Real time anomaly prediction in distributed systems using Hierarchical Temporal Memory

Pratik Singh, Anirudha Tambolkar
Department of Computer Science
North Carolina State University
{psingh22, atambol}@ncsu.edu

Abstract
Many supervised and unsupervised machine learning algorithms have been implemented to detect anomalies in the system performance. However, many of these implementations fail to meet a balance between the accuracy of prediction and resource intensive computation. This arises due to their inability to handle streaming data and real-time predictions using limited computing and storage capacities. In this paper, we use Hierarchical Temporal Memory (HTM) model[1] to predict system performance metrics and use it to predict SLO violations with high lead time. HTMs are especially designed for learning, remembering and processing of temporal sequences and can learn continuously without any manual intervention. HTM sequence memory continuously adapts to changing statistics in the data, hence we expect high prediction accuracy of this model with good lead time. We plan to evaluate HTMs on cloud infrastructure using Amazon Web Services and use RUBiS[5] online auction benchmark to train and test our model.

Keywords
Hierarchical Temporal Memory; Unsupervised System Behavior Learning; Cloud Computing; Anomaly Prediction; Artificial Neural Networks; Distributed Systems, Amazon Web Services (AWS)

1 Introduction

Although modern distributed systems provide features such as high availability, scalability and robustness to the system, it is still prone to failures due component malfunction. Such failures are manifested by various faults and can affect the performance of the system as well as end user experience. Often, the quality of services offered by vendors is bound by the service level objective (SLO) document. An example of such a service would be infrastructure as a service (IaaS) where the provider of the IaaS must ensure that SLO is not violated at any time. The violation of SLO can cause huge financial loss to both the provider and the client and hence efficient mechanisms must be built to minimize SLO violations. In other cases where no explicit guarantees are made regarding the quality of service, unchecked failures can affect user experience and reduce vendor's credibility as well. Such service providers often rely on advertisements for revenue and therefore, user experience can become an important factor. Thus in either case, it is a necessity to detect faults at early stages proactively so that the services built on top of distributed systems meets certain quality requirements.

Our approach to predict anomalies in a distributed system begins with collection of system level metrics from various components. The metrics can be considered as periodic flow of data that can be processed using a streaming dataflow engine. Therefore, our approach to tackle the problem of collecting metrics involves leveraging pre-existing tools that are popularly used by data streaming and analytics applications. The system under observation is RUBiS, which we deploy in a multi node environment. We use Apache Kafka for building real time data pipeline, where the nodes hosting RUBiS shall produce metrics data. This data can be consumed and processed by HTM engine. In addition to that, we use various services of AWS to host the infrastructure. Our approach intends to make the metrics collection and anomaly prediction system

reliable and scalable. AWS provides the tools and resources to make this possible.

For anomaly prediction in system performance, machine learning algorithms should have these properties:

- They should be computationally less expensive due to which they take less time to predict the next state of the system. This guarantees high lead time when a system anomaly is about to happen which in turn allows more time to apply suitable changes to the system to mitigate the predicted anomalous state.
- The model should be able to learn continuously without the need to retrain the model periodically. Batch leaning models cannot be used in real-time system anomaly detection as they can miss the changing trends in the data in between the batches and also, they need manual intervention to retrain the model.
- The model should provide good noise tolerance, i.e., it should be able to identify similar input sequences even if the input sequences are changed to a certain extent. The system performance metrics like CPU utilization percentage may have different absolute values but may represent the same semantic state of the system.
- Also, the real-world time series generally contains semantics that can span multiple time steps. The minimum number of previous timesteps required to make the next prediction is a parameter that the machine learning algorithm should learn automatically from the dataset.

HTMs which is an artificial neural network is based on the concepts of how the neocortex in the brain works: have all these properties and hence it is an ideal candidate for real time anomaly prediction in IaaS cloud systems. This model has been specially designed for learning, remembering and processing of temporal sequences. This model also provides good noise tolerance and is able to identify similar system states even if a large percentage of noise is added to the input. For HTMs, the real world input data is encoded into sparse distributed representations(SDRs). SDRs[2] are the internal representation of the individual neurons in the neural network. They are bit array[0,1] of a fixed size with semantic meaning. Different states in the SDRs is able to represent the semantics of the encoded input data efficiently. So, similar input data will have more number of overlapping in the SDRs. When the number of overlapping between two SDRs are above a certain automatically learned threshold, the two input data are said to be similar. This is how the HTM learns to recognise spatial patterns. Now Temporal Memory algorithm[4] is used to learn transition of spatial patterns. In the Temporal Memory algorithm, when a neuron becomes active, it forms connections to other neurons that were active in the previous step. Neurons can hence predict when they will become active by looking at their connections. If all the neurons do this, collectively they can store and recall temporal sequences, and they can predict what is likely to happen next.

2 Related Works

Cloud computing has dominated the IT infrastructure services industry in the last two decades. Cloud infrastructure is prone to anomalies due to inherent nature of large distributed systems. So there has been continuous research in predicting anomalies in cloud infrastructure using machine learning algorithms. These researches showed how supervised and unsupervised learning techniques can be applied to predict system anomalies. Supervised learning requires labelled data (normal and anomalous data) while unsupervised learning techniques do not require any labelling and hence they are inherently more suitable for real time anomaly detection. This is because labelling of the data is a difficult job which sometimes requires manual supervision. The most common supervised algorithms are supervised neural networks, parameterization of training model, support vector machine learning, k-nearest neighbors, Bayesian networks and decision trees. K-nearest neighbor (k-NN) is one of the most conventional nonparametric techniques that are used in supervised learning for anomaly detection. The most common unsupervised algorithms are self-organizing maps (SOM), K-means, C-means, expectation-maximization meta-algorithm (EM),

adaptive resonance theory (ART), and one-class support vector machine. Our work is based on the research paper Continuous Online Sequence Learning with an Unsupervised Neural Network Model[3]. We intend to learn its effectiveness for anomaly prediction in a cloud based environment.

3 Online System Behavior Learning

We chose to use the HTM learning technique for online learning to achieve scalable and efficient system behavior learning. During learning, the HTM uses a competitive learning process to adjust the weight vectors of different neurons. SDR classifier learn associations between a given state of the Temporal Memory at time t, and the value that is to be fed into the Encoder at time t+n (where n is the number of steps into the future you want to predict. t+1, t+5, t+2 - or all three!).

The SDR Classifier must update its weight matrix in order to learn and make more accurate predictions/inferences. When the SDR Classifier is first initialized, all weights are 0. Each iteration, the SDR Classifier makes a prediction about the input n steps in the future. This prediction is in the form of a probability distribution. We can represent it like so:

$$y=(y1,y2,...,yk)$$

where, y is our computed probability distribution. Each index/element of y is the likelihood computed by the SDR Classifier of seeing that bucket index used by the encoder. Each time step we also get a target distribution. This is simply the probability distribution that we want our predicted distribution, y, to match. We can model it like so:

$$z=(z1,z2,...,zk)$$

where, z is the target distribution.

All of z's indexes/elements will be 0, except for one. The "on" element is the one that matches the bucket used by the encoder to encode the input at that time step. At time t we obtain an input to the encoder, and thus a target distribution, z. If we are doing 5 step prediction, then we will have received and stored the activation pattern from the Temporal Memory at time t−5. So, we use the stored activation pattern from time t−5 to make a prediction, y. Fortunately for us, we also have the actual input for time t. So we can compare our prediction, y, to the actual target distribution, z, and see how close we were. Then we can adjust out weight matrix, W, to try and make better predictions. We need to calculate error scores for each index of y. For that, we use the following formula:

$$error_j = z_j - y_j$$

An error>0 means SDR classifier's prediction was too low, an error<0 means it was too high.

We multiply the error by an alpha, α,(learning rate. set to 0.05) which lets us speed up or slow down the rate at which the SDR Classifier learns/adapts. The product of this operation is the value we will use to update our matrix:

$$update_j=\alpha(error_j)$$

(Note that this means each output unit in the SDR gets a single update value, $update_j$, for all of its columns in the weight matrix, W. Also note that α must be greater than 0, or learning cannot occur.)

Once we have the update value for each j/matrix-row, we simply go into the weight matrix, W, grab each row, go to each of the columns whose input units were active at time t−5, and add our update value to whatever weight value exists there already. We can model the whole update process like so:

$$W'_{ij}=W_{ij}+\alpha(z_j-y_j)$$

And again, we only do this for the weights that apply to input units that were active at t−5.

4 System Design and Implementation

The following sections describes the design of the solution proposed in terms of the distributed systems as well as the prediction service.

4.1 Architecture of Distributed System

The system architecture proposed here is based following goals in mind. The design tries to emulate corporate infrastructures which consist of a network address translator (NAT), public and private networks, virtual private network (VPN), firewalls

and routers. This consideration would allow such our solution to be easy to integrate in pre-existing infrastructure architectures. Figure 1 gives a high level view of the design used in our implementation.

Figure 1: Architecture of the system under observation and the prediction service built on top of it. The entire system is deployed on using AWS Cloudformation [6] and uses VPC, EC2 [7] and S3 service. We used Cloudformation templates to maintain the whole infrastructure as code and enabled single click deployment.

One of the biggest factors is ability of our solution to scale. The test bed of our system is Rubis, which is inherently a distributed system. It can scale on its own with its modular architecture, and along with that the amount of workload it can support can increase as well. Since our solution depends on the amount of data generated by RUBiS, the system we design around it should be able to handle the increased workload. Our implementation relies on Kafka broker to form this data pipeline. Kafka is a natural choice for streaming data due to its scalable nature. The response time received from the Web Server on Rubis Clients and CPU metrics of Web Server at that epoch time are produced on to two topic on Kafka as shown in Figure 2. The messages from these topic are aggregated together into a single topic that represents the unlabeled data stream for the HTM engine. Aggregation of these two data streams is required because the rate at which they are produced is different.

Figure 2: Web Server and Rubis Clients produce two different types of data stream. The data is then aggregated together as a third data stream and consumed by HTM model.

We chose to observe the CPU metrics for every epoch, however the response time data points are generated by each RUBiS client independently depending on the workload used. Thus there can be several data points for response time for a given epoch. The aggregation module combines all the response times for an epoch and also calculates the average response time, maximum response time, total number of requests and number of violations that occurred during that moment. Here the messages from the two stream are merged into a single aggregated data point for each epoch. The messages are

represented as key value pairs in a JSON object that can be consumed by HTM engine as unlabeled data.

Another design choice consisted of defining a threshold value for response time beyond which a response would be categorized as a violation. An average web request takes around 100ms. The test environment we use consists of a private network only and therefore we expect the response time to be much lesser. We chose a threshold of 70ms for our experiments. We collect these metrics every 1 second.

4.2 Architecture of Predictive Models

Three models were built and tested in our experiments. The first model is a three component model comprising of single layers of spatial pooler, temporary memory and SDR classifier. The input CPU utilization data is fed into a Random scalar data encoder which converts rav CPU utilization data into its corresponding stream of bits vector. This bit vector is fed to the spatial pooler. The spatial pooler learns the bit patterns in this input via spatial pooler memory which consists of columns of neurons. The spatial pooler generates another bit vector as its output which is the input signal for the temporal memory.

The temporal pooler learns the bit patterns in this input via temporal memory which consists of columns of neurons. These neurons forms new connections to other neurons and increase or decrease the strength of the connections between the already connected neurons when the learning mode is set on. The temporal pooler generates another bit vector as its output which is the input signal for the sparse distributed representation(SDR) classifier. The SDR classifier generates the future step predictions and anomaly score of the present input signal.

The second model comprises of two single layers of spatial pooler, temporary memory and SDR classifier. It is similar to the first model but it has an extra layer of spatial and temporal memory which can capture sequences which the first model may miss as it is less powerful. The temporal pooler learns the bit patterns in this input via temporal memory which consists of columns of neurons. These neurons forms new connections to other neurons and increase or decrease the strength of the connections between the already connected neurons when the learning mode is set on. There are chances of underfitting with model 1 whereas model 2 might overfit the data model capable of learning relatively more complex associations in the input signal, whereas Model 1 is useful for learning simple associations and Model 2 is the intermediate.

Figure 3: Model 1

Figure 4: Model 2 and 3

The third model comprises of two single layers of spatial pooler, temporary memory and SDR classifier along with a feedback connection between the layers of the temporal pooler. Model 3 is the most powerful

4.3 Workload Characterization

The workload generated by RUBiS clients uses the following probabilistic distribution for the requests made to the web server:

Action/Page Visited	**Probability**
Home	0.01
Register	0.01
Browse Items	0.58
Sell	0.05
About Me	0.15
End Of Session	0.2

Table 1: Probabilistic distribution of web requests

TPC workload characterizes around 80 percent of requests to be browse requests. A browse request is any request that reads data as such from the web server. Our workload probabilistically makes a browse request around 75 percent of the time.

5 Results and Analysis

Unsupervised Anomaly Prediction summary: SDR Classifier computes a raw anomaly score that measures the deviation between the model's predicted input and the actual input. The raw anomaly score will be 0 if the current input is perfectly predicted, 1 if it is completely unpredicted, or somewhere in between depending on the similarity between the input and the prediction. If two completely different inputs are both possible and predicted, receiving either input will lead to a 0 anomaly score. Any other input will generate a positive anomaly score. Changes to the underlying system are also handled gracefully due to the continuous learning nature of HTMs. If there is a shift in the behavior of the system, the anomaly score will be high at the point of the shift, but will automatically degrade to zero as the model adapts to the "new

normal". This works well for predictable scenarios but in many practical applications, the underlying system is inherently noisy and unpredictable. In these situations it is often the change in predictability that is indicative of anomalous behavior. Rather than thresholding the raw score directly, we use anomaly likelihood value from SDR classifier which is based on the maximum likelihood principle by using the prediction history of the HTM model. A single spike in input will not lead to a significant increase in anomaly likelihood value but a series of spikes will. Hence it is more stable and ideal to be used for anomaly detection on noisy data. We define the system state to be anomalous when,

$$\text{Anomaly Likelihood Score} \geq 1 - e$$

where e is a value we select for the model.

The prediction model works as described below:

1. The models first learns on the training data. The inference/learning mode of spatial pooler and temporary memory is enabled during training. The model is configured using model params, to provide 7 steps ahead predictions as well. The model is trained on non-anomalous data. At the end of the training phase, the models are saved in pickle files for later use.

2. In the testing phase, the models are recreated from the pickle files. Each model consumes the current input from the stream(with inference mode ON), and provides the next 7 steps predictions. Then each model is fed the predicted 7 steps(with inference mode OFF), which gives us the future anomalous score values. When any of the future anomaly scores are greater than anomaly score threshold, then the model predicts an anomaly.

3. In the testing phase, CPU anomaly is introduced at different instances in the testing stream data. We injected this bug 53 times during the half hour RUBiS simulation. In CPU anomaly, we know that it takes 50 secs on an average to make the current state anomalous. Say the CPU anomaly was introduced at t1, then after t1+50 sec, SLO violations will occur. We observed this value (50 sec) by running it many time and observing the SLO feedback stream.

4. The main thread maintains a list of previous predictions to keep track of when the model predicted an anomaly. Using the SLO Violation feedback, the thread updates the prediction list and continuously classifies the predictions into true positive, false positive, true negative or false negative prediction.

5. Average lead time is the measure of the time lead provided by advance alarms in the true positive cases. For example, let's say the CPU anomaly was deployed in the webserver at time t1, and the model raised an alarm at t1+x seconds. Then the lead time provided by the model was 50 - x (50 secs = Max lead time for CPU anomaly). We take the average of this quantity for all true positive cases and we call it the average lead time.

Note - A prediction is TP, if the model predicted anomaly it at t1, and SLO violation(s) occur during (t1, t1+50). A prediction is FP, if the model predicted anomaly it at t1, and no SLO violations occur during (t1, t1+50). A prediction is TN, if the model does not predicted any anomaly at t1, and no SLO violations occur during (t1, t1+50). A prediction is FN, if the model does not predicted any anomaly at t1, and SLO violation(s) occur during (t1, t1+50).

Results: Model 1 seems to be working fine. But Model 2 and Model 3 predict anomaly very frequently(very high percentage of False Positives). Model 2,3 very frequently give anomaly scores close to 1 even for non-anomalous data.

Note - The red data point in the ROC curve highlights the position at which the maximum accuracy of the model is achieved in Model 1 in the figure below.

Model 2, 3 were found having a very high false positive rate throughout the ROC experiments.

Model 1:

Anomaly Score Threshold = 0.98

Avg Lead Time 12.0 sec

Model Accuracy : 95.91 %

Model 2:

Anomaly Score Threshold = 0.99

Avg Lead Time 43.77

Model Accuracy: 4.74 %

Model 3:

Anomaly Score Threshold = 0.99

Avg Lead Time 46.45

Model Accuracy: 4.75 %

6 Conclusion

Our results show that HTM can achieve high prediction accuracy and raise advance alarms with average 12 seconds lead time. HTMs are quick to train and takes around 20-50 ms for prediction, which makes it practical for large-scale cloud computing infrastructures.

7 Future Works

In addition to predicting the anomalies, it is also imperative to take measures to maintain an optimum system state. RUBiS, this would mean adding the necessary nodes (web or database server) in the cluster to mitigate the increased workloads. We can leverage AWS APIs to create an action service in conjunction to the prediction service that would take an appropriate action when anomalies occur.

AWS EC2 Autoscaling service allows its users to set health checks on nodes within an autoscaling cluster. This is a reactive approach to scaling and our proposed solution provides an alternative scaling mechanism. Moreover, AWS EC2 Autoscaling service can also scale down if the workload decreases. This provides us with another challenge that involves predicting low request rates and taking some action to scale down resources.

Another improvement to our solution would be to consider other system metrics such as memory, I/O and network to characterize the performance. This would make the prediction service much more holistic.

References

[1] An Overview of Hierarchical Temporal Memory: A New Neocortex Algorithm http://ieeexplore.ieee.org.prox.lib.ncsu.edu/stamp/stamp.jsp?tp=&arnumber=6260285

[2] Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory https://arxiv.org/ftp/arxiv/papers/1503/1503.07469.pdf

[3] Continuous Online Sequence Learning with an Unsupervised Neural Network Model https://www.mitpressjournals.org/doi/10.1162/NECO_a_00893

[4] Overview of Temporal-Memory-Algorithm https://numenta.com/assets/pdf/temporal-memory-algorithm/Temporal-Memory-Algorithm-Details.pdf

[5] RUBiS: Rice University Bidding System. http://rubis.ow2.org

[6] Infrastructure as Code: AWS Cloudformation https://d1.awsstatic.com/whitepapers/managing-your-aws-infrastructure-at-scale.pdf

[7] Amazon elastic compute cloud. http://aws.amazon.com/ec2/