

2020

## A Hierarchical Temporal Memory Sequence Classifier for Streaming Data

Jeffrey Barnett

Follow this and additional works at: [https://nsuworks.nova.edu/gscis\\_etd](https://nsuworks.nova.edu/gscis_etd)



Part of the [Computer Sciences Commons](#)

### Share Feedback About This Item

---

This Dissertation is brought to you by the College of Computing and Engineering at NSUWorks. It has been accepted for inclusion in CCE Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact [nsuworks@nova.edu](mailto:nsuworks@nova.edu).

A Hierarchical Temporal Memory Sequence Classifier  
for Streaming Data

by

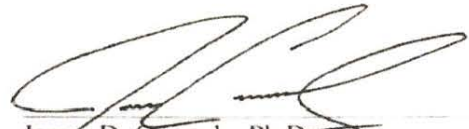
Jeffrey V. Barnett

A dissertation submitted in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy  
in  
Computer Science

College of Computing and Engineering  
Nova Southeastern University

2020

We hereby certify that this dissertation, submitted by Jeffrey Barnett conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.




James D. Cannady, Ph.D.  
Chairperson of Dissertation Committee

07/30/2020  
Date



Wei Li, Ph.D.  
Dissertation Committee Member


7/30/2020  
Date



Peixiang Liu, Ph.D.  
Dissertation Committee Member

7/30/2020  
Date

Approved:



Meline Kevorkian, Ed.D.  
Dean, College of Computing and Engineering

07/30/2020  
Date

College of Computing and Engineering  
Nova Southeastern University

2020

An Abstract of a Dissertation Submitted to Nova Southeastern University  
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Hierarchical Temporal Memory Sequence Classifier  
for Streaming Data

by  
Jeffrey V. Barnett  
2020

Real-world data streams often contain concept drift and noise. Additionally, it is often the case that due to their very nature, these real-world data streams also include temporal dependencies between data. Classifying data streams with one or more of these characteristics is exceptionally challenging. Classification of data within data streams is currently the primary focus of research efforts in many fields (i.e., intrusion detection, data mining, machine learning). Hierarchical Temporal Memory (HTM) is a type of sequence memory that exhibits some of the predictive and anomaly detection properties of the neocortex. HTM algorithms conduct training through exposure to a stream of sensory data and are thus suited for continuous online learning. This research developed an HTM sequence classifier aimed at classifying streaming data, which contained concept drift, noise, and temporal dependencies. The HTM sequence classifier was fed both artificial and real-world data streams and evaluated using the prequential evaluation method. Cost measures for accuracy, CPU-time, and RAM usage were calculated for each data stream and compared against a variety of modern classifiers (e.g., Accuracy Weighted Ensemble, Adaptive Random Forest, Dynamic Weighted Majority, Leverage Bagging, Online Boosting ensemble, and Very Fast Decision Tree). The HTM sequence classifier performed well when the data streams contained concept drift, noise, and temporal dependencies, but was not the most suitable classifier of those compared against when provided data streams did not include temporal dependencies. Finally, this research explored the suitability of the HTM sequence classifier for detecting stalling code within evasive malware. The results were promising as they showed the HTM sequence classifier capable of predicting coding sequences of an executable file by learning the sequence patterns of the x86 EFLAGS register. The HTM classifier plotted these predictions in a cardiogram-like graph for quick analysis by reverse engineers of malware. This research highlights the potential of HTM technology for application in online classification problems and the detection of evasive malware.

## **Acknowledgments**

First and foremost, my appreciation goes out to my wife Hanan, whose support was unwavering, and understanding and compassion went beyond anything that a husband could ever expect.

Secondly, my heartfelt thanks go out to my dissertation committee of Drs. Cannady, Liu, and Li for agreeing to support my research and for giving their time, talents, and feedback. A special thanks to Dr. Cannady, who first inspired me with his course on Artificial Intelligence in 2003 and has made a lifelong impact on my interest in academia. I'd also like to thank several of my students who provided various forms of support: Joshua for explaining some of the nuances of Python, Felix, and Jacob for helping me record the statistical results from hundreds of output files. Finally, this work would not have been possible without the many HTM tutorial videos and live data streams provided by Numenta's Matt Taylor, whose recent passing shocked the HTM community. As Matt would often say, "its time for HTM school!"

## Table of Contents

<b>Approval</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Equations</b>	<b>xii</b>

### Chapters

<b>1. Introduction</b>		<b>1</b>
Background	1	
Problem Statement	3	
Goals	5	
Objectives	6	
Relevance and Significance	6	
Barriers and Issues	7	
Definition of Terms	9	
Summary	15	
<b>2. Literature Review</b>		<b>17</b>
Overview	17	
Current State of Classifiers	17	
Current Approaches for Addressing Concept Drift	18	
Temporal Patterns in Data Streams	23	
Evasive Malware	25	
Malware Evasion Techniques	31	
Hierarchical Temporal Memory	37	
Summary	46	
<b>3. Methodology</b>		<b>47</b>
Overview	47	
Classification	47	
Data Sets	48	
HTM Classifier	50	
HTM Algorithm	50	
HTM Classifier Encoders	51	
Experiments	52	
Statistical Significance Validation	58	
Computing Resources Used	59	

Summary	60
<b>4. Results</b>	<b>62</b>
Experiment I (Artificial Data Streams)	63
Experiment II (Real-world Data Streams)	72
Experiment III (Simulated Malware)	76
Statistical Analysis	89
Comparisons to Other Literature	91
Summary of Results	96
<b>5. Conclusions, Implications, Recommendations, &amp; Summary</b>	<b>97</b>
Conclusions	97
Summary	98
Implications	100
Recommendations	101
<b>Appendices</b>	<b>103</b>
<b>A:</b> Hyperplane Features Description for Experiment I	103
<b>B:</b> LED Features Description for Experiment I	104
<b>C:</b> Random Tree Features Description for Experiment I	105
<b>D:</b> SEA Features Description for Experiment I	106
<b>E:</b> Electricity Features Description for Experiment II	107
<b>F:</b> Airlines Features Description for Experiment II	108
<b>G:</b> Poker Features Description for Experiment II	109
<b>H:</b> SDR Encoder Dictionaries	110
<b>I:</b> Electricity Spatial Pooler, and Temporal Memory Specs	112
<b>J:</b> Simulated Malware using Stalling Code	113
<b>K:</b> Assembly Language Sorting Algorithm	114
<b>L:</b> IDAPro Python script to disassemble x86 executables	132
<b>M:</b> Executable Spatial Pooler and Temporal Memory Specifications	135
<b>N:</b> HTM Classifier (EFLAGS Version) Python Code	136
<b>Reference List</b>	<b>152</b>

## List of Tables

### Tables

1. Classifier Server Hardware	60
2. Malware Analysis Software	60
3. HTM Classifier Accuracy - Artificial Data sets with 0% noise	64
4. HTM Classifier Accuracy - Artificial Data sets with 10% noise	65
5. HTM Classifier Avg. Run-Time (secs) - Artificial Data sets with 0% noise	66
6. HTM Classifier Avg. Run-Time (secs) - Artificial Data sets with 10% noise	66
7. HTM Classifier Avg. RAM-Usage (K) - Artificial Data sets with 0% noise	67
8. HTM Classifier Avg. RAM-Usage (K) - Artificial Data sets with 10% noise	68
9. Accuracy (%) of the Classifiers with Real-World Data	74
10. Average run-time (in seconds) of the Classifiers with Real-World Data	75
11. Average RAM usage (in K) of the Classifiers with Real-World data sets	75
12. HTM classifier Malware Analysis EFLAGS prediction accuracy % 1-step	76
13. HTM classifier Malware Analysis EFLAGS prediction accuracy % 5-step	76
14. HTM classifier Malware Analysis RAM usage in K	77
15. Average classifier rank including Artificial and Real-World data sets	90
16. Nemenyi p-values, with no further adjustment	91
17. Comparison to Other Research - Average Accuracy	92
18. Comparison to Other Research - Average CPU run-time (seconds)	94
19. Hyperplane Features Description for Experiment I	103
20. LED Features Description for Experiment I	104
21. Random Tree Features Description for Experiment I	105
22. SEA Features Description for Experiment I	106



23. Electricity Features Description for Experiment II	107
24. Airlines Features Description for Experiment	108
25. Poker Features Description for Experiment II	109

## List of Figures

### Figures

1. Major Milestones in the Evolution of Evasive Techniques	26
2. SDR Overlap set. Showing random SDR y not matching x'	42
3. Classifier Server Applications and File Structure	54
4. x86 EFLAGS diagram	56
5. Malware Analysis Machine	57
6. Malware Analysis Network	58
7. Abrupt and Recurrent concept drift – SEA data with 0% noise	69
8. Gradual and recurrent concept drift– SEA data with 0% noise	70
9. Abrupt and Recurrent concept drift – SEA data with 10% noise	71
10. Gradual and recurrent concept drift– SEA data with 10% noise	71
11. Electricity dataset: HTM Accuracy – 1 and 5 step predictions	73
12. Stalling code: HTM Carry Flag Accuracy - 1 and 5 step predictions	78
13. Sorting code: HTM Carry Flag Accuracy - 1 and 5 step predictions	78
14. Stalling code: HTM Parity Flag Accuracy - 1 and 5 step predictions	79
15. Sorting code: HTM Parity Flag Accuracy - 1 and 5 step predictions	80
16. Stalling code: HTM Auxiliary Flag Accuracy - 1 and 5 step predictions	81
17. Sorting code: HTM Auxiliary Flag Accuracy - 1 and 5 step predictions	81
18. Stalling code: HTM Zero Flag Accuracy - 1 and 5 step predictions	82
19. Sorting code: HTM Zero Flag Accuracy - 1 and 5 step predictions	83
20. Stalling code: HTM Sign Flag Accuracy - 1 and 5 step predictions	84
21. Sorting code: HTM Sign Flag Accuracy - 1 and 5 step predictions	84
22. Stalling code: HTM Trap Flag Accuracy - 1 and 5 step predictions	85

23. Sorting code: HTM Trap Flag Accuracy - 1 and 5 step predictions	86
24. Stalling code: HTM Direction Flag accuracy - 1 and 5 step predictions	87
25. Sorting code: HTM Direction Flag accuracy - 1 and 5 step predictions	87
26. Stalling code: HTM Overflow Flag accuracy - 1 and 5 step predictions	88
27. Sorting code: HTM Overflow Flag accuracy - 1 and 5 step predictions	89
28. Nemenyi test results with a 90% confidence level	91

## List of Equations

### Equations

1. Match	11
2. Overlap	11
3. Friedman statistic	59

# Chapter 1

## Introduction

### Background

The process of analyzing data to discover hidden relationships and predict future trends has a lengthy history. Commonly referred to as knowledge discovery in databases before the 1990s (Fayyad et al., 1996), the term is now more widely known as data mining. Data mining comprises three intertwined scientific disciplines: statistics, artificial intelligence, and machine learning.

Over the last twenty years, advances in processing power and speed have enabled consumers of big data to move beyond manual, tedious, and time-consuming practices to rapid, effortless, and computerized data analysis. The more complicated the data sets gathered, the more potential there is to discover relevant insights. Telecommunications providers, retailers, banks, manufacturers, and insurers are just a few examples of users with interest in using data mining to find relationships in streaming data. Everything from price optimization, demographics to how the economy, risk, competition, and social media are affecting their business models, revenues, and customer relationships.

Data mining makes heavy use of models generated by classifiers. Unfortunately, these generated models become quickly obsolete due to the occurrence of changes, also known as "concept drift" (Mouchaweh, 2016). More formally, concept drift is a term used to describe changes in the statistical properties of an object or learned structure that occur over time, which eventually leads to a drastic drop in classification accuracy. Concept drifts are generally categorized as being either abrupt, gradual, or recurrent (Bifet et al., 2017; Mouchaweh, 2016). Abrupt or sudden concept drift occurs when the

distribution of a concept has remained unchanged for a long time, then changes in a few steps to a significantly different one ).

An example of this would be the sudden shift of customers' shopping habits from one product to another based on some personal change in taste. Gradual concept drift occurs when the new concept replaces the old slowly over time (Bifet et al., 2017). All forms of concept drift are a significant issue, especially when learning from data streams as it requires learners to be adaptive to dynamic changes (Wang et al., 2017).

The limitation of existing classifiers is that almost all assume that the distribution of the data is independent and identical (IID) (Bifet et al., 2017; Duong et al., 2018). In a streaming environment, no part of the IID assumption remains valid. Many of the modern-day data streams, by their very nature, contain temporal features (e.g., electricity usage, airline flight data, intrusion detection, stock market prices). It is also often the case that for specific periods the labels or classes of instances are correlated (aka temporal dependence) (Bifet et al., 2017).

In 2004, Jeff Hawkins, a renowned neuroscientist and software developer, designed the Hierarchical Temporal Memory (HTM) framework as a type of neural network. HTM represents a shift away from the artificial neuron typically used in machine learning and artificial neural networks (Hawkins et al., 2017). The HTM framework consists of artificial neurons that mimic their biological counterparts' use of binary synapses and learns by modeling the growth of new synapses and the decay of unused synapses. As a result, HTM receives its training through exposure to a stream of sensory data; this exposure is what determines the HTMs' capabilities (Hawkins, 2011a). This research builds upon earlier work by (Hawkins et al., 2017), leveraging the underlying neocortical theories resident within the HTM framework. Based on studies of

the neocortex, HTM regions make inference and prediction on complex data streams. HTM algorithms also learn temporal sequences (dependencies) that exists in a data stream. Hawkins (2011c) states that identifying temporal dependencies is difficult because (a) the system may not know when sequences start and end. (b) there may be overlapping sequences occurring at the same time. (c) learning has to occur continuously and has to happen in the presence of noise.

Hawkins (2011c) further goes on to explain that inference in an HTM region continuously analyzes streaming data and matches them to previously learned sequences. An HTM region can identify temporal patterns, but usually, it is more fluid, analogous to how you can recognize a melody starting from anywhere. Because HTM regions use distributed representations, its use of sequence memory and inference make HTMs a viable solution for handling concept drifts, noisy data, and finding temporal dependencies within in streaming data.

This report described an HTM sequence classifier that can classify streaming data that contains concept drift, noise, and temporal dependencies. Such a classifier is potentially applicable to a wide array of real-world applications, including the detection of malware that utilizes evasion techniques such as stalling.

### **Problem Statement**

There currently does not exist an effective method for classifying sequential data in data streams. The overwhelming volume of data coupled with concept drift, noise, and temporal dependencies leads to a drastic drop in classification accuracy (Dongre & Malik, 2014). A great deal of active research utilizing Deep Learning techniques is currently underway in an attempt to address this problem; however, an effective sequence classifier continues to elude researchers.

Many fields such as cybersecurity, weather forecasting, and data mining would benefit a great deal if an effective sequence classifier capable of identifying anomalies in streaming data existed. Government agencies, private institutions, and citizens rely heavily on computer networks to store their private information. The problem with this dependency is that it creates a critical area of vulnerability that is exploited by malware authors. According to Jadhav, Vidyarthi, and Hemavathy (2016), the advent of malware that can modify itself using evasion techniques threatens to undermine the integrity, safety, and security of information that is needed by every organization. Cannady (2013) stated that while researchers have conducted a great deal of research in intrusion detection, a reliable solution has yet to surface. The Commander of United States Cyber Command testified before the Senate Armed Services Committee that, "Cyber-attacks could hamper our military forces, interfering with deployments, command, and control, and supply functions, in addition to the broader impact such events could have across our society" (Rogers, 2016, p. 2).

The effectiveness of evasive malware and its associated financial costs are substantial. According to Gandel (2015), Beale chief executive officer of Lloyds, a British insurance company that is known for specializing in obscure risks such as hack coverage, reports that cyberattacks cost businesses as much as \$400 billion a year. Beale goes on to report that the demand for cyber insurance has grown considerably in recent years. The costs of evasive malware are more than just financial; it can also have high political costs as well. According to McAfee (2017), the US Democratic National Committee attack in the fall of 2016 was likely conducted using a well-known password-stealing virus named Fareit that had been modified to use one or more evasion techniques.



Understanding the methods and techniques employed by malware authors is crucial to the development of software that is capable of defending against evasive malware (Barria et al., 2016). In this regard, anti-malware vendors make extensive use of signature-based techniques for identifying malware. Intrusion detection systems can only monitor processes for a limited amount of time before labeling them as benign. According to Osorio et al. (2015), the problem is that evasive malware can take advantage of this limitation by delaying their harmful behavior long enough to exceed the intrusion detection systems window for identifying malicious behavior. Evasive malware often employs methods such as implementing do nothing code (stalling code), waiting for user interaction or invoking sleep calls to alter the timing sequence of its behavior as a means to evading detection by intrusion detection systems.

In response to this problem, this research developed an HTM sequence classifier that classifies streaming data with concept drift, noise, and temporal dependencies. Also, this research evaluated the HTM classifier for its potential as a solution for detecting malware, which utilizes stalling code (trivial instructions) to alter its timing sequence as a means of avoiding detection by intrusion detection systems.

### **Dissertation Goal**

The goal of this research was to develop a sequence classifier that can classify data in a data stream containing concept drift, noise, and temporal dependencies. The HTM sequence classifier was compared against other concept drift oriented classifiers using four artificial and three real-world data streams that contained concept drift, noise, and temporal dependencies. This research also examined the use of the HTM sequence classifier as a potential solution for detecting evasive malware by training it on a toy model that contained stalling code.

## **Objectives**

Three objectives were achieved based on the stated goal:

1. Analyze and compare the HTM sequence classifier against other well-known concept drift oriented sequence classifiers on artificial data streams that contain concept drift (i.e., abrupt, gradual, and recurrent), noise, but without temporal dependencies.

2. Analyze and compare the HTM sequence classifier against other well-known concept drift oriented sequence classifiers on real-world data streams that contain concept drift (i.e., abrupt, gradual, and recurrent), noise, and temporal dependencies.

3. Test the HTM sequence classifier as a potential solution for identifying evasive malware containing stalling code.

## **Relevance and Significance**

Despite recent advancements in machine learning, modern classifiers are still susceptible to concept drift. According to Madireddy et al. (2019), high-performance computing (HPC) systems, used to solve complex computational problems, suffered from concept drift stemming from root causes such as disk-hog, network-hog, disk-busy, and packet-loss faults. According to Park, Seo, Jeong, and Kim (2018), many network intrusion detection systems have to rebuild their models, which are computationally expensive due to concept drift. Adding to the problem, the detection of concept drift itself is a challenging subject that is popular in current academic research efforts.

According to (Jadhav et al., 2016), sandboxing (dynamic malware detection) also has suffered from its own set of limitations stemming from concept drift. Malware writers often embed in their code the ability to discover virtualized environments by checking for live internet access, or specific system properties inherent to virtualized environments. Malware writers often employ a "wait and seek" (aka dormant malware), a technique

where knowing the execution time limitations of sandboxes, the malware waits until this time has passed (Osorio et al., 2015).

Osorio et al. (2015) also showed that malware that uses polymorphic and metamorphic obfuscation techniques combined with "sandboxing evasion techniques" reduced the effectiveness of both static detection (signature matching), and dynamic detection (sandboxing).

### **Barriers and Issues**

According to Lavin & Ahmad (2015), detecting anomalies in streaming data is a difficult task, because it requires detectors to process data and to make decisions in real-time. Though the academic community has proposed a wide variety of anomaly-based classifiers, benchmarking these techniques adequately concerning their strengths and weaknesses has fallen short of that needed by industry to warrant investment interest (Tavallae et al., 2010). Drew et al., (2016) add that the problem of detecting malware is made even more difficult due to malware developers attempting to avoid the detection of their polymorphic software by constantly changing the algorithm's appearance while keeping its functionality. According to Jadhav et al. (2016), evasive malware generally falls into two main categories, polymorphic and metamorphic. Polymorphic malware can change its appearance, whereas metamorphic malware can automatically re-code itself each time it propagates or is distributed (Jadhav et al., 2016). Evasive malware makes the job of the Intrusion Detection Systems (IDS) even more difficult as current-day solutions struggle to keep pace. Cheng, Tay, and Huang (2012) found that machine learning methods like support vector machines (SVMs) and neural networks used for intrusion detection, generally suffer from long training times, require parameter tuning, and do not perform well in identifying evasive malware.

Earlier research conducted by Wong and Stamp (2006) support Cheng et al., (2012) findings as they discovered that several malware authors had released virus creation kits capable of producing viruses which shared only a small amount of similarity thereby making them extremely difficult to reproduce and thus difficult to detect. Strategies to mitigate the risks associated with this barrier are currently under consideration by looking for candidate solutions that focus on the evasive technique itself rather than the malicious software.

Another potential barrier is that training data is difficult to obtain, especially test data that adequately simulates real-world attacks by evasive malware. According to Moustafa and Slay (2015), the traditional data sets KDD98, KDDCUP99, and NSLKDD used for evaluating the effectiveness of Network Intrusion Detections Systems (NIDS) are out of date and do not reflect modern-day network attacks. The website VirusShare.com contains 33,892,901 samples of the real world viruses in the form of executable files. These samples are packaged into and cataloged into chunks of 65,535 zipped samples per chunk.

### ***Assumptions***

The majority of real-world data streams inherently contain a combination of one or more variants of concept drift (e.g., abrupt, gradual, or recurrent), noise, and temporal dependencies.

Originally an assumption was made that HTM architecture can not accurately classify data without temporal dependencies. It turned out that while this was true, the HTM architecture was able to detect concept drift.

There is a concern as to how much memory and CPU processing power/time will be required to implement an HTM sequence classifier in a real-time data stream, along with

the continuous learning capability that is needed to identify novel attacks. The three cost measures (Accuracy, CPU Run-Time, and RAM Usage) will be used to monitor this concern.

### ***Limitations***

I was unable to find malware labeled as containing stalling code. The sheer number of data samples available from VirusShare.com (33,892,901) made manual reverse engineering efforts impractical.

### ***Delimitations***

One of the goals of this research was to seek a potential solution for the detection of evasive malware that utilized stalling code to evade Intrusion Detections Systems (IDS). The Rombertik malware code snippets published by Giron and Kolbitsch (2015) provide a real-world example of evasive malware that employs the stalling code behavior to evade detection.

### **Definition of Terms**

***Active duty cycle:*** A moving average denoting the frequency of column activation (Hawkins et al., 2017).

***AND:*** See “Intersection” (Hawkins et al., 2017).

***Binary vector:*** An array of bits. SDRs are represented as a binary vector. For the purposes of this research, SDRs are binary vectors, using the notation  $\mathbf{x} = [b_0, \dots, b_{n-1}]$  for an SDR  $x$  (Hawkins et al., 2017).

***Bit:*** A single element of an SDR. Can be in either ON (1) or OFF (0) states (Hawkins et al., 2017).

**Concept Drift:** Žliobaite, Pechenizkiy, and Gama (2016) state that “Concept drift in machine learning and data mining refers to the change in the relationships between input and output data in the underlying problem over time” (p. 1).

**Column:** An HTM region is organized in columns of cells. The Spatial Pooler (SP) operates at the column-level, where a column of cells function as a single computational unit (Hawkins et al., 2017).

**Distal dendrite segment:** Forms synapses with cells within the layer. Every cell has many distal dendrite segments. If the sum of the active synapses on a distal segment exceeds a threshold, then the associated cell enters the predicted state. Since there are multiple distal dendrite segments per cell, a cell’s predictive state is the logical OR operation of several constituent threshold detectors. (Hawkins et al., 2017).

**Encoder:** Converts the native format of data into an SDR that can be fed into an HTM system (Hawkins et al., 2017).

**Hierarchical Temporal Memory (HTM):** A theoretical framework for both biological and machine intelligence (Hawkins et al., 2017).

**HTM learning algorithms:** Describes the set of algorithms in HTM (Hawkins et al., 2017).

**Inhibition:** The mechanism for maintaining sparse activations of neurons. (Hawkins et al., 2017).

**Inhibition radius:** The size of a column’s local neighborhood, within which columns may inhibit each other from becoming active. (Hawkins et al., 2017).

**Intersection:** Of two sets A and B, the intersection is the set that contains all elements of A that also belong to B, but no other elements; the AND operation, denoted  $A \cap B$  (Hawkins et al., 2017).

**Matching:** A match between two SDRs is determined by checking if the two encodings overlap sufficiently. See “Overlap” definition below. For two SDRs  $\mathbf{x}$  and  $\mathbf{y}$ :

$$\mathit{match}(\mathbf{x}, \mathbf{y} | \theta) \equiv \mathit{overlap}(\mathbf{x}, \mathbf{y}) \geq \theta \quad (1)$$

If  $\mathbf{x}$  and  $\mathbf{y}$  have the same cardinality  $w$ , an exact match can be determined by setting  $\theta = w$  (number of active bits chosen to be on).

**Mini-column:** See “Column.”

**Noise:** Meaningless or corrupt data. In SDRs this manifests as randomly flipped ON and OFF bits (Hawkins et al., 2017).

**NuPIC:** Numenta Platform for Intelligent Computing. An open-source community working on HTM (Hawkins et al., 2017).

**OR:** See “Union” (Hawkins et al., 2017).

**Overlap duty cycle:** A moving average denoting the frequency of the column’s overlap value being at least equal to the proximal segment activation threshold (Hawkins et al., 2017).

**Overlap:** The similarity between two SDRs is determined using an overlap score. The overlap score is the number of ON bits in common, or in the same locations, between the vectors. If  $\mathbf{x}$  and  $\mathbf{y}$  are two SDRs, then the overlap can be computed as the dot product:

$$\mathit{overlap}(\mathbf{x}, \mathbf{y}) \equiv \mathbf{x} \cdot \mathbf{y} \quad (2)$$

***Permanence threshold:*** If a synapse's permanence is above this value, it is considered to be fully connected (Hawkins et al., 2017).

***Proximal dendrite segments.*** Forms synapses with feed-forward inputs. The active synapses on this type of segment are linearly summed to determine the feed-forward activation of a column (Hawkins et al., 2017).

***Receptive field:*** The input space that column can potentially connect to (Hawkins et al., 2017).

***Sparse distributed representation (SDR):*** Binary representations of data comprised of many bits with a small percentage of the bits active (1's). The bits in these representations have semantic meaning and that meaning is distributed across the bits (Hawkins et al., 2017).

***Sparsity:*** In a binary vector, the ON bits as a percentage of total bits. Sparsity: At any point in time, a fraction of the  $n$  bits in vector  $\mathbf{x}$  will be ON and the rest will be OFF. Let  $s$  denote the percent of ON bits. Generally, in sparse representations,  $s$  will be substantially less than 50%. (Hawkins et al., 2017).

***Spatial Pooler:*** One of the HTM learning algorithms. In an HTM region, the Spatial Pooler learns the connections to each column from a subset of the inputs, determines the level of input to each column and uses inhibition to select a sparse set of active columns (Hawkins et al., 2017).

***Spatial Pooling:*** a learning mechanism fundamental to both the neocortex and Hierarchical Temporal Memory (HTM) tasked with processing inputs from many different sources without any prior knowledge of what these inputs represent, how many input bits there will be, and what spatial patterns may exist in the input. The



Spatial Pooler does this by accepting an input vector and translating it into an output vector of a different size with a sparse number of activated bits. The output vector of the Spatial Pooler represents mini-columns (Hawkins et al., 2017).

**Synapse:** A junction between cells. In the Spatial Pooling algorithm, synapses on a column's dendritic segment connect to bits in the input space (Hawkins et al., 2017). A synapse can be in the following states:

- Connected – permanence is above the threshold.
- Potential – permanence is below the threshold.
- Unconnected – does not have the ability to connect.

**Temporal Memory:** Learns sequences of patterns over time, and predicts the next pattern as an SDR at the level of cells in columns (Hawkins et al., 2017).

**Temporal Pooler:** One of the HTM learning algorithms. The Temporal Pooler groups together SDRs that are predictable by the lower layer, forming a single representation for many different SDRs (Hawkins et al., 2017).

**Union:** The union of two sets A and B is the set of elements which are in A, in B, or in both A and B; the OR operation, denoted  $A \cup B$  (Hawkins et al., 2017).

**Vector cardinality:** The number of non-zero elements in a vector, or the  $l_0$ -norm. Let  $w$  denote the vector cardinality, which is defined as the total number of ON bits in the vector. If the percent of ON bits in vector  $\mathbf{x}$  is  $s$ , then  $w_{\mathbf{x}} = s \times n = \|\mathbf{x}\|_0$ . (Hawkins et al., 2017).

**Vector size:** Number of elements in a 1-dimensional vector. In an SDR  $\mathbf{x} = [b_0, \dots, b_{n-1}]$ ,  $n$  denotes the size of a vector. Equivalently, we say  $n$  represents the total number of

positions in the vector, the dimensionality of the vector, or the total number of bits.  
(Hawkins et al., 2017).

### ***List of Acronyms***

APK: Android App Packages

ANN: Artificial Neural Network

ARF: Adaptive Random Forest

AWE: Accuracy Weighted Ensemble

AWS: Amazon Web Services

CAN: Controller Area Network

DBN: Deep Belief Neural

DoS: Denial of Service

DWM: Dynamic Weighted Majority

ESN: Echo State Network

EVB: Encrypted Virus program Body

FCG: Function Call Graphs

HMM: Hidden Markov Model

HPC: High-Performance Computing

HTM: Hierarchical Temporal Memory

HQSOM: Hierarchical Quilted Self-Organizing Map

IDS: Intrusion Detection System

IP: Internet Protocol

IPS: Intrusion Prevention System

LevBag: Leveraging Bag

LSTM: Long Short-Term Memory

MPF: Memory-Prediction Framework  
ML: Machine Learning  
NAB: Numenta Anomaly Benchmark  
NIDS: Network Intrusion Detection System  
OBoost: Online Boosting  
RBC: Rule-Based Classifier  
RBM: Restricted Boltzmann Machine  
SDM: Sparse Distributed Memory  
SDR: Sparse Distributed Representation  
SOM: Self-Organizing Map  
SP: Spatial Pooler  
SQL: Structured Query Language  
SYN: Synchronization  
SVM: Support Vector Machine  
TCP: Transmission Control Protocol  
TP: Temporal Pooler  
UDP: User Datagram Protocol  
VDR: Virus Decryption Routine  
VFDT: Very Fast Decision Tree  
VM: Virtual Machine

## **Summary**

This chapter provided an overview of the problem of designing a sequence classifier for sequential data in data streams. As depicted earlier, much of the modern data streams contain concept drift, noise, and temporal dependencies. This research

leveraged the underlying theoretical framework of HTM, which utilizes memory that records time changing or "temporal" patterns. This research overcame the primary limitation of finding labeled malware that contained stalling code by creating an executable based on the Rombertik malware published by Lastline.com. This research has developed a new classifier based on the HTM architecture that can classify data within a data stream containing concept drift, noise, and temporal dependencies.

## **Chapter 2**

### **Review of the Literature**

#### **Overview**

This research focused primarily on exploring the appropriateness of HTM toward developing a sequence classifier capable of classifying data within data streams containing concept drift, noise, and temporal dependencies. Because of its ability to learn sequences over time, HTM will enable the sequence classifier algorithm to identify programs that use stalling code. The following areas of literature are essential to the proposed research:

- The current state of classifiers
- Current approaches to addressing concept drift
- Temporal patterns in real-world data streams
- Evasive Malware
- Malware evasion techniques
- Hierarchical Temporal Memory

#### **Current State of Classifiers**

The classification problem continues to receive a great deal of attention within the research community. However, most modern approaches to the classification problem incorrectly assume that data is stationary (Bifet et al., 2017). Current classification and data mining literature by Bifet et al., (2017), Mouchaweh (2016), and others have published that most real-world data streams consist of (possibly infinite) sequences of items, each having a timestamp and therefore a temporal order. Temporal dependencies

within data streams present two main algorithmic challenges, with the first challenge being the need to extract information in real-time, resulting in approximated solutions to use less time and memory. The second challenge is that the data often evolves, so classifier models must adapt when there are changes in the data.

Barcelo-Rico et al. (2016), developed a semi-supervised classification system for detecting Advanced Persistent Threats (APT) to large organizations. They employed a process that involved a human expert to first label the training data, followed by training their classifier on both labeled and unlabeled data. The authors trained the classifier using three computational intelligence methods: genetic programming, decision trees, and support vectors. Their solution achieved an 80% accuracy rating. However, their solution required human expert labeling and did not address concept drift, noise, or temporal dependencies.

### **Current Approaches for Addressing Concept Drift**

Ghameshi et al. (2019) developed an ensemble-based concept drift sensitive classifier named Evolutionary Adaptation to Concept Drifts (EACD). Ghomeshi et al. (2019) utilized random subspaces of features from a pool of features to create different classifiers for the ensemble. Each classifier consists of decision trees that were built at various times over the data stream. Ghomeshi et al. (2019) then employed replicator dynamics, a popular evolution and prestige-biased learning in game theory, along with a genetic algorithm that mutates existing classifier models to adapt to different concept drifts. Those classifiers with higher performance stay in the ensemble while with a lower performance are gradually removed. Ghomeshi et al. (2019) compared the EACD classifier with various other classifiers (i.e., ARF, DWM, LevBag, OAUE, and OSBoost classifiers) over four artificial and five real-world data sets. Ghomeshi et al. (2019)

reported that the EACD demonstrated performances comparable to that of the other classifiers in their experiments.

Gözüaçik et al. (2019) developed an unsupervised method for detecting concept drift, Discriminative Drift Detector (D3). D3 uses a sliding window to monitor changes in the feature space and requires an existing classifier (one without a drift adaptation mechanism). D3 uses a fixed sliding window of the latest data divided into two sets: the old and the new. An arbitrary classifier then distinguishes between the two groups based on the classifiers' overall performance on the two sets. Gözüaçik et al. (2019) reported that the D3 scored an accuracy of 86.69% with the Electricity data set, 75.59% Poker hand data set, and 85.29% with rotating hyperplane dataset. Gözüaçik et al. (2019) acknowledged that D3 could not detect real concept drifts that are caused by changes only in conditional distributions of  $P(y|X)$ . Furthermore, the authors recognize that D3 detects drifts unnecessarily when the change in the  $P(X)$  does not affect  $P(y|X)$ .

Madireddy et al. (2019), address concept drift using a model that identifies the location of events that lead to the concept drift through an online Bayesian changepoint detection method, followed by retraining the model on the data collected just before the drift. Madireddy et al. (2019) address the temporal learning problem through the use of long short-term-memory-based recurrent neural networks to build the model. Madireddy et al. (2019) reported that their concept-drift-aware models obtained 58.8% accuracy improvement. The authors acknowledge that their model requires a large shift in the drift to be effective.

Goel & Batra (2019) developed the Ensemble-based Online Diversified Drift detection (En-ODDD) algorithm for detecting concept drift. En-ODDD uses a trigger-based drift detection mechanism on a block-based ensemble framework. The ensemble

addresses concept drift by building experts based on their current performance on the most recent data chunks. The experts are then updated by pruning using a sliding-window based deterioration scheme based on how well it is currently performing. Goel & Batra (2019) augmented the usual incremental classifier training for the En-ODDD by incorporating an online bagging approach that then updates the ensemble experts. Goel & Batra state that online bagging increases the predictive accuracy of their ensemble because diverse learners are better suited at handling concept drift. Goel & Batra (2019) experiments consisted of nine other block-based classifiers (i.e., DDD, OzaBag, AUE2, ACE, DWM, WMA, LevBag, ARF, NSE). Goel & Batra's (2019) experiments utilized cost measures consisting of prediction accuracy, model cost, training time, and testing time with 12 artificial and three real-world datasets and report that En-ODDD outperformed all of the classifiers in their experiments.

Zhang & Chen (2019) developed the Weighted classification and Update algorithm for data streams based on Concept Drift Detection (WUDCDD) classifier. WUDCDD uses emerging patterns to build and update the base classifier based on computing a performance value that combined the Mahalanobis and  $\mu$  detection standards for determining error rates. The general process is to (a) build several integration classifiers (b) determine if concept drift has occurred and (c) update the classifiers based on the classification error rate derived from the Mahalanobis and  $\mu$  detection methods. Zhang & Chen (2019) solution is a form of the basic sliding window algorithm popular in many classifier solutions.

Pesaranghader et al. (2018) employ a pool-based classifier solution (TORNADO) for classifying data within data streams that experience concept drift. Their framework implements a reservoir of diverse classifiers that operate in parallel paired together with a



variety of drift detection algorithms. At any point in time, they select the classifier-drift detection pair, which currently yields the best performance. When determining the best performing classifier/drift-detection pair, Pesaranhader et al. (2018) take into consideration memory usage, runtime, drift detection delay, along with the number of false positives and false negatives. Additionally, Pesaranhader et al. (2018) developed two drift detection algorithms (e.g., FHDDMS, and FHDDMS<sub>add</sub>). The FHDDMS algorithm creates a stack of sliding windows of different sizes. The windows monitor the streams using bitmaps and signal an alarm for concept drift based on threshold values. Pesaranhader et al. (2018) propose that the detection of concept drift occurs faster and is more accurate occurs when using sliding windows of various sizes. The FHDDMS<sub>add</sub> drift detection algorithm is a variant of FHDDMS that employs data summaries, instead of bitwise operations.

Yang et al. (2018) developed an ensemble Extreme Learning Machine (ELM) to address both gradual and abrupt concept drifts (CELM). CELM takes advantage of the speed that ELMs typically provide and applies Locally Linear Embedding (LLE) to reduce the dimensions of data blocks. Learning is achieved via an online sequential learning mechanism that updates the classifiers when the change of data stream is small, only retraining the entire model when necessary. CELM then categorizes the current data stream into one of three types: stable, warning, and concept drift, with the difference being the amount of the current error percentage that the classifier(s) are experiencing. Yang et al. (2018) report that CELM can detect both gradual and abrupt concept drifts by using their online sequential learning and concept drift detection mechanisms. However, Yang et al. (2018) also acknowledge that CELM still has some scalability problems with the number of hidden nodes and their learning algorithm.

Sun et al. (2016) designed a pool-based classifier focused on handling recurrent concept drift. To aid in the detection of recurrent concept, Sun et al. (2016) develop the Distribution-Based Detection Method (DBDM), which detects changes by comparing the distribution of data in different time windows based on the Bernstein inequality which associates the expected value with variance. Additionally, Sun et al. (2016) developed an algorithm for dealing with previously seen concept drifts (recurring) named the Recurrent Detection and Prediction (RDP). Each time a concept drift is detected, RDP first looks to see if the current concept drift matches any contained in the current graph model; if not, the new concept drift is added to the graph model.

Zhang et al. (2016) developed a concept drift resilient classifier based on the calculation of correlation coefficients and information entropy. The correlation coefficient, also known as the Pearson correlation coefficient, describes the relationship between the two equal interval variables; information entropy is then used as a tool for fitting their classification models. The generalized procedure is to (a) process the data stream in blocks, (b) determine if concept drift has occurred using the correlation coefficients of the current block and comparing it to that of previous blocks which provides an entropy value, and (c) after updating the model, it is saved in a classifier pool to describe previous concepts for use in detecting recurring concept drifts.

Dehghan et al. (2016) also ensemble-based classifier similar to that of Zhang et al. (2016). Dehghan et al. (2016) employ methods that explicitly detect changes and adapts the learner to cope with the new concept. The detection of concept drifts is done by processing samples one by one and monitoring the error of the ensemble classification method. Dehghan et al. (2016) then count the number of samples with the possibility of drift and measuring the distances between these samples. If the classifier detects concept

drift, it will then be trained on the new concept and added to the pool of existing classifiers.

It is noteworthy to mention that none of the research in this section addressed temporal dependencies. While managing concept drift is an essential aspect in the design of a classifier for data streams, the next section identifies the importance of also addressing temporal dependencies.

### **Temporal Patterns in Data Streams**

Zhong et al. (2019) developed a deep learning-based classification framework for remotely sensed time series for California economic crops. Zhong et al. (2019) classified summer crops using Landsat Enhanced Vegetation Index (EVI) time series using two deep learning models, Long ShortTerm Memory (LSTM), and one-dimensional convolutional (Conv1D) layers. The LSTM model remembers values over arbitrary time intervals, either long or short. Zhong et al. (2019) report that the LSTM improved the efficiency of depicting temporal patterns at various frequencies, which was a desirable feature in the analysis of crop growing cycles that often consist of varying lengths. The Conv1D model employed one-dimensional filters to capture the temporal pattern or shape of the input series. Zhong et al. (2019) reported that the optimized Conv1D model had the highest test set accuracy of 85.54% when compared to other deep learning and non-deep learning models and the LSTM model performing the worst with 82.41% accuracy. Zhong et al. (2019) concluded that the LSTM was not an appropriate model for their classification experiment

Lange et al. (2019) describe a proof-of-concept for classifying news articles using features defined by extracting and normalizing temporal expressions. Lange et al. (2019) propose using temporal expressions and their characteristics for feature selection used in

their classification approach. Lange et al. (2019) use WEKA, an open source machine learning software, to train k-Nearest-Neighbours and Decision Trees on the features previously selected. Using datasets consisting of both English and German newspapers, Lange et al. (2019) reported that their experiments resulted in 69.3% accuracy for a decision tree and 68.2% accuracy for a 9-NN classifier. These results were achieved using the generalized normalized values extended with their temporal relation to the publication date.

Amine et al. (2019) developed an approach for discovering temporal information within raw data obtained from camera sensors. Their algorithm, Complex Temporal Dependencies (CTD)-Miner, searched for temporal dependencies between state streams by transforming raw sensor data sequences into a symbolic time-interval series representation known as Temporal Abstraction (TA). While the authors' work demonstrated that it is possible to integrate video analysis methods into a data analysis process, they did not address concept drift nor the classification problem.

Chiba et al. (2018) developed the DOMAINPROFILER for discovering malicious domain names that are likely to be used by malware authors in the future. Chiba et al. (2018) research focused on exploiting temporal variation patterns (TVPs) that exist within domain names. Chiba et al. (2018) define TVPs as the time-series behavior of each domain name within various domain name lists, and they observed that both legitimate and malicious domain names vary dramatically in domain name lists over time. The DOMAINPROFILER is comprised of a monitoring and profiling module. The monitoring module collects three types of information for use in profiling module: (i) domain name lists, (ii) historical DNS logs which contain time series collections of the mappings between domain names and IP addresses, and (iii) the ground truth used to

label the training dataset. The profiler module consists of three steps that utilize the information collected from the monitoring module:

1. Identify the TVPs for each input target domain.
2. Append DNS-based features to the output of step 1. This consist of target domain names with identified TVPs.
3. Apply a Random Forest machine learning algorithm for detecting/predicting possible malicious domain names.

Chiba et al. (2018) report that DOMAINPROFILER predicted malicious domain names 220 days beforehand with a true positive rate of 0.985.

One of the more recent research efforts with finding temporal dependencies in data streams was conducted by (Duong et al., 2018). Their work focused on spotting concept drift by finding change points within the data stream using linear higher-order Markov processes. The authors developed a k-order Candidate Change Point (CCP) model that exploits the temporal dependencies between data within the stream. Their model employs a sliding window approach to calculate the probability of finding change points using the time-based dependency information or factors between different observed data points in a stream. Duong et al. (2018) experiments proved to be extremely fast and accurate over large data streams. However, their research design was for detecting temporal dependencies and not as a classifier.

### **Evasive Malware**

The first virus software attack against a computer network occurred in 1971 with the Creeper Virus written by Bob Thomas at BBN Technologies (Rajesh et al., 2015; Barría and Cubillos 2016). This attack was an experimental program that self-replicated and infected DEC PDP-10 computers that used the TENEX Operating System. The

Reaper, the first antivirus software, was then developed to delete this virus. Now, more than 40 years later, the sophistication of both Intrusion Detection Systems (IDS) and malware has significantly increased with malware authors employing an ever-increasing number of evasion tactics aimed at avoiding detection of their malware. A technological war has ensued since this time between the authors of malware and the developers of intrusion detection applications (Rajesh et al. 2015; Jadhav et al. 2016).

As reported by McAfee Labs (2017), the first known virus that attempted to defend itself from anti-malware was the MS-DOS virus Cascade. This virus encrypted part of its code, thereby making its contents unreadable by security analysts. Figure 1 depicts the first large-scale use of obfuscation by evasive malware was the PowerShell virus, which utilized Windows commands to hide.

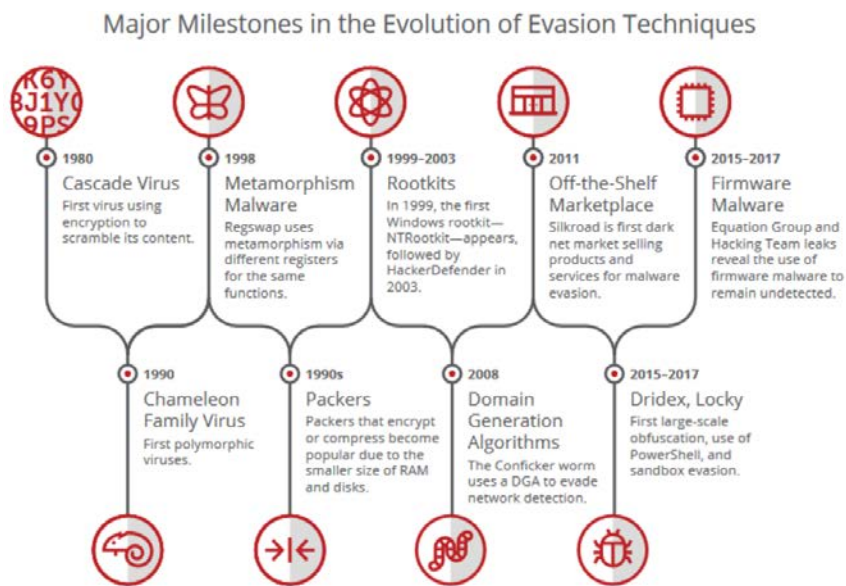


Figure 1. Major Milestones in the Evolution of Evasive Techniques. McAfee Labs Threats Report (2017), p. 11.

Malware detection solutions fall primarily into two broad categories: knowledge-based (inclusive of signature-based) and anomaly-based. Signature-based detection relies

on an existing signature database to detect known malware infections, and anomaly-based detection identifies abnormal behaviors in the data or system. According to Cannady (2000), anomaly detection comes with a cost of high false-positive rates because any deviation from the norm results in an alert to the administrator. The primary advantage of anomaly-based intrusion detection is the capability to detect new or unknown attacks since the new malware (whose signature is not available) would also generate abnormal behaviors.

Yaacob et al. (2010) pursued an approach for detecting network attacks based on a time series model. They presented a model named Autoregressive Integrated Moving Average (ARIMA) to predict regular network traffic that triggered an early warning if anomalous network traffic behavior occurred. ARIMA detects abnormal activity by employing a probability model that utilizes statistical functions that reside within the International Mathematics and Statistics Library (IMSL) C Numerical Library. Yaacob et al. (2010) claimed that by comparing forecasted network traffic to real-time network traffic that it is possible to raise an alert if the specific difference exceeds a preset threshold; in their implementation, they set a limit of 15% for the threshold. Yaacob et al. (2010) focused their solution on User Datagram Protocol (UDP) flooding and Transmission Control Protocol (TCP) synchronization (SYN) attacks. UDP flooding was identified by measuring the volume of incoming and outgoing traffic and defining the model of normality. ARIMA detected TCP SYN flooding by predicting the correct ratio of SYN packets to the number of completed TCP handshaking sessions. Yaacob et al. (2010) showed that ARIMA successfully identifies DoS (Denial of Service) attacks as abnormal data. However, they acknowledged that ARIMA does not perform well in low volume networks consisting of less than 1 Megabytes of traffic per second.

Network attacks that distribute themselves over time are difficult to detect. Cannady (2013) addressed this problem by taking an approach that recognizes temporally distributed attacks based on a modified Hierarchical Quilted Self-Organizing Map (HQSOM). HQSOMs are an extension of the original Self-Organizing Map (SOM) algorithm written by Kohonen (2001). Cannady's (2013) primary motivation behind using the HQSOM was to implement an algorithm that realized the concepts described in the Memory-Prediction Framework (MPF). MPFs include feedback loops that facilitate prediction, similar to the idea that there are structures in the brain that are responsible for processing a variety of different signals and share many structural similarities.

Cannady (2013) utilized a "leaky integrator," which combined an adaptive learning parameter with variable spatial and temporal clustering to associate the components of the attack. During the evaluation of the prototype, Cannady (2013) injected the attack patterns for three network attacks (e.g., Internet Protocol (IP) Spoofing, Low-rate DoS, and Teardrop) into the network data stream at three distinct periods of temporal distribution (e.g., 200, 500, and 800 milliseconds). Cannady (2013) then evaluated the HQSOMs ability to identify attack patterns at different levels of temporal distribution. Cannady (2013) reported that the accuracy of the HQSOM in detecting attack patterns was highest when the decay rate was at the lowest setting and lowest when the decay rate was at the highest setting. Cannady (2013) also reported that by enabling the leaky nodes to decay more slowly, the indicators of the temporally distributed attack patterns could be more easily detected. Cannady's (2013) work demonstrated the ability of HQSOM to identify temporally distributed network attacks by utilizing leaking nodes effectively.



Jadhav et al. (2016) state that there does not exist a signature-based solution that is not susceptible to trivial obfuscation techniques by evasive malware, and that can accurately adapt its signature database. Due to this inherent weakness with signature-based solutions, anti-malware vendors are not able to keep pace with malware authors who utilize evasion techniques. Vendors struggle to generate and implement innovations capable of detecting altered or new forms of malware, thus creating a need for constant patching, which equates to an endless cycle of adaptation-patching between the malware author and anti-malware vendor. Anti-malware vendors do not currently have a means for breaking this adapt/patch cycle as the current signature-based solution paradigm is susceptible to malware evasion techniques and requires constant updating, often after the damage is already done (Marpaung et al., 2012). Similar research by Singh et al. (2012) concurs that Machine Learning (ML) methods are not effective against malware with characteristics that change over time, because they require the ML model to retrain continually. Research by Canfora et al. (2015) supported Singh et al. (2012) findings in that they showed that obfuscation methods that perform trivial code transformations thwarted current signature-based detection solutions. Also, Canfora et al. (2015) concluded that signature-based techniques for identifying malware are susceptible to a variety of malware evasion techniques, thus locking anti-malware developers and malware developers into a never-ending cycle of adaptation-patching.

Even non-signature-based solutions for intrusion detection have their drawbacks, as Alom et al. (2015) reported that the primary disadvantage of anomaly-based intrusion detection is that it is more prone to generating false positives. Alom et al. (2015) explored an approach that applies a Deep Belief Neural (DBN) network to the intrusion detection problem that stacks a Restricted Boltzmann Machine (RBM) and a generative stochastic

Artificial Neural Network (ANN) that learns a probability distribution over its set of inputs. Alom et al. (2015) built upon work previously done by Hinton and Salakhutdinov (2006), which showed that RBMs can be stacked and trained greedily to form what is now called DBNs. Hinton and Salakhutdinov explained that DBNs are graphical models that learn to extract a deep hierarchical representation of the training data. Alom et al. (2015) reported that the resulting DBN network identifies any unknown attack in a dataset supplied to it with an approximately 97.5% success rate.

Moh et al. (2016) employed a hybridized strategy to address attacks that attempt to evade detection by making small changes to their signatures. Moh et al. (2016) combined the real-time efficiency of signature-based pattern matching that doesn't require training with the ability to detect new attack patterns that supervised machine learning methods provide. Moh et al. (2016) focused on Structured Query Language (SQL)-injection attacks and combined the two strategies by implementing a multi-stage log analysis architecture. The resulting prototype combined traditional Bayes Net algorithms with Kibana, a commercial application from Amazon Web Services (AWS). Bayes Net provided machine learning characteristics, and Kibana provided pattern matching characteristics. The architecture of the prototype consisted of three main parts: Log Generation, Data Preprocessing, and Detection. Moh et al. (2016) alternated the strategy order of the two-stage system as well to determine if this made any difference in the accuracy of their approach. Moh et al. (2016) reported that their experimental results showed that the hybridized two-stage system detected significantly more SQL injections than either a standalone or a single-stage system.

Evasive malware is also commonly grouped into other descriptive categories, such as the ones listed in the McAfee Labs Threats Report (2017), which categorized

evasion techniques into three broad categories (Anti-security, Anti-sandbox, and Anti-analyst). Regardless of the numerous malware categories that are prevalent in the current literature, the fact remains that evasive malware has proven effective against modern detection techniques employed by IDS and Intrusion Prevention Systems (IPS).

### **Malware Evasion Techniques**

Rouse (2010) reported that polymorphic malware commonly changes itself by employing a virus decryption routine (VDR) to alter its code using any variety of techniques. The resulting Encrypted Virus program Body (EVB) deploys; however, Rouse (2010) comments that the EVB remains the same after each iteration through the VDR, which makes this type of malware a little easier to identify.

Later research by Rastogi et al. (2014) supports that from Rouse (2010) that once a computer is infected, metamorphic malware constantly rewrites itself so that each succeeding version of the code looks different than the previous one but still has the same behavior. Signature-based Network Intrusion Detection Systems (NIDS) have a difficult time recognizing that the various versions of the malware are the same program. The longer the malware stays in a computer, the more versions it produces, making it increasingly difficult for the NIDS to detect, quarantine, and disinfect (Canfora et al. 2015; Rastogi et al. 2013; & Rouse 2010).

Canfora et al. (2015) findings also support that of Rastogi et al. (2014) and Rouse (2010) that modern NIDS strategies can not keep pace with evolving malware. Canfora et al. (2015) compared 57 anti-malware solutions against a dataset of 5560 malware attacks. The authors developed a transformation engine and subjected the dataset of malware attacks to it. Their transformation engine performed the following transformations on each attack:

1. Disassembled the original program, reorganizing its contents, and then reassembled the program.
2. Repacked the code by embedding a unique developer signature key.
3. Changed the package name, which altered the package and class names.
4. Renamed the identifiers through a random string generator
5. Encoded the strings and arrays within the code using a cipher.
6. Call indirection that changed the call graph of the application.
7. Code reordering that modified the instruction order of the code by inserting simple *goto* instructions without losing the original runtime execution trace.
8. Embedded junk code that did not affect the overall functionality of the code (i.e., no-op instructions).

Canfora et al. (2015) reported that a broad set of antimalware tools did not detect the transformed malware even when, before applying the transformations, they did.

Sosha et al. (2012) examined the use of constructing malware signatures based on their execution profiles. Sosha et al. (2012) extracted the execution profiles from kernel data structure objects as opposed to the traditional signature generation method that relies on byte sequence matching. The authors posited that kernel objects are an equal representation of code executed in the operating system kernel. Thus characteristics of kernel objects' can provide the basis for deriving evasion-resistant malware signatures. Sosha et al. (2012) presented a prototype signature generation tool (SigGENE) which utilized malware profiles built using the profiling process developed in Windows 7 SP1. The authors reported that, out of 63 test samples, their prototype detected 100% of the

evasive malware with 0 false positives. They also acknowledged the following deficiencies in their approach:

- Computationally expensive in the profiling stage.
- Ineffective against evasive malware that uses behavior monitoring strategies to avoid detection.
- They are limited in scope to only kernel Objects. Authors recommend expanding the approach to include file objects.

Marpaung et al. (2012) also surveyed malware evasion techniques employed by malware developers to evade detection by NIDS. They examined a variety of evasion techniques such as obfuscation, fragmentation and session splicing, application-specific violations, protocol violations, inserting traffic at IDS, and reuse attack evasion methods. Marpaung et al. (2012) asserted that sandboxing is the most effective countermeasure against evasive malware. They explained that sandboxing is effective against obfuscation techniques such as polymorphic code and encrypted sessions because it separates untrusted programs, users, and websites into a limited virtual environment with tightly controlled resources. Marpaung et al. (2012) concluded that simple string matching signature-based detection methods are no longer adequate and that evasion techniques are constantly evolving at a faster pace than detection strategies for detecting them. However, Kruegal (2013) contradicts Marpaung et al. (2012), in that malware authors can exploit several vulnerabilities of sandboxing through the use of stalled code in which the malware performs some useless computation that gives the appearance of normal activity.

Lim and Nicsen (2015) surveyed the most frequent types of malware found in network traffic and found that approximately 86% of the malware was evasive. Their

findings showed that nearly 75% of the evasive malware used a runtime compression algorithm (packed); 86% possessed an anti-debugging capability, and 1% employed anti-Virtual Machine (VM) techniques. Lim and Nicsen (2015) presented a model for the detection of evasive malware (MAL-Eve), which utilized several of the most popular evasion techniques used by malware authors: packing, anti-debugging, and anti-virtualization. Lim and Nicsen (2015) demonstrated that their model was capable of detecting various evasive malware with an accuracy of 98.16% and a false positive rate of 1.45% with an average processing time of 3.22 seconds for file sizes below 100 Kbytes. Lim and Nicsen (2015) acknowledged that the primary drawback of their model is that it is static, requiring resampling and retraining for any modification to existing evasive malware or new malware altogether. They posited that using a combination of static and behavior features might improve their models' detection capability.

Instead of focusing on malware signatures, Sosha et al. (2015) studied how evasive malware can thwart both static signature-based and dynamically based methods of detection. Sosha et al. (2015) successfully demonstrated that malware could overwhelm static IDS solutions with a large volume of signatures for a single attack. They also showed that malware could thwart sandboxing solutions by analyzing the machine environment. To address these evasion methods, Sosha et al. (2015) presented an automaton model for polymorphic and metamorphic detection that combined a mixture of static and dynamic malware detection methods called "segmented sandboxing." The first step in their approach was to determine if the potentially malicious code is a direct or polymorphic match to any seminal malware byte string in their malware signature database. Sosha et al. (2015) then utilized a controlled environment to

compare the effects that the malicious code had on system behavior and looked for similarities with other known malicious behavior patterns. The authors reported detection rates ranging from 82% to 100% and false positive and false negative rates, typically under 4%.

Similar to Sosha et al. (2015), Barria and Cubillos (2016) also studied the methods, techniques, procedures, and tools that malicious code authors use to update their malware against new detection measures utilized by IPS/IDS vendors. Barria and Cubillos (2016) found that malware authors used one or more of the following techniques to obfuscate their code, encryption, polymorphism, or metamorphism. The authors also found that encryption techniques require malware authors to update their code manually and are extremely difficult to maintain. In contrast, those that are automated (polymorphic, metamorphic) are relatively easy and becoming more popular with malware authors. Barria and Cubillos (2016) concluded that the development of a classification system that can classify malware into one or more of these groups is critical in the design of any future IPS/IDS. Similar research by Bushan et al. (2016) showed that malware authors, realizing the effectiveness of encryption in avoiding detection, take advantage of Self-Extracting Archive (SFX) to bypass antivirus software. SFX files contain within itself the software needed to extract the encrypted file(s) that it contains.

Additionally, Bushan et al. (2016) reported that if those files are password protected that the antivirus software is unable to examine the contents and thus are rendered ineffective. Bushan et al. (2016) also reported that malware authors sometimes employ a silent SFX technique, where an infected archive can self-extract and self-execute the malware contained within it without triggering any antivirus software or

alerting the user. Bushan et al. (2016) concluded that most current antivirus software is not capable of detecting malware packed using the silent-SFX techniques and that SFX is an advantageous method in the delivery of malicious code.

Jadhav et al. (2016) also examined and categorized evasion techniques which they found in current evasive malware. Jadhav et al. (2016) grouped the evasive malware into four primary evasion categories (Environment Awareness, Obfuscating internal data, Timing based evasion, and Confusing automated tools). Additionally, they identified the strategy by malware designers to searching online databases to see if their malware signatures' were present and, if found, would utilize custom encryption routines to obfuscate their malware so that it hardly resembled the original code signature. Jadhav et al. (2016) concluded that polymorphic and metamorphic coding effectively defeats signature-based IDS' by altering malware machine code sequences, thereby making the signatures useless.

Choliy and Gao (2017) studied evasion techniques that malware authors used to evade detection. The authors' research centered around the use of Function Call Graphs (FCG) extracted from Android App Packages (APK) in the discovery of malware on devices that utilize the Android mobile operating system. Choliy and Gao (2017) focused primarily on a method known as ACTS (App topologiCal signature through graphleT Sampling), which is commonly used by antivirus software developers for identifying malware. ACTS works by extracting graphlet statistics from an FCG and differentiates between benign app samples and malware. Choliy and Gao (2017) observed that malware authors were able to circumvent ACTS by creating function calls to manipulate the FCG of their software to resemble a legitimate app.



Choliy and Gao (2017) concluded that by adding edges or nodes and edges, that they were able to manipulate a malicious FCG to sufficiently change its graphlet frequency distribution vector enough to fool ACTS into classifying it as legitimate software.

### **Hierarchical Temporal Memory**

Hawkins and Blakeslee (2004) promoted that there are many things humans find easy to do that are beyond that of modern-day computers. Tasks such as visual pattern recognition, understanding spoken language, recognizing a musical piece based solely on a few notes, are easy for humans. Despite nearly 50 years of research, we have few viable algorithms for achieving human-like performance on a computer (Numenta, 2011) for these types of recognition problems. In humans, the neocortex performs these recognition tasks. HTM is a biologically-constrained theory of intelligence based on neuroscience and the physiology and interaction of pyramidal neurons in the neocortex of the mammalian brain (Hawkins and Blakeslee 2004).

#### ***Early realization of HTMs in malware detection***

Bonhoff (2007) researched the application of HTM and CLA theories (Hawkins and Blakeslee; 2004) as a solution for intrusion detection. Bonhoff (2007) implemented an early beta version of the HTM software under extremely prohibitive research licensing agreement referred to as "Zeta1". Unlike later versions published by Numenta Inc, Zeta1 did not incorporate a feedback mechanism, which is fundamental to Hawkins' HTM theories. While Bonhoff (2007) highlighted flaws in the 2004 implementation of HTM theory, he did not find flaws in the HTM theory itself. Khangamwa (2010), conducted similar research to that of Bonhoff (2007) on the application of Hawkins et al. (2004) HTM and CLA theories as a solution for intrusion detection utilizing Numenta's NuPIC

version 1.6.1. Unlike the Zeta1 version implemented by Bonhoff (2007), this version included a feedback loop and more effective Spatial Pooling (SP) and Temporal Pooling (TP) algorithms. Khangamwa concluded that the NuPIC platform was suitable for solving the problem of intrusion detection through either an anomaly-based detection approach or a misuse detection based approach. However, Khangamwa (2010) reported that the anomaly-based detection approach provided a better solution than misuse detection.

### ***Sparse Distributed Memory***

Sparse Distributed Memory (SDM) began in 1974 as a paper written by Pentti Kanerva for a class on human memory given by Gordon Bower of Stanford's psychology department (Kanerva, 1990). Flynn et al. (1989) later formalized the definition of SDM as:

a generalized random-access memory (RAM) for long binary words (1000 bits or longer). The main attribute of the memory is sensitivity to similarity, meaning that a word can be read back not only by giving the original write address but also by giving one close to it as measured by the Hamming distance between addresses.

(p. 1)

### ***Sparse Distributed Representations***

Hawkins (2011), derived a data structure similar to the sparse distributed memory proposed by Kanerva (1990). Hawkins (2011a) speculated that despite neurons in the neocortex being highly interconnected, inhibitory neurons guaranteed that only a small percentage of the neurons are active at one time, implying that only a tiny percentage of active neurons represented information in the brain. This kind of encoding is called a sparse distributed representation (SDR). Hawkins (2011a) goes on to state that

"Sparse means that only a small percentage of neurons are active at one time. Distributed means that the activations of many neurons are required to represent something. A single active neuron conveys some meaning, but it must be interpreted within the context of a population of neurons to convey the full meaning." (p. 11)

### ***Formal Definitions and Notation***

The following formal definitions and notations come from Ahmad & Hawkins, 2015, p. 2.

**SDR:** Given a population of  $n$  neurons, their instantaneous activity is represented as an  $n$ -dimensional vector of binary components, e.g.,  $x = [b_0, \dots, b_{n-1}]$ .

Typically these vectors are highly sparse, i.e., a small percentage of the components are 1. We use  $w_x$  to denote the number of components in  $x$  that are 1.

**Overlap:** Similarity between two SDR encodings is determined using an overlap score. The overlap score is the number of bits that are ON in the same locations in both vectors. If  $x$  and  $y$  are two binary SDRs, then the overlap can be computed as the dot product as per equation (2). Notice the absence of typical distance metrics (i.e., Hamming, Euclidean) to quantify similarity. The overlap function derives some useful properties (i.e., union, intersection), which would not hold with these distance metrics (Hawkins et al., 2017).

**Matching:** A match between two SDRs is recognized if their overlap exceeds some threshold  $\theta$  as shown in equation (1) where typically  $\theta$  is set such that  $\theta \leq w_x$  and  $\theta \leq w_y$ . (p. 2)

### ***Example usage of SDR, Overlap, and Matching***

For two SDRs  $x$  and  $y$ , given equation (1), if  $x$  and  $y$  have the same cardinality  $w$ , an exact match can be determined by setting  $\theta = w$  (number of active bits chosen to be on). When designing an encoder, we must choose the number of active bits,  $w$ , to have in each representation (SDR). The number of active bits does not change regardless of the input. In this case, if  $\theta$  is less than  $w$ , the overlap score will indicate a mismatch.

Consider an example of two SDR vectors:

$$\mathbf{x} = [010000000000000000001000000000001100000000]$$

$$\mathbf{y} = [100000000000000000001000000000001100000000]$$

Both vectors have size  $n = 40$ ,  $s = 0.1$ , and  $w = 4$  (active bits). The overlap between  $x$  and  $y$  is 3; i.e., there are three ON bits in common positions of both vectors. Thus the two vectors match when the threshold is set at  $\theta = 3$ , but they are not an exact match. Note that a threshold larger than either vector's cardinality (i.e.,  $\theta > w$ ) implies a match is not possible (Hawkins et al., 2017).

It is important to note that  $w$  represents the number of active bits “on.” The encoder ensures that every SDR has  $w$  bits activated bits. Notice that this formula does not use a typical distance metric, such as Hamming or Euclidean, to quantify similarity. The absence of a distance metric is a significant difference between SDR's and SDM's. With this type of overlap, some useful union and difference properties are derived, which would not hold with Hamming or Euclidean distance metrics. Consider an example of two SDR vectors:

$$x = [0100\ 0000\ 0000\ 0000\ 0011\ 0000\ 0000\ 0001\ 1000\ 0000]$$

$$y = [1000\ 0000\ 0000\ 0000\ 0011\ 0000\ 0000\ 0001\ 1000\ 0000]$$

Both vectors have size  $n = 40$ , and  $w = 5$ . The overlap between  $x$  and  $y$  is 4, and thus the two vectors match when  $\theta = 4$ . Parameters typical of current HTM implementations are listed below.

HTM parameters:

$n = 1024$  to  $65,536$ , representing the length of an SDR vector.

$w = 10$  to  $40$ , representing the number of ON bits in an SDR vector.

$s = 0.05\%$  to  $2.0\%$ , representing the sparsity, where  $s = w/n$  (Ahmad & Hawkins, 2015).

Theta ( $\theta$ ) is the minimum number of overlapping bits from SDR  $x$  and some random SDR  $y$  before we consider SDR  $y$  to be a match of SDR  $x$ .

According to Ahmad and Hawkins (2015), it is not necessary to keep track of all  $w$  bits in the original SDR  $x$  when checking for overlap (Figure 2). SDRs provide the ability to compare against a subsampled version of a vector reliably. That is, recognizing a large distributed pattern by matching a small subset of the active bits within the larger pattern. As stated by Ahmad and Hawkins (2015), Let  $x$  be an SDR vector and let  $x'$  be a subsampled version of  $x$ , such that  $w_{x'} \leq w_x$ . The subsampled vector  $x'$  will always match  $x$ , as long as  $\theta \leq w_{x'}$ , but as you increase the subsampling, the chance of false-positive increases.

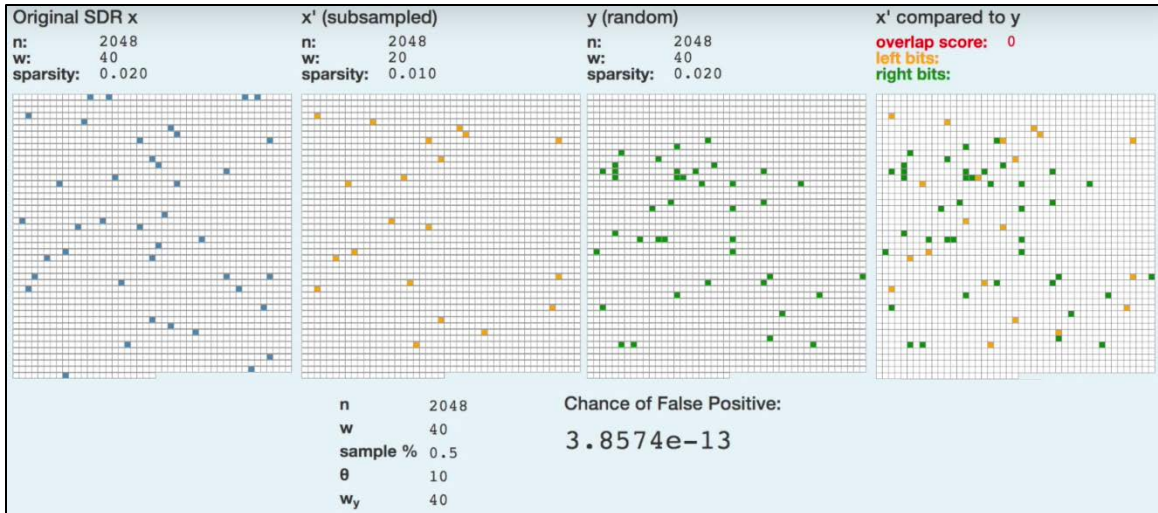


Figure 2. SDR Overlap set. Showing random SDR  $y$  not matching  $x'$ .

Evasive malware semantic similarity: Is defined as a similarity (overlap) score of an algorithms' behavior(s) that is  $\geq \theta$  to that of stalling code.

***General guidelines for encoding SDRs.***

According to Purdy (2015), four essential properties of SDRs must hold true when encoding data:

1. Semantically similar data should result in SDRs with overlapping active bits.
2. The same input should always produce the same SDR as output.
3. The output should have the same dimensionality (total number of bits) for all inputs.
4. The output should have similar sparsity for all inputs and have enough one-bits to handle noise and subsampling.

***Recent advancements in HTM technology.***

Numenta Inc. (2011) incorporated a feedback loop into their algorithms, which allowed Ahmad and Hawkins (2015) to further research aimed at deriving the mathematical properties of HTMs. Ahmad and Hawkins sought to take advantage of specific features that HTMs possess, such as bounds and scaling laws, performance

characteristics, and ideal parameters. Ahmad and Hawkins (2015) showed that SDRs could be used to perform robust classification despite the noise and random deletions by using the union property. They concluded that under the right conditions, SDRs enable a massive capacity to learn temporal sequences and form the basis for highly robust classification systems.

According to Cui, Surpur et al. (2016), Cui, Ahmad, et al. (2016), and Ahmad and Hawkins (2015), HTM sequence memory lends itself to continuous online learning as well as handling branching temporal sequences by maintaining multiple predictions until there is sufficient disambiguating evidence. These features are ideal in an online environment with streaming live data that may contain a lot of noise as well as ambiguity. Cui, Surpur, et al. (2016) analyzed the properties of HTM sequence memory and applied them to sequence learning and prediction problems that utilized streaming data. Cui, Surpur, et al. (2016) reported the following findings:

- HTM does not require the use of sliding windows for learning.
- HTMs learn from each data point using unsupervised Hebbian-like associative learning mechanisms.
- The SDRs used in HTM possess a very large coding capacity and allow simultaneous representations of multiple predictions with minimal collisions.
- HTM sequence memory achieved performance comparable to that of Long Short-Term Memory (LSTM) networks.
- As a strict one-pass algorithm with access to only the current input, it may take HTMs longer to learn sequences with very long-term dependencies.

Cui, Ahmad, et al. (2016), reported that HTM sequence memory achieved comparable prediction accuracy to four statistical and machine learning techniques:

ARIMA, a statistical method for time-series forecasting; extreme learning machine (ELM), a feedforward network with sequential online learning and two recurrent networks LSTM and Echo State Network (ESN).

These findings suggest that an HTM solution would apply to detect evasive malware. HTMs are not just limited to the detection of malware. Fu et al. (2015) used an HTM solution for the diagnosis of diseases in patients. In their work, the Fu et al. (2015) encoded patient medical data into SDR's, which included up to 49 different symptoms (i.e., sneezing, flu, tonsillitis, etc.). Their experimental results showed that they were able to predict diseases with an accuracy of 86.1%.

***HTM suitability as a classifier.***

Irmanova et al. (2018) conducted preliminary research using circuits based on HTMs for sequence learning of handwritten character images. In their simulation, they showed that HTM algorithms suitable for symbol order recognition and learning sequences from character images. Irmanova et al. (2018) intend to use the results of this preliminary research as a step towards using HTMs for solving sequence learning tasks (i.e., spell checking). A direct correlation can be made between recognizing words with recognizing stalling code in that they both refer to language, the former written human language, and the later being machine language.

Wang et al. (2018) conducted preliminary research that developed a distributed anomaly detection system using HTMs to enhance the security of a vehicular controller area network bus. The goal of their study was to detect attacks on a car's network in real-time as it was being driven. HTMs were used for detecting abnormal sequences on the Controller Area Network (CAN) bus of the vehicle. The data on CAN bus data consisted of a list of (ID, data payload) pairs indexed with timestamps. This work also



demonstrated the strong propensity for HTMs to be used as a classifier for identifying stalling code given the similarities between its streaming data and that of the running executables (e.g., Program Counter, Register Values).

According to Ahmad (personal communication, April 18, 2018), Numenta Vice President of Research, Numenta has not attempted to build a sequence classifier for malware yet but has an interest in doing so; however, Ahmad acknowledges the reason that this is difficult is that building SDRs that capture the behavior of malware over time is extremely tedious.

Lavin and Ahmad (2015) observed that existing benchmarks are designed for static datasets and are not suitable for evasive malware. They added that most academic research in intrusion detection involves an artificial separation of training and test sets; unfortunately, this artificiality does not correctly capture the characteristics of real-time streaming data. Lavin and Ahmad (2015) went on to develop an evaluation benchmark that they assert is suitable for the evaluation of real-time anomaly detection algorithms, the NAB (Numenta Anomaly Benchmark). A concern is that depending solely on the benchmark developed by Numenta, the developers of Hierarchical Temporal Memory (HTM) technology, may not be entirely objective. The amount of system resources that will be required to implement an HTM solution that will be robust enough for practical use as an Intrusion Detection System is an unknown (Khangamwa, 2010). Also, Cui et al. (2016) state that "Real-world sequence learning deals with noisy data sources where sensor noise, data transmission errors and inherent device limitations frequently result in inaccurate or missing data" (pp. 1531). Therefore, the HTM sequence classifier and any SDR must be capable of filtering out the noise.

## **Summary**

A tremendous amount of research in the development of classifiers for streaming data is currently underway. However, much of this research focuses exclusively on either concept drift, noise, or temporal dependencies in isolation. Similarly, a great deal of ML research is also underway in detecting anomalous activities within data streams indicative of malicious malware.

Thus, the design of an application that takes into account the effect of concept drift, noise, and temporal dependencies would help to fill a gap in modern classifier literature.

## Chapter 3

### Methodology

The goal of this research was to develop a sequence classifier that can classify data in a data stream containing concept drift, noise, and temporal dependencies. The approach this research utilized was an HTM framework with SDRs constructed for each of the respective data sets.

#### **Classification**

Bifet et al. (2017) formally define the classification problem as:

given a set of labeled instances or examples of the form  $(x,y)$ , where  $x = x_1, \dots, x_k$  is a vector of feature or attribute values, and  $y$  is one of  $n_c$  different classes, also regarded as the value of a discrete attribute called the class. The classifier building algorithm builds a classifier or model  $f$  such that  $y = f(x)$  is the predicted class value for any unlabeled example  $x$ . (p. 85)

#### ***Classifier Evaluation***

According to Bifet et al. (2017), the main challenge in evaluating classifiers is knowing when a classifier is outperforming another classifier only by chance, and when there is a statistical significance to that claim. As recommended by Bifet et al. (2017), the classifier evaluation framework for this research included the following parts:

**Error estimation method.** According to Bifet et al. (2017), the traditional method of splitting the dataset into disjoint training and test sets is computationally too expensive in a stream setting. Therefore this research implemented the prequential method for error estimation, which takes into account that more recent examples are more important than older ones. This evaluation method requires a sliding window.

**Cost measure of the process.** As recommended by Bifet et al. (2017), in addition to accuracy %, estimation for the combined cost of performing the learning and prediction process in terms of time and memory are added in the form of CPU Run-Time and RAM-Usage.

### **The Data Sets**

The data sets in this research consisted of artificial and real-world data streams, along with simulated and real-world malware. Both the manufactured and real-world data streams contained temporal and non-temporal data sets. According to (Bifet et al., 2017), two essential characteristics of any classification method in the stream setting are: (1) limitations exist in terms of memory and time and (2) classification models must be able to adapt to possible changes in the distribution of the incoming data. Therefore, in all experiments, the inspection of the data sets was restricted to a single pass for each classifier, and concept drifts inserted into each artificial data stream. The real-world data streams may or may not contain their own unique forms of concept drift unique to their application. The real-world data sets do contain temporal dependencies also based on their individual use.

### ***Artificial Data Streams***

The following four data stream generators provided by the python package scikit-multiflow were employed to simulate data streams containing abrupt, gradual, and recurrent concept drifts: the SEA generator (Street & Kim, 2001), the Hyperplane generator (Hulten et al., 2001), the Random Tree (RT) generator (Hulten & Domingos, 2002), and the LED generator (Breiman et al., 1984).

Using the research conducted by Ghomeshi et al., (2019) as a template, each generator created ten different stream variants (files) containing one million data sets

each. Additionally, each respective generator created an additional set of 10 variants containing 10% noise as well. Each generator introduced abrupt, gradual, and recurrent concept drifts into each variant via its respective default parameters. A different random seed parameter for each run ensured unique variants. For the Hyperplane generator, the parameters were: noise percentage, random seed, number of drifting attributes, and magnitude of changes (Appendix A). For the LED generator, different variants were built by tweaking the number of attributes that have concept drift and the random seed number (Appendix B). For the RT generator, the parameters were: random seed number, along with the number of features and classes (Appendix C). Finally, for the SEA generator, the parameters were: noise percentage, random seed, and classification parameters for each variant, which served to introduce noise and concept drift (Appendix D).

### ***Real-World Data Streams***

Three real-world data streams were selected for experiment II. following real-world data streams were selected:

**Electricity data set.** The electricity data set by Harries (1999) collected from the Australian New South Wales electricity market is a popular ML dataset that contains temporal dependencies. This data set reflects market prices that are not fixed but rather affected by demand and supply. The Electricity dataset contains 45,312 instances. Each instance contains eight attributes, and the target class specifies the change of the price (whether it goes up or down) according to its moving average over the last 24 hours (Appendix E).

**Airlines data set.** This data set consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008 (Appendix F).

This data set is a smaller subset consisting of 539,383 instances out of the original 120 million instances.

**Poker data set.** This data set from the UCI Machine Learning Repository consists of 1,000,000 instances and 11 attributes (Appendix G). The UCI repository describes the data set as follows:

Each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes. There is one Class attribute that describes the "Poker Hand." The order of cards is important, which is why there are 480 possible Royal Flush hands as compared to 4.

### **HTM Classifier**

Bifet et al. (2017) explain that *classification* seeks to predict which group a new instance may belong to and that two important characteristics of any classification method in the stream setting are (a) that limitations exist in terms of memory and time, and (b) that classification models must be able to adapt to possible changes in the distribution of the incoming data. Therefore, any design of the HTM classifier must possess these two characteristics to be suited to the stream setting.

### **HTM Algorithm**

This research implemented v2.0.22 of the Community Fork of the nupic.core C++ repository, with Python bindings, which can be found at <https://github.com/htm-community/htm.core>. This repository maintains an actively developed C++ core library and implements the theory as described by Hawkins et al. (2017).

## **HTM Classifier Encoders**

The nupic.core C++ repository comes with several default encoders as described by Purdy (2016). This research utilized the basic scalar, date, and category encoders along with creating custom encoders for the airlines and poker data sets built by combining the existing basic encoders (Appendix H).

### ***Scalar Encoder***

A scalar encoder encodes a numeric (floating point) value into an array of bits. A scalar encoder with a range from 0 to 100 with  $n=12$  and  $w=3$  will produce the following encodings:

- 1 becomes 111000000000
- 7 becomes 111000000000
- 15 becomes 011100000000
- 36 becomes 000111000000

The first thing to notice is that values that fall into the same bucket are represented identically as with 1 and 7. The second thing to notice is that for values that fall into separate buckets (i.e. 7 and 15), the closest buckets share the most overlapping bits. So 7 and 15 share two overlapping bits while 15 and 36 share only one bit and 7 and 36 do not share any bits.

### ***Date Encoder***

The date encoder encodes a date according to encoding parameters specified in its constructor. The input to a date encoder is a `datetime.datetime` object. The output is the concatenation of several sub-encodings, each of which encodes a different aspect of the date. Various optional parameters are available (i.e. season, dayOfWeek, weekend, holiday).

### ***Category Encoder***

The category encoder encodes a list of discrete categories (described by strings) that aren't related to each other. Except for the poker data set, the remaining real-world data sets contained binary '0' or '1' categories. In contrast, the poker data set contained a range of classes from '0' to '9' as described earlier.

### ***Construction of Additional Encoders***

Purdy (2015) stated that "the first step to designing an encoder is to determine each of the aspects of the data that you want to capture ... that the encoder should create representations that overlap for inputs that are similar in one or more of the characteristics chosen" (p. 2). For this research, specialized encoders were designed for the airline dataset that replaced airport names with SDRs of the latitude/longitude, altitude, and airport size (small, medium, large) for each airport in the dataset (Appendix H). The electricity and poker data sets utilized existing scalar and category encoders.

### **Experiments**

Three groups of experiments were conducted and referenced as experiments I, II, and III. Experiments I and II utilized artificial and real-world data streams, respectively. At the same time, Experiment III employed a simulated malware sample derived from code snippets of the Rombertik virus provided by Giron and Kolbitsch (2015) and an Assembly language sorting algorithm written by Sag (2012) for comparison.

Experiments I (artificial data streams) and II (real-world data streams) compared the HTM classifier against several state-of-the-art classifiers for non-stationary data stream classification (Ghomeshi et al., 2019). The list of classifiers included the Accuracy Weighted Ensemble (AWE) (Wang et al., 2003), Adaptive Random Forest (ARF) (Gomes & Enembreck, 2014), Dynamic Weighted Majority (DWM) (Kolter &



Maloof, 2007), Leveraging Bag (LevBag) (Bifet et al., 2010), Online Boosting (OBoost) (Wang & Pineau, 2016), and the Very Fast Decision Tree (VFDT) (Hulten et al., 2001).

Experiments I and II, implemented the classifier evaluation framework, including error estimation, performance evaluation measures, statistical significance, and cost measure recommended by Bifet et al. (2017). The classifier server was a stand-alone machine for experiments I and II (Figure 3) and are not networked to the malware analysis lab (Figure 5). Experiment III connected the classifier server to a malware analysis machine that provided two Virtual Machine Environment (VME) guest machines running on Windows 8.1 and Windows 10 Operating Systems, respectively (Figure 4).

### ***Experiment I Data Set Generation***

Expanding on the research of Ghomeshi et al. (2017), which did not include noise, this experiment generated two data sets consisting of 0% and 10% noise, respectively, for every artificial data stream using the previously mentioned generators. Additionally, to create a sufficient amount of data, each data set consisted of 10 different variants (files) containing 1,000,000 data points each, with each classifier tested on all variants. The three forms of concept drift (abrupt, gradual, and recurrent) were manually introduced into each variant in the instance numbers 200K, 400K, 600K, and 800K. The first five variants of each data set contained two abrupt concept drifts with a width (width of concept drift change) of one at the instance numbers 200K, 600K, and two recurrent concept drifts with the same width at instance numbers 400K and 800K. The second five variants of each data set contained two gradual concept drifts with a width of 10,000 at the instance numbers 200K, 400K, and 600K, and one recurrent concept drift with the same width are added at the instance number 800K. There were two different evaluation

runs for this experiment. The first run utilized the data sets containing 0% noise, and the second run utilized the data sets containing 10% noise.

**Experiment II Data Set**

As described earlier, experiment II utilized the popular ML data sets of electricity, airlines, and poker hand (Appendices E, F, and G). Using Ghomeshi et al., (2019) research as a guide, each experiment was repeated ten times over the same data stream. This research assumes that concept drift, noise, and temporal dependencies inherently exists within these three real-world data sets.

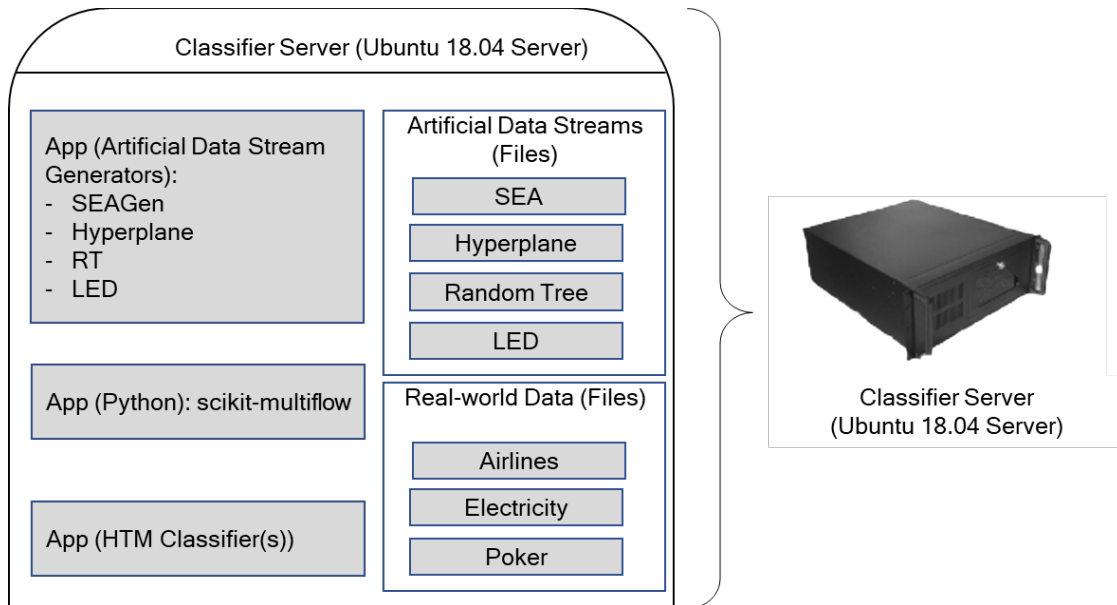


Figure 3. Classifier Server Applications and File Structure.

**Experiment III Predicting Stalling Code**

This research postulates that registers within a Von Neumann based machine accurately describe its machine state at any given time. This research also proposes that a running executable creates temporal dependencies between its machine states.

The purpose of experiment III was to assess the predictive performance of the HTM classifier against a known example of stalling code. This analysis was achieved by

providing the HTM classifier the EIP, EAX, EBX, ECX, EDX, and EFLAGS register values resulting from executing each Line of Code (LOC) during dynamic disassembly of the executable. The HTM classifier then made a 1-step prediction for each of the individual flag bit values in the EFLAGS register for the next machine state. A plot of the HTM classifiers' accuracy for each flag bit (Figure 4) after receiving each machine state was generated (see Chapter 4) for visual inspection.

Additionally, a 5-step prediction plot was generated for analysis to address Cui et al. (2016) claim that HTMs are capable of making multiple simultaneous predictions. Cui et al. (2016) go on to state that a good sequence learning algorithm should be able to make multiple predictions due to different temporal contexts creating multiple possible future outcomes. An example of this would be the American patriotic song "America," with lyrics written by Samuel Francis Smith having the same melody as that of the national anthem of the United Kingdom, "God Save the Queen."

Figure 4 shows a detailed diagram of the EFLAGS register. "The EFLAGS register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation. Some instructions test and manipulate individual processor flags." (Irvine, 1999, p.40)

The flags within the EFLAGS register divided into two groups (e.g., Control and Status flags). Irvine (1999) states, "The control flags control the CPU's operation (i.e., cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode)." (p. 40) The control flags are the Trap and Direction flags.

The status flags reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags.

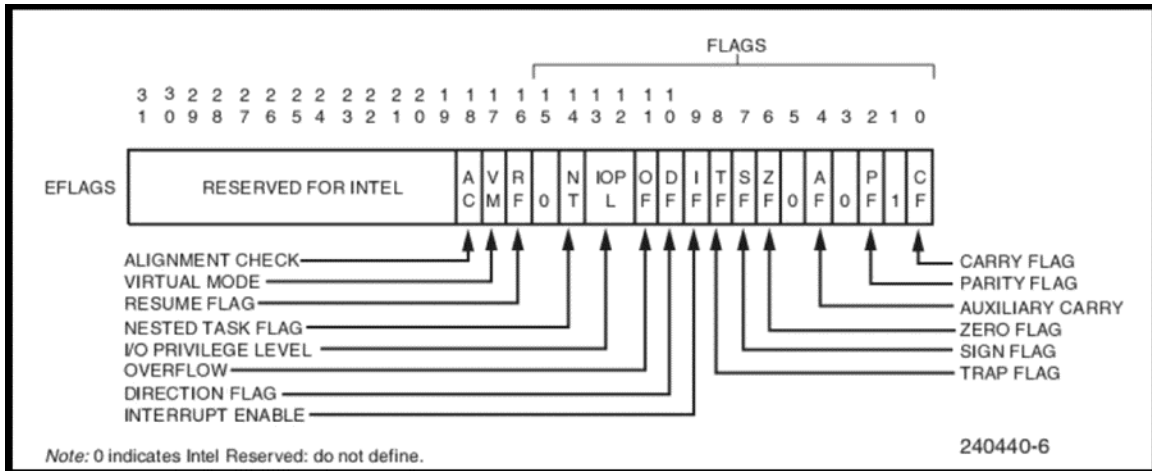


Figure 4. x86 EFLAGS diagram. Reprinted from Windows Malware Analysis Essentials (p. 70), by Marak (2015). Windows Malware Analysis Essentials. Packt Publishing. Kindle Edition.

An exhaustive search for labeled malware executables containing stalling code was unsuccessfully conducted. While the Virusshare.com malware repository contained nearly 34 million examples of malware executables, none of these were explicitly labeled as containing stalling code. Manually reverse engineering these files was impractical, even with the reverse engineering tools provided by IDAPro. In this case, snippets of a variant of the Rombertik virus provided by *lastline* authors Giron and Kolbitsch (2015) served to create a simulated malware example for analysis (Appendix J). It was observed that the Rombertik virus employs stalling logic that contains two very large looping constructs that are each  $O(n)$  complexity at labels `loc_4EEFD2` and `loc_4EEFE2`, comprising 30 million and 7.6 billion iterations respectively (Appendix J). Giron and Kolbitsch (2015) identify these sections of the Rombertik virus as stalling code. An

assembly language sorting algorithm of  $O(n^2)$  complexity written by Sag (2012) and modified to suit the requirements of this research was implemented for comparison (Appendix K).

A malware analysis environment was created to prevent live malware undergoing reverse engineering from infecting the classifier server. The host machine ran VM Workstation Pro 15.0 and created two VME Guest for malware analysis and running the IDAPro python scripts that performed dynamic disassembly of the executable and fed the resulting machine states to a local HTM Input Stream Server which forwarded the machine states to the classifier server via a socket as depicted in Figures 4 and 5.

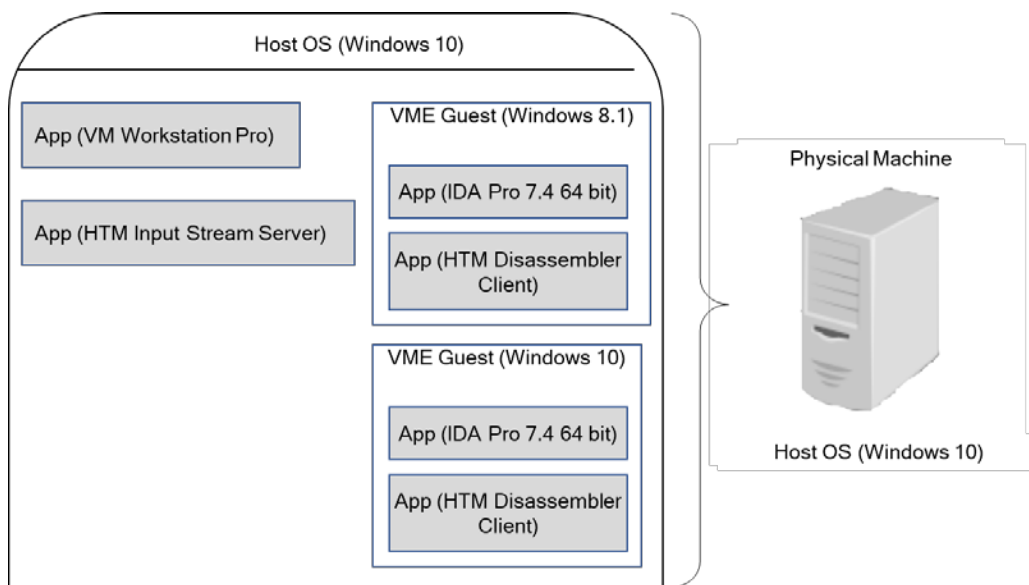


Figure 5. Malware Analysis Machine

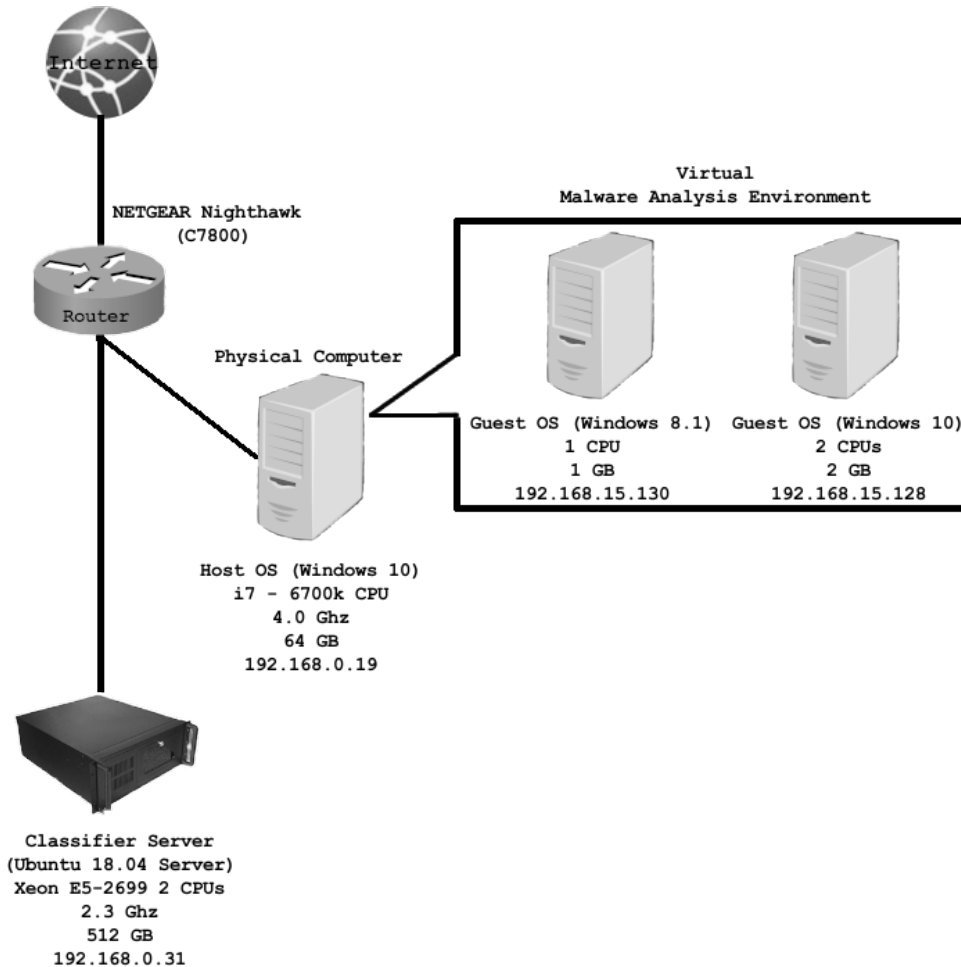


Figure 6. Malware Analysis Network

### Statistical Significance Validation

As was mentioned earlier by Bifet et al. (2017), research has shown that in most cases, streaming data is not independently and identically distributed (IID). Therefore, a non-parametric test (sometimes called a distribution-free test) was implemented. While Bifet et al., (2017) recommended the McNemar (1947) statistic for determining the statistical significance of differences between classifiers, it is only designed to compare two classifiers against one another. Like Ghomeshi et al., (2019), this research utilized the Friedman omnibus test for this comparison as it is better suited for comparing more than two classifiers. The Friedman test ranks the classifiers separately, with the best

performing algorithm getting the rank of 1, the second-best rank of 2, and so on. The formal definition of the Friedman test comes from Demsar (2006) as described below.

Let  $r_i^j$  be the rank of the  $j$ -th of  $k$  algorithms on the  $i$ -th of  $n$  data sets. The Friedman test compares the average ranks of algorithms,  $R_j = \frac{1}{N} \sum_i r_i^j$ . Under the null hypothesis, which states that all the algorithms are equivalent and so their ranks  $R_j$  should be equal, the Friedman statistic

$$\chi_F^2 = \frac{12N}{k(k+1)} \left[ \sum_j R_j^2 - \frac{k(k+1)^2}{4} \right] \quad (3)$$

is distributed according to  $\chi_F^2$  with  $k - 1$  degrees of freedom, when  $N$  and  $k$  are big enough (as a rule of a thumb,  $N > 10$  and  $k > 5$ ). (p. 11)

The Nemenyi posthoc test was implemented to find the groups of classifiers that differed after the Friedman test rejected the null hypothesis that the performance of the classifiers was the same.

### **Computing Resources Used**

This research was conducted using the classifier server hardware listed in Table 1 running on an Ubuntu 18.04 Server Operating System and a Windows 10 PC with a 4.00 GHz Intel Core i7-6700K Central Processing Unit, 64 Gigabytes of Random Access Memory, and a 2 Terabyte hard drive using the malware analysis software in Table 2.

**Table 1**

*Classifier Server Hardware*

---

<u>Item Description</u>
StarTech.com 12U AV Rack Cabinet
TYAN S7076GM2NR Tyan Motherboard
CORSAIR AX1600i, 1600 Watt, Digital Power Supply
Asus 24x DVD-RW Serial-ATA Internal OEM Optical Drive
2 x Intel Xeon E5-2697 v3 2.6 GHz LGA 2011-3 Server Processors
SAMSUNG 970 EVO M.2 2280 2TB PCIe Internal Solid State Drive
Micron 5200 5210 Ion 3.84 TB Solid State Drive - SATA 600-2.5" Drive
Crucial with 512GB (16 x 32GB) DDR4 PC4-21300 2666MHz RDIMM
APC UPS 1500VA Smart-UPS
LG 27UD88-W 27-inch 4K Ultra HD IPS LED-lit Computer Monitor
Rybozen 7-Port USB 3.0 Hub
TRIPP LITE 1U Rackmount Keyboard with KVM Cable Kit
2 x AC Infinity CLOUDPLATE T1, Rack Mount Fan Panel 1U
AC Infinity CONTROLLER 12, Thermal Fan Controller, Rack Mount 1U
XFX Radeon RX 560 1295MHz,4gb GDDR5
StarTech.com 1U Server Rack Rails
StarTech.com 1U Adjustable Vented Server Rack Mount Shelf
C-Zone Front Panel USB Hub
2 x Noctua NH-U9DX i4, Premium CPU Cooler

---

**Table 2**

*Malware Analysis Software*

---

<u>Item Description</u>
IDA Pro 32 bit and 64 bit version
VM Workstation Pro 15.0
Windows 8 (VM)
Windows 10 (VM)

---

**Summary**

This section described the research methodology utilized by this research for designing and evaluating an HTM classifier for streaming data. Both artificial and real-world data sets were identified along with a process for developing simulated evasive



malware containing stalling code. Finally, this section described the statistical techniques for determining the statistical significance of the HTM classifier and identified the resources needed to conduct the experiments.

## Chapter 4

### Results

This chapter provides the results of the experiments designed to evaluate the appropriateness of the HTM sequence classifier as an application in classifying artificial and real-world data streams containing concept drift, noise, and temporal dependencies. Additionally, this chapter provides the results of evaluating the HTM classifier as a possible technology for the detection of stalling code. The experimental design followed similar research conducted by Ghomeshi et al. (2019), as described in chapter 3.

Experiment I used the prequential evaluation method to evaluate the HTM sequence classifier on the following artificial (synthetic) data streams containing abrupt, gradual, and recurrent concept drift: Hyperplane, LED, RT, and SEA. The HTM classifier was then evaluated against the ARF, AWE, DWM, LevBag, OBoost, and VFDT classifiers using cost measures consisting of accuracy, memory consumption, and processor time.

Experiment II used the prequential evaluation method to evaluate the HTM sequence classifier on the following real-world data streams: Electricity, Airlines, and Poker. In similar fashion to experiment I, the HTM classifier was then evaluated against the ARF, AWE, DWM, LevBag, OBoost, and VFDT classifiers using the same cost measures as previously mentioned.

Experiment III explored the use of the HTM sequence classifier in recognizing stalling code within an executable file containing simulated evasive malware. This experiment depicted the potential of HTM sequence classifiers as a possible method for analyzing the behaviors of malware.

## **Experiment I (Artificial Data Streams)**

Experiment I implemented the classifier evaluation framework developed by Bifet et al. (2017) to evaluate the performance of all the previously listed classifiers on artificial data streams. It is important to note that even though the artificial data streams contained various forms of concept drift, they lacked temporal dependencies. Experiment I consisted of two separate runs with data sets containing 0% and 10% noise, respectively.

### ***Accuracy***

Tables 3 and 4 show the average accuracy for the classifiers over the four artificial data sets using the prequential error estimation approach. As can be seen from the tables, the HTM classifiers' best performance came in at 56% and 54% accuracy on the SEA datasets containing 0% and 10% noise, respectively. The HTM classifiers' higher accuracy rating on the SEA data set may be attributed to it consisting of only two features and one label, as opposed to the Random Tree data set containing 29 features and one label. It is interesting to note that the HTM classifiers' accuracy performance of 52% on the RT data set (which has a high degree of randomness) is comparable to that of the AWE, DWM, LevBag, and OBoost classifiers' accuracy percentage despite the absence of temporal dependencies within the data. This result would suggest that HTMs are more resilient to noise and randomness than traditional classifiers.

The HTM classifiers' poor performance on the synthetically generated data streams is likely due to the absence of temporal dependencies. As stated by Hawkins (2011a), "Time plays a crucial role in learning, inference, and prediction." (p. 12). In HTM, the Temporal Memory algorithm implements sequence memory. The algorithm learns

sequences of Sparse Distributed Representations (SDRs) formed by the Spatial Pooling algorithm and makes predictions of what the next input SDR will be (Hawkins et al., 2017). Therefore when the data stream is missing temporal dependencies, the Temporal Memory algorithm is unable to learn, thus affecting performance.

**Table 3**

*HTM Classifier Accuracy - Artificial Data sets with 0% noise*

---

Containing Abrupt, Gradual, Recurrent Concept Drifts

Dataset	Criteria	ARF	AWE	DWM	LevBag	OBoost	VFDT	HTM
Hyper	Ave.	0.943	0.974	0.997	0.770	0.820	0.972	<b>0.501</b>
	$\sigma$	0.021	0.003	0.001	0.002	0.002	0.017	0.000
	Min	0.917	0.971	0.996	0.769	0.818	0.951	0.500
	Max	0.976	0.979	0.998	0.773	0.825	0.995	0.501
LED	Ave.	0.998	0.998	1.000	0.630	0.837	1.000	<b>0.500</b>
	$\sigma$	0.001	0.001	0.000	0.022	0.004	0.000	0.000
	Min	0.997	0.997	1.000	0.622	0.834	1.000	0.500
	Max	1.000	1.000	1.000	0.692	0.848	1.000	0.500
RT	Ave.	0.814	0.593	0.609	0.648	0.674	0.874	<b>0.522</b>
	$\sigma$	0.004	0.000	0.000	0.095	0.002	0.008	0.001
	Min	0.809	0.592	0.608	0.618	0.673	0.878	0.522
	Max	0.821	0.594	0.609	0.919	0.679	0.859	0.523
SEA	Ave.	0.995	0.962	0.949	0.938	0.957	0.955	<b>0.560</b>
	$\sigma$	0.001	0.001	0.000	0.001	0.001	0.001	0.001
	Min	0.995	0.961	0.948	0.937	0.956	0.954	0.559
	Max	0.996	0.964	0.950	0.939	0.958	0.956	0.562

Table 4 depicts the performance of the classifiers when 10% noise is introduced into the data streams. It was found that the HTMs' best performance was with the SEA data stream, with an average accuracy of 54%. It is observed in Table 4 the degradation of accuracy throughout all of the classifiers. The most significant drop being that of the LevBag classifier on the LED data stream with an average accuracy of 40%. It was also observed that as noise was introduced, the HTM classifier remained relatively stable as compared to the other classifiers.

**Table 4**

*HTM Classifier Accuracy - Artificial Data sets with 10% noise*

---

Containing Abrupt, Gradual, Recurrent Concept Drifts

---

<u>Dataset</u>	<u>Criteria</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Hyper	Ave.	0.860	0.884	0.879	0.682	0.697	0.884	<b>0.500</b>
	$\sigma$	0.019	0.002	0.002	0.001	0.001	0.013	0.000
	Min	0.838	0.882	0.876	0.680	0.696	0.868	0.500
	Max	0.889	0.886	0.881	0.684	0.700	0.902	0.500
LED	Ave.	0.758	0.762	0.562	0.404	0.526	0.763	<b>0.509</b>
	$\sigma$	0.001	0.001	0.002	0.000	0.000	0.001	0.005
	Min	0.758	0.761	0.560	0.404	0.525	0.761	0.506
	Max	0.760	0.763	0.564	0.405	0.527	0.764	0.513
RT	Ave.	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	$\sigma$	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	Min	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	Max	n/a	n/a	n/a	n/a	n/a	n/a	n/a
SEA	Ave.	0.904	0.886	0.885	0.795	0.767	0.870	<b>0.541</b>
	$\sigma$	0.000	0.001	0.000	0.001	0.001	0.000	0.001
	Min	0.903	0.885	0.884	0.794	0.765	0.869	0.540
	Max	0.905	0.887	0.886	0.796	0.769	0.871	0.542

*Note:* The Random Tree (RT) data set was excluded from this run as it inherently contains noise.

### ***CPU run-times***

Tables 5 and 6 show the average run-time for the classifiers over the four artificial data sets using the prequential error estimation approach. As can be seen from Table 5, the HTM classifier run-time ranks 4<sup>th</sup>, 3<sup>rd</sup>, 3<sup>rd</sup>, and 6<sup>th</sup> on the Hyperplane, LED, RT, and SEA data sets containing 0% noise, respectively. The HTM classifiers' performance was relatively fast, compared to the other classifiers, which is likely due to the efficiency of the Temporal Memory algorithm. As per Cui et al., (2016), the Temporal Memory algorithm needs to continuously learn from the data streams and is designed to rapidly adapt to changes to learn new patterns.

**Table 5**

*HTM Classifier Avg. Run-Time (secs) - Artificial Data sets with 0% noise*

---

Containing Abrupt, Gradual, Recurrent Concept Drifts

<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Hyper	13776.03	4417.03	1209.69	28610.06	50473.21	331.73	<b>10346.99</b>
LED	29842.33	43475.40	3792.89	73243.41	61545.17	422.19	<b>9439.73</b>
RT	16115.15	14213.91	4735.79	92246.17	123796.25	458.20	<b>11832.48</b>
SEA	12469.20	2814.93	843.76	21311.02	25690.63	181.13	<b>22659.88</b>

Table 6 shows the impact of CPU run-time with the data sets containing 10% noise. The HTM classifier ranking changed very little; however, it is interesting to note that the AWE, VFDT, and HTM classifiers' performance remained relatively the same, while the ARF, DWM, LevBag, and OBoost classifiers suffered performance losses. Especially impacted was the ARF classifier, doubling the amount of CPU-time to complete its runs on all four data sets.

**Table 6**

*HTM Classifier Avg. Run-Time (secs) - Artificial Data sets with 10% noise*

---

Containing Abrupt, Gradual, Recurrent Concept Drifts

<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Hyper	22446.87	4498.01	1425.74	30333.96	52635.70	327.41	<b>9600.14</b>
LED	60329.59	43803.50	8385.59	69143.31	98845.17	1300.78	<b>10872.84</b>
RT	n/a	n/a	n/a	n/a	n/a	n/a	<b>n/a</b>
SEA	24038.47	2530.06	844.74	21844.44	27593.68	177.85	<b>23423.78</b>

*Note:* The Random Tree (RT) data set was excluded from this run as it inherently contains noise.

### **RAM usage**

Tables 7 and 8 show the average RAM usage for the classifiers over the four artificial data sets using the prequential error estimation approach. Out of all of the cost measures considered, average RAM usage turned out to be the Achilles heel for HTM.

The HTM classifier required nearly 132 GBs of RAM for the Hyperplane, LED, and SEA data sets, respectively, and 722 GBs for the RT data sets, thereby resulting in heavy usage of SWAP memory. Although not recorded as a cost measure, it was noticed that CPU temperatures reached as high as 190 degrees Fahrenheit during the majority of these runs. Much of the memory consumption can be attributed to the size of the SDRs needed to represent the data. Higher sparsity in the SDRs of the data improved the HTMs' performance, but at higher consumption of memory.

**Table 7**

*HTM Classifier Avg. RAM-Usage (in K) - Artificial Data sets with 0% noise  
Containing Abrupt, Gradual, Recurrent Concept Drifts*

<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Hyper	51856.63	618.60	52.87	5257.62	5401.25	4586.66	<b>131198722.66</b>
LED	3888.89	3779.19	262.47	10714.83	7098.88	318.66	<b>131191578.13</b>
RT	71710.50	1747.54	234.87	13060.88	13227.31	17164.55	<b>722070304.84</b>
SEA	10076.32	228.57	30.89	2522.09	2649.77	1184.90	<b>131690084.23</b>

Table 8 shows the impact of adding 10% noise to the data streams. The HTM, AWE, LevBag classifiers RAM consumption remained relatively the same throughout all 3 data sets, while the ARF, DWM, OBoost, and VFDT saw sizable increases.

In summary, Tables 3 – 8 suggest that the introduction of noise into the artificial data streams resulted in a decrease in the cost measures of average accuracy, and an increase in both run-time and RAM usage for all but the HTM classifier. These results suggest the HTM classifier is relatively scalable to large data sets. However, the poor performance in both accuracy and RAM usage likely rules out the HTM classifier as a practical solution for classifying artificial data sets missing temporal dependencies.

**Table 8***HTM Classifier Avg. RAM-Usage (in K) - Artificial Data sets with 10% noise*

Containing Abrupt, Gradual, Recurrent Concept Drifts							
<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Hyper	150370.67	618.55	85.29	5264.65	5373.65	6013.00	<b>131199908.20</b>
LED	30208.44	3836.24	524.63	10713.23	10898.32	1300.78	<b>131201768.03</b>
RT	n/a	n/a	n/a	n/a	n/a	n/a	n/a
SEA	197398.28	228.65	<b>30.60</b>	2529.93	2657.64	2301.57	<b>131721509.55</b>

*Note:* The Random Tree (RT) data set was excluded from this run as it inherently contains noise.

### ***Concept drift***

Figure 7 shows the effect of concept drift on the HTM classifiers' accuracy percentages over the SEA data stream at both 1-step and 5-step predictions with 0% noise. As described in Chapter 3, abrupt and recurrent concept drifts were manually inserted into the data stream, at instance numbers 200K, 400K, 600K, and 800K. The data stream began with the SEA function  $f_0$  and abruptly shifted to  $f_1$  with a width (width of concept drift change) of one at the instance number 200K and experienced a shift back to the function  $f_0$  at 400K thereby creating a recurrent concept drift. There was another abrupt concept drift at 600K with a function shift from  $f_0$  to  $f_2$ , and a final recurrent concept drift was encountered at 800K with a function shift from  $f_2$  back to  $f_1$ .

It is observed that in Figure 7, at instance 200K, where the first abrupt concept drift occurred ( $f_1$ ), resulted in the HTM classifiers' average accuracy percentage to begin a gradual drop from ~56% down to ~54%. The classifier did not cope well with the first drift. However, once the first recurrent drift occurred at instance 400K ( $f_0$ ), the HTM classifier's accuracy percentage began to improve to nearly 57%. The accuracy percentage dropped once again when the next abrupt concept drift ( $f_2$ ) occurred at the 600K point, but surprisingly a gradual improvement was noticed beginning at the 700K



mark. The sudden improvement at the 700K mark implies that the HTM classifier is becoming more sensitive to concept drifts and alters its predictions accordingly. Finally, the last recurrent concept drift was experienced at the 800K mark without a drop in average accuracy. This suggests that the HTM classifier’s temporal memory was better able to deal with the transition since it had seen ( $f_1$ ) earlier.

When comparing the accuracy percentages for the 1-step and 5-step predictions, it was surprising to observe that the HTM classifiers’ average accuracy percentages were relatively identical. These results support the claim by Cui et al. (2016) that HTM technology is capable of making multiple simultaneous predictions.

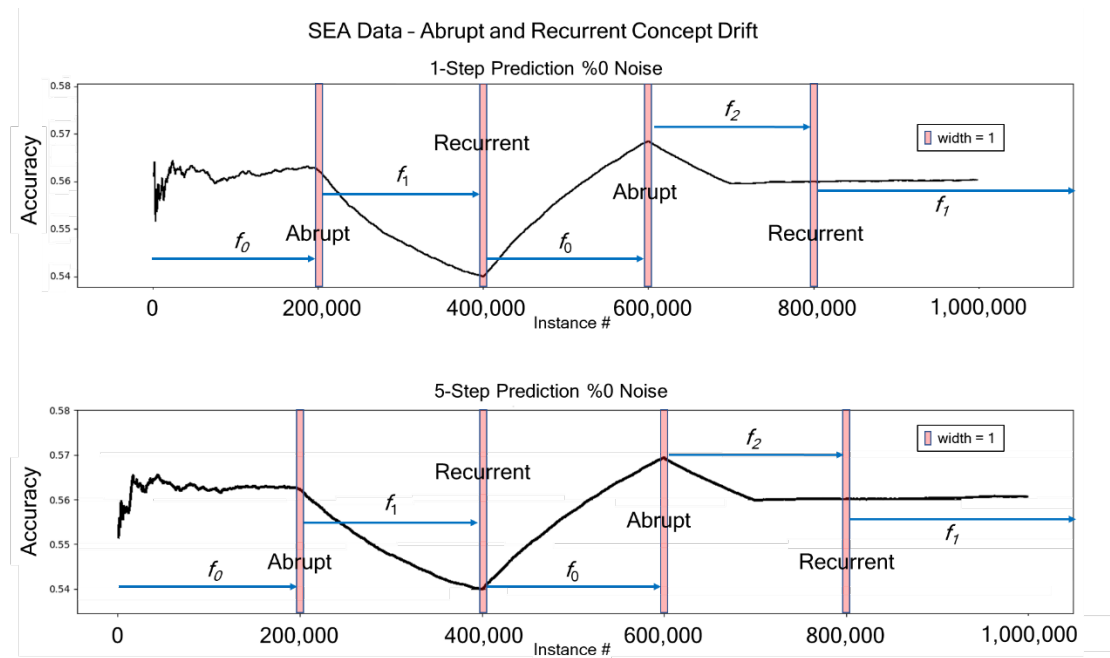


Figure 7. Abrupt and Recurrent concept drift accuracy – SEA data with 0% noise - 1 and 5 step predictions.

The same observations are made from Figure 8, which plotted the HTM classifiers’ performance with gradual and recurrent concept drift. As with Figure 7, the data stream begins with the SEA function  $f_0$  and gradually shifted to  $f_1$  with a width of 10000 centered at instance number 200K and experienced a recurrent concept drift

centered at 400K back to function  $f_0$ ; followed another gradual concept drift centered at 600K to function  $f_0$  and finally the last recurrent concept drift occurring at the 800K mark with a shift in function from  $f_2$  back to the previously seen function of  $f_1$ .

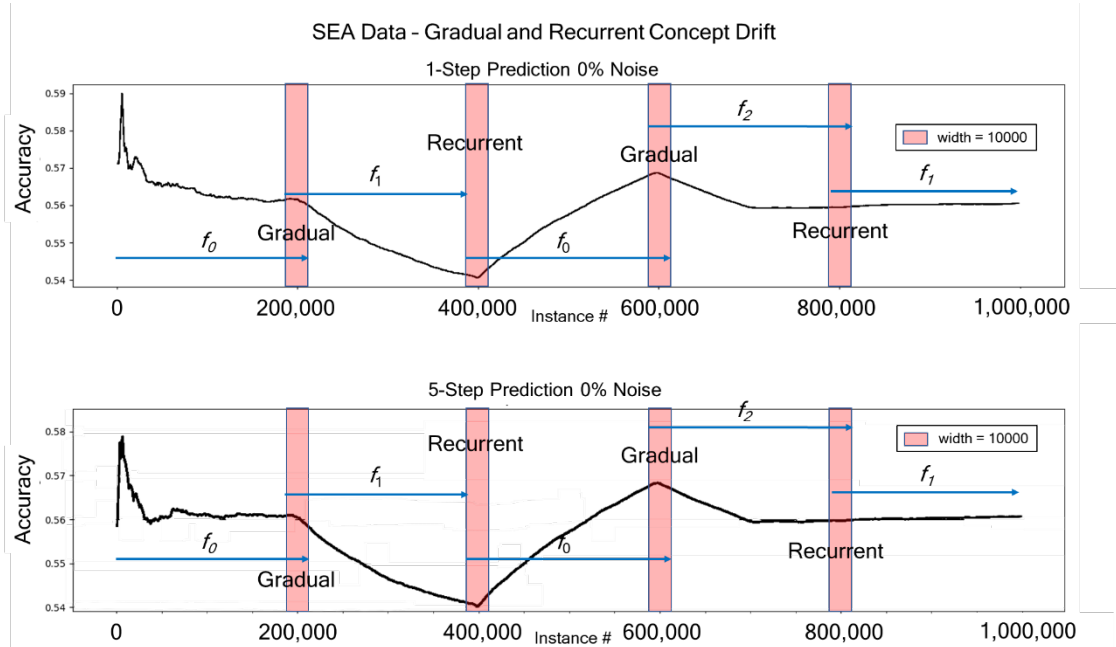


Figure 8. Gradual and recurrent concept drift accuracy – SEA data with 0% noise - 1 and 5 step predictions.

The behavior of the HTM classifier with gradual and recurrent concept drift (Figure 8), is nearly identical to that of abrupt and recurrent concept drift (Figure 7). The primary difference being that the slopes of the drops and increases in accuracy percentage are not as steep.

Figures 9 (abrupt and recurrent) and 12 (gradual and recurrent) depicted the HTM classifiers' behavior when 10% noise was introduced into the SEA data stream. Aside from the early randomness of the accuracy percentage, attributed to the random instantiation of temporal memory algorithm, the behavior of the HTM classifier is relatively the same as that shown in Figures 6 and 7.

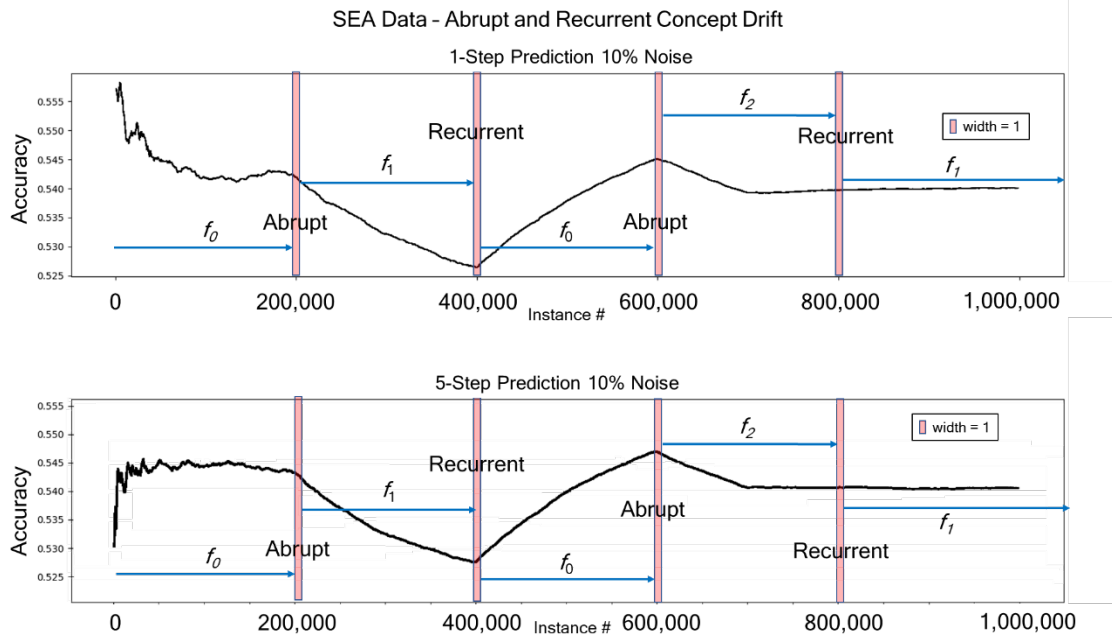


Figure 9. Abrupt and recurrent concept drift accuracy – SEA data with 10% noise - 1 and 5 step predictions.

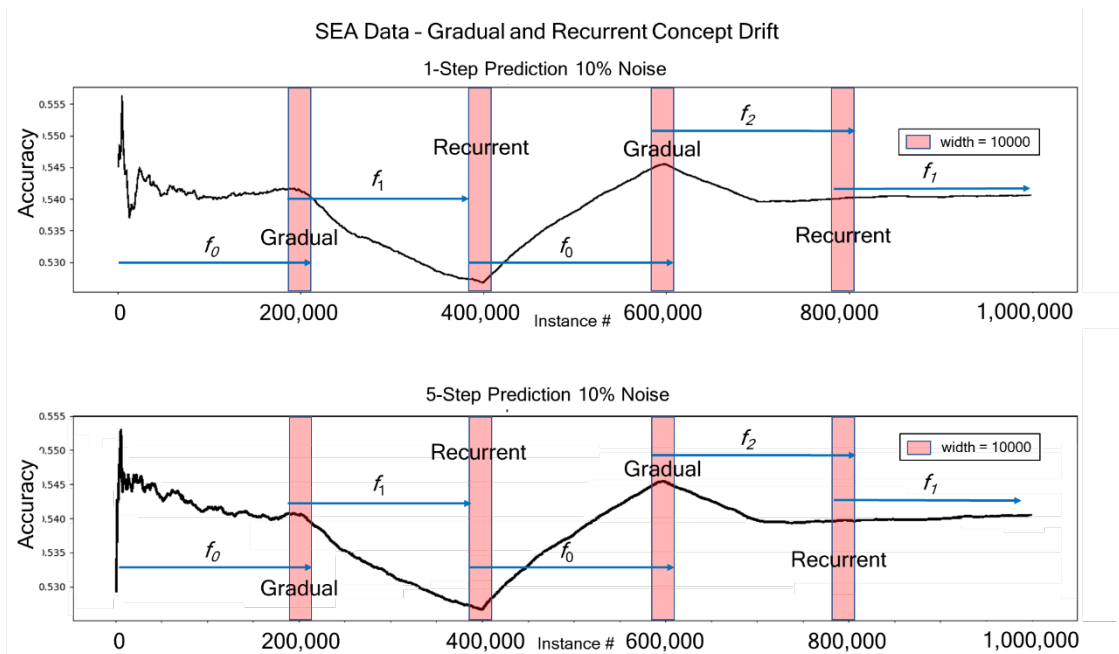


Figure 10. Gradual and recurrent concept drift accuracy – SEA data with 10% noise - 1 and 5 step predictions.

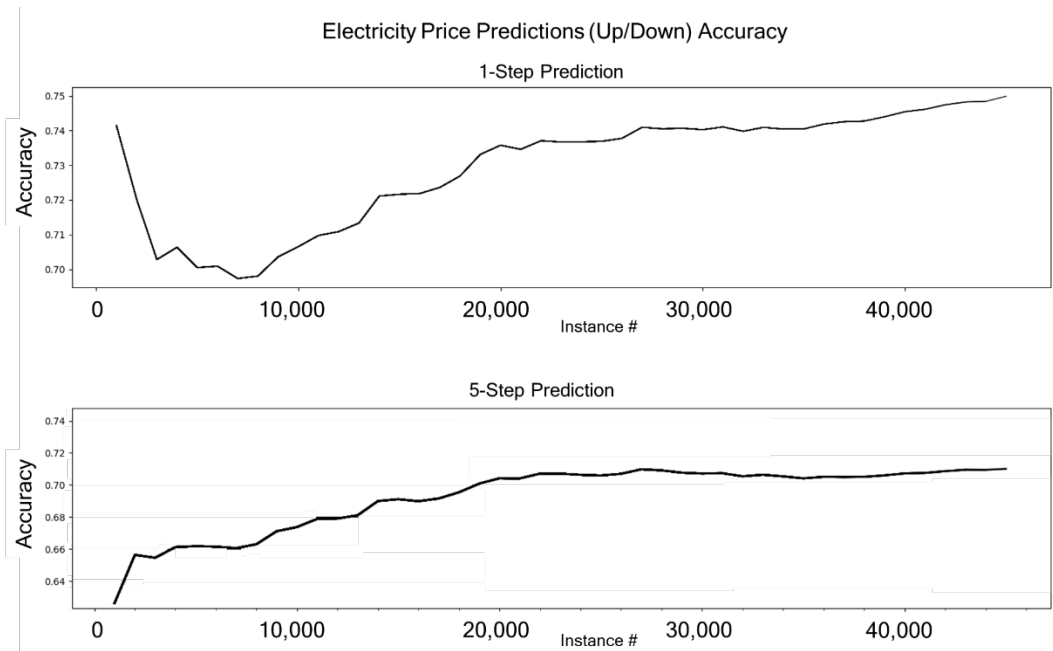
## **Experiment II (Real-world Data Streams)**

Experiment II was nearly identical to Experiment I, with the primary difference being that the data sets came from real-world data streams, and concept drift and noise were not manually added. Instead of variants, each experiment was repeated ten times over the same data stream.

### ***Electricity Price Predictions (Up/Down)***

As shown in Table 9 and Figure 11, the HTM classifier achieved an average accuracy of 75%. While only 6<sup>th</sup> best when compared to the other classifiers, the HTM classifiers' performance of 75% was comparable to that of the OBoost (79%) and VFDT (80%) classifiers; and superior to that of the AWE (72%) classifier. It is likely that the HTM classifier would have performed better had the electricity data set contained more than 45k records.

Figure 11 plots the price prediction accuracy of the HTM classifier on the electricity data set. It was observed that the HTM classifiers' accuracy percentage for the 1-step prediction begins at 74% but drops to ~70% near the 2500 instance number. This drop is due to the HTM classifiers' temporal memory algorithm learning the temporal dependencies within the data. It was observed at instance number 8000, that the HTM classifier began to steadily improve the accuracy of its predictions consistently until the end of the data stream. It is interesting to note that the 5-step prediction accuracy is relatively the same as that of the 1-step with the primary difference coming at the beginning of the data stream in which the two predictions converge near the 8000 instance number.



*Figure 11. Electricity Price Predictions (Up/Down) Accuracy – 1 and 5 step predictions*  
***Airline Flight Predictions (On-Time/Late)***

As shown in Table 9, the HTM classifier ranked 4<sup>th</sup> in average accuracy on the airlines data set. The HTM classifiers' average accuracy of 60% out-performed that of the AWE (57%), LevBag (56%), and the OBoost (55%) classifiers; and gave comparable results to those of the ARF (66%), DWM (61%), and VFDT (64%) classifiers. It was likely that the HTM classifiers' better performance was attributable to temporal dependencies and the customized SDR encoder for the airport field in the data. As detailed in Appendix A, the airport data field was transformed from a text string in the original file to an SDR encoding, which consisted of the actual latitude/longitude, altitude, and size (small, medium, large) of each airport. It is speculated that increasing the sparsity (size) of the encodings and temporal memory settings may have improved the accuracy percentage of the HTM classifier but likely at the expense of the CPU run-time (Table 10) and RAM-Usage (Table 11) cost measures.

### *Poker Hand Predictions*

As shown in Table 9, the HTM classifier ranked last amongst the classifiers tested on the poker data set, its average accuracy percentage of 44% was comparable to that of all but the ARF and VFDT. The poor accuracy percentage likely was due to the poker data set containing ten classes (Appendix H).

**Table 9**

*Accuracy (%) of the Classifiers with Real-World Data*

---

Compared using the Prequential Error Estimation Method

<u>Dataset</u>	<u>Criteria</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Airlines	Ave.	0.66	0.57	0.61	0.56	0.55	0.64	<b>0.60</b>
	$\sigma$	0.00	0.00	0.00	0.00	0.00	0.00	0.01
	Min	0.66	0.57	0.61	0.56	0.55	0.64	0.59
	Max	0.67	0.57	0.61	0.56	0.55	0.64	0.60
Elec	Ave.	0.88	0.72	0.80	0.85	0.79	0.80	<b>0.75</b>
	$\sigma$	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Min	0.88	0.72	0.80	0.85	0.78	0.80	0.75
	Max	0.89	0.72	0.80	0.85	0.79	0.80	0.75
Poker	Ave.	0.54	0.48	0.47	0.46	0.47	0.52	<b>0.44</b>
	$\sigma$	0.00	0.00	0.00	0.00	0.00	0.00	0.01
	Min	0.53	0.48	0.47	0.46	0.47	0.52	0.43
	Max	0.54	0.48	0.47	0.46	0.47	0.52	0.44

---

### *CPU run-times*

Table 10 shows the average CPU run-time of the classifiers on the real-world data sets. The HTM classifier ranked 4<sup>th</sup> for the airlines data set, 7<sup>th</sup> for the electricity data set, and 4<sup>th</sup> for the poker data set. It is likely that the HTM's 7<sup>th</sup> place ranking for the electricity data set is due to its small number of records (45K). It was observed that the larger data sets of airlines and poker, with 539K and 1000K respectively, demanded more CPU time as shown by the ARF's, LevBag's, and OBoost's results.

**Table 10***Average CPU run-time (in seconds) of the Classifiers with Real-World Data*

Compared using the Prequential Error Estimation Method

<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Airlines	10527.6	2393.76	1529.12	14301.71	20546.3	206.789	<b>7082.84</b>
Electricity	502.912	136.094	56.083	1217.428	1624.033	10.375	<b>1967.761</b>
Poker	29044.56	24.60	6334.51	31327.38	49277.20	1767.51	<b>10593.25</b>

***RAM usage***

Table 11 shows the average RAM usage in Kilobytes consumed by each of the classifiers over the real-world data sets. It was observed that the HTM classifier performed the poorest of all the classifiers, even with the smaller electricity data set. This was attributed to the large SDR needed to encode the features of the data sets along with the large memory requirements for the Spatial and Temporal poolers (Appendix I). The large memory requirements of the HTM classifier were not unexpected. Ahmad and Scheinkman (2019) demonstrated the benefits of high dimensional sparse representations; however, an increase in sparsity directly increases memory consumption. The sparsity of the SDRs had to be significantly increased (Appendix H) to obtain the accuracy percentages recorded in Table 9. As Table 11 shows, the increase in SDR size resulted in a drastic increase in memory consumption.

**Table 11***Average RAM usage (in K) of the Classifiers with Real-World data sets*

Compared using the Prequential Error Estimation Method

<u>Dataset</u>	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
Airlines	43471.83	450.16	59.25	4100.61	4211.00	2384.37	<b>33593500</b>
Electricity	7638.76	507.22	66.83	4461.69	4482.07	291.33	<b>5214611.33</b>
Poker	47235.76	570.19	124.62	124.62	5269.99	5416.49	<b>66683675.8</b>

### Experiment III (Simulated Malware)

The purpose of experiment III was to assess the predictive performance of the HTM classifier against a known example of stalling code. This analysis was achieved by providing the HTM classifier the EIP, EAX, EBX, ECX, EDX, and EFLAGS register values resulting from executing each Line of Code (LOC) during dynamic disassembly of an executable via a IDAPro Python script and plotting the results. The next step was to provide the HTM classifier an executable from a sorting algorithm and plot the results. The resulting plots (Figures 12 – 27) were then compared against each other for analysis.

As can be seen in Tables 12 and 13, the HTM achieved high accuracy scores in predicting all eight flags bits. Furthermore, Table 13 demonstrates the multi-prediction capabilities of the HTM classifier by delivering high accuracy scores for 5-step predictions.

**Table 12**

*HTM classifier Malware Analysis EFLAGS prediction accuracy %*

1-step predictions

<u>Algorithm</u>	<u>CARRY</u>	<u>PARITY</u>	<u>AUX</u>	<u>ZERO</u>	<u>SIGN</u>	<u>TRAP</u>	<u>DIR</u>	<u>OVERFLOW</u>
Stalling	0.99	0.89	0.97	0.99	0.93	1	1	1
Sorting	0.97	0.89	0.97	0.97	0.98	1	1	0.99

**Table 13**

*HTM classifier Malware Analysis EFLAGS prediction accuracy %*

5-step predictions

<u>Algorithm</u>	<u>CARRY</u>	<u>PARITY</u>	<u>AUX</u>	<u>ZERO</u>	<u>SIGN</u>	<u>TRAP</u>	<u>DIR</u>	<u>OVERFLOW</u>
Stalling	0.99	0.88	0.94	1.00	0.93	1.00	1.00	1.00
Sorting	0.95	0.76	0.95	0.92	0.96	1.00	1.00	0.99

Table 14 shows the RAM usage of the HTM classifier for both the stalling code and the sorting algorithm. It was observed that the sorting algorithm required consumed



nearly twice the amount of RAM as that of the stalling algorithm despite having the same SDR encoders, Spatial and Temporal Pooler parameters (Appendices H and M). It was likely that this was the result of the learning process of the Temporal Pooler, as the sorting algorithm had of  $O(n^2)$  while the stalling code had  $O(n)$  complexity. Simply stated, it was harder for the HTM classifier to learn the temporal dependencies of the sorting algorithm.

**Table 14**

*HTM classifier Malware Analysis*

RAM usage in K	
<u>Algorithm</u>	<u>RAM</u>
Stalling	9469554.69
Sorting	17830265.63

The following list of Figures (12 – 27) is the result of the HTM classifier predicting each flag bit of the x86 EFLAGS register given the state of the machine by the IDAPro Python script for each LOC that was disassembled for both the stalling and sorting algorithms.

***Findings: Carry Flag***

As per Irvine (1999), “The Carry flag (CF) is set when the result of an unsigned arithmetic operation is too large to fit into the destination.” (p. 40)

Figures 12 and 13 depict the HTM classifiers' accuracy of prediction the CF for both the stalling code and sorting algorithm. It was observed that the stalling code quickly flatlined, while the sorting algorithm fluctuated throughout the entirety of its execution.

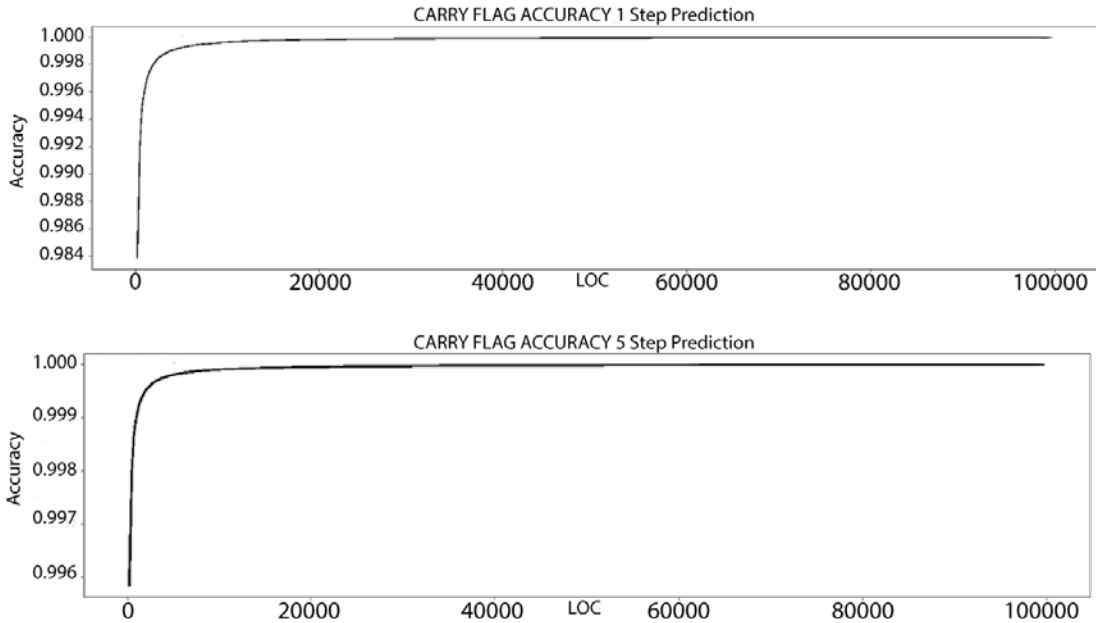


Figure 12. Stalling code: HTM Carry Flag Accuracy - 1 and 5 step predictions

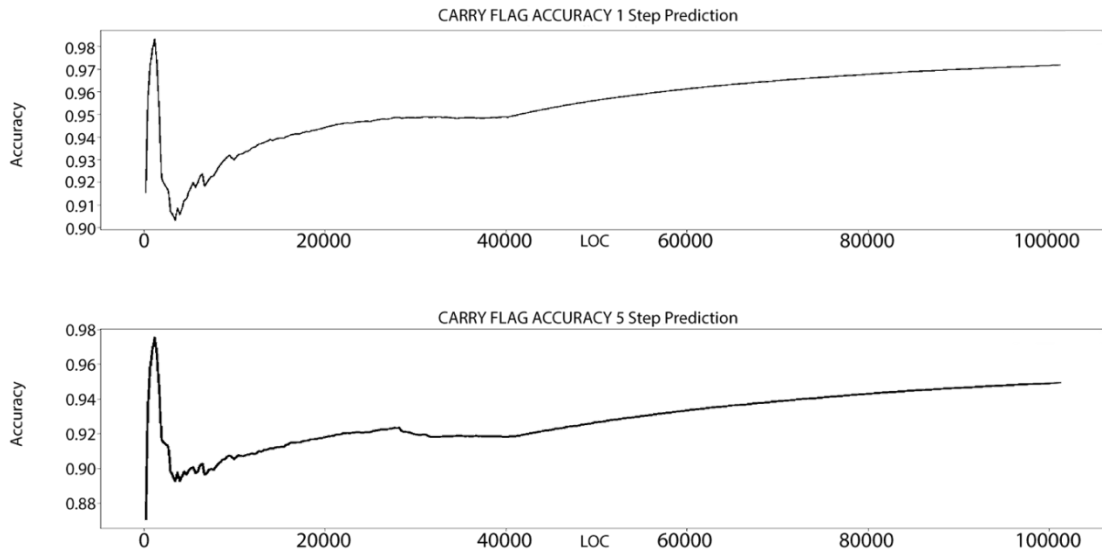


Figure 13. Sorting code: HTM Carry Flag Accuracy - 1 and 5 step predictions.

**Findings: Parity Flag**

As per Irvine (1999),

The Parity flag (PF) is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error

checking when there is a possibility that data might be altered or corrupted. (p. 41)

Figures 14 and 15 depict the HTM classifiers' prediction accuracy of the PF for both the stalling code and sorting algorithms. Although somewhat similar, subtle fluctuations in the sorting algorithm plot can be seen, while those of the stalling code approach a flatline.

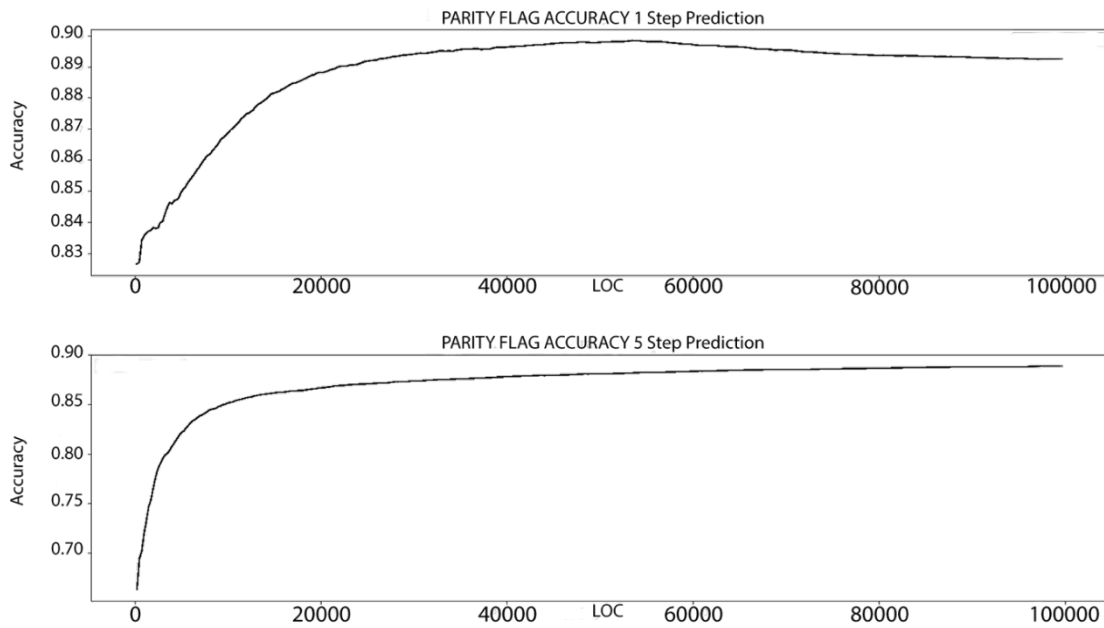


Figure 14. Stalling code: HTM Parity Flag accuracy - 1 and 5 step predictions.

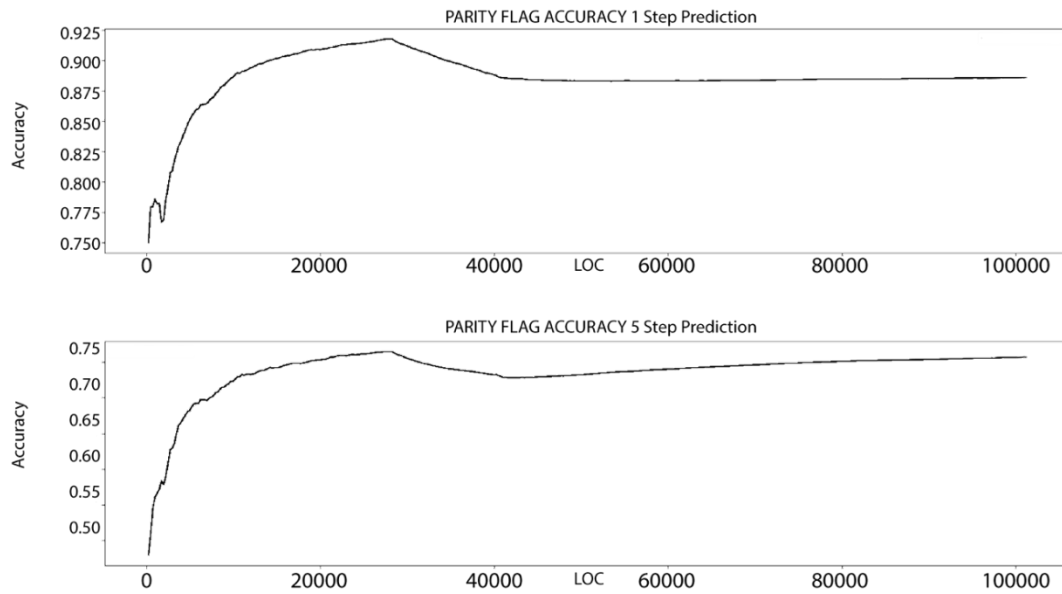


Figure 15. Sorting code: HTM Parity Flag accuracy - 1 and 5 step predictions.

### ***Auxiliary Flag***

As per Irvine (1999), “The Auxiliary Carry flag (AC) is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.” (p. 41)

Figures 16 and 17 depict the HTM classifiers AC prediction accuracies. It was observed that both the stalling code and sorting algorithm had similar results; however, subtle fluctuations can be seen in the plot of the sorting algorithm.

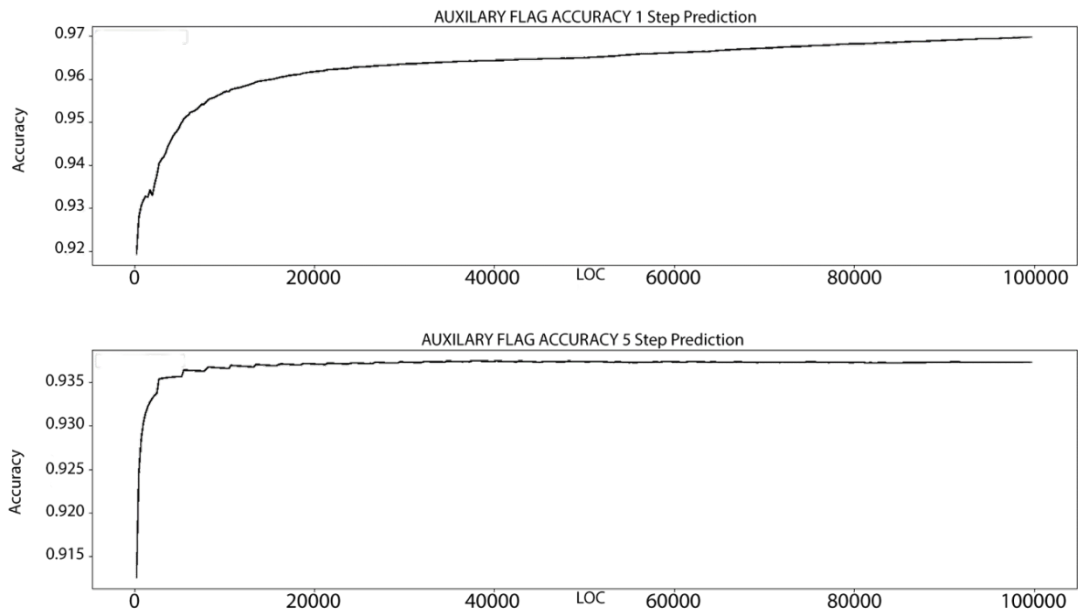


Figure 16. Stalling code: HTM Auxiliary Flag accuracy - 1 and 5 step predictions.

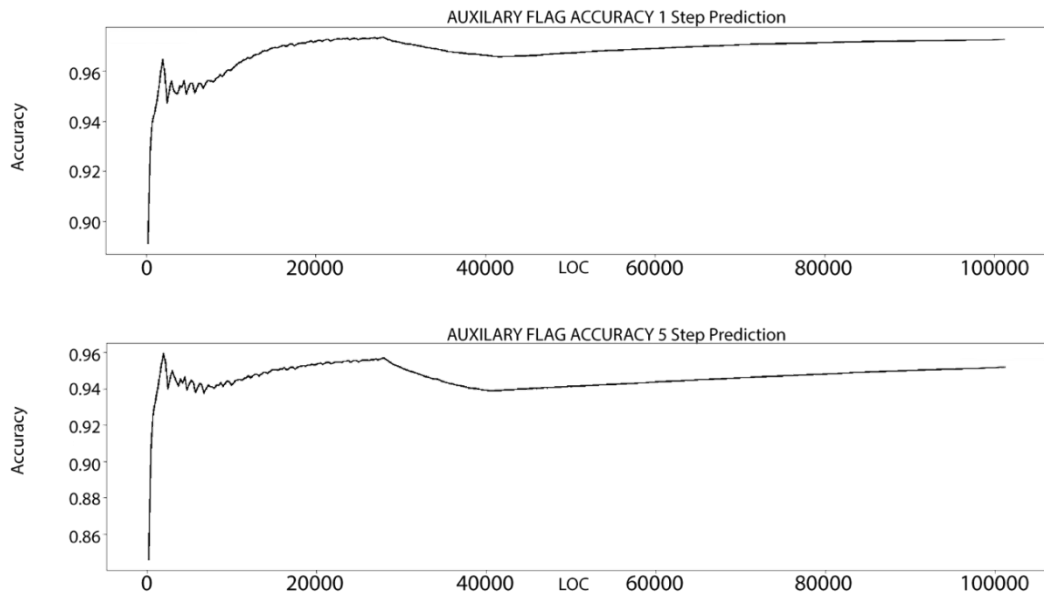
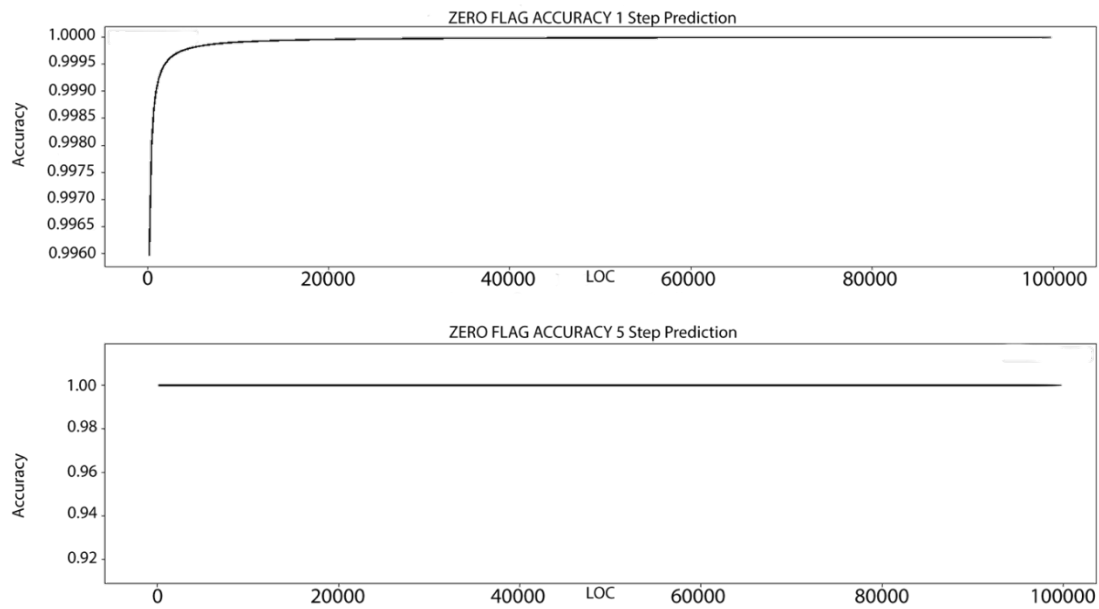


Figure 17. Sorting code: HTM Auxiliary Flag accuracy - 1 and 5 step predictions.

**Findings: Zero Flag**

As per Irvine (1999), “The Zero flag (ZF) is set when the result of an arithmetic or logical operation generates a result of zero.”

Figures 18 and 19 show the results of plotting the HTM classifiers’ ZF accuracy predictions. It was observed that the stalling code continues its trend of flatlining EFLAGS register, while the sorting algorithm shows heavy use.



*Figure 18.* Stalling code: HTM Zero Flag accuracy - 1 and 5 step predictions.

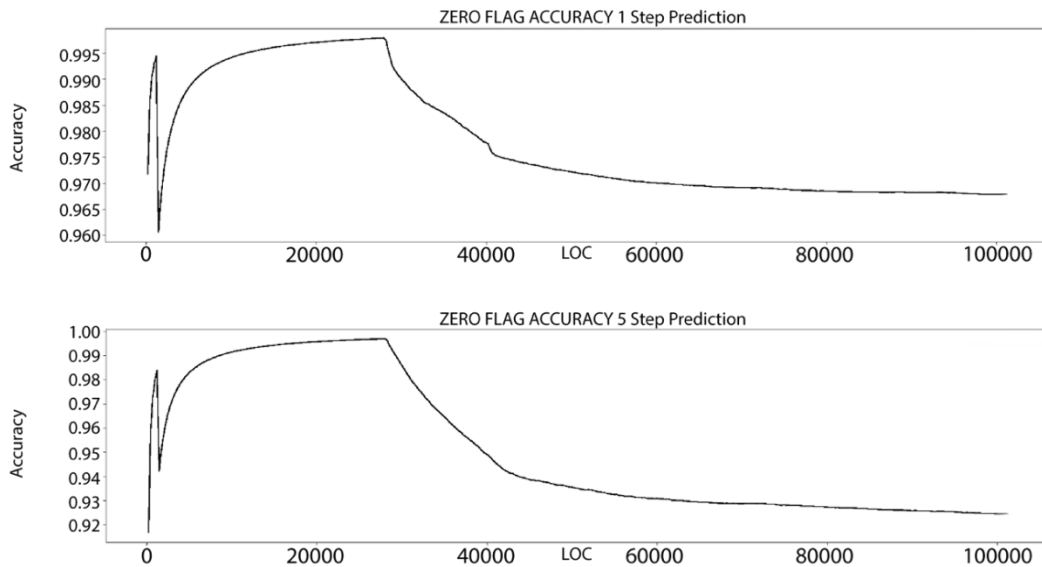


Figure 19. Sorting code: HTM Zero Flag accuracy - 1 and 5 step predictions.

**Findings: Sign Flag**

As per Irvine (1999), “The Sign flag (SF) is set when the result of an arithmetic or logical operation generates a negative result.” (p. 40)

Figures 20 and 21 depict the behavior of the SF for the stalling code and sorting code executables. It is observed that the stalling code plot gradually flatlined at ~92%, while the sorting algorithm fluctuated a great deal until the 20K LOC mark, at which point it begins to stabilize. These figures reinforce the idea that stalling code doesn’t want to draw attention to itself by performing a large amount of ALU computations.

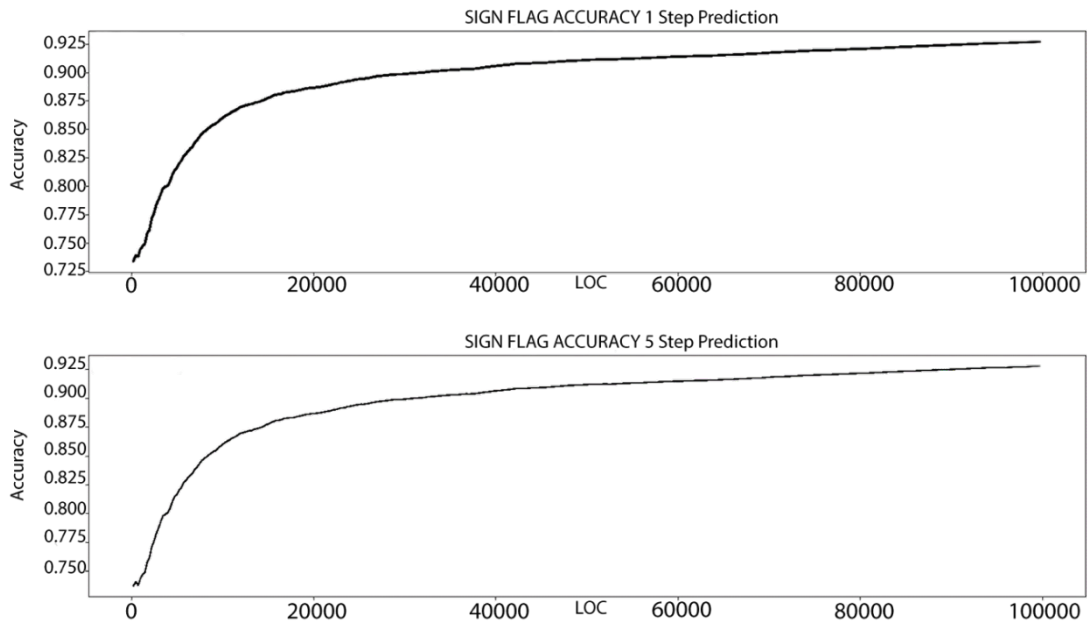


Figure 20. Stalling code: HTM Sign Flag accuracy - 1 and 5 step predictions.

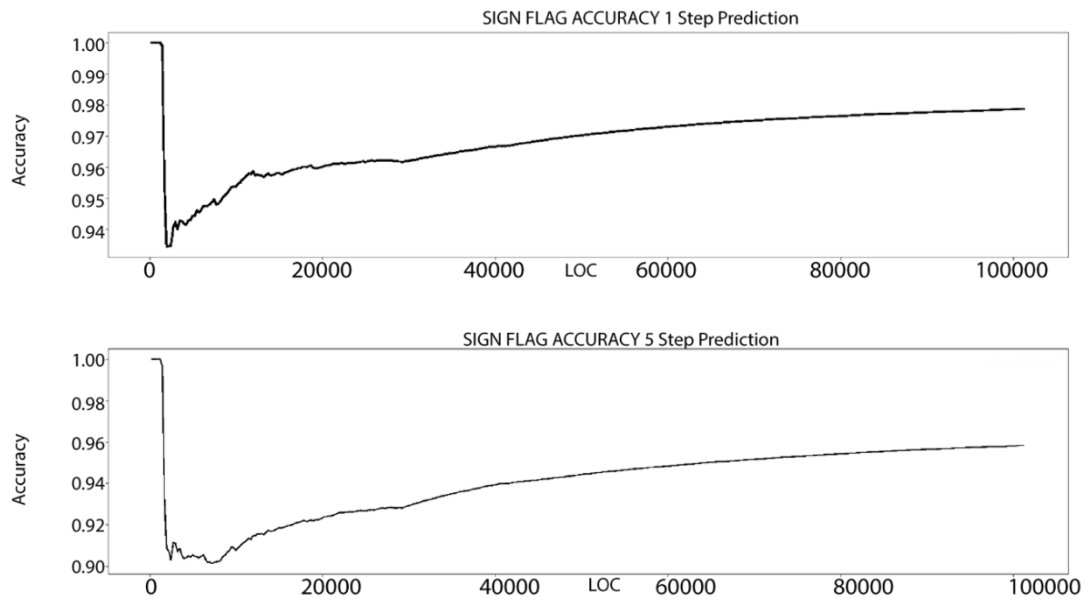


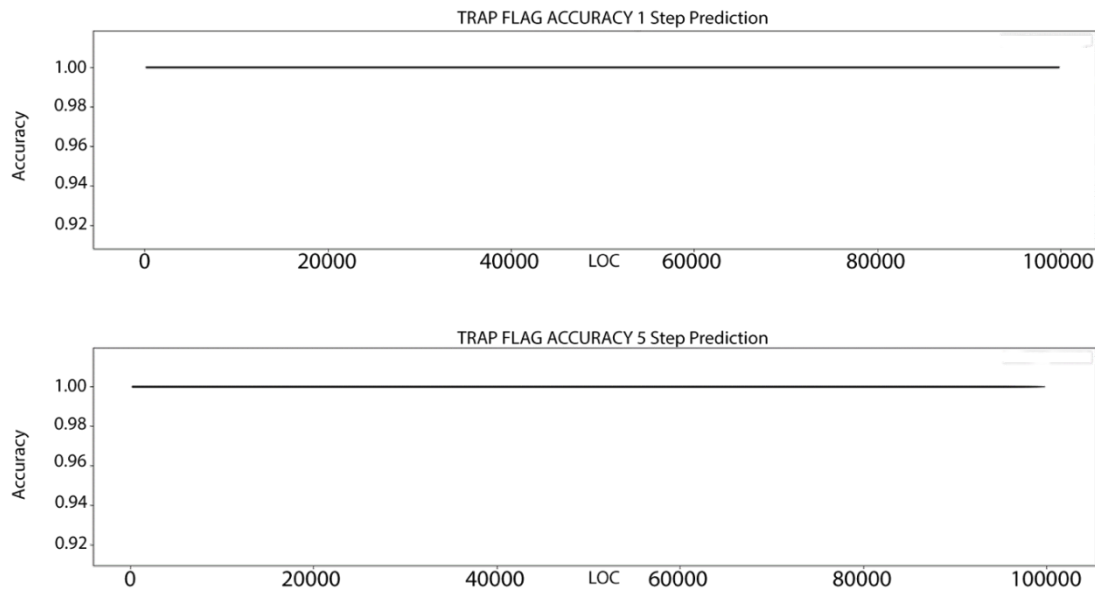
Figure 21. Sorting code: HTM Sign Flag accuracy - 1 and 5 step predictions.



### ***Findings: Trap Flag***

As defined by Sikorski et al. (2012), “The trap flag (TF) is used for debugging. The x86 processor will execute only one instruction at a time if this flag is set.” (p. 72)

Figures 17 and 18 show the behavior of the TF for the stalling code and sorting code executables. When the TF was set, the processor executes one instruction and then generates an exception (Sikorski & Honig, 2012), also known as single-stepping. It was observed that the TF is set immediately upon executing the simulated malware example in both Figures 17 and 18. The IDAPro disassembler likely sets the TF flag to step through and disassemble each instruction and record the state of the machine. A common malware evasion technique is to check the TF during run-time and cease its malicious behavior if this flag is set or to turn it off to prevent dynamic analysis.



*Figure 22. Stalling code: HTM Trap Flag accuracy - 1 and 5 step predictions.*

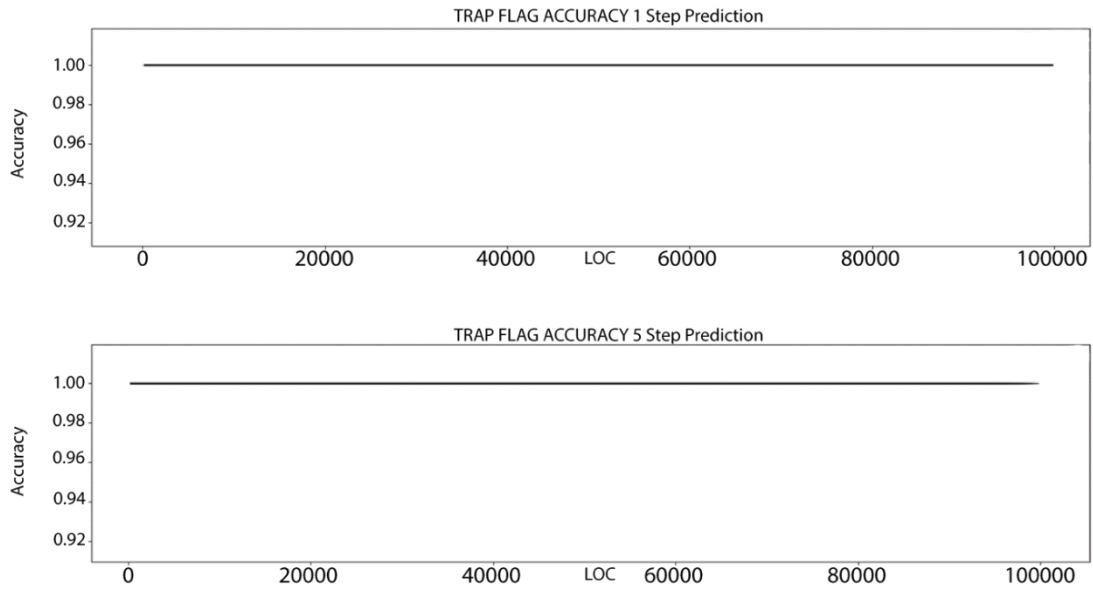


Figure 23. Sorting code: HTM Trap Flag accuracy - 1 and 5 step predictions.

**Findings: Direction Flag**

As stated by Irvine (1999) “The Direction Flag (DF) determines whether the index register is incremented or decremented during each iteration of a string primitive instruction.” (p. 383)

Figures 17 and 18 depict the behavior of the DF flag for the stalling code and sorting code executables. The DF flag is primarily used for string operations where the source and destination registers are incremented if the flag is set to 0 and decremented if it is set to 1. Neither the stalling code nor the sorting code performed any string operations. Therefore little observation can be made from these two figures.

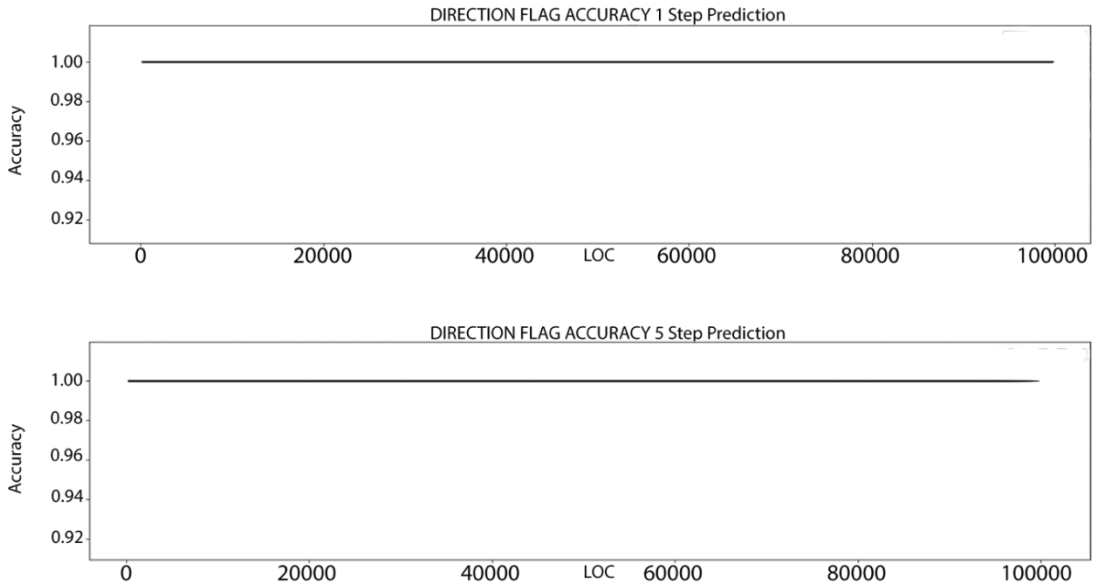


Figure 24. Stalling code: HTM Direction Flag accuracy - 1 and 5 step predictions.

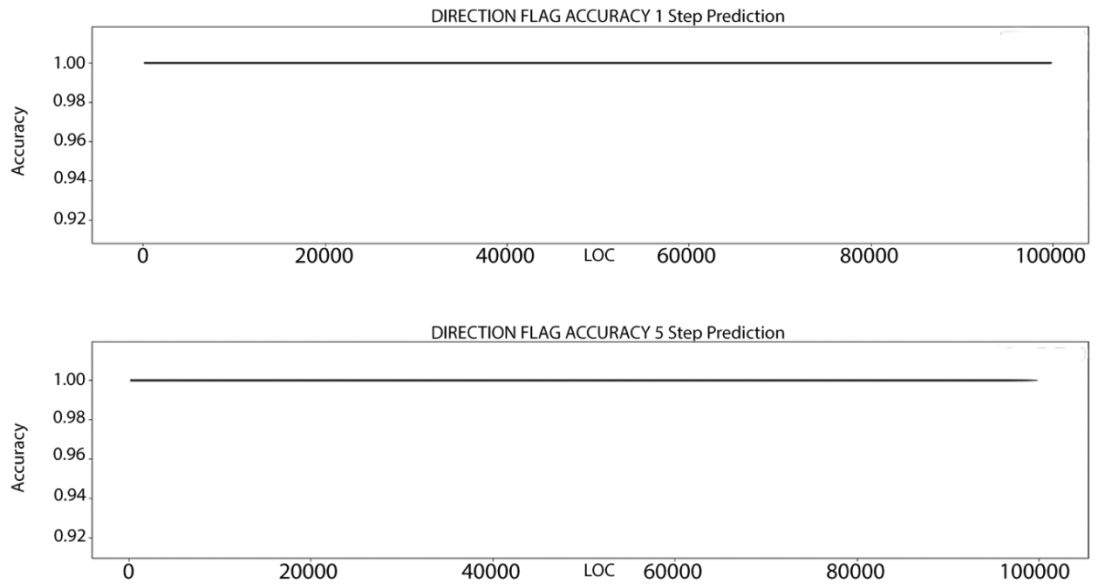


Figure 25. Sorting code: HTM Direction Flag accuracy - 1 and 5 step predictions.

**Findings: Overflow Flag**

As defined by Irvine (1999), “The Overflow flag (OF) is set when the result of a signed arithmetic operation is too large or too small to fit into the destination.” (p. 41)

Figures 21 and 22 show the behavior of the OF for the stalling code and sorting code executables. It was observed that the stalling code OF was immediately set and never changed, while that of the sorting algorithm fluctuated throughout the entirety of the experiment, resulting in lower accuracy. The results of Figures 21 and 22 implies that stalling code has the tendency to flatline the OF, while that of the sorting algorithm fluctuates more frequently. This is likely due to malware authors not wanting to draw too much attention to their malware.

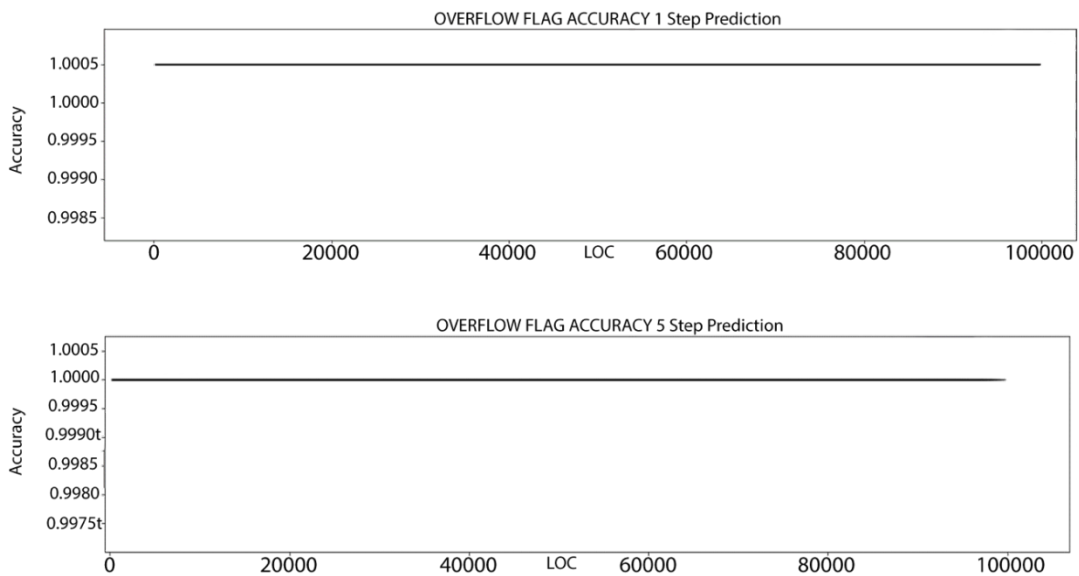


Figure 26. Stalling code: HTM Overflow Flag accuracy - 1 and 5 step predictions.

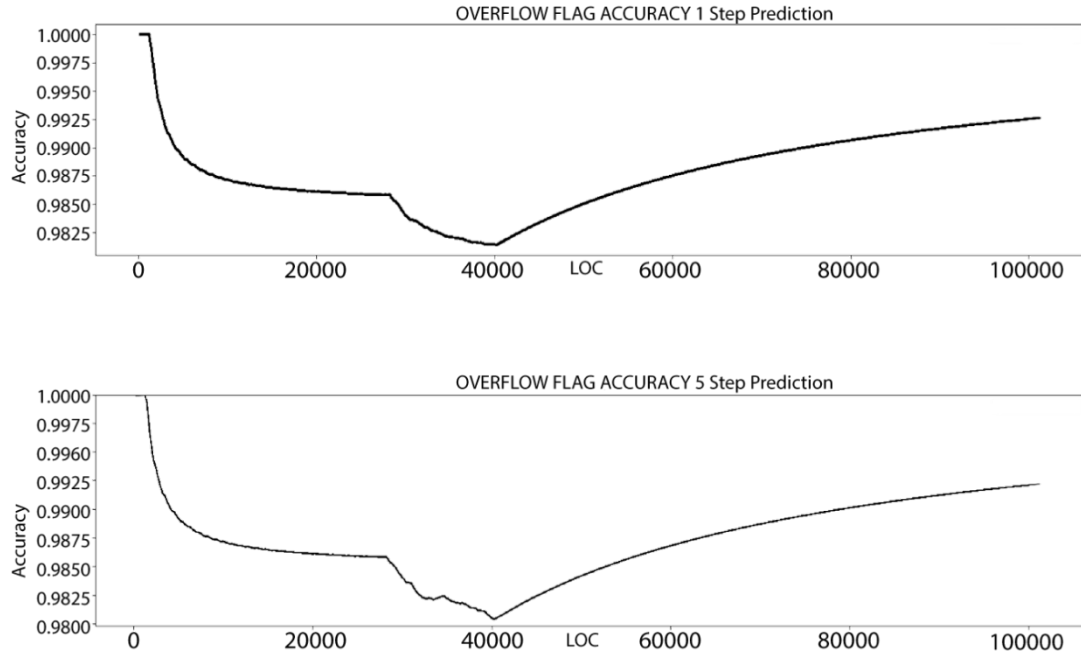


Figure 27. Sorting code: HTM Overflow Flag accuracy - 1 and 5 step predictions.

### Statistical Analysis

As stated by (Bifet et al., 2017), the main challenge in classifier analysis is to know when a classifier is outperforming another classifier only by chance, and when there is a statistical significance to that claim. Thus in order to show that the HTM classifier was statistically significant when compared to other classifiers, we first show that the null hypothesis is false.

Claim: All seven classifiers have the same probability distribution.

$H_0$ :  $\tilde{X}_{ARF} = \tilde{X}_{AWE} = \tilde{X}_{DWM} = \tilde{X}_{LevBag} = \tilde{X}_{OBoost} = \tilde{X}_{VFDT} = \tilde{X}_{HTM}$

$H_A$ : at least two classifiers differ from each other.

Table 15 shows the average rank of each classifier included in the experiments I and II. Note that for  $k = 7$  and  $N = 7$ , as there are seven classifiers and seven different datasets, the resulting value of the Friedman test statistic is :  $\chi_F^2 = 22.54$  with 6 degrees of

freedom at significance level  $\alpha = 0.05$ . The resulting p-value of 0.000965 allowed for the rejection of the omnibus null hypothesis that all samples (groups) are from the same distribution. Thus we can conclude that the accuracy values of the 7 seven classifiers are significantly different as the value of 22.54 is greater than the critical value of 12.5916.

After the null-hypothesis was rejected, I proceeded with the Nemenyi posthoc test. The critical value with  $k = 7$  and  $\alpha = 0.10$  is  $q_{0.10} = 3.11$ . The performance of any two classifiers is considered significantly different if their corresponding average ranks differ by at least the critical difference (CD). Figure 28 graphically represents the comparison of the classifiers based on their critical differences, and Table 15 shows the classifier rankings in table format. Both Figure 28 and Table 15 shows that the HTM classifier is significantly different than the ARF, DWM, and VFDT classifiers but not significantly different from the AWE, LevBag, and OBoost classifiers.

**Table 15**

<i>Average classifier rank including Artificial and Real-World data sets</i>						
<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OBoost</u>	<u>VFDT</u>	<u>HTM</u>
1.93	4.04	3.29	5.14	4.71	2.43	6.43

Table 16 shows the resulting Nemenyi p-values. Post-hoc p-values of all possible pairs (classifiers) are compactly represented as a lower triangular matrix. Each numerical entry is the p-value of row/column pair, i.e., the null hypothesis that the classifier represented by its particular column name is different from the classifier represented by its particular row name.

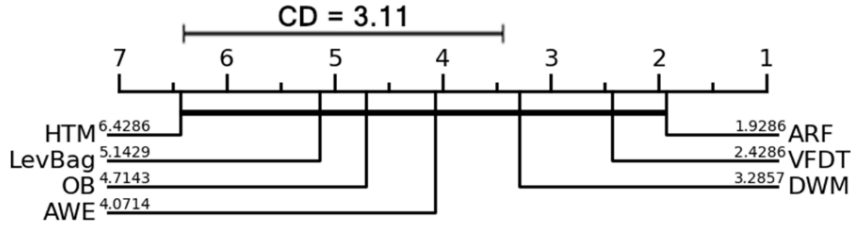


Figure 28. Nemenyi test results with a 90% confidence level.

Table 16

*Nemenyi p-values, with no further adjustment*

	<u>ARF</u>	<u>AWE</u>	<u>DWM</u>	<u>LevBag</u>	<u>OB</u>	<u>VFDT</u>
AWE	0.51	-	-	-	-	-
DWM	0.9036	0.9937	-	-	-	-
LevBag	0.0789	0.9682	0.677	-	-	-
OB	0.1931	0.9979	0.8797	1	-	-
VFDT	0.9995	0.7897	0.9899	0.22	0.43	-
HTM	0.0019	0.3884	0.0928	0.92	0.75	0.01

Note: Nemenyi p-values computed using tools from <https://astatsa.com/FriedmanTest/>

### Comparison to Other Literature

This research followed that of Ghomeshi et al. (2019), who developed a classifier based on evolutionary algorithms to cope with different types of concept drifts in non-stationary data streams. It is noteworthy that Ghomeshi et al. (2019) generated their artificial data sets and ran their classifier experiments using MOA, which is an open-source framework for data stream mining and is a Java-based service. In contrast, this research utilized the data generation and classifier tools provided by scikit multi-flow, which is a Python add-in package. Tables 24 and 25 below are reprinted from Ghomeshi et al. (2019). Incorporated into Tables 24 and 25 are the results of this research for comparison purposes.

Ghomeshi et al. (2019) classifier accuracy results were similar to that of this research conducted. However, there was a trend for their results to be more in alignment

with this research’ runs that contained 10% noise. Ghomeshi et al. (2019) did not mention inserting noise in their publication.

It was observed that there was a great deal of difference in the average CPU run-times between Ghomeshi et al. (2019) and this work for all experiments conducted. The contrast in results for both classifier accuracy and CPU run-times was likely due to architectural and implementation differences between MOA and scikit multiflow.

**Table 17**

<u>Comparison to Other Research</u>			
<u>Average Accuracy</u>			
<u>Reference</u>	<u>Data Set</u>	<u>Classifier</u>	<u>Accuracy</u>
Ghomeshi et al. (2019)	Hyper.	ARF	88.17
This Research (from Table 3 0% noise)	Hyper.	ARF	94.30
This Research (from Table 4 10% noise)	Hyper.	ARF	86.00
Ghomeshi et al. (2019)	LED	ARF	74.05
This Research (from Table 3 0% noise)	LED	ARF	99.80
This Research (from Table 4 10% noise)	LED	ARF	76.00
Ghomeshi et al. (2019)	RT	ARF	78.24
This Research (from Table 3 0% noise)	RT	ARF	81.40
This Research (from Table 4 10% noise)	RT	ARF	--
Ghomeshi et al. (2019)	SEA	ARF	88.67
This Research (from Table 3 0% noise)	SEA	ARF	99.50
This Research (from Table 4 10% noise)	SEA	ARF	90.40
Ghomeshi et al. (2019)	Airlines	ARF	63.53
This Research (from Table 9)	Airlines	ARF	66.00
Ghomeshi et al. (2019)	Elec	ARF	92.17
This Research (from Table 9)	Elec	ARF	88.00
Ghomeshi et al. (2019)	Poker	ARF	84.19
This Research (from Table 9)	Poker	ARF	54.00
Ghomeshi et al. (2019)	Hyper.	DWM	89.64
This Research (from Table 3 0% noise)	Hyper.	DWM	99.70
This Research (from Table 4 10% noise)	Hyper.	DWM	87.90
Ghomeshi et al. (2019)	LED	DWM	75.05
This Research (from Table 3 0% noise)	LED	DWM	99.98
This Research (from Table 4 10% noise)	LED	DWM	56.20



Ghomeshi et al. (2019)	RT	DWM	59.49
This Research (from Table 3 0% noise)	RT	DWM	60.90
This Research (from Table 4 10% noise)	RT	DWM	--
Ghomeshi et al. (2019)	SEA	DWM	87.72
This Research (from Table 3 0% noise)	SEA	DWM	94.90
This Research (from Table 4 10% noise)	SEA	DWM	88.20
Ghomeshi et al. (2019)	Airlines	DWM	63.97
This Research (from Table 9)	Airlines	DWM	61.00
Ghomeshi et al. (2019)	Elec	DWM	75.73
This Research (from Table 9)	Elec	DWM	80.00
Ghomeshi et al. (2019)	Poker	DWM	74.37
This Research (from Table 9)	Poker	DWM	47.00
Ghomeshi et al. (2019)	Hyper.	LevBag	91.03
This Research (from Table 3 0% noise)	Hyper.	LevBag	77.00
This Research (from Table 4 10% noise)	Hyper.	LevBag	68.20
Ghomeshi et al. (2019)	LED	LevBag	74.22
This Research (from Table 3 0% noise)	LED	LevBag	63.00
This Research (from Table 4 10% noise)	LED	LevBag	40.40
Ghomeshi et al. (2019)	RT	LevBag	90.91
This Research (from Table 3 0% noise)	RT	LevBag	64.80
This Research (from Table 4 10% noise)	RT	LevBag	--
Ghomeshi et al. (2019)	SEA	LevBag	87.59
This Research (from Table 3 0% noise)	SEA	LevBag	93.80
This Research (from Table 4 10% noise)	SEA	LevBag	79.50
Ghomeshi et al. (2019)	Airlines	LevBag	59.42
This Research (from Table 9)	Airlines	LevBag	56.00
Ghomeshi et al. (2019)	Elec	LevBag	92.09
This Research (from Table 9)	Elec	LevBag	85.00
Ghomeshi et al. (2019)	Poker	LevBag	88.52
This Research (from Table 9)	Poker	LevBag	46.00
Ghomeshi et al. (2019)	Hyper.	OBoost	85.85
This Research (from Table 3 0% noise)	Hyper.	OBoost	82.00
This Research (from Table 4 10% noise)	Hyper.	OBoost	69.70
Ghomeshi et al. (2019)	LED	OBoost	74.15
This Research (from Table 3 0% noise)	LED	OBoost	83.70
This Research (from Table 4 10% noise)	LED	OBoost	52.60
Ghomeshi et al. (2019)	RT	OBoost	85.30
This Research (from Table 3 0% noise)	RT	OBoost	67.40
This Research (from Table 4 10% noise)	RT	OBoost	--
Ghomeshi et al. (2019)	SEA	OBoost	85.56
This Research (from Table 3 0% noise)	SEA	OBoost	95.70

This Research (from Table 4 10% noise)	SEA	OBoost	76.70
Ghomeshi et al. (2019)	Airlines	OBoost	61.98
This Research (from Table 9)	Airlines	OBoost	55.00
Ghomeshi et al. (2019)	Elec	OBoost	88.02
This Research (from Table 9)	Elec	OBoost	79.00
Ghomeshi et al. (2019)	Poker	OBoost	84.31
This Research (from Table 9)	Poker	OBoost	47.00

**Table 18**

Comparison to Other Research

*Average CPU run-time (seconds)*

<u>Reference</u>	<u>Data Set</u>	<u>Classifier</u>	<u>CPU run-time</u>
Ghomeshi et al. (2019)	Hyper.	ARF	208.00
This Research (from Table 3 0% noise)	Hyper.	ARF	13776.03
This Research (from Table 4 10% noise)	Hyper.	ARF	22446.87
Ghomeshi et al. (2019)	LED	ARF	188.00
This Research (from Table 3 0% noise)	LED	ARF	29842.33
This Research (from Table 4 10% noise)	LED	ARF	60329.59
Ghomeshi et al. (2019)	RT	ARF	394.00
This Research (from Table 3 0% noise)	RT	ARF	16115.15
This Research (from Table 4 10% noise)	RT	ARF	--
Ghomeshi et al. (2019)	SEA	ARF	751.00
This Research (from Table 3 0% noise)	SEA	ARF	12469.20
This Research (from Table 4 10% noise)	SEA	ARF	24038.47
Ghomeshi et al. (2019)	Airlines	ARF	495.00
This Research (from Table 9)	Airlines	ARF	10527.60
Ghomeshi et al. (2019)	Elec	ARF	7.73
This Research (from Table 9)	Elec	ARF	502.91
Ghomeshi et al. (2019)	Poker	ARF	167.00
This Research (from Table 9)	Poker	ARF	29044.56
Ghomeshi et al. (2019)	Hyper.	DWM	130.00
This Research (from Table 3 0% noise)	Hyper.	DWM	1209.69
This Research (from Table 4 10% noise)	Hyper.	DWM	1425.74
Ghomeshi et al. (2019)	LED	DWM	851.00
This Research (from Table 3 0% noise)	LED	DWM	3792.89
This Research (from Table 4 10% noise)	LED	DWM	8385.59
Ghomeshi et al. (2019)	RT	DWM	195.00
This Research (from Table 3 0% noise)	RT	DWM	4735.79
This Research (from Table 4 10% noise)	RT	DWM	--

Ghomeshi et al. (2019)	SEA	DWM	98.00
This Research (from Table 3 0% noise)	SEA	DWM	843.76
This Research (from Table 4 10% noise)	SEA	DWM	844.74
Ghomeshi et al. (2019)	Airlines	DWM	66.00
This Research (from Table 9)	Airlines	DWM	1529.12
Ghomeshi et al. (2019)	Elec	DWM	1.48
This Research (from Table 9)	Elec	DWM	56.08
Ghomeshi et al. (2019)	Poker	DWM	46.00
This Research (from Table 9)	Poker	DWM	6334.51
Ghomeshi et al. (2019)	Hyper.	LevBag	144.00
This Research (from Table 3 0% noise)	Hyper.	LevBag	28610.06
This Research (from Table 4 10% noise)	Hyper.	LevBag	30333.96
Ghomeshi et al. (2019)	LED	LevBag	246.00
This Research (from Table 3 0% noise)	LED	LevBag	73243.41
This Research (from Table 4 10% noise)	LED	LevBag	69143.31
Ghomeshi et al. (2019)	RT	LevBag	207.00
This Research (from Table 3 0% noise)	RT	LevBag	92246.17
This Research (from Table 4 10% noise)	RT	LevBag	--
Ghomeshi et al. (2019)	SEA	LevBag	409.00
This Research (from Table 3 0% noise)	SEA	LevBag	21311.02
This Research (from Table 4 10% noise)	SEA	LevBag	21844.44
Ghomeshi et al. (2019)	Airlines	LevBag	531.00
This Research (from Table 9)	Airlines	LevBag	14301.71
Ghomeshi et al. (2019)	Elec	LevBag	5.12
This Research (from Table 9)	Elec	LevBag	1217.43
Ghomeshi et al. (2019)	Poker	LevBag	81.00
This Research (from Table 9)	Poker	LevBag	31327.38
Ghomeshi et al. (2019)	Hyper.	OBoost	93.00
This Research (from Table 3 0% noise)	Hyper.	OBoost	50473.21
This Research (from Table 4 10% noise)	Hyper.	OBoost	52635.70
Ghomeshi et al. (2019)	LED	OBoost	174.00
This Research (from Table 3 0% noise)	LED	OBoost	61545.17
This Research (from Table 4 10% noise)	LED	OBoost	98845.17
Ghomeshi et al. (2019)	RT	OBoost	148.00
This Research (from Table 3 0% noise)	RT	OBoost	123796.25
This Research (from Table 4 10% noise)	RT	OBoost	--
Ghomeshi et al. (2019)	SEA	OBoost	162.00
This Research (from Table 3 0% noise)	SEA	OBoost	25690.63
This Research (from Table 4 10% noise)	SEA	OBoost	27593.68
Ghomeshi et al. (2019)	Airlines	OBoost	74.00
This Research (from Table 9)	Airlines	OBoost	20546.30

Ghomeshi et al. (2019)	Elec	OBoost	2.06
This Research (from Table 9)	Elec	OBoost	1624.03
Ghomeshi et al. (2019)	Poker	OBoost	64.00
This Research (from Table 9)	Poker	OBoost	31327.38

## Summary of Results

Experiment I showed that although the HTM classifier's accuracy percentages were in the mid 50% range, the classifier was sensitive to abrupt, gradual, and recurrent concept drift and noise. Furthermore, it can be concluded that even without temporal dependency within the data stream, the HTM classifier demonstrated an ability to learn new patterns relatively quickly. These observations suggest that despite the lack of temporal dependencies within the SEA data stream, the HTM classifier showed an ability to learn new patterns and adjust to concept drift, and noise within artificial data streams.

The results of experiment II demonstrated the HTM classifier suitable for classifying real-world data in two out of the three cost measures (i.e., average accuracy and CPU run-time). However, the high cost in memory usage, which coincided with that of experiment I, is something that must be considered.

Experiment III successfully identified the behavioral tendencies of the Rombertik virus through the machine states that resulted from executing its code. The plots derived from tracing the HTM classifiers' performance on all eight flag bits demonstrate a new tool that can aid reverse engineers of malware to identify whether an executable contains stalling code.

## Chapter 5

### Conclusions, Summary, Implications, & Recommendations

This research demonstrated an HTM sequence classifier capable of classifying data within a data stream containing concept drift, noise, and temporal dependencies. The HTM sequence classifier was evaluated on both artificially generated data streams (e.g., Hyperplane, LED, Random Tree, and SEA) and real-world data streams (e.g., Airlines, Electricity, and Poker), and compared its performance against several modern classifiers (e.g., ARF, AWE, DWM, LevBag, Online-Boost, and VFDT) using the cost metrics of average accuracy percentage, CPU run-time, and RAM usage. Additionally, this research evaluated the potential of an HTM classifier for malware analysis via a simulated example of the Rombertik stalling code within an executable file.

#### Conclusions

The HTM classifier proved effective in detecting abrupt, gradual and recurrent concept drift within artificial data streams, including those with 10% noise as shown in Figures 7, 8, 9 and 10 of experiment I. However, the HTM classifiers' accuracy metric was poor due to the data streams lacking temporal dependencies as shown in Tables 3 and 4.

The application of noise to the artificial data streams provided surprising results in that it was unexpected to see the significant drop in the accuracy of many of the other classifiers. The OBoost classifier suffered the most significant reduction in accuracy from 83.71% down to 52.6%, while LevBag dropped from 62.96% down to 40.41% for the LED data stream (Table 4). Meanwhile, the HTM classifier remained relatively stable at

50%. The HTMs' stable performance could imply that the HTM classifier exhibits a behavior that is resistant to noise.

Experiment II showed that the HTM classifier performed reasonably well when predicting real-world data using the cost measures of average accuracy and average CPU run-time; however, the HTM classifiers' average RAM usage was exceptionally high.

Experiment III showed that the HTM classifier is well suited to predicting machine states (i.e., CPU Flags), as shown in Figures 12 – 27. The HTM classifier was capable of producing plotting graphs, similar to that of an electrocardiogram, with a high accuracy percentage, thereby providing a valuable analysis tool in malware analysis.

## **Summary**

The primary purpose of this research was to develop a new sequence classifier that can classify data in a data stream that contained concept drift, noise, and temporal dependencies. This research demonstrated that a sequence classifier based on HTM architecture achieved this goal but is better suited to data streams that contain temporal dependencies.

As Bifet et al. (2017) pointed out, most streaming environments contain temporally related data that, during certain periods, their labels correlate (i.e., network attack and intrusion detection). However, despite recent research that advanced the discovery of novel cyber-attacks such as that conducted by Burgio (2019), there still exists a gap in the published literature on streams that contain temporal dependencies. This research demonstrated that an HTM based approach is applicable toward classifying data within data streams that contain temporalness as experiments I, II, and III show. This research also showed that the HTM based approach is capable of detecting concept drift.

Finally, the secondary goal of this research was to explore the potential of HTM as a solution for detecting evasive malware that contains stalling code. As demonstrated in experiment III, this research generated a sample executable that contained stalling code based on the Rombertik virus, and successfully analyzed the live executable via a combination of an IDA Pro Python script and an HTM classifier that generated an electrocardiography style plot that can be utilized to study and identify behavior patterns of programs.

An implied goal of this research was to determine the fitness of HTM technologies as a classifier capable of classifying all data streams. As experiments I and II demonstrated, the HTM classifier is more suitable for temporal data streams than non-temporal data streams. This determination came by using the four-part classifier evaluation framework, described by Bifet et al. (2017). The first part, error estimation, was accomplished using the prequential approaches for error estimation.

The fourth part, identifying cost measures, identified classifier accuracy, CPU time, and RAM usage to measure the HTM sequence classifiers' performance against the ARF, AWE, DWM, LevBag, Online-Boost, and VFDT classifiers.

### ***Experiment I***

Showed that HTMs are not suitable for classifying artificial data streams; however, it did prove that the HTM classifier is capable of detecting concept drift. It also highlights the fact that many efficient classifiers already exist for classifying artificially generated data that contain concept drift. Finally, experiment I showed that with the introduction of 10% noise, the test set of classifiers' performance in accuracy, CPU-time, and RAM-usage began to degrade sharply.

### ***Experiment II***

Results showed the HTM classifier was capable of classifying data within data streams containing temporal data. While the accuracy results of experiment II were average in comparison to the other classifiers, it was encouraging to see that this technology is sensitive to temporal data streams within real-world data. As such, this approach offers a new option for classifying streaming data that contain temporal dependencies.

### ***Experiment III***

Experiment III outlined a process for using HTM technology in dynamic malware analysis. This experiment demonstrated the potential of HTMs in analyzing malware with the specific behavior of utilizing stalling code as a means of avoiding detection. This experiment compared the simulated stalling executable against a normal sorting executable. Comparing the EFLAG graphs of the two algorithms showed the stalling code to behave very predictably, leading to the HTM classifier to obtain high accuracy predictions quickly. The resulting plots showed that the Rombertik stalling code flatlined the EFLAGS, making it very recognizable.

### **Implications**

Given the ever-increasing level of temporal dependencies that exist within real-world data streams, the need for classifiers that can find and apply temporal dependencies when labeling data quickly would be of great benefit. Furthermore, the effectiveness of evasive malware at evading detection by IDS through the use of simple tactics such as stalling code forces classifiers to frequently retrain, resulting in a degradation of performance and an increase in the consumption of system resources. However, as discussed, finding temporal dependencies within real-world data streams (including



network traffic), could allow IDS and classifiers to avoid performance degradation and consumption of system resources. Thus, the use of HTM technologies in finding temporal dependencies might help to improve the effectiveness of these applications.

### **Recommendations**

While this research has highlighted the potential of HTM based classifiers in classifying real-world streaming data, there is ample opportunity to extend this research to improve the HTM classifiers' accuracy and for use in multiple machine learning domains. The HTM classifiers' accuracy is highly dependent upon modifying the parameters for the Temporal Memory and Spatial Pooler, along with highly customized encoders unique to each data stream. One such example was the simple attempt by this research to create a geospatial encoder for use in the Airline dataset. Improving the robustness and quality of the geospatial encoder while maintaining the four essential SDR properties, as described by Purdy (2015), would be useful in real-world data streams that contain features for geospatial locations.

In the domain of malware analysis, finding labeled malware stalling examples does not exist. Researchers must individually reverse engineer known malware executables to search for the existence of stalling code, which is time-intensive. The lack of labeled stalling code will offer an opportunity to extend experiment III by labeling real-world malware samples from online repositories such as Virusshare.com for the presence of stalling code. The labeling of this repository for the existence of stalling code would benefit evasive malware researchers by providing a rich pool of malware examples without the need for individually reverse engineering each one.

Currently, standard statistical functions (e.g., ROC, AUC, Friedmans' Test) for HTM's do not exist. Common statistical software applications (e.g., R, MATLAB, SAS,

Python, Java) contain a rich repository of statistical analytical tools that would be useful in evaluating the performance of HTM classifiers. Adding HTM statistical functions to any of these applications would allow the researcher to focus on improving the performance of HTMs rather than concentrating on designing customized methods for producing common statistical measurements and graphs.

Another avenue of inquiry would be to allow for the provision of advanced parallelism of HTM. Parallelism could help in improving the amount of time needed for the Temporal Memory algorithm to learn sequential patterns. Applications such as <https://dask.org/> could provide the necessary infrastructure to parallelize HTMs. Finally, my research demonstrated the potential of HTM in the implementation of machine learning and data mining.

## Appendix A

### Hyperplane Features Description for Experiment I (10 files ea. 1 million data pts)

(Hulten et al., 2001)

Table 19

*Hyperplane Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	n_features	int (Def: 10)	The number of attributes to generate. Higher than 2.
2	n_drift_features	int (Def: 2)	The number of attributes with drift. Higher than 2.
3	mag_change	float (Def: 0.0)	Magnitude of the change for every example. 0.0 to 1.0
4	noise_%	float (Def: 0.05)	% of noise to add to the data. 0.0 to 1.0
5	sigma_%	int (Def: 0.1)	% of prob.that the direction of change is reversed. 0.0 to 1.0

## Appendix B

### LED Features Description for Experiment I (10 files ea. 1 million data pts)

(Breiman et al., 1984)

Table 20

*LED Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	random_state	int, RandomState instance or None, optional (default=None)	If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by <i>np.random</i> .
2	noise_percentage	float (Default: 0.0)	The probability that noise will happen in the generation. At each new sample generated, a random probability is generated, and if that probability is equal or less than the noise_percentage, the selected data will be switched
3	has_noise	bool (Default: False)	Adds 17 non relevant attributes to the stream.

## Appendix C

### Random Tree Features Description for Experiment I (10 files ea. 1 million data pts)

[https://scikit-](https://scikit-multiflow.readthedocs.io/en/stable/api/generated/skmultiflow.data.RandomTreeGenerator.html#skmultiflow.data.RandomTreeGenerator)

[multiflow.readthedocs.io/en/stable/api/generated/skmultiflow.data.RandomTreeGenerator.html#skmultiflow.data.RandomTreeGenerator](https://scikit-multiflow.readthedocs.io/en/stable/api/generated/skmultiflow.data.RandomTreeGenerator.html#skmultiflow.data.RandomTreeGenerator)

Table 21

*Random Tree Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	tree_random_state	int (Default: None)	Seed for random generation of tree.
2	sample_random_state	int (Default: None)	Seed for random generation of instances.
3	n_classes	int (Default: 2)	The number of classes to generate.
4	n_cat_features:	int (Default: 5)	The number of categorical features to generate. Categorical features are binary encoded, the actual number of categorical features is $n\_cat\_features \times n\_categories\_per\_cat\_feature$
5	n_num_features	int (Default: 5)	The number of numerical features to generate.
6	n_categories_per_cat_feature	int (Default: 5)	The number of values to generate per categorical feature.
7	max_tree_depth	int (Default: 5)	The maximum depth of the tree concept.
8	min_leaf_depth	int (Default: 3)	The first level of the tree above MaxTreeDepth that can have leaves.
9	fraction_leaves_per_level	float (Default: 0.15)	The fraction of leaves per level from min_leaf_depth onwards.

## Appendix D

### SEA Features Description for Experiment I (10 files ea. 1 million data pts)

(Street & Kim, 2001)

Table 22

*SEA Feature List*

---

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	classification_function	int (Default: 0)	Which of the four classification functions to use for the generation. This value can vary from 0 to 3, and the thresholds are, 8, 9, 7 and 9.5.
2	random_state:	int, RandomState instance or None, optional (default=None)	If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by <i>np.random</i> .
3	balance_classes	bool (Default: False)	Whether to balance classes or not. If balanced, the class distribution will converge to a uniform distribution.
4	noise_percentage	float (Default: 0.0)	The probability that noise will happen in the generation. At each new sample generated, a random probability is generated, and if that probability is higher than the noise percentage, the chosen label will be switched. From 0.0 to 1.0.

---

## Appendix E

### Electricity Features Description for Experiment II (45,312 instances)

(Harries et al., 2003)

Table 23

*Electricity Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	class (target)	nominal	change of the price (UP or DOWN) (1 or 0)
2	date	numeric	date between 7 May 1996 to 5 December 1998. Here normalized between 0 and 1
3	day	numeric	day of the week (1-7)
4	period	numeric	time of the measurement (1-48) in half hour intervals over 24 hours. Here normalized between 0 and 1
5	nswprice	numeric	New South Wales electricity price, normalized between 0 and 1
6	nswdemand	numeric	New South Wales electricity demand, normalized between 0 and 1
7	vicprice	numeric	Victoria electricity price, normalized between 0 and 1
8	vicdemand	numeric	Victoria electricity demand, normalized between 0 and 1
9	transfer	numeric	scheduled electricity transfer between both states, normalized between 0 and 1

## Appendix F

### Airlines Features Description for Experiment II (539,381 instances)

<http://stat-computing.org/dataexpo/2009/>

Table 24

#### *Airlines Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	Year	int (normalized)	1987-2008
2	Month	int (normalized)	12-Jan
3	DayofMonth	int (normalized)	31-Jan
4	DayOfWeek	int (normalized)	1 (Monday) - 7 (Sunday)
5	DepTime	int (normalized)	actual departure time (local, hhmm)
6	CRSDepTime	int (normalized)	scheduled departure time (local, hhmm)
7	ArrTime	int (normalized)	actual arrival time (local, hhmm)
8	CRSArrTime	int (normalized)	scheduled arrival time (local, hhmm)
9	UniqueCarrier	int (normalized)	<u>unique carrier code</u>
10	FlightNum	int (normalized)	flight number
11	TailNum	int (normalized)	plane tail number
12	ActualElapsedTime	int (normalized)	in minutes
13	CRSElapsedTime	int (normalized)	in minutes
14	AirTime	int (normalized)	in minutes
15	ArrDelay	int (normalized)	arrival delay, in minutes
16	DepDelay	int (normalized)	departure delay, in minutes
17	Origin	int (normalized)	<u>origin IATA airport code</u>
18	Dest	int (normalized)	<u>destination IATA airport code</u>
19	Distance	int (normalized)	in miles
20	TaxiIn	int (normalized)	taxi in time, in minutes
21	TaxiOut	int (normalized)	taxi out time in minutes
22	Cancelled	int (normalized)	was the flight cancelled?
23	CancellationCode	int (normalized)	reason for cancellation (A = carrier, B = weather, C = NAS, D = security)
24	Diverted	int (normalized)	1 = yes, 0 = no
25	CarrierDelay	int (normalized)	in minutes
26	WeatherDelay	int (normalized)	in minutes
27	NASDelay	int (normalized)	in minutes
28	SecurityDelay	int (normalized)	in minutes
29	LateAircraftDelay	int (normalized)	in minutes



## Appendix G

### Poker Features Description for Experiment II (999,999 instances)

<https://archive.ics.uci.edu/ml/datasets/Poker+Hand>

Table 25

*Poker Feature List*

<u>No.</u>	<u>Name</u>	<u>Type</u>	<u>Description</u>
1	S1 "Suit of card #1"	Ordinal (1-4)	{Hearts, Spades, Diamonds, Clubs}
2	C1 "Rank of card #1"	Numerical (1-13)	(Ace, 2, 3, ... , Queen, King)
3	S2 "Suit of card #2"	Ordinal (1-4)	{Hearts, Spades, Diamonds, Clubs}
4	C2 "Rank of card #2"	Numerical (1-13)	(Ace, 2, 3, ... , Queen, King)
5	S3 "Suit of card #3"	Ordinal (1-4)	{Hearts, Spades, Diamonds, Clubs}
6	C3 "Rank of card #3"	Numerical (1-13)	(Ace, 2, 3, ... , Queen, King)
7	S4 "Suit of card #4"	Ordinal (1-4)	{Hearts, Spades, Diamonds, Clubs}
8	C4 "Rank of card #4"	Numerical (1-13)	(Ace, 2, 3, ... , Queen, King)
9	S5 "Suit of card #5"	Ordinal (1-4)	{Hearts, Spades, Diamonds, Clubs}
10	C5 "Rank of card 5"	Numerical (1-13)	(Ace, 2, 3, ... , Queen, King)
11	CLASS "Poker Hand"	Ordinal (0-9)	"Poker Hand" 0: Nothing in hand; not a recognized poker hand 1: One pair; one pair of equal ranks within five cards 2: Two pairs; two pairs of equal ranks within five cards 3: Three of a kind; three equal ranks within five cards 4: Straight; five cards, sequentially ranked with no gaps 5: Flush; five cards with the same suit 6: Full house; pair + different rank three of a kind 7: Four of a kind; four equal ranks within five cards 8: Straight flush; straight + flush 9: Royal flush; {Ace, King, Queen, Jack, Ten} + flush

## Appendix H

### SDR Encoder Dictionaries

#### *Hyperplane*

```
'encoderDictionary': {  
  "scalerXY":  
    {'size': 100, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 10, 'activeBits': 10},  
  "category":  
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},  
},
```

#### **LED**

```
'encoderDictionary': {  
  "scalerXY":  
    {'size': 10, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},  
  "category":  
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 9, 'activeBits': 5},  
},
```

#### **RT**

```
'encoderDictionary': {  
  "scalerInt":  
    {'size': 10, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},  
  "scalerFloat":  
    {'resolution': 0.88, 'size': 700, 'sparsity': 0.02},  
  "category":  
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 9, 'activeBits': 5},  
},
```

#### **SEA**

```
'encoderDictionary': {  
  "scalerXY":  
    {'size': 100, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 10, 'activeBits': 10},  
  "category":  
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},  
},
```

#### *Airlines*

```
'encoderDictionary': {  
  "airline":  
    {'size': 128, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 17, 'activeBits': 4},  
  "flight_no":  
    {'size': 128, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 1, 'maximum': 7814, 'activeBits': 4},  
  "lat":  
    {'size': 1024, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 13, 'maximum': 72, 'activeBits': 8},  
},
```

```

      "long":
        {'size': 1024, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': -177, 'maximum': 145, 'activeBits':
8},
      "alt":
        {'size': 1024, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': -54, 'maximum': 9070, 'activeBits':
8},
      "airport_size":
        {'size': 16, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 2, 'activeBits': 5},
      "length_of_flight":
        {'size': 2048, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 655, 'activeBits': 5},
      "category":
        {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},
      "time":
        {'timeOfDay': (30, 1), 'weekend': 21}
    }

```

### ***Electricity***

```

'encoderDictionary': {
  "scalerXY":
    {'size': 2048, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 10, 'activeBits': 10},
  "category":
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 5},
  "time":
    {'timeOfDay': (30, 1), 'weekend': 21}
},

```

### ***Poker***

```

'encoderDictionary': {
  "suit":
    {'size': 128, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 1, 'maximum': 4, 'activeBits': 5},
  "rank":
    {'size': 128, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 1, 'maximum': 13, 'activeBits': 3},
  "category":
    {'size': 0, 'radius': 0, 'category': 1, 'resolution': 0, 'minimum': 0, 'maximum': 9, 'activeBits': 5}
},

```

### ***Executables***

```

'encoderDictionary': {
  "register":
    {'size': 1024, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 1000000, 'maximum': 4967295,
'activeBits': 3},
  "flag":
    {'size': 16, 'radius': 0, 'category': 0, 'resolution': 0, 'minimum': 0, 'maximum': 1, 'activeBits': 2},
},

```

# Appendix I

## Real-World Dataset Spatial Pooler, and Temporal Memory Specifications

```
'predictor': {'sdrc_alpha': 0.0005},

'sp': {'boostStrength': 9.0,
       'columnCount': 8192,
       'localAreaDensity': 0.04395604395604396,
       'potentialPct': 0.85,
       'synPermActiveInc': 0.04,
       'synPermConnected': 0.13999999999999999,
       'synPermInactiveDec': 0.006},

'tm': {'activationThreshold': 10,
       'cellsPerColumn': 64,
       'initialPerm': 0.21,
       'maxSegmentsPerCell': 2048,
       'maxSynapsesPerSegment': 256,
       'minThreshold': 5,
       'newSynapseCount': 32,
       'permanenceDec': 0.1,
       'permanenceInc': 0.1},

'anomaly': {
  'likelihood':
    { 'probationaryPct': 0.1,
      'reestimationPeriod': 100
    } # These settings are copied from NAB
}
```

## Appendix J

### Simulated Malware using Stalling Code

<https://www.lastline.com/labsblog/exposing-rombertik-turning-the-tables-on-evasive-malware/>

```
*****
;
; Program Name: stall.asm
; Programmer: Jeffrey V. Barnett
; Class: PH D Dissertation
; Date: February 12, 2017
; Purpose:
; Simulate Stalling Code
*****
.486
.model flat, stdcall
.stack 100h

ExitProcess PROTO Near32 stdcall, dwExitCode:dword ; capitalization not necessary
; capitalization not necessary

.data ; this is the data area
iX dword 0
ptrB dword iX

sY word 4 dup(5,22)
bVal byte 'GEORGE WASHINGTON'
qVal qword -55
iY dword 3 dup(25,67)

.code ; this is the code area
_start:
mov ebx, ptrB

loc_4EEFD2:
mov edx, edx
inc dword ptr [ebx]
cmp dword ptr [ebx], 1C9C381h
jnz short loc_4EEFD2
xor eax, eax
mov [ebx], eax

loc_4EEFE2:
mov ecx, ecx
inc dword ptr [ebx]
cmp dword ptr [ebx], 376EAC81h
jnz short loc_4EEFE2

INVOKE ExitProcess,0
PUBLIC _start
END

end
```

## Appendix K

### Assembly Language Sorting Algorithm

<https://github.com/beckysag/masm-random-integers/blob/master/prog05.asm>

```
TITLE Prog05.asm)

; Original Author: Rebecca Sagalyn
; Modified by Jeff Barnett 02/08/2020
; Course / Project ID: CS271 #05           Date: 2/28/13
; Description:
; 1. Introduce the program.
; 2. Get a user request in the range [min=10 .. max=200].
; 3. Generate request random integers in the range [lo=100 ..
hi=999], storing them in consecutive elements of an array.
; 4. Display the list of integers before sorting, 10 numbers per
line.
; 5. Sort the list in descending order (i.e., largest first).
; 6. Calculate and display the median value, rounded to the nearest
integer.
; 7. Display the sorted list, 10 numbers per line.

;*****
*****
;*                                     SHARED
DATA                                     *
;*****
*****
        .486
        .model flat, stdcall
        .stack 1000h

        includelib ..\..\Irvine\kernel32.lib
        include ..\..\Irvine\Irvine32.inc
        include ..\..\Irvine\VirtualKeys.inc
        include ..\..\Irvine\macros.inc
        includelib ..\..\Irvine\User32.lib
        includelib ..\..\Irvine\Irvine32.lib

; Win32 Console handle
;STD_OUTPUT_HANDLE EQU -11 ; predefined Win API constant
(magic)
;FILE_ATTRIBUTE_NORMAL equ 80h
;OPEN_EXISTING EQU 3 ;Parameter for opening an existing file
;NULL EQU 0
; get standard handle
; type of console handle
GetStdHandle PROTO, nStdHandle:DWORD
```

```

ExitProcess PROTO Near32 stdcall, dwExitCode:dword ;
capitalization not necessary
GetProcessHeap          PROTO
HeapAlloc                PROTO,mHeap:HANDLE, dwFlags:DWORD,
dwBytes:DWORD ; number of bytes to allocate
HeapFree                PROTO,mHeap:HANDLE,
dwFlags:DWORD,lpMem:DWORD

; capitalization not necessary
local_1 EQU DWORD PTR [ebp-4]
local_2 EQU DWORD PTR [ebp-8]
local_3 EQU DWORD PTR [ebp-12]
MIN = 10
MAX = 400
LO = 100
HI = 999

.data
introl          BYTE    "Sorting Random Integers
Original Becky Sag, Modified by Jeff Barnett",02h,0Dh,0Ah,0Ah
                BYTE    "This program generates 999
random numbers in the range [100 .. 999], ",0Dh, 0Ah
                BYTE    "displays the original list,
sorts the list, and calculates the median value.",0Dh, 0Ah
                BYTE    "Finally, it displays the list
sorted in desc order.",0Dh,0Ah,0Dh,0Ah,0
prompt         BYTE    0Dh, 0Ah,"How many numbers should be
generated? [100 .. 999]: ",0
err_str        BYTE    "Invalid input",0
strUn          BYTE    "The unsorted random numbers:",0
SIZE_UN= ($ - strUn)
prompt2        BYTE    "The median is: ",0
strSort        BYTE    "The sorted list:",0
SIZE_SORT = ($ - strSort)
request        DWORD    ?
iarr           DWORD    MAX    DUP(?)
lf             DWORD    ?
rt            DWORD    ?
outHandle      DWORD    ?
; handle to standard

console output device
fHandle        DWORD    ?
; handle to output file

fname          BYTE
"C:\Users\barnettjv\Desktop\numbers.txt",0 ; file
name

buff           BYTE 5 DUP(0)
; buffer pointer

hHeap         HANDLE ?
pArray        DWORD ? ; pointer to array

;*****
*****

```

```

;*
PROCEDURES
;*****
*****

.code

;-----
-----
main PROC
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
        ; init handle
    mov     [outHandle], eax
        ; store handle in outHandle

; Intro
    push   OFFSET introl
        ; @introl
    call   intro
        ; Introduce program

; Get number from user, store in request

    push   OFFSET err_str
        ; @err_str
    push   OFFSET prompt
        ; @prompt
    push   OFFSET request
        ; @request
    call   getData
        ; Get user data
    ;exit

; Generate random numbers into file, one per line

    call   Randomize
    push   OFFSET buff
        ; @buff

    push   OFFSET fhandle
        ; @fhandle
    push   OFFSET fname
        ; @fname (file to write to)
    push   request
        ; request
    call   writeNums

; Read numbers from file into array

    INVOKE GetProcessHeap
    .IF eax == NULL ; cannot get handle
        jmp finished
    .ELSE
        mov hHeap,eax ; handle is OK
    .ENDIF

    INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 900000
    .IF eax == NULL

```



```

        mWrite "HeapAlloc failed"
        jmp finished
    .ELSE
        mov pArray,eax
    .ENDIF
    ;mallocptrHeap, ecx                ; Allocate space
for our new string
    push  OFFSET buff
        ; @buff
    push  pArray
        ; @arr
    push  OFFSET fname
        ; @fname (file to write to)
    push  request
        ; request
    call  readNums
; Display unsorted list
    ;push  OFFSET strUn
    ;push  pArray
    ;push  request
    ;call  displayList
; Sort List
    ;push  pArray
        ; @arr
    ;push  request
        ; request
    ;call  sortList
; Display sorted list
    push  OFFSET strSort
    push  pArray
    push  request
    call  displayList

; Display median
; Calculate and display the median value, rounded to the nearest
integer.
finished:
    ;INVOKECloseHandle, fHandle
    ;close file handle
main ENDP

```

```

;-----
;-----
intro PROC
; Introduces program and programmer, and describes program.
; Receives: [ebp+8] = @intro1
; Returns: nothing
; Proconditions: none
; Registers changed: none
;-----
;-----
    push  ebp
    mov   ebp, esp

```

```

        push    edx
        mov     edx, [ebp+8]
                ; introduce program & programmer
        call   WriteString
        pop     edx
        pop     ebp
        ret     4
intro ENDP

```

```

;-----
-----
getData PROC
; Prompts user to enter number of integers, in range [min..max],
then validates number
; Receives: [ebp+8] = @request, [ebp+12] = @prompt, [ebp+16] =
@err_str
; Returns: number entered by user in request
; Proconditions:none
; Registers changed: none
;-----
---
```

```

        push    ebp
        mov     ebp, esp
        pushad
        mov     edi, [ebp+8]
                ; @request in edi

```

```

RequestNum:
        mov     edx, [ebp+12]
                ; edx = @prompt
        ;call   WriteString
                ; tell user to enter a number
        ;call   ReadDec
                ; save number in eax
        mov     eax, MAX

```

```

; verify: request >= MIN && request <= MAX
        cmp     eax, MIN

        jl     InvalidRequest
                ; if request < MIN, reprompt
        cmp     eax, MAX
        jg     InvalidRequest
                ; if request > MAX, reprompt
        jmp     ValidRequest
                ; else, continue

```

```

InvalidRequest:
        mov     edx, [ebp+16]
                ; edx = @err_str
        call   WriteString
        call   Crlf

```

```

        jmp             RequestNum
                                ; re-prompt

ValidRequest:
        mov             [edi], eax
                                ; store number in request

        popad
        pop             ebp
        ret             12
getData ENDP
;-----
---

;-----
---
DecToASCII PROC
; input = 3 digits dec number
; buff = 4 byte array of BYTES
; uses: eax, ebx, ecx, edx, esi, ebp
; reg changed: eax, ebx, edx, esi
;-----
---
        push     ebp
        mov     ebp, esp
        pushad

        mov     ax, [ebp+8]
ebx = dec num
        mov     edx, [ebp+12]
edx
; @buff in
        mov     ecx, 3
loop counter
L1:
; loop sets buff[0], buff[1], buff[2]
        mov     bl, 10
bl = 10
        div     bl
; AH = digit (7), AL = quotient (65)
        mov     bl, ah
bl = ah = 7
        add     bl, 48
bl = ascii form of digit
        mov     ah, 0
ax = 65 (for next DIV instruction)
        mov     [edx+ecx-1], bl
buff[ecx] = ascii digit
        loop    L1

; set buff[3], buff[4]
        mov     al, 13

```

```

        mov     [edx+3], al
        mov     al, 10
        mov     [edx+4], al

        popad
        pop     ebp
        ret     8
DecToASCII ENDP
;-----
---

;-----
---
readNums PROC
; Description: Read numbers from file into array
; Receives: ebp+8 = request, ebp+12 = @fname, ebp+16 = @arr
; Returns: arr, with request numbers read from file
; Proconditions: numbers are written one per line in file [fname]
;                numbers are all 3 digits (leading
zeros if needed)
;                console handle has been initialized
; Registers changed: none
;-----
---
        LOCAL  pFname:DWORD
        LOCAL  filehandle: DWORD
        pushad
        mov     eax, [ebp+12]
                ; @fname
        mov     pFname, eax
                ; @fname
        mov     esi, [ebp+16]
                ; @arr

        INVOKE CreateFile,
                ; open file [fname] for reading
                pFname, GENERIC_READ,
                DO_NOT_SHARE, NULL, OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL, 0
        mov     filehandle, eax
                ; store file handle in fhandle

        mov     ecx, [ebp+8]
                ; init ecx (loop counter) to request
L1:
                ; for i = request, i >0, i--
        pushad
        INVOKE ReadFile,
                ; Read number from file into buffer
                filehandle,
                ; handle
                [ebp+20],
                ; buffer pointer

```

```

        5,
                                ; number of bytes to read
        NULL,
                                ; num bytes read
        0
                                ; overlapped execution flag

    popad

    mov     edi, [ebp+20]
        ; edi = @buff
    mov     eax, 0
    ; get hundreds digit into edx
    mov     al, [edi]
    sub     al, 48
        ; hundreds digit
    mov     bl, 100
    mul     bl
        ; eax = hundreds digit *
100
    mov     edx, eax
        ; store in edx
    inc     edi
    ; get tens digit into edx
    mov     eax, 0
    mov     al, [edi]
    sub     al, 48
        ; eax = digit in tens place
    mov     bl, 10
    mul     bl
        ; eax = tens digit * 10
    add     edx, eax
        ; add to edx, edx = first two
digits
    inc     edi
    ; get ones digit into edx
    mov     al, [edi]
    sub     al, 48
        ; eax = digit in ones place
    add     edx, eax
        ; add to edx (edx = all 3
digits)
    ; store in array
    mov     [esi], edx
    add     esi, 4
        ; esi points to next array
element
    loop   L1
        ; loop

    INVOKE CloseHandle, fHandle
    ;close file handle
    popad

```

```

        ret 16
readNums ENDP
;-----
---

;-----
---
writeNums PROC
; Description: Generate random numbers and write to file, one number
per line
; Receives: ebp+8 = request, ebp+12 = @fname, ebp+16 = @fhandle,
ebp+20 = @buff
; Returns: none
; Proconditions: request != null
;
; console handle has been initialized
; Registers changed: none
;-----
---

        push    ebp
        mov     ebp, esp
        sub     esp, 4
                ; make space for 1 local var
        push    esi
        push    eax
        push    ebx
        push    ecx
        push    edx
        mov     esi, [ebp+16]
        ; store @fhandle in esi
        mov     ecx, [ebp+8]
        ; store request in ecx (loop counter)

        ; range = hi - lo + 1
        mov     eax, HI
        sub     eax, LO
        inc     eax
                ; eax = hi - lo + 1
        mov     local_1, eax
        ; store "range" in local1

        push    ecx
                ; save ecx before Win API function
        INVOKE CreateFile,
        ; create/overwrite file [fname] for writing
                [ebp+12], GENERIC_WRITE,
                DO_NOT_SHARE, NULL, OPEN_ALWAYS,
                FILE_ATTRIBUTE_NORMAL, 0
        pop     ecx
        mov     [esi], eax
                ; store file handle in fhandle

L1:
                ; while count < request

```

```

mov          eax, local_1
; eax = range
call   RandomRange
; random num in eax

push [ebp+20]
; @buff
push eax
; random number
call DecToASCII
; convert random num to ascii digits

push  ecx
; save ecx before Win API function
INVOKE WriteFile,
; write to file
    [esi],
    ; file handle
    [ebp+20],
    ; buffer pointer
    5,
    ; number of bytes to write
    NULL,
    ; num bytes written
    0
    ; overlapped execution flag
pop      ecx
loop    L1
; sub 1 from ecx, or leave if ecx == 0
LeaveArr:
INVOKE CloseHandle, fHandle
pop      edx
pop      ecx
pop      ebx
pop      eax
pop      esi
mov      esp, ebp
; reset esp, remove local var
pop      ebp
ret      16
; clean up 4 32-bit variables

writeNums ENDP
;-----
---

;-----
---

sortList PROC
; Description:
; Receives: 2 params
; Returns:
; Proconditions:
; Registers changed:

```

```

;-----
---
    push    ebp
    mov     ebp, esp
    push    esi
    push    eax
    push    ebx
    push    ecx
    mov     edi, esp                ;
edi points to location before adding array

    ;mov     esi, [ebp+12]          ; store @arr in
esi
    mov     ebx, [ebp+8]          ; store
request in ebx (size)
    mov     eax, ebx                ;
move size to eax
    mov     ecx, 4                ;
move 4 to ecx
    mul     ecx
    ; multiple array size * 4 to get total size
    add     eax, 4                ;
space for 1 more DWORD variable
    sub     esp, eax                ;
make space for array
    ; now edi - 4 is single variable ; edi - eax is start of
array
    mov     esi, esp                ;
esi points to start of result array

    push    [ebp+12]              ; @arr
    push    0                      ; start
    push    [ebp+8]                ; size
    push    esi                    ; @result
    call   merge

    mov     esp, edi
    pop     ecx
    pop     ebx
    pop     eax
    pop     esi
    ;mov     esp, ebp              ; reset esp,
remove local var
    pop     ebp
    ret     8
sortList ENDP
;-----
---
;-----
---
merge PROC
;-----
---
```



```

;push  ebp                                ; these two lines done
by OS
;mov   ebp, esp                            ;
LOCAL  left:DWORD
LOCAL  right:DWORD
LOCAL  i:DWORD
LOCAL  len:DWORD
LOCAL  dist:DWORD
LOCAL  r:DWORD
LOCAL  l:DWORD                            ; l and r are to the positions
in the left and right subarrays
push  eax
push  ebx
push  edi
push  ecx
push  edx
push  esi

; ebp + 8 = @result
; ebp + 12 = right
; ebp + 16 = left
; ebp + 20 = @arr

edi   mov     edi, [ebp+20]                ; store @arr in
edi   mov     esi, [ebp+8]                 ; store @result
in esi
      mov     eax, [ebp+12]
      mov     right, eax                    ; store
right
      mov     eax, [ebp+16]
      mov     left, eax                     ; store
left
      mov     l, eax                        ; l = left

; base case: one element (if r == l+1, return)
      add     eax, 1                        ; add 1 to left
(in eax)
      cmp     eax, right
      je     LeaveProc                     ; if left+1 =
right, exit

; else
; set len = right - left
      mov     eax, right
      sub     eax, left                    ; right - left in
eax
      mov     len, eax                     ; length = right
- left

; set dist = right - left / 2
      mov     ebx, 2                        ; ebx = 2
      mov     edx, 0

```

```

        div            ebx                ; eax =
(right - left) / 2
        mov            dist, eax        ; dist = (right -
left) / 2

        ; set r = left + mid_distance
        mov            ebx, left        ; ebx =
left
        add            ebx, eax        ; ebx =
left + dist
        mov            r, ebx          ; r = left
+ dist

; sort each subarray
        ; push parameters for first call
        push    [ebp+20]                ; @arr
        push    left                    ; left
        push    r                        ; left +
dist
        push    [ebp+8]                 ; @result
        call    merge                   ; recursive call
on left subarray (from 0 -> midpoint)

        ; push parameters for second call
        push    [ebp+20]                ; @arr
        push    r                        ; left +
dist
        push    right                   ; right
        push    [ebp+8]                 ; @result
        call    merge                   ; recursive call
on right subarray (from midpoint -> max)

; merge arrays together
        ; Check to see if any elements remain in the left array;
        ; if so, we check if there are any elements left in the right
array;
        ; if so, we compare them.
        ; Otherwise, we know that the merge must use take the element
from the left array

        mov            i, 0
        ; i = 0
BeginFor:
        ;-----
        ; for(i = 0; i < len; i++)
        mov    ebx, i
        mov    eax, len
        cmp    i, eax
        ; compare i to len
        jge    LeaveFor
        ; if i >= len, exit for-loop

```

```

        ;if (l < left + dist) AND (r == right || max(arr[l],
arr[r]) == arr[l])
        ; if l >= r
        mov     eax, left
        add     eax, dist
        ; eax = left + dist
        cmp     l, eax
        ; compare l to left+dist
        jge     FromRight

second part
        ; if here, first part is true, now check
arr[l])
        ; (r == right || max(arr[l], arr[r]) ==
and go to FromLeft
        ; if either one is true, whole thing is true,
        ; (r == right)
        ; edi ->arr
        ; esi ->result

        ; check: (max(arr[l], arr[r]) == arr[l])
        ; find max(arr[l], arr[r])
        mov     eax, l
        mov     ebx, 4
        mul     ebx
        mov     ecx, eax
        ; ecx = l * 4
        mov     eax, r
        mul     ebx
        mov     edx, eax
        ; edx = r * 4

        mov     eax, [edi+ecx]
        ; arr[l] in eax
        mov     ebx, [edi+edx]
        ; arr[r] in ebx
        cmp     eax, ebx
        jge     LeftMax
        RightMax:
        ; if left >= right
        ; arr[r] > arr[l]
        mov     eax, [edi+edx]
LeftMax:
        ;arr[l] >= arr[r]

        ; eax = max

already

        ; is max == arr[l]?
        cmp     eax, [edi+ecx]
        je     FromLeft
        ; if true, FromLeft

```

```

; else check
second condition

; check: r == right
mov     eax, r
mov     ebx, right
cmp     eax, ebx
je      FromLeft
; if this isnt true, then second
condition is false, so whole condition is false
; go to FromRight
jmp     FromRight

FromLeft:
;result[i] = arr[l];
mov     eax, l
mov     ebx, 4
mul     ebx
; eax = l * 4
mov     ecx, [edi+eax]
; move arr[l] to ecx

mov     eax, i
mul     ebx
; eax = i * 4
mov     [esi+eax], ecx
; result[i] = arr[l]
;l++;
add     l, 1
jmp     ContinueFor

;else
FromRight:
;result[i] = arr[r];
mov     eax, r
mov     ebx, 4
mul     ebx
; eax = r * 4
mov     ecx, [edi+eax]
; move arr[r] to ecx

mov     eax, i
mul     ebx
; eax = i * 4
mov     [esi+eax], ecx
; result[i] = arr[r]
mov     ebx, [esi+eax]

;r++;
add     r, 1

ContinueFor:
add     i, 1

```

```

        jmp          BeginFor
; end for-loop
;-----
LeaveFor:

        mov         eax, left
        mov         i, eax           ; i = left
        mov         lf, eax
        mov         ebx, right
        mov         rt, ebx
        mov         eax, [esi]
For2:
;-----
; Copy the sorted subarray back to the input
; for(i = left; i < right; i++)
        mov         eax, i
        cmp         eax, right
        jge        Leave2           ; if i >= right,
leave loop

        ; arr[i] = result[i - left];
        mov         eax, i
        sub         eax, left
        ; eax = i - left
        mov         ebx, 4
        mul         ebx
        ; eax = 4 * (i - left)
        mov         ecx, eax
        ; ecx = 4 * (i - left)
        mov         eax, i
        mul         ebx
        ; eax = 4 * i
        mov         edx, eax
        ; edx = 4 * i

        mov         eax, [esi+ecx]
; eax = result[i - left]
        mov         [edi+edx], eax
; arr[i] = result[i - left]

        add         i, 1
        jmp         For2
;-----
Leave2:

LeaveProc:
        pop         esi
        pop         edx
        pop         ecx
        pop         edi
        pop         ebx
        pop         eax
        ret         16           ; remove 4
parameters from stack

```

```

merge ENDP
;-----
---

;-----
---
displayMed PROC
; Description:
; Receives:
; Returns:
; Proconditions:
; Registers changed:
;-----
---
        ret
displayMed ENDP
;-----
---

;-----
---
displayList PROC
; Description:
; Receives:
; Returns:
; Proconditions:
; Registers changed:
;-----
---
        push    ebp
        mov     ebp, esp
        sub     esp, 12                ; make space for
local vars
        push    esi
        push    eax
        push    ebx
        push    ecx
        push    edx
        mov     edx, [ebp+16]          ; store @title in
edx
        mov     esi, [ebp+12]          ; store @arr in
esi
        mov     ebx, [ebp+8]           ; store request
in ebx
        mov     local_1, 1                ;
"columnCount", initialized to 1
        call    Crlf
        call    Crlf
        call    WriteString              ; print title
        call    Crlf

        mov     local_2, 00202020h      ; move 3 spaces
to local_2

```

```

        mov         local_3, 1                ;
"columnCount" in local_3
        mov         ecx, 0                    ; set
"count" to 0 in ecx
L1:
        cmp         ecx, ebx                  ; while
count < request
        jge         LeaveArr
        cmp         local_3, 10               ; check
colCount to see if new line needed
        jle         SameLine                 ; if
colCount > 10, new line
        mov         local_3, 1                ; reset
colCount to 1
        call        Crlf                      ; move to new
line
SameLine:
        mov         eax, [esi]                ;
arr[count]
        call        WriteDec
        mov         edx, ebp
        sub         edx, 8                    ; move
address of local variable to edx
        call        WriteString
        add         esi, 4                    ; esi
points to next index
        add         ecx, 1                    ;
increment index number
        add         local_3, 1
        jmp         L1                        ;
loop
LeaveArr:
        call        Crlf
        pop         edx
        pop         ecx
        pop         ebx
        pop         eax
        pop         esi
        mov         esp, ebp                  ; reset esp,
remove local var
        pop         ebp
        ret         12
displayList ENDP

```

```

;-----
---
```

END main

## Appendix L

### IDAPro Python script to disassemble x86 executables

```
#!/usr/bin/env python3

"""
This script demonstrates using the low-level tracing hook (dbg_trace)
It can be run like: ida[t].exe -B -Strace.py -Ltrace.log file.exe
"""

import socket
import sys
import time

from idaapi import *
from idautils import *
from idc import *
import idautils
import ida_ua

class TraceHook(DBG_Hooks):

    def __init__(self):
        DBG_Hooks.__init__(self)
        # calculate the limits before actually running the process so min_ea and max_ea
        # store the limits of the current database's segments
        self.min_ea = get_inf_attr(INF_MIN_EA)
        self.max_ea = get_inf_attr(INF_MAX_EA)
        self.traces = 0
        self.epReached = False
    def dbg_trace(self, tid, ea):
        #def dbg_trace(self, tid, ea, sock):
        # Log all traced addresses
        if ea < self.min_ea or ea > self.max_ea:
            raise Exception("Received a trace callback for an address outside this database!")
        eax = get_reg_val("EAX")
        ecx = get_reg_val("ECX")
        edx = get_reg_val("EDX")
        ebx = get_reg_val("EBX")
        esp = get_reg_val("ESP")
        ebp = get_reg_val("EBP")
        esi = get_reg_val("ESI")
        edi = get_reg_val("EDI")
        eip = get_reg_val("EIP")

        self.traces += 1

        out = idc.GetDisasm(ea)
        cf = idc.get_reg_value("CF")
        pf = idc.get_reg_value("PF")
        af = idc.get_reg_value("AF")
        zf = idc.get_reg_value("ZF")
```



```

sf = idc.get_reg_value("SF")
tf = idc.get_reg_value("TF")
df = idc.get_reg_value("DF")
of = idc.get_reg_value("OF")
print("%08X %08X %08X %08X %08X %d%d%d%d%d%d%d%d" %
(ea,eax,ebx,ecx,edx,cf,pf,af,zf,sf,tf,df,of))

instr = idutils.DecodeInstruction(ea)
# log disassembly and ESP for call instructions
#if instr and instr.itype in [NN_callni, NN_call, NN_callfi]:
    #print("call insn: %s" % generate_disasm_line(ea,
GENDSM_FORCE_CODE|GENDSM_REMOVE_TAGS))
    #print("ESP=%08X" % get_reg_val("ESP"))

return 1

def dbg_run_to(self, pid, tid=0, ea=0):
# this hook is called once execution reaches temporary breakpoint set by run_to(ep) below
if not self.epReached:
    refresh_debugger_memory()
    print("reached entry point at 0x%X" % cpu.Eip)
    print("current step trace options: %x" % get_step_trace_options())
    self.epReached = True

# enable step tracing (single-step the program and generate dbg_trace events)
request_enable_step_trace(1)
# change options to only "over debugger segments" (i.e. library functions will be traced)
request_set_step_trace_options(ST_OVER_DEBUG_SEG)
request_continue_process()
run_requests()

def dbg_process_exit(self, pid, tid, ea, code):
    print("process exited with %d" % code)
    print("traced %d instructions" % self.traces)
    return 0

def do_trace():

debugHook = TraceHook()
debugHook.hook()

# Start tracing when entry point is hit
ep = get_inf_attr(INF_START_IP)
enable_step_trace(1)
set_step_trace_options(ST_OVER_DEBUG_SEG|ST_OVER_LIB_FUNC)
run_to(ep)

while get_process_state() != 0:
    wait_for_next_event(1, 0)

if not debugHook.epReached:
    raise Exception("Entry point wasn't reached!")

if not debugHook.unhook():
    raise Exception("Error uninstalling hooks!")

```

```
del debugHook
# we're done; exit IDA
qexit(0)

# load debugger module so that rest of the script works
# load_debugger("linux", 0)
# load_debugger("mac", 0)
load_debugger("win32", 0)
do_trace()
```

Command prompt command to run script:

```
ida.exe -B -Shtm_trace.py -Ltrace2.log stall2.exe
```

## Appendix M

### Executable Spatial Pooler and Temporal Memory Specifications

```
'predictor': {'sdrc_alpha': 0.0001},
'sp': {'boostStrength': 3.0,
      'columnCount': 2048,
      'localAreaDensity': 0.04395604395604396,
      'potentialPct': 0.85,
      'synPermActiveInc': 0.04,
      'synPermConnected': 0.13999999999999999,
      'synPermInactiveDec': 0.006},

'tm': {'activationThreshold': 10,
      'cellsPerColumn': 32,
      'initialPerm': 0.21,
      'maxSegmentsPerCell': 64,
      'maxSynapsesPerSegment': 32,
      'minThreshold': 5,
      'newSynapseCount': 20,
      'permanenceDec': 0.1,
      'permanenceInc': 0.1},

'anomaly': {
  'likelihood':
    { # 'learningPeriod': int(math.floor(self.probatinaryPeriod / 2.0)),
      # 'probationaryPeriod': self.probatinaryPeriod-
default_parameters["anomaly"]["likelihood"]["learningPeriod"],
      'probationaryPct': 0.1,
      'reestimationPeriod': 100
    } # These settings are copied from NAB
}
```

## Appendix N

### HTM Classifier (EFLAGS Version) Python Code

```
# HTM Classifier
# Customized for EFLAGS
# Author: Jeffrey V. Barnett
# Based on the Numenta hotgym.py example
# Date 02/Aug/2020

import time
from memory_profiler import memory_usage

import csv
import datetime
import os
import numpy as np
import random
import math
import array as arr

from htm.bindings.sdr import SDR, Metrics
from htm.encoders.rdse import RDSE, RDSE_Parameters
from htm.encoders.scalar_encoder import ScalarEncoder,
ScalarEncoderParameters

from htm.bindings.algorithms import SpatialPooler
from htm.bindings.algorithms import TemporalMemory
from htm.algorithms.anomaly_likelihood import \
    AnomalyLikelihood # FIXME use TM.anomaly instead, but it gives
worse results than the py.AnomalyLikelihood now
from htm.bindings.algorithms import Predictor
import matplotlib.pyplot as plt
from htm.bindings.algorithms import Classifier

_EXAMPLE_DIR = os.path.dirname(os.path.abspath(__file__))
#_INPUT_FILE_PATH = os.path.join(_EXAMPLE_DIR,
"\SEA_Variant50_00.csv")
_INPUT_FILE_PATH = "/mnt/faststorage/Real World Data
Streams/Malware/dump1000.csv"
default_parameters = {

    'encoderDictionary': {
        "register":
            {'size': 1024, 'radius': 0, 'category': 0, 'resolution':
0, 'minimum': 1000000, 'maximum': 4967295, 'activeBits': 3},
        "flag":
            {'size': 16, 'radius': 0, 'category': 0, 'resolution':
0, 'minimum': 0, 'maximum': 1, 'activeBits': 2},
```

```

},

'predictor': {'sdrc_alpha': 0.0001},
'sp': {'boostStrength': 3.0,
       'columnCount': 2048,
       'localAreaDensity': 0.04395604395604396,
       'potentialPct': 0.85,
       'synPermActiveInc': 0.04,
       'synPermConnected': 0.13999999999999999,
       'synPermInactiveDec': 0.006},
'tm': {'activationThreshold': 10,
       'cellsPerColumn': 32,
       'initialPerm': 0.21,
       'maxSegmentsPerCell': 64,
       'maxSynapsesPerSegment': 32,
       'minThreshold': 5,
       'newSynapseCount': 20,
       'permanenceDec': 0.1,
       'permanenceInc': 0.1},
'anomaly': {
    'likelihood':
        { # 'learningPeriod':
          int(math.floor(self.probatinaryPeriod / 2.0)),
            # 'probatinaryPeriod': self.probatinaryPeriod-
default_parameters["anomaly"]["likelihood"]["learningPeriod"],
            'probatinaryPct': 0.1,
            'reestimationPeriod': 100
          } # These settings are copied from NAB
    }
}

}

#@profile
def main(parameters=default_parameters, argv=None, verbose=True):
    np.seterr(divide='ignore', invalid='ignore')
    start_time = time.time()
    random.seed(time.time())
    HTMseed = int(time.time())
    if verbose:
        import pprint
        print("Parameters:")
        pprint.pprint(parameters, indent=4)
        print("")

    # Read the input file.
    records = []
    print("Reading file ...")
    with open(_INPUT_FILE_PATH, "r") as fin:
        reader = csv.reader(fin)
        headers = next(reader)
        next(reader)
        next(reader)

```

```

        for record in reader:
            records.append(record)

    # Make the Encoders.  These will convert input data into binary
    representations.

    registerEncoderParams = ScalarEncoderParameters()
    registerEncoderParams.size =
parameters["encoderDictionary"]["register"]["size"]
    registerEncoderParams.radius =
parameters["encoderDictionary"]["register"]["radius"]
    registerEncoderParams.category =
parameters["encoderDictionary"]["register"]["category"]
    registerEncoderParams.resolution =
parameters["encoderDictionary"]["register"]["resolution"]
    registerEncoderParams.minimum =
parameters["encoderDictionary"]["register"]["minimum"]
    registerEncoderParams.maximum =
parameters["encoderDictionary"]["register"]["maximum"]
    registerEncoderParams.activeBits =
parameters["encoderDictionary"]["register"]["activeBits"]
    registerEncoder = ScalarEncoder(registerEncoderParams)

    flagEncoderParamsCat = ScalarEncoderParameters()
    flagEncoderParamsCat.size =
parameters["encoderDictionary"]["flag"]["size"]
    flagEncoderParamsCat.radius =
parameters["encoderDictionary"]["flag"]["radius"]
    flagEncoderParamsCat.category =
parameters["encoderDictionary"]["flag"]["category"]
    flagEncoderParamsCat.resolution =
parameters["encoderDictionary"]["flag"]["resolution"]
    flagEncoderParamsCat.minimum =
parameters["encoderDictionary"]["flag"]["minimum"]
    flagEncoderParamsCat.maximum =
parameters["encoderDictionary"]["flag"]["maximum"]
    flagEncoderParamsCat.activeBits =
parameters["encoderDictionary"]["flag"]["activeBits"]
    flagEncoder = ScalarEncoder(flagEncoderParamsCat)

    encodingWidth = ((5 * registerEncoder.size) + ( 6 *
flagEncoder.size))
    enc_info = Metrics([encodingWidth], 999999999)

    # Make the HTM.  SpatialPooler & TemporalMemory & associated
    tools.
    spParams = parameters["sp"]
    sp = SpatialPooler(
        inputDimensions=(encodingWidth,),
        columnDimensions=(spParams["columnCount"],),
        potentialPct=spParams["potentialPct"],
        potentialRadius=encodingWidth,
        globalInhibition=True,
        localAreaDensity=spParams["localAreaDensity"],

```

```

        synPermInactiveDec=spParams["synPermInactiveDec"],
        synPermActiveInc=spParams["synPermActiveInc"],
        synPermConnected=spParams["synPermConnected"],
        boostStrength=spParams["boostStrength"],
        wrapAround=True
    )
    sp_info = Metrics(sp.getColumnDimensions(), 999999999)

    tmParams = parameters["tm"]
    tm = TemporalMemory(
        columnDimensions=(spParams["columnCount"],),
        cellsPerColumn=tmParams["cellsPerColumn"],
        activationThreshold=tmParams["activationThreshold"],
        initialPermanence=tmParams["initialPerm"],
        connectedPermanence=spParams["synPermConnected"],
        minThreshold=tmParams["minThreshold"],
        maxNewSynapseCount=tmParams["newSynapseCount"],
        permananceIncrement=tmParams["permanenceInc"],
        permananceDecrement=tmParams["permanenceDec"],
        predictedSegmentDecrement=0.0,
        seed=HTMseed,
        maxSegmentsPerCell=tmParams["maxSegmentsPerCell"],
        maxSynapsesPerSegment=tmParams["maxSynapsesPerSegment"]
    )
    tm_info = Metrics([tm.numberofCells()], 999999999)

    # setup likelihood, these settings are used in NAB
    anParams = parameters["anomaly"]["likelihood"]
    print(len(records))
    probationaryPeriod =
    int(math.floor(float(anParams["probationaryPct"]) * len(records)))
    learningPeriod = int(math.floor(probationaryPeriod / 2.0))
    CF_anomaly_history =
    AnomalyLikelihood(learningPeriod=learningPeriod,

    estimationSamples=probationaryPeriod - learningPeriod,

    historicWindowSize=len(records),

    reestimationPeriod=anParams["reestimationPeriod"])

    PF_anomaly_history =
    AnomalyLikelihood(learningPeriod=learningPeriod,

    estimationSamples=probationaryPeriod - learningPeriod,

    historicWindowSize=len(records),

    reestimationPeriod=anParams["reestimationPeriod"])

    AF_anomaly_history =
    AnomalyLikelihood(learningPeriod=learningPeriod,

    estimationSamples=probationaryPeriod - learningPeriod,

```

```

historicWindowSize=len(records),

reestimationPeriod=anParams["reestimationPeriod"])

    ZF_anomaly_history =
AnomalyLikelihood(learningPeriod=learningPeriod,

estimationSamples=probationaryPeriod - learningPeriod,

historicWindowSize=len(records),

reestimationPeriod=anParams["reestimationPeriod"])
    SF_anomaly_history =
AnomalyLikelihood(learningPeriod=learningPeriod,

estimationSamples=probationaryPeriod - learningPeriod,

historicWindowSize=len(records),

reestimationPeriod=anParams["reestimationPeriod"])
    TF_anomaly_history =
AnomalyLikelihood(learningPeriod=learningPeriod,

estimationSamples=probationaryPeriod - learningPeriod,

historicWindowSize=len(records),

reestimationPeriod=anParams["reestimationPeriod"])

    CF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])
    PF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])
    AF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])
    ZF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])
    SF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])
    TF_predictor = Predictor(steps=[1, 5],
alpha=parameters["predictor"]['sdr_alpha'])

    predictor_resolution = 1

    # Iterate through every datum in the dataset, record the inputs
& outputs.
    CF_inputs = []
    PF_inputs = []
    AF_inputs = []
    ZF_inputs = []
    SF_inputs = []
    TF_inputs = []

```



```

CF_anomaly = []
CF_anomalyProb = []
CF_predictions = {1: [], 5: []}

PF_anomaly = []
PF_anomalyProb = []
PF_predictions = {1: [], 5: []}

AF_anomaly = []
AF_anomalyProb = []
AF_predictions = {1: [], 5: []}

ZF_anomaly = []
ZF_anomalyProb = []
ZF_predictions = {1: [], 5: []}

SF_anomaly = []
SF_anomalyProb = []
SF_predictions = {1: [], 5: []}

TF_anomaly = []
TF_anomalyProb = []
TF_predictions = {1: [], 5: []}

encoding_time = time.time()

A = SDR(registerEncoder.size)
B = SDR(registerEncoder.size)
C = SDR(registerEncoder.size)
D = SDR(registerEncoder.size)
E = SDR(registerEncoder.size)
F = SDR(flagEncoder.size)
G = SDR(flagEncoder.size)
H = SDR(flagEncoder.size)
I = SDR(flagEncoder.size)
J = SDR(flagEncoder.size)
K = SDR(flagEncoder.size)

prevPrediction = float('nan')

print("Prequential Evaluation")
print("Evaluating 1 target(s).")
print("Evaluating...\n")
for count, record in enumerate(records):
    EIP = int(record[0],16)
    EAX = int(record[1],16)
    EBX = int(record[2],16)
    ECX = int(record[3],16)
    EDX = int(record[4],16)
    CFLAG = int(record[5])
    PFLAG = int(record[6])
    AFLAG = int(record[7])
    ZFLAG = int(record[8])
    SFLAG = int(record[9])

```

```

TFLAG = int(record[10])

CF_inputs.append(CFLAG)
PF_inputs.append(PFLAG)
AF_inputs.append(AFLAG)
ZF_inputs.append(ZFLAG)
SF_inputs.append(SFLAG)
TF_inputs.append(TFLAG)

if (count % 1000 == 0):
    print ("Count: %s" % count)
# print("EIP\t\tEAX\t\tEBX\t\tECX\t\tEDX\t\tFlags:  C P A Z
S T")
    # print("%s %s %s %s %s \t\t%s %s %s %s %s %s" %
(EIP,EAX,EBX,ECX,EDX,CFLAG,PFLAG,AFLAG,ZFLAG,SFLAG,TFLAG))

# Call the encoders to create bit representations for each
value.  These are SDR objects.
EIPBits = registerEncoder.encode(EIP)
EAXBits = registerEncoder.encode(EAX)
EBXBits = registerEncoder.encode(EBX)
ECXBits = registerEncoder.encode(ECX)
EDXBits = registerEncoder.encode(EDX)
CBits = flagEncoder.encode(CFLAG)
PBits = flagEncoder.encode(PFLAG)
ABits = flagEncoder.encode(AFLAG)
ZBits = flagEncoder.encode(ZFLAG)
SBits = flagEncoder.encode(SFLAG)
TBits = flagEncoder.encode(TFLAG)

A = EIPBits
B = EAXBits
C = EBXBits
D = ECXBits
E = EDXBits
F = CBits
G = PBits
H = ABits
I = ZBits
J = SBits
K = TBits

L = SDR(2 * registerEncoder.size).concatenate(A, B)
#Concatenate EIP, EAX
M = SDR(3 * registerEncoder.size).concatenate(L, C)
N = SDR(4 * registerEncoder.size).concatenate(M, D)
O = SDR(5 * registerEncoder.size).concatenate(N, E)
P = SDR( (5 * registerEncoder.size) +
(flagEncoder.size)).concatenate(O, F)
Q = SDR((5 * registerEncoder.size) + (2 *
flagEncoder.size)).concatenate(P, G)
R = SDR((5 * registerEncoder.size) + (3 *
flagEncoder.size)).concatenate(Q, H)

```

```

        S = SDR((5 * registerEncoder.size) + (4 *
flagEncoder.size)).concatenate(R, I)
        T = SDR((5 * registerEncoder.size) + (5 *
flagEncoder.size)).concatenate(S, J)
        # print("A= %s" % A)
        # print("B= %s" % B)
        # print("C= %s" % C)
        # print("D= %s" % D)

        encoding = SDR(encodingWidth).concatenate(T, K)
        # print("encoded = %s" % encoding)
        enc_info.addData(encoding)

        # Create an SDR to represent active columns, This will be
populated by the
        # compute method below. It must have the same dimensions as
the Spatial Pooler.
        activeColumns = SDR(sp.getColumnDimensions())

        # Execute Spatial Pooling algorithm over input space.
        sp.compute(encoding, True, activeColumns)
        sp_info.addData(activeColumns)

        # Execute Temporal Memory algorithm over active mini-
columns.
        tm.compute(activeColumns, learn=True)
        tm_info.addData(tm.getActiveCells().flatten())
        #print("Active Cells: %s" % tm.getActiveCells().flatten())
        # Predict what will happen, and then train the predictor
based on what just happened.
        CF_pdf = CF_predictor.infer(tm.getActiveCells())
        PF_pdf = PF_predictor.infer(tm.getActiveCells())
        AF_pdf = AF_predictor.infer(tm.getActiveCells())
        ZF_pdf = ZF_predictor.infer(tm.getActiveCells())
        SF_pdf = SF_predictor.infer(tm.getActiveCells())
        TF_pdf = TF_predictor.infer(tm.getActiveCells())
        for n in (1, 5):
            if CF_pdf[n]:
                CF_predictions[n].append(np.argmax(CF_pdf[n]) *
predictor_resolution)
            else:
                CF_predictions[n].append(float('nan'))

            if PF_pdf[n]:
                PF_predictions[n].append(np.argmax(PF_pdf[n]) *
predictor_resolution)
            else:
                PF_predictions[n].append(float('nan'))

            if AF_pdf[n]:
                AF_predictions[n].append(np.argmax(AF_pdf[n]) *
predictor_resolution)
            else:
                AF_predictions[n].append(float('nan'))

```

```

        if ZF_pdf[n]:
            ZF_predictions[n].append(np.argmax(ZF_pdf[n]) *
predictor_resolution)
        else:
            ZF_predictions[n].append(float('nan'))

        if SF_pdf[n]:
            SF_predictions[n].append(np.argmax(SF_pdf[n]) *
predictor_resolution)
        else:
            SF_predictions[n].append(float('nan'))

        if TF_pdf[n]:
            TF_predictions[n].append(np.argmax(TF_pdf[n]) *
predictor_resolution)
        else:
            TF_predictions[n].append(float('nan'))

    CF_anomalyLikelihood =
CF_anomaly_history.anomalyProbability(CFLAG, tm.anomaly)
    CF_anomaly.append(tm.anomaly)
    CF_anomalyProb.append(CF_anomalyLikelihood)
    CF_predictor.learn(count, tm.getActiveCells(), int(CFLAG /
predictor_resolution))

    PF_anomalyLikelihood =
PF_anomaly_history.anomalyProbability(PFLAG, tm.anomaly)
    PF_anomaly.append(tm.anomaly)
    PF_anomalyProb.append(PF_anomalyLikelihood)
    PF_predictor.learn(count, tm.getActiveCells(), int(PFLAG /
predictor_resolution))

    AF_anomalyLikelihood =
AF_anomaly_history.anomalyProbability(AFLAG, tm.anomaly)
    AF_anomaly.append(tm.anomaly)
    AF_anomalyProb.append(AF_anomalyLikelihood)
    AF_predictor.learn(count, tm.getActiveCells(), int(AFLAG /
predictor_resolution))

    ZF_anomalyLikelihood =
ZF_anomaly_history.anomalyProbability(ZFLAG, tm.anomaly)
    ZF_anomaly.append(tm.anomaly)
    ZF_anomalyProb.append(ZF_anomalyLikelihood)
    ZF_predictor.learn(count, tm.getActiveCells(), int(ZFLAG /
predictor_resolution))

    SF_anomalyLikelihood =
SF_anomaly_history.anomalyProbability(SFLAG, tm.anomaly)
    SF_anomaly.append(tm.anomaly)
    SF_anomalyProb.append(SF_anomalyLikelihood)
    SF_predictor.learn(count, tm.getActiveCells(), int(SFLAG /
predictor_resolution))

```

```

        TF_anomalyLikelihood =
TF_anomaly_history.anomalyProbability(TFLAG, tm.anomaly)
        TF_anomaly.append(tm.anomaly)
        TF_anomalyProb.append(TF_anomalyLikelihood)
        TF_predictor.learn(count, tm.getActiveCells(), int(TFLAG /
predictor_resolution))

        #print("Count: %s" % count)

        # Print information & statistics about the state of the HTM.
        # print("Encoded Input", enc_info)
        # print("")
        # print("Spatial Pooler Mini-Columns", sp_info)
        # print(str(sp))
        # print("")
        # print("Temporal Memory Cells", tm_info)
        # print(str(tm))
        # print("")
        print("Shifting predictions")
        # Shift the predictions so that they are aligned with the input
they predict.
        for n_steps, CF_pred_list in CF_predictions.items():
            for x in range(n_steps):
                CF_pred_list.insert(0, float('nan'))
                CF_pred_list.pop()

        for n_steps, PF_pred_list in PF_predictions.items():
            for x in range(n_steps):
                PF_pred_list.insert(0, float('nan'))
                PF_pred_list.pop()

        for n_steps, AF_pred_list in AF_predictions.items():
            for x in range(n_steps):
                AF_pred_list.insert(0, float('nan'))
                AF_pred_list.pop()

        for n_steps, ZF_pred_list in ZF_predictions.items():
            for x in range(n_steps):
                ZF_pred_list.insert(0, float('nan'))
                ZF_pred_list.pop()

        for n_steps, SF_pred_list in SF_predictions.items():
            for x in range(n_steps):
                SF_pred_list.insert(0, float('nan'))
                SF_pred_list.pop()

        for n_steps, TF_pred_list in TF_predictions.items():
            for x in range(n_steps):
                TF_pred_list.insert(0, float('nan'))
                TF_pred_list.pop()

        print("Calculating accuracies")

```

```

# Calculate the predictive accuracy, Root-Mean-Squared
CF_accuracy = {1: 0, 5: 0}
CF_accuracy_samples = {1: 0, 5: 0}

for idx, inp in enumerate(CF_inputs):
    for n in CF_predictions: # For each [N]umber of time steps
ahead which was predicted.
        val = CF_predictions[n][idx]
        if not math.isnan(val):
            CF_accuracy[n] += (inp - val) ** 2

    for n in sorted(CF_predictions):
        CF_accuracy[n] = (CF_accuracy[n] / CF_accuracy_samples[n])
** .5
    print("CF Predictive Error (RMS)", n, "steps ahead:",
CF_accuracy[n])

# Show info about the anomaly (mean & std)
print("CF_anomaly Mean", np.mean(CF_anomaly))
print("CF_anomaly Std ", np.std(CF_anomaly))

# Calculate the predictive accuracy, Root-Mean-Squared
PF_accuracy = {1: 0, 5: 0}
PF_accuracy_samples = {1: 0, 5: 0}

for idx, inp in enumerate(PF_inputs):
    for n in PF_predictions: # For each [N]umber of time steps
ahead which was predicted.
        val = PF_predictions[n][idx]
        if not math.isnan(val):
            PF_accuracy[n] += (inp - val) ** 2
            PF_accuracy_samples[n] += 1
    for n in sorted(PF_predictions):
        PF_accuracy[n] = (PF_accuracy[n] / PF_accuracy_samples[n])
** .5
    print("PF Predictive Error (RMS)", n, "steps ahead:",
PF_accuracy[n])

# Show info about the anomaly (mean & std)
print("PF_anomaly Mean", np.mean(PF_anomaly))
print("PF_anomaly Std ", np.std(PF_anomaly))

# Calculate the predictive accuracy, Root-Mean-Squared
AF_accuracy = {1: 0, 5: 0}
AF_accuracy_samples = {1: 0, 5: 0}

for idx, inp in enumerate(AF_inputs):
    for n in AF_predictions: # For each [N]umber of time steps
ahead which was predicted.
        val = AF_predictions[n][idx]
        if not math.isnan(val):
            AF_accuracy[n] += (inp - val) ** 2
            AF_accuracy_samples[n] += 1

```

```

    for n in sorted(AF_predictions):
        AF_accuracy[n] = (AF_accuracy[n] / AF_accuracy_samples[n])
** .5
    print("AF Predictive Error (RMS)", n, "steps ahead:",
AF_accuracy[n])

    # Show info about the anomaly (mean & std)
    print("AF_anomaly Mean", np.mean(AF_anomaly))
    print("AF_anomaly Std ", np.std(AF_anomaly))

    # Calculate the predictive accuracy, Root-Mean-Squared
    ZF_accuracy = {1: 0, 5: 0}
    ZF_accuracy_samples = {1: 0, 5: 0}

    for idx, inp in enumerate(ZF_inputs):
        for n in ZF_predictions: # For each [N]umber of time steps
ahead which was predicted.
            val = ZF_predictions[n][idx]
            if not math.isnan(val):
                ZF_accuracy[n] += (inp - val) ** 2
                ZF_accuracy_samples[n] += 1
    for n in sorted(ZF_predictions):
        ZF_accuracy[n] = (ZF_accuracy[n] / ZF_accuracy_samples[n])
** .5
    print("ZF Predictive Error (RMS)", n, "steps ahead:",
ZF_accuracy[n])

    # Show info about the anomaly (mean & std)
    print("ZF_anomaly Mean", np.mean(ZF_anomaly))
    print("ZF_anomaly Std ", np.std(ZF_anomaly))

    # Calculate the predictive accuracy, Root-Mean-Squared
    SF_accuracy = {1: 0, 5: 0}
    SF_accuracy_samples = {1: 0, 5: 0}

    for idx, inp in enumerate(SF_inputs):
        for n in SF_predictions: # For each [N]umber of time steps
ahead which was predicted.
            val = SF_predictions[n][idx]
            if not math.isnan(val):
                SF_accuracy[n] += (inp - val) ** 2
                SF_accuracy_samples[n] += 1
    for n in sorted(SF_predictions):
        SF_accuracy[n] = (SF_accuracy[n] / SF_accuracy_samples[n])
** .5
    print("SF Predictive Error (RMS)", n, "steps ahead:",
SF_accuracy[n])

    # Show info about the anomaly (mean & std)
    print("SF_anomaly Mean", np.mean(SF_anomaly))
    print("SF_anomaly Std ", np.std(SF_anomaly))

    # Calculate the predictive accuracy, Root-Mean-Squared
    TF_accuracy = {1: 0, 5: 0}

```

```

TF_accuracy_samples = {1: 0, 5: 0}

for idx, inp in enumerate(TF_inputs):
    for n in TF_predictions: # For each [N]umber of time steps
ahead which was predicted.
        val = TF_predictions[n][idx]
        if not math.isnan(val):
            TF_accuracy[n] += (inp - val) ** 2
            TF_accuracy_samples[n] += 1
    for n in sorted(TF_predictions):
        TF_accuracy[n] = (TF_accuracy[n] / TF_accuracy_samples[n])
** .5
    print("TF Predictive Error (RMS)", n, "steps ahead:",
TF_accuracy[n])

# Show info about the anomaly (mean & std)
print("TF_anomaly Mean", np.mean(TF_anomaly))
print("TF_anomaly Std ", np.std(TF_anomaly))

print("CF Accuracy: %s" % CF_accuracy)
#Plot the Predictions and Anomalies.
if verbose:
    try:
        import matplotlib.pyplot as plt
    except:
        print("WARNING: failed to import matplotlib, plots
cannot be shown.")
        return -CF_accuracy[5]

    plt.figure(1)
    plt.subplot(1, 1, 1)
    plt.title("CARRY FLAG Accuracy")
    plt.xlabel("LOC")
    plt.ylabel("Accuracy")
    plt.plot(np.arange(len(CF_accuracy[1])), CF_accuracy[1],
'bs')
    plt.legend(labels=('C Flag Accuracy'))
    plt.show()

    plt.figure(1)
    plt.subplot(2, 1, 1)
    plt.title("CARRY FLAG Predictions")
    plt.xlabel("LOC")
    plt.ylabel("C Flag")
    plt.plot(np.arange(len(CF_inputs)), CF_inputs, '^k:',
            np.arange(len(CF_inputs)), CF_predictions[1], 'bs',
            np.arange(len(CF_inputs)), CF_predictions[5],
'green', linestyle='dashed', alpha=0.75)
    plt.legend(labels=('C Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

    plt.subplot(2, 1, 2)
    plt.title("CARRY FLAG Anomaly Score")
    plt.xlabel("LOC")

```



```

plt.ylabel("CARRY FLAG")
CF_inputs = np.array(CF_inputs) / max(CF_inputs)
plt.plot(np.arange(len(CF_inputs)), CF_inputs, '^k',
         np.arange(len(CF_inputs)), CF_anomaly, 'blue')
plt.legend(labels=('Input', 'CF_anomaly Score'))

plt.figure(2)
plt.subplot(2, 1, 1)
plt.title("CARRY FLAG Predictions")
plt.xlim(500, 600)
plt.xlabel("LOC")
plt.ylabel("C Flag")
plt.plot(np.arange(len(CF_inputs)), CF_inputs, '^k:',
         np.arange(len(CF_inputs)), CF_predictions[1], 'bs',
         np.arange(len(CF_inputs)), CF_predictions[5],
'green', linestyle='dashed', alpha=0.75)
plt.legend(labels=('C Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("CARRY FLAG Anomaly Score")
plt.xlabel("LOC")
plt.ylabel("CARRY FLAG")
CF_inputs = np.array(CF_inputs) / max(CF_inputs)
plt.plot(np.arange(len(CF_inputs)), CF_inputs, '^k',
         np.arange(len(CF_inputs)), CF_anomaly, 'blue')
plt.legend(labels=('Input', 'CF_anomaly Score'))

plt.figure(3)
plt.subplot(2, 1, 1)
plt.title("PARITY FLAG Predictions")
plt.xlabel("LOC")
plt.ylabel("PARITY FLAG")
plt.plot(np.arange(len(PF_inputs)), PF_inputs, '^k',
         np.arange(len(PF_inputs)), PF_predictions[1], 'bs',
         np.arange(len(PF_inputs)), PF_predictions[5],
'green', linestyle='dashed', alpha=0.75)
plt.legend(labels=('P Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("PARITY FLAG Anomaly Score")
plt.xlabel("LOC")
plt.ylabel("P Flag")
PF_inputs = np.array(PF_inputs) / max(PF_inputs)
plt.plot(np.arange(len(PF_inputs)), PF_inputs, '^k',
         np.arange(len(PF_inputs)), PF_anomaly, 'blue', )
plt.legend(labels=('Input', 'PF_anomaly Score'))
plt.show()

plt.subplot(2, 1, 1)
plt.title("AF_predictions")
plt.xlabel("LOC")
plt.ylabel("A Flag")

```

```

plt.plot(np.arange(len(AF_inputs)), AF_inputs, 'red',
         np.arange(len(AF_inputs)), AF_predictions[1],
'blue',
         np.arange(len(AF_inputs)), AF_predictions[5],
'green', )
plt.legend(labels=('A Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("AF_anomaly Score")
plt.xlabel("LOC")
plt.ylabel("A Flag")
AF_inputs = np.array(AF_inputs) / max(AF_inputs)
plt.plot(np.arange(len(AF_inputs)), AF_inputs, 'red',
         np.arange(len(AF_inputs)), AF_anomaly, 'blue', )
plt.legend(labels=('Input', 'AF_anomaly Score'))
plt.show()

plt.subplot(2, 1, 1)
plt.title("ZF_predictions")
plt.xlabel("LOC")
plt.ylabel("Z Flag")
plt.plot(np.arange(len(ZF_inputs)), ZF_inputs, 'red',
         np.arange(len(ZF_inputs)), ZF_predictions[1],
'blue',
         np.arange(len(ZF_inputs)), ZF_predictions[5],
'green', )
plt.legend(labels=('Z Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("ZF_anomaly Score")
plt.xlabel("LOC")
plt.ylabel("Z Flag")
ZF_inputs = np.array(ZF_inputs) / max(ZF_inputs)
plt.plot(np.arange(len(ZF_inputs)), ZF_inputs, 'red',
         np.arange(len(ZF_inputs)), ZF_anomaly, 'blue', )
plt.legend(labels=('Input', 'ZF_anomaly Score'))
plt.show()

plt.subplot(2, 1, 1)
plt.title("SF_predictions")
plt.xlabel("LOC")
plt.ylabel("S Flag")
plt.plot(np.arange(len(SF_inputs)), SF_inputs, 'red',
         np.arange(len(SF_inputs)), SF_predictions[1],
'blue',
         np.arange(len(SF_inputs)), SF_predictions[5],
'green', )
plt.legend(labels=('S Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("SF_anomaly Score")

```

```

plt.xlabel("LOC")
plt.ylabel("S Flag")
SF_inputs = np.array(SF_inputs) / max(SF_inputs)
plt.plot(np.arange(len(SF_inputs)), SF_inputs, 'red',
         np.arange(len(SF_inputs)), SF_anomaly, 'blue', )
plt.legend(labels=('Input', 'SF_anomaly Score'))
plt.show()

plt.subplot(2, 1, 1)
plt.title("TF_predictions")
plt.xlabel("LOC")
plt.ylabel("T Flag")
plt.plot(np.arange(len(TF_inputs)), TF_inputs, 'red',
         np.arange(len(TF_inputs)), TF_predictions[1],
'blue',
         np.arange(len(TF_inputs)), TF_predictions[5],
'green', )
plt.legend(labels=('T Flag', '1 Step Prediction, Shifted 1
step', '5 Step Prediction, Shifted 5 steps'))

plt.subplot(2, 1, 2)
plt.title("TF_anomaly Score")
plt.xlabel("LOC")
plt.ylabel("T Flag")
TF_inputs = np.array(TF_inputs) / max(TF_inputs)
plt.plot(np.arange(len(TF_inputs)), TF_inputs, 'red',
         np.arange(len(TF_inputs)), TF_anomaly, 'blue', )
plt.legend(labels=('Input', 'TF_anomaly Score'))

plt.show()
return -CF_accuracy[5]

if __name__ == '__main__':
    #main()
    mem_usage = memory_usage(main)
    print("HTM - Prequential - Size(kB): %4f" % (max(mem_usage) *
1000))

```

## References

- Ahmad, S., & Luiz, S. (2019). *How can we be so dense? The benefits of using highly sparse representations*. Cornell University. Retrieved 2020, from <http://arxiv.org/abs/1903.11257>
- Ahmad, S., & Hawkins, J. (2015). *Properties of sparse distributed representations and their application to hierarchical temporal memory*. Retrieved 2020, from <https://arxiv.org/abs/1503.07469>
- Alom, M., Venkata, R., B., & Taha, T. (2015). Intrusion detection using deep belief networks. *2015 National Aerospace and Electronics Conference (NAECON)*, 339–344.
- Amine El Ouassouli, Robinault, L., & Scuturici, V. (2019). Mining complex temporal dependencies from heterogeneous sensor data streams. *In Proceedings of the 23rd International Database Applications & Engineering Symposium (IDEAS '19)*, (23), 1–10.
- Barcelo-Rico, F., Esparcia-Alcazar, A., & Villalon-Huerta, A. (2016). Semi-supervised classification system for the detection of advanced persistent threats. *Recent Advances in Computational Intelligence in Defense and Security, Studies in Computational Intelligence*, 621, 225–248.
- Barria, C., Cordero, D., Cubillos, C., & Palma, M. (2016). Proposed classification of malware based on obfuscation. *2016 6th International Conference on Computers Communications and Control (ICCCC)*, 37–44
- Bifet, A., Gavaldá, R., Holmes, G., & Pfahringer, B. (2017). *Machine learning for data streams: with practical examples in MOA* (Adaptive computation and machine learning series). MIT Press.

- Bifet, A., Holmes, G., & Pfahringer, B. (2010). Leveraging bagging for evolving data streams. *ECML PKDD 2010: Machine Learning and Knowledge Discovery in Databases*, 135–150.
- Bonhoff, G. (2007). Using hierarchical temporal memory for detecting anomalous network activity. Retrieved 2020, from <http://www.dtic.mil/dtic/tr/fulltext/u2/a482820.pdf>
- Breiman, L., Friedman, J., Stone, C. J., & Olsh, R. (1984). *Classification and regression trees* (Wadsworth Statistics/Probability). Chapman and Hall.
- Bushan, S., Kumar, P., Kumar, A., & Sharma, V. (2016). Scan time antivirus evasion and malware deployment using Silent-SFX, *2016 Spring International Conference on Advances in Computing*, 1–4.
- Canfora, G., Sorbo, A. D., Mercaldo, F., & Visaggio, C. A. (2015). Obfuscation techniques against signature-based detection: A case study. *2015 Mobile Systems Technologies Workshop*, 21–26.
- Cannady, J. (2013). The detection of temporally distributed network attacks using an adaptive hierarchical neural network. *2013 World Congress on Nature and Biologically Inspired Computing*, 5–9.
- Cannady, J. (2000). Next generation intrusion detection: autonomous reinforcement learning of network attacks. In *Proceedings of the 23rd National Information Systems Security Conference*, 1-12.
- Cheng, C., Tay, W. T., & Huang, G. (2012). Extreme learning machines for intrusion detection. *The 2012 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

- Chiba, D., Yagi, T., Akiyama, M., Shibahara, T., Mori, T., & Goto, S. (2018). Domain profiler toward accurate and early discovery of domain names. *International Journal of Information Security*, (17), 661–680.
- Choliy, A., Li, F., & Gao, T. (2017). Obfuscating function call topography to test structural malware detection against evasion attacks, *2017 International Conference on Computing*, 808–813.
- Cui, Y., Ahmad, S., & Hawkins, J. (2016). Continuous online sequence learning with an unsupervised neural network model. *Neural Computation*, 28(11), 2474–2504.
- Cui, Y., Surpur, C., Ahmad, S., & Hawkins, J. (2016). A comparative study of HTM and other neural network models for online sequence learning with streaming data. *2016 International Joint Conference on Neural Networks (IJCNN)*, 1530–1538.
- Dehghan, M., Beigy, H., & ZareMoodi, P. (2016). A novel concept drift detection method in. *Intelligent Data Analysis*, 20, 1329–1350.
- Demsar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7, 1–30.
- Dongre, P., & Malik, L. (2014). A review on real time data stream classification and adapting to various concept drift scenarios, *2014 IEEE International Advance Computing Conference (IACC)* , 533-537.
- Drew, J., Hahsler, M., & Moore, T. (2017). Polymorphic malware detection using sequence classification methods and ensembles. *EURASIP Journal on Information Security*, (55). Retrieved 2020, from <http://doi.org/10.1186/s13635-017-0055-6>
- Duong, Q., Ramampiaro, H., & Norvag, K. (2018). Applying temporal dependence to detect changes in streaming data. *Applied Intelligence*, 4805–4823.

- Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine* *V, 17*, 37–54.
- Flynn, M., Kanerva, P., & Bhadkamkar, N. (1989). *Sparse Distributed Memory: Principles and Operation*. Retrieved 2020, from <http://i.stanford.edu/pub/cstr/reports/csl/tr/89/400/CSL-TR-89-400.pdf>
- Fu, Y., Guo, X., Xie, Y., Zhang, D., & Li, H. (2015). Disease diagnosis supported by hierarchical temporal memory. *2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 863–870.
- Gandel, S. (2015). Lloyd's CEO: Cyber attacks cost companies \$400 billion every year. Retrieved 2020, from <http://fortune.com/2015/01/23/cyber-attack-insurance-lloyds/>
- Ghomeshi, H., Gaber, M., Medhat, & Kovalchuk, Y. (2019). EACD: Evolutionary adaptation to concept drifts in data streams. *Data Mining and Knowledge Discovery*, *33*, 663–694.
- Giron, J., & Kolbitsch, C. (2015). *Exposing Rombertik – turning the tables on evasive malware*. *Lastline*. Retrieved 2020, from <http://www.lastline.com/labsblog/exposing-rombertik-turning-the-tables-on-evasive-malware/>
- Goel, K., & Batra, S. (2019). Dynamically updated diversified ensemble-based approach for handling concept. *Turkish Journal of Electrical Engineering & Computer Sciences*, *28*, 556–574.
- Gözüaçık, Ö., Büyükçakır, A., Bonab, H., & Can, F. (2019). Unsupervised concept drift detection with a discriminative classifier. *In Proceedings of the 28th ACM*

- International Conference on Information and Knowledge Management (CIKM '19)*, 2365–2368. Retrieved 2020, from <https://doi.org/10.1145/3357384.3358144>
- Harries, M. (1999), SPLICE-2 comparative evaluation: electricity pricing. *Technical report, The University of South Wales*. <https://www.openml.org/d/151>
- Hawkins, J., Lewis, M., Klukas, M., Purdy, S., & Ahmad, S. (2019). A framework for intelligence and cortical function based on grid cells in the neocortex. *Frontiers in Neural Circuits*, 12, 121, 1-14.
- Hawkins, J., Ahmad, S., Purdy, S., & Lavin, A. (2017). *Biological and Machine Intelligence (BAMI)*. [Unpublished manuscript].
- Hawkins, J. (2011a). *Hierarchical Temporal Memory (HTM) Whitepaper*. Numenta Research Papers. <http://numenta.com/neuroscience-research/research-publications/papers/hierarchical-temporal-memory-white-paper/>
- Hawkins, J. (2011b). *Hierarchical Temporal Memory, including HTM Cortical Learning Algorithms*. Hierarchical Temporal Memory (HTM) Whitepaper. Retrieved 2020, from <http://numenta.com/assets/pdf/whitepapers/hierarchical-temporal-memory-cortical-learning-algorithm-0.2.1-en.pdf>
- Hawkins, J., & Blakeslee, S. (2004). On Intelligence. *New York, NY: Henry Holt and Company, LLC*.
- Hinton, G., & Salakhutdinov, S. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786), 504–507.
- Huang, G.-B., Zhu, Q.-Y., & Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3), 489–501.
- Hulten, G., & Domingos, P. (2002). Mining complex models from arbitrarily large databases in constant time. *In Proceedings of the eighth ACM SIGKDD*



- international conference on Knowledge discovery and data mining (KDD '02)*, 525–531.
- Hulten, G., Spencer, L., & Domingos, P. (2001). Mining time-changing data streams. *In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01)*, 97–106.
- Irvine, K. R. (1999). *Assembly language for Intel-based computers* (3rd ed.). Prentice-Hall.
- Irmanova, A., Krestinskaya, O., & James, A. (2018). Image-based HTM word recognizer for language processing. *2018 IEEE International Conference on Consumer Electronics-Asia*, 206–212.
- Jadhav, A., Vidyarthi, D., & Hemavathy, M. (2016). Evolution of evasive malwares: A Survey. *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, 641–646.
- Kanerva, P. (1990). *Sparse distributed memory*. MIT Press.
- Khangamwa, G. (2010). Detecting network intrusions using hierarchical temporal memory. *Social Informatics and Telecommunications Engineering*, 64, 41–48.
- Kolter, J. Z., & Maloof, M. A. (2007). Dynamic weighted majority: an ensemble method for drifting concepts. *Journal of Machine Learning Research*, (8), 2755–2790.
- Kruegel, C. (2015). *Evasive malware exposed and deconstructed*. RSA Conference 2015. Retrieved 2020, from [https://www.rsaconference.com/writable/presentations/file\\_upload/crwd-t08evasive-malware-exposed-and-deconstructed.pdf](https://www.rsaconference.com/writable/presentations/file_upload/crwd-t08evasive-malware-exposed-and-deconstructed.pdf)

- Lange, L., Alonso, O., & Strötgen, J. (2019). The power of temporal features for classifying news articles. *In Companion Proceedings of The 2019 World Wide Web Conference (WWW '19)*. Association for Computing Machinery, 1159–1160.
- Lavin, A., & Ahmad, S. (2015). Evaluating Real-Time anomaly detection algorithms -- The Numenta anomaly benchmark. *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 38–44.
- Lim, C., & Nicsen. (2015). Mal-EVE: Static detection model for evasive malware, 10th EAI International Conference on Communications and Networking in China, 283–288.
- Madireddy, S., Balaprakash, P., Carns, P., Latham, R., Lockwood, G. K., Ross, R., Snyder, S., & Stefan, M. W. (2019). Adaptive learning for concept drift in application performance modeling. *In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*, (79), 1–11.
- Marak, V. (2015). *Windows malware analysis essentials*. Packt Publishing and Birmingham.
- Marpaung, S., Sain, M., & Lee, H. (2012). Survey on malware evasion techniques: state of the art and challenges. *14th International Conference on Advanced Communication Technology (ICACT)*, 744–749.
- McAfee Labs. (2017). McAfee labs threats report June 2017. Retrieved 2020, from <https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-jun-2017.pdf>
- McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153–157.

- Moh, M., Pininti, S., Doddapaneni, S., & Moh, T. (2016). Detecting web attacks using multi-stage log analysis. *International Advanced Computing Conference 2016 IEEE 6th International Conference on Advanced Computing*, 733–738.
- Mouchaweh, S. (2016). Learning from data streams in dynamic environments. Springer.
- Moustafa, N., & Slay, J. (2015). UNSW-NB15: A comprehensive data set for network intrusion detection systems (*UNSW-NB15 network data set*). *Conference: Military Communications and Information Systems Conference (MilCIS)*, 1–8.
- Numenta Inc. (2011). Hierarchical temporal memory, including HTM cortical learning algorithms. Retrieved from [http://numenta.org/resources/HTM\\_CorticalLearningAlgorithms.pdf](http://numenta.org/resources/HTM_CorticalLearningAlgorithms.pdf)
- Osorio, F. C., Qiu, H., & Arrott, A. (2015). Segmented sandboxing - A novel approach to Malware polymorphism detection. *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 59–68.
- Park, S., Seo, S., Jeong, C., & Kim, J. (2018). Network intrusion detection through online transformation of eigenvector reflecting concept drift. *In Proceedings of the First International Conference on Data Science, E-learning and Information Systems (DATA '18)*, (17), 1–4.
- Pesaranghader, A., Viktor, H., & Paquet, E. (2018). Reservoir of diverse adaptive learners and stacking fast hoeffding drift detection methods for evolving data streams. *Mach Learn*, (107), 1711–1743.
- Purdy, S. (2015). Encoding Data for HTM Systems, Numenta, Inc, Redwood City, CA,. Retrieved 2020, from <https://arxiv.org/abs/1602.05925>

- Rajesh, B., Reddy, Y., & Reddy, B. (2015). A survey paper on malicious computer worms. *International Journal of Advanced Research in Computer Science & Technology*, 3(2), 161–167.
- Rastogi, V., Chen, Y., & Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1), 99–108.
- Rogers, M. S. (2016). Statement of Admiral Michael S. Rogers Commander United States Cyber Command before the Senate Armed Services Committee, Retrieved 2020, from [https://www.armed-services.senate.gov/imo/media/doc/Rogers\\_04-0516.pdf](https://www.armed-services.senate.gov/imo/media/doc/Rogers_04-0516.pdf)
- Rouse, M. (2010). *Metamorphic and polymorphic malware*. Metamorphic; polymorphic malware. <http://searchsecurity.techtarget.com/definition/metamorphic-and-polymorphicmalware>
- Sag, B. (2012). *MASM Random Number Sorter*. <http://github.com/beckysag/masm-random-integers>
- Sikorski, M., & Honig, A. (2012). *Practical malware analysis*. William Pollock; San Francisco.
- Singh, A., Walenstein, A., & Lakhotia, A. (2012). Tracking concept drift in malware families. *In Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 81–92.
- Sosha, A., Liu, C., Gladyshev, P., & Matten, M. (2012). Evasion-Resistant Malware Signature Based on Profiling Kernel Data Structure Objects, *2012 7th International Conference on Risks and Security of Internet and Systems*, 1–8.

- Street, W. N., & Kim, Y. (2001). A streaming ensemble algorithm (*SEA*) for large-scale classification In. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 377–382.
- Tavallae, M., Stakhanova, N., & Ghorbani, A. (2010). Toward credible evaluation of anomaly-based intrusion-detection methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(5), 516–524.
- Sun, Y., Wang, Z., Bai, Y., Dai, H., & Nahavandi, S. (2018). A classifier graph based recurring concept detection and prediction approach, *2018*, 1–13.
- Wang, C., Zhao, Z., Gong, L., Zhu, L., Liu, Z., & Cheng, x. (2017). A distributed anomaly detection system for In-Vehicle network using HTM. *IEEE Access*, 6, 9091–9098.
- Wang, B., & Pineau, J. (2016). Online Bagging and Boosting for Imbalanced Data Streams. in *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 12, pp. 3353-3366, 28(12), 3353–3366.
- Wang, H., Fan, W., Yu, P. S., & Ha, J. (2003). Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '03)*, 226–235.
- Wong, W., & Stamp, M. (2006). Hunting for metamorphic engines. *Journal in Computer Virology*, 2, 211–229.
- Yaacob, A., Tan, I., Chien, S., & Tan, H. (2010). ARIMA based network anomaly detection, Second International Conference on Communication Software and Networks, 2010. *ICCSN '10*, 205-209.

- Yang, R., Xu, S., Feng, L., & Feng, L. (2018). An ensemble extreme learning machine for data stream classification. *Algorithms*, 11(107), 1–16.
- Zhang, B., & Chen, Y. (2019). Research on detection and integration. *EURASIP Journal on Wireless Communications and Networking*, 86, 1–7.
- Zhang, B., Xue, L., Wang, W., Qin, S., & Wang, D. (2016). Model updating mechanism of concept drift detection in data stream based on classifier pool. *EURASIP Journal on Wireless Communications and Networking*, 217, 1–8.
- Zhong, L., Hu, L., & Zhou, H. (2019). Deep learning based multi-temporal crop classification. *Remote Sensing of Environment*, 221, 430–443.
- Žliobaite I, Pechenizkiy, M., & Gama, J. (n.d.). (2016) An Overview of Concept Drift Applications. *An overview of concept drift applications*. Big Data Analysis: New Algorithms for a New Society. Studies in Big Data, vol 16. Springer, Cham.