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 the 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 the HTM model as input and tries to learn patterns in it. It will then try to identify anomalies by comparing the real data with the predicted data from learning, with a 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 of the neocortex- a large, sophisticated, and complex part of the human brain. The 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 data and making 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 the 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 that is built in a way that tries to simulate the way the neocortex processes information in the 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 the storage, processing, and robustness of data. The spatial pooler creates a new SDR from the output from the 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 prediction patterns based on learned patterns and the input data. The component Homeostatic Plasticity Controller ensures that new patterns are learned by the system over time. Figure 1 shows how input data is processed in an HTM system.

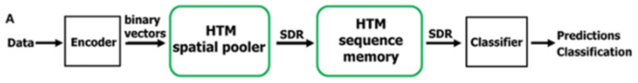
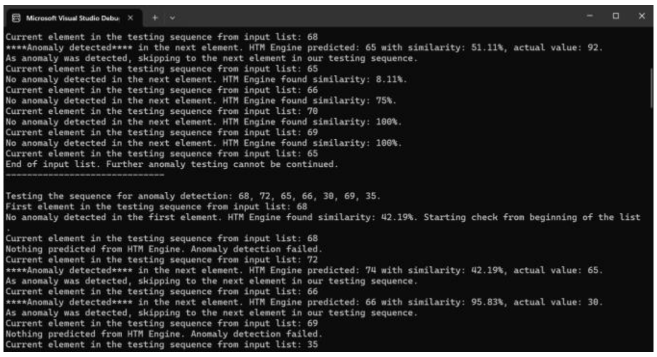


Figure 1: HTM pipeline.

II. METHODS

We are going to use Neocortex API [2], which is based on HTM CLA, for implementing our sample project in the 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 like training, but with added anomalies randomly added inside it. We are going to read data from both folders and train our HTM model using it. After that, we are going to take a part of the 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 [65,75] 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. output result of combined numerical sequence data from training folder (without anomalies) and predicting folder (with anomalies) can be found here.



```
Current element in the testing sequence from input list: 68
****Anomaly detected**** in the next element. HTM Engine predicted: 65 with similarity: 51.11%, actual value: 92.
As anomaly was detected, skipping to the next element in our testing sequence.
Current element in the testing sequence from input list: 65
No anomaly detected in the next element. HTM Engine found similarity: 8.11%.
Current element in the testing sequence from input list: 66
No anomaly detected in the next element. HTM Engine found similarity: 75%.
Current element in the testing sequence from input list: 70
No anomaly detected in the next element. HTM Engine found similarity: 100%.
Current element in the testing sequence from input list: 69
No anomaly detected in the next element. HTM Engine found similarity: 100%.
Current element in the testing sequence from input list: 65
End of input list. Further anomaly testing cannot be continued.
------------------------------

Testing the sequence for anomaly detection: 68, 72, 65, 66, 30, 69, 35.
First element in the testing sequence from input list: 68
No anomaly detected in the first element. HTM Engine found similarity: 42.19%. Starting check from beginning of the list
.
Current element in the testing sequence from input list: 68
Nothing predicted from HTM Engine. Anomaly detection failed.
Current element in the testing sequence from input list: 72
****Anomaly detected**** in the next element. HTM Engine predicted: 74 with similarity: 42.19%, actual value: 65.
As anomaly was detected, skipping to the next element in our testing sequence.
Current element in the testing sequence from input list: 66
****Anomaly detected**** in the next element. HTM Engine predicted: 66 with similarity: 95.83%, actual value: 30.
As anomaly was detected, skipping to the next element in our testing sequence.
Current element in the testing sequence from input list: 69
Nothing predicted from HTM Engine. Anomaly detection failed.
Current element in the testing sequence from input list: 35
```

Figure 2: Console Output of Anomalies

We are going to use multisequencelearning class of Neocortex API as a base of our project. It will help us 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 is initialized. After that, the HTM Classifier, Cortex layer, and HomeostaticPlasticityController are initialized.

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

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

d. After that, 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.

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. In the beginning, we have ExtractSequencesFromFolder method of CsvSequenceFolder class to read all the files placed inside a folder. These classes keep track of the read sequences in a list of numerical sequences that will be used repeatedly in the future. To handle non-numeric data, some classes have incorporated exception handling inside. With the Trimsequences technique, data can be trimmed. It returns a numeric sequence after trimming one to four components (numbers 1 through 4) from the start. Both the methods are given in listing 2.

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

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

Listing 2: Important methods in CsvSequenceFolder class

b. After that, the method ConvertToHTMInput of CSVToHTMInputConverter class is there which 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: ConvertToHTMInput method

c. After that, we have ExecuteHTMModelTraining method of HTMTrainingManager class to train our model using the converted sequences. The numerical data sequences from training (for learning) and predicting folders are combined before training the HTM engine. This class returns our trained model object predictor.

```
.....
MultiSequenceLearning learning = new
MultiSequenceLearning();
predictor = learning.Run(htmInput);
.....
.....
List<List<double>> combinedSequences = new
List<List<double>>(sequences1);
combinedSequences.AddRange(sequences2);
.....
```

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

d. We will use HTMAnomalyExperiment 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 that we are going to help run this project from the beginning.

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

2. After that, we use CsvSequenceReader 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.

```
.....
CsvSequenceFolder testSequencesReader = new
CsvSequenceFolder(_predictingCSVFolderPath);
var inputSequences =
testSequencesReader.ExtractSequencesFromFolder();
var trimmedInputSequences =
CsvSequenceFolder.TrimSequences(inputSequences);
.....
```

Listing 5: Trimming sequences in testing data

Path to training and predicting folder is set as default and passed on the constructor, or can be set inside the class manually.

```
.....
_trainingCSVFolderPath =
Path.Combine(projectBaseDirectory, trainingFolderPath);
_predictingCSVFolderPath =
Path.Combine(projectBaseDirectory,
predictingFolderPath);
.....
```

Listing 6: Path to training and predicting folder

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 7. 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 7: Passing of testing data sequences to DetectAnomaly method

Exception handling is present, such that errors thrown from DetectAnomaly method can be handled (like passing of non-numeric values, or the number of elements in the list of less than two).

e. DetectAnomaly method is the most important part of the 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 a trained model predictor to predict the next element for comparison. We use an anomaly score to quantify the comparison, by taking the 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 an anomaly, then we will not use it to detect our second element. We will directly start from the second element. Otherwise, we will start from the 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.

DetectAnomaly is the main method from the ExtractSequencesFromFolder class which detects anomalies in our data. It traverses each value of a list one by one in a sliding window manner and uses a trained model predictor to predict the next element for comparison. We use an anomaly score to quantify the comparison and detect anomalies; if the prediction crosses a certain tolerance level, it is declared as an anomaly. 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. We can 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 8.

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

Listing 8: Using trained model to predict data

Here, assume that item passed to the model is of int type with value 8. We can use this to analyze how prediction works. When this is executed,

```
//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: 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.

We will then use this to detect anomalies.

When we iteratively pass values to DetectAnomaly method using our sliding window approach, we will not be able to detect anomaly in the first element. 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 an anomaly, then we will not use it to detect our second element. We will directly start from the second element. Otherwise, we will start from the first element as usual.

Now, 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 an 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 use anomaly score (difference ratio) for comparison with our already preset threshold. When it exceeds, probable anomalies are found. To run this project, use the following class/methods given in [Program.cs].

```
HTMAnomalyExperiment tester = new
HTMAnomalyExperiment();
tester.ExecuteExperiment();
```

Listing 10: using program.cs class methods

III. Results

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 [72, 67, 66, 90, 69, 97], we get the following output with respective accuracies.

Start of the raw output:

Testing the sequence for anomaly detection: 72, 67, 66, 90, 69, 97.

Current element in the testing sequence from input list: 72

No anomaly detected in the next element. HTM Engine found similarity to be: 95.83%. Starting check from the beginning of the list.

Current element in the testing sequence from input list: 67

No anomaly detected in the next element. HTM Engine found similarity to be: 83.33%.

Current element in the testing sequence from input list: 66

****Anomaly detected**** in the next element. HTM Engine predicted: 70 with similarity: 100%, actual value: 90. Skipping to the next element in the testing sequence due to detected anomaly.

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: 69

****Anomaly detected**** in the next element. HTM Engine predicted: 72 with similarity: 100%, actual value: 97. Skipping to the next element in the testing sequence due to detected anomaly.

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

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

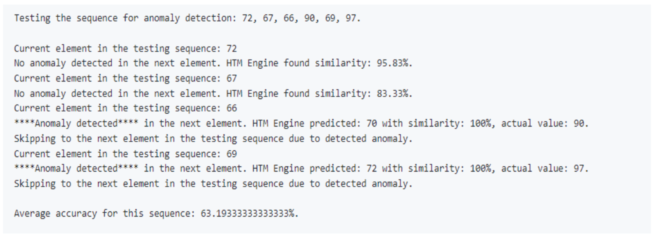
The average accuracy for this sequence: 63.19333333333333%.

As we can see, the accuracy rate ranges from 50% to 70%. In an anomaly detection algorithm, a high degree of accuracy on the sequence is desired. Our machine's hardware specifications prevent us from running programs with a lot of cycles and sequences. Nevertheless, by executing more data sequence and cycle, accuracy can be increased.

As mentioned earlier, we have considered values within range of [65,75] to be non-anomalous. Anything outside of this range is considered an anomaly. We can then easily analyze our output data and calculate the accuracy.

After the project's execution, we obtained the following Outputs.

One testing sequence for anomaly detection and average accuracy for this sequence given below: Testing the sequence for anomaly detection: [72, 67, 66, 90, 69, 97].



```
Testing the sequence for anomaly detection: 72, 67, 66, 90, 69, 97.

Current element in the testing sequence: 72
No anomaly detected in the next element. HTM Engine found similarity: 95.83%.
Current element in the testing sequence: 67
No anomaly detected in the next element. HTM Engine found similarity: 83.33%.
Current element in the testing sequence: 66
****Anomaly detected**** in the next element. HTM Engine predicted: 70 with similarity: 100%, actual value: 90.
Skipping to the next element in the testing sequence due to detected anomaly.
Current element in the testing sequence: 69
****Anomaly detected**** in the next element. HTM Engine predicted: 72 with similarity: 100%, actual value: 97.
Skipping to the next element in the testing sequence due to detected anomaly.

Average accuracy for this sequence: 63.19333333333333%.
```

Figure 3: testing sequence for anomaly detection and average accuracy for this sequence.

IV. Discussion

Accuracy rate indicates the ratio of missed anomalies (actual anomalies that are classified as normal). Having a lower accuracy 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 high accuracy. Having a high accuracy rate is also desirable but not essential, increased accuracy rate, like when normal cases are marked as anomalies, can lead to unnecessary investigation of anomalous data and wasted effort. A lower value of both 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 accuracy rate is 50 to 70 percent, 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 constraints. This can be improved if high computation resources are used and more time is used for training the model in other platforms, like the cloud. The results can be improved if we can use more amount of data and tune the hyper-parameters of our HTM model for the 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].

[4] Anomaly detection: Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM computing surveys (CSUR), 41(3), 15.

[5] HTM: Numenta (2022). HTM technology. https://numenta.com/technology/htm/.

[6] Neocortex: Mountcastle, V. B. (1997). The columnar organization of the neocortex. Brain, 120(4), 701-722.K. Elissa, "Title of paper if known," unpublished.

[7] Continuous learning: Ahmad, S., & Hawkins, J. (2016). How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. arXiv preprint arXiv:1601.00720.

[8] Applications of HTM: Lavin, A., & Ahmad, S. (2015). Evaluating Real-time Anomaly Detection Algorithms - The Numenta Anomaly Benchmark. In Proceedings of the 14th International Conference on Machine Learning and Applications (ICMLA) (pp. 38-44).

[9] HTM for stock price prediction: Razavian, A. S., Amini, N., & Abolhasani, M. (2017). A deep learning framework for financial time series using stacked auto-encoders and long-short term memory. Journal of finance and data science, 3(3), 141-149.