Senior Thesis in Mathematics

Exploring Methods of High-Dimensional Data Analysis

Author:
Victor de Fontnouvelle

Advisor:
Dr. Vin De Silva

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

December 11, 2019

Abstract

We will explore various methods of analyzing high-dimensional data. We'll investigate techniques that make inferences about the underlying structure of the data, cluster the data, or reduce the dimension of the data. We'll explain these methods, and apply them to real-world datasets.

Contents

Chapter 1

Introduction

1.1 Inference About the Structure of the Data

Cohomology analysis and Helmholtz decomposition provide insight on the structure of the data. Cohomology analysis provides a broader framework for detecting clusters, holes, and higher-order features within the data. Helmholtz decomposition is a specific application of cohomology analysis which seeks to find an ordered list which best accounts for a weighted graph.

1.2 Clustering

Mapper clusters the data, by mapping clusters of points onto intervals on the real line using a filter function, and connecting overlapping clusters.

1.3 Dimensionality Reduction

Calculating the eigenvalues of the laplacian matrix reduces the dimension of the data. The laplacian is a symmetric matrix encoding the pairwise distances between points, normalized by row. This matrix can be thought of as a linear operator encoding heat flows. Given an input of initial temperatures, it outputs the changes in temperatures after one time step. Eigenvectors corresponding to low eigenvalues thus correspond to stable temperature configurations. The eigenvectors are perpendicular, and thus eigenvectors corresponding to slightly higher eigenvalues often capture geometric structure existing in the data. Additionally, nearby points will have similar values in the eigenvectors, and thus the eigenvectors are also a useful tool for reducing the dimension of the data.

Chapter 2

Review of Literature

Papers by Gunnar Carlsson [1] and Vin de Silva [3] explain the intuition behind cohomolgy analysis, and provides several examples. Carlsson [2] and deSilva [3] also describe the algorithm used to compute cohomology, which will be useful if I choose to implement it.

Curto [4] and Ulmer [5] both provide various examples of the uses of homology analysis, both in detecting underlying structure, and in providing fingerprints that identify different phenomena. Curto analyzes a dataset representing connection strengths between neurons in rats responsible for spatial recognition. Curto first keeps a certain proportion of edges ρ s.t. $0\langle\rho\langle1$, keeping those edges which are the strongest. Curto then runs cohomology analysis on this graph to determine the Betti curve. Curto determined that the Betti curve obtained from spatially organized neurons is different than the Betti curve that would be obtained from neurons with random structure. Cohomolgy analysis thus provides a method for detecting spatial neurons. Ulmer uses cohomology analysis to evaluate two different models of social interaction for aphids roaming in a dish. Standard measures that compare the models to real data include angular momentum, and average distance to closest neighbor. Ulmer found that cohomology analysis provided an equally strong measurement for assessing model accuracy.

A paper by Jiang [6] explains the Helmholtz decomposition that converts ranked data into ordinal data. It provides both the algorithm for the decomposition, as well as three examples of its use. Jiang uses Helmholtz decomposition to rank movies. Most users rate several movies, so each time a user rated two movies, this introduced an edge from one movie to the other indicating the user's preference. Jiang used Helmholtz decomposition to create an absolute index of currency values based off trading rates. Jiang also used Helmholtz decomposition to rank websites based off of connection strengths. Jiang found that Helmholtz decomposition performed as well as some of the standard methods for website ranking, indicating that it is a useful tool.

Carlsson [1] also explains the Mapper tool, and provides examples of its use. I have already used the algorithm described in this paper to analyze several datasets.

Singh [7] describes the intuition behind the laplacian analysis, which I have also implemented.

Chapter 3

Helmholtz Decomposition

3.1 Explanation of Helmholtz Decomposition

3.1.1 Motivation

Ranking various alternatives based off of comparative data is a common problem. Examples include ranking movies based off of user reviews, ranking tennis players based off of individual match results, and ranking Google search results. In most instances, there exists no ranking that is consistent with every observation. Helmholtz decomposition finds the ranking that best accounts for the observed data. Additionally, the ranking produced by Helmholtz decomposition is composed of real-valued numbers, providing more precise information about how close certain elements are than a simple ordinal ranking would.

The matrix $\bar{Y}$ represents all pairwise rankings, and the matrix W represents the weights of each pairwise ranking. Some examples will help to clarify.

Imagine we would like to rank four tennis players: Ann, Bob, Carl, and Dan. We are given the following data about match outcomes:

Winner	Loser
Ann	Bob
Ann	Carl
Ann	Carl
Carl	Ann
Bob	Carl
Bob	Carl
Carl	Bob
Bob	Dan
Dan	Bob
Carl	Dan

Each match counts as one point for the winner, and negative one points for the loser. We fill in the entries of $\bar{Y}$ by averaging the outcome of the relevant matches for each entry. For example, we set Y_{20}, corresponding to matches between Ann and Carl to $\frac{1+1+(-1)}{3} = 0.33$, because Ann beat Carl twice and Carl beat Ann once. We fill in the entries of W by counting the number of matches for each entry. For example, $w_{02} = 3$, as Ann and Carl played three matches together. The full matrices $\bar{Y}$ and W are as follows:

$$\bar{Y} = \begin{bmatrix} 0 & -1 & -0.33 & 0 \\ 1 & 0 & -0.33 & 0 \\ 0.33 & 0.33 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$w = \begin{bmatrix} 0 & 1 & 3 & 0 \\ 1 & 0 & 3 & 2 \\ 3 & 3 & 0 & 1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$$

Now imagine we'd like to rank three movies, A, B, and C, based off the reviews of users X, Y, and Z. Suppose the users have provided the following ratings, on a scale of 1 to 5:

	A	B	C
X		4	2
T	5	3	4
Z	4		4

We fill in $\bar{Y}$ to contain the average user preference corresponding to each entry. For example, Y_{12}, the entry corresponding to movies B and C, has value $\frac{2+(-1)}{2}$, as user X had a preference of magnitude 2 for movie B whereas user Y had a preference of magnitude -1. We fill in W to contain the number of users who reviewed both movies corresponding to the relevant entry. For example, $Y_{12} = 2$ because two users reviewed both movies B and C. We thus have

$$\bar{Y} = \begin{bmatrix} 0 & 2 & 0.5 \\ 2 & 0 & 0.5 \\ 0.5 & 0.5 & 0 \end{bmatrix}$$

$$w = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix}$$

More generally, we consider any skew-symmetric matrix $\bar{Y} \in M_n$ (recall that $A \in M_n$ is skew-symmetric when $A^T = -A$) along with a symmetric matrix W. We would like to find a vector $s \in M_{nx1}$ that minimizes $||\text{grad}s - \bar{Y}||_{2,W}$. Note that the norm is computed with respect to the inner product space defined by W. More precisely, $\langle X, Y\rangle_W = \Sigma_{\{i,j\}\in E} w_{ij} X_{ij} Y_{ij}$.

3.1.2 Intuition

The Hodge Decomposition Theorem provides that $\bar{Y}$ can be expressed as the sum of three orthogonal components:

1. The gradient flow G
2. The curl flow C
3. The harmonic flow H

The gradient flow G represents a comparison matrix that corresponds directly to a vector v where each item is assigned a real value. G is the gradient of v, thus each entry is computed as $G[ij] = v[j] - v[i]$.

The harmonic flow H represents a ranking that is curl-free and divergence-free. This means that any three items in H will have pairwise rankings that are logically consistent. Specifically, $H[ij] + H[jk] + H[ki] = 0, \forall i, j, k$.

Because $\text{div}(H) = 0$, H contains plausible values in the sense that it could have been produced by real-world data. A harmonic flow thus indicates that there are cycles in the graph with more than three edges, where the edge weights don't sum to zero.

The curl flow C is the image of curl*, the adjoint of the curl. Nonzero values of C thus indicate cycles of length three whose edge weights don't sum to zero.

The gradient flow provides the ranking that minimizes the least-squared residual, while the harmonic and curl flows characterize the residual. Specifically, $\bar{Y} - H = G + C$. A large curl flow indicates that the ordering of items ranked closely together is unreliable, while a large harmonic flow indicates that the ordering of items ranked further apart is unreliable. For example, if the curl flow is large but the harmonic flow is small, this indicates that the ranking is valid at a larger scare, but that the specific ordering of closely-ranked alternatives isn't very precise.

3.1.3 Representation of Matrices

It is necessary to have matrix representations of δ_0 (grad), δ_1 (curl), δ_0^*, and $delta_1^*$ in order to compute the gradient, curl, and harmonic flows. Throughout this section, f, α, and A refer to maps from edges, vertices, and triples into $\mathbb{R}$, respectively.

The gradient operator assigns values to edges based on the difference of the values of the endpoints. Specifically, $\alpha((a, b)) = -f(a) + f(b)$. For example, when the graph has four vertices and all edge weights are nonzero, the matrix representation of the gradient is

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} ab \\ ac \\ ac \\ bc \\ bd \\ cd \end{bmatrix}$$

We can verify the correctness of this matrix by examination. For example, the first row corresponds to the edge (a, b) and thus we want $\alpha((a, b)) = -f(a) + f(b)$. Indeed, the value -1 in the first column corresponds to a, and the value $+1$ in the second column corresponds to b. If any of the edge weights are zero, we remove the corresponding row from the gradient matrix. For example, if $W((a,b))=0$, we would remove the first row from the matrix above.

The curl operator assigns values to triples based off the values of edges. Specifically, $A((a,b,c)) = \alpha(b,c) - \alpha(a,c) + \alpha(a,b)$. For example, when the graph has four vertices and all edge weights are nonzero, the matrix representation of the curl is:

$$\begin{bmatrix} 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} ab \\ ac \\ ad \\ bc \\ bd \\ cd \end{bmatrix} = \begin{bmatrix} abc \\ abd \\ acd \\ bcd \end{bmatrix}$$

To check correctness, we examine the triple (a,b,c). We would want to assign this triple the value $\alpha((b,c)) - \alpha((a,c)) + \alpha((b,c))$. We can verify that indeed the row corresponding to edge (a,b) has values $+1$, -1, and $+1$ at the columns corresponding to the edges (b,c), (a,c), and (b,c). If any of the edge weights are zero, we remove the corresponding column from the matrix. If any of the triples contain an edge with zero weight, we remove the corresponding row from the matrix.

We now compute δ_0^* and δ_1^*. Let $M_{\delta_0^*}$ and M_{δ_0} denote the matrix representations of δ_0^* and δ_0, respectively. By definition of the adjoint operator, we must have that:

$$\langle \delta_0 f, \alpha \rangle = \langle f, \delta_0^* \alpha \rangle \tag{3.1}$$

$\delta_0 f$ and α are in C^1, thus their inner product includes multiplication by edge weights, and is computed as $\sum_{i,j} (\delta_0 f)_{ij} \alpha_{ij} W_{ij}$. f and δ_0^* are in C^0, thus their inner product does not include edge weights. So in order for 3.1 to be valid, we need δ_0^* to include multiplication by edge weights. Then multiplying the rows of M_{δ_0} by the corresponding edge weights and transposing the result will yield $M_{\delta_0^*}$. Specifically, $M_{\delta_0^*} = (DM_{\delta_0})^T$ where $D = \text{diag}(W_{ij})$ for all nonzero edges (i,j).

Similarly, in the case of the curl operator we must have:

$$\langle \delta_1 \alpha, A \rangle = \langle \alpha, \delta_1^* A \rangle \tag{3.2}$$

This time α and $\delta_1^* A$ are in C^1 while $\delta_1 \alpha$ and A are in C^2. Thus only the inner product $\langle \alpha, \delta_1^* A \rangle$ includes multiplication by edge weights. Then dividing the rows of M_{δ_1} by the corresponding edge weights and transposing the result will yield $M_{\delta_1^*}$. Specifically, $M_{\delta_1^*} = (M_{\delta_1} D')^T$ where $D' = \mathrm{diag}(\frac{1}{W_{ij}})$ for all nonzero edges (i, j).

3.1.4 Computation of the Gradient, Curl, and Harmonic Flows

Having defined the necessary matrix representations, we are ready to compute the three flows, which we denote G, C, and H. The gradient flow is the orthogonal projection of Y onto $\mathrm{im}(\delta_0)$. So $G = \delta_0(\delta_0^* \delta_0)^+ \delta_0^* Y$, where $^+$ denotes the Moore-Penrose pseudo-inverse.

The curl flow is the orthogonal projection on to $\mathrm{im}(\delta_1^*)$. We thus hope to find $A \in C^2$ which is the least squares solution to $Y = \delta_1^* A$. Note that this equation is in the inner product space C_1 which includes multiplication by edge weights. To solve this equation in the standard inner product space, it is equivalent to multiply both sides by $\sqrt{D}$, and solve $\sqrt{D} Y = \sqrt{D} \delta_1^* A$, where $D = \mathrm{diag}(W_{ij})$. Then $C = \delta_1^* A$.

The harmonic flow is $\ker \delta_0^* \cap \ker \delta_1 = \ker \Delta_1$ where $\Delta_1 = \delta_1^* \delta_1 + \delta_0 \delta_0^*$. Then $D\Delta_1 = \Delta_1^T D$. So $D^{1/2} \Delta_1 D^{-1/2} = D^{-1/2} \Delta_1^T D^{1/2}$. So $S = D^{1/2} \Delta_1 D^{-1/2}$ is symmetric, and we can compute its pseudo-inverse. We thus have that the matrix representation of the projection in to $\ker \Delta_1$ is $P = D^{-1/2}(S^+ S) D^{1/2}$. So $H = (I - P)Y$.

3.2 Application of Helmholtz Decomposition

We apply Helmholtz decomposition to a dataset of matches from all major professional tennis tournaments in 2017. There are 528 players and 3831 matches. Each match functions as a comparison between two players. The entries in the $\bar{Y}$ matrix are computed as the fraction of games the winner won minus 0.5. For example, if player B won 0.7 of the games in a match against player A, the entry at index (A, B) in $\bar{Y}$ would be $0.7 - 0.5 = 0.2$. We experiment with four different weighting methods:

1. "Uniform" : All matches are weighted equally

2. “Sets” : A match is weighted by the number of sets played
3. “Games” : A match is weighted by the number of games played
4. “Tourney” : A match is weighted more if it is played in a later round in the tournament. Final-round matches have a weight of 1, semifinal matches a weight of 0.5, quarterfinal matches a weight of 0.25, and so on.

We take a subset of the data consisting of only those matches involving the top 50 players as chosen using “uniform” weighting. We apply Helmholtz decomposition to this smaller dataset using each of the four different weighting methods. We then compare the three flows for each weighting method, where the flows are represented as their fraction of the whole matrix Y, using the Frobenius norm with respect to the C^1 inner product $\langle A, B\rangle = \sum_{i,j} A_{ij}B_{ij}W_{ij}$. The results are as follows:

Weighting Method	Gradient	Curl	Harmonic
Uniform	0.368911	0.630049	0.001040
Sets	0.361733	0.637242	0.001025
Games	0.351814	0.646887	0.001299
Tourney	0.424746	0.574440	0.000814

Note that all weighting methods produce similar results except for “tourney,” in which case the gradient flow is higher and the curl flow lower. This suggests that “tourney” weighting allows for the most consistent ranking of players. This is not intuitive: we would not expect a match to be more reflective of the payers’ true skill just because it occurs in a later round.

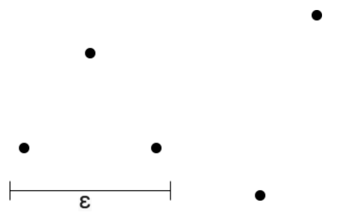
Chapter 4

Cohomology Analysis

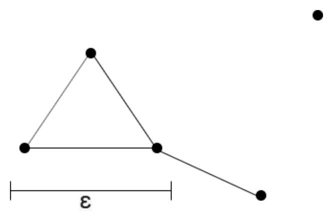
4.1 Explanation of Cohomology Analysis

Cohomology analysis allows us to understand the structure of high-dimensional data. For two-dimensional data, clusters and holes are very easy to spot with the human eye. However, as the number of dimensions grows large, clusters and holes are impossible to visualize, and are often lost when common dimension reduction techniques such as Principle Components Analysis are applied. Cohomology analysis provides a method to detect such features without losing any information.

It works as follows: we start off with a parameter $\epsilon = 0$. We gradually increase it, connecting any two points whose distance is less than ϵ (using whatever distance metric the user has provided). Consider this example:



Dataset of Points



After Connecting

We also connect sets of points, where the set is any size. For example, a set of two points is a line, three points is a triangle, and four points is a tetrahedron. These sets of connected points are called simplices, and the set of all simplices for a given value of ϵ is called a simplicial complex. An n-simplex is a simplex containing $n + 1$ points.

There are two common types of simplicial complexes:

- Cech complex: connect a set of points $P = \{p_1, .., p_n\}$ whenever $\cap_{i=1}^{n} B_\epsilon(p_i) \neq \emptyset$.

- Vietoris-Rips (Rips) Complex: Connect a set of points $p = \{p_1, ..., p_n\}$ whenever $B_\epsilon(p_i) \cap B_\epsilon(p_j) \neq \emptyset \forall i, j \in [1, n]$. Informally, we conenct a set of points whenever their pairwise distances are less than ϵ.

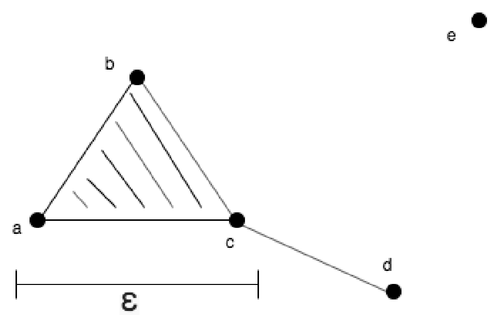
In general, given a dataset of points $P = \{p_1, .., p_n\}$, a simplicial complex $S = \{s_1, .., s_m\}$ is a collection of simplices, where each simplex s_i is a subset of $s_justified$, and where all subsets of $s_justified$ are also included in S.

We next consider the boundary operator δ_n. We are given a subset of faces $Q = \{f_1, .., f_m\}$ along with arbitrary values $V = \{v_1, .., v_m\}$. Each face $f_i = \{p_{i_1}, .., p_{i_n}\}$ is a subset of P of size n. If there exists a simplex S of size $n + 1$, where all subsets of size n correspond to faces, then the boundary operator evaluates the simplex of size $n + 1$. If there is no such simplex, the boundary operator is zero.

The boundary of a simplex $s = \{p_1, .., p_n\}$ is computed as

$$\Sigma_{i=1}^{n} (-1)^{i-1} value(\{p_1, ..., p_{i-1}, p_{i+1}, p_n\})$$

where $value(P)$ is the value of the face whose set of points is P. An example will help clarify the concept of the boundary operator. Suppose that we have the following simplicial complex:

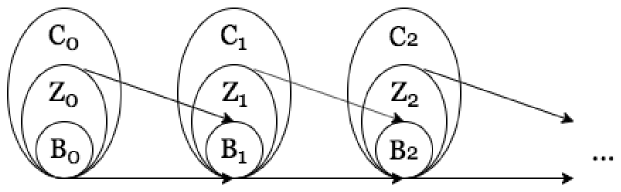


Let $Q = \{\{c\}, \{d\}, \{e\}\}$. Then Q contains the 1-simplex $\{c, d\}$, thus $\delta_0(Q) = (-1)^0 value(\{c\}) + (-1)^1 value(\{d\}) = v(c) - v(d) = 2 - 3 = -1$.

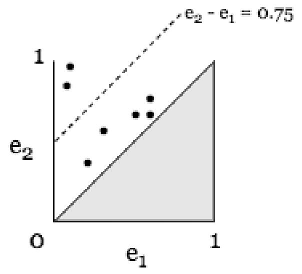
Let $Q' = \{\{a, b\}, \{a, c\}, \{b, c\}\}$, and $v = \{1, 2, 3\}$. Then Q' contains the 2-simplex $\{a, b, c\}$, so $\delta_1(Q') = value(\{a, b\}) - value(\{a, c\}) + value(\{b, c\}) = 1 - 2 + 3 = 2$.

n-cocycles are elements of $\ker(\delta_n)$. They correspond to values that are "consistent" with faces in the next dimension. n-coboundaries are values derived from the previous dimension. The coboundaries will thus always be "consistent," and so are a subset of the coboundaries.

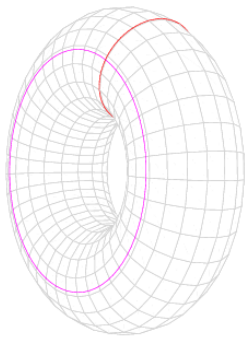
B_n commonly refers to the set of n-coboundaries, Z_n to the set of n-cocycles, and C_n to all other sets of values that are neither coboundaries nor cocycles. This map is a helpful visualization:



As we increase ϵ, the number of cocycles will change. In the next section we'll describe an algorithm for keeping track of the cocycles. For each cocycle, we can construct an interval $[\epsilon_1, \epsilon_2]$, where ϵ_1 is the value of ϵ when the cocycle was created, and ϵ_2 is the value of ϵ when the cocycle was destroyed. Persistent cocycles have a large value for $\epsilon_2 - \epsilon_1$. We can plot all cocycles to visualize which ones are persistent. For example, if ϵ ranges from 0 to 1, we might get the plot:



By setting an arbitrary threshold for $\epsilon_2 - \epsilon_1$, we can count the number of "significant" n-cocycles. In this case, we set the threshold to 0.75, and get 2 significant n-cocycles. In general, the number of significant n-cocycles is called the n^{th} Betti number. Consider the example of a torus:



It has one connected component, thus its 0th Betti number is 1. It has two holes (in red), so its 1st Betti number is 2. It has one cavity (the interior of the torus), thus its 2nd Betti number is 1. If we were to sample points from the torus, run our algorithm, and choose appropriate cutoffs for $\epsilon_2 - \epsilon_1$, we would expect to see exactly these results.

4.2 Explanation of Algorithm

The inputs to our algorithm are:

1. $s_1, s_2 ... s_n$: A list of simplices in the order that they are added
2. $t_1, t_2 ... t_n$: The time at which each simplex arrived

Our algorithm will output a list of all cocycles that existed, along with the times at which they were created and destroyed. Recall that cocycles which persist for a long time indicate important features of the data.

The algorithm maintains the following data at all times:

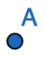
1. I: A list of the indices of live cocycles
2. C: A list of all cocycles, where each cocycle includes the value of each simplex of equal dimension to the cocycle.
3. T: The time of creation and destruction for each cocycle

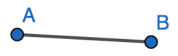
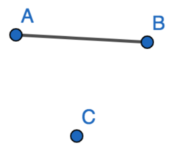
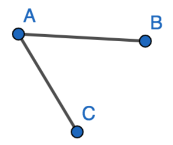
The algorithm works as follows: for each $i \in [1, n]$, we extend all live cocycles in C with dimension equal to s_i by assigning s_i a value of 0. Next, we compute the coboundary of each cocycle. There are two cases:

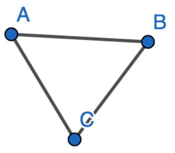
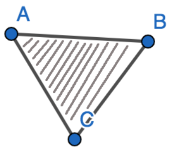
1. If all coboundaries are 0, we add a new cocycle c_i to C with a value of 1 for s_i and 0 for all other simplices of the same dimension as s_i. We add i to I, and set $T[i][0] = t_i$, recording the time of creation of c_i.

2. Some element of a coboundary c_x is nonzero. In this case we remove from C the last cocycle c_x with a nonzero coboundary. We also remove x from I. We adjust all other live cocycles c_y with a nonzero coboundary by setting $c_y = c_y - \frac{b_y}{b_x} c_x$ where b_y and b_x are the coboundaries of c_y and c_x, respectively.

Here is a simple example of the algorithm running. For clarity, we only include the live cocycles in C. Note that the actual algorithm would want to store all cocycles, so that upon termination it can describe all cocycles and their associated time of creation and destruction.

i	Current Simplices	Data Updates
0		$I = \{\}$ $C = []$
1		C Extension: $[]$ Coboundaries: $[]$ <u>New Data</u> $I = \{1\}$ $C = [[a : 1]]$
2		C Extension: $[[a : 1, b : 0]]$ Coboundaries: $[[]]$ <u>New Data</u> $I = \{1, 2\}$ $C = [[a : 1, b : 0],$ $[a : 0, b : 1]]$

3		C Extension: $[[a:1, b:0],$ $[a:0, b:1]]$ Coboundaries: $[[ab:-1]$ $[ab:1]]$ <u>New Data</u> $I=\{2\}$ $C=[[a:1, b:1]]$
4		C Extension: $[[a:1, b:1, c:0]]$ Coboundaries: $[[ab:0]]$ <u>New Data</u> $I=\{1,4\}$ $C=[[a:1, b:1, c:0],$ $[a:0, b:0, c:1]]$
5		C Extension: $[[a:1, b:1, c:0],$ $[a:0, b:0, c:1]]$ Coboundaries: $[[ab:0, ac:-1]$ $[ab:0, ac:1]]$ <u>New Data</u> $I=\{1\}$ $C=[[a:1, b:1, c:1]]$

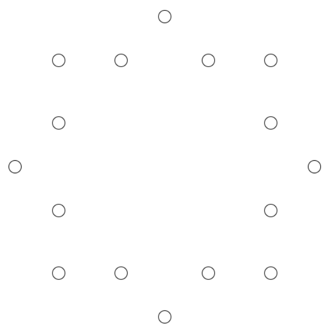
6		C Extension: $[[a:1, b:1, c:1]]$ Coboundaries: $[[ab:0, ab:0, bc:0]]$ <u>New Data</u> $I = \{1, 6\}$ $C = [[a:1, b:1, c:1],$ $[ab:0, ac:0, bc:1]]$
7		C Extension: $[[a:1, b:1, c:1],$ $[ab:0, ac:0, bc:1]]$ Coboundaries: $[[ab:0, ac:0, bc:0]$ $[abc:1]]$ <u>New Data</u> $I = \{1\}$ $C = [[a:1, b:1, c:1]]$

The algorithm will output the time of creation and destruction for each cocycle:

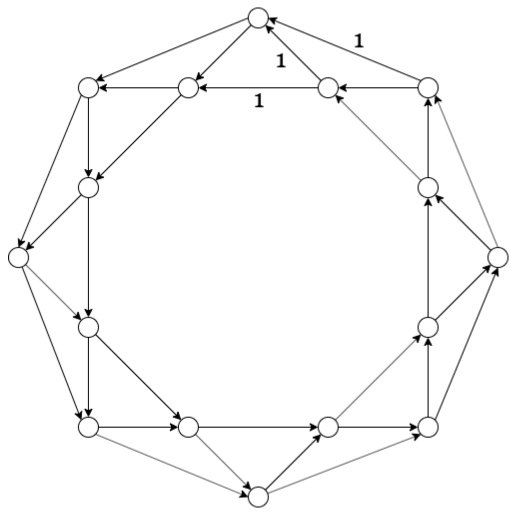
Index	Cocycle	Time Created	Time Destroyed
1	[a: 1, b: 1, c: 1]	t_1	(none)
2	[a: 0, b: 1]	t_2	t_3
4	[a: 0, b: 0, c: 1]	t_4	t_5
6	[ab: 0, ac: 0, bc: 1]	t_6	t_7

4.3 Explanation of Circular Coordinates

In may instances, our data has one or more major holes that we'd like to capture. Consider the following data:



Imagine we choose the following 1-cocycle (all unlabeled edges have the value zero):

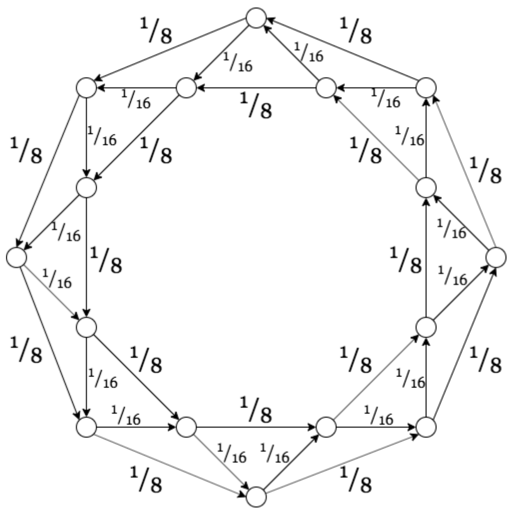


We can verify that the boundary of both triangles involving nonzero edges are zero. Thus all boundaries are zero, so we have a cocycle.

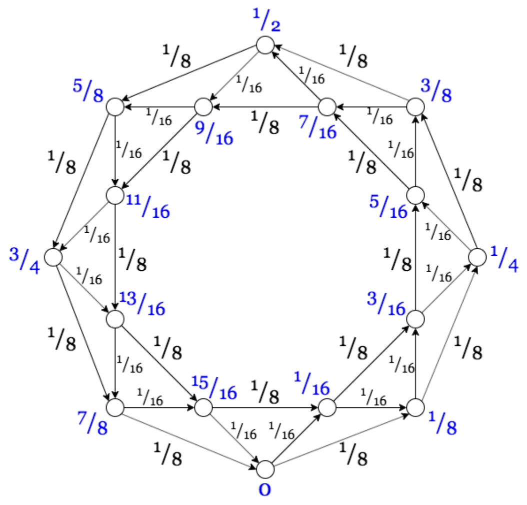
Intuitively, this cocycle will capture the winding number of a path: how many times it travels counterclockwise around the circle. For example, if the path travels clockwise but counterclockwise once, then it has a winding number of $2 - 1 = 1$.

We would like a cocycle that does the same job, but minimizes the $L2$ norm, which takes into account the squares of all edges. Formally, the L2 norm is $\sqrt{\Sigma_{i,j \in edges} d(p_i, p_j)^2}$. In this case, the L2 norm is $\sqrt{(1^2 + 1^2 + 1^2)} = \sqrt{3}$

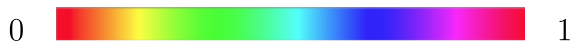
Minimizing the L2 norm gives the following cocycle:



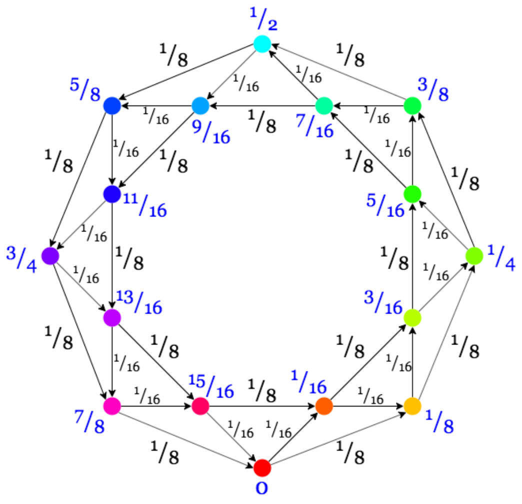
We can now assign a value to each vertex. We arbitrarily choose some vertex to be 0, and then increment by edge values as we travel around the circle. This gives us the following result:



Notice that we go from $\frac{7}{8}$ back to 0, and also from $\frac{15}{16}$ back to 0. This is because we are operating in $\mathbb{R}$ mod 1. To aid visualization, we color each vertex with the color corresponding to its value on this RGB strip:



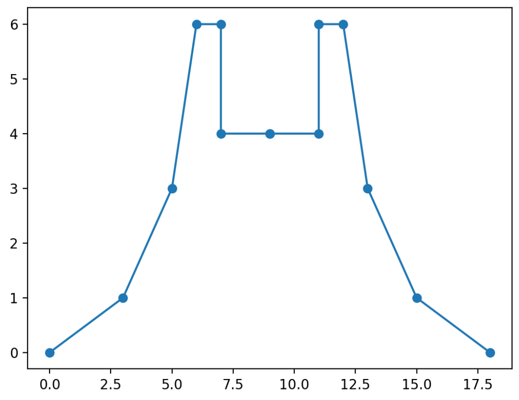
The result is as follows:

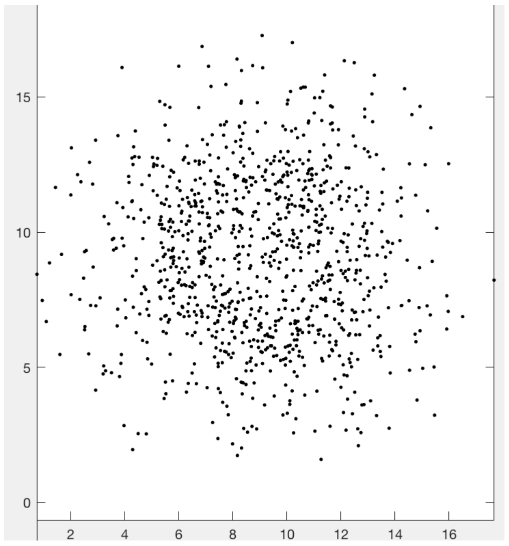


Indeed, we see that each vertex has a value corresponding to its position around the circle.

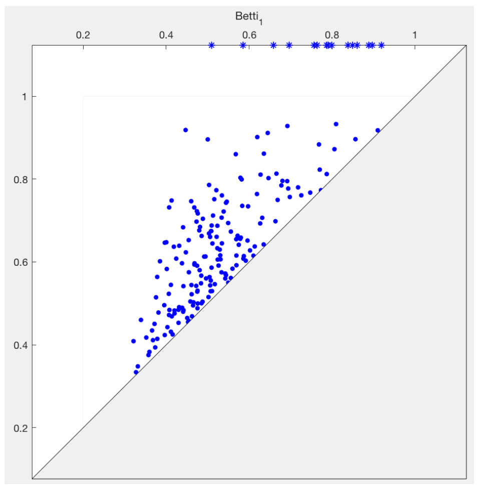
4.4 Toy Example for Circular Coordinates

We apply circular coordinate analysis to detect a hole in a normal distribution. To create our data, we define a roughly normal distribution with a hole in the center, and sample points on a two-dimensional plane according to this normal distribution:

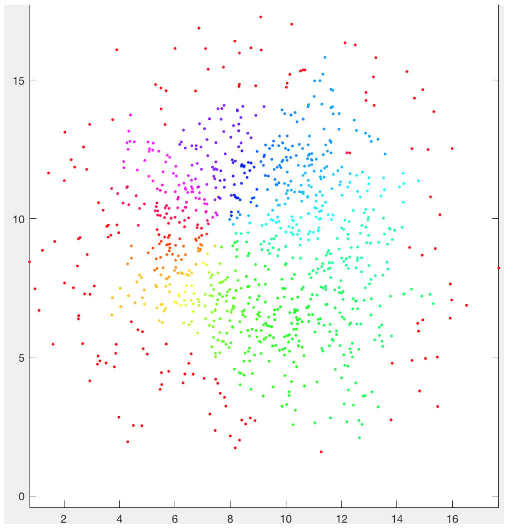

Normal Distribution


Sampled Points

Using the standard distance metric, we compute the persistence diagram as well as the circular coordinates for the most persistent cocycle:



Persistence Diagram


Circular Coordinates

Indeed, the 1-cocycle with $\epsilon_2 - \epsilon_1$ have captured a hole. However, we see that the hole it captured is slightly off-centered: it is to the top-left of the "actual" center of the hole. This was because the largest space with no points happened to not be in the center. This metric is thus subject to too much variance.

In order to capture the true center of the hole, we need to reduce variance by selecting a new metric. The metric we use is

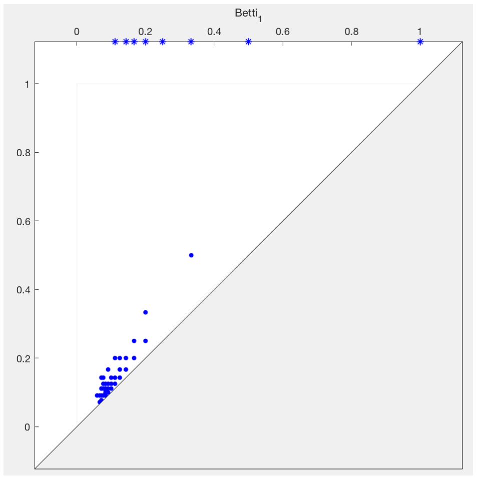
$$d(p_i, p_j) = \frac{1}{\min(\{density(p_i), density(p_j)\})}$$

where $density(p_i)$ is the number of points within a radius of 0.5 of p_i.

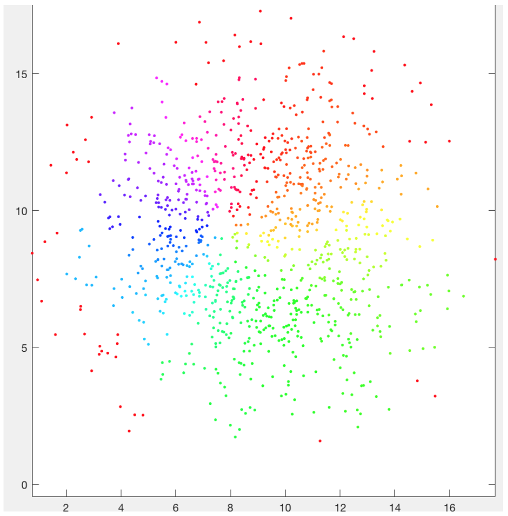
If we stopped here, our algorithm would connect all pairs of points. To prevent this, we set a limit of 0.8, so that points whose euclidean distance exceeds 0.8 are never connected.

In effect, this metric should cause our algorithm to first connect points in high-density regions, and only later to connect points in low-density regions.

The results are as follows. We plot the persistence diagram, as well as the circular coordinates for the most persistent cocycle.



Persistence Diagram



Circular Coordinates

Indeed, the 1-cocycle with maximal $\epsilon_2 - \epsilon_1$ captures the hole. It seems to be an improvement over the usual metric, as the center of the cocycle corresponds to the actual center of the hole.

4.5 Application of Circular Coordinates

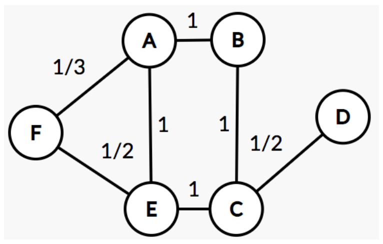
4.5.1 Background

I applied the circular coordinates algorithm to a dataset of Bach chorales. Each chorale is represented as a sequence of chords, where each chord is an array of notes. For example, the C chord corresponds to:

$$\begin{array}{cccccccccccc} C & C\# & D & D\# & E & F & F\# & G & G\# & A & A\# & B \\ \left[\ 1\right. & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \left. 0\ \right] \end{array}$$

I connect two chords X and Y whenever either X follows Y or Y follows X. The distance between X and Y is the inverse of the number of connections between them. This ensures that chords which follow each other often are "closer," and thus treated as stronger edges by our algorithm.

For example, suppose one chorale was this sequence of chords: [A, B, C, D, C, E, F, A, E, F, A, F]. Then we'd have the graph:

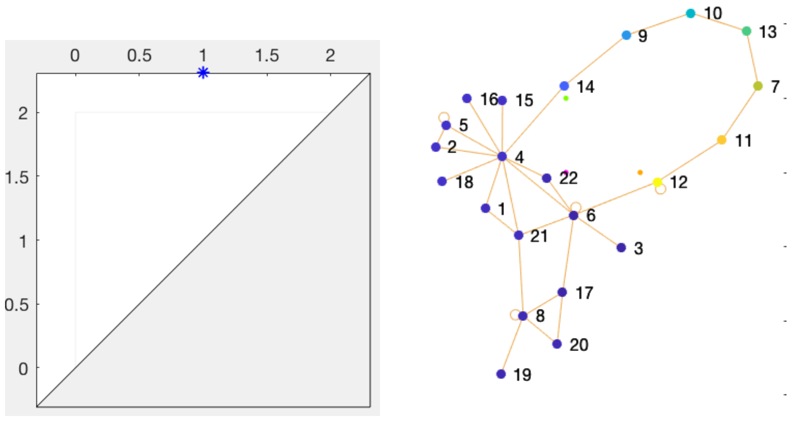


Notice, for example, that the closest two nodes (in terms of edge weight) are A and F. They were connected three times, and thus the weight of the edge between them is $\frac{1}{3}$.

Our dataset contained 162 chorales. I'll describe the results obtained from applying our circular coordinates algorithm to three chorales: one short, one medium, and one long:

4.5.2 Small Chorale

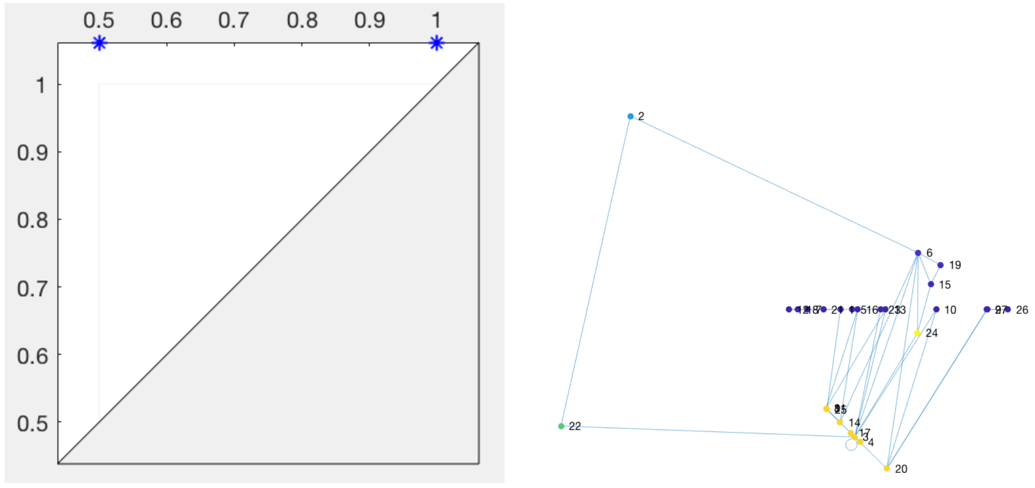
This chorale has 49 chords, with 22 distinct chords (the other 27 being repeats). The persistence diagram indicates that there is one hole. We see that the hole corresponds to a sequence of nine chords.



Persistence Diagram Circular Coordinates

4.5.3 Medium Chorale

This chorale had 82 chords, 27 of them distinct. We plot the persistence diagram, and the circular coordinates for the circular coordinates corresponding to the most persistent hole:

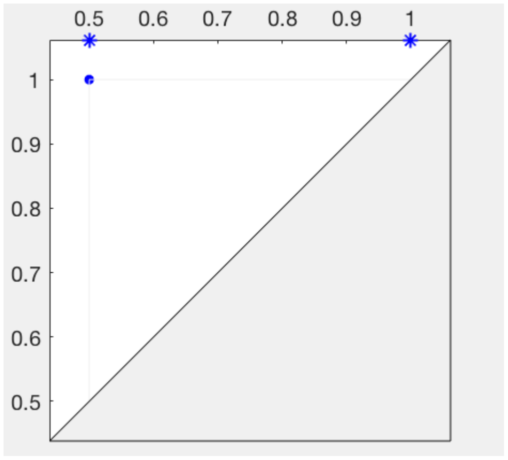


Persistence Diagram Circular Coordinates

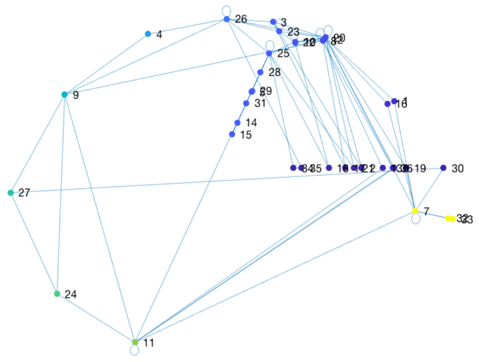
We notice that the algorithm picked up on a sequence of four chords:

4.5.4 Large Chorale

This chorale had 162 chords, 36 of of them distinct. We plot the persistence diagram, and the circular coordinates for the circular coordinates corresponding to the most persistent hole:



Persistence Diagram



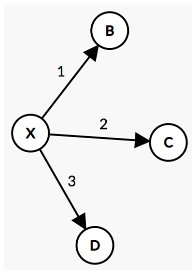
Circular Coordinates

It appears that the algorithm found a loop of length 9. Notice that chords that are part of this loop sometimes link to other chords that are not their immediate neighbors. This didn't happen for the shorter chorales. The fact that this happened means that each chord in the loop had more connections to each of its neighbors in the loop, and thus stronger edges. So our algorithm was able to detect the loop despite the noise caused by additional edges cutting across the loop.

4.5.5 Music Generation

We hoped to use this circular coordinate information to generate songs. We hoped that these generated songs would sound better than those naïvely generated by a simple bigram model.

In a standard bigram model, the mass given to each neighbor is simply the edge weight. If we have the graph:



then X will transition to B with probability $\frac{1}{6}$, to C with probability $\frac{2}{6} = \frac{1}{3}$, and to D with probability $\frac{3}{6} = \frac{1}{2}$. We denote that $m_x(y) = w_{xy}$.

To incorporate the circular coordinates, we instead use the mass function:

$$m_x(y) = w_{xy}(1 - L) + sigmoid(w_{xy}) * L * (D_{xy} + 0.5)$$

D_{xy} is the difference between circular coordinate values for x and y. Adding 0.5 makes this range from between 0 and 1. We hoped that this would encourage the song to progress around the circle.

"*sigmoid*" refers to a variant of the standard sigmoid function. Its effect was to make a value of 0 when $w_{xy} = 0$, and value 1 when $w_{xy} > 0$. This makes it impossible to make transitions not seen in the data.

Setting $L = 0$ makes our mass function the same as in the bigram model. Setting $L = 1$ makes the mass function only care about circular coordinates, diregarding edge weights. We decided that setting it somewhere in between would five the best results. Unfortunately, it didn't seem that the model that incorporated circular coordinates produced significantly better-sounding music than the bigram model did. We next discuss a few reasons for why this could be the case.

4.5.6 Limitations

We did not have information about what key each chord was in. It is thus likely that key changes occurred during the chorales, which we weren't able to account for.

Additionally, the circular coordinates algorithm doesn't account for which direction the transition between chords goes. For example, suppose a song has many transitions from the G chord to the C chord, but never vice versa. Our algorithm would juts make a strong edge between G and C, and thus be just as likely to make a transition from G to C as from C to G.

4.5.7 Future Work

Annotating the chords with key information and only applying the algorithm to sections of a song that are in the same key may produce better results

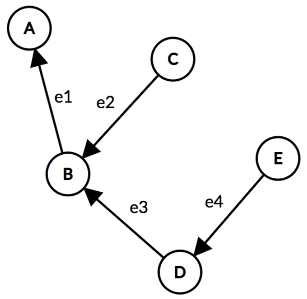
Chapter 5

Laplacian Eigenvector Analysis

5.1 Explanation of Laplacian Eigenvector Analysis

Intuitively, the Laplacian matrix L is used to calculate the net outward flow at each point in a graph. Given a vector v of node values, Mv gives the outward flow. Thus the eigenvectors of the laplacian matrix corresponding to small eigenvalues represent stable flows. This means that nodes close to one another in the graph must have similar corresponding values in the eigenvector. Furthermore, the eigenvectors of L are orthogonal, thus the first few eigenvectors of L provide an effective way to reduce the dimension of data.

Consider the following graph:



Suppose that we have the following function that assigns values to each node A, B, C, D, and E, in that order:

$$f = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

Let the matrix K encode the edges, with one column for each edge. Because this is a directed graph, within each column the entry for the source node of an edge is assigned value -1 while the destination is assigned the value 1. For undirected graphs, both values would be 1.

$$K = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

The gradient in this case denotes the flow along each edge. For example, $grad(e_1) = f(A) - f(B)$. So we can represent the gradient as follows:

$$grad(f) = K^T f = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

Also note that Kv will calculate the net outward flow at each point for any given vector field v. Recall that the divergence of a vector field is the net outward flux at each point. For a graph, the divergence denotes the net outward gradient flow at each point. Calculating the net outward flow for the vector field corresponding to the gradient will give us the graph's divergence. For example, $div(B) = -1*grad(e_1)+1*grad(e_2)+1*grad(e_3) = 2+2+1 = 5$. In general, we have:

$$div(n) = K * grad = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} -2 \\ 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 5 \\ -2 \\ 0 \\ 1 \end{bmatrix}$$

Substituting in the formula for the gradient, we have $div = KK^T f$. The matrix $L = KK^T$ is called the Laplacian. In this case, we have

$$L = KK^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Note that the value of L_{ii} will simply be the number of edges adjacent to node i. This because we are effectively taking the inner product of one row with itself, and since $-1^2 = 1^2 = 1$ this is equivalent to counting the number of nonzero entries in the row. For L_{ij} with $i \neq j$, we are taking the inner product of two different rows. The value at each index of the row will be -1 if there is an edge between nodes i and j and 0 otherwise. This is because the only time that two rows of K (equivalent to one row of K and one column of K^T share a nonzero entry in the same column is when there is an edge between the two nodes corresponding to these rows.

Let D be the diagonal matrix with the degrees of each node on the diagonal. In this case,

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Let A be the adjacency matrix. The value of A_{ij} will be 1 if there is an edge between nodes i and j and 0 otherwise. In this case, we have

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that $L = D - A$. In this example, all edges had the same weight of 1, but in other cases the edge weights can vary. Thus in general, the degree of a node is not the number of adjacent edges, but rather the sum of the weights of its adjacent edges.

We would like to incorporate some idea of normalization into the Laplacian. For example, some nodes may have much higher degrees than others. We would like the entries in those rows to still have similar values. We'd thus like to divide each cell by the degree of the two nodes involved. This is the intuition behind the graph laplacian, where we divide each cell by L_{ij} by $\sqrt{degree(i) * degree(j)}$, the geometric mean of the degrees of the nodes i and j. We can accomplish this as follows:

$$\begin{aligned} L^{sym} &= D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}} \\ &= D^{-\frac{1}{2}}DD^{-\frac{1}{2}} - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \\ &= I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \text{ (Because D is diagonal)} \end{aligned}$$

We then compute the eigenvectors $v_0 ... v_{n-1}$ and eigenvalues $e_0 ... e_{n-1}$ of L^{sym}. Intuitively, the eigenvectors corresponding to the smallest eigenvalues represent states without much flow between nodes. Thus all connected nodes will have similar corresponding values in the eigenvector.

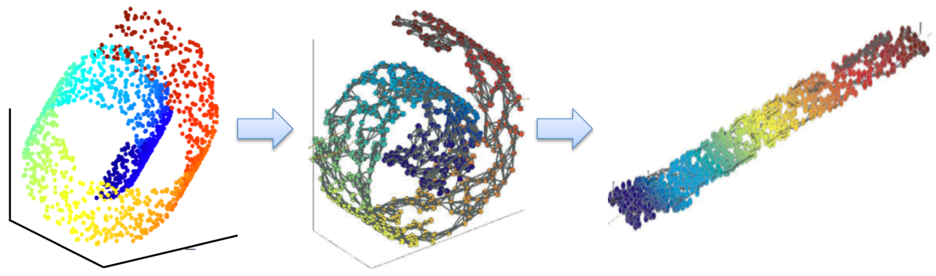
Also note that because L and L^{sym} are symmetric, then the eigenvalues are orthogonal. We can verify that this is the case: choose two distinct eigenvectors x and y with distinct eigenvalues λ and μ:

$$\lambda\langle x, y\rangle = \langle \lambda x, y\rangle = \langle Lx, y\rangle = \langle x, L^T y\rangle = \langle x, Ly\rangle = \langle x, \mu y\rangle = \mu\langle x, y\rangle$$

Thus $(\lambda - \mu)\langle x, y\rangle = 0$. But $\lambda \neq \mu$, thus $x \perp y$.

Because the eigenvectors are orthogonal, the information they provide is not redundant. Thus the first few eigenvectors of L^{sym} provides an effective way to reduce the dimension of the data. Note that for L, 1 is an eigenvector with eigenvalue 0. Similarly, the first eigenvector and eigenvalue are not informative for L^{sym}. We thus throw out very first eigenvector, and only consider the first few after that one.

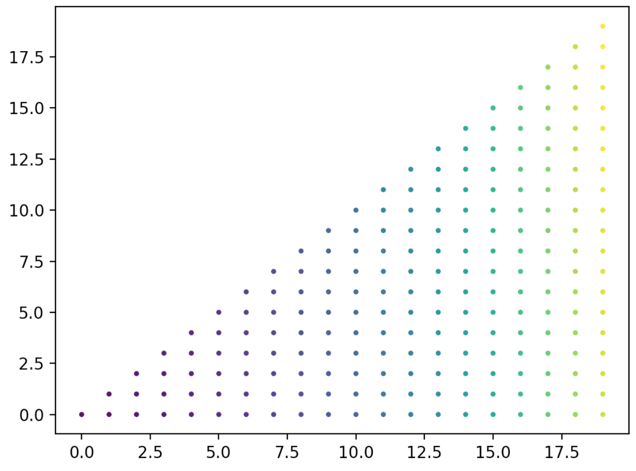
Also note that this method of dimension reduction only preserves local structure. Points that are connected to each other will have similar values, but points that are not connected may not - even if they are very close by. Consider the following example:



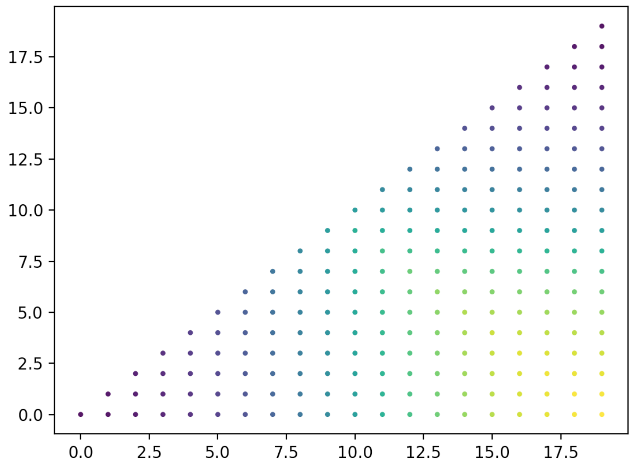
We see that points that were initially close by, but were not connected, can end up with very different values. Thus while the first few eigenvectors of the symmetric normalized Laplacian provide information about the structure of the data, it is important to remember that this structure is defined in terms of connected nodes rather than physical distance.

5.2 Application of Laplacian Eigenvector Analysis

We first calculate the eigenvectors of the laplacian of points sampled from a triangle. Here is the triangle, colored by the 2nd and 3rd eigenvectors:



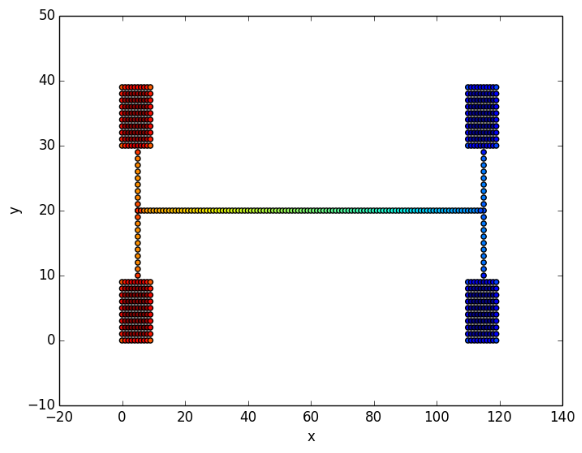
2nd Eigenvector



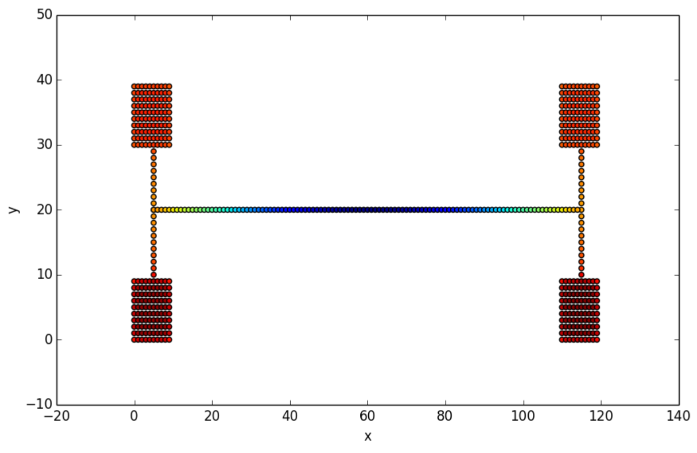
3rd Eigenvector

We notice that the 2nd eigenvector captures left-to-right variation: points in the bottom right have different values from those in the top left. Taken together, these two eigenvectors will successfully indicate what part of the triangle a point is located in.

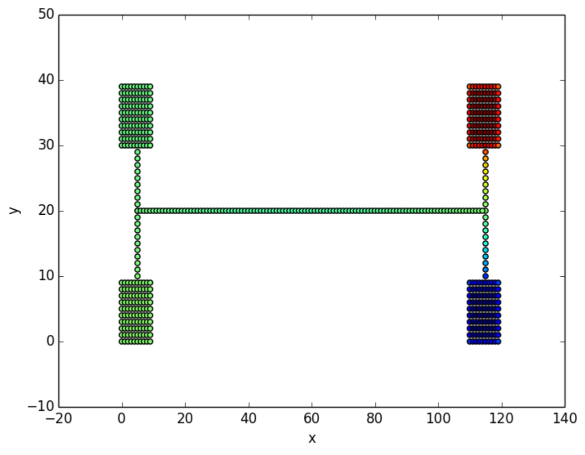
We now calculate the eigenvectors of the laplacian for connected components We plot the 2nd, 3rd, 4th, and 5th eigenvectors as before:



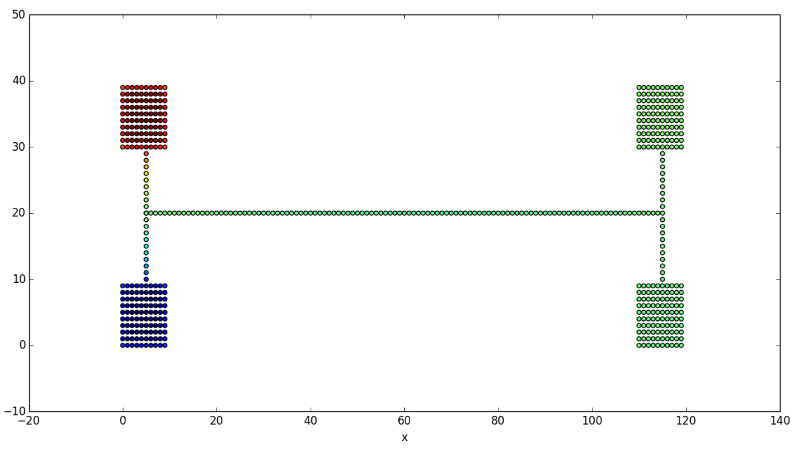
2nd Eigenvector



3rd Eigenvector



4th Eigenvector



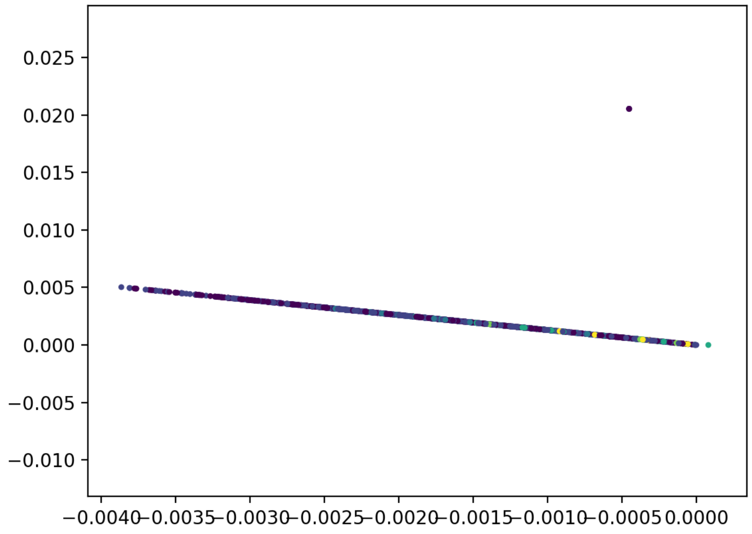
5th Eigenvector

Notice that the 2nd eigenvector, for example, captures left-to-right variation, whereas the 4th and 5th eigenvectors single out the different quadrants. Taken together, these eigenvectors will also provide us with information about where a point is located.

Because these examples are in low dimensions, we can tell where a point is without the aid of the eigenvectors of the laplacian But in higher dimensions where our vision fails, the information provided by these eigenvectors can be very useful.

We now calculate the eigenvectors of the laplacian for a dataset of wine quality. There are 11 input variables, such as “pH,” “density,” “fixed acidity,” and so forth. The output variable is a quality score ranging form 1 to 10, as judged by self-professed experts.

We plot the 2nd and 3rd eigenvectors of the laplacian, coloring each point with the judged quality. Darker colors like purple indicate higher quality. The results are as follows:



2nd Eigenvector vs. 3rd Eigenvector

We note that there is not much interesting structure in the data. Only the 2nd eigenvector captured some variance, as evidenced by the fact that the points are spread out mainly along the axis corresponding to the 2nd eigenvector. The 3rd did not contribute much. Further, there doesn’t appear to be a strong pattern in terms of wine quality. There do appear to be more lower-quality wines toward the right side of the plot, but there are also many higher quality wines there as well.

One reason that there the eigenvectors of the laplacian don’t capture a strong pattern may be that there is in fact no strong pattern. Wine tasting is a notoriously unreliable art, as many self-professed experts can’t tell the difference between a cheap wine and a more expensive one in a blind taste test.

Chapter 6

Mapper

6.1 Explanation of Mapper

In *Topology and Data*, Gunnar Carlsson introduces Mapper, a way to visualize the structure of high-dimensional data [1]. Mapper is insensitive to the metric used, and allows the user to visualize the structure at various levels of resolution, thereby detecting features that persist over time.
The user starts off with a set of points X, together with a metrix d. The Mapper algorithm works in five stages:

1: Filter. The user chooses a reference metric space (Z, d_z) and a filter function $\rho : X \to Z$. We call this function a filter.

The image of ρ will be used for clustering, so we would like $d_z(\rho(x), \rho(x'))$ to be small when $d(x, x')$ is small. For example, when z is $\mathbb{R}$, the graph laplacian is a good choice, where our graph connects all pairs of points (x, x') in X with edge weight inversely proportional to $d(x, x')$. As discussed last chapter, the graph laplacian assigns a similar value to nearby points that are connected.

2: Cover. The user selects a covering $\mathcal{U}$ of Z. $\mathcal{U}$ is a set $\{\mathcal{U}_1, ..., \mathcal{U}_n\}$ where $\forall x \in X$, $x \in \cup_{\alpha \in [1,n]} \mathcal{U}_\alpha$.

Intuitively, this is a set of overlapping intervals that capture all points in X. The final clustering will depend on the overlap between elements of the covering to make connections between clusters. Thus the user should ensure that there is adequate overlap.

The user can choose coverings with smaller intervals in order to get a more detailed view of the structure, or larger intervals in order to get a coarser view of the structure. The user can also make small modifications to the size of the intervals to see which features persist across different scales. The persistent features are likely the most important.

3: Group. $\forall \alpha \in [1, n]$, we construct the group $X_\alpha = \{x \in X | \rho(x) \in \mathcal{U}_\alpha\}$. Intuitively, this corresponds to the points in X that fall into each element of our covering.

4: Cluster The user selects some small $\epsilon \in \mathbb{R}$. Within each group X_α, we apply single-linkage clustering, where we connect any two points x and x' if $d(x, x') < \epsilon$.

The user should ensure that ϵ is not so small that it splits a legitimate cluster in two, and not so large that it merges two legitimate clusters into one (where "legitimate clusters" are the clusters that we think the algorithm should find).

We now understand the benefit Mapper provides by being metric insensitive. Many modern datasets have a metric that encodes a rough notion of similarity. Mapper only uses this metric to check if two points are within some threshold distance of each other, thus it will provide the same result under all continuous transformations of the distance metric, provided that the user adjusts ϵ appropriately, It is also fairly insensitive to minor changes in the metric, as all that Mapper cares about is whether the threshold has been met or not.

We now have a set of clusters parametrized by (α, c), where $\alpha \in [1, n]$ and c is one of the distinct clusters found in x_α.

5: Connect. We construct a simplicial complex with vertex set (α, c), creating a k-simplex $\{(\alpha_0, c_0), ..., (\alpha_k, c_k)\}$ when all those clusters share a point. Intuitively, we connect overlapping clusters.

Summary: The user has a set of points X along with a metric d. They choose a filter ρ, a covering $\mathcal{U}$, and a threshold ϵ. Given this data and these parameters, Mapper clusters the data, allowing the user to visualize its structure.

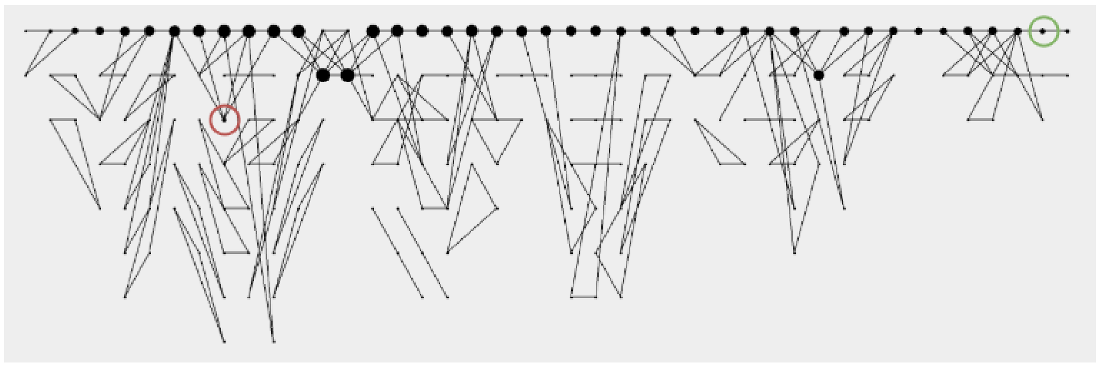
Simple Example: To illustrate how Mapper works, we consider the following example.

6.2 Application of Mapper

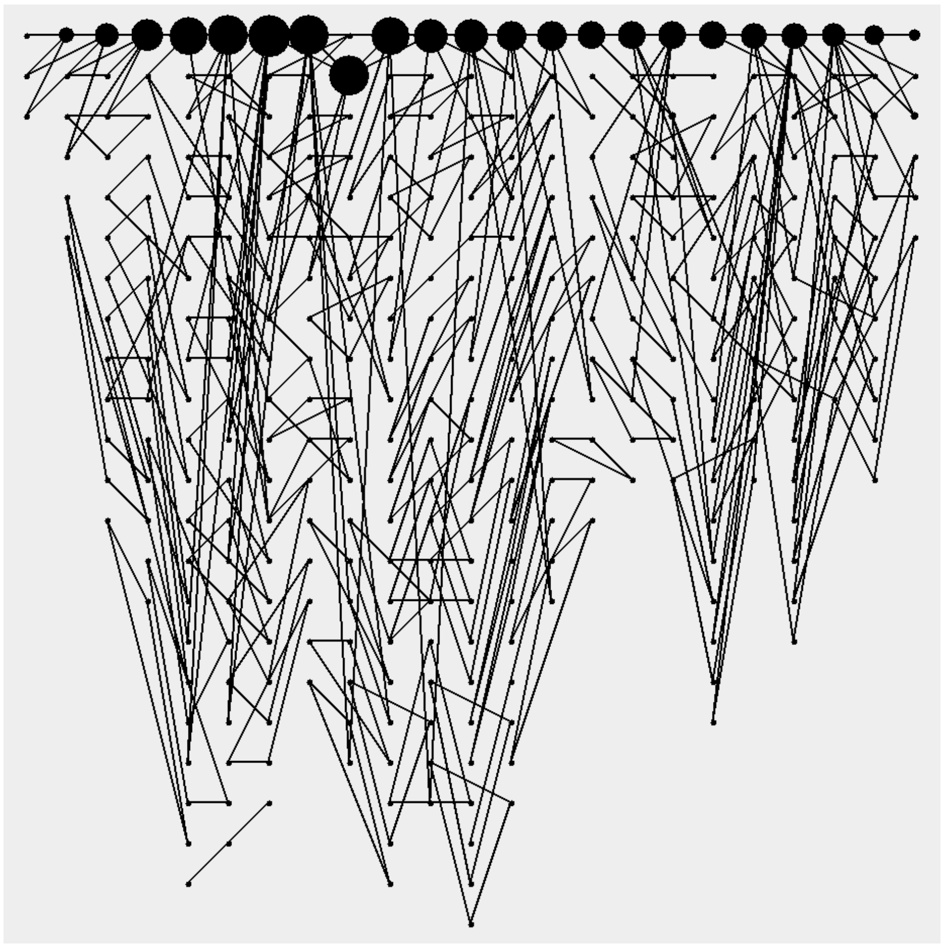
I applied Mapper to a dataset of 1567 universities. There were nine features:

- Graduation rate
- Percent of first-time undergraduate students given financial aid
- Average amount of aid per student given by the government
- Total price for in-state students living on campus
- Total price for out-of-state students living on campus
- Admissions rate
- Yield (percent of admitted students who attend)
- Number of instructional faculty
- Number of service staff

I used the first eigenvector of the Laplacian as the filter function. I chose a window size and stride length such that there were 43 non-empty windows. The results are as follows:



We see that Mapper did not find much interesting structure. In each window, almost all of the points in that window got grouped into one cluster. I reduced the threshold for linkage in the single-linkage clustering by two-thirds, in an attempt to split apart the larger clusters, and find more interesting structure. This did not work. In each window, several small points split off from the large cluster, but the large cluster was preserved:



The fact that all points get grouped into one cluster indicates that within each window, all data points were pretty close. Mapper suggests that the structure of this data is indeed linear.

It did appear that different windows contained different categories of colleges. For example, 16 of the 27 colleges in the cluster highlighted in green were religious colleges. This is far fewer than the average. I was not able to count all religious colleges in the dataset, but internet research indicates that about twenty percent of colleges have a religious affiliation. The cluster highlighted in red contained fairly prestigious colleges:

- Haverford College
- Brown University
- Bates College
- Carnegie Mellon University
- Trinity College
- Wake Forest University
- Rhode Island School of Design
- Skidmore College

Although the clusters do seem to be informative, note that this information was captured by the eigenvectors of the laplacian alone, not by Mapper as whole

Chapter 7

Conclusion

As computers have become ubiquitous, we've collected data on all sorts of phenomena including schools, social media, healthcare, and genetics. These datasets have many features, and thus high-dimensional datasets are very common. Being able to both visualize them and detect patterns in them is very important. In this paper, we've explored several methods to do both of those things. Using Mapper and calculating the eigenvectors of the Laplacian are both useful tools for data visualization. Helmholtz decomposition and cohomology analysis are both useful methods for detecting patterns in the data. We introduced the novel idea of applying cohomology analysis to detect patterns in music, but there are a huge variety of applications that are currently unexplored. Future work could continue to find new domains where these tools are applicable, as well as develop improved tools for data analysis.

Bibliography

[1] Carlsson, Gunnar. “Topology and data.” *Bulletin of the American Mathematical Society* 46.2 (2009): 255-308.

[2] Zomorodian, Afra, and Gunnar Carlsson. “Computing persistent homology.” *Discrete & Computational Geometry* 33.2 (2005): 249-274.

[3] De Silva, Vin, Dmitriy Morozov, and Mikael Vejdemo-Johansson. “Persistent cohomology and circular coordinates.” *Discrete & Computational Geometry* 45.4 (2011): 737-759.

[4] Giusti, Chad, et al. “Clique topology reveals intrinsic geometric structure in neural correlations.” *Proceedings of the National Academy of Sciences* 112.44 (2015): 13455-13460.

[5] Ulmer, M., Lori Ziegelmeier, and Chad M. Topaz. “Assessing biological models using topological data analysis.” *arXiv preprint arXiv:1811.04827* (2018).

[6] Jiang, Xiaoye, et al. “Statistical ranking and combinatorial Hodge theory.” *Mathematical Programming* 127.1 (2011): 203-244.

[7] Singh, Gurjeet, Facundo Mmoli, and Gunnar E. Carlsson. “Topological methods for the analysis of high dimensional data sets and 3d object recognition.” *SPBG.* 2007.