Computing Persistent Homology of a Simplicial Complex

Etta Rapp, Elisheva Sigfried

Topics in Topology: Computational Topology

Spring 2020 -- Final Project

Abstract

Persistent homology is a tool to measure the significance of topological features. This can be useful in applications such as recognizing and discounting the effects of noise in data. This paper will review the theory of persistent homology, specifically the persistent homology of a simplicial complex. It will then discuss our implementation of an algorithm to compute the persistent homology of a simplicial complex given its incidence matrix. We conclude with several examples implementing this algorithm and mention some applications of persistent homology.

Introduction

Persistent homology is a tool to characterize the topological features of a simplicial complex while tracking which features are most significant. Later in this paper we will review how this is technically computed, but we will first give a brief overview of what persistent homology means. We can think of the process of computing persistent homology as breaking down the simplicial complex into individual simplices and then building it back up step by step. We begin with a null simplex and then add simplices one at a time in order of increasing dimension. Each stage of this process yields a new subcomplex of the simplicial complex. This process of ordering simplices and creating subcomplexes is done through a filtration function. We calculate persistent homology by studying the homology of each subcomplex and computing the persistence of individual homology generators. These homology generators represent topological features of the simplicial complex, and their persistence is an indication of their importance in characterizing the shape of the simplicial complex. Through this process, we create a persistence diagram which represents the persistent homology of the simplicial complex.

Mathematical Background

The key to computing persistent homology is a filtration function. We begin computing persistent homology with a simplicial complex called K. The filtration function of K is defined as a monotonic function that maps each simplex in K to a real number ($f: K \to R$). The condition that f is monotonic means that the filter function value of a simplex will be no smaller than the filter function value of any of the simplex's faces (if σ_i is a face of σ_j with $i < j$ then $f(\sigma_i) \leq f(\sigma_j)$.) This holds true along the increasing chain of faces. (For a chain $(\sigma_1, \sigma_2, \cdots, \sigma_m)$, we have $f(\sigma_i) \leq f(\sigma_{i+1}), \forall\ 1 \leq i \leq m-1$ where m is the number of simplices in K.) This ordering of the simplices of K is called a compatible ordering, or simplex-ordering, of K with regard to f.

The filter function of K defines subcomplexes of K, with a subcomplex K_i defined such that $K_i = K(a_i) = f^{-1}(-\infty, a_i]$ for $a_i \in R$. We start with one simplex of K and add simplices one at a time according to their ordering as defined by the filtration function, yielding a subcomplex of K at each stage, until we add the final simplex and the result is K itself. Due to the monotonicity of the filtration function, simplices will be added in order of their dimensions. The resulting sequence of subcomplexes ($\emptyset = K_0 \subseteq K_1 \subseteq K_2 \subseteq \ldots \subseteq K_n = K$) when we apply the filtration function is referred to as the filtration of K. The existence of a filtration function is necessary in computing persistent homology.

Once we have a filtration of K, we can study the topological changes that occur as we progress through the sequence of subcomplexes. We do this by looking at the homology groups of each subcomplex. We keep track of when a new homology generator is born and when it dies as we go through the sequence of subcomplexes. A homology generator is born when it first appears and then dies when it merges with an older generator. (Because of the Elder Rule, when two generators merge, the generator with the later birth time merges into the generator with the earlier birth time.) We calculate the persistence of a homology generator as its death time minus its birth time. It is important to note that not all homology generators will die. If a homology generator does not die, it is called an

essential generator, and we define its time of death as infinity. Essential generators have an infinite persistence and are the nontrivial homology generators of *K*.

For each dimension of homology generators that we want to study, we examine the entire filtration of K, focusing on the homology generators of the dimension in question. For example, if we are studying the persistence of the 0-dimensional generators (connected components), we look at the homology of the filtration of *K* and track the persistence of the 0-dimensional generators. Then if we want to study the 1-dimensional generators (holes), we would follow the same steps, computing the homology of each subcomplex in the filtration of *K* and computing the persistence of the 1-dimensional homology generators.

Computation of persistence is technically accomplished through an inclusion map from subcomplex K_i to K_j, $i < j$, that induces a group homomorphism $F_p^{i,j}$ on the homology groups of each dimension p ($F_p^{i,j} : H_p(K_i) \rightarrow H_p(K_j)$). We say that a homology generator h is born at time $a_i = f(\sigma_i)$ if $h \in H_p(K_i)$ and $h \notin img(F_p^{i-1,i})$, and that it dies at time $a_j = f(\sigma_j)$ if $F_p^{i,j-1}(h) \notin img(F_p^{i-1,j-1})$ and $F_p^{i,j}(h) \in img(F_p^{i-1,j})$. Formally, the persistence of a homology generator h that was born with the creation K_i and died as the subcomplex went from K_{j-1} to K_j is $a_j - a_i$.

Once we have computed the persistent homology of a simplicial complex, there are two main ways to represent it: a persistence diagram and a persistence barcode. In this paper we will mainly discuss the persistence diagram. A persistence diagram is a graph where one point is plotted to represent each homology generator that resulted from the filtration. The *x* and *y* coordinates of each point represent its birth time and death time respectively. (Recall that the birth time of a generator born with the creation of subcomplex K_i is a_i such that $a_i = f(\sigma_i)$ and $K_i = f^{-1}(-\infty, a_i]$.) The persistence of a homology generator is the vertical (or horizontal) distance from the point to the diagonal $y = x$. Points that are further from the diagonal of a persistence diagram represent homology generators that are

more significant in a simplicial complex. A persistence diagram either encodes the persistence of homology generators of just one dimension, with a separate diagram for each dimension, or it can encode the persistence of homology generators of all dimensions using different symbols to represent homology generators of different dimensions.

Alternatively, persistent homology can be represented in a persistence barcode. In a persistence barcode, the x axis corresponds to time, and each homology generator is represented as a bar on the graph extending from its time of birth to its time of death. In a persistence barcode, the persistence of a homology generator is the length of its bar. Generators that have longer bars thus have a greater persistence and are more important in the topological structure of the simplicial complex. Similarly to persistence diagrams, persistence barcodes can either encode homology generators of a specific dimension or can plot all dimensions on one plot using different symbols.

We can compute the persistent homology of a simplicial complex through an algorithm that takes as input an incidence matrix that contains all the boundary matrices of the simplicial complex. This matrix is built by listing all simplices of K in increasing order of dimensions along both the columns and rows of a matrix. This ordering of simplices is the ordering that results from applying the filtration function to K. We set the $[i, j]$ position in the matrix to 1 if the simplex σ_i represented by row i is a codimension one face of of simplex σ_j represented by column j. Otherwise, the entry in position $[i, j]$ is 0. For example, the incidence matrix of an unfilled triangle with vertices a, b, and c is represented in Figure 1.

	a	*b*	*c*	*ab*	*ac*	*bc*
a	0	0	0	1	1	0
b	0	0	0	1	0	1
c	0	0	0	0	1	1
ab	0	0	0	0	0	0
ac	0	0	0	0	0	0
bc	0	0	0	0	0	0

Figure 1

The Persistent Homology Algorithm and our Code

We now describe the algorithm developed by Edelsbrunner *et al.* that computes the persistent homology of a simplicial complex from its incidence matrix. This algorithm reduces the incidence matrix into a reduced form called R. We can then derive the

homology and persistent homology of the associated simplicial complex from *R*. Our code implements this algorithm and displays its results.

The incidence matrix of a given simplicial complex is created as described above. In summary, we first list all the simplices in our simplicial complex in order of the results of applying the filtration function to them. Each row and each column in the incidence matrix corresponds to the simplex in the list of simplices that has the same index as the row or column (see Figure 1). The incidence matrix is populated with 0's and 1's; we refer to the row index of the lowest "1" in a column as the "lowest 1" of the column. If a column contains only zeros, its lowest 1 is undefined.

The reduced matrix R is defined by the property that no two columns in R have the same lowest 1. In order to compute R, the algorithm goes through the columns of the matrix from left to right. For each column, as long as there is a column to the left of it with the same lowest 1 as it, we add that column to it, repeating until there is no such column. After going through all the columns of the matrix, the result is the reduced matrix R in which all lowest 1's are unique or undefined.

Our code implements the algorithm to reduce the incidence matrix to *R* in a few steps. First, we define a function called `getLows` that takes as input a matrix and returns a list of its lowest 1's. This function's running time is $O(n^2)$: for each of n columns, it searches the column from bottom to top until it finds a one. If the lowest 1 of a column is undefined, we set it to `None` to increase the clarity and efficiency of our code. We also define a helper function `equalLow` which takes as input a list of lows and an index *j* and returns the index of a column to the left of *j* which has the same lowest 1 as *j*. If there is no such column, it returns None. This function runs in O(n), since it scans through the list of lowest 1's at most once.

We then define our function that reduces the matrix. This function, called `REDUCE`, takes as input an incidence matrix and returns a reduced matrix *R*. The algorithm first calls `getLows` on the input matrix. It then runs left to right across the columns, and for each column *j*, as long as the `equalLow` of *j* is not None, it adds the column with an equal lowest 1 to *j* using modular arithmetic with modulus two. Our code updates the lowest 1 of *j* as it performs this addition, adding the columns from bottom to top and noting when it first encounters a 1. This does not add to the time complexity, and is faster than calling `getLows` on the matrix after each iteration of adding two columns. The overall time complexity of the `REDUCE` function is $O(n^3)$.

Once we have this reduced matrix, we can derive a lot of information about our simplicial complex from it. The number of zero columns corresponding to p-simplices is z_p, the rank of the p-cycles of the simplicial complex, and the number of non-zero columns corresponding to (p+1)-simplices is b_p, the rank of the p-boundaries of the simplicial complex. From this information we can calculate the Betti numbers of the simplicial complex since the p-th Betti number is $z_p - b_p$. Thus we can derive the homology of a simplicial complex from its reduced matrix R.

Our code implements this operation, finding the p-th Betti number, with a function called `computeBetti`. In order to compute Betti numbers, in addition to the incidence matrix, our algorithm needs to know which columns in the incidence matrix correspond to what simplices of which dimension. We therefore create a list of the ranges of indices of columns representing simplices of each dimension. Our `computeBetti` function takes this list as input along with the reduced matrix. `computeBetti` calls `getLows` on the reduced matrix, and sums up the zero and non-zero rows for each dimension. Non-zero rows are rows whose lowest 1 is defined, zero-rows are rows whose lowest 1 is None. From z_p and b_p, `computeBetti` computes the Betti number and then returns z_p, b_p, and the Betti numbers for each dimension. This function's run time is O(n).

Not only does the reduced incidence matrix of a simplicial complex tell us its homology, it also encodes information about the complex's persistent homology. There are two different cases of when we add a point to a persistence diagram. If row *i*, corresponding to a p-simplex, contains the lowest 1 of column *j*, corresponding to a (p+1)-complex, then (a_i , a_j) is a point in the p-dimensional persistence diagram of the simplicial complex. This point represents a homology generator that is born at time a_i and dies at time a_j . Additionally, if column *i* corresponding to a p-simplex is a zero-column but row *i* does not contain the lowest 1 of any column, then (i, ∞) is a point in the persistence diagram of the simplicial complex. This point corresponds to an essential generator born at time a_i .

These rules enable us to generate a list of the points in the persistence diagram of a simplicial complex from its reduced matrix. Our code does this in a function called `persistence`. The function takes as input a dimension *p*, the reduced matrix *R*, and the list of which columns correspond to simplices of which dimension. It calls `getLows` on the matrix, and then it finds all the points in the persistence diagram for the given dimension. For non-essential classes, the code goes through all rows corresponding to p-simplices. If a row *i* contains a lowest 1, we find the column *j* that it is the lowest 1 of, and add (i+1, j+1) to the list of points in the p-dimensional persistence diagram. (We add one to *i* and *j* because the indexing of rows and columns are zero-based but the indices in the persistence diagram are not.) For essential classes, for every zero column *i* we check if the *i*-th row of the matrix contains the lowest 1 of a column. If it does not, we add (i+1, INF) to the list of points. We use the string "INF" as a placeholder to represent infinity. The `persistence` function runs in $O(n^2)$ time and returns a list of points.

In order to compare persistence diagrams, we compute a norm for each persistence diagram. This enables us to summarize an entire p-dimensional persistence diagram in one number which represents the significance of the features of dimension p in the corresponding simplicial complex. To compute the norm, we first draw a vertical and a horizontal line from each point in the persistence diagram to the diagonal of the diagram.

These two lines along with the diagonal form a triangle for each point, whose top vertex is the point and whose base is the diagonal. We then sum up the areas of these triangles to yield the norm. This norm is representative of the significance of features in a simplicial complex because if a generator has a greater persistence, it is farther from the diagonal of the persistence diagram, so its triangle will have a greater area.

To implement this calculation in our code, we simply define a function called `norm` that computes this metric. The `norm` function takes as input a list of the points in a persistence diagram, which is the output of the `persistence` function. For each point in the list, the function finds the difference between the two coordinates of the point, squares it, and divides it by two. This yields the area of the described triangle because the absolute value of the difference between the two coordinates is equal to both the vertical and the horizontal distance from the point to the diagonal, which is both the base and the height of the triangle whose area we are computing. For a point whose second coordinate is INF, we replace INF on a case by case basis with a value that is larger than all the finite coordinates of all the points in the diagram. The `norm` function returns the sum of these triangle areas for all of the points in the input list.

We include in our code a simple plot function to plot persistence diagrams. The plots include a diagonal and a scatterplot of the points returned by the `persistence` function. These plots place points whose second coordinate is INF along the top border of the plot.

Computer Experiments and Results

We ran our code on a few examples to test it and demonstrate what it does. Our `main` function calls `REDUCE`, `computeBetti`, `persistence`, `norm`, and `plot` functions on four examples: an unfilled triangle, a filled triangle, an unfilled tetrahedron, and a filled tetrahedron. We replace INF values with 20 in both plotting and computing the norm of persistence diagrams because it is larger than all of the birth and death times in our

examples. This isn't a perfect solution, but it is the simplest way to handle infinite values in this case.

The output of our code is as expected. In every example 0-th Betti number is 1 since there is one connected component. The 1st Betti number is 1 for the unfilled triangle and 0 otherwise. The 2nd Betti number is 1 for the unfilled tetrahedron and 0 otherwise. Higher Betti numbers are all 0 since all of our examples are all in 3-dimensional space and therefore have no higher dimensional homology generators. Our persistence diagrams are also as expected, see the output of our code for more details.

Conclusion

In recent years, topology, which has been considered a purely theoretical subfield of mathematics, has been found to have many real world applications. These applications range from biology to finance to computer vision. Because real world data is almost never as nice and theory, persistent homology is especially important in discounting noise and other external factors when studying trends in data. Persistent homology is also particularly relevant because it can be calculated efficiently by a computer, as we have demonstrated in this paper.

References Cited

"Chapter VII.1" *Computational Topology: an Introduction*, by Herbert Edelsbrunner and John Harer, American Mathematical Society, 2010, pp. 149–156.

Gidea, Marian. "Topics in Topology". Stern College, Yeshiva University. Spring 2020. Lecture notes and lecture.

D. Freedman and C. Chen. "Algebraic Topology for Computer Vision". July 11, 2010.

H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. Discrete and Computational Geometry, 28(4):511–533, 2002.

(This is the source of the algorithm)