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

Implementation of SDR Classifier

Chetan Chadha
Chetan.chadha@stud.fra-uas.de

Jagdeep Singh
Jagdeep.singh@stud.fra-uas.de

Maria Majid
Maria.majid@stud.fra-uas.de

Sidra Hussain
Sidra.hussain@stud.fra-uas.de

Sanjana Sharma
Sanjana.sharma@stud.fra-uas.de

Vishakha Babulal
Vishakha.babulal@stud.fra-uas.de

***Abstract*—The development of Hierarchical Temporal Memory (HTM) for Artificial Neural Networks has led to an advancement in the already known Cortical Learning Algorithm (CLA) classifier. The improved classifier is called Sparse Distributed Representation (SDR) classifier which unlike the CLA classifier uses feed forward neural network with maximum likelihood estimation for prediction and classification. In this paper the SDR Classifier is implemented using Numenta's documented and tested approach. The SDR Classifier, unlike CLA Classifier is established for continuous learning to reinforce correct predictions in its updating weight matrix and additionally to penalize incorrect predictions. This is executed using Softmax algorithm and Learning in the current case. The results show the proposed classifier updates its weight matrix for reaching accurate prediction probability.**

Keywords— ***Feed-Forward neural network, probabilistic distribution, Maximum Likelihood estimation***

I. INTRODUCTION

In this era of Artificial intelligence and Machine learning, the Human neocortex inspired cognitive learning algorithm Hierarchical temporal memory (HTM) is undergoing so many developments in order to provide Human neocortex functionalities like storing, learning, predicting or inferring wide-ranging data sequences [1].

HTM-SDR Hierarchy consists of different stages including Encoder, Spatial pooler, Temporal Memory and SDR Classifier. In HTM-SDR Structure, a special form of input data is required to be processed, which is called Sparse Distributed Representation SDR (a large array of 0's and 1's bits). The on bits "1" indicate the active neurons, whereas off bits "0" correspond to inactive neurons [1]. Encoder takes the human understandable data as Input and converts it into Machine readable format. Spatial Pooler then processes this input further, actives some columns as per its algorithm and produces Sparse binary vectors. Spatial pooler's output then acts as input for Temporal memory, where the active columns from Spatial pooler activates their cells. Temporal memory observes and learns the SDRs pattern from Spatial pooler and predicts the next value for the relative sequences on basis of its previous knowledge [2]. SDR Classifier then receives vector of active cells from Temporal Memory as input and performs further cognitive functions like prediction/inference and learning.

The main objective of this paper is to implement HTM-SDR Classifier using Numenta's documented and verified approach. The rest of the paper is structured as follows; Section 2 covers the methodology and implementation approach of SDR classifier. Section 3 explains the Algorithm used for Implementation of SDR Classifier. Results and conclusion are presented in section 4 and 5 respectively.

II. METHODOLOGY

The main inspiration behind HTM-SDR Classifier is to let the machine perform advanced functionalities and cognitive tasks same as human brain. This time-based prediction framework is designed to perform future data prediction based on learning and memorizing the data previously fed into it. SDR Classifier is an essential element in HTM framework as it is responsible to detect and learn the relationship between the Temporal Memory's present state at time t and the future value at t+n, where n indicates no. of stages in future to be inferred.

This section further splits into three subsections, in which Input requirements for SDR Classifier, Prediction/Inference function and Learning function of SDR classifier will be discussed.

Case 1. *Input Requirement for SDR Classifier*

Fig. 1. elaborates HTM-SDR network and the type of inputs required for SDR classifier is to perform cognitive functions like inference/prediction and learning.

SDR Classifier receives three different inputs at every time instance. Temporal memory provides first input to SDR Classifier, which is the vector of uniquely activated cells for predicted value. After receiving input from Temporal memory, SDR classifier expands its weight matrix according to activated cells, by appending new columns (Input units).

At the same instant, Encoder provides the Bucket Index and record number of the current input [3]. Encoder's main task is not only to convert the given input value to Sparse Distributed Representation "SDR" format, but encoder is also responsible to store the given input value into Buckets and assign Bucket index & Record number to it. The storage capacity of each bucket is only one value at a time. SDR classifier obtains these two inputs from encoder to perform Bucket prediction at a particular time step. To evaluate Bucket index of the given input, Encoder uses the following Equation (1) [4].

$$\boldsymbol{Bucket\ Index}\ (\boldsymbol{I}) = \boldsymbol{floor}[\boldsymbol{No.of\ Buckets} * \frac{(\boldsymbol{given\ value} - \min \boldsymbol{value})}{\boldsymbol{Range}}] \quad (1)$$

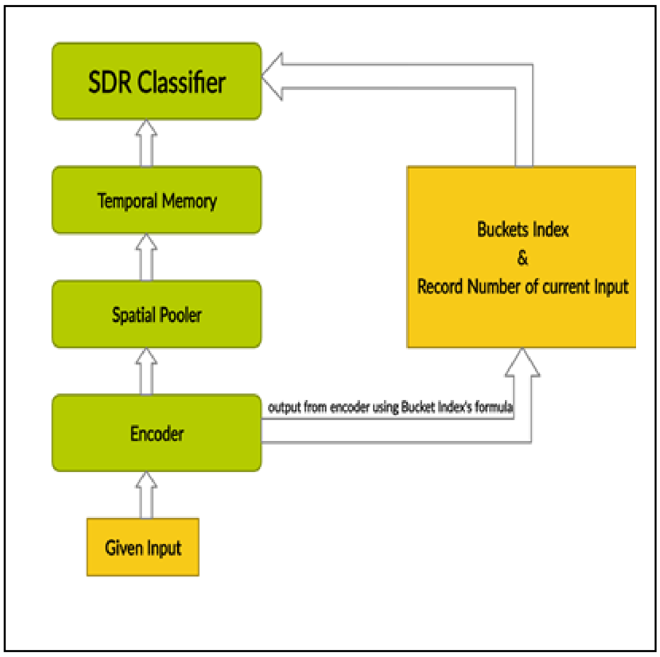

SDR Classifier
Temporal Memory
Spatial Pooler
Encoder
Given Input
Buckets Index & Record Number of current Input
output from encoder using Bucket Index's formula

Fig. 1. HTM-SDR Hierarchy

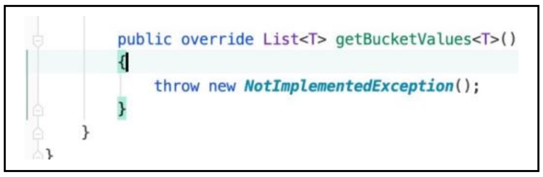
While implementing SDR Classifier in C#, we came across one limitation regarding the inputs from Encoder, which is discussed in the following “Open issue”.

I. Open Issue:

To make the SDR classifier work accurately in real time, the support of lower layers of HTM framework is required, more specifically complete implementation of Encoder is required.

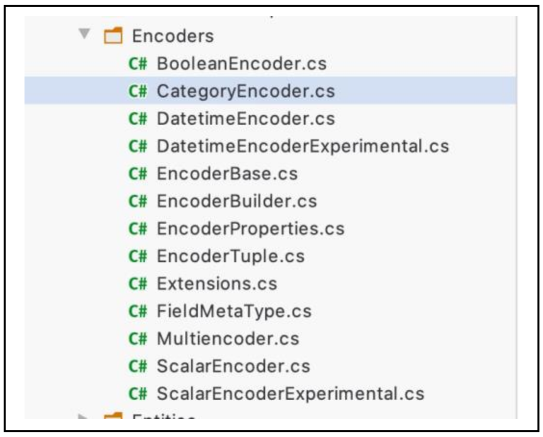
As discussed earlier, Encoder provides two actual inputs (Bucket Index & Bucket Value) to SDR classifier for Bucket prediction. Therefore, a full-fledged operating Encoder is needed in order to fulfil SDR Classifier’s input requirement.

However, Fig 2 shows that the method for retrieving the bucket values is unimplemented by Category Encoder Fig 3 because of which, SDR classifier is unable to get them. So, without having the bucket value prediction cannot happen since the role of classifier is to predict the bucket index not the bucket value. Therefore, to make the full model work, the method for storing bucket value in encoder must be implemented.



```
public override List<T> getBucketValues<T>()
{
    throw new NotImplementedException();
}
    }
}
```

Fig. 2. Unimplemented Method of Category Encoder



```
▼ Encoders
    C# BooleanEncoder.cs
    C# CategoryEncoder.cs
    C# DatetimeEncoder.cs
    C# DatetimeEncoderExperimental.cs
    C# EncoderBase.cs
    C# EncoderBuilder.cs
    C# EncoderProperties.cs
    C# EncoderTuple.cs
    C# Extensions.cs
    C# FieldMetaType.cs
    C# Multiencoder.cs
    C# ScalarEncoder.cs
    C# ScalarEncoderExperimental.cs
```

Fig. 3. List of Encoders

Case 2. SDR Classifier’s Prediction Function

A feed forward neural network is applied by SDR classifier, in which vector of active cells from Temporal memory (bits in activation pattern) serves as input. In other words, number of active bits corresponds to number of input units. To evaluate the activation levels of Output units through weight matrix, Weighted sum equation is used [3]

$$\boldsymbol{a_j} = \sum_{i=1}^{N} \boldsymbol{W_{ji}}\, \boldsymbol{x_i} \quad (2)$$

Where,
$\boldsymbol{a_j}$ is activation level of j^{th} Output unit
N is the number of Input unit columns (N=3 as per below stated Weight matrix)
$\boldsymbol{W_{ji}}$ refers to weighted values used by j^{th} Output (j=row in Weight matrix) for i^{th} Input unit (i=column in Weight matrix)
$\boldsymbol{x_i}$ is the activation state of i^{th} Input unit (it can be either on ‘1’ or off ‘0’)

According to Equation (2), ***Weighting*** and ***Summing*** are the two processes involved in computation of Output’s activation levels from weight matrix. In ***Weighting*** process, weighted values of each input unit are scaled by their activation state, which can be either 0 or 1. In ***Summing*** process, these values are updated in rows of Output unit of weight matrix and then added together to determine activation level of each output unit. To elaborate this computation, the following general weight matrix is under consideration.

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

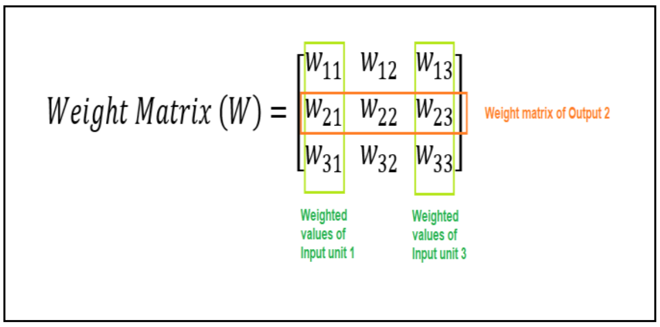

Weight Matrix (W) =
W11 W12 W13
W21 W22 W23
W31 W32 W33
Weight matrix of Output 2
Weighted values of Input unit 1
Weighted values of Input unit 3

Fig. 4. Weight matrix for activation levels computation

Fig. 4. shows that the columns of the weight matrix are Input units and rows of weight matrix represents Output units.

Since Input unit 1 and 3 are active according to Fig 4, therefore the activation state x_i will be 1 for them. Activation state for Input unit 2 will be 0. The activation level of 2^{nd} Output unit can be computed using Weighted sum Equation (2) as follows,

$$a_2 = \sum_{i=1}^{3} W_{2i}\, x_i$$

$$a_2 = (w_{21} * 1) + (w_{22} * 0) + (w_{23} * 1)$$

Next step is to apply ***Softmax Algorithm*** to each activation level, which will calculate probability distribution for Bucket prediction of upcoming or future value. ***Softmax function*** can be evaluated using Equation (3) as follows [3]

$$y_k = \frac{e^{ak}}{\sum_{i=1}^{k} e^{a_i}} \qquad (3)$$

Where,
$\boldsymbol{y_k}$ is the probability $[0 \leq y_k \leq 1]$ to check k^{th} Bucket in order to predict next coming value.
$\boldsymbol{a_k}$ is the activation level of k^{th} output unit.
$\boldsymbol{k}$ represents number of buckets utilized by encoder (number of Output rows in Weight matrix)
$\boldsymbol{a_i}$ represents activation level of each output unit, which was evaluated above using Weighted sum equation.

Softmax function will be applied to each bucket in order to find the probability distribution. Bucket with the highest probability value y_k has the utmost chance of containing the upcoming or future value in it.

Case 3. SDR Classifier's Learning Function

Another capability of SDR classifier is that it learns and updates its Weight matrix, whenever its bucket prediction turns incorrect. This learning feature makes SDR classifier efficient enough to provide 100% accurate predictions in future. At the beginning, SDR classifier initializes a Weight matrix with weight value "0". At every iteration, it learns and revises its Weight matrix.

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

$$\boldsymbol{y = (y_1, y_2, y_3, \dots \dots, y_k)}$$

At each iteration, ***target distribution*** values will be provided. This value would be 1 for that particular bucket which was used by encoder to encode the input at that time step, and it would be 0 for the other buckets. SDR classifier obtains Target distribution by seeing the bucket index input coming from encoder. Target distribution can be symbolized by

$$\boldsymbol{z = (z_1, z_2, z_3, \dots \dots, z_k)}$$

The ***probability distribution elements*** y are being compared with ***target distribution elements*** z in order to check whether the predicted bucket matches the actual bucket? This comparison will be accomplished by computing errors for each element of probability distribution y, utilizing the following formula

$$\boldsymbol{Error\ (E_j) = z_j - y_j}$$

Where,
$\boldsymbol{j}$ represents the rows of Weight matrix (output unit), see Fig.4.

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'.

$$\boldsymbol{Update_j = \alpha(E_j)}$$

Alpha is decided prior to the construction of SDR for fast adaptability to learning. It is essentially larger but closer to zero. The product of alpha and error, called updated value, is adjusted for the active columns of the weight matrix for the input in hand.

$$\boldsymbol{W'_{ij} = W_{ij} + \alpha(z_j - y_j)}$$

$$\boldsymbol{W'_{ij} = W_{ij} + Update_j} \qquad (4)$$

Equation (4) shows how SDR classifier updates its Weight matrix in order to learn through its incorrect predictions.

III. Algorithm

We established that the SDR Classifier functions by receiving inputs from the encoder and the temporal memory, in the code it is specified accordingly:

a. Inputs by the encoder at instant t: This is implemented by a list of object `classification` with two values. `classification[0]` includes the bucket index and `classification[1]` includes the bucket value at that index.

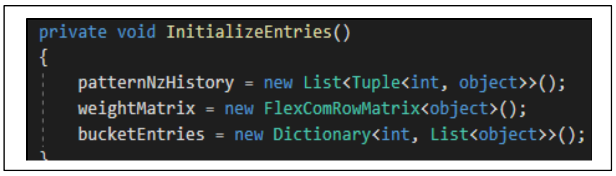
b. Record number of the current iteration: This is implemented using integer variable `recordNum`.

c. Activated bit pattern of temporal memory for the input at instant t+n: This is implemented by initiating a data structure `patternNz`(1d array of integer) which stores activated cells of temporal memory.

The running code is divided into four parts:

I. Initialization

Three new elements are initialized by using the method `InitializeEntries`. Elements include `weightMatrix` which is a list of lists of objects representing row and column of our matrix, dictionary of `bucketEntries` that stores bucket value for each input and `patternNzHistory` which stores the history of patterns processed by our classifier. The dimensions are established further in the code according to the input received.



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

Fig. 5. Initializing Bucket Entries

For example: Considering `classification[0]= 5` will increase the probability of the 5th bucket in the SDR classifier `and patternNZ[]={1,6}` represents activated cells in the temporal memory.

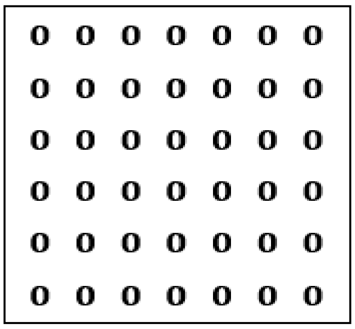


Fig. 6. Starting Empty Weight Matrix

II. Inference

- The activated pattern array that we received as an input provides the columns of the weight matrix to be dealt with, in this case 1st and 6th, refer Fig. 7. According to these known columns, for every row/ bucket/output unit the weights in the activated columns are added and stored in `outputActivationSum`. It is a double type array with size equal to the maximum bucket/ row. This is performed by the method `inferSingleStep(int[] patternNz)`.

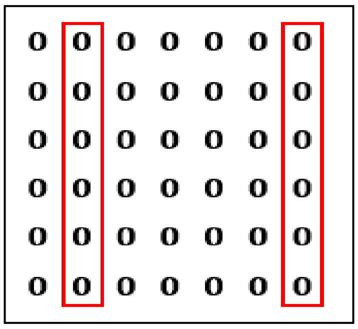
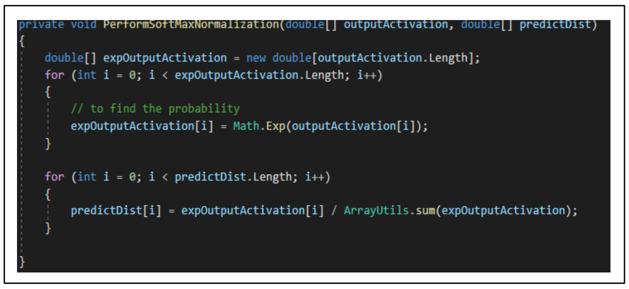


Fig. 7. Activated Input units 1 & 5

- The method then calls another method `PerformSoftMaxNormalization(double[]outputActivationSum,double[]predictDist)`, where `predictDist` is an empty 1-D array having same size as `outputActivationSum`. The probability is then calculated of each bucket/row using SoftMax theorem. The `predictDist` array is filled with the probabilities of each bucket and is returned by the method `inferSingleStep`.



```
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. 8. 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

III. Error Computation

- For all buckets, using the probabilities an error is computed as an offset from the desired probabilities (which is 1 for 5th bucket and 0 for all other buckets in our case since we received 5th bucket as an input from the encoder) to the computed probabilities. This is performed by the method `CalculateError` which receives `List<Object> classification` as an input containing two values, bucket index of the input and its bucket value. The method returns double type array of error which contains errors to be compensated in the activated columns for each bucket/ row.

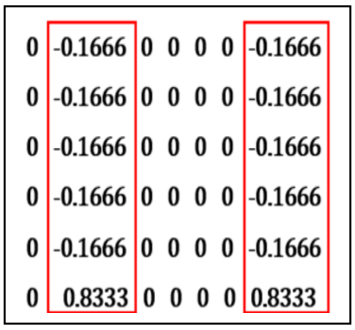
- **Error Computed: E (-0.1666, -0.1666, -0.1666, -0.1666, -0.1666, 0.8333)**

IV. Updating the Weight Matrix

Before updating, alpha value is chosen initializing the SDRClassifier object. Alpha is important to adapt the weight matrix during learning. This value is multiplied to all error entries and is then updated for the activated columns of the weight matrix as a form of learning.

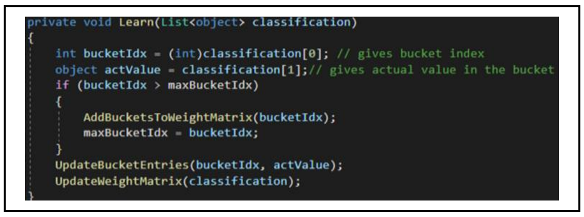
This is performed in following steps in the code:

- Method `Compute(int recordNum, List<object> classification, int[] patternNz)` is called and the current `patternNZ` is stored with its serial number in `patternNzHistory` list.
- To ensure the weight matrix is large enough to store all activated pattern values received as an input, the largest index in patternNZ is stored as 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 method `Learn(List<Object> classification)` is then called and in this method it is ensured that weight matrix is large enough to accommodate the received bucket/output unit for the input.
- To do that, the method AddBucketsToWeightMatrix(int bucketIdx) is called to grow the rows of the weight matrix as much as the maximum bucket index, if maximum bucket index is greater than the current bucketIdx.
- `Learn` method finally calls `UpdateWeightMatrix(List<Object> classification)`. For each pattern listed in `patternNzHistory` it updates the weight matrix with the computed error for each pattern. It is to ensure the weight matrix learns for all possible patterns received so 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. 9. 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);
}
```

Fig. 10. Learn Method

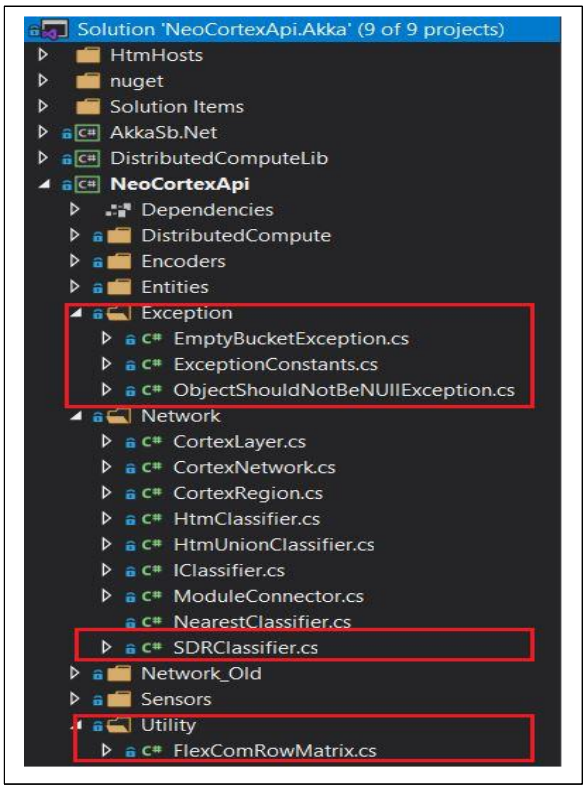
TABLE II. PREDICTED PROBABILITY DISTRIBUTION

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

Table II shows serial probabilities of each bucket, after performing SoftMax on the updated weight matrix. The 5th bucket has the highest probability as the SDR Classifier learnt the target inputs high occurrence in this iteration.

IV. C# CODE

Under the solution "NeoCortexapi.Akka" in HTM folder, there is a Network folder which contains the main implemented class "SDRClassifier.cs". Along with that, some other classes in folders "Exception" and "Utility" are also implemented, as highlighted in Fig 11.

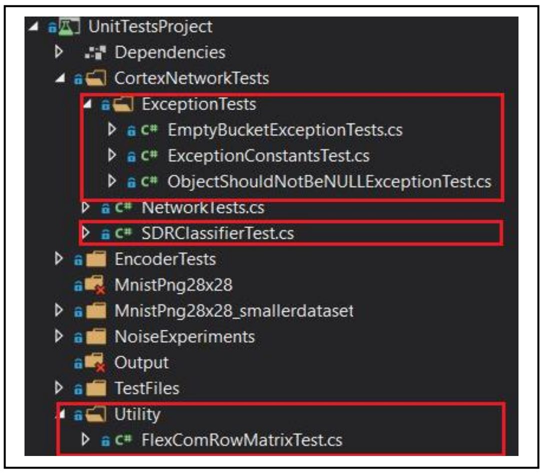


```
Solution 'NeoCortexApi.Akka' (9 of 9 projects)
  HtmHosts
  nuget
  Solution Items
  AkkaSb.Net
  DistributedComputeLib
  NeoCortexApi
    Dependencies
    DistributedCompute
    Encoders
    Entities
    Exception
      EmptyBucketException.cs
      ExceptionConstants.cs
      ObjectShouldNotBeNUllException.cs
    Network
      CortexLayer.cs
      CortexNetwork.cs
      CortexRegion.cs
      HtmClassifier.cs
      HtmUnionClassifier.cs
      IClassifier.cs
      ModuleConnector.cs
      NearestClassifier.cs
      SDRClassifier.cs
    Network_Old
    Sensors
    Utility
      FlexComRowMatrix.cs
```

Fig. 11. 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 12.



```
UnitTestsProject
  Dependencies
  CortexNetworkTests
    ExceptionTests
      EmptyBucketExceptionTests.cs
      ExceptionConstantsTest.cs
      ObjectShouldNotBeNULLExceptionTest.cs
    NetworkTests.cs
    SDRClassifierTest.cs
  EncoderTests
  MnistPng28x28
  MnistPng28x28_smallerdataset
  NoiseExperiments
  Output
  TestFiles
  Utility
    FlexComRowMatrixTest.cs
```

Fig. 12. Implemented Unit tests

Firstly, there is “ExceptionTests” folder, which consists of three classes. The purpose of these exception tests is to specify the reason behind disruptions encountered by the code during execution.

The class “SDRClassifierTest.cs” contains different test cases, which are implemented to test the functionality of different parameters and methods existing in the main code.

In the Utility folder, there is another class named “FlexComRowMatrix.cs”, which is implemented to check the functionality of SDR Classifier’s Weight matrix.

V. Results

The most important element of an SDR classifier is the weight matrix. It is essentially updated as the classifier learns on every predicted value from temporal memory. To give a better overview of our results we have chosen the weight matrix to dictate the performance of the implemented SDR classifier.

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:

  A key thing to be noted is the number of iterations to be performed for each input. These can be decided prior to the SDR construction. More the iterations, better the learning. At each preceding iteration the SDR computes its process for the pattern that came prior to the current pattern as well as for keen learning. It is observed that wth each iteration the results get refined.

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 the instant but as established, it also adjusts the coming error for the pattern prior to it, continuously learning for all received inputs, illustrated 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.

$$\begin{array}{ccccc}
[0 & 1 & 2 & 3 & 4] \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{array}$$

Fig. 13. Initial Weight matrix

$$\begin{array}{ccccc}
[0 & 1 & 2 & 3 & 4] \\
0 & -0.2 & 0 & 0 & -0.2 \\
0 & -0.2 & 0 & 0 & -0.2 \\
0 & -0.2 & 0 & 0 & -0.2 \\
0 & -0.2 & 0 & 0 & -0.2 \\
0 & 0.8 & 0 & 0 & 0.8
\end{array}$$

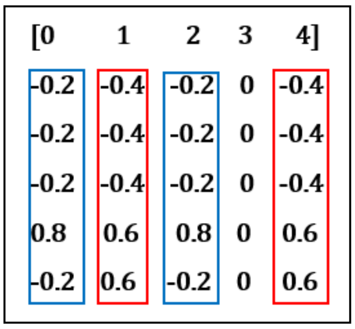
Fig. 14. Updated matrix

- **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. 15. Updated Weight Matrix on pattern [1, 4] & pattern [0, 2]

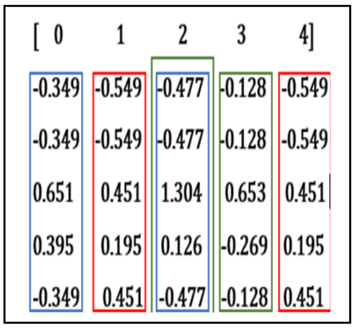
- **Input 3:** Iteration 2nd, Input bucket index 2 and activated pattern {2, 3}

  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. 16. 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

SDR Classifier has proved to be better in performance than a CLA classifier. In this paper, we aimed to implement the SDR classifier based on the tested approach of Numenta. We validate and benchmark that SDR, unlike CLA, does not give outliers in prediction probability due to reinforcing the imperative weight matrix at each turn. In addition to that, incorrect predictions are ensured to be penalized by successive inputs using as much iterations as the system's capacity at every preceding pattern. The results attest that multiple iterations enhance continuous learning and warrants precise outcomes.

REFERENCES

[1] A. James, "Deep Learning Classifiers with Memristive Networks", vol. 14, pp. 174-179, 2019, DOI: 10.1007/978-3-030-14524-8

[2] S. Ahmad, M. Lewis, "Temporal memory algorithm", Technical Report Version 0.5, 2017, Numenta Inc.

[3] Dillon A, "SDR Classifier", Sept 2016, https://hopding.com/sdr-classifier#title

[4] Purdy S, "BaMI-Encoders", Technical Report Version 0.4, Numenta Inc.