POLITECNICO
MILANO 1863

Advanced Programming for
Scientific Computing

lmapper, predmap:
two Python packages for Predictive Topological Data Analysis

Martino Milani

May 27, 2019

Abstract

A fundamental problem in data analysis is the visualization of high dimensional datasets.
G. Carlsson et al. in [3] propose an algorithm called Mapper that visualizes the dataset through a simplicial complex that aims at approximating the topology of the dataset. Few implementations of this algorithm already exist [15, 22]; in this work we propose lmapper, a Python package that provides an alternative, efficient and easily extensible implementation of Mapper. We benchmarked lmapper against [22] on two different datasets and verified that in both cases lmapper is faster and more robust to noise. We then present a companion Python package, filterutils, written in C++ with the use of OpenMP for higher performances.
Palma et al. in [18] proposed a binary classification algorithm based on Mapper; in this work we developed on top of lmapper a third package, predmap, implementing such algorithm. With predmap we were able in a simple case (majority vote) to retrieve the same results presented in [18].

Contents

1 Topological Data Analysis and Mapper

One way for scientists to make sense of complex data is to identify groups (clusters) of similar data points. Instead of thinking of each data point as an individual entity we wish to identify a small number of meaningful groups and being able to label them makes scientists able to describe datasets with only a little information; the data point is seen as part of a bigger entity: the cluster which it belongs to. By neglecting differences between data points belonging to the same cluster, it is possible to achieve a very effective and compressed description of the dataset. Many clustering algorithms have been developed so far; most common algorithms, such as Gaussian Mixture Models, are founded on a probabilistic model that is build on the data. Other algorithms include optimization methods (k-means for example), density-based methods and hierarchical algorithms [9].

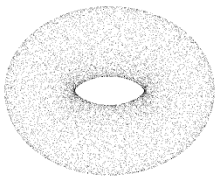


Figure 1: A data point cloud sampled from a torus in $\mathbb{R}^3$

In the work of G. Carlsson et al. [12] a new algorithm called Mapper, based on a different approach, was presented. The observed points are thought not as sampled from a distribution, such as in GMM, but are thought to be sampled from an ideal object $\mathcal{O}$ with a particular shape; Mapper tries to visualize this shape. The concept of shape of a high-dimensional object is although non trivial, but tools to study the shape of such high-dimensional objects were already available in literature before Mapper: Reeb graphs. Reeb graphs manage to describe in a compact way information about an object $\mathcal{O}$ by keeping track of the evolution of the topology of the connected components of the level sets $f^{-1}(c), c \in \mathbb{R}$ of a function $f : \mathcal{O} \to \mathbb{R}$[6] [1]. The object $\mathcal{O}$ is however only an abstraction; inspired from the Reeb graphs, Mapper is an algorithm that implements this same idea but on the finite collection of data points, where the notion of "connected component" and level set are not well defined as on the ideal object $\mathcal{O}$. The output of Mapper can be thought of as the skeleton of the shape of the object $\mathcal{O}$ from which the points are imagined to be sampled. Thanks to a careful visual analysis of this skeleton we can improve our understanding of the dataset, and thus it is sometimes possible to perform a finer and more meaningful clustering of the datasets than with traditional algorithms [12]. In the following paragraph a first example of an application of Mapper is presented, in order to introduce the reader to the algorithm. With this example we want to provide the reader just with a first intuition of Mapper and introduce him to concepts such as filter function and distance; in the next section a formal definition of Mapper is provided.

A first example In this first introductory example, taken from [12], a three dimensional dataset is analyzed, since it is possible to visualize the data point cloud and get an understanding

of the "shape" of the dataset just by looking at it.
The set of data points X is represented in figure 2. A shape of a hand is recognizable. First, we need to define a distance d on the data point cloud, in order to be able to introduce a notion of "closeness" between points, onto which a notion of connectedness of the data point cloud is based. In this way Mapper will be able to recognize points belonging to the same finger as part of the same connected component. A connected component of the object $\mathcal{O}$ is subset of $\mathcal{O}$ that cannot be represented by the union of two other disjoint subsets.
For the next definition the concept of metric space is needed. The readers not familiar with metric spaces can think for now about a metric space as a finite collection of points X equipped with a function d that associates to each couple of points $x, y \in X$ a scalar value that represent a measure of how far these two points are. For a formal definition of metric space, see definition 1.5.

Definition 1.1. [*data point cloud*] A **data point cloud** is a *finite* metric space (X, d) where $X \subset \mathbb{R}^n$ and d is some distance in $\mathbb{R}^n$.

For this example, the euclidean distance $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$ is chosen. After this step, Mapper requires to define a function f on the data point cloud (X, d), called a *filter function*.

Definition 1.2. [*filter function*] A **filter function** is a continuous function $f\colon (X, d) \to \mathbb{R}$.

The chosen filter function $f : (X, d) \to \mathbb{R}$ is the x-coordinate of the points in X. In figure 2 B) the points are colored by the value of the filter function, where blue means low values and red high values. The image $f(X) \subset \mathbb{R}$ is then binned into six, equally spaced, overlapping intervals $\{B_i \subset \mathbb{R}\colon \bigcup_{i=1}^{6} B_i = f(X)\}$. The data point cloud (X, d) is then subdivided into six corresponding subsets $\{A_i \subset X\colon A_i = f^{-1}(B_i)\}_{i=1}^{6}$ represented in figure 2 C. Notice that, since the intervals B_i were overlapping, the subsets $A_i = f^{-1}(B_i)$ in figure 2 C share some points between them. On each of the six subsets A_i , a clustering algorithm that makes use of the distance d defined on X is run. The clusters obtained represent in some way the notion of connected components of each $A_i \subset X$: if two points were close enough to be clustered together, they belonged to the same connected component. One node for each connected component obtained so far is created (figure 2 D). An edge is drawn only between the nodes representing connected components that share at least one point.

In the next section we adopt a rigorous approach to define Mapper. If the reader needs further examples, we suggest reading the paper of G. Carlsson *et al.* [12] where the authors provide an introduction to Topological Data Analysis and Mapper and give three examples of applications of Mapper to represents the shape of datasets representing player performance data from the NBA, voting data of the House of the Representatives and gene expression data from breast tumors.

1.1 Introduction to topology

Many of the following definitions, notations and theorems, although sometimes slightly reformulated for better following the purpose of this work, have been taken from the book of James R. Munkres, *Topology*[14]. We'll state, along with some examples, the basic ingredients that will lead us to the definition of Mapper. If the reader notices to have difficulties in following

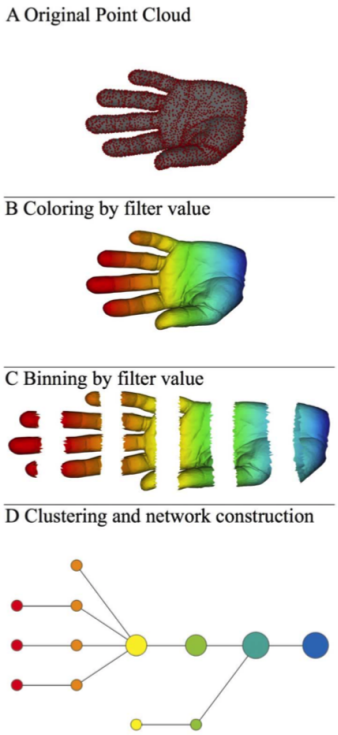


Figure 2: A data point cloud with the shape of a hand; A) the data point cloud X B) X is colored by the values of the filter function f C) X is split into six subsets $\{A_i\}_{i=1}^{6}$ D) each connected component of each subset A_i is represented by a node in the graph. An edge is drawn between the nodes corresponding to connected components that share at least one point. *Figure from G. Carlsson et al, [12]*

this section, he can refer to Chapter 2 of [14], *Topological Spaces and Continuous Functions* for a more complete explanation.

Topology, open sets and continuity To study shapes, one could choose many different ways. "Choosing a way" to study shapes means choosing at what point an object can be deformed before considering that its shape changed [2]. Let's consider a simple example: although the two following objects look very different

$$\mathcal{R}, \quad R$$

it can still be recognized in both of them the same shape: the shape of the letter R. Yet, the former is much more bended and curled than the latter. Topology formalizes and extends this intuitive notion of shape, that it is independent by transformations such as stretching and bending. These transformations are "continuous" in the sense that nearby points before such transformations are nearby also after. In other words, through the glasses of topology an object has the same shape no matter what continuous transformation is applied to it.

Definition 1.3. [*Topological Space*] A **topology** on a set X is a family $\mathcal{T}$ of subsets of X with the following three properties:

- both the empty set $\varnothing$ and X belong to $\mathcal{T}$
- $\mathcal{T}$ is closed with respect to *any* kind of union (finite or infinite, countable or even uncountable)
- $\mathcal{T}$ is closed with respect to *finite* intersections

The ordered pair $(X, \mathcal{T})$ is called **topological space**

Any element of the topology $\mathcal{T}$ is called **open set**. A way to define a topology on X would be then to define a collection of subsets of X containing X itself or the empty set $\varnothing$, and taking the closure of such collection with respect to any kind of union and to finite intersections. However, to define a topology on a set X one often first defines a smaller and easier to describe collection of subsets of X, called *basis* for $\mathcal{T}$ and builds $\mathcal{T}$ from that.

Definition 1.4. [*basis*] A **basis** for a topology $\mathcal{T}$ on the set X is a collection of subsets $\mathcal{B}$ of X that satisfies the following properties:

- for each element x of X, there exists at least one element B of $\mathcal{B}$ such that x is contained in B.
- if x is contained in the intersection of two basis elements B_1 and B_2 , then there exists a third basis element B_3 such that $x \in B_3 \subset B_1 \cap B_2$

Once given a basis, a topology on X can be constructed by defining as *open* any set $U \subset X$ such as for every x in U there exist a basis element B such that x is contained in B and B is contained in U. To see that the collection of open sets defined in this way defines a topology in the sense of definition 1.3, see *Munkers*, [14]. This construction is more practical and easy to handle in proving theorems and properties: to see in an example how it can be used, see the proof of lemma 1.1.

Another way to build a topology on a set X is by starting defining a *metric* on the set X.

Definition 1.5. [*metric space*] A **metric** on a set X is a function

$$d : X \times X \to \mathbb{R}$$

such that

- $d(x, y) \geq 0$ for every $x, y \in X$, $d(x, y) = 0$ if and only if $x = y$
- $d(x, y) = d(y, x)$ for every $x, y \in X$
- $d(x, y) + d(y, z) \geq d(x, z)$ for every $x, y, z \in X$

The ordered pair (X, d) is called a **Metric Space**

An ϵ ball $B(x, \epsilon)$ is the subset of X such that

$$B_d(x, \epsilon) := \{y \in X : d(y, x) < \epsilon\}$$

Once defined a metric on X, we can build the following topology on X:
Definition 1.6. [*metric topology*] The **metric topology** induced on the metric space (X, d) by the distance d is the topology whose basis is the family of all the ϵ-balls $B_d(x, \epsilon)$ for every $x \in X$ and every $\epsilon > 0$.

It can be checked that the family of all the ϵ-balls respect the definition 1.4.
Proposition. A subset U of a metric space (X, d) is *open* in the metric topology if, given any $x \in U$, there exists a positive, real number $\epsilon > 0$ such that, given any point $y \in X : d(x, y) < \epsilon \implies y \in U$.

Or equivalently, U is *open* if every point in U has a neighborhood contained in U.
Thanks to the geometric meaning of the metric d at the base of this construction, the metric topology respects many desired properties "for free", i.e it separates points (*Hausdorff property*).
Remark. Starting from a metric space (X, d) it is *always* possible to build a topology in the way we described above, but the converse is *not* aways true. Topological spaces $(X, \mathcal{T})$ for which it is possible to find a metric d that induces the topology $\mathcal{T}$ are called **metrizable spaces**

Now we introduce the concept of continuous function. We use the more general topological definition of continuity, even if less intuitive of the more common metric definition.
Definition 1.7. [*continuous function*] A function

$$f : (X, \mathcal{T}) \to (Y, \mathcal{T}')$$

between two topological spaces $(X, \mathcal{T})$ and $(Y, \mathcal{T}')$ is **continuous** if

$$U' \in \mathcal{T}' \implies f^{-1}(U') \in \mathcal{T}$$

The function f is continuous if the pre-image of each open set in Y is an open set in X. Note that such a definition uses the topologies of both X and Y. The property of f of being continuous depends not only on the function f itself, but also on the two topologies that we define on X and Y. Fixed $\mathcal{T}$, the finer the topology $\mathcal{T}'$ on the codomain, the harder it is for a function f to

be continuous, since f has to be smooth enough and well-behaved to ensure that the pre-image of every set in $\mathcal{T}'$ belongs to $\mathcal{T}$. Vice versa, the more coarse the topology $\mathcal{T}'$, the easier it is for f to be continuous i.e. functions even not so well-behaved could become at a certain point continuous.

Definition 1.8. [*homeomorphism*] A continuous function $f : X \to Y$ is called **homeomorphism** if its inverse $f^{-1} : Y \to X$ is also continuous.

A *homeomorphism* defines a bijection between the two topologies of X and Y. This definition states that not only the preimage of each open set of Y is an open set in X, but also the image of an open set in X is an open set in Y. Properties of the space $(X, \mathcal{T})$ invariant through homeomorphisms are called *topological properties*. To familiarize with the concept of basis of a topology, and to see how the two structures of *metric* and *metric topology* interact, we think it is instructive to prove that the topological definition of continuity is actually equivalent to the most intuitive metric definition, the famous $\epsilon - \delta$ definition.

Lemma 1.1. A function

$$f : (X, d_X) \to (Y, d_Y)$$

between two metric spaces (X, d_X) and (Y, d_Y) is **continuous** in the sense of definition 1.7 with respect to the two metric topologies $\mathcal{T}, \mathcal{T}'$ of X, Y if and only if, given $x \in X$, for every $\epsilon > 0$ there exist a $\delta > 0$ such that $y \in B_{d_X}(x, \delta) \implies f(y) \in B_{d_Y}(f(x), \epsilon)$

Proof. Suppose f is continuous. Fix $x \in X, \epsilon > 0$. The ϵ-ball $B_{d_Y}(f(x), \epsilon)$ is an element of the basis of the metric topology $\mathcal{T}'$. By hypothesis then, $f^{-1}(B_{d_Y}(f(x), \epsilon))$ is an element of the metric topology $\mathcal{T}$ of X. Since $x \in f^{-1}(B_{d_Y}(f(x), \epsilon))$, by proposition 1, there must exist a $\delta = \delta(\epsilon)$ such that $B_{d_X}(x, \delta) \subset f^{-1}(B(f(x), \epsilon))$ i.e. $f(B_{d_X}(x, \delta)) \subset B_{d_Y}(f(x), \epsilon)$
Now let's prove the opposite implication. Suppose that given $x \in X$, for every $\epsilon > 0$ there exist a $\delta > 0$ such that $y \in B_{d_X}(x, \delta) \implies f(y) \in B_{d_Y}(f(x), \epsilon)$. Now, fix an open subset U of Y. By proposition 1, for every element $y \in U$ there exist $\epsilon = \epsilon(y)$ such that $B(y, \epsilon(y)) \subset U$. By hypothesis, $\forall x : x \in f^{-1}(y)$ there exists a $\delta(\epsilon)$ such that the δ-ball $B_{d_X}(x, \delta)$ is mapped into the ball $B_{d_Y}(y, \epsilon) \subset U$. We just proved that for every $x \in f^{-1}(U)$ $\exists \delta : B_{d_X}(x, \delta) \subset f^{-1}(U)$. But this is exactly the definition of open set given in proposition 1. □

Homeomorphic spaces are in the same in the eye of topology; examples of topological properties are: connectedness, branches, holes and cavities. It is also possible to define a relaxed version of equivalence relation between two topological spaces, called homotopy equivalence. Two continuous maps g, h: $(X, \mathcal{T}) \to (Y, \mathcal{T}')$ are homotopic if there is a sequence of continuous functions f_t, $t \in [0, 1]$ such that $\forall x \in X$, $f_0(x) = g(x)$, $f_1(x) = h(x)$ and the sequence $\{f_t\}$is continuous in t (with respect to the product topology).

Definition 1.9. [*homotopy equivalence*] Homotopy equivalence can be defined first between functions and then between topological spaces as follows:

- Two continuous maps g and h between topological spaces X and Y are homotopic if there is a continuous map H, H: $X \times [0, 1] \to Y$ such that for all $x \in X$ $H(x, 0) = g(x)$ and $H(x, 1) = h(x)$.
- Two topological spaces X and Y are homotopy equivalent if there exist two continuous maps f: $X \to Y$ and g: $Y \to X$ such that $f \circ g$ and $g \circ f$ are homotopic to the identity map of Y and X respectively.

Connected components, covers In our daily experience the fact that an object is "all one piece" or not is important: a broken cup is not the same thing as an integer cup. When studying the "shape" of a space, this concept of connectedness plays the same important role. For this purpose we introduce the following definition.

Definition 1.10. [*connected space*] A **connected space** is a topological space that cannot be represented as union of two disjoint, non-empty open sets. A subset of a topological space X is a **connected set** if it is a connected space when viewed as a subspace of X.

We are now interested in studying when and how a space $(X, \mathcal{T})$ can be subdivided in connected components: for this we present the following basic result.

Lemma 1.2. Given a point $x \in (X, \mathbb{T})$, we define

$$C(x) = \bigcup \{C \subset X,\ C \text{ is connected and } x \in C\}$$

as the union of all connected subspaces of X that contain the point x. Then:

- Each subspace $C(x)$ is connected and closed.
- If $x, y \in X$, then $C(x)$ and $C(y)$ are either equal or disjoint.
- Every nonempty connected subspace of X is contained in a unique $C(x)$

The set $C(x)$ is called the **connected component** of x; the lemma shows that it is the largest connected subspace of X containing the point x. Since any two connected components of X are either equal or disjoint, we get a partition of the space X into largest connected subsets [19].

To partition a space is indeed important; often it will be needed to split a space into components, not necessarily connected, to study them one at the time. For this we continue defining the notion of cover of a space:

Definition 1.11. [*cover*] A **cover** of a topological space $(X, \mathcal{T})$ is a collection of subsets $\mathcal{A} = \{A_i : A_i \subset X\}_{i \in J}$ such that their union contains the space X: $X \subset \bigcup A_i$. If the subsets A_i are open inside $\mathcal{T}$, we call $\mathcal{A}$ an **open cover** of X.

Definition 1.12. [*pullback cover*] Given a continuous function between two topological spaces

$$f : (X, \mathcal{T}) \to (Y, \mathcal{T}')$$

and a cover $\mathcal{B}$ of the co-domain Y, the **pullback cover** of $\mathcal{B} = \{B_i\}_{i \in I}$ through f is the collection of sets $\mathcal{A} = \{A_i : A_i = f^{-1}(B_i)\}$

If $\mathcal{B}$ is an open cover of $(Y, \mathcal{T}')$, it is immediate to verify that $\mathcal{A}$ is an actual *open* cover of X due to the continuity of f and to common properties of unions and pre-images.

In section 1.3 we'll make use of the notion of *connected components of the pullback cover*; by this term we'll refer to the set made by the *largest* connected subspaces of the subsets A_i:

$$\{A_i^{(j)} \ :\ A_i \in \mathcal{A},\ A_i = \bigcup_{j \in J_i} A_i^{(j)},\ A_i^{(j)} \text{is a largest connected subspace of } A_i\}$$

Notice that the connected components of the same element A_i of the cover $\mathcal{A}$ are disjoint (lemma 1.2) but connected components of two different elements A_i, A_k of the cover $\mathcal{A}$ could have non empty intersection.

Reeb Graph The understanding of Reeb graphs is propedeutic to the comprehension of Mapper; the aim of this paragraph is to deepen what a Reeb graph is, providing the mathematical definitions of the concepts necessary to build such a construction. Some examples at the end of are provided.

Definition 1.13. [*relation, equivalence relation*] A **relation** R on a set A is a subset of the cartesian product $A \times A$. An **equivalence relation** on a set A is a relation R on A such that:

- $(x, x) \in R$ for every x in A
- if $(x, y) \in R$, then $(y, x) \in R$
- if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$

We write $(x, y) \in R$ or $x \sim y$ equivalently.

Definition 1.14. [*equivalence class*] Given a set A and an equivalence relation $\sim$ on A, the equivalence class of an element a in A is the set

$$\{x \in A : x \sim a\}$$

It can be proven that the collection of all the equivalence classes is a partition of A i.e. they are all disjoint and their union correspond to A.

Definition 1.15. [*quotient set*] The set of all equivalence classes is the **quotient set** of A by $\sim$ and is indicated as $A/\sim$

Observe that the relation R on the topological set X

$$R = \{(x, y) \in X : x, y \text{ belong to the same largest connected component}\}$$

is an equivalence relation. It's important to keep this in mind while following the next sections.

Definition 1.16. [*quotient map*] Let $(X, \mathcal{T})$ and $(Y, \mathcal{T}')$ be two topological spaces; let $p : (X, \mathcal{T}) \to (Y, \mathcal{T}')$ a *surjective* map. The map p is said to be a **quotient map** if a subset U of Y is open in Y if and only if $p^{-1}(U)$ is open in X

Note that p is required to be only surjective: a bijective quotient map is indeed a homeomorphism.

Definition 1.17. [*quotient topology*] Let $(X, \mathcal{T})$ be a topological space and A a set. If $p : (X, \mathcal{T}) \to A$ is surjective, then there exists one and only one topology $\mathcal{T}'$ on A such that $p : (X, \mathcal{T}) \to (A, \mathcal{T}')$ is a quotient map. The topology $\mathcal{T}'$ is called **quotient topology** induced by p.

The topology $\mathcal{T}'$ is defined in a straightforward way by those subsets U of A such that $p^{-1}(U) \in \mathcal{T}$. It can be proven that this family is indeed a topology.

Example. Let p be the map of the real line $\mathbb{R}$ with the usual metric topology $\mathcal{T}$ onto the three point set $A = \{a, b, c\}$ defined by

$$p(x) = \begin{cases} a & \text{if } x < 0 \\ b & \text{if } x \in [0, 1] \\ c & \text{if } x > 1 \end{cases}$$

One can check that the quotient topology on A induced by p is given by

$$\mathcal{T}' = \{\{a\}, \{c\}, \{a, c\}, \mathbb{R}, \varnothing\}$$

Definition 1.18. [*quotient space*] Let $(X, \mathcal{T})$ be a topological space, and let $\mathcal{P}$ be a partition of X into disjoint subsets whose union is X. Let $p : X \to \mathcal{P}$ be the surjective map that maps each point of X into the subset of the partition $\mathcal{P}$ containing it. The space $(\mathcal{P}, \mathcal{T}')$ where $\mathcal{T}'$ is the quotient topology induced by p is called **quotient space** of X.

Definition 1.19. [*Reeb graph*] Given a topological space X and a continuous function $f : X \to \mathbb{R}$, define the equivalence relation $\sim$ on X where $p \sim q$ whenever p and q belong to the same connected component of a single level set $f^{-1}(c)$ for some real c. Consider the map $p : X \to X/\sim$ that maps each point of X into its equivalence class. The **Reeb graph** is the quotient space $(X/\sim, \mathcal{T}')$ induced by the map p.

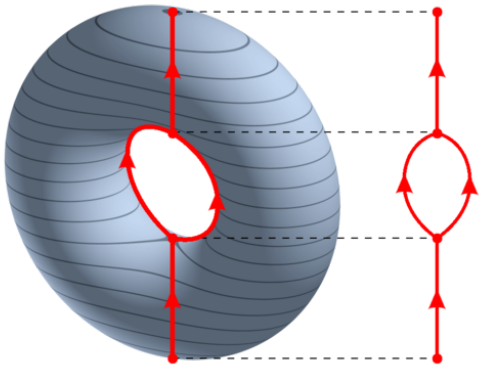


Figure 3: A Reeb graph of the torus.

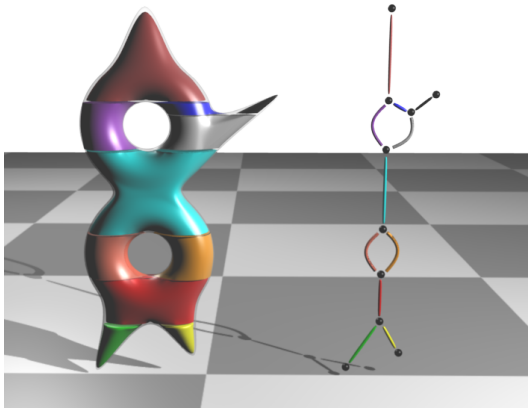


Figure 4: A Reeb graph of the double torus. Image taken from [6]

In figure 3 the topological space $(X, \mathcal{T})$ is the torus, and the function f is the height function. The horizontal "slices" are the level sets of the torus with respect to f. The Reeb graph keeps track of the changes of the connected components of the level sets providing an effective representation of the torus. The Reeb graph can be thought as the skeleton of the topological space $(X, \mathcal{T})$, where the connected components of the level sets collapse into a point providing a condensed, summarized representation of the original space X. Another example can be seen in 4.

1.2 Simplices and Simplicial Complexes

In the example discussed in section 1.2, we saw that the output of Mapper was an undirected graph. This was only a simple case; the output of Mapper is a higher dimensional object, called *simplicial complex*. We are interested in simplicial complexes because by the means of a suitable cover of the space X, the Nerve theorem **??** provides a way to construct a simplicial complex that is homotopy equivalent to X. The outline of this section is the following: first we'll provide the reader with a purely combinatorial notion of a simplicial complex. We'll provide an example to better illustrate this concept. Then, we'll introduce the concept of *realization of a simplicial complex* in a vector space, that will further help the reader familiarize with this concept since it provides a more intuitive visualization of the simplicial complex.

Definition 1.20. [*Simplicial Complex*] Given a finite set X, a **simplicial complex** $\mathcal{K}$ on X is a collection of subsets of X such that:

- every singleton of X is in $\mathcal{K}$ i.e. $\{x\} \in \mathcal{K}$ for every $x \in X$
- if $\gamma \subset X$ is in $\mathcal{K}$, then every subset of γ must be in $\mathcal{K}$ as well.

Each $\gamma \in \mathcal{K}$, $\gamma \subset X$ is called **simplex** of dimension $|\gamma| - 1$, where $|\gamma|$ is the cardinality of the subset γ .

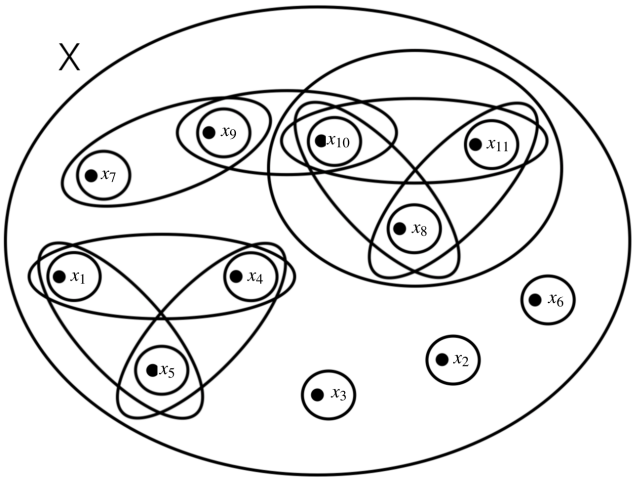


Figure 5: A visualization of a possible collection of subsets of X that satisfy the definition 1.20 of *abstract simplicial complex* on the finite set X

In figure 5 we provide an example of a simplicial complex on the set $X = \{x_i, i = 1, 2, ..., 11\}$. The elements of the abstract simplicial complex in figure 5 are:

- the 11 singletons $\{x_i\}, i = 1, 2, ..., 11$, called *simplices* of dimension 0.
- the *simplices* $\{x_1, x_4\}, \{x_4, x_5\}, \{x_5, x_1\}, \{x_7, x_9\}, \{x_9, x_{10}\}, \{x_{11}, x_{10}\}, \{x_8, x_{10}\}, \{x_8, x_{11}\}$ of dimension 1.
- the *simplex* $\{x_8, x_{10}, x_{11}\}$ of dimension 2.

A few more useful definitions:

Definition 1.21. [*n-face*] Every simplex σ of dimension n which is a subset of a higher-dimensional simplex γ is called **n-face** of the simplex γ.

- the *0-faces* are called *vertices*
- the *1-faces* are called *edges*

Definition 1.22. [*facet*] A **facet** is a simplex which is not the *face* of any other higher-dimensional simplex

The **dimension of a simplicial complex** is the largest dimension of any of its faces. A simplicial complex is said to be a **homogeneous K-complex** if any of its *facets* has the same dimension K. Let's continue the example in figure 5. We can now see that the complex represented is a *2-dimensional complex*, since its largest *facet* has dimension 2. The *1-faces* (or *edges*) of the two-dimensional simplex $\{x_8, x_{10}, x_{11}\}$ are the three one-dimensional simplices $\{x_{11}, x_{10}\}, \{x_8, x_{10}\}, \{x_8, x_{11}\}$ and its *vertices* are the zero-dimensional simplices $\{x_8\}, \{x_{10}\}, \{x_{11}\}$. It is useful to get a more geometric representation of a simplicial complex. To do so, we'll now build what is called the **realization of a simplicial complex** in a vector space.

Definition 1.23. [*linearly independent points*] The points $\{v_i\}_{i=0}^m \subset \mathbb{R}^n, 1 \leq m \leq n$ are said to be **linearly independent** if the m vectors $\{v_i - v_0\}_{i=1}^m$ are linearly independent.

Definition 1.24. [*convex hull*] Given a set of points $V = \{v_i\}_{i=1}^m \subset \mathbb{R}^n$, the **convex hull** of V is the set of the convex linear combinations of the elements of V:

$$\{\sum_{i=1}^m t_i v_i \ : \ \sum_{i=1}^m t_i = 1\}$$

Definition 1.25. [*realization*] Given a finite set X and an K-dimensional simplicial complex $\mathcal{K}$ on X, choose $\{v_x\}_{x \in X}$ linearly independent points in $\mathbb{R}^K$. The **realization of** $\mathcal{K}$ is the union of the convex hulls of $\{v_x\}_{x \in \gamma}$ for every simplex $\gamma \in \mathcal{K}$

In figure 6 we provide a realization of the simplicial complex represented in figure 5. The 2-simplex is represented by a triangle (the convex hull of three independent points in $\mathbb{R}^2$), the 1-simplices are represented by segments (the convex hulls of two independent points in $\mathbb{R}^2$) and the vertices are represented by simple points.

Definition 1.26. [*nerve complex*] Given a finite cover $\mathcal{A} = \{A_i\}_{i \in I}$ of a topological space X, its **nerve complex** is a simplicial complex $N(\mathcal{A})$ on the set I of indices where

$$N(\mathcal{A}) = \left\{ J \subset I : \bigcap_{j \in J} A_j \neq \varnothing \right\}$$

Example Take the set X as the segment in figure 7, and as open cover $\mathcal{A}$ of X the subsets $A_1,\ A_2,\ A_3,\ A_4,\ A_5$

$$X = \bigcup_{i \in I} A_i,\ I = \{1, 2, 3, 4, 5\}$$

The *nerve* of such a cover is the complex N illustrated both in its combinatorial form and in its realization. We can observe that there are five *1-dimensional simplices* (or *edges*), one for every couple of subsets A_i in the cover that overlap: (A_1, A_2), (A_2, A_3), (A_1, A_3), (A_3, A_4), (A_4, A_5). Furthermore, the subsets A_1, A_2, A_3 all overlap i.e. $A_1 \cap A_2 \cap A3 \neq \varnothing$, and this leads to the creation of one *2-dimensional simplex* with vertices the three subsets. In the realization of the

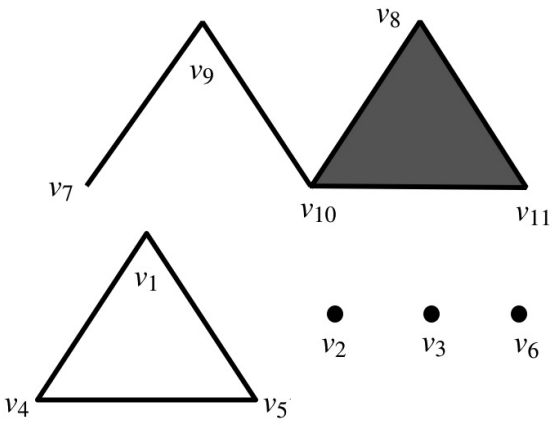


Figure 6: A possible realization of the simplicial complex in figure 5

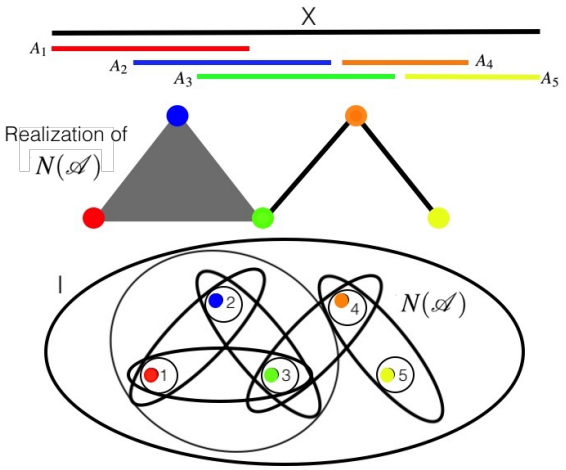


Figure 7: the simplicial complex $N(\mathcal{A})$, nerve of the open cover $\{A_i\}$ of the set X

nerve complex, the triangle represents the 2-dimensional simplex and the five edges (three edges of the triangle plus the two other edges) represent the 1-dimensional simplices.

The Nerve Theorem From an open cover $\mathcal{A} = \{A_i\}_{i\in I}$ of the topological space X it is possible to construct a simplicial complex on the index set I that is an "approximation" of X.
Definition 1.27. [*nerve of a cover*]
For a topological space X and its covering $\mathcal{A} = \{A_i\}_{i\in I}$ we call nerve of the cover the simplicial complex on I:

$$N(X, \mathcal{A}) = \left\{ J \subset I \mid \bigcap_{i\in J} A_i \neq \emptyset \right\}$$

An important property, called the Nerve theorem, states that under a specific condition on the cover $\mathcal{A}$, the nerve $N(X, \mathcal{A}))$ is homotopy equivalent to X.
Proposition 1 (Nerve Theorem)**.** Let $\mathcal{A} = \{A_i\}_{i\in I}$ be a open cover of a topological space X, if the following property holds

$$\forall J \subset I \bigcap_{i\in J} A_i \text{ is either empty or contractible to a point}$$

then the nerve $N(X, \mathcal{A})$ is homotopy equivalent to X.

Homotopy equivalent spaces have the important property of sharing the same "shape", or better, the same homology (*Munkres,* [14]). In other words, the Nerve theorem tells us that, for a dataset X sampled from a topological space $\mathcal{O}$ and a suitable open cover $\mathcal{A}$ of X , the shape (homology) of the nerve $N(X, \mathcal{A})$ is the same of the one of the ideal space $\mathcal{O}$.

1.3 Mapper

The output of Mapper is a simplicial complex that can be thought as the skeleton of the abstract object O from which the data point cloud (X, d) is sampled. We saw how the Reeb graph is capable of providing an effective skeletonization of a space; Mapper can be thought as a pixelized version of the *Reeb graph*, in the sense that Mapper provides an *approximated* skeleton of the data point cloud (X, d). Taking inspiration from [5], hereunder we define a general construction,that illustrates the idea behind Mapper. We call this construction pixelized Reeb graph.
Definition 1.28. [*pixelized Reeb graph*] Let $(X, \mathcal{T})$ and $(Y, \mathcal{T}')$ be topological spaces and let $f : (X, \mathcal{T}) \to (Y, \mathcal{T}')$ be a well-behaved and continuous map. Let $\mathcal{B} = \{B_i\}_{i\in I}$ be a finite open cover of Y. The **pixelized Reeb graph** arising from these spaces is defined to be the nerve simplicial complex of the connected components of the pullback cover.
Remark. (on definition 1.28) Consider $(X, \mathcal{T})$ a topological space and $f : (X, \mathcal{T}) \to (R, \mathcal{T}')$, where $\mathcal{T}'$ is the metric topology. Then consider the following option for $\mathcal{B} = \{B_\alpha\}_{\alpha\in A}$: $B_\alpha = (\alpha-\epsilon, \alpha+\epsilon)$ for $\alpha \in A = \{k\epsilon'\}_{k\in Z}$, $\epsilon' \in (0, 2\epsilon)$ for some fixed $\epsilon > 0$. This correspond to "ϵ-thick level sets", which induce a relaxed notion of Reeb graphs.

In other words: given a topological space $(X, \mathcal{T})$ we map it through a continuous function f in $(Y, \mathcal{T}')$. Then, we define a cover $\mathcal{B}$ of Y and we obtain the pullback cover $\mathcal{A} = \{A_i\}_{i\in I}$ on X. Then for each subset $A_i \in \mathcal{A}$ we take its partition $\{A_i^{(j)}\}_{j\in J_i}$ made by its connected components,

whose existence is assured by theorem 1.2. We build then the nerve complex of the pullback cover split into its largest connected subspaces i.e. $N(\{A_i^{(j)}, i \in I, j \in J_i\})$
A data point cloud X can be thought of as sampled from an underlying space with a particular shape that we are interested to capture and visualize through a process as close as possible to the pixelized Reeb graph. To do so, we need a similar notion of connected subspace but for a data point cloud. To do so we use a common clustering algorithm for metric spaces, the *simple linkage algorithm*. The clusters obtained will be our "connected components". Linkage algorithms depend on a parameter ϵ fixed *a priori*; different choices of this parameter lead to different partitions of the data point cloud. The smaller ϵ, the more this notion of connectedness is strong, and the more the algorithm will lead to the creation of many but small clusters. On the contrary, the bigger ϵ, the more relaxed the notion of connectedness and the more the algorithm creates few but bigger clusters. The output of Mapper will be sensible to changes of this parameter, that needs to be suitably tuned.
Definition 1.29. [*Single Linkage algorithm, connected components in a metric space*] Fix a positive real number $\epsilon > 0$. An ϵ-path in X is a set

$$P_\epsilon \;=\; \{x \in X \;: \forall x_0, x_n \in P_\epsilon \; \exists \{x_1, x_2, ..., x_{n-1}\} \subset P_\epsilon : d(x_i, x_{i+1}) < \epsilon \; \forall i = 0, ..., n-1\}$$

Two points x, y in the data point cloud (X, d) belong to the same cluster (or "connected component") if there exists an ϵ-path that connects the two i.e. $\exists \; P_\epsilon \;:\; x, y \in P_\epsilon$

Now that we have this notion of connectedness for metric spaces, we can modify slightly the definition 1.28 to have Mapper:
Definition 1.30. [*Mapper*] Let (X, d) be a data point cloud and $(Y, \mathcal{T}')$ be a topological space. Let $f : (X, \mathcal{T}) \to (Y, \mathcal{T}')$ be a continuous map, where $\mathcal{T}$ is the metric topology. Let $\mathcal{B} = \{B_i\}_{i \in I}$ be a finite open cover of Y. Fix a real parameter $\epsilon > 0$ for the single linkage algorithm. The **Mapper construction** arising from these data is defined as the simplicial complex induced by the nerve of the connected components $A_i^{(j)}$ (in the sense of definition 1.29) of the pullback cover $\mathcal{A} = \{A_i\}_{i \in I} \;=\; f^{-1}(\mathcal{B})$

$$M(X, d, \epsilon, f, \mathcal{B}) \;=\; N(\{A_i^{(j)}\})$$

Before moving to an example, it must be specified that in this section we used the single linkage algorithm to define a notion of connectedness in metric spaces; any other kind of clustering algorithm can be used, as soon as it leads to a partition of each element of the pullback cover $\mathcal{A}$. Some options could be other linkage algorithms, such as complete linkage and average linkage algorithms [4].

1.3.1 Example

We present an example of application of Mapper, where the typical choice of parameters is used. Furthermore, we discuss the importance of a correct choice of the ϵ parameter in the simple linkage algorithm. To summarize the choice of parameters for this example,

- $X \subset \mathbb{R}^2$ is the data point cloud represented in figure 8 sampled from a circle with some noise.
- $Y = \mathbb{R}$ endowed with the euclidean metric topology.

- d is the usual euclidean metric in $\mathbb{R}^2$
- $f : X \to \mathbb{R}$ is the height function $(x, y) \to y$
- $\mathcal{B}$ is the cover set by the colored intervals in figure 8

$$\mathcal{B} = \{B_1 = (b3, +\infty),\ B_2 = (b_1, b_4),\ B_3 = (-\infty, b_2)\} \text{ where } b_1 < b_2 < b_3 < b_4$$

- ϵ varies

Let's consider a data point cloud $(X, d) \subset \mathbb{R}^2$ represented in figure 8. We consider the usual euclidean distance d, as filter function $f : X \to \mathbb{R}$ the usual height function, and as cover $\mathcal{B}$ the cover given by the three colored overlapping intervals in figure 8. We represent the pullback cover by coloring the points by the same color of the interval that they are mapped to. Points that are mapped by f on a value belonging to an overlapping zone, are colored by both colors of the corresponding overlapping intervals. We represents clusters by explicitly drawing one of the possible ϵ-path. In figure (a) ϵ was chosen too small: the result is that the green point at the bottom is too far form the other green points and so it gets clustered alone, it's not considered "connected" to the other green points. Figure (b) is a "good" approximation of a circle: the linkage algorithm led to an intuitive clustering of the data point cloud, where all the green points are considered connected together, as well as all the blue ones. The violet points are grouped in two distinct clusters: the points on the left and the points on the right, since it is impossible to find a violet point on the left that is less than *epsilon* away to any of the violet points on the right. If we choose an ϵ big enough that it is possible to find a violet point on the left and a violet point on the right that are less than epsilon away, all the violet points will be considered "connected" i.e. belonging to the same cluster. This will make the Mapper look like a straight line, loosing any information about the hole in the middle of the data point cloud.

Playing with toy examples like this it is possible to understand what happens in many different configurations of the parameters $(d, f, \mathcal{B}, \epsilon)$ of Mapper. Many different configurations lead indeed to many different outputs. In the next section some choices for these parameters common in literature are presented.

1.3.2 Selection of parameters

In this subsection the cover $\mathcal{B} = \{B_i\}_{i \in I}$ is assumed to be built by n equally spaced, overlapping intervals B_i. Only the dependence of Mapper on the distance d, on the filter function $f : X \to \mathbb{R}$ and on ϵ is developed. Common choices of d, f and ϵ made in practical applications found in [12][17] are discussed.

1.3.3 Filter function

Geometric filters It is common to use filter functions that depend only on the distance d. Such filters $f : (X, d) \to \mathbb{R}$ are often a measure of the local density of the data point cloud or a measure of a centrality of the point x. One typical example of centrality measure is the L_∞-centrality f_∞:

$$f_\infty(x) = \max_{y \in X} d(x, y)$$

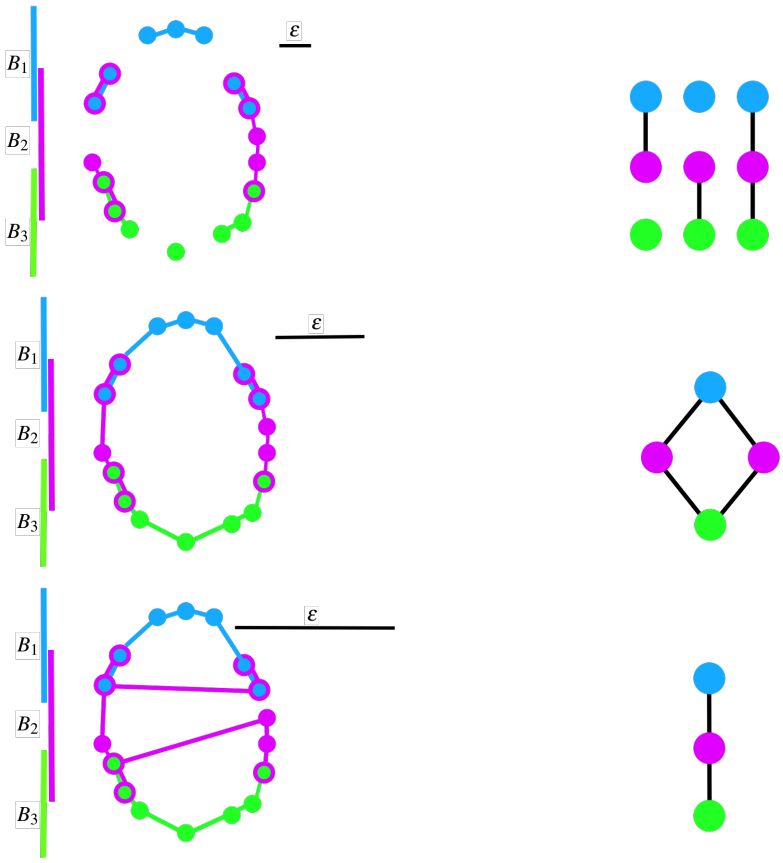


Figure 8: different choices of ϵ lead to different Mappers.

High values of f correspond to points "far" from the center of the data point cloud; the level sets $f^{-1}(B_i)$ will be consequently made by points at about the same "distance" from the center of the point cloud. Filters that depend only on the distance d are called *geometric filters*.

More filters On the other hand, a possible choice is to take as filter $f_P : (X, d) \to \mathbb{R}$ the projection of the points on one direction of choice. This was the choice made in the introductory example, where f was chosen as the projection on the first coordinate of the points $x \in \mathbb{R}^3$. Consider the data matrix $X \in \mathbb{R}^{m\times n}, n > m$, where each column is made by the vector of the m components of the corresponding data point x. The most common direction onto which to project the data point cloud is the first principal direction of the data matrix X: to get an intuition of why this could be a good choice, consider the following reasoning.
Any matrix X can be decomposed in its singular value decomposition

$$X = USV^T$$

where U is an $m \times m$ unitary matrix, S is a $m \times n$diagonal matrix and V is an $n \times n$ orthonormal matrix. The column vectors of V are called principal directions of X. Imagine that the data points are generated in an i.i.d. fashion from some distribution $\mathcal{D}_X$. Assume that we have preprocessed the data to subtract the mean from each row. The empirical covariance K is given by

$$n \cdot K = \sum_{i=1}^{n} x_i x_i^T = XX^T = USV^T VS^T U^T = USS^T U^T = US_m^2 U^T$$

where S_m is the $m \times m$ diagonal matrix consisting of the m first columns of S. Consider the linear transformation of the features $\hat{X} = U^T X$. The empirical covariance $\hat{K}$ is given by

$$n \cdot \hat{K} = \hat{X}\hat{X}^T = U^T XX^T U = U^T US^2 U^T U = S_m^2$$

This means that $\hat{X}$ is a linear transformation of the features such that the empirical covariance $\hat{K}$ is diagonal i.e. the transformed features are uncorrelated. Furthermore, the first transformed feature is the one with the highest variance, given by the square of the first singular value divided by n: $\frac{1}{n} \cdot (S)_{1,1}^2$. From this point of view, the intuition of why projecting on the first principal component could be a good idea is the following: to each point x, the filtered value $f_P(x)$ represents the value of the component of x with the highest variance in the linearly transformed, uncorrelated feature space.
Another interpretation, less probabilistic but more geometric, is the following: one can imagine the first principal direction as the direction along which the data point cloud is distributed. In the introductory example, it was natural to project the "hand" on the x axis since it was possible to see that the hand was aligned, or "pointing" in the x-direction. Such a choice makes the projections $f(X) \subset \mathbb{R}$ as spread along the real axis as possible (or, in the previous interpretation, with the highest variance as possible) and thus ease the process of finding a suitable cover $\mathcal{B}$.
Another example of filter is the following: $f(x) = ||x||_{L^p}^k$[17]. This is a measure of how much a point is far from the origin, and it is useful when analyzing data that represent deviations form a normal value, centered in the origin. How to choose p and k is not clear however, and several experiments have to be made depending on the particular application to find the combination of (p, k) that gives the best output.

There is a high degree of arbitrariness in choosing an appropriate filter function. An advantage of geometric filters with respect to more general filters is their independence on the particular choice of coordinates used to represent the data matrix X. However, they introduce limited amount of information in the algorithm and they are limited to density and centrality measures since they depend only on the distance d.

1.3.4 Clustering algorithm and ϵ

Our interpretation of clustering together two points consist in considering them to belong to the same "connected component" of the data point cloud. Any clustering algorithm could be used, however the most common in literature are the *linkage algorithms*. Linkage is an algorithm where at each step clusters are merged together into bigger and bigger ones, until there are no clusters left closer to each other than a threshold ϵ.

Definition 1.31. [*Linkage Algorithm*] Start with defining each element as a cluster on its own. Define a distance $d_L(A, B)$ between two clusters A, B. At each step, the two closest clusters are combined into one, until there are no more clusters close to each other by less than a threshold valued ϵ.

The choice of the cluster distance d_L defines different kind of linkage algorithms. $d_L(A, B) = \min_{x\in A, y\in B} d(x, y)$ defines the **Single Linkage** algorithm; $d_L(A, B) = \max_{x\in A, y\in B} d(x, y)$ defines the **Complete Linkage** algorithm; $d_L(A, B) = \frac{1}{|A|\cdot|B|}\sum_{x\in A}\sum_{y\in B} d(x, y)$ defines the **Average Linkage** algorithm. Whatever choice is made on d_L, different methods can be chosen to determine ϵ. Hereunder two possible methods are presented:

Definition. [*Histogram method*] Given an integer k, create a histogram of k equally spaced bins of the number of *merges* of clusters as function of the threshold value ϵ. Select ϵ to be the threshold corresponding to the first gap in the histogram. If no gap is observed, the parameter k has to be chosen smaller until a gap is observed.

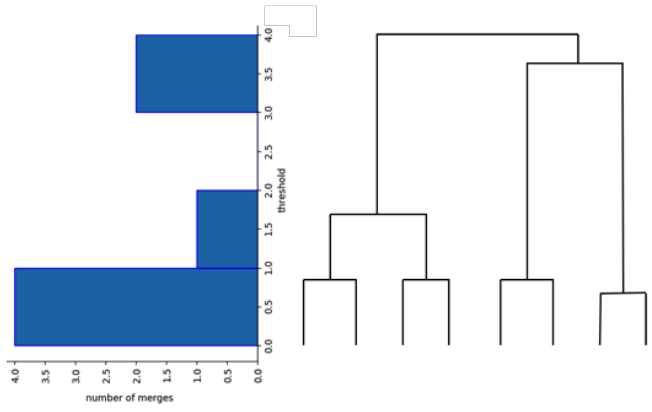


Figure 9: The histogram method

An example is provided in figure 9. On the right, the dendogram of a possible linkage is showed. On the left, the corresponding histogram with $k = 4$ is build. In this example the threshold ϵ would then be set at the value $\epsilon = 2$.

Notice that a high value of k produces a finer histogram, and consequently the first gap will be observed with lower values of ϵ.
Definition. [*Simple gap method*] Proceed with the linkage algorithm chosen. Increase the threshold value linearly in time. As soon as for a fixed amount of time δt new merges of clusters are not observed, the algorithm is stopped.

Notice that both methods need an extra parameter to be chosen, again. For the histogram method k must be fixed, and for the single gap method δt must be fixed. In conclusion, there is no unique way to properly fix ϵ. Some experiments have to be made to find a suitable value for each particular case. Furthermore, there's no unique definition of "suitable": as saw in figure 8, it is hard to judge only by the corresponding different outputs of Mapper when ϵ is "well set". The choice of ϵ is ultimately left to the researcher, who has to try different methods, interpret the result and make a "reasonable" choice.

1.3.5 Distance d

In Mapper, the distance $d : X \times X \to \mathbb{R}$ intervenes *only* in the clustering step of the level sets $f^{-1}(B_i)$. In all the definitions given until now d was always referred to as a metric respecting the three axioms of definition 1.5. However, to successfully execute any of the linkage algorithms described above, it can be observed that is not strictly necessary for d to satisfy the axiom of identity of the indiscernibles i.e. $d(x, y) = 0 \implies x = y$. A function $d : X \times X \to \mathbb{R}$ that satisfies only the properties of positivity and triangle inequality is called a **dissimilarity metric**. When adopting such a dissimilarity metric on X, two points x, y belonging to the same level set such that $d(x, y) = 0$ will always be clustered together, no matter the choice of ϵ. This is useful to filter out some noise in the data. The *mismatch distance* described in [11] that will be presented in Chapter 2 is an example of dissimilarity metric. When defining a dissimilarity metric on X instead of a proper metric, all the definitions given previously in this section can be adapted consequently.
In applications, the notion of metric can be relaxed even more. Indeed, one of the most used "distances" on X is the Pearson correlation distance. By Pearson correlation distance one of the two following definitions can be chosen:

$$d_{\text{Pearson, 1}} : (x, y) \to 1 - |\text{Corr}(x, y)|$$

$$d_{\text{Pearson, 2}} : (x, y) \to 1 - \text{Corr}^2(x, y)$$

where $\text{Corr}(x, y) = \frac{s_{xy}}{s_x s_y}$ and s_x, s_y are the sample standard deviations, and s_{xy} is the sample covariance. Notice that it is possible to come up with examples that prove that these definitions of distance do not respect even the triangle inequality.

2 Previous Implementations

In this section we briefly describe the most used libraries at the moment that implement the Mapper algorithm. In particular, we focus on discussing their architecture and design choices, highlighting the corresponding pros and cons.

2.1 Kepler Mapper

Kepler Mapper Kepler Mapper is a pure Python implementation of the Mapper algorithm. It relies on clustering and filter functions implemented in the popular package Scikit-learn. However, compared to other implementations described further on, it is limited. It implements only one type of cover, and furthermore it does not compute the nerve of such cover, but only its one-dimensional skeleton. In other words, it completely ignores simplices with dimension $d, d >= 1$. Such a one-dimensional skeleton is nothing else than a simple undirected graph.
As a filter function, the user can provide a Scikit-learn API compatible object (an object with a fit_transform() method) or a user-made numpy.ndarray containing the filter values of the data points. As a clustering algorithm, Kepler Mapper uses Scikit-learn API compatible clustering algorithms, that means a class with the following methods: get_params(), fit(), and the following attribute: _labels.

At the present Kepler Mapper is under constant development (as at the 10th of March 2019 the GitHub repo has 482 commits, 8 branches, 6 releases, last published commit the 6th of March 2019).

Kepler Mapper implements a very effective visualization of the output graph. The package provides a base HTML file, and uses Jinja2 to modify such a base HTML file with the data contained in the output graph to produce a new HTML file that renders the output graph.

Architecture This package (as in September 2018) is implemented in an OOP fashion, and consists of only three classes: KeplerMapper, Cover, GraphNerve. Figure 12 illustrates this simple architecture. This architecture is really simple if compared to the more complex one of Python Mapper (a second implementation of Mapper) illustrated in figure 13.

Example We present an easy use-case example of this package, discussing how the steps of Mapper were implemented in the package.

```
# Import the class
import kmapper as km

# Some sample data
from sklearn import datasets
data, labels = datasets.make_circles(n_samples=5000, noise=0.03, factor=0.3)

# Initialize
mapper = km.KeplerMapper(verbose=1)

# Fit to and transform the data
```

```
12 projected_data = mapper.project(data, projection=[0,1]) # X-Y axis
13
14 # Create dictionary called 'graph' with nodes, edges and meta-information
15 graph = mapper.map(projected_data, data, nr_cubes=10)
16
17 # Visualize it
18 mapper.visualize(graph, path_html="make_circles_keplermapper_output.html",
19 title="make_circles(n_samples=5000, noise=0.03, factor=0.3)")
```

Listing 1: Kepler Mapper example

- Step 1: evaluating the filter function f.
  The filter values $f(X)$ are calculated in line 12. Internally, the KeplerMapper object uses functions imported from Numpy and Scikit-Learn to calculate the projection on the x-y axis of the data points. The user is just requested to insert as arguments of the *project*() method a list of indices identifying the axis onto which to project the data.

- Step 2: computing the pullback cover $\mathcal{B}$.
  This step is done in line 15. The default cover implemented in Kepler Mapper is a uniform cover with 10 intervals and a percentage of overlap of 20%. The user can modify this clustering algorithm by passing to the method *KeplerMapper.map*() a Scikit-learn API compatible clusterer in the following way:

```
1 from kmapper.cover import Cover
2 graph = mapper.map(projected_data, data, nr_cubes=10, coverer=Cover(
    nintervals=15, perc_overlap=0.4))
```

- Step 3: clustering the pre images $\{B_i\}$.
  This step is done again in line 15. The default clustering scheme implemented in Kepler Mapper is DBSCAN, implemented in *sklearn.cluster.DBSCAN*(). The user can modify this clustering algorithm by passing to the method KeplerMapper.map() a Scikit-learn API compatible clusterer in the following way:

```
1 custom_clusterer = sklearn.cluster.DBSCAN(eps=0.5, min_samples=3)
2 graph = mapper.map(projected_data, data, nr_cubes=10, clusterer=
    custom_clusterer)
```

- Step 4: building the nerve complex.
  This step is done again in line 15 and the nerve complex is stored as a Python dictionary returned by the method *KeplerMapper.map*(). Such Python dictionary has the following keys: "nodes", "links", "simplices", and "metadata" where *graph*["*links*"] and *graph*["*simplices*"] are return values of a call to a GraphNerve functor.

- Step 5: visualizing the graph.
  Line 18. *KeplerMapper.visualize*() is a really interesting method. In 35 lines of code it converts the information contained in the graph dictionary to a very effective HTML/D3.js visualization thanks to Jinja templates. See [22] for the source code. See figure 11 for an example.

Figure 10: KeplerMapper Github statistics

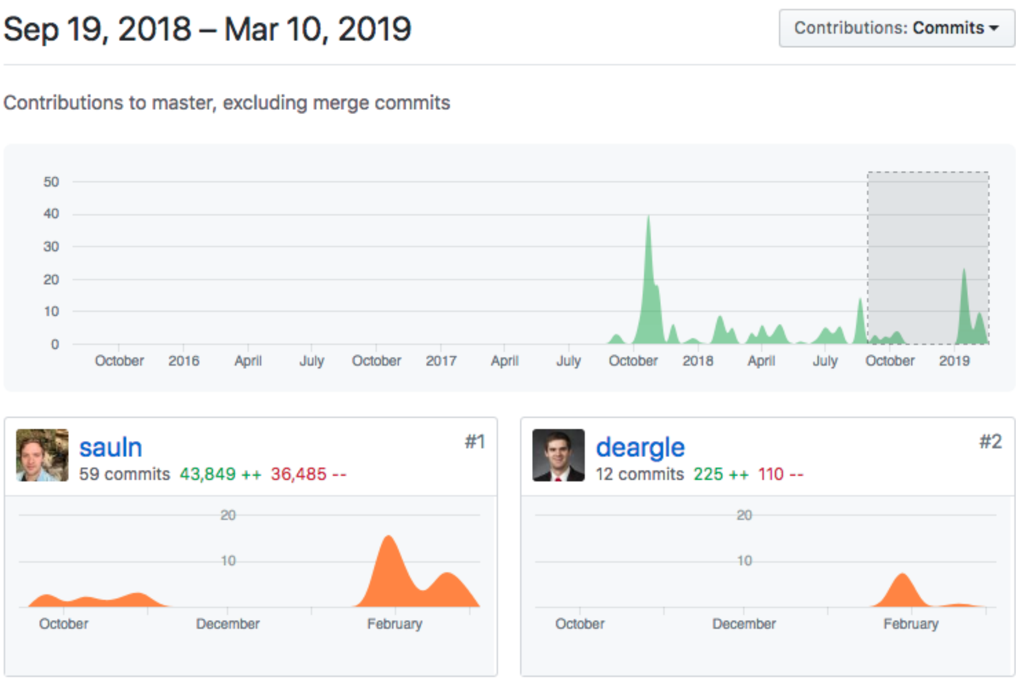
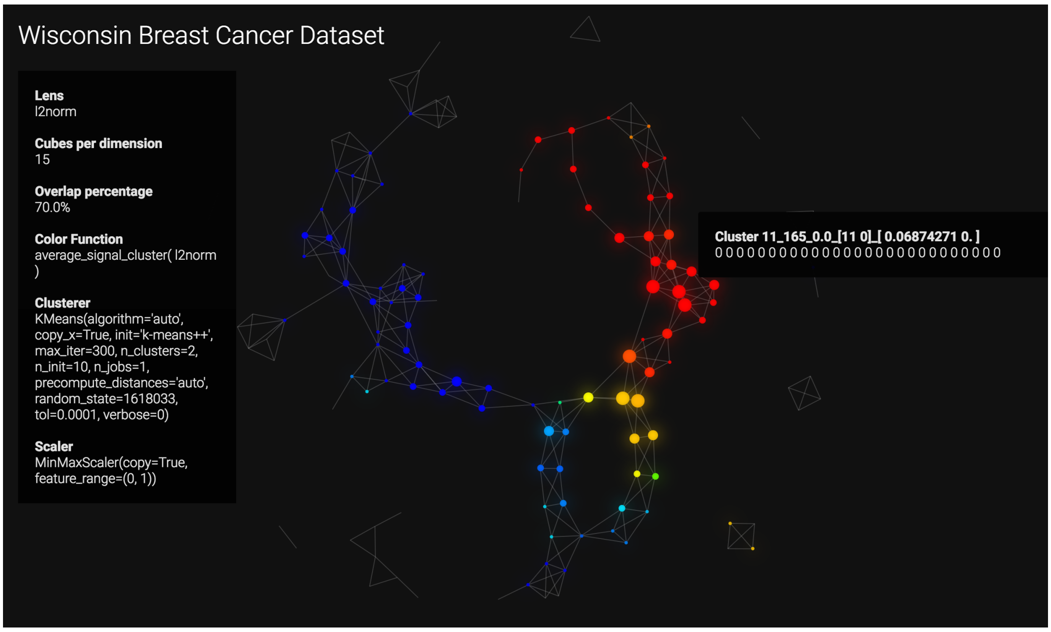


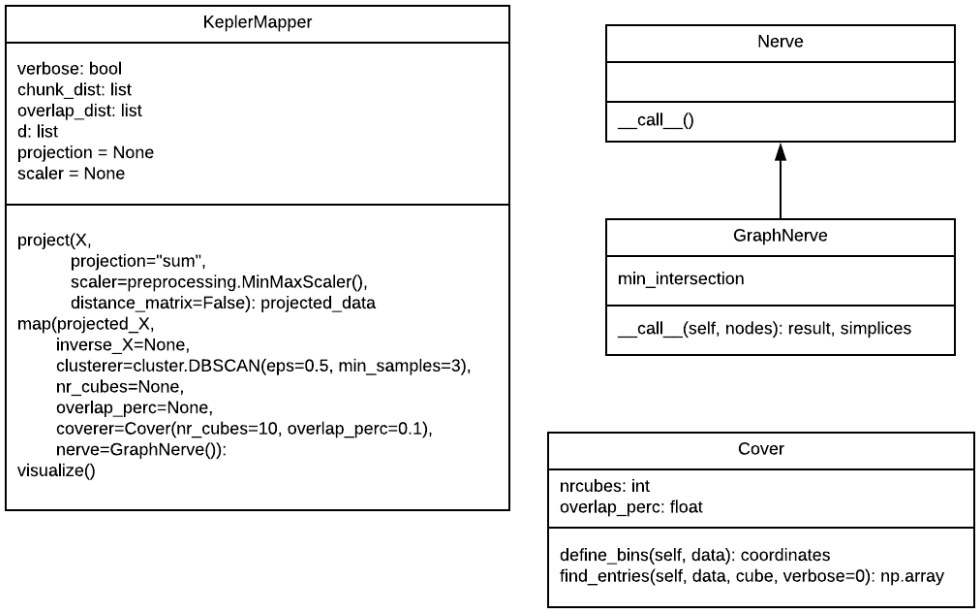
Figure 11: KeplerMapper output example



2.2 Python Mapper (PyMapper)

Python Mapper is a Python package developed by Daniel Müllner as his PhD thesis. It comes with two companion packages written in C++: *cmappertools* and *fastcluster*. Compared to Kepler Mapper, it is expected to be:

Figure 12: Kepler Mapper Architecture (as in September 2018)



- Comprehensive, thanks to the implementation of more filters, more covers.
- Performant, thanks to a custom C++ backend and thus compiled code and efficient multi threading.
- Easy to use, thanks to the graphical interface provided.

However, this package has the following cons:

- It is outdated and not maintained any more (Mapper: current release dated April 19, 2017. Cmappertools: current release dated March 2, 2016)
- The code is not documented. A little documentation is provided for the use of the grapical interface, but there's no documentation about the native Python API.
- No examples are provided with the code. However, it is possible to see the code generated by the GUI while playing around with simple datasets. An example of code generated by the GUI is given in Listing 2
- Code's architecture is in my opinion messy and poorly designed: there is no common interface of different filter classes and cover classes. There's a gigantic class *MapperOutput* that is almost impossible to handle, given the many methods it offers. The code is organized as a mix of object oriented programming and functional programming making it really hard to understand how a modification will affect the whole system. The API of the package is not clear: in other words, it is not clear what is exposed to the user and what should be private.
- C++ code is really hard to understand, since the original C-Python API is used instead of binding libraries such as PyBind11 (used in lmapper's companion package fastfilter). This

makes the binding code really verbose and implementation-dependent.

- It's really hard to install because of deprecated dependencies.
- Some C++ implementations of clustering algorithms in *fastcluster* are now obsolete, since they have been included in more recent releases of SciPy.

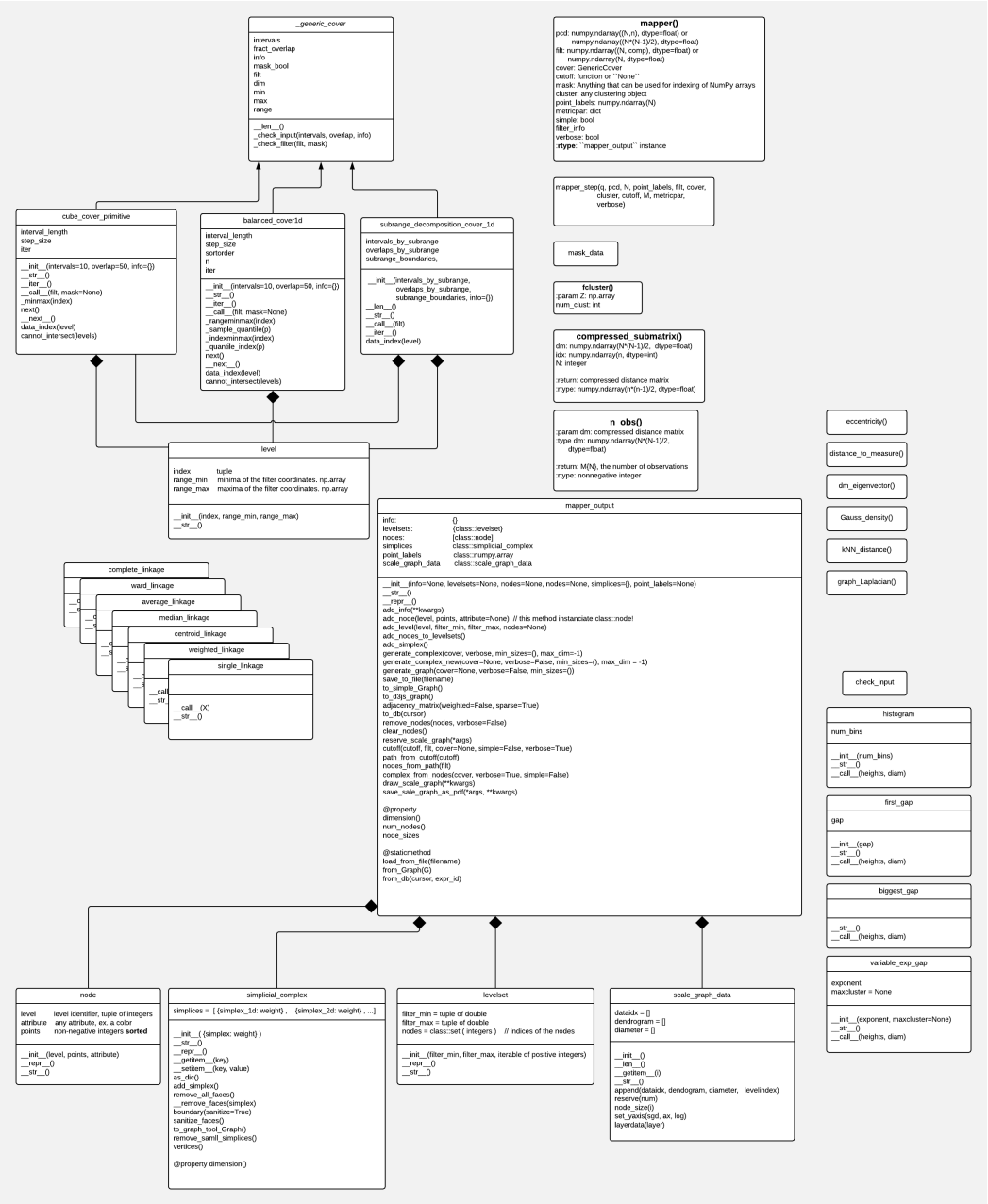
In figure 13 a simplified UML class diagram of the core code of PyMapper is showed.

```
'''
    Python Mapper script
    Generated by the Python Mapper GUI
'''
import mapper
import numpy as np
import matplotlib.pyplot as plt

'''
    Step 1: Input
'''
import gzip
filename = '/Users/martinomilani/Documents/mapper/exampleshapes/cat-reference.csv
    .gz'
with gzip.open(filename, 'r') as inputfile:
    data = np.loadtxt(inputfile, delimiter=',', dtype=np.float)
# Preprocessing
point_labels = None
mask = None
Gauss_density = mapper.filters.Gauss_density
kNN_distance  = mapper.filters.kNN_distance
crop = mapper.crop
data, point_labels = mapper.mask_data(data, mask, point_labels)
'''
    Step 2: Metric
'''
intrinsic_metric = False
if intrinsic_metric:
    is_vector_data = data.ndim != 1
    if is_vector_data:
        metric = Euclidean
        if metric != 'Euclidean':
            raise ValueError('Not implemented')
    data = mapper.metric.intrinsic_metric(data, k=1, eps=1.0)
is_vector_data = data.ndim != 1
'''
    Step 3: Filter function
'''
if is_vector_data:
    metricpar = {'metric': 'euclidean'}
    f = mapper.filters.eccentricity(data,
        metricpar=metricpar,
        exponent=1.0)
else:
    f = mapper.filters.eccentricity(data,
        exponent=1.0)
# Filter transformation
```

```
47 mask = None
48 crop = mapper.crop
49 '''
50     Step 4: Mapper parameters
51 '''
52 cover = mapper.cover.cube_cover_primitive(intervals=15, overlap=50.0)
53 cluster = mapper.single_linkage()
54 if not is_vector_data:
55     metricpar = {}
56 mapper_output = mapper.mapper(data, f,
57     cover=cover,
58     cluster=cluster,
59     point_labels=point_labels,
60     cutoff=None,
61     metricpar=metricpar)
62 cutoff = mapper.cutoff.first_gap(gap=0.1)
63 mapper_output.cutoff(cutoff, f, cover=cover, simple=False)
64 mapper_output.draw_scale_graph()
65 plt.savefig('scale_graph.pdf')
```

Listing 2: PyMapper example automatically generated from the GUI

Figure 13: PyMapper Architecture (simplified)



3 Lmapper

3.1 Intro

The architecture has been inspired to the previous implementation of Daniel Müllner [15]. Compared to Kepler Mapper, this architecture is complex but more modular. However, the API of [15] is really complex to use, and thus we aimed to design a simpler API similar to Kepler Mapper's one.
The approach followed for the development of this package was a test-driven one: first a basic test for the package was written, and then the code necessary to make it work was developed. Once the first test was passed, this procedure was iterated with a more complex test containing more advanced features. This approach was chosen since in this way the development was always guided by a specific goal (pass the test). We present here under the first two tests that were designed and we comment them.

```
import lmapper as lm
from lmapper.filter import Projection
from lmapper.cover import UniformCover
from lmapper.cluster import Linkage
from lmapper.cutoff import FirstGap
from lmapper.datasets import cat

def main():
    data = cat()
  filter = Projection(ax=0)
  cover = UniformCover(nintervals=15,
                       overlap=0.4)
  cutoff = FirstGap(0.05)
  cluster = Linkage(method='single',
                    metric="euclidean",
                    cutoff=cutoff)
  mapper = lm.Mapper(data=data,
                       filter=filter,
                       cover=cover,
                       cluster=cluster)
  mapper.fit()
  mapper.plot()


if __name__ == "__main__":
  main()
```

Listing 3: First test for Mapper

This test shows the main components and design choices of the package.

Import statements The code is structured in a pure OOP fashion. The package is composed by five base modules: _mapper.py, filter.py, cover.py, cluster.py, cutoff.py, datasets.py. Each module contains the definitions of the classes implementing the corresponding algorithms.

main() First, one object for the filter, one object for the cover, one object for the cutoff method used in the clustering step, one object for the clustering step are instantiated. Given the many parameters of Mapper, in this way each step of the Mapper pipeline is isolated in an object that contains its set of parameters, avoiding long lists of parameters that could make the API confusing.
These objects (filter, cover, cluster) are then used to instantiate a mapper object together with the data that has to be analyzed. The mapper object has to expose to the client user two methods: fit() and plot().
It can be seen that such test looks more readable that the code (in Listing 2) necessary for PyMapper, but however it lets the user set all the parameters that Kepler Mapper hides and does not let the user choose. The package is clearly modular: if in the future more algorithms would be necessary (i.e. a new way of building a cover, or a new clustering algorithm) it would be necessary to implement a corresponding new class respecting the API to correctly interact with the other modules, expose it to the user and all the future patches of the package would make all the old client code still compatible.

```
import lmapper as lm
from lmapper.filter import Projection
from lmapper.cover import UniformCover
from lmapper.cluster import Linkage
from lmapper.cutoff import FirstGap
from lmapper.datasets import cat

def main():
  filter = Projection(ax=0)
  cover = UniformCover(nintervals=15,
                       overlap=0.4)
  cutoff = FirstGap(0.05)
  cluster = Linkage(method='single',
                    metric="euclidean",
                    cutoff=cutoff)
  data = cat()
  mapper = lm.Mapper(data=data,
                       filter=filter,
                       cover=cover,
                       cluster=cluster)
  mapper.fit()
  mapper.plot()

  cluster = Linkage(method='average',
                    metric='correlation',
                    cutoff=cutoff)
  mapper.set_params(cluster=cluster)
  mapper.fit()
  mapper.plot()


if __name__ == "__main__":
  main()
```

Listing 4: Second test for Mapper

This test was designed to solve the following issue: Mapper has a lot of parameters, and to set them correctly a lot of trial-and-error has to be done by the user to find a good choice of parameters. All the previous implementations up to our knowledge, when a parameter is changed, need to recalculate everything from scratch, making the trial-and-error procedure really slow. However, Mapper is an algorithm that has the following clear pipeline:

Mapper pipeline

1. Calculate the filter values
2. Calculate the pull-back cover
3. Cluster each element of the pullback cover
4. Build the simplicial complex from the elements of the pullback cover

Each one of these steps is completely independent from the previous ones; this means that if we fit a Mapper object, and then we decide to try to fit it changing of the clustering method, there's no need to recompute the filter values and the pullback cover, and the computations could restart from step (3). To enable this we need to design the Mapper class to store all the intermediate information needed to restart the computation from whichever step, and make it "aware" of the changes to the previous set of parameters through the method set_params().

Base architecture In figure 14 the architecture for a first implementation of the package necessary to pass the first two tests is illustrated. The final version of the package is more complex, but this is sufficient to illustrate the high level design choices that were respected during the whole development.

Three base classes *class Filter, class Cover, class Cluster* defining the API of the backend of the package are defined, one for each of the first three steps of the Mapper pipeline. Every class implementing an algorithm for a pipeline step has to derive from one of these classes, in order to be compatible with the package. *class Complex* is responsible for the fourth and last step of the Mapper pipeline. The classes exposed to the user would be only *class Mapper, class Projection, class BalancedCover, class Linkage*. All the other classes, explained in detail in Section 3.2, will be part of the backend of the package, and thus should not be used in any client code.

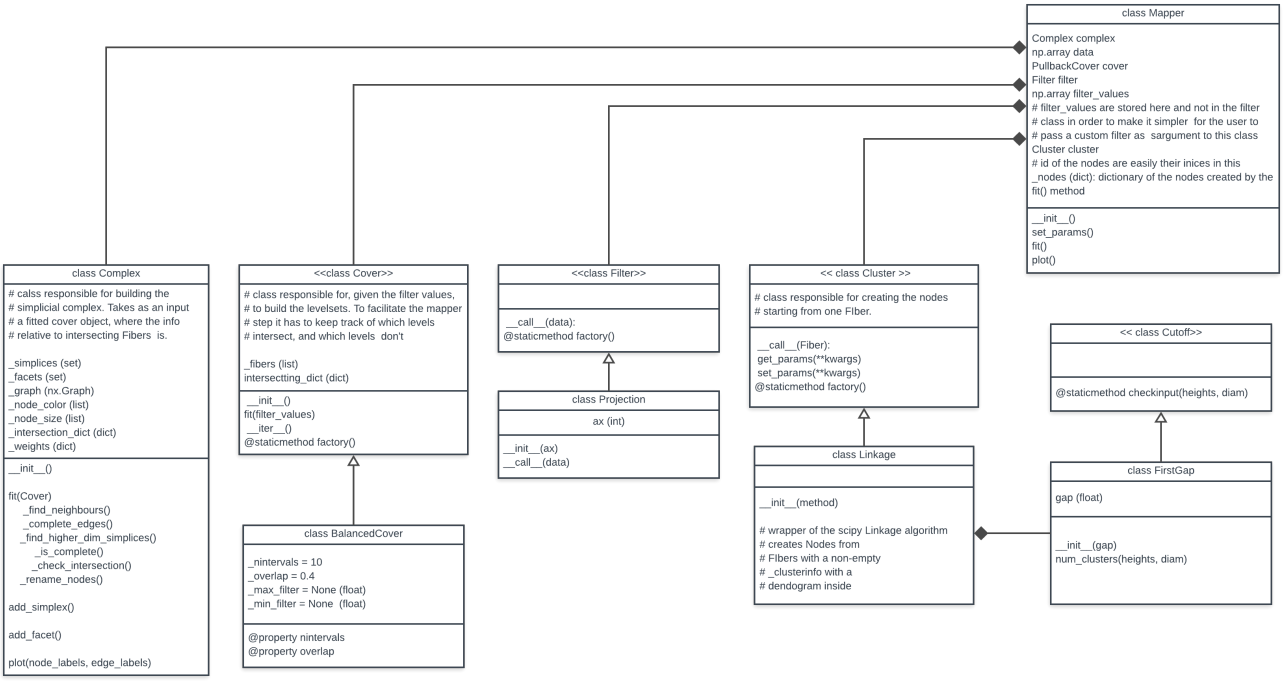
3.2 Description of the classes

In figure 15 we present the UML graph of the final version of the package. In this section we present the classes in a top-to-bottom order in order to first present the classes implementing the most high level algorithms, and only at the end present the classes implementing the low level details.

3.2.1 *class Mapper*

```
1 import lmapper as lm
2 from lmapper.filter import Projection
```

Figure 14: Simplified Mapper Architecture (simplified)

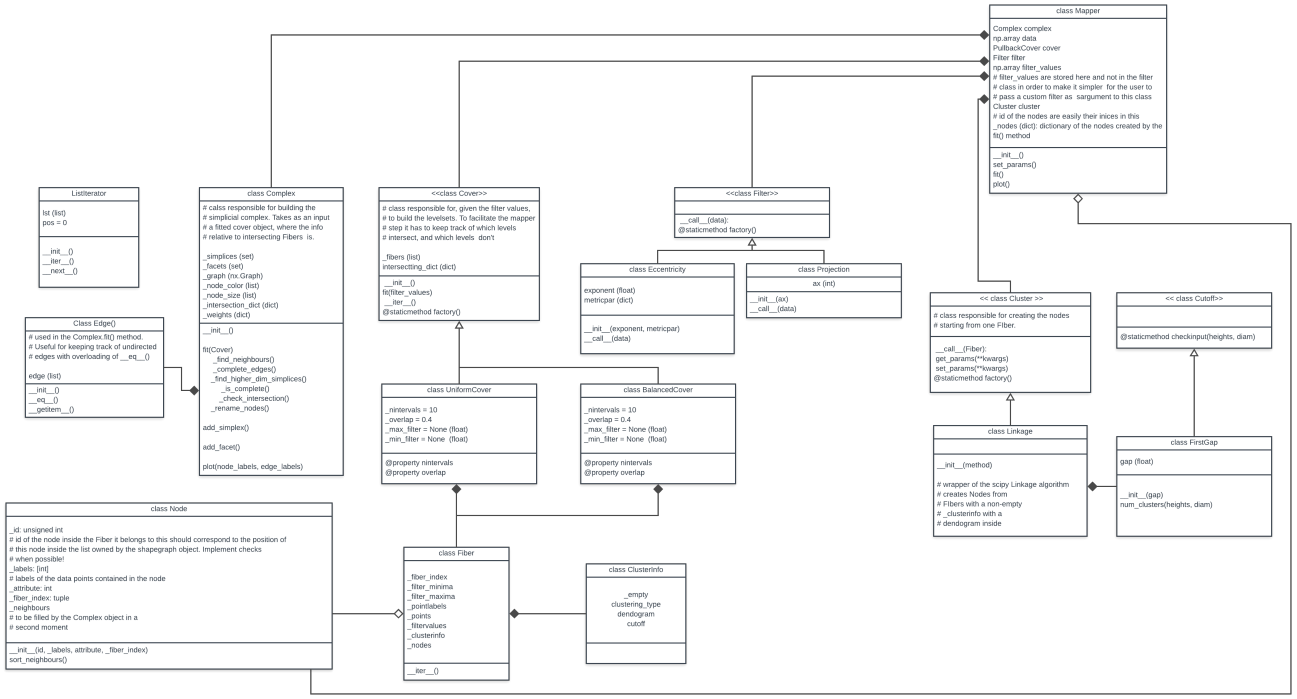



```
from lmapper.cover import UniformCover
from lmapper.cluster import Linkage
from lmapper.cutoff import FirstGap
from lmapper.datasets import cat

data = cat()
filter = Projection(ax=0)
cover = UniformCover(nintervals=15,
                     overlap=0.4)
cluster = Linkage(method='single',
                  cutoff=FirstGap(0.05)
mapper = lm.Mapper(data=data,
                   filter=filter,
                   cover=cover,
                   cluster=cluster)
mapper.fit()
mapper.plot()
```

Listing 5: Example for the Mapper class

```
import lmapper as lm
from lm.datasets import cat

data = cat()
mapper = lm.Mapper(data=circles)
mapper.fit()
```

Figure 15: Full Mapper Architecture




```
mapper.plot()
```

Listing 6: Example 2 for the Mapper class. The Mapper object can be initialized just with the data matrix.

```
import lmapper as lm
from lm.datasets import cat

data = cat()
mapper = lm.Mapper(data=data,
                   filter='Projection',
                   cover='UniformCover',
                   cluster='Linkage')
mapper.fit()
mapper.plot()
```

Listing 7: Example 3 for the Mapper class. The arguments of the init function can be strings.

```
import lmapper as lm
from lmapper.cover import UniformCover
from lmapper.cluster import Linkage
from lmapper.cutoff import FirstGap
from lmapper.datasets import cat

def ProjectionOnFirstCoordinate(x):
  return x[:, 0]

cover = UniformCover(nintervals=15,
```

```
                        overlap=0.4)
cluster = Linkage(method='single',
                  cutoff=FirstGap(0.05)
data = cat()
mapper = lm.Mapper(data=data,
                   filter=ProjectionOnFirstCoordinate)
mapper.fit()
mapper.plot()
```

Listing 8: Example 4 for the Mapper class. The filter given to the init function of the Mapper object can be a Python function.

As shown in the example the init method requires four arguments: the data matrix as a two-dimensional numpy array, where the $n-th$ row represents the coordinates of the $n-th$ data point; the filter object, the cover object, and the cluster object.
We present now the main original features of this implementation of the Mapper algorithm, that haven't been implemented neither in Kepler Mapper neither in Python Mapper.

Optimized calls to fit() The main feature of this implementation is that it is optimized for minimizing the computational time required for multiple calls to the fit() method. Since there's no way of algorithmically test the goodness of a Mapper parameters set, the only way to find a good combination of parameters is to manually explore the parameter space and observing the corresponding output graph. This is illustrated in the following example.

```
import lmapper as lm
from lmapper.filter import Projection
from lmapper.cover import UniformCover
from lmapper.cluster import Linkage
from lmapper.cutoff import FirstGap
from lmapper.datasets import cat

filter = Projection(ax=0)
cover = UniformCover(nintervals=15,
                     overlap=0.4)
cluster = Linkage(method='single',
                  cutoff=FirstGap(0.05)
data = cat()
mapper = lm.Mapper(data=data,
                   filter=filter,
                   cover=cover,
                   custer=cluster)
mapper.fit()
mapper.plot()

# we want to see what happens if we change the clustering step
# from a Single Linkage to an Average Linkage,
# mantaining all the rest of the Mapper parameters equal.
# For this purpose we use the set_params() method

mapper.set_params(cluster= Linkage(method='average', cutoff=FirstGap(0.05)))

# After having changed the parameters, we need to fit the Mapper object
mapper.fit()

```

```
31 # Now we can plot the new mapper graph to compare it with the previous one
32 mapper.plot()
```

Since in practice it is normal to alternate many calls to fit() and set_params() to find a good parameter set, the fit() method has been implemented in order to avoid to alway recompute everything from scratch. The Mapper algorithm has indeed a clear pipeline presented in section 2.1. If as in the example above we change only the parameters of the clustering step, it is useless to recompute the filter values and the cover, already calculated before. For this reason, the Mapper class keeps track internally of its state, what has changed compared to the previous state, and the new call to the fit() method performs only the computations needed. In the example, only the clustering step will we done before computing the output graph.

3.2.2 filter.py

This module implements some common filters.

class filter.Filter Python do not provide the opportunity of defining abstract classes. However, given the non trivial complexity of the package, we chose to the define anyway a base class for each step of of the Mapper pipeline to provide a clear way of understanding the interface of the backend API. *Each one of these four classes* class filter.Filter, class cover.Cover, class cluster.Cluster, class cutoff.Cutoff *was written in order that every time a new filter, cover, cluster, cutoff has to be implemented, it will be implemented as a class derived from one of these corresponding base classes.* For example, a projection filter has to be implemented as child class of the corresponding Filter class: class Projection(Filter). The programmer that will be expanding the package by adding a new filter, will have to read the code of the Filter class to see which methods have to be overridden, and which methods are already provided in the base class. In this way these four base classes act as a guideline for the programmer to make sure that every newly added class will respect the backend API of the package.
The class Filter simply declares the __call__() operator raising a NotImplementedError exception, indicating to the programmer that any new filter has to override such a method. Since Python is not type-checked (although the package **typing** could be used from Python 3.5) it was chosen to put clear indications of the expected parameter and the return value type In the docstring of the filter.Filter.__call__() method. The same procedure has been used for all the methods of the thee other base classes class filter.Filter, class cover.Cover, class cluster.Cluster, class cutoff.Cutoff raising a NotImplementedError exception.
A static factory method is implemented. Note that the code of filter.Filter.factory() does not have to be updated every time a new child class is added to the package thanks to the use of the globals() method.

class filter.Eccentricity(filter.Filter) This implements the eccentricity filter as presented in [15]. In short, the eccentricity filter value of the $n-th$ data point is the p-norm of the $n-th$ row of the distance matrix

$$Eccentricity(x_i; p) = \left(\sum_{k=0}^{N} d(x_i, x_k)^p \right)^{\frac{1}{p}}$$

As explained above in section 3.2.2 we can see that this class is a child of filter.Filter. It implements an __init__() method initializing the attributes that every cover object must have and overrides the method __call__() as indicated in the docstring of the same method of the base class filter.Filter.__call__().
Since the computation of the eccentricity is computationally very expensive because of the calculation of the full distance matrix, the implementation of the Python functions eccentricity() and my_distance() were implemented in C++ and are available in the module **filterutils**. They provide an efficient parallel implementation of the most CPU-intensive part of the algorithm through the use of OpenMP.
When the package **lmapper** is imported, the module **filterutils** is imported. If the latter import fails, the user is notified about the failure with a printed message on the terminal and a slower, fully Python implementation of the functions eccentricity() and my_distance() provided in the **filter.py** module is used instead.
A last method, filter.Eccentricity.for_assignment_only() is defined. However, this method is meant to be used from the companion package **predmap** and not to be exposed to the final user of the package. This implementation is however not clean and prone to bugs, since it makes the implementations of the two packages profoundly interdependent. This method is necessary as the filter.Filter.__call__() method is meant to operate on two-dimensional np.ndarray representing the whole data matrix in order to vectorize as much as possible the computations to achieve speedup. However, we'll see that in **predmap** it is necessary to compute the filter values *of only one new data point belonging to the test set after having already computed the filter values of the whole original data matrix of the training set.* These two different situations of calculating filter values require two different implementations. It is not clear if the implementation needed in the second scenario was to be implemented in **predmap** or in **lmapper**. For now it has been chosen to add this method filter.Eccentricity.for_assignment_only(). A more elegant solution is left as further work.

3.2.3 cover.py

This module defines some classes to implement different ways of creating a pullback cover.

class cover.Cover As explained above in section 3.2.2 class cover.Cover is a base class to define the common interface that any new implementation of a new cover method must respect. Similar to filter.Filter it implements an __init__() method initializing the attributes that every cover object must have. It implements a method cover.Cover.find_intersecting_dict() that outputs a dictionary that keeps track of which fibers intersect. This implementation is valid as soon as the filter values are one dimensional, and each Fiber object represents is the counter image of an interval on the real line. When the package will be expanded with multi-dimensional filters, or with more complicated ways of building a cover, this method will become obsolete and must be changed.

class cover.UniformCover(OverlapCover) It derives from class OverlapCover(Cover). This latter class was introduced since class UniformCover(OverlapCover), KeplerCover(OverlapCover), BalancedCover (OverlapCover) share the common idea of being defined by overlapping intervals on the image of

the filter $f(X) \subset \mathcal{R}$. We can see that class cover.UniformCover(OverlapCover) just needs to define the cover.UniformCover.fit() method, in which the particular construction of this specific cover is implemented. Note that "constructing a cover" for an OverlapCover means finding the variables list_of_as and list_of_bs that are respectively the list of the leftmost extremes and the rightmost extremes of the intervals $I_j = (a_j, b_j)$ forming the open cover $\mathcal{I} = \{I_j\}$ of $f(X)$. Once defined these intervals, all the rest of the implementation is taken care of by the method cover.OverlapCover.find_entries() that finds the corresponding data points belonging to each preimage $f^{-1}(I_j)$ and instantiating an object of class cover.Fiber for each preimage $f^{-1}(I_j)$. The output of cover.OverlapCover.find_entries() is a list of cover.Fiber objects, and a dictionary created by the cover.Cover.find_intersecting_dict() method.

class cover.Fiber class cover.Fiber is an iterable data structure containing informations relative to each preimage A_i of the pullback cover $\mathcal{A} = \{A_i\}$. It stores the indices of the data points that the corresponding preimage B_i contains and their filter values. The implementation of such a data structure could have been avoided. In fact, it stores on memory information that is duplicated in the system (in the Mapper object), and that could have been easily retrieved at run time each time it was needed. However, implementing such data structure and duplicating the information makes the algorithm faster and the implementation easier, at the cost of an increased memory requirement. This choice was made also to ease the implementation of the package predmap: such package will have to access these informations fast and often, thus the increased memory price is negligible compared to the increased simplicity of the code and to the consequent speedup.

3.2.4 complex.py

This module implements the classes needed to create the Nerve Complex output of the Mapper algorithm.

class complex.Node Simple data structure responsible for storing the information of one node of the final complex, result of the cluster call on a Fiber object. To be clear, at the moment of instantiating a cover.Fiber object the attribute cover.Fiber._nodes is an empty list. After having called the method cluster.Linkage.__call__(cover.Fiber) the attribute cover.Fiber._nodes becomes populated by a list of objects complex.Node, each one corresponding to a cluster found by the clustering method.

class complex.Complex Class responsible for storing all the simplices, edges, nodes of the Nerve Complex output of the Mapper algorithm. It also implements a method for drawing te 1-skeleton of the nerve complex. It implements four method: complex.Complex.fit(), complex.Complex.add_simplex(), complex.Complex.add_facet(), complex.Complex.plot().
While the complex.Complex.add_simplex(), complex.Complex.add_facet() methods are trivial, we deepen the details of the more complicated complex.Complex.fit() method. Starting from a cover.Cover object, the complex.Complex.fit(cover.Cover) method has access to all the fiber.Fiber objects stored

inside the cover.Cover object, and thus to all the complex.Node objects stored in the different cover.Fiber._nodes attribute. complex.Complex.fit() looks for intersections between all these complex.Node objects, creating the corresponding Nerve Complex, as explained in section 1.2. Such Nerve complex is stored as a list of simplices (a simplex is represented as a tuple of int, where each int is the id of a complex.Node) in the complex.Complex._simplices attribute. The corresponding 1-d skeleton is stored in the networkx.Graph attribute complex.Complex._graph. The choice of using a networkx.Graph is due to the fact that this class implements a easy to use drawing method networkx.Graph.draw() that we can reuse providing the complex.Complex class with a method complex.Complex.plot() that simply wraps the networkx.Graph.draw() method.

3.2.5 cluster.py

This module implements some clustering algorithms for the clustering step of Mapper's pipeline.

class cluster.ClusterInfo Simple data structure containing information about the parameters of the specific clustering algorithm used to produce the complex.Nodes. Objects of class cluster.Clusterinfo are instantiated by a cluster call on a fiber (cluster.Linkage.__call__(cover.Fiber)) and are owned by the corresponding cover.Fiber objects in the attribute cover.Fiber._clusterinfo.

class cluster.Cluster As explained above in section 3.2.2 class cluster.Cluster is a base class to define the common interface that any new implementation of any new cluster method must respect. The interface is simple: as we introduced before these objects must have a __call__ method, and the two common get_params and set_params methods.

class cluster.Linkage Example of cluster.Cluster child class. The cluster.Linkage.__call__ method takes a cover.Fiber object, accesses to its data points and cluster them wrapping a scipy.cluster.hierarchy.linkage object. Each cluster is then represented by one complex.Node object, that is owned by the corresponding fiber in its attribute cover.Fiber._nodes.

3.2.6 cutoff.py

This module implements the methods described in [17] to determine where to cut the dendrogram produced by a linkage algorithm to define the optimal clusters for the Mapper algorithm.

class cutoff.Cutoff As explained above in section 3.2.2 class cutoff.Cutoff is a base class to define the common interface that any new implementation of any new cutoff method must respect.

class cutoff.FirstGap This class implements the First Gap method illustrated by Carlsson et al. in [17]. It is used by the cluster.Linkage class to determine the number of clusters to create starting from the full linkage matrix.

4 PredMap

4.1 Intro

predmap implements a method for building binary classifiers based on the Mapper clustering algorithm, ad described in [**?**].

Given a finite set X and real-valued function $f : X \to \mathbb{R}$, Mapper represents X as a graph whose nodes are subsets of X. Typically, X is viewed as a metric subspace of an ambient metric space $(\mathbb{M}, d)$, and the Mapper nodes are found by means of clustering algorithms exploiting this metric structure.

Assuming ground truth labels to be known for all points in X, a simple binary classification model on $\mathbb{M}$ based on Mapper may be defined as follows: first, a Mapper graph is built on X and – up to ambiguities which may arise when different Mapper nodes intersect – the model's output on each point in X is declared to be the majority label within the Mapper node containing that point. Then, given a choice of mapping between $\mathbb{M} \setminus X$ and the set of Mapper nodes, the model's estimate for $x \in \mathbb{M} \setminus X$ is declared to be the majority label within the assigned node. The algorithm is improved upon this approach by a method for refining initial labels in the training phase, effectively decomposing Mapper nodes into regions of high purity. Furthermore, an assignment procedure which is in keeping with the spirit of Mapper is implemented.

4.2 The algorithm

Binary classification with Mapper: a first approach

Let $\mathcal{D}$ be a training dataset for a binary classification problem, i.e. $\mathcal{D}$ comprises a sequence X of points in, say, a topological space and a corresponding sequence Y of known class labels belonging to $\{0, 1\}$. Suppose a clustering method on X is given, producing a collection of subsets $\mathcal{C} = \{C_i : \quad C_i \subset X, \quad \bigcup_i C_i = X\}$, together with a rule $\alpha : X \to \mathcal{C}$ for assigning new points to the clusters of X. A simple classifier $h : X \to \{0, 1\}$ can then be defined as follows:

1. Each cluster is assigned the most common class label among its constituents i.e. if the data points in the cluster C_0 are mostly belonging to the class labeled with 1, then the *majority class* of cluster C_0 will be $\text{majority}(C_0) = 1$.
2. A new point x is assigned to a cluster C through a well defined mapping $\alpha : \mathbb{M} \to \mathcal{C}$ to a cluster $C = \alpha(x)$. The point x is then classified as follows:

$$h(x) = \text{majority}(C) = \text{majority}(\alpha(x))$$

Thus we may construct a first Mapper-based classification algorithm provided we complement Mapper with a suitable such rule α.

Because of the fact that Mapper produces a set of overlapping clusters $\mathcal{C} = \{C_i\}$, a first phase called *disambiguation* is introduced in the algorithm such that from the initial set of overlapping

clusters $\mathcal{C}$ produces a set of disjoint clusters $\mathcal{C}' = \{C'_i : C'_i \subset C_i, \quad \bigcup C'_i = X, \quad C'_i \cap C'_j = \delta_{ij}\}$. For simplicity we'll use the notation $\mathcal{C}$ to refer to the *disambiguated* partition $\mathcal{C}'$.

Binary classification with Mapper: how to improve

Assuming that $\mathcal{C}$ partitions X, we seek to improve our model by defining some exceptions to rule 2 of section 4.2. This means that we look for special, well defined cases where the predicted label of a point x belonging to the test set will be *the opposite* of majority$(C = \alpha(x))$. We begin by finding regions that contain a high proportion (over 50%) of misclassified points.

For $x \in X$ we let $I_{x,0} := \{f(x)\} \subset \mathbb{R}$ and, for any $k = 1, \ldots, |\alpha(x)| - 1$, we denote by $I_{x,k} \subset \mathbb{R}$ the minimal closed interval containing the k nearest neighbours of $f(x)$ according the Euclidean metric in $\mathbb{R}$ and among the elements of $f(\alpha(x))$. We call the collection of all such intervals as $\mathcal{I}_x := (I_{x,k})_{k=0}^{|\alpha(x)|-1}$.
Definition 4.1. With $x \in X$ and $1 \leq k \leq |\alpha(x)| - 1$, let

$$C_{x,k} := f^{-1}(I_{x,k})$$

We also let $C_{x,0} := \{x\}$.

An intermediate measure of misclassification severity will be given by the following function:
Definition 4.2. With $x \in X$ and $0 \leq k \leq |\alpha(x)| - 1$, we denote by $\phi_x(k)$, resp. $\tau_x(k)$, the number of points in $C_{x,k}$ misclassified, resp. correctly classified, by majority$(\alpha(x))$. Finally, we define $g_x(k) := \phi_x(k) - \tau_x(k)$.

The k's for which we have positive values of g_x indicate that there is a majority of misclassified points in $C_{x,k}$. This suggests that switching estimated labels on some of the $C_{x,k}$ may be statistically sound. We now introduce a function on X which we call the *density detector*. It will provide us with a way of ranking points in X which will be later used to construct refinements of the original Mapper nodes and perform such switchings.
Definition 4.3. For any $\Gamma \in \mathbb{N}_0$, the density detector $\Delta_\Gamma : X \to \mathbb{R}_{\geq 0}$ with weight $\beta \in \mathbb{R}^+$ is defined as follows. Let $x \in X$, then

$$\Delta(x) = \max_{0 \leq k \leq |\pi(x)|-1} \delta_x(k), \quad \text{where} \quad \delta_x(k) := g_x^+(k)^\beta \frac{\phi_x(k)}{|C_{x,k}|}$$

and g_x^+ is the positive part of g_x. We also let $\hat{k}(x) = \arg\max_{0 \leq k \leq |\pi(x)|-1} \delta_x(k)$ and $\hat{I}(x) := I_{x,\hat{k}(x)}$.

We now give a prescription, depending on a parameter $0 \leq \lambda$, for identifying subsets of each Mapper node in which estimated labels are to be changed.
Definition 4.4. *PredMap classifier* $h(x)$

$$h(x) = \begin{cases} \text{majority}(\alpha(x)) & \text{if } f(x) \notin F_{\alpha(x)} \\ \neg\text{majority}(\alpha(x)) & \text{if } f(x) \in F_{\alpha(x)} \end{cases}$$

where for each cluster C the corresponding set F_C is built as a union of some intervals $\hat{I}(x)$ defined before:

$$F_C = \bigcup_{x \in L_C} \hat{I}(x)$$

where $L_C = \{x \in C : \triangle(x) > \lambda \cdot \text{Score}(C; a)\}$ where $\text{Score}(C) : \mathcal{C} \to \mathbb{R}^+$ is a *score function* (maybe depending on some parameters) that has to be designed such that a cluster C with high score value will correspond to a bigger set L_C, resulting in a higher probability for a point $x : \alpha(x) = C$ to be such that $h(x) = \neg\text{majority}(C)$

4.3 Description of the Classes

4.3.1 _predmap.py

class _predmap.BinaryClassifier Class responsible for the implementation on top of a mapper graph of the binary classification task as designed by Francesco Palma and Thomas Boys

Example of use:

```
import lmapper as lm
from lmapper.filter import Projection
from lmapper.cover import BalancedCover
from lmapper.cluster import Linkage
from lmapper.datasets import synthetic_dataset
import predmap as mapp


filter = Projection(ax=2)
cover = BalancedCover(
                    nintervals=20,
                    overlap=0.4)
cluster = Linkage(method='single')
data, response = synthetic_dataset()
mapper = lm.Mapper(
                  data=data,
                  filter=filter,
                  cover=cover,
                  cluster=cluster)
mapper.fit()

predictor = mapp.BinaryClassifier(
                            mapper=mapper,
                            response_values=response,
                            _lambda=0.4)
predictor.fit()

x0 = [0.3, 0.4, 0.7]

predictor.predict(x0)
```

As it can be seen in the example this class provides to the user two methods: _predmap.BinaryClassifier.fit() and _predmap.BinaryClassifier.predict().

The first method performs all the calculations described in section 4.2. Namely, it performs the *disambiguation* step, builds all the values $\triangle_\Gamma(x), \kappa_\Gamma(x)$ and $I_\Gamma(x)$ for each point x in the training set, and builds for each cluster C the set $F_C = \bigcup_{x \in L_C} I_\Gamma(x)$.

The method _predmap.BinaryClassifier.predict() takes as an input the test set $\mathcal{D}_{test}$, and for each point $x \in \mathcal{D}_{test}$ resolves the assignment $\alpha(x)$ and returns the value $h(x)$:

$$h(x) = \begin{cases} \text{majority}(\alpha(x)) & \text{if } f(x) \notin F_{\alpha(x)} \\ \neg\text{majority}(\alpha(x)) & \text{if } f(x) \in F_{\alpha(x)} \end{cases}$$

4.3.2 disambiguated_node.py

class disambiguated_node.DisambiguatedNode The aim of this class is to extend the class lmapper.complex.Node with attributes and methods necessary to implement the predictive Mapper algorithm. Each object of class predmap.disambiguated_node.DisambiguatedNode represents one of the clusters C of the *disambiguated* partition $\mathcal{C}$. In particular, it has to construct and store the set F_C through the procedure described in the section above. To summarize the responsibilities of this class:

1. each object stores a reference to the corresponding lmapper.complex.Node object that it extends.
2. the method predmap.disambiguated_node.DisambiguatedNode._finduniquelabels performs the disambiguation step.
3. the method predmap.disambiguated_node.DisambiguatedNode._setmajorityvote finds majority(C) and stores some information necessary to compute the score function value for the corresponding node.
4. the method predmap.disambiguated_node.DisambiguatedNode._applyscorefunction calculates $S(C; a)$ where the parameter $a \in [0, 1]$.
5. the method predmap.disambiguated_node.DisambiguatedNode._computeintervals calculates the density detector Δ_Γ and $\kappa_\Gamma(x)$. the method predmap.disambiguated_node.DisambiguatedNode._intervalstoflip finds and stores the set $F_C = \bigcup_{x \in L_C} I_\Gamma(x)$ where $L_C = \{x \in C : \triangle_\Gamma(x) > \lambda \cdot \text{Score}(C; a)\}$

5 filterutils: a C++ extension with the use of OpenMP

filterutils is a Pyhton module written in C++ with the use of the binding library PyBind11 [10] that makes use of OpenMP to provide low-level control of the parallelization of two important CPU-intensive tasks: the computation of the eccentricity filter function. While the final goal would be to implement more and more parts of the algorithm in C++ and with the use of OpenMP, the complexity of developing a Python package in C++ is however non trivial. Although the biggest bottleneck of the algorithm is the clustering step, it was observed that in numerous use cases the computation of the eccentricity represents a big bottleneck of the algorithm, comparable to the one of the clustering step. The computation of the eccentricity is however simpler - with respect to the clustering step - to implement, and simpler to optimize, since the problem is purely its being CPU intensive, while the clustering step, is also heavily memory intensive, and thus requires much more care to be correctly optimized. For these reasons it was chosen to start with the parallelization of the eccentricity, leaving the parallelization of the clustering step for further developments.

Challenges

To write this simple package it was necessary to deal with the following problems.

The binding code When writing a Python function in C++, the steps to follow are the following:

1. Parse the Python objects received as arguments in C objects.
2. perform the computations with the C++ objects to produce other C++ objects.
3. Parse the C++ objects, results of the computation, to construct Python objects to return to the Python interpreter.

The code necessary to perform the steps 1 and 3 is called *binding code*. Python provides a Python/C++ API that however has the following disadvantages: it is really verbose and produces complicated code that is hardly readable. It is hard to use. For these reasons we decided to use PyBind11 [10], a C++ library developed by Wenzel Jacob et al. at EPFL that aims at solving these issues. From PyBind11 documentation, "pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. Its goals and syntax are [..] to minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection."

How to parallelize the computations To parallelize the computations, it was chosen to use OpenMP since the code is supposed to run on multi-core systems and not on clusters, thus making the shared memory abstraction of OpenMP the most coherent with the hardware where the code is supposed to run, as opposed to message passing abstractions.

filterutils.cpp implements - for now - just two Python functions, `my_distance()` and `eccentricity()`. These functions are imported in the `lmapper.filter.py` module and used in the implementation of the class `lmapper.filter.Eccentricity` As a remark, a more complete package, fastfilter.cpp - that in our intention would replace completely the module `lmapper.filter.py` providing a full C++

implementation of the classes in lmapper.filter.py - is under development but unfortunately still bugs at run time. For the interested readers in such package one could find the code necessary to bind C++ classes (and not only functions) to the Python interpreter.

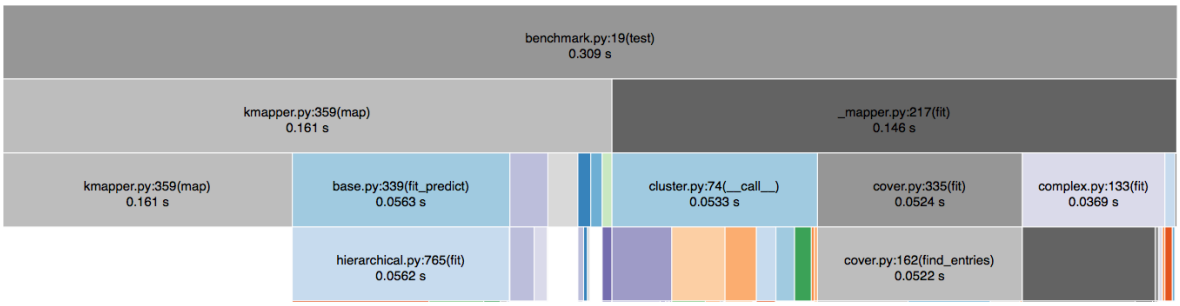
6 Benchmark

6.1 lmapper VS kmapper

We benchmarked lmapper against Kepler Mapper. To benchmark it a specific cover was implemented in lmapper.cover.KeplerCover in order to implement the same method to build the cover. In both cases the filter used is a projection, since Kepler Mapper do not implement the Eccentricity filter. For this reason it is not possible to take advantage of the filterutils package. As a clustering algorithm it was used a single linkage algorithm, where for lmapper the cutoff.FirstGap class has been used to find the number of clusters, whereas for kmapper it was only possible to set a predetermined number of clusters, that has been chosen to be 2. Note that the performances of lmapper are highly dependent on the parameter chosen to initialize cutoff.FirstGap. Specifically, the smaller the parameter, the more clusters will be created and the slower will be the method complex.Complex.fit affecting the overall performances of the package. For this benchmark this parameter has been set in order to obtain a graph the most similar to the one obtained by Kepler Mapper.

Hardware used: Mac Book Pro late 2011 13", with a 32 nm "Sandy Bridge" 2.4 GHz Intel "Core i5" processor (2435M), 8 GB of 1333 MHz DDR3 SDRAM in two modules of 4GB each.

Figure 16: Synthetic dataset: performance comparison



On the synthetic dataset lmapper managed to get approximately a 10% speedup compared to kmapper. The main topological features (the four branches and the ring at the center of the data point cloud) are correctly identified by both packages; furthermore we can see how the graph output of lmapper has much less noise than the graph obtained by kmapper. In the graph output of kmapper there are two small extra connected components, plus there are many little nodes connected to the biggest connected component. This was possible thanks to the class lmapper.cutoff.FirstGap that for each set of the cover calculates the number of clusters (nodes) to create from the dendrogram; in kmapper the number of clusters has to be fixed; in this case it was set to 2. This is what causes the creation of many little nodes (noise) in the kmapper output.

In conclusion, lmapper is faster and more robust to noise.

On this second dataset lmapper is approximately 20% faster than kmapper; again we see that lmapper's graph is again more robust to noise. There's only one connected component, and the main topological feature (the ring) is still captured.

Figure 17: Synthetic dataset: Left: graph obtained with Kepler Mapper. Right: graph obtained with lmapper

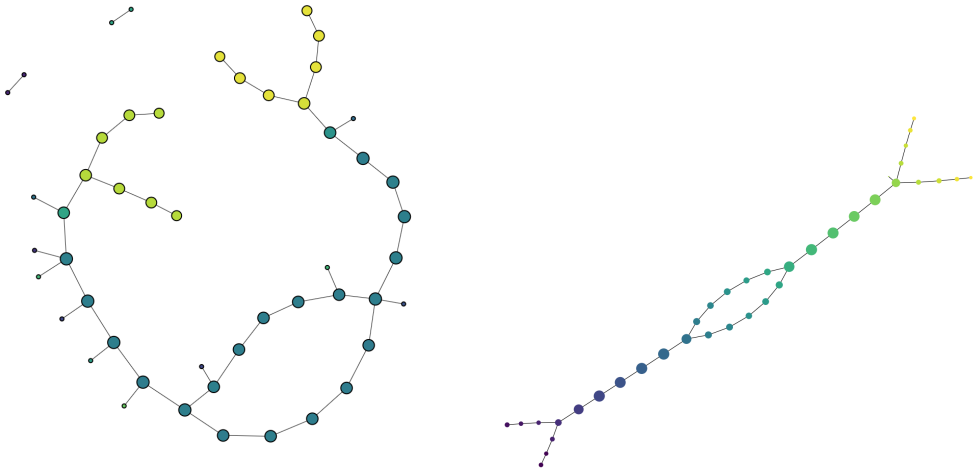


Figure 18: Wisconsin breast cancer dataset: performance comparison

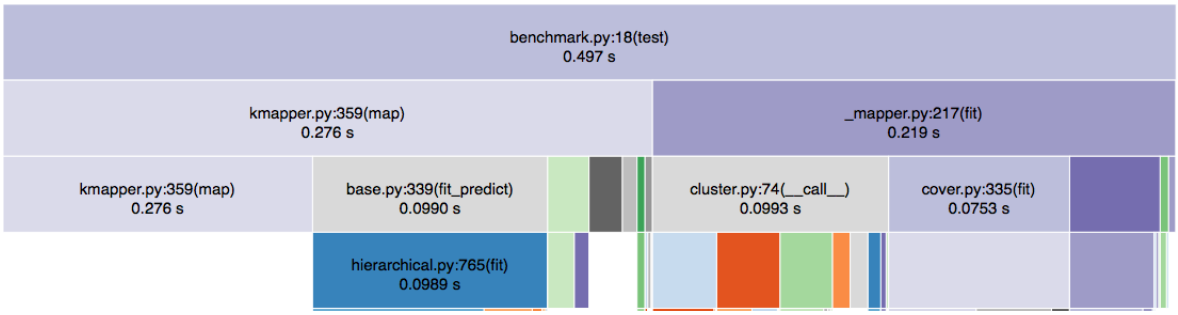
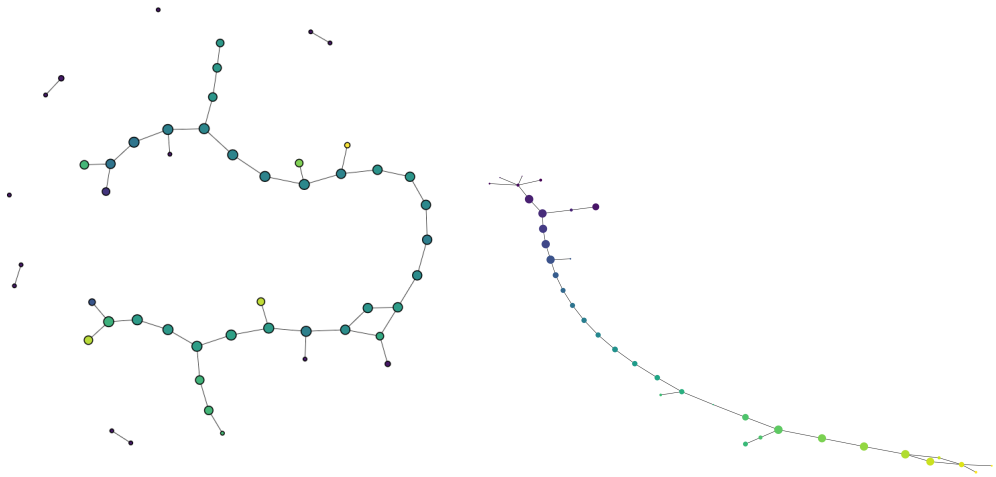


Figure 19: Wisconsin breast cancer dataset. Left: graph obtained with Kepler Mapper. Right: graph obtained with lmapper



6.2 filterutils

For an Eccentricity filter calculated with exponent set to 10 and whith an euclidean distance, the filterutils package manage to get the following speedup (for reproducibility, run the file lmapper/test/test_filterutils.py)

Threads	Speedup
1	1
2	1.83

Accuracy

To see the results, execute the test_filterutils.py file in the lmapper test folder. On our machine we were limited to only 1 and 2 threads, but it would be interesting to test the scalability of the package on a machine capable of running more threads in parallel.

6.3 predmap

With predmap we managed to obtain the same results of Francesco Palma's master thesis (see the test folder of the package) Here a table confronting the results. The small differences in the synthetic datasets are due to the fact that the dataset is not exactly the same used in Palma's work, since it has been regenerated through a random process and thus it is slightly different. The difference in the Wisconsin dataset however means that there's some slight difference in the algorithms.

	Palma's results	predmap (majority vote)
Synthetic dataset	71.01%	70.85%
Wisconsin breast cancer dataset	79.10%	79.78%

Accuracy

Conclusions To conclude, the pure Python implementation of lmapper obtained a speedup of 10% and 20% compared to Kepler Mapper on the synthetic dataset of Palma and the Cat dataset of [15]. filterutils shows good speedup in calculating the eccentricity filter. Thanks to the implementation of the cutoff in the cutoff.py module the output is more robust to noise. predmap was capable of reproducing very closely the results of Francesco Palma's master thesis.

References

[1] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *Theoretical Computer Science*, 392(1):5 – 22, 2008. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2007.10.018. URL http://www.sciencedirect.com/science/article/pii/S0304397507007396. Computational Algebraic Geometry and Applications.

[2] Robert Bruner. What is topology? a short and idiosynchratic answer, 2000. URL http://www.math.wayne.edu/~rrb/topology.html.

[3] G. Carlsson. Topology and data. *Bull. Amer. Math. Soc*, 46:255–308, 2009. URL https://doi.org/10.1090/S0273-0979-09-01249-X.

[4] Hinrich Schütze Christopher D. Manning, Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge University Press, 2008. URL https://nlp.stanford.edu/IR-book/#anchor01.

[5] Tamal K. Dey, Facundo Mémoli, and Yusu Wang. Multiscale mapper: Topological summarization via codomain covers. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 997–1013, 2016. doi: 10.1137/1.9781611974331.ch71. URL https://doi.org/10.1137/1.9781611974331.ch71.

[6] Harish Doraiswamy. *Reeb graphs: computation, visualization and applications*. PhD thesis, Indian Institute of Science, Bangalore, 6 2012.

[7] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[8] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *JSTOR: Applied Statistics*, 28(1):100–108, 1979.

[9] Christian Hening, Marina Meila, Fionn Murtagh, and Roberto Rocci. *Handbook of Cluster Analysis*. Chapman and Hall, 2015.

[10] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 ? seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

[11] Rachel Jeitziner, Mathieu Carriere, Jacques Rougemont, Steve Oudot, Kathryn Hess, and Cathrine Brisken. Two-tier mapper: a user-independent clustering method for global gene expression analysis based on topology, 2018.

[12] P. Y. Lum, G. Singh, A. Lehman, T. Ishkanov, M. Alagappan, J. Carlsson, G. Carlsson, and Mikael Vilhelm Vejdemo Johansson. Extracting insights from the shape of complex data using topology. *Scientific Reports*, 3, 2 2013. ISSN 2045-2322. doi: 10.1038/srep01236.

[13] Milani M. Extending ttmap. https://github.com/MartMilani/ExtendingTTMap, 2018.

[14] James R. Munkres. *Topology*. Prentice Hall, 2000.

[15] Daniel Müllner and Aravindakshan Babu. Python mapper: An open-source toolchain for data exploration, analysis and visualization, 2013.

[16] Monica Nicolau, Robert Tibshirani, Anne-Lise Børresen-Dale, and Stefanie S. Jeffrey. Disease-specific genomic analysis: identifying the signature of pathologic biology. *Bioinformatics*, 23(8):957–965, 2007. doi: 10.1093/bioinformatics/btm033. URL http://dx.doi.org/10.1093/bioinformatics/btm033.

[17] Monica Nicolau, Arnold J. Levine, and Gunnar Carlsson. Topological based data analysis identifies a subgroup of breast cancers with a uniqu mutational profile and excellent survival. *PNAS*, 108(17), 4 2011.

[18] Francesco Palma, Thomas Boys, Martino Milani, and Martina Scolamiero. Predictive models in topological data analysis: the case of mapper.

[19] Christian Schnell. Notes on geometry and topology i. Lecture Notes, Stony Brook University, Department of Mathematics, 2014. URL https://www.math.stonybrook.edu/~cschnell/mat530/notes/.

[20] Robert Tibshirani, Trevor Hastie, Balasubramanian Narasimhan, and Gilbert Chu. Diagnosis of multiple cancer types by shrunken centroids of gene expression. *Proceedings of the National Academy of Sciences*, 99(10):6567–6572, 2002. ISSN 0027-8424. doi: 10.1073/pnas.082099299. URL http://www.pnas.org/content/99/10/6567.

[21] Marc J. van de Vijver, Yudong D. He, Laura J. van 't Veer, Hongyue Dai, Augustinus A.M. Hart, Dorien W. Voskuil, George J. Schreiber, Johannes L. Peterse, Chris Roberts, Matthew J. Marton, Mark Parrish, Douwe Atsma, Anke Witteveen, Annuska Glas, Leonie Delahaye, Tony van der Velde, Harry Bartelink, Sjoerd Rodenhuis, Emiel T. Rutgers, Stephen H. Friend, and René Bernards. A gene-expression signature as a predictor of survival in breast cancer. *New England Journal of Medicine*, 347(25):1999–2009, 2002. doi: 10.1056/NEJMoa021967. URL https://doi.org/10.1056/NEJMoa021967. PMID: 12490681.

[22] Hendrik Jacob van Veen and Nathaniel Saul. Keplermapper. http://doi.org/10.5281/zenodo.1054444, nov 2017.