A SCALABLE IMPLEMENTATION OF MAPPER FOR TOPOLOGICAL DATA ANALYSIS VIA VANTAGE POINT TREES

LUCA SIMI

Abstract. The Mapper algorithm is a powerful tool used in Topological Data Analysis to extract valuable information about the shape of point clouds. One significant drawback of many existing open-source libraries for Mapper is a suboptimal approach to constructing open covers. This naive implementation does not scale well in high dimensions and hampers the overall performance of Mapper. In this study, we propose a novel methodology for building open covers for Mapper using vp-trees. By employing this approach, we develop a more efficient algorithm capable of handling high-dimensional data with improved scalability. Additionally, our methodology produces a simplified and more concise Mapper Graph, enhancing the interpretability of the results. To facilitate the adoption of our methodology, we introduce the *tda-mapper* Python library. This library, hosted at `https://github.com/lucasimi/tda-mapper-python`, implements our proposed approach for building open covers. With tda-mapper, users can effortlessly leverage the benefits of our methodology in their own analyses. Lastly, we conduct comprehensive benchmarks to assess the performance of tda-mapper in comparison to widely-used open-source alternatives. These benchmarks provide quantitative evidence of the superior scalability and efficiency of tda-mapper, further solidifying its position as a reliable and powerful option for Mapper-based analyses.

1. Introduction

In recent years, Topological Data Analysis (TDA) has gained significant momentum in the field of data science due to its ability to extract valuable insights from complex datasets. TDA utilizes topological methods that are resilient to noise and dimensionality challenges, making it a robust mathematical framework for data analysis. One particularly notable tool in TDA is the *Mapper algorithm* [13]. Mapper is an efficient technique that constructs a graph from the given data, revealing important topological properties of the underlying space. By estimating connectivity features of data, Mapper provides a visual representation that facilitates exploration and interpretation. The effectiveness of Mapper was initially demonstrated in the analysis of medical data, as showcased in the pioneering work by Singh, Memoli and Carlsson [13]. Since then, Mapper has proven to be a versatile and powerful tool for data visualization, capable of uncovering hidden patterns even in high-dimensional datasets. Unlike conventional algorithms, such as clustering algorithms or Principal Component Analysis (PCA), Mapper excels at visualizing data by preserving the same number of connected components as the original dataset. This feature alone renders Mapper more reliable and superior for shape exploration and pattern extraction, especially when dealing with high dimensional data. Overall, Mapper's ability to handle complexity, extract meaningful insights, and provide a visual format for data exploration makes it an invaluable tool in the field of data science.

Data exploration is inherently interactive, requiring a lot of tweaks and fine-tuning in order to extract relevant information from data. For this reason, overall performance of software for Mapper plays a crucial role in driving a widespread adoption across the industry. Although there are a few widely-used libraries for Mapper, used both in reaserch and industry, they predominantly prioritize correctness over performance.

The original description of Mapper in [13] includes what the authors refer to as *statistical implementation*, which has now become a de-facto standard in several open source implementations such as *Python Mapper* [11], *Kepler Mapper* [15], and *giotto-tda* [14]. However, the description of the statistical implementation in [13] gives the definition of an *open cover* (often called *cubical*

Date: February 14, 2024.

cover) but lacks an explicit algorithm for its construction. Regrettably, this critical step has often been neglected both in the industry and in scientific production, resulting in the adoption of naive procedures to implement it. Consequently, many open source libraries, including the aforementioned ones, present significant performance drawbacks. In this study, we provide a practical enhancement to address this issue and improve the performance of Mapper.

Recently, a number of alternative algorithms, known as *Mapper-type* algorithms, have been developed. Two such algorithms, called *Ball Mapper* (abbreviated as *BM*) and *Mapper on Ball Mapper* (abbreviated as *MoBM*), were introduced in [4] and [5] respectively. Also in the field of neuroscience new Mapper-like algorithm came to light, for example *NeuMapper* introduced in [7]. Unlike traditional Mapper algorithms that use rectangles to construct the open cover, these new algorithms employ open balls. Although these alternative algorithms enhance the construction of the open cover, they come at the cost of changing the shape of the open sets. In our work, we present a novel and more efficient approach to compute Mapper-type algorithms, including *BM* and *MoBM*, by utilizing *vp-trees*. Our methodology enables us to construct open covers more effectively and efficiently.

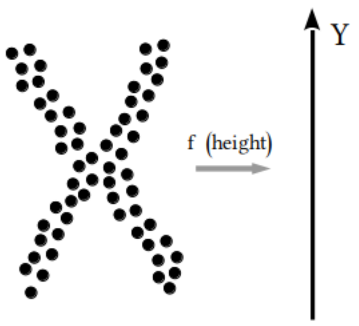
This methodology has been implemented and released by the author as the Python package called *tda-mapper*. It can be accessed at `https://github.com/lucasimi/tda-mapper-python`. As far as the author knows, there is currently no other open source package available that contains the same implementation as presented here. In the following sections, we will provide a detailed explanation of the core ideas behind our methodology and shed light on the implementation of *tda-mapper*. We will introduce the fundamental mathematical and technical concepts that form the basis of this software. Moreover, we will conduct benchmarks to compare the performance of *tda-mapper* against [11], [15], and [14]. This will help us gauge the effectiveness and efficiency of *tda-mapper* in relation to its counterparts.

Remark 1.1. Throughout our work, whenever we analyze a dataset denoted as X, we will assume that it is a finite and discrete sample of an unknown topological space referred to as $\tilde{X}$. Studying the topology of a finite, discrete topological space is a straightforward task, as its only topological invariant is its cardinality. Therefore, when we examine the topology of a dataset X, our main focus is actually on understanding the topology of $\tilde{X}$. In essence, when we mention our desire to comprehend the shape of X, we truly aim to gain insights into the topology of $\tilde{X}$ by examining the sample X.

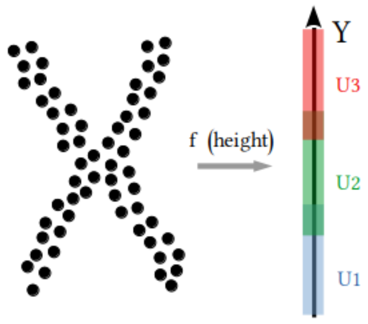
2. The Mapper Algorithm

In this section, we provide a concise overview of Mapper, based on the work by [13]. Mapper operates on a dataset X and its output is determined by the following steps:

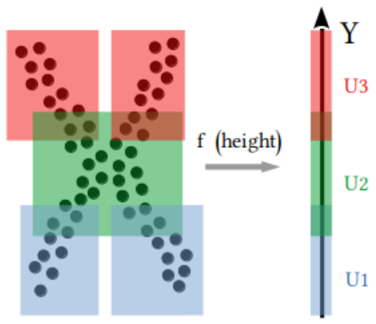
(1) Let f be a *lens*, which we define as a continuous map $f\colon X \to Y$, where Y is a parameter space that can have a lower dimension than X, although this is not always the case. We will work with the general setting where Y can have any dimension. Common choices for the lens f include *statistics* of any order, *projections*, *entropy*, *density*, *eccentricity*, and more. As an example, we chose X as an X-shaped point cloud in $\mathbb{R}^2$, and f as the *height function*, which is the projection of X onto the y-axis.



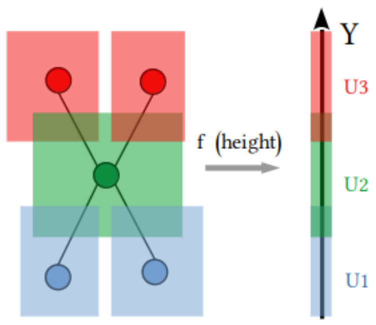
(2) Next, we proceed by constructing an *open cover* for $f(X)$. In other words, we create a collection $\{U_\alpha\}_\alpha$ of open sets such that their union covers the entire image $f(X)$, i.e., $f(X) = \bigcup_\alpha U_\alpha$. It is important to note that the sets in this open cover may intersect with one another, and they inherit their topology from the parent space Y. In the picture you can see an open cover of $f(X)$ made by three colored open sets U_1, U_2 and U_3.



(3) For each element U_α in the selected cover, we define V_α as the preimage of U_α under the function f. It is clear that the collection $V_{\alpha\alpha}$ forms an open cover of X. Next, we proceed to partition each open set V_α into a disjoint union denoted as $V_\alpha = \amalg_\beta C_{\alpha,\beta}$. The clusters $C_{\alpha,\beta}$ are obtained from a chosen clustering algorithm. The resulting family $C_{\alpha,\beta_{\alpha,\beta}}$ is referred to as a refined open cover for X. In the picture you can see the pullbacks of U_1, U_2 and U_3, with their corresponding colors. Moreover the pullbacks of U_1 and U_3 has been split according to some possible clusters.



(4) We construct the *Mapper Graph* as the undirected graph $G = (V, E)$ defined by the following rule: the set V contains a vertex $v_{\alpha,\beta}$ for every local cluster $C_{\alpha,\beta}$, while the set E contains the edge $e = (v_{\alpha_1,\beta_1}, v_{\alpha_2,\beta_2})$ only if their corresponding local clusters intersect, i.e., when $C_{\alpha_1,\beta_1} \cap C_{\alpha_2,\beta_2} \neq \emptyset$. In the picture, by following colors, we can finally see how clusters and intersections are summarized into the mapper graph.



The intuition around Mapper can be explained as follows: Steps 1 to 3 work together to create an open cover for X, which can be visualized as a patchwork of smaller sets. These smaller sets have a simpler shape compared to X, making them easier to analyze. Next, we apply the pullback cover. By considering the pullbacks of these smaller sets, we can constrict the range of values taken by the continuous map f. This simplifies the study of f. Once the pullback cover is established, we proceed to clustering. This step involves grouping together similar elements and constructing a graph that represents the connections between local clusters. In an ideal scenario, we aim to prevent different *connected components* from merging into a single node in the Mapper Graph.

The theoretical foundation of Mapper is rooted in the *Nerve Theorem* (see [1], [16]). According to this theorem, it is required that each intersection of two different open sets is either empty or *simply connected* (meaning it can be continuously deformed into a point without any holes) in order to apply it. For Mapper, this condition translates to the requirement that each intersection of any two clusters is either empty or simply connected. However, it is important to note that the Mapper algorithm as a whole may not satisfy this condition. This is because clustering does not guarantee that clusters correspond to connected components, and clustering algorithms do not generally preserve simple-connectedness. Despite this limitation, whenever the hypothesis of the Nerve Theorem holds, it can be claimed that X and its Mapper Graph have the same number of connected components.

2.1. **Cubical Cover.** In the original definition of Mapper in [13], the authors utilize an open cover consisting of consecutive intervals with fixed *length* and *overlap*. They introduce two parameters to characterize this open cover. The first parameter is the *length* w of the intervals, while the second parameter is the *overlap* $p \in (0,1)$; this denotes the fraction of w representing the intersection length δ between any two consecutive intervals in the cover.

Definition 2.1. Let $0 < n \in \mathbb{N}$ and $p \in (0,1)$.

(1) Consider the closed interval $[m, M] \subseteq \mathbb{R}$ and let $w = \frac{M-m}{n(1-p)}$ and $\delta = pw$. The *cubical cover* of $[m, M]$ is the collection of open sets

$$CC_{[m,M]}(n,p) = \{I_0, \dots, I_i, \dots, I_{n-1}\}$$

such that $I_i = [m, M] \cap (a_i, b_i)$ with

$$a_i = m + i(w-\delta) - \delta/2$$
$$b_i = m + (i+1)(w-\delta) + \delta/2.$$

(2) Given $X \subseteq \mathbb{R}$ compact, the *cubical cover* of X is defined as

$$CC_X(n,p) = CC_{[m,M]}(n,p)$$

where $m = \min(X)$ and $M = \max(X)$.

(3) Let $X \subseteq \mathbb{R}^k$ compact. The *cubical cover* of X

$$CC_X(n,p) = \{I_{1,j_1} \times \dots \times I_{k,j_k} | (I_{1,j_1}, \dots, I_{k,j_k}) \in CC_{X_1}(n,p) \times \dots \times CC_{X_k}(n,p)\}$$

where $X_j \subseteq \mathbb{R}$ is the projection of X on the j-axis.

Remark 2.2. We report here some facts that easily follow from the definition in the case $X = [m, M]$.

- $b_i - a_i = w$ for every $i = 0, \dots, n-1$
- $a_0 = m - \delta/2$ and $b_{n-1} = M + \delta/2$
- $a_{i+1} - b_i = \delta$ for every $i = 0, \dots, n-1$.
- $[m, b_0 - \delta/2), \dots, [a_i + \delta/2, b_i - \delta/2), \dots [a_{n-1} + \delta/2, M]$ is a partition of $[m, M]$.

Example 2.3. Consider $X = [0, 12]$, we have

$$CC_{[0,12]}(4, 2/5) = \{[0,4), (2,7), (5,10), (8,12]\}.$$

From the definition of cubical cover it's very easy to find an algorithm which constructs the cubical cover on each projection separately and then build the open cover on X. This is what we call *naive cubical cover*

Algorithm 1 Naive Cubical Cover

```
Input: Any integer $n > 0$, $p \in (0, 1)$.
Output: $CC(n, p)$.
  for $i = 1, \dots, k$ do
    $CC_i(n, p) \leftarrow \{I_{i,1}, \dots, I_{i,n}\}$ be the cubical cover on the projection of $X$ on the $j$-axis;
  end for
  $CC(n, p) \leftarrow \{R = \prod_{i=1}^{k} I_{i,j_i} | R \neq \emptyset, 1 \leq j_i \leq n\}$.
  return $CC(n, p)$
```

Remark 2.4. Algorithm 1 presents a straightforward yet inefficient method for obtaining the cubical cover. Indeed, this approach becomes computationally expensive for Mapper when dealing with high-dimensional lenses, i.e. when $f(X) \subseteq Y$ is high dimensions. Even when many products are empty, their number can grow rapidly and introduce additional computational overhead to the entire Mapper process. To illustrate this issue, consider the following example: if $f(X) \subset \mathbb{R}^k$ lies along the diagonal, an appropriate cover for $f(X)$ could be achieved using a number of rectangles proportional to the number of intervals n. Nevertheless, Algorithm 1 would construct an open cover for each projection initially and then iterate through all possible rectangles, resulting in a total of n^k steps.

2.2. **Ball Cover.**

When a dataset X is embedded in a Euclidean space, it is always possible to construct an open cover using rectangles, as in cubical cover. However, in cases where coordinates are not available, such as in a generic metric space, a different approach is required since rectangles may not have a well-defined meaning. An alternative option in such cases is to consider open balls as a suitable choice for constructing an open cover. We recall here some basic definitions:

Definition 2.5. Let X be a topological space. A *pseudometric* on X is a map $d \colon X \times X \to \mathbb{R}$ such that

- $d(x, y) \geq 0$ for every $x, y \in X$, and $d(x, x) = 0$ for every $x \in X$;
- d is *symmetric*, i.e. $d(x_1, x_2) = d(x_2, x_1)$ for every $x_1, x_2 \in X$;
- d satisfies the *triangle inequality*, i.e.: $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$ for every $x_1, x_2, x_3 \in X$.

We say that d is a *metric* when $d(x, y) = 0$ implies $x = y$. An *open ball* of center $p \in X$ and radius $\epsilon > 0$ is defined as $B_d(p, \epsilon) = \{x \in X | d(p, x) < \epsilon\}$.

Definition 2.6. Given a dataset X and a metric d on X, a *ball cover* of radius $r > 0$ on X is any open cover where every set is an open ball of radius r.

The approach of building a ball cover in the context of Mapper is not new, it was presented in [4] and [5], called *ϵ-net*, where the author introduced an alternative to Mapper, called *Ball Mapper*. In that setting a ball cover of X is built by incrementally adding open balls of fixed radius, until every point is covered by at least one open ball (See Algorithm 2).

Algorithm 2 ϵ-net

```
Input: Let $X$ be a dataset, $d$ a pseudometric on $X$, and let $\epsilon > 0$.
Output: A ball cover of $X$.
  $S \leftarrow X$, as a set
  $C \leftarrow \emptyset$
  while $S \neq \emptyset$ do
      take a point $p \in S$
      $B \leftarrow B_d(p, \epsilon)$                ▷ the ball of radius $\epsilon$, centered in $p$
      Add $B$ to $C$
      for $q \in B$ do
          Remove $q$ from $S$
      end for
  end while
  return $C$
```

Remark 2.7. Covering a dataset $X \subseteq \mathbb{R}^k$ with open balls centered in X is enough to prevent the flaw of Algorithm 1, because the number of open balls is expected to be proportional to n^d, where d is the dimension of X, which is typically lower than the representational dimension k. This remark is also presented in [5], where the author introduces *MoBM*.

2.3. **Proximity Net.** In this work, we present a more flexible version of the ϵ-net concept that we call *proximity-net*. This modified approach allows for a wider range of input parameters. To achieve this, we introduce a new parameter known as the *proximity* function, denoted as $b\colon X \to \mathcal{P}(X)$. The proximity function, for any given point $p \in X$, returns a collection of points $b(p)$ (where $p \in b(p)$) that are considered to be *near* to p. Instead of constructing balls as in the traditional ϵ-net approach, we modify the algorithm to collect the open sets obtained by invoking the proximity function b. This modification allows us to construct a broader range of potential open covers utilizing the same algorithm. In light of these modifications, we refer to our algorithm as the proximity-net algorithm, as illustrated in Algorithm 3.

Algorithm 3 proximity-net

```
Input: Let $X$ be a dataset, and let $b$ be a proximity function
Output: An open cover of $X$
  $S \leftarrow X$, as a set
  $C \leftarrow \emptyset$
  while $S \neq \emptyset$ do
      take a point $p \in S$
      $B \leftarrow b(p)$
      Add $B$ to $C$
      for $q \in B$ do
          Remove $q$ from $S$
      end for
  end while
  return $C$
```

Remark 2.8. We can write an identity which models the time complexity of proximity-net. Let $T(Y)$ be the time complexity of running proximity-net on a dataset $Y \subseteq \mathbb{R}^k$ of dimension $d \leq k$. It's easy to see that we can write $T(Y) = \tilde{T}_b(Y) + R(Y)T_b(Y)$, where

- $\tilde{T}_b(Y)$ is the time complexity of constructing the proximity function b;
- $R(Y)$ is the number of open sets obtained from calling the proximity function b in order to cover Y;
- $T_b(Y)$ is the time complexity of computing a single instance of the proximity function b.

Definition 2.9. Let (Y, d) be a pseudometric space. For each $\epsilon > 0$ we define the *ball proximity* function $BP(d, \epsilon)\colon Y \to \mathcal{P}(Y)$ by setting for every $y \in Y$

$$BP(d, \epsilon)(y) = B_d(y, \epsilon) = \{u \in Y | d(y, u) < r\} .$$

Remark 2.10. It's clear to see that ϵ-net can be obtained from proximity-net by supplying the map $BP(d, \epsilon)$ as the proximity function.

Definition 2.11. Let n be a positive integer and $p \in (0, 1)$. Consider the interval $[m, M] \subseteq \mathbb{R}$ and let $w = \frac{M-m}{n(1-p)}$ and $\delta = pw$. Let $a_i = m+i(w-\delta)-\delta/2$ and $b_i = m+(i+1)(w-\delta)+\delta/2$. We define the *cubical proximity* function $CP_{[m,M]}(n, p)\colon [m, M] \to \mathcal{P}([m, M])$ by setting for every $x \in [m, M]$

$$CP_{[m,M]}(n, p)(x) = [m, M] \cap (a_i, b_i) \iff a_i + \delta/2 \leq x < b_i - \delta/2$$

For any $X \subseteq \mathbb{R}^k$ compact we can define for every $x \in X$

$$CP_X(n, p)(x) = CP_{X_1}(n, p)(x_1) \times \ldots \times CP_{X_k}(n, p)(x_k)$$

where X_j is the projection of X on the j-axis.

3. Vantage Point Trees

A *range query* is a function that returns $B(p, \epsilon)$ for any $p \in X$, where p is the *query point* and ϵ is the *query radius.* There are numerous algorithms and data structures available in scientific literature for efficiently performing range queries. One popular example is *kd-tree*, which effectively partition the space to minimize the computation of pairwise distances. However, in the context of implementing Ball Mapper [4], the author has dismissed the use of kd-trees due to their limited theoretical advantage in high-dimensional spaces. Since the cover in Ball Mapper is typically constructed on X, which often involves high dimensions, the potential gain from employing kd-trees is minimal. Conversely, when it comes to MoBM [5], using a specialized data structure for range queries can prove advantageous. This is because the open cover for MoBM is built on the (typically low-dimensional) space $f(X)$. Therefore, using a dedicated data structure for range queries can enhance the performance of MoBM.

In this study, our primary focus is to accommodate a wide range of scenarios and utilize a general lens $f\colon X \to Y$. It is important to note that the target space, Y, does not necessarily have to be Euclidean or coordinate-based, but rather encompasses a broader category of pseudometric spaces. To ensure maximum flexibility and suitability for our tasks involving range queries, we have opted to utilize a *metric tree* data structure rather than a kd-tree. More specifically, we have chosen the highly renowned and versatile *Vantage Point Tree* data structure, commonly known as *vp-tree.* This decision was made due to its superior performance in high-dimensional settings, proving to be more efficient compared to kd-trees. It is worth highlighting that vp-trees transcend the constraints of a Euclidean space and are able to function effectively within any pseudometric space. Consequently, they serve as an optimal choice for our purposes.

Remark 3.1. The ability to work with pseudometrics, rather than just metrics, is an invaluable feature of vp-trees that we will leverage in our implementation.

In the following paragraphs, we will provide a concise overview of vp-trees tailored to our specific requirements. For a detailed understanding of the original definitions and findings regarding vp-trees, we point the interested readers to [17] and [2].

Remark 3.2. While we do have a preference for vp-trees, it is important to acknowledge that in the context of Euclidean spaces, there are other data structures available for consideration, apart from metric trees. One such alternative is *range trees*, which could potentially be useful in constructing the cubical cover. Range trees are particularly well-suited for interval queries and may be worth exploring.

For the sake of readability we include a definition of vp-trees that suits our needs.

Notation 3.3. In the following, we will introduce a recursive notation for binary trees. To represent an empty tree, we will use the notation $\{\}$. For any non-empty binary tree T, we will represent it as $T = Tree\{x, L, R\}$. Here, x represents a piece of information attached to the node, L denotes the tree rooted at the left child of T, and R denotes the tree rooted at the right child of T. When a tree T is terminal, it can be represented as $T = Tree\{x, \{\}, \{\}\}$. However, for improved readability, we prefer to write $T = Tree\{x\}$ in this case. In the case of a non-empty binary tree $T = Tree\{x, L, R\}$, we will use the notation $T.left$ to denote the tree rooted at the left child of T, and $T.right$ to denote the tree rooted at the right child of T. Finally, for any tree T, we will use the notation $T.leaves$ to represent its leaves.

Definition 3.4. Let X be a dataset, and d a pseudometric on X. A *vp-tree on* (X, d) is any binary tree T which satisfies the following properties:

(1) For each $x \in X$ there exists a leaf $Tree\{x\} \in T.leaves$.
(2) If T is non-terminal we have $T = Tree\{(p, \rho), L, R\}$, where $p \in X$ and $\rho > 0$, such that:
   - $d(p, x) \leq \rho$ whenever $x \in L.leaves$;
   - $d(p, x) \geq \rho$ whenever $x \in R.leaves$.

We can summarize the most important features of vp-trees:

- Building a balanced vp-tree is achieved through recursive splitting of the dataset in half at each step (refer to Algorithm 4). This process requires an asymptotic time of $O(|X| \log(|X|))$ in total.
- Range queries can be significantly more efficient with vp-trees than with linear scans. When using a linear scan, which involves looping through all elements in X, it takes $|X|$ steps to find the points of $B(q, \epsilon)$. This results in an asymptotical time complexity of $O(|X|)$. On the other hand, by employing a well-balanced vp-tree, a range query typically requires fewer steps. The triangle inequality satisfied by the metric allows us to visit only a single child at each step when certain conditions are met (refer to Figure 1). For this reason precisely estimating the average time complexity of range queries is challenging due to its dependency on the dataset (see reference [2]), but we can confidently state that it is bounded between $O(\log(|X|))$ and $O(|X|)$. In situations where the query radius is sufficiently small, we expect the query radius to less likely fall across two children at the same time. For this readon we expect that in such cases the time complexity of range queries is comparable to $O(log(|X|))$. Additionally, vp-trees often outperform kd-trees, particularly in high dimensions.

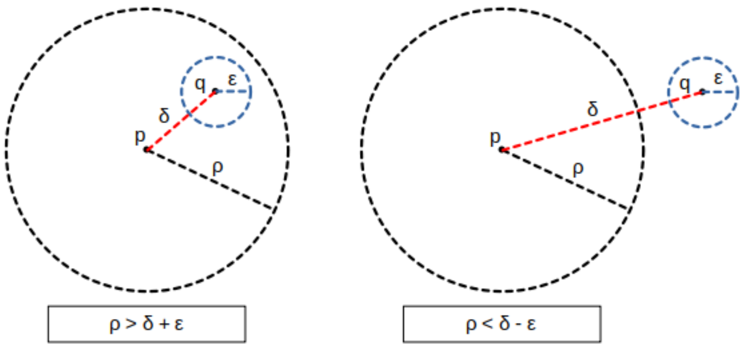


FIGURE 1. The only two possible settings when a branch can be skipped in vptree.search, while visiting node $Tree\{(p, \rho), L, R\}$, wher q is the query point and ϵ is the query radius.

Remark 3.5. The time complexity of executing a range query is influenced by the count of computed pairwise distances. When utilizing vp-trees, this count is generally a fraction of the upper bound $|X|$.

Remark 3.6. Using Remark 2.8, we can give a rough estimation of the time complexity of ball cover in two different cases.

- In the first case, when range queries are performed with linear scans, we have

$$T(Y) = R(Y)T_{BP}(Y) = O(n^d \cdot n) = O(n^{d+1}).$$

- In the second case, when range queries are performed using vp-trees, we have

$$T(Y) = \tilde{T}_b(Y) + R(Y)T_b(Y) = O(|Y| \log |Y|) + O(n^d)T_{BP}(Y).$$

Since $T_{BP}(Y)$ is at most $O(n)$ we can claim that vp-trees give (at worst) the same time complexity as linear scans. However, when $T_{BP}(Y) = O(\log |Y|)$ the asymptotic time complexity is far lower with vp-trees.

To enhance readability, we provide the two primary algorithms related to vp-trees. The first algorithm (Algorithm 4) constructs a vp-tree for a given dataset and metric. The second algorithm (Algorithm 5) performs the actual range query for a given query point q and query radius ϵ.

Algorithm 4 vp.build

```
Input: Let $X = [x_0, \dots, x_{n-1}]$ be a dataset, and let $d\colon X \times X \to \mathbb{R}$ be a metric.
Output: A vp-tree on $(X, d)$.
  if $X = \emptyset$ then:
      $T \leftarrow Tree\{\}$, the empty tree
  else
      $p \leftarrow$ chose in $X$. Let $\rho$ be the median of $d(p, x)$ for $x \in X$.
      Reorder $X$ such that $d(p, x_i) \leq \rho$ for $i < n/2$ and $d(p, x_i) \geq \rho$ for every $i \geq n/2$ (Use quickselect)
      $L \leftarrow vp.build([x_0, \dots, x_{n/2-1}])$
      $R \leftarrow vp.build([x_{n/2}, \dots, x_{n-1}])$
      $T \leftarrow Tree\{root = (p, \rho), left = L, right = R\}$
  end if
  return $T$
```

Algorithm 5 vp.search

```
Input: $T = vp.build(X, d)$, where $X$ is a dataset, and $d$ a metric on $X$. Let $q$ be a query point,
  and $\epsilon > 0$
Output: $B_d(q, \epsilon) = \{x \in X | d(x, q) < \epsilon\}$.
  if $T = Tree\{\}$ then
      $S \leftarrow \emptyset$
  else if $T = Tree\{root = x\}$ then
      if $d(q, x) < \epsilon$ then
          $S \leftarrow \{x\}$
      else
          $S \leftarrow \emptyset$
      end if
  else
      $(p, \rho) \leftarrow T.root$
      $\delta \leftarrow d(p, q)$
      if $\rho \geq \delta + \epsilon$ then
          $S_R \leftarrow \emptyset$
      else
          $S_R \leftarrow vp.search(T.right, q, \epsilon)$
      end if
      if $\rho \leq \delta - \epsilon$ then
          $S_L \leftarrow \emptyset$
      else
          $S_L \leftarrow vp.search(T.left, q, \epsilon)$
      end if
      $S \leftarrow S_L \cup S_R$
  end if
  return $S$
```

4. Cubical Cover

By utilizing the definition of ball cover and the efficient computation method provided by vp-trees, we can compute the cubical cover while eliminating the performance degradation encountered in Algorithm 1. Before delving into the specifics of this approach, it is necessary to establish a set of notation. Initially, we define a helper function γ_n that takes a dataset Y and projects it onto the hypercube $[0, n]^k$.

Definition 4.1. Let $X \subseteq \mathbb{R}^k$ compact. Let $m_j = \min_{x \in X} x_j$, and $M_j = \max_{x \in X} x_j$. For every integer $n > 0$ we define $\gamma_n : X \to [0, n]^k$ as

$$\gamma_n(x)_j = n \frac{x_j - m_j}{M_j - m_j}, \quad j = 1, \ldots, k.$$

Remark 4.2. Given $m_j < M_j$ for $j = 1, \ldots, k$, with $Y = \prod_{j=1}^k [m_j, M_j]$, the map $\gamma_n : Y \to [0, n]^k$ is a bijection. The map $\gamma_n^{-1} : [0, n]^k \to Y$ is given by

$$\gamma_n^{-1}(x)_j = m_j + x_j \frac{M_j - m_j}{n}, \quad j = 1, \ldots, k.$$

Definition 4.3. Given a proximity function $b : X \to P(X)$ and a map $f : Z \to X$, we can consider the map $c : Z \to P(Z)$ defined by $c(z) = f^{-1}(b(f(z)))$. The map c is then a proximity function for Z that we call *pullback proximity function* and denote with f^*b.

Definition 4.4. Let $f : X \to Y$ be a map of topological spaces. If d is a pseudometric on Y, the *pullback pseudometric* of d under f is the pseudometric f^*d, defined by setting for each $u, v \in X$

$$f^*d(u, v) = d(f(u), f(v)).$$

Proposition 4.5. *Let $X \subseteq \mathbb{R}^k$ compact. Let n be a positive integer and $p \in (0,1)$. Let $\rho\colon [0,n]^k \to [0,n]^k$ defined by $\rho(x)_j = \lfloor x_j \rfloor + \frac{1}{2}$. Then we have*

$$CP_X(n,p) = BP_X\left(\gamma_n^* d_\infty, \frac{1}{2-2p}\right) \circ \gamma_n^{-1} \circ \rho \circ \gamma_n.$$

Proof. Without repeting ourselves we will use the notation introduced within Definitions 2.1 and 4.1. We start the proof with a preliminary observation: for any $x \in X$ we have

$$\begin{aligned}(\rho \circ \gamma_n)(x)_i &= \lfloor \gamma_n(x)_i \rfloor + \frac{1}{2} \\ &= \left\lfloor n \frac{x_i - m_i}{M_i - m_i} \right\rfloor + \frac{1}{2} \\ &= \left\lfloor \frac{x_i - m_i}{w_i - \delta_i} \right\rfloor + \frac{1}{2}, \qquad (1)\end{aligned}$$

Take any $x \in X$ and let $CP_X(n,p)(x) = I_{1,j_1} \times \ldots \times I_{k,j_k}$ for some $j_i, \ldots, j_k$ (see Definition 2.1). Using the definition of I_{i,j_i}, this condition can be written as the following inequality

$$a_{j_i} + \frac{\delta}{2} \leq x_i < b_{j_i} - \frac{\delta}{2},$$

which together with 1 is equivalent to $(\rho \circ \gamma_n)(x)_i = j_i + \frac{1}{2}$. Moreover observe that

$$\begin{aligned}(\gamma_n^* d_\infty)((\gamma_n^{-1} \circ \rho \circ \gamma_n)(x), y) &= d_\infty((\rho \circ \gamma_n)(x), \gamma_n(y)) \\ &= d_\infty\left(j_i + \frac{1}{2}, \gamma_n(y)\right). \qquad (2)\end{aligned}$$

The proof then follows a double inclusion argument. In order to prove the first inclusion take any $y \in CP_X(n,p)(x)$. Following the definition of $CP_X(n,p)$ we can write

$$a_{j_i} < y_i < b_{j_i}.$$

Using the definition of γ_n we have

$$\frac{a_{j_i} - m_i}{w_i - \delta_i} < \gamma_n(x)_i < \frac{b_{j_i} - m_i}{w_i - \delta_i}$$

which is equivalent to

$$-\frac{1}{2-2p} < \gamma_n(x)_i - j_i - \frac{1}{2} < \frac{1}{2-2p}$$

and can be written as

$$\left| \gamma_n(y)_i - \left(j_i + \frac{1}{2}\right) \right| < \frac{1}{2-2p}.$$

Repeating this for every i and using 2, we obtain that $y \in BP_X\left(\gamma_n^* d_\infty, \frac{1}{2-2p}\right)((\gamma_n^{-1} \circ \rho \circ \gamma_n)(x))$. In order to prove the other inclusion we can follow the same arguments in a backward direction, and this concludes the proof. □

5. Development of tda-mapper

The development of *tda-mapper* [12] was primarly motivated by the exploration of alternative methods for constructing open covers for Mapper, driven by the goal of implementing a more efficient approach. While major open-source implementations like *Python Mapper* [11], *Kepler Mapper* [15], and *giotto-tda* [14] can in principle handle high-dimensional lenses, they all fall short of usability, as they all rely on Algorithm 1 with its well-documented limitations (discussed in Remark 2.4). Indeed, a common thread among them is the usage of a function call to `itertools.product`, as seen in their source code. This function, described in Python's official documentation available at `https://docs.python.org/3/library/itertools.html#itertools.product`, is used to perform a nested loop on each one-dimensional open cover, which is exactly the essence of Algorithm 1.

5.1. **Code Architecture.** The library *tda-mapper* is designed to provide a comprehensive set of modules through a single central package called `tdamapper`. These modules offer various features and we will provide an overview of the architectural decisions made during development. Throughout the development process, one of the objectives was to create an API that is easy to understand and use. To achieve this, we adopted an object-oriented approach and took inspiration from the well-respected *scikit-learn* APIs. This ensures familiarity for the intended user base of *tda-mapper*. Additionally, we made efforts to keep the API of *tda-mapper* similar to the APIs provided by other popular libraries such as *giotto-tda* and *Kepler Mapper*. This allows users to smoothly transition between these libraries and leverage their existing knowledge. By considering these factors, we aim to provide a user-friendly and seamless experience for users of *tda-mapper*, making it easier for them to explore and utilize the library's full potential.

5.1.1. *Core module.* In the module `tdamapper.core`, we have implemented the essential functions responsible for computing the Mapper graph. Additionally, we have encapsulated the entire logic of the Mapper within the `MapperAlgorithm` class for convenience.

5.1.2. *Cover module.* Inside the module `tdamapper.cover`, the implementation of *proximity-net* (Algorithm 3) can be found in the function `proximity_net`. In addition, this module also contains several cover algorithms based on proximity-net. These algorithms are implemented as subclasses of `ProximityNetCover`, and are expected to implement the methods `fit` and `search`:

- The `fit` method is intended for fine-tuning internal parameters and structures that enhance the execution efficiency of the `search` method. It works similarly to the `fit` method of *scikit-learn* estimators.
- The `search` method is the actual proximity function employed as a parameter for proximity-net.

We offer several cover algorithms, which are defined by the following classes within the `tdamapper.cover` module:

- `TrivialCover`: This class implements the *trivial proximity function*, where the entire dataset is returned for each input point.
- `BallCover`: This class implements the proximity function BP using vp-trees for efficient computation.
- `KNNCover`: This implementation uses vp-trees for K-nearest neighbor (KNN) search.
- `CubicalCover`: This class implements the proximity function CP using `BallCover` as stated in Proposition 4.5.

Remark 5.1. To compute the Mapper graph, the first step is to construct an open cover on the image $f(X)$. Subsequently, the pullback under f is performed to obtain an open cover of X. In the case of `BallCover`, it is possible to obtain the pullback cover of X with a more direct approach, by using the pullback of the metric.

To understand this, suppose we have an open cover $\mathcal{U} = \{B_d(y_i, \epsilon) | i = 1, \dots, m\}$ for $f(Y)$, where $y_i = f(p_i)$ for $i = 1, \dots, m$. In this scenario we can write $f^*\mathcal{U} = \{f^{-1}B_d(y_i, \epsilon) | i = 1, \dots, m\} = \{B_{f^*d}(p_i, \epsilon) | i = 1, \dots, m\}$. Therefore, it is feasible to construct $f^*\mathcal{U}$ by employing *proximity-net* directly on X by using the pseudo-metric f^*d, without constructing the pullback of the open cover.

However, in our code, we chose not to follow this approach since it would only be applicable for `BallCover`. We want all the different classes in `tdamapper.cover` to share the same API, which suggests a more generic approach. For this reason, to compute pullbacks with minimal overhead, we opted for the `search` method to return indices instead of points.

5.1.3. *VPTree.* We have implemented our own version of the vp-tree data structure within the `tdamapper.utils.vptree` module to optimize it for our specific use-case. Our implementation allows each leaf of the vp-tree to contain multiple items by stopping the construction when the splitting circle is *small*, either in terms of its cardinality or in terms of its radius (smaller than a given threshold). This optimization is beneficial both for range queries and for K-nearest neighbor (KNN) queries. When the visited node during a search becomes smaller than the query, the search operation collapse into a faster brute force linear scan. Currently, there are two different implementations available for vp-trees in our codebase:

- `tdamapper.utils.vptree`: This module implements the `VPTree` class using a binary tree structure and utilizes recursive methods for construction and search operations. Both range search and KNN search methods are implemented by providing different arguments to the same recursive method. This implementation was the first implementation of vp-trees in tda-mapper and serves as a reference implementation for regression testing.
- `tdamapper.utils.vptree_flat`: This module implements the `VPTree` class by using an array and utilizes iterative methods for construction and search operations.

Since iterative methods are generally considered better and safer than recursive ones, we have chosen to default to using the `tdamapper.utils.vptree_flat.VPTree` implementation in the `BallCover` and `KNNCover` classes. However, we still offer the option for users to use `tdamapper.utils.vptree` if desired.

5.1.4. *Clustering Algorithms.* Clustering algorithms are often susceptible to failure in practical scenarios, leading to potential failures in the overall Mapper functionality. To address this issue, we have developed a wrapper class called `FailSafeClustering` that acts as a safeguard when using clustering algorithms. This wrapper class accepts a clustering algorithm, compatible with `sklearn.cluster` estimators, and in case of a failure indicated by a `ValueError` exception, it falls back to a trivial clustering approach. In this approach, all data points are assigned to a single cluster.

In addition to the fail-safe mechanism, we have also implemented a novel clustering algorithm named `MapperClustering`. This algorithm clusters data based on the connected components of the Mapper graph. By leveraging the structural properties of the Mapper graph, this approach provides a unique way to group data points and extract meaningful information from complex datasets.

5.2. **Dependencies and testing.** The implementation of *tda-mapper* relies on several software dependencies, including `networkx` [8], `numpy` [9], `matplotlib` [10], and `plotly` [3]. These dependencies are essential for various functions and features of the Mapper algorithm.

- `networkx` is used to generate and manipulate the Mapper graph, which is the primary result of the algorithm.
- `numpy` is necessary for numeric computations, particularly for the `CubicalCover` function.
- `matplotlib` and `plotly` are used to create plots for the Mapper graph, providing visualization options.

Additionally, there is a weaker dependency on `sklearn` for test purposes. The `sklearn` library is used to ensure that the custom-defined estimators in *tda-mapper* are compatible with `sklearn`. This compatibility is not complete because *tda-mapper* includes classes like `BallCover` that can work with datasets defined as iterables of custom classes rather than just `numpy.ndarray`. This choice allows for greater flexibility in input data. The compatibility is primarily tested using sklearn's utilities for testing estimators, and the test coverage for compatibility is currently around 96%.

Overall, the software dependencies in *tda-mapper* are crucial for its functionality and enable users to generate Mapper graphs and visualize them effectively. The optional compatibility with `sklearn` ensures that the implementation aligns with widely-used machine learning standards.

```
import numpy as np
from sklearn.datasets import load_digits

X, y = load_digits(return_X_y=True)

from sklearn.cluster import AgglomerativeClustering
from sklearn.decomposition import PCA

from tdamapper.core import *
from tdamapper.cover import *
from tdamapper.clustering import *
from tdamapper.plot import *

lens = PCA(2).fit_transform(X)
mapper_algo = MapperAlgorithm(
    cover=CubicalCover(
        n_intervals=10,
        overlap_frac=0.25),
    clustering=AgglomerativeClustering())
mapper_graph = mapper_algo.fit_transform(X, lens)
mapper_plot = MapperPlot(X, mapper_graph, iterations=50)

fig = mapper_plot.with_colors(
    colors=y,
    cmap='jet',
    agg=np.nanmedian)
    .plot(title='digit', width=512, height=512)
fig.show(config={'scrollZoom': True})
```

5.3. **Benchmarks.** To evaluate the performance and scalability of our approach, we conducted a series of measurements to compute the Mapper graph's running time. During these benchmarks, we consistently maintained a specific type of open cover while systematically varying the lens dimension. We compared the running time of three different libraries, namely *Giotto-TDA* [14], *Kepler Mapper* [15], and *tda-mapper* [12]. This comparative analysis provides valuable insights into the behavior of these implementations when dealing with high-dimensional data. To simplify the process, we used *Principal Component Analysis* (PCA) as the lens, with the number of components ranging from 1 to 5. To ensure the reliability of our benchmarks, we utilized well-known datasets publicly available at the UCI Machine Learning Repository [6]. Our experiments were conducted on Python 3.10, leveraging *giotto-tda* 0.6.0, *Kepler Mapper* 2.0.1, and *tda-mapper* 0.3.0. All running time measurements are presented in seconds to provide a clear understanding of the performance.

Running time for digits (1797×64)			
lens features	Giotto-TDA	Kepler Mapper	tda-mapper
1	0.1	0.1	0.3
2	1.6	0.3	0.2
3	116.7	2.0	0.3
4	6951.7	15.1	0.5
5	???	109.4	0.8

Running time for MNIST_784 (70000×784)			
lens features	Giotto-TDA	Kepler Mapper	tda-mapper
1	7.0	32.8	6.0
2	27.8	73.6	11.5
3	429.0	204.7	16.3
4	20012.0	678.6	24.7
5	???	2260.8	40.1

References

[1] Karol Borsuk. "On the imbedding of systems of compacta in simplicial complexes". eng. In: *Fundamenta Mathematicae* 35.1 (1948), pp. 217–234. URL: http://eudml.org/doc/213158.

[2] Sergey Brin. "Near Neighbor Search in Large Metric Spaces". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 574–584. ISBN: 1558603794.

[3] *Collaborative data science.* Montreal, QC: Plotly Technologies Inc. URL: https://plot.ly.

[4] Paweł Dłotko. *Ball mapper: a shape summary for topological data analysis.* 2019. arXiv: 1901.07410 [math.AT].

[5] Paweł Dłotko, Davide Gurnari, and Radmila Sazdanovic. *Mapper-type algorithms for complex data and relations.* 2023. arXiv: 2109.00831 [math.AT].

[6] D. Dua and C. Graff. *UCI machine learning repository.* University of California, Irvine, School of Information; Computer Sciences. URL: http://archive.ics.uci.edu/.

[7] Caleb Geniesse, Samir Chowdhury, and Manish Saggar. "NeuMapper: A scalable computational framework for multiscale exploration of the brain's dynamical organization". In: *Network Neuroscience* 6.2 (June 2022), pp. 467–498. ISSN: 2472-1751. DOI: 10.1162/netn_a_00229.

[8] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference.* Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[9] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[10] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[11] D. Müllner and A. Babu. "Python Mapper: An open-source toolchain for data exploration, analysis, and visualization". In: (2013). URL: http://danifold.net/mapper.

[12] Luca Simi. *lucasimi/tda-mapper-python: v0.3.0.* Version v0.3.0. Feb. 2024. DOI: 10.5281/zenodo.10642382. URL: https://doi.org/10.5281/zenodo.10642382.

[13] Gurjeet Singh, Facundo Memoli, and Gunnar Carlsson. "Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition". In: *Eurographics Symposium on Point-Based Graphics.* Ed. by M. Botsch et al. The Eurographics Association, 2007. ISBN: 978-3-905673-51-7. DOI: 10.2312/SPBG/SPBG07/091-100.

[14] Giotto-TDA Tauzin. "A Topological Data Analysis Toolkit for Machine Learning and Data Exploration". In: (2020). arXiv: 2004.02551 [cs.LG].

[15] et al. Van Veen. "Kepler Mapper: A flexible Python implementation of the Mapper algorithm". In: *Journal of Open Source Software* 4.42 (2019), p. 1315. DOI: 10.21105/joss.01315.

[16] André Weil. "Sur les théorèmes de de Rham." In: *Commentarii mathematici Helvetici* 26 (1952), pp. 119–145. URL: http://eudml.org/doc/139040.

[17] Peter N. Yianilos. "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces". In: *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA.* Ed. by Vijaya Ramachandran. ACM/SIAM, 1993, pp. 311–321. URL: http://dl.acm.org/citation.cfm?id=313559.313789.