Anomaly Detection for Time Series Data

Manpreet Singh,[1] Manan Hora,[2] Edet Nkposong[3]
Ajay Krisha Borra[4], Metarya Ruparel[5] Rama Rao[6]
Rajdeep Dua[7]

August 11, 2017

Abstract

Anomaly detection is an important part of maintaining the health of large scale cloud services. Large scale services need techniques that provide accuracy for diagnosis and Management actions. In this paper, we discuss certain methods for real-time anomaly detection including traditional machine learning techniques like K-Means Clustering, Neural Networks and Hierarchical Temporal Memories (HTM's).

1 Introduction

Data centers and Software, today operate and compute at scale of Peta-bytes and more. Solving complex computational problems, high availability and scalability of systems are the top priorities, while serving the consumers. Instrumenting and monitoring hidden layers of data centers like rack life cycle management, software management is complex in nature. Any failures or malfunctions in the systems running at such large scale can lead to very expensive losses. Einstein Monitor aka Anomaly detector is designed to predict or detect the problems ahead of time thereby enabling advanced diagnoses to determine remedies.

Anomaly is described as a deviation from the expected behavior of a system. Anomaly detection should be done continuously, as long as a system is running.

2 Background

Servers are getting more powerful and the hardware is becoming more commodity. Running and maintaining thousands of servers with continuous feature enhancements and software updates makes statistical analysis of service health data more complex than ever. We have now reached a point where we need software tooling to augment our human analytical skills to master this challenge. To simplify the process and better separate real issues from background noise, we need machine learning techniques for anomaly detection.

The process of encoding human experiences into rules that create alerts is hard, due to the number of metrics and dependency between them. Most of the data cannot be alerted because it is periodic. Advanced knowledge and software like Einstein monitor (Einstein name here refers to Salesforce IQ platform), lets the team pro-actively address these issues before an outage occurs and customer trust is impacted.

Monitoring, which is the practice of observing systems and determining if they are healthy is getting harder since we are now managing many more systems (servers and micro-services) with much more data than ever before. Many of us are used to monitoring manually and visually by actually traversing the charts on the computer or using thresholds with systems like Nagios.

2.1 Why Threshold-Based Alerting is Not Efficient?

Detection of server or application related problems, also known as 'Anomaly Detection' is more nuanced than simply raising a flag when a particular metric is above a certain predefined threshold because:

- The value of metrics is dependent on both internal (such as the actual state of the hardware/servers) and external factors (number of transactions, seasonality, etc.)
- Metric values also vary over time-periods: for example: cpu may be high on certain days of the week than others, and during certain times of the year because of increase in usage during those periods.
- Often, multiple metrics move together and a real problem may exist only if two or three (or more) given metrics show anomalous behavior instead of just one.

Furthermore, thresholds actually represent one of the main reasons that monitoring is hard. Setting a threshold on a metric requires an Engineer to make a decision about the right value to configure. The difficulty is that there is no right value. A static threshold is just that, never changes over time. Systems are neither similar nor static. Every system is different from one another, and even individual systems changes both over the short term and long term.

For these reasons, threshold-based alerting generates several false positives. As a result, Site Reliability Engineers (SRE) often spend a non-trivial amount of time trying to figure out which alerts imply real problems with the system. Our project aims to reduce that time by building an anomaly detection system which generates very few false positives, and presents data in a format that is easy to interpret, so that effort can instead be focused on fixing issues.

2.2 High-Level Features

We are looking to build an anomaly detection system which can learn patterns from data and make predictions for metric values over time with more than 90 percent accuracy. We then use these predicted values to label incoming data-points as anomalous or normal based on how much they deviate from the expected value for the corresponding time-stamp. The KPI for this would be a reduction in the number of false positives generated. The system will also create tickets every time it raises an anomaly.

In current phase of work we use system level features and app level features separately, with next phase of work we plan to correlate the server and app features into a single vector for learning patterns.

3 Machine Learning Approaches

Before we dive deep, let's go through the basics of what different models are. Machine learning algorithms discover patterns in data, and construct mathematical models using these discoveries. We can then use the models to make predictions on future data.

3.1 Clustering

KMeans clustering is a class of unsupervised learning, which is used when we have unlabeled data. The algorithm finds cluster centroids that minimize the distance between data points and the nearest centroid.

KMeans algorithm only works with numerical data points vs any categorical features. The next step is to identify the cluster boundaries. Anomalies are further away from their cluster centers than normal data points because they deviate from normal behavior. After learning the cluster boundaries of each cluster we can make predictions on new data points. As the new data point comes in, we select the closest cluster to that data point by examining the distance between the new data point and each of the cluster center. Following which we check if the data point's distance is larger than the cluster boundary. If it is, we consider it as an anomalous data point, since it is outside the cluster. If not, it's considered as a standard data point since it's inside the cluster. Elaborated in Fig 1.

We then explored Hierarchical Temporal Memories(HTMs), which is discussed in the next section.

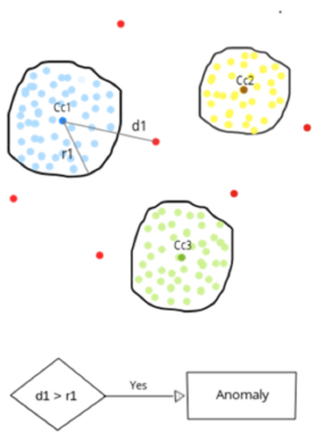


Figure 1: K-means for Anomaly Detection

3.2 Hierarchical Temporal Memories

Hierarchical Temporal Memory (HTM) is a biologically motivated machine intelligence technology that mimics the architecture and processes of the neocortex.

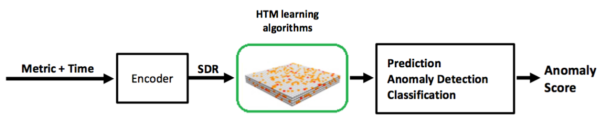


Figure 2: HTM for Anomaly Detection

The feature or metric value along with time are supplied to an encoder that converts them into a sparse distributed representation (SDR). SDRs enable several useful properties such as generalization across data streams, resistance to noise, and the attachment of semantic meaning to data points. And then next a sequence of SDRs is fed into the HTM algorithms. HTM algorithms simulate a small slice of the neocortex and are responsible for learning temporal sequences in the server/app metric data streams. Temporal sequences are nothing but patterns over time. HTM learning algorithms learn the temporal patterns present in the data continuously and new patterns are replaced by old patterns in the same way we remember recent events.

3.3 Recurrent Neural Networks and Long Short Term Memories

The Recurrent Neural Network (RNN) is a generalization of feed-forward neural networks to sequences. Given a sequence of inputs $(x_1, ..., x_T)$, a standard RNN computes a sequence of outputs $(y_1, ..., y_T)$. The RNN can easily map sequences whenever the alignment is known between the inputs the outputs ahead of time.

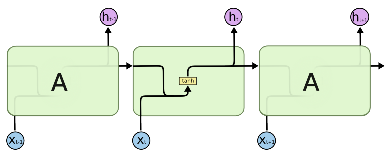


Figure 3: RNN for Anomaly Detection

Regular neural networks have been proven to be "universal function approximators". A feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function. However, ANNs (Artificial Neural Networks) cannot deal with sequential/temporal data. The reason for this is that they have no memory. When analyzing sequential data, it becomes important to look at information in context. The value in each cell must be examined in context of all the values before it.

Regular neural networks do not have persistent memory, but Recurrent Neural Networks do. For analyzing large chunks of sequential data, we need models which have long term memory. This led us to use LSTM (Long Short Term Memory) networks, which are a special kind of RNNs, capable of learning long-term dependencies.

LSTMs have chain like structure similar to basic RNN but the repeating module has a different structure. Rather than having a single neural network layer, there are four, interacting in a very special way.

Each layer of the LSTM network consists of gates which allow the network to forget certain information which is likely to be useless for future processing and retain other information which is likely to be used. A gate is essentially a mathematical function like the sigmoid function which takes a vector as input.

The Vanilla RNN Cell is described below :

$S_t = \phi(W * S_{t-1} + U * x_t + b)$
ϕ is the activation function.
$S_t \; \epsilon \; R^n$ - Current State
$S_{t-1} \; \epsilon \; R^n$ - Prior State
$x_t \; \epsilon \; R^m$ - Current Input
$W \; \epsilon \; R^{nXm}, U \; \epsilon \; R^{mXn}, b \; \epsilon \; R^n$ - weights, bias
n - state size
m- input size

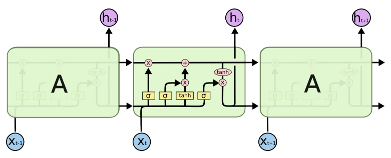


Figure 4: LSTM for Anomaly Detection

4 Data Pipeline

Einstein Monitor data pipeline involves a sequence of processing and learning stages. These stages are executed in order, and the input data is transformed as it passes through each stage in the pipeline. Typical steps involve Discover, Ingest, Process, Persist, Integrate, Analyze and Expose.

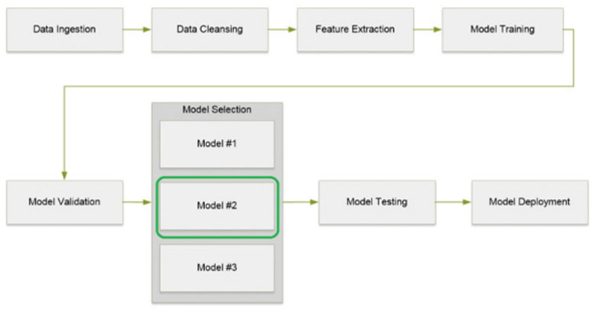


Figure 5: Data Pipeline for Anomaly Detection

4.1 Dataset

Taking the right data in the right format is key to success for any machine learning problem. Learning algorithms typically work with three types of data sets training, validation, and test. All three should randomly sample a larger body of data.

4.1.1 System Features

System features or vector is described below (Ref. Table 1), We took 0.6 million data points and used $300,000$ as training set and $300,000$ as test set.

Table 1: System Features

CPU User
CPU System
CPU Wait
System Interrupt
System Softirq
Memory Used
Memory Free
System:Interrupt
System:Softirq
System:Short-term
System:Midterm
System:Longterm

4.1.2 Application Features

Application features or vector (from core app) is described below (Ref. Table 2), We took $10k$ data points and used $5k$ as training set and $5k$ as test set.

Table 2: Application Features

Sample Count (No. of stack samples)

4.1.3 DataSource

We ingested the system data from Argus a time-series data collection, visualization, and monitoring service that provides both near-real-time and historical metric data. Argus has a fairly straightforward REST API, which is what we used to extract the data.

We ingested the application data from Warden service which was aggregated to build time-series database.

4.2 Storage

- Data is stored as rows under single column family in a HBase table.
- Telemetry data timestamps are used as the row key to group the features in the HBase column family.
- Data is stored under different name spaces based on the transformations applied during preprocessing.

4.3 Preprocessing

4.3.1 Normalization

Normalization is the method of rescaling one or more attributes to a common range. This means that the largest value for each attribute is 1 and the smallest value is 0. Data was normalized to contain values between (0,1).

4.3.2 Dealing With Missing Data

We encountered missing data points. In order to maintain consistency, we filled in the missing data-points with the mean of the remaining values for each column. This would not lead to any significant deviation or misinterpretation of actual data since the percentage of missing data points was less than 10 % in most cases.

4.3.3 Dimensionality Reduction

One way to speed up Machine Learning (or, equivalently allow for greater number of iterations in the same amount of time) is to reduce the number of features in the dataset. This is precisely the motivation to perform PCA- Principal Component Analysis.

Principal Component Analysis is a Statistical procedure that uses orthogonal transformation to extract the principal components of a feature set. The general idea is to reduce an n-dimensional dataset to a k-dimensional dataset in such a way that the k dimensions selected are the ones which contribute the most to the variance in the data (hence the term 'Principal' Components). PCA can also be used to figure out how many dimensions contribute most to the variance in the data. After implementing PCA on our dataset, we found that 99 % of the variance in the data was contributed by just two variables. We then went on to perform dimensionality reduction from 10 dimensions to 2 dimensions before building the model. This helped speed up processing, thus allowing for more layers, greater number of iterations and epochs, which further leads to higher accuracy.

5 Experiments and Results

We evaluated and compared the performance of various learning techniques like KMeans Clustering, Long short term memories (LSTMs) and Hierarchical temporal memory (HTM).

5.1 KMeans Clustering

We experimented with KMeans clustering applying different values of k. For k equal to 2, One cluster for anomalies and one for non-anomalies the result was a 40-60 distribution of data. This was obviously not useful for us since we can say with certainty that neither of those clusters correspond to anomalies.

5.2 LSTMs

We then moved on to LSTMs. As discussed in the theory in section 3.3, LSTMs are ideal when dealing with time-series data.

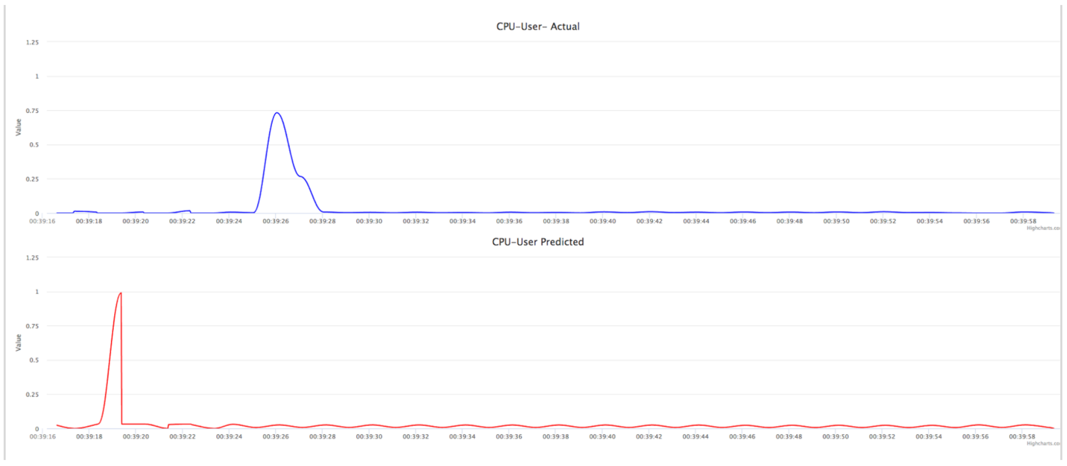
- Our file consisted of 600,000 data points, each with a unique time-stamp.
- Training set: The first 300,000 data points with 10 features each.
- Testing/Validation set: The next 300,000 with 10 features each.

5.2.1 Result Loss

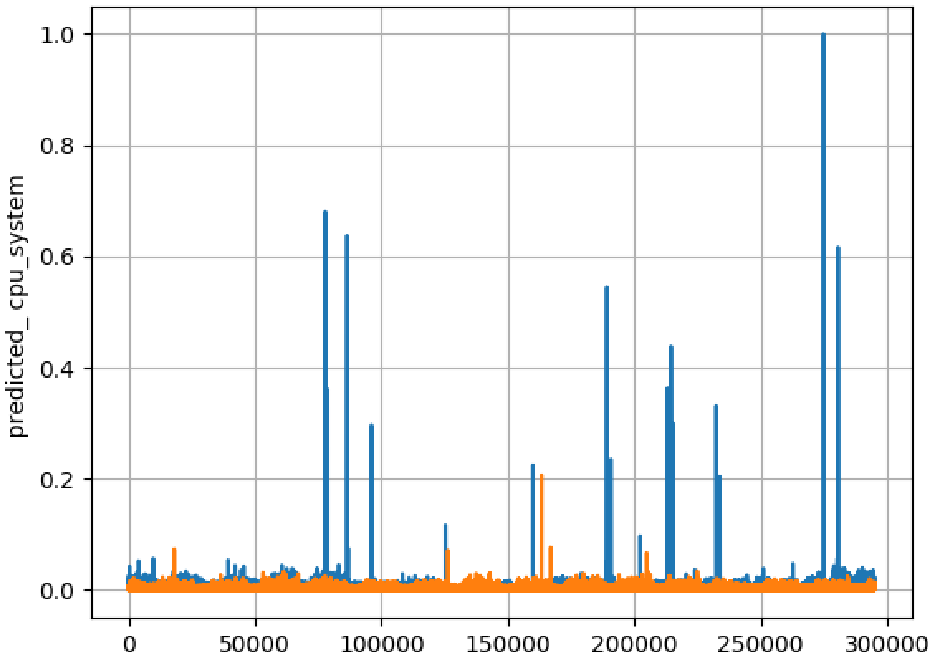
Mean Squared Error Loss: 0.05

5.2.2 Graphs- Actual vs Predicted, Over Six Months for system data

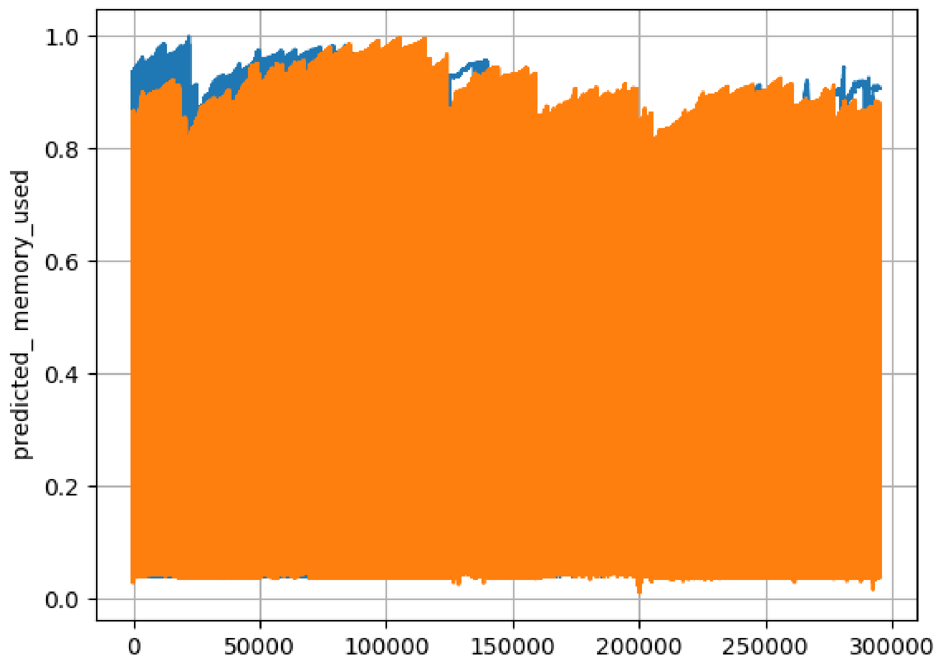
The following graph shows the "actual" vs "predicted" samples. The prediction graph predicts and shows values 5 points ahead of actual graph. As you can see in the graph, when an anomaly happened, the actual and predicted CPU values are very close.



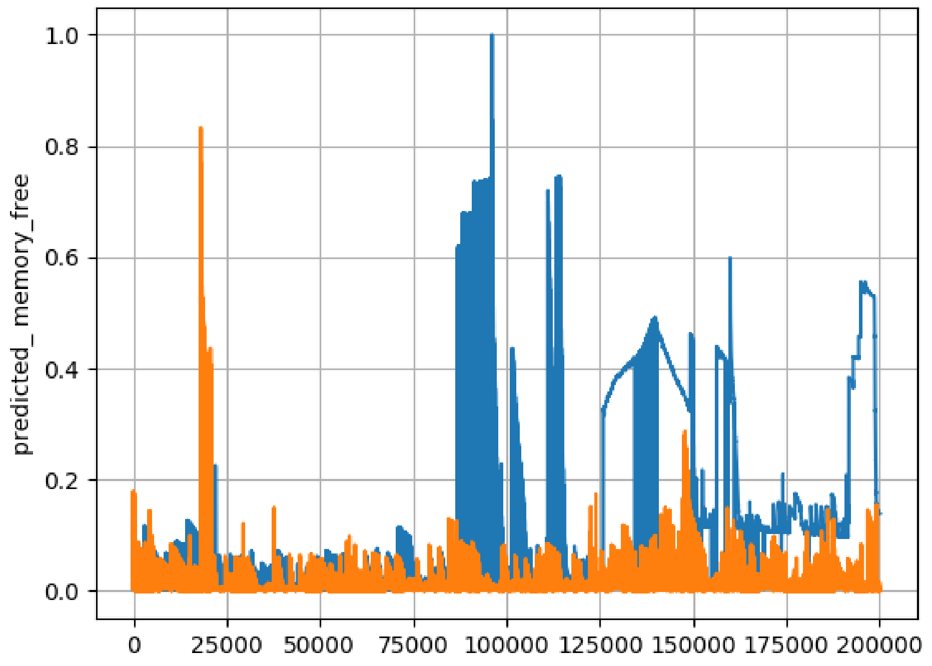
In the following graphs, the blue portion represents the actual values of the metric over time while the orange portion represents the predicted values over the same time period.



CPU-System



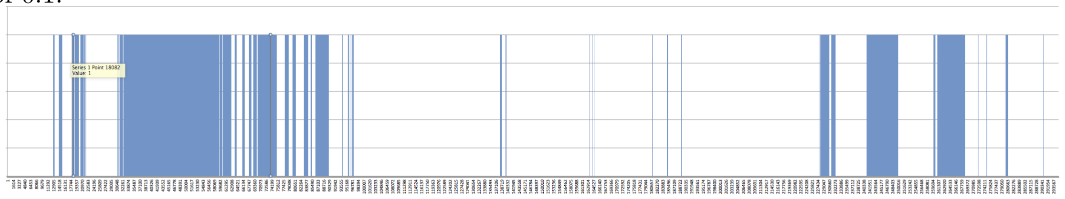
Memory Used



Free Memory

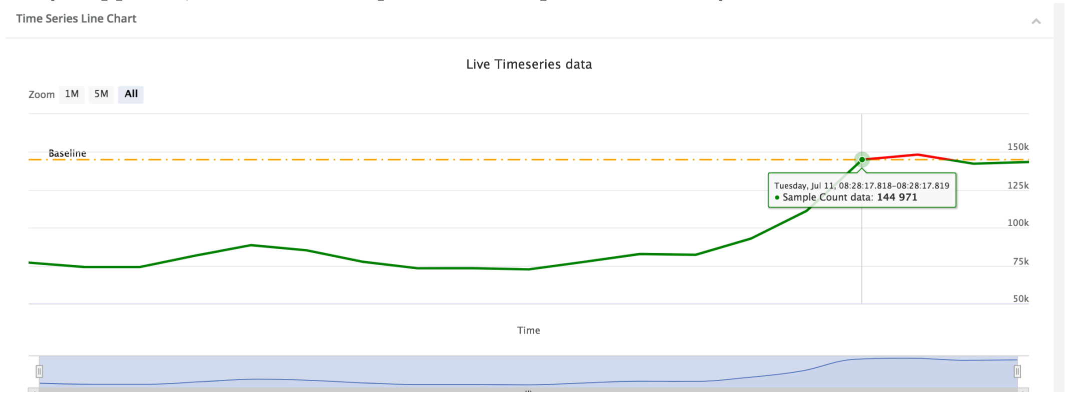
- After generating predictions, we took a threshold-based approach to anomaly detection: Given a (real) data-point, we compare it against the predicted/expected data-point corresponding to the same time-stamp as the input data-point. If the mean-squared difference taken across all features is above a certain threshold, then we raise it as an anomaly.

In the following graph, the blue lines represents anomalies, the whitespace represents 'normal' data points. These have been computed for a time period of 6 months with an anomaly-threshold of 0.1.

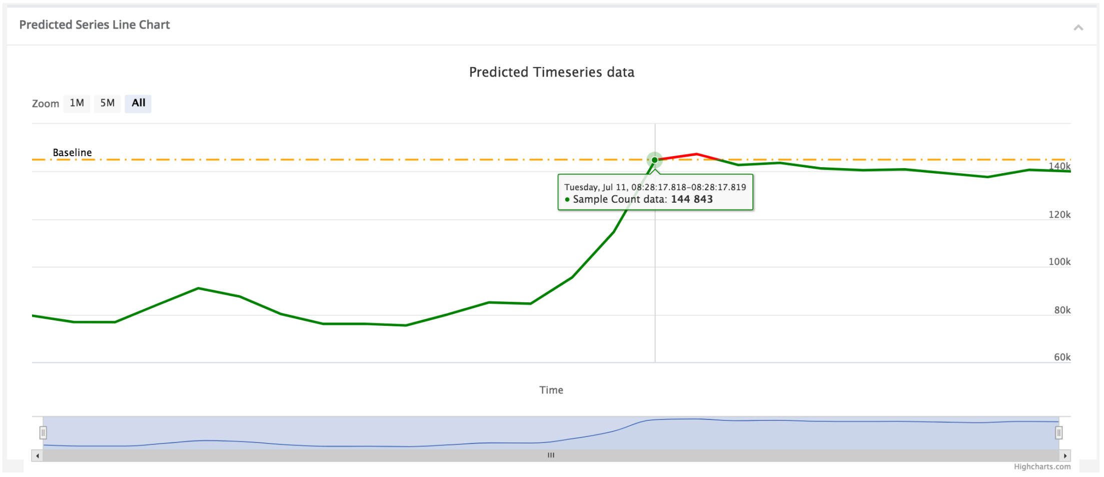


5.2.3 Graphs- Actual vs Predicted, for application data

The following graph shows the "actual" vs "predicted" warden samples. The prediction graph predicts and shows values 10 points a head of actual graph. As you can see in the graph, when an anomaly happened, the actual and predicted sample count is very close.



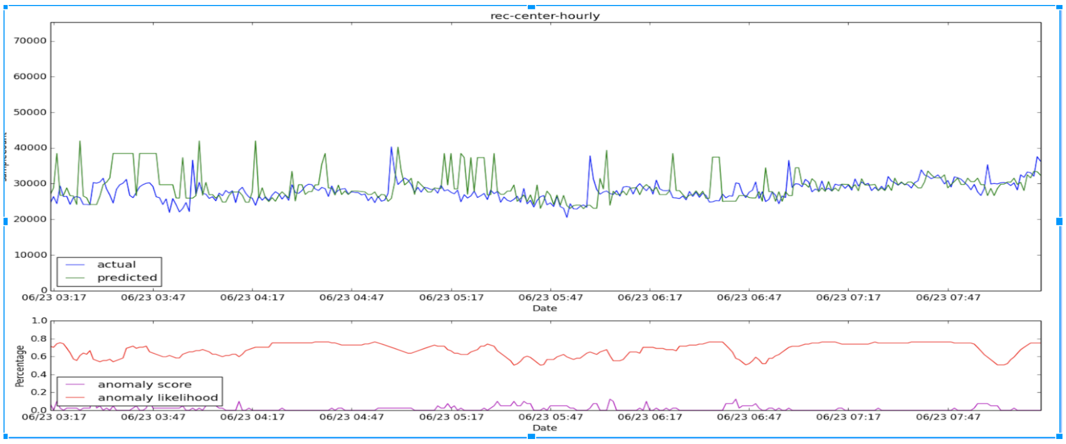
Warden Time-series Data



Predicted Warden Time-series Data

5.3 HTMs

We used HTMs for making predictions on application level data as shown below. Similar experiment was conducted for system features. Using those predictions, we perform anomaly detection as described in section 5.2.1.



6 Conclusion

Out of the three approaches that we investigated for anomaly detection, we got satisfactory results with two of them HTMs and LSTMs. The success of a problem solving approach is determined by how well it measures against its KPIs. Our LSTM-based system for server anomaly detection was able to predict metrics with a loss of 5%. While HTMs have given us fairly accurate results (as shown in the graph above), at present, we do not have the mathematical proof to say that HTMs perform objectively better or worse than LSTMs.

So far, we have examined only one specific use case for this system- anomaly detection on server level data. While we have not yet validated the anomalies, we have validated our predictions against real data. This system also has several other applications:

- Capacity Planning: Anomaly Detection at pod level data can be immensely helpful in capacity planning. Presently, the capacity engineers monitor metrics individually and if a metric crosses a certain threshold, they take that as an indicator of the need for more machines. However, several times, the spikes in metrics are only temporary and do not actually indicate a real need for more machines. With an Anomaly Detection system in place which looks at multiple metrics over time, it can potentially aid capacity engineers by providing knowledge of how likely a given spike is likely to point to a real need for more machines.

- Anomaly Detection at the Application level: If we have Anomaly Detection Systems for both application-level and server-level, it can be useful in triaging issues. For example, if there is an increase in response time on the core app, with anomaly detection systems on both server and application-levels, we can figure out where the problem lies.

In conclusion, what we have here is a generalizable system for time series analysis, forecasting and anomaly detection.

References

[1] Mike Schuster and Kuldip K. Paliwal, Bidirectional Recurrent Neural Networks, Trans. on Signal Processing 1997.

[2] Alex Graves, Santiago Fernandez, and Jurgen Schmidhuber, Multi-Dimensional Recurrent Neural Networks, ICANN 2007.

[3] Kai Sheng Tai, Richard Socher, and Christopher D. Manning, Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks, arXiv:1503.00075 / ACL 2015.

[4] Oriol Vinyals, Samy Bengio, Manjunath Kudlur, "Order Matters: Sequence to sequence for sets", ICLR 2016.

[5] Junyoung Chung, Sungjin Ahn, Yoshua Bengio, "Hierarchical Multiscale Recurrent Neural Networks", arXiv:1609.01704.

[6] Sepp Hochreiter and Jurgen Schmidhuber, Long Short-Term Memory, Neural Computation 1997.

[7] Hierarchical Temporal Models.
https://en.wikipedia.org/wiki/Hierarchical_temporal_memory

[8] Recurrent Neural Networks.
https://en.wikipedia.org/wiki/Recurrent_neural_network
https://ayearofai.com/rohan-lenny-3-recurrent-neural-networks-10300100899b