Anomaly Prediction in Large-Scale Distributed Systems

Varsha Nagarajan
North Carolina State University
vnagara2@ncsu.edu

Rajat Narang
North Carolina State University
rnarang@ncsu.edu

Prashanth Mallyampatti
North Carolina State University
pmallya@ncsu.edu

ABSTRACT

Performance anomaly prediction is crucial for long running, large scale distributed systems. Many existing monitoring systems are built to analyze system logs produced by distributed systems for troubleshooting and problem diagnosis. However, inspection of such logs are non-trivial owing to the difficulty in converting text logs to vectorized data. This becomes infeasible with the increasing scale and complexity of distributed systems. Few other effective methods employ statistical learning to detect performance anomalies. However, most existing schemes assume labelled training data which require significant human effort to create annotations and can only handle previously seen anomalies. In this paper, we present two unsupervised anomaly prediction algorithms based on Self Organizing Maps and Long Short-Term Memory networks. We implemented a prototype of our system on Amazon Web Services and conducted extensive experiments to model the system behavior of Cassandra. Our analysis and results show that both these algorithms pose minimal overhead on the system and are able to predict performance anomalies with high accuracy and achieve sufficient lead time in the process.

KEYWORDS

Distributed Systems, Anomaly prediction, Self-organizing maps, Long Short-Term Memory, neural networks

1 INTRODUCTION

Anomaly detection systems are an integral part of any complete package of well-maintained distributed systems. It is the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data. A dynamic environment like a production process contains information that is changing with time, which needs to be understood in order to gain insights into the behaviour of the system so that we can flag any irregularities that happen. Systems evolve with time and we need techniques that can intelligently model this environment. We need these anomalies to be detected early on so that preventive actions can be taken. Hence, our model needs the ability to detect subtle changes in patterns that are not apparent. We need to be able to do all this in an unsupervised manner because anomalous events can be new and so we cannot depend on pre-programmed thresholds.

Researches have been going on in this field to identify the anomalies at the earliest and if possible, give an outline for its repair. Nowadays machine learning is used as a tool for detecting and analyzing network intrusion, CPU usage anomalies, throughput factor, systems logs, and many other performance and system metrics. New algorithms are being developed consistently, where each address a different aspect of anomaly detection. Some of the most notable approaches are around system diagnosis and monitoring using console logs/system traces, and raw system metrics. Since the earliest days of software, developers have used free-text console logs to report internal states, trace program execution, and report runtime statistics. Since log files are obtained with very minimal overhead, it has been an efficient way to report anomalies in some systems. But further researches have shown that system logs might not always indicate all the states and can be subject to issues such as multiple interpretation, difficulty to convert text logs to vectorized data, difficulty in deciphering multithreaded system logs and high log content to handle (for large systems) to name a few.

To overcome these issues, our model should be able to capture temporal and spatial dependencies in each metric's (CPU utilisation, memory usage etc.) time series. It should be able to encode inter-correlations between different pairs of time series and should be robust to noise. Since modern systems have got auto-recovery capabilities and robustness to short term anomaly caused by temporal turbulence, not all anomalies lead to failures. Therefore, it would be good to have different levels of anomaly scores based on the severity of various incidents. Using raw system metrics such as CPU usage, network hogs, memory leaks, I/O interactions provides a sophisticated platform for such analysis. Any anomalous behavior should account for at-least one of the above metrics, thus the unknown category of anomalies is reduced significantly. Small daemon processes can be deployed on host systems to collect the metric data from distributed systems (some come with inbuilt metric collector) with very less overheads. These system metrics are quantifiable and can be easily fed into machine learning algorithms for automated analysis. The predictive power of such learning algorithms resulted in increasing focus on machine learning algorithms for improved and accurate anomaly detection and prediction. We will primarily be exploring the potential of LSTMs and Self-Organizing Maps for unsupervised anomaly detection.

We implemented LSTMs, exploiting their property of detecting and remembering short term patterns over long periods of time and assess its predictive power. Long Short-Term Memory Units, or LSTMs, was proposed by German researchers Sepp Hochreiter and Juergen Schmidhuber [11] as a solution to the vanishing gradient problem. LSTMs preserve the errors that have been seen and can be back propagated. This back propagation happens through time and layers. A cell state, forget gate, and a output gate form an LSTM cell which operate on various activation functions. LSTMs makes use of these gates to decide if it should keep or forget information. So when a LSTM sees a new data coming in, it tries to forget any long-term information that it decides it may no longer need. Then it learns which parts of the new input are worth using, and saves them into its long-term memory. After this, LSTM learns to focus on the selective parts of the saved long-term memory which can be brought in as working memory for immediate use for future predictions. Using this property of LSTMs, they can also be modeled as the encoder and decoder components of Autoencoders. An autoencoder primarily focuses on reconstructing the input fed to it and by using LSTMs, we are able to feed in sequences of temporal data and model their reconstruction with minimum error. This motivated us to use LSTM Autoencoders to capture and model normal system behavior thus enabling them to effectively detect even the slightest of perturbations. Such perturbations, based on certain thresholds/tolerance, could be flagged as an anomalous event.

Self-organizing map (SOM) is an unsupervised deep leaning algorithm that was proposed by Kohenen [12] and is based on Winner-Take-All algorithm. It maps a high-dimensional space to a low dimensional map space (usually two dimensions) while preserving the topological properties of the original input space. The idea behind this was the intuition that neural networks could partition into different indicator regions based on the external input. These regions could correspond to different states in the system and the euclidean distance of the new observations from these partitions (BMUs: Best Matching Units) would be used as an anomaly indicator.

Both our approaches do not require any labelled training data, allowing us to perform anomaly predictions in an unsupervised fashion. We only rely on the collected low-level system metrics for making our predictions. We leverage the fact that before a failure actually occurs, there are unusual changes in system metrics which provide hints about possible unexpected or anomalous events. By analyzing these early deviations from normal system behaviours, we make our predictions.

A prototype of our system was deployed on EC2 instances provided by Amazon Web Services. We conducted extensive experiments using Cassandra as our underlying distributed system. Our experiments show that SOMs are able to achieve a recall of 87% and precision of 82% for CPU faults and a recall of 78% and precision of 71% for memory faults with an average lead time of 4 seconds for CPU and 5 seconds for memory faults. LSTM Autoencoders, on the other hand, are able to achieve a recall of 88% and precision of 79% for CPU faults and recall of 77% and precision of 70% for memory faults with an average lead time of 12 seconds for CPU and 10 seconds for memory faults. We observe that both our proposed algorithms perform anomaly predictions with good accuracy and sufficient lead time.

The rest of the paper is organized as follows: Section 2 presents the design details of LSTM and SOM outlining the approach, algorithm, training procedure, data prepossessing, hyper-parameter tuning, unsupervised anomaly prediction and anomaly cause inference. Section 3 presents the experimental evaluation. Section 4 discusses the related work and finally section 5 concludes the paper.

2 SYSTEM DESIGN

In this section, we provide a detailed overview of the two proposed algorithms for anomaly prediction in distributed systems. We first describe our unsupervised anomaly prediction algorithms and how it applies to the problem we seek to address. Next, we discuss about the training and subsequent alerts and inference mechanisms employed.

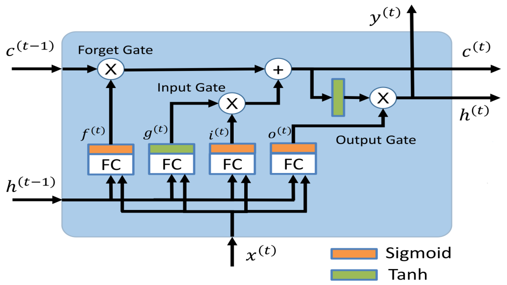


Figure 1: LSTM Cell

2.1 Long Short-Term Memory

Tools and algorithms designed for monitoring distributed systems should be scalable and must provide fast predictions. An anomaly, not predicted well ahead of time, does not provide ample scope for mitigation or for employing corrective measures. Achieving a combination of scalable, highly available, responsive and accurate performance anomaly prediction systems is a very challenging task. We first need to model a learning scheme that does not pose a significant overhead on the system. It must be scalable in order to induce behavior models for capturing patterns across a wide range of system metrics. Moreover, the system metrics collected from real-world distributed applications are often fluctuating due to dynamic workloads or noise interference and hence requires a robust learning scheme. Long Short-Term Memory networks are able to accurately model such complex patterns in data and are less susceptible to noisy interference. They have provided state-of-the-art results in many tasks involving modeling sequential data and so we chose to work with this algorithm and evaluate its efficiency in making real-time anomaly prediction.

2.1.1 ***LSTMs.*** Long Short Term Memory networks, popularly referred to as "LSTMs", are a special kind of recurrent neural network, capable of learning long-term dependencies in data. These networks are very powerful in modeling temporal data and hence are naturally suited for modeling dynamic system behavior by inferring patterns in resource usages. They have three gates that regulate the flow of information through the network and their cell states enable them to capture and retain the latent representation of important features which could subsequently be applied to making accurate predictions. Figure 1 shows how a simple LSTM cell looks like.

2.1.2 ***Approach.*** Our approach operates on a very simple premise that any anomalous usage pattern is not a normal system behavior. We essentially try to identify instances or patterns in data that deviate from normal behavior. As a solution, we propose an LSTM-Autoencoder model. An autoencoder is an unsupervised deep learning algorithm which consists of an encoder and a decoder that seeks to learn a lower-dimensional latent representation of the input that can be used for nearly lossless reconstruction of the input pattern. The network aims on learning this latent representation by trying to minimize the reconstruction errors by backpropagating gradient updates. In order to model temporal system metrics data, we design an LSTM Autoencoder (ref. Figure 2), which functions very similar

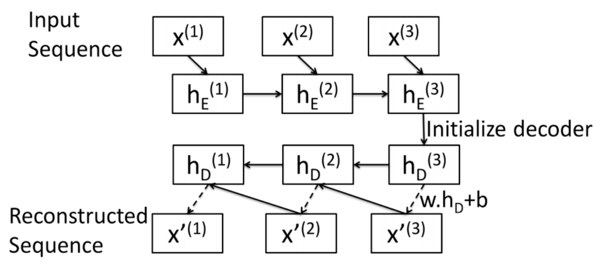


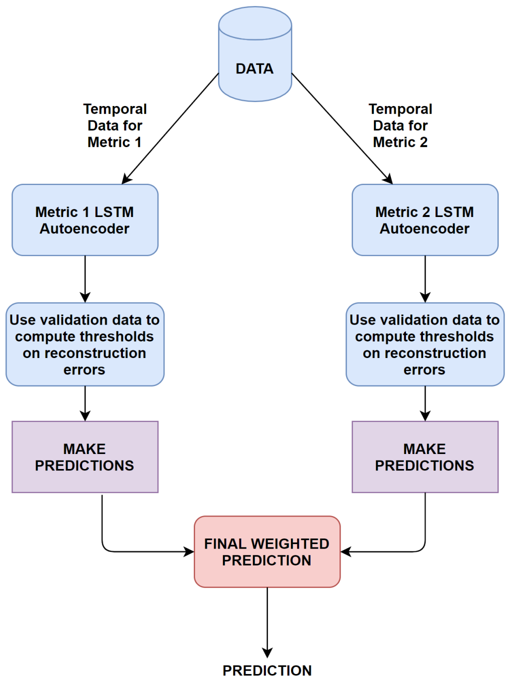
Figure 2: LSTM Autoencoder

to the usual autoencoders, with the only difference being that the encoder and decoder components are LSTM networks instead of multilayer perceptrons. The encoder learns a vector representation of the input time-series and the decoder uses this representation to reconstruct the time-series with the objective function being the minimization of the reconstruction errors.

The intuition here is that since the algorithm operates in an unsupervised fashion, the model will be trained only on normal usage patterns i.e., patterns which are expected to be seen in the normal functioning of the system. Hence, the LSTM encoder-decoder pair would have only seen normal instances during training and would have learnt to reconstruct them. When fed an anomalous sequence, it might not be able to reconstruct it well, leading to higher reconstruction errors compared to the reconstruction errors for the normal sequences. By identifying a threshold on the reconstruction error tolerance, we make our predictions.

2.1.3 ***Algorithm.*** In order to effectively model emergent system behaviors in large-scale distributed systems, we collect temporal data of usage patterns of various system resources. A dedicated LSTM Autoencoder model is spawned for each of the system metrics and a weighted prediction is used to identify anomalous patterns in the resource usages. Figure 3 provides a detailed overview of our proposed model architecture. Often times, in distributed systems, effects of only certain features manifest themselves and can be termed as potential features to "watch-out" for sending alerts. For example, certain systems can be I/O intensive or memory-intensive, making I/O or memory their potential features to look out for. In such cases, notifying the algorithm to watch out for these metrics could result in effective anomaly predictions. Hence, we propose an algorithm that is tailored to model temporal data for each of the system metrics individually and then a weight is assigned to each of these metrics to mark their importance. These weights can be configured based on the distributed system that is being monitored. Thus, we propose a flexible model that can be extended to any distributed system environment.

2.1.4 ***Training Procedure.*** We jointly train both the encoder and decoder components of the LSTM Autoencoder to reconstruct the input sequences which represent usage patterns of normal system behavior. The input sequences, in our case, are system metrics data which indicate resource usage trends over time. The encoder learns a fixed-length vector representation of the input sequence and the decoder uses this representation to reconstruct the sequence using



DATA
Temporal Data for Metric 1
Temporal Data for Metric 2
Metric 1 LSTM Autoencoder
Metric 2 LSTM Autoencoder
Use validation data to compute thresholds on reconstruction errors
Use validation data to compute thresholds on reconstruction errors
MAKE PREDICTIONS
MAKE PREDICTIONS
FINAL WEIGHTED PREDICTION
PREDICTION

Figure 3: LSTM Autoencoder Anomaly Predictor Model

the current hidden state and the values predicted at the previous timestep.

Consider a usage pattern $X = \{x^{(1)}, x^{(2)}, ..., x^{(N)}\}$ of length N, where each point $x^{(i)}$ indicates a value of system metric at a given timestep. We train the model to reconstruct this sequence $X' = \{x'^{(1)}, x'^{(2)}, ..., x'^{(N)}\}$, with the objective function being the minimization of the below reconstruction loss:

$$\sum_{X \in S} \sum_{i=1}^{N} ||x^{(i)} - x'^{(i)}||^2 \tag{1}$$

where S is the chunks of input sequences fed to the model.

These reconstruction errors are then used to obtain a likelihood of incoming pattern being anomalous. In order to do so, the training phase is essentially divided into two parts. First, the model is trained on resource usage patterns collected during normal system behavior. After the training concludes, the normal resource usage patterns collected further on are fed to the model and their reconstruction errors are used to estimate the parameters of the Normal Distribution $\mathcal{N} = (\mu, \sigma)$ using maximum likelihood estimation. Post this, three standard deviations away from the mean of this normal distribution is used as a tolerance for observed reconstruction errors in new sequences. If the model's reconstruction errors for predictions made on the new incoming pattern lie beyond three standard deviations of the mean as identified above, an alert is flagged. The number of standard deviations away from the mean that could be tolerated without flashing an alert was treated as a hyperparameter in our analysis.

2.1.5 ***Data Preprocessing.*** Before the sequences are fed to the model, they are preprocessed by employing feature-wise standard

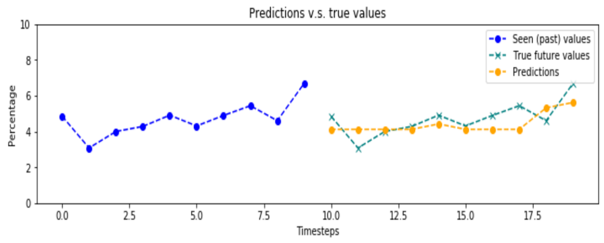


Figure 4: LSTM-Autoencoder: Reconstructed Sequences for CPU usage patterns

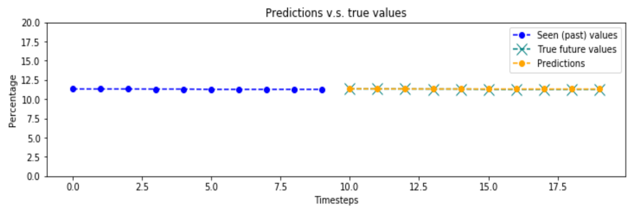


Figure 5: LSTM-Autoencoder: Reconstructed Sequences for Memory usage patterns

scaling. This is essential in order to avoid bias towards larger values. Certain metric data might exhibit huge usage values while other metrics might have relatively smaller values. In order to remove such bias, all data points are standardized before being fed to the model. From the purview of LSTMs, standardizing values makes gradient calculations simpler, thus speeding up the learning process.

2.1.6 ***Hyperparameter Tuning.*** Hyperparameter tuning is an integral component while designing any machine learning algorithm. Not all parameters work the same in different usecases, hence fine-tuning of parameters often result in superior performance. For this reason, we conducted extensive experimentation to analyze the optimum number of LSTM layers, bidirectional vs unidirectional LSTMs, number of LSTM units in each layer, optimizer function, learning rate, number of epochs for which the model must be trained and the window size for the input sequences.

We observed that using bidirectional LSTMs resulted in lower reconstruction errors and the patterns in input sequences were captured accurately. Figures 4 and 5 show the predicted values plotted against true future values. On experimenting with the number of LSTM layers, we found 2 layers in the encoder and 1 layer in the decoder to be optimum. Fewer layers were underfitting and more layers did not offer substantial improvements for the increased training time. In case of learning rates we observed that with higher learning rates, the validation accuracy and losses over the epochs were pretty bumpy, indicating steep jumps across the local minima resulting in delayed and sometimes sub-optimal convergence. Adam optimizer with a learning rate of 0.001 seemed to work best in our case. Figures 6 and 7 show the validation and training losses observed. As for the number of epochs, we trained our model for 50 epochs to note the trend of validation accuracy and losses and observed that beyond some 20-25 epochs, the model begins to overfit. Hence, we use early stopping with a patience of 10 epochs as a criterion to decide exactly when to stop training. We also carefully investigated the optimal batch size and window length and fixed on a batch size of 32 and window length of 60 timesteps that seemed to work best in our case. Table 1 provides a comprehensive list of the range of values tried for different hyperparameters.

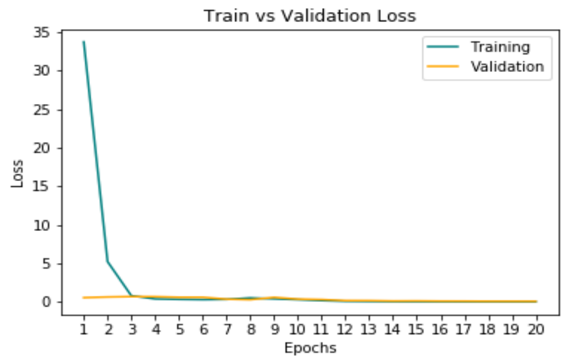


Figure 6: Training and Validation Loss (Memory)

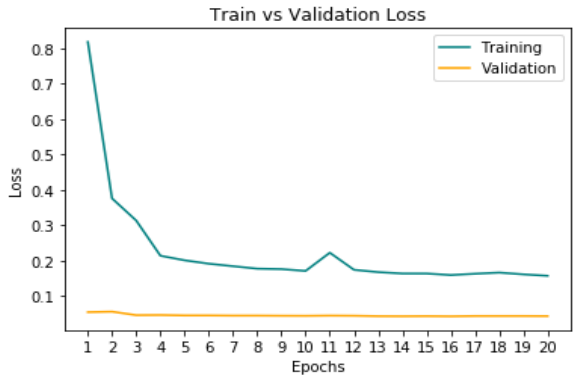


Figure 7: Training and Validation Loss (CPU)

Table 1: Hyperparameter Tuning Ranges for LSTM Autoencoder

Hyperparameters	***Range***
Learning Rate	0.01, 0.001, 0.0001
Optimizer	RMSProp,SGD, Adam
Epochs	15, 20, 30, 50
Batch Size	16, 32, 64, 128
Window Size	5, 10, 20, 60, 120

2.1.7 ***Unsupervised Anomaly Prediction.*** In distributed systems, the actual occurrence of performance anomalies is usually preceded by certain anomalous changes exhibiting themselves during system operation. Possible indicators could be a gradual increase in the CPU or memory usage values. A fault does not cause an SLO violation immediately. Often times, there is a time gap between the occurrence of a fault and the violation of service-level objectives.

We aim to design prediction models that could allow the system administrator to take full advantage of this time-gap by providing early-on predictions with sufficient lead time.

In order to identify performance anomalies, the online LSTM Autoencoder model continuously analyzes the stream of resource usage patterns and tries to reconstruct these input sequences. The reconstruction errors are then used to obtain a likelihood of incoming pattern being anomalous. As already discussed, the reconstruction errors observed during the second part of the training phase are used to estimate the parameters of the Normal Distribution $\mathcal{N} = (\mu, \sigma)$ using maximum likelihood estimation. Post this, three standard deviations away from the mean of this normal distribution is used as a tolerance for observed reconstruction errors in new sequences. If the model's reconstruction errors for predictions made on the new incoming pattern lie beyond three standard deviations of the mean as identified above, an alert is flagged. The idea is that if our model was able to reconstruct the input sequence with very low reconstruction error, then it must have probably seen such sequences before. Since the model is essentially trained on normal data, this would mean that the observed sequence indicates normal system behavior. However, if the reconstruction error is huge, it would mean that our model has not seen such sequences before, indicating a possible anomaly that requires attention.

2.1.8 ***Anomaly Cause Inference.*** In addition to sending alerts for notifying the system administrator about possible performance anomalies, narrowing down the scope of the problem by identifying the affected system and providing hints about the possible root cause could be extremely beneficial. Hence, we designed our model keeping both anomaly prediction and anomaly cause inference as prime objectives which made such inferences possible.

As already mentioned, we propose a system that is tailored to model temporal data for each of the system metrics individually using separate LSTM Autoencoder models. Using the predictions made by each of these individual models, the root cause metric can be easily identified. Additionally, we note the individual deviations in reconstruction errors for this metric data for each of the nodes in the distributed system and using this information, possible affected systems can also be pin-pointed. Thus, we propose a system that is not only effective in making accurate anomaly predictions but also provides a robust fault localization and cause inference mechanism.

2.2 Self Organizing Maps

2.2.1 ***Modeling System Behavior.*** Self-organizing map (SOM) is an unsupervised clustering algorithm that was proposed by Kohenen and is based on Winner-Take-All algorithm. It maps a high-dimensional space to a low dimensional map space (usually two dimensions) while preserving the topological properties of the original input space. The idea behind this was the intuition that neural networks could partition into different indicator regions based on the external input. These regions could correspond to different states in the system and the euclidean distance of the new observations from these partitions (BMUs: Best Matching Units) would be used as an anomaly indicator. Moreover, SOMs are computationally inexpensive, hence ideal for a real production system. Since SOM is an unsupervised deep learning technique, it also does not require any labelled training data.

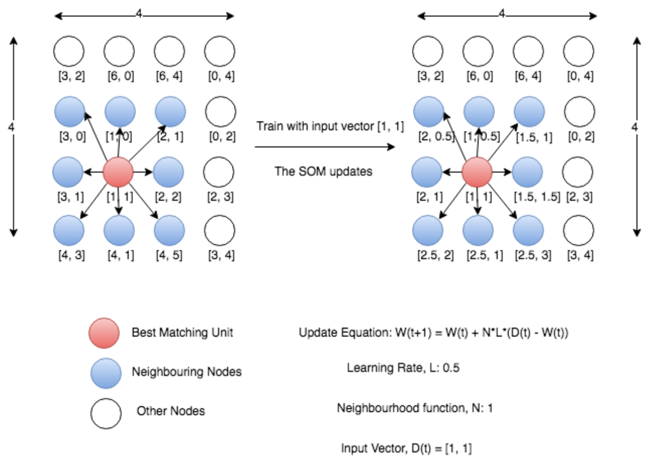


Figure 8: SOM training process

2.2.2 ***Algorithm.*** For each of the nodes in our distributed system, we continuously collect system metrics, formulate them into a vector representation of the form D(t) = $[x_c, x_m]$, where x_c and x_m denotes two example system metrics. These vectors are then fed as inputs to train and subsequently test our SOM model.

An SOM is composed of a set of nodes arranged in a matrix as shown in the Figure 8. Each node has a weight vector and a coordinate in the map. Weight vectors should be of the same length as the sample input vectors, because nodes are essentially a generalization of the input we have used to train the SOM. Weight vectors are updated based on the values of the sample input vectors in the training data. Self-organising maps have two different phases to model system behaviours: training and mapping. We first describe the training (learning) phase.

2.2.3 ***Training Procedure.*** During training, SOMs uses a competitive learning process to adjust the weight vectors of different nodes. The competitive learning process works by comparing the Euclidean distance of the input measurement vector to each node's weight vector in the map. The node with the smallest Euclidean distance is selected as the best matching unit. The values of that node along with its neighbouring nodes are then updated, thus exhibiting a classic "Winner-Take-All" methodology. The general formula for updating the weight vector of a given node at time t is given by the Equation 2. We use $W(t)$ and $D(t)$ to define the weight vector and the input vector respectively at time instance t. $N(v, t)$ denotes the neighbourhood function (e.g., a Gaussian function) which depends on the lattice distance to a neighbour node v. $L(t)$ denotes a learning coefficient that can be applied to modify how much each weight vector is changed as the learning proceeds.

$$W(t+1) = W(t) + N(v,t)L(t)((D(t) - W(t)) \qquad (2)$$

When each input vector has been used to update the map multiple times, learning is complete. At this point, the weight vectors of nodes represent a generalization of the whole measurement vector space.

2.2.4 *Data Preprocessing.*

When applying SOM to a real learning system, several metric preprocessing challenges must be addressed in order to achieve efficiency. Different system metric values can have very different ranges in their raw form. For instance, in an example distributed system, memory usages might vary between 8GB and 13GB whereas CPU usage could range from 0 to 100. In order to remove the bias towards large values, we normalize our incoming data to a standard scale with values restricted to lie between 0 and 1.

2.2.5 *Setting Hyperparameter and Initialization.*

Before we start using our SOM model for training and making predictions, a decision has to be made on the optimal size of the map that must be used for effectively modeling the normal system behaviour. A rule of thumb is to set the size of the grid for the dimensionality reduction task to contain $\lfloor 5 * sqrt(N) \rfloor$ nodes, where N denotes the number of incoming samples that are to be analyzed. The size of the grid is set to the power of 2 closest to above number. E.g. if your dataset has 150 samples, $\lfloor 5 * sqrt(150) \rfloor = 61$ and the closest power of 2 is 64. Hence, a map 8-by-8 should perform well. After experimentation with different values, we observed that a learning factor of 0.7, having the neighbourhood of each node to be 4, and setting the neighbourhood to be a Gaussian gives the best results. We initialize the weight vectors of the SOM by randomly sampling from the incoming data.

2.2.6 *Unsupervised Anomaly Prediction.*

In distributed systems, before the performance anomalies manifest themselves, we observe anomalous changes in system metrics such as CPU usage or memory usage, with these values suddenly exhibiting a gradual increase. A fault doesn't immediately cause a service-level objectives (SLO) violation and often times there is a time gap from when a fault occurs and an SLO violation is observed. Therefore, at any given time, a system can be thought to be operating in one of three states: normal, pre-failure, or failure. The system typically first enters the pre-failure state before entering the failure state. Figure 9 demonstrates the movement of the system from a pre-failure state to a failure state and we note that the characteristic of SOM to maintain the topological properties of nodes helps us in making accurate anomaly predictions. We can notify the system administrator, i.e., raise an alarm, when the system leaves the normal state but has not yet entered the failure state, so that they can take the necessary preventive action.

We compute the neighbourhood area size for each node in the SOM to decide which node represents what system state. Since SOM uses competitive learning, during training, when we update the weight vectors of a particular node (the best matching unit), we also update the weights of its neighbours. This implies that weights of frequently trained nodes will be very similar to the weights of their neighbouring nodes. Since a system operates in a normal state most of the time, majority of our training data also has inputs samples which represent normal states. This leads to nodes representing normal states to be trained more often in comparison to nodes representing pre-failure or failure states and this results in formation of clusters of nodes representing different normal system behaviours. We calculate a neighbourhood area size value for each node by examining their immediate neighbours. As our lattice topology is a two-dimensional grid, this means we examine the

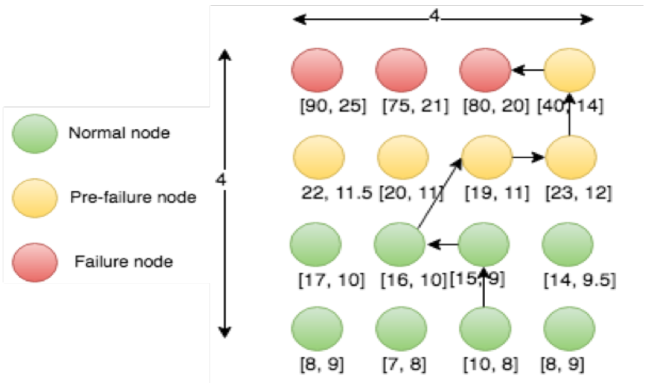


Figure 9: Example path showing system evolution from normal to anomaly

top, left, right, and bottom neighbours. We calculate the Manhattan distance between two nodes N_i, N_j with weight vectors $W_i = [w_{1,i}, ..., w_{k,i}]$, $W_j = [w_{1,j}, ..., w_{k,j}]$ using Equation 3.

$$M(N_i, N_j) = \sum_{l=1}^{k} (w_{l,i} - w_{l,j})^2 \tag{3}$$

Neighbourhood area size for a node N_i is calculated as the sum of Manhattan distance between the node N_j and its top, left, right and bottom immediate neighbours denoted by N_T, N_L, N_R, N_B, as it defined in Equation 4.

$$S(N_i) = \sum_{X \subseteq \{N_T, N_B, N_R, N_L\}} M(N_i, X) \tag{4}$$

A node is determined as normal or anomalous by looking at the neighbourhood area size of that node. If the neighbourhood area size is small, the node is believed to be in a tight cluster of nodes, which would essentially mean that the node represents some normal state. On the other hand, a large neighbourhood area value would imply that the node is not close to other nodes and hence could represent a possible anomalous state. We shall now discuss how we defined these small area size and large area size for our map.

If the threshold is set too high, then based on the configurations of our proposed SOM model, we will raise an alarm only when the input samples map to a node that is farther away from other nodes and qualifies so based on the above defined threshold. Since the threshold is very strict, only large deviations will be considered for alarming the system administrator. This would lead to many false negative indications in our system and we might miss the identification of some true anomalies. On the other hand, if the threshold is too low, every small deviation would be considered a possible anomaly and our system would end up sending a large number of false positive indications. Additionally, neighbourhood area size values vary from application to application depending on the range of values in that particular dataset. To address this issue, we set the threshold value based on a percentile instead of a fixed value. We sort all calculated neighbourhood area size values and set the threshold value to be the value at a selected percentile. We found a percentile value of 85% is able to achieve good results across all datasets in our experiments.

2.2.7 ***Anomaly Cause Inference.***

We try to tackle the highly non-trivial task of determining the root cause of an anomaly by giving a hint as to what metrics are the top contributors. This provides a hint to the system administrator of where they can start looking when trying to debug the issue. In order to do so, we follow a methodology very similar to the one outlined in one of the popular researches [5] in the domain. We exploit the property of SOM which seeks to preserve the topology of the input space. The idea is to look at the difference between anomalous nodes and normal nodes, and output the metrics that differ most as faulty metrics. That is, when we map a measurement sample to an anomalous node, we calculate the distance from the mapped anomalous node to a set of nearby normal nodes. Here, it is necessary to avoid comparing with anomalous neighbour nodes as they represent unknown states and therefore may give incorrect anomaly cause hints. We examine the neighbourhood area value for each node first. If it is above our threshold, we ignore it and move on to the next node in our neighbourhood, because this implies that the node represents anomalous state, we are looking for nodes representing normal states. If we don't find any normal node in the anomalous node's neighbourhood, we expand our distance calculation to include more nodes in the map. In order to ensure we get a good representation of normal metrics, we select N normal nearby nodes.

Once a set of normal nodes has been found, we calculate difference between the individual metric values of each normal node and those of the anomalous node. Since the change can be positive or negative, we take the absolute value of the calculated difference. We then sort the metric differences from the highest to the lowest to determine a ranking order. After this process completes, we will have N metric ranking lists. Finally, we examine the ranking orders of each of the N rankings to determine a final order. To do this, we use majority voting. Each list votes for which metric it had identified as having the largest difference in values. We then output the metric with the most votes as the first ranked metric, the metric that has the 2nd most is the second ranked metric, and so on. To break ties we output the metric that happens to come first in the output list construction.

3 EXPERIMENTAL EVALUATION

3.1 Experimental Setup

Our underlying distributed system is Cassandra where a single cluster of 3 nodes was deployed on Amazon EC2 instances. These were *t2.xlarge* instances with 4 CPU cores and 16 GB RAM. A separate *t2.medium* instance was dedicated as a Monitoring System, which hosts the *Collectd* server, *Prometheus* and a *prediction engine*. The Cassandra cluster was set with a replication factor of 3, and number of tokens for each of the node was set to 256.

Each of the nodes in the Cassandra cluster has *collectd* clients running on them, which periodically collects system metrics for various resource usages and sends this data to the *collectd* server hosted on the monitoring server. All data sent to the *collectd* server are dumped into the time-series database offered by Prometheus for effective querying during the online prediction phase.

3.2 Workload Generation

Dynamic workloads were generated to test the performance of our proposed real-time anomaly prediction algorithms. We primarily focused on generating CPU and Memory workloads, which we found from our investigations, to be the potential bottlenecks for Cassandra. These workloads were generated using *Yahoo Cloud Serving Benchmark* (YCSB). We used this tool to induce dynamic workload affecting various system parameters such as CPU and Memory. We carefully administered different parameter specifications to generate the exact amounts of workload that could be sustained by our setup. Since our experiments were run with limited resources, careful consideration of various parameters was necessary to reproduce dynamic workloads close to a real production system without compromising the computational limitations of our setup. Following were the specifications we employed:

- **CPU:** By varying operation count and thread count we were able to generate a variety of CPU usage patterns similar to real-world production systems. Typical workload consisted of around 10000 reads and 10000 write operations with 4 threads operating in parallel.
- **Memory:** We triggered write-intensive workloads for generating memory usage patterns close to the real production system. Identifying the right combination of operation counts, threads and number of records that do not compromise our limited-resource setup was a challenge.

3.3 Fault Injection

Faults were injected into the system to check the correctness and responsiveness of our models in detecting them as an anomaly and alerting in real time. All faults were made to last for 1-2 minutes.

1) *CPU fault*: Record-read and record-update intensive faults were injected into the system using YCSB to cause CPU hogging (CPU fault). This fault was spawned on all 4 CPU cores with an increased thread count and it took approximately 30 seconds to achieve the desired fault.

2) *Memory fault*: Due to limited system resources, memory fault injection using YCSB was potentially difficult, as Cassandra was crashing persistently with less than 40% of memory usage. Hence, to exert pressure externally, we used *stress-ng* for memory fault injection. This fault caused large memory allocations, and high page swaps leading to high memory usage. We triggered this fault on each CPU core in isolation and also tried triggering it on a combination of cores. In order to effectively trigger this fault, we first find the available free memory and then generate a fault to occupy 98% of this identified free memory. We were able to inject this memory fault to its full specified capacity in approximately 25 seconds.

3.4 Evaluation Methodology

In cases where we have unbalanced representation of positive and negative classes, i.e., anomalous and non-anomalous classes, a standard receiver operating characteristic (ROC) curve is considered most effective in understanding the predictions made my the machine learning algorithms. This is particularly true while dealing with resource usage patterns during performance anomalies in distributed systems as such occurrences are way less in comparison to normal system behavior. Hence, we plot ROC curves, which essentially shows the trade-off between true positive rate and false positive rate, to evaluate the anomaly prediction accuracy of our model.

In order to attribute a prediction as a true positive or false positive prediction, we follow the same mechanism outlined here [5]. A prediction model is said to make a true positive prediction if it raises an anomaly alert at time t_1, and an anomaly indeed occurred at time t_2, where $t_1 < t_2 < W$, where W denotes the upper-bound of the anomaly pending time. Consequently, if the predictor model raises an alert and no anomaly occurs within the time $t_1 + W$, we tag this as a false positive. In addition to prediction accuracy, we also evaluate the effectiveness of our proposed model on basis of the achieved *lead time*, which can be thought of as how advance in time are we sending out an alert.

3.5 Results and Analysis

3.5.1 ***Prediction Accuracy Results.*** In this section, we present the anomaly prediction accuracy results for both our models. As discussed above, we plot the receiver operator characteristics (ROC) curve for accurate assessment of our model's performance. For our LSTM Autoencoder model, the ROC curve is plotted by varying the reconstruction error threshold ranging from 1 to 5 standard deviations away from the mean of the fitted Gaussian distribution. In case of our SOM based model, we acquire the ROC curve by adjusting the neighbourhood area size percentile threshold (i.e. 70th percentile to 98th percentile). In order to compare our results with other popularly used algorithms, we train a *k-NN* model in the exact same setting as our proposed models and evaluate the results. An ROC curve is plotted by adjusting the *kth* nearest neighbor distance threshold.

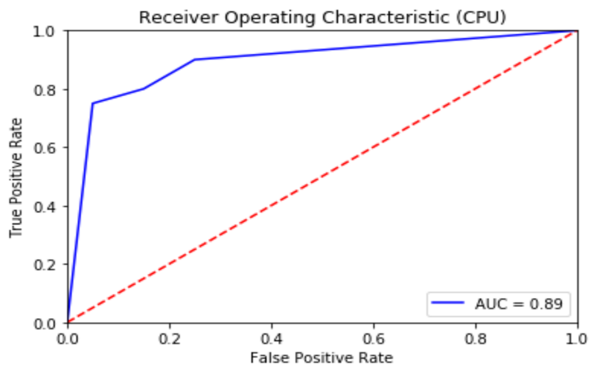


Figure 10: LSTM: Receiver Operating Characteristics (CPU)

Table 2: LSTM: Precision-Recall-F1

	Precision	Recall	F1 Score
CPU	0.789	0.882	0.833
Memory	0.7	0.77	0.733

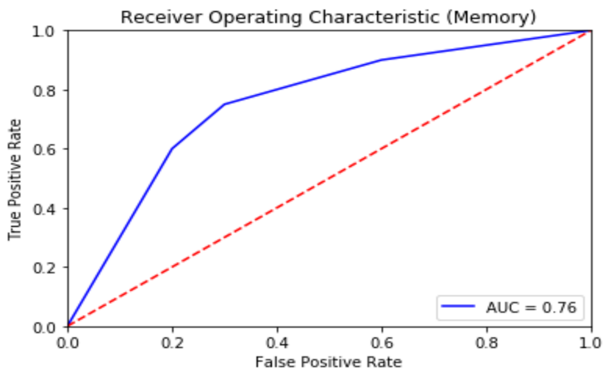


Figure 11: LSTM: Receiver Operating Characteristics (Memory)

Table 3: SOM: Precision-Recall-F1

	Precision	Recall	F1 Score
CPU	0.82	0.87	0.845
Memory	0.71	0.78	0.743

Figures 10, 11, 12 and 13 show the obtained ROC curves for our proposed models. For both our SOM and LSTM based models, we observe that predictions in case of injected CPU faults is fairly good, with a high true positive rate of about 80%. However, the models do not perform as expected in modeling and predicting memory faults, with true positive rate as low as 60%. One possible reason for this could be the inherent difficulty in modeling memory usage metrics of Cassandra, our case-study distributed system. The usage patterns remained consistent regardless of load generation, indicating that we might need really huge production systems for capturing the effect of the generated dynamic workload on the memory resource usage. With current resource limitations, this could not be explored but we do note it down as a future work.

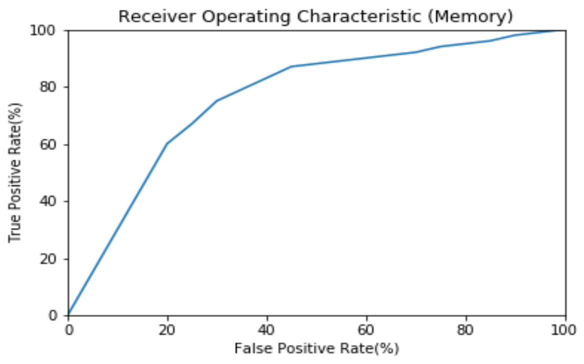


Figure 12: SOM: Receiver Operating Characteristics (Memory)

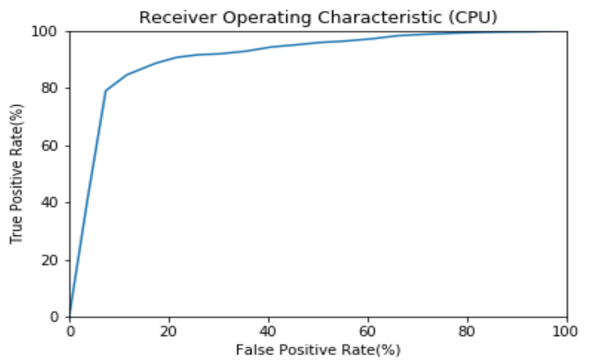


Figure 13: SOM: Receiver Operating Characteristics (CPU)

Coming to the prediction scores, Tables 2 and 3 outline the precision, recall and F1 scores for our models. We observe a high F1 score for both CPU and memory fault injection usecases. This indicates that our model is able to identify both normal system behavior and performance anomalies in almost all cases. We also compare our results with the baseline *k-NN* model and from the ROC curves in Figure 14 and 15, we can say that our proposed models are superior to *k-NN* based models when tasked with performance anomaly predictions. Additionally, Tables 4, 5, 6 and 7 provide the confusion matrix for our SOM and LSTM models. The difference in numbers across models is mainly because LSTMs identify anomalies across input patterns in a configured window size whereas predictions in SOM is made on every input datapoint. For SOM, best results were obtained when the neighbourhood area size percentile threshold was set to 85th percentile. And in case of LSTM Autoencoder, an encoder comprising of 2 LSTM layers and a one LSTM layer decoder, with ReLU activations resulted in best results.

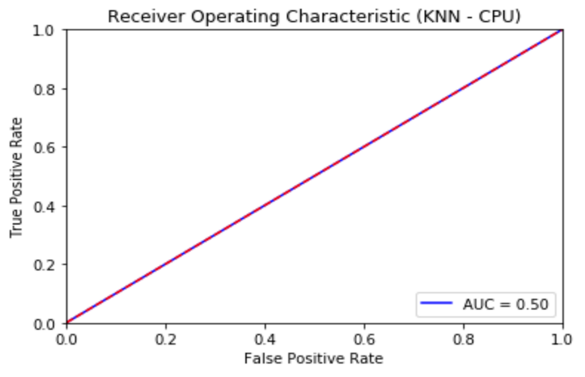


Figure 14: KNN: Receiver Operating Characteristics (CPU)

3.5.2 ***Lead Time Results.*** The observed lead time for the predicted performance anomalies for both CPU and memory fault injections are outlined in the Figure 16. For the LSTM Autoencoder model, we achieved an average lead time of 10 seconds for memory fault and 12 seconds for CPU fault, with a maximum lead time of 101 seconds (memory fault) and 98 seconds (CPU fault). Similarly, for SOM model, we observe an average lead time of 4 seconds for CPU faults and 5 seconds for injected memory faults. We observe that LSTM Autoencoder model provides a better lead time in comparison to SOM. This is because, LSTM models are able to effectively detect even the most minor perturbations in input sequence, enabling them to quickly and accurately identify deviations from normal patterns.

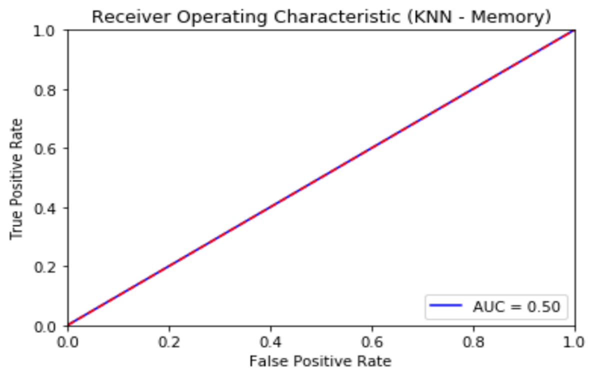


Figure 15: KNN: Receiver Operating Characteristics (Memory)

Table 4: SOM: CPU Faults Confusion Matrix

	Actual Anomaly	Actual Not Anomaly
Predicted Anomaly	104	26
Predicted Not Anomaly	23	330

Table 5: LSTM: CPU Faults Confusion Matrix

	Actual Anomaly	Actual Not Anomaly
Predicted Anomaly	15	4
Predicted Not Anomaly	2	21

Table 6: SOM: Memory Faults Confusion Matrix

	Actual Anomaly	Actual Not Anomaly
Predicted Anomaly	98	40
Predicted Not Anomaly	27	310

Table 7: LSTM: Memory Faults Confusion Matrix

	Actual Anomaly	Actual Not Anomaly
Predicted Anomaly	14	6
Predicted Not Anomaly	4	22

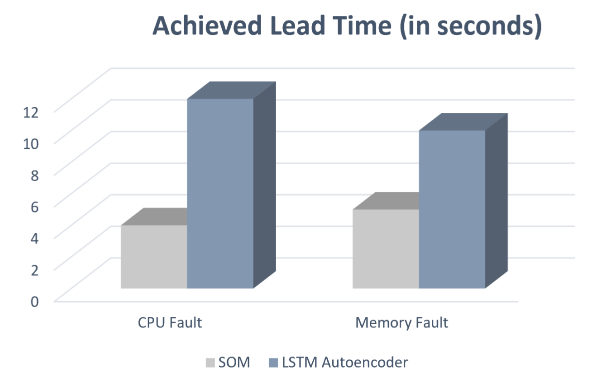


Figure 16: Achieved lead time for SOM and LSTM (y axis represents lead time in seconds)

3.5.3 ***Anomaly Cause Inference and Fault Localization.*** As mentioned earlier, anomaly cause inference and fault localization can be very helpful for quick recovery of affected systems. Hence, we assess our model's performance from this aspect. For LSTM Autoencoder, we observed an accurate root cause inference in all our conducted experiments. The reason for this effectiveness is our model architecture. In order to accurately model the temporal data for each of the system metrics, separate LSTM Autoencoder models are spawned for each of the system metric, which effectively capture resource usages patterns for all systems in the distributed environment. Thus, for any and all anomalous perturbations, the root cause metric can easily be inferred by identifying the LSTM model that made the prediction of an anomaly. For example, if an LSTM model capturing CPU usage patterns predicts an anomaly, CPU will be considered as a possible root cause. Additionally, by identifying the individual deviations in reconstruction errors for a given metric data for each of the nodes in the distributed system, fault can be localized to specific nodes that exhibited huge reconstruction errors. Due to this flexibility, our proposed LSTM Autoencoder model was able to pin-point the exact anomalous nodes in all of our experiments. However, we did see quite a false positive node indications in normal system operations mainly because of the inherent difficulty in modeling Cassandra memory usage patterns. Our SOM model was also able to correctly identify the faulty metric in most failure cases. This is due to characteristic of SOM to preserve the topological properties of the input space which proves to be useful not just for prediction, but for diagnosis as well.

3.5.4 ***System Overhead and Prediction Time.*** In order to pose minimal overhead on all systems in the distributed environment, we designed a monitoring system where the monitoring server is isolated from the distributed system it monitors. By dedicating a separate server for anomaly predictions, we are able to achieve a system that poses no overhead on the distributed environment. As for agents running on the distributed system for collecting system metrics, they pose negligible overhead. Also, we observed that our proposed algorithms use only 0.4% of CPU and 0.2% of memory of the monitoring server, when in operation. The time taken by our models to make predictions are also minimal, with SOM model taking 0.176 seconds and LSTM Autoencoder model taking 0.56 seconds. While LSTMs are fairly complex models, tuning them to reduce their prediction time is pretty challenging and it can decrease only so much and it might, in all cases, be more than our SOM model. But, we believe that the increased *lead time* offered in case of LSTMs compensates for the excess time taken in making predictions.

4 RELATED WORK

The idea of using machine learning methods for anomaly detection and prediction in large scale systems is not something new. Researches conducted in this domain can be broadly categorized into supervised, semi-supervised and unsupervised learning methods. Supervised approaches learn a classifier using labeled instances belonging to healthy and faulty categories and then predicts the category of the test instance. This method relies on the trained classifier to accurately identify known and unknown anomalies by observing the unseen data. Semi-supervised methods construct a model from the given training dataset that represents healthy behavior and test the model's likelihood of generating the observed test instance. Unsupervised techniques learn models using unlabelled data and detect anomalies under the assumption that majority of the training instances indicate healthy behavior.

In anomaly detection, popular supervised machine learning techniques include support vector machine (SVM), k-nearest neighbor (KNN), Naive Bayes and Decision Trees. Traditional Support vector machine approaches were often used for classifying faulty instances [6], [9], [22], [23]. In [21], Shin and Kim proposed a hybrid classification method that combines One-Class SVM with the nearest mean classifier (NMC)[2]. The feature subset selection algorithm in these methods improved classifier performance in lesser dimensional data. However, these methods took a long time to train and exhibited poor classification accuracy.

While supervised methods provide reasonable accuracy, studies suggest that they are only capable of accurately detecting previously seen anomalies and do not perform so well over new or unseen anomalies. In [6], Eskin and Stolfo propose a geometric framework for unsupervised anomaly detection by mapping data elements to two feature maps: data dependent normalization feature map and spectrum kernel based feature map. Anomalies were detected by determining which points lie in the sparse region of the feature space. Several popular clustering-based algorithms such as K-Means, SNN [14], ROCK [8], and DBSCAN [7] are also increasingly used for the task of anomaly detection. They group instances based on mutual similarity and assume that healthy samples are within a cluster and faulty samples fall outside of any cluster. These algorithms require fixing on the number and width/radius of the clusters and are very sensitive to outliers and initial seeds.

Artificial neural networks are becoming increasingly popular for predicting anomalies by capturing information embedded in temporal data. One approach for temporal anomaly detection has been to build multilayer perceptron based prediction models and use the prediction errors (the difference between the predicted values and the actual values) to compute an anomaly score [15]. A wide variety of simple to complex prediction models have been employed. In [3] a simple window-based approach is used to calculate the median of recent values as the predicted value, and a threshold on the prediction errors is used to flag outliers. In [10] the authors build a one-step-ahead prediction model. A data point is considered an anomaly if it falls outside a prediction interval computed using the standard deviation of the prediction errors. The authors compare different prediction models: naive predictor, nearest cluster, multilayer perceptron, and a single-layer linear network.

In particular, approaches based on self-organizing maps (SOMs) [13] of artificial neural networks have proven to be effective in identifying both known and unknown attacks [9], [20]. Several approaches related to the application of SOMs in anomaly detection have been examined in [9]. Experiments were conducted with different partitions of the training data: 10 percent partition, normal only connections and 50/50 stratified set consisting of equal numbers of faulty and normal connections. Diverse improvements have been proposed to make SOMs powerful and accurate. [20] proposed a hierarchy network built on a SOM architecture with no neighborhood or transfer functions. Randomly selected subsets were used to train the hierarchical net and each level of the hierarchical map was modeled as a simple winner-take-all K-Map. In [19], authors proposed a Naive Bayes SOM which proved effective for multidimensional visualization by enabling Naive Bayes model to perform topological mappings.

Recently, Long-Short Term Memory (LSTM) [16], [25], [17], have been gaining popularity due to their property to be able to detect and remember short term patterns over long periods of time. In [18], LSTM encoder decoder models time series temporal dependency and achieves better generalizing capability than traditional methods. In [10] the authors build a one-step-ahead prediction model. A data point is considered an anomaly if it falls outside a prediction interval computed using the standard deviation of the prediction errors. With respect to RNNs, stacked LSTM are used for anomaly detection in time series in [16]. The model takes only one time step as input and maintains LSTM state across the entire input sequence. The model is trained on normal data and made to predict multiple time steps. Thus each observation has multiple predictions made at different times in the past. In [4] LSTMs are employed to detect collective anomalies in network security domain. The model is used to detect anomalies in KDD 1999 dataset. In [24], Zhang and Chawla et al. considered the temporal dependency, noise resistance and the interpretation of severity of anomalies by using a convolutional LSTM network with attention technique. Another kind of neural network called the Hierarchical Temporal Memory (HTM) also has the ability to learn and remember long term dependencies. Ahmad and Lavin introduced HTM and its application in anomaly detection for high velocity streaming data [1]. Drawing our cues from the power of LSTM based networks to model time-series data rather accurately, we were motivated to try a variant of this model, which is an LSTM Autoencoder, for making predictions. Additionally, we also investigate the power of SOMs in modeling multi-variate time-series data for making predictions about possible anomaly. Our SOM based algorithm closely follows [5], which leverages SOMs to capture emergent behaviours and predict unknown anomalies.

5 DISCUSSIONS AND FUTURE WORK

A dynamic environment like a production process contains information that is changing with time, which needs to be understood in order to gain insights into the behaviour of the system so that we can flag any irregularities that happen. In this paper, we presented two anomaly prediction algorithms based on Self Organizing Maps and Long Short-Term Memory networks respectively. We implemented a prototype of our system on Amazon Web Services (AWS) and conducted extensive experiments to model the system behavior of Cassandra, our case-study distributed system. Our analysis and results show that both these algorithms pose minimal overhead on the system and are able to predict performance anomalies with high accuracy and achieve sufficient lead time in the process. Our algorithms are essentially application-agnostic and can be suited for monitoring any distributed system. However, there are few things that are yet to be explored. Our proposed algorithms found it difficult to effectively model the memory usage patterns of Cassandra, thus sending out quite a few false positive alerts. Current computational limitations restricted us from experimenting further and identifying the root cause. As a future work, we intend to deploy our system on a larger AWS instances, in order to effectively capture the resource usage patterns. With respect to fault localization, our LSTM Autoencoder model was able to correctly pin-point the affected (faulty) node, but in few cases, it was also identifying other normal nodes as faulty. We plan to identify the root cause for such false positive indications and intend to make our algorithms more robust in accurately identifying anomalies with minimal false positives.

REFERENCES

[1] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. 2017. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing* 262 (2017), 134–147.

[2] Bernhard B. SchÃűlkopf, John C. Platt, John C. Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. 2001. Estimating the Support of a High-Dimensional Distribution. *Neural Comput.* 13, 7 (July 2001), 1443–1471. DOI: http://dx.doi.org/10.1162/089976601750264965

[3] S. Basu and M. Meckesheimer. 2007. Automatic Outlier Detection for Time Series: An Application to Sensor Data. *Knowledge and Information Systems* 11 (2007), 137–154.

[4] Loïc Bontemps, Van Loi Cao, James McDermott, and Nhien-An Le-Khac. 2017. Collective Anomaly Detection based on Long Short Term Memory Recurrent Neural Network. *CoRR* abs/1703.09752 (2017). http://arxiv.org/abs/1703.09752

[5] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th international conference on Autonomic computing*. ACM, 191–200.

[6] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy, and Sal Stolfo. 2002. A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. In *Applications of Data Mining in Computer Security*. Kluwer.

[7] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507

[8] S. Guha, R. Rastogi, and K. Shim. 1999. ROCK: a robust clustering algorithm for categorical attributes. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*. 512–521. DOI: http://dx.doi.org/10.1109/ICDE.1999.754967

[9] H. Gunes Kayacik, A. Nur Zincir-Heywood, and Malcolm I. Heywood. 2007. A Hierarchical SOM-based Intrusion Detection System. *Eng. Appl. Artif. Intell.* 20, 4 (June 2007), 439–451. DOI: http://dx.doi.org/10.1016/j.engappai.2006.09.005

[10] D. J. Hill and B. S. Minsker. 2010. Anomaly Detection in Streaming Environmental Sensor Data: A Data-Driven Modeling Approach. *Environmental Modelling Software* 25 (2010), 1014–1022.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[12] Teuvo Kohonen. 1990. The self-organizing map. *Proc. IEEE* 78, 9 (1990), 1464–1480.

[13] T. Kohonen, M. R. Schroeder, and T. S. Huang (Eds.). 2001. *Self-Organizing Maps* (3rd ed.). Springer-Verlag, Berlin, Heidelberg.

[14] M. Steinbach L. ErtÃűz and V. Kumar. 2004. *Finding Topics in Collections of Documents: A Shared Nearest Neighbor Approach.* Springer US, Boston, MA, 83–103. DOI: http://dx.doi.org/10.1007/978-1-4613-0227-8_3

[15] C. C. Aggarwal M. Gupta, J. Gao and J. Han. 2014. Outlier Detection for Temporal Data: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 26 (2014), 2250–2267.

[16] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. 2015. Long Short Term Memory Networks for Anomaly Detection in Time Series. In *ESANN*.

[17] Z. Meng, S. Watanabe, J. R. Hershey, and H. Erdogan. 2017. Deep long short-term memory adaptive beamforming networks for multichannel robust speech recognition. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 271–275. DOI: http://dx.doi.org/10.1109/ICASSP.2017.7952160

[18] Song D. Chen H. Cheng W. Jiang G. Qin, Y. and G Cottrell. 2017. A dual-stage attention-based recurrent neural network for time series prediction. IJCAI.

[19] Gonzalo Ruz and D Pham. 2012. NBSOM: The naive Bayes self-organizing map. *Neural Computing and Applications* 21 (09 2012), 1319–1330. DOI: http://dx.doi.org/10.1007/s00521-011-0567-9

[20] S. T. Sarasamma, Q. A. Zhu, and J. Huff. 2005. Hierarchical Kohonenen net for anomaly detection in network security. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 35, 2 (April 2005), 302–312. DOI: http://dx.doi.org/10.1109/TSMCB.2005.843274

[21] Donghyuk Shin and Saejoon Kim. 2009. Nearest Mean Classification via One-Class SVM. In *Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization - Volume 01 (CSO '09)*. IEEE Computer Society, Washington, DC, USA, 593–596. DOI: http://dx.doi.org/10.1109/CSO.2009.388

[22] G. Shu and D. Lee. 2007. Testing Security Properties of Protocol Implementations - a Machine Learning Based Approach. In *27th International Conference on Distributed Computing Systems (ICDCS '07)*. 25–25. DOI: http://dx.doi.org/10.1109/ICDCS.2007.147

[23] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. 2009. Review: Intrusion Detection by Machine Learning: A Review. *Expert Syst. Appl.* 36, 10 (Dec. 2009), 11994–12000. DOI: http://dx.doi.org/10.1016/j.eswa.2009.05.029

[24] Cheny Y. Fengz X. Lumezanuy C. Chengy W. Niy J. Zongy B. Cheny H. Chawla N. Zhang C., Songy D. 2018. A Deep Neural Network for Unsupervised Anomaly Detection and Diagnosis in Multivariate Time Series Data.

[25] Yu-Long Zhou, Ren-Jie Han, Qian Xu, and Wei-Ke Zhang. 2018. Long Short-Term Memory Networks for CSI300 Volatility Prediction with Baidu Search Volume. *Concurrency and Computation: Practice and Experience* (05 2018). DOI: http://dx.doi.org/10.1002/cpe.4721