mathrm{VR}_{\epsilon_j}(X)\right) = f^n_{\epsilon_j} \circ \cdots \circ f^n_{\epsilon_i}$. It is important to mention that we have a finitely amount of non isomorphisms because we have a finite amount of points.

[1] The reason we can do this is that the particular case where two elements image is the same happens when those features merge. We want to preserve the more persistent feature alive because it is more useful for later analysis but in the case both are equal it is irrelevant.

Moreover, the boundary maps are *morphism of persistence modules* $\eta_x : V \to W$ are a collection of linear maps and a particular case of natural transformations, $\eta_x : V(x) \to W(x)$ such that $\eta_y \circ f_{xy} = f_{xy} \circ \eta_x$ for each pair $x \leq y$. In our case, $\partial_n^j \circ f_{ij}^n = f_{ij}^{n-1} \circ \partial_{n-1}^i$ as we previously discussed. What is relevant is the fact that the chain groups of our sequence of Vietoris-Rips complexes generate persistence modules which are related by face maps.

Let $\{X_s\}_{s\in\mathbb{R}}$ be a set of filtered complexes with simplicial maps $f_{st} : X_s \to X_t$ for each pair $s \leq t$ such that $\forall r \leq s \leq t$, $f_{rt} = f_{st} \circ f_{rs}$. Its *persistent simplicial homology* with $\Bbbk$-coefficients is the sequence of persistence modules $H_n(X_s;\Bbbk)$ with structure maps $H_n(f_{st}) : H_n(X(s);\Bbbk) \to H_n(X(t);\Bbbk)$

Going back to our particular case, the set of complexes will be the set of VR-complexes $\{\mathrm{VR}_\epsilon(X)\}_{\epsilon\in\mathbb{R}}$ with the inclusion composition. Moreover, the persistent simplicial homology will be $\{H_\bullet(\mathrm{VR}_\epsilon(X))\}_{\epsilon\in\mathbb{R}}$ with the transformations induced by the inclusions.

It is remarkable that sequences of homology groups are persistence modules themselves: each $\{H_n(\mathrm{VR}_\epsilon(X))\}_{\epsilon\in\mathbb{R}}$ with the structure maps are the composition of the linear maps, $\sigma_{ij} : H_n\left(\mathrm{VR}_{\epsilon_i}(X)\right) \to H_n\left(\mathrm{VR}_{\epsilon_j}(X)\right) = H_n(f_{\epsilon_j}) \circ \ldots \circ H_n(f_{\epsilon_i})$.

Note that, roughly speaking, $C_\bullet(X_\epsilon)$ are "columns" that depend on ϵ while the persistence modules $C_n(X_\bullet)$ are "rows" that depend on n. Each column contains all the information from the simplicial complex generated with the specific ϵ value while each row contains all the information from the nth dimensional features.

3.4 Barcodes and Diagrams

All of this comes together into the *persistence diagram*, a multiset of points (x, y) in $\mathbb{R} \times (\mathbb{R} \cup +\infty)$. To be more precise, given a persistence module, its associated diagram has, for each pair $s \leq t$, the number of points (x, y) in the multiset such that $x \leq s \leq t < y$ is equal to the rank of f_{st}. Basically, the multiplicity of each point (x, y) is equal to the number of features that are born at x and die at y. A persistence diagram can be seen in the figure: 3.3.

Persistence diagrams portray topological information of the original finite metric set as a plot. Therefore, the main objective of persistence homology has been accomplished. Nevertheless, it is not all persistence homology has to offer, but the beginning of its utility. For starters, there is multitude of other ways to show the data analogous to the persistence diagram. One of those ways is through *persistence barcodes*. A persistence diagram is a multiset of points in $(x, y) \in \mathbb{R} \times (\mathbb{R} \cup +\infty)$, whereas a persistence barcode is a multiset of intervals $[x, y) \subset \mathbb{R} \cup +\infty$. Persistence barcodes are generated analogously to persistence diagrams. Furthermore, some authors even define one of the two objects through the clear isomorphism that gets the point (x, y) to the interval $[x, y)$ and vice versa.

We will now explain the correct way to interpret the different graphics of figure:3.3 to properly understand the two types of graphic and how to interpret the different topological features represented. Before starting, remember that if we calculate the

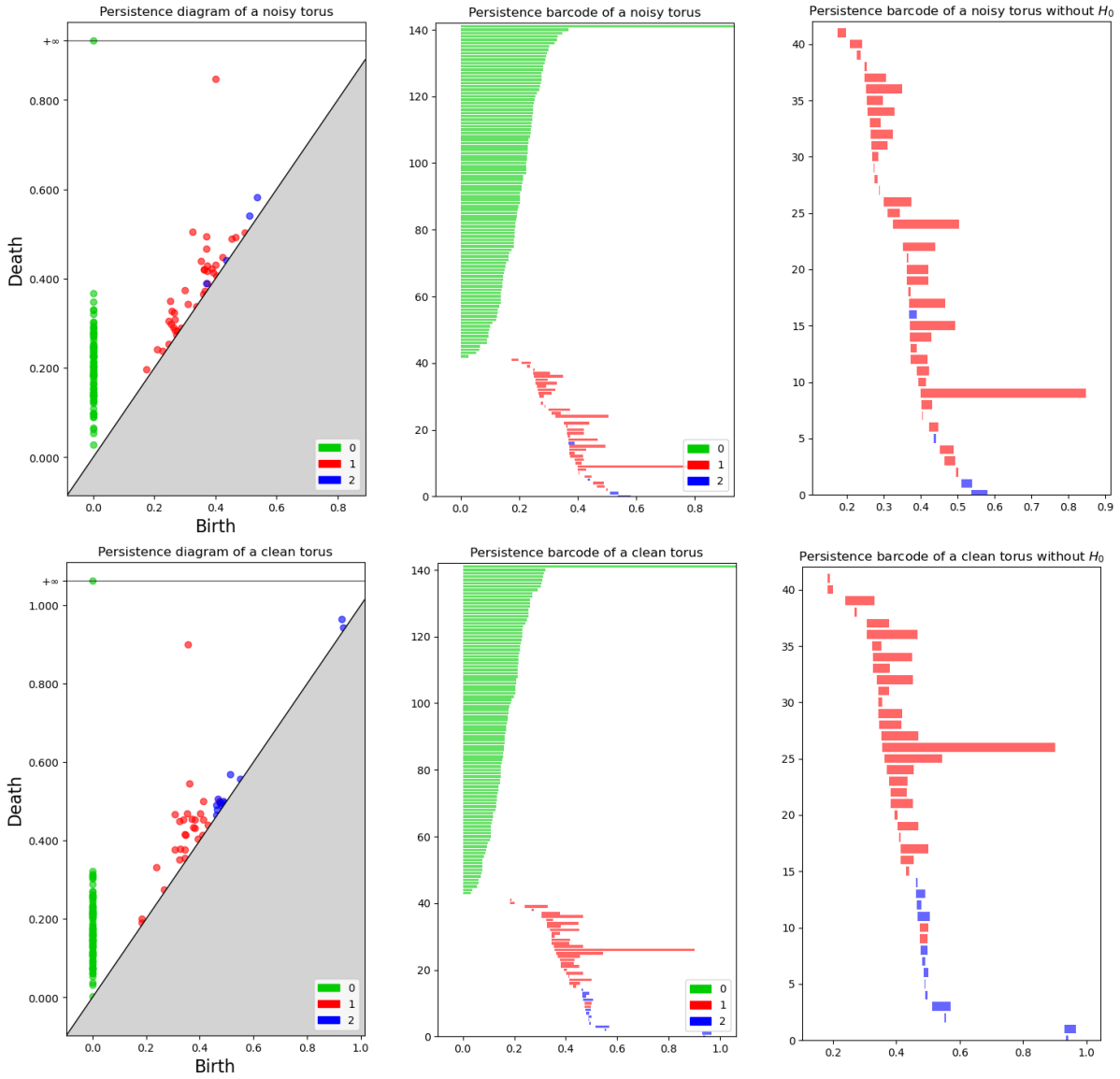


Figure 3.3: In the image the persistence diagrams and barcodes of two torus. The first row has some noise applied to its points while the second row has no noise at all. First column is a persistence diagram, second column is persistence barcode with all three calculated homology group but the magnitude of the 0-homology group makes it hard to see the other two. In consequence we included the third column, which are the diagram of the homology groups excluding the 0th.

homology groups as in section:2.3, the homology groups of the torus are: $H_0 \simeq \mathbb{Q}, H_1 \simeq \mathbb{Q} \times \mathbb{Q}, H_2 \simeq \mathbb{Q}$,

In the persistence diagram, each point stands for a topological feature that we have discerned through homology groups. The color of the point depict the dimension of the topological feature they represent and the dimension of the homology group they come from. Here green indicate path-connected components from H_0, red being one dimensional holes from H_1 and blue being two dimensional holes from H_2. The x-axis represents its birth, the first value of ϵ where the VR-complex manifested the specific topological feature, while the y-axis determine the last value of ϵ that manifested it. There are no points under the diagonal because the last value of ϵ that generates a complex with a certain feature needs to be bigger than or equal to the first value that generates a complex with that feature.

As we can see, all the 0-dimensional features are born at 0 (representing each an individual point) and they start to die as multiple points merge into bigger path-connected components. Observe that the span of a feature depends on how big the difference between the values of birth and death are. Therefore, a point being close to the diagonal represents a feature that is only expressed in a very specific scale and that are usually the least relevant features. On the other hand, a point being not close to the diagonal means that it represents a topological feature that is expressed in a big range of scales. In short, there is a direct correlation between the farthest away a point is from the diagonal and the relevance and prominence of the topological feature it represents. For instance, in our case the big hole in the torus is the most relevant feature and is represented by the isolated red point in the middle of the graph. It is remarkable that the presence of noise makes it less obvious but is one of the most intact features.

In the same way each point represents a feature in the diagram, in the persistence barcodes each feature is represented by a bar. Likewise, the dimension of the feature they represent is showed by color and we used the same color pallette for an easier comparison. Instead of expressing birth and death as two different coordinates, the persistence barcode expresses them by the values at where the bar starts and ends respectively. Bars are ordered by the value at which the start and, in case two or more bars share the same value of birth, those are ordered by the value at which they die. The barcodes from the third column share the same values as the ones from the second column but without the bars that represent 0-dimensional features. This way, it is easy to appreciate the behavior of 1 and 2-dimensional features without sacrificing the information of the original barcode graphs provide.

Similarly to our analysis of the diagrams, it is easy to see that all the green bars start at 0 and begin to disappear as components merge. Looking at the 0-components is useful to detect the amount of different path-connected components and how far away they are. If we then look at the third column and at the higher dimensional features, the relationship between the information conveyed here and that conveyed in the persistence diagrams is not so obvious. Instead of having to look at the distance to the diagonal to see the difference in value between birth and death, we just have to look at the amplitude of the bar: The longer each bar is the more persistent to scale changes it is. Now the big hole of the torus is represented by a bar that is quite larger than the

rest instead of a point that is farther from the diagonal. With that is now trivial that the distance from the diagonal in one representation is matched by the length of the bar in the other. From here it is easy to compare the 2-dimensional features in both graphs to fully grasp how they interact

As we mentioned, the persistence diagram allows for more information extraction than just the information displayed. For instance, it might be useful to compare the topological information of totally different sets through the persistence diagrams that they generate. To do that comparison, the *p-Wasserstein distance* is defined between two persistence diagrams X, Y as:

$$W_p(X,Y) = \inf_{\phi:X\to Y}\left[\sum_{x\in X}\left\|x-\phi(x)\right\|_\infty^p\right]^{1/p}$$

Where $||-||_\infty$ is the traditional L_∞ norm and $\phi : X' \to Y'$ ranges over all possible bijections between two arbitrary multi-subsets of $X' \subset X$ and $Y' \subset Y$ respectively and then adding $\emptyset$ to both X and Y. For elements $(x_1,x_2) \in X \setminus X'$ or $(y_1,y_2) \in Y \setminus Y'$, the value of $\left\|x-\phi(x)\right\|_\infty$ is defined as $\frac{|x_2-x_1|}{2}$. The reason we add $\emptyset$ to both sets is because, the way the distance is defined, some points will need to be paired with the empty set. If we look at figure: 3.4, those points are the ones that are paired with the diagonal and not another point (getting its distance as the distance to the diagonal). It is also important to remark that some authors define the Wasserstein distance as $W_p[L_q]$, giving the opportunity to change the used norm.

The distance defined by the limit $p \to \infty$ is known as the *bottleneck distance*. It is defined as:

$$W_\infty(X,Y) = \inf_{\phi:X\to Y}\left[\sup_{x\in X}\left\|x-\phi(x)\right\|_\infty\right]$$

The bottleneck distance measures, roughly speaking, the worst discrepancy in the best matching between two persistence diagrams. It is by far the most used distance function between persistence diagrams because it is the easiest distance to calculate. We can see on figure: 3.4 how we represent the bottleneck distance on top of the persistence diagrams of the two torus.

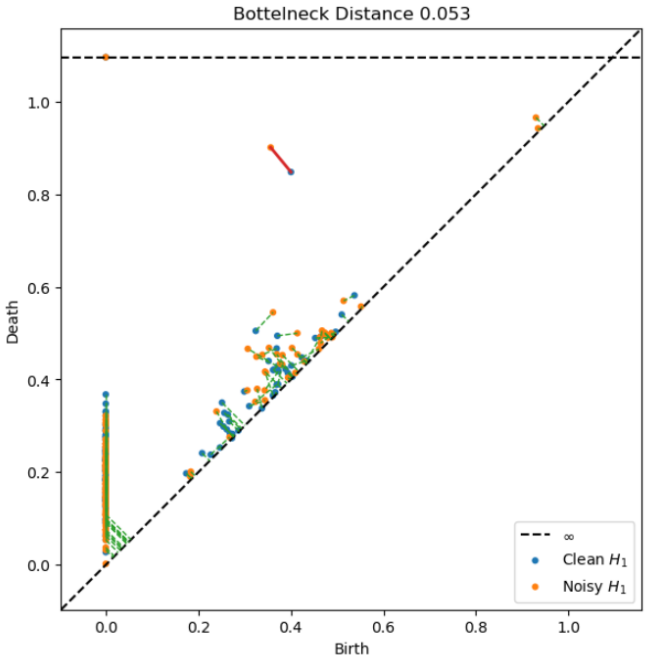


Figure 3.4: In the image a representation of the bottleneck distance between the persistence diagrams of the two torus seen before in 3.3. Both persistence diagrams can be seen represented by the different points. Green segments group together the points that are grouped together. Notice that some points are grouped with the diagonal, those are the ones that would mathematically be paired with $\emptyset$. Finally, biggest bottleneck distance is represented as a red segment between the two more distant points (the 1 dimensional holes)

CHAPTER 4

COMPUTATIONAL APPLICATIONS

In the end of the previous chapter we explained the two most basic applications of persistence homology: the representation of the homology groups up to interpretation and the comparison of those same features between different objects (or the same object with noise).

In this chapter we will show some more complex applications of TDA, mention some alternatives to the options discussed in chapters 2 and 3 and how these alternatives are used to previously prepare the data. Therefore improving the extraction of topological information that will later be transformed, filtered and cleaned to be finally displayed or used in one of a multitude of ways. However, this time we will focus on the practical aspect of TDA and we will peek into some packages and how they interpret the different definitions we saw. All of this will be illustrated by the construction of a torus using the Vietoris-Rips complex as a showcase of the theoretic example in Section 2.3. To further understand the computational representations, this familiar and simple case will be used to explain how some packages operate the different topological objects that we previously defined: how they are constructed and how the different operations are computed.

Packages overview

The TDA packages we will be explaining and working with are *GUDHI 3.7.1* [7], *Giotto-TDA 0.5.1* [8] and *Ripser 1.2.1* [9] while also using *kepler-mapper* and several other more widely used and generalist packages like *NumPy*. *GUDHI* is the state of the art, it is implemented in C++ 14, though we are using it through its Python bindings provided by the same team. It is open source under the MIT license[1].

While *GUDHI* provides low-level functionality with a wider range of options, the majority of packages are specialized into narrower, particular applications. For instance,

[1] many modules have GPL3 and LGPL dependencies

Ripser is built exclusively for Vietoris-Rips persistence computation and outperforms other generalist implementations (like *GUDHI*) in computation time and memory efficiency. It is also licensed under the MIT license and is written entirely in C++. On the other hand, *Giotto-TDA* is highly adapted towards machine learning, with a tight integration with *scikit-learn* and being optimized with regards to batch processing. It differs from the other two packages in that it is part of the *Giotto* family of open-source projects, distributed under the GNU AGPLv3 license and developed in C++ but exclusively bounded to Python.

The reason behind using these packages and not others started behind the intention of using *Giotto*: the possibility of implementing TDA in conjunction with machine learning, the full bindings in Python and the quality of the documentation made it look a lot more attractive than the alternatives. Once we started diving into the package and starting comparing it to other packages we discovered that it had dependencies to *GUDHI* and *Ripser* and the importance of these packages in the state of the art. Therefore, we thought it was wise to cover them too.

Similarly to how a sequence of functors and evaluations had to be applied to a set of points to obtain the definition of persistence diagram, these packages utilize a sequence of functions, classes and objects to transform the input data into a usable resource for further computation or a clean representation of the topological features. Like many algorithms, TDA heavily relies on the use of matrices to optimize computation time and memory usage.

After properly explaining the general idea of the implementation and how to express and work with the different theoretical concepts, we will review some examples and explain some functions, how different packages treat the data and the generated topological objects. It is useful that different packages have different implementations of the same theoretical concepts. After explaining how some of the packages work, we will show some examples to tie all previous knowledge together and consolidate the understanding of the process.

4.1 Data treatment, background and implementation details

Since the construction of a VR-complex depends solely on the distance between points, the computation of a VR-complex through a distance matrix is quite trivial: given a maximum threshold ϵ, if an element of matrix satisfies $x_{i,j} < \epsilon$ then it means that the ith and jth points are connected in the resulting complex. It is remarkable that in the construction of the distance matrix, the order in the set of points is necessary once again.

However, the way packages store the information of a simplex is not trivial and there are alternatives to the VR-complex that are not as straightforward. For instance, `gudhi.RipsComplex` objects are 1-skeleton graphs formed by the edges of length less than the given threshold. When creating a simplicial complex from this graph with `RipsComplex.create_simplex_tree()` , the class first builds the graph and inserts it into the data structure. Only then it inserts the different simplices that correspond to the faces of the complex. The result of the operation is an object of type

`gudhi.SimplexTree`, which stores the data as described in [10]. In short, all faces of the simplicial complex are explicitly stored in a trie whose nodes are in bijection with the faces of the complex.

Nevertheless, the input will not always be a distance matrix nor it will be feasible to compute the generation of the complex through VR. Hence the need for alternatives to VR-persistence and for transformation tools that let us convert a series of different inputs into more desirable formats for improved computation and wider application. This includes alternatives to VR like the *Alpha persistence*, the *Witness persistence* [11] and derivatives of these three in the case of having a point cloud input; *Flagser persistence* in case of having a weighted adjacency matrix generated from a graph input (or a point cloud input transformed into a graph); or cubical persistence used on image inputs. On top of all of this, one could also apply VR to a distance matrix created from a time series data set using the Pearson correlation coefficient. Clearly the amount of possibilities is overwhelming, proving the flexibility and utility of TDA. We won't prove the efficiency, robustness or accuracy of these functions, since this would be outside the scope of the project. Nevertheless we will give a brief summary and references for some of them.

For instance, the *Alpha persistence* and the *Witness persistence* use the finite cells of the *Delaunay triangulation* to generate a simplicial complex and calculate its homology. As with Vietoris Rips, Alpha complexes are equivalent to the Čech complex topologically with an unbound radii, making all the statements in previous chapters true in that case. However, the Alpha complex can be computed efficiently only in very low dimensions since in high dimensions it requires a threshold radii to be efficient and, in that case, there are no good stability theorems. As a result of these inconveniences, the *GUDHI* implementation of this algorithm comes with the parameter `precision = 'safe'` that can be `"fast"`, `"safe"` or `"exact"`, defaulting on `"safe"` for a middle ground of fast computation and a guaranteed small multiplicative error [2]. Similarly, the *Witness persistence* lacks any good stability theorem and can fail to reconstruct the homotopy type even in simple examples [12] making this algorithm only useful in conjunction with standard statistical tools.

Due to the lack of a good high dimensional alternative, the focus of the research returned to Vietoris Rips to find some improvements in the efficiency of the algorithm, using the *cochain complex*: the dual notion of chain complex. Cochain complexes have a very similar structure to a chain complex with the difference that, in cochain complexes, boundary maps increase dimension instead of reducing it. Applying homology to a Cochain complex results in what is called *cohomology* and, even if the premise might seem more complicated, is more natural than homology in many applications. For instance, the computational cost of calculating persistence cohomology is way lower [13]. Moreover, algebraic relationships between the persistence modules of the two have been established, resulting in ways to compute the same results more efficiently.

This and other discoveries resulted in *Ripser*, which also uses computational short-

[2] The documentation in [7] goes into detail and is reccomended as a further reading for *Witness complex* and other methods

cuts and chain complex properties to improve the computation time by a factor of more than *40x*. Moreover, it also improves the memory efficiency by a factor of more than *15x*, hence creating the fastest and most memory-efficient implementation for the Vietoris-Rips barcodes and diagrams. However, part of the computational shortcuts make it impossible to retrieve the simplicial complex or other intermediate information.

In the same vein, *Giotto* is focused on functions that directly transform a point cloud or a distance matrix into a persistence diagram represented by triplets, trading the ability to acces some intermediate objects for efficiency. These resulting triplets are conformed by two `float` values and an `integer` that represent (respectively) the filtration values at which the feature appears and disappears (birth and death) and the dimension of the feature. For instance, a H_1 feature that is born at $\epsilon_i = 0.17$ and dies at $\epsilon_j = 1.43$ would be represented by the triplet `[0.17, 1.43, 1]`. *Giotto* is organized through classes like `gtda.homology.VietorisRipsPersistence` that are generated with parameters that focus on the distance functions and the limitations it needs for optimization. Therefore, the object created is a computational function that is akin to a functor like $\mathrm{VR}_\epsilon(-, d)$ that can be applied to different point clouds.

On the contrary, *GUDHI* is oriented towards generating individual objects that represent simplices of the form $\mathrm{VR}_\epsilon(X)$. The main difference is that there is a middle step that generate a data object that represents the simplicial tree and can be converted into an array that represent the persistence homology multiset that is used to plot persistence diagrams and persistence barcodes. To better understand this we will later look into the *GUDHI* class in 4.2.

The main reason behind this difference is that *Giotto* intends the user to generate one way of computing a simplex from a given value and then apply that new function to a multitude of objects and in a very efficient way (remember that it is tailored towards machine learning) and directly computes persistence diagrams. If the reader has any experience using the *scikit-learn* package will notice the similarities with the *scikit-learn* models: *Giotto* is build with the idea of applying TDA in mass to data to adjust estimators that rely on topological features. On other words, it tries to be a topological version of *scikit-learn*.

It is also remarkable that the reason that *Giotto* resembles *Ripser* is because the first uses the second as a backend with modified bindings to optimize Vietoris-Rips persistent homology (it also uses *GUDHI* and other TDA packages backened) [3].

The combination of the flexible approach of *Giotto* with the usage of very optimized code for the specific most common tasks results in an adaptable package that sacrifices low-level functionality for efficiency and compatibility with more inputs and outputs. Moreover, the package also includes an extension to `sklearn.pipeline` that, with the classes of the package that are oriented towards creating particular functors and functions, allows the user to generate custom sequences of transformations easily. Eventually, we will see that this is not only great at cross validating and hence machine learning: it is one of the reasons why *Giotto*'s Mapper implementation is so user friendly. Furthermore, we found in *Giotto-TDA* one of the widest range of supported input

[3] https://giotto-ai.github.io/gtda-docs/0.5.1/modules/homology.html

formats: Most packages support, directly or indirectly, inputs in the form of distance matrices, point clouds and graphs. On top of that, *Giotto-TDA* has tools that support images and time series, in addition of multiple ways of processing the inputs, filtering and representing the extracted information.

Giotto-TDA offers the user ways to extract distance matrices from geodesic distances from graphs. If the input is a point cloud instead, there are multiple ways to convert it into a graph, directly into a persistence diagram or into a distance matrix. Similarly, if the input is a set of time series, it could be transformed into a point cloud or into a distance matrix. There are multiple options to transform the resulting matrices into a persistence diagram with the possibility of limiting homology dimensions or edge length to improve on computation time. Moreover, the modularity of the package allows us to mix combinations in any way that makes sense, and most of the options have a wide range of parameters that allow us to change things such as the metric used, the weights assigned between points and more.

As a final note before continuing with the examples, the same amount of options and flexibility that exists for the process of transforming an input into a persistence diagram exists after the diagram is created and towards its representation and usage. Usually the diagram is first filtered and scaled, discarding the features that exist for a very short period of time since they are generally irrelevant and applying a basis change that converts the diagonal into the x-axis. Since the lower triangle provides no information, using that space in the diagram results in improved readability. After that, the information can be presented as is, as a landscape curve, a Betti curve or as a vector for machine learning purposes.

Summarising: *GUDHI* is the state of the art, a generalist package that is one of the more complete and optimized packages available. We recommend it for general TDA research and use. *Ripser*, on the other hand, is a specialized package that is the most optimized way of calculating VR-persistence. It is advised to use it to do VR-persistence in conjunction with other packages that don't have it integrated (as *Giotto* does). Finally, *Giotto* stands as a middle ground between the other two packages: it is focused in a very specific direction (usage towards and in conjunction with machine learning) but it provides a general usage case with a big fan of options like *GUDHI*.

It is worth mentioning the existence of other packages and further reading. The main competitor to *GUDHI* is *DIPHA*, since those are the two best implementations after *Ripser* and both are very complete and flexible packages. Similarly to how *Giotto* covers a niche in the TDA branch, some other specialized packages like *Kmapper*, *JavaPlex*, *JHoles* and *Dionysus* between others exist. These packages might do more than one thing but all of them cover some specialised niche in a different way than the rest of the packages. For further reading, [14] is a thorough comparison among multiple packages.

4.2 The torus

In section 3.4 we remade the original theoretical example we did in section 2.3 , calculating and representing the homology groups, but in a practical way. The persistence

diagrams examples are a result of applying persistence homology to sets of points in the surface of a torus (with and without noise). The geometric interpretation of practical examples born from discrete data sets, the code and computational operations are so different that it can be hard to perceive how it all comes together.

As a first coding example we will show and explain the code that we use to generate those torus and the associated persistence diagrams as well as a visualizing tool that displays the VR-complexes generated from those torus at different scale values. We start with this example because we have already looked at the torus from a more theoretical perspective before in section: 2.3. Hence, this is the third look at the homology groups of the torus, the persistence homology technique and the combination of the three examples should improve the overall understanding of all of them.

We represent some of the complexes that are generated to calculate the homology at distinct values of ϵ. Hopefully, this graphical interpretation helps understanding all those aspects and makes an easy introduction into the way the packages work by showing how different aspects of the package are used, bringing a better comprehension of the overall process.

In both examples, use the TDA package *GUDHI* in conjunction with *Numpy* and *Plotly* for ploting the result. To generate the persistence diagram, we generate a discrete set of points over the surface of the torus that we worked with in the examples. In the code below we see the extract where we generate the noisy discrete set. We convert the data set into a `gd.RipsComplex()` object and we apply to it the `.create_simplex_tree()` method of finally calculate the persistence diagram with the `.persistence()` method to the resulting simplex tree.

```
"""
The code converts the points ino a Gudhi.RipsComplex data structure, that is
↪ similar to a one skeleton graph
Then it converts that one skeleton graph into a tree of simplices that represent
↪ the directed simplicial complexes.
We will limit the Max_dimension=3 restriction to ensure only the homology groups
↪ of dimension 3 or lower (and accordingly the simplices of dimension 3 and
↪ lower) are computed. This greatly improves computation.
Finally we transform the directed into the multiset that represents the
↪ persistence diagram.
"""

    data1 = make_point_torus(n_points = 100, noise = 0.3)
    rc1 = gd.RipsComplex(points = data1)
    vr1 = rc1.create_simplex_tree(max_dimension = 3)
    vrp1 = vr1.persistence()
```

In the code it is easy to appreciate the most significant *GUDHI* features: it is heavily object-oriented, treating each intermediate object as its own class with its own functions. This can be seen as it treats Rips Complexes as a different, intermediate structure

from a simplex tree. From here we can use some plotting functions from *GUDHI* or any other package. In our case, we use *plotly* to plot the diagrams from figure: 3.3.

The `RipsComplex` class is just nearly exclusively used to create a `SimplexTree` object that has a lot more broad use. In the previous example we used it to generate the persistence diagram and in the next example we will use its `.get_skeleton()` feature to extract some features from the 2-dimensional skeleton of the complex. The function returns a generator of the 2-dimensional skeleton but with the NumPy package we can do:

```
1    np.array([s for s in st.get_skeleton(2)])
```

With this we would get an array of the elements of the skeleton. We can extract information from all simplices because every element of the skeleton is a list with the points in the simplex and the filtration value (the ϵ) where the simplex is generated. Note that the order in the set of points is crucial to the way the package works with the points, referencing each point in the data set and in the complex just by its order number.

With all of this in mind, we put some conditions to only get the 2-dimensional simplices (otherwise the 0 and 1 dimensional simplices would have been extracted too) and we also put it so that we only get the simplices that are generated at a scale value of less than or equal to 1. We chose this arbitrary number because the torus over which we generate the points has a distance between the center of the tube and the center of the torus of 0.5. Therefore, with a filtration of 1 the torus will be fully covered and plotting even more simplices does not make much sense. The resulting plots can be seen in the figure: 4.1

```
1 points = make_point_torus(200)
2 rc = gudhi.RipsComplex(points=points)
3 st = rc.create_simplex_tree(max_dimension=2)
4 triangles = np.array([s[0] for s in st.get_skeleton(2) if len(s[0])==3 and s[1]
  ↪  <= 1])
```

The code can be found entirely in the appendix A.2. The resulting array will not have the coordinates of each point but just the index of the points from the original `np.array` that conform the 2-simplex. In the full code it can be seen how we use this to get the coordinates of each point and we even do this dynamically to get the triangles that conform the VR realization of the torus at different values of ϵ and then the resulting `np.array` is used to plot those triangles and see the torus that is generated.

4.3 Mapper

As we have just seen, visualizing high-dimensional data and interpreting the topological data analysis is not an easy task. The example we just saw succeeded just because we used a lower dimensional, easy to understand example. In more practical examples this will not always be the case. Therefore, we believe it is necessary to introduce a tool that,

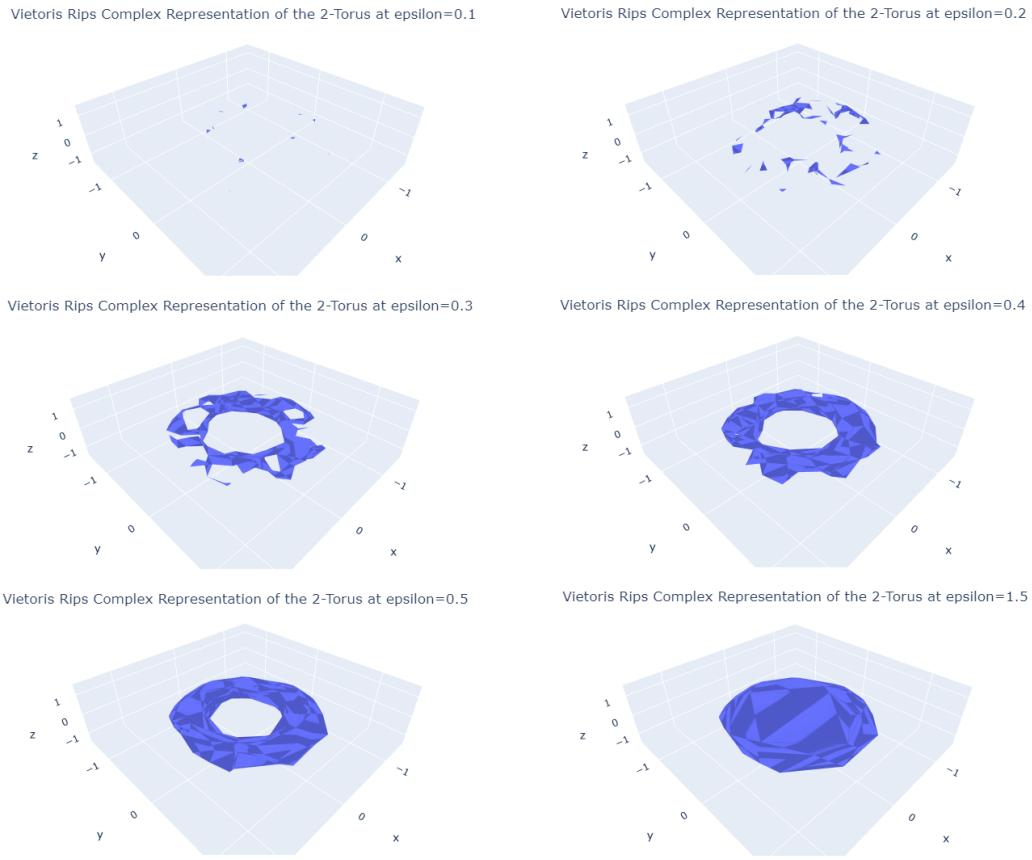


Figure 4.1: In the image we can see how the VR-complex of the torus evolves at different values of ϵ and it's representation.

once TDA discovers persistence homology features, it gives the user some visualization options. Therefore, we introduce *Mapper* [2], that was made with the objective of finding a computational method for extracting and expressing basic information as simplicial complexes using topological methods.

The Mapper algorithm is a TDA technique that is based on the idea of partial clustering. It combines dimensionality reduction, clustering and graph network techniques. One of the best features it presents it that it is not strictly tied to any particular dimensionality reduction or clustering algorithm. We will provide a rough explanation of the theoretical framework of how it works, focusing more closely on how it can be used and providing a code example.

Similarly to how Persistence Homology is based around the use of VR and Čech complexes to have a good sense of structure, Mapper uses the *nerve* complex instead. As we have seen in section 3.1, there is a fundamental relation between the VR and the Čech complex, which is a particular case of nerve complex. This relationship in the structure on which these two algorithms are built is fundamental in the way they complement each other: Persistence Homology is a great tool for extracting information about the topological features of the data, but that information alone might be too hard

to decipher making it impossible to portray the general structure of the data. Mapper starts with the same idea: converting the data into a simplical complex through the use of the nerve. Even so, Mapper is focused on portraying the general structure and shape of the data. The way this is achieved is through the usage of the nerve complex in conjunction of a variety of more classical techniques in data analysis.

The Mapper algorithm starts by reducing the dimension of the data (through what is called a lens function), then it constructs an overlapping cover over the resulting reduced dimension space and applies clustering to the preimage of the lens function of each set of the cover. The resulting clusters form a new cover over the original space that will overlap thanks to the overlap of the original cover. Applying the nerve complex to the set of resulting clusters, and the resulting simplicial complex will have edges between the clusters that share any points. The resulting simplicial complex can be geometrically realized, resulting in a compressed representation of the dataset, which maintains the structure fantastically.

Before explaining the particular differences of some packages, we will go over an outline of the different steps. These steps are the same in all Mapper algorithms even though, as explained, each step has a lot of room for flexibility.

- The first step is projecting the data into a lower dimensional space using a *filter function* or lens. These functions are usually standard dimensionality reduction algorithms like PCA, MDS, UMAP, t-SNE, Isomap or even density-based methods like distance to the first k-nearest neighbors.
- Then an overlapping cover of the projected data is fixed. Usually it is done by using n-dimensional cubes with constant length that overlap in each dimension.
- For each interval, we apply a cluster algorithm to its preimage. Any technique can be used but the most usual are k-means, hierarchical clustering and DBSCAN.
- Since the cover we created had some overlapping, there should be some points that appear in multiple covers and, therefore, in multiple clusters.
- Finally, the resulting cluster is plotted. Thanks to its construction, the result is a graph that can be plotted as an image but also could be plotted as an interactive application that let us move the dots respecting the topological interactions. All the usual plotting options are available including the possibility to use coloration dependent on an outer function.

The overall process is modular, having a variety of options and paths in each step that are all compatible, generating a huge amount of possible applications. This amount of choice makes it so it is very likely that there is a combination that fits the data. However, finding the best performing combination for each case is a really hard task that makes it show the user capabilities and knowledge of both the data and the algorithms.

We will see how both *Giotto* and *Kmapper* implement the Mapper algorithm. We already talked about *Giotto* and as usual it implements the Mapper algorithm by extending some of the *scikit-learn* classes and using pipelines. Once the user decides which options to use in each step, which lens function to use, which cover to generate and the de-

sired clustering algorithm, it makes a pipeline using the `gtda.mapper.make_mapper_pipeline` function that generates a new function that applies all the steps automatically.

Kmapper, on the other hand, is a package that is oriented solely towards the representation of data. We will not go into too much detail regarding the package's operation. It was included to showcase some of the particularities of *Giotto-tda* by showing a more traditional Mapper package. Due to its orientation towards machine learning, most of *Giotto*'s functions are built with the purpose of being applied a large number of times to many different datasets. *Kmapper* and most other Mapper algorithms are instead tailored towards representing individual huge datasets, with each use of Mapper using completely different assets. Where *Giotto* prioritizes defining a Mapper function with set options, *Kmapper* goes for a more complex representation of the data and an interactive application.

4.3.1 Applying Mapper to a classical dataset

As a first example, we will tackle a classic problem of data analysis: the Iris dataset. We will apply Mapper with the most general clustering options towards one of the most well-known datasets to compare it to the usual results with classical tools. With *Giotto* we create a pipeline with PCA [4] as a filter function, a cubical cover, and the `DBSCAN()` cluster method [5] and we apply it to the dataset.

```
# We will use PCA with 3 components as our lens function for simplicity.
filter_func = PCA(n_components=3)
# The cubical cover is the default option due to its simplicity.
cover = CubicalCover(n_intervals = 10, overlap_frac = 0.30)
# We define DBSCAN() as our cluster algorithm.
clusterer = DBSCAN()
# Initialise pipeline.
pipe_iris = make_mapper_pipeline(
    filter_func=filter_func,
    cover=cover,
    clusterer=clusterer,
    verbose=True,
    n_jobs=n_jobs,
)

fig = plot_static_mapper_graph(pipe_iris, iris, color_data =
↪  datasets.load_iris().target)
fig.show(config = {'scrollZoom': True})
```

[4] PCA is the default option in the field because its simplicity but other options might be more appropiate for specific datasets

[5] DBSCAN stand for Density-based spatial clustering of applications with noise, and its name is self explanatory. On top of being a great clustering algorithm overall, its strengths are specially well fitted for a large number of small scale sets to cluster as in Mapper.

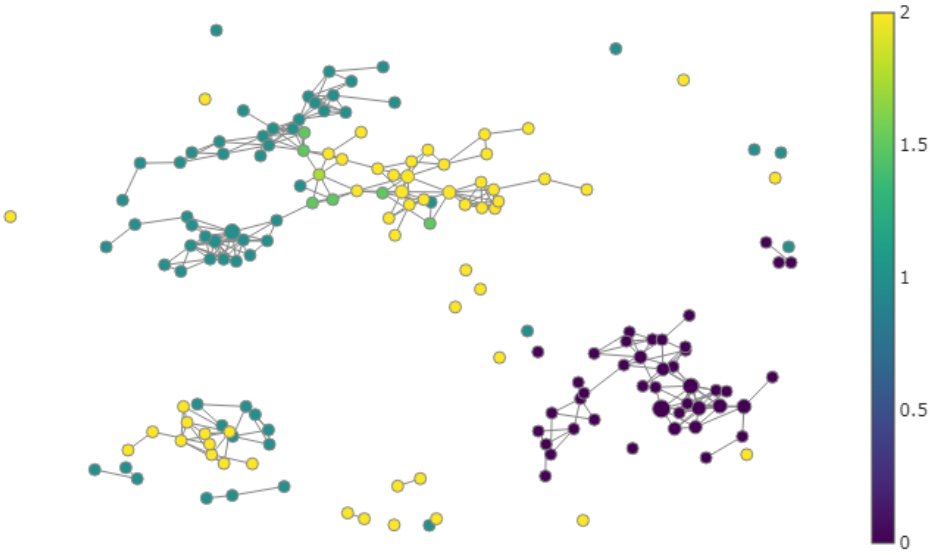


Figure 4.2: In the image we can see how the Mapper Algorithm plots the Iris database and has some of the same problems as usual statistical techniques.

In the code it is easy to see how we define each of the important functions that we went over in the outline and use a pipeline to tie them all together and apply it to the dataset for plotting. The resulting representation can be seen in Figure 4.2.

The results are quite similar than to normal techniques: It is quite easy to see that of the three types of iris flower, one is easily distinguished while the other two variants are quite similar. There is no extraordinary insight over the data, mostly because the topological structure of the 2 similar types of flower are close in the parameters that are used and also because the reason that this specific dataset is used as a practice for beginners is that it is quite easy to extract nearly all extractable information with classical tools [15].

With this first example we did not choose an example where Mapper does surpass classical techniques. Instead, we first started showcasing an example with very similar performance to classical tools and the most basic options to showcase how it represents a result that should be familiar to the user. Finally, we will follow up with an easy artificial case before continuing with a real example of Mapper usage for research.

4.3.2 Mapper applied to a discrete set of points in the surface of a torus

Everyone with some experience with clustering or general data analysis knows that working with circumferences with most of the usual tools and methods is painfully hard for even the most easy examples. As a recurring example of the TDA utility, we will showcase the Mapper algorithm applying it to a set of points in the surface of the torus.

The full code can be seen in the annex A.3, though the only notable difference with

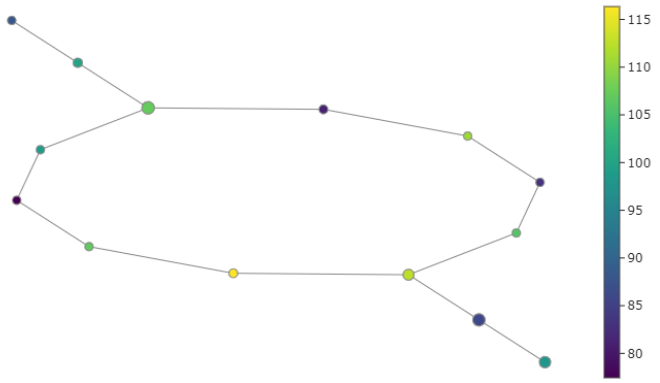


Figure 4.3: In the image we can see how the Mapper Algorithm percieves the circular nature of the Torus and gives insight of its most important feature: it's central hole.

the code used with the Iris dataset is that this time we used a mere projection of 1 of the axis as a lens function. The reason we used this oversimplified option is to showcase that even with the most basic assets, Mapper successfully showcases the most notable topological features. In a nearly identical way, in the *Giotto* documentation an example of Mapper applied to concentric circumferences can be seen [6].

4.3.3 Mapper applied to cancer research

By now it should already be clear that Mapper is an algorithm that can be used to showcase similar insight as the classical tools while also being able to easily showcase non-linear features (like circular structures) where most traditional techniques will have a hard time. It is obvious that, once applied to very complicated datasets, Mapper might reveal some insight that was oversight with the more usual tools. The next example uses *Kmapper* to apply Mapper to a cancer dataset. As always, full code on the Annex A.4:

```
    # Create a custom 1-D lens with Isolation Forest
    model = ensemble.IsolationForest(random_state=1714)
    model.fit(X)
    lens1 = model.decision_function(X).reshape((X.shape[0], 1))

    # Create another 1-D lens with L2-norm
    lens2 = mapper.fit_transform(X, projection="l2norm")

    # Combine both lenses to get a 2-D [Isolation Forest, L^2-Norm] lens
    lens = np.c_[lens1, lens2]

    # Define the simplicial complex (similar to how Giotto uses pipeline)
    scomplex = mapper.map(lens,
```

[6] https://giotto-ai.github.io/gtda-docs/latest/notebooks/tmp/mapper_quickstart.html?highlight=mapper

```
                        X,
                        cover = km.Cover(n_cubes = 15, perc_overlap = 0.7),
                        clusterer = sklearn.cluster.KMeans(n_clusters = 2,
                        ↪  random_state = 3471))
```

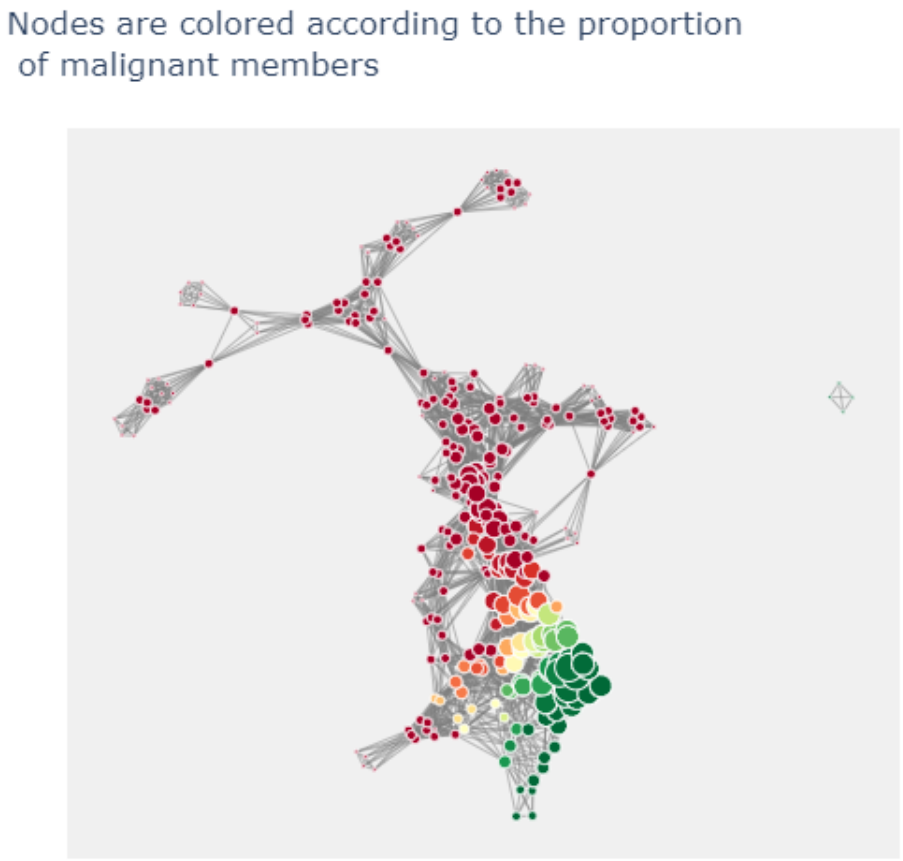


Figure 4.4: In the image we can see a representation of a database of cancer patients in Wisconsin

As we can see, the code uses the `mapper.map` function in a very similar way as *Giotto* uses pipelines. The main difference is that it takes the data as parts of the inputs and the output is the resultant graph instead of outputting a function that can be used with data to generate graphs. It can also be seen that we used a much more complicated lens function, getting a two dimensional representation of the point where one dimension is the L_2 norm (the euclidean norm) and the other the resulting classification of a random isolation tree.

The reason we chose this specific lens is quite enlightening: The lens that is based

on the euclidean norm disperses the points instead of clustering them. This might be counter-intuitive but doing this, once we do a cover of the result, it will better distribute the points in the different subsets. Moreover, having this dispersion given by one of the lens, we can focus on a value that makes biological sense and highlights special features from the data. For instance, the anomaly score given by the isolation tree makes a lot of sense studying cancer cells that mutate.

The resulting plot can be seen in 4.4 and it gives a good amount of insight of how to approach the data. It is important to remark that evidence in the field of cancer research points to cancer being born from one cell that mutates. All research is focused with that in mind: any cancer started as one cell that mutates in some way or another [16]. Having that in mind, it is quite easy to see that the green portion (where most cases are benign) indicates cancers that mutated the less while red dots represent malign cancers usually related to more mutated cases. The geometrical form of the data indicates us the different ramifications that should be tied to different mutations of the cancer. Doing a cross validation between the items in each section or branch and the result of other classical techniques could end in finding a type of cancer that has not been properly classified.

The utilization of the Mapper method not only results in a better understanding of the general structure of the data. In some cases the resulting plot will not match previous results and a further inspection might reveal possible improvements of the classifications of the data. This was the case in [2]: After applying Mapper, the resulting plot showed a divergence in the sample that did not mach with the insight provided by other techniques.

Further inspection revealed that Hierarchical clustering failed to identify this subset of samples, separating them into different clusters with low confidence (See figure 4.5. Moreover, those tumors do not correspond to any previously reported cancer expression subtype. Additional research and validation confirmed the hypothesis: it was a new class of tumors and they were discovered.

4.4 Summary and other applications

In this chapter we showed how Mapper compares to classical clustering in basic cases, how it gives insight over complicated databases, how it succeeds at analyzing non-linear structures and how it can identify groups that cluster analysis fails to spot. Moreover, we used and explained the basic structure and usage of two different packages.

We mostly focused on *Mapper* and, to a lesser extent, on *Persistent Homology* but TDA has a lot more applications than the ones we saw here. For starters, those same methods can be applied to image processing, to time series or even to graph theory, giving amazing and unique results. Moreover, it is worth recalling that the main attractiveness of *Giotto-tda* is the orientation of the package towards Machine Learning.

The main reason behind it is that, whereas *Mapper* was a hard tool to use but provided great, easy to comprehend results and insight, *Persistent Homology* is just the opposed. Once one understands the basics behind it is really easy to use and apply to data sets, but with the coveat that the resulting insight can be hard to interpret or even

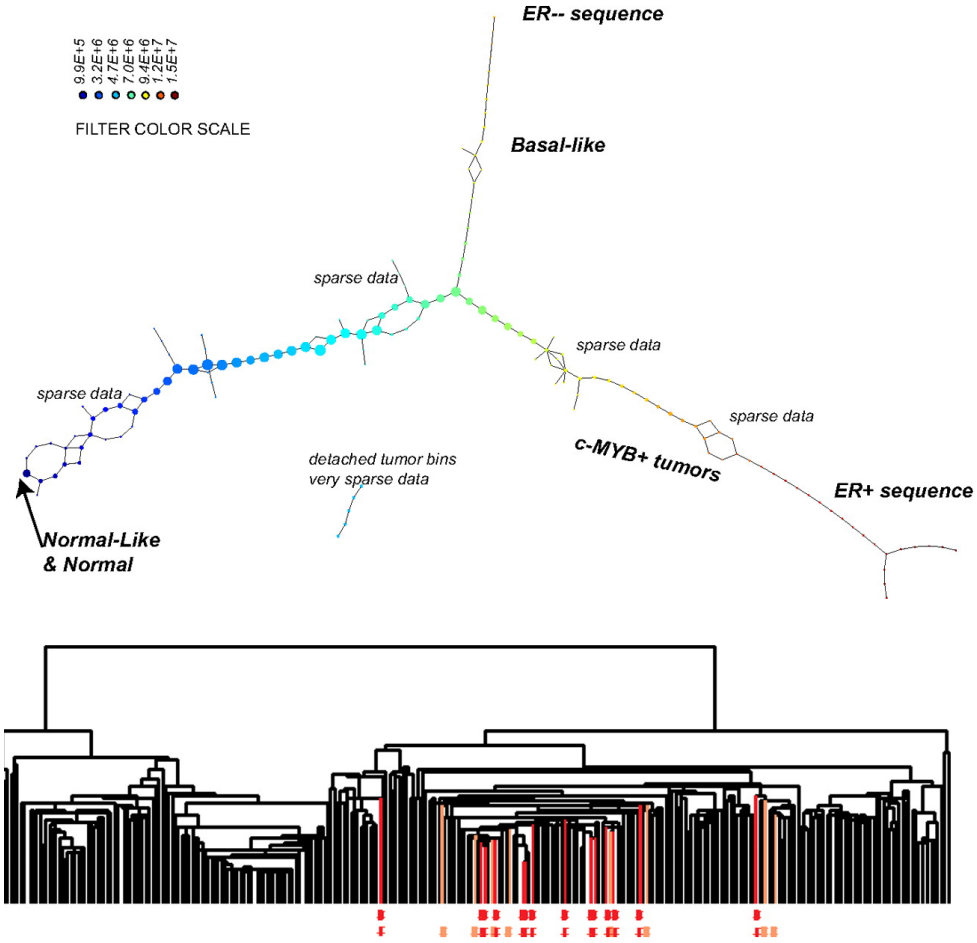


Figure 4.5: In the upper image we can see the Mapper representation of the dataset. In the lower image, bins defining the c-MYB+ cancer group were colored (red for thigh no outliers group and orange for larger group containing outliers). It is easy to see that classical clustering does not succeed at classifying this data and Mapper gives great insight. Source: [2] From Monica Nicolau, Arnold J. Levine, Gunnar Carlsson, Proceedings of the National Academy of Sciences.

comprehend. However, through Machine Learning, one could use that data without the need of human interpretation.

Giotto it is fully integrated with *scikit-learn* for that reason: once we apply persistence homology to a set, the resulting persistence diagram is usually treated and filtered through a *Persistence Landscape* or a *Betti Curve*. All of this results in the extraction of the main features of the set as a vector and then finally using it together with Machine Learning to get an estimator. In short, for every set we express its topological features as a vector with the usage of *Giotto* and then we apply machine learning techniques with *scikit-learn* to that vector for proper classification of shapes through topological features and machine learning.

This is just another example of the endless possibilities of Topological Data Analysis. However, we didn't include any application using it for the same reason we didn't cover some *Mapper* modifications for better covering: they start to get out of the introductory scope of the project and we needed to stop somewhere.

CHAPTER 5

CONCLUSIONS AND FURTHER READING

The main goal of this project was to introduce the reader into TDA, explaining the *Persistence Homology* and the *Mapper* algorithms. Furthermore, we wanted to explore the possibilities of this algorithm and the state of their implementations as well as understanding how and when to apply them. In order to achieve all of this, we needed to start explaining the theory where this algorithms stem from.

Consequently, we start this project in chapter 2 introducing the basic bricks that are used in the different steps of the process: Simplices, chain groups and homology groups. We explained all of this objects explaining their properties, how to construct them, how to use them and the properties that present that are useful. We ended the chapter by summarizing the relationship that there is between the different objects introduced and more importantly, proving the functorality that provides our algorithms of flexibility and strength.

To follow up, in chapter 3, we explained how we can use what was already introduced to create specific and convenient objects from our data in the *Vietoris Rips* and *Čech* complex. Moreover, we also showed how those creations might contain useful information through the Niyogi-Smale-Weinberger theorem [1] and how *Persistent Homology* extracts and represent that information.

Last, we got to the chapter 4, where we finally got to the centerpiece of the work: a study of the computational applications of the most relevant TDA algorithms. First, we reviewed some of the prominent and (to our opinion) more interesting packages of the scene, going over their actual status, their main features and the differences in their implementation.

We then used the packages to make an interactive tool to help understand what the package did through an example that should be familiar to the reader. Then we introduced the *Mapper* algorithm and used it in a myriad of examples.

Overall, it should be well established by now that these two algorithms are trustwor-

thy and reliable. As we showcased with the examples, there is great utility in representing the topological shape of the data as well as numerically researching the topological features of the data. Both of those options bring insight and a unique point of view towards the study and research of the data that, through the usage of advanced algebraic topology techniques, gives a way more natural understanding of the data.

Even then, it also true that it is a really young branch of the Data Analysis and there is still a lack of knowledge to understand when it is worth it to use these tools over the classical ones. Additionally, persistent homology requires a good understanding of mathematics to even comprehend the resulting information that they provide, making it really hard to use for researchers from other areas.

As a result, most packages and applications have room of improvement or are still in development. Moreover, the computation times with big inputs and high precision are in most cases unpractical. However, there already are ways around most problems and constant research is being made. As mentioned in the end of chapter 4, it is out of scope going in depth over all of the cases, modifications, and applications, but it is worth mentioning their existence.

Some of those methods include: *Alpha complex*, *Witness complex*, the *Zig-Zag persistence* and many others. All of these alternatives fill different niches: some are modifications of the already presented methods that solve problems of very specific cases, others improve computational speed at the cost of precision or even proof.

For improving computational time, the *Witness complex* might seem appealing: The general idea is to construct the simplicial complex on a "landmark" set , reducing the number of points, noise, and computation time. Although being very attractive due to efficiency, it lacks a stability theorem similar to the Niyogi-Smale-Weinberger and the dependence on choice of landmarks is not well understood [17]. Consequently, it has been shown that *Witness complex* fails to reconstruct even simple examples [18].

Even then, there are ways to work around those problems, statistical methods that can be applied to the usage of *Witness complex* as well as cases where data has noise, is incomplete, etc. These methods are varied, complex and require some statistical background. If the reader wants to deepen this we reccomend chapter 3 of [19], as well as [11] for a better understanding of the *Witness complex*.

APPENDIX A

ANNEXES

A.1 Persistence diagram

The code used to generate the persistence diagrams and barcodes in fig: 3.3 and the bottelneck in fig: 3.4

```
#prepare all the packages
import numpy as np
from pathlib import Path
import gudhi as gd
from gtda.plotting import plot_point_cloud
import plotly
import pandas as pd  # Not a requirement of giotto-tda, but is compatible with
↪  the gtda.mapper module
import persim

# TDA magic
from gtda.mapper import (
    CubicalCover,
    make_mapper_pipeline,
    Projection,
    plot_static_mapper_graph,
    plot_interactive_mapper_graph,
    MapperInteractivePlotter
)

# ML tools

from gtda.plotting import plot_point_cloud
from sklearn import datasets
```

```
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA

# a function to generate sets of points over a torus
def make_point_torus(n_points: int, noise: float=0.1, R: float = 0.75, r:
↪  float=0.25):
    torus=[]
    for n in range(n_points):
        u=np.random.rand()*np.pi*2
        v=np.random.rand()*np.pi*2
        torus.append([
                    (R + r*np.cos(u)) * np.cos(v) + noise *
                    ↪  (np.random.rand(1)[0] - 0.5),
                    (R + r*np.cos(u)) * np.sin(v) + noise *
                    ↪  (np.random.rand(1)[0] - 0.5),
                    r*np.sin(u) + noise * (np.random.rand(1)[0] - 0.5),
                ])
    return np.asarray(torus)

# the different noise values will be great to observe how persistent homology
↪  behaves in front of noise
data1 = make_point_torus(n_points= 100, noise= 0.3)
data2 = make_point_torus(n_points= 100, noise= 0)

#convert the points into Gudhi RipsComplex data structure
RC1=gd.RipsComplex(points=data1)
RC2=gd.RipsComplex(points=data2)

#convert the one skeleton graphs into two trees of simplices that represent the
↪  two directed simplicial complexes. Max_dimension=3 because we don't expect
↪  anything more than H3 and that way the computation time is lower (otherwise
↪  it calculates all the different high dimensional simplices that are already
↪  closed)
VR1=RC1.create_simplex_tree(max_dimension=3)
VR2=RC2.create_simplex_tree(max_dimension=3)

#Convert the trees of simplices data structure into the multisets that represent
↪  the persistence diagrams
VRP1=VR1.persistence()
VRP2=VR2.persistence()

#plot the barcodes
ax = gd.plot_persistence_barcode(persistence= VRP1,
    legend=True)
ax.set_title("Persistence barcode of a noisy torus")
plt.show()

```

```
ax = gd.plot_persistence_barcode(persistence= VRP2,
    legend=True)
ax.set_title("Persistence barcode of a clean torus")
plt.show()

#plot the persistence diagrams

ax = gd.plot_persistence_diagram(persistence= VRP1,
    legend=True)
ax.set_title("Persistence diagram of a noisy torus")
plt.show()

ax = gd.plot_persistence_diagram(persistence= VRP2,
    legend=True)
ax.set_title("Persistence diagram of a clean torus")
plt.show()

#transform the list used to generate diagrams and barcodes into arrays
VRP12   = np.array([s[1] for s in VRP1])
VRP22   = np.array([s[1] for s in VRP2])

# Use the persim package to calculate the bottleneck distance and return the
↪  matching (the pairings of elements of VRP12 and VRP22 that generate the
↪  lower higher distance)
d, matching = persim.bottleneck(
    VRP12,
    VRP22,
    matching=True
)


#Finally plot the bottelneck distance and the respective matching
persim.bottleneck_matching(VRP12, VRP22, matching, labels=['Clean $H_1$', 'Noisy
↪  $H_1$'])
plt.title("Bottelneck Distance {:.3f}".format(d))
plt.show()
```

A.2 Vietoris-Rips visualization

The code used to generate the VR complexes discussed in 3.1:

```
#prepare all the packages
import numpy as np
from pathlib import Path
import pandas
```

```
import gudhi
from gtda.plotting import plot_point_cloud
import plotly
from plotly.graph_objs import graph_objs as go
import ipywidgets as widgets
from matplotlib.widgets import Slider

# a function to generate the random points
def make_point_torus(n_points: int, noise: float=0.1, R: float = 0.75, r:
    float=0.25):
    torus=[]
    for n in range(n_points):
        u=np.random.rand()*np.pi*2
        v=np.random.rand()*np.pi*2
        torus.append([
                    (R + r*np.cos(u)) * np.cos(v) + noise *
                    (np.random.rand(1)[0] - 0.5),
                    (R + r*np.cos(u)) * np.sin(v) + noise *
                    (np.random.rand(1)[0] - 0.5),
                    r*np.sin(u) + noise * (np.random.rand(1)[0] - 0.5),
                ])
    return np.asarray(torus)

# points is an array given by the defined function, rc the VR complex of it and
    st the Simplex tree (st is the actual complex, rc is used solely for
    persistence diagrams)
points = make_point_torus(200)
rc = gudhi.RipsComplex(points=points)
st = rc.create_simplex_tree(max_dimension=2)

#st.get_skeleton(2) returns the simplices of the skeleton of dimension 2 as a
    pair: the list of the points that generate the simplex and the value of
    epsilon needed for the simplex to be generated
triangles = np.array([s[0] for s in st.get_skeleton(2) if len(s[0])==3 and s[1]
    <= 1])

#plot the points that we have randomly generated before going into the VR
    complex representation
plot_point_cloud(points)

# use plotly offline mode to genereate the widget
plotly.offline.init_notebook_mode()
from plotly.offline import iplot

#define epsilon as the value of the slider
epsilon = widgets.FloatSlider(
    value = 0.2,
```

```
    min = 0.075,
    max = 2,
    step = 0.0001,
    description = 'Epsilon:',
    readout_format = '.4f'
)

#mesh puts together the cordinates of the different points (x,y,z) and the elements that conform the 2-simplices (i,j,k)
mesh = go.Mesh3d(
    x = points[:, 0],
    y = points[:, 1],
    z = points[:, 2],
    i = triangles[:, 0],
    j = triangles[:, 1],
    k = triangles[:, 2]
)

#the figure structure that the widget will use.
fig = go.FigureWidget(
    data = mesh,
    layout = go.Layout(
        title = dict(
            text = 'Vietoris Rips Complex Representation of the 2-Torus'
        ),
        scene = dict(
            xaxis = dict(nticks = 4, range = [-1.5, 1.5]),
            yaxis = dict(nticks = 4, range = [-1.5, 1.5]),
            zaxis = dict(nticks = 4, range = [-1.5, 1.5])
        )
    )
)

#given a value of epsilon, calculate the VR of that value, its points and the 2-simplicies that they generate
def view_torus(epsilon):
    if epsilon < 0.075:
        epsilon = 0.075
    triangles = np.array([s[0] for s in st.get_skeleton(2) if len(s[0]) == 3 and s[1] <= epsilon])
    fig.data[0].i = triangles[:, 0]
    fig.data[0].j = triangles[:, 1]
    fig.data[0].k = triangles[:, 2]
    iplot(fig)

widgets.interact(view_torus, epsilon = epsilon);
```

```
86
```

A.3 MAPPER

```
1  import sys
2  # Data wrangling
3  import numpy as np
4  import pandas as pd  # Not a requirement of giotto-tda, but is compatible with
   ↪  the gtda.mapper module
5
6  # Data viz
7  from gtda.plotting import plot_point_cloud
8
9  # TDA magic
10 from gtda.mapper import (
11     CubicalCover,
12     make_mapper_pipeline,
13     Projection,
14     plot_static_mapper_graph,
15     plot_interactive_mapper_graph,
16     MapperInteractivePlotter
17 )
18
19 # ML tools
20 from sklearn import datasets
21 from sklearn.cluster import DBSCAN
22 from sklearn.decomposition import PCA
23
24 # Import the Iris dataset, only the data
25 iris = datasets.load_iris().data[:, :4]
26
27 # We configure a Mapper pipeline, choosing a filter function, a cover, and
   ↪  clustering arguments
28
29 # Define filter function - can be any scikit-learn transformer, we will use the
   ↪  classical PCA
30 from sklearn.decomposition import PCA
31 filter_func = PCA(n_components=3)
32 # Define cover
33 cover = CubicalCover(n_intervals=10, overlap_frac=0.30)
34 # Choose clustering algorithm - default is DBSCAN
35 clusterer = DBSCAN()
36
37 # Configure parallelism of clustering step
```

```
n_jobs = 1

# Initialise pipeline
pipe_iris = make_mapper_pipeline(
    filter_func=filter_func,
    cover=cover,
    clusterer=clusterer,
    verbose=True,
    n_jobs=n_jobs,
)

# We finally plot the result in a very basic way

fig = plot_static_mapper_graph(pipe_iris, iris)
fig.show(config={'scrollZoom': True})


```

As a continuation of the code before, we generate a new pipeline but we apply it to the torus

```
    torus=make_point_torus(200)


    # Define filter function - can be any scikit-learn transformer, this time we
    ↪  use a simple projection
    filter_func = Projection(columns=[1])
    # Define cover
    cover = CubicalCover(n_intervals=10, overlap_frac=0.3)
    # Choose clustering algorithm - default is DBSCAN
    clusterer = DBSCAN()

    # Configure parallelism of clustering step
    n_jobs = 1

    # Initialise pipeline
    pipe_torus = make_mapper_pipeline(
        filter_func=filter_func,
        cover=cover,
        clusterer=clusterer,
        verbose=False,
        n_jobs=n_jobs,
    )

    # Finally we just plot the result
    fig = plot_static_mapper_graph(pipe_torus, torus)
```

```
25    fig.show(config={'scrollZoom': True})
```

A.4 Mapper applied to cancer

In this case, most of the code is a result of trying to improve the resulting plot. Multiple plots are generated with this code, some of them are not used in the main chapters but is included because will be used in the presentation.

```
1     import sys
2     try:
3         import pandas as pd
4     except ImportError as e:
5         print("pandas is required for this example. Please install with conda or
          ↪  pip  and then try again.")
6         sys.exit()
7
8     import numpy as np
9     import sklearn
10    from sklearn import ensemble
11    import kmapper as km
12    from kmapper.plotlyviz import *
13
14    import warnings
15    warnings.filterwarnings("ignore")
16
17    import plotly.graph_objs as go
18    from ipywidgets import (HBox, VBox)
19
20    #First we visualize the resulting graph via a color_function that associates
      ↪  to lens data their x-coordinate distance to min, and colormap these
      ↪  coordinates to a given Plotly colorscale. Here we use the brewer
      ↪  colorscale with hex color codes.
21
22    # Data - the Wisconsin Breast Cancer Dataset
23    # https://www.kaggle.com/uciml/breast-cancer-wisconsin-data
24    df = pd.read_csv('/kaggle/input/breast-cancer-wisconsin-data/data.csv')
25    feature_names = [c for c in df.columns if c not in ["id", "diagnosis"]]
26    df["diagnosis"] = df["diagnosis"].apply(lambda x: 1 if x == "M" else 0)
27    X = np.array(df[feature_names].fillna(0))
28    y = np.array(df["diagnosis"])
29
30    # Create a custom 1-D lens with Isolation Forest
31    model = ensemble.IsolationForest(random_state=1714)
32    model.fit(X)
33    lens1 = model.decision_function(X).reshape((X.shape[0], 1))
```

```
34
35      # Create another 1-D lens with L2-norm
36      mapper = km.KeplerMapper(verbose=0)
37      lens2 = mapper.fit_transform(X, projection="l2norm")
38
39      # Combine both lenses to get a 2-D [Isolation Forest, L^2-Norm] lens
40      lens = np.c_[lens1, lens2]
41
42      # Define the simplicial complex (similar to how Giotto uses pipeline)
43      scomplex = mapper.map(lens,
44                            X,
45                            cover = km.Cover(n_cubes = 15,
46                                             perc_overlap=0.7),
47                            clusterer=sklearn.cluster.KMeans(n_clusters=2,
48                                                             random_state=3471))
49
50
51
52      pl_brewer = [[0.0, '#006837'],
53                   [0.1, '#1a9850'],
54                   [0.2, '#66bd63'],
55                   [0.3, '#a6d96a'],
56                   [0.4, '#d9ef8b'],
57                   [0.5, '#ffffbf'],
58                   [0.6, '#fee08b'],
59                   [0.7, '#fdae61'],
60                   [0.8, '#f46d43'],
61                   [0.9, '#d73027'],
62                   [1.0, '#a50026']]
63
64
65
66      color_values = lens [:,0] - lens[:,0].min()
67      my_colorscale = pl_brewer
68      kmgraph,  mapper_summary,
69      colorf_distribution = get_mapper_graph(scomplex,
70                                             color_values,
71                                             color_function_name='Distance to x-min',
72                                             colorscale=my_colorscale)
73
74      # assign to node['custom_tooltips']  the node label (0 for benign, 1 for
        ↪  malignant)
75      for node in kmgraph['nodes']:
76          node['custom_tooltips'] = y[scomplex['nodes'][node['name']]]
77
78
79
```

```
80  #Since the chosen colorscale leads to a few light colors when it it used for
    ↪ histogram bars, we set a black background color to make the bars
    ↪ visible:
81
82  bgcolor = 'rgba(10,10,10, 0.9)'
83  y_gridcolor = 'rgb(150,150,150)'# on a black background the gridlines are
    ↪ set on  grey
84
85
86
87
88
89  plotly_graph_data = plotly_graph(kmgraph, graph_layout='fr',
    ↪ colorscale=my_colorscale,
90                                   factor_size=2.5, edge_linewidth=0.5)
91  layout = plot_layout(title='Topological network representing the<br>  breast
    ↪ cancer  dataset',
92                       width=620, height=570,
93                       annotation_text=get_kmgraph_meta(mapper_summary),
94                       bgcolor=bgcolor)
95
96  fw_graph = go.FigureWidget(data=plotly_graph_data, layout=layout)
97  fw_hist = node_hist_fig(colorf_distribution, bgcolor=bgcolor,
98                          y_gridcolor=y_gridcolor)
99  fw_summary = summary_fig(mapper_summary, height=300)
100 dashboard = hovering_widgets(kmgraph,
101                              fw_graph,
102                              ctooltips=True, # ctooltips = True, because we
                                ↪ assigned a label to each
103                                                #cluster member
104                              bgcolor=bgcolor,
105                              y_gridcolor=y_gridcolor,
106                              member_textbox_width=600)
107
108 #Update the fw_graph colorbar, setting its title:
109
110 fw_graph.data[1].marker.colorbar.title = 'dist to<br>x-min'
111
112
113
114 #finally we print the reulting graph
115
116 VBox([fw_graph, HBox([fw_summary, fw_hist])])
```

BIBLIOGRAPHY

[1] P. Niyogi, S. Smale, and S. Weinberger, "Finding the homology of submanifolds with high confidence from random samples," *Discrete & Computational Geometry*, vol. 39, pp. 419–441, 2008. 1.1, 3.1.1, 3.1.3, 5

[2] G. Singh, F. Mémoli, G. E. Carlsson *et al.*, "Topological methods for the analysis of high dimensional data sets and 3d object recognition." *PBG@ Eurographics*, vol. 2, pp. 091–100, 2007. 1.2, 4.3, 4.3.3, 4.5

[3] J. Lee, *Introduction to topological manifolds*. Springer Science & Business Media, 2010, vol. 202. 2.1.3

[4] S. Eilenberg and N. E. Steenrod, "Axiomatic approach to homology theory," *Proceedings of the National Academy of Sciences*, vol. 31, no. 4, pp. 117–120, 1945. 2.4

[5] D. Kozlov, "Algorithms and computation in mathematics vol. 21," 2008. 3.1.1

[6] J. Latschev, "Vietoris-rips complexes of metric spaces near a closed riemannian manifold," *Archiv der Mathematik*, vol. 77, no. 6, pp. 522–528, 2001. 3.1.3

[7] T. G. Project, *GUDHI User and Reference Manual*, 3rd ed. GUDHI Editorial Board, 2023. [Online]. Available: https://gudhi.inria.fr/doc/3.7.1/ 4, 2

[8] G. Tauzin, U. Lupo, L. Tunstall, J. B. Pérez, M. Caorsi, A. Medina-Mardones, A. Dassatti, and K. Hess, "giotto-tda: A topological data analysis toolkit for machine learning and data exploration," 2020. 4

[9] U. Bauer, "Ripser: efficient computation of Vietoris-Rips persistence barcodes," *J. Appl. Comput. Topol.*, vol. 5, no. 3, pp. 391–423, 2021. [Online]. Available: https://doi.org/10.1007/s41468-021-00071-5 4

[10] J.-D. Boissonnat and C. Maria, "The simplex tree: An efficient data structure for general simplicial complexes," *Algorithmica*, vol. 70, pp. 406–427, 2014. 4.1

[11] V. De Silva and G. E. Carlsson, "Topological estimation using witness complexes." in *PBG*, 2004, pp. 157–166. 4.1, 5

[12] L. J. Guibas and S. Y. Oudot, "Reconstruction using witness complexes," *Discrete & computational geometry*, vol. 40, no. 3, p. 325, 2008. 4.1

[13] V. De Silva, D. Morozov, and M. Vejdemo-Johansson, "Dualities in persistent (co) homology," *Inverse Problems*, vol. 27, no. 12, p. 124003, 2011. 4.1

[14] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington, "A roadmap for the computation of persistent homology," *EPJ Data Science*, vol. 6, pp. 1–38, 2017. 4.1

[15] P. S. Hoey, "Statistical analysis of the iris flower dataset," *University of Massachusetts At Lowell, Massachusetts*, 2004. 4.3.1

[16] P. C. Nowell, "The clonal evolution of tumor cell populations: Acquired genetic lability permits stepwise selection of variant sublines and underlies tumor progression." *Science*, vol. 194, no. 4260, pp. 23–28, 1976. 4.3.3

[17] F. Chazal, V. De Silva, and S. Oudot, "Persistence stability for geometric complexes," *Geometriae Dedicata*, vol. 173, no. 1, pp. 193–214, 2014. 5

[18] L. J. Guibas and S. Y. Oudot, "Reconstruction using witness complexes," *Discrete & computational geometry*, vol. 40, no. 3, p. 325, 2008. 5

[19] R. Rabadán and A. J. Blumberg, *Topological data analysis for genomics and evolution: topology in biology.* Cambridge University Press, 2019. 5