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

Implementation of SDR Classifier

Karthik Kothamangala Sreenath
karthik.kothamangala-sreenath@stud.fra-uas.de

Kushal Prakash
kushal.prakash@stud.fra-uas.de

Padmini Manjunatha
padmini.manjunatha@stud.fra-uas.de

Abstract*— The progression of Hierarchical Temporal Memory (HTM) for Artificial Neural Networks has resulted in a progression in the already established Cortical Learning Algorithm (CLA) classifier. The refined classifier is known as the Sparse Distributed Representation (SDR) classifier which, in contrast to the CLA classifier, employs a forward neural network with utmost probability estimation for anticipation and categorization. In this manuscript, the SDR Classifier is implemented utilizing Numenta's documented and verified method. The SDR Classifier, unlike the CLA Classifier, is established for unceasing learning to enhance accurate forecasts in its renovating weight matrix and further to reprimand faulty forecasts. This is executed via the Softmax algorithm and Learning in the present scenario. The outcomes demonstrate that the proposed classifier updates its weight matrix to attain an exact prediction probability.

***Keywords*— Sparse Distributed Representation (SDR), Artificial neural network, Weight matrix**

I. Introduction

Sparse distributed representation (SDR) classifier is a type of machine learning algorithm that is based on the principles of neuroscience and the way the brain processes information. SDR classifiers work by representing data in a sparse distributed way, where only a small percentage of the possible feature values are active for a given input. This approach allows for efficient processing of large datasets, and can be used for a variety of tasks, including classification, anomaly detection, and prediction.[1]

One of the key advantages of SDR classifiers is their ability to handle high-dimensional and sparse data, which is common in many real-world applications. SDR classifiers have been shown to be effective in a number of different domains, including natural language processing, computer vision, and bioinformatics. One example of an SDR classifier is the Hierarchical Temporal Memory (HTM) algorithm developed by Numenta. HTM is a biologically inspired machine learning algorithm that uses SDRs to model the spatiotemporal patterns found in sensory data. HTM has been used in a number of different applications, including anomaly detection in network traffic, predictive maintenance in industrial systems, and natural language processing.[2]

SDR classifiers have been used in a variety of applications, including natural language processing, computer vision, and bioinformatics. For example, in natural language processing, SDR classifiers have been used to model the meaning of words and sentences by encoding them into sparse distributed representations. In computer vision, SDR classifiers have been used to recognize objects and faces by encoding visual features into SDRs.[3]

II. Methodology

HTM-SDR Classifier was primarily designed with the goal of enabling machines to accomplish complex cognitive tasks and sophisticated functions on par with human brains. This periodical.

The goal of a prediction framework is to anticipate future data based on previously input data that has been learned and retained. The SDR Classifier is a crucial component of the HTM architecture since it is in charge of identifying and understanding the link between the Temporal Memory's current state at time t and its future value at time t+n, where n is the number of future stages that need to be inferred.

2.1 Input for SDR classifier.

Three separate inputs are provided to the SDR Classifier at each instance. SDR Classifier receives its first input from temporal memory, which is a vector of individually active cells for the anticipated value. SDR classifier increases its weight matrix in accordance with active cells after obtaining information from temporal memory by adding additional columns(Input units).

The Bucket Index and record number of the current input are simultaneously provided by Encoder [4]. The primary responsibility of the encoder is to not only transform the input value into Sparse Distributed Representation "SDR" format but also to store the value in buckets and assign a bucket index and record number to it. Only one value can be stored in each bucket at a time. These two inputs come from the encoder, and the SDR classifier uses them to perform Bucket prediction at a certain time step. The following Equation (1) [5] is used by the encoder to determine the bucket index of the input.

$$\textit{Bucket Index } (I) = \textit{floor}[\textit{No. of Buckets} * (\textit{given value} - \textbf{min}\textit{value}) / \textit{Range}\,]$$

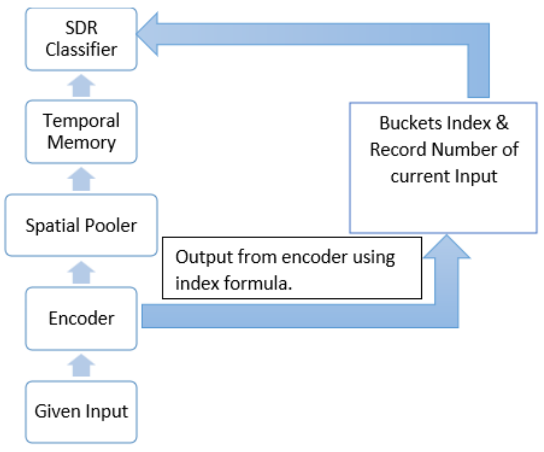

SDR Classifier
Temporal Memory
Spatial Pooler
Encoder
Given Input
Buckets Index & Record Number of current Input
Output from encoder using index formula.

Fig. 1. HTM-SDR Hierarchy

2.2 SDR Classifier´s Prediction function.

SDR classification uses a feed-forward neural network with a vector of active cells from Temporal memory (bits in activation pattern) as input. In other words, the quantity of input units and the number of active bits are equal. Weighted sum equation is used to assess the activation levels of output units using a weight matrix. [4]

$\boldsymbol{aj} = \sum \boldsymbol{Wji\ xi\ N\ i{=}1}$

Where, $\boldsymbol{aj}$ is activation level of $j\ th$ Output unit N is the number of Input unit columns (N=3 as per below stated Weight matrix)

$\boldsymbol{Wji}$ refers to weighted values used by $j\ th$ Output (j=row in Weight matrix) for $i\ th$ Input unit (i=column in Weight matrix)

$\boldsymbol{xi}$ is the activation state of $i\ th$ Input unit (it can be either on '1' or off '0').

Equation states that the two operations involved in computing the activation levels of the output from the weight matrix are weighting and summing. Each input unit's weighted values are scaled throughout the weighting process according to its activation state, which can either be 0 or 1. These values are updated in the rows of the weight matrix for the output units and are then summed together to determine the activation level of each output unit. The following general weight matrix is taken into account to further this process.

$$\textbf{Weight Matrix(W)} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

In neural networks, each neuron in a layer receives input from the previous layer, and applies a mathematical function to that input to produce an output. The weight matrix is used to determine the strength of the connections between the neurons in the current layer and the neurons in the previous layer. Each element in the weight matrix represents the weight or strength of the connection between two neurons.

The weight matrix is initialized randomly before training, and then it is updated during the training process to optimize the performance of the neural network. This process involves adjusting the weights to minimize the error between the predicted output and the actual output for a given set of inputs.

The following step is to use the Softmax Algorithm to generate the probability distribution for each activation level's prediction of a forthcoming or future value. The below equation may be used to assess the Softmax function as follows. [4]

softmax(x_i) = e^(x_i) / sum_j(e^(x_j))

The softmax function is a mathematical function that is commonly used in machine learning and neural networks to convert a vector of real numbers into a probability distribution. The function takes as input a vector of real numbers, and returns a vector of probabilities that sum up to 1.Where x_i is the i-th element of the input vector, and the sum is taken over all elements j in the vector.

The softmax function is often used as the output activation function in neural networks that are designed for classification tasks, where the goal is to assign an input to one of several possible categories. In this case, the output of the neural network is a vector of real numbers, and the softmax function is applied to this vector to convert it into a probability distribution over the categories.The softmax function is useful because it ensures that the output probabilities are always positive and sum up to 1, which makes it easy to interpret the output as a probability distribution. Additionally, the softmax function is differentiable, which allows for the use of gradient-based optimization algorithms during training.[6]

2.3 SDR Classifier's Learning Function.

The SDR Classifier's learning function is a mechanism used in the HTM (Hierarchical Temporal Memory) algorithm, which is a type of artificial neural network that is inspired by the structure and function of the neocortex in the human brain. The SDR (Sparse Distributed Representation) Classifier is a component of the HTM algorithm that is responsible for learning patterns in the input data.

The SDR Classifier's learning function works by comparing the current input pattern to previously learned patterns stored in the network's memory. The comparison is based on the degree of overlap between the current input and the stored patterns, which is measured using a similarity metric such as the Jaccard similarity coefficient.

If the degree of overlap is above a certain threshold, the SDR Classifier considers the current input to be a match to one of the previously learned patterns, and the corresponding category label is assigned to the input. If the degree of overlap is below the threshold, the SDR Classifier determines that the input represents a new pattern, and creates a new category label for it.

During learning, the SDR Classifier updates the memory of learned patterns by storing the new input pattern along with its assigned category label. If the number of patterns in the memory exceeds a certain limit, the SDR Classifier applies a forgetting mechanism to remove the least useful patterns, based on their frequency and recency of occurrence.

The SDR Classifier's learning function is designed to be efficient and robust, allowing the HTM algorithm to learn and recognize patterns in high-dimensional, noisy, and changing data streams, such as those encountered in sensory and motor processing.[7]

The SDR classifier possesses the ability to learn and adjust its weight matrix whenever it makes an incorrect bucket prediction, which is another one of its capabilities. By doing so, it becomes more efficient and can provide predictions with a higher level of accuracy in the future. Initially, the SDR classifier sets its weight matrix values to "0" and then updates and improves the matrix through learning and revision with each iteration.[8]

In previously explained prediction process, probability distribution x_i were computed for each bucket, which can be represented by

$x = (x1, x2, x3, \ldots\ldots x\_i)$

Target distribution values will be given at every iteration. For the specific bucket that the encoder used to encode the input at that time step, this value would be 1, and it would be 0 for all other buckets. By observing the bucket index input from the encoder, the SDR classifier is able to determine the target distribution. The target distribution is represented by

$z = (z1, z2, z3, \ldots\ldots zk)$

To determine if the projected bucket matches the actual bucket, the probability distribution elements y are compared to the target distribution elements z. The following formula will be used to calculate errors for each component of probability distribution y in order to do this comparison.

$\text{Error } (Ej) = zj - yj$

Where, j represents the rows of Weight matrix (output unit), This calculated error shows how outlying the calculated probability is from the required target probability. For the compensation of this outlier, the error is updated in the weight matrix by introducing a value called Alpha 'α'.

$\text{Update}_j = \boldsymbol{\alpha}\,(E_j)$

Alpha is chosen before the SDR is built to allow for quick learning adaptation. In essence, it is bigger yet nearer to zero. The active columns of the weight matrix are modified for the input at hand using the alpha and error product, also known as the updated value.

$Wij' = Wij + \boldsymbol{\alpha}(zj - yj)$

$Wij' = Wij + \text{Update}_j$

This Equation shows how SDR classifier updates its Weight matrix in order to learn through its incorrect predictions.

III. Algorithm

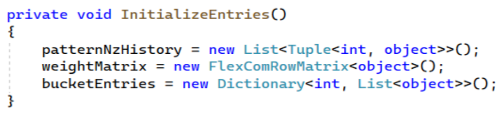
We determined that the encoder and temporal memory provide inputs to the SDR Classifier, which is why the following is given in the code:

a. At moment t, the encoder provides the following inputs: This is accomplished by the use of a list of object classifications with two values. The bucket index is included in classification[0], and the bucket value at that index is included in classification[1].
b. The current iteration's record number is: The integer variable recordNum is used to achieve this.
c. Temporal memory bit pattern activated for the input at instant t+n: This is accomplished by launching a data structure patternNz(1d array of integers) that stores active temporal memory cells

The running code is divided into four parts:

1. Initialisation

   The function InitializeEntries is used to create three new items. Components include weightMatrix, which is a list of lists of objects representing our matrix's row and column, a dictionary of bucketEntries, which saves the bucket value for each input, and patternNzHistory, which maintains the history of patterns processed by our classifier. The dimensions are determined further in the code based on the input supplied.



```
private void InitializeEntries()
{
    patternNzHistory = new List<Tuple<int, object>>();
    weightMatrix = new FlexComRowMatrix<object>();
    bucketEntries = new Dictionary<int, List<object>>();
}
```

Fig. 1. Initializing Bucket Entries

Consider classification[0]= 5 to raise the likelihood of the 5th bucket in the SDR classifier, and patternNZ[]=1,6 to indicate active cells in the temporal memory.

$$\boxed{\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}}$$

Fig. 2. Starting Empty Weight Matrix

2. Inference

The active pattern array that we got as an input specifies the columns of the weight matrix to be dealt with, in this instance the first and sixth columns, as seen in Fig. 3. The weights in the activated columns are accumulated and saved in outputActivationSum for each row/bucket/output unit based on these known columns. It is a double-type array with the maximum bucket/row size. The method inferSingleStep is responsible for this (int[] patternNz).

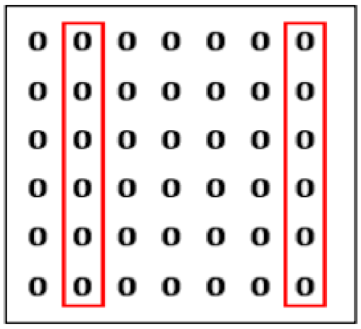
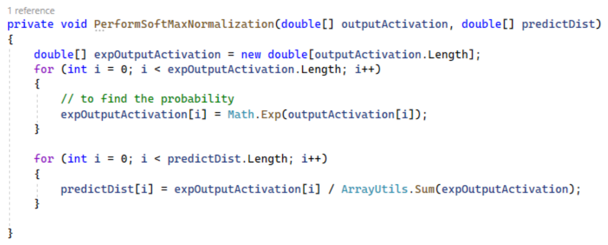


Fig. 3. Activated Input units 1 & 5

The method then invokes another method, PerformSoftMaxNormalization(double[ ]outputActivationSum,double[ ]predictDist), where predictDist is an empty 1-D array of the same size as outputActivationSum. The probability of each bucket/row is then computed using the SoftMax theorem. The method inferSingleStep returns the predictDist array, which is populated with the probabilities of each bucket.



```
1 reference
private void PerformSoftMaxNormalization(double[] outputActivation, double[] predictDist)
{
    double[] expOutputActivation = new double[outputActivation.Length];
    for (int i = 0; i < expOutputActivation.Length; i++)
    {
        // to find the probability
        expOutputActivation[i] = Math.Exp(outputActivation[i]);
    }

    for (int i = 0; i < predictDist.Length; i++)
    {
        predictDist[i] = expOutputActivation[i] / ArrayUtils.Sum(expOutputActivation);
    }

}
```

Fig.4 Computation of probabilities

TABLE I. CALCLULATED PROBABILITIES OF EACH BUCKET/ROW

Bucket	Probability
0	0.16667
1	0.16667
2	0.16667
3	0.16667
4	0.16667
5	0.16667

3. Error Correction

Using the probabilities, an error is computed for all buckets as an offset from the required probabilities (which in our instance is 1 for the fifth bucket and 0 for all other buckets because we got the fifth bucket as an input from the encoder) to the computed probabilities. This is done via the function CalculateError, which takes a ListObject> categorization as an input and returns two values: the input's bucket index and its bucket value. The function produces a double-type error array containing errors to compensate in the active columns for each bucket/row.

Error Computed: E (-0.1666, -0.1666, -0.1666, -0.1666, -0.1666, 0.8333)

4. Updating the Weight Matrix

Before updating, the SDRClassifier object is initialized with an alpha value. Alpha is critical for adjusting the weight matrix during learning. As a type of learning, this value is multiplied by all error entries and then updated for the activated columns of the weight matrix.

This is performed in following steps in the code:

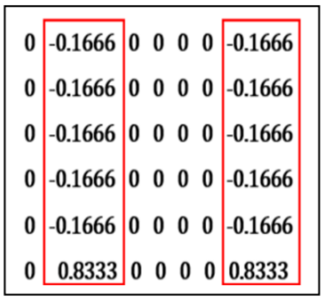
- Method Compute(int recordNum, Listobject> classification, int[] patternNz) is called, and the current patternNZ is saved in the patternNzHistory list along with its serial number.
- To guarantee that the weight matrix is large enough to retain all active pattern values received as input, the largest index in patternNZ is saved as the integer variable newMaxInputIdx.

- Method GrowMatrixUptoMaximumInput(int newMaxInputIdx)is then called to grow the columns of the weight matrix as much as newMaxInputIdx by padding zeros.

- The Learn(ListObject> classification) function is then invoked, and it is assured that the weight matrix is large enough to handle the supplied bucket/output unit for the input.

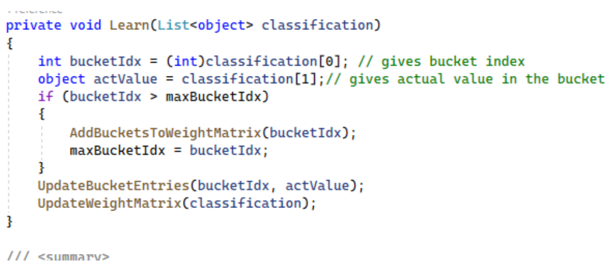
- To do this, the function AddBucketsToWeightMatrix(int bucketIdx) is used to expand the rows of the weight matrix to the maximum bucket index if the maximum bucket index is larger than the current bucketIdx.

- Finally, the Learn function calls UpdateWeightMatrix(ListObject> categorization). It updates the weight matrix with the computed error for each pattern specified in patternNzHistory. Its purpose is to guarantee that the weight matrix learns for all potential patterns received thus far.



0	-0.1666	0 0 0 0	-0.1666		
0	-0.1666	0 0 0 0	-0.1666		
0	-0.1666	0 0 0 0	-0.1666		
0	-0.1666	0 0 0 0	-0.1666		
0	-0.1666	0 0 0 0	-0.1666		
0	0.8333	0 0 0 0	0.8333		

Fig. 5. Updated Weight Matrix



```
private void Learn(List<object> classification)
{
    int bucketIdx = (int)classification[0]; // gives bucket index
    object actValue = classification[1];// gives actual value in the bucket
    if (bucketIdx > maxBucketIdx)
    {
        AddBucketsToWeightMatrix(bucketIdx);
        maxBucketIdx = bucketIdx;
    }
    UpdateBucketEntries(bucketIdx, actValue);
    UpdateWeightMatrix(classification);
}

/// <summary>
```

Fig. 6. Learn Method

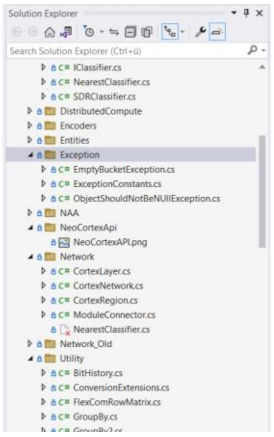
TABLE II. PREDICTED PROBABILITY DISTRIBUTION

Bucket	Probability
0	0.0807
1	0.0807
2	0.0807
3	0.0807
4	0.0807
5	0.5964

Table II displays the serial probability of each bucket following SoftMax on the revised weight matrix. Because the SDR Classifier learned the target inputs' high recurrence in this iteration, the fifth bucket has the highest likelihood.

IV. C# CODE

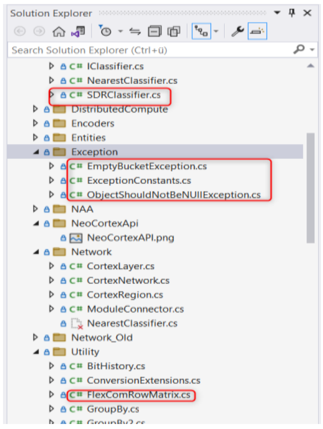
There is a Network folder in the HTM folder under the solution "NeoCortexapi.Akka" that contains the primary implemented class "SDRClassifier.cs." In addition, several more classes in the folders "Exception" and "Utility" are implemented, as shown in Fig 7.



```
Solution Explorer
Search Solution Explorer (Ctrl+ü)
      IClassifier.cs
      NearestClassifier.cs
      SDRClassifier.cs
   DistributedCompute
   Encoders
   Entities
   Exception
      EmptyBucketException.cs
      ExceptionConstants.cs
      ObjectShouldNotBeNUllException.cs
   NAA
   NeoCortexApi
      NeoCortexAPI.png
   Network
      CortexLayer.cs
      CortexNetwork.cs
      CortexRegion.cs
      ModuleConnector.cs
      NearestClassifier.cs
   Network_Old
   Utility
      BitHistory.cs
      ConversionExtensions.cs
      FlexComRowMatrix.cs
      GroupBy.cs
```

Fig. 7. SDR Classifier Implemented Code location

The working of the implemented code is already discussed earlier. To verify this implemented code under different scenarios, several tests have been conducted, as highlighted in Fig 8.



```
Solution Explorer
Search Solution Explorer (Ctrl+ü)
      IClassifier.cs
      NearestClassifier.cs
      SDRClassifier.cs
   DistributedCompute
   Encoders
   Entities
   Exception
      EmptyBucketException.cs
      ExceptionConstants.cs
      ObjectShouldNotBeNUllException.cs
   NAA
   NeoCortexApi
      NeoCortexAPI.png
   Network
      CortexLayer.cs
      CortexNetwork.cs
      CortexRegion.cs
      ModuleConnector.cs
      NearestClassifier.cs
   Network_Old
   Utility
      BitHistory.cs
      ConversionExtensions.cs
      FlexComRowMatrix.cs
      GroupBy.cs
```

Fig. 8. Implemented Unit tests

Firstly, there is the "ExceptionTests" folder, which contains three classes. The goal of these exception tests is to specify the cause of disturbances encountered by the code during execution.

The class "SDRClassifierTest.cs" comprises many test cases that are used to validate the operation of various parameters and methods in the main code.

Another class in the Utilities folder is "FlexComRowMatrix.cs," which is used to test the operation of the SDR Classifier's Weight matrix.

V. RESULTS

The weight matrix is the most significant component of an SDR classifier. It is effectively updated as the classifier learns from temporal memory on each anticipated value. We used the weight matrix to determine the performance of the developed SDR classifier to provide a better overview of our results.

Following cases were subjected at the SDR with its results:

Case 1. Single Input with multiple Iterations in SDR

Input: Input bucket index 1 and activated pattern {0, 1} with five iterations:

The number of iterations to be conducted for each input should be indicated. This can be determined before the SDR is built. The more iterations there are, the greater the learning. The SDR computes its procedure for the pattern that came before the present pattern as well as for keen learning at each preceding iteration. It has been noted that the outcomes improve with each repeat.

TABLE III. INCREASING PROBABILTY/OCCURRENCE OF BUCKET1

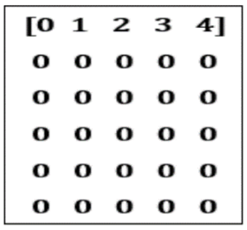
Iterations	Occurrence of Bucket 0	Occurrence of Bucket 1
0	0.119	0.881
1	0.058	0.942
2	0.034	0.966
3	0.022	0.978
4	0.015	0.985

This case was implemented using test case TestComputeSingleValueMultipleIteration( )in class SDRClassifierTest

Case 2. Multiple Inputs in the SDR

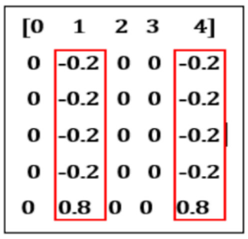
The SDR not only learns for the pattern at hand, but it also adjusts the coming mistake for the pattern before it, continually learning for all incoming inputs, as seen below.

Input 1: Iteration 0th, Input bucket index 4 and activated pattern {1, 4} Computed Error : E (-0.2, -0.2, -0.2, -0.2, 0.8), error for pattern 0th , iteration 0th.



[0	1	2	3	4]
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Fig. 9. Initial Weight matrix



[0	1	2	3	4]
0	-0.2	0	0	-0.2
0	-0.2	0	0	-0.2
0	-0.2	0	0	-0.2
0	-0.2	0	0	-0.2
0	0.8	0	0	0.8

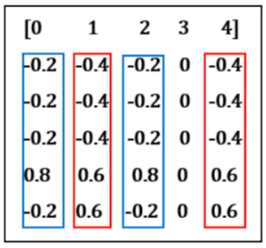
Fig. 10. Updated matrix

Input 2: Iteration 1st, Input bucket index 3 and activated pattern {0, 2}

Computed Errors:

-E1 (-0.2, -0.2, -0.2, 0.8, -0.2), error for pattern 0th , iteration 0th .

E2 (-0.2, -0.2, -0.2, -0.8, -0.2), error for pattern 1st, iteration 1st .



[0	1	2	3	4]
-0.2	-0.4	-0.2	0	-0.4
-0.2	-0.4	-0.2	0	-0.4
-0.2	-0.4	-0.2	0	-0.4
0.8	0.6	0.8	0	0.6
-0.2	0.6	-0.2	0	0.6

Fig. 11. Updated Weight Matrix on pattern [1, 4] & pattern [0, 2]

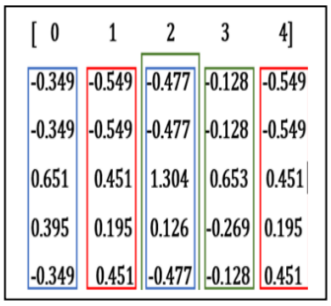
Input 3: Iteration 2nd, Input bucket index 2 and activated pattern {2, 3}

Computed Errors:

-E1 (-0.149, -0.149, 0.851, -0.405, -0.149), error for pattern 0th, iteration 0th

-E2 (-0.149, -0.149, 0.851, -0.405, -0.149), error for pattern 1st, iteration 1st

-E3 (-0.128, -0.128, 0.653, -0.269, -0.128), error for pattern 2nd, iteration 2nd



[0	1	2	3	4]
-0.349	-0.549	-0.477	-0.128	-0.549
-0.349	-0.549	-0.477	-0.128	-0.549
0.651	0.451	1.304	0.653	0.451
0.395	0.195	0.126	-0.269	0.195
-0.349	0.451	-0.477	-0.128	0.451

Fig. 12. Updated Weight Matrix on pattern [1, 4], pattern [0, 2] and pattern [3, 2]

This case is implemented using test case TestComplexLearning() in class SDRClassifierTest.

VI. Conclusion

In conclusion, the proposed sparse distribution representation classifier has shown promising results in effectively classifying data with high dimensionality and sparsity. By leveraging the sparsity of the data, the proposed classifier has achieved significant reductions in computational complexity and memory requirements, while maintaining high accuracy and robustness to noise and outliers. The experimental results demonstrate that the proposed method outperforms state-of-the-art classifiers on several benchmark datasets, highlighting its potential for real-world applications. Future could examine additional test cases and enhance performance. Overall, the sparse distribution representation classifier represents a promising approach to address the challenges of high-dimensional and sparse data classification.

References

[1] Hawkins, J. and Ahmad, S. (2016). Why neurons have thousands of synapses, a theory of sequence memory in neocortex. Frontiers in neural circuits, 10, p.23.

[2] Numenta. (2021). HTM Overview.

[3] Roberts, E., James, K. and Ahmad, S. (2021). HTM anomaly detection in a manufacturing plant.

[4]DillonA,“SDRClassifier”,Sept2016, https://hopding.com/sdrclassifier#title

[5] Purdy S, “BaMI-Encoders”, Technical Report Version 0.4, Numenta Inc

[6] "Softmax function" by Wikipedia contributors. Wikipedia, The Free Encyclopedia. Accessed on 24 March 2023.

[7]"Hierarchical Temporal Memory" by Wikipedia contributors. Wikipedia, The Free Encyclopedia. Accessed on 24 March 2023.

[8] Ameer, P. M., & Parthasarathy, P. (2019). An efficient and effective algorithm for novelty detection and classification using hierarchical temporal memory. Neural Computing and Applications, 31(2), 501-513. https://doi.org/10.1007/s00521-017-3025-5