Anomaly detection in user behavior of websites using Hierarchical Temporal Memories

Using Machine Learning to detect unusual behavior from users of a web service to quickly detect possible security hazards.

VICTOR BERGER
Anomaly detection in user behavior of websites using Hierarchical Temporal Memories

Using Machine Learning to detect unusual behavior from users of a web service to quickly detect possible security hazards.

VICTOR BERGER

Master’s Thesis at CSC
Supervisor: Pawel Herman
Examiner: Anders Lansner
Abstract

This Master’s Thesis focuses on the recent Cortical Learning Algorithm (CLA), designed for temporal anomaly detection. It is here applied to the problem of anomaly detection in user behavior of web services, which is getting more and more important in a network security context.

CLA is here compared to more traditional state-of-the-art algorithms of anomaly detection: Hidden Markov Models (HMMs) and t-stide (an N-gram-based anomaly detector), which are among the few algorithms compatible with the online processing constraint of this problem.

It is observed that on the synthetic dataset used for this comparison, CLA performs significantly better than the other two algorithms in terms of precision of the detection. The two other algorithms don’t seem to be able to handle this task at all. It appears that this anomaly detection problem (outlier detection in short sequences over a large alphabet) is considerably different from what has been extensively studied up to now.
Contents

1 Introduction 1
   1.1 Problem statement, objectives and scope .................. 2
   1.2 Outline ............................................. 2

2 Background 3
   2.1 Related Work ......................................... 3
       2.1.1 Non-temporal Anomaly detection .................. 3
       2.1.2 Anomaly detection on sequences .................. 3
   2.2 HTM & CLA Fundamentals ................................. 4
       2.2.1 HTM structure & CLA ............................. 4

3 Methods 7
   3.1 Data: Web applications access logs ...................... 7
       3.1.1 Contents of log files ............................ 7
       3.1.2 Specific problem constraints ..................... 8
       3.1.3 Pre-processing hypothesis ......................... 8
       3.1.4 Dataset used .................................... 9
   3.2 Application of CLA to logs data ......................... 9
       3.2.1 URL encoding .................................... 10
       3.2.2 Anomaly measure for CLA ........................ 10
       3.2.3 CLA parameters .................................. 11
       3.2.4 HTM regions stacking ............................. 12
   3.3 Comparison models .................................... 12
       3.3.1 t-stide .......................................... 12
       3.3.2 HMM ............................................. 13
   3.4 Performance measures .................................. 13

4 Results 15
   4.1 HTM-CLA .............................................. 15
   4.2 t-stide ................................................ 18
   4.3 Hidden Markov Models .................................. 19

5 Discussion 23
   5.1 Intrinsic problem difficulty .......................... 23

iv
5.2 Interpretation of the results ............................... 23
5.3 The interpretability question ............................... 24

6 Conclusions and future work ............................... 25

Bibliography ..................................................... 27

Appendices ....................................................... 29

A Algorithm of generation of the dataset ..................... 31
    A.1 Sitemap ................................................... 31
    A.2 Anomalies ................................................ 32
    A.3 Content Generation ...................................... 32
Chapter 1

Introduction

Anomaly detection is a class of problem gaining traction in the domain of computer security. As systems are monitored more and more closely and attacks are getting more and more elaborate, traditional rule-based systems for raising alerts are becoming insufficient. Thus, anomaly detection techniques based on machine learning are being considered, to make monitoring systems more dynamic and adaptive[1, 2]. The goal is to detect anomalous event in the infrastructure before the potential attack reaches its critical point.

Learning to detect anomalies in monitoring data is inherently an unsupervised process: each infrastructure is unique and cannot be easily formally described, so there are no ground truth labels on the data. Nowadays, monitoring systems are being structured as "Artificial Immune Systems" (AIS): algorithms that learn a representation of the systems given the history of past events, and raising alerts anytime something deviates from this baseline [3].

This formulation of anomaly detection can be interpreted similarly to the data mining problem of "outlier detection". This question is handled by algorithms such as one-class support vector machines [4], cluster-analysis [5] or auto-encoder-based methods [6].

This Master’s Thesis is focused on unsupervised online anomaly detection in sequences: given a stream of relatively short sequences, train a model to classify each sequence as normal or anomalous, while only processing each sequence once.

These constraints are typically met in processing log data from web servers: sequences are the actions taken by each user of the service, and the goal is to detect users having an usual behavior on the website. On very high-traffic websites, the amount of data this represents is so large that storing it to process it offline is not an option, hence the anomaly detection algorithm needs to work online.

This particular problem does not appear to have been extensively reported in the literature. On the other hand, the recent Cortical Learning Algorithm (CLA) [7] has been successfully applied to temporal sequence prediction [8] and has been considered to have a good potential for generalizing to more complex data. In this thesis, I study how CLA performs in application to a challenging problem of
detecting anomaly in the behavior of users of a website.

1.1 Problem statement, objectives and scope

This Master’s Thesis focuses on the application of CLA to unsupervised online anomaly detection in sequences, instantiated as the detection of anomalous behavior from users of a website. Given a stream of user sessions, the objective is to classify them as "normal" or "anomaly", each session being a sequence of the actions of a user (here, the temporally ordered list of the URLs he or she visited).

This context implies specific properties of the data: the number of possible URLs can be very large (or even infinite for websites with dynamical content), while the length of each sequence is on average relatively short (most of the users visiting a website know what they are looking for and do not wander long).

Another constraint is the fact that, for a large website, the log stream is very large, and cannot easily be stored to be processed in batch over several iterations. As such, the algorithms must work online, and process the data at least as quickly as it arrives.

I study here the performance of Hierarchical Temporal Memories (HTMs) with the Cortical Learning Algorithm (CLA) on the problem of online unsupervised anomaly detection in sequences, and compare it to two traditional algorithms among the few that can be trained with these constraints: Hidden Markov Models (HMMs) and t-stide [9] (an n-gram-based approach).

These algorithms are compared using anomaly detection performance (taking into account specificity and sensitivity) and computing time, thus providing insights into the scalability potential.

This comparison was performed on a synthetic dataset for two reasons. First of all, datasets of real logs of user sessions on a website are difficult to obtain due to privacy considerations. Secondly, using a synthetic dataset allowed me to work on a simplified version of this problem (website with a relatively small architecture) so that it was actually computable given the resources at my disposition.

1.2 Outline

The next chapter of this thesis gives background elements about anomaly detection algorithms and the context in which they can be trained, as well as an overall description of the CLA. Then in Chapter 2 I describe the considerations that were made to implement the evaluated models, the dataset they were evaluated on and the evaluation measures. Chapter 4 introduces the numerical results of this evaluation, and Chapter 5 my interpretation of them.
Chapter 2

Background

2.1 Related Work

2.1.1 Non-temporal Anomaly detection

The original problem of anomaly detection can be formulated as "outlier detection": given a dataset, find in it points that do not follow the global statistic repartition.

A deeply explored approach to such anomaly detection is the use of One Class Support Vector Machines (OCSVM), using both supervised [10] and unsupervised [10, 11] training algorithms.

Another approach to consider is Kernel-based Online Anomaly Detection (KOAD) [12], a method using a dynamic dictionary approximating the normal state of the dataset, and processing the datapoints in an online fashion. It allows continual unsupervised learning to adapt to variations of the analyzed stream.

2.1.2 Anomaly detection on sequences

An obvious approach to handling sequences is to use windowing on the sequence to create fixed-length datapoints, and use them with previously described algorithms. These approaches work for simple sequential data, but cannot be considered for capturing mid or long-term temporal dependency, as the state space grows exponentially with the length of the window: for an alphabet $\Sigma$ and windows of size $n$, the effective alphabet size handled by the non-temporal method will be $|\Sigma|^n$.

Other methods have been developed specifically to handle temporal sequences. Approaches based of Finite-State-Automatas [13], and mostly markovian techniques such as Probabilistic-Suffix-Trees [14], and more generally Hidden Markov Models (HMM) [9, 15, 16].

However, many of these methods are not applicable to online training and exploitation, because they either require to work on the dataset as a whole, or need to be able to process it several times. Therefore, for comparative analysis I have chosen t-stide [9] as a representative of window-based algorithms that is online-compatible and HMMs with online capability [17].
2.2 HTM & CLA Fundamentals

HTMs are learning structures introduced by the American company Numenta. They have been designed with inspiration of the cerebral cortex structure with much more fidelity than more classic artificial neural networks, and were proposed to account for cortex-like processing of memorized information [7]. It is a quite recent approach. The potential of HTMs has not yet been fully demonstrated even though they have already been shown to successfully address certain pattern recognition problems involving a temporal context [18].

2.2.1 HTM structure & CLA

An HTM construction is composed of regions as show in figure 2.1, each one composed of two blocks: a Pattern Memory and a Transition Memory. The Pattern Memory recognizes spatial patterns in the input, while the Transition Memory learns temporal patterns over the spatial patterns recognized by the Pattern Memory.

![Figure 2.1. Representation of an HTM hierarchy composed of 4 blocks. Information flows from the input (at the bottom) to the output (at the top), but also laterally within each region.](image)

While different approaches are possible regarding HTMs, the CLA has been developed specifically to handle temporal streams of data [7].

Pattern Memory

This block works on the input of the region, which is supposed to be in a Sparse Data Representation (SDR). Meaning that it is a set of boolean inputs, which at each step roughly 2% of them are activated. This representation is necessary for
the algorithm, and has interesting properties regarding resilience and combination [19].

The Pattern Memory is composed of a set of neurons, each one with a set of potential synapses linked to a random subset of the inputs. Each of this potential synapses has a permanence value between 0 and 1, and is sensitive to its associated input if this value is above a certain threshold (a typical value being 0.2), in which case the synapse is said to be connected. The training process involves adjusting these permanence values.

Each of these neurons is considered active in the output if at least a certain number of its synapses are connected to inputs that are active. When a neuron gets activated, it inhibits its neighbors so that only a few "most activated" neurons are actually propagated to the next level\(^1\). This is to preserve the sparsity of the data representation.

**Transition Memory**

Each of the outputs of the Pattern Memory is mapped to a column of neurons in the Transition Memory, and if activated activates this column. When a column is activated, two cases are possible: either some neurons of this column were in a "predictive" state following previous input, and they get activated, or none was predictive, and they all get activated.

Then, each neuron has several dendrites, each with a set of potential synapses with other neurons of other column, and with permanence values. If enough synapses are connected to active neurons, the dendrite gets activated, and if at least one of its dendrite is activated, the neuron gets into 'predictive' state, in anticipation of being activated by the next input, as represented in figure 2.2.

The output of this block (and thus of the region) is a set of boolean values associated with each column, being true if the associated column has either some active or predictive neurons. It can be interpreted as the union of "the columns that are active" and "the columns that could probably be active next step".

\[^1\text{The amount of activated neurons is an hyperparameter of the model, but is typically set to 8-10\%.}\]
Figure 2.2. Representation of a neuron from the transition memory of an HTM. The green pattern is the input from the pattern memory (a random subset of the input neurons of the whole structure). The blue inputs are the dendrites, reporting the "active" state of other neurons at previous timestep. The neuron has two outputs: the "predictive" on the left that is forwarded to the dendrite of other neurons at next timestep, and the active one. Both are merged for the final input of the transition memory.
Chapter 3

Methods

3.1 Data: Web applications access logs

3.1.1 Contents of log files

In this work I focus on analyzing access logs of a website. The data that can typically be used is the history of all requests made by client of this website. A typical webserver will provide in its logs:

- date and hour
- origin IP address
- target website
- queried URL
- protocol used (HTTP or HTTPS)
- size of the request
- user-agent\(^1\) of the user’s browser
- the HTTP code\(^2\) of the answer
- the size of the answer

Of all these features, many are already handled by other rule-based detectors (suspicious user-agents, blacklisted IPs, ...), and this thesis focuses on behavioral

\(^1\)An user-agent is a text string provided by web browser identifying which software they are and their capabilities. It can be used to discriminate between a desktop computer or a mobile browser and serve a different website, for example.

\(^2\)An HTTP code is a numeric value representing whether a request was successful, and provide additional context about the success or failure. Common ones are 200 for "OK", 404 for "Page not found" or 500 for "Internal server error".
anomalies from users of a website. I thus focus on the information about what the
user did: date and hour, origin IP and queried URL (target website is omitted, as
I assume to be working on the logs of a single website).

This information allows to reconstruct the sessions of each user, as the succession
of web pages they visited on the website.

3.1.2 Specific problem constraints

To be applicable on a real-scale website, a few constraints must be taken in account:

Unsupervised learning: there is no known information of which user sessions
contain anomalies and which sessions are free of them, or when the anomalies
take place precisely: it is highly dependent on the website. The algorithm
thus needs to learn by itself what is "normal" and what is "not normal", by
observing the logs content and building a statistical profile of it.

Online learning: On high-traffic websites, keeping a large portion of the logs
history in memory to process it in batch is not possible. The flow of this data
stream is often much too large (hundreds of thousands of requests per second)
and must be processed online as much as possible.

Stable complexity: The algorithm must continually learn on the data stream
for long periods of time. As such, the average per sequence time complexity
should not depend on the amount of data that has already been processed. An
algorithm with an internal state growing to store more and more information
about the data it has processed would get slower and slower as the amount of
work it must do to process each new sequence increases, which would not be
compatible with it running continuously on an online architecture.

3.1.3 Pre-processing hypothesis

The original stream of events from a webserver log is just the succession of all
requests it received. All user sessions that occurred in parallel are mixed in the
same stream of data.

Expecting an anomaly detection algorithm to track all concurrent sessions at
the same time would be extremely optimistic: a high-traffic website can easily have
thousands of sessions occurring in parallel. For this reason I have worked under
the assumption that the data is pre-processed to regroup requests into actual user
sessions, which then are given to the algorithm as a stream a sequences of requests,
each sequence being an user session. The algorithm is then expected to mark each
sequence as "Normal" or "Anomalous".

This pre-processing fits into the constraints described in the previous subsection
in terms of complexity and memory overhead: most user sessions are quite short (a
few minutes each), and the cost of reconstructing a session is in the same order of
magnitude of storing the log data.
3.1.4 Dataset used

Traffic histories of real websites are difficult to use as a resource, for privacy and security reasons, they cannot be made public. Another limitation is that the amount of data coming from a real website would be too large to handle in this thesis. In order to solve this issue, I use a generated dataset, using Markov processes. Two Markov processes are defined: one is the baseline normal data, the other generates sequences resembles the ones from the first, but cannot be generated by it (for example, the second process can generate a session representing an user that buys something without first logging in to his or her account, which is against the rules of normal behavior of the first process).

The complete dataset is an ordered set of sequences, each randomly drawn from an non-uniform Bernoulli distribution over the two Markov processes. The goal of the anomaly detection algorithms is thus to discriminate which Markov process generated the sequences. As I generated the data, I have the ground truth of which process generated which sequence. As stated in the problem definition, I cannot use this ground truth to train the models but I use it to evaluate their performance.

The dataset used there have the following properties:

- it has 20,000 sequences
- each sequence is around 10-20 urls long on average (but can be shorter or longer)
- there are 75 different urls in total
- around 1% of the sequences are anomalies

The detailed process of generation of the dataset is described in Appendix A.

3.2 Application of CLA to logs data

To apply the CLA to previously described data as a stream of sequence of URLs, the following is done for each sequence:

- Reset the internal state of the HTM (deactivate all neurons).
- Encode the URLs in SDR (as described in 3.2.1)
- Input sequentially the SDRs in the HTM
- The final anomaly score of the sequence is the maximum of the anomaly scores of each input\(^3\).

This is then a score-based classifier, and finally a threshold must be chosen to separate the two classes ‘normal’ and ‘anomaly’.

\(^3\)The first input is excluded as its anomaly score is always 1, given no neuron was active at previous step.
3.2.1 URL encoding

To be understood by the CLA, the URLs need to be translated into SDR format. The most straightforward way to do this translation is to use a categorical encoder [20]: create a random SDR for each possible input, if the size of input is large enough (often 1024 or 2048 is largely sufficient), the risk of collision is practically zero. However, it is possible to benefit from a relevant property of the SDRs: the union (bitwise OR) of a few SDRs can still reasonably be used as an SDR [19]. These properties can be used to encode in the data the fact that for example /foo/bar is more similar to /foo/baz than to /qqx.

Given an URL, for example /foo/bar/baz, an SDR is computed for each segment using the categorical encoder. So here I have 3 SDRs: one for foo, one for bar and one for baz. Then, final encoding will be the union of these encodings.

For example, imagining a 16-bits input, if the 3 encodings were:

- foo: 0001000000100000
- bar: 0000000010000001
- baz: 0000100001000000

Then, the final encoding for /foo/bar/baz would be 0001100011100001.

This can help the model to handle large categories in some website structure, as well as make use of the hierarchical nature of the URLs. If a web application has a lot of different pages like /articles/title-of-the-article, the model can potentially understand that all pages in the /articles/ category are quite similar.

3.2.2 Anomaly measure for CLA

Given the output of a transition memory, it is possible to measure how unexpected the input was, by comparing the activated columns at time $t$ with the predictive columns at time $t-1$.

The exact formula of this anomaly score is given by [20]:

$$anomaly(t) = \frac{|A_t - (P_{t-1} \cap A_t)|}{|A_t|}$$

where $A_t$ is the set of active CLA columns at time $t$ and $P_{t-1}$ the set of predictive CLA columns at time $t-1$.

This gives a score between 0 and 1, which counts the fraction of columns that were active while not predicted at previous step: how unexpected the input was.

From this anomaly score for each item of a sequence, an anomaly score can be derived for the entire sequence as the maximum of the anomalies of its items.
3.2.3 CLA parameters

HTMs can be configured via several meta-parameters (the Numenta framework allows for 14 parameters to be changed). A random sweep of this space parameter showed that their values have a large impact on the behavior of the algorithm. However, the correlation between the choice of parameters and the quality of the result is highly non-trivial. There is at the time of writing of this paper little heuristics to evaluate a priori which values should be chose for these parameters. The number of columns and number of neurons by column have a large impact on the performance, as they define the connectivity of the graph of neural connections and thus the maximum complexity of the model. However the impact of the other parameters are difficult to isolate from each other.

All these parameters are:

.activationFraction: Percent of neurons that should be activated in the output of the HTM

.globalInhibition: Should the inhibition of neighbors columns be computed globally (by sampling the N most active columns) or locally (each active column inhibites its neighbors)

.synPermActiveInc: Increment step of the permanence of active input synapses

.synPermInactiveDec: Decrement step of the permanence of inactive input synapses

.synPermConnected: Permanence threshold to consider a potential input synapse as connected

.cellsPerColumn: Number of neurons in each column

.initialPermanence: Initial value of permanence for dentrite synapses

.connectedPermanence: Permanence threshold to consider a potential dentrite synapse as connected

.minThreshold: Number of synapses that must be active for their input branch to be active

.maxNewSynapseCount: Number of synapses of a newly-created dentrite branch

.permenanceIncrement: Increment step of the permanence of active dentrite synapses

.permenanceDecrement: Decrement step of the permanence of inactive dentrite synapses

.activationThreshold: Number of synapses that must be active for their dentrite branch to be active
Table 3.1. Parameters used to configure the CLA.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Used value</th>
<th>Possible range</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>activationFraction</td>
<td>0.085</td>
<td>[0; 1]</td>
<td>0.020</td>
</tr>
<tr>
<td>globalInhibition</td>
<td>True</td>
<td>{True, False}</td>
<td>True</td>
</tr>
<tr>
<td>synPermActiveInc</td>
<td>0.003</td>
<td>[0; 1]</td>
<td>0.010</td>
</tr>
<tr>
<td>synPermInactiveDec</td>
<td>0.020</td>
<td>[0; 1]</td>
<td>0.008</td>
</tr>
<tr>
<td>synPermConnected</td>
<td>0.650</td>
<td>[0; 1]</td>
<td>0.100</td>
</tr>
<tr>
<td>cellsPerColumn</td>
<td>3</td>
<td>N*</td>
<td>8</td>
</tr>
<tr>
<td>initialPermanence</td>
<td>0.500</td>
<td>[0; 1]</td>
<td>0.500</td>
</tr>
<tr>
<td>connectedPermanence</td>
<td>0.200</td>
<td>[0; 1]</td>
<td>0.500</td>
</tr>
<tr>
<td>minThresold</td>
<td>4</td>
<td>N*</td>
<td>10</td>
</tr>
<tr>
<td>maxNewSynapseCount</td>
<td>20</td>
<td>N*</td>
<td>20</td>
</tr>
<tr>
<td>permanenceIncrement</td>
<td>0.250</td>
<td>[0; 1]</td>
<td>0.100</td>
</tr>
<tr>
<td>permanenceDecrement</td>
<td>0.010</td>
<td>[0; 1]</td>
<td>0.010</td>
</tr>
<tr>
<td>activationThresold</td>
<td>9</td>
<td>N*</td>
<td>8</td>
</tr>
</tbody>
</table>

A simple random search allowed me to find parameters seemingly appropriate for our class of problems, but no attempt to fine-tune them precisely was done. The actual parameters used are given in Table 3.1, they are different from the default values of parameters of the Numenta framework, which resulted in poor performance of this problem.

3.2.4 HTM regions stacking

As the output of a HTM is also an SDR, it can be given as input to an other HTM. This is a central structure as described in the CLA whitepaper [7]: HTM regions are building blocks for bigger hierarchies.

These hierarchies can be structurally related to deep convolutional networks [21]: each layer recognizes small and simple patterns, but these patterns are constructed on the patterns recognized by the previous layer. As a whole, deep convolutional networks can recognize very complex patterns. Here HTM hierarchies follow the same idea, but recognizing temporal patterns as well as spatial ones.

3.3 Comparison models

3.3.1 t-stide

The stide (sequence time-delay embedding) algorithm [9] detects anomalous sequences by first being trained on a dataset of normal sequences, and then comparing sub-sequences of the test data of a certain fixed length with the ones it was trained on, and reporting the mismatches. An anomaly score being given by the amplitude of this mismatch using the Hamming distance.
This algorithm cannot be directly applied to this thesis’ problem, as I do not have a dataset of normal data to pre-train the model on. However a variation of stide, t-stide, has also been proposed [9]. It works by additionally taking into account relative frequencies of these subsequences. As a baseline test, I adapt t-stide to build this relative frequency database at runtime, and use it on the fly to measure the anomaly. This variant is the same as classifying each sequence using a t-stide classifier trained on all the previous sequences.

3.3.2 HMM

A natural choice to model sequences is using HMMs. The HMM is trained using an online Baum-Welch algorithm [17]. For each sequence, I first evaluate the likelihood of it being generated by this model using the well known forward algorithm before training the HMM on it.

The number of hidden states is a parameter of the model, and the number of possible observations is set to a number large enough so that every URL in the dataset can be mapped to a single observation.

This likelihood $l_n$ of sequence $n$ is then compared to an average of the previous sequences\footnote{This average can be computed in a very efficient way incrementally, so the memory cost associated with it is effectively $O(1)$: $avg_n = \frac{l_n}{n} + \frac{n-1}{n} avg_{n-1}$} in log-space to compute a score, as the likelihood can be several orders of magnitude different from each other:

$$anomaly(n) = -\log(l_n) - \frac{1}{n-1} \sum_{k=1}^{n-1} \log(l_k)$$

This anomaly score is then associated with a threshold to make a classifier.

3.4 Performance measures

In the data considered in this work, the two classes ('normal' and 'anomaly') are very unbalanced: anomalies often represent less than 1% of the data. This makes the use of plain accuracy as a performance indicator very misleading: if 99% of the dataset is in the same class, a classifier reporting always this class will have an accuracy score of 99%, even though it is a really bad classifier.

Every considered model associates an anomaly score to each sequence. This score represents "how much" the models considers this sequence as anomalous. To build a complete classifier from this, it is needed to add a threshold, in order to discriminate between the two classes. This thresold can typically be chosen using a Receiver Operating Characteristic (ROC) analysis. However, measures like Precision and
Recall are more in line with the expectations of such anomaly detection algorithms, they are computed from the confusion matrix like this:

\[
Recall = \frac{TP}{TP + FN} \quad Precision = \frac{TP}{TP + FP}
\]

These metrics are equivalent to ROC analysis [22], but more visually representative of the practical objectives and costs associated with our problem.

Recall represents the fraction of real positives that have been detected by the classifier, Precision measures the fraction of true positives in the points the classifier reported as positive. These two measures can be used in correlation with an estimation of the cost of handling false positives and false negatives, to build a cost-driven score.

However, these measures are biased by the composition of the dataset, and while directly correlated to the practical usefulness, they thus cannot give a generic indication of the performance. Thus I will also compare the algorithms using an unbiased measure: Matthews correlation coefficient (MCC)[23],

Defined from the confusion matrix like this:

\[
MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}
\]

It ranges from $-1$ to $1$, with $1$ being the perfect classifier, 0 a random classifier, and $-1$ the classifier that is always wrong.

In all cases, any model trained on this problem is expected to behave poorly at first, as it needs to adapt to the dataset. As such, the first 1000 sequences of the dataset are discarded as a "training period", and I only use the rest of the dataset to evaluate its performances.

Lastly, time complexity is also taken into account for the evaluation. The model is supposed to run online, as such, it should not slow-down as time advances (at least not beyond a certain threshold), or it will not be able to process the new data quickly enough, and it will fall behind. Some models have a constant complexity (HMMs for example) and are not subject to this issue. Others, like HTM, re-arrange their internal state as learning goes, and can get slower as the state (in the case of HTMs, the graph of neural connections) gets bigger. Also, in the case of HTM, the complexity evolution is not easily predictable, so I measure it as a sliding average of the processing time of each sequence divided by the number of requests in it.
Chapter 4

Results

The evaluation of each model is done in a similar way: each sequence of the dataset is given to the model, one after the other. For each sequence the anomaly score given by the model is recorded, as well as the time the model took to process this sequence. The order of the sequence is not changed: as these models are time-dependant, the order in which they see the sequence has an impact on their performance. These anomaly scores are then used to compute Precision-Recall curves and MCC score for various values of the classification threshold.

4.1 HTM-CLA

I compare the impact of the main factor determining the complexity of the algorithms\(^1\): the number of columns in a region, as well as configurations stacking two regions, either by having the second region the same size as the first, or half of it (the column sizes have been chosen as successive powers of two\(^2\), mostly by mimicking the numbers shown in Numenta’s documentation [20]). Figure 4.1 shows Recall-Precision curves and Tables 4.1-4.2 report the best MCC score reached across the different classification threshold values, for single and two-region stack respectively.

First of all, it can be observed that for the considered configurations, the double-region stacks has significantly lower performance than single-region models. Run-time wise, two-region models are around 2 times slower than single-region ones, which was to be expected.

It can be observed that the performance correlates with the number of columns up to a point: the single-region with 2048 columns performs on average as well as the one with 1536, but both performs on average better than the one with 1024. But the gain of doubling the number of columns is not very large: as we can see

\(^1\)Temporal complexity of the CLA is mostly driven by the size of the connection graph between neurons, which is mostly determined by the number of columns and the number of neurons in each column.

\(^2\)1536 is 1024 * 1.5, it has been added as an intermediate between 1024 and 2048
the average Precision-Recall curves of the 3 single-region models are very similar to each other.

Regarding the best performing region – the one with 2048 columns, the threshold value yielding the highest MCC score has a recall of 0.22 and a precision of 0.52. Relative to the dataset of 20,000 sequences, that means that the classifier reported 84 sequences as anomalies, of which 44 were real anomalies (out of 200 anomalies in the dataset).

The usability of such a model depends a lot on the relative cost of dismissing a false positive versus handling the consequences of a false negative (meaning the infrastructure might get compromised). Under the assumption that the former is much cheaper than the latter, even a model with a precision of around 50% can provide good help to a security team. This assumption is based on the fact that the cost of a false-positive is only time from the security team for investigation, while the cost of a false-negative is potentially an attack on the system that remains undetected. Such an attack, depending on its aim and success, can be of large cost to the company operating the monitored system.

![Recall-Precision curves of HTM results on the dataset.](image)

**Figure 4.1.** Recall-Precision curves of HTM results on the dataset. The labels of the curves represent the number of columns used for the HTM, both for single-region models and with two stacked regions. Recall and Precision are computed as a mean over 5 simulations of the CLA on the dataset for each curve, the shaded region represents the variance on both axis.

Regarding time complexity, the results are given in Figure 4.2. It can be observed that while time performance changes a lot at the beginning during the initial
Table 4.1. Average over 5 simulations of the best MCC scores reached across the classification threshold spectrum, for some single-region HTM setups.

<table>
<thead>
<tr>
<th>columns count</th>
<th>512</th>
<th>1024</th>
<th>1536</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC</td>
<td>0.252 ± 0.023</td>
<td>0.275 ± 0.022</td>
<td>0.303 ± 0.013</td>
<td>0.319 ± 0.027</td>
</tr>
</tbody>
</table>

Table 4.2. Average over 5 simulations of the best MCC scores reached across the classification threshold spectrum, for some double-region stacking of HTM.

<table>
<thead>
<tr>
<th>columns count</th>
<th>1024-1024</th>
<th>1536-768</th>
<th>1536-1536</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC</td>
<td>0.125 ± 0.018</td>
<td>0.146 ± 0.011</td>
<td>0.145 ± 0.014</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>columns count</th>
<th>2048-1024</th>
<th>2048-2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC</td>
<td>0.130 ± 0.026</td>
<td>0.126 ± 0.020</td>
</tr>
</tbody>
</table>

In the learning phase, it stabilizes over time towards a near constant average, which is correlated with the number of columns (as one could expect, bigger models are slower to train and evaluate).

Figure 4.2. Average processing time per request in seconds over a sliding window of 100 sequences. The labels of the curves represent the number of columns used for the HTM, both for single-region models and with two stacked regions. The shaded areas represent the standard deviation. Mean and standard deviation are computed over the 100 points of the sliding window times 5 simulations, so a total of 500 points.
4.2 t-stide

Results when using the t-stide approach are presented in Figure 4.3, for a subsequence size of 8 and various threshold values. Subsequence sizes from 3 to 10 were tested, for all threshold presented in the figure, with very similar but slightly worse results. Across all these configurations the best MCC score achieved was 0.038.

It can be seen that t-stide performs barely better than a random classifier on our dataset. This is understandable, as our problem’s context is quite different from the one t-stide was developed for. This original problem involved traces of system calls, meaning the size of the alphabet was relatively restricted (the number of different system calls) and there were a lot of similar subsequences in a "normal" dataset (a computer program is very deterministic in what it normally does, especially the programs considered in [9], which are relatively simple tools). The problem examined in this thesis is of different nature: the alphabet is very large and the normal sequences present a lot of variability.

Regarding time complexity, t-stide was so fast to run on our dataset that the relative processing time of different sequences cannot be measured reliably\footnote{The fact of actually recording this time measurably slows down the program}: running the whole dataset takes only a few seconds. However, from the structure of the

![Figure 4.3. Recall-Precision curves of t-stide results on the dataset, with a subsequence size of 8, and different threshold values.](image-url)
algorithm it can be anticipated that the complexity grows as a logarithmic function of the number of different subsequences observed.

### 4.3 Hidden Markov Models

Various sizes of hidden states for the HMM were considered. Recall-Precision curves are shown in Figure 4.4, and best MCC scores are reported in Table 4.3.

![Recall-Precision curves](image)

**Figure 4.4.** Recall-Precision curves of HMM results on the dataset, with different number of hidden states of the model.

<table>
<thead>
<tr>
<th>Hidden state size</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC</td>
<td>0.074</td>
<td>0.074</td>
<td>0.073</td>
<td>0.075</td>
<td>0.079</td>
<td>0.080</td>
<td>0.080</td>
</tr>
</tbody>
</table>

**Table 4.3.** Best MCC scores reached across the threshold spectrum, for each hidden state size of HMM.

It can be observed that all these HMMs perform similarly and rather poorly. The number of hidden states does not affect the capacity of HMMs to represent the dataset. This is likely due to the fact that the average length of sequence is smaller than the alphabet size. As a consequence, each sequence only contains a small subset of all possible observations. This interferes a lot with the training, and as a consequence the HMM seems unable to link observation and hidden state, and simply acts as a frequency measure of all symbols.
I also measured the processing time on the HMMs, which is stable, and report it in Figure 4.5. HMMs show a relatively constant processing time with regular slow spikes. However, the HMM with 200 hidden states (which is the slowest measured) still processes the dataset one order of magnitude faster than the HTMs.
Figure 4.6. Recall-Precision curves of a random classifier and the best performing instance of t-stide, HMM and HTM measured. Note that the Precision axis is in logarithmic scale.
Chapter 5

Discussion

5.1 Intrinsic problem difficulty

The problem addressed in this thesis has special features that render it difficult to approach for traditional anomaly detection algorithms.

First of all, the alphabet on which sequences are defined is very large if each URL is considered as a possible letter. On the other hand, the sequences are relatively short: many user sessions are not longer than 20 requests, while the alphabet in the small dataset used here had 75 different URLs (and this number will be much larger on a real world website). This means that each sequence will only show a small portion of the alphabet, and many letters are susceptible to appear very rarely. The encoding method using SDR for the CLA helps mitigating this by using the hierarchical structure of the URLs, but the combinatorics remain large.

Secondly, the data can be processed only once. This means the algorithms may be sensitive to the order in which the sequences occur. Specifically, an initial fraction of the data cannot be used for prediction at all, as the model will at first output garbage, needing to first learn a little the data. This also means that if by chance an anomaly is very frequent during this first part, the model may quickly learn to label it as normal.

Considering these characteristics, it cannot be expected from any algorithm to achieve perfect performance on real-world data.

5.2 Interpretation of the results

I observed that classic state-of-the-art temporal anomaly detection techniques – HMMs and t-stide – cannot reliably handle the problem. It is likely explained by the data characteristics described above: it is composed of short sequences over a very large alphabet. Both methods appear to handle this case very poorly. They were developed for different scenarios, where sequences are relatively long compared to the size of the alphabet.

It is very apparent for the HMM, for which the size of the hidden state does not
seem to have any impact on the performance. Suggesting the model is completely incapable of learning any temporal pattern on this data (as even an hidden state of size 1 gives roughly the same performance). Basic tests on HMMs with varying sequence lengths and alphabet size seems to confirm that this class of models struggles to learn when the size of the alphabet gets significantly larger that the average size of the sequence. However, I could not find any literature sources to confirm or contradict this observation.

The new algorithm tested in this work, CLA, seems to handle this better, reaching precision scores around 70%. This is still not production-ready performance, but the models were not fine-tuned, and better results can certainly be reached.

5.3 The interpretability question

HTM is far less interpretable than HMMs and t-stide, which create a rather explicit representation of the sequences they model. It is essentially a black-box approach given our current understanding of the algorithm.

The main rationale in favor of CLA is that it mimics neural behavior of the brain, which is good at anticipating and recognizing anomalies. However, while HTMs and CLA are being experimentally tested with positive result, there is not yet a solid mathematical study of its behavior. This renders these models very experimental, with very few heuristics available for tuning parameters for a given task, and thus making it difficult to evaluate a priori what their field of application is.

They also have a large number of meta-parameters, which interact with each other in an hardly predictable manner (for now at least). The random search I did to choose the parameter set and the fact that the ones I used were different from what the documentation from Numenta suggests [20] implies these meta-parameters are not universal. This may indicate that tuning CLA parameters is particularly prone to overfitting, depending on how the dataset used to tune them is representative of the real data that will be processed.
Chapter 6

Conclusions and future work

HTMs are already marketed by Numenta for anomaly detection in numerical time-series such as stock market values\(^1\). My work shows that their capacity to model temporal sequences can be generalized to much more complex data with encouraging results. These results are promising particularly in comparison with other competing techniques, such as t-stide and HMMs. They thus show potential application as statistical online anomaly detectors for computer security, being able to model temporal sequences with more complexity than what current state-of-the-art techniques are capable.

In this work, I did not attempt to optimize the parameters to obtain the best possible results. This is not an easy task as evaluating HTMs on a dataset is significantly slower than with other models, and the large meta-parameter space to explore has no simple relation to the performance of the model. Optimizing HTMs in such a context will probably require hyper-parameters tuning algorithms [24, 25], such as Bayesian optimization [26] or genetic algorithms [25]. Special care must be taken to have representative datasets for this tuning, in order to avoid overfitting.

Another consideration is that other applicable algorithms could be identified to compare them against CLA on this problem, as traditional methods like t-stide and HMM are clearly unfit in this case.

The output of HTMs also contains a representation of its input in the temporal context of the preceding inputs. This representation was not used except for calculating the anomaly score and stacking regions. It could be possible to associate HTMs with another classifier like a feedforward neural network, which given the anomaly score and the output of the HTM could compute a revised anomaly score. This second classifier could be trained in a semi-supervised fashion, to include feedback, and dismiss events that are known to be false positives.

In general, it can be concluded that HTMs combined with other tools and human supervision may produce a reliable performance as detectors of attacks on websites, delivered a security team of an enterprise, willing to reduce as much as possible the number of undetected attacks.

\(^1\)http://numenta.com/htm-studio/


Appendix A

Algorithm of generation of the dataset

A.1 Sitemap

The dataset will emulate a simplified web shop. The map of accessible urls is the following:

/  
/account  
/account/myorders  
/account/preferences  
/admin  
/currentorder  
/currentorder/validate  
/login  
/register  
/shop  
/shop/group1  
/shop/group1/article1-[1-15]  
/shop/group2  
/shop/group2/article2-[1-15]  
/shop/group3  
/shop/group3/article3-[1-15]

For a quick description of this map

- the /account/ category, which regroups the list of previous orders of the customer, as well as an interface to change their preferences

- the /admin page, inaccessible to customers

- the /currentorder category, regroups the items the customer has selected. They can go to /currentorder/validate to actually buy them.
• the /login and /register pages, allowing the user to login to their account or create one
• the /shop category, containing 3 groups of items, with 15 articles in them

A.2 Anomalies

In this general structure, 3 generic kinds of anomalies are considered:

access to the admin page: no customer should be able to access it
access to preferences without login: the customer goes to an account page without first logging-in.
validate empty order: the customer validates an order without seeing at least one item (an thus the order is empty)

If any of these happen in a sequence, it should be labeled as an anomaly.

A.3 Content Generation

Each generated sequence represents the interaction of a customer with the website. For each sequence, the customer is assigned with a set of generic tasks (there can be repetition) from:

• check their preferences
• check their old orders
• check their current order
• search for an item (moves to a random item page)
• browse a category (moves to a category page, and maybe check some items in it)

As well as some random parameters:

• will finish the session by buying something?
• comes from a search engine? (the session starts on a random item page rather than the homepage of the website)
• does the user have an account? (if not, they’ll need to register)

The user will then proceed to execute their set of actions in a random order, logging-in (or registering their account) before the first action that requires it, and finish by validating or not an order depending on the parameter.

Each sequence has a probability of containing an anomaly (I used an 1% chance), which is then introduced appropriately depending on which anomaly it is.