Advances in Topology-Based Graph Classification

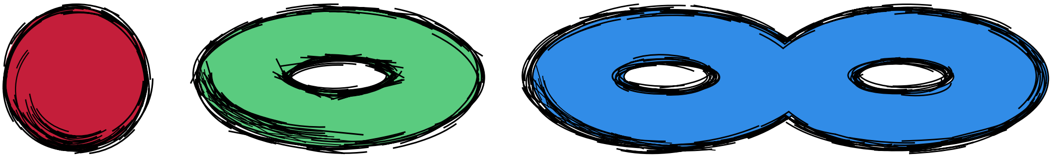
Bastian Rieck

Pseudomanifold

MLCB D BSSE ETH zürich

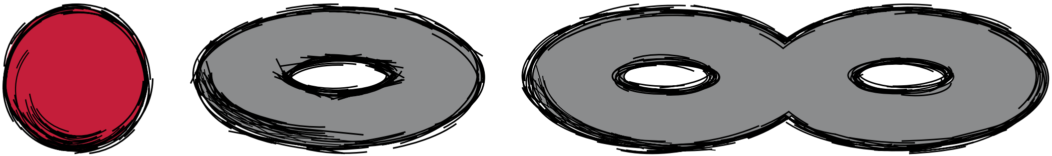
What is topology?

Studying the abstract *shape* of objects



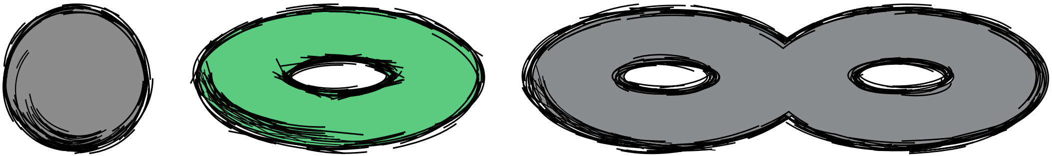
What is topology?

Studying the abstract *shape* of objects



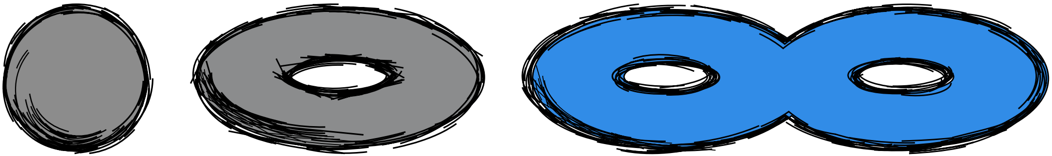
What is topology?

Studying the abstract *shape* of objects



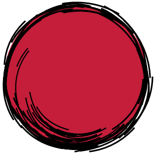
What is topology?

Studying the abstract *shape* of objects



Betti numbers

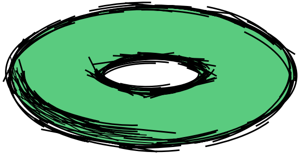
Counting d-dimensional holes



$$\beta_0 = 1, \beta_1 = 0, \beta_2 = 1$$

Betti numbers

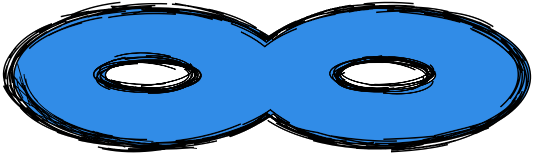
Counting d-dimensional holes



$$\beta_0 = 1, \beta_1 = 2, \beta_2 = 1$$

Betti numbers

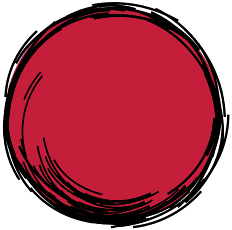
Counting d-dimensional holes



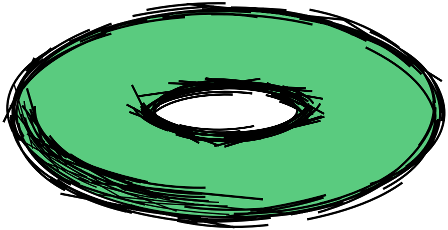
$\beta_0 = 1, \beta_1 = 4, \beta_2 = 1$

Betti numbers

Counting d-dimensional holes

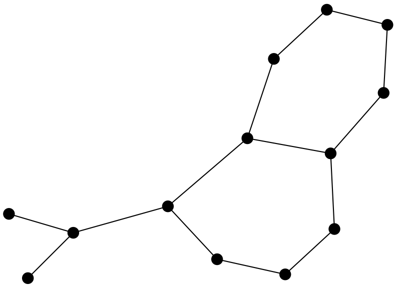


$\beta_0 = 1, \beta_1 = 0, \beta_2 = 1$

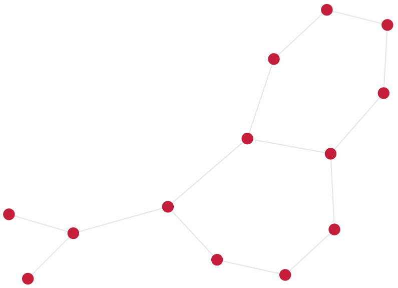


$\beta_0 = 1, \beta_1 = 2, \beta_2 = 1$

The shape of a graph

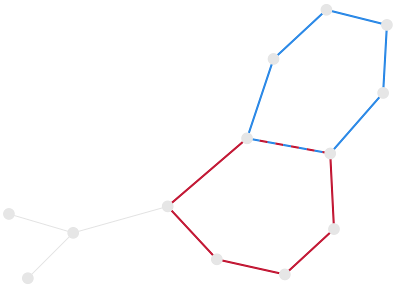


The shape of a graph



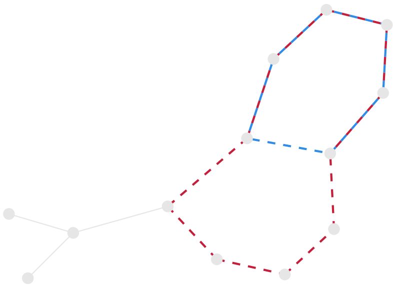
Connected components

The shape of a graph



Cycles

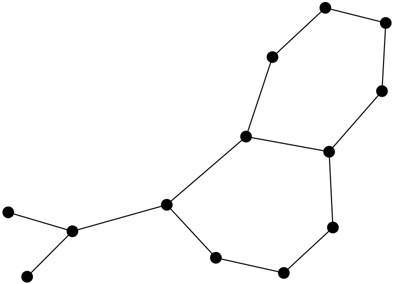
The shape of a graph



Alternative cycles

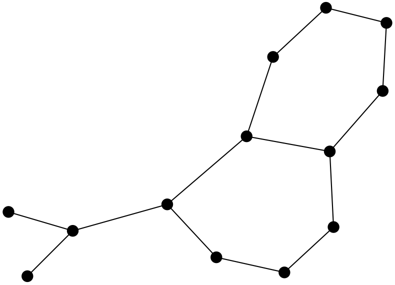
The Betti numbers of a graph

A graph with n vertices, m edges, and k connected components has $\beta_0 = k$ and $\beta_1 = m + k - n$.

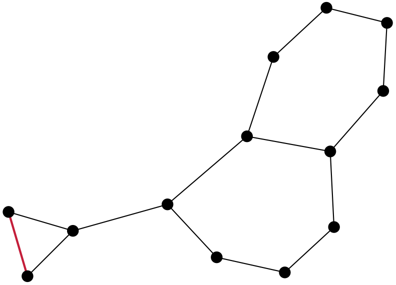


$$\beta_0 = 1, \beta_1 = 14 + 1 - 13 = 2$$

Comparing two graphs using Betti numbers



$$\beta_0 = 1, \beta_1 = 14 + 1 - 13 = 2$$



$$\beta_0 = 1, \beta_1 = 15 + 1 - 13 = 3$$

Properties of Betti numbers

Betti numbers are invariant under graph isomorphism, i.e. if $\mathcal{G}$ and $\mathcal{G}'$ are isomorphic, their Betti numbers will coincide (the other direction is not true).

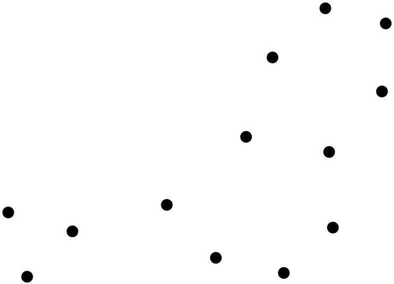
Some 'heretical' thoughts

- Graph isomorphism is too restrictive for many purposes anyway; we want 'near-isomorphism' or *isometry*.

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

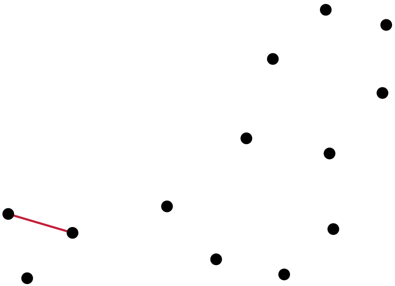


$$\beta_0 = 13, \beta_1 = 0 + 13 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

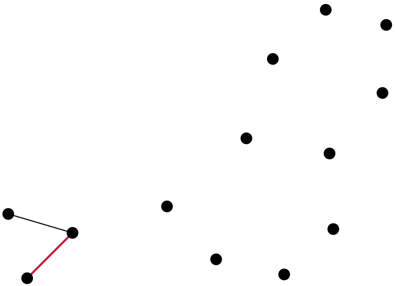


$$\beta_0 = 12, \beta_1 = 1 + 12 - 13 = 0$$

Persistent homology

Intuition

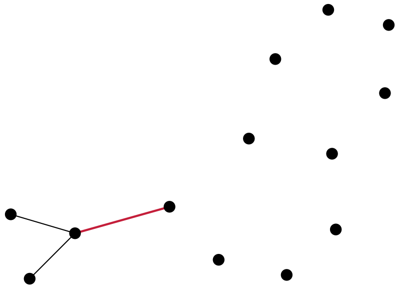
Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!



$$\beta_0 = 11, \beta_1 = 2 + 11 - 13 = 0$$

Persistent homology
Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

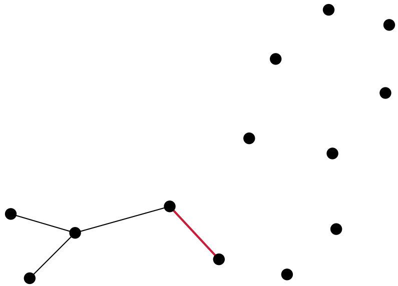


$$\beta_0 = 10, \beta_1 = 3 + 10 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

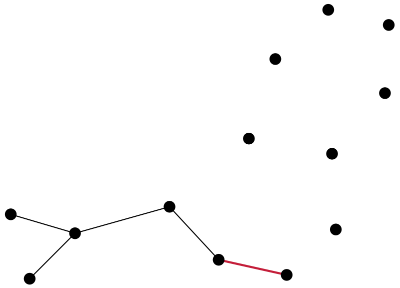


$$\beta_0 = 9, \beta_1 = 4 + 9 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

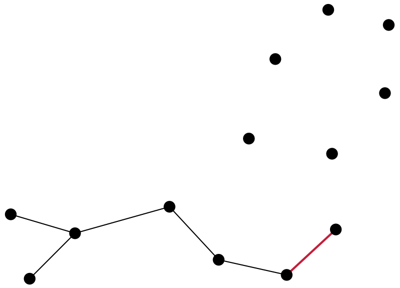


$$\beta_0 = 8, \beta_1 = 5 + 8 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

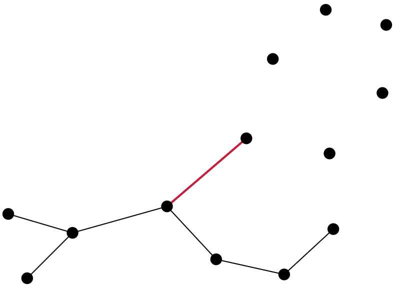


$$\beta_0 = 7, \beta_1 = 6 + 7 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

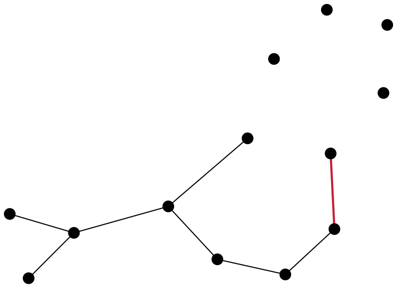


$$\beta_0 = 6, \beta_1 = 7 + 6 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

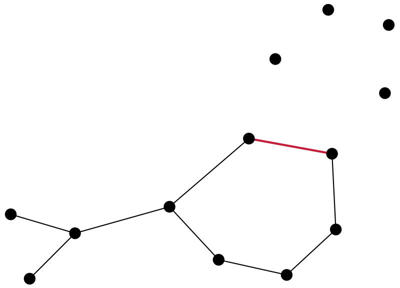


$$\beta_0 = 5, \beta_1 = 8 + 5 - 13 = 0$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

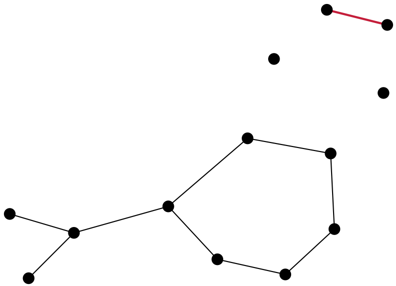


$$\beta_0 = 5, \beta_1 = 9 + 5 - 13 = 1$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

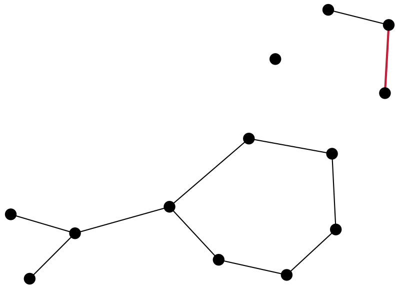


$$\beta_0 = 4, \beta_1 = 10 + 4 - 13 = 1$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

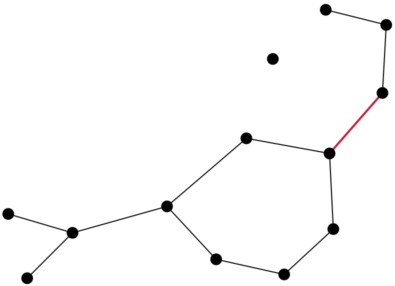


$$\beta_0 = 3, \beta_1 = 11 + 3 - 13 = 1$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

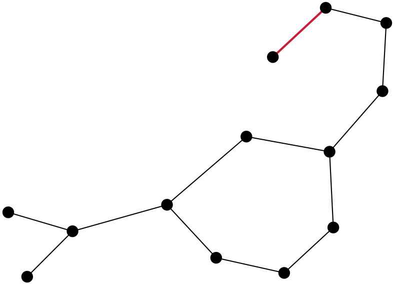


$$\beta_0 = 2, \beta_1 = 12 + 2 - 13 = 1$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

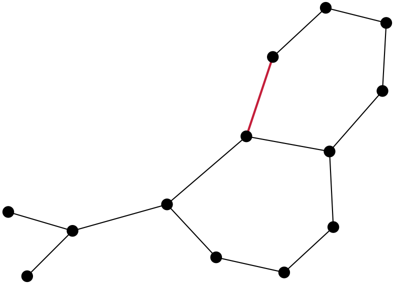


$$\beta_0 = 1,\, \beta_1 = 13 + 1 - 13 = 1$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

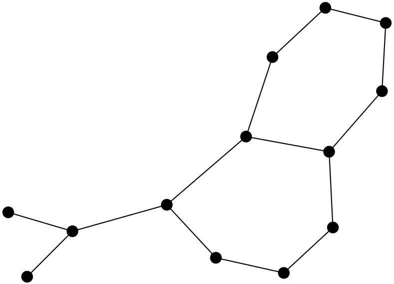


$$\beta_0 = 1, \beta_1 = 14 + 1 - 13 = 2$$

Persistent homology

Intuition

Suppose we have *weights* on the edges. If we add them in ascending order of their weight, we can watch as topological features change!

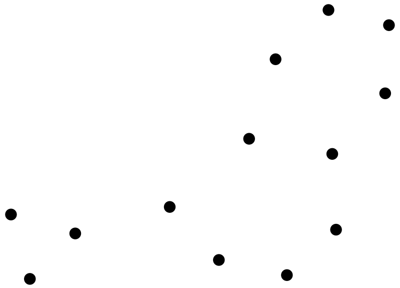


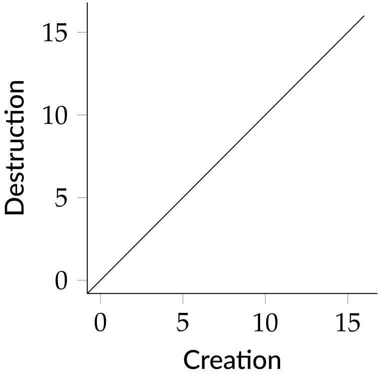
$$\beta_0 = 1, \beta_1 = 14 + 1 - 13 = 2$$

Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

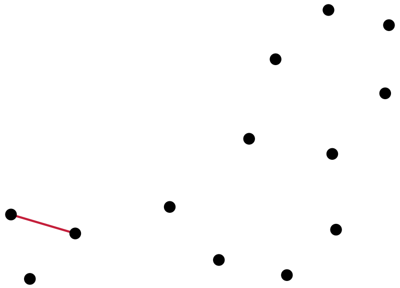


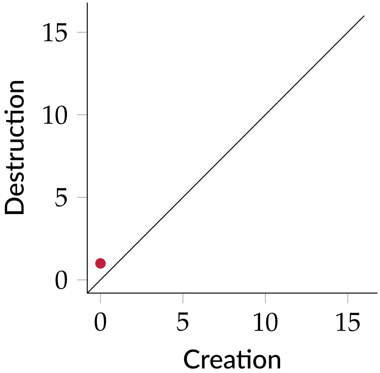


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

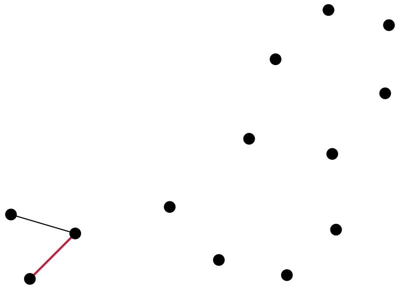


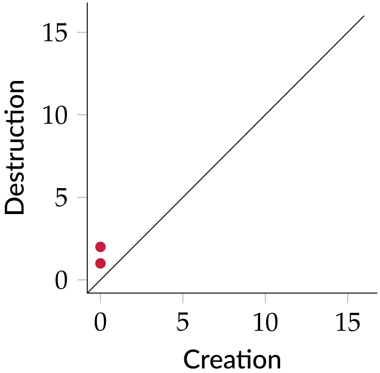


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

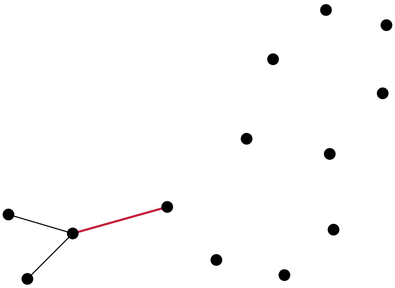


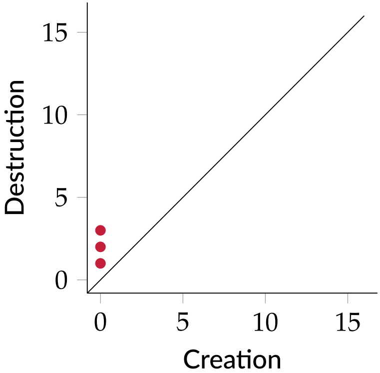


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

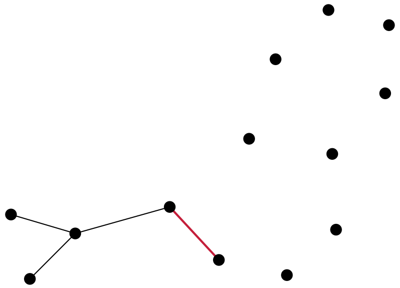


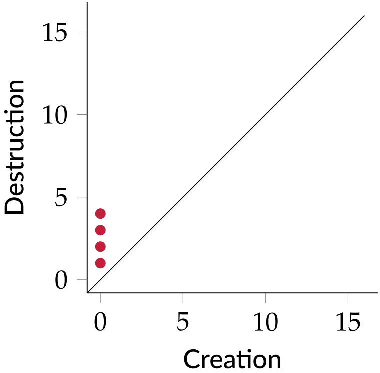


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

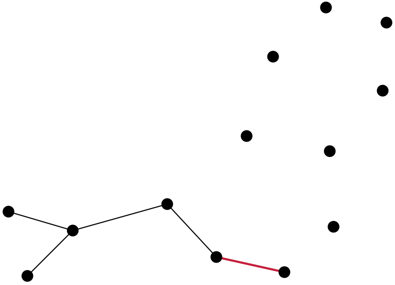


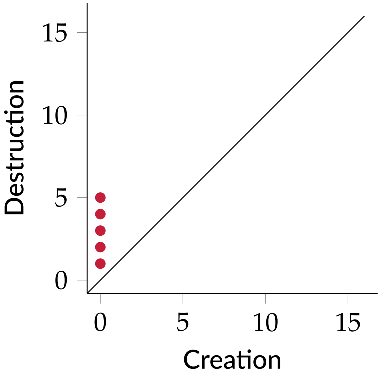


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

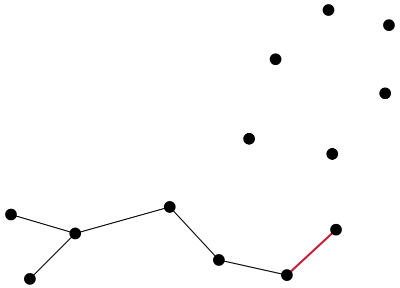


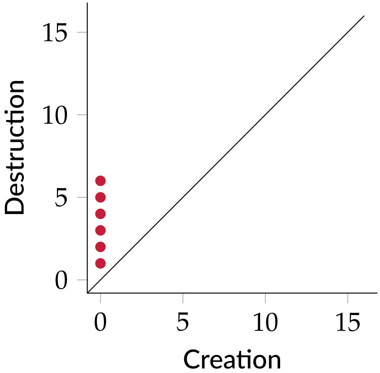


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

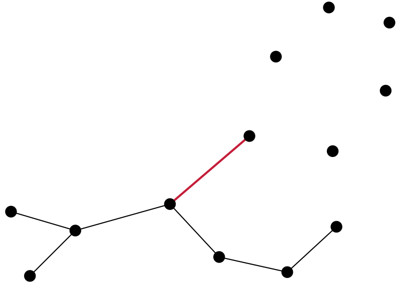


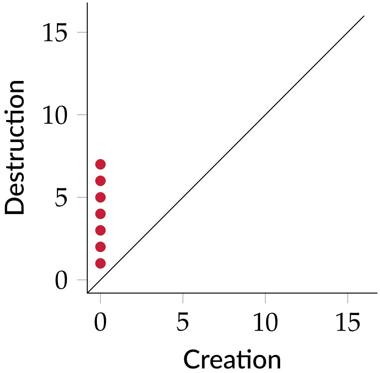


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

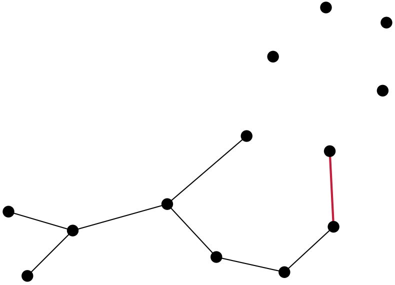


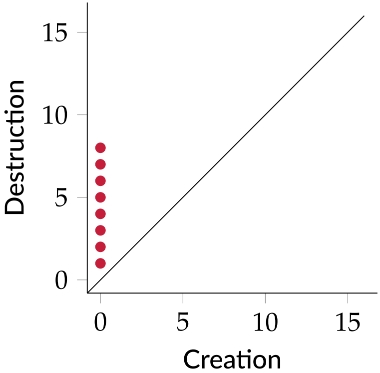


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

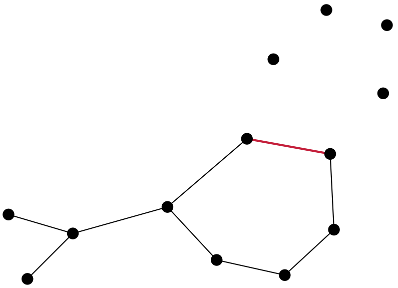


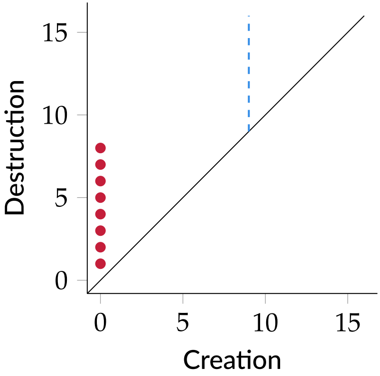


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

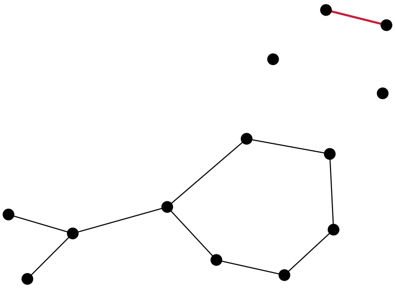


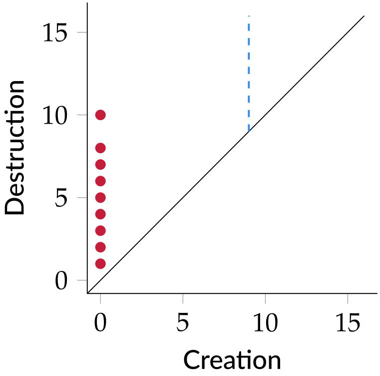


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

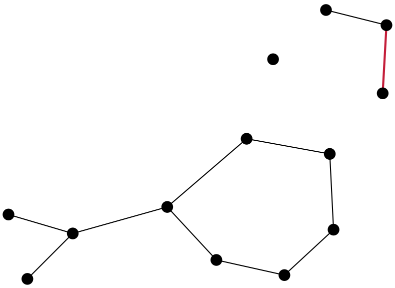


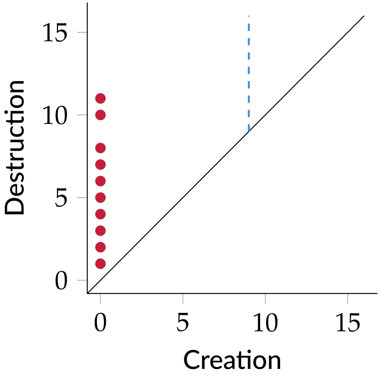


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

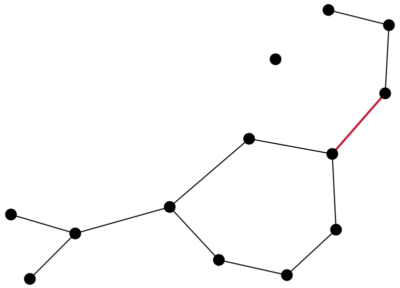


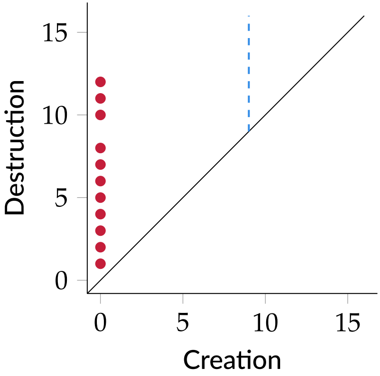


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

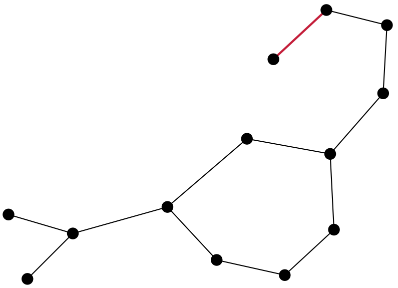


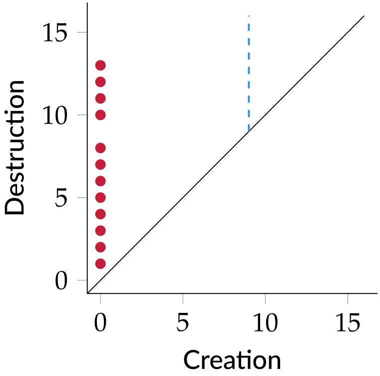


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

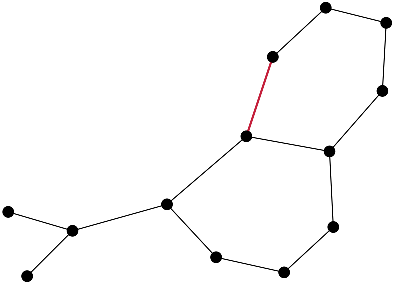


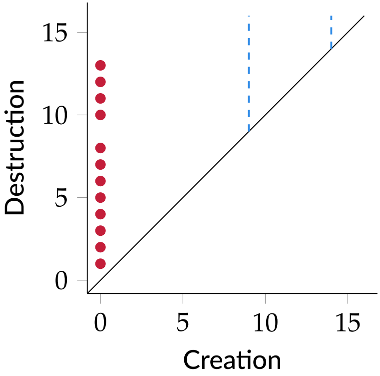


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.

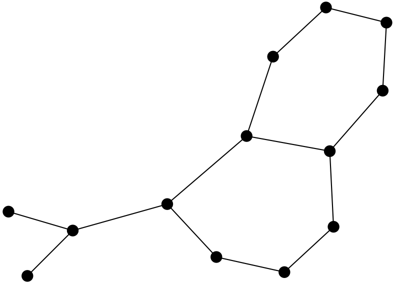


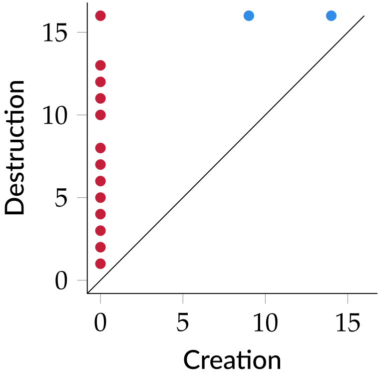


Persistent homology

Intuition, continued

Store information about features in a *persistence diagram*. A tuple (c, d) indicates that a topological feature was created at step c and destroyed at step d.





Persistent homology

Some concepts

We are calculating topological features of a *filtration* of graphs, i.e. a sequence of subgraphs satisfying

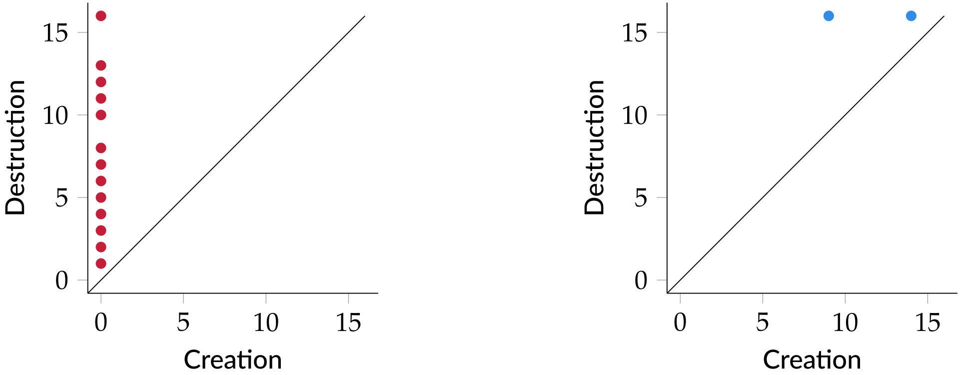
$$\mathcal{G}_0 \subseteq \mathcal{G}_1 \subseteq \ldots \mathcal{G}_{k-1} \subseteq \mathcal{G}_k = \mathcal{G},$$

where $\mathcal{G}$ is the 'original' graph.

Persistent homology has a rich mathematical foundation that permits its use in the context of *simplicial complexes*, i.e. generalised graphs, and many other types of data structures.

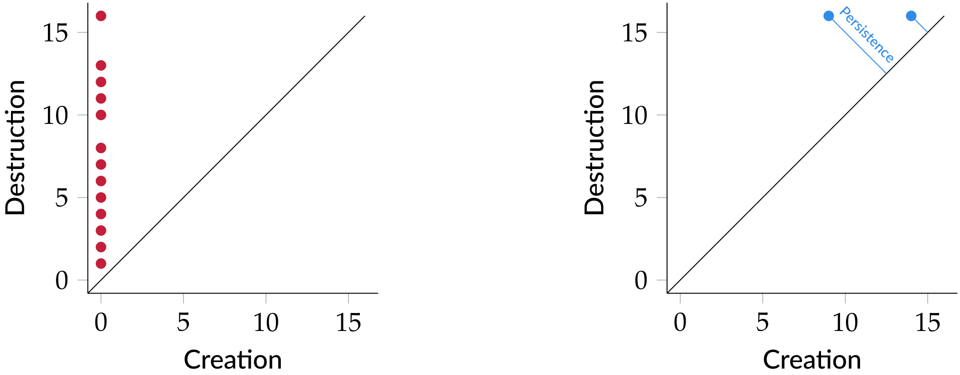
Some formal properties

Persistent homology assigns a graph $\mathcal{G}$ with a function $f\colon \mathcal{G} \to \mathbb{R}$ a set of *persistence diagrams*, describing the topological features of $\mathcal{G}$, as 'measured' via f.



Some formal properties

Persistent homology assigns a graph $\mathcal{G}$ with a function $f\colon \mathcal{G} \to \mathbb{R}$ a set of *persistence diagrams*, describing the topological features of $\mathcal{G}$, as 'measured' via f.



$$\operatorname{pers}(c, d) := |d - c|$$

Using persistence diagrams with machine learning pipelines

Persistence diagrams are multisets in $\mathbb{R} \times \mathbb{R} \cup \{\infty\}$, which can make their use in machine learning somewhat cumbersome. There are two schools of thought for integrating them:

1. *Vectorise* the diagram (i.e. create high-dimensional feature vectors)!
2. Change the *architecture* to include them!

Using persistence diagrams with machine learning pipelines

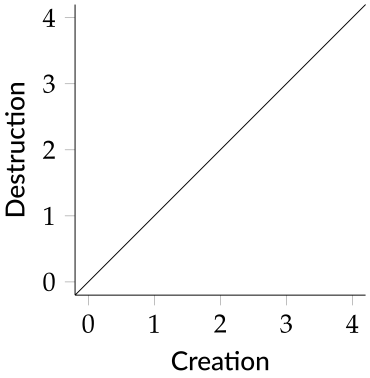
Persistence image calculation

The persistence image amounts to a *density estimation* (with appropriate weights). This image can be made into a high-dimensional feature vector, and integrated into any machine learning algorithm.

A filtration for unlabelled & unweighted graphs

Use a filtration based on *vertex degrees*. Set $f(v) := \deg(v)$ for every vertex v and $f(u, v) := \max(\deg(u), \deg(v))$ for every edge (u, v) to obtain $f\colon \mathcal{G} \to \mathbb{R}$.

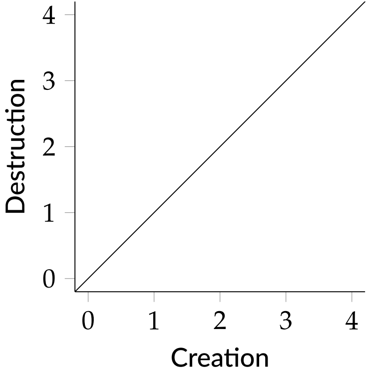
$f^{-1}((-\infty, 0])$



A filtration for unlabelled & unweighted graphs

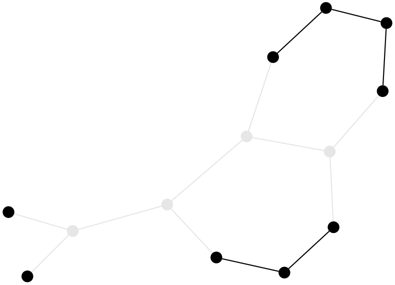
Use a filtration based on *vertex degrees*. Set $f(v) := \deg(v)$ for every vertex v and $f(u,v) := \max(\deg(u), \deg(v))$ for every edge (u,v) to obtain $f \colon \mathcal{G} \to \mathbb{R}$.

$f^{-1}((-\infty, 1])$

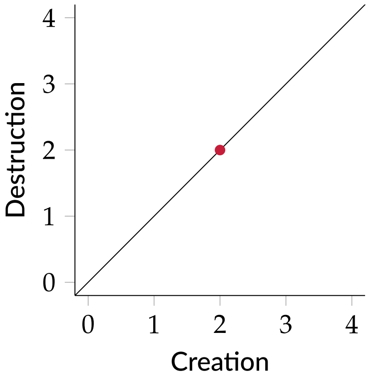


A filtration for unlabelled & unweighted graphs

Use a filtration based on *vertex degrees*. Set $f(v) := \deg(v)$ for every vertex v and $f(u,v) := \max(\deg(u), \deg(v))$ for every edge (u,v) to obtain $f\colon \mathcal{G} \to \mathbb{R}$.

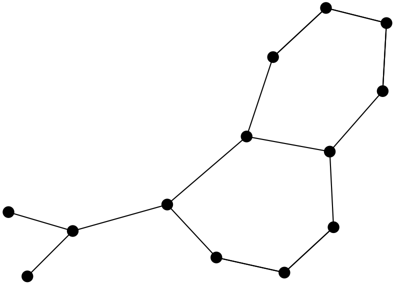


$f^{-1}((-\infty, 2])$

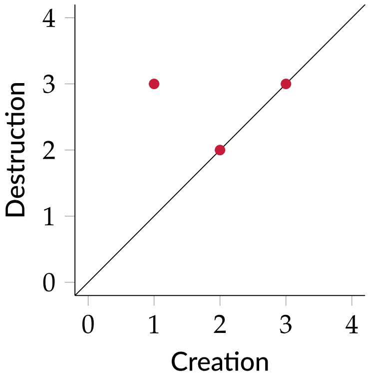


A filtration for unlabelled & unweighted graphs

Use a filtration based on *vertex degrees*. Set $f(v) := \deg(v)$ for every vertex v and $f(u,v) := \max(\deg(u), \deg(v))$ for every edge (u,v) to obtain $f\colon \mathcal{G} \to \mathbb{R}$.

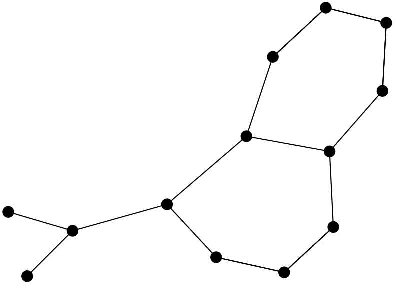


$f^{-1}((-\infty, 3])$

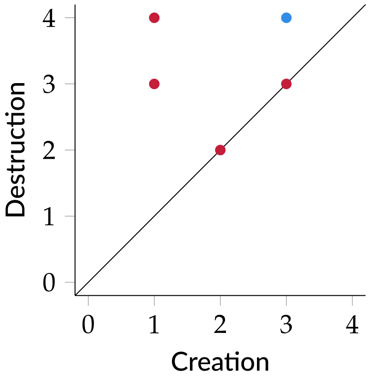


A filtration for unlabelled & unweighted graphs

Use a filtration based on *vertex degrees*. Set $f(v) := \deg(v)$ for every vertex v and $f(u,v) := \max(\deg(u), \deg(v))$ for every edge (u,v) to obtain $f\colon \mathcal{G} \to \mathbb{R}$.



$f^{-1}((-\infty, 3])$



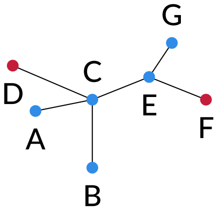
The Weisfeiler–Lehman test for graph isomorphism

1. *Create* a colour for each node in the graph (based on its label or its degree).
2. *Collect* colours of adjacent nodes in a multiset.
3. *Compress* the colours in the multiset and the node's colour to form a new one.
4. *Continue* this relabelling scheme until the colours are stable.

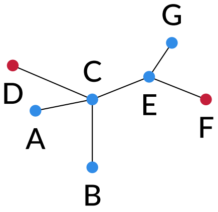
If the compressed labels of two graphs *diverge*, the graphs are *not* isomorphic!

The other direction is not valid! Non-isomorphic graphs can give rise to coinciding compressed labels.

Weisfeiler–Lehman iteration & subtree feature vector

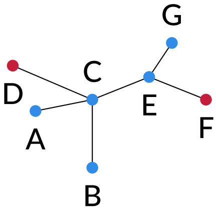


Weisfeiler–Lehman iteration & subtree feature vector



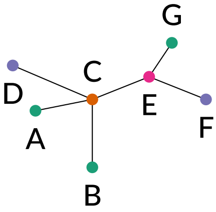
Node	Own label	Adjacent labels
A	🔵	🔵
B	🔵	🔵
C	🔵	🔵 🔵 🔵 🔴
D	🔴	🔵
E	🔵	🔵 🔵 🔴
F	🔴	🔵
G	🔵	🔵

Weisfeiler–Lehman iteration & subtree feature vector



Node	Own label	Adjacent labels	Hashed label
A	●	●	●
B	●	●	●
C	●	● ● ● ●	●
D	●	●	●
E	●	● ● ●	●
F	●	●	●
G	●	●	●

Weisfeiler–Lehman iteration & subtree feature vector



Label				
Count	3	1	2	1

$$\Phi(\mathcal{G}) := (3, 1, 2, 1)$$

Compare $\mathcal{G}$ and $\mathcal{G}'$ using a *kernel function*.

Kernels

$$\mathrm{k}(\mathcal{G}, \mathcal{G}') := \langle \Phi(\mathcal{G}), \Phi(\mathcal{G}') \rangle$$

$$\mathrm{k}(\mathcal{G}, \mathcal{G}') := \exp\left(-\|\Phi(\mathcal{G}) - \Phi(\mathcal{G}')\|^2 / \left(2\sigma^2\right)\right)$$

A Persistent Weisfeiler–Lehman Procedure for Graph Classification

Proceedings of the 36th International Conference on Machine Learning

A Persistent Weisfeiler–Lehman Procedure for Graph Classification

Bastian Rieck *1 Christian Bock *1 Karsten Borgwardt 1

Abstract

The Weisfeiler–Lehman graph kernel exhibits competitive performance in many graph classification tasks. However, its subtree features are not able to capture connected components and cycles, topological features known for characterising graphs. To extract such features, we leverage propagated node label information and transform unweighted graphs into metric ones. This permits us to augment the subtree features with topological information obtained using *persistent homology*, a concept from topological data analysis. Our method, which we formalise as a generalisation of Weisfeiler–Lehman subtree features, exhibits favourable classification accuracy and its improvements in predictive performance are mainly driven by including cycle information.

1. Introduction

Graph-structured data sets are ubiquitous in a variety of different application domains, each of them posing a separate challenge while also requiring different tasks to be solved. A common task involves *graph classification*, for which a variety of methods exists. These methods comprise convolutional neural networks (Duvenaud et al. 2015), recurrent neural networks (Lei et al. 2017), or Hilbert space methods (Vishwanathan et al. 2010), the latter also being referred to as *graph kernels*. While several approaches for defining graph kernels exist, the most common one uses the $\mathcal{R}$-convolution framework (Haussler 1999), which makes it possible to define the similarity between two graphs as a function of the similarity of their substructures.

Substructures that have been used for graph classification range from graphlets (Shervashidze et al. 2009), i.e. small non-isomorphic graphs of fixed size, over shortest paths (Borgwardt & Kriegel 2005), to random walks (Gärtner et al. 2003, Kashima et al. 2003, Sugiyama & Borgwardt 2015). One of the most powerful substructures is the set of *subtree patterns* (Ramon & Gärtner 2003), i.e. patterns based on rooted subgraphs of a graph. Their computational complexity made their applicability somewhat limited, until the Weisfeiler–Lehman (WL) graph kernel framework (Shervashidze & Borgwardt 2009, Shervashidze et al. 2011) was developed. Properly trained, it still constitutes the state-of-the-art method for many graph classification tasks. The framework is based on the idea of iteratively propagating (node) label information through a graph, leading to a feature vector representation that can be used to assess the dissimilarity of two graphs.

One of the disadvantages of this framework is that its relabelling step, i.e. the step in which subtree patterns are being *compressed*, is somewhat "brittle": labels are only compared with a Dirac kernel, making their dissimilarity a coarse function. Moreover, the subtree feature vector only contains counts of compressed labels and can neither account for their relevance with respect to the topology of the graph nor capture connected components and cycles, both of which are important and interpretable features for characterising graphs (Rieck et al. 2018, Sizemore et al. 2017). We thus propose an enhancement of the original WL stabilisation procedure that uses recent advances in topological data analysis (Munch 2017) to alleviate these issues. Our contributions are as follows:

- We measure the relevance of topological features (connected components and cycles) in graphs and use them to define a novel set of WL subtree features, which we show to be a generalised version of the original ones.
- We develop a topology-based kernel that uses an iterative variant of the WL stabilisation procedure to classify non-attributed graphs.
- We demonstrate that our proposed features perform favourably on a range of graph classification benchmark data sets. In particular, we empirically show that the inclusion of *cycle* information yields classification accuracy improvements over state-of-the-art methods.

*Equal contribution [1]Department of Biosystems Science and Engineering, ETH Zurich, 4058 Basel, Switzerland. Correspondence to: Bastian Rieck <bastian.rieck@bsse.ethz.ch>, Karsten Borgwardt <karsten.borgwardt@bsse.ethz.ch>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

Christian Bock — chrs_bock

Karsten Borgwardt — kmborgwardt

- The Weisfeiler–Lehman algorithm vectorises labelled graphs
- Persistent homology captures relevant topological features
- We can *combine* them to obtain a *generalised* formulation
- This requires a distance between multisets

Digression

A distance between label multisets

Let $A = \{l_1^{a_1}, l_2^{a_2}, \dots\}$ and $B = \{l_1^{b_1}, l_2^{b_2}, \dots\}$ be two multisets that are defined over the same label alphabet $\Sigma = \{l_1, l_2, \dots\}$.

Transform the sets into count vectors, i.e. $\vec{x} := [a_1, a_2, \dots]$ and $\vec{y} := [b_1, b_2, \dots]$.

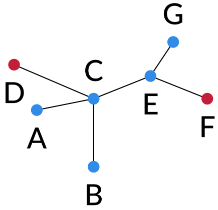
Calculate their *multiset distance* as

$$\mathrm{dist}_{\mathfrak{M}}(\vec{x}, \vec{y}) := \left(\sum_i |a_i - b_i|^p \right)^{\frac{1}{p}},$$

i.e. the p^{th} Minkowski distance, for $p \in \mathbb{R}$. Since nodes and their multisets are in one-to-one correspondence, we now have a metric on the graph!

Multiset distance

Example for $p = 1$



$$
\begin{aligned}
\operatorname{dist}_{\mathfrak{M}}(C, E) &= \operatorname{dist}_{\mathfrak{M}}\left(\{\textcolor{blue}{\bullet}^3, \textcolor{red}{\bullet}^1\}, \{\textcolor{blue}{\bullet}^2, \textcolor{red}{\bullet}^1\}\right) \\
&= \operatorname{dist}_{\mathfrak{M}}([3, 1], [2, 1]) \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
\operatorname{dist}_{\mathfrak{M}}(C, A) &= \operatorname{dist}_{\mathfrak{M}}\left(\{\textcolor{blue}{\bullet}^3, \textcolor{red}{\bullet}^1\}, \{\textcolor{blue}{\bullet}^1\}\right) \\
&= \operatorname{dist}_{\mathfrak{M}}([3, 1], [1, 0]) \\
&= 3
\end{aligned}
$$

Extending the multiset distance to a distance between vertices

Use vertex label from *previous* Weisfeiler–Lehman iteration, i.e. $\mathrm{l}_{v_i}^{(h-1)}$, as well as $\mathrm{l}_{v_i}^{(h)}$, the one from the *current* iteration:

$$\mathrm{dist}_{\mathfrak{V}}(v_i, v_j) := \left[\mathrm{l}_{v_i}^{(h-1)} \neq \mathrm{l}_{v_j}^{(h-1)}\right] + \mathrm{dist}_{\mathfrak{M}}\left(\mathrm{l}_{v_i}^{(h)}, \mathrm{l}_{v_j}^{(h)}\right) + \tau$$

$\tau \in \mathbb{R}_{>0}$ is required to make this into a proper metric. Else, the expression can become zero even though the vertices themselves are *not* equal.

This turns *any* labelled graph into a weighted graph whose persistent homology we can calculate!

Persistence-based Weisfeiler–Lehman algorithm

1. Perform Weisfeiler–Lehman iteration for a number of steps.
2. In each step, obtain a filtration using the vertex distance.
3. Store persistence-based features.

Why does this work?

For graphs, there is a one-to-one mapping between topological features and vertices/edges! Vertices correspond to connected components, while edges correspond to cycles.

Persistence-based Weisfeiler–Lehman feature vectors

Connected components

$$\Phi_{\text{P-WL}}^{(h)} := \left[\mathfrak{p}^{(h)}\left(l_0\right), \mathfrak{p}^{(h)}\left(l_1\right), \dots\right]$$

$$\mathfrak{p}^{(h)}\left(l_i\right) := \sum_{\mathrm{l}(v)=l_i} \operatorname{pers}\left(v\right)^p,$$

Cycles

$$\Phi_{\text{P-WL-C}}^{(h)} := \left[\mathfrak{z}^{(h)}\left(l_0\right), \mathfrak{z}^{(h)}\left(l_1\right), \dots\right]$$

$$\mathfrak{z}^{(h)}\left(l_i\right) := \sum_{l_i \in \mathrm{l}(u,v)} \operatorname{pers}\left(u, v\right)^p,$$

Persistence-based Weisfeiler–Lehman feature vectors

Connected components

$$\Phi^{(h)}_{\text{P-WL}} := \left[\mathfrak{p}^{(h)}\left(l_0\right), \mathfrak{p}^{(h)}\left(l_1\right), \dots\right]$$
$$\mathfrak{p}^{(h)}\left(l_i\right) := \sum_{\mathrm{l}(v)=l_i} \operatorname{pers}\left(v\right)^p,$$

Cycles

$$\Phi^{(h)}_{\text{P-WL-C}} := \left[\mathfrak{z}^{(h)}\left(l_0\right), \mathfrak{z}^{(h)}\left(l_1\right), \dots\right]$$
$$\mathfrak{z}^{(h)}\left(l_i\right) := \sum_{l_i \in \mathrm{l}(u,v)} \operatorname{pers}\left(u, v\right)^p,$$

Bonus

We can re-define the vertex distance to obtain the original Weisfeiler–Lehman subtree features (plus information about cycles):

$$\operatorname{dist}_{\mathfrak{V}}(v_i, v_j) := \begin{cases} 1 & \text{if } v_i \neq v_j \\ 0 & \text{otherwise} \end{cases}$$

A Persistent Weisfeiler–Lehman Procedure for Graph Classification

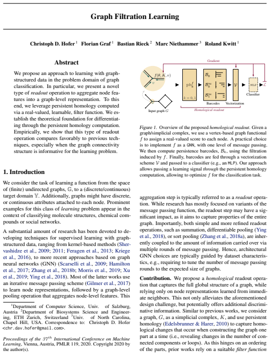
Classification results & summary

	D & D	MUTAG	NCI1	NCI109	PROTEINS	PTC-MR	PTC-FR	PTC-MM	PTC-FM
V-Hist	78.32 ± 0.35	85.96 ± 0.27	64.40 ± 0.07	63.25 ± 0.12	72.33 ± 0.32	58.31 ± 0.27	68.13 ± 0.23	66.96 ± 0.51	57.91 ± 0.83
E-Hist	72.90 ± 0.48	85.69 ± 0.46	63.66 ± 0.11	63.27 ± 0.07	72.14 ± 0.39	55.82 ± 0.00	65.53 ± 0.00	61.61 ± 0.00	59.03 ± 0.00
RetGK*	81.60 ± 0.30	90.30 ± 1.10	84.50 ± 0.20		75.80 ± 0.60	62.15 ± 1.60	67.80 ± 1.10	67.90 ± 1.40	63.90 ± 1.30
WL	79.45 ± 0.38	87.26 ± 1.42	85.58 ± 0.15	84.85 ± 0.19	76.11 ± 0.64	63.12 ± 1.44	67.64 ± 0.74	67.28 ± 0.97	64.80 ± 0.85
Deep-WL*		82.94 ± 2.68	80.31 ± 0.46	80.32 ± 0.33	75.68 ± 0.54	60.08 ± 2.55			
P-WL	79.34 ± 0.46	86.10 ± 1.37	85.34 ± 0.14	84.78 ± 0.15	75.31 ± 0.73	63.07 ± 1.68	67.30 ± 1.50	68.40 ± 1.17	64.47 ± 1.84
P-WL-C	78.66 ± 0.32	90.51 ± 1.34	85.46 ± 0.16	84.96 ± 0.34	75.27 ± 0.38	64.02 ± 0.82	67.15 ± 1.09	68.57 ± 1.76	65.78 ± 1.22
P-WL-UC	78.50 ± 0.41	85.17 ± 0.29	85.62 ± 0.27	85.11 ± 0.30	75.86 ± 0.78	63.46 ± 1.58	67.02 ± 1.29	68.01 ± 1.04	65.44 ± 1.18

Try it out

Graph Filtration Learning

Proceedings of the 37th International Conference on Machine Learning



Graph Filtration Learning

Christoph D. Hofer [1] **Florian Graf** [1] **Bastian Rieck** [2] **Marc Niethammer** [3] **Roland Kwitt** [1]

Abstract

We propose an approach to learning with graph-structured data in the problem domain of graph classification. In particular, we present a novel type of *readout* operation to aggregate node features into a graph-level representation. To this end, we leverage persistent homology computed via a real-valued, learnable, filter function. We establish the theoretical foundation for differentiating through the persistent homology computation. Empirically, we show that this type of readout operation compares favorably to previous techniques, especially when the graph connectivity structure is informative for the learning problem.

Figure 1. Overview of the proposed *homological readout*. Given a graph/simplicial complex, we use a vertex-based graph functional f to assign a real-valued score to each node. A practical choice is to implement f as a `GNN`, with one level of message passing. We then compute persistence barcodes, $\mathcal{B}_k$, using the filtration induced by f. Finally, barcodes are fed through a vectorization scheme $\mathcal{V}$ and passed to a classifier (e.g., an `MLP`). Our approach allows passing a learning signal *through* the persistent homology computation, allowing to optimize f for the classification task.

1. Introduction

We consider the task of learning a function from the space of (finite) undirected graphs, $\mathbb{G}$, to a (discrete/continuous) target domain $\mathbb{Y}$. Additionally, graphs might have discrete, or continuous attributes attached to each node. Prominent examples for this class of *learning* problem appear in the context of classifying molecule structures, chemical compounds or social networks.

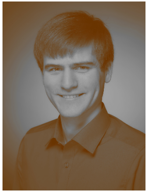
A substantial amount of research has been devoted to developing techniques for supervised learning with graph-structured data, ranging from kernel-based methods (Shervashidze et al., 2009; 2011; Feragen et al., 2013; Kriege et al., 2016), to more recent approaches based on graph neural networks (GNN) (Scarselli et al., 2009; Hamilton et al., 2017; Zhang et al., 2018b; Morris et al., 2019; Xu et al., 2019; Ying et al., 2018). Most of the latter works use an iterative message passing scheme (Gilmer et al., 2017) to learn node representations, followed by a graph-level pooling operation that aggregates node-level features. This aggregation step is typically referred to as a *readout* operation. While research has mostly focused on variants of the message passing function, the readout step may have a significant impact, as it aims to capture properties of the entire graph. Importantly, both simple and more refined readout operations, such as summation, differentiable pooling (Ying et al., 2018), or sort pooling (Zhang et al., 2018a), are inherently coupled to the amount of information carried over via multiple rounds of message passing. Hence, architectural GNN choices are typically guided by dataset characteristics, e.g., requiring to tune the number of message passing rounds to the expected size of graphs.

Contribution. We propose a *homological* readout operation that captures the full global structure of a graph, while relying only on node representations learned from immediate neighbors. This not only alleviates the aforementioned design challenge, but potentially offers additional discriminative information. Similar to previous works, we consider a graph, G, as a simplicial complex, K, and use persistent homology (Edelsbrunner & Harer, 2010) to capture homological changes that occur when constructing the graph one part at a time (i.e., revealing changes in the number of connected components or loops). As this hinges on an ordering of the parts, prior works rely on a suitable *filter function*

[1]Department of Computer Science, Univ. of Salzburg, Austria [2]Department of Biosystems Science and Engineering, ETH Zurich, Switzerland [3]Univ. of North Carolina, Chapel Hill, USA. Correspondence to: Christoph D. Hofer <chr.dav.hofer@gmail.com>.

Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

Christoph Hofer



Florian Graf

Marc Niethammer

MarcNiethammer

Roland Kwitt

rkwitt1982

Digression

Graph neural networks in a nutshell

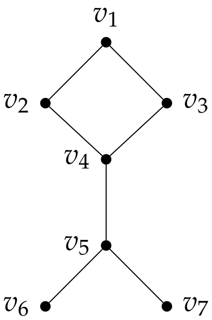
- Learn node representations h_v based on aggregated attributes a_v
- Aggregate them over neighbourhoods
- Iteration k contains information up to k hops away
- Repeat iteration K times

$$a_v^{(k)} := \texttt{aggregate}^{(k)}\left(\left\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\right\}\right)$$
$$h_v^{(k)} := \texttt{combine}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$
$$h_{\mathcal{G}} := \texttt{readout}\left(\left\{h_v^{(K)} \mid v \in \mathfrak{V}_{\mathcal{G}}\right\}\right)$$

This terminology follows the paper *How powerful are graph neural networks?* by Xu et al., presented at the International Conference on Learning Representations 2019.

Digression

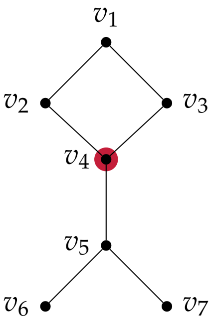
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

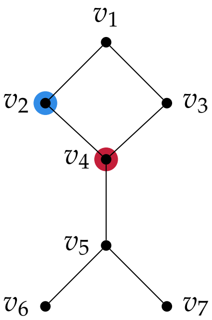
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

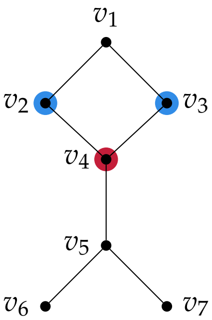
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

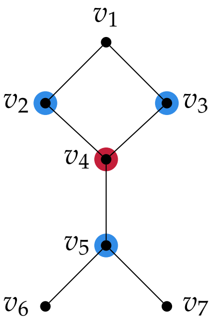
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

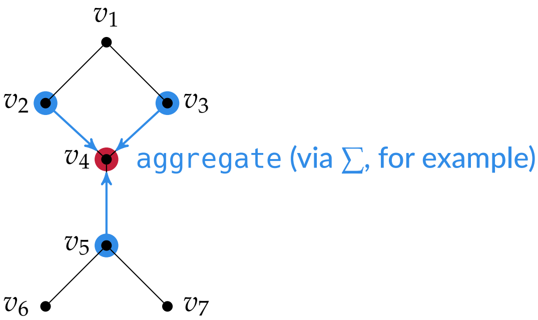
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

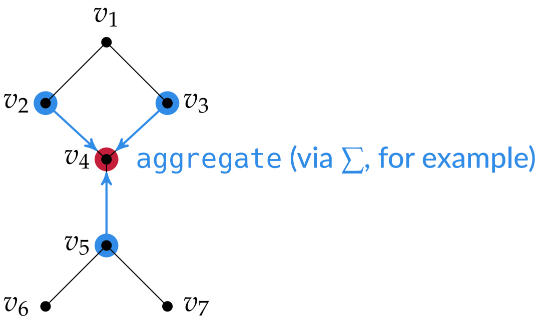
Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Digression

Message passing in graphs



Here, $v_i \in \mathbb{R}^d$ is a d-dimensional attribute vector (use one-hot encoding for labels).

Repeat this process multiple times and update the vertex representations accordingly. Use a `readout` function to obtain a graph-level representation.

Learning graph filtrations

Motivation

- When classifying graphs with TDA, we often employ a *filter function* $f\colon \mathfrak{V} \to \mathbb{R}$, such as $f(v) := \deg(v)$.
- We typically extend f to edges by setting $f(\{u, v\}) := \max\{f(u), f(v)\}$.
- How can we *learn* f end-to-end?
- Crucial ingredient: a *differentiable* coordinatisation scheme!

Learning graph filtrations

Details

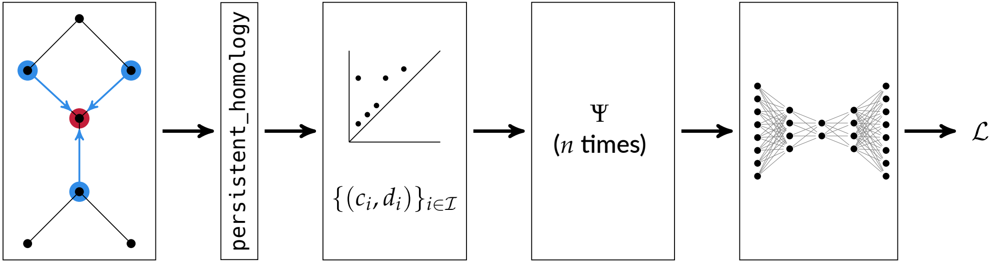
Use a differentiable *coordinatisation* scheme of the form $\Psi\colon \mathcal{D} \to \mathbb{R}$. Letting $p := (a, b)$ denote a tuple in a persistence diagram, we have

$$\Psi(p) := \frac{1}{1 + \|p - c\|_1} - \frac{1}{1 + \operatorname{abs}(r - \|p - c\|_1)},$$

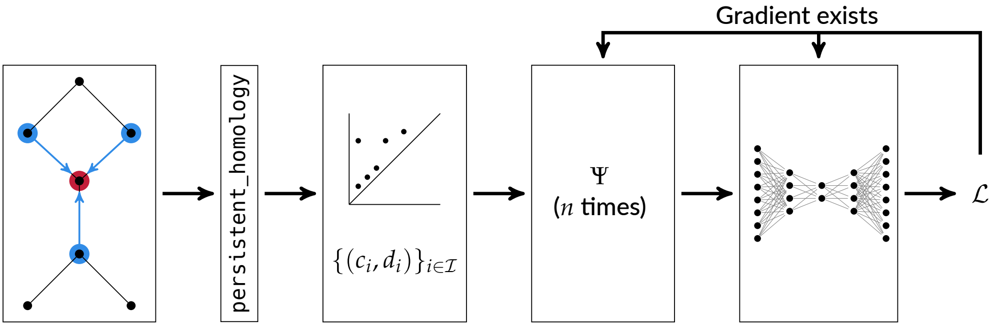
with $c \in \mathbb{R}^2$ and $r \in \mathbb{R}_{>0}$ being *trainable* parameters. The whole diagram is represented as a sum over each individual projections.

Using n different coordinatisations, we obtain a differentiable embedding of a persistence diagram into $\mathbb{R}^n$.

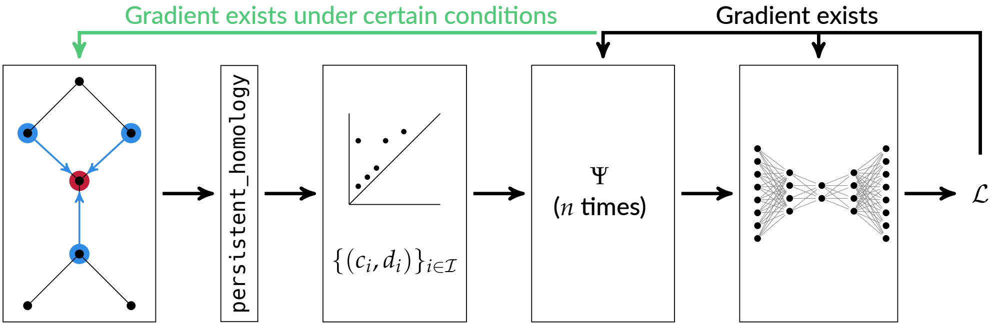
A `readout` function based on persistent homology



A `readout` function based on persistent homology



A `readout` function based on persistent homology



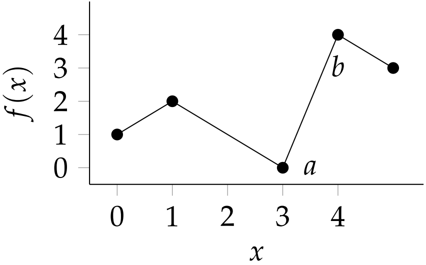
Why does this work?

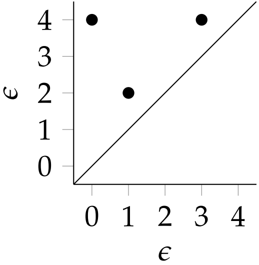
Terminology

- Let $f\colon \mathcal{G} \to \mathbb{R}$ be a function on a graph. Persistent homology can be seen as a map from $(\mathcal{G}, f)$ to $\{(c_i, d_i)\}_{i \in \mathcal{I}}$.
- Let $\mathcal{S}$ be a map from points in the persistence diagram to simplex pairs (vertices and edges), i.e. $\mathcal{S}(c_i, d_i) = (\sigma_i, \tau_i)$. We write $\mathcal{S}(\cdot)$ to denote the map for a single point.
- Depending on the filtration, we can also map a simplex to one of its vertices. For the sublevel set filtration, we have a map $\mathcal{V}$ with $\mathcal{V}(\sigma) := \arg\max_{v \in \sigma} f(v)$.
- Finally, let $\mathcal{P} := (\mathcal{P}_c, \mathcal{P}_d)$, with $\mathcal{P}_c := \mathcal{V} \circ \mathcal{S}(c_i)$ and $\mathcal{P}_d := \mathcal{V} \circ \mathcal{S}(d_i)$.

Why does this work?

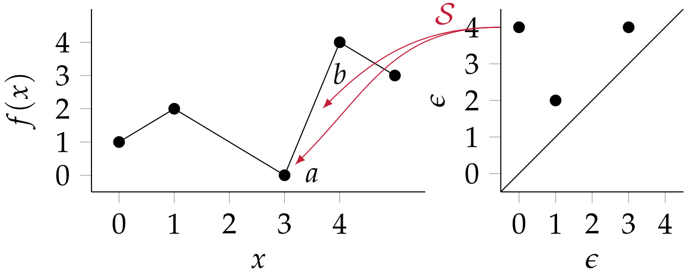
Example





Why does this work?

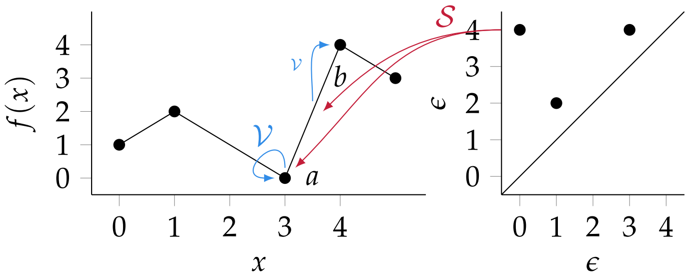
Example



We have $\mathcal{S}(0,4) = (\{a\},\{a,b\})$.

Why does this work?

Example

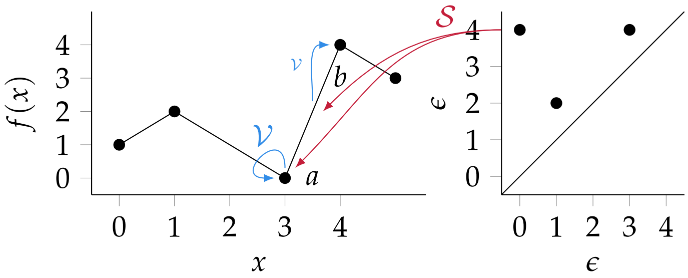


We have $\mathcal{S}(0,4) = (\{a\},\{a,b\})$.

We have $\mathcal{V}(\{a\}) = x_3$ and $\mathcal{V}(\{a,b\}) = x_4$.

Why does this work?

Example



We have $\mathcal{S}(0,4) = (\{a\},\{a,b\})$.

We have $\mathcal{V}(\{a\}) = x_3$ and $\mathcal{V}(\{a,b\}) = x_4$.

We have $\mathcal{P}(0,4) = (\mathcal{V} \circ \mathcal{S})(0,4) = (x_3, x_4)$.

Why does this work?

Gradient calculation sketch

- If the function values are *distinct*, then $\mathcal{P}$ is *unique*.
- If the function values are *distinct*, then $\mathcal{P}$ is *constant* in some neighbourhood.

Assume that f depends on $\theta = (\theta_1, \theta_2, \ldots)$. We then have $f(\mathcal{P}_c(c_i)) = c_i$, and, since $\mathcal{P}$ is constant,

$$\frac{\partial c_i}{\partial \theta_j} = \frac{\partial f(\mathcal{P}_c(c_i))}{\partial \theta_j} = \frac{\partial f(v_i)}{\partial \theta_j} = \frac{\partial f}{\partial \theta_j}(v_i),$$

i.e. the partial derivative is equivalent to the derivative of the function evaluated at the image of the map $\mathcal{P}_c$.

This formulation and proof is due to the paper *Topological Function Optimization for Continuous Shape Matching* by Poulenard et al., which appeared in Computer Graphics Forum, Volume 37, Issue 5, pp. 13–25.

Graph Filtration Learning

Obtaining a filter function f

Use a single GIN-ϵ layer with one level of message passing (`1-GIN`) with hidden dimensionality 64, followed by a two-layer MLP.

`GIN-1`, $h = 64$ $\longrightarrow$ `MLP`$(64, 64, 1)$ with sigmoid activation

Hence, $f\colon \mathfrak{V} \to [0, 1]$.

We can initialise f using the vertex degree or uniform weights (plus a symbolic perturbation to ensure gradient existence).

Graph Filtration Learning

Practical results & summary

Method	IMDB-BINARY	IMDB-MULTI
1-GIN (GFL)	74.5$\pm$4.6	49.7$\pm$2.9
1-GIN (SUM)	73.5$\pm$3.8	50.3$\pm$2.6
1-GIN (SP)	73.0$\pm$4.0	50.5$\pm$2.1
Baseline	72.7$\pm$4.6	49.9$\pm$4.0
PH	68.9$\pm$3.5	46.1$\pm$4.2

Try it out

- We can *learn* a scalar-valued filtration function in an end-to-end fashion.
- The `readout` function integrates nicely into existing architectures.
- Predictive performance is better than 'raw' persistent homology (with only a single level of message passing).

Summary

- Topological features improve graph classification tasks.
- Recent advances make persistent homology *differentiable*!
- This is only just the beginning; need to handle higher-dimensional features, different filtrations, and much more...
- The future belongs to *hybrid* 🐍🦁 models!

❤ My co-authors Christian Bock, Karsten Borgwardt, Max Horn, Roland Kwitt, and Michael Moor for providing feedback, illustrations, and a plethora of high-quality discussions!