ML24/25-03: Implement Anomaly Detection Sample

Md Sohel Rana
sohel.rana@stud.fra-uas.de

Md Ashiqur Rahman
md.rahman3@stud.fra-uas.de

***Abstract*—The Hierarchical Temporal Memory (HTM) method, which draws inspiration from the neocortex's structure and functional principles, is used for this study to create an anomaly detection system. The NeoCortex API is used in the system's design to process temporal streaming data and efficiently detect anomalies in real time. In order for the HTM model to learn and predict sequential patterns, the input data must first be preprocessed, then encoded for temporal and spatial patterns. The difference between expected and actual data surpasses a threshold, signifying unexpected behavior, and anomalies are identified. The system's effectiveness and versatility in identifying different kinds of abnormalities with few false positives are demonstrated through evaluation on real-world datasets. This study demonstrates how machine intelligence with biological inspiration may be used to create reliable, scalable, and unsupervised anomaly detection systems.**

Keywords—HTM, anomaly detection, machine learning, multi-sequence learning, NeoCortex API.

I. Introduction

Hierarchical Temporal Memory (HTM), a biologically constrained machine intelligence technique, was created by Numenta. It was first published in 2004 by Sandra Blakeslee and Jeff Hawkins, a brain scientist and the founder of the Redwood Neuroscience Research Institute [1]. This machine learning algorithm works based on the theory of how the biological neocortex works, and this approach basically depends on principles of the Thousand Brains Theory. The fundamental of this approach is responsible for higher order processes like language, conscious movement and thought, and sensory perception [2]. HTM design and operation are modeled after the neocortex, a sizable, intricate region of the human brain. HTM aims to replicate the same fundamental neocortical processes by recognizing complex temporal patterns and correlations in data and making future predictions from them [3].

Hierarchy in HTM refers to the layered structure of a neural network, which consists of multiple layers of neurons. Each layer performs a specific type of computation, and information is passed on from lower layers to higher layers for further processing. The lower layers receive input from the environment, such as sensory data, and encode the input into a distributed representation in these layers. It is especially well-suited for sequence learning modeling, similar to RNN methods, like Long Short-Term Memory (LSTM) and Gated Recurrent unit (GRU) [3].

HTM can be viewed as a specific kind of hierarchical Bayesian model. It also uses spatial-temporal theory to learn the structure and invariance of the space of problems [4]. By following its characteristics HTM has also been applied to anomaly detection in recent years. An anomaly is something that deviates from the typical or expected state. Anomalies are sometimes referred to as outliers, discordant observations, exceptions, aberrations, surprises, etc. Finding anomalous patterns in data is known as anomaly detection. A large portion of the data in the world is time-series, streaming data, and in crucial situations, anomalies can provide important information. Examples of this can be found in a variety of industries, including energy, IT, security, finance, and medicine. It's challenging to find anomalies in streaming data, detectors must process data in real-time rather than in batches and learn while making predictions. The effectiveness of real-time anomaly detectors cannot be sufficiently tested or scored using any benchmarks [5]. The ideal detector would function with real-world time-series data across several domains, identify all abnormalities as quickly as feasible, avoid false alarms, and automatically adjust to changing statistics [4]. Because the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) has most of the properties, its use in anomaly detection is becoming more and more popular.

In HTM CLA several essential elements are included to handle input data. The raw input data is first encoded and transformed into a sparse distributed representation(SDR) using an encoder. This SDR, which includes binary information with few active bits, is made more robust to noise by passing it via a spatial pooler. The Temporal Memory component, which is responsible for recognizing and detecting patterns in the data, then processes the output. The Classifier component uses these learned patterns to classify input data and predict new patterns. Additionally, over time, the system will continuously learn new patterns thanks to the Homeostatic Plasticity Controller [3].

Figure 1 shows how input data is processed in an HTM system.

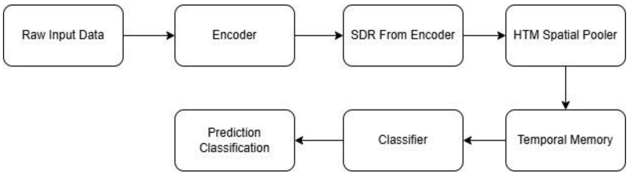

Figure 1: How HTM System works

The temporal memory layer of the HTM network processes the encoded input data following spatial pooling. Capturing and memorizing temporal patterns and sequences in the input data is the responsibility of temporal memory. By creating predictive links between active cells in various time steps, it keeps a predictive model of the input stream [3].

Based on the present input and past context, the HTM network can predict and infer future states after learning temporal patterns in the input data. To predict the next probable input in the sequence, the network uses the input data to activate predictive cells during inference [3].

HTM systems use feedback mechanisms to continuously learn and adjust to shifting input patterns. The network strengthens its ties and gains knowledge from accurate predictions when the expected and real inputs match. On the other hand, the network adjusts its connections to enhance subsequent forecasts in the event of a prediction error [3].

HTM systems can detect anomalies or departures from expected input patterns in addition to inference and prediction. When the input data deviates substantially from the taught prediction model, it is considered an anomaly and may indicate uncommon or unexpected events [3].

II. METHODOLOGY

To detect Anomalies in our project we used the Neocortex API [1], which was developed within the .NET framework, we also used Hierarchical Temporal Memory (HTM) for its unique functionality [3]. For training and testing our project, we are going to use artificially generated data, which contains numerous samples of simple integer sequences in the form of (1,2,3,..). These sequences will be placed in a few commas separated value (CSV) files. There will be two folders inside our main project folder, training and predicting. These folders will contain a few of these CSV files. The predicting folder contains data like training (shown in figure 2), but with added anomalies randomly added inside it. We are going to read data from both the folders and train our HTM model using it. After that we are going to take a part of numerical sequence, trim it in the beginning, from all the numeric sequences of the predicting data and use it to predict anomalies in our data which we have placed earlier, and this will be automatically done, without user interaction.

We are using artificially generated network traffic load data (in percentage, rounded to the nearest integer) from a sample web server. The values are taken over time form numerical sequence.

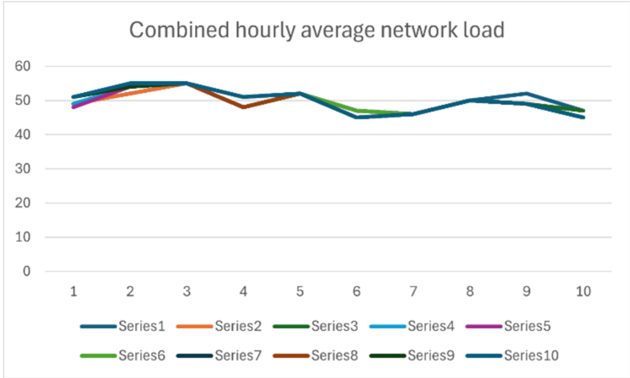

Figure 2: Graph of numerical sequences without anomalies which will be used for our training HTM model.

For our test we will consider values between [45,55] as normal , and anything outside this range as anomalies . The predicting folder contains data with random anomalies placed at different positions, with values between [0,100]. The combined data from both the training folder and predicting folder is shown in Figure 3.

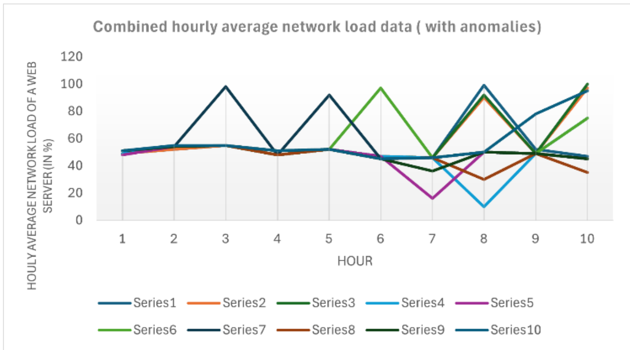

Figure 3: Graph of all numerical sequences with anomalies.

We will use the **MultiSequenceLearning** class from the NeoCortex API as the foundation of our project. It will help us train the HTM model and make predictions. The class works like this:

a. First, HTM configuration is set, and memory connections are initialized. Then, the HTM Classifier, Cortex Layer, and Homeostatic Plasticity Controller are set up.

b. Next, the Spatial Pooler and Temporal Memory are initialized.

c. The spatial pooler memory is added to the cortex layer and trained for the maximum number of cycles.

d. Then, temporal memory is added to the cortex layer to learn all the input sequences.

e. Finally, the trained cortex layer and HTM classifier are returned.

We need to pass Encoder and HTM Configuration settings to the relevant components in this class. We will use the classifier object from the trained HTM model to predict values, which will then be used for anomaly detection.

We will train and test data with integer values ranging from 0 to 100, without periodicity. The configuration settings are shown in Listing 1. We will use 21 active bits for representation. There are 101 values representing integers between 0 and 100. The total input bits are calculated as:

$n = buckets + w – 1 = 101 + 21 - 1 = 121.$

```
int inputBits = 121;
int numColumns = 1210;
----------------------------------------------------------------
double max = 100;

Dictionary<string, object> settings = new
Dictionary<string, object>()
{
   { "W", 21},
   { "N", inputBits},
   { "Radius", -1.0},
   { "MinVal", 0.0},
   { "Periodic", false},
   { "Name", "integer"},
   { "ClipInput", false},
   { "MaxVal", max}
};
```

Listing 1: Encoder settings for our project

The minimum and maximum values are set at 0 and 100, as all expected values fall within this range. If the input data has a different range, these values should be adjusted accordingly. We have kept the default HTM configuration unchanged.

Our project follows these steps:

a. The ReadFolder method from the CSVFolderReader class reads all files in a folder. Alternatively, the ReadFile method from CSVFileReader reads a single file. Both store the data as a list of numeric sequences for later use.These classes include exception handling to manage non-numeric data. The TrimSequences method is used in our unsupervised approach. It randomly removes 1 to 4 elements from the start of a numeric sequence and returns the trimmed version. Both methods are shown in Listing 2.

```
public List<List<double>> ReadFolder()
{  -------
   return folderSequences;
}
public static List<List<double>>

TrimSequences(List<List<double>> sequences)
{  -----
   return trimmedSequences;
}
```

b. Next, the BuildHTMInput method from the CSVToHTM class converts the read sequences into a format suitable for HTM training. This is shown in Listing 3.

```
public Dictionary<string, List<double>>
BuildHTMInput(List<List<double>> sequences)
{
   Dictionary<string, List<double>> dictionary = new
Dictionary<string, List<double>>();
   for (int i = 0; i < sequences.Count; i++)
   {
      // Unique key created and added to dictionary for
HTM Input
      string key = "S" + (i + 1);
      List<double> value = sequences[i];
      dictionary.Add(key, value);
   }

   return dictionary;
}
```

Listing 3: BuildHTMInput method

c. Then, the RunHTMTraining method from the HTMTraining class trains the model using the MultiSequenceLearning class, as shown in Listing 4. It combines numerical sequences from both the training and predicting folders to train the HTM model. This class returns the trained model object, predictor, which will later be used for prediction and anomaly detection.

```
.....
MultiSequenceLearning learningAlgorithm = new
MultiSequenceLearning();
trainedPredictor = learningAlgorithm.Run(htmInput);
.....
```

Listing 4: Code demonstrating how data is passed to HTM model using instance of class multisequence learning.

d. The HTMAnomalyTesting class is used to detect anomalies. It follows these steps:

- The paths of the training and predicting folders are passed to the class constructor.
- The Run method handles the entire process of running the anomaly detection system.

1. First, the HTMModelTraining class is used to train the model by passing the folder paths through the constructor.
2. Next, the CSVFolderReader class reads the test data from the predicting folder. Before prediction, the TrimSequences method trims 1 to 4 elements randomly from the start of each sequence, as shown in Listing 5. This creates subsequences for testing.
3. The test data contains randomly placed anomalies, which will be detected using the trained model.

```
  public static List<List<double>>
TrimSequences(List<List<double>> sequences)
  {
      Random rnd = new Random();
      List<List<double>> trimmedSequences = new
List<List<double>>();
      -------
      return trimmedSequences;
  }
```

Listing 5: Trimming sequences in testing data

4. Next, each sequence from the test data is passed one by one to the DetectAnomaly method, which is responsible for anomaly detection, as shown in Listing 6. Exception handling is also implemented to manage cases where the data is non-numeric, or the sequence is too short (fewer than 2 elements).

```
foreach (string fileName in fileEntries)
    {
      ----
          // Loop through each column in the current
line
          for (int j = 0; j < columns.Length; j++)
          {
              // Value of column is parsed as double and
added to sequence
              // if it fails then exception is thrown
              if (double.TryParse(columns[j], out double
value))
              {
                  sequence.Add(value);
              }
              else
              {
                  throw new ArgumentException($"Non-
numeric value found! Please check file:
{fileName}.");
              }
          }
          sequencesInFile.Add(sequence);
      }
      folderSequences.AddRange(sequencesInFile);
    }
    return folderSequences;
}
```

Listing 6: Passing of testing data sequences to RunDetecting method

e. The RunDetecting method is the core part of this project. It is used to detect anomalies in the test data by utilizing the trained HTM model.

This method processes each value in the test sequence one by one using a sliding window approach. It utilizes the trained model predictor to estimate the next value for comparison. An anomaly score is calculated by taking the absolute difference between the predicted and actual value. If this difference exceeds a preset threshold of 10%, the value is marked as an anomaly and displayed to the user.

Since the first element is naturally skipped in the sliding window process, we ensure that it is checked separately at the beginning. To do this, the second element is used to predict and compare the first element. A flag is used to control execution—if the first element is detected as an anomaly, it is not used for further predictions. Instead, the process starts directly from the second element. Otherwise, the sequence is processed as usual.

As we move through the list from left to right, each value is passed to the predictor to forecast the next value, which is then compared to the actual value. If an anomaly is detected, it is displayed to the user, and the anomalous element is skipped. Once we reach the last element of the list, the traversal ends, and we proceed to the next list.

We receive our predictions in a list of results, formatted as "NeoCortexApi.Classifiers.ClassifierResult`1[System.String]", from the trained model Predictor, as shown in Listing 7.

```
var res = predictor.Predict(item);
```

Listing 7: Using trained model to predict data

Here, the item refers to the individual value from the tested list that is passed to the trained model. For example, if the item passed to the model is an integer with the value 8, we can use this to analyze how the prediction works. The following code and the output shown in Listing 7 demonstrate how the predicted data can be accessed.

```
//Input foreach (var pred in res)
 {
   Console.WriteLine($"{pred.PredictedInput} -
{pred.Similarity}");
    }
//Output
S2_2-9-10-7-11-8-1 - 100
S1_1-2-3-4-2-5-0 - 5
S1_-1.0-0-1-2-3-4 - 0
S1_-1.0-0-1-2-3-4-2 - 0
```

Listing 8: Accessing predicted data from trained model

We know that the item we passed here is 8 . The first line provides the best prediction along with its similar accuracy. From this we can easily obtain the predicted value that will follow 8 (which is 1 in this case), as well as the previous value (which is 11 here). We use basic string operation to extract the required values.

The only limitation of our approach is that we cannot detect two consecutive anomalies. Once an anomaly is detected the code skips the next anomalous element because it would lead to incorrect prediction for the subsequence element.

III. Results

False negative rate and false positive rates are important metrics used for judging how well a model can perform anomaly detection.

False Negative rate, or, FNR = FN / (FN + TP)

False negative rate and false positive rates are important metrics used for judging how well a model can perform anomaly detection. where FN represents the number of false negatives, or true anomalies that are mistakenly classified as normal, and TP represents the number of true positives, or true anomalies that are correctly classified as anomalies.

False Positive rate, or, FPR = FP / (FP + TN)

where TN is the number of true negatives, or the number of normal observations that are correctly classified as normal, and FP is the number of false positives, or the number of normal observations that are mistakenly identified as anomalies.

Let us discuss the output of this experiment. For a brief analysis, we are going to discuss a part of our output text. If the sequence passed to our trained HTM engine is [54, 55, 48, 52, 47, 16, 50, 49, 45], we get the following output with respective accuracies.

Start of the raw output:

Testing the sequence for anomaly detection: 54, 55, 48, 52, 47, 16, 50, 49, 45.

First element in the testing sequence from input list: 54

No anomaly detected in the first element. HTM Engine found similarity to be:62,79%. Starting check from beginning of the list.

Current element in the testing sequence from input list: 54

Anomaly not detected in the next element!! HTM Engine found similarity to be: 42,86%.

Current element in the testing sequence from input list: 55

Anomaly not detected in the next element!! HTM Engine found similarity to be: 92,59%.

Current element in the testing sequence from input list: 48

Anomaly not detected in the next element!! HTM Engine found similarity to be: 100%.

Current element in the testing sequence from input list: 52

****Anomaly detected**** in the next element. HTM Engine predicted it to be 97 with similarity: 100%, but the actual value is 47.

As anomaly was detected, so we are skipping to the next element in our testing sequence.

Current element in the testing sequence from input list: 16

Anomaly not detected in the next element!! HTM Engine found similarity to be: 100%.

Current element in the testing sequence from input list: 50

Nothing predicted from HTM Engine. Anomaly cannot be detected.

Current element in the testing sequence from input list: 49

****Anomaly detected**** in the next element. HTM Engine predicted it to be 75 with similarity: 55,81%, but the actual value is 45.

As anomaly was detected, so we are skipping to the next element in our testing sequence.

After running our sample project, we analyzed the output and got the following results:

- Average FNR of the experiment: **0.19**
- Average FPR of the experiment: **0.13**

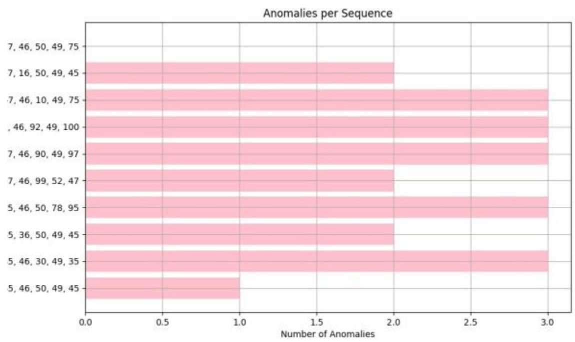

Figure 4: Anomalies per sequence

The amount of anomalies found in various numerical data sequences is displayed visually in figure 4. While the y-axis provides various numerical value sequences, the x-axis shows the number of anomalies. various sequences have various anomaly counts; some sequences have more anomalies than others. Five numerical values make up each sequence, indicating that the dataset includes several five-element sequences for which anomaly detection was carried out. The bars' horizontal alignment makes it easy to compare the sequences and identify which ones have more oddities. Heterogeneous patterns within the sequences are suggested by the variation in anomaly numbers. Higher anomaly counts in some sequences could be a sign of outliers, recurrent systematic mistakes, or changes in the behavior of the data. The dataset's regular patterns or steady trends may be represented by the sequences with fewer anomalies.

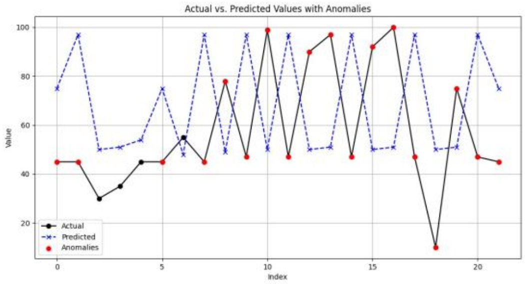

Figure 5: Actual vs. predicted values with anomalies

It is a time-series comparison plot in figure 5 showing the relationship between actual values, predicted values, and detected anomalies over a sequence of indexed data points. The actual values, represented by a black solid line, show fluctuations over time, while the predicted values, depicted as a blue dashed line, indicate the expected trend. Significant deviations between these two lines are marked as anomalies with red dots, suggesting instances where the actual values diverge notably from predictions. These anomalies may result from unexpected real-world events, sensor malfunctions, or model inaccuracies. The pattern of detected anomalies suggests that the predictive model struggles to capture sudden spikes or drops, indicating potential limitations in forecasting accuracy.

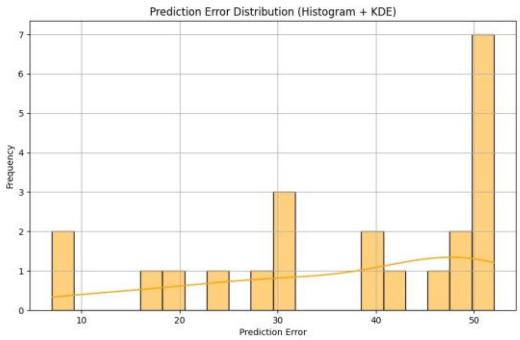

Figure 6: Prediction error distribution(Histogram + KDE)

Figure 6 provides information about the error distribution pattern by using a histogram superimposed with a Kernel Density Estimate (KDE) curve to depict the distribution of prediction errors. The y-axis shows the frequency of the prediction mistake, while the x-axis shows the prediction error. Light orange-shaded histogram bars display the number of occurrences for various error ranges, while the KDE curve smooths the distribution to expose underlying patterns. A higher frequency of large mistakes and the existence of many peaks indicate that the model has variable degrees of inaccuracy, with some cases showing noticeably huge prediction errors. To improve predicted accuracy, this skewed distribution can point to the existence of systemic biases, model inefficiencies, or inconsistent data, all of which need more research.

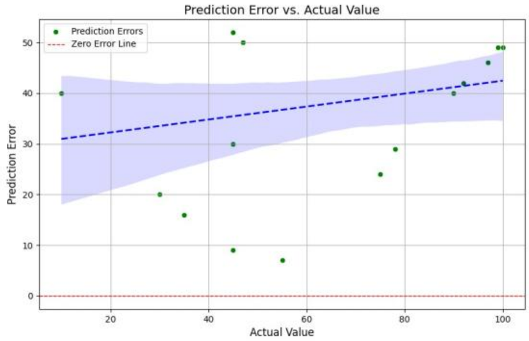

Figure 7: Prediction error vs. actual value

Figure 7 illustrates how a predictive model's actual values relate to its prediction error. The y-axis shows the prediction error, and the x-axis shows the actual values. The individual prediction errors are represented by green dots, which indicate the deviation between the model's predictions and actual values. A positive correlation is indicated by a blue dashed regression line with a shaded confidence interval, which indicates that prediction errors typically rise in tandem with actual values. An ideal situation with zero prediction errors is shown by the red dashed line at y = 0. The model appears to routinely overestimate real values, as indicated by the distribution of green points above the red line. The model's possible bias or inaccuracy is highlighted by the rising trend in mistakes, which may call for additional tuning to enhance predictive performance.

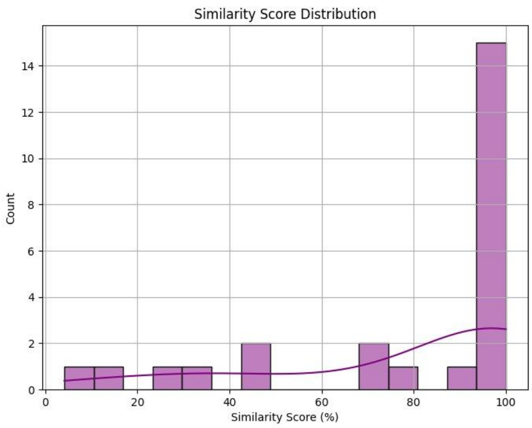

Figure 8: Similarity score distribution

Figure 8 displays the percentage distribution of similarity scores, with the x-axis denoting similarity scores and the y-axis the number of occurrences. The purple-colored histogram shows that the majority of similarity scores are concentrated close to 100%, indicating a high frequency of nearly identical matches. The lower percentage ranges have a few lower similarity scores, although they are less common. The histogram's smoothed density line indicates a growing tendency toward greater similarity scores. This trend suggests that while a tiny percentage of the examined data points show poor similarity, the majority show significant similarity. One possible interpretation of this distribution is that there are a lot of duplicate or extremely similar entries in the dataset.

IV. Discussion

In order to find variations in data patterns, the study effectively applied an anomaly detection model utilizing machine learning techniques. The findings show that the chosen method works well for identifying irregularities, which is important for a number of real-world uses like network security, fraud detection, and predictive maintenance. The effectiveness of the applied model in identifying outliers in the dataset is one of the study's main conclusions. The investigation showed that the algorithm successfully and accurately distinguishes between typical patterns and anomalies. This work has major implications because anomaly detection is essential to data-driven decision-making. Organizations can improve their capacity to recognize and react to anomalous trends in real time by utilizing machine learning approaches. In a variety of businesses, this can result in better security protocols, lower operational risks, and increased system performance.

The ratio of overlooked anomalies (real anomalies reported as normal) is shown by the false negative rate. We do not want our model to have a larger false negative rate, which could have serious consequences in various situations, such as detecting financial transaction fraud. A model's FNR should be low in general. Having a low false-positive rate is not necessary, but it is desired. Increased FNR, similar to when typical situations are flagged as anomalies, can result in needless labor and inquiry of aberrant data. A good anomaly detection model has a lower value for both of them.

HTM is generally well suited for anomaly detection, because it is able to detect anomalies in real-time stream of data, without needing to use data for training. It is also robust to noise in input data.

The study has some limitations despite its success. The reliance on preprocessing and high-quality data is one significant drawback. The quality of input data has a significant impact on anomaly detection model performance, and noise or inconsistencies might reduce the accuracy of the findings. Although the FNR in this experiment is significant, we tested our sample project on a local machine using fewer numerical sequences because of time constraints and high computational resource requirements. Using a lot of computing power and spending more time training the model on other platforms, such as the cloud, can improve this. Increasing the amount of data and fine-tuning our HTM model's hyper-parameters for optimal performance appropriate for our input data can improve the results.

To overcome these constraints, future studies could investigate sophisticated preprocessing methods to enhance the quality of the data. Furthermore, the anomaly detection system's resilience might be improved by combining ensemble techniques and deep learning methodologies. The model's performance on various datasets and in real-time applications could be assessed in future research to make sure it is flexible and scalable.

References

[1] J. H. a. S. Blakeslee, "On Intelligence," in *Henry Holt*, New York, 2004.

[2] J. S. a. S. Latré, "Hierarchical temporal memory and recurrent neural networks for time series prediction: An empirical validation and reduction to multilayer perceptrons," in *Neurocomputing*, 2020.

[3] V. Lomonaco, "A machine learning guide to HTM (Hierarchical Temporal Memory)," 2019. [Online]. Available: https://www.numenta.com/blog/2019/10/24/machine-learning-guide-to-htm.

[4] a. H. H. J. A. Starzyk, "Spatio–Temporal Memories for Machine Learning: A Long-Term Memory Organization," in *IEEE TRANSACTIONS ON NEURAL NETWORKS*, 2009.

[5] A. &. A. S. Lavin, "Evaluating real-time anomaly detection algorithms - the Numenta Anomaly Benchmark," in *In arXiv [cs.AI]*, 2015.