2021 SPRING CS6170 PROJECT 1 REPORT
DUE: MAR 4TH, 2021

SANGHOON KWAK

1. Submitted Files

1. `report.pdf` – Project 1 report file.
2. Source codes
   - `part1.py` – Source code for Part 1.
   - `part2(boundary).py` – Source code for Part 2, with *Ripser* filtration out of extracted boundary points.
   - `part2(heightFtn).py` – Source code for Part 2, with sublevel filtration where the height function is motivated from [2, Section 5.1].
3. Data
   - Input data
     - `data\octa.txt` and `data\cylinder.txt`
     - Ten `MPEG-7` image samples: `data\MPEG\bat-19.gif, beetle-13.gif, butterfly-10.gif, camel-16.gif, cattle-3.gif, crown-10.gif, device3-5.gif, dog-7.gif, horse-18.gif, octopus-12.gif`
   - Output barcodes – *written as from the longest to the shortest.*
     - `part1_barcodes\octa_longest_8_barcodes.txt` – Contains the longest eight persistence barcodes of dimension 1 from `octa.txt`.
     - `part1_barcodes\cylinder_longest_barcode.txt` – Contains the longest persistence barcode of dimension 1 from `cylinder.txt`.
     - Ten `part2_barcodes(boundary)\XYZ.gif.txt` files – Each file contains persistence barcodes of dimension 0 for each `MPEG-7` image, using *Ripser* filtration.
     - Ten `part2_barcodes(heightFtn)\XYZ.gif.txt` files – Each file contains persistence barcodes of dimension 0 for each `MPEG-7` image, using sublevel filtration.
4. Screenshots
   - `PDiagram(octa).png`, `PDiagram(cylinder).png` – Persistence diagrams for part 1.
   - `XYZ-execution-log.png` – The execution logs for each program in part 1 and 2. This contains the measured time it took for each program to yield output.

Date: Feb 27th, 2021.

2. Part 1: Point cloud to barcodes

2.1. **Summary of Source code `part1.py`.** The source code `part1.py` is quite straightforward. It first reads the data from `octa.txt` and `cylinder.txt` and converts it into Ripser-readable data structure, `ndarray`. Then use Ripser to process this data to extract barcodes for dimension 0 and 1. (Note the default value for `maxdim` parameter of `ripser` is 1, which indicates the maximum dimension of persistence homology for Ripser to calculate.) We sort it in decreasing order by the *lifespan(=death-birth)* of each barcode, so that we can pick the longest barcodes at ease. Finally, write those barcodes into `.txt` file, where each line has the format of `[birth, death)`.

2.2. **Topological Spaces from which the Data is Sampled.** For `octa.txt`, we can observe that there are four barcodes of length(lifespan) $190 \sim 194$ with death $213 \sim 216$, and four of length $47 \sim 52$ with death $67 \sim 70$. See the left diagram of Figure 1. The rest of the barcodes of dimension 1, has length < 14 (the 9th longest one is $\sim [19.14, 33.26)$). That being said, we can guess that the ambient topological space for `octa.txt` might have **four big loops of diameter** ~ 214 [1] and **four medium-sized loops of diameter** ~ 68, and a lot of possibly smaller loops of diameter $\lesssim 33$. Meanwhile, noting that all but one of 0th and the most of 1st persistence homology die within $20 \sim 32$ range, we can conclude that the data is separated by distance $20 \sim 32$. Therefore, we can say that the loops of diameter $\lesssim 33$ are in fact the noises from the distribution of sample points, rather than represent a meaningful loop in the ambient topological space.

Similarly, for `cylinder.txt`, the longest barcode has of length ~ 1.44 with death 1.74. The second longest one is $\sim [0.29, 0.69)$ and the rest of the barcodes of dimension 1 have $\lesssim 0.2$ life span. See the right diagram of Figure 1. Thus, we can say that **longest barcode represents a loop surrounding the (hollow) cylinder X, which accounts for $\beta_1(X) = 1$. Plus, the fact that this longest one dies at 1.74 tells us that the width of the cylinder is ~ 1.74.** Just for fun, to explain the origin of the second longest outlier $[0.29, 0.69)$, we guess the cylinders formed by the Rips complex at $d \in [0, 29, 0.69)$ have a *hole* on the side. If this is the case, then the two loops from the top and bottom of the cylinder are not homotopic, representing **distinct** 1st homology classes of the cylinder. When $d = 0.69$, the hole disappears, and the two loops are now homotopic to form a single 1-cycle of the cylinder, until $d = 1.74$, in which the loop will bound a 2-disk.

3. Part 2: Image to barcodes

Here we have done Part 2 with two different methods, given in the project 1 document. The first one is to use the boundary points of a given image, from which we run Ripser for 0-th persistence barcodes. Hence, the first one uses a usual Vietoris-Rips filtration as in Part 1. The other one is to

[1] Note *Ripser* uses diameter as parameter, rather than a radius from each vertex.

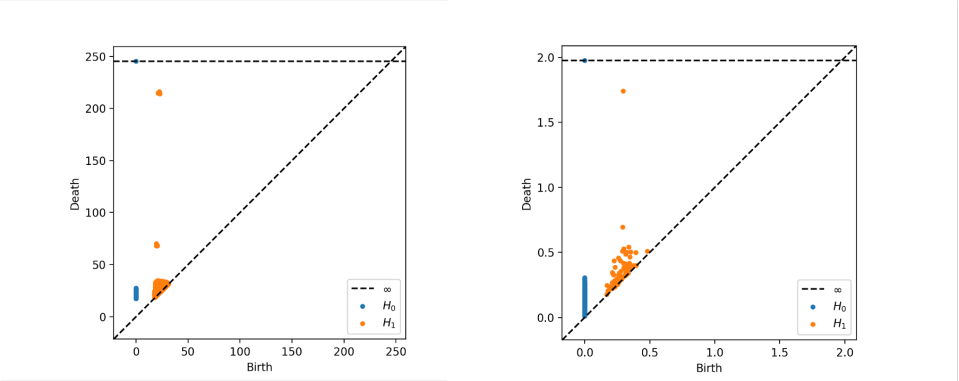

FIGURE 1. Persistence Diagrams for `octa.txt`(left) and `cylinder.txt`(right).

use a filter function motivated from [2, Section 5.1], while seeing the non-black pixels to be vertices and connecting edges between them if they have distance less than the given threshold. Hence, the second one uses a filtration by the sublevel sets generated by the filter function. In the following, we will present one by one in detail.

3.1. **Summary of Source code `part2(boundary).py`.** How this is implemented is not that far from `part1.py`. The only difference arises in the **pre-processing** data part, in which we are effectively extracting the boundary points from the given image. The rest of the process is just to feed those boundary points to Ripser, while giving `maxdim` parameter to be 0 to save resources.

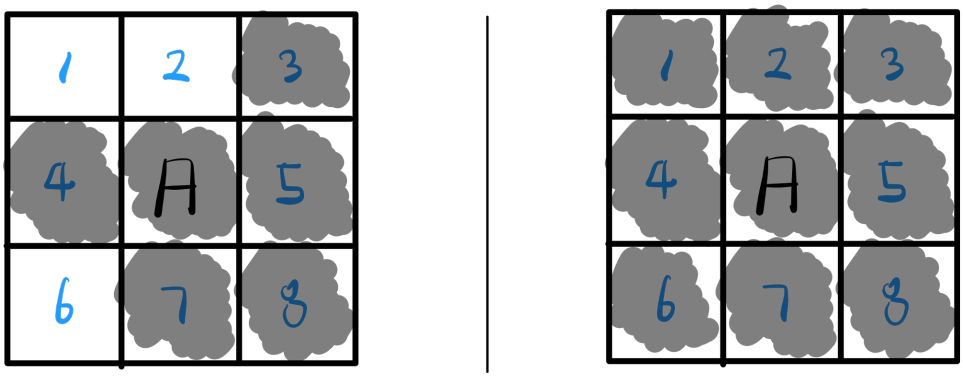

FIGURE 2. How to find a boundary point of an image. Compare the pixel A with its eight adjacent pixels. If any one of them (1–8) differs from the center pixel A, we conclude A is a boundary point. For example, A on the left is a boundary point, but A on the right is not a boundary point.

The idea of picking boundary points is to compare a given point with the eight adjacent pixels to it. (See Figure 2). Again, to reduce unnecessary calculations, we only pick the pixels of the form $(even, even)$, which is sufficient for persistence homology calculation since we are seeing all adjacent pixels to see if it is boundary points. Eventually, all the pixels of the image will be swept to be compared, so this does not make any serious underfitting error.

3.2. **Summary of Source code `part2(heightFtn).py`.** Here we do not use the Ripser to find the barcodes. Instead, we will make a filtration out of image using a filter function in [2, Section 5.1].

First, convert sample the non-black pixels to be vertices where the the points are sampled with $\lfloor \frac{\texttt{threshold}}{\sqrt{2}} \rfloor$-steps, where `threshold` is a lower bound for the distances between two vertices to form an edge. [2] Then sort the vertex set `vSet` in increasing order by the value of height function, where the height function $f : \texttt{vSet} \to \mathbb{R}$ is defined to be:

$$f(v) = \frac{1}{r}\langle v - b, d\rangle,$$

where b is the barycenter of the image, d is the unit direction vector and r is a normalizer. For this project, we chose b to be the center of the image, and r to be $\max(\texttt{height}, \texttt{width})$ of the image. While calculating the height function of each vertex `v`, **we set `v.birth` to be its value of the height function**. This is because we use the sublevel set of f to be our filtration, and it is exactly $f(v)$ when v is born in the sublevel set $f^{-1}((-\infty, f(v)])$.

After sorting `vSet`, by comparing the distances of all the pairs of vertices, we have the edge set `eSet` as well. To make use of our sorting on `vSet`, take consistent notation of edges $[u, v]$ where u is older than (or has the same age as) v. Note that each edge is born at the birth time of its younger incident vertex. However, since it is the edges that can possibly give death to the vertices, we proceed by taking a loop for `eSet`.

For each edge in `eSet`, we make use of the **union-find** structure [3] to see if each edge connects the same or different components. More precisely, as the one goes by *young dies first*, we have put the older one to parent of younger ones, so for each `union` operation, the root vertices are the ones that need their death time updated. To do this, say an edge $e = [u, v]$ connects two vertices u, v in different components, whose roots are x, y. Note v is younger than u, by our earlier sort of vertices and our construction of `eSet`. Then if we assume y is younger root than x, then grant y death by setting `y.death = v.birth`. If u, v were already in the same component, then do nothing and continue to the next edge. We set the default value for death to be `inf`, so the vertices whose death times are not updated will have `inf` death time.

[2] The reason why we divide by $\sqrt{2}$ is to avoid situation where one sampled vertex has another pixel of distance $\leq$ `threshold`, yet none of those nearness is realized as an edge – the underfitting of the connectedness of sampled vertices from the image.

[3] This part of using union-find is inspired from [1, Section VII.2]

After the loop is finished, we just collect all of the (birth,death) pairs of each vertex in `vSet` and summarize into files `XYZ.gif.txt`.

3.3. **Discussion on the Performance and Outcome.**

3.3.1. *Outcome.* For the `boundary`, all of the 0-th persistence barcodes have one of infinite length, which represents that the the Rips complex of each boundary point will be in fact connected to a single component. This was expected result by the construction of Rips complex. More interestingly, for the images like `butterfly-10.gif`, `cattle-3.gif`, and `horse-18.gif`, they generated barcodes of nontrivial length 2 (the gap for pixel sampling). This is because they had an interesting 'strips' on their image, so that they are included as boundary inside of the contour of image. Those strips formed a separate component at the first few stages, but they died not long after as they were near to the contour of the images.

When it comes to finding the number of components – finding the rank of 0th homology –, `heightFtn` did a better job than `boundary`. Compared to `boundary`, `heightFtn` extracted more than one barcodes of infinite length for `butterfly-10.gif`, `cattle-3.gif`, and `dog-7.gif` while maintaining one infinite barcode for `horse-18.gif`. This is because the "threshold" for connecting two different components was fixed for `heightFtn`, in contrast to `boundary` using the increasing threshold to connect every vertex in the end. It is interesting to note that it still gave one infinite barcode for `horse-18.gif`, and one possible reason for this is the strips on horse were too "narrow"(compared to the given threshold) so that they are regarded as a noise in the image, rather than counted as a separate component. However, for the highly sparse image like `dog-7.gif`, it generated too many barcodes of infinite length, which is expected by its sparseness, but not an optimal result.

3.3.2. *Performance.* Overall execution time was way faster for `boundary`, than `heightFtn`. It took 30.41 seconds for `boundary` to process the ten images, but it took 737.27 seconds for `heightFtn`, which is about 24 times longer. One reason to address this difference is that there are far less pixels to be considered in `boundary`, since we are only considering boundary points of an object. On the other hand, essentially the number of points in consideration for `heightFtn` is generally more than the number of boundary points, because now it counts the interior points as well.

However, if we normalize the time taken for each of image by dividing it by total time, more interesting pattern appears. See Table 1. The Figure 3 and 4 show the images that required `boundary` and `heightFtn` comparatively extra time to process, respectively. Here we summarize our main observations on the 'preference' of each program.

Observation. *`boundary` is not effective with the images with* ***scattered pixels****, while `heightFtn` is not effective with those with* ***condensed pixels****.*

Filtration	bat	beetle	butterfly	camel	cattle	crown	device	dog	horse	octopus	total
Boundary	4.1%	2.1%	**13.0%**	3.1%	**24.5%**	1.5%	5.8%	**38.9%**	4.5%	2.5%	30.41(s)
HeightFtn	**13.8%**	0.3%	3.6%	4.5%	**42.7%**	0.1%	**28.4%**	3.9%	1.8%	0.9%	737.27(s)

TABLE 1. Comparison of portion of time usage for each image for each of two filtrations. For brevity, we omitted the indices of each class of images. We highlighted with bold letters that show significant performance difference.

It is easy to see why this observation makes sense. The less the data points, the more effective both of the programs. Images with scattered pixels inevitably have more boundary points that may not be in the *contour* of the image, that is, the 'real' boundary of the topological space from which the points are sampled from. Therefore, the more data points will be fed to *Ripser* in `boundary` if we deal with images with scattered pixels. On the other hand, condensed pixels literally mean there are more points compared to those are scattered. Hence, there would be more vertices to be considered for the filtration in `heightFtn`. An interesting thing to note is that those scattered/condensed properties are not exclusive. That is, if there are non-negligible amounts of both scattered and condensed parts in the image(such as `cattle-3.gif` in Figure 3, 4), it will require both `boundary` and `heightFtn` to take more time.

FIGURE 3. Images with relatively scattered pixels among the image set. Note the middle image `cattle-3.gif` has both scattered and condensed part, but the scattered part(circled) has about the same size as the whole `butterfly-10.gif`.

Possibly ways to fix those issues for each program are:

- For `boundary`, we give more strict condition for pixels to be qualified as a boundary point. For example, instead of just checking their adjacent(=1-neighborhood) points, we can now require the points to satisfy some "cluster" condition as well, like there should be at least 40% of points from its "n-neighborhood"
- For `heightFtn`, we can vary the threshold for connecting edges between vertices, depending on the "density" of non-black pixels in sublevel sets. For example, in high-density sublevel sets we give

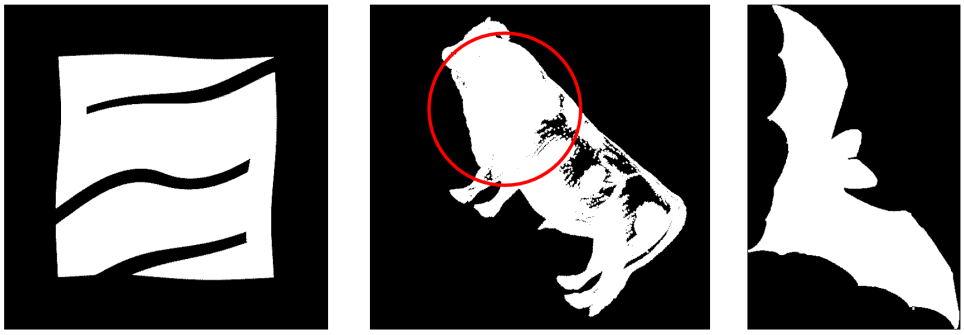

FIGURE 4. Images with relatively condensed images among the image set. Note the middle image `cattle-3.gif` has both scattered and condensed part, but the condensed part(circled) has about the same size as the half of `bat-19.gif`.

high threshold distance for two vertices to be connected, and in low-density ones, we lower the threshold distance.

References

1. Herbert Edelsbrunner and John Harer, *Computational topology: an introduction*, American Mathematical Soc., 2010.
2. Roland Kwitt, Christoph Hofer, Andreas Uhl, and Marc Niethammer, *Deep learning with topological signatures*, Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 1633–1643.