

AMSI Summer School – 2026

Computational and Combinatorial Algebraic Topology: Group Project – Report

Group 6: Zihan Lin, Kevin Tran, Brice Arrigo & Jude Hine

February 4, 2026

Abstract

This report details the computational project for Computational and Combinatorial Algebraic Topology at AMSI Summer School 2026. Using the GUDHI library in Python, we apply persistent homology to classify different geometric configurations of planar point clouds. In addition, we demonstrate how persistence barcodes can be used to identify and remove outliers in noisy data, recovering the underlying topological structure.

Contents

1	Generate the dataset	2
2	Cluster the dataset	4
3	Noise Influence	8
4	Outlier points	10
5	Removing outliers	12
6	Conclusion	13

1 Generate the dataset

The code for generating circles is adapted from the code presented in the week three tutorial.

The following function generates a point cloud containing 200 points, sampling from two disjoint concentric noisy circles.

```
def generate_two_disjoint_concentric_noisy_circle(n=200, radius1=1,
radius2=2, center=(0, 0), noise_std=0.1):
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx, cy = center
    base = np.column_stack([cx + radius1 * ux, cy + radius1 * uy])
    noise = np.random.rand(100, 2)
    points1 = base + noise_std*radius1*noise
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx, cy = center
    base = np.column_stack([cx + radius2 * ux, cy + radius2 * uy])
    noise = np.random.rand(100, 2)
    points2 = base + noise_std * radius1 * noise
    points = np.vstack([points1, points2])
    return points
```

The code below generates a point cloud containing 200 points, sampling from two disjoint noisy circles that are not nested.

```
def generate_two_disjoint_noisy_circle(n=200, radius=1, center1=(0,0),
center2=(3,0), noise_std=0.1):
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx1, cy1 = center1
    base = np.column_stack([cx1 + radius * ux, cy1 + radius * uy])
    noise = np.random.rand(100, 2)
    points1 = base + noise_std * radius * noise
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx2, cy2 = center2
```

```
base = np.column_stack([cx2 + radius * ux, cy2 + radius * uy])
noise = np.random.rand(100, 2)
points2 = base + noise_std * radius * noise
points = np.vstack([points1, points2])
return points
```

The code below generates a point cloud containing 200 points, sampling from two adjacent noisy circles.

```
def generate_two_adjacent_noisy_circle(n=200, radius=1, center1=(0,0),
center2=(2,0), noise_std=0.1):
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx1, cy1 = center1
    base = np.column_stack([cx1 + radius * ux, cy1 + radius * uy])
    noise = np.random.rand(100, 2)
    points1 = base + noise_std * radius * noise
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=100)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx2, cy2 = center2
    base = np.column_stack([cx2 + radius * ux, cy2 + radius * uy])
    noise = np.random.rand(100, 2)
    points2 = base + noise_std * radius * noise
    points = np.vstack([points1, points2])
    return points
```

We store the three classes of point clouds in a numpy array, **dataset**.

```
dataset = []
for i in range(0,3):
    for j in range(0,100):
        if i == 0:
            pc = generate_two_disjoint_concentric_noisy_circle()
        elif i == 1:
            pc = generate_two_disjoint_noisy_circle()
        else:
            pc = generate_two_adjacent_noisy_circle()
        dataset.append(pc)
dataset=np.array(dataset)
```

2 Cluster the dataset

We first analyse the persistence barcodes qualitatively in order to identify distinguishing features between the different classes. These features are then quantified via suitable interval counting heuristics on the persistence diagrams, which are used to cluster our dataset.

Fig. 1 is the persistence barcode of the point cloud obtained from sampling two disjoint concentric noisy circles.

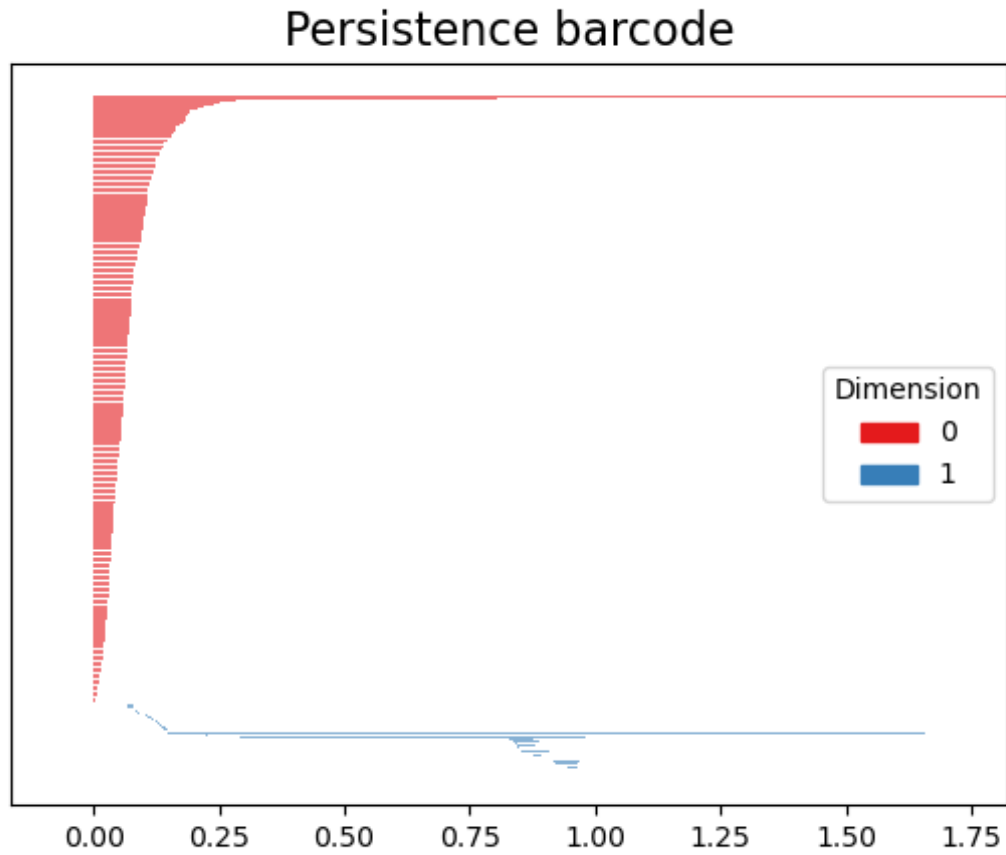


Figure 1: persistence barcode of sample from two concentric circles.

Fig. 2 is a persistence barcode of the point cloud obtained from sampling two adjacent noisy circles.

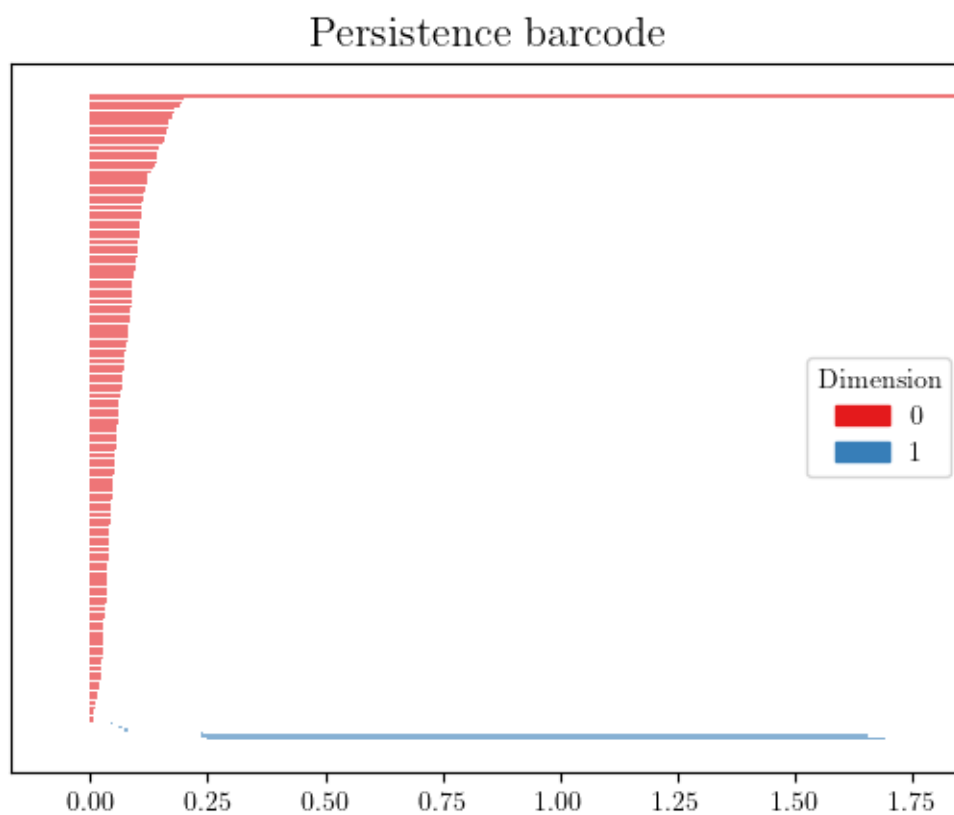


Figure 2: persistence barcode of sample from two adjacent circles.

Fig. 3 is a persistence barcode of the point cloud obtained from sampling two disjoint noisy circles that are not nested.

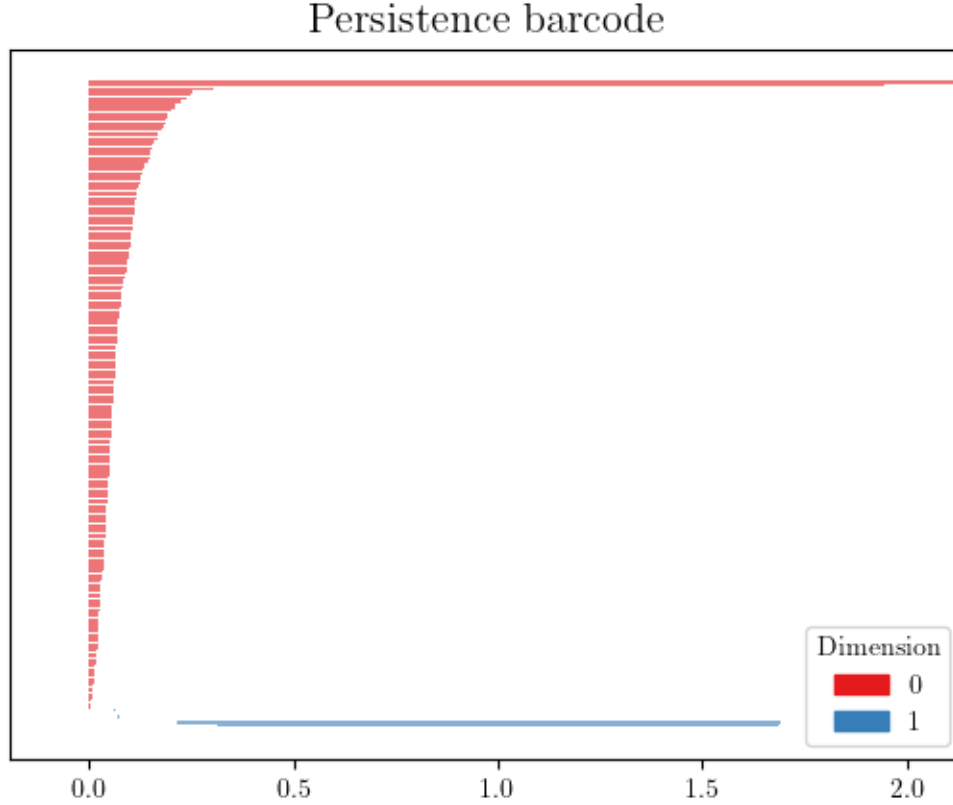


Figure 3: persistence barcode of sample from two disjoint circles.

Examining dimension 0 in Fig. 1, persistence intervals with significantly longer lifetimes than those observed in the other two classes.

Looking at dimension 1 of Fig. 1, we see the persistence barcodes are noticeably different from the other two classes, Figs. 2 and 3, whose dimension 1 barcodes consists of only two intervals with a relatively long lifespan.

We see that Fig. 2 has only one long barcode in dimension 0, since the adjacent circles only have one connected component. But the other two classes have more the one long dimension 0 barcode.

So, in summary, we will check for the following properties:

1. if dimension 0 has two very long intervals, then the point cloud is comes from either nested or disjoint circles;
2. if dimension 1 has many intervals, then the point cloud comes from two nested circles; and

3. if it is neither of the two, then it must be two adjacent circles.

We first present a psuedocode version of the algorithm, followed by its implementation in python.

```

Input: data — nested array of point clouds
Output: classification — list with elements in [0,1,2], corresponding
        to nested, disjoint and adjacent circles respective.

Let PH = []
Let classification = []
for each point cloud in data
    Compute Rips filtration
    Calculate persistence
    Let  $H\backslash_0, H\backslash_1$  = persistence intervals for dimensions 0 and 1
    Append [ $H\backslash_0, H\backslash_1$ ] to PH

for each persistence interval in PH
    if the largest finite death time  $H\backslash_0 \geq 0.8$  and the second
        largest finite death time  $H\backslash_0 \geq 0.2$ 
        if the number of intervals in  $H\backslash_1 \geq 12$ 
            the point cloud comes from two nested circles
            append 0 to classification
        else:
            the point cloud comes from two disjoint circles
            append 1 to classification
    if the largest finite death time  $H\backslash_0 \leq 0.5$ 
        the point cloud comes from two adjacent circles
        append 2 to classification
return classification

```

Where the constants were chosen empirically through trial and error. This was done by examining the largest persistence lifetimes for each barcode.

This is practically already the python algorithm, which is as follows

```

def classify_my_data(data):
    PH = []
    classification = []
    for points in data:
        rips = gudhi.RipsComplex(points=points)
        st = rips.create_simplex_tree(max_dimension=2)
        st.persistence()

```

```

H0 = st.persistence_intervals_in_dimension(0)
H1 = st.persistence_intervals_in_dimension(1)
PH.append([H0,H1])

for persistence_interval_pair in PH:
    persistence_0 , persistence_1 = persistence_interval_pair[0] ,
        persistence_interval_pair[1]
    persistence_0 , persistence_1 = persistence_0[::-1] , persistence_1
        [::-1]
    if persistence_0[1][1]>=0.8 and persistence_0[2][1] >= 0.2:
        if len(persistence_1) >= 12:
            classification.append(0)
        else:
            classification.append(1)
    if persistence_0[1][1]<=0.5:
        classification.append(2)
return classification

result = classify_my_data(dataset)
np.unique(result , return_counts=True)

```

The last line returns the frequency of each element in the list. Which running on our generated dataset returns

```
(array([0, 1, 2]), array([100, 100, 100], dtype=int64))
```

So we have successfully used persistent homology to come up with a heuristics approach to classifying our point clouds. While there are clear opportunities to refine these heuristics and improve robustness, this method proves to be highly effective for the dataset under consideration. We will see in Section 3 how this approach breaks down with more noise.

3 Noise Influence

In Section 1 when generating our dataset, we fixed `noise_std = 0.1`. We observed setting `noise_std = 0.3` fail to classify most of the data. For example,

```
(array([0, 1, 2]), array([ 45, 1, 100], dtype=int64))
```

indicates that while the algorithm continues to correctly identify the adjacent circle configurations, the increased variance in the point clouds makes it difficult to reliably distinguish between nested and disjoint circles. In particular, some of the detected classes may correspond to false positives.

Increasing further to `noise_std = 0.5`, causes the classification procedure to fail almost entirely, as the persistence-based heuristics no longer provide sufficient separation between the different configurations. For example,

```
(array([0, 2]), array([ 1, 116], dtype=int64))
```

shows that the algorithm is no longer able to detect the disjoint circle configuration, as the corresponding cluster is completely absent.

This problem makes sense within the theoretical framework of persistent homology. This is because the distance between two disjoint concentric noisy circles or two disjoint noisy circles that are not nested are both 1. So when noise is over 0.5, the differences in persistence homology among them will be distorted. For example, two connected components may be considered one connected component now, which will influence PH_0 .

Further empirical evidence showing the influence of noise on the accuracy of our algorithm is the following noisy point cloud with and its associated barcode.

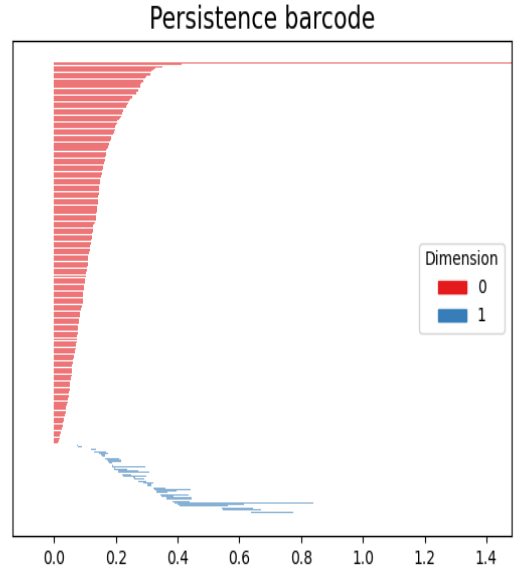
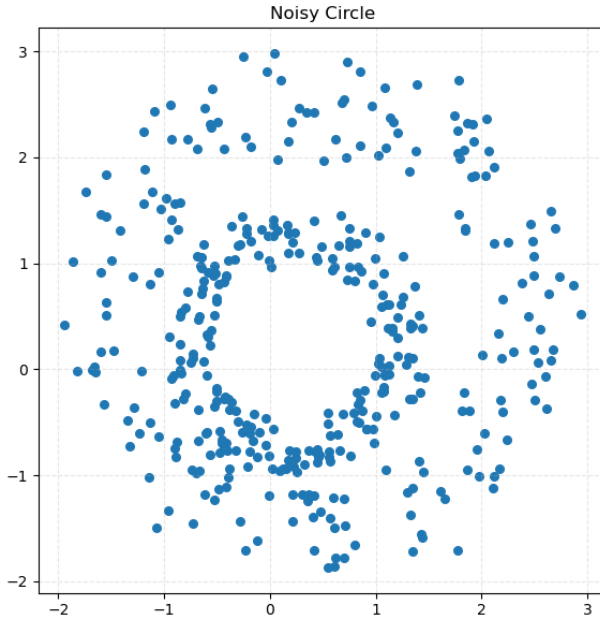


Figure 4: Point cloud sampled from two concentric circles with `noise_std = 0.5`.

Figure 5: Barcode of Rips complex of point cloud.

We can see Fig. 5 has a noticeably different structure in dimension 1, to that of its less noisy counterpart Fig. 1. Visually showing how increased sampling noise leads to less accurate classification based on persistent homology.

4 Outlier points

The code below generates a point cloud, P_1 , consisting of 200 points, sampling a noisy circle and creates a second point cloud P_2 by adding 20 outlier points to P_1 . We then display the persistence diagrams of P_1 and P_2

```
def generate_random_points(n=20, xlim=(-1.0, 1.0), ylim=(-1.0, 1.0)):
    x = np.random.uniform(xlim[0], xlim[1], size=n)
    y = np.random.uniform(xlim[0], xlim[1], size=n)
    return np.column_stack([x, y])

def generate_noisy_circle(n=200, radius=1, center=(0, 0), noise_std=0.1):
    theta = np.random.uniform(0.0, 2.0 * np.pi, size=n)
    ux = np.cos(theta)
    uy = np.sin(theta)
    cx, cy = center
    base = np.column_stack([cx + radius * ux, cy + radius * uy])
    noise = np.random.rand(n, 2)
    points = base + noise_std * radius * noise
    return points

P1 = generate_noisy_circle()
P2 = np.vstack((P1, generate_random_points()))
rips_complex = gudhi.RipsComplex(points=P1)
st = rips_complex.create_simplex_tree(max_dimension=2)
a = st.persistence()
gudhi.plot_persistence_barcode(a, legend=True)
plt.show()

rips_complex = gudhi.RipsComplex(points=P2)
st = rips_complex.create_simplex_tree(max_dimension=2)
a = st.persistence()
gudhi.plot_persistence_barcode(a, legend=True)
plt.show()
```

P_1 is a noisy circle, with only one connected component and one cycle. So only 1 dimension of PH_0 and PH_1 will live for a long time. In contrast, the outlier points in P_2 will lead to more long living classes in persistent homology since some of them may be far from the circle, forming new connected components and edges when the filtration index is small.

We see this information in the persistence barcodes of P_1 and P_2 in Figs. 6 and 7 respectively.

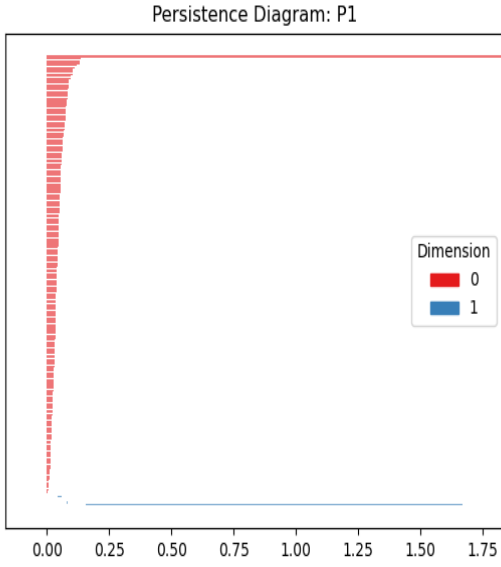


Figure 6: Persistence diagram of P_1 .

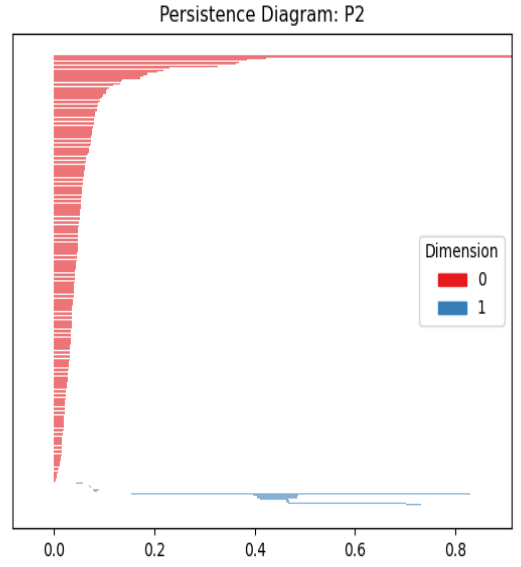


Figure 7: Persistence diagram of P_2 .

Notice, the influence of outlier points is visually clear from Fig. 7. With these observations, we are ready to recover a persistence diagram similar to P_1 .

5 Removing outliers

As discussed in Section 4, the points further away from the circle have greater influence over the persistent homology of P_2 . So to recover a persistence diagram similar to that of P_1 , we will remove points which are far away from other points.

Let the *neighbourhood* of a point p be the area where the distance between p and any point in this area is less than 0.3. A point p is an *outlier point* if the neighbourhood of p contains fewer than 5 points, we will remove any outlier point from the point cloud. Note, however, the points we remove may not be the 20 outlier points we added before.

The following is an implementation of this outlier removal algorithm.

```
D = cdist(P2, P2)
neighbors = (D < 0.3).sum(axis=1)
clean = neighbors > 5
P2_clean = P2[clean]
rips_complex = gudhi.RipsComplex(points=P2_clean)
st = rips_complex.create_simplex_tree(max_dimension=2)
a = st.persistence()
gudhi.plot_persistence_barcode(a, legend=True)
plt.show()
```

Where `P2_clean` is the recovered point cloud, with persistence barcode Fig. 8

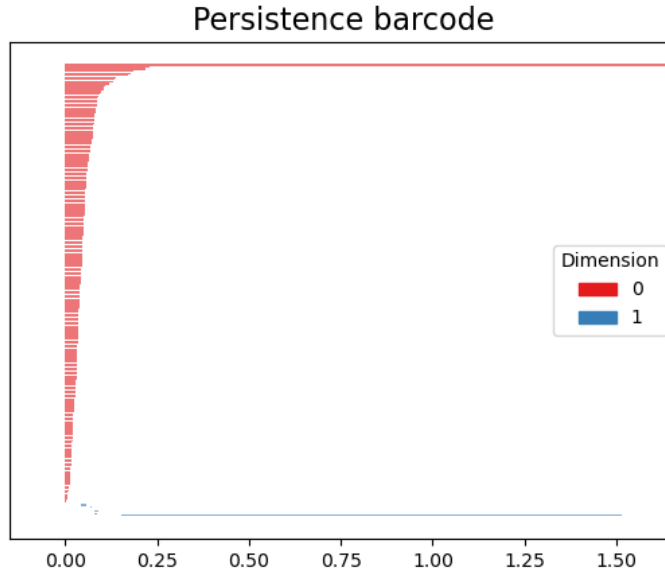


Figure 8: Persistence barcode for cleaned P_2 .

Which is similar to the persistence diagram for P_1 . This demonstrates that removing points which have only a small number of neighbours within a suitably chosen open neighbourhood is effective at removing outliers and recovering the underlying topological structure of the data.

6 Conclusion

In conclusion, we have demonstrated the utility of persistent homology as a tool for data analysis. However, as with all tools, it has its limitations. The performance of persistence-based methods is sensitive to noise and sampling density, which can obscure meaningful topological features. Moreover, the computational cost of Vietoris-Rips filtrations and the interpretation of persistence diagrams must be taken into account. Nevertheless, persistent homology remains a powerful approach for extracting qualitative structural information from complex data.