WASP Software Engineering Course Module 2023: Assignment
Isaac Ren

Topological Data Analysis (TDA) is a field of research in mathematics that applies theoretical results from algebraic topology to real-life data. Topology deals with the shape of mathematical spaces, ignoring more structured information such as distances and smoothness, and algebraic topology quantifies the spaces' features using algebraic objects: homotopy groups, homology spaces, cohomology rings, etc. These objects are called topological invariants and can be more easily encoded in data structures than the topological spaces themselves, and thus lend themselves to real-life applications. In particular, homology spaces are vector spaces that can be represented by their dimension or by a list of basis vectors, and are the main component of the most common method in TDA: persistent homology. The general setup is as follows. Consider a dataset in a high-dimensional space. Given a threshold number r, insert edges for each pair of vertices that are at most r away from each other, triangular facets for triples of vertices at most r away from some common point, etc. This defines a simplicial complex, depending on r, whose homology varies as a function of r. By studying how the homology spaces of this complex vary for increasing values of r, we get an idea of what global topological features the dataset has, even if the original data is noisy. More generally, studying graphs that arise from data, such as crystals, proteins, or co-authorship, to name a few, and in particular studying graph invariants such as eigenvalues, can be considered a part of TDA. This gives an alternative or complementary approach to analyzing data: while traditional statistical methods such as principal component analysis extract linear information from data, TDA only considers the shape of the dataset, with the hope of revealing nonlinear features that would not have been detected otherwise. Research in TDA consists both of proving theoretical results of stability and robustness for their methods, as well as working with researchers in other domains to apply TDA methods to their datasets.

In this essay I will discuss the following observations. TDA is used in machine learning, and more generally is applied in smaller settings where the Agile manifesto and other similar principles are more easily followed. One subdomain of TDA is the definition of metrics on complex spaces, which is relevant when doing boundary value testing. TDA can also be applied to sentiment analysis as an alternative method. Recent interactions between TDA and (behavioral) software engineering include studying knowledge transmission on GitHub and automatically determining input specifications of parsers. I will conclude with some remarks about public communication of software engineering and machine learning, as well as some personal thoughts on explainable AI.

TDA has been implemented in machine learning pipelines, most notably with the construction of differentiable topological loss functions: see implementations such as `TopologyLayer` [4] and `torch_topological` [5] for Python. An example of such a loss function is one that maximizes one-dimensional persistent homology, that is, the size of empty holes in a dataset. Focusing on zero-dimensional persistent homology allows neural networks to use information about the connectivity of the input dataset, as zero-dimensional homology counts the number of connected components. The features detected by topological machine learning can be detected by other, more traditional methods, but might be slower or more inaccurate. If the engineer working on the neural network has some functionality in mind, if they wanted the program to look at connected components or ring-like shapes, then it would be reasonable for them to consider TDA. This is one approach to verification in machine learning, which has not been really formalized: TDA can help if the engineer specifies some topologically interesting output.

TDA is a relatively new field, having been introduced at the beginning of the twenty-first century. Thus its applications in other domains remain relatively small. At KTH, the TDA group has had collaborations in medical research with teams at Karolinska Institute (pertaining to brain diseases, in particular Parkinson's and Alzheimer's) and the Institute of Science and Technology Austria (classification of microglia in mice). In these relatively small collaborations, there is not much of the technical bureaucracy

that can be found in well-established software companies, and so the principles of the Agile manifesto can be respected. The collaborators can all communicate with each other about their specific skillset and knowledge, and the methods of data analysis can be readily adapted to changes in the available data, the observed results, and discussions. Since the situations described here are specific experiments in academic research, and oftentimes the tools are used by only one person, there is very little work devoted to documentation. The specifics of the implementation of TDA concepts are usually glossed over in reports and publications, and the tools are overly specified to the problem at hand. In this sense, while the software works for the time being, we neglect the technical debt that we incur, and thus make future work more difficult. Being in research, we can hope that new theoretical developments, such as new topological invariants or algorithms, will make past work obsolete, but from the point of view of software engineering this is not ideal. There do exist larger groups more focused on creating good TDA tools for software engineering, where documentation is much better: aside from the machine learning tools cited above, there is tda-giotto [9] and Gudhi [6].

Boundary value testing for programs depends on a choice of distance for the input and output of a given program. When strings are involved, for instance, we can use the difference in length of strings. More generally, there are methods based on Kolmogorov complexity [3]. Once the spaces of inputs and outputs are equipped with metrics, we can compute the differential of programs. TDA pipelines are programs to which these methods can be applied, and the metrics used in TDA can be considered for other boundary value tests for appropriate data. Specifically, in TDA, metrics between point clouds are extensively studied and effectively computed: for instance, Wasserstein distances, which is inspired by optimal transport theory. These metrics allow us to observe how persistent homology varies when varying the input dataset [2], and so we can identify boundary cases where persistent homology changes drastically with a small variation of the input.

Semantic and sentiment analysis is another field where TDA can contribute as an alternative form of data analysis. After embedding words into a high-dimensional space, using a method like word2vec, which embeds words as vectors of its neighboring words, we can use a topological tool called Mapper [7] which will produce a graph where related words are close together. This can be used for sentiment analysis in technical domains, where general methods are insufficient: indeed, specific fields such as software engineering use language differently, and with different connotations, than general language. For instance, the word “kill” in software engineering has a roughly neutral connotation, since it usually refers to ending a computer process, while in colloquial English it has a strong negative connotation. This discrepancy can be captured in a Mapper graph produced from SE-specific text. In this graph, the word “kill” will have neighbors such as “exit” and “suspend,” and thus can inform the sentiment classifier of the actual sentiment of the word.

Behavioral software engineering (BSE) is at the intersection of psychology and software engineering: it studies psychology in the context of SE, and studies SE with the explicit assumptions that people, including programmers, are not rational and are affected by many factors. BSE also looks at group behavior, since software development is rarely a lone effort. One of the main concepts in BSE is bias. In software engineering, bias appears on all levels. At the individual level, there are cognitive biases, such as confirmation bias, anchoring bias, and availability bias. As a consequence, individual engineers may hold inaccurate beliefs about the correctness of their code or ignore potential ways of changing their code when asked to review it. On a group level, there are various dynamics at play. Groups develop in stages that start with emotion and relationship building before arriving at a truly productive state. In groups, the notion of diversity emerges: having a diverse group, not only in socioeconomic terms but also in beliefs and technical background, is crucial for a healthy working environment. Finally, on an organizational level, company culture and politics affect software development. For instance, employee turnover rate will determine how well goals are communicated to senior members and recent hires. Consideration of BSE leads to new guidelines for software engineers that focus more on human needs, such as the Agile manifesto. Being more conscious of engineers’ mental state leads to more satisfying work and better

teams. Even if this is not the most productive or efficient for hard metrics such as number of commits or development time, psychologically aware software engineering is more durable in the long run. One instance where TDA was used to study BSE is the article "A topological analysis of communication channels for knowledge sharing in contemporary GitHub projects" by Tantisuwankul et al. [8]. This paper looks at the different ways of communication on GitHub, including GitHub pages, changelogs, licences, etc. For a given project, the existence and volume of communication through various channels is embedded in a high-dimensional space, and then the authors use the software Mapper to produce a human-readable graph that clusters similar projects together. The paper concludes that projects use different communication channels depending on age and popularity.

Requirements engineering is one of the first steps in software development, where engineers specify the context in which the software will work: what inputs will it consider and what outputs it will produce. One specific way of evaluating the performance of the software is through boundary value testing, which will search for the boundary between expected and unexpected behavior. When the code is available and simple to read, one can do a formal inspection in order to check that the requirements are satisfied. However, for more complex software such as deep neural networks, this method is not possible, so we need more automated methods. In the manuscript "Topological differential testing" by Ambrose, Huntsman, Robinson, and Yutin [1], the authors test the input specifications of various parsers meant to recognize PDF. This is of interest, for instance, in cybersecurity, where we would like to avoid improperly formatted files that may contain viruses. The authors take a topological approach: for each file, they determine the subset of parsers that validate the file, and then for each subset of parsers, they count the number of files validated only by those parsers. They then construct the simplicial complex whose vertices are the parsers and whose simplices are the subsets of parsers such that the number of files validated only by the subset is greater than the number of files validated by any of its subsets. They call this simplicial complex the *de facto* file format, with the idea that this structure describes the true file format.

Software engineering is unavoidable in the modern world. Its presence is not always clear in the collective conscience, although concrete examples such as large language models and diffusion models, which produce easily understandable text and images, are making AI research more well known to the general public. Despite the fact that software underlies almost everything in developed societies, the actual inner workings of software and software engineering are not properly communicated to the general public. It's known that the representation of programmers in popular media is dramatized and deformed into fictional tropes such as the lone hacker, which undermine the inherently social nature of modern software engineering. When people outside of SE are unaware of the culture withing software engineering, the inattention can lead to more toxic environments. Conflicts and harassment can be left unresolved because society does not value the social aspects in software companies. By presenting software engineering as a group effort, with all of the interpersonal dynamics that come with it, we can give the general public a better representation of software development and improve standards for SE teams.

One of the selling points of TDA is that it produces tools for visualizing data. For instance, the Mapper algorithm groups together (topologically) similar points in a higher dimensional space and returns a graph in two or three dimensions. This sort of explainable data analysis is also crucial for the future of AI and machine learning. For instance, I would love to see a large language model that trains while keeping explicit track of where it draws its text from: in some sense, it generates sources along with the text, except that these sources are real and not hallucinated. This is not a rigorous idea, since the model will choose expressions that it sees repeatedly in many different contexts and so the idea of a single source text is not really correct, but if such a language model could cite facts that it states, it could truly become a good education tool. More abstractly, one could ask a generative model to produce some text or art of a certain genre, and when the user is interested in a certain aspect, the program could cite real authors and artists that correspond to that aspect of the genre. This would of course also address the ethical and legal problem of stealing artistic works without proper credit. In any case, deep neural

networks are the most successful tools of modern software engineering and are likely to stay around for the next decade, affecting daily life in more and more deep ways. It is therefore important for the limitations and pitfalls of SE to enter the public conscience and actively address ethical problems within the field.

References

[1] K. Ambrose, S. Huntsman, M. Robinson, and M. Yutin. Topological differential testing, 2020.

[2] M. Carriere, F. Chazal, Y. Ike, T. Lacombe, M. Royer, and Y. Umeda. Perslay: A neural network layer for persistence diagrams and new graph topological signatures. In S. Chiappa and R. Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 2786–2796. PMLR, 26–28 Aug 2020.

[3] F. Dobslaw, F. de Oliveira Neto, and R. Feldt. Boundary value exploration for software analysis. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 346–353, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[4] R. B. Gabrielsson, B. J. Nelson, A. Dwaraknath, and P. Skraba. A topology layer for machine learning. In S. Chiappa and R. Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 1553–1563. PMLR, 26–28 Aug 2020.

[5] F. Hensel, M. Moor, and B. Rieck. A survey of topological machine learning methods. *Frontiers in Artificial Intelligence*, 4, 2021.

[6] C. Maria, J.-D. Boissonnat, M. Glisse, and M. Yvinec. The gudhi library: Simplicial complexes and persistent homology. In H. Hong and C. Yap, editors, *Mathematical Software – ICMS 2014*, pages 167–174, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[7] G. Singh, F. Memoli, and G. Carlsson. Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition. In M. Botsch, R. Pajarola, B. Chen, and M. Zwicker, editors, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2007.

[8] J. Tantisuwankul, Y. S. Nugroho, R. G. Kula, H. Hata, A. Rungsawang, P. Leelaprute, and K. Matsumoto. A topological analysis of communication channels for knowledge sharing in contemporary github projects. *Journal of Systems and Software*, 158:110416, 2019.

[9] G. Tauzin, U. Lupo, L. Tunstall, J. B. PÃ©rez, M. Caorsi, A. M. Medina-Mardones, A. Dassatti, and K. Hess. giotto-tda: : A topological data analysis toolkit for machine learning and data exploration. *Journal of Machine Learning Research*, 22(39):1–6, 2021.