University of Bergen
Department of Informatics

A Benchmarking Suite for Persistent Homology

Author: Sondre Bergsvåg Risanger
Supervisor: Nello Blaser

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2024

Abstract

Topological data analysis (TDA), which is based on persistent homology, is a way of examining the underlying topological features of data. Machine learning pipelines based on TDA have proven effective in several areas of research; despite this, there does not yet exist a standardized benchmarking suite for TDA-based machine learning, especially focusing on testing different vectorization methods.

This thesis presents a benchmarking suite with four synthetic tasks, which was developed to test the ability of TDA pipelines to extract and use topological features of point clouds in machine learning. The underlying benchmarking tool used by the suite is an important contribution of this thesis, allowing the user to implement the different steps of the TDA pipeline separately, such that different combinations of TDA pipeline components can be tested. The benchmarking tool also reduces the amount of code the user must implement, as steps like splitting the datasets and performing model selection is implemented as a part of the benchmarking tool.

The usefulness of the benchmarking suite was evaluated by having it benchmark two non-TDA pipelines which were used as a baseline, and two TDA pipelines. The non-TDA pipelines did not perform well on the tasks, while the TDA pipelines performed well, outperforming the non-TDA pipelines on all tasks. This indicates that the benchmarking suite is suited for its intended purpose. While the suite was intended to also contain real-world datasets, that goal was ultimately not reached.

Acknowledgements

I would first like to thank my supervisor, Nello Blaser. His advice and feedback, as well as our many meetings throughout the writing of this thesis, have been invaluable, helping me stay on the right course.

I would also like to thank my fellow students, both for their feedback on my thesis, and for allowing me to hone my skills by letting me give them feedback on their work.

Finally, I would like to thank my friends and family, not only for their support during the writing of this thesis, but also for their support in all of my endeavors.

Sondre Bergsvåg Risanger
Monday 3rd June, 2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and motivation

Topological data analysis (TDA) is a way of analyzing data by looking at its shape rather than looking at the data directly. Taking a point cloud (a set of points) as an example, one may use combinations of the points to create a simplicial complex, a sort of generalization of graphs using components of dimensions higher than 1. The combinations included in the simplicial complex will vary based on some parameter r, commonly denoting a scale parameter for maximum distances between the combined points. We can then analyze the shape of the point cloud by looking at how the simplicial complex evolves as r changes. Those different simplicial complexes are together called a filtered simplicial complex. When analyzing the shape, we are more specifically interested in the appearances and disappearances of holes in the complex. This is known as the persistent homology of the complex, which is summarized in a persistence diagram. The persistence diagram is then vectorized to make it more suitable for machine learning [4]. This describes the general TDA pipeline: Generate a filtered simplicial complex and compute its persistent homology, which is then vectorized.

The use of TDA in machine learning has proven to be useful in several areas of research, for example tumor classification [9], neuroscience [19], and genomics [11]. Despite this, there is, at least to my knowledge, no standardized benchmarking suite for testing the performance of new TDA methods, in particular new vectorization methods. There does exist a Python package called *giotto-deep* which is intended for benchmarking of TDA pipelines; however, it is not an automated, standardized benchmarking suite, and it is built specifically for use with PyTorch [3].

The existence of standardized benchmarks have contributed to advancements in areas like object recognition, where ImageNet is a notable example [21]; model architectures such as AlexNet and ResNet were evaluated using ImageNet, and were winning entries in the ImageNet Large-Scale Visual Recognition Challenge in their respective years [13, 12].

This thesis therefore aims to make a benchmarking suite for TDA in machine learning with a predetermined set of tasks intended to test the capabilities of TDA pipelines specifically, allowing for easy comparison between pipelines. The focus will be on supervised learning.

1.2 Objectives

The objectives of this thesis can be split into two parts: The benchmarking tool itself, and the tasks it uses for testing.

First, the benchmarking tool should be as flexible as possible, such that most, if not all, TDA pipelines can be tested. This must be balanced with its ease of use, minimizing the amount of extra work needed to implement the pipeline for testing.

Second, the tasks included should be a combination of synthetic datasets with known ground truths, generated as needed by the benchmarking tool, and real-world datasets to test the pipelines in real-world scenarios. The synthetic tasks should include both classification tasks and regression tasks where the goal is to distinguish topological features. The included real-world datasets should be chosen based on the strengths and weaknesses of TDA pipelines found by the benchmarking results of the synthetic tasks.

1.3 Thesis outline

The rest of the thesis is structured as follows: Chapter 2 introduces the background theory used in the thesis work; Chapter 3 explains how the benchmarking tool is implemented and how to use the benchmarking suite, and describes the pipelines used to test the suite; Chapter 4 details how each of the synthetic datasets is generated and presents the results of each pipeline on the datasets; Chapter 5 discusses the results and evaluates the benchmarking suite against the objectives of the thesis; Chapter 6 gives suggestions for future work on the suite.

Chapter 2

Background

2.1 Probability theory

Different probability distributions are used in this thesis, as they are central in the process of generating data. This section will describe these distributions. It will also describe a sampling method know as rejection sampling, which is used in one of the data generators.

2.1.1 Probability distributions

The material in this section is based on Castañeda et al. [6] and will introduce the different probability distributions used in this thesis. The beta distribution and the multivariate normal distribution will be explained in detail. For the uniform and normal distributions, however, I assume that the reader is familiar with them and will thus only introduce the notation used.

The uniform distribution will use the notation $\mathcal{U}(A)$ to refer to a uniform distribution on the set A, e.g. $\mathcal{U}((0,1))$ for a uniform distribution in the interval $(0,1)$, or $\mathcal{U}(\mathbb{S}^d)$ for a uniform distribution on the d-sphere. The normal distribution will use the notation $\mathcal{N}(\mu, \sigma^2)$ with mean μ and standard deviation σ. Lastly, the notation $x \sim D$ will denote a variable x sampled from a distribution D.

Beta distribution

The beta distribution is a continuous distribution in the interval $(0, 1)$ with the parameters $a, b > 0$. This thesis will use the notation $\beta(a, b)$ when referring to the beta distribution.

To define its probability density function, its beta function has to be defined first, given by

$$B(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx.$$

Using the beta function, the probability density function of the beta distribution is defined as

$$f(x) = \begin{cases} \frac{1}{B(a,b)}x^{a-1}(1-x)^{b-1} & \text{if } x \in (0,1) \\ 0 & \text{otherwise.} \end{cases}$$

Here, the beta function normalizes the expression, making the area under the curve equal to 1 for valid values of a and b.

Figure 2.1 shows some of the different distributions generated by varying the a and b values of the beta distribution. For example, $\beta(0.5, 0.5)$ is U-shaped, $\beta(1, 1)$ is the same as $\mathcal{U}((0, 1))$, and $\beta(5, 5)$ has a bell curve shape.

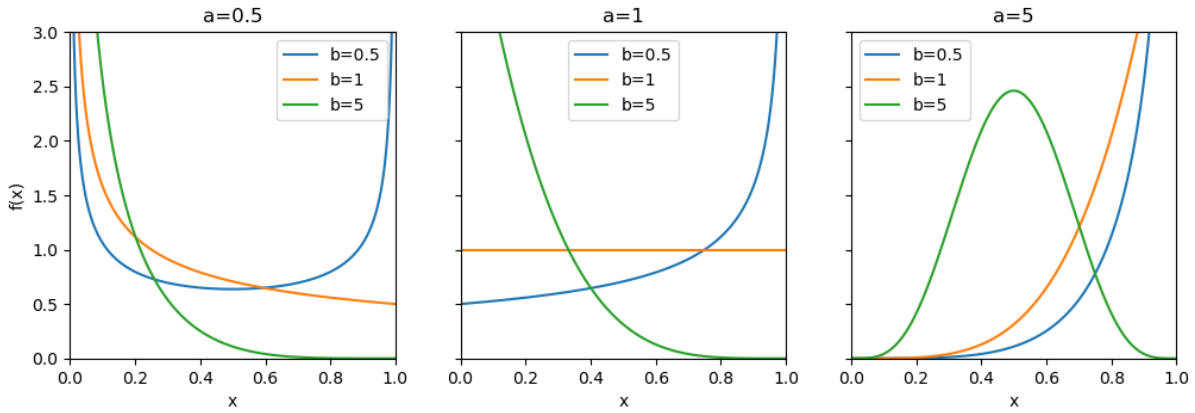

Figure 2.1: The probability density function of the beta distribution for varying values of a and b.

Multivariate normal distribution

A multivariate normal distribution is a distribution such that for vectors $\vec{x} := (x_1, ..., x_n)$ sampled from the distribution, any linear combination $\sum_{i=1}^{n} \alpha_i x_i$ has univariate normal distribution. Note that this section uses row vectors.

To generate an n-dimensional normally distributed vector, it uses an n-dimensional vector $\vec{\mu}$ as the mean parameter, and a positive semidefinite symmetric $n \times n$ matrix Σ as the variance parameter. With these parameters, the multivariate normal distribution will use the notation $\mathcal{N}(\vec{\mu}, \Sigma)$.

One way to sample from a multivariate normal distribution is to use the eigendecomposition of the variance matrix, i.e. $\Sigma = A\Lambda A^T$, where Λ is the diagonal matrix consisting of the eigenvalues, and A is the matrix of the corresponding eigenvectors. A vector $\vec{y}$ is generated with components $y_i \sim \mathcal{N}(0, \Lambda_{i,i})$. This vector is then transformed such that we get a resulting vector

$$\vec{x} \sim \mathcal{N}(\vec{\mu}, \Sigma) = \vec{y}A + \vec{\mu}.$$

This method can be used to sample from any multivariate distribution.

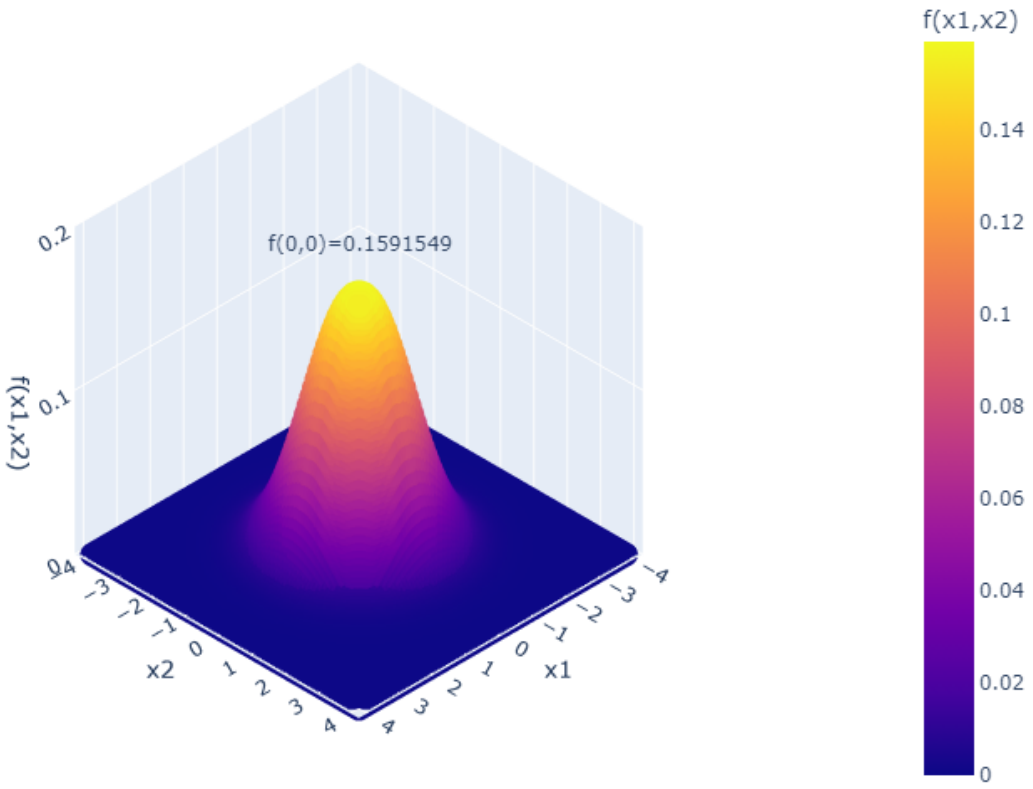

Figure 2.2: The probability density function of the standard bivariate normal distribution, where $x1$ and $x2$ denote the first and second component of $\vec{x}$ respectively, and $f(x1, x2)$ is $f(\vec{x})$. The figure shows the function in the interval $x_1, x_2 \in [-4, 4]$, with the annotated point at $f(0, 0)$ being the maximum of the function.

The multivariate normal distribution has some useful properties. For example, for a sampled vector $\vec{x} \sim \mathcal{N}(\vec{\mu}, \Sigma)$ with a diagonal Σ, each vector component x_i is independent

such that $x_i \sim \mathcal{N}(\mu_i, \Sigma_{i,i})$. This means that a standard multivariate normal distribution $\mathcal{N}(\vec{0}, I)$ has independent components, each from the standard normal distribution $\mathcal{N}(0, 1)$.

The probability density function of the multivariate normal distribution is given by

$$f(\vec{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}}(\det \Sigma)^{-\frac{1}{2}} \exp\left[-\frac{1}{2}(\vec{x} - \vec{\mu})\Sigma^{-1}(\vec{x} - \vec{\mu})^T\right].$$

Since the function uses the inverse of Σ, it only works for positive *definite* Σ. To give an idea of how it looks, figure 2.2 shows the function for the standard bivariate normal distribution.

2.1.2 Rejection sampling

In computing, random number generators are usually based on a uniform distribution. Then, to sample from a non-uniform probability density function, some method must be used to change the final distribution of the generated values. One of the simplest and most efficient of these is the inversion method. It applies the inverse of a target distribution's cumulative density function to uniformly generated values, which results in correctly distributed random values [15]. This of course assumes that we know the inverse of the cumulative density function, which is not always the case. We might therefore consider another method which is more flexible, called rejection sampling.

Rejection sampling only requires that we have a random number generator with probability density function $\pi(x)$ and a target distribution D with probability density function $p(x)$ such that

$$M\pi(x) \geq p(x)$$

for some constant $M > 0$. If we sample an $x' \sim \pi(x)$ and a $y' \sim \mathcal{U}((0, M\pi(x')))$ and accept x' only if $y' \leq p(x')$, the accepted values will be distributed according to D. The probability of x' being accepted is then the ratio of the value of each of the functions at x',

$$\frac{p(x')}{M\pi(x')}.$$

In this thesis, the random number generator will have a uniform distribution, which means that we can view rejection sampling more intuitively as uniformly sampling a random point in a box, and accepting x' if the point is under the curve of p [20].

To show how this works, rejection sampling will be shown as a way to sample from the beta distribution. This only works for $a, b \geq 1$, since the density of the beta distribution

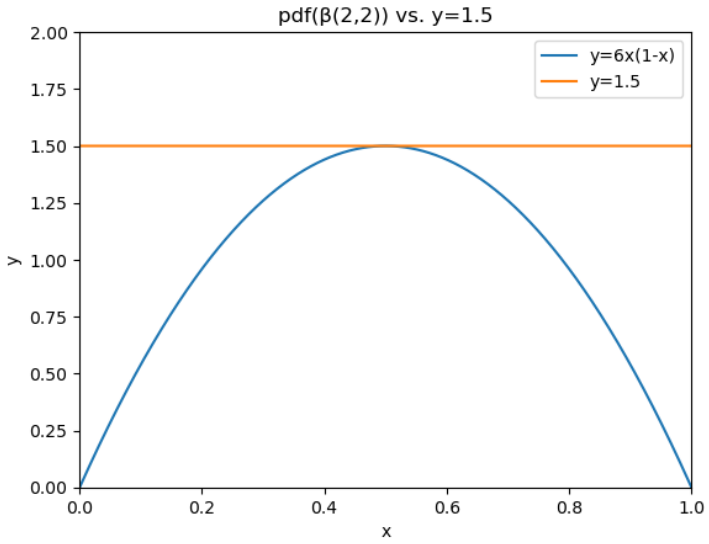

Figure 2.3: The probability distribution function of $\beta(2,2)$, $6x(1-x)$, plotted against 1.5 times the probability distribution function of $\mathcal{U}((0,1))$, $y = 1.5$. Since the scaled function of the distribution we can sample from has a higher value for all $x \in (0,1)$ than that of the target distribution, we can use rejection sampling to indirectly sample from the target distribution.

approaches ∞ for a or b less than 1. This is not a problem however, as this thesis will only use the beta distribution with $a, b \geq 1$.

Using $a, b = 2$ as an example, the probability density function of $\beta(2,2)$ is $p(x) = 6x(1-x)$, and the probability density function of $\mathcal{U}((0,1))$ is $\pi(x) = 1$. The maximum of $p(x)$ is 1.5, so $p(x) \leq M\pi(x)$ for $M \leq 1.5$ and we set $M = 1.5$. The functions $p(x)$ and $1.5\pi(x)$ are plotted in figure 2.3. We can then sample two values, $x' \sim \mathcal{U}((0,1))$ and $y' \sim \mathcal{U}((0,1.5))$, and accept x' if $y' \leq 6x'(1-x')$. The accepted x' will then be distributed according to $\beta(2,2)$.

Note that this is only intended as an example of how to use rejection sampling, and how to sample from a beta distribution. While this method becomes decreasingly efficient as a and b increase, much more efficient methods of sampling from the beta distribution exist [20].

2.2 Topology

Topology is closely related to geometry as both are used to describe shapes, but while geometry uses metrics like distances and angles to describe objects, topology is more

generalized and measures shapes based on invariance under smooth deformations. That is, if two objects can be "reshaped" into each other and back without any tearing or gluing, they are topologically equivalent, and are said to be homeomorphic.

A common example of this is the topological equivalence between a coffee mug and a donut (more formally known as a torus). The two objects are geometrically distinct, but if we imagine a coffee cup made of some pliable material, we could reasonably expect to be able to reshape it into a torus without any gluing or tearing, as shown in figure 2.4.

Our use case for topology is data analysis. Using point clouds as an example, there are many well-known tools for specific cases of point clouds, like linear regression for points lying in an approximate line, or clustering algorithms for points in several distinct clusters. However, each of these situations requires us to choose a suitable way of analyzing the data, which may become difficult for data with less common or less clear shapes. Topological data analysis gives us tools to analyze the shape of several types of data without a priori knowledge [4].

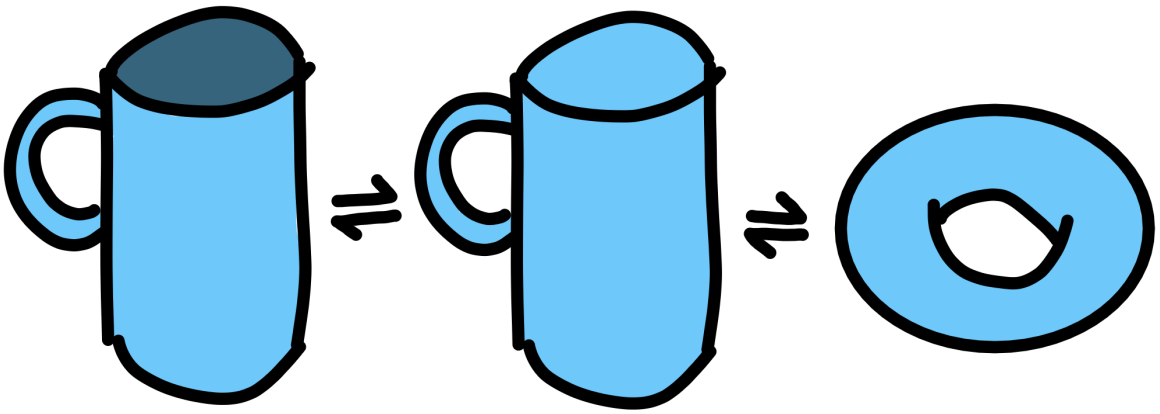

Figure 2.4: The continuous deformation of a coffee mug to a torus. The bottom of the mug is expanded to fill the cylinder, which is then compressed to match the thickness and curvature of the handle, resulting in a torus. These operations may also be reversed to deform a torus to a coffee mug.

To describe topology more formally, we need to define topological spaces. For a set of points $\mathbb{X}$, we can make a set of its subsets, denoted $\mathcal{U}$, which define the open sets. Here, $\mathcal{U}$ is said to define the topology on $\mathbb{X}$, giving the topological space ($\mathbb{X}$, $\mathcal{U}$). The following conditions also need to be satisfied for it to be a valid topology:

1. $\mathbb{X}$ and the empty set $\emptyset$ are in $\mathcal{U}$.

2. The intersection of any finite number of sets in $\mathcal{U}$ are in $\mathcal{U}$.

3. The union of any number of sets in $\mathcal{U}$ are in $\mathcal{U}$.

A simple and useful way of defining a topology is by defining a base for the topology. For a set $\mathbb{X}$, a collection $\mathcal{B}$ of subsets of $\mathbb{X}$ is a base given two conditions:

1. For each element $x \in \mathbb{X}$, there exists a $B \in \mathcal{B}$ such that $x \in B$.

2. If x is in an intersection of two elements $B_1, B_2 \in \mathcal{B}$, x must also be in a $B_3 \in \mathcal{B}$ given by the intersection of B_1 and B_2.

This base is then said to generate a topology on $\mathbb{X}$ where a set $U \subseteq \mathbb{X}$ is open if each $x \in U$ is also contained in some $B \subseteq \mathbb{X}$, $B \in \mathcal{B}$.

Since this thesis will be mostly concerned with point clouds, it will be helpful to have some connection between point clouds and topological spaces. The point clouds discussed here will lie in some space $\mathbb{R}^n$, which is a metric space. A metric space is a set $\mathbb{X}$ with a distance function $\delta_{\mathbb{X}} \colon \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. This distance function must satisfy some conditions:

1. Distances must be non-negative such that $\delta_{\mathbb{X}}(x_1, x_2) \geq 0$.

2. The distance $\delta_{\mathbb{X}}(x_1, x_2)$ is 0 if and only if x_1 and x_2 are identical.

3. The function is symmetric, i.e., $\delta_{\mathbb{X}}(x_1, x_2) = \delta_{\mathbb{X}}(x_2, x_1)$.

4. The triangle inequality holds, that is, $\delta_{\mathbb{X}}(x_1, x_3) \leq \delta_{\mathbb{X}}(x_1, x_2) + \delta_{\mathbb{X}}(x_2, x_3)$.

In the case of $\mathbb{R}^n$, the distance function is the Euclidean distance function $\delta_{\mathbb{R}^n}(\vec{x}, \vec{y}) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2}$.

Metric spaces can be used to define topological spaces. For a metric space with a set $\mathbb{X}$, open sets of its topological space are given by the $U \subset \mathbb{X}$ where each element $x \in U$ has some open ball around it contained in U, i.e., $B_r(x) \coloneqq \left\{ y \in \mathbb{X} \mid \delta_{\mathbb{X}}(x, y) < r \right\} \subseteq U$ for some radius r [18].

There are some properties topological spaces may have, Hausdorff and second countable, which will be useful to define: If any two points $x, y \in \mathbb{X}$, $x \neq y$ have open sets $x \in U_1 \in \mathcal{U}$, $y \in U_2 \in \mathcal{U}$ with empty intersection, the topological space defined by $(\mathbb{X}, \mathcal{U})$ is said to be Hausdorff. If the base of a topological space is countable, the space is said to be second countable.

2.2.1 Manifolds

This thesis will mostly discuss a type of topological space called (n-)manifolds. These are spaces which locally, i.e. if we look at the space immediately around a point, are homeomorphic to an open subset of $\mathbb{R}^n$. To be more specific, it will discuss 2-manifolds in particular, also known as surfaces, which locally look like the plane [10].

Before describing the manifolds relevant to this thesis, it would be useful to accurately define manifolds. To do so, we first need the concepts of charts and atlases. Consider a topological space $(\mathbb{X},\mathcal{U})$. If we have a homeomorphism from an $U \in \mathcal{U}$ to an open subset of $\mathbb{R}^n$, we call it a chart. We can then make a collection of charts, called an atlas, such that each point $x \in \mathbb{X}$ lies in the domain of at least one chart. An n-manifold can then be defined as a topological space which has an atlas, and is Hausdorff and second countable [18].

As the condition of the existence of an atlas may imply, manifolds can be entirely represented in Euclidean space, which we then call an immersion. If the immersion has no overlapping points, it is called an embedding.

There are two properties which can be used to tell surfaces apart: the genus g of the surface, and the orientability. The genus is used to count how many closed curves we can cut the surface by without separating it into several components. For example, any closed curve on a sphere will divide it into at least two components, while one may draw a closed curve around the tube of a torus and get a cylinder which is connected. Orientability is used to describe whether a surface has an inside and outside or not. A sphere, for instance, is orientable with a clear inside and outside. The Klein bottle, however, is a surface with no clear inside or outside, and is therefore non-orientable.

As one might assume from the examples above, the simplest non-trivial n-manifold is what is known as the n-sphere, being orientable and having genus 0. The most well-known examples are the circle for $n = 1$, and the sphere for $n = 2$. The n-sphere has several embeddings in $\mathbb{R}^n$, but the most common type of embedding is the set of points with distance r from the origin. Usually, when discussing the generation of n-spheres in this thesis, it will be referring to the unit n-sphere, which is the embedding mentioned above with $r = 1$.

A torus is the orientable 2-manifold of genus 1, and is often embedded in R^3 as a donut. One may intuitively describe this embedding as the surface of revolution of a circle.

Figure 2.5 shows a unit sphere, a donut embedding of the torus, and an example of a genus 2-torus.

Figure 2.5: A sphere, a torus, and a genus 2-torus, shaded to convey depth.

To get orientable surfaces of genus $g > 1$, one may take the connected sum of several toruses to make a genus g torus. To get the connected sum of two surfaces, one draws a closed curve on each surface to remove a piece homeomorphic to an open disk, and connect the two surfaces by the closed curves [10]. An example of this is shown in figure 2.6, where the ends of two toruses are removed such that the toruses can be connected to make a genus 2-torus.

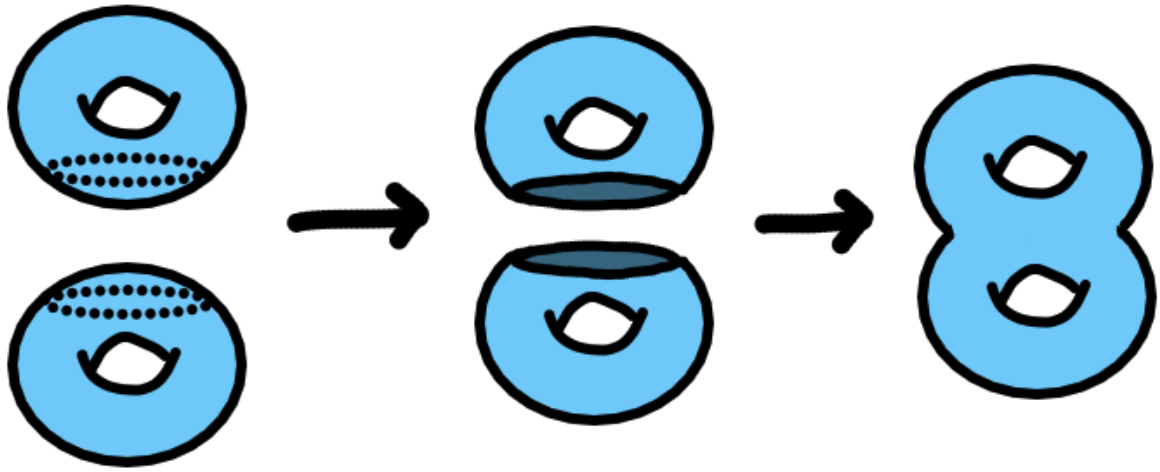

Figure 2.6: The genus 2-torus as the connected sum of two toruses. A piece homeomorphic to an open disk is removed from the end of each torus, and they are glued together where the disks were removed from, resulting in a genus 2-torus.

2.2.2 Simplicial complexes

The goal of topological data analysis is to analyze the shape of the data to give some information about its topology. Point clouds, being just collections of points, do not have much structure as is. Therefore, the first step in the topological data analysis process is to generate a simplicial complex based on the data, to give it shape.

Simplicial complexes are collections of n-simplices, which can be viewed as the simplest n-dimensional objects. For example, for dimensions 0 through 3, their respective n-simplices are colloquially known as the point, the line segment, the triangle and the tetrahedron. Figure 2.7 below shows examples of these simplices.

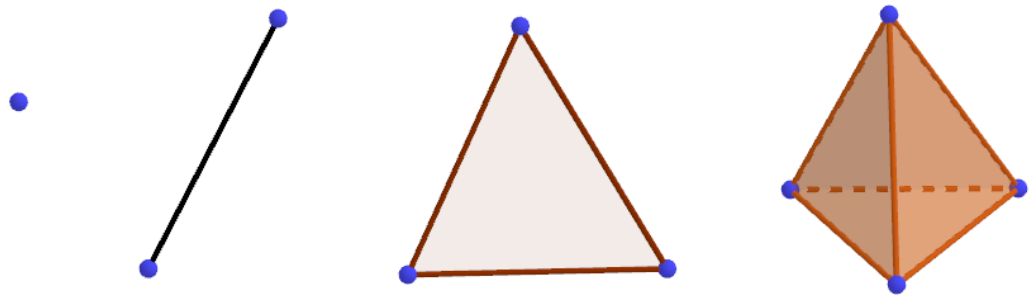

Figure 2.7: The n-simplices for $0 \leq n \leq 3$, in order.

An n-simplex is denoted by the set of its vertices $\{x_0, x_1, \ldots, x_n\}$ where the vectors given by $x_i - x_0$ are linearly independent, and is formally constructed as the collection of points given by

$$\left\{\sum_{i=0}^{n} \lambda_i x_i \mid \lambda_i \geq 0, \sum \lambda_i = 1\right\}.$$

This is known as the convex hull of the vertices. We can also take convex hulls of proper subsets of the vertices of an n-simplex to get lower-dimensional simplices, known as its proper faces, which bound its volume. It is also worth noting that a simplex is a face of itself, but not a proper face, as its vertices are an improper subset of its vertices [10].

To construct a simplicial complex K, we only need to define a collection of simplices σ_i satisfying the following conditions:

1. For each simplex $\sigma \in K$, all its proper faces τ must also be contained in K.

2. The intersection of two simplices in K must either be empty or a face of both simplices.

With these conditions, it becomes clear that a simplicial complex consisting of only 0-simplices and 1-simplices is homeomorphic to a graph where the 0-simplices map to the vertices, and the 1-simplices map to the edges. Thus, one might view simplicial complexes as generalizations of graphs for higher dimensions [18].

We still need some way to generate simplicial complexes from point clouds. To do this, we can construct different types of complexes. This thesis will use the alpha complex.

To define the alpha complex, we first need to define Voronoi cells and closed balls. A Voronoi cell of a point x in a set of points $\mathbb{X} \subseteq \mathbb{R}^n$ is given by all points in $\mathbb{R}^n$ for which x is the closest point of $\mathbb{X}$, i.e.,

$$V_x = \big\{p \in \mathbb{R}^n \mid \delta(x,p) \leq \delta(y,p),\ y \in \mathbb{X}\big\}.$$

A closed ball is simply all points in $\mathbb{R}^n$ within a radius r of x, similar to the open ball except that it includes the points with distance exactly r.

To get the alpha complex with scale parameter r, we first take the intersection of each point's Voronoi cell and closed ball at radius r, denoted $R_x(r)$ for a point $x \in \mathbb{X}$. With these $R_x(r)$, we can define the alpha complex as

$$\text{Alpha}(r) = \big\{\sigma \subseteq \mathbb{X} \mid \bigcap_{x \in \sigma} R_x(r) \neq \emptyset\big\}.$$

This means that if the $R_x(r)$ of n points overlap, an $(n-1)$-simplex is created with those points as vertices; two $R_x(r)$ overlapping creates an edge, three $R_x(r)$ overlapping creates a triangle, and so on.

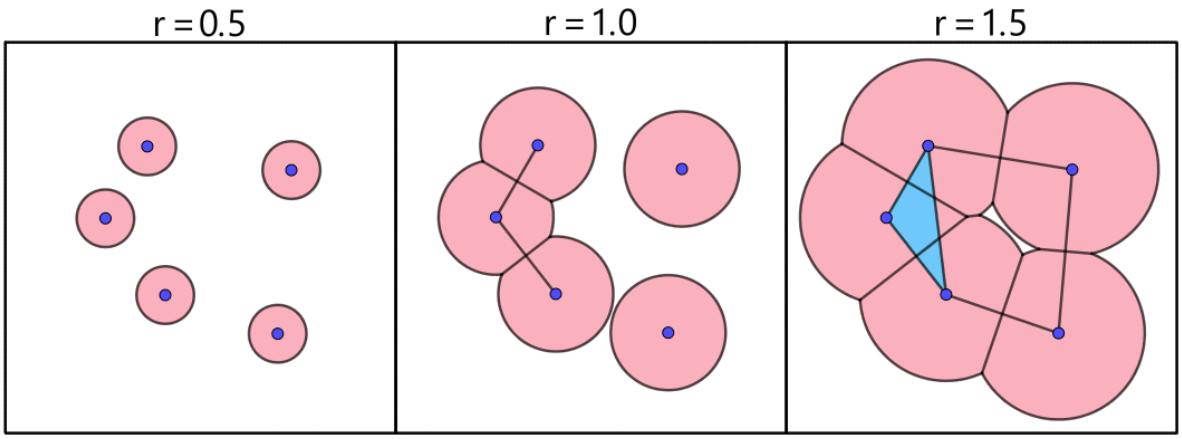

Figure 2.8: Alpha complexes of the same set of points for $r = 0.5$, $r = 1.0$, and $r = 1.5$. As the red areas (representing $R_x(r)$) start to overlap, 1-simplices (lines between points) and 2-simplices (blue triangles between points) are added.

An example of alpha complexes for increasing r with the same set of points is shown in figure 2.8. What is interesting to note here is that as r increases and edges are added, some edges may create cycles in the complex which are not filled in by triangles, as can be seen with the four rightmost points at $r = 1.5$. There appears to be a "hole" in the complex, analogous to the genus of manifolds, which will disappear as r increases further.

Thus, it is not only interesting to us how the Alpha complex looks, but also how it changes as r is increased. In fact, for a finite point cloud, the complex will only change a finite number of times as r approaches infinity. We can then describe a series of complexes

$$\emptyset = \text{Alpha}_0 \subseteq \ldots \subseteq \text{Alpha}_k \subseteq \ldots \subseteq \text{Alpha}_n \tag{2.1}$$

with each different Alpha complex as r increases, where Alpha_n is the complex as r approaches infinity. This series of complexes is known as a filtration [10]. In particular, this paper will refer to $Alpha_n$ and its filtration as a filtered simplicial complex.

Generating these filtered simplicial complexes is the first step of topological data analysis. The next step is to get topological information from them using persistent homology, which is covered in the next section.

2.3 Persistent homology

Homology can be viewed as studying the number of holes in a simplicial complex. Using a filtered simplicial complex, we may compute what are called its homology groups at

several values of r to check its "holedness" at that value. We are, however, unsure about which value of r is the closest to the ground truth for the point cloud. Therefore, we instead use persistent homology to study the persistence of the holes as the complex evolves [18].

This section will first introduce group theory and homology groups, which lay the foundation for persistent homology.

2.3.1 Group theory

This section, introducing group theory concepts which are either necessary or useful for constructing homology groups, is based on Ledermann and Weir [14].

Group basics

This section will cover the basics of group theory, i.e., the definitions of groups and subgroups.

A group G is defined as a set G along with a binary operation which satifies the following conditions:

Closure: If a and b are in G, their product $ab = c$ must also be in G.

Associativity: For any $a, b, c \in G$, $(ab)c = a(bc)$.

Identity element: G must contain an identity element e such that $ae = ea = a$ for all $a \in G$.

Inverse element: Each element $a \in G$ has an inverse element $a^{-1} \in G$ such that their product is the identity element, i.e., $aa^{-1} = a^{-1}a = e$.

Additionally, an Abelian group is a group where the operation also satisfies commutativity, that is, $ab = ba$ for any $a, b \in G$. It should also be noted that in cases where the operation is addition, to avoid confusion, a product ab is instead written explicitly as the sum $a + b$, and an inverse element a^{-1} is written as $-a$.

One example of a group is $(\mathbb{R}, +)$, the group of real numbers with addition. It is easy to see that this is a valid group: Any sum of its elements is in $\mathbb{R}$, addition is associative,

0 is the identity element such that $a + 0 = a$ for any $a \in \mathbb{R}$, and each element $a \in \mathbb{R}$ has an inverse element $-a \in \mathbb{R}$ such that $a + (-a) = 0$. We can also see that since addition is commutative, $(\mathbb{R}, +)$ is an Abelian group.

For a group G, we may take a subset H of G and use it along with the operation of G to define a subgroup H, written as $H \leqq G$. This subgroup must also satisfy closure, and contain the identity element of G and the inverse elements.

As an example, the group of integers with addition, $(\mathbb{Z}, +)$, is a subgroup of $(\mathbb{R}, +)$. This is easy to check: $\mathbb{Z}$ is a subset of $\mathbb{R}$ and contains the identity element 0, any sum of integers is an integer, and the inverse element of an integer is itself an integer.

For the group of integers with addition, note that any element can be made using 1 or its inverse -1 ($1 + 1 = 2$, $(-1) + (-1) = -2$, and so on). We can therefore call 1 the generator of $\mathbb{Z}$. In general, for a group generated by elements $a, b, \ldots \in G$ and their inverses, we use the notation $\text{gp}\{a, b, \ldots\}$. These elements may be redundant, using $\text{gp}\{1, 2\} = \mathbb{Z}$ as an example.

Quotient groups

This section introduces quotient groups, which is what homology groups are defined as.

We can construct something called cosets based on an element of a group G and a subgroup $H \leqq G$. There are left cosets, defined as $gH \coloneqq \{gh \mid h \in H\}$ for some element $g \in G$. Similarly, there are right cosets $Hg \coloneqq \{hg \mid h \in H\}$. When $gH = Hg$ for all elements of G, we say that H is a normal subset of G, denoted $H \trianglelefteq G$. The condition of commutativity means that each subgroup of an Abelian group is trivially normal.

With a group G and a normal subgroup $H \trianglelefteq G$, we can define a quotient group G/H. Its set is the set of all unique cosets of H, i.e., $\{gH \mid g \in G\}$, and the operation is the multiplication of sets, $aHbH \coloneqq \{ah_1bh_1 \mid ah_1 \in aH, ah_2 \in bH\}$ for two sets aH and bH.

To see that the quotient group satisfies the conditions of a group, it will be useful to first show that $HH = H$. H is closed, so any hH, $h \in H$ is a subset of H, and H contains the identity element, so $H = eH \subseteq HH \subseteq H$. In other words, HH can only contain elements of H, and it must also contain all elements of H, so $HH = H$.

We can now check that G/H is a group, keeping in mind that $aH = Ha$ since H is normal:

- Closure is given by $aHbH = abHH = abH$ for any $a, b \in G$ by the closure of G.
- Associativity, $(aHbH)cH = aH(bHcH)$, $a, b, c \in G$, is satisfied as multiplication of sets is associative.
- The identity element is H: $HaH = aHH = aH$ for any $a \in G$.
- The inverse of an element aH is $a^{-1}H$: $aHa^{-1}H = aa^{-1}HH = eH = H$ for any $a \in G$.

Thus, G/H defines a group.

A simple example of a quotient group is $\mathbb{Z}/2\mathbb{Z}$, where $2\mathbb{Z} \coloneqq \{2 \cdot k \mid k \in \mathbb{Z}\} = \{\ldots, -2, 0, 2, 4, \ldots\}$. The quotient group is then the set $\{g+2\mathbb{Z} \mid g \in \mathbb{Z}\}$, which we quickly see only contains two cosets, $\{\ldots, -2, 0, 2, 4, \ldots\}$ for even g and $\{\ldots, -1, 1, 3, 5, \ldots\}$ for odd g.

The elements within each coset are said to be equivalent, relative to the subgroup; we thus call these cosets equivalence classes. For each equivalence class, we may pick one element as a representative for the class. In this case, we can use [0] to denote the coset containing 0, and [1] for the coset containing 1. We then see that these equivalence classes form the group of integers with addition modulo 2, as $0 \equiv 2 \pmod 2 \equiv -2 \pmod 2 \equiv \ldots$ and $1 \equiv 3 \pmod 2 \equiv -1 \pmod 2 \equiv \ldots$.

Homomorphisms

This section introduces homomorphisms, which are used to define the groups of which the homology group is a quotient group.

We can define maps between groups. These maps are called homomorphisms (not to be confused with homeomorphisms), and are maps $\theta : G \to G'$ mapping all elements of one group to elements of the other. Here, we call $\theta g = g' \in G'$ the image of g under θ. A homomorphism only needs to satisfy one condition: For all $a, b \in G$, $(\theta a)(\theta b) = \theta(ab)$.

From this condition, we can derive two useful properties of homomorphisms. First, the identity of G is mapped to the identity of G' (denoted e'), as shown by $(\theta e)(\theta g) = \theta(eg) = \theta g = e'(\theta g)$ for $g \in G$. Second, the inverse of an element $g \in G$ is mapped to the inverse of θg. Since $(\theta g)(\theta g^{-1}) = \theta(gg^{-1}) = \theta(e) = e'$, we see that θg and θg^{-1} must be each other's inverses.

There are two sets associated with homomorphisms which will be useful to us, the kernel and the image. The kernel of a homomorphism θ is denoted as $\ker \theta$. We define it as the set of all elements of G which are mapped to e', that is, $\ker \theta \coloneqq \{g \mid g \in G, \theta g = e'\}$. The image of θ, written as $\operatorname{im} \theta$, is the image of the set G under θ, i.e., $\operatorname{im} \theta \coloneqq \{g' \mid g' \in G', g' = \theta g\}$.

We will also need to know that the kernel and the image of a homomorphism θ form groups with the operation of G and G' respectively. Since their operations come from groups, we know that they are associative. Both of them have an identity element as $e \in G$ maps to $e' \in G'$. Closure for the kernel is given by $(\theta a)(\theta b) = e'e' = e' = \theta(ab)$. For the image, closure is given by $(\theta a)(\theta b) = a'b' = \theta(ab)$. The kernel has inverse elements since $(\theta g)(\theta g^{-1}) = \theta(gg^{-1}) \Rightarrow e'(\theta g^{-1}) = e'$, meaning that for a $g \in \ker \theta$, its inverse g^{-1} must also map to the identity. The inverse elements of the image is trivially given by the previously stated fact that $\theta g^{-1} = (g')^{-1}$ for any $g \in G$.

With these group concepts, we can now introduce homology groups.

2.3.2 Homology

Homology is, as mentioned earlier, a way of counting the holes of a simplicial complex. This is done by computing the homology groups of the complex, which in turn requires three other kinds of groups: p-chain groups, p-cycle groups, and p-boundary groups. This section, which is based on Edelsbrunner and Harer [10], will explain these groups, and show a matrix reduction algorithm for computing homology.

A p-chain is a sum of p-simplices, written with coefficients a_i and p-simplices σ_i as $\Sigma a_i \sigma_i$. The coefficients will in our case be $a_i \in \mathbb{Z}_2$, such that for any p-chain, each p-simplex is either in it or not.

The p-chain group C_p of a complex K consists of the set of all its p-chains, with the operation being addition of its coefficients. Thus, the p-chain group can be written as the group generated by all p-simplices, $\operatorname{gr}\{\sigma \mid \dim \sigma = p\}$. Since each simplex is unique, none of the generators are redundant. Note that since the operation is addition, the chain groups are Abelian.

There exist homomorphisms from each chain group of dimension p to the chain group of dimension $(p-1)$. These are called boundary homomorphisms, or boundary maps,

$\partial_p : \mathsf{C}_p \to \mathsf{C}_{p-1}$, and map each p-simplex to the sum of its dimension $(p-1)$ faces. The boundary of a p-simplex σ defined by the vertices $\{x_0, \ldots, x_p\}$ can be written as

$$\partial_p \sigma = \sum_{i=0}^{p} \{x_0, \ldots, \overline{x_i}, \ldots, x_p\}$$

where the bar denotes that the vertex is not included in the face. Since a p-chain is a sum of p-simplices, the boundary of a p-chain c can be written as the sum of its simplices' boundaries,

$$\partial_p(c) = \partial_p(\sum^{i \in I} a_i \sigma_i) = \sum^{i \in I} a_i \partial_p(\sigma_i), \; I = \{i \mid \dim \sigma_i = p\}$$

satisfying the homomorphism condition, i.e., $\theta(a)\theta(b) = \theta(ab)$.

With the boundary homomorphisms, we can use the kernel of ∂_p and the image of ∂_{p+1} to form two subgroups of the p-chain group: The p-cycle group Z_p and the p-boundary group B_p, respectively. The p-cycle group consists of all p-chains where the dimension $(p-1)$ faces are shared between the simplices an even number of times, such that they cancel out and the boundary is 0 (making them part of the kernel). The p-boundary group simply consists of all p-chains that are the boundary (i.e., the image) of some $(p+1)$-chain.

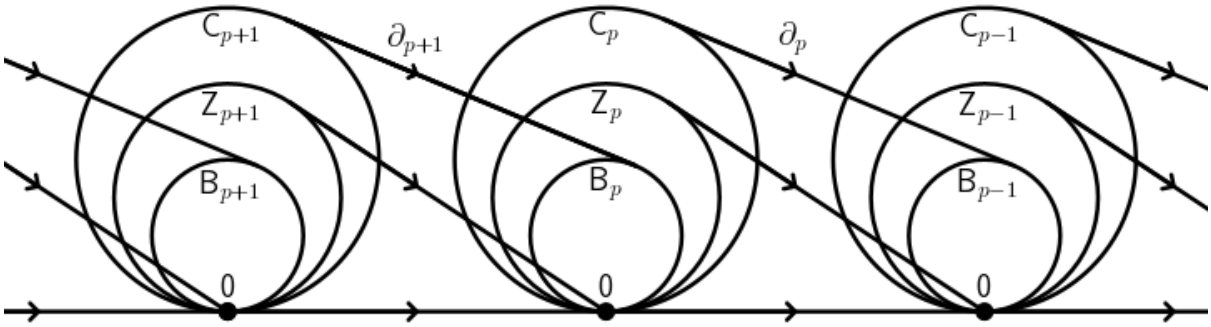

Figure 2.9: The relationships between the chain, cycle, and boundary groups for dimensions $p+1$, p, and $p-1$, along with boundary maps. This clearly illustrates how B_p is a subgroup of Z_p, which itself is a subgroup of C_p. It also shows Z_p as $\ker \partial_p$, and B_p as $\operatorname{im} \partial_{p+1}$. Figure adapted from Edelsbrunner and Harer [10].

There is an important relationship between the p-chain group, p-cycle group, and the p-boundary group, namely $\mathsf{B}_p \trianglelefteq \mathsf{Z}_p \trianglelefteq \mathsf{C}_p$, as shown in figure 2.9. The fact that B_p is a (normal) subgroup of Z_p might not be obvious, so it will need to be explained. Since they are both subgroups of C_p, it is clear that the relationship holds as long as the

boundaries are a subset of the cycles. The fundamental lemma of homology states that $\partial_p(\partial_{p+1}\sigma) = 0$, i.e., the boundary of a boundary is zero. We can see this by first writing the boundary of a simplex as $\partial_{p+1}\sigma = \sum_{i=0}^{p+1} \tau_i$ where τ_i is the face without vertex x_i. This notation can be extended to $\tau_{i,j}$, denoting a dimension $(p-1)$ face of σ without vertices x_i and x_j. It is then easy to see that for any vertices $x_i, x_j \in \sigma$, $\tau_{i,j}$ will be added to $\partial_p(\partial_{p+1}\sigma)$ twice (by $\partial_p\tau_i$ and $\partial_p\tau_j$), canceling it out. This means that the fundamental lemma of homology holds, and im $\partial_{p+1} \subseteq \ker \partial_p$ as required.

Using the p-cycle groups and the p-boundary groups, we can finally define the p-th homology group. It is given by their quotient group, i.e., $\mathsf{H}_p = \mathsf{Z}_p/\mathsf{B}_p$. The equivalence classes of a homology group are called homology classes, and their elements are then the cycles which are equivalent relative to the boundaries. This means that each homology class represents a unique combination of holes, such that log_2 of the number of p-homology classes is the number of p-dimensional holes in the complex. This number is commonly called the p-th Betti number, written as β_p.

Each of the groups discussed here has a rank according to its number of irredundant generators. For the p-chain group, for example, its rank is the number of p-simplices. The rank of the p-th homology group is rank $\mathsf{Z}_p -$ rank B_p, and is equivalent to the p-th Betti number. This means that we may find the number of p-dimensional holes in the complex by finding the ranks of the p-cycle group and the p-boundary group. To do so, we need to define boundary matrices, which we will use a matrix reduction algorithm on to get the ranks.

The boundary matrices of a complex are defined for each dimension, such that the columns of a p-boundary matrix represent the p-simplices, and the rows represent the $(p-1)$-simplices. The element of each cell is 1 if the row's simplex is a face of the column's simplex, and 0 if it is not. The matrix thus encodes the p-th boundary map, ∂_p. As an example, the 1-boundary matrix of a simplicial complex representing a hollow triangle, $K = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}\}$, is

$$\begin{array}{c} \\ \{x_1\} \\ \{x_2\} \\ \{x_3\} \end{array} \begin{array}{c} \begin{array}{ccc} \{x_1, x_2\} & \{x_1, x_3\} & \{x_2, x_3\} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \end{array}.$$

The matrix reduction algorithm is used to convert an $m \times n$ matrix to Smith normal form, where the entries $e_{1,1}, \ldots, e_{d,d}$ for some d are 1, and the rest of the entries of the matrix are 0. To perform the algorithm, start with $x = 1$:

- Find $k \geq x$, $l \geq x$ such that $e_{k,l} = 1$.
  - If such an entry does not exist, the algorithm is finished.
- Swap row k with row x, and column l with column x.
- For each row i, $x < i \leq m$, if $e_{i,x} = 1$, add row x to row i.
- For each column j, $x < j \leq n$, if $e_{x,j} = 1$, add column x to column x.

Repeat the steps for $x = 2$, $x = 3$, etc., until the algorithm is finished.

The final matrix then provides information about the ranks of some groups the following way: As each p-simplex and $(p-1)$-simplex respectively corresponds to one column and one row each, the width n of the matrix is the rank of the p-chain group, and the height m of the matrix is the rank of the $(p-1)$-chain group. The rank of the matrix, d, is equivalent to the rank of the $(p-1)$-boundary group, i.e., the group defined by the image of the p-th boundary map. The number of zero columns, $n - d$, is the rank of the p-cycle group, i.e. the group defined by the kernel of the p-th boundary map.

To show how this works, we will calculate β_1 of the simplicial complex K defined earlier, which will require the rank of Z_1 and the rank of B_1. Since there are no 2-simplices and therefore no 2-boundary matrix, rank B_1 is trivially zero. To find rank Z_1, we will perform the matrix reduction algorithm on the 1-boundary matrix of K. First, we perform the steps for $x = 1$:

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{r_2 = r_2 + r_1} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{c_2 = c_2 + c_1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Note that no row or column swap was performed as $e_{1,1}$ was already 1. We repeat the steps for $x = 2$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \xrightarrow{r_3 = r_3 + r_2} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \xrightarrow{c_3 = c_3 + c_2} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

For $x = 3$, we see that there are no entries $e_{k,l} = 1$ for $k \geq x, l \geq x$, meaning that the algorithm is finished. Since there is one zero column in the resulting matrix, rank $\mathsf{Z}_1 = 1$, and $\beta_1 = \text{rank } \mathsf{Z}_1 - \text{rank } \mathsf{B}_1 = 1 - 0 = 1$. This is correct, since K has one 1-dimensional hole.

Now, if we fill the hole by adding the 2-simplex $\{x_1, x_2, x_3\}$ to K, we get a 2-boundary matrix which we can reduce:

$$\begin{array}{c} \\ \{x_1, x_2\} \\ \{x_1, x_3\} \\ \{x_2, x_3\} \end{array} \begin{array}{c} \{x_1, x_2, x_3\} \\ \left[\begin{array}{c} 1 \\ 1 \\ 1 \end{array}\right] \end{array} \xrightarrow[r_2 = r_2 + r_1]{} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \xrightarrow[r_3 = r_3 + r_1]{} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

This means that rank $\mathsf{B}_1 = 1$, such that $\beta_1 = 1 - 1 = 0$, and the hole has disappeared as expected.

Lastly, it should be noted that since the 0-simplices have no proper faces, the boundary of a 0-chain is always 0. Thus, $\mathsf{Z}_0 = \mathsf{C}_0$, and the rank of the 0-cycle group is equal to the number of vertices. Additionally, the boundary group contains information about which vertices are connected by edge paths. This leads to the 0-th homology group encoding the connected components of the complex; each vertex is in the same homology class as the vertices it is connected to. This in turn means that the rank of the 0-th homology group is the number of connected components in the complex, and a 0-dimensional hole can be intuitively explained as being a connected component.

Now that groups and homology have been defined, persistent homology can be defined.

2.3.3 Persistent homology

The core idea of persistent homology is to examine how the homology of a filtration changes as the parameter of its complex increases, as opposed to examining the homology of the complex at one specific parameter value. This makes the analysis less sensitive to noise in the data [10], and makes it easier to approximate its underlying space [4]. In this thesis, the complex will be the Alpha complex, the parameter of which is the radius r of the balls around each point.

The persistence of a hole is calculated as the difference between the value of r when it is "born" (i.e., when it appears), and the value of r when it "dies" (i.e., when it disappears). In this thesis, the persistence information of a filtration will be encoded in a persistence diagram, where each hole is represented as a point. A persistence diagram most often has birth-death axes, which means that each point has coordinates (b, d) corresponding to its hole's birth and death; this means that all points lie in the upper left triangular,

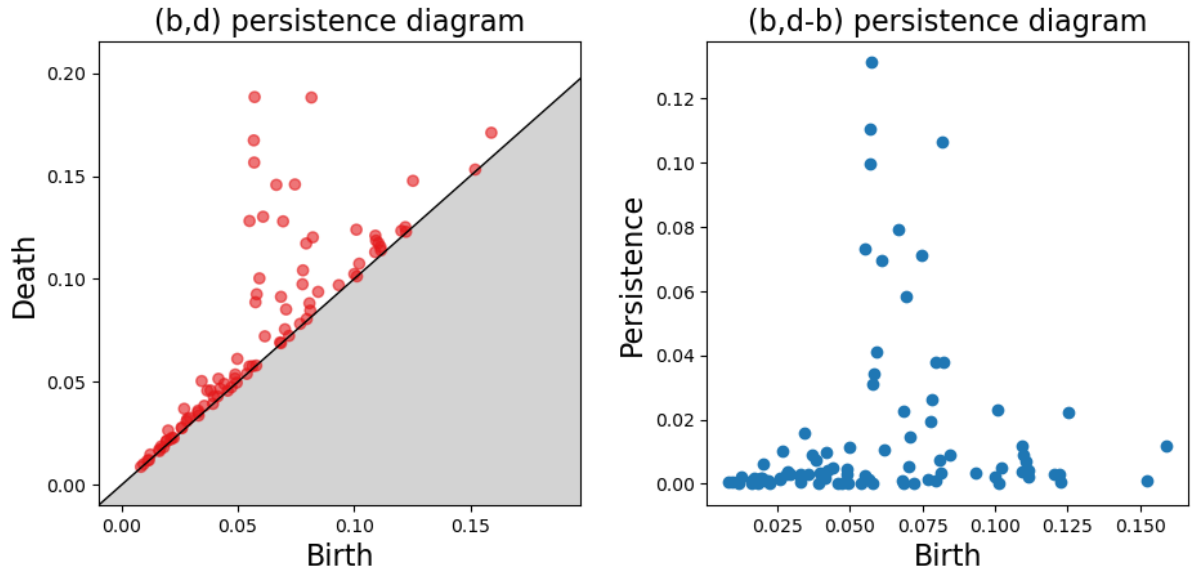

Figure 2.10: Examples of two types of persistence diagrams. The diagram on the left has points with birth-death coordinates, while the diagram on the right has points with birth-persistence coordinates.

as a hole cannot die before it is born. Sometimes, a persistence diagram may have birth-persistence axes instead, where each point has coordinates $(b, d - b)$ corresponding to its hole's birth and persistence. Examples of these two types of diagrams are shown in figure 2.10, both showing the same filtration. Note that if a hole never dies, like the final, single connected component of a complex, its death value is set to ∞. This is plotted in a persistence diagram as a point on a line at the top of the diagram representing infinity.

Similarly to Betti numbers in homology, persistent homology has persistent Betti numbers. The p-th persistent Betti numbers are written as $\beta_p^{i,j}$, and denote the number of p-dimensional holes which are born at $r \leq i$, and whose deaths are at $r > j$. This can be read from a persistence diagram as the number of points to the upper left of the point (i, j), that is, all points with $b \leq i$ and $d > j$. Persistent Betti numbers can be useful when analyzing a filtration, as highly persistent holes in an interval may indicate that the interval is close to a parameter value r approximating the ground truth for the point cloud being analyzed.

To generate the persistence diagram of the p-th homology group of a filtration, we need to know when each p-dimensional hole is born and dies. This may be done by calculating the homology for each complex of the filtration individually. However, there is a more efficient method for doing this, again using a boundary matrix and a matrix reduction algorithm.

First, we need to define an ordering for the simplices of a filtration. For a filtered simplicial complex K, we may assign to it a function f from its simplices to $\mathbb{R}$. The value of the function for each simplex σ will be the lowest value of r for which σ is a part of the complex. We can then index the simplices in order of appearance. For any two simplices $\sigma_i, \sigma_j \in K$, $i < j$ if $f(\sigma_i) < f(\sigma_j)$ or if σ_i is a face of σ_j. In the case where simplices are tied, they may be indexed in any order. This makes it so that $\{\sigma_i \mid i \in [1, k]\}$ is a valid subcomplex of K for any $k \leq n$ [10].

Using the ordered simplices, we can construct a new type of boundary matrix based on the filtration. For a filtered simplicial complex with m simplices, the boundary matrix is an $m \times m$ matrix where each simplex σ_i of the complex is represented by row and column i. In other words, in this boundary matrix, all simplices of all dimensions are in both the rows and columns. Again, like with the previous boundary matrix, the value of each element $e_{i,j}$ is 1 if column j represents a p-simplex and row i represents one of its $(p-1)$-dimensional faces, and 0 if not.

The matrix reduction algorithm for persistent homology is simpler than the one for homology. We first need to define low(j), indicating the highest i (i.e., the visually lowest row) for which $e_{i,j} = 1$; if j is a zero column, low(j) is 0. The algorithm is then:

Input: Boundary matrix D
Output: Reduced boundary matrix R
begin
  for *column* $j = 1$ **to** m **do**
    while *there is a column* $k < j$ *for which* $low(k) = low(j) \neq 0$ **do**
      Add column k to column j;
    end
  end
end

In the reduced matrix, if low(j) $\neq 0$ for a column j representing a $(p+1)$-simplex, that means a p-dimensional hole is born at $f(\sigma_{\text{low}(j)})$ and dies at $f(\sigma_j)$ [18]. We thus have the persistence intervals $[b, d)$ of the filtration, adding an interval $[0, \infty)$ for the final connected component of the filtration.

To demonstrate how this works, we will make the persistence diagram of the point cloud with points $x_1 = (-0.5, 1.5)$, $x_2 = (-1, 0)$, and $x_3 = (1, 0)$. The filtered simplicial complex is then $K = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_1, x_2\}, \{x_2, x_3\}, \{x_1, x_3\}, \{x_1, x_2, x_3\}\}$, with $f(\{x_i\}) = 0$ for each vertex $\{x_i\}$, $f(\{x_1, x_2\}) = \sqrt{0.625}$, $f(\{x_2, x_3\}) = 1$,

$f(\{x_1, x_3\}) = \sqrt{1.125}$, and $f(\{x_1, x_2, x_3\}) = \sqrt{1.25}$. Its boundary matrix is

$$\begin{array}{r} \\ \{x_1\} \\ \{x_2\} \\ \{x_3\} \\ \{x_1, x_2\} \\ \{x_2, x_3\} \\ \{x_1, x_3\} \\ \{x_1, x_2, x_3\} \end{array} \begin{array}{c} \begin{array}{ccccccc} \{x_1\} & \{x_2\} & \{x_3\} & \{x_1, x_2\} & \{x_2, x_3\} & \{x_1, x_3\} & \{x_1, x_2, x_3\} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

which we reduce according to the persistent homology algorithm:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{c_6 = c_6 + c_5} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{c_6 = c_6 + c_4} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This results in the points $(f(\{x_2\}), f(\{x_1, x_2\})) = (0, \sqrt{0.625})$, $(f(\{x_3\}), f(\{x_2, x_3\})) = (0, 1)$, and $(f(\{x_1\}), \infty) = (0, \infty)$ for H_0, and $(f(\{x_1, x_3\}), f(\{x_1, x_2, x_3\})) = (\sqrt{1.125}, \sqrt{1.25})$ for H_1. The resulting persistence diagram is shown in figure 2.11.

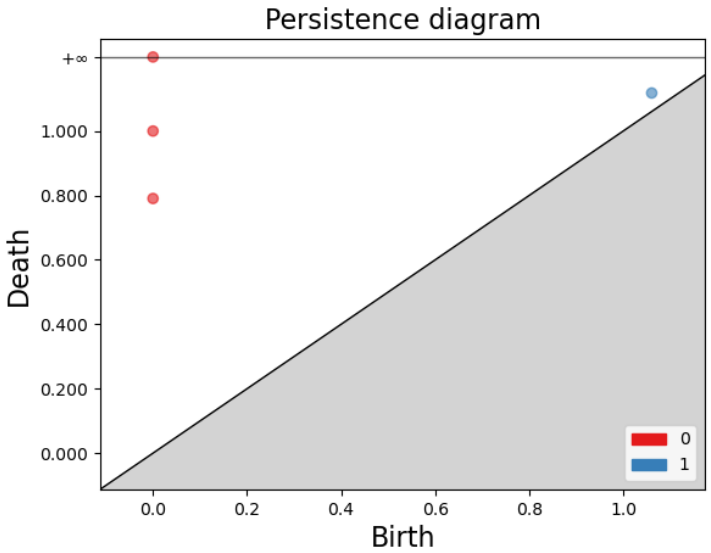

Figure 2.11: The persistence diagram for H_0 and H_1 of the filtration K.

There are more efficient algorithms for computing these steps however, implemented by Python modules such as Gudhi [22] which will used throughout the thesis.

The persistence diagrams are difficult to use directly as input for machine learning. For example, the number of points may vary between diagrams. Therefore, additional processing is done to vectorize the diagrams to a predetermined number of features, as will be detailed in the next section.

2.4 Vectorization

While persistence diagrams are useful for encoding topological information about a dataset, there are methods to vectorize the diagrams to make them more suitable for machine learning. Two different vectorization methods will be described here: Persistence landscapes, and persistence images.

2.4.1 Persistence landscape

The persistence landscape was introduced by Bubenik [2] and provides a way to convert from different discrete topological summaries to a series of continuous functions. The k-th persistence landscape function of a filtration is given by

$$\lambda_k(r) = \sup(m \geq 0 \mid \beta^{r-m,r+m} \geq k),$$

where $\beta^{r-m,r+m}$ is the sum of $\beta_p^{r-m,r+m}$ for all dimensions p. It is also possible to use only the persistent Betti numbers for some dimensions, e.g. to generate the persistence landscape of the first homology group, H_1.

In our case however, we want to convert persistence diagrams to persistence landscapes. For a birth-death persistence diagram with points (b_i, d_i), the paper defines the function of the k-th persistence landscape as

$$\lambda_k(r) = k\text{-th largest value of } \max(\min(r - b_i, d_i - r), 0),\ r \in \mathbb{R}.$$

Here, like with the function defined for filtrations, one may choose to either include the points for all H_p, or only include the points for select H_p.

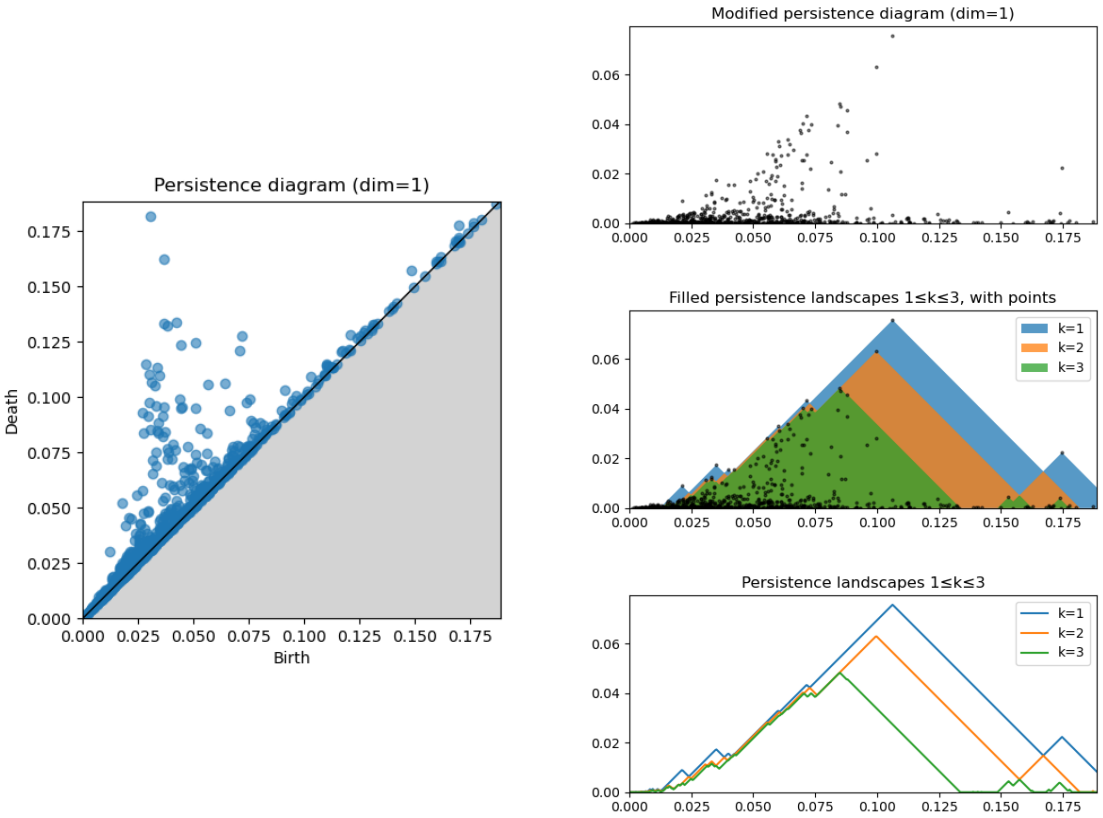

Figure 2.12: The figure on the left shows an H_1 persistence diagram. The top-right figure shows it rotated and scaled. The middle-right figure shows the modified persistence diagram superposed on the filled persistence landscapes. The bottom-right figure shows the final persistence landscapes for $1 \leq k \leq 3$ generated by the original persistence diagram.

These functions can be explained intuitively by slightly modifying the persistence diagram and comparing it to its persistence landscapes. First, rotate the diagram by 45 degrees counter-clockwise, then scale it down by a factor of $\sqrt{2}$. Afterwards, draw a triangle down from each of the points. $\lambda_k(t)$ is then given by the highest point at time step t where at least k triangles overlap. Figure 2.12 demonstrates this process for an H_1 persistence diagram.

Note that the landscape functions λ_k are continuous. To vectorize the functions, a sample range $I \in \mathbb{R}^2$ and a resolution $r \in \mathbb{N}$ are chosen to sample r evenly spaced points of the function in the interval I. This is done for each desired λ_k, usually given by the range $1 \leq k \leq k_{max}$ for a given number of landscapes k_{max}. Those vectors are then concatenated to give the final vectorization.

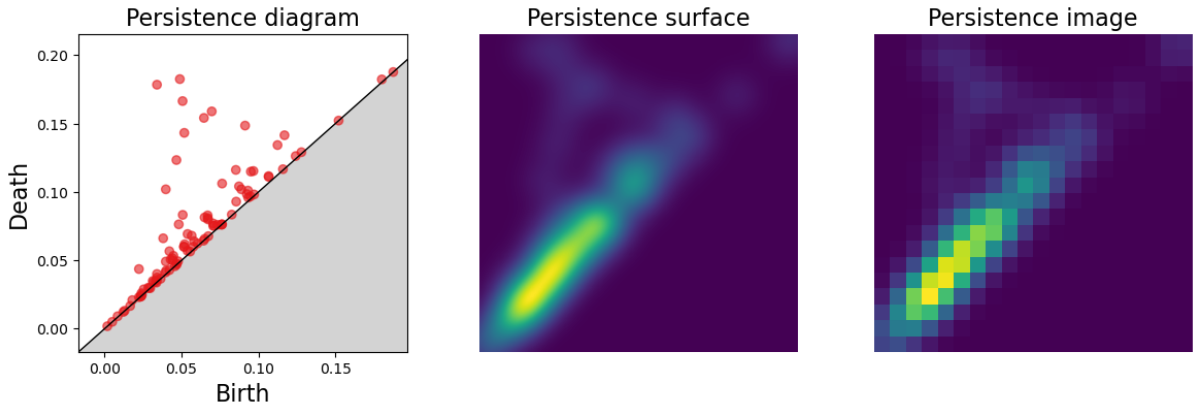

Figure 2.13: Example of how a persistence image is generated from a persistence diagram. A Gaussian is applied around each point in the persistence diagram to create a persistence surface, which is then discretized to make a persistence diagram.

2.4.2 Persistence image

The persistence image, developed by Adams et al. [1], is a very intuitive way of vectorizing a persistence diagram by converting it into an image. First, a persistence surface $\rho(z) : \mathbb{R}^2 \to \mathbb{R}$ is generated by applying a probability distribution function, usually the Gaussian, centered at each point, and performing a weighted sum of the resulting values. This may be thought of as applying a Gaussian blur to the persistence diagram; the variance of the Gaussian should be noted as a parameter of interest. Then, for a given resolution $r \in \mathbb{N} \times \mathbb{N}$ and image range $im\_r \in \mathbb{R}^2 \times \mathbb{R}^2$, the persistence surface is cropped to the image range and discretized to an image with dimension r, such that the value of each pixel is the double integral of the corresponding area of the cropped persistence surface. An example of this process is shown in figure 2.13. To get the final vectorization, the pixel values are concatenated.

Additionally, as noted in the paper, the diagram may be preprocessed to better fit the persistence image method. For example, the lower right triangular of a birth-death diagram is empty, meaning that the persistence image would gain no information from that area. Therefore, diagrams are usually transformed to use birth-persistence axes before generating persistence images.

The weighting function might also be varied by making it constant, i.e. each point has equal weight, or by making it linear such that each point has weight equal to its persistence. The latter could be useful for either focusing on more persistent points, or helping their visibility in diagrams with many points of low persistence.

Chapter 3

Benchmarking tool implementation

In this chapter, I will introduce the benchmarking tool and how it is implemented. Note that this chapter is about the benchmarking tool itself, and not the tasks implemented as part of the benchmarking suite, as those will be detailed in the next chapter.

First, I will explain how to use the benchmarking tool, and give an overview of how it works. Afterwards, I will provide a more detailed explanation of the implementation of the benchmarking tool, along with the reasoning behind the design choices. Then, I will briefly describe the initial version of the benchmarking tool, which was scrapped in favor of making the current version. Finally, I will describe the pipelines which will be used to evaluate the benchmarking suite.

3.1 Overview

3.1.1 Usage

The benchmarking suite is provided as a function *benchmark*, with required parameters being *fscs*, *phs*, and *vecs*. *fscs* and *phs* respectively provide the benchmarking tool with functions to generate filtered simplicial complexes (FSC) and compute persistent homology (PH), while *vecs* provides the benchmarking tool with classes corresponding to vectorization methods. It should also be noted that the function is seeded to make the results reproducible, taking an integer as a seed, defaulting to 42.

The FSC and PH functions must take two parameters each, the first being the input data, and the second being the seed, in case the function involves randomness.

The vectorization classes must have *fit* and *transform* methods for the vectorization methods, attributes *classifier* and *regressor* corresponding to model classes for use with classification and regression tasks, and vectorization parameter ranges and hyperparameter ranges, available through the attributes *vec_parameter_ranges* and *hyperparameter_ranges*. Additionally, its methods and attributes must satisfy certain conditions:

- The initialization method of the class must take the seed as the only parameter.
- The model classes must have two initialization parameters, the first being the hyperparameters and the second being the seed.
- The model classes must have *fit* and *predict* methods similar to the scikit-learn interface.
- The *fit* method of the vectorization class must take two parameters, the first being the input data on which the vectorizer is fitted and the second being the vectorization parameters.
- The *vec_parameter_ranges* and *hyperparameter_ranges* attributes must be dictionaries. Each key must be a tuple where the first element is the name of the parameter, and the second element is one of the strings "discrete" and "continuous", depending on whether the value of the entry is a list with predefined parameter values ("discrete") or a list with the endpoints of a range ("continuous").

The benchmarking tool contains base classes for the vectorization and model classes, which can be extended to make it easier for the user to implement them.

Each of the parameters *fscs*, *phs*, and *vecs* must be a list of dictionaries. For *fscs* and *phs*, each dictionary must have the following entries:

- "fns", which is itself a dictionary, with the keys being the names of the functions and the values being the functions.
- "in", a string stating the input type of the functions in "fns".
- "out", a string stating the output type of the functions in "fns".

For *vecs*, each dictionary must have the following entries:

- "classes", a dictionary with key-value pairs corresponding to the names of the classes and the classes themselves (not instances of the classes).
- "in", a string stating the input type of the vectorization methods in "classes".

The "in" and "out" entries are used to make combinations consisting of an FSC function, a PH function, and list of vectorization classes, where the components are compatible; that is, the output type of the FSC function must match the input type of the PH function, and the output type of the PH function must match the input type of the vectorization classes.

3.1.2 Benchmarking process

The benchmarking suite consists of four tasks. The training, validation, and test datasets for these tasks are generated as needed, based on a handful of parameters. The benchmarking tool tests the TDA pipelines on each task several times, each time with varying parameter combinations. What these tasks are and which parameters are used will be explained in the next chapter. For now, it suffices to know that the tasks and task parameters are pre-determined.

With the prerequisites explained, I can now go through the process of the benchmarking tool, which is also shown in figure 3.1.

The first step in the benchmarking tool is making the FSC, PH, and vectorization combinations as described earlier, because each combination of the FSC function, the PH function, and any of the vectorization classes in the list constitutes a TDA pipeline. Note that FSC function and PH function combinations may be referred to as FSC + PH.

After making the TDA pipelines, the benchmarking tool iterates over each task and its parameter combinations, testing the pipelines whose FSC function has input type corresponding to the task's data type, e.g. "pc" for tasks whose datasets consist of point clouds.

The pipelines are tested by first passing the dataset to the FSC function, then passing its output to the PH function; this section is timed as the FSC + PH time. Afterwards, the tool iterates over the vectorization classes associated with the FSC + PH.

For each vectorization class, vectorization parameter and hyperparameter optimization is done using random search; a maximum of 20 vectorization parameters combinations and 20 hyperparameter combinations are generated based on the provided ranges. The tool then iterates over the vectorization parameter combinations, and for each combination, it vectorizes the output from the PH function and iterates over the hyperparameter combinations.

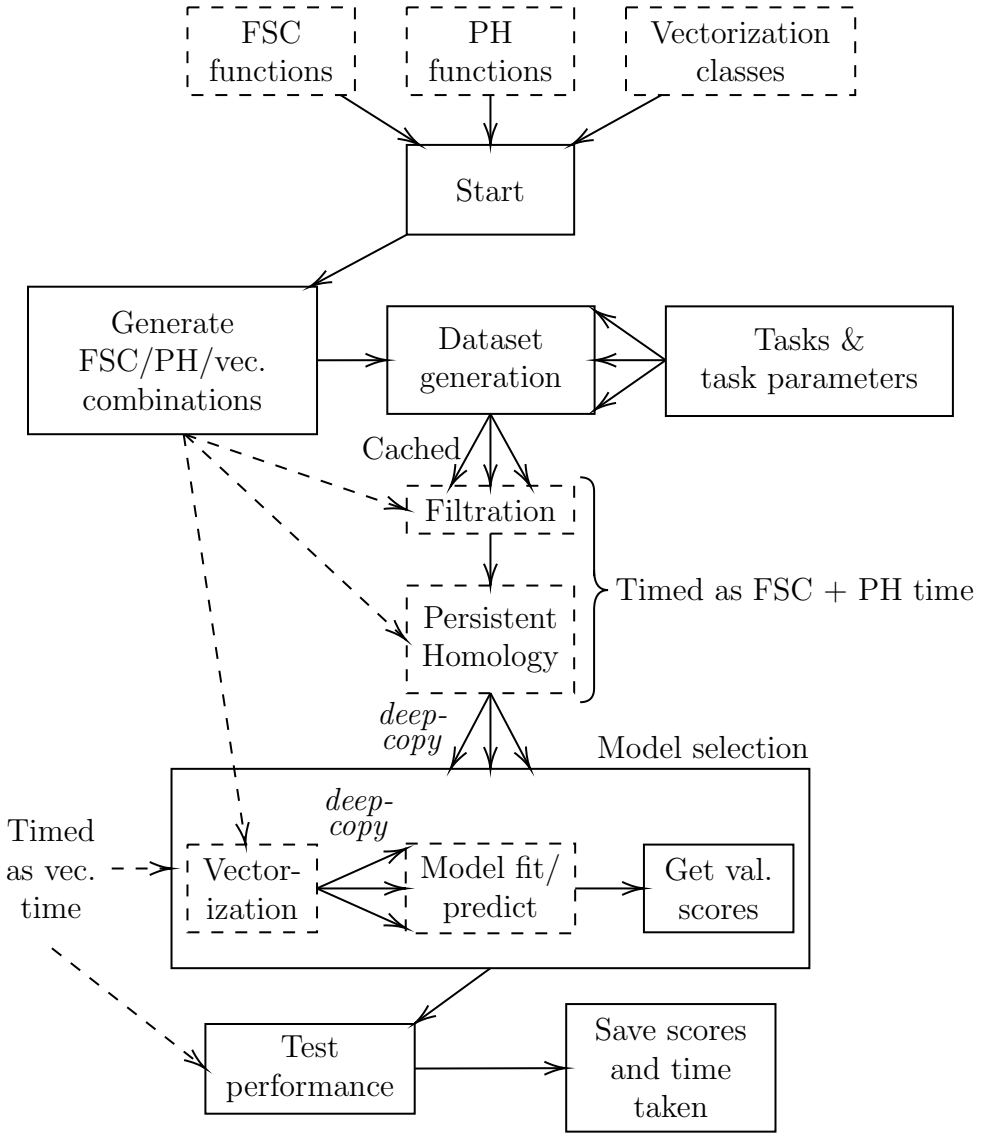

Figure 3.1: Flowchart showing the workflow of the benchmarking tool. The boxes with solid lines indicate that the step is performed by the benchmarking tool, and the boxes with striped lines indicate steps which use user implemented code. *deepcopy* next to an arrow means that data from the preceding step is copied using *deepcopy* before being used in the next step.

For each of the hyperparameter combinations, the tool fits the model corresponding to the task type (i.e., the *classifier* or the *regressor*) on the training data. Afterwards, the trained model is given a validation score based on its predicted values for the validation data, which is stored as the validation score for that combination of vectorization parameters and hyperparameters. When each vectorization parameter and hyperparameter combination has been given a validation score, the combination with the highest score is chosen for testing. One may therefore view this as an extended model selection.

The dataset is vectorized using the selected vectorization parameters, and a model with the selected hyperparameters is trained on the combined training and validation data. Then, the model is given a test score based on its predicted values for the testing data. The time taken for the model selection and testing is timed as the vectorization time.

Finally, the best validation score and the test score for the FSC function, PH function, and vectorization class combination is stored in a *scores.json* file, along the FSC + PH time and the vectorization time. Note that the pipelines are scored by accuracy for classification tasks, and MSE for regression tasks.

The benchmarking tool also caches and copies the data at certain points to speed up the benchmarking process, and to prevent the pipelines from tampering with the data given to other pipelines. When each dataset is generated, it is also cached, allowing the benchmarking tool to quickly reload the dataset for each FSC + PH. Additionally, when data which is intended to be reused is given to user implemented code, it is first copied using *deepcopy*. This most notably happens when the vectorization class uses the output of the PH function, and when the model uses the vectorized data.

3.2 Implementation details

This section will describe the finer details of the benchmarking tool not explored in the previous section, while explaining the reasoning behind the design choices.

First, the random number generator used throughout the benchmarking suite, both in the tool itself and when generating the datasets, is NumPy's RandomState. It is a legacy generator, meaning it will not be updated, and is therefore suited for use in the benchmarking suite to ensure that the results obtained are consistent across multiple versions of NumPy [17].

The perhaps most notable design choice of the benchmarking tool is directly splitting up the pipeline into the FSC, PH and vectorization parts, as opposed to having the user implement whole pipelines. There are multiple advantages to this approach.

One advantage is the possibility of implementing multiple FSC functions to handle different types of data, for example point clouds and images, while still being able to use the same PH functions and vectorization classes. It also allows the user to test the impact of using different simplicial complexes without having to re-implement the PH functions and vectorization classes. Having the user implement the vectorization classes and their model classes in a standardized way also makes the model selection for each pipeline more unified, making it easier to compare the pipelines. Most important for this thesis, however, is the ease with which multiple vectorization methods can be implemented and tested. The same FSC + PH can be in combination with several vectorization classes, such that the user only has to implement the different vectorization classes, and not an entire pipeline for each vectorization method.

Using the same FSC + PH also helps speed up the testing. As mentioned earlier, the FSC, PH, and vectorization combinations are pre-generated such that each FSC + PH is associated with their compatible vectorization classes. In addition to allowing pipelines with the same FSC + PH to reuse its output, as explained in the previous section, it also simplifies checking if all the vectorization classes have been tested for a particular FSC + PH; if they have, the FSC + PH does not need to be run and can safely be skipped, saving time.

The requirements for implementing the FSC functions, PH functions, and vectorization classes were kept somewhat low to allow the pipelines to be sufficiently flexible. One might for example make a non-TDA pipeline to use as a baseline by simply making both the FSC and PH functions the identity function, and making the vectorization class transform the point clouds to the mean and variance of each coordinate.

The tasks have parameters, which are divided into constant parameters and variable parameters. The constant parameters stay the same for each dataset generated for the task, while the variable parameters are combined and iterated over. An example of what a variable parameter might be is the number of points per point cloud, or the amount of noise applied when generating the dataset.

This approach makes it easier to test for the different pipelines' stability for varying amount of noise, and their performance with a varying amount of information available (e.g., the number of points per point cloud), eliminating the need to add each task and

parameter combination individually. The variable parameters might also be applicable to the tasks based on real-world datasets, for example varying the total size of the dataset to check the impact on the total running time. It also makes it easier to save and read the pipelines' scores for each dataset; this will be explained later.

I should also note that the pre-generation of the task and task parameter combinations is used to show the current progress of the benchmarking while it is running.

The choice of dividing the timings of the FSC + PH and the vectorization part was motivated by the fact that the same FSC + PH may be used for different vectorization methods. It therefore makes sense to look at both the FSC + PH time taken, which is shared between them, and the vectorization time, which will be different. This makes it easier to compare the time taken for those different vectorization methods, while still allowing for comparing the total time taken for a TDA pipeline, and the time taken for a baseline non-TDA pipeline.

Having the model selection and vectorization so tightly connected was done for several reasons: First, one might want to vary the complexity of the model based on the vectorization method. Second, it easily allows for combining the vectorization parameter search with the hyperparameter search in a natural way. I should mention here that for simplicity, the benchmarking tool iterates over the same hyperparameter combinations for each vectorization parameter combination, as this is not expected to have a significant impact on the results. Lastly, having the model classes be a part of the vectorization class allows for model-level vectorization methods such as PersLay [5]. The model can then be changed by the vectorization class according to the vectorization parameters, and the benchmarking tool will retrieve the model with the correct parameters.

Including the model selection and testing in the vectorization time was done because they are majorly decided by the vectorization class. The models and hyperparameters used can vary significantly between the different vectorization classes. Additionally, the choice of vectorization method can affect the time taken to train and use the model, depending on the number of features generated by the vectorization.

The choice of having a maximum of 20 vectorization parameter combinations and 20 hyperparameter combinations was done to keep the total running time of the benchmark low for the purposes of this thesis; optimally, these numbers would be set higher to perform more thorough searching.

Using accuracy as the performance measure for classification tasks, and MSE for regression tasks, was done because they are simple and well-known performance measures

for their respective purposes. For the accuracy in particular, it is a reasonable choice because the datasets used by the benchmarking suite have balanced labels.

Lastly, the scores are saved to *scores.json* after each pipeline has finished testing. This allows for continuing the suite if an error occurs, or if the benchmarking is stopped some other way, without having to rerun tests.

The structure of *scores.json* is the same as the iterations: The upper level is the task, followed by a level for each of the variable parameters of the task. The following level contains each of the pipelines tested, in the form "*FSC name*, *PH name*, *vectorization name*". The scores and the times taken are saved under their respective pipelines, making it easy to compare their performance.

3.3 Initial version

The initial version of the benchmarking tool differed from the current version in a few ways.

The largest difference was that it was pipeline-based, instead of having the current FSC, PH, and vectorization split. That meant that the user would have to implement the entire pipeline, from the filtration step to the model selection and test set prediction, for every combination of FSC function, PH function, and vectorization method. When the possibility came up of having tasks with different types of data, for example images and point clouds, I started rewriting the benchmarking tool to the current version so this could be supported.

The initial version of the tool also did not yet have variable parameters for the tasks, meaning that the logic for handling the tasks also had to be rewritten.

Also majorly impacted by the rewrite was the way the scores were saved. Since each task only had one parameter combination, and each pipeline was added in its entirety, the scores were simply saved as each pipeline's test score and time taken for each task, which could very easily be converted to a variety of other formats, like an Excel spreadsheet or a CSV file. This proved unfeasible for the current version, and so JSON was chosen as the format for the score file.

3.4 Pipelines

To evaluate the suitability of the benchmarking suite for testing TDA pipelines, I implemented two non-TDA pipelines, to use as a baseline, and two TDA pipelines, which will be described in this section.

The model classes and hyperparameter ranges used by the pipelines are the same. The classification model and the regression model are the *MLPClassifier* and *MLPRegressor* of scikit-learn respectively. The hyperparameter ranges are 1-5 hidden layers of size 32, sampled discretely, and the exponent of the learning rate is between -4 and 0, sampled continuously and uniformly; the learning rate used is then 10^x for $x \sim \mathcal{U}((-4, 0))$. Additionally, the models use early stopping.

The FSC function and PH function for the non-TDA pipelines is the identity function, as mentioned earlier. One of the non-TDA pipelines concatenates the points of each point cloud as its "vectorization" method, while the other computes the mean and variance of each coordinate for each point cloud. Neither of these have any vectorization parameters.

The TDA pipelines use the same FSC function and PH function. The FSC function uses Gudhi's AlphaComplex class to make a filtration for each point cloud based on the alpha complex. These filtrations are sent to the PH function, which uses Gudhi to compute the persistent homology of each filtration. The PH function outputs the H_1 persistence intervals, which are used as the input for the vectorization.

The first TDA pipeline uses the persistence image as its vectorization method. Its parameter ranges are: 4, 8, 10, 12, and 16 for the resolution of both dimension (always producing a square image), chosen discretely; $[-3, 3]$ for the exponent of the variance of the Gaussian applied to each point, making the variance 2^x for $x \sim \mathcal{U}((-3, 3))$; $[-1, 2]$ for the exponent of the weight function applied to each point, making the weight function $p^{\max(0,x)}$, where p is the persistence of the point and $x \sim \mathcal{U}((-1, 2))$.

The second TDA pipeline uses the persistence landscape as its vectorization method, with the following parameter ranges: 1-5 for the number of landscapes, chosen discretely; 32, 64, 128, 256, and 512 for the resolution, chosen discretely.

Note that because the vectorization methods are only used in one pipeline each, the pipelines will be referred to by their vectorization method.

Chapter 4

Synthetic benchmarks

4.1 Method

This section will explore in detail how the benchmarking tasks are created. In the first section, each of the different point cloud generation methods will be explained, both the theory behind the implementations and how they work in practice. The second section will then describe how the generators were used to make the different benchmarks.

4.1.1 Manifold sampling

Generating point clouds on manifolds is generally a simple task, given a parameterization of the manifold. However, if the parameters are drawn from uniform distributions, the distribution of the points on the surface of the manifold might not be uniform, depending on the curvature of the manifold [8]. The reason this is important is because for the point sampling, the goal is to obtain a uniformly sampled ground truth from which points can either be evenly or, perhaps more interesting, unevenly downsampled.

A simple example of uniform parameter distributions leading to non-uniform point distribution is the 2-sphere. It may be parameterized by two angles, one for the longitude and one for the latitude. For the longitude, if we sample an angle $\theta \in [0, 2\pi)$ from a uniform distribution, we get an evenly distributed circle. When generating an angle $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ for the latitude however, the uniform distribution yields an unevenly sampled sphere. As the latitude goes further from the equator, the radius of the circle gets smaller

while the probability of a point being on that circle stays the same, meaning that points will be more concentrated around the poles. A sphere sampled this way is shown in figure 4.1.

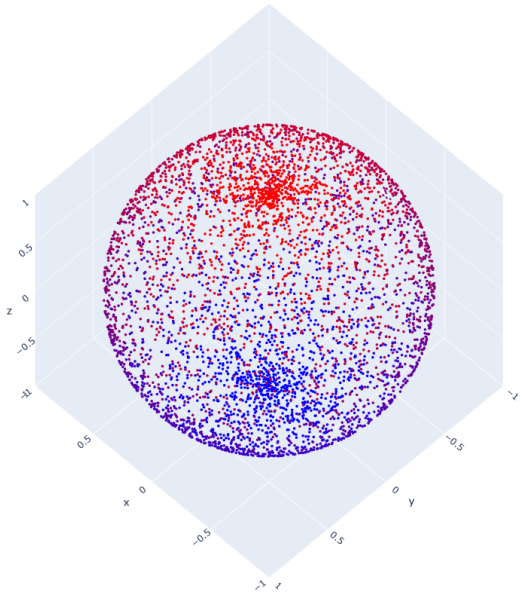

Figure 4.1: 2-sphere generated by uniformly sampling the longitudinal and latitudinal angles for each point. The points are colored by their y-value to show the poles more clearly.

Mathematically, this can be proven by comparing the probabilities and surface areas for two different sections of the sphere. The probability of $0 \leq \phi \leq \frac{\pi}{4}$ and $\frac{\pi}{4} \leq \phi \leq \frac{\pi}{2}$ is trivially the same given a uniform distribution. The surface areas can be calculated by integrating the circles of latitude for the two intervals. For the unit sphere, the radius of a circle of latitude is given by $\cos\phi$, thus the formula for the circumference is:

$$2\pi \cos\phi$$

We then integrate over the two intervals to get their respective surface areas:

$$\begin{aligned}\int_0^{\frac{\pi}{4}} 2\pi\cos\phi\, d\phi &= 2\pi \int_0^{\frac{\pi}{4}} \cos\phi\, d\phi \\ &= 2\pi \left[\sin\phi\right]_0^{\pi/4} \\ &= 2\pi \left(\frac{1}{\sqrt{2}}\right) \\ &= \underline{\underline{\sqrt{2}\pi}}\end{aligned}$$

$$\begin{aligned}\int_{\frac{\pi}{4}}^{\frac{\pi}{2}} 2\pi\cos\phi\, d\phi &= 2\pi \int_{\frac{\pi}{4}}^{\frac{\pi}{2}} \cos\phi\, d\phi \\ &= 2\pi \left[\sin\phi\right]_{\pi/4}^{\pi/2} \\ &= 2\pi \left(1 - \frac{1}{\sqrt{2}}\right) \\ &= \underline{\underline{(2-\sqrt{2})\pi}}\end{aligned}$$

Since $\sqrt{2}\pi \neq (2-\sqrt{2})\pi$, and the probability of a point being on either section is the same, their sampling densities must differ.

n-sphere

There are convenient parameterizations for n-spheres which use n angles; for example $(\cos\theta\sin\phi, \sin\theta\sin\phi, \cos\phi)$ with $\theta \in [0, 2\pi)$ and $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ generates a unit 2-sphere. The problem is that, as demonstrated earlier, this results in an uneven sphere (for $n > 1$). A much simpler, but perhaps less intuitive way to generate a sphere is to simply draw each coordinate from a probability distribution and scale the resulting vector to the desired length. Using the uniform distribution $\mathcal{U}((-1, 1))$ gives a sphere like the one shown in figure 4.2. Using the normal distribution, however, results in an evenly sampled sphere (figure 4.3) because of the rotational symmetry of the normal distribution [16].

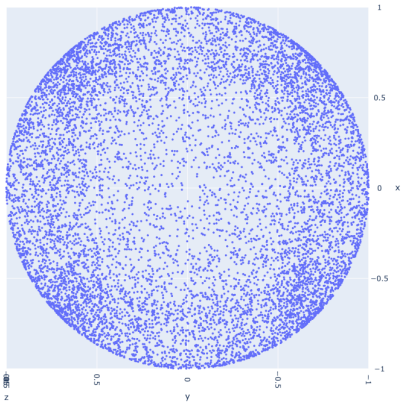

Figure 4.2: Top-down view of a 2-sphere generated by normalizing points with coordinates sampled from $\mathcal{U}((-1, 1))$.

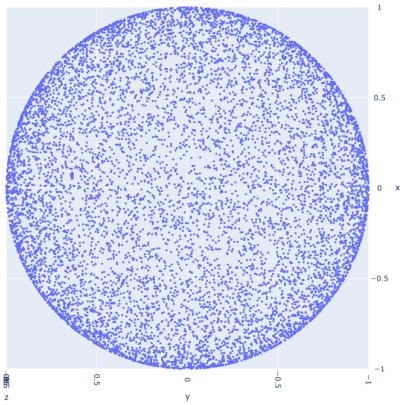

Figure 4.3: Top-down view of a 2-sphere generated by normalizing points sampled from $\mathcal{N}(\vec{0}, I)$.

My implementation of the n-sphere generator defaults to a standard normal distribution (mean $\vec{0}$, covariance I), but a custom multivariate generator function may be used to get other coordinate distributions on the sphere. The examples below show how the sphere looks for multivariate normal distributions with non-standard mean (figure 4.4), and non-standard covariance matrix (figure 4.5).

Point sampling with noise is also supported in my implementation, which varies the distances of the points from the center. The formula used for the final distance r_f of each

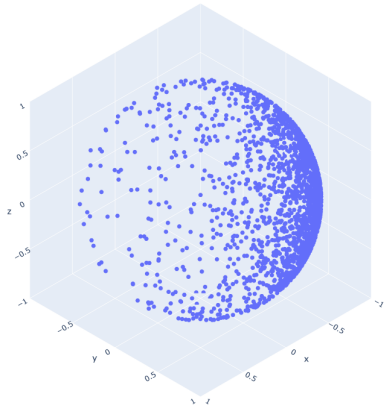

Figure 4.4: 2-sphere with points sampled from $\mathcal{N}((-1, 1, 0), I)$, which were then normalized.

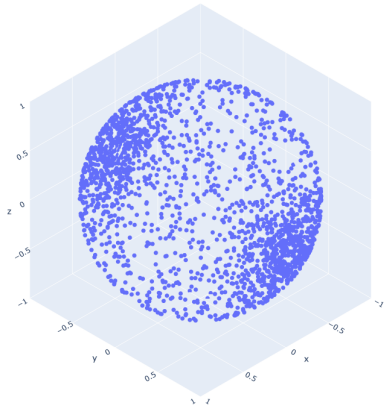

Figure 4.5: 2-sphere with points sampled from $\mathcal{N}(\vec{0}, \text{diag}(1, 10, 1))$, which were then normalized.

point is $r_f = r\,(1 + \alpha x)$, where $x \sim D$ for a probability distribution D, with the default being the standard normal distribution, and $\alpha \in \mathbb{R}$ is a given *noise_scale* parameter; note that setting $\alpha = 0.0$ creates a sphere without noise.

Before settling on the final noise function, a few others were considered. One example is $r_f = r \cdot 2^{\alpha x}$ which forces a positive magnitude, but as it is asymptotic to 0 in the negative direction, its resulting points tended to be concentrated towards the origin. Another method tested was to multiply each norm by αx before scaling the point, but this proved to be very unpredictable; for points with lower norms especially, the probability was relatively high of a point's final distance being an order of magnitude larger than r.

Power spherical

The power spherical [7] is an alternative to the standard n-sphere point sampling. It takes a direction vector and a concentration scalar to generate a point cloud concentrated in that direction on the sphere. Figures 4.6 below show the power spherical for different concentrations, with the same direction. Note that a concentration of zero is a special case, giving a uniform sphere.

To explain how the power spherical works, I will go through the algorithm step by step, generating an n-sphere with m points, direction $\vec{\mu} \in \mathbb{R}^{n+1}$, and concentration $\kappa \in \mathbb{R}$. First, we uniformly sample m points from an $(n-1)$-sphere, denoting the collection of points S, and draw m samples from the beta distribution $\beta\left(\frac{n}{2} + \kappa, \frac{n}{2}\right)$ which are

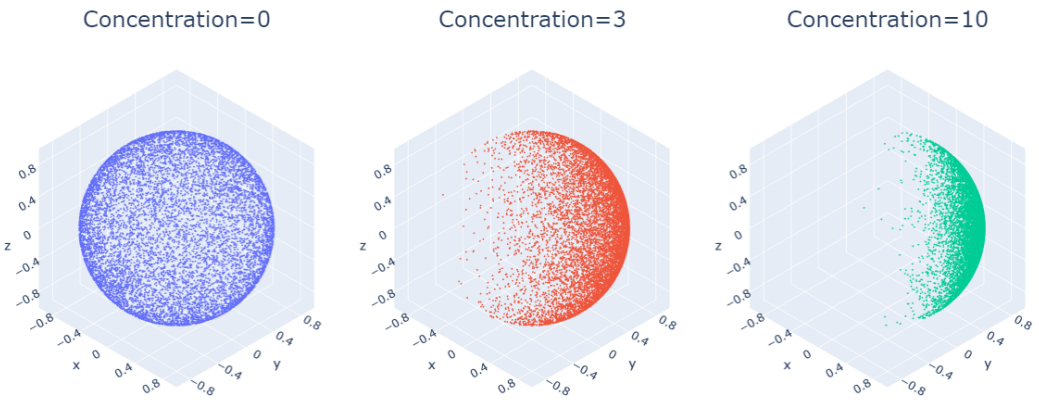

Figure 4.6: The power spherical at various concentration values, with the same direction.

concatenated to make $\vec{b}$. This beta distribution is then transformed to lie in the interval $[-1, 1]$ using $\vec{t} = 2\vec{b} - 1$. Then, each point s_i in S is scaled by $\sqrt{1 - t_i^2}$, resulting in S'. The n-sphere itself, Y, is made by concatenating $\vec{t}$ and S', effectively distributing the points of S across the n-sphere with the correct concentration.

At this point, the sphere itself is complete, but the concentration is in the "default" direction $\vec{e}_1 = [1, 0, ..., 0]^\top$, so we need to move it to $\vec{\mu}$. The way this is done in the paper is by reflecting the sphere about a hyperplane going through the origin.

$$\vec{u} = \frac{\vec{e}_1 - \vec{\mu}}{||\vec{e}_1 - \vec{\mu}||_2}$$

defines a unit vector which is normal to the plane of reflection. Assuming Y is a $d \times m$ matrix, the final sphere (with correct concentration and direction) is given by $X = (I_d - 2\vec{u}\vec{u}^\top)Y$.

Noisy point sampling is also possible for the power spherical. Since the sphere is generated with a radius of 1, each point can be scaled at the end by $r_f = r\,(1 + \alpha x)$, $x \sim D$ for some probability distribution D, defaulting to the standard normal distribution, to get the wanted noise level α (and radius).

Torus

A torus is commonly embedded in $\mathbb{R}^3$ as the surface of revolution of an off-center circle. This means that it can be parameterization with two angles, one for the axis of revolution,

and one for the off-center circle. The torus parameterization also relies on two radii. One is the major radius R, which is the distance of the circle's translation. The other is the minor radius r, which is the radius of the revolved circle itself.

Generating a torus may be done using the two angles, $\theta \in [0, 2\pi)$ for the revolved circle and $\phi \in [0, 2\pi)$ for the revolution, using the parameterization

$$\begin{pmatrix} (R + (r\cos\theta))\cos\phi \\ (R + (r\cos\theta))\sin\phi \\ r\sin(\theta) \end{pmatrix}.$$

Sampling the angles from uniform distributions results in a non-uniformly sampled torus, shown in figure 4.7. There is no simple trick to uniformly sample from a torus, but it is possible by rejection sampling θ (recall the description of rejection sampling in section 2.1.2). While this wastes a lot of computation, it does result in a uniform torus.

A paper by Diaconis et al. [8] describes the process. They found that the probability density function of the target distribution from which θ should be sampled is

$$p(\theta) = \frac{1}{2\pi}(1 + \frac{r}{R}\cos\theta),\ 0 \leq \theta < 2\pi.$$

Requiring r and R to satisfy $0 < r \leq R$, $p(\theta)$ has a maximum value of $1/\pi$, occurring when $r = R$ and $\theta = 0$. The probability density function of $\mathcal{U}((0, 2\pi))$ is $\pi(\theta) = 1/2\pi$, and we can set $M = 2$ such that $2\pi(\theta) \geq p(\theta)$. This means that we can use rejection sampling to sample from $p(\theta)$: Sample $x' \sim \mathcal{U}((0, 2\pi))$ and $y' \sim \mathcal{U}((0, 1/\pi))$. If $y' \leq p(x')$, accept x'.

The paper also included an implementation in R for the rejection sampling, which I based my Python implementation of the torus point sampler on.

Similarly to the previously described sphere samplers, the torus sampler also supports noisy sampling. For the toruses, noise is applied to the minor radius r to shift the point along its surface normal. The final minor radius used is calculated using $r_f = r\,(1 + (\alpha x))$, where $\alpha \in \mathbb{R}$ is the given *noise_scale* and $x \sim D$ for a probability distribution D, defaulting to the standard normal distribution. As with the spheres, this formula allows both for predictable scaling of the noise, and for non-noisy toruses at $\alpha = 0$.

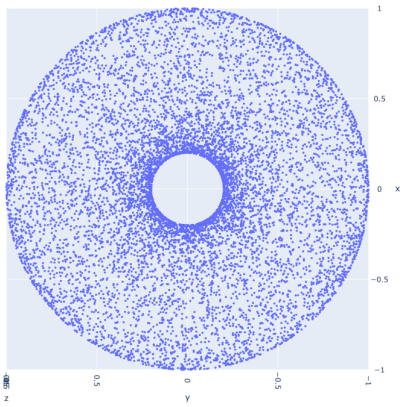

Figure 4.7: Top-down view of a non-uniformly sampled torus generated using uniformly sampled angles.

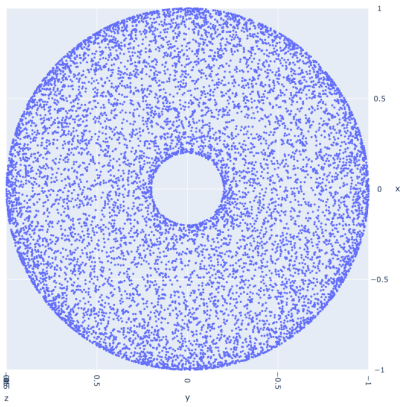

Figure 4.8: Top-down view of a uniformly sampled torus. It was generated with uniformly sampled angles of revolution and rejection sampled angles for the revolved circle.

Genus g torus

While the torus itself has genus 1, for $g \geq 1$ it is generalizable to any genus g torus by simply removing adjacent "caps" of g toruses and gluing them together. Figure 4.9 shows toruses of genus 2 and 3.

The size of the cap is determined by a cut-off value in the range $[-1, 1]$, corresponding to the cosine of the revolved circle. That is, -1 removes almost the entire end of the torus, 1 keeps almost the entire end of the torus, and 0 (the default) cuts off at the middle of the end piece. Genus 2 toruses with cut-offs -0.5, 0.0, and 0.5 are shown in figure 4.10.

The implementation for the genus g torus uses a modified version of the normal torus generator. First, the torus to generate a point for is chosen. Then a two-stage rejection sampling is done, consisting of: 1) the normal rejection sampling; 2) checking that the point is not a part of one of the removed caps. If the point is accepted, noise is applied similarly to the normal torus, and the point is moved to the correct torus. If a point is rejected, the torus for which to generate a point is again chosen at random. This ensures that each of the toruses has the correct point density, which is important to take into account as their surface areas are different.

Like the standard torus sampler, the noise is implemented such that the final minor radius of each point is given by $r_f = r\,(1 + \alpha x)$, $x \sim D$, i.e. the point is moved along the surface normal.

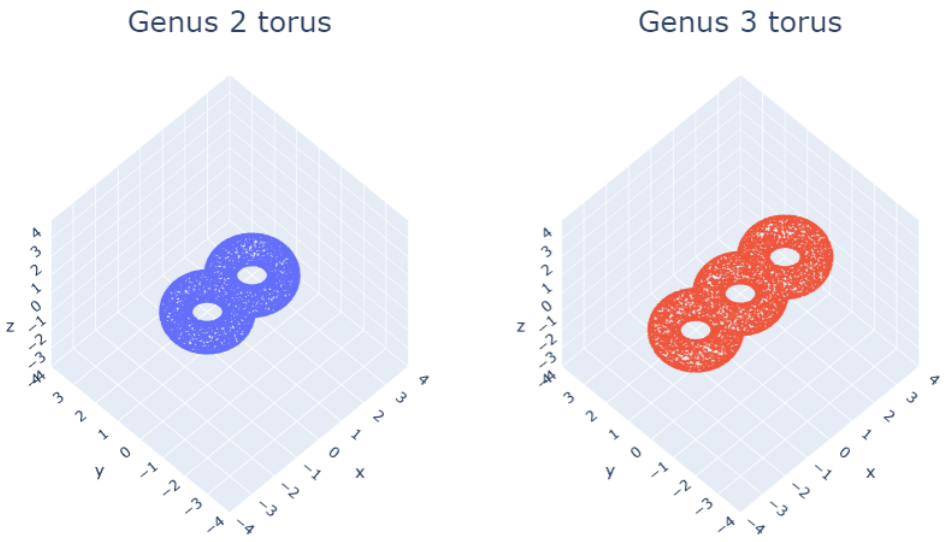

Figure 4.9: Toruses with genus 2 and 3.

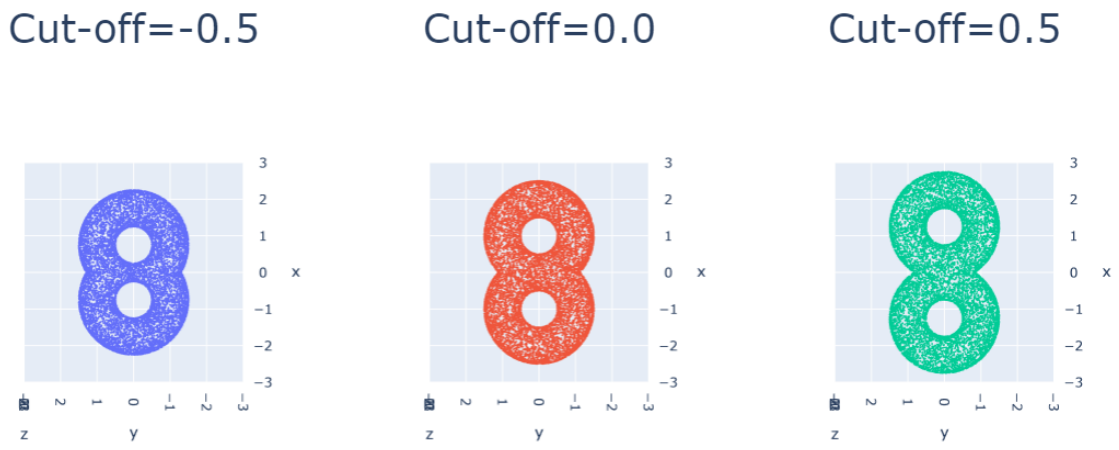

Figure 4.10: Genus 2 toruses with cut-offs -0.5, 0.0, and 0.5.

4.1.2 Creating benchmarks

The different manifold sampling methods detailed in the previous section are used to generate different synthetic benchmarks, each constructed to test different aspects of the TDA-based pipelines compared to the baselines. This section will explain in detail each of the synthetic benchmarks.

First, I should note the parameters which can be varied when generating the datasets: *n_clouds*, the number of point clouds, as well as *n_points*, the number of points per point cloud. The split ratios of the dataset can also be varied with the *split_ratios* parameter, and must be a tuple of three numbers summing to 1. The three numbers represent the ratios of the dataset to be used for the training set, the validation set, and the test set respectively. Lastly, the noise scale(s) used when generating the dataset can be varied with the *noise_scales* parameter. If it is a number, it is used as the noise scale for the training, validation, and test sets. If it is a tuple of two numbers, the first number will be used as the noise scale for the training and validation sets, while the second number will be used for the test set.

Most of these parameters will be shared between the different tasks: 1000 point clouds are generated for each dataset, with *split_ratios*=(0.6, 0.2, 0.2). Also, each task iterates over the same set of noise scales:

- 0.0, to test with no noise.
- 0.1, to test how the pipelines handle a small amount of noise.
- (0.1, 0.2), to test how the pipelines handles a higher amount of noise in the test data than in the training data.
- 0.2, to test how the pipelines handle a larger amount of noise.
- (0.2, 0.4), to test how the pipelines handle a much higher amount of noise in the test data than in the training data.

Lastly, each individual point cloud will be scaled to have a mean of 0 and a standard deviation of 1 for all coordinates; the unscaled point clouds diverge significantly enough in these measures that during the initial testing of the benchmarking suite, the mean/-variance pipeline outperformed the other pipelines on almost all datasets. The scaling is therefore applied to remove this information, making the suite more relevant to TDA.

Sphere/torus classification

The first benchmarking task is a simple sphere/torus binary classification task. It is intended as a very simple baseline task where TDA-based pipelines are expected to have an accuracy close to 1, given sufficiently many points and low noise.

For this task, datasets are generated with 50, 100, and 250 points per point cloud; in combination with the noise scales, this results in 15 different datasets for the task.

For each point cloud, the dataset generator switches between making a 2-sphere and a torus, resulting in an even number of spheres and toruses. Additionally, the actual noise scale n' for each point cloud is varied such that $n' = n|x|$ where $x \sim \mathcal{N}(0, 1)$ and n is the base noise scale for the entire dataset. The goal of this random noise scale is to get more varied data. This method of varying the actual noise scale is also used when generating datasets for the other tasks.

Sphere/genus g torus binary classification

The sphere/genus g torus binary classification task is made similarly to the regular sphere/torus classification task, alternating between generating 2-spheres and toruses. When it generates a torus, however, it also randomly selects a genus $1 \leq g \leq 5$ for the torus. The task is still a binary classification task with equal probability of either class, but having a random genus for the torus increases the complexity of the task as there are several different objects corresponding to one class.

This task uses a higher number of points per point cloud to be able to more accurately represent the toruses of higher genus. The different numbers of points per cloud iterated over are 250, 500, and 1000; combined with the noise scales, there are 15 different datasets for this task as well.

It should also be noted that the cut-off for the toruses is chosen at random as $\tanh(x),\ x \sim \mathcal{N}(0, 0.2)$, providing more variation to the data.

Sphere/genus g torus regression

In the sphere/genus g torus regression task, the genus of the manifold from which the point cloud is sampled is chosen entirely at random using a uniform distribution. In contrast to the sphere/genus g torus binary classification task, the task here is to determine the genus of the surface from which the point cloud is sampled, making it more complex than both of the previous tasks. Note, however, that the method for choosing the cut-off for the toruses is the same.

The parameters iterated over for the task are the same as its corresponding classification task, again resulting in 15 datasets in total for the task.

Power spherical concentration regression

For this task, each point cloud is sampled based on the 3-dimensional power spherical, with varying concentration and direction. The goal is to estimate the concentration of each point cloud. The direction will be sampled uniformly from a 2-sphere, while the concentration is sampled as the absolute value of $x \sim \mathcal{N}(0, 2)$.

Datasets are generated with 50, 100, and 250 points per point cloud, which combined with the noise scales again results in 15 datasets.

4.2 Results

The pipelines described in section 3.4 were tested on the tasks detailed in the previous section. This section will present the results for each of the tasks. The full results can be found in appendix A.

4.2.1 Sphere/torus classification

This section will present the results for the sphere/torus classification task. The full results can be found in table A.1.

The non-TDA pipelines had low accuracies: The validation and test accuracies of the concatenated points pipeline was around 0.5 on all datasets with some variance, though its

accuracies were only in the range 0.425-0.58. The mean/variance pipeline had validation and test accuracies of 0.5 on all datasets.

Both TDA pipelines outperformed the non-TDA pipelines on all datasets. The accuracies of the persistence image pipeline were mostly in the range 0.7-1.0; for the datasets with $n\_points = 100$ and $noise\_scales$ of 0.1 and (0.1, 0.2), it had a validation accuracy of 0.98 and test accuracies of 0.63 and 0.575 respectively. The persistence landscape pipeline was generally outperformed by the persistence image pipeline, with accuracies in the range 0.625-1.0. Its accuracies did, however, decrease less than the persistence image pipeline's accuracies as the noise increased, eventually outperforming it on the datasets with $n\_points = 250$ and $noise\_scales$ 0.2 and (0.2, 0.4). Both of the pipelines achieved a test accuracy of 1.0 for $n\_points = 250$, $noise\_scales = 0.0$.

The time taken by the non-TDA pipelines was much lower on all datasets than the time taken by the TDA pipelines.

The FSC + PH times of the non-TDA pipelines were negligible, while the FSC + PH times of the TDA pipelines were around 3 seconds for the datasets with $n\_points = 50$, 7 seconds for those with $n\_points = 100$, and 22 seconds for those with $n\_points = 250$.

The concatenated points pipeline had a vectorization time of around 0.6 seconds for the datasets with n_points 50 and 100, increasing to about 1 second for the datasets with $n\_points = 250$. The mean/variance pipeline had a vectorization time of approximately 0.4 seconds on all datasets. The vectorization time of the persistence image pipeline was around 15, 16, and 21 seconds for the datasets with n_points 50, 100, and 250 respectively. For the persistence landscape, the vectorization time was about 24 seconds on almost all datasets, with a few taking close to 20 seconds and 27 seconds.

4.2.2 Sphere/genus g torus binary classification

This section will present the results for the sphere/genus g torus binary classification task. The full results can be found in table A.2.

The non-TDA pipelines' performance on this task was very similar to their performance on the previous task, with the accuracies of the concatenated points pipeline being around 0.5, and the mean/variance pipeline's accuracies being 0.5 on all datasets.

The overall performance of the TDA pipelines was also comparable to their performance on the previous task, with their average accuracies being slightly higher. It should be noted, however, that the accuracies of the persistence image pipeline decreased less with increasing *noise_scales* than the persistence landscape pipeline's accuracies. The persistence landscape pipeline only outperformed the persistence image pipeline on the dataset with $n\_points = 1000$ and $noise\_scales = 0.0$, where it had validation and test accuracies of 1.0, compared to 1.0 and 0.995 respectively for the persistence image pipeline.

The FSC + PH times for the non-TDA pipelines were again close to 0, with a maximum of around 0.04 seconds. The TDA pipelines' FSC + PH times were much higher, taking around 22, 50, and 105 seconds for the datasets with *n_points* 250, 500, and 1000 respectively.

The vectorization times of the non-TDA pipelines also stayed quite low, with the concatenated points pipeline taking less than 3 seconds on any dataset, and the mean/-variance pipeline never taking more than 0.5 seconds.

The TDA pipelines' vectorization times were quite high. The persistence image pipeline's times increased from around 22 seconds on the datasets with $n\_points = 250$ to about 78 seconds on the datasets with $n\_points = 1000$ and *noise_scales* 0.2 and (0.2, 0.4). For the persistence landscape pipeline, the increase was not as significant, from about 29 seconds to 55 seconds.

4.2.3 Sphere/genus g torus genus regression

This section will present the results for the sphere/genus g torus genus regression task. The full results can be found in table A.3.

The MSEs of the non-TDA pipelines were quite high, with the concatenated points pipeline having MSEs ranging from around 2.5 to 26.6, and the mean/variance pipeline having much less variance, from about 2.5 to 3.33.

Again, the TDA pipelines performed better than baseline, although in some cases the difference was small; for example, on the dataset with $n\_points = 250$ and $noise\_scales = (0.2, 0.4)$, the non-TDA pipelines had test MSEs of about 2.94, while the persistence image and persistence landscape pipelines had about 2.2 and 2.67 respectively. In other cases, however, they performed very well, like on the dataset with $n\_points = 1000$ and $noise\_scales = 0.0$, where the persistence image pipeline had a test MSE of about 0.11,

and the persistence landscape pipeline test MSE was around 0.16. The persistence image pipeline generally performed better than the persistence landscape pipeline.

The FSC + PH times for each of the pipelines for *n_points* 250, 500, and 1000 were similar to those in the previous task.

The vectorization times of each pipeline were much higher for this regression task than for the previous tasks. For the concatenated points pipeline, the vectorization times varied from around 2.5 seconds on the datasets with $n\_points = 250$ to about 8.5 seconds on the datasets with $n\_points = 1000$. The mean/variance pipeline had vectorization times of about 1.5 seconds on all datasets. The vectorization times of the persistence image pipeline ranged from about 47 seconds on some of the datasets with $n\_points = 250$ to almost 98 seconds on datasets with $n\_points = 1000$; for the persistence landscape pipeline, the vectorization times varied from about 80 seconds at best to almost 146 seconds at worst.

4.2.4 Power spherical concentration regression

This section will present the results for the power spherical concentration regression task. The full results can be found in table A.4.

The test MSEs of the non-TDA pipelines were around 1.57, 1.56, and 1.24 for *n_points* 50, 100, and 250 respectively, with the concatenated points pipeline having instances of test MSEs of about 2.0 and 2.5. The TDA pipelines generally performed much better, with test MSEs ranging from about 0.54 to 1.21 for $n\_points = 50$, 0.51 to 1.08 for $n\_points = 100$, and 0.38 to 0.93 for $n\_points = 250$; note that this excludes the persistence landscape pipeline's MSEs on the datasets with $n\_points = 100$ and *noise_scales* 0.1 and (0.1, 0.2), where it had validation MSEs of approximately 0.51, and test MSEs of about 1.53.

In general, the persistence image pipeline performed better than the persistence landscape pipeline.

The FSC + PH times for this task were about the same for each pipeline and each of the *n_points* as those for the sphere/torus classification task.

For the concatenated points pipeline, the vectorization times were about 1.3 seconds, 1.7 seconds, and 2.7 seconds for *n_points* 50, 100, and 250. The vectorization times of the mean/variance pipeline were about 1.4 seconds for all datasets.

The TDA pipelines again had much longer vectorization times than the non-TDA pipelines. For the persistence image pipeline, they were around 55 seconds for each dataset. The vectorization times of the persistence landscape pipeline varied more: about 50 seconds, 60 seconds and 73 seconds for *n_points* 50, 100, and 250.

Chapter 5

Discussion

This chapter will briefly discuss the results obtained testing the different pipelines using the benchmarking suite, before discussing the benchmarking suite and the results in light of the objectives of this thesis. The code repository containing the benchmarking suite and the files used to run the test can be found in appendix B.

The TDA pipelines were found to perform better than the non-TDA pipelines for all tasks and task parameters in the benchmarking suite. The non-TDA pipelines were unable to obtain information from the data. For the concatenated points pipeline, the issue is likely that the point cloud is invariant to the order of the points, while model used assumes that the order of the features matters; for the mean/variance pipeline, the issue is clearly that each point cloud has a mean of 0 and a variance of 1 in all dimensions. The TDA pipelines, however, were able to use persistent homology to gather information about the shapes of the point clouds, making it possible for them to perform well; there were a few cases of the TDA pipelines having high validation performance and low test performance, indicating overfitting. It is also worth noting that the TDA pipelines often performed well on the datasets with high *noise_scales*.

The amount of time taken for the TDA pipelines is a clear downside compared to the non-TDA pipelines tested. However, it is a reasonable trade-off given their relative performance.

Comparing the performances and time measured for each of the TDA pipelines, it seems that the persistence image is a better vectorization method overall than the persistence landscape. The vectorization times for the persistence image pipeline was in most cases lower than those of the persistence landscape pipeline, and it also tended to have better validation and test performance.

5.1 Thesis objectives

The first part of the thesis' objectives was to make the benchmarking tool flexible and easy to use, which I think the benchmarking tool presented is.

The way the FSC, PH, and vectorization split is implemented allows the user to test a wide variety of TDA pipelines, with implicit support for different machine learning frameworks, like scikit-learn and PyTorch. The benchmarking tool also allows for testing different compatible FSC function, PH function, and vectorization class combinations, in case the user wants to test the impact of using different types of simplicial complexes. It is also flexible enough to allow for the implementation of non-TDA pipelines to use as a baseline.

The standardized format required for the inputs of the *benchmark* function helps the user define which components are compatible in an intuitive way; each component in the pipeline with a certain input type is compatible with the components of the previous step with matching output type. This, in addition to the base classes available for the model classes and vectorization classes, makes it quicker to implement different pipelines. The fact that the tool performs things such as training/validation/test dataset splitting and model selection also reduces the amount of code the user has to write.

The second part of the objectives was related to the tasks to be included in the benchmarking suite. One objective was to include synthetic benchmarks for both classification and regression tasks, based on distinguishing topological features. This was achieved with the sphere/torus classification task, the sphere/genus g torus binary classification task, and the sphere/genus g torus genus regression task, which are generated as needed with support for a variety of parameters. The TDA pipelines outperformed the non-TDA pipelines used as the baseline; however, it should be noted that only two non-TDA methods were tested, and there might be other non-TDA methods with the potential to outperform the TDA pipelines on the included tasks.

Another objective was to include real-world datasets in the benchmarking suite. In the end, there was not enough time left to properly search for suitable real-world datasets and include them, partly due to the shift in development of the benchmarking tool from the initial version to the final version. The next chapter will, however, include a discussion about what should be considered when choosing real-world datasets, based on the results obtained from the synthetic benchmarks.

Chapter 6

Future work

The benchmarking suite has a lot of potential for future development, which will be explored in this chapter.

First, and perhaps most important, is the inclusion of real-world datasets in the benchmarking suite. While there was not enough time to properly include any, the results of the synthetic benchmarks help us set a few requirements for potential real-world datasets.

For the sphere/genus g torus genus regression task, the TDA pipelines took around two minutes for 1000 point clouds with 1000 points per point cloud. Based on those times, it is reasonable to limit the potential tasks based on real-world datasets to about 1000 point clouds and 1000 points per point cloud, at least for regression tasks with point clouds as data; having larger datasets could make the benchmarking take too long, especially considering the possibility of a higher number of TDA pipelines, or a higher maximum amount of vectorization parameter and hyperparameter combinations.

It would also be useful to focus on real-world datasets with a probable or proven correlation between the labels and the underlying shape of the data; more specifically, the search for real-world datasets should start with those for which TDA-based approaches have already proven to be effective.

The total running time required for the benchmarking suite could be reduced by combining the datasets with the same noise scale for the training and validation sets, for example those with *noise_scales* 0.1 and (0.1, 0.2); assuming that the rest of the parameters are the same, those datasets will have the same model selection, meaning

that the model selection could be performed once per pipeline, and the model could be tested on test sets with different amounts of noise. This would, however, require a significant change in the benchmarking tool, and perhaps also the dataset generators.

The benchmarking tool could also provide the user with a couple of options for performing more thorough benchmarking. As mentioned earlier, it could allow the user to do benchmarking with a higher maximum number of vectorization parameter and hyperparameter combinations. The tool currently only supports random search for the (hyper)parameter optimization; however, it could be useful to have a *no_max* parameter to allow for grid search, given that all parameter ranges are discrete. Having an *n_runs* parameter could also be of interest to the user, to perform several runs of the benchmarking suite with a different seed each time. The result could then be the average or the best of the runs.

While the current suite only uses 3-dimensional point clouds, it would be interesting and useful to have synthetic benchmarks with manifolds of higher dimensions, to see how the time taken changes with higher dimensions. Having tasks with manifolds in a variety of dimensions embedded in the same space, with the task being to find the dimension of the underlying space, could be useful for testing whether the TDA pipelines are able to find lower-dimensional features in high-dimensional data.

The TDA pipelines generally performed well on the power spherical concentration regression task, despite it not being directly related to the topological features of the underlying space; it would therefore be interesting to look at tasks using different distributions on manifolds, similar to the power spherical.

The benchmarking tool is designed to support testing on different types of data, but the suite currently only contains data in the form of point clouds. It would therefore be useful to add datasets with other types of data, for example images, to see how well TDA-based approaches handle different kinds of data.

Lastly, the benchmarking tool should be updated to support different options for how the *scores.json* file should be structured. The current structure makes it easy to compare the performances of the different pipelines on each of the datasets. However, it could also be practical for the user to have the results grouped by the pipelines instead of the tasks. This could potentially make it easier to see how a new vectorization method performs across different datasets.

Bibliography

[1] Henry Adams, Sofya Chepushtanova, Tegan Emerson, Eric Hanson, Michael Kirby, Francis Motta, Rachel Neville, Chris Peterson, Patrick Shipman, and Lori Ziegelmeier. Persistence images: A stable vector representation of persistent homology. *Journal of Machine Learning Research 18 (2017), Number 8, 1-35*, July 2015. doi: 10.48550/ARXIV.1507.06217.

[2] Peter Bubenik. Statistical topological data analysis using persistence landscapes. *Journal of Machine Learning Research, 16 (2015), 77-102*, July 2012. doi: 10.48550/ARXIV.1207.6437.

[3] Matteo Caorsi, Raphael Reinauer, and Nicolas Berkouk. giotto-deep: A python package for topological deep learning. *Journal of Open Source Software*, 7(79):4846, November 2022. ISSN 2475-9066. doi: 10.21105/joss.04846.

[4] Gunnar Carlsson and Mikael Vejdemo-Johansson. *Topological data analysis with applications*. Cambridge University Press, 2021. ISBN 9781108838658.

[5] Mathieu Carrière, Frédéric Chazal, Yuichi Ike, Théo Lacombe, Martin Royer, and Yuhei Umeda. Perslay: A neural network layer for persistence diagrams and new graph topological signatures. April 2019. doi: 10.48550/ARXIV.1904.09378.

[6] Liliana Blanco Castañeda, Viswanathan Arunachalam, and Delvamuthu Dharmaraja. *Introduction to Probability and Stochastic Processes with Applications*. Wiley, June 2012. ISBN 9781118344972. doi: 10.1002/9781118344972.

[7] Nicola De Cao and Wilker Aziz. The power spherical distribution. June 2020. doi: 10.48550/ARXIV.2006.04437.

[8] Persi Diaconis, Susan Holmes, and Mehrdad Shahshahani. Sampling from a manifold. June 2012. doi: 10.48550/ARXIV.1206.6913.

[9] Olga Dunaeva, Herbert Edelsbrunner, Anton Lukyanov, Michael Machin, Daria Malkova, Roman Kuvaev, and Sergey Kashin. The classification of endoscopy images with persistent homology. *Pattern Recognition Letters*, 83:13–22, November 2016. ISSN 0167-8655. doi: 10.1016/j.patrec.2015.12.012.

[10] Herbert Edelsbrunner and John L. Harer. *Computational Topology: An Introduction*. American Mathematical Society, Providence, R.I, 2010. ISBN 9781470467692.

[11] Georgina Gonzalez, Arina Ushakova, Radmila Sazdanovic, and Javier Arsuaga. Prediction in cancer genomics using topological signatures and machine learning. In Nils A. Baas, Gunnar E. Carlsson, Gereon Quick, Markus Szymik, and Marius Thaule, editors, *Topological Data Analysis*, Abel Symposia, pages 247–276. Springer International Publishing, 2020. ISBN 9783030434083. doi: 10.1007/978-3-030-43408-3_10.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. December 2015. doi: 10.48550/ARXIV.1512.03385.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[14] Walter Ledermann and Alan J. Weir. *Introduction to group theory*. Longman mathematics series. Longman, Harlow, second edition, 1996. ISBN 0582259541.

[15] Luca Martino, David Luengo, and Joaquín Míguez. *Independent Random Sampling Methods*. Springer International Publishing, 2018. ISBN 9783319726342. doi: 10.1007/978-3-319-72634-2.

[16] Mervin E. Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Communications of the ACM*, 2(4):19–20, apr 1959. doi: 10.1145/377939.377946.

[17] NumPy. Legacy random generation — numpy v1.26 manual.
URL: `https://numpy.org/doc/stable/reference/random/legacy.html`. Online; accessed June 1, 2024.

[18] Raúl Rabadán and Andrew J. Blumberg. *Topological data analysis for genomics and evolution*. Cambridge University Press, Cambridge, 2019. ISBN 9781107159549.

[19] Bastian Rieck, Tristan Yates, Christian Bock, Karsten Borgwardt, Guy Wolf, Nicholas Turk-Browne, and Smita Krishnaswamy. Uncovering the topology of time-varying fmri data using cubical persistence. June 2020. doi: 10.48550/ARXIV.2006.07882.

[20] Christian P. Robert and George Casella. *Monte Carlo statistical methods*. Springer texts in statistics. Springer, New York, NY, 2nd edition, 2004. ISBN 9780387212395. Literaturverz. S. [591] - 622.

[21] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, April 2015. ISSN 1573-1405. doi: 10.1007/s11263-015-0816-y.

[22] The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, version 3.9.0, 2023.
URL: `https://gudhi.inria.fr/doc/3.9.0/`.

Appendix A

Benchmark results

Sphere/torus classification

Table A.1: The results for each of the pipelines on the sphere/torus classification task. The five columns are, in order from left to right, the vectorization method used by the pipeline, its validation accuracy, its test accuracy, its FSC + PH time, and its vectorization time. The header above each group of results denotes the dataset from which the results were obtained.

Vec. method	**V. accuracy**	**T. accuracy**	**FSC + PH time (s)**	**Vec. time (s)**
Sphere/torus classification, $n\_points = 50$, $noise\_scales = 0.0$				
Concat.	0.560000	0.535000	0.001000	0.629048
Mean/var.	0.500000	0.500000	0.001000	0.403031
Pers. Im.	0.975000	0.955000	2.839778	14.907672
Pers. Lsc.	0.750000	0.780000	2.839778	23.181477
Sphere/torus classification, $n\_points = 50$, $noise\_scales = 0.1$				
Concat.	0.570000	0.465000	0.001001	0.607055
Mean/var.	0.500000	0.500000	0.001001	0.391043
Pers. Im.	0.930000	0.850000	3.112748	15.229616
Pers. Lsc.	0.725000	0.705000	3.112748	23.279111
Sphere/torus classification, $n\_points = 50$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.570000	0.460000	0.002001	0.585074
Mean/var.	0.500000	0.500000	0.002001	0.408635
Pers. Im.	0.930000	0.800000	3.106755	15.167710
Pers. Lsc.	0.725000	0.645000	3.106755	23.153241
Sphere/torus classification, $n\_points = 50$, $noise\_scales = 0.2$				
Concat.	0.550000	0.555000	0.001002	0.639049
Mean/var.	0.500000	0.500000	0.001002	0.397038
Pers. Im.	0.795000	0.775000	3.253766	15.363204
Pers. Lsc.	0.675000	0.675000	3.253766	23.370855
Sphere/torus classification, $n\_points = 50$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.550000	0.550000	0.002001	0.646563

Mean/var.	0.500000	0.500000	0.002001	0.424050
Pers. Im.	0.795000	0.700000	3.231068	15.312769
Pers. Lsc.	0.675000	0.625000	3.231068	23.081333
Sphere/torus classification, $n\_points = 100$, $noise\_scales = 0.0$				
Concat.	0.530000	0.505000	0.001001	0.629150
Mean/var.	0.500000	0.500000	0.001001	0.398028
Pers. Im.	1.000000	0.995000	6.622982	16.187864
Pers. Lsc.	0.915000	0.920000	6.622982	24.698642
Sphere/torus classification, $n\_points = 100$, $noise\_scales = 0.1$				
Concat.	0.520000	0.545000	0.002001	0.587060
Mean/var.	0.500000	0.500000	0.002001	0.427847
Pers. Im.	0.980000	0.630000	7.241061	16.453200
Pers. Lsc.	0.880000	0.865000	7.241061	26.457679
Sphere/torus classification, $n\_points = 100$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.520000	0.550000	0.001001	0.576054
Mean/var.	0.500000	0.500000	0.001001	0.397032
Pers. Im.	0.980000	0.575000	7.280749	16.435762
Pers. Lsc.	0.880000	0.770000	7.280749	26.346954
Sphere/torus classification, $n\_points = 100$, $noise\_scales = 0.2$				
Concat.	0.515000	0.450000	0.002001	0.651056
Mean/var.	0.500000	0.500000	0.002001	0.400035
Pers. Im.	0.890000	0.830000	7.631363	16.949578
Pers. Lsc.	0.785000	0.780000	7.631363	24.781306
Sphere/torus classification, $n\_points = 100$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.515000	0.425000	0.003001	0.648573
Mean/var.	0.500000	0.500000	0.003001	0.404034
Pers. Im.	0.890000	0.725000	7.783599	16.988455
Pers. Lsc.	0.785000	0.705000	7.783599	24.822800
Sphere/torus classification, $n\_points = 250$, $noise\_scales = 0.0$				
Concat.	0.580000	0.460000	0.004999	0.966611
Mean/var.	0.500000	0.500000	0.004999	0.413031
Pers. Im.	1.000000	1.000000	19.509545	17.953882
Pers. Lsc.	1.000000	1.000000	19.509545	19.675240
Sphere/torus classification, $n\_points = 250$, $noise\_scales = 0.1$				
Concat.	0.540000	0.470000	0.011000	1.077602
Mean/var.	0.500000	0.500000	0.011000	0.422042
Pers. Im.	0.990000	0.995000	21.813271	20.989073
Pers. Lsc.	0.975000	0.980000	21.813271	22.172493

Sphere/torus classification, $n\_points = 250$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.540000	0.450000	0.010001	1.071605
Mean/var.	0.500000	0.500000	0.010001	0.410036
Pers. Im.	0.990000	0.835000	22.044628	20.830277
Pers. Lsc.	0.975000	0.815000	22.044628	22.109638
Sphere/torus classification, $n\_points = 250$, $noise\_scales = 0.2$				
Concat.	0.570000	0.455000	0.010000	1.014088
Mean/var.	0.500000	0.500000	0.010000	0.430049
Pers. Im.	0.940000	0.875000	22.837749	23.167748
Pers. Lsc.	0.905000	0.880000	22.837749	26.876685
Sphere/torus classification, $n\_points = 250$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.570000	0.465000	0.009999	0.996089
Mean/var.	0.500000	0.500000	0.009999	0.419111
Pers. Im.	0.940000	0.735000	23.042230	23.111767
Pers. Lsc.	0.905000	0.800000	23.042230	27.129955

Sphere/genus g torus binary classification

Table A.2: The results for each of the pipelines on the sphere/genus g torus binary classification task. The five columns are, in order from left to right, the vectorization method used by the pipeline, its validation accuracy, its test accuracy, its FSC + PH time, and its vectorization time. The header above each group of results denotes the dataset from which the results were obtained.

Vec. method	**V. accuracy**	**T. accuracy**	**FSC + PH time (s)**	**Vec. time (s)**
Sphere/genus g torus binary classification, $n\_points = 250$, $noise\_scales = 0.0$				
Concat.	0.545000	0.500000	0.006999	0.964096
Mean/var.	0.500000	0.500000	0.006999	0.451060
Pers. Im.	1.000000	1.000000	20.301996	18.357755
Pers. Lsc.	0.995000	0.985000	20.301996	26.201866
Sphere/genus g torus binary classification, $n\_points = 250$, $noise\_scales = 0.1$				
Concat.	0.525000	0.440000	0.006000	0.982092
Mean/var.	0.500000	0.500000	0.006000	0.405028
Pers. Im.	0.975000	0.985000	22.148046	21.892747
Pers. Lsc.	0.965000	0.925000	22.148046	28.939345
Sphere/genus g torus binary classification, $n\_points = 250$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.525000	0.435000	0.010000	0.983086

Mean/var.	0.500000	0.500000	0.010000	0.410029
Pers. Im.	0.975000	0.825000	22.185488	21.873529
Pers. Lsc.	0.965000	0.800000	22.185488	29.144387
Sphere/genus g torus binary classification, $n\_points = 250$, $noise\_scales = 0.2$				
Concat.	0.540000	0.465000	0.010000	1.119658
Mean/var.	0.500000	0.500000	0.010000	0.423066
Pers. Im.	0.915000	0.915000	22.967715	23.472778
Pers. Lsc.	0.900000	0.885000	22.967715	28.882437
Sphere/genus g torus binary classification, $n\_points = 250$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.540000	0.475000	0.011000	1.087611
Mean/var.	0.500000	0.500000	0.011000	0.403029
Pers. Im.	0.915000	0.830000	23.219722	23.530050
Pers. Lsc.	0.900000	0.740000	23.219722	28.550457
Sphere/genus g torus binary classification, $n\_points = 500$, $noise\_scales = 0.0$				
Concat.	0.555000	0.500000	0.009000	1.445639
Mean/var.	0.500000	0.500000	0.009000	0.437029
Pers. Im.	1.000000	1.000000	42.947647	26.798304
Pers. Lsc.	1.000000	0.985000	42.947647	23.376776
Sphere/genus g torus binary classification, $n\_points = 500$, $noise\_scales = 0.1$				
Concat.	0.545000	0.515000	0.014508	1.637171
Mean/var.	0.500000	0.500000	0.014508	0.435067
Pers. Im.	0.985000	0.985000	48.564628	36.626728
Pers. Lsc.	0.975000	0.960000	48.564628	30.215218
Sphere/genus g torus binary classification, $n\_points = 500$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.545000	0.495000	0.012000	1.631148
Mean/var.	0.500000	0.500000	0.012000	0.436057
Pers. Im.	0.985000	0.930000	48.794981	36.430006
Pers. Lsc.	0.975000	0.785000	48.794981	29.952077
Sphere/genus g torus binary classification, $n\_points = 500$, $noise\_scales = 0.2$				
Concat.	0.545000	0.505000	0.021000	1.527236
Mean/var.	0.500000	0.500000	0.021000	0.430029
Pers. Im.	0.950000	0.915000	50.375267	41.856408
Pers. Lsc.	0.955000	0.915000	50.375267	36.654224
Sphere/genus g torus binary classification, $n\_points = 500$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.545000	0.500000	0.023505	1.529648
Mean/var.	0.500000	0.500000	0.023505	0.424658
Pers. Im.	0.950000	0.855000	50.461986	41.615478
Pers. Lsc.	0.955000	0.815000	50.461986	36.666449

Sphere/genus g torus binary classification, $n\_points = 1000$, $noise\_scales = 0.0$				
Concat.	0.550000	0.440000	0.023002	2.246289
Mean/var.	0.500000	0.500000	0.023002	0.467539
Pers. Im.	1.000000	0.995000	92.323428	49.590144
Pers. Lsc.	1.000000	1.000000	92.323428	34.938439
Sphere/genus g torus binary classification, $n\_points = 1000$, $noise\_scales = 0.1$				
Concat.	0.540000	0.435000	0.027998	2.375285
Mean/var.	0.500000	0.500000	0.027998	0.457540
Pers. Im.	0.975000	1.000000	105.459805	68.313326
Pers. Lsc.	1.000000	1.000000	105.459805	48.661074
Sphere/genus g torus binary classification, $n\_points = 1000$, $noise\_scales = (0.1, 0.2)$				
Concat.	0.540000	0.440000	0.038506	2.415780
Mean/var.	0.500000	0.500000	0.038506	0.474029
Pers. Im.	0.975000	0.940000	104.239886	69.210200
Pers. Lsc.	1.000000	0.930000	104.239886	49.451481
Sphere/genus g torus binary classification, $n\_points = 1000$, $noise\_scales = 0.2$				
Concat.	0.540000	0.450000	0.021998	2.800839
Mean/var.	0.500000	0.500000	0.021998	0.468535
Pers. Im.	0.985000	0.975000	110.940959	78.066266
Pers. Lsc.	0.965000	0.910000	110.940959	54.185399
Sphere/genus g torus binary classification, $n\_points = 1000$, $noise\_scales = (0.2, 0.4)$				
Concat.	0.540000	0.450000	0.019999	2.826986
Mean/var.	0.500000	0.500000	0.019999	0.478031
Pers. Im.	0.985000	0.835000	111.160298	78.294521
Pers. Lsc.	0.965000	0.770000	111.160298	55.055532

Sphere/genus g torus genus regression

Table A.3: The results for each of the pipelines on the sphere/genus g torus genus regression task. The five columns are, in order from left to right, the vectorization method used by the pipeline, its validation MSE, its test MSE, its FSC + PH time, and its vectorization time. The header above each group of results denotes the dataset from which the results were obtained.

Vec. method	**V. MSE**	**T. MSE**	**FSC + PH time (s)**	**Vec. time (s)**
Sphere/genus g torus genus regression, $n\_points = 250$, $noise\_scales = 0.0$				
Concat.	2.533007	2.944623	0.022999	2.515782

Mean/var.	2.534250	2.940057	0.022999	1.717132
Pers. Im.	0.281423	1.551076	21.776293	67.015139
Pers. Lsc.	0.703417	0.697766	21.776293	101.671465
Sphere/genus g torus genus regression, $n\_points = 250$, $noise\_scales = 0.1$				
Concat.	2.570799	2.933698	0.025999	2.553350
Mean/var.	2.534250	2.940057	0.025999	1.680630
Pers. Im.	0.808571	0.863227	22.776890	54.713003
Pers. Lsc.	0.982640	1.346195	22.776890	86.779535
Sphere/genus g torus genus regression, $n\_points = 250$, $noise\_scales = (0.1, 0.2)$				
Concat.	2.570799	2.938824	0.028001	2.506724
Mean/var.	2.534250	2.940057	0.028001	1.691635
Pers. Im.	0.808571	1.333292	22.822453	54.708579
Pers. Lsc.	0.982640	2.143104	22.822453	88.624736
Sphere/genus g torus genus regression, $n\_points = 250$, $noise\_scales = 0.2$				
Concat.	2.536149	2.944653	0.026003	2.544219
Mean/var.	2.534250	2.940057	0.026003	1.783232
Pers. Im.	1.176763	1.401469	23.281565	47.292475
Pers. Lsc.	1.519971	1.716604	23.281565	80.427614
Sphere/genus g torus genus regression, $n\_points = 250$, $noise\_scales = (0.2, 0.4)$				
Concat.	2.536149	2.944653	0.027000	2.539731
Mean/var.	2.534250	2.940057	0.027000	1.675183
Pers. Im.	1.176763	2.204815	23.412917	47.637253
Pers. Lsc.	1.519971	2.666258	23.412917	80.125224
Sphere/genus g torus genus regression, $n\_points = 500$, $noise\_scales = 0.0$				
Concat.	2.721876	3.303597	0.028000	3.585086
Mean/var.	2.590487	3.332027	0.028000	1.486637
Pers. Im.	0.208775	0.234262	46.494550	72.459275
Pers. Lsc.	0.309218	0.343988	46.494550	117.698810
Sphere/genus g torus genus regression, $n\_points = 500$, $noise\_scales = 0.1$				
Concat.	2.695191	26.644284	0.029505	3.577353
Mean/var.	2.590487	3.332027	0.029505	1.478161
Pers. Im.	0.623970	0.597963	49.384149	68.310423
Pers. Lsc.	0.620516	0.840746	49.384149	118.413491
Sphere/genus g torus genus regression, $n\_points = 500$, $noise\_scales = (0.1, 0.2)$				
Concat.	2.695191	26.644284	0.030999	3.654902
Mean/var.	2.590487	3.332027	0.030999	1.465110
Pers. Im.	0.623970	1.151525	49.523718	67.669887
Pers. Lsc.	0.620516	1.553139	49.523718	117.577823

Sphere/genus g torus genus regression, $n\_points = 500$, $noise\_scales = 0.2$				
Concat.	2.673796	9.554685	0.031504	3.542438
Mean/var.	2.590487	3.332027	0.031504	1.496243
Pers. Im.	1.133883	1.284909	50.913331	61.476951
Pers. Lsc.	1.402922	1.361152	50.913331	107.181650
Sphere/genus g torus genus regression, $n\_points = 500$, $noise\_scales = (0.2, 0.4)$				
Concat.	2.673796	9.564072	0.031505	3.543358
Mean/var.	2.590487	3.332027	0.031505	1.459675
Pers. Im.	1.133883	2.225023	50.664580	61.867141
Pers. Lsc.	1.402922	2.535438	50.664580	108.438160
Sphere/genus g torus genus regression, $n\_points = 1000$, $noise\_scales = 0.0$				
Concat.	2.871638	4.855017	0.030507	7.842873
Mean/var.	2.803473	2.977581	0.030507	1.512616
Pers. Im.	0.087360	0.105875	102.848403	95.407976
Pers. Lsc.	0.152201	0.155298	102.848403	145.641761
Sphere/genus g torus genus regression, $n\_points = 1000$, $noise\_scales = 0.1$				
Concat.	2.798160	4.811952	0.029999	8.066624
Mean/var.	2.803473	2.977581	0.029999	1.557668
Pers. Im.	0.403862	0.467015	109.281283	94.126250
Pers. Lsc.	0.496314	0.716646	109.281283	140.428404
Sphere/genus g torus genus regression, $n\_points = 1000$, $noise\_scales = (0.1, 0.2)$				
Concat.	2.798160	4.843832	0.043508	8.447542
Mean/var.	2.803473	2.977581	0.043508	1.578140
Pers. Im.	0.403862	1.629145	109.452761	94.443418
Pers. Lsc.	0.496314	2.197214	109.452761	142.148820
Sphere/genus g torus genus regression, $n\_points = 1000$, $noise\_scales = 0.2$				
Concat.	2.861088	4.581490	0.046004	8.943905
Mean/var.	2.803473	2.977581	0.046004	1.573125
Pers. Im.	1.346547	0.901616	112.715025	96.306923
Pers. Lsc.	1.678204	1.334627	112.715025	122.476285
Sphere/genus g torus genus regression, $n\_points = 1000$, $noise\_scales = (0.2, 0.4)$				
Concat.	2.861088	4.617554	0.032508	9.239039
Mean/var.	2.803473	2.977581	0.032508	1.636793
Pers. Im.	1.346547	2.163825	113.689675	97.843237
Pers. Lsc.	1.678204	2.432236	113.689675	121.646036

Power spherical concentration regression

Table A.4: The results for each of the pipelines on the power spherical concentration regression task. The five columns are, in order from left to right, the vectorization method used by the pipeline, its validation MSE, its test MSE, its FSC + PH time, and its vectorization time. The header above each group of results denotes the dataset from which the results were obtained.

Vec. method	**V. MSE**	**T. MSE**	**FSC + PH time (s)**	**Vec. time (s)**
Power spherical concentration regression, $n\_points = 50$, $noise\_scales = 0.0$				
Concat.	1.773554	1.549748	0.041003	1.283187
Mean/var.	1.748508	1.565125	0.041003	1.398116
Pers. Im.	0.511938	0.537841	2.651738	58.637333
Pers. Lsc.	0.741372	0.576489	2.651738	49.222471
Power spherical concentration regression, $n\_points = 50$, $noise\_scales = 0.1$				
Concat.	1.794144	1.802386	0.001000	1.338607
Mean/var.	1.748508	1.565125	0.001000	1.384104
Pers. Im.	0.644236	0.768679	2.930337	58.633793
Pers. Lsc.	0.808097	0.743127	2.930337	48.486941
Power spherical concentration regression, $n\_points = 50$, $noise\_scales = (0.1, 0.2)$				
Concat.	1.794144	1.801468	0.000000	1.361163
Mean/var.	1.748508	1.565125	0.000000	1.389116
Pers. Im.	0.644236	0.858819	2.993226	58.430226
Pers. Lsc.	0.808097	0.743499	2.993226	48.602125
Power spherical concentration regression, $n\_points = 50$, $noise\_scales = 0.2$				
Concat.	1.770820	1.549782	0.001001	1.219614
Mean/var.	1.748508	1.565125	0.001001	1.400128
Pers. Im.	0.722095	0.821064	3.155736	55.915487
Pers. Lsc.	0.846283	0.834495	3.155736	50.185985
Power spherical concentration regression, $n\_points = 50$, $noise\_scales = (0.2, 0.4)$				
Concat.	1.770820	1.549782	0.000000	1.226101
Mean/var.	1.748508	1.565125	0.000000	1.402141
Pers. Im.	0.722095	1.114149	3.141823	56.057408
Pers. Lsc.	0.846283	1.208090	3.141823	47.927477
Power spherical concentration regression, $n\_points = 100$, $noise\_scales = 0.0$				
Concat.	1.527460	1.522628	0.002002	1.855170
Mean/var.	1.527651	1.558181	0.002002	1.432116
Pers. Im.	0.393172	0.516281	6.303984	57.755501
Pers. Lsc.	0.442568	0.676745	6.303984	62.330203

Power spherical concentration regression, $n\_points = 100$, $noise\_scales = 0.1$				
Concat.	1.524533	1.522847	0.001000	1.652484
Mean/var.	1.527651	1.558181	0.001000	1.419103
Pers. Im.	0.557579	0.730179	7.071849	53.509495
Pers. Lsc.	0.505433	1.532150	7.071849	62.609599
Power spherical concentration regression, $n\_points = 100$, $noise\_scales = (0.1, 0.2)$				
Concat.	1.524533	1.522847	0.001001	1.678207
Mean/var.	1.527651	1.558181	0.001001	1.414100
Pers. Im.	0.557579	1.080832	7.124576	54.731180
Pers. Lsc.	0.505433	1.532150	7.124576	63.386528
Power spherical concentration regression, $n\_points = 100$, $noise\_scales = 0.2$				
Concat.	1.527458	2.007598	0.018000	1.687656
Mean/var.	1.527651	1.558181	0.018000	1.436134
Pers. Im.	0.745100	0.875043	7.611580	51.259758
Pers. Lsc.	0.708549	0.887840	7.611580	60.194247
Power spherical concentration regression, $n\_points = 100$, $noise\_scales = (0.2, 0.4)$				
Concat.	1.527458	2.515888	0.001000	1.741638
Mean/var.	1.527651	1.558181	0.001000	1.449629
Pers. Im.	0.745100	1.008929	7.625408	51.352604
Pers. Lsc.	0.708549	1.028614	7.625408	60.257998
Power spherical concentration regression, $n\_points = 250$, $noise\_scales = 0.0$				
Concat.	1.707283	1.236698	0.009504	2.814244
Mean/var.	1.693242	1.236744	0.009504	1.454123
Pers. Im.	0.393743	0.384196	18.737776	51.980587
Pers. Lsc.	0.446086	0.394168	18.737776	76.716535
Power spherical concentration regression, $n\_points = 250$, $noise\_scales = 0.1$				
Concat.	1.711469	1.237368	0.022000	2.673320
Mean/var.	1.693242	1.236744	0.022000	1.434118
Pers. Im.	0.601968	0.473908	21.705134	55.377315
Pers. Lsc.	0.675465	0.508252	21.705134	73.608826
Power spherical concentration regression, $n\_points = 250$, $noise\_scales = (0.1, 0.2)$				
Concat.	1.711469	1.237368	0.038000	2.783275
Mean/var.	1.693242	1.236744	0.038000	1.432115
Pers. Im.	0.601968	0.571433	21.680995	55.066148
Pers. Lsc.	0.675465	0.637965	21.680995	71.358160
Power spherical concentration regression, $n\_points = 250$, $noise\_scales = 0.2$				
Concat.	1.905845	1.267326	0.035999	2.747344
Mean/var.	1.693242	1.236744	0.035999	1.454132

Pers. Im.	0.720727	0.520395	22.786781	61.716527
Pers. Lsc.	0.795692	0.671828	22.786781	73.215675
Power spherical concentration regression, $n\_points = 250$, $noise\_scales = (0.2, 0.4)$				
Concat.	1.905845	1.277438	0.037510	2.728504
Mean/var.	1.693242	1.236744	0.037510	1.412106
Pers. Im.	0.720727	0.848335	23.129259	61.405952
Pers. Lsc.	0.795692	0.927691	23.129259	73.000244

Appendix B

Code repository

The code for the benchmarking suite, as well the files used to run the tests presented in this thesis, are available on GitHub: `https://github.com/sondrebr/MastersThesis`.

The benchmarking suite is contained in the *BenchTDA* folder, which can be imported as a Python package. The folder contains several files:

- *__init__.py*, exporting the *benchmark* function and the base classes for the model and vectorization classes.
- *_base_classes.py*, containing the base classes for the model and vectorization classes.
- *_benchmark.py*, containing the benchmarking tool itself, along with the different tasks and task parameters.
- *datasets.py*, containing the dataset generators for the different tasks.
- *manifolds.py*, containing the code used to sample from each of the manifolds.

The *ExamplePipelines* folder includes the files *fscs.py*, *phs.py*, and *vecs.py*, which contain the FSC function for the TDA pipelines, the PH function for the TDA pipelines, and the model classes and vectorization classes for all of the pipelines.

The *run_benchmark.py* file can be used to run the tests presented in this thesis, using the included conda environment in the *environment.yml* file.