TIME SERIES ANALYSIS FOR BOTNET DETECTION

by

Taylor Henderson
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

_____	Dr. Robert Simon, Thesis Director
_____	Dr. Eric Osterweil, Committee Member
_____	Dr. Jessica Lin, Committee Member
_____	Dr. David Rosenblum, Chairman, Department of Computer Science
_____	Dr. Kenneth Ball, Associate Dean for Research and Graduate Studies

Date: _____ Fall Semester 2020
George Mason University
Fairfax, VA

Time Series Analysis for Botnet Detection

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Taylor Henderson
Bachelor of Science
George Mason University, 2018

Director: Dr. Robert Simon, Professor
Department of Computer Science

Fall Semester 2020
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my fiancée Kirsten Van Nortwick. Thank you for always supporting me.

Acknowledgments

I would like to thank Micheal Brown for his leadership and being a wonderful mentor. I would also like to thank Dr. Simon for his outstanding guidence and mentorship throughout this process.

Table of Contents

List of Tables

List of Figures

Abstract

TIME SERIES ANALYSIS FOR BOTNET DETECTION

Taylor Henderson

George Mason University, 2020

Thesis Director: Dr. Robert Simon

Cyber attacks are becoming more prevalent and increasingly threaten to have devastating consequences. Many of these attacks use multiple distributed devices controlled by a single person or group from a remote location, and are commonly referred to as botnet attacks. There are multiple reasons why botnet detection is challenging. First, botnets use covert communication measures and actively attempt to mask their communication. Second, the command and control (C&C) of these devices may not come from a single source but instead from peer to peer (P2P) bot communication. Third, network traffic is inherently very noisy and has high dimensionality both in the data's continuous nature and the number of variables. Finally, massive botnet data collections are generally incomplete, and real-world data is challenging to find. These factors complicate performing botnet data analytics through well-known approaches, such as time series analysis techniques.

Recent results in topological data analysis (TDA) have shown great promise in analyzing noisy, large scale, and incomplete time series data sets. This thesis explores using TDA persistence landscapes (PL-TDA) to transform a multi-attribute time series into a single attribute time series, which can then be analyzed using existing time series/data mining techniques. We first perform a robustness analysis on existing PL-TDA computational

methods in the presence of noise. We then propose an algorithm using plane-sweeping methods that decrease the PL-TDA runtime. This algorithm utilizes another result that demonstrates a linear-time approach to finding the top landscapes that appear when PL-TDA is computed. Following that we show how to implement a processing pipeline for PL-TDA on botnet data. Finally we show that our new algorithms maintain accuracy while decreasing runtime. This work assists future network and systems researchers by giving them a new technique to effectively process network traffic analysis capturing the inherent topological properties.

Chapter 1: Introduction

Cyber attacks are becoming more prevalent and increasingly threaten to have devastating consequences. Many of these attacks use multiple distributed devices controlled by a single person or group from a remote location and are commonly referred to as botnet attacks. The consequences of these attacks include possible interference in the 2020 [1] United States election. Additionally, botnets are used to steal banking information, deny access to major websites, and other criminal activity at scale [2][3].

There are multiple reasons why botnet detection is challenging. First, botnets use covert communication measures and actively attempt to mask their communication. This creates an arms race that pushes the generalization of machine learning models to the max. Second, the command and control (C&C) of these devices may not come from a single source but instead from peer to peer (P2P) bot communication. As a result, both device and network-wide communication patterns must be analyzed in real-time to detect communication patterns and act before the malicious actions have already taken place. Third, network traffic is inherently very noisy and has high dimensionality in the data's continuous nature and the number of variables. Although this is a large amount of information to learn off of, it is unwieldy for most models. The data must be filtered and compressed, extracting the critical features and reducing dimensionality for real-time detection. Finally, massive botnet data collections are generally incomplete, and real-world data is challenging to find. Useful data is imperative in order to train a model. Without good data, even the best models will have no chance in the wild. These factors complicate data mining and data analytics tasks that rely on well-known approaches, such as time series analysis techniques.

Topological data analysis is the study of techniques from algebraic topology applied to real-world data. Essentially it works by mapping geometric structures into algebraic constructs that can be computationally measured. It is concerned with quantifying the

persistent homology of point clouds. Persistent homology measures the importance of n-dimensionality "holes" in a point cloud. It works by measuring topological properties at different resolutions. These techniques allow a deeper understanding of the data and help machine learning models interpret the data. Recent results in topological data analysis (TDA) have shown great promise in analyzing noisy, large scale, and incomplete data sets in computer systems and networking [4], financial early warning signs [5], and time series motion datasets [6]. These papers use persistence landscapes from topological data analysis (PL-TDA) to compare time window to time window changes by computing the difference of the L^p norms of the persistence landscapes. This technique tracks persistent homology evolution over time and uses this new feature space of topological changes for classification.

1.1 Research Contributions

This thesis explores using TDA persistence landscapes to transform a multi-attribute time series into a single attribute time series. This is done to take advantage of univariate time-series analysis and data mining techniques. Experiments are conducted to see if this TDA technique has the power to reduce the botnet detection problem to a smaller dataset while still performing competitively.

For brevity, we refer to Topological Data Analysis using Persistence Landscapes as PL-TDA. For PL-TDA to work well, the hyperparameters, such as window size and maximum rips radius, must be correctly chosen. It is also critically important that it is well understood how robust persistence landscapes are in the presence of noisy data. We experimentally explore this issue.

We also present a novel output-sensitive algorithm to compute the persistence landscapes. An output-sensitive algorithm means that the input size N and the output size M effect time complexity. This results in minimal information being computed that is not of interest to the end-user. It can be shown with the provided algorithm that it will only compute a constant amount of extraneous work in proportion to each of the topological features, measured in birth-death pairs, that appear in the output. A substep that is needed

to get this to work is a method to determine if a given birth-death pair will ever appear in the top k persistence landscapes. A proof for a linear-time method to accomplish this is described. This algorithm can be used with any persistence landscape generation method.

Finally, a critical element in any time series analysis is the steps necessary to (pre)process existing data sets into a form that can be analyzed. We describe how to accomplish this. Using our processing pipeline enables us to experimentally evaluate the strengths and weaknesses of our proposed algorithms. Experimentally analyzing a large scale data set consisting of over 500,000 net flows containing 16 different types of botnets, we determined the strength of our approach.

This work assists future network and systems researchers by giving them a new technique to process network traffic, capturing the inherent topological properties effectively. We also provide insight into the robustness of persistence landscapes in the presence of noisy data.

1.2 Thesis Roadmap

Chapter 2 provides an overview of the necessary topological and machine learning knowledge to understand the persistence landscape problem and how it can be applied as a dimensionality reduction technique for botnet detection. This chapter also summarizes the relevant recent advances in both PL-TDA and machine learning for botnet detection. Chapter 3 examines the robustness of PL-TDA under various types of noisy data. Chapter 4 introduces a novel output-sensitive algorithm to the k persistence landscape generation problem. Also, a novel property of persistence barcodes allows efficient filtering of birth-death pairs for the k persistence landscape problem. Chapter 5 provides an overview of the experimental setup and environment for the experimental validation of the thesis, including the data processing pipeline. Lastly, Chapter 5 also provides the details of the experiments run and the significance of the results. This includes experimental validation of the novel algorithms from Chapter 4 and intuition for how to set the hyperparameters for the k persistence landscape generation problem.

Chapter 2: Background and Related Work

This Chapter provides a brief background in the techniques underlying Topological Data Analysis and Machine learning. We also present related work in this area.

2.1 Topology and Topological Data Analysis

We now describe basic topological structures and concepts. It is not an exhaustive explanation of topology or abstract algebra. For a rigorous mathematical introduction to the topic, one should turn to [7].

Topological data analysis (TDA) is a group of techniques that take advantage of a dataset's topology to gain further insight into the data. It was proposed to deal with datasets that are noisy, incomplete, and have high dimensionality. Recently algorithms for efficiently computing meaningful representations for point clouds have been introduced for statistical learning. The TDA approach this thesis uses is persistent homology. Specifically, persistent homology is represented by persistence landscapes. The choice of persistence landscapes over other representations is because it is relatively easy to compute and is easy to visualize and represent, which causes it to lend itself to machine learning applications. It is stable under perturbations and has a Banach space structure[8]. Persistent homology captures the geometry of a point cloud by quantifying the shape and lifetime of the various n-dimensional "holes" within the point cloud.

2.1.1 Overview

TDA comes from a branch of mathematics called algebraic topology. Algebraic topology studies topological spaces using tools from abstract algebra, where abstract algebra is the study of algebraic structures such as rings, fields, and vector spaces. Topological spaces are

a set of points, each having a neighborhood of points where the neighborhood satisfies a set of axioms. By studying a topological space, one can gain insight into the structure of the space from which the points were sampled.

Simplices and Simplicial Complex The starting point in TDA is the analysis of simplicial complexes. Informally simplices generalize the notion of line or triangle to n dimensions. More formally, a n-simplex is a polytype of n dimensions defined as the convex hull of its $n+1$ vertices. A simplicial complex is similar to a graph where each subgraph of n points that is a complete graph creates an $n-1$ dimensional simplex. For each complete graph, G in the simplicial complex C, each subgraph of G is also a complete graph that forms its own simplex, which is also apart of C. Simplices in a simplicial complex are connected along the shared lower-dimensional simplices. If no shared lower-dimensional simplices exist between two simplices, then they are not connected.

K-Skeleton The k-skeleton is a subset of a simplicial complex where only simplices up to a dimention n are kept. This is used when a specific type of topology is in question. When it comes to choosing the k-skeleton to compute persistence over, most of the literature agrees that when using the rips filtration on time series data, the 2-skeleton makes the most sense. When we use the 2-skeleton, it captures holes caused by loops in the time series. This allows the model to determine how the time series' cyclic structure changes over time and is used in [4] to get their promising results. Others have had success using other filtration processes, such as the lower star filtration, which [9] showed promising results when classifying the instruments found in an audio track.

Filtrations Filtrations play a central role in TDA. A filtration is a set of subobjects of some algebraic structure. This set is indexed and totally ordered. Filtrations extract topological properties from a point cloud. Filtrations accomplish this by translating a point cloud into an ordered set of simplicial complexes. One standard filtration used in topological data analysis is the Vietoris-Rips filtration. The Vietoris-Rips filtration, commonly

known as the rips filtration, uses a point cloud's geometric properties to derive topological properties. If one had a collection of points, one of the simplest ways to translate it into a simplicial complex would be to start by defining a threshold t and a distance function $d(x, y)$. Then all $s \subseteq S$, where $\forall x, y \in s, d(x, y) < t$ form a simplex in the simplicial complex. The Vietoris–Rips first sets the threshold to 0 and then increases it up to a max rips radius r. Whenever increasing the rips radius causes a change in the simplicial complex, the transformation between the two complexes is recorded. This results in an ordered set of simplicial complexes/transformations between complexes that encode a given point cloud's geometry.

2.1.2 Presistant Homology and Persistent Cohomology

Persistent homology and cohomology are both used in TDA and, in most cases, produce the same results when their results are represented in a persistence representation. The details of how these are calculated are unnecessary from a practitioner's perspective: though, it is essential to have an intuitive understanding of how these concepts work.

Persistent homology and cohomology transform an ordered set of simplicial complexes into birth-death pairs. These birth-death pairs represent the first and last threshold from the filtration that a topological property was observed during the filtration, respectively. Persistent homology and cohomology can be computed in various dimensions where the dimension determines the type of topological objects that are being tracked. For example, in 0-d persistent homology, connected components are being tracked. In 1-d persistent homology 2-d holes are being tracked, and n-d persistent homology (n-1)-d voids are being tracked. It should be noted that some features may have a death of inf if the feature still exists at the end of the filtration.

2.1.3 Presistence Representations

There are multiple ways to visualize the birth-death pairs. Barcodes are made by creating a number line of the threshold value. Then for each birth-death pair, a line is created

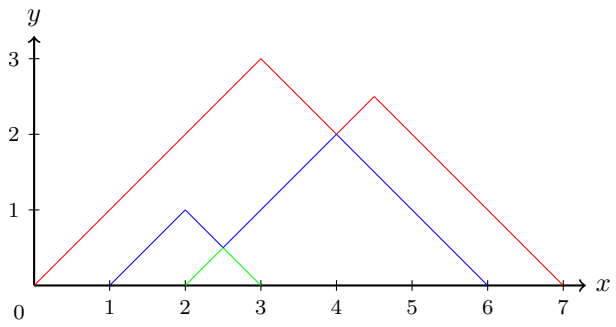

Figure 2.1: Presistance Landscape. λ_1 is red, λ_2 is blue, and λ_3 is green.

starting at the birth and ending at the death. Usually, these lines are drawn not to intersect each other to create a more pleasing visualization. Another persistence representation is the persistence diagram, where the birth-death pairs are plotted in 2-d. The persistence representation used in this thesis is the persistence landscape.

Presistance Landscapes

To compute the persistence landscape, we take the birth-death pairs, and for each pair, we create 2 line segments. The set of both of these line segments is a persistence mountain. The first line segment will start at the (b, 0) where b is the birth of that pair and ends at (((d-b)/2)+b, (d-b)/2). The second line segment will begin at (((d-b)/2)+b, (d-b)/2) and ends at (d, 0). Suppose a homology feature has a greater "lifespan" (the difference between its birth and death). In that case, its mountain in the persistence landscape will peak higher than other homology feature with a shorter "lifespan. This quantifies a feature's importance overall and at a specific threshold by observing the height of its mountain. Once we create all of these line segments and plot them on the same graph, we have a persistence landscape. We can then define the λ_k landscape as the set of k-max values at all thresholds in the diagram. If we take the λ_1 landscape, we can then think of this value as being the highest strength of any homology feature at every threshold. The λ_2 landscape would be the "ridge-line" if all the points in λ_1 were removed. An example can be seen in Figure 2.1

The persistence landscape is used because it is stable under perturbation as well as

exist's in a Banach or Hilbert space.

2.2 Machine Learning

Here we provide a basic overview of general machine learning techniques.

2.2.1 Metrics

The metrics used to assess the success of a given model are very important. Choosing the wrong metric to report can bias the results for specific problems. Two metrics that are widely accepted for classification and anomaly detection are the F1 measure and MCC.

F1 Measure The F1 measure, or just F1, handles unbalanced classes by using the harmonic mean of the precision and recall as seen in (2.1). Precision is the true positives divided by the number of inputs labeled positive, and recall is the true positives divided by the number of positive inputs. The F1 ranges from 0 to 1.

$$F_1 = \frac{percision * recall}{percision + recall} \tag{2.1}$$

MCC The MCC is a technique that combines the false positive (FP), false negative (FN), true positive (TP), and true negative (TN) into a single value by taking the geometric mean of all the values as seen in (2.2). This creates a more robust value that is less biased for unbalanced classes. The MCC ranges from -1 to 1.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{2.2}$$

2.2.2 Feature Engineering

Feature engineering is transforming raw input features to make it easier for the model to learn. Two standard feature engineering techniques are used in our data processing pipeline.

Minmax Normalization

Minmax normalization scales all numerical fields to be on the same scale (0-1). This is done by subtracting the minimum value of a feature from every value of that feature and then dividing by the maximum minus the minimum value of the feature. Normalization ensures that a feature's scale does not influence its significance in the model. This is a concern because TDA uses distance metrics to calculate the topology, and a feature being disproportional large compared to its actual significance will hurt the model.

One-hot Encoding

One-hot encoding finds numerical representations of categorical data enabling the use of euclidean distance. The method works by transforming a categorical feature f which takes on n possible values into n features, one for each of the possible values of f. The value for all of the new features is set to 0. If the value for feature f of a sample s takes on the value a k, then the new feature f_k which encodes the feature value k is set to 1 for sample s. We use one-hot encoding on protocol type and flow direction for our experiments.

2.2.3 Generalization

Generalization is the primary goal of machine learning. It is the objective of creating a model that works well with data that is not seen during training. This happens when the model captures the data's true underlying nature and not overfitting to the training samples. Generalization can be assessed through proper testing training split, where new samples are seen in testing that were not seen in training, and by reporting proper metrics on the model's performance.

2.2.4 Classification

Classification in machine learning is the task of predicting a data's class label given a set of examples. These labels usually come from a finate set. This thesis deals with a two class classification problem: labeling each time-series as malicious or benign.

2.2.5 Datasets

Choosing datasets to test the method on is very important, and there are many pitfalls. Datasets can introduce unforeseen bias in the results. When creating a classification model, the number of examples of each class must be considered, especially if the ratio of classes is heavily skewed. This will happen in our problem if we have significantly fewer examples of malicious traffic vs. regular traffic. We do not want a model that is less likely to predict malicious just because we have a poor balance in our dataset. Ideally, each class in the dataset will have the same amount of examples. If there are C classes, we want $1/C\%$ of the total examples dedicated to each class. For example, if we have ten classes, then each class would ideally account for 10% of the total data. This problem can be addressed using a metric that accounts for unbalanced datasets, such as using the F1 or MCC instead of standard accuracy.

Along this line, we also must ensure that our dataset is representative of what would be seen in the wild if this model were to be implemented in practice. The problem of any network classification task is inherently extremely noisy as users run multiple different types of applications simultaneously. Each of these applications will behave differently, and the actions of the end-users and applications are incredibly complex. This is extremely difficult to emulate in a synthetic dataset. Thus, having a dataset captured on a live network with real users is the only reliable way to ensure proper representation of regular traffic. Unfortunately, for privacy reasons, it is hard to find public datasets that have this attribute.

Another factor for the realism of the dataset is that bad actors are actively developing new botnets. As such, it is a near certainty that our model will interact with botnets that it has never seen before. This is similar to the generalization required of machine learning models in other application fields but even more difficult. Normally one would train a model to generalize to new examples of the classes that have been seen in the training dataset. This is similar to being able to correctly classify examples of specific botnet implementations, which the model has already seen in training. On top of this, in botnet detection, the model also attempts to correctly classify new implementations of botnets of which use

similar protocols that have been seen before. Such as a new type of botnet that runs on top of HTTP. We though attempt to classify botnets for which we have seen nothing like it in the past: whether it runs on a new protocol or distributes tasks from the master to the bots in a new way. This is a challenging task as it involving gaining a deep understanding of what it means to have remote, covert command and control (C&C) over a collection of devices. Ideally, in a realistic dataset, we would have instances of new types of botnets and train with and without these new types of botnets to quantify how our model generalizes in these different ways.

It is also essential to consider what datasets others are currently using in the field. Even if a better dataset exists, if others in the field are using a possibly worse dataset, it is crucial to run one's model through this dataset to compare to other models in the field.

Lastly, there are many different formats to provide a network dataset. A common method is to use a pcap file. Pcap is a file format that captures network data that goes across a wire. It is a verbose way to capture networking data and is the gold standard in the field. Similar to a pcap file is argus data. Open argus is an open-source project out of Georgia Tech designed to read from a network interface or a file and extract valuable statistics that can be ingested into network monitoring devices and machine learning models [10]. It can compute 145 different attributes from the raw network data. It addresses many of the privacy concerns of providing a raw network capture. The drawback of accessing data in this format is that the dataset provider is in charge of much of the feature engineering. Thus if one wishes to explore a feature that the dataset provider did not calculate using argus, it can be impossible to recover such information.

Open argus was chosen for this project when pcap data is used as it is easy to work with and provides a wealth of features to choose from during the feature selection process with no added work to the researcher. The added benefit of processing both live network feeds and recorded pcap files allows training on pcap files, followed by seamless deployment to a live network.

2.2.6 Representative Pattern Matching (RPM)

RPM is a univariate time series analysis classification technique. It is designed to efficiently determines the representative patterns for each class and removes common patterns between classes. It uses these representative patterns to transform the data by calculating the minimal distance for each representative pattern from every sample time series. These distances are used as the new feature space for classification. SVM's are used for the classification algorithm in this new feature space. RPM is competitive with the top models for time series classification [11]. RPM is the classification tool used by TSAT (Time Series Analysis Tool)[4]. As will be seen, TSAT has been modified to incorporate the algorithms developed in this thesis.

2.3 Literature Survey

This section describes related work in Persistence Landscapes and Botnet detection.

2.3.1 Persistence Landscapes

Persistence landscapes were first introduced in [8] in 2012 as a topological summary. This summary was a transformation of the existing barcode summary that was commonly used beforehand. The barcode summary and persistence landscapes are dual. The advantage of persistence landscapes over the barcode summary, and persistence diagrams are that Bubenik et al. [8] showed that statistics on persistence landscapes are meaningful. They also showed that persistence landscapes were similar under perturbation (stable), two persistence landscapes could easily be compared (computing a lower-bound on the bottleneck and Wasserstein distances) and provided a statistical theory for them. The significance of this being that it made persistence landscapes viable for use as an input to statistical machine learning models.

Bubenik et al. [12] released another paper in 2014 discussing how to compute persistence landscapes optimally. They provide a method to compute persistence landscapes with

arbitrary precision as well as an approximation grid-based method. Additionally, they include a method for computing the linear combination of persistence landscapes. The algorithm for the exact computation of persistence landscapes is optimal in the worst case with a time-bound of $O(n \log n + Kn)$ where n is the number of birth-death pairs, and K is the number of nonzero landscapes one wants in return. Although this algorithm is optimal in the worst case, the author states that it might be possible to increase the computation time in specific cases using a plane sweep algorithm. Their code is available online.

In 2015 Perea et al. [13] introduced a "theoretical framework for the topological study of time series data" [13]. The main focus was on detecting the periodicity of data using persistence diagrams. Perea et al. used a sliding window over EKG, EEG, and MEG data and found that 1-d persistence using the Vietoris-Rips filtration was strongest when the window size is set to the underlying system's natural frequency.

Seversky et al. [6] proposed a standardized way to use TDA techniques on time-series in 2016. They introduced a TDA time-series processing pipeline that has since gained traction in the field. Additionally, they released and described a standardized data set (TS-TOP) to test topological data analysis methods. The data set consisted of both single and multivariate time-series data from a variety of domains. Their work focuses on the classification of time-series data. They noted that data from natural sources, such as walking, had more success with their models. They also noted that there seemed to be an evolution of the topological landscape over time that should be explored further [6].

These papers, among others, led Gidea et al. [5] to use sliding window persistence landscapes to analyze financial data from 2007–2009 in an attempt to find early warning signs of the dotcom crash of 2000 and the Lehman bankruptcy of 2008. The method Gidea et al. [5] used consisted of moving a sliding window over the data set. For each window, Gidea et. al. computed 1-d persistence landscape using the Vietoris-Rips filtration and then computed the L^p norm of the landscape. They then plotted the result as a 1-d time-series and applied standard time-series analysis techniques to detect a change in trading trendings leading up to both crashes. They successfully detected changes in the L^p norms leading

up to both crashes and believe this is a viable early warning sign for detecting financial crashes. The results of [5] showed the viability of topological data analysis for prediction and classification problems. This early success for persistence landscapes led others also to explore time-series datasets from other domains as well.

Postol et al. [4] applied the techniques seen in [5] and [12] to IoT network traffic in an attempt to classify IoT devices. They used a real-world data set of 183 IoT devices consisting of 17 different types spanning many of the top types of devices. This data was put through a Vietoris-Rips filtration, a persistence landscape was calculated on the birth-death pairs and finally put through an L^p norm similar to [5]. Postol et al. [4] also chose to use 1-d persistence for their Vietoris-Rips calculations. They found a window size equivalent to one work-week to be the best for their data set. They believe this is because any significant behavior change would take place on this scale. They used a grid search to find this parameter. For their 1-d signal analysis, they used Symbolic Aggregate approXimation (SAX), a symbolic representation for time-series data, and RPM [4]. They were able to achieve an F1 score upwards of 0.77 and an MCC upwards of 0.66 using this technique on a multivariate version of the data set. They found that their method performs best when given a large corpus of data [4]. They also found that when the corpus was not large enough, the method performs worse than conventional techniques. They believe that this is because their method needs a critical amount of data before they can begin to understand the underlying patterns.

2.3.2 Botnet Detection

Botnet detection is a multivariate time series problem that requires in-depth domain knowledge for effective feature selection and engineering. There are many open problems in this area, and at large, it has been lagging behind the general advances in machine learning. This is seen as botnet fingerprinting is still openly accepted as a must-have for any reasonable security approach. There have recently been improvements in the capabilities of botnet detection problems with advances in deep-learning and dimensionality reduction techniques.

Specifically, auto-encoders, convolutional neural networks (CNN), reinforcement learning, and topological data analysis (TDA) have shown great promise.

Various implementations of neural networks have become popular instead of more traditional statistical anomaly detection and signature-based methods. This change has led to significant improvements in the generalization of models to novel botnet types and evasion techniques. Botnet detection is different from many other types of traditional machine learning problems, as there are adversaries who are continually attempting to fool models. This, in part, has resulted in some people refusing to describe their models and feature engineering techniques fully. This has, in the past, slowed the progress of the field. Probabilistic machine learning models have taken this problem head-on as they require less effort to adapt to changing challenges than traditional statistical models.

A Survey of Botnet and Botnet Detection [14] was presented at the IEEE Conference on Emerging Security Information in 2009. It discusses what botnets are, as they were still emerging as a significant threat. It then describes the main types of botnets and examples for each one. These types are signature-based, anomaly detection, DNS-based detection, and mining-based detection. We see in [15] that little has changed regarding what types of methods are used for detection, with the significant difference being that more machine-learning approaches have become popular.

It is interesting to see from [14] that the vast majority of the methods are not protocol independent. Specifically, only two are protocol-independent: both of which are machine learning models. This promise of more generalizable results is a likely reason for seeing an increase of researchers applying machine learning models to the botnet detection problem.

A recent paper in this area is BotMiner [16]. BotMiner is composed of two clustering models. It works by clustering over both communication patterns and malicious activity patterns. It then determines which devices are similar in both models to identify botnet activity. The paper does not present specific, quantifiable results to evaluate the effectiveness of the model.

A survey on network-based botnet detection methods [15] was published in the Wiley

Online Library's Security and Communication Networks journal in 2014. This survey gives extensive coverage of various findings, state-of-the-art models, and problems facing the field. The work in [15] gives a good overview of the break down of what methods have been used in botnet detection and what papers to refer to get more information on specific implementations. Artificial neural networks (ANN) and support vector machines (SVM) do not appear in any of the listed papers. As pointed out in the survey, it is difficult to compare models because there is no data set that all of the models have used. One useful insight is that there are four types of features in any of the models – bot behavior, botnet behavior, temporal behavior, and protocol behavior. It is rare for a model to take advantage of all three types of features, and one model only takes advantage of one (with the mode taking advantage of two of the four).

In [17] Garcia et al. emphasize that there is a lack of comparison between botnet detection algorithms. As a result, they present novel error metrics to address the problem. The new metrics are based on network administrators' needs that would be interacting with the algorithms day-to-day. These include improvements such as having the error classifying IP addresses instead of NetFlows [17]. This new approach is four new error metrics that are similar to TPR, FPR, TNR, and FNR. In addition to these new error metrics, they introduce two new algorithms for botnet detection: BClus and CAMNEP. BClus is a clustering algorithm that separates the network capture into time windows and then IP address. It embeds domain knowledge through the selection of which NetFlow features to extract. CAMNEP combines various state-of-the-art anomaly detection models to model each user's behavior on the network. It then detects deviations from each user's expected behavior. They also released their data set to test their findings. Luckily, they did release their data set to the general public, but they only released the NetFlow of the data instead of the raw network capture. This makes it near impossible to add certain features in the future.

They compare these two methods to BotHunter [18]. BClus was too aggressive with labeling IPs as malicious, while CAMNEP is too cautious in labeling an IP.

BotHunter [18] uses statistical anomaly detection to identify the sequence of steps botnets and malware usually take in the process of infecting a computer. It comprises two systems SCADE (Statistical sCan Anomaly Detection Engine) and SLADE (Statistical PayLoad Anomaly Detection Engine). SCADE is a snort plugin that attempts to detect when inbound or outbound port scans are happening on a machine. SLADE is an anomaly detection program for finding anomalous payloads. It accomplishes this by determining when the lossy n-gram frequency deviates from the norm. The final part is a correlation engine designed to detect the sequence of an attack. This embeds domain knowledge into the problem. This makes it very brittle to new types of attacks as it is specifically designed only to detect a specific type of attack. It has reasonable success at detecting the types of attacks it was designed to find.

The authors state that the model is "capable of accurately flagging both well-studied and emergent bots" [18]. This was tested on a virtual network and a live honeypot.

Although it was good when discovered, Bot Hunter is not the best we can do. This was shown in [17], and a more general statistical machine learning approach could help improve the accuracy of results.

BoTShark [19] explores the use of convolutional neural networks and stacked autoencoders to solve the problem of botnet detection. They use the ISCX 2014 botnet data set and do minor feature engineering to argus's raw output. They use two stacked autoencoders for greater feature extraction. The output of these stacked autoencoders is put through the softmax function for classification. They use the autoencoders to go from 27 down to 10 features for the softmax function. The CNN that they use has two convolution layers and a fully connected layer.

Unfortunately, they do not report too many statistics, but it looks like the model could learn very well. This is inspiring as the neural network models they used are not very complicated, and extending their results to more complex ones should be able to better capture the data. They reported a true positive rate of 0.91 and a false positive rate of 0.13 for the stacked autoencoders and 0.92 and 0.15, respectively, for the CNN. A significant

benefit of the autoencoder model is that it can do automatic feature extraction. This will allow security researchers to spend less time analyzing a new botnet protocol when one is found and should allow them to classify them right away.

Although a more specialized model may get a better metric in specific workloads, it is the general applicability of these models, which is the most interesting. This leads to many further questions that the field should look into, including trying other types of neural network models and if a larger model that takes in different features can generalize better.

The approach described in [20] makes a compelling case for the use of neural networks for botnet detection. It uses multiple data sets and highly tuned features, along with a reinforcement learning approach. The model is first shown a training set to train a multilayer neural network, after which it is released into the wild, where it continues to improve its model through reinforcement learning.

2.3.3 Related Work Evaluation

It is challenging to compare different approaches overall method because almost no two algorithms have used the same data set. Another shortcoming of the field is that there is little to no comparison between methods, thus making it hard to reach general conclusions. There has been an effort to create canonical data sets, but researchers have little motivation to use them over their private data sets. This might be changing as [20] did provide a comprehensive analysis of their algorithm.

Chapter 3: Persistence Landscapes and Noisy Data

The work in [8] and [12] provides the computational foundations for the use of persistence landscapes. However, it is desirable to more deeply explore this approach's robustness in the presence of noise and in the difficulty of setting hyperparameters such as window size for time series analysis. This will help guide our use of PL-TDA in analyzing botnets and other types of noisy network data.

3.1 PL-TDA and Noisy Data

This experiment is motivated by the experiments shown in [12]. Those experiments sampled points take from $\mathcal{R}^D$ mapped to a lower-dimensional sphere S^d. The question is, how well does PL-TDA perform in determining the value of d?

As can be seen in Figure 3.1 PL-TDA captures the number of circles because there is an equal number of mountains of considerable height. The low mountains are caused by the noise in the data from the cycles of the circles forming.

When we surround one circle in another, PL-TDA creates two mountains of different

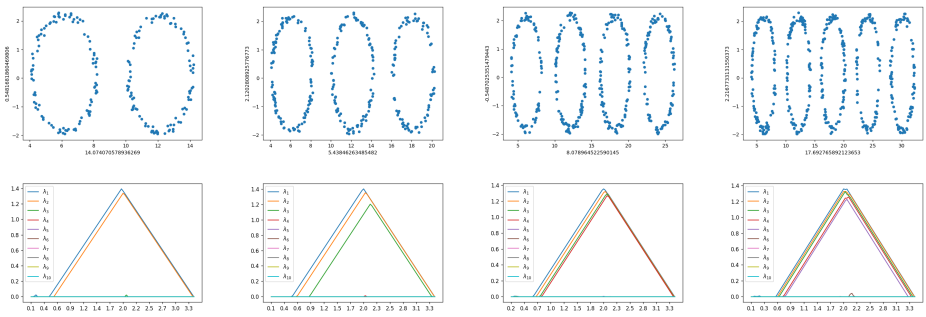

Figure 3.1: Various point clouds and their corresponding persistance landscapes

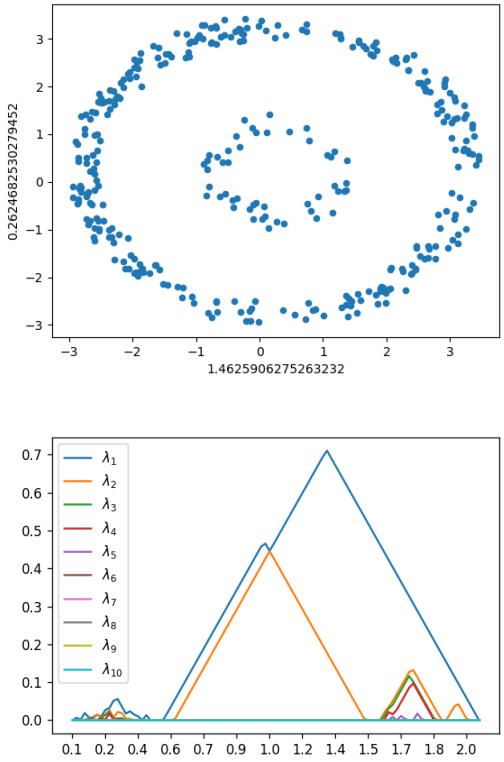

Figure 3.2: Two circles and their corresponding persistance landscapes

persistence. Additionally, when the two features merge, we see many small mountains in Figure 3.2.

I added two types of noise to this dataset: creating a feature that is less precise and dedicating fewer points to the feature and instead adding random data. Figure 3.3 and Figure 3.4 show that PL-TDA is still able to capture the important features of the point cloud even with high levels of noise. 50% of data can be dedicated to uniform noise over the feature's space, and the feature can still be extracted with relative ease. When dimensions of only random points are added to the data, the model can only extract features when two or fewer of these new dimensions are added. After that, it is not possible to extract the features reliably.

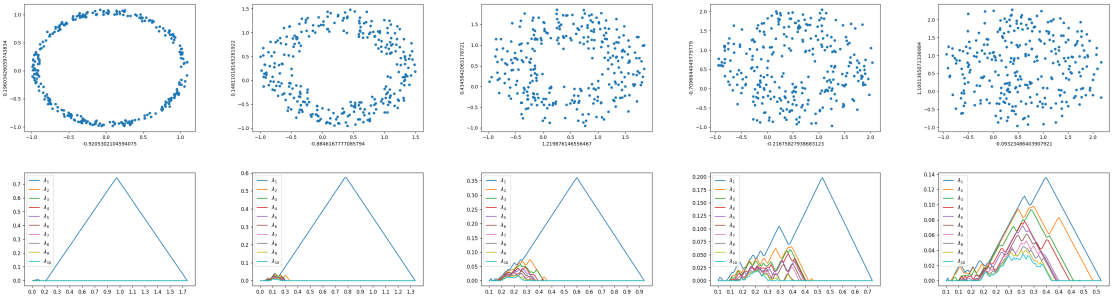

Figure 3.3: Uniform noise on a single circle of radius 1. Level of uniform noise added from left to right: 0.1, 0.5, 0.9, 1.1, 1.3

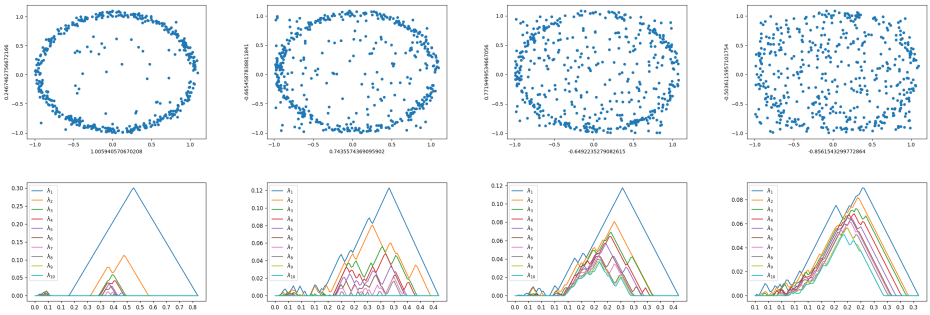

Figure 3.4: Various percentage of points dedicated to uniform noise with a circle of radius 1. Percentage dedicated to noise from left to right: 10%, 30%, 50%, 70%

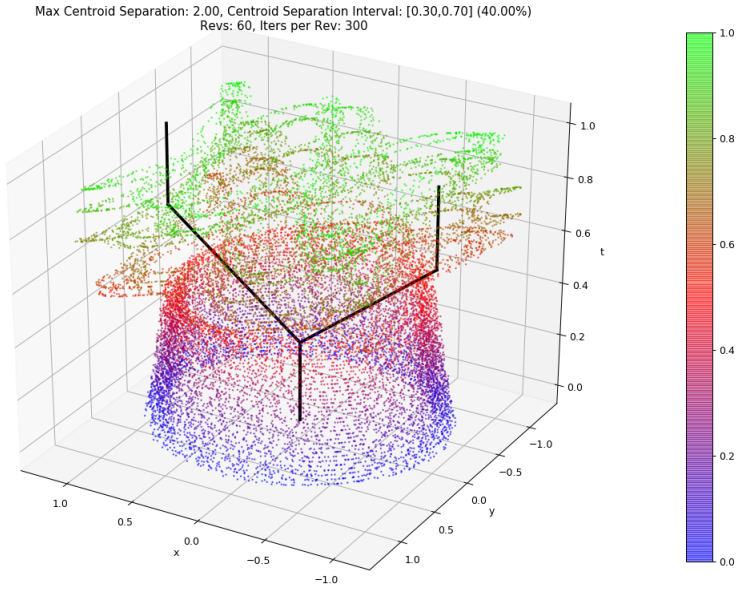

Figure 3.5: Particle over time.

3.2 PL-TDA Hyperparameter Intuition

3.2.1 Window Size

To quantify the method's success at various window sizes, the synthetic data set from Postol et al. [4] was used. This synthetic data set is a multivariate time series obtained by windowing and tunable time steps of values of a simulated particle's movement around a single gravitational object that splits into two. This data is plotted over time in Figure 3.5.

PL-TDA should detect a change in topology, represented by a change in L^p norm when the orbit of the particle changes from a single circular orbit to an orbit more representative of a figure-eight once the object it is orbiting around splits into two partway through the simulation. The experiment is set up to have 300 samples per orbit, and that interval timing was not changed as the orbit increased.

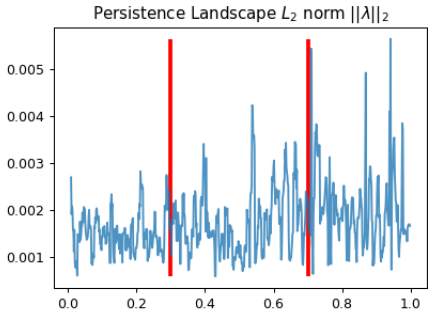

Figure 3.6: Window size 150 with 10% uniform noise.

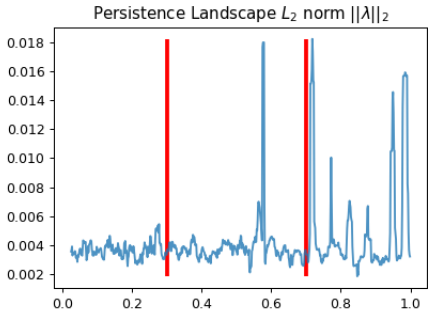

Figure 3.7: Window size 450 with 10% uniform noise.

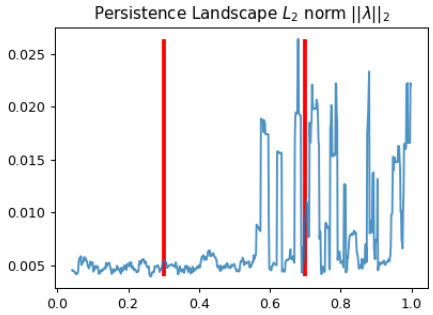

Figure 3.8: Window size 750 with 10% uniform noise.

It can be seen from Figures 3.6, 3.7, and 3.8 the most significant difference before and after the orbit change happens after a window size of 750. The area between the red lines is where the centroid is diverging. This is approximately the period of the particle during the latter part of the experiment. This makes intuitive sense if a time series has a constant period using splitting the dataset into time windows of equal length is a technique in feature engineering. Unfortunately, not all time-series are cyclic, and when this is the case setting, the window size is not as easy.

3.2.2 Resilience to Noise

Two types of noise were tested: observation noise (noise within a previously existing dimension) and dimensions of pure noise (adding dimensions of nothing but random values to a previously clean data set). Both types of noise are tested with uniform random noise and gaussian noise. During these tests, we found that TDA seemed to be reasonably resilient to observation noise. There is still a clear difference in the L^p norms before and after the orbit change. This is true even when every data point is shifted by uniform noise up to 50 percent of the dataset's original range as seen in Figures 3.9, 3.10, 3.11 and 3.12.

When dimensions were added to the data set containing only random noise, either uniform or Gaussian, TDA had significant trouble finding a difference in the data as seen

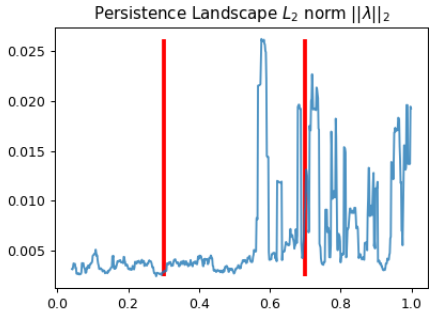

Figure 3.9: Window size 750 with 10% uniform noise.

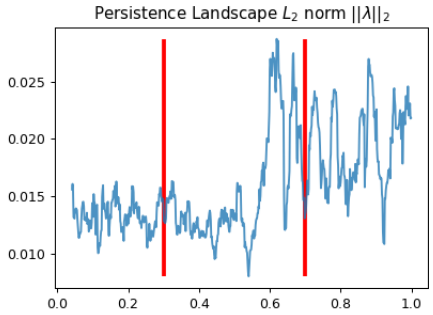

Figure 3.10: Window size 750 with 30% uniform noise.

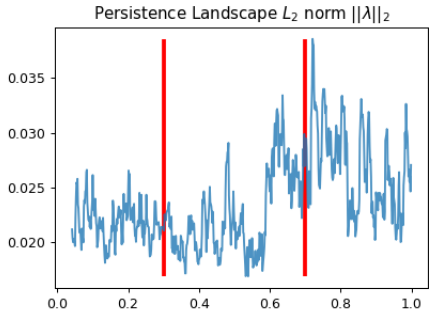

Figure 3.11: Window size 750 with 50% uniform noise.

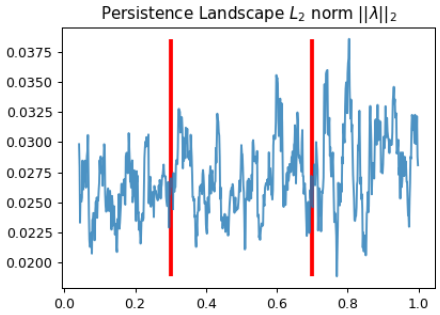

Figure 3.12: Window size 750 with 60% uniform noise.

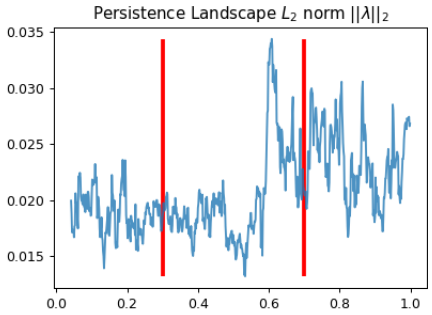

Figure 3.13: Window size 750 with 0 added dimension at 40%.

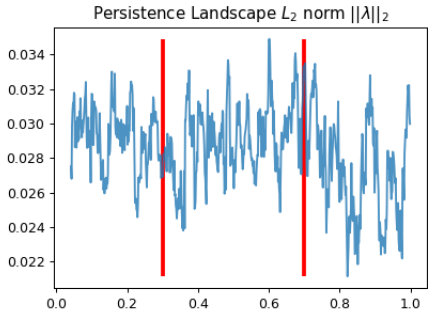

Figure 3.14: Window size 750 with 1 added dimension at 40%.

in Figures 3.13, 3.14 3.15 and 3.16.

Although it was able to pull out differences when the random dimensions were small compared to the other features' overall range, this is unlikely to be the case for real-world data analysis as data is often normalized before it is put through most models. When this is done to the data, TDA has trouble pulling out any meaningful change. This is important to be aware of when using TDA as it requires cleaned data to get any result in some cases. It would be interesting to see if any n-dimensionality reduction or denoising techniques might work well as a preprocessing. This would remove these useless dimensions and allow TDA

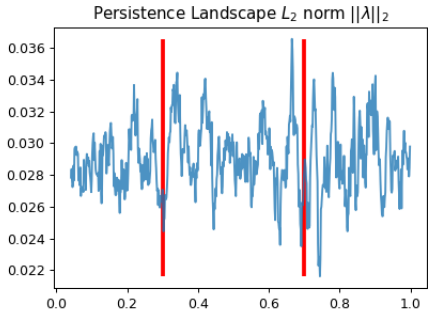

Figure 3.15: Window size 750 with 2 added dimension at 40%.

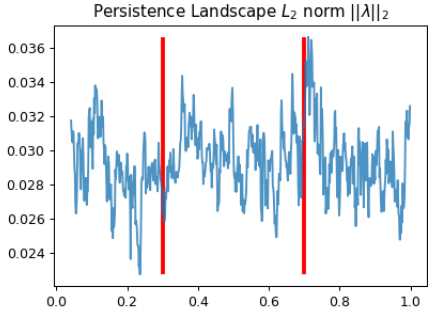

Figure 3.16: Window size 750 with 3 added dimension at 40%.

to be more successful in more application domains.

3.2.3 Maximum Rips Radius

The maximum rips radius is used to determine the maximum threshold when computing the "connectedness" between two points. This provides filtering on the birth-death pairs by removing all birth-death pairs with a death after the defined threshold: determining the data set's large topological feature captured. This is because PL-TDA will drop any birth-death pair with a death of infinity. This is because if the pair is kept in the persistent landscape the L^p norm of all of the landscapes that contain that birth-death pair will be infinity. Therefore we should set the maximum radius parameter to the minimum threshold that contains the death of all of the features of interest.

It should be noted that increasing the max rips radius can not affect already discovered birth-death pairs. Being aware of the magnitude of the largest birth-death pair is useful when determining what value to set the max rips radius to and can help determine after the results have been processed if a feature of a given size was discovered or not. A red flag is if the persistent landscapes' scale varies greatly where no topological change should be happening. Of course, this could be a topological property of the data set that was previously unknown. Either way, it is something that should be investigated. It can also be useful in determining if the max rips radius is too large. If there is extraneous information at the right side of the persistent landscape where no feature should be, it is simple to determine how to set the max rips radius to cut out these features. It should also be possible to do an extra filtering step in the opposite direction by beginning the rips filtration at the base of the first important persistent landscape. This would naturally make the filtration less resilient to noise but could be a simple way to speed up computations.

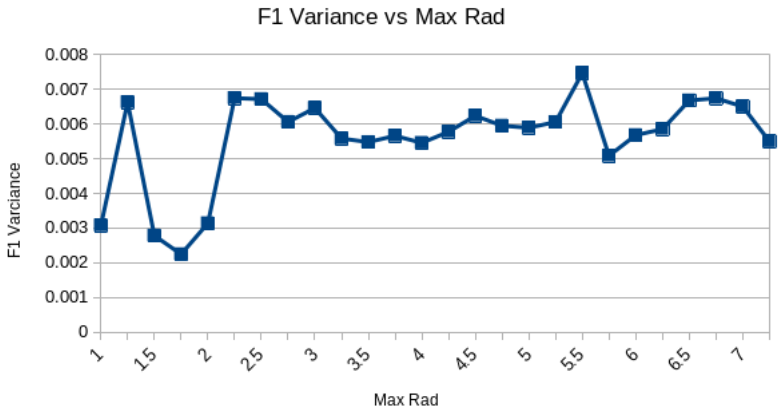

Figure 3.17: Average F1 vs max rips radius.

3.3 Hyperparameter Graphs

Looking at the best hyperparameters for the ISCX 2014 botnet dataset gives insight into setting these parameters. The max rips radius and window size hyperparameters in Figures 3.17, 3.18, 3.19 and 3.20 shows both hyperparameters' mean and variance become fairly stable at larger values. The conclusion is to set the values in the medium to high range of possible values for initial analysis. It is possible to tune each hyperparameter to squeeze out the best performance. In the tests run, doing a grid search over a couple of hundred possible settings led to minor improvements in weighted F1 and MCC.

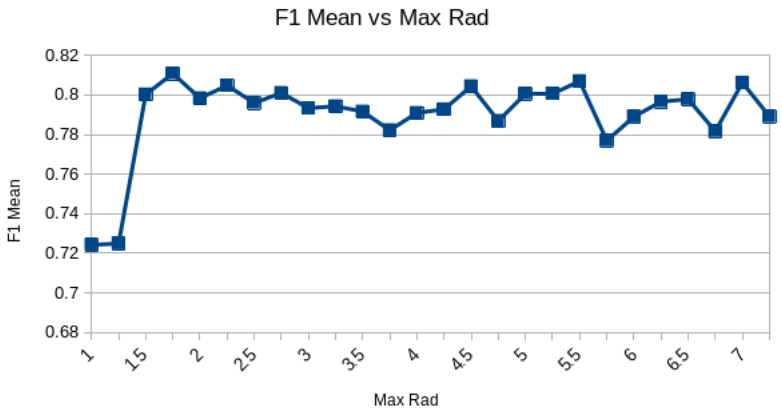

Figure 3.18: Average F1 vs max rips radius.

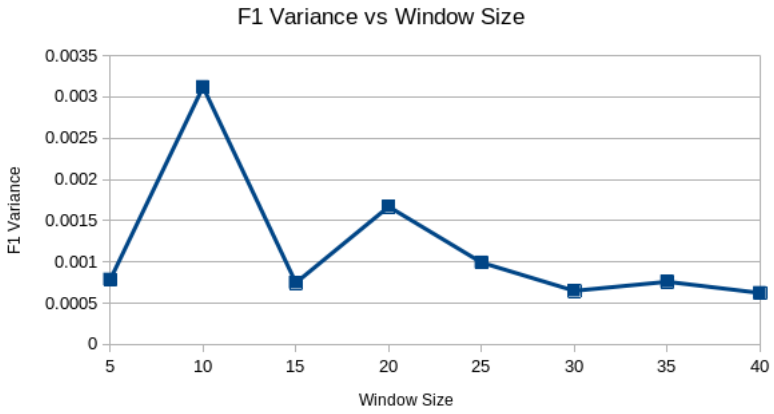

Figure 3.19: F1 variance vs window size.

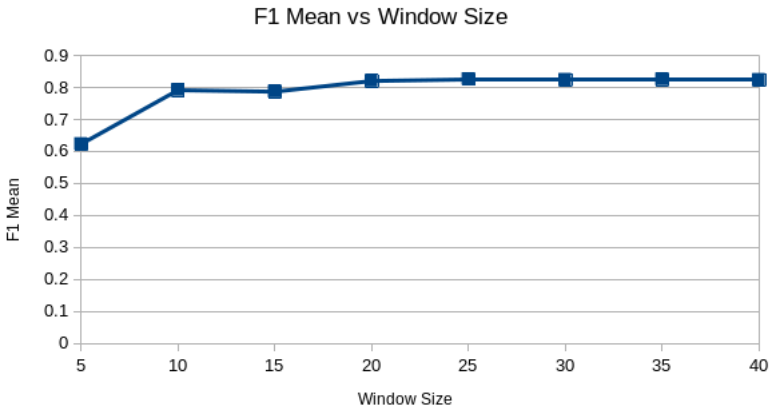

Figure 3.20: Average F1 vs window size.

Chapter 4: Efficient Algorithms for PL-TDA

Two novel algorithms are presented in this section that, when combined, result in a significant speed-up in the time needed to compute the top K persistence landscapes given a set of birth-death pairs. There is first an output-sensitive algorithm for computing the top K persistence landscapes. Then there is a preprocessing step of the set of birth-death pairs that will, in optimal linear time, removes all birth-death pairs that can never appear in the top K landscapes. This method can be used with any method for computing persistence landscapes.

4.1 Plane-Sweep Landscape Generation

This section provides an output-sensitive algorithm for computing persistence landscapes by taking advantage of the geometric structure of the problem. The inspiration for this finding stems from Bubenik et al. [12] where it was stated that it might be possible.

4.1.1 Overview

Given a collection of every birth-death pair and every intersection between birth-death pairs (including which two pairs are involved in the intersection) it is trivial to determine the top K landscapes. All one has to do is perform a plane-sweep of the data and keep track of the ordering of landscapes based on y coordinate. We only need to track when the mountains entered and left the top K persistence landscapes. This technique's disadvantage is that every intersection must be calculated, which upper bounds our algorithm by $O(n^2)$. Given all of this data to start with, then it would be the optimal algorithm.

Fortunately, through careful modifications to the above algorithm, it is possible only to perform a known amount of work for each birth-death pair that appears in the top K

landscapes. This creates an output-sensitive algorithm. This is the algorithm is built up in this section. The first stage is the definition of a preprocessing step that eliminates all birth-death pairs that do not appear in the top K landscapes. The definition of this preprocessing step is in Section 4.2. We use the findings of Section 4.2 to run another plane-sweep algorithm before we generate the persistence landscapes to filter out birth-death pairs the no not appear in the top K landscapes.

Secondly, we define an algorithm that checks for intersections only with birth-death pairs currently apart of the top k-landscapes. This means that all of those intersections must be apart of the final diagram. The key to this algorithm comes when we add a new pair to the top K landscapes. If there are less than K birth-death pairs alive, then the new pair is automatically added to the top K landscapes. To add new birth-death pairs into the top K landscapes when there are already K alive pairs, we check for an intersection with the new pair (once we see its birth) and the bottom mountain in the top K landscapes if that bottom mountain has a negative slope. If this is the case, the problem is such that the two mountains must intersect. Additionally, we only check for intersections with paris not apart of the top K when a new mountain becomes the bottom mountain or changes slope. If a mountain has just become the bottom mountain, it must have a negative slope and be the next mountain to die in the top K landscapes. That check is with the next alive landscape, which must be taking its place in the top K landscapes (this is the oldest landscape that is not in the top K landscapes). The two mountains must intersect, and the resulting intersection point must be in the top K landscapes because of 4.2.

This algorithm is inspired by the line segment intersection algorithm from the Dutch computational geometry book [21]. When observing the persistence landscape problem from an abstract level, the two problems are very similar.

4.1.2 Explanation

Alogorithm 1 contains the critical points of the algorithm but does leave out some book-keeping. The rest of this section aims to fill in those gaps.

Algorithm 1 Compute top K persistence landscapes

Precondition: *pairs* is a list of all birth-death pairs and *maxLambda* is the number of landscapes to calculate

```
1: function ComputeLandscapes(pairs, maxLambda)
2:     events ← generateInitialEventPoints(pairs)
3:     for e ∈ events do
4:         if isStartPoint(e) then pos ← addToStatus(e)
5:             if intersection ←intersectsWithNeighbor(e) then
6:                 events.add(intersection)
7:             if pos < maxLambda then
8:                 λ[pos].append(e)                    ▷ λ is the landscapes to be returned
9:         else if isPeakPoint(e) then
10:            if intersection ←intersectsWithNeighbor(e) then
11:                events.add(intersection)
12:        else if isFlipPoint(e) then
13:            e1, e2, pos1, pos2 ← flipPoints(e)
14:            if intersection ← intersectsWithNeighbor(e1) then
15:                events.add(intersection)
16:            if intersection ← intersectsWithNeighbor(e2) then
17:                events.add(intersection)
18:            if pos1 < maxLambda then
19:                λ[pos1].append(e1)
20:            if pos2 < maxLambda then
21:                λ[pos2].append(e2)
22:        else if isEndPoint(e) then
23:            if pos < maxLambda then
24:                λ[pos].append(e)
25:            removeFromStatus(e)
26:    return λ[ ]
```

As noted in Subsection 4.1.1, the algorithm is inspired by the plane-sweep line segment intersection in [21]. Our algorithm differs from the original algorithm in the following keys ways. We still have an ordering, but now our geometric objects consist of two line segments, defined by three points: birth, peak, and death. One line segment goes from birth to peak, and the other line segment goes from peak to death. Our status structure is the ordering of birth-death pairs where having a higher y-coordinate at the sweep line's current position denotes a higher position in the status structure. We should also note that our sweep line always starts on the far left of our collection of objects, perpendicular to the x-axis. Our event points are the birth, peak, and death points that define each mountain and any intersection points between mountains discovered through the algorithm. We check for intersection points the same way as in the line segment intersection algorithm: only checking for intersections between two mountains if they are neighbors in the status structure.

The algorithm does the following given a set of birth-death pairs. First, it generates the initial event points. The initial event points correspond to the following for each birth-death pair: the birth, the peak, and the death of a given topological feature.

We need to take different actions at each event point. At a birth point, we must add the corresponding mountain to the status structure. We luck out here because, from the definition of birth-death pairs, we can guarantee that when a "mountain" first appears, it must be lower than any other mountain in the status structure. As a result, we do not have to search the status structure to determine where it should be inserted to maintain the ordering. This fact allows us to get away with a simple status structure with constant insert and access times, such as a linked list or array. We must also check to see if this new mountain intersects with its neighbors, but because it is at the bottom of the status structure, it can only have one neighbor: the pair above it. We only check for this intersection if the neighbor above is after its peak. If it is still rising, it could switch positions with mountains above it before it intersects with our new pair. If our new pair intersects with its neighbor, add the intersection point to the event structure. A Fibonacci heap named eventTree is the event structure used here. Finally, if our new pair has a position in the status structure in

the top K, that means it is a part of one of the top K landscapes. As such, we keep track of it to report later. We use K linked lists in our implementation to return to the user at the end of the algorithm. This point is added to the respective linked list.

For peak points, we have a similar story. First, we check in with our bookkeeping to see if the peak point belongs to a mountain that is part of the top K landscapes. If it is, we track it in the way previously mentioned: adding it to the end of the respective linked list. Next, we check for intersections as our mountain has changed slopes and could now intersect with any mountains below it. If we find an intersection, add it to the eventTree.

For an intersection/flip point, we first need to exchange the points in question within the status structure. After this step, each of these points has a new neighbor. We must check for intersections with the new neighboring points. If we find an intersection, we carry out the usual actions to add it to our eventTree. Finally, if the new position of either of the two points respective mountains involved in the flip ends up in the top K, we track this new point in the linked list. If this flip caused one of the mountains to leave the top K landscapes, we remove the point altogether as it cannot reappear in the top K landscapes. This ensures that we do not compute unneeded intersections.

Lastly, we can guarantee that the corresponding mountain must be in the bottom position in the status structure for a death point. If this were not the case, then there would be at least one other point, the one below it, that would have to end earlier or intersect with our mountain before it could die itself. The reason is that all mountains must end at the same y coordinate, they all have the same slope after their peak, and each mountain consists of only two line segments. As a result, we can delete the bottom mountain from the status structure and be confident that it is the one the end-point is referencing. Before we do this, we check to see if it is currently part of the top K landscapes. If it is, we save the point to its respective linked list.

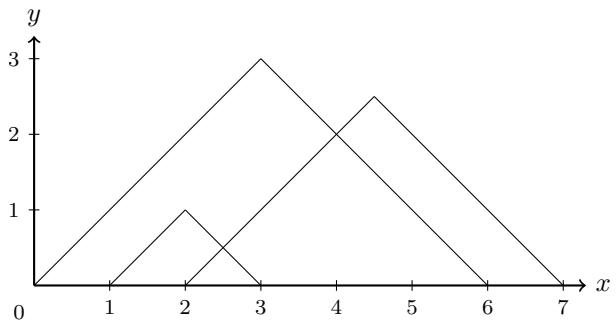

Figure 4.1: Corresponding persistence landscape.

4.1.3 Algorithm Walk Through

If the birth-death pairs input into the algorithm is (4.1) (corresponding to a persistence landscape in Figure 4.1) then the algorithm will run through the following states in order. The birth-death pairs are referred to as x, y, z where $x = (0, 6), y = (1, 3), z = (2, 7)$. Subscripts refer to whether the point is a birth, peak, or death point. The values for the starting evenHeap are in (4.2).

$$\{(0,6), \quad (1,3), \quad (2,7)\} \tag{4.1a}$$

$$x \quad y \quad z \tag{4.1b}$$

$$x_b = (0,0) \quad x_p = (3,3) \quad x_d = (6,0) \tag{4.2a}$$

$$y_b = (1,3) \quad y_p = (2,1) \quad y_d = (3,0) \tag{4.2b}$$

$$z_b = (2,0) \quad z_p = (4.5,2.5) \quad z_d = (7,0) \tag{4.2c}$$

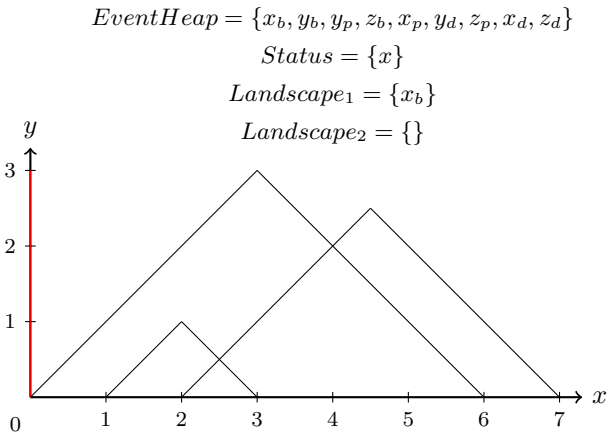

Figure 4.2: After processing x birth

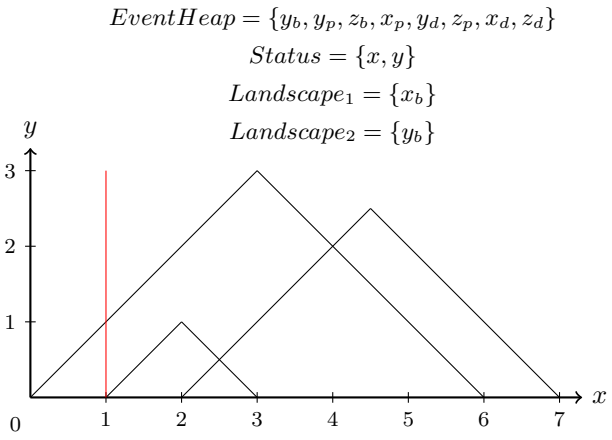

Figure 4.3: After processing y birth

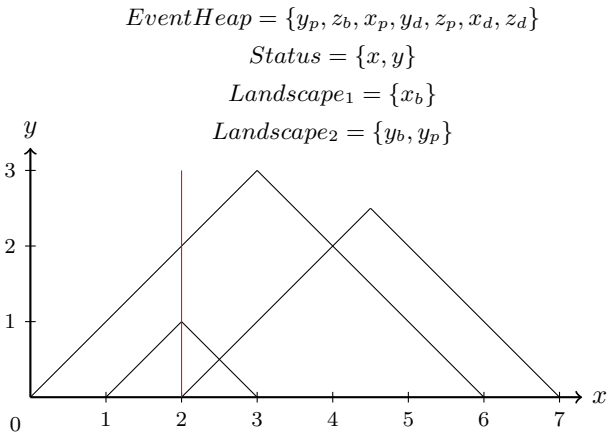

Figure 4.4: After processing y peak

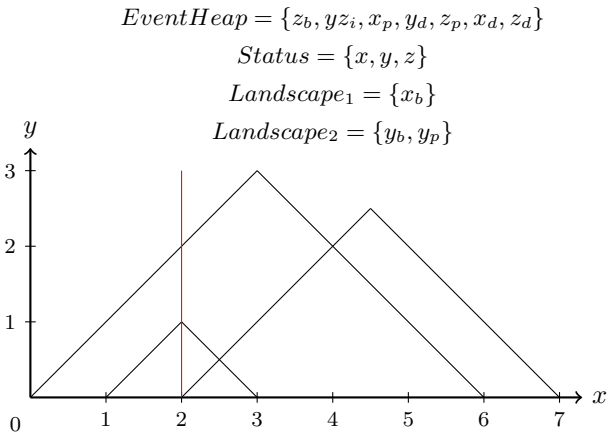

Figure 4.5: After processing z birth

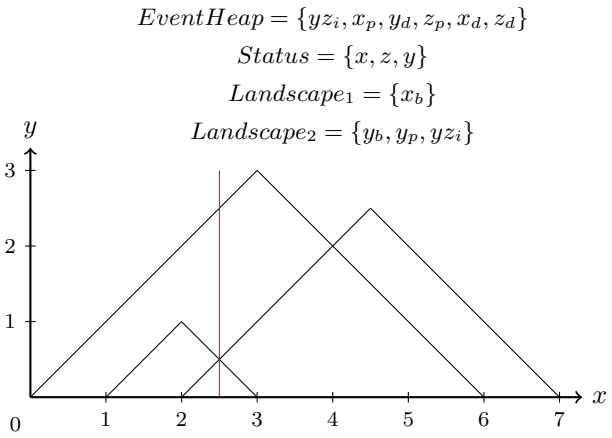

Figure 4.6: After processing xy intersection

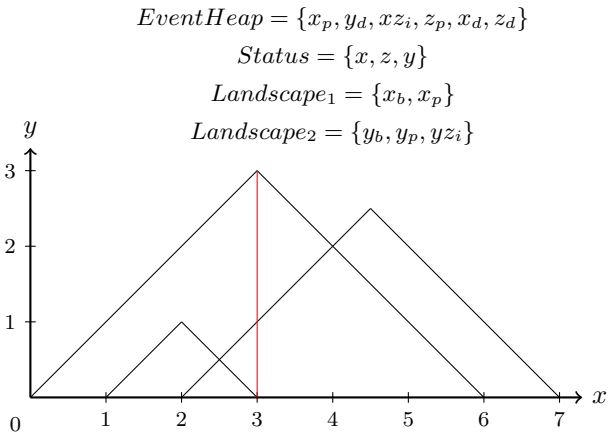

Figure 4.7: After processing x peak

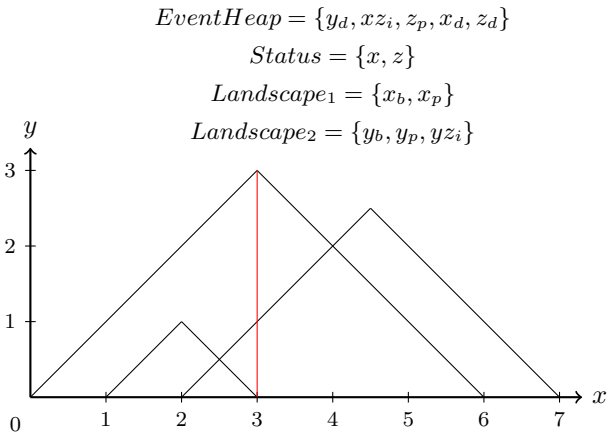

Figure 4.8: After processing y death

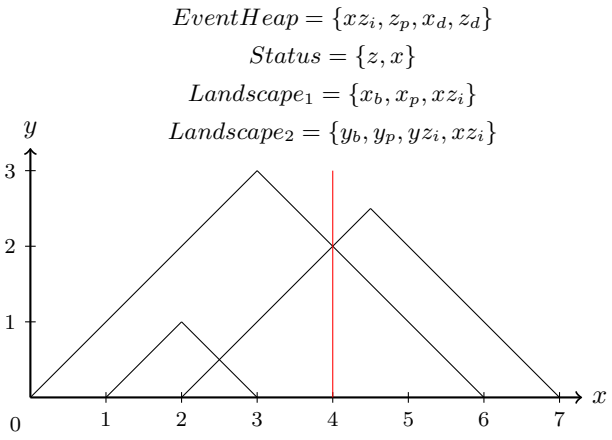

Figure 4.9: After processing xz intersection

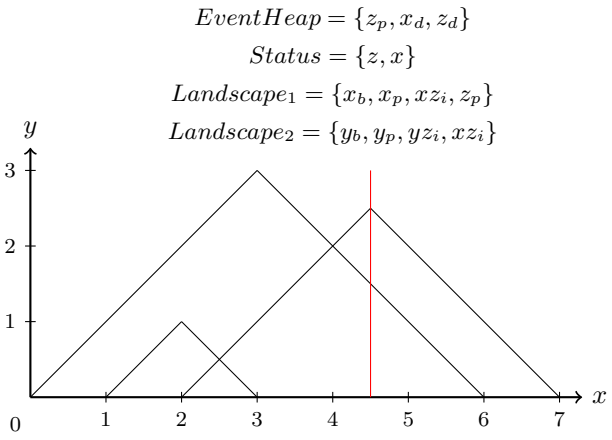

Figure 4.10: After processing z peak

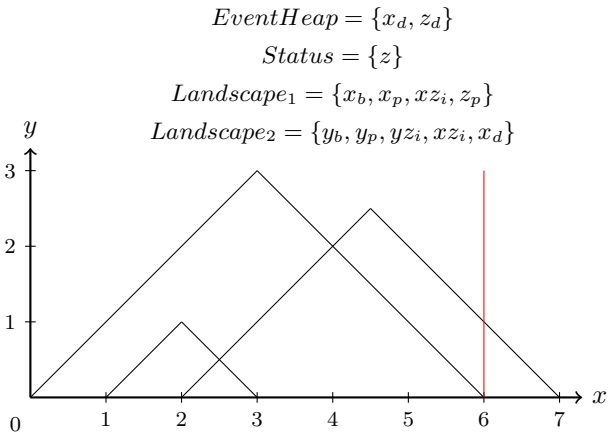

Figure 4.11: After processing x death

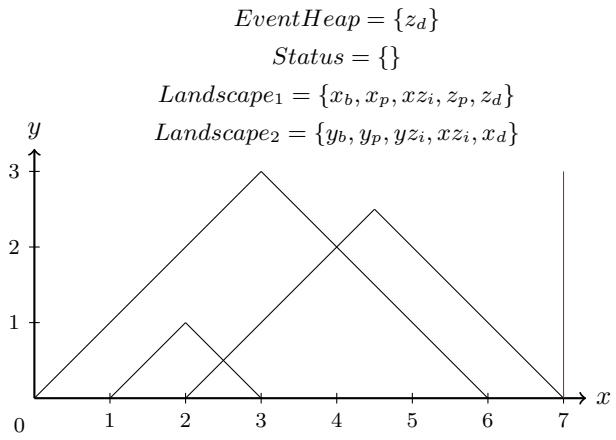

Figure 4.12: After processing z death

4.1.4 Runtime

The algorithm's runtime is relatively complex to analyze, especially when combined with the results from 4.2. First, for every event point, we only do constant work. This is because the five possible actions for an event are as follows: add to status (birth), check for intersection (birth, peak, intersection), add to a linked list (all), swap two points (intersection), and remove from the status structure (death). For all of these actions, we maintain pointers between all the structures and, with proper bookkeeping, only incur an addition of $\Theta(1)$ runtime per event if we use a good heap. We use a Fibonacci heap for the analysis here.

The complicated analysis comes when observing the eventTree structure. To obtain an upper bound, we count the number of events added to the structure and the cost for adding and removing each event. The total number of events is a combination of the initial event points with the total number of intersections. The initial events are $O(n)$, and to add each of them to the heap is $\Theta(1)$. The number of intersections is upper bounded by $O(n^2)$, but this would only be if all birth-death pair intersected, which is unlikely. This fact is taken advantage of by our algorithm being output-sensitive. Our runtime can be defined in terms of the total intersections in the landscape as we do not incur any extra work by checking for pairs that don't exist as defined above (it is a $\Theta(1)$ check when we are already doing $O(1)$

work regardless of the result). We use I to denote the total number of intersection points. Thus our total number of events is $O(n + I)$.

To determine the cost of each heap operation, we upper bound the heap's size. This is the combination of having every initial event in a heap and having the maximum number of intersections in the heap. To upper bound the number of intersections in the heap, we use the fact that every active birth-death pair only checks for intersections if it is in the top K landscapes. Additionally, each pair in the top K landscapes only checks for intersections with the current direction in which it is headed (up or down). Finally, we only check for intersections with our neighbors, there must be at least 2 points involved in each intersection, and we do not double count intersections. Taking all of this into account, we are upper bounded by $K/2$ ($O(K)$) intersections in the heap at any point in time. Observing that $K \leq n$ we can upper bound our heap size by $\Theta(n)$. Thus our total runtime for the entire algorithm is $\Theta((n + I)\log(n))$

Like most output-sensitive algorithms, this algorithm perform worses than the approach from [12] in the worst case, but there are cases where it performs better. Section 5.4 shows that there is a speed-up in practice vs. the optimal approach from [12].

4.1.5 Integration

The way persistent landscapes are used in our implementation requires taking the integral of the landscapes. Because we have a discrete function, this is a trivial task, and no external library is needed. All one must do is find the area under each line segment to the x-axis for every persistence landscape. This is accomplished easily in $O(n)$ where n is the number of points in all the persistence landscapes.

4.2 Barcode Filtering

The main variables that affect the amount of time it takes to generate persistent landscapes are the number of birth-death pairs and intersections between birth-death pairs. During

real-world applications of the persistent landscape model, only the top K landscapes are kept in the final representation. However, there has been no time-efficient method for determining if a given birth-death pair is in the top K landscapes. This section presents a new property of persistent barcodes, which leads to an optimal $O(N)$ algorithm for determining this desirable property.

4.2.1 Overview

Given a set of birth-death pairs, create a barcode diagram where all barcodes are parallel lines, and barcodes with an earlier birth begin at a higher y-coordinate than those with later births.

Perform a line sweep scan of the barcodes, maintaining an ordering on the lines where higher y-coordinates are higher in the status structure. Keep track of all barcodes that appear in the top K. This process can be thought of as an observer looking straight down on the barcodes that can see through $K-1$ barcodes without loss of generality. If the observer can see the barcode, then it appears in the top K landscapes. An algorithm to accomplish this in $O(N)$ time if the lines are sorted, or $O(N\log(N))$ if sorting is required, follows.

The following proof will show that a barcode appears in the top K barcodes seen by the observer during the algorithm if and only if it appears in the top K persistent landscapes. The proof outline is as follows: First, it is shown that if the observer can see a barcode, then all previous barcodes either are subsets of the given barcode or one of the following two cases. If the birth of another barcode is less than our given barcode's birth, it is proven that they must either cross before our given barcode's corresponding persistent landscape's peak or die before our given barcode birth. The other case is for barcodes born after our given barcodes birth and is not a subset of our given barcode. For these, it is proven that they must cross after our given barcodes peak or have their ranges not intersect. These facts are enough to prove that the ordering of the lines from the line sweep algorithm stated above corresponds to the max position of the birth-death pairs in the persistence landscape.

4.2.2 Proof

Theorem 4.2.1. *Given a line, l, perpendicular to a set of barcodes arranged as a barcode diagram where barcodes with lower births have higher y coordinates, the line segment with the highest y-coordinate that intersects l will be apart of the highest persistence landscape.*

Proof. If there is only one birth-death pair intersecting the line l, then it is evident that there exists an x-coordinate when our birth-death pair, p, is apart of the highest persistence landscape (the current x-coordinate where we are at, which contains no other birth-death pairs).

If multiple birth-death pairs are intersecting l, there exists a point where the birth-death pair that has the highest y-coordinate, p, is higher than all of the previous birth-death pairs (who are not currently intersecting l) while being above all birth-death pairs below it that are currently intersecting l.

We prove that all previous lines must have crossed p before p peaked, and all other birth-death pairs either cross p after p's peak or do not cross p at all.

1. Previous barcodes

   As a result of 4.2.4, all birth-death pairs must peak before they can cross any birth-death pairs below them. Additionally, for a pair of birth-death pairs cross, one must be traveling to its peak while the other is traveling to its death. It then follows that all previous birth-death pairs either intersect p before its peak or they do not intersect at all. If we can observe a given birth-death pair from our observer, there is no birth-death pair which is a superset of p such as in Lemma 4.2.2. Therefore, the only way for a previous birth-death pair to not intersect our given birth-death pair would be for the range of p to not overlap with the other birth-death pair's range. So if we can see p from the observer, any birth-death pair whose birth is less than p^b must either not intersect or intersect p before p's peak.

2. Other intersecting Lines

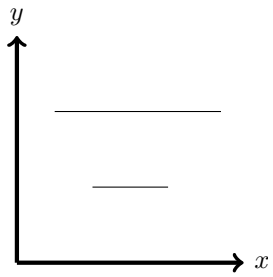

Figure 4.13: Barcode superset example (Lemma 4.2.2)

Refer to 4.2.4

□

Lemma 4.2.2. *If a barcode p_1 is a superset of another barcode p_2, then their corresponding persistence mountains do not cross.*

Proof. All persistence mountains, birth-death pairs in the persistence landscape, have the same slope to their peak and the same slope after their respective peak: which is always halfway between the birth and death points. As a result, if one barcode has a birth p_1^b and death p_1^d and another barcode has a birth p_2^b and death p_2^d such that $p_1^b > p_2^b$ and $p_1^d < p_2^d$, the persistence mountain corresponding to the p_1 barcode will never cross p_2 (as seen in Figure 4.13). Additionally, because all persistence mountains begin at $y = 0$, the barcode p_1 exists inside of p_2 when plotted and, therefore, does not affect if p_2 appears in the top K landscapes. □

Lemma 4.2.3. *Given two birth-death pairs, p_1 and p_2, if $p_1^b < p_2^b$ and $p_1^d < p_2^d$ then p_1 and p_2 must intersect (case in Figure 4.14)*

Proof. The initial ordering of the two lines based off of y-coordinate right after p_2 is born is (p_1, p_2) and the ordering right before p_1 dies is (p_2, p_1). If this were not the case, then p_2 would end before p^1 because they share the same slope right before they die, once their slope changes to -1 it cannot change, and they die at the same y-coordinate in the persistence landscape. As a result p_1 and p_2 must intersect. □

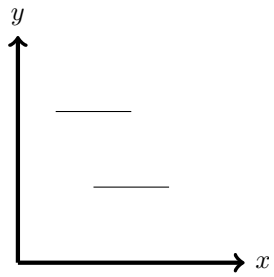

Figure 4.14: Barcode intersection example

Lemma 4.2.4. *Given two birth-death pairs, p_1 and p_2, if $p_1^b < p_2^b$ and $p_1^d \leq p_2^d$, then p_1 must reach its peak before p_1 and p_2 cross.*

Proof. All mountains in the persistence landscape have the same slope to their respective peaks and after the peak to their death. As a result, two birth deaths cannot cross while both of them are either on their way to their respective peaks or deaths; one must be on its way to a peak while the other is on its way to its death when they cross.

If p_2 peaked before p_1, then they would not cross. The reason is that the slope after the peak is the same for both. Because p_2 is already inside of p_1 in the persistence landscape, if p_2 peaked before p_1, it will end before p_1, and we will have Lemma 4.2.2. Therefore p_2 cannot peak before p_1 because $p_1^d \leq p_2^d$.

If p_2 and p_1 shared a death point, then that means the line segments they define after their peaks are colinear because they have an equal slope, by definition, and share a point, their death. In order for them to share this line, p_1 must peak before p_2 because p_1 has an earlier birth and, therefore, would peak higher than p_2. It must, as a result, begin its descent to its birth before p_2 begins its descent in order for them to meet up on the same line. Therefore p_1 peaks before p_2 if they share a death point, and they cross after p_1's peak.

If $p_1^d < p_2^d$, then Lemma 4.2.3 shows they intersect in the persistence landscape. This cross happens when one pair is traveling to its peak while the other is traveling to its death (as shown in Lemma 4.2.3). The pair that is traveling to its peak at the point of intersection

ends after the one that is traveling to its death. The reason is that after the intersection, the pair traveling to its peak will be higher than the other pair. Additionally, once a pair has begun its descent, it cannot change slopes. As a result, the pair that is on its descent at the intersection point must die first. This proves that because p_2 ends after p_1, p_1 peaked before the intersection, or else p_1 would end after p_2.

□

Theorem 4.2.5. *If a birth-death pair is the n-th closest to the observer than it is apart of the n-th persistence landscape*

Proof. Applying the logic from 4.2.6 recursively leads immediately to this conclusion. □

Lemma 4.2.6. *If a birth-death pair is the second closest to the observer than it is apart of the second persistence landscape*

Proof. If a birth-death pair is second closest to the observer, that means that there is only one birth-death that is closer than our given birth-death pair to the observer. Imagine removing this closest birth-death pair. The result of this operation would be that the second closest birth-death pair is now the closest birth-death pair. Because adding in a birth-death pair can only move other birth-death pairs down in rank by a maximum of one position in the ordering. This shows that our given birth-death pair must be apart of the second persistence landscape. □

4.2.3 Importance

The result of this new property of barcodes is that we can now determine in $O(N)$ if sorted, or $O(N \log(N))$ if unsorted if a given birth-death pair appears in the top K landscapes. This limits the number of intersections I from Subsection 4.1 to only be the intersections that appear in the final output.

Chapter 5: Botnet Detection with PL-TDA

This chapter describes botnet detection with PL-TDA. We first describe our implementation strategy, and then discuss the preprocessing pipeline. We next detail the botnet data sets we examined, and finally describe the results of our experiments.

5.1 Implementation

Preprocessing and feature engineering are critical steps for all time series efforts. For this thesis Python 3.8 with NumPy and pandas were used. These Python packages are standard for this task in the field. Ripser was the topological data analysis package used as it was shown in [22] to be the best for the task and has a Python implementation using [23]. TSAT (Timeseries Analysis Tool) ran representative pattern matching (RPM) for the classification of the resulting data from the TDA stage.

Using the datasets described below, we treat each pcap packet capture as a set of points. Given these ordered sequence of points, the pipeline does the following to transform a multi-attribute time series into a univariate time series. First, it separates the time series into ordered sets of equal size determined by a user-defined window size w. It creates the first of these by looking at the first w points in the time series. To create the next ordered set, it will shift this window over by s points, where s is the number of skip points defined by the user. This results in the points s to $w + s$ being the next window. It continues this process until it cannot skip over s points and still maintain a window size of w. For each of these windows, it will compute the persistence landscape using the Vietoris–Rips filtration.

We must convert each of these landscapes into a single value that can represent the overall homology of the window and can be compared to other windows. We will use the L^2 norm of each of these landscapes. The L^p norm is a valid choice, as [8] showed it provides

Table 5.1: Hardware Specifications

	HEDT	Server
Memory	48GB	125 GB
CPU	Intel Xeon Silver 4114	2x Intel Xeon E5-2640 v4
OS	Windows 10	VMWare Sphere
User-level OS	Windows 10	Ubuntu 18.04LTS
Multiuser system	no	yes

a Banach space structure. The L^2 norms form a natural ordering based on the window they were calculated. This ordering on the L^2 norms forms a univariate time-series. From this point, one could use any univariate time series analysis technique for classification or anomaly detection in order to either classify or detect anomalies, respectively. We used RPM and TSAT, described in Section 2.2.6, for our time series analysis.

For the experiments described in this chapter we used two different machines: A high-end desktop (HEDT) and a multiuser server. For all experiments containing relative speed-up results, all other users were logged off the multiuser system apart from user experimenting. This was done to minimize the experimental noise introduced into the results from other users' activities. Times were captured by taking the wall time difference before and after the TDA step only (dimensionality reduction stage). This was done to demonstrate the time savings from changing the algorithms used. The exact specifications of each machine are in table 5.1.

5.2 Botnet Datasets

We examined two Botnet datasets. The first is ISCX botnet 2014 from the University of New Brunswick Canadian Institute for Cybersecurity [24]. This dataset has the three main attributes of a good dataset: generality, realism, and representativeness. They accomplished this by including many different types and manifestations of botnets from multiple data collection points. The data format is multiple pcap files. Having access to the raw pcap files greatly increases the different types of techniques that can be evaluated on the dataset

as pcap is the industry standard for network capture and replay. The result is a greater number of people who can use the dataset to evaluate their algorithm, making it easier for the dataset to become a standard for evaluation.

The training dataset contains seven different botnets. Additionally, the dataset is nearly perfectly balanced. Each of which is either IRC, HTTP, or P2P, with 43.92% of the traffic being malicious. The testing set contains 16 different botnets. Each of these botnets is one of the previously seen types from the training dataset, with 44.97% being malicious. All of the original botnet types appear in the testing dataset. This enables one to test the previously mentioned different types of generalization for the model in question.

The other dataset, which is looked at briefly in this thesis, is [17]. This dataset provided only the argus file for evaluation with minimal attributes. Having access to the data in this format made it difficult, or impossible in some cases, to modify the dataset to add particular missing features. It also only contained a small number of botnets per test/train set, and the dataset was much smaller than the ISCX 2014 botnet dataset [24]. These reasons pushed us to switch to [24] part way through our evaluation.

5.3 Preprocessing

The different datasets need to be processed in different ways because the data was provided in different formats with different features. The input for RPM through TSAT is a time series json split into different example time series each of the same length. Feature engineering was only done to the ISCX dataset. This is simply a result of the most effort being put toward this dataset. This is different from the network datasets which needed to be split into equal length example time series. The steps taken for these two datasets are as follows:

1. pcap to CSV and feature selection using argus
2. sample labeling in python
3. feature engineering in python (only done on the ISCX dataset)

4. test/train split

5. CSV to JSON in python

Feature Selection using Argus The ISCX dataset comes as pcap files, which means that feature selection is needed. Feature selection is made using open argus. The features are in table 5.2. These features are chosen out of the possible 145 is because, as stated by [19], not all of the available features are thought to be helpful in the botnet detection problem [19]. These are very similar to the features that are used in [19]. The reason for the difference is because some of the features never changed in value in the ISCX botnet dataset and thus added no information. If these features in the dataset were to change, it might be worthwhile to add these features back into the dataset. More feature extraction could be done to the dataset than what was done here, but because this thesis aims to explore TDA's power in botnet detection and not to achieve the highest metrics, this was not explored more in-depth.

The output from open argus is customizable to most formats: CSV is used here. The source IP, destination IP, and port were not used during classification to enable better generalization to new networks.

Time-Series Split The input into TSAT needs to be formatted as a collection of time-series. Thus we must decide how to group the data and where to split the different samples. We decided to order all of the data sequentially according to time and then split the data into 500 sample chunks with no overlap between any chunks. This number was chosen as it was the minimum seen to allow for patterns to be found using representative pattern matching (RPM). This results in having the maximum number of samples possible for testing and training and ensuring that botnets are detected as soon as possible if deployed in a real-time system.

Data labeling A result of the data provided by the ISCX botnet data being provided as pcap files means that the data is unlabeled. ISCX does provide a list of the malicious

Table 5.2: Features extracted from the ISCX 2014 Botnet dataset

	Argus Argument
Source IP address	saddr
Destination IP address	daddr
Direction of transaction	dir
Protocol	proto
Record total duration	dur
Total transaction packet count	pkts
Packet count from source to destination	spkts
Packet count from destination to source	dpkts
Source interpacket arrival time (mSec)	sintpkt
Source Idle interpacket arrival time (mSec)	sintpktidl
Destination interpacket arrival time (mSec)	dintpkt
Destination idle interpacket arrival time (mSec)	dintpktidl
Total Transaction bytes	bytes
Source to destination transaction bytes	sbytes
Destination to source transaction bytes	dbytes
Source active interpacket arrival time (mSec)	sintpktact
Destination active interpacket arrival time (mSec)	dintpktact
Mean of the flow packet size transmited by the src (initiator)	smeansz
Mean of the flow packet size transmited by the dst (target)	dmeansz
Minimum packet size for traffic transmited by the src	sminsz
Minimum packet size for traffic transmited by the dst	dminsz
Maximum packet size for traffic transmited by the src	smaxsz
Maximum packet size for traffic transmited by the dst	dmaxsz
Bits per second	load
Source bits per second	sload
Destination bits per second	dload
Packets per second	rate
Source packets per second	srate
Destination packets per second	drate

IP's seen in the network capture, however. The data is labeled so that if there is a bad connection in the sample chunk, then the entire chunk is labeled as malicious. If there is no bad connection in the chunk, then the chunk is labeled as benign.

Feature Engineering Extra feature engineering was done on top of the initial feature selection from open argus in some experiments. If this is done in an experiment, the following process is followed. The data is read into a python file as a CSV, then min-max normalization, and one-hot encoding are done using pandas and NumPy. Then, features with all NaN values are removed to shrink the dataset's size, as these features provided no additional information to the model. Lastly, the data is exported as a CSV.

Test/Train Split The testing and training data set are manipulated in a subset of the experiments. If there is a manipulation of the testing and training data in an experiment, the following process is followed. A random value between 0 and 1 is assigned to each sample chunk. If the value is greater than a user-defined threshold value, then that value is assigned to the training set. If it is less than the value, it is assigned to the testing set. This was only done when the original training and testing datasets were not used, but instead, the original training dataset was split into new training and testing sets.

CSV to JSON The final stage of the data preprocessing pipeline is a conversion from CSV to JSON. This is done for compatibility reasons to use RPM through the time-series analysis tool (TSAT). The conversion is done using pandas in python and is provided along with the other source code on github.com/tph5595/.

5.4 Plane Sweep Algorithm Analysis Validation

The purpose of these experiments is to see if the plane-sweep algorithm from 4.1 performed better than the original algorithms presented in [12].

Table 5.3: Algorithm runtime (sec) at the 95% confidence level

	Original	Plane-sweep	Plane-sweep with filtering
Mean	758.2–866.2	700.2–709.7	399.2–430.7
Variance	20370.3–50050.9	143.7–353.2	1739.1–4273.0

Table 5.4: Mean weighted MCC and weighted F1

	Original	Plane-sweep	Plane-sweep with filtering
Weighted MCC	0.0452	0.504	0.51
Weighted F1	0.023	0.093	0.0999

5.4.1 Method

A grid search is done using the ISCX botnet 2014 data set on both algorithms to test this. The grid search is performed on the window size (5–40 with a step size of 5) and the max rips radius (1–5 with a step size of 1). Experiments were run with the barcode filtering step keeping every birth-death pair and only keeping the top 20^2 pairs. This is done to determine what is providing the speedup. The results can be found in Table 5.3. There is a speedup from both the barcode filtering step as well as the plane-sweep persistent landscape algorithm. Filtering out barcodes to this level did not affect the weighted F1 or weighted MCC at the 95% confidence level but did have a meaningful effect on the time 5.4. Further experiments are done to see what percentage of birth-death pairs are needed before there is a significant drop off in the model's quality. With a 95% confidence level, there is no difference in the weighted F1 or MCC when only keeping the top persistence landscape vs. keeping all of the persistence landscapes when using the ISCX botnet 2014 dataset. Also, with a 95% confidence level, the plane-sweep algorithm will perform better on average than the algorithm from Bubenik et al. [12].

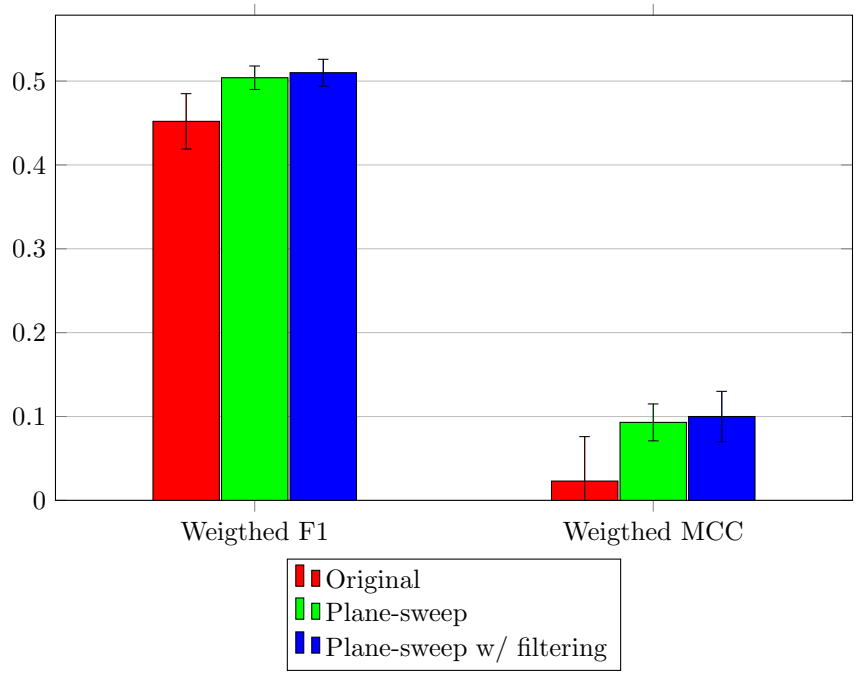

Figure 5.1: Mean weighted MCC and weighted F1 (95% confidence level)

Table 5.5: Max Metrics on different datasets

	Original test/train set	Split training	Test/train equal
Weighted MCC	0.246	0.727	0.993
Weighted F1	0.595	0.867	0.997

5.4.2 Generalization

The initial findings on the ISCX botnet 2014 data set were underwhelming compared to the success achieved using TDA on other data sets [5]. The result was a weighted F1 of 0.595 and a weighted MCC of 0.246 when using the provided testing and training data sets. In order to determine why the model was performing poorly on this data set compared to other models, such as [19], the provided training data set is split into testing (30%) and training (70%) sets. This helped diagnosing the problem because it removes the variable of new botnet types in the original testing data set as can be seen in Table 5.5.

Training the model on this new data set increased the weighted F1 to 0.867 and the weighted MCC to 0.727. This confirms the model was having trouble generalizing to new types of botnets. This could result from not enough training data, which is known to affect performance, as shown by Postol et al. [4]. This shows that this TDA method is capable of reducing the dimensionality of the data into a single dimension while keeping the data separable.

Experiments were conducted to quantify the effect of the loss of information caused by using TDA for time-series dimensionality reduction for botnet classification. The model was trained and tested on the ISCX training data set. The result was that the model was able to achieve a weighted F1 of 0.997 and weighted MCC of 0.993 (only one misclassified example). This can be seen in Table 5.5. This confirms TDA's ability to reduce the dimensionality of botnet time-series data from 26 dimensions to 1 with minimal loss of information. This instills confidence that univariate time series analysis techniques can be used on multivariate data. There is a greater understanding of how to analyze univariate than multivariate time-series data—thus, opening the door for these techniques to be used on multivariate time series data. Further work should be done with TDA with other time-series analysis models, techniques, and features to determine the best combination.

It should be noted that the best parameters for all of the different experiments in this section were the same: window size 20 and max rips radius of 4. There is no reason this should have been the case as different training sets were created for all different experiments.

Therefore, it is logical to assume that some attribute is inherent to the network that is causing this. This helps with training various TDA implementations on the same network as it seems only a single search for the correct parameters is needed.

Chapter 6: Conclusion

This work intended to show the power of capturing changes in topology using persistence landscapes from topological data analysis for botnet detection. The technique has the power to transform multiattribute time-series data into univariate time-series. This feature can be used standalone and be added back in with the original data as a feature engineering technique. Importantly, this technique has shown success in a noisy dataset where traditional techniques have trouble.

6.1 Accomplishments

I have shown that persistence landscapes are a viable dimensionality reduction technique for botnet detection. This was shown through experimental validation using the ISCX 2014 botnet dataset. I have shown minimal information lost using this technique and that the data is still separable after the technique is applied. Additionally, intuition for how to set the hyperparameters is given. This comes from analyzing the experiments conducted and observations on the definitions behind the traditionally opaque topological vocabulary. This information arms future researchers to be able to use the techniques presented immediately.

I have also provided a substantial 2.27x speed-up for the computation of the persistence landscapes. This speed-up is shown to have no drawbacks from the perspective of weighted F1 and MCC. This speed-up is the result of a novel output-sensitive algorithm. Additionally, a new property of persistence barcodes was discovered that allows this algorithm to be used to its fullest potential.

6.2 Limitations of the Work

The current technique is unable to generalize well to unseen botnets. This could result from not enough data to learn, an issue with the features the model is given, or with the hyperparameters chosen. Additionally, no automated method for determining the hyperparameters was found, nor was any insight gained into how to set the hyperparameters for a dataset given the dataset's statistics. This should be possible given enough data from properly tuned models on various datasets.

Also, a result of the persistence landscape seemingly performing just as well with only the top persistence landscape means that new algorithms can be developed that perform faster given this new constraint. Part of this new algorithm is that the barcode filtering step could be combined with the persistence landscape generation stage.

6.3 Future Work

Future work should be done using this technique on more datasets, combinations of features, and different models. It is likely the case that the output from this model favors a specific type of data or pairs well with specific types of models. Additionally, it should be explored if adding this technique to state-of-the-art models improves their performance. This including using reinforcement learning and other deep learning models.

Further work should quantify persistence landscapes' success in other security, systems, and networking areas. This technique may succeed in these other domains as the data can be similar to that of botnet detection.

Future work needs to be dedicated to determining a way to inform the hyperparameter search through attributes of a given dataset or type of dataset. This will significantly speed up the time to discover new uses for persistence landscapes as time can be spent running new experiments and not worrying about fine-tuning hyperparameters.

There should be work done on speeding up the current algorithm. This comes from the fact that keeping all of the landscapes gave statistically equivalent results when only the

top landscape was kept and intermediary settings. It should be possible to achieve significant speed-ups by directly computing the L^p norm of all landscapes or the top landscape. Creating a multithreaded version of the algorithm should also be explored.

Additionally, there should be ways to speed-up the calculation of birth-death pairs on time-series data. This is a new application of these techniques, and most of the packages available are intended to take in a point cloud. There is a lot of information overlap between time windows in the implementation of persistence landscapes for time-series used in this thesis. Taking advantage of this information should lead to speed-ups and best practices to compute topological properties in the future.

Bibliography

Bibliography

[1] E. Nakashima and J. Greene, "Microsoft seeks to disrupt russian criminal botnet it fears could seek to sow confusion in the presidential election," Oct 2020. [Online]. Available: https://www.washingtonpost.com/technology/2020/10/12/microsoft-trickbot-ransomware/

[2] Kaspersky, "Zeus virus," Oct 2018. [Online]. Available: https://usa.kaspersky.com/resource-center/threats/zeus-virus

[3] "Famous ddos attacks — the largest ddos attacks of all time." [Online]. Available: https://www.cloudflare.com/learning/ddos/famous-ddos-attacks/

[4] M. Postol, C. Diaz, R. Simon, and D. Wicke, "Time-series data analysis for classification of noisy and incomplete internet-of-things datasets," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 2019, pp. 1543–1550.

[5] M. Gidea and Y. Katz, "Topological data analysis of financial time series: Landscapes of crashes," *Physica A: Statistical Mechanics and its Applications*, vol. 491, 10 2017.

[6] L. M. Seversky, S. Davis, and M. Berger, "On time-series topological data analysis: New data and opportunities," *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1014–1022, 2016.

[7] A. J. Zomorodian, *Topology for Computing*. USA: Cambridge University Press, 2009.

[8] P. Bubenik, "Statistical topological data analysis using persistence landscapes," *Journal of Machine Learning Research*, vol. 16, no. 3, pp. 77–102, 2015. [Online]. Available: http://jmlr.org/papers/v16/bubenik15a.html

[9] J. Liu, S. Jeng, and Y. Yang, "Applying topological persistence in convolutional neural network for music audio signals," *CoRR*, vol. abs/1608.07373, 2016. [Online]. Available: http://arxiv.org/abs/1608.07373

[10] C. Bullard. (2020) Argus project. [Online]. Available: https://openargus.org/

[11] X. Wang, J. Lin, P. Senin, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, and S. Frankenstein, "Rpm: Representative pattern mining for efficient time series classification," in *EDBT*, 2016.

[12] P. Bubenik and P. Dlotko, "A persistence landscapes toolbox for topological statistics," *CoRR*, vol. abs/1501.00179, 2015. [Online]. Available: http://arxiv.org/abs/1501.00179

[13] J. A. Perea and J. Harer, "Sliding windows and persistence: An application of topological methods to signal analysis," *Foundations of Computational Mathematics*, vol. 15, no. 3, pp. 799–838, Jun 2015. [Online]. Available: https://doi.org/10.1007/s10208-014-9206-z

[14] M. Feily, A. Shahrestani, and S. Ramadass, "A survey of botnet and botnet detection," in *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, 2009, pp. 268–273.

[15] S. GarcÃa, A. Zunino, and M. Campo, "Survey on network-based botnet detection methods," *Security and Communication Networks*, vol. 7, no. 5, pp. 878–903, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.800

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. USA: USENIX Association, 2008, p. 139–154.

[17] S. García, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *Computers & Security*, vol. 45, pp. 100 – 123, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404814000923

[18] G. Gu, P. Porras, V. Yegneswaran, and M. Fong, "Bothunter: Detecting malware infection through ids-driven dialog correlation," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/bothunter-detecting-malware-infection-through-ids-driven

[19] S. Homayoun, M. Ahmadzadeh, S. Hashemi, A. Dehghantanha, and R. Khayami, *BoT-Shark: A Deep Learning Approach for Botnet Traffic Detection*. Cham: Springer International Publishing, 2018, pp. 137–153.

[20] M. Alauthman, N. Aslam, M. Al-kasassbeh, S. Khan, A. Al-Qerem, and K.-K. Raymond Choo, "An efficient reinforcement learning-based botnet detection approach," *Journal of Network and Computer Applications*, vol. 150, p. 102479, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S108480451930339X

[21] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008.

[22] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington, "A roadmap for the computation of persistent homology," *EPJ Data Science*, vol. 6, no. 1, Aug 2017. [Online]. Available: http://dx.doi.org/10.1140/epjds/s13688-017-0109-5

[23] C. Tralie, N. Saul, and R. Bar-On, "Ripser.py: A lean persistent homology library for python," *The Journal of Open Source Software*, vol. 3, no. 29, p. 925, Sep 2018. [Online]. Available: https://doi.org/10.21105/joss.00925

[24] E. Biglar Beigi, H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani, "Towards effective feature selection in machine learning-based botnet detection approaches," in *2014 IEEE Conference on Communications and Network Security*, 2014, pp. 247–255.

Curriculum Vitae

Taylor Henderson received his Bachelor of Science in Computer Science from George Mason University in 2014. During his undergraduate studies, he served as both a teaching assistant and research assistant at George Mason University. Additionally, he helped form a startup focused on transitioning ICS security research from Pacific Northwest National Laboratory and Idaho National Laboratory into the private sector. His research interests include machine learning and systems and networking security. He plans to continue his studies at George Mason University to earn to Ph.D. in Computer Science.