ML23/24-08: Implement Anomaly Detection Sample

Information Technology Course
Module Software Engineering
by Damir Dobric / Andreas Pech

Hasibuzzaman
hasibuzzaman@stud.fra-uas.de

Md Ikbal Husen Chowdhury
md.chowdhury@stud.fra-uas.de

Suva Sarkar
suva.sarkar@stud.fra-uas.de

Abstract— HTM (Hierarchical Temporal Memory) is a machine learning algorithm that processes timeseries data in a distributed manner using a hierarchical network of nodes. It is biologically inspired, both architecturally and functionally, by the neocortex of a human brain. It is possible to train each node, or column, to identify patterns in the input data. Information processing, pattern recognition, and future prediction based on prior knowledge can all be done using this. It is a potential method for predicting and detecting anomalies in a range of industries, including banking and healthcare. By utilizing an HTM model to learn several basic numeric integer sequences as input, we have attempted to create an anomaly detection sample and attempt to identify patterns within it. Then, after setting a tolerance threshold, it will compare the actual data with the anticipated data from learning in an attempt to find anomalies. This paper gives a prototype algorithm implementation along with a thorough description of anomaly detection approaches.

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

I. INTRODUCTION

Much of the world's data is streaming, time-series data, where anomalies give significant information in critical situations; examples abound in domains such as finance, IT, security, medical, and energy. Yet detecting anomalies in streaming data is a difficult task, requiring detectors to process data in real-time, not batches, and learn while simultaneously making predictions. There are no benchmarks to adequately test and score the efficacy of real-time anomaly detectors. The perfect detector would detect all anomalies as soon as possible, trigger no false alarms, work with real-world time-series data across a variety of domains, and automatically adapt to changing statistics [4]

Based on the fundamental ideas of the Thousand Brains Theory, the Hierarchical Temporal Memory (HTM) algorithm is a machine learning technique. Its design and operation are modeled after the neocortex, a sizable, intricate region of the human brain. It is thought that the neocortex processes sensory data, facilitates cognition, and stores and retrieves memories. [1]
By identifying intricate temporal patterns and relationships in data and drawing predictions about the future from it, HTM attempts to mimic the same basic functions of the neocortex. Like RNN techniques like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), it is especially well-suited for sequence learning modeling. In HTM, the term "hierarchy" describes a neural network's tiered architecture. HTM networks are made up of several neuronal layers. Every stratum carries out a distinct kind of calculation and data is sent between tiers below and above for additional processing. The environment provides input to the lower layers, such as sensory data. In these levels, the input is encoded into a distributed representation. Information can be represented in a distributed manner when only a small portion of it is active to signify the presence of a feature. To create sophisticated representations, the upper layers include data from the lower layers. As a result, the network is able to recognize more intricate and abstract patterns in the input data. [1]

In fact, the neocortex, or outer layer of the brain, and its structure and functions serve as inspiration for the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA), a machine learning algorithm. The neocortex is a densely interconnected network of neurons structured in layers that is in charge of higher-order cognitive processes in humans, including language processing, sensory perception, spatial reasoning, and decision-making. By using a hierarchical structure of regions and layers, each made up of computational units known as cortical columns, HTM CLA aims to imitate this design.[2]

There are a few main parts to HTM CLA that handle input data. The encoder first encodes the unprocessed input data. The sparse distributed representation (SDR), which is given to a spatial pooler, aids in the conversion of the raw input data. SDR is a technique for employing a limited number of active bits to encode information in binary format. This makes data processing, storage, and resilience more efficient. Using the encoder's output, the spatial pooler generates a new SDR that is more noise-resistant. The Temporal Memory component then processes this output. It is in charge of identifying and picking up on patterns in data.

The classifier component classifies input data by using the patterns it has learned from the temporal memory. Using previous data, it uses the identified patterns to forecast upcoming sequences. Within the HTM CLA system, the homeostatic plasticity controller is in charge of controlling the learning process. It guarantees that over time, the system will continue to be stable and flexible.

The first step in input data processing is encoding, where raw sensory data is transformed into a sparse distributed representation (SDR). Encoding real-world data into SDRs is a very important process to understand in HTM. Semantic meaning within the input data must be encoded into a binary representation. Encoders need to be first initialized with pre-defined settings using its constructor. HTM systems typically employ various encoding techniques tailored to the type of input data, such as scalar, spatial, or temporal encoding. [1] [3]

Once the input data is encoded, it is fed into the spatial pooling layer of the HTM network. Spatial pooling involves mapping the high-dimensional input space onto a lower-dimensional space of active columns. This process helps create a stable and distributed representation of the input data, making it more robust to noise and input variations. [1]

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

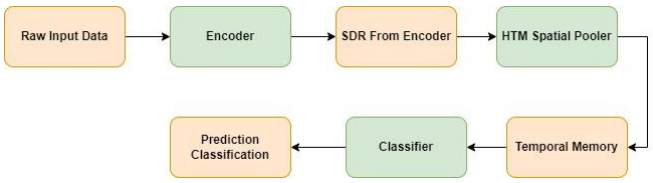

Figure 1: HTM System

After spatial pooling, the encoded input data is processed by the temporal memory layer of the HTM network. Temporal memory is responsible for capturing and learning temporal sequences and patterns in the input data. It maintains a predictive model of the input stream by forming predictive connections between active cells in different time steps. [1]

Once the HTM network has learned temporal patterns in the input data, it can make predictions and infer future states based on the current input and past context. During inference, the network activates predictive cells based on the input data, allowing it to anticipate the next likely input in the sequence. [1]

HTM systems continuously learn and adapt to changing input patterns through feedback mechanisms. When the predicted input matches the actual input, the network reinforces its connections and learns from the correct predictions. Conversely, when there's a prediction error, the network adjusts its connections to improve future predictions. [1]

In addition to inference and prediction, HTM systems can also detect anomalies or deviations from expected input patterns. Anomalies are identified when the input data significantly diverges from the learned predictive model, indicating potential unusual or unexpected events. [1]

II. Methods

We are going to use NeoCortex API [2], which is based on HTM CLA, for implementing our sample project in C#/.NET framework. For training and testing our experiment, we are going to use artificially generated network load 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 named AnomalyDetectionSample, train_data (or learning) (shown in Figure 2) and predict_data (shown in Figure 3). These folders will contain a few of these CSV files. predict_data folder contains data similar to training, but with added anomalies [Figure 3] randomly added inside it. We are going to read data from both train_data and predict_data folders then train them in our machine by using HTM Model. After that 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.

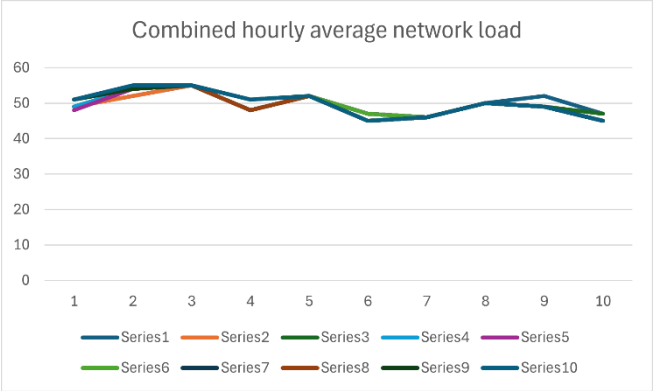

Figure 2: Graph of numerical sequences without anomalies which will be used for our training HTM model.

As artificially generated network traffic load data(in percentage, rounded off to the nearest integers) of a sample web server. The values of this load, taken over time, are represented as numerical sequences. For testing our prototype project, we will consider the values inside [45,55] as normal values, and anything outside it to be anomalies. Our predicting data comprises of anomalies between values between [0, 100] placed at random indexes. Combined data from both train_data and predict_data folder are given in Figure 3.

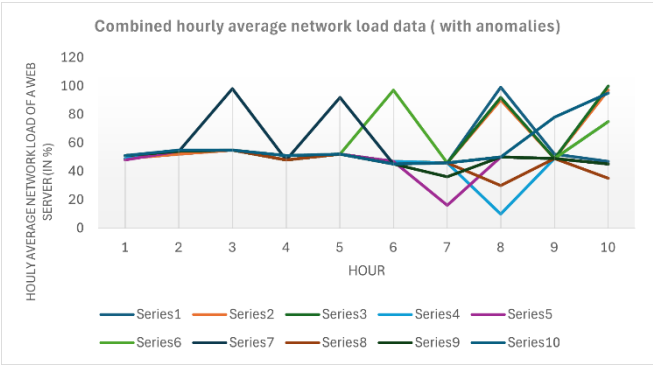

Figure 3: Graph of all numerical sequences with anomalies.

We are going to use multisequence learning class of NeoCortex API as base of our project. It will help use with both training our HTM model and using it for prediction. The class works in the following way: [6]

a. HTM Configuration is taken and memory of connections are initialized. After that, HTM Classifier, Cortex layer and Homeostatic Plasticity Controller are initialized.

b. After that, Spatial Pooler and Temporal Memory is initialized.

c. After that, spatial pooler memory is added to cortex layer and trained for maximum number of cycles.

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

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

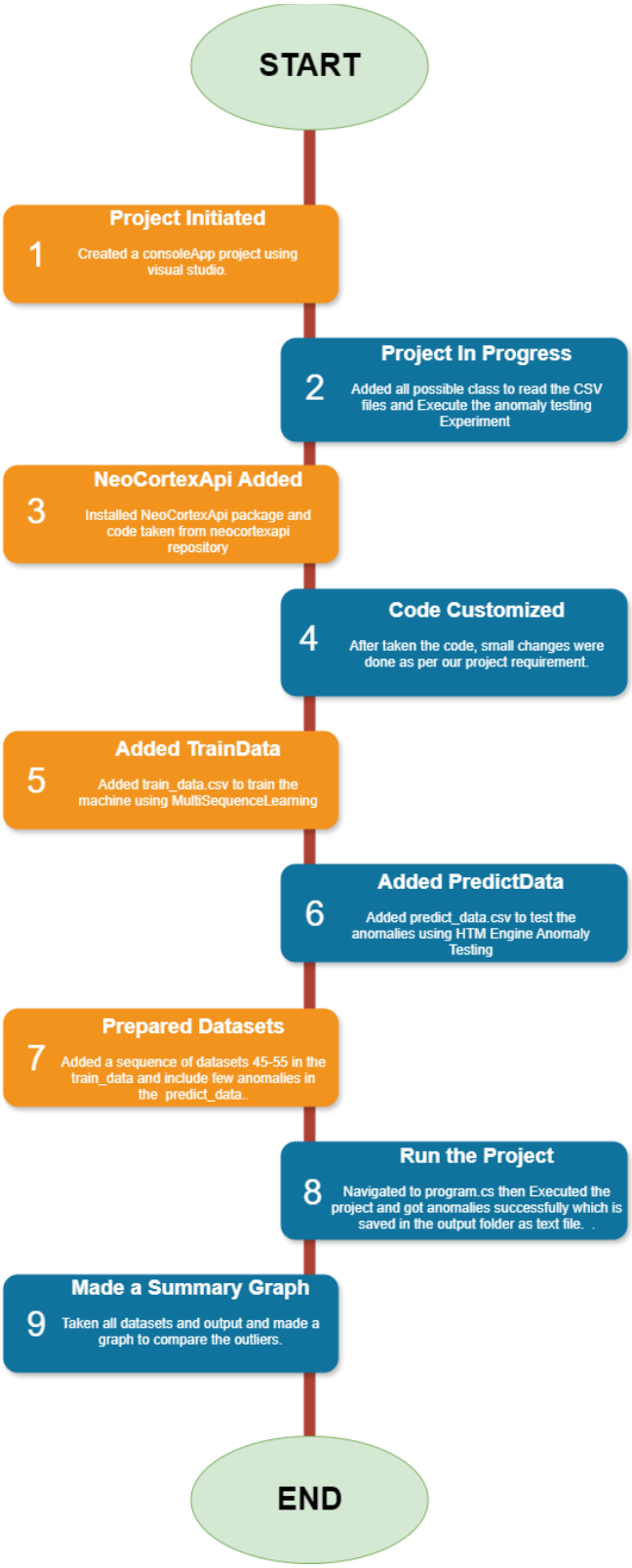

Figure 4: Working Process for Anomaly Detection

Encoder and HTM Configuration settings are needed to be passed to relevant components in this class. We are going to use the classifier object from trained HTM to predict value, which will be eventually used for anomaly detection.

We are going to train and test data between the range of integer values between 0 and 100 with no periodicity, so we are using the following settings given in listing 1. We are taking 21 active bits for representation. There are 101 values which represent integers between [0, 100]. We are calculating our input bits using n = buckets + w – 1 = 101+21-1 = 121. [3]

```
int inputBits = 121;
int numColumns = 1210;
.....…
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

Minimum and maximum values are set to 0 and 100 respectively, as we are expecting all the values to be in this range only. In other cases, these values must be changed depending on the input data. We have made no changes to the default HTM Config. [5]

Our project is executed in the following steps:

a. We have ReadFolder method of CSVReader_Folder class to read all the files placed inside a folder. Alternatively, we can use ReadFile method of CSVReader_File class to read a single file; it works in a similar way, except that it reads a single file. These classes store the read sequences to a list of numeric sequences, which will be used in a number of occasions later. These classes have exception handling implemented inside for handling non-numeric data. Data can be trimmed using TrimSequences method, which will be used in our unsupervised approach. Trimsequences method trims one to four elements (Number 1 to 4 is decided randomly) from the beginning of a numeric sequence and returns it. Both the methods are given in listing 2.

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

public static List<List<double>>
TrimSequences(List<List<double>> sequences)
        {        ....
         return trimmedSequences;
        }
```

Listing 2: Important methods in CSVReader_Folder class

b. After that, we have the method BuildHTMInput of CSVToHTM class converts all the read sequences to a format suitable for HTM training. It is shown in listing 3.

```
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. After that, we have the RunHTMTraining method of HTMTraining class, to train our model using multisequence class, as shown in listing 4. We will also combine the numerical data sequences from training (for learning) and predicting folders, and train the HTM model using this data. This class will return our trained model object predictor, which will be used later for prediction/anomaly detection.

```
.....
MultiSequenceLearning learning = new
MultiSequenceLearning(); predictor =
learning.Run(htmInput);
.....
```

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

d. We will use HTMAnomalyTesting class to detect anomalies. This class works in the following way,

- We pass on the paths of the training (learning) and predicting folder to the constructor of this class.
- The Run method encompasses all the important steps which we are going to help running this project from the beginning.

  1. At first, we start our model training using HTMModeltraining class by passing paths of the training and predicting folder path using constructor.

  2. After that, we use CSVReader_Folder class to read our test data from predict_data folder. Before starting our prediction, we use TrimSequences method of this class to trim a few elements in the front before testing, as shown in listing 5. This method trims between 1 to 4 elements of a sequence. The number between 1 to 4 is decided randomly, and it essentially returns a subsequence. We will use this data for predicting anomalies. Please note that the data read from predicting folder contains anomalies at random indexes in different sequences.

```
CSVReader_Folder testSequencesReader = new
CSVReader_Folder(_predictingFolderPath);
var inputSequences =
testSequencesReader.ReadFolder();
var trimmedInputSequences =
CSVReader_Folder.TrimSequences(inputSequences);
predictor.Reset();
```

Listing 5: Trimming sequences in testing data

  3. After that we pass on each sequences of the test data one by one to DetectAnomaly method. DetectAnomaly method is the method responsible for anomaly predictions in our data as shown in listing 6. We also placed an exception handling to handle non-numeric data, or if a testing sequence is too small (below 2 elements).

```
foreach (List<double> sequence in
trimmedInputSequences)
{
   double[] sequenceArray = sequence.ToArray();
   List<string> sequenceOutputLines = new
List<string>();

   try
   {
      sequenceOutputLines = DetectAnomaly(predictor,
sequenceArray, _tolerance);
   }
   catch (ArgumentException ex)
   {
      Console.WriteLine($"Exception caught:
{ex.Message}");
   }

   experimentOutputList.Add(sequenceOutputLines);
}
```

Listing 6: Passing of testing data sequences to RunDetecting method

e. RunDetecting method is the most important part of code in this project. We use this to detect anomalies in our test data using our trained HTM model.

f. After that, we use a TextOutput class to store our console information into text file to the output folder inside the project folder and calculate the accuracy with the TextOutput class.

```
public static class TextOutput
{
   public static double TrainingTimeInSeconds { get; set;
}
   public static string? OutputPath { get; set; }
   public static double TotalAvgAccuracy { get; set; }
}
```

Listing 7: Storing console output data where anomalies are detected or not to the output path.

This method traverses each value of the tested sequence one by one in a sliding window manner, and uses trained model predictor to predict the next element for comparison. We use an anomaly score to quantify the comparison, by taking absolute value of the difference between the predicted value and real value. If the prediction (absolute difference ratio) crosses a certain tolerance level (threshold value), preset to 10%, it is declared as an anomaly, and outputted to the user.

In our sliding window approach, naturally the first element is skipped, so we ensure that the first element is checked for anomaly in the beginning. So, in the beginning, we use the second element of the list to predict and compare the previous element (which is the first element). A flag is set to control the command execution; if the first element has anomaly, then we will not use it to detect our second element. We will directly start from second element. Otherwise, we will start from first element as usual.

When we traverse the list one by one to the right, we pass the value to the predictor to get the next value and compare the prediction with the actual value. If there's anomaly, then it is outputted to the user, and the anomalous element is skipped.

Upon reaching to the last element, we can end our traversal and move on to next list.

We get our prediction in a list of results in format of "NeoCortexApi.Classifiers.ClassifierResult`1[System.String] " from our trained model Predictor as shown in Listing 7.

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

Listing 8: Using trained model to predict data

Here, the item is the individual value from the tested list which is passed on the trained model. Let us assume that item passed to the model is of int type with value 8. We can use this to analyze how prediction works. The following code and the output given in listing 8 demonstrates how the predicted data can be accesed.

```
//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 9: Accesing predicted data from trained model

We know that the item we passed here is 8. The first line gives us the best prediction with similarity accuracy. We can easily get the predicted value which will come after 8 (here, it is 1), and previous value (11, in this case). We use basic string operations to get our required values.

The only downside in our approach is that we cannot detect two anomalies which are placed side by side, because as soon as an anomaly is detected, the code ignores the next anomalous element, as the anomalous element will result in incorrect predictions in the element next to it.

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)

where FN is the number of false negatives (i.e., the number of true anomalies incorrectly identified as normal), and TP is the number of true positives (i.e., the number of true anomalies correctly identified as anomalies).

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

where FP is the number of false positives (i.e., the number of normal observations incorrectly identified as anomalies) and TN is the number of true negatives (i.e., the number of normal observations correctly identified as normal).

Let us discuss the output of this experiment. For a brief analysis, we are going to discuss a part of our output next. If the sequence passed to our trained HTM engine is [52, 55, 48, 52, 47, 46, 99, 52, 47], we get the following output with respective accuracies.

Start of the raw output:

Testing the sequence for anomaly detection: 74, 75, 68, 72, 67, 66, 10, 68, 85.

Current element in the testing sequence: 74

Nothing predicted from HTM Engine. Anomaly detection failed.

Current element in the testing sequence: 75

No anomaly detected in the next element. HTM Engine found similarity: 95.83%.

Current element in the testing sequence: 68

No anomaly detected in the next element. HTM Engine found similarity: 100%.

Current element in the testing sequence: 72

No anomaly detected in the next element. HTM Engine found similarity: 91.67%.

Current element in the testing sequence: 67

No anomaly detected in the next element. HTM Engine found similarity: 87.5%.

Current element in the testing sequence: 66

****Anomaly detected**** in the next element. HTM Engine predicted: 70 with similarity: 100%, actual value: 10.

Skipping to the next element in the testing sequence due to detected anomaly.

Current element in the testing sequence: 68

****Anomaly detected**** in the next element. HTM Engine predicted: 72 with similarity: 100%, actual value: 85.

Skipping to the next element in the testing sequence due to detected anomaly.

Average accuracy for this sequence: 63.888888888888886%.

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

- Average FNR of the experiment: 0.65
- Average FPR of the experiment: 0.24

IV. Discussion

False negative rate indicates the ratio of missed anomalies (actual anomalies that are classified as normal). Having a higher false negative rate in our model is not desirable at all, and might be consequential in many cases, for example- fraud detection in financial transactions, etc. Generally, a model should have low FNR. Having low False positive rate is also desirable but not absolutely essential, Increased FNR, like when normal cases are marked as anomalies, can lead to unnecessary investigation of anomalous data and wasted effort. A lower value of both of them makes a good model for anomaly detection.

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.

In this experiment, we can see that the FNR is high, but we have tested our sample project in our local machine with less number of numerical sequences due to high

computational resource requirements and time constraint. This can be improved if high computation resources are used and more time is used for training the model in other platforms, like cloud. The results can be improved if we can use more amount of data and tune the hyper-parameters of our HTM model for best performance suited for our input data higher false negative rate in our model is not desirable at all, and might be consequential in many cases, for example- fraud detection in financial transactions, etc.

Generally, a model should have low FNR. Having low False positive rate is also desirable but not absolutely essential, Increased FNR, like when normal cases are marked as anomalies, can lead to unnecessary investigation of anomalous data and wasted effort. A lower value of both of them makes a good model for anomaly detection.

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.

In this experiment, we can see that the FNR is high, but we have tested our sample project in our local machine with less number of numerical sequences due to high computational resource requirements and time constraint. This can be improved if high computation resources are used and more time is used for training the model in other platforms, like cloud. The results can be improved if we can use more amount of data and tune the hyper-parameters of our HTM model for best performance suited for our input data.

V. REFERENCES

[1] A machine learning guide to HTM (Hierarchical Temporal Memory). (2019, October 24). Numenta. https://www.numenta.com/blog/2019/10/24/machine-learning-guide-to-htm/

[2] NeoCortexApi : https://github.com/ddobric/neocortexapi

[3] Dobric, D. (n.d.). source/Documentation/Encoders.md at master · ddobric/neocortexapi

[4] Lavin, A., & Ahmad, S. (2015). Evaluating real-time anomaly detection algorithms - the Numenta Anomaly Benchmark. In arXiv [cs.AI]. http://arxiv.org/abs/1510.03336

[5] Dobric, D. (n.d.). source/Samples/NeoCortexApiAnomaly/README.md at master · ddobric/neocortexapi.

[6] Sourav Paul, S. (n.d.). Team_anomaly: C#.net Implementation of Hierarchical Temporal Memory Cortical Learning Algorithm.

[7]