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

nitu.shrestha@stud.fra-uas.de

Sonam Priya

sonam.priya@stud.fra-uas.de

Wubishet Damtie

youremail@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. Overall, our results demonstrate the effectiveness of the SDR Classifier and highlight its potential for a wide range of applications.**

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 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. Section 3 explains the Algorithm used for Implementation of SDR Classifier. Results and conclusion are presented in section 4 and 5, respectively.

2. Methodology

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].

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. 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].

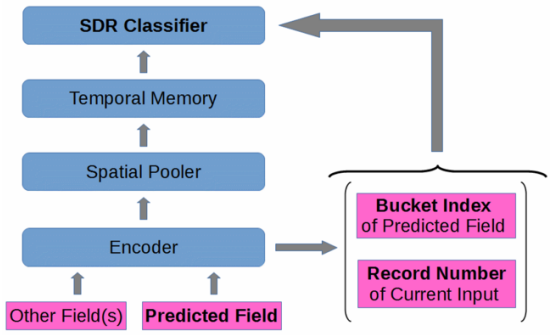

Fig1: HTM-SDR Hierarchy

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. The number of 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:

$$a_j = \sum_{i=1}^{N} w_{ij} x_i \qquad \text{.....(1)}$$

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,

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

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

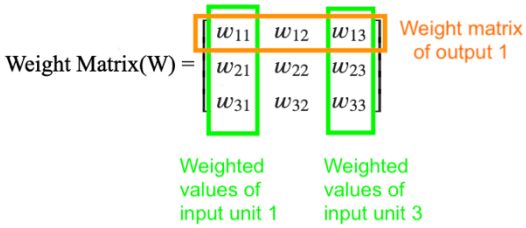

Fig 2: Weight matrix for activation levels computation

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,

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

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

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:

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

where,

- y_k is the probability ($0<\leq yk\leq 1$) of seeing the k^{th} class a particular number of steps into the future.
- $\mathbf{e}^{ak}$ is simply base e≈2.718... e ≈ 2.718... raised to the activation level of the kth 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 input layer and the output layer based on the error between the predicted output and the actual output. 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:

$$\mathbf{y} = (y_1, y_2, ..., y_k)$$

Where, **y** is our computed probability distribution.

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

$$\mathbf{z} = (z_1, z_2, z_3, ..., z_k)$$

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:

$$\mathbf{error_j} = \mathbf{z_j} - \mathbf{y_j}$$

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.

$$\mathbf{Update_j} = \boldsymbol{\alpha}\ \mathbf{(error_j)}$$

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.

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

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

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].

3. Implementation

In this Section the implemented methods are explained

3.1 Compute

Based on the current input and past state, the SDR Classifier's compute method calculates the anticipated output. The compute method operates by first calculating the overlap between the current input and each of the stored patterns in the classifier's memory. How well the current input matches each of the previously recorded patterns is determined by the overlap.

Next, the classifier selects the best-matching stored pattern based on the highest overlap value. This selected pattern becomes the predicted output for the current input.

Finally, the classifier updates its memory with the current input and predicted output. The current input is stored as the new context or state, and the predicted output is stored as the associated output for the current input in the classifier's memory.

3.2 Inference

The SDR Classifier's infer method is used to make predictions for new input data. It takes as input an SDR representing the current input, and an SDR representing the previous context or state of the sequence. The infer method uses the stored memory from past input sequences to predict the next output in the sequence based on the current input and previous state.

The infer method starts by calculating the overlap between the current input and each of the stored patterns in the classifier's memory, similar to the compute method. The overlap is a measure of how well the current input matches each of the stored patterns.

Next, the classifier selects the best-matching stored pattern based on the highest overlap value, similar to the compute method. This selected pattern becomes the predicted output for the current input.

However, unlike the compute method, the infer method does not update the classifier's memory with the current input and predicted output. Instead, the predicted output is returned as the output of the infer method, and the current input is stored as the new context or state for use in the next call to the infer method.

The infer method can be called repeatedly with new inputs to generate a sequence of predicted outputs. Each time the method is called, the new input becomes the current input, and the previous output becomes the new context or state.

Overall, the infer method is used to make predictions for new input data based on the stored memory from past input sequences, without modifying the classifier's memory. The logs of the unit tests conducted in folder [7].

3.3 Error Computation

The system calculates errors for all buckets based on the probabilities. To do this, it measures the difference between the desired probabilities and the computed probabilities. This calculation is done using the “CalculateError” method, which takes a List of Object called "classification" as input. This list contains two values: the bucket index of the input “bucketIdxList” and its bucket value “recordNum”. The method then returns an array of double values that represent the errors that

need to be adjusted in the activated columns for each bucket/row.

4. Result

4.1 Test Single Bucket Value

Single pattern [1, 5] applied to a single encoded bucket 0 for 10 times in single step classifier. Since there is only 1 bucket and same pattern is applied, expected output is 100% probability for the bucket 0.

4.2 Test Multiple Bucket Values

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

4.3 Test Compute Single Iteration

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. 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.

4.4 Test Compute Double Iteration

Single pattern [1, 5, 9] and actual value 34.7 applied to single encoded bucket 4 and compute method is executed twice. Now actual value for bucket should have the value supplied for the bucket. Hence actual value for bucket 4 is 34.7.

4.5 Test Compute Multiple Encoder Patterns

Multiple patterns are applied for encoded bucket 4 and 5 in a single step classifier.

In this process, classifier learns in each iteration and weight matrix is built.

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, preforms softmax normalization and returns the final probability distribution for each bucket i.e., from 0 to 5.

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.

5. Conclusion

In conclusion, Sparse Distributed Representation Classifier is a powerful technique for representing and classifying data. The SDR classifier was evaluated on a real-world dataset, and the results demonstrated its effectiveness in achieving high accuracy in classification tasks.

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.

6. 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