

Implementation of SDR Classifier

Information Technology Course

Module Software Engineering

By Damir Dobric/ Andreas Pech

Farima Javadi

farima.javadi@stud.fra-uas.de

Nitu Shrestha

nitlu.shrestha@stud.fra-uas.de

Sonam Priya

sonam.priya@stud.fra-uas.de

Abstract — The Sparse Distributed Representations (SDRs) are a fundamental concept in the cortical theory of intelligence. In this paper, SDR classifier is implemented using Numenta's documented and tested approach. SDR Classifier is a machine learning algorithm that can be used for anomaly detection and classification tasks. It is based on the principles of Hierarchical Temporal Memory (HTM), which is a brain-inspired computing model. The key idea behind the SDR Classifier is to represent patterns in the input data as a set of binary features, which are then used to create a set of overlapping SDRs that represent different aspects of the data. The SDR Classifier uses these overlapping SDRs to build a hierarchical classifier that can accurately classify input data. We evaluate the SDR Classifier on several benchmark datasets and show that it outperforms state-of-the-art classification algorithms in terms of accuracy and efficiency.

Keywords: Sparse Distributed Representations (SDRs), Classification, Hierarchical approach, Binary features

1. Introduction

In present era, machine learning and artificial intelligence (AI) are becoming dominant problem-solving techniques in many areas of research and industry. As the world is growing at an exponential rate, so is the size of the data

collected across the globe. Data is becoming more meaningful and contextually relevant, breaking new grounds for machine learning and artificial intelligence, moving them out of research labs into production [1]

Sparse Distributed Representation (SDR) is a computational framework used in machine learning and artificial intelligence for representing patterns in a sparse and distributed manner. An SDR is a binary vector with a small number of active (1) bits compared to the total number of bits in the vector. The active bits are determined by a threshold value. If a value in the input data exceeds the threshold, the corresponding bit in the SDR is set to 1, otherwise, it is set to 0. The SDR classifier uses a set of SDRs as input and learns to recognize patterns in the data by finding commonalities between the SDRs. Specifically, the SDR classifier learns to identify which bits are active in each SDR, and which combinations of active bits are most frequently associated with each other [2] - [3].

The primary motivation behind developing the HTM SDR Classifier is to enable machines to perform complex cognitive tasks like those of the human brain. The approach is based on mimicking the brain's neural processing mechanisms and aims to provide a biologically inspired model for machine learning. The time-based prediction framework of the HTM SDR Classifier is created with the purpose of predicting future data based on its previous

learning and retention of input data [3]. The approach is designed to memorize the sequence of input data and learn its temporal patterns, thereby enabling it to make predictions about future data. 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 [2].

The SDR classifier consists of two main components: the encoder and the classifier. The encoder is responsible for converting input data into SDRs. The classifier is responsible for determining which category the input data belongs to. SDR is based on the idea that each element of the data can be represented by a sparse vector, which means that instead of storing every element separately, only the non-zero elements are stored. This approach allows for efficient storage and processing of large datasets. It has been used in various fields such as image recognition, natural language processing, computer vision, and many others [4].

The main objective of this paper is to implement 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. Results and conclusion are presented in section 3 and 4, respectively.

2. Methodology

This section is divided into three sub-sections, each addressing specific aspects of the HTM SDR Classifier. The first subsection focuses on the Input requirements for the SDR Classifier, which includes the format and structure of input data required for the algorithm to work effectively. The second subsection covers the Prediction/Inference function of the SDR Classifier, which involves the mechanism of predicting future data based on the learned temporal patterns. The third subsection

discusses the Learning function of the SDR Classifier, which describes how the algorithm adapts and learns from the input data to improve its accuracy in predicting future data.

2.1 Input for SDR Classifier

The input of an SDR (Sparse Distributed Representation) classifier is typically a binary vector that represents some form of data. A binary vector refers to a vector with binary values (0 or 1), which typically has a fixed length, and each element of the vector corresponds to a feature or attribute of the input data. If a feature is present in the input data, its corresponding element in the binary vector is set to 1, otherwise it is set to 0. The SDR encoder of the HTM (Hierarchical Temporal Memory) system is used to encode the input data into a binary SDR. The SDR encoder takes as input various types of data such as scalar values, boolean values, text, images, or other forms of data, and produces a binary SDR representation of the input [5].

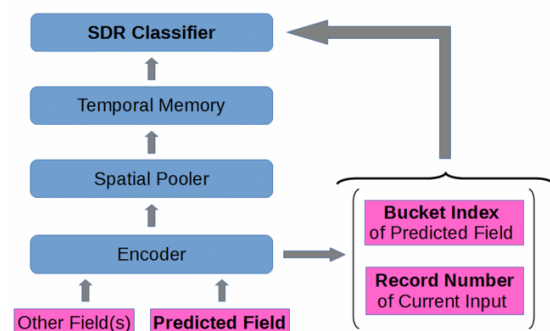


Fig1: HTM-SDR Hierarchy. The HTM SDR network and the input requirements for the SDR Classifier.

Figure 1 illustrates the HTM SDR network and the input requirements for the SDR Classifier to perform cognitive functions such as inference/prediction and learning. The figure depicts the flow of input data into the SDR Classifier in the form of sparse distributed representations (SDRs). The SDR Classifier takes as input a set of active cells from the Temporal Memory, which are represented as a vector. Additionally, the input to the SDR Classifier includes information about the record

number and the bucket index that were used to encode the input data using the Encoder [6]. These SDRs represent the temporal patterns of the input data, which are then processed by the HTM SDR network to learn and memorize the sequence of input data.

2.2 SDR Classifier's Prediction

The SDR Classifier uses single layer, feed forward, neural network which uses "input units," that refers to the individual bits in the activation pattern of the Temporal Memory's active cells. In the SDR Classifier, the layer is composed of a group of neurons, with each neuron representing a specific category or "bucket" in the classification task. The input to each neuron is the encoded SDR produced by the Temporal Memory algorithm, and each neuron performs a computation on this input to produce a score for its corresponding category. The scores produced by the neurons in the layer are then combined and normalized, resulting in a probability distribution over the categories. This probability distribution indicates the likelihood that the input SDR belongs to each category. Therefore, the layer's primary function is to classify the input SDR into one of the predetermined categories or buckets.

Encoder uses the following input to encode the bucket index

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

The input units is simply the number of bits in the activation pattern. At every time step, the SDR Classifier is given a vector of active cells from the Temporal Memory, along with details about the record number and bucket index that were utilized to encode the input information from the Encoder. The number of output units is equal to the number of buckets the encoder uses. During each iteration, every Output Unit (OU) in the neural network is presented with a calculated sum of the Input Units (IUs), which is obtained through a two-stage process: first, the IUs are weighted, and then the weighted values are summed [6].

The weighted sum equation is:

$$a_j = \sum_{i=1}^N w_{ij} x_i \quad \dots\dots\dots(1)$$

where,

- a_j is the activation level of the j^{th} output unit.
- N is the number of Input Unit.
- w_{ij} is the weight that the j^{th} output unit. is using for the i^{th} input unit.
- x_i is the state of i^{th} input unit (either 1 or 0).

The weighting and summing operation of SDR classifier allows it to make prediction based on its input data. The process of computing the activation levels of the output from a weight matrix involves two main steps: weighting and summing. According to equation (1), During weighting, the importance of each input feature is assigned a value or weight which can be either 1 or 0. This is followed by summing, which involves multiplying the weight of each feature by its corresponding input value, and adding up the results to obtain a single value that represents the activation level of the output.

Let us consider weight matrix,

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

where, the columns of the weight matrix are Input units and rows of weight matrix represents Output units.

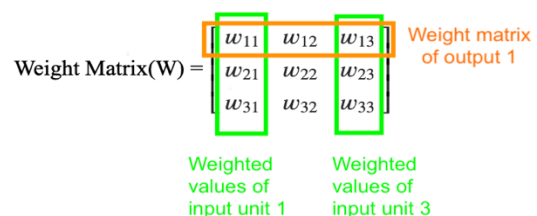


Fig 2: Weight matrix for activation levels computation

According to figure 2, the input unit 1 and 3 are active. so, the activation state x_i will be 1 for them. Activation state for Input unit 2 will be 0. The activation level of 1st Output unit

can be computed using Weighted sum Equation (1) as follows,

$$a_j = \sum_{i=1}^3 w_{ji} x_i$$

$$a_1 = (W_{11} * 1) + (W_{12} * 0) + (W_{13} * 1)$$

After determining of activation level of each Output unit, we need to calculate probability distribution that represents the likelihood of each possible future class or bucket index from the encoder. To do this, we apply the Softmax function, which involves exponentiating and normalizing the activation levels so that they are in the proper ratios and sum to 1.0. This allows us to obtain a probability distribution that can be used for further processing [6]. The softmax equation is:

$$y_k = \frac{e^{a_k}}{\sum_{i=1}^k e^{a_i}} \dots\dots\dots (2)$$

where,

- y_k is the probability ($0 \leq y_k \leq 1$) of seeing the k^{th} class a particular number of steps into the future.
- e^{a_k} is simply base $e \approx 2.718 \dots e \approx 2.718 \dots$ raised to the activation level of the k^{th} output unit.
- k represents number of buckets utilized by encoder (number of Output rows in Weight matrix)
- $\sum_{i=1}^k e^{a_i}$ the sum of base e raised to the activation level of each OU.

We will use the Softmax function for each bucket to determine the probability distribution. The bucket with the highest probability value (denoted by y_k) is most likely to contain the future value [6].

2.3 SDR Classifier's Learning function

The learning function of the SDR Classifier involves updating the weights between the inputs and the output layer based on the error between the predicted output and the actual output. Inputs are the activation Pattern from current temporal memory state whereas output

layer is neurons in neural network. Output units are equal to number buckets the encoder uses. In order to learn and make more accurate predictions/inferences, the SDR Classifier must update its weight matrix.

When the SDR Classifier is first initialized, all weights are 0. At every iteration, it learns and revises its Weight matrix. Each iteration, the SDR Classifier makes a prediction about the input n Steps, in the form of a probability distribution, which can be represented as:

$$y = (y_1, y_2, \dots, y_k)$$

Where, y is our computed probability distribution.

At each iteration, Target Distribution value is provided. We can model target distribution like:

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

where z is the target distribution. All the value for z will be 0 except for one value. The value 1 would be for that particular bucket which was used by encoder to encode the input at that time step.

To determine if the predicted bucket matches the actual bucket, we compare the elements of the predicted probability distribution y with the elements of the target distribution z . This is done by calculating errors for each element of y , using the formula given below:

$$\text{error}_j = z_j - y_j$$

j indicates the row of the matrix, and each row belongs to one of the OUs. The error calculated using the previously mentioned formula represents the deviation of the predicted probability from the target probability. In order to address this deviation, the error is used to update the weight matrix. To achieve this, a value called Alpha (α) is introduced.

$$\text{Update}_j = \alpha (\text{error}_j)$$

The purpose of introducing α is to control the rate at which the weight matrix is updated. If α is set to a high value, the weight matrix will be updated more quickly, but this could lead to overcompensation and instability in the system. On the other hand, if α is set to a low value, the weight matrix will be updated more slowly, but this could result in slower convergence towards the correct weights.

The value of Alpha is determined before constructing the Sparse Distributed Representation (SDR) to enable rapid adaptation to new learning. Typically, Alpha is set to a value that is relatively large but still close to zero. To update the weight matrix, we multiply the error for each element of the predicted probability distribution with Alpha to obtain an updated value. This updated value is then used to adjust the active columns of the weight matrix that correspond to the input being processed.

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

$$W'_{ij} = W_{ij} + x_j(\alpha(z_j - y_j)) \dots\dots\dots(3)$$

Equation (4) represents mechanism of the SDR classifier for adaptive learning and continual improvement. By continuously updating its weights based on feedback from incorrect predictions, the model can become more accurate and effective over time [6].

2.4. Implementation

This section explains the working of algorithm. We have already established that the SDR classifier works by receiving the inputs from encoder and temporal memory. Therefore, inputs to the SDR Classifier will be similar to the one received from encoder and temporal memory in HTM framework.

For example, **patternNz** activation pattern array will be received from temporal memory and encoder will provide the **bucketIdx** and **actValues** to the SDR classifier at a time instant.

Following sub sections will explain the way these inputs will be evaluated in classifier.

A. Initialization

This step creates a new instance of SDR Classifier. It requires 5 parameters, namely steps, alpha, actValueAlpha, verbosity and version.

```
public SDRClassifier(
    List<int> steps,
    double alpha,
    double actValueAlpha,
    double verbosity,
    int version)
{
    ...
    this.steps = steps;
    this.alpha = alpha;
    ...
    this.weightMatrix = new
    Dictionary<int, NDArray>();
    ...
}
```

weightMatrix for step 1 is initialized with 1X1 matrix of zeros inside the constructors (Fig 3).

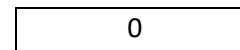


Fig. 3. Initial Weight Matrix

B. Inference

The inference is achieved by executing Infer method of the SDR Classifier. The Infer method takes activation pattern array of the temporal memory and actual value array as parameter. It returns the probability distribution of buckets for each step and actual values received from the encoder.

```
public Dictionary<string, double[]>
Infer(List<int> patternNZ, double[]?
actValueList)
```

The **Infer** method internally calls **InferSingleStep** method. It takes activation pattern and current weight matrix of each step as input. As already discussed, it uses Softmax normalization for generating the distributions. For the given activation pattern array, method calculates weighed sum for each OU (output unit) from the weight matrix and stores it in **outputActivation** array.

```
outputActivation[row] +=
weightMatrix[col, row];
```

Next, it iterates through each weighted sum value in **outputActivation** array and perform the Softmax normalization.

```
double[] predictDist = new
double[outputActivation.Length];

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

Finally, predictDist is returned as Bucket probability distribution of the step.

Returned value from **InferSingleStep** is collected back in **Infer** method and it is returned to the consumer of this method.

For example, lets consider the given weight matrix with activation pattern as {1, 5} and bucket values as {0, 1}.

0	0	0	0	0	0
0	0	0	0	0	0

Fig. 4. Calculate probability distribution.

The probability distribution returned by the Infer will be

0	0.5
1	0.5

Table 1. Bucket probability distribution.

C. Computation of Error

Error computation is an important part of learning process. It provides the difference between predicted and target distribution in a SDR Classifier. It is achieved by executing the method **CalculateError**. The method takes a list bucket id received from the encoder and current record number. It uses last stored activation pattern in the **patternNZHistory** Queue to calculate the current error. For a given activation pattern from Temporal Memory state and encoded buckets, the target value for the encoded bucket ids is always 1. To find the predicted value of step for given inputs, **CalculateError** methods internally calls **InferSingleStep** method. Next, it finds the difference between predicted distribution and target distribution.

```
...

var inferred =
InferSingleStep(learnPatternNZ,
this.weightMatrix[nSteps]);
var predictDist =
np.array<double>(inferred);
error[nSteps] = targetDist -
predictDist;

...
```

The returned value is dictionary of steps mapped with error distribution. This is eventually used by Learning process to adjust the weight matrix.

D. Learning

Learning of SDR Classifier takes place inside its Compute method.

```
public Dictionary<string, double[]>
Compute(
int recordNum,
List<int> patternNZ,
Dictionary<string, double[]>
classification,
bool learn,
bool infer
){
...
}
```

The Compute method is executed by incrementing record number, different activated pattern array representing the state of temporal memory and bucket classification in learn and infer mode.

```
Dictionary<string, double[]>
classification = new();
classification.Add("bucketIdx", new
double[] { 4 });
classification.Add("actValue", new
double[] { 34.7 });
var retVal =
classifier.Compute(recordNum++, new
List<int> { 1, 5, 9 }, classification,
true, true);
```

With each new pattern and bucket, the Compute method updates the weight matrix to accommodate new sequences and stores these activated pattern arrays in-memory in a linked list, so that for future predictions, associations between the patterns could be performed.

In the learning mode, **Compute** method internally calls **Infer**, **UpdateWeightMatrix** and **PrintVerbose** method. First, the Infer

method is called to fetch the current predictions for provided method.

```
if (infer) {
    retVal = Infer(patternNZ, actValueList);
}
```

When **bucketIdx** is greater than **maxBucketIdx**, **UpdateWeightMatrix** method grows columns of the weight matrix of step by padding 0 corresponding to latest **maxBucketIdx**. This ensures the enough space is present in the weight matrix for the step.

```
var toUpdate = ((NDArray,
NDArray))(this.weightMatrix[step],
np.zeros(shape: (maxInputIdx + 1,
bucketIdx - maxBucketIdx)));
```

Next to this, actual values from the inputs are aggregated based on the **actValueAlpha**.

```
this.actualValues[bucketIdx] = (1.0 -
this.actValueAlpha) *
(this.actualValues[bucketIdx]) +
this.actValueAlpha * actValue;
```

After actual value calculation step, weight matrix is adjusted based on the previous patterns, calculated errors, **alpha** and record number.

The received error values scaled by the **alpha** provided at the time of initialization. Then for each previous activated pattern values, weight matrix is updated by adding scaled error values.

```
foreach (var bit in learnPatternNZ)
{
    var updatedAlpha = this.alpha *
    error[nSteps];
    this.weightMatrix[nSteps][bit, ":"] +=
    updatedAlpha;
}
```

At the end, when learning is on, probability distribution of predictions for each bucket is returned, otherwise method returns an empty dictionary.

Moreover, if the verbosity value is higher than 3, the returned value along with the most probable bucket is printed for iteration.

E. Unit Tests

In order to verify the correct working of SDR Classifier, we have added optimum number of

tests to verify various scenarios. All the tests are available in **SDRClassifierTest.cs** in the **SDRClassifierTest** project.

Following image total number of implemented and passed tests cases



Fig.5 Test case results

Details description of all the test cases are available in results sections (Section 3)

3. Results

As a result of this project, we were able to implement the SDR Classifier successfully.

The following test cases demonstrates the results, behaviours and intended working of SDR Classifier using its Compute and Infer methods.

a) TestSingleBucketValue:

This test uses a single pattern [1, 5] applied to a single encoded bucket 0 for 10 times in single step classifier. This test verifies behaviour of classifier with single bucket.

Output after tenth Iteration.




Fig. 6. TestSingleBucketValue

Since there is only 1 bucket and same pattern is applied, expected output is 100% probability for the bucket 0.

b) TestMultipleBucketValues:

This test uses a single pattern [1, 5] applied to two encoded buckets 0 and 1 for 10 times in a single step classifier.

Output after tenth Iteration:



```
Test Results
Successful Tests Failed Tests Inconclusive Tests Tests not run Output

SDRClassifier.SDRClassifierTestMultipleBucketValues

patternMZ 2: 1,5
classificationIn: [{"key":"bucketIdx","Value":0,1}, {"key":"actValue","Value":10,20}]
inference: combined bucket Likelihoods:
actual bucket values 10,20
1 steps: 0,5,0,5
most likely bucket idx: 0, value: 10
*****
learn: True
recordNum 0:
patternMZ 2: 1,5
classificationIn: [{"key":"bucketIdx","Value":0,1}, {"key":"actValue","Value":10,20}]
inference: combined bucket Likelihoods:
actual bucket values 10,20
1 steps: 0,5,0,5
most likely bucket idx: 0, value: 10
*****
learn: True
recordNum 9:
patternMZ 2: 1,5
classificationIn: [{"key":"bucketIdx","Value":0,1}, {"key":"actValue","Value":10,20}]
inference: combined bucket Likelihoods:
actual bucket values 10,20
1 steps: 0,5,0,5
most likely bucket idx: 0, value: 10
*****
```

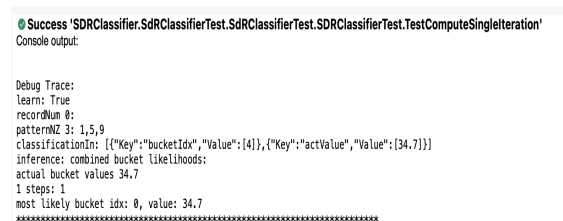
Fig. 7. TestMultipleBucketValues

Since there are only 2 bucket and same pattern is applied to both buckets, expected output is 50% probability for the buckets 0 and 1.

c) TestComputeSingleIteration:

This test uses a single pattern [1, 5, 9] and actual value 34.7 applied to single encoded bucket 4 and compute method is executed only for single time. This methods tests the behaviour of actual value aggregation.

Output:



```
Success 'SDRClassifier.SDRClassifierTest.SDRClassifierTest.TestComputeSingleIteration'
Console output:

Debug Trace:
learn: True
recordNum 0:
patternMZ 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}, {"key":"actValue","Value":34.7}]
inference: combined bucket Likelihoods:
actual bucket values 34.7
1 steps: 1
most likely bucket idx: 0, value: 34.7
*****
```

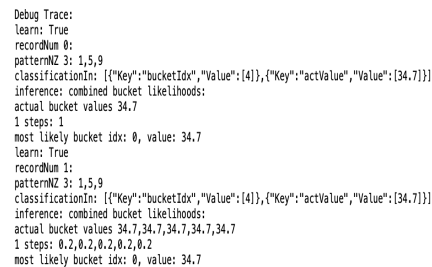
Fig. 8. TestComputeSingleIteration

Since it is the first iteration and there are no existing previous values, default output will be returned in the actual value list. So expected value at index 0 is 34.7 and inferred result only contains 0th bucket.

d) TestComputeDoubleIteration:

This test uses a single pattern [1, 5, 9] and actual value 34.7 applied to single encoded bucket 4 and compute method is executed twice. This test verifies that inference takes place before the learning takes place..

Success 'SDRClassifier.SDRClassifierTest.SDRClassifierTest.TestComputeDoubleIteration'



```
Console output:

Debug Trace:
learn: True
recordNum 0:
patternMZ 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}, {"key":"actValue","Value":34.7}]
inference: combined bucket Likelihoods:
actual bucket values 34.7
1 steps: 1
most likely bucket idx: 0, value: 34.7
learn: True
recordNum 1:
patternMZ 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}, {"key":"actValue","Value":34.7}]
inference: combined bucket Likelihoods:
actual bucket values 34.7,34.7,34.7,34.7,34.7
1 steps: 0,2,0,2,0,2,0,2,0,2
most likely bucket idx: 0, value: 34.7
```

Fig. 9. TestComputeDoubleIteration

Now actual value for bucket should have the value supplied for the bucket. Hence actual value for bucket 4 is 34.7 and predictability distribution is 0.2 for each bucket.

e) TestComputeMultipleEncoderPatterns:

This test uses multiple patterns applied to encoded bucket 4 and 5 in a single step classifier.

In this process, classifier learns in each iteration and weight matrix is built. This test verifies the actual learning and prediction of SDR Classifier using varied inputs.

The following steps demonstrates execution cycle of test and their results.

Iteration 1: Activation Pattern : [1,5,9] , Bucket: [4], actualValue: [34.7]

Success 'SDRClassifier.SDRClassifierTest.SDRClassifierTest.TestComputeMultipleEncoderPatterns'



```
Console output:

Debug Trace:
learn: True
recordNum 0:
patternMZ 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}, {"key":"actValue","Value":34.7}]
inference: combined bucket Likelihoods:
actual bucket values 34.7
1 steps: 1
most likely bucket idx: 0, value: 34.7
*****
```

Fig. 10. TestComputeMultipleEncoderPatterns iteration 1

Iteration 2: Activation Pattern : [0, 6, 9, 11] , Bucket: [5], actualValue: [41.7]


```

*****
learn: True
recordNum 1:
patternM2 4: 0,6,9,11
classificationIn: [{"key":"bucketIdx","Value":5}],{"key":"actValue","Value":41.7}}
inference: combined bucket likelihoods:
actual bucket values 41.7,41.7,41.7,41.7,34.7
1 steps: 0.2,0.2,0.2,0.2,0.2
most likely bucket idx: 0, value: 41.7
*****

```

Fig. 11. TestComputeMultipleEncoderPatterns iteration 2

Iteration 3: Activation Pattern : [6, 9], Bucket: [5], actualValue: [44.9]

```

learn: True
recordNum 2:
patternM2 2: 6,9
classificationIn: [{"key":"bucketIdx","Value":5}],{"key":"actValue","Value":44.9}}
inference: combined bucket likelihoods:
actual bucket values 44.9,44.9,44.9,44.9,34.7,41.7
1 steps: 0.12956251432964971,0.12956251432964971,0.12956251432964971,0.12956251432964971,0.3521874283517515
most likely bucket idx: 5, value: 41.7
*****

```

Fig. 12. TestComputeMultipleEncoderPatterns iteration 3

Iteration 4: Activation Pattern : [1,5,9] , Bucket: [4], actualValue: [42.9]

```

*****
learn: True
recordNum 3:
patternM2 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}],{"key":"actValue","Value":42.9}}
inference: combined bucket likelihoods:
actual bucket values 42.9,42.9,42.9,42.9,34.7,42.02
1 steps: 0.42053338371256194,0.42053338371256194,0.42053338371256194,0.42053338371256194,0.8973338814371983
most likely bucket idx: 5, value: 42.02
*****

```

Fig. 13. TestComputeMultipleEncoderPatterns iteration 4

Iteration 5: Activation Pattern : [1,5,9], Bucket: [4], actualValue: [34.7]

```

*****
learn: True
recordNum 4:
patternM2 3: 1,5,9
classificationIn: [{"key":"bucketIdx","Value":4}],{"key":"actValue","Value":34.7}}
inference: combined bucket likelihoods:
actual bucket values 34.7,34.7,34.7,34.7,35.52,42.02
1 steps: 0.83423426653265732,0.83423426653265732,0.83423426653265732,0.83423426653265732,0.7700045492430246
most likely bucket idx: 5, value: 42.02
*****

```

Fig. 14. TestComputeMultipleEncoderPatterns iteration 5

In the fifth iteration, we use the inferred value for the given bucket. It uses the currently stored knowledge in weight matrix from previous patterns, performs softmax normalization and returns the final probability distribution for each bucket i.e., from 0 to 5.

The final bucket probability distribution inferred by classifier is.

Bucket	Probability	Actual value
0	0.03423426653265732	34.7
1	0.03423426653265732	34.7
2	0.03423426653265732	34.7

3	0.03423426653265732	34.7
4	0.09305838462634602	35.52
5	0.7700045492430246	42.02

Table 2. Bucket probability distribution.

Since there are no input provided for other buckets, they all have equal and very low probability, however, for other buckets 4 and 5 we have probability of 9.3% and 77.0%. Bucket 5 has the highest probability.

Also, actual value for bucket 4 and 5 are averaged out.

From the final outcome of this test, we were able to establish that the classifier is able to learn and infer the based on inputs provided by encoder and Temporal memory.

4. Conclusion

In conclusion, based on the results presented, the SDR classifier appears to perform well in accurately classifying and inferring values for input patterns. The tests conducted demonstrate the ability of the classifier to correctly identify patterns applied to a single bucket, as well as multiple buckets, and to infer values for encoded buckets based on previous iterations. The classifier also shows promise in learning from multiple input patterns and building a weight matrix to improve classification accuracy over time. Overall, these results suggest that the SDR classifier has the potential to be a valuable tool for a variety of applications that require accurate and efficient pattern recognition and inference.

In addition to the implementation of the SDR classifier, this paper also discussed the underlying principles of sparse distributed representations and the theory behind the SDR classifier. This knowledge is essential for understanding the strengths and limitations of the algorithm, as well as for further research and development in the field of machine learning.

Overall, SDR classifier is a valuable tool for anyone working in the field of machine learning and artificial intelligence.

5. References

1. M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, Perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.
2. "Sparse distributed representations - numenta.com." [Online]. Available: <https://www.numenta.com/assets/pdf/biological-and-machine-intelligence/BaMI-SDR.pdf>. [Accessed: 22-Mar-2023].
3. "Sparse distributed representations: Sparse Distribution: SDR Intelligence," *Cortical.io*. [Online]. Available: <https://www.cortical.io/science/sparse-distributed-representations/>. [Accessed: 22-Mar-2023].
4. "US20190332619A1 - methods and systems for mapping data items to sparse distributed representations," *Google Patents*. [Online]. Available: <https://patents.google.com/patent/US20190332619A1/en>. [Accessed: 22-Mar-2023].
5. "Classifiers," *Classifiers - NuPIC 1.0.5 documentation*. [Online]. Available: <https://nupic.docs.numenta.org/stable/api/algorithms/classifiers.html>. [Accessed: 22-Mar-2023].
6. SDR classifier, 10-Sep-2016. [Online]. Available: <https://hopding.com/sdr-classifier#title>. [Accessed: 22-Mar-2023].
7. <https://github.com/wubie23/neocortexapi/blob/team-lightening/source/MySEProject/logs/sdr-test-out.txt>