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

Implement Anomaly Detection Sample

Mahbubur Rahman
mahbubur.rahman@stud.fra-uas.de

Md Rakibul Islam
rakibul.islam@stud.fra-uas.de

Md Zihadul Islam Joni
zihadul.joni@stud.fra-uas.de

***Abstract*— HTM (Hierarchical Temporal Memory) is a machine learning algorithm, biologically inspired, both structurally and functionally, by neocortex of a human brain, which uses a hierarchical network of nodes to process time-series data in a distributed way. Each node, or column, can be trained to learn, and recognize patterns in input data. This can be used to process information, recognize, and identify patterns and make future predictions based on past learning. It is a promising approach for anomaly detection and prediction in a variety of applications in sectors, such as healthcare, finance, etc. We have implemented an anomaly detection sample using HTM, such that it learns multiple simple numeric integer sequences given to HTM model as input and try to learn patterns in it. It will then try to identify anomalies by comparing the real data with the predicted data from learning, with certain threshold of tolerance.**

Keywords—HTM, anomaly detection, machine learning, multi-sequence learning

I. INTRODUCTION

The Hierarchical Temporal Memory (HTM) algorithm is a machine learning algorithm based on the core principles of the Thousand Brains Theory. It takes its inspiration from the structure and function from neocortex- large, sophisticated, and complex part of the human brain. Neocortex is believed to be responsible for processing sensory information, cognition, and storing and retrieval of memory.

HTM tries to imitate the same fundamental working process of the neocortex by learning complex temporal patterns and relationships in a data and make future predictions from it. It is particularly suited for sequence learning modeling, similar to RNN methods like Long short term memory (LSTM) and Gated recurrent unit (GRU)[1]. Hierarchy in HTM refers to the layered structure of a neural network. HTM networks consist 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 processing it further. The lower layers receive input from the environment, like sensory data. The input is encoded into a distributed representation in these layers. Distributed representation is a way of representing information where only a few parts are active to indicate presence of some feature. The higher layers assimilate information from lower layers to form complex representations. This allows the network to identify more complex and abstract patterns in input data.

The Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) is a machine learning algorithm which is built in a way which tries to simulate the way neocortex processes information in human brain.

HTM CLA consists of a few major components for processing input information. In the beginning, the raw input data is encoded by the encoder. It helps to convert the raw input data into sparse distributed representation (SDR), which is then fed to a spatial pooler. SDR is a way of representing information in binary format using a small number of active bits. This allows for efficiency in storage, processing, and robustness of data. Spatial pooler creates new SDR from the output from encoder, and converts it into SDR more robust to noise. This output is then processed by the Temporal Memory component. It is responsible for recognizing and learning patterns in data. The Classifier component then classifies the input data based on patterns it has learned previously. It also makes predictions about the future patterns based on learned patterns and the input data. The component Homeostatic Plasticity Controller ensures that new patterns is learned by the system over time. Figure 1 shows how input data is processed in an HTM system.

Figure 1: HTM pipeline

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 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 comma separated value (CSV) files. There will be two folders inside our main project folder, training (or learning) and predicting. These folders will contain a few of these CSV files. Predicting folder contains data similar to training, 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.

As artificially generated data, we are going to take the network traffic load (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 sample 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 training and predicting folder is given in Figure 2.

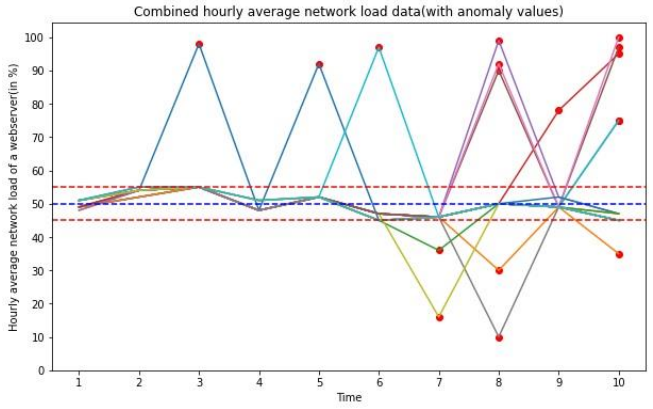

Figure 2: Graph of all numerical sequences which will be used for our training HTM model

We are going to use multisequencelearning 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:

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

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},
        };
```

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.

Our project is executed in the following steps:

a. We have ReadFolder method of CSVFolderReader class to read all the files placed inside a folder. Alternatively, we can use ReadFile method of CSVFileReader 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 CSVFolderReader class

b. After that, we have the method BuildHTMInput of CSVToHTMInput 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 RunHTMModelLearning method of HTMModeltraining 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 run 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 CSVFolderReader class to read our test data from predicting 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.

```
CSVFolderReader testseq = new
CSVFolderReader(_predictingFolderPath);
var inputtestseq = testseq.ReadFolder();
var triminputtestseq =
CSVFolderReader.TrimSequences(inputtestseq);
```

Listing 5: Trimming sequences in testing data

  3. After that we pass on each sequence 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> list in triminputtestseq)
        {
            double[] lst = list.ToArray();
            try
            {
                DetectAnomaly(myPredictor, lst);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine($"Exception caught:
{ex.Message}");
            }
        }
```

Listing 6: Passing of testing data sequences to DetectAnomaly method

e. DetectAnomaly 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.

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 anomalyscore 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 the last element, we can end our traversal and move on to the 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 7: 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 8: Accessing 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: 52, 55, 48, 52, 47, 46, 99, 52, 47.

First element in the testing sequence from input list: 52

No anomaly detected in the first element. HTM Engine found similarity to be:92.31%. Starting check from beginning of the list.

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: 16.67%, but the actual value is 55.

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: 48

****Anomaly detected**** in the next element. HTM Engine predicted it to be 75 with similarity: 51.06%, but the actual value is 52.

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: 47

Nothing predicted from HTM Engine. Anomaly cannot be detected.

Current element in the testing sequence from input list: 46

****Anomaly detected**** in the next element. HTM Engine predicted it to be 50 with similarity: 100%, but the actual value is 99.

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: 52

Anomaly not detected in the next element!! HTM Engine found similarity to be: 16.67%.

Current element in the testing sequence from input list: 47

End of input list. Further anomaly testing cannot be continued.

This is the end of the raw output.

As mentioned earlier, we have considered values within range of [45,55] to be non-anomalous. Anything outside of this range is considered an anomaly. We can then easily analyze our output data and calculate the FNR and FPR as following:

Anomaly detection results:

FN: 0

FP: 2

TN: 6

TP: 1

FNR = FN / (FN + TP) = 0

FPR = FP / (FP + TN) = 2/(2+6) = 0.25

Figure 3 illustrates the output for one input sequence, where the green values are the anomalies determined by our trained HTM model, and red marked value is the actual anomaly value in this sequence. Note that TP value (here, it is 99) is marked in both colours.

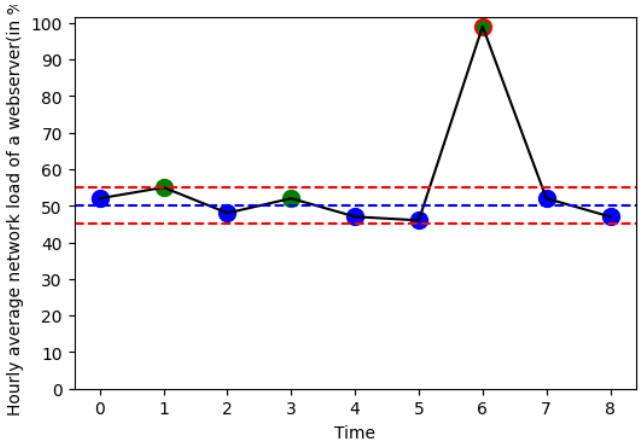

Figure 3: Graph of a simple numerical sequence to demonstrate how anomaly detection is calculated from output

After running our sample project, we analyzed the total output and calculated the average to get 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.

V. REFERENCES

[1] *"A Machine Learning Guide to HTM (Hierarchical Temporal Memory),"Numenta, https://www.numenta.com/blog/2019/10/24/machine-learning-guide-to-htm/ (accessed Mar. 28, 2023).*

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

[3] "Encoders." [Online]. Available: https://www.numenta.com/assets/pdf/biological-and-machine-intelligence/BaMI-Encoders.pdf. [Accessed: 29-Mar-2023].