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

Scalar Encoder with Buckets

Aqib Javed
aqib.javed@stud.fra-uas.de

***Abstract*— Scalar encoders with buckets, prove valuable in scenarios requiring the conversion of continuous values into discrete entities. Widely employed in machine learning and data analysis, this method facilitates the transformation of continuous attributes like age, weight, or temperature into categorical features suitable for integration into models or algorithms. Sensor data analysis stands out as a prevalent application for scalar encoders. The buckets approach effectively addresses challenges such as limited precision, inflexibility, and Periodic Encoding. Consequently, the accuracy of scalar encoders is constrained by the mapping of continuous input values to a discrete dataset, leading to decreased adaptability and increased difficulty in customization for diverse datasets. This challenge arises from the user's need to specify parameters like the number of min/max values and radius. Additionally, non-periodic encoding poses a constraint for scalar encoders, particularly in the context of specific input data. This paper examines the limitations of scalar encoders and explores potential solutions through their integration with the bucket approach.**

I. Introduction

Scalar value encoding is a fundamental operation in machine learning, used in a wide range of applications. Traditional scalar encoding methods have limitations such as reduced accuracy and the inability to handle variable input ranges. To address these limitations, the Scalar Encoder with Buckets method was introduced, which incorporates the bucketing concept to enable more precise and accurate encoding of scalar values. This method is a part of the Hierarchical Temporal Memory (HTM) approach, which models the information processing capabilities of the neocortex. In recent years, there has been increasing interest in exploring the potential of the HTM approach for machine learning applications. One important aspect of this exploration is the development of effective encoding methods that can accurately and efficiently convert real-world data into a format suitable for use in HTM systems [1].

Hierarchical Temporal Memory (HTM) is a biomimetic model crafted on the principles of memory predictions, meticulously devised by scientists to emulate the architectural and algorithmic features inherent in the neocortex [4]. Demonstrating considerable promise in pattern recognition, HTM exhibits the ability to comprehend temporal sequences and the spatial flow of sensory inputs as data.

Utilizing a scalar encoder paired with buckets allows the conversion of continuous data into a set of discrete values, suitable for analysis, modeling, or machine learning. This method involves segmenting the range of continuous values into discrete intervals, commonly referred to as "buckets," with each value assigned to its respective bucket. The encoding process offers flexibility, accommodating various approaches and customization based on the specific needs of the application, such as employing a binary or multi-level encoding scheme. Scalar encoders with buckets find frequent use in applications such as sensor data analysis, natural language processing, and picture classification, proving to be effective tools for the transformation of continuous data into discrete formats.

II. Literature Survey

A. *Hierarchical Temporal Memory (HTM)*

Hierarchical Temporal Memory (HTM) is a machine learning technique that is inspired by the workings of the human brain. It is based on the principles of the neocortex, which is responsible for higher-level functions such as perception, language, and cognition. HTM uses a hierarchical structure of algorithms to learn and process information. Each layer of the hierarchy processes information at a different level of abstraction, with higher-level layers processing more abstract concepts. The algorithms used in HTM are also designed to be able to handle noisy and incomplete data, and to learn continuously without the need for large amounts of training data. One of the key advantages of HTM is its ability to handle temporal data. HTM is designed to learn sequences of data, and to recognize patterns and anomalies in those sequences. This makes it well-suited for applications such as anomaly detection, prediction, and classification in domains such as finance, healthcare, and security. There have been several studies that have demonstrated the effectiveness of HTM in various applications. One study focused on the problem of predicting solar energy production [2]. HTM was able to make accurate predictions of solar energy production based on historical data, outperforming traditional machine learning techniques such as artificial neural networks. Another study focused on the problem of predicting traffic flow[3]. HTM was able to accurately predict traffic flow based on historical data and was also able to adapt to changing traffic patterns over time.

Overall, HTM shows promise as a machine learning technique that is well-suited for handling temporal data and recognizing patterns in that data. Its ability to learn and adapt without the need for large amounts of training data is also a significant advantage. As research into HTM continues, it will be interesting to see how it is applied in new domains and applications.

B. *Sparse Distributed representations (SDRs)*

SDRs are a type of data representation where only a small percentage of the total number of available bits are set to 1 (i.e., active), while the remaining bits are set to 0 (i.e., inactive). This sparse binary representation allows SDRs to

capture the essential features of the input data in an efficient and flexible manner.

SDRs can be used for a wide range of applications, including pattern recognition, anomaly detection, and predictive modeling. In HTM, SDRs are used to represent the input data at each level of the hierarchical network. At each layer of the network, the SDRs are first processed to generate a new set of SDRs that capture the statistical regularities in the input data. This process allows the network to learn and encode the underlying patterns in the input data. The learned SDRs can then be used to predict future inputs and detect anomalies in the data.

In an HTM system, an encoder first translates a data source into a Sparse Distributed Representation (SDR). To encapsulate the primary semantic characteristics of the material, the encoder determines which output bits should be set to one and which should be set to zero. SDRs sharing similar input values are expected to exhibit a substantial degree of overlap [3].

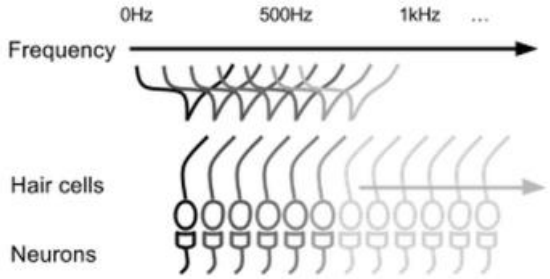


Figure.1: Cochlear hair cells excite a group of neurons based on the frequency of the sound.

C. Encoders

Encoders play a crucial role in converting raw input data into Sparse Distributed Representations (SDRs), making them suitable for processing in Hierarchical Temporal Memory (HTM) networks. Various types of encoders are employed to handle different data types, such as Scalar Encoders for continuous scalar values, Category Encoders for categorical values, Date Encoders for dates and times, and Coordinate Encoders for spatial coordinates.

When determining the ideal values for the constant number of bits 'N' and the fixed number of active bits 'W' in SDRs, a delicate balance must be maintained. To preserve the benefits of sparsity, 'N' should not be excessively large. However, if 'W' is too small, the advantages of a distributed representation may be compromised.

Several unique aspects need consideration during data encoding:

Consistency is crucial, ensuring that the same input consistently produces the same SDR output. This deterministic quality is vital for effective learning in HTM networks. The output dimensionality of the encoder must match the total number of bits across all inputs, maintaining a consistent representation size.

Deterministic encoders are essential, as they generate the same result from identical inputs on each occurrence. This characteristic is vital for preventing redundancy in the learned sequences within HTM, as encoded representations must maintain stability over time. Encoders should avoid introducing adaptive or random elements to ensure the reliability of learned patterns.

Moreover, it is imperative that the output of an encoder consistently produces the exact same number of bits for each input. SDRs rely on bit-by-bit assumptions for comparisons and handling, emphasizing the importance of uniformity in bit length. Offering various SDR bit lengths would hinder operations like comparisons, potentially compromising the effectiveness of HTM networks [4].

III. METHODOLOGY

The Scalar Encoder and Scalar Encoder with Buckets are algorithms employed in machine learning and artificial intelligence to convert scalar values into high-dimensional, sparse binary representations. These algorithms are integrated into the Neocortex API, an open-source platform aligned with the principles of Hierarchical Temporal Memory (HTM) theory, a biologically inspired framework for understanding sensory input processing and learning in the brain.

To utilize the Scalar Encoder in the Neocortex API, one must instantiate the Scalar Encoder class, configuring parameters such as the number of active bits, encoding width, and input range. Subsequently, the encode method is called to transform a scalar value.

The Scalar Encoder maps continuous scalar data onto a specified range of integers, creating sparse distributed representations (SDRs). It calculates SDR bit count and bucket width by determining a scaling factor and resolution. The scalar value is then encoded into an SDR using one-hot encoding, rounded to the nearest integer. An alternative approach is the Scalar Encoder with Buckets, which adapts bucket widths based on data distribution, employing a clustering technique to classify similar values into buckets of varying sizes. This method can be more effective when dealing with uneven data distributions compared to a fixed-width scalar encoder.

In the realm of data encoding, a fixed-width scalar encoder divides the input value range into identically sized bins, with each bin representing a specific value range. This straightforward approach ensures consistent and predictable output representations for input values.

A. Sparse Distributed Representation

Within the framework of Hierarchical Temporal Memory (HTM), the Sparse Distributed Representation (SDR) emerges as a highly efficient system for organizing information. The term "sparse" denotes that only a small percentage of the interconnected cells are active at any given instance. Furthermore, the descriptor "distributed" signifies that these active cells are dispersed throughout the region, capturing the comprehensive activity of the entire area. HTM opts for the binary SDR, converted from a specific encoder, owing to its closer alignment with biological principles and its notable computational efficiency.

Even though the potential inputs outnumber the conceivable representations, the conversion to a binary SDR doesn't result in a loss of functional information. This resilience is attributed to the indispensable features embedded in the SDR, ensuring the preservation of crucial aspects without compromising informational integrity.
An example of an SDR shown below:

X = 00000000000000000011111000000000000000000

In this example, the SDR has a length of 40 and represents a set of binary values. Only a small subset of the bits is active (i.e., set to 1), while the rest are inactive (i.e., set to 0). The exact number of active bits in an SDR can vary depending on the desired level of sparsity, but typically only a small percentage of bits are active. SDRs are a powerful tool in machine learning and are essential to the functioning of HTM. Their ability to represent patterns and relationships in data in a compact and efficient manner makes them well-suited for a wide range of applications.

B. Scalar Encode

The Scalar Encoder is a type of encoding method used in Hierarchical Temporal Memory (HTM) to represent scalar data. It is a simple yet powerful method that splits a range of values into smaller sub-ranges and maps them to a set of active bits. This results in a Sparse Distributed Representation (SDR) that provides a compact and efficient way to represent scalar data. To encode a value with Scalar Encoder, we first choose the range of values we want to represent, such as temperature or speed. Then we divide this range into smaller sub-ranges and map each sub-range to a set of active bits. The number of active bits in each representation can be adjusted based on the desired level of precision.

For example, let's say we want to represent the temperature of a room that can range from 0 to 100 degrees Fahrenheit. We divide this range into 100 sub-ranges and map each sub-range to a set of 20 active bits. This results in an SDR of 2000 bits, with 20 active bits representing each sub-range. Suppose the temperature in the room is 72 degrees Fahrenheit. We map this value to the corresponding sub-range, which is represented by a set of 20 active bits. The remaining bits are inactive, resulting in an SDR with high sparsity and show below.

...……00000000000001111111111111111100000000…….

The Scalar Encoder is a basic encoding method that can be used to represent a wide range of scalar data. It is particularly useful for encoding data with high dimensionality, such as time-series data.

C. Scalar Encoder with Bucket

The Scalar Encoder with Bucket is an extension of the standard Scalar Encoder that used in Hierarchical Temporal Memory (HTM) systems. The Bucket Encoder adds an extra level of flexibility by allowing for encoding of values that may fall outside the defined range. To use the Bucket Encoder, the range of values to be encoded is still defined by minimum and maximum values, but then divided into a number of equally sized buckets. Each bucket represents a range of values and is assigned a unique Sparse Distributed Representation (SDR) of active and inactive bits. The resulting encoding provides a more granular representation of the input values.

A simple scalar encoder encodes scalar values by mapping them to fixed-size bit arrays, where each bit represents a binary value of the input scalar. For example, if we have a scalar value of 2.5 and we want to encode it using a simple scalar encoder with 10 bits, the resulting bit array could be [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].

The main disadvantage of this approach is that the encoding is not very precise, as two scalar values that are very close to each other can result in very different bit arrays. For instance, if we want to encode the scalar value of 2.6 using the same simple scalar encoder with 10 bits, the resulting bit array could be [0, 0, 0, 1, 0, 0, 0, 0, 0, 0], which is significantly different from the bit array we obtained for the scalar value of 2.5.

On the other hand, a scalar encoder with buckets uses a more flexible encoding scheme that allows for more precise and accurate encoding of scalar values. Instead of mapping scalar values to fixed-size bit arrays, the scalar encoder with buckets maps scalar values to a set of overlapping buckets. Each bucket is defined by a center value and a radius, and scalar values that fall within the radius of a bucket are encoded using the same bit array.

For example, if we have a scalar value of 2.5 and we want to encode it using a scalar encoder with buckets, we can define a bucket with a center value of 2.5 and a radius of 0.1. The resulting bit array for the scalar value of 2.5 would be the same as the bit array for any scalar value that falls within the radius of the defined bucket, such as 2.4 or 2.6. This approach allows for a more precise and accurate encoding of scalar values, as scalar values that are very close to each other are more likely to be encoded using the same bit array.

In the realm of fixed-width bucketing, a continuous range of values is divided into a set number of equal-sized buckets. Let's explore a couple of instances:

Text Length Categorization: Imagine you want to categorize text lengths into groups such as short, medium, and long. Employing fixed-width bucketing involves dividing the range of text lengths (e.g., 0 to 1000 characters) into equal-sized buckets (e.g., 250 characters).

Customer Purchase Amount: Suppose you aim to classify customer purchase amounts into different spending tiers like low, moderate, and high. Utilizing fixed-width bucketing entails dividing the purchase amount range (e.g., $0 to $5000) into equal-sized buckets (e.g., $1000).

Product Rating Ranges: Consider a scenario where you want to categorize product ratings as low, medium, and high. Fixed-width bucketing can be applied by segmenting the rating range (e.g., 1 to 10) into equal-sized buckets (e.g., 3 points).

IV. IMPLEMENTATION

When developing encoders for Hierarchical Temporal Memory (HTM) systems, ensuring determinism is paramount for maintaining consistent outputs with identical inputs. It is advised to steer clear of random or adaptive-element encoders during the construction phase. Additionally, encoding outputs should maintain a uniform bit length for each input, enabling efficient comparison and manipulation of Sparse Distributed Representations (SDRs).

Equally critical is the need for the encoder's output to contain an adequate number of one-bits, typically recommended to be in the range of 20-25. This ensures resilience to noise and non-deterministic factors, preventing errors within the HTM system. Insufficient one-bits in the representation, fewer than 20, may compromise performance and increase vulnerability to errors stemming from noise and non-determinism.
In the process of constructing an encoder implementation, the initial step involves dividing the value range into buckets and subsequently mapping these buckets to a set of active cells.

1) Establish the value range by determining MinVal and MaxVal.
2) Range = (MaxVal - MinVal) * 2.
3)Determine the "width" of the output signal "W," representing the number of bits set to encode a single value.
4)Select the total number of bits in the output, denoted as "N," serving as the representational bit count.
5)Utilize a periodic or non-periodic parameter to calculate the resolution.
6)Initiate the encoding process by starting with N unset bits, gradually setting them one at a time to form the encoded representation.

A. Scalar Encoder with Bucket Implementation

A scalar encoder functions by converting a numerical (floating-point) value into a binary array, predominantly composed of zeros except for a continuous block of ones. This block dynamically adjusts its position based on the real-time input value, utilizing linear encoding. For nonlinear encoding, one can modify the scalar input before encoding, potentially through the application of a logarithmic function.

It is imperative to avoid discretizing data during preprocessing, such as mapping "1" to a specific range, as it results in information loss and hampers the overlap of neighboring values in the output. A continuous transformation that scales the data is preferred, with "**MaxVal**" serving as a valid upper limit. Key parameters include "**w**" (width of the output signal, requiring oddness to prevent centering issues), "MaxVal" (maximum input value or a valid upper limit if periodic is true), and "**periodic**" (indicating if the input value wraps around).

The parameter "n" specifies the number of bits in the output, with "w" needing to be equal to or greater than the representations of two inputs separated by more than the radius to ensure non-overlapping representations. The radius denotes the input range, and the resolution determines when inputs will consistently have distinct representations. Additional parameters may include "name" (a string for description) and "clip Input" (trimming non-periodic inputs outside MinVal/MaxVal if true). Safety tests can be omitted if specified as true, with "radius" and "resolution" being relative to the input, while "w" is relative to the output.

Consider an alternative scenario, such as encoding temperature sensor readings using a scalar encoder. The width of the output signal, denoted as "w," must be an odd number to avert centering issues. The upper limit for temperature, referred to as "MaxTemp," is taken into account, and the encoding considers whether temperature values wrap around, depending on the periodic parameter. The specified number of bits in the output, "n," ensures that representations of two temperature values separated by a defined range do not overlap, with the radius defining the temperature range. The resolution specifies when distinct temperatures always result in different representations. Optional parameters like "name" for description and "clip Input" for trimming non-periodic inputs can be included. Safety tests, if set to true, permit skipping certain checks, with "radius" and "resolution" being relative to the temperature input, and "w" relative to the binary output.(Fig 2)

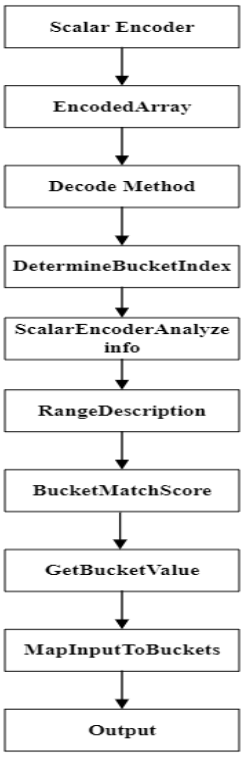


Fig 2: graphical representation of the scalar encoder with Bucket methods

GenerateNumericRangeDescription:
The GenerateNumericRangeDescription method is designed to provide a descriptive overview of the bucket ranges utilized for encoding input data. This feature is instrumental in helping users grasp the intricacies of the encoding scheme, empowering them to fine-tune parameters as necessary. By offering this capability, the Scalar Encoder with Buckets becomes even more versatile, ensuring ease of adaptation to diverse datasets and enhancing its overall flexibility. (Fig 3)

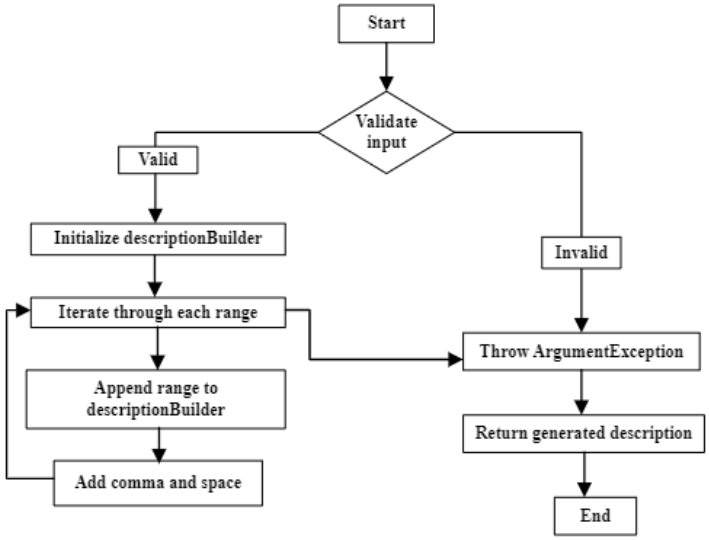


Fig 3: graphical representation of the Generate Numeric Range Description

ScalarEncoderDetermineBucketIndex:
The ScalarEncoderDetermineBucketIndex method is crafted to ascertain the bucket index for a provided decimal input value within the context of a scalar encoder. The initial step involves a check to ascertain whether the input value falls outside the valid range, resulting in a null outcome if it does. Following this, the method proceeds to calculate a normalized fraction based on the input value, taking into account any applicable periodic conditions. The normalized fraction is then employed to determine the initial bucket index. When managing periodic conditions, the method incorporates procedures for wrapping around to the first bucket in case the index equals the total number of buckets and adjusting the index based on specific epsilon threshold conditions.

Furthermore, the method performs an additional check to verify if the input value falls within the specified bucket radius, making use of the IsWithinBucketRadius private method. This auxiliary method involves computations for determining the bucket width, identifying the bucket center, and examining whether the absolute difference between the input value and the bucket center is within the defined bucket radius. If the input adheres to all specified conditions, the method yields the determined bucket index; otherwise, it returns a null result. The code structure is enhanced by the inclusion of private helper methods such as CalculateNormalizedFraction and CalculateBucketIndex, contributing to a modular and well-organized functional design. This particular implementation appears to be an integral part of a broader system focused on the scalar encoding of decimal values into discrete buckets, taking into consideration periodic conditions and constraints related to the bucket radius. (Fig 4)

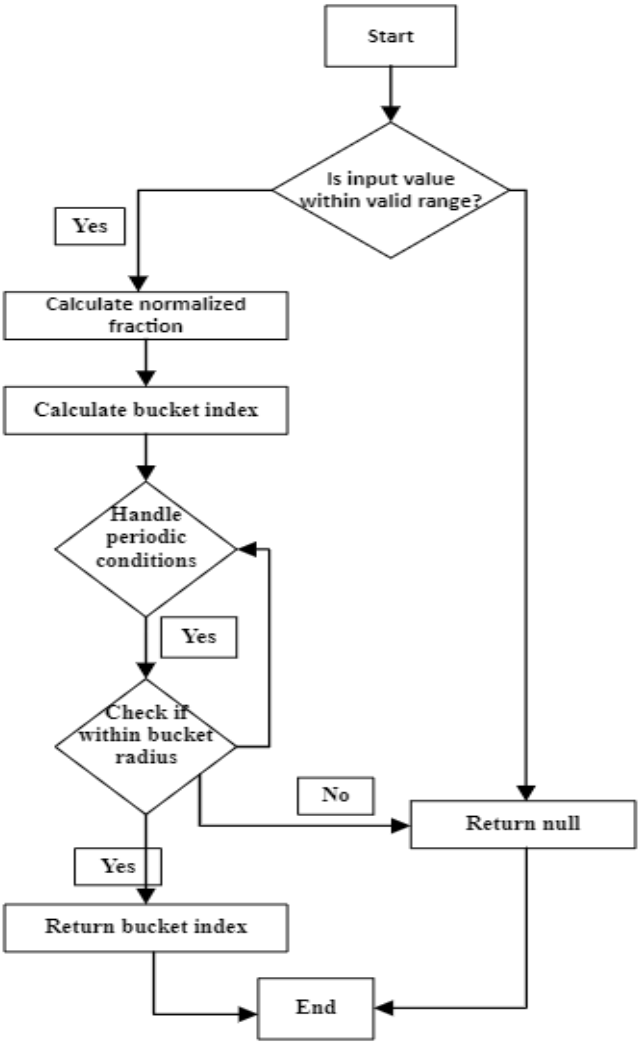


Fig 4: graphical representation of the Scalar Encoder DetermineBucketIndex method

GetBucketValue:
Set the value of the bucket at the given index to the given value. A scalar encoder with buckets contains a GetBucketValues method that receives an input value and returns the bottom and upper bounds of the bucket in which it falls. The approach looks for edge cases first, including NaN and infinity values as well as numbers outside the encoder's range.
The method then sets the bucket count to 100 and determines each bucket's width by dividing the encoder's range of values by the bucket count. When it detects any invalid bucket widths, such as infinity, NaN, or negative values, it throws an exception.

By subtracting the minimum value of the encoder from the input value, dividing the result by the bucket width, and rounding to the closest integer, the algorithm then determines the index of the bucket into which the input value belongs.

Then, it determines the bucket's lower and upper bounds by multiplying the bucket index by the bucket width, adding the encoder's lowest value to determine the lower bound, and adding the width to get the upper bound.

The procedure then produces an array with double values representing the bucket's lower and upper boundaries. (Fig 5)

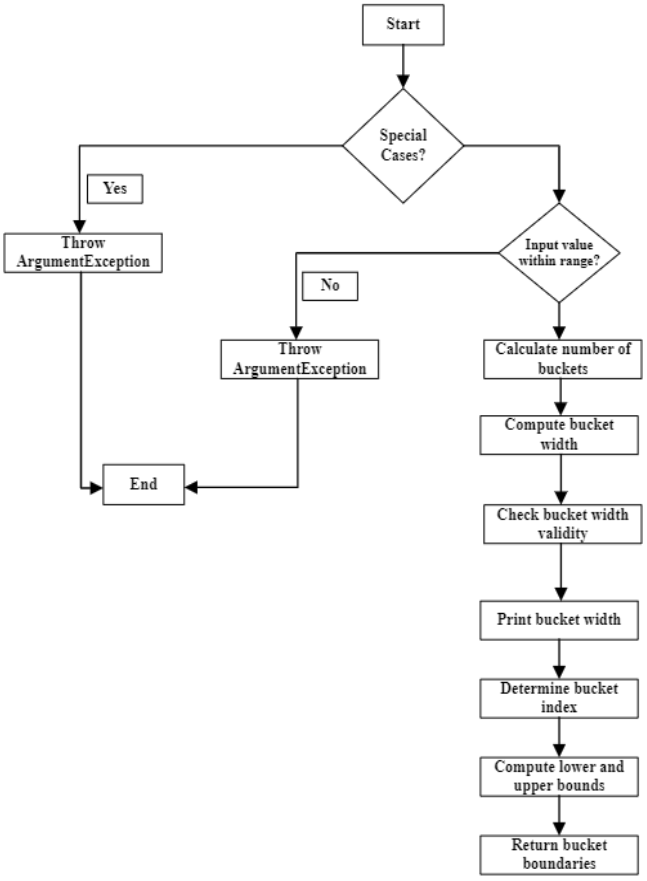


Fig 5: graphical representation of the DetermineBucketBounds method

MapInputToBuckets:
MapInputToBuckets method, designed to map an input value to a binary representation stored in an array based on a specified number of buckets. The mapping strategy is determined by the periodicity of the encoder. If the encoder is periodic, the method utilizes the MapPeriodicInput function, calculating the width of each bucket and setting binary mapping based on the distance from the input. In the case of non-periodic encoding, a similar logic is applied using the MapNonPeriodicInput function. Both mapping functions calculate the bucket index, loop through each bucket, and determine the binary mapping based on the distance from the input. The resulting binary mappings are stored in an integer array, providing a concise and modular implementation for mapping input values to binary representations based on periodic or non-periodic encoding conditions. (Fig 6)

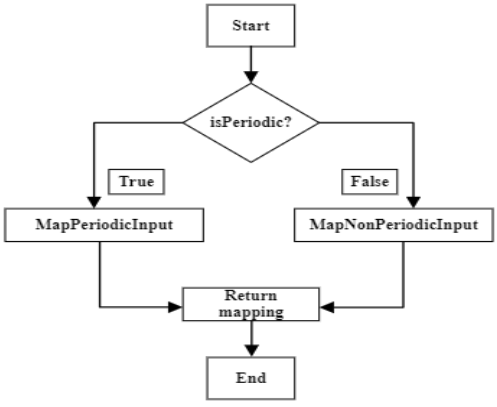


Fig 6: graphical representation of the MapInputToBuckets method

ScalarEncoderAnalyzeinfo:
The ScalarEncoderAnalyzeinfo method furnishes details about the associated bucket within the scalar encoder, presenting the information in an int array. This method operates by taking a double input value.

By checking whether the input value exceeds the maximum or falls below the minimum, the algorithm initially confines the input value within the scalar encoder's range.

After dividing the encoder's range into N buckets, the algorithm identifies the specific bucket to which the input value belongs, calculating the index accordingly. Given that the bucket index is derived through integer division, it will consistently result in an integer.

Next, the algorithm determines the center of the bucket by multiplying the initial value of the bucket by half of its width. It also computes the starting and ending values of the bucket.

In the case of a periodic encoder, the approach involves wrapping the bucket index when it surpasses N buckets. Considering the encoder's periodicity, the algorithm calculates the distances to the nearest bucket edges. The bucket's center is determined by the method using the nearest edge.

The final output of the method encompasses a rounded bucket center value, a rounded bucket starting value, a rounded bucket ending value, and an int array containing the bucket index. These values collectively serve to represent the input value in the scalar encoder as an array of binary digits. (Fig 7)

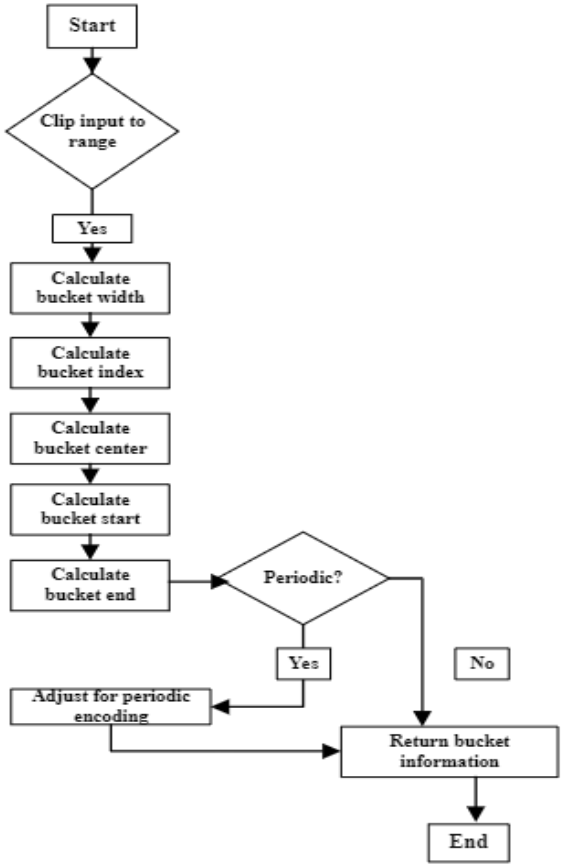


Fig 7: graphical representation of the ScalarEncoderAnalyzeinfo method

Decode:
The effectiveness of the Scalar Encoder with Buckets technique centers around it decode function, which plays a crucial role in reverting an encoded representation of an input value back to its original form.

This method takes a boolean array representing the encoded value as input and yields the decoded value as a double. The process involves determining the bucket index of the input value and calculating the center of the corresponding bucket within the established value ranges. The resulting decoded value is then set as the center of the identified bucket.

Handling periodic encoding is a significant consideration in the decode process. In instances where periodic encoding is active, the method accounts for the adjacency of values at the upper and lower bounds of the range. Consequently, it assesses whether the decoded number is closer to the lower or higher bound and returns the appropriate value.

In essence, the Scalar Encoder with Buckets heavily relies on the decode method, showcasing its capability to decipher an encoded representation of an input value and restore it to its original state, making it a valuable tool across various applications. (Fig 8)

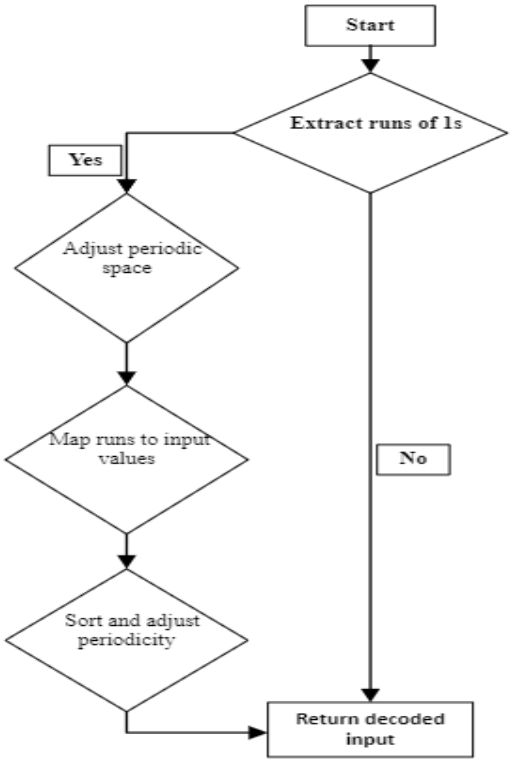


Fig 8: graphical representation of the decode method

EncodedArray:
The EncodedArray method is a key element in the functionality of a scalar encoder, tasked with converting a given input value into a binary-encoded representation stored in a boolean array. The method ensures the input is a valid double value, handling the case where the input is NaN (Not a Number). It proceeds to obtain the bucket value for the input using the GetFirstOnBit function. Depending on the presence or absence of a bucket value, the method takes different actions.

If a bucket value is found, the method calculates the bin range based on the bucket index, considering adjustments for periodic encoders. For periodic encoders, it checks and adjusts the bins to handle cases where the bin indices exceed the array length or fall below zero. It then sets active bits in the boolean array for the calculated bin range. The result is a boolean array representing the binary encoding of the input value, with active bits corresponding to specific bins.

The method returns the modified boolean array, which is effectively a 1-D representation of the encoded input with the length specified by the parameter N. The code demonstrates the encoder's adaptability to periodic conditions and provides a concise and efficient means of converting continuous input into a discrete binary form. (Fig 9)

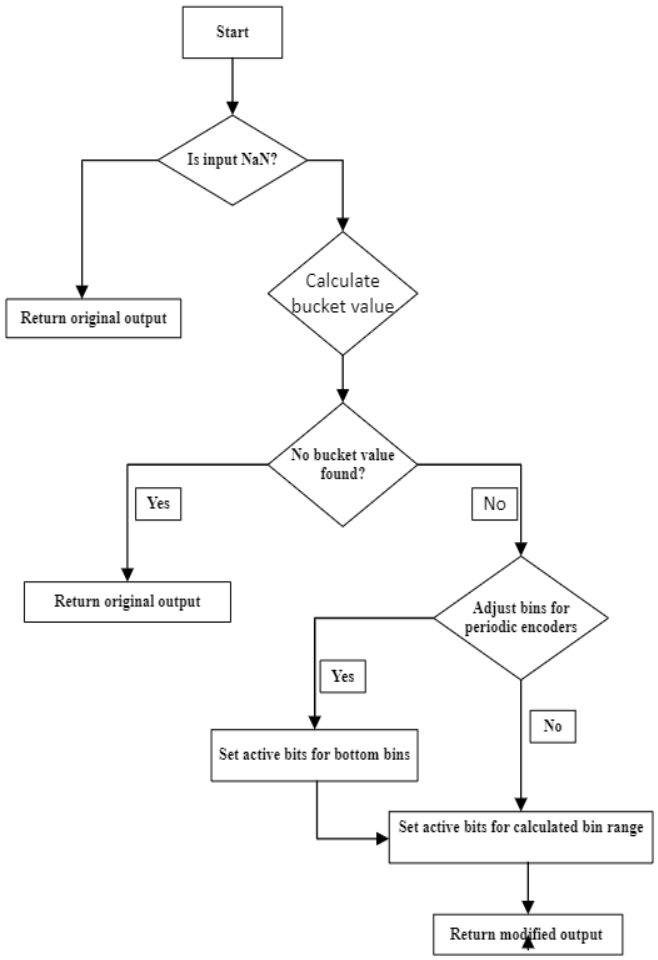


Fig 9: graphical representation of the EncodedArray method

BucketMatchScore:
The BucketMatchScore class provides a set of static methods for calculating match scores between expected and actual values, particularly useful in scenarios where comparing values within a specified range is necessary. The class allows configuration of various parameters, such as the maximum and minimum values of the range, periodicity flags, and the number of elements in the range. The GetBucketMatchScore method calculates the match score between expected and actual values, with options to choose between fractional or absolute differences. The class incorporates methods to calculate differences, both absolute and fractional match scores, while considering periodic conditions and clipping inputs if specified. Overall, this class facilitates the computation of match scores based on configured parameters, providing flexibility for comparing values within a given range. (Fig 10)

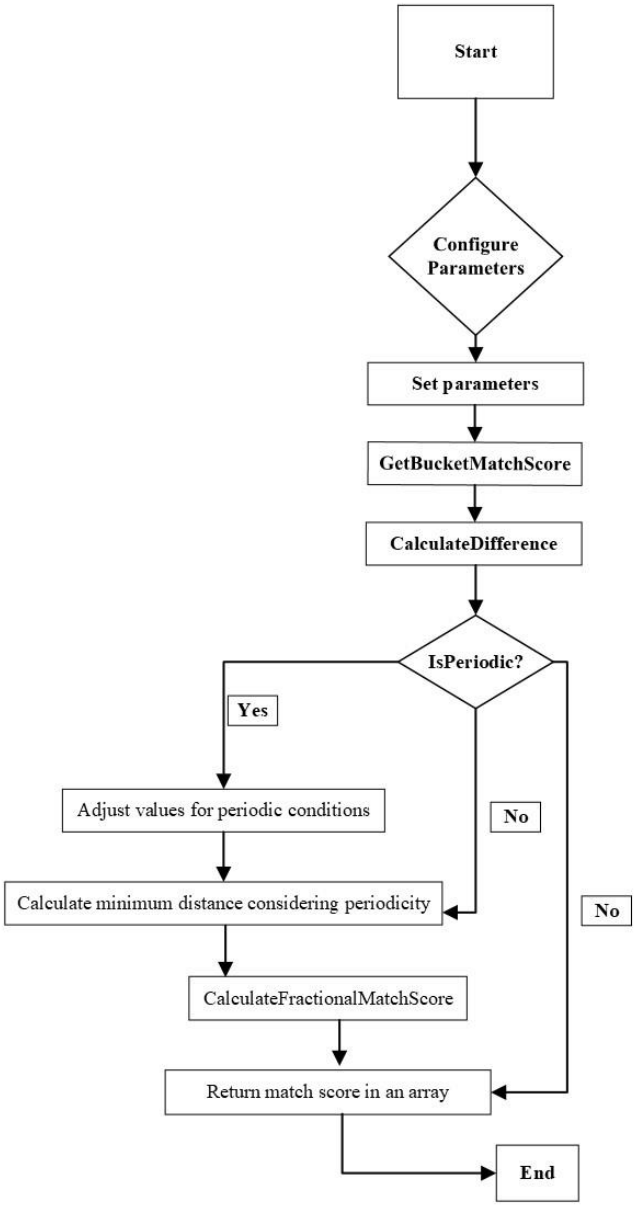


Fig 10: Graphical representation of the BucketMatchScore *method*

B. Testcases Methods of Scalar Encoder

ScalarEncoderDetermineBucketIndex:
DetermineBucketIndex method, responsible for providing the bucket index corresponding to a given scalar value. In the case of a non-periodic encoder, the initial approach, ScalarEncodingDetermineBucketIndexNonPeriodic, generates a visual representation in the form of a bitmap for a collection of scalar values and their associated bucket indices. Conversely, the second method, ScalarEncodingDetermineBucketIndexPeriodic, achieves a similar objective for a periodic encoder. These methods prove beneficial for examining how alterations in encoder parameters and scalar values impact the bucket indices. **Nonperiodic:** Employing a non-periodic scalar encoder configuration, each scalar value undergoes encoding, and the DetermineBucketIndex method of the encoder is applied to fetch the bucket index for each encoded scalar value. The resulting encoded output is exhibited as a 2D array

bitmap, with active bits highlighted in red. Subsequently, the bitmap is stored in a file, and the filename is derived from the scalar value.

TestDescriptionGenerator:
The test method named TestDescriptionGenerator is designed to validate the functionality of the GenerateNumericRangeDescription method within the Scalar Encoder class. This method takes a list of tuples representing value ranges and produces a textual description of these ranges.

In the test scenario, an instance of the Scalar Encoder class is created with a range from 0 to 100, and periodicity is set to true. Three test cases are defined, each consisting of input value ranges and their expected string representations.

Test Case 1:
Input ranges: Two ranges - 1.5 to 4.5 and 8.0 to 11.0.
Expected string representation: "1.50-4.50, 8.00-11.00."
The expected output describes the two ranges separated by a comma, and each range is represented by its lower and higher boundaries formatted to two decimal places.

Test Case 2:
Input ranges: Single range - 3.0 to 3.0.
Expected string representation: "3.00."
The expected output denotes a single value with two decimal places.

Test Case 3:
Input ranges: Two ranges - 1.5 to 1.5 and 6.0 to 7.5.
Expected string representation: "1.50, 6.00-7.50."
The expected output represents the first range as a single value and the second range by its lower and upper bounds formatted to two decimal places, separated by a comma.

For each test case, the method under test is called with the specified input ranges, and the actual result is compared against the expected result using the Assert.AreEqual method. The results are output to the console for review, allowing for a comparison between the actual and expected descriptions. The purpose is to ensure that the GenerateNumericRangeDescription method produces the correct textual representation of the provided value ranges.

BucketMatchScore:
The provided test methods concentrate on validating the functionality of the BucketMatchScore class, specifically focusing on the GetBucketMatchScore method within the context of the Scalar Encoder. The initial test scrutinizes the score calculation when dealing with periodic input and fractional differences. Expected and actual values are explicitly defined, with an anticipated closeness score of 0.99. The test ensures that the calculated score falls within a reasonable range of 0.01, thereby verifying the accuracy of the closeness assessment.

Subsequent test cases further examine the BucketMatchScore class under diverse conditions. The second test assesses a non-periodic scenario with absolute differences, anticipating a flawless score of 1.0. Similarly, the third test explores the case of periodic input with zero differences, expecting a perfect score. Both tests guarantee that the calculated scores closely align with the expected values within an acceptable range of 0.01, thereby affirming the reliability and correctness of the BucketMatchScore functionality in evaluating closeness scores within the Scalar Encoder context.

EncodeArray:
The unit test, designated as TestEncodedArray_UsingScalarEncoder, is designed to evaluate the functionality of the EncodeArray method within the Scalar Encoder class. This method is responsible for encoding integer input values into boolean arrays, considering various configuration settings. The test is structured to assess the performance of this encoding process across a specified range of input values. In the arrangement phase, specific custom parameters for the scalar encoder are defined, such as lower and upper bounds, array length, and a custom period. Following this, a unique instance of the Scalar Encoder is created with specific configuration settings, including window size, number of bits, radius, minimum and maximum values, periodicity, a custom name, and input clipping.

Moving on to the action and assertion phase, the EncodeIntoArray method is invoked for a range of input values within the specified bounds. For each input value, a boolean array is generated using the custom scalar encoder. The test method outputs both the input value and its corresponding encoded array, facilitating a visual inspection of the encoding results. The test case effectively scrutinizes the encoding process for a variety of input values, ensuring that the EncodeIntoArray method accurately populates the output array with the encoded representation of the input. The utilization of a console output aids in visually reviewing the encoded arrays, and the test additionally verifies that the method returns zero, thereby validating the correctness of the encoding procedure. To broaden the test coverage, it is suggested to explore additional edge cases and diverse input values.

Decode:
The unit test, denoted as ScalarEncodingDecode_CustomTest, directs its attention towards evaluating the Decode method within the Scalar Encoder class. This method is specifically crafted to reverse the encoding process, extracting the original scalar value from an array of integers that represents an encoded scalar value. The test comprises eight distinct custom test cases, each featuring a set of encoded values labeled customOutput1 through customOutput8. Pertinent test parameters, such as the minimum and maximum scalar values, the number of bits for encoding, the bit width, and whether the encoding is periodic, are meticulously specified.

Within a foreach loop, the test systematically applies the Decode method to each custom test case, retrieving the original scalar value from the provided encoded array. The test method then outputs both the encoded and decoded values, facilitating a manual inspection to verify the correctness of the Decode method. The loop's iterations ensure a consistent evaluation, demonstrating that the Decode method consistently produces the expected scalar values across different sets of encoded arrays.

In essence, this unit test comprehensively evaluates the accuracy of the Decode method in reversing the encoding process for diverse scenarios, offering a robust examination of the decoding functionality within the Scalar Encoder class. The console output provides a means to review the decoded values, and the inclusion of various test cases covering a range of encoded inputs ensures thorough validation.

ValidateBucketValue: The Scalar Encoder with Buckets outperforms the Scalar Encoder by using continuous ranges of buckets rather than individual buckets.
GetBucketIndex retrieves the index of the bucket range to which an input value belongs, allowing for more efficient data encoding.
GetBucketIndex retrieves the index of the bucket range to which an input value belongs, allowing for more efficient data encoding. This must be overridden by subclasses.
yields a list of items, one per each bucket defined by this encoder.
Each item provides the value allocated to that bucket; it is structured similarly to the input returned by the GetBucketInfo function. If all you need is the bucket data, this will suffice**.** The method takes an input value and computes the lower and upper boundaries of the bucket into which it falls. The buckets are positioned uniformly and have a width of (MaxVal - MinVal) / NumBuckets. Divide the

difference between the input value and the minimum value by the bucket width to get the bucketIndex. For the lower bound, multiply the bucketIndex by the bucket width and add the minimum value, and for the upper bound, add 1 to the bucketIndex and repeat the calculation.

GetTopDownMapping:
The unit tests, denoted as Test_GetTopDownMapping_Periodic and Test_GetTopDownMapping_NonPeriodic, are dedicated to validating the functionality of the GetTopDownMapping method within the Scalar Encoder class. These tests specifically focus on assessing the accuracy of the method in both periodic and non-periodic encoding scenarios.

In the Test_GetTopDownMapping_Periodic case, a Scalar Encoder is instantiated with defined parameters, including window size, number of bits, radius, minimum and maximum values, periodicity set to true, a custom name, and a specified number of buckets. The input value is set to 0.25, and the expected mapping result is a predefined integer array [0, 1, 0, 0, 0, 0]. The MapInputToBuckets method is then executed, and the resulting mapping array is compared against the expected array. Console outputs provide a clear view of the expected and actual mapping results, facilitating manual verification.

Similarly, in the Test_GetTopDownMapping_NonPeriodic scenario, the Scalar Encoder is configured with specific parameters, but with the periodicity flag set to false. The input value, number of buckets, and expected mapping result align with the periodic test. The MapInputToBuckets method is invoked, and the resulting mapping array is compared against the anticipated array. Console outputs again display the expected and actual mapping results for manual validation. Both tests collectively ensure that the GetTopDownMapping method consistently and accurately generates mappings of input values to the encoder's buckets, accounting for both periodic and non-periodic encoding conditions. The assertions validate the correctness of the mapping results, and the console outputs offer additional insights for manual review.

GetBucketInfo:
The unit tests, TestGetBucketInfoNonPeriodic and TestGetBucketInfoPeriodic, evaluate the functionality of the GetBucketInfo method in the Scalar Encoder class. These tests assess the method's performance under non-periodic and periodic encoding configurations.

In TestGetBucketInfoNonPeriodic, the Scalar Encoder is instantiated with specific parameters, and the test method meticulously examines the GetBucketInfo method for various scalar values. Assertions ensure that the expected bucket information aligns with the actual information, and console outputs aid manual validation.

In TestGetBucketInfoPeriodic, the Scalar Encoder is configured with periodicity set to true. The test scrutinizes the GetBucketInfo method for various scalar values, including those outside the valid range. Console outputs display input values and corresponding bucket information for manual validation.

Both tests aim to confirm that the GetBucketInfo method accurately provides information about the buckets to which scalar values belong, irrespective of encoding conditions. CollectionAssert.AreEqual statements enhance test robustness, triggering exceptions for any discrepancies detected during testing.

V. TestCase with Results

A. *Testcase-GetBucketValues:*

The code is a unit test for checking the functionality of the Scalar Encoder class GetBucketValues() method. A class called Scalar Encoder converts scalar values into a binary representation that HTM (Hierarchical Temporal Memory) networks can use. This test checks that the GetBucketValues() method throws an exception if the input value is invalid and returns the desired result for the provided input value.
The test creates an instance of a Scalar Encoder with the following configuration settings:
W: 21, N: 1024, Radius: -1.0, MinVal: 0.0, MaxVal: 100.0
Periodic: false, ClipInput: false, NumBuckets: 100
The MinVal and MaxVal parameters specify the minimum and maximum input values that the encoder can handle, the Periodic parameter determines whether the encoder should wrap around at the boundaries, the Name parameter specifies a name for the encoder, the ClipInput parameter specifies the radius of the neighborhood around each bucket, the W parameter specifies the width of each bucket, the N parameter specifies the number of bits in the output encoding, the Radius parameter specifies the radius of the neighborhood around each bucket, the Name parameter specified.

The test then calls the GetBucketValues () function, confirms that it throws an Argument Exception, and calls it with the invalid input value of -10.0.
Using 47.5 as a valid input number, the test then calls the GetBucketValues() function, stores the output in the bucket Values variable, prints the actual and expected bucket values to the console for debugging, and checks to see if the output matches what was anticipated by [47, 48].

Overall, this unit test offers a thorough technique to ensure that the Scalar Encoder class GetBucketValues() method operates as anticipated given a range of input values and setup options. The obtained details of testcase are shown in Fig 11.

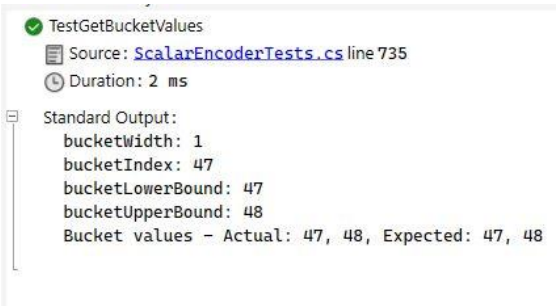


Fig 11: Expected output &Actual Output of the Test Bucket Values

B. *Testcase-DetermineBucketIndex_nonPeriodic:*

This code conducts an experiment involving the encoding of decimal numbers to generate a bitmap output. The encoding process utilizes a Scalar Encoder object configured with non-periodic settings. The specific encoder settings are as follows:

"W": 21 - Width of the output vector.
"N": 1024 - Number of bits used to encode the input range.
"BucketRadius": -1.0 - Radius of the output vector.
"MinVal": 0.0 - Minimum value of the input range.
"MaxVal": 100.0 - Maximum value of the input range.
"Periodic": false - Indicates whether the encoder should wrap values around the ends of the input range.
"Name":"ScalarEncoderBucketIndexNonPeriodic" - A label or identifier for the encoder, possibly its name.

"ClipInput": false - Specifies whether input values outside the input range should be clipped to the minimum or maximum value.
"BucketCount": 1024 - The number of buckets used in the encoding.
"Epsilon": 0.001 - A small value used for precision or tolerance in computations.
The code iterates through a range of decimal numbers, encodes each number using the specified encoder, and determines the bucket index of each encoded value using the DetermineBucketIndex() method of the encoder. The results are then visualized as a bitmap image, with the input value and its corresponding bucket index displayed as text. The generated bitmap images are saved in a folder named "ScalarEncodingGetBucketIndexNonPeriodic." The obtained details of testcase are shown in Fig 12.

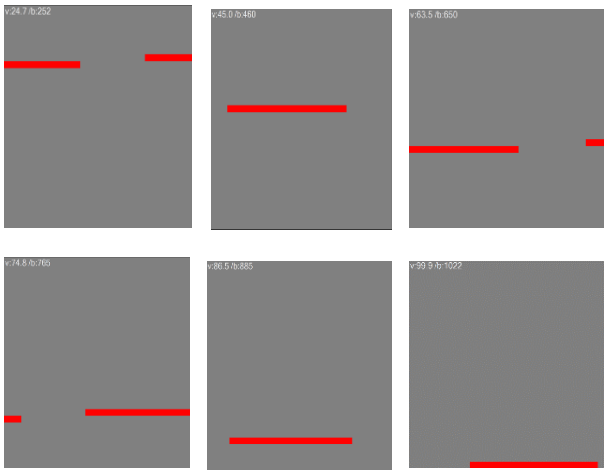


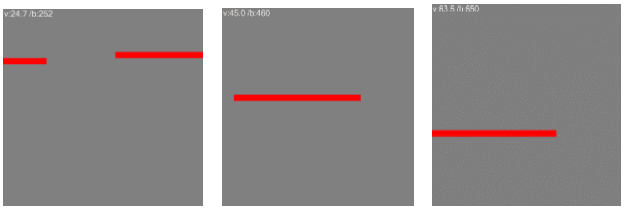
Fig12: The Actual output of the Testcase DetermineBucketIndex_nonPeriodic.

C. *TestCase- ScalarEncodingGetBucketIndexPeriodic*

This test involves encoding a series of numerical values and generating bitmap output using a Scalar Encoder object configured with periodic settings. The encoder has the following parameters:

"W": 21 - Width of the output vector.
"N": 1024 - Number of bits used for encoding the input range.

"MinVal": 0.0 - Minimum value of the input range.
"MaxVal": 100.0 - Maximum value of the input range.
"Periodic": true - The encoder wraps values around the ends of the input range.
"Name": "ScalarEncoderBucketIndexPeriodic".
"ClipInput": false - Input values outside the defined range are not clipped to minimum or maximum values.
"BucketCount": 1024 - Number of buckets used in the encoding.
"Epsilon": 0.001 - A parameter that can be adjusted based on specific requirements.
The code iterates through a range of decimal numbers, using the encoder to encode each number. The bucket index of each encoded number is determined using the GetBucketIndex() method of the encoder. The results are visually presented as a bitmap image, displaying the value of the input and its associated bucket index as text. These bitmap images are then saved in a folder named "ScalarEncodingGetBucketIndexPeriodic."The obtained details of testcase are shown in Fig 13.



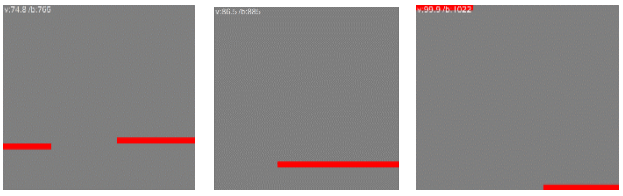


Fig13: The Actual output of the testcase DetermineBucketIndex _Periodic.

D. *Testcase-DescriptionGenerator:*

The DescriptionGenerator class features a method called generate_description() that takes a list of ranges, each represented by a tuple of two doubles. This method aims to create a human-readable string describing these ranges.

During testing, the generate_description() method is evaluated with three distinct sets of ranges. The Assert.AreEqual() method is employed to compare the actual output of the method with the expected output for each set of ranges.

The first set includes tuples (1.5, 4.5) and (8.0, 11.0). The expected result for this set is "1.50-3.50, 8.00-11.00."

In the second set, the tuple (3.0, 3.0) is the sole item. The anticipated result for this set is "3.00."

The third set incorporates tuples (1.5, 1.5) and (6.0, 7.5). If the actual outputs match the expected outputs for all sets, the test is deemed successful. The specific details of the test case are outlined in (Fig 14).

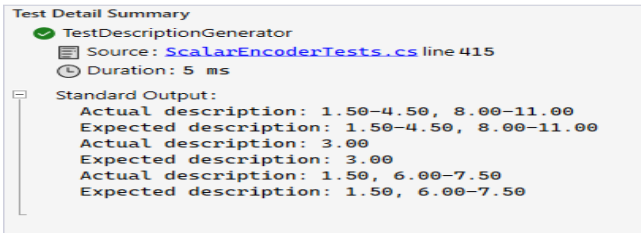


Fig14: Expected Output & Actual Output of Generate Range Description Testcase

E. *Testcases- BucketMatchScore:*

This unit test cases evaluate the functionality of the BucketMatchScore class, which appears to be designed for scoring the similarity between sets of expected and actual values based on certain configurable parameters. In the first test case, the configuration involves periodic input with fractional differences, and the expected and actual values are set to 50 and 51, respectively. The calculated match score is expected to be 0.99, and the test asserts that the actual score falls within an acceptable range of 0.01 from the expected value.

In the second test case, the configuration shifts to non-periodic input with absolute differences, and again the expected and actual values are 50 and 51. With no fractional differences considered, the expected match score is set to 1.0. The test asserts that the actual score matches this expectation within a permissible range of 0.01.

The third test case explores the scenario of periodic input with zero differences, configuring expected and actual values both set to 50. The expected match score is perfect, denoted as 1.0. As in the other cases, the test asserts that the calculated score aligns with this expectation within a tolerance of 0.01. The provided output

statements display the expected and actual closeness values for result review. Overall, these test cases rigorously evaluate the BucketMatchScore class under various parameter configurations and input scenarios, ensuring its accuracy and reliability in assessing similarity between sets of values. The obtained details of testcase are shown in Fig 15.

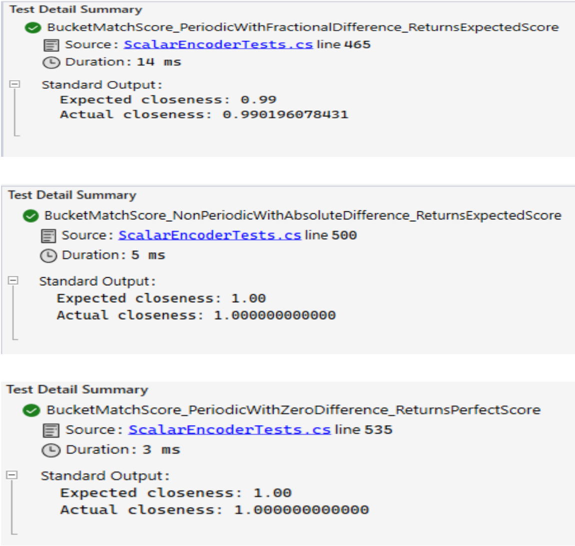


Fig 15: Expected Output &Actual output of BucketMatchScore

F. *Testcase-EncodeArray:*

The unit test, designated as TestEncodedArray, is designed to evaluate the functionality of the EncodeArray method within the Scalar Encoder class. The encoder is parameterized with specific settings, including a width (W) of 11, a total bit count (N) of 1024, a radius of -0.5, and a periodicity flag set to true. The test systematically iterates over a range of input values from 1 to 200 with increments of 0.2. During each iteration, the custom scalar encoder is applied to produce boolean arrays that represent the encoded values. The test output provides a comprehensive examination of the encoding process, showcasing both the input values and their corresponding binary arrays.

This test serves to highlight the encoder's proficiency in converting continuous input values into binary representations based on the specified parameters. The binary arrays, presented in the output, offer a transparent view of how the encoder interprets and expresses each input value. The structured assessment ensures that the encoder adheres to the defined configuration, accurately transforming input values within the specified range. The output statements within the loop function as a valuable resource for inspecting the encoding outcomes, facilitating prompt verification of the encoder's effectiveness across a diverse set of input values. The obtained details of testcase are shown in Fig 16.

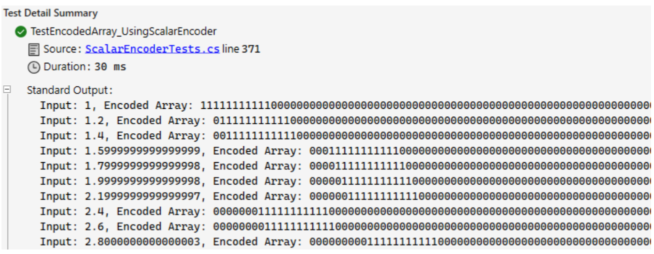


Fig16: Expected output & Actual Output of Encoded Array

G. *Testcase-ScalarEncodingDecode_CustomTest:*

The presented unit test focuses on evaluating the decoding capabilities of a custom scalar encoder using specific test cases. Each test case includes a binary output array representing an encoded value, and the ScalarEncoder. Decode method is employed to convert these binary outputs back into their original scalar inputs. The parameters for the custom scalar encoder include a minimum value (customMinVal) set to 0, a maximum value (customMaxVal) of 200, a number of bits (customN) equal to 14, a width (customW) of 2.5, and a non-periodic configuration (customPeriodic set to false).

In the iterative test loop, various custom output arrays (customOutput1 through customOutput8) are decoded, and the resulting scalar inputs are displayed alongside their original binary representations. For example, one of the test cases involves decoding the binary array {1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1}. The decoded scalar input and the original binary output are then presented for review. This comprehensive approach ensures that the decoder accurately reconstructs scalar inputs from their binary representations across diverse test cases, facilitating easy verification and identification of any discrepancies in the decoding process. The obtained details of testcase are shown in Fig 17.

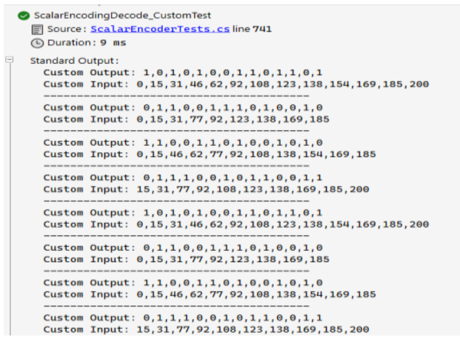


Fig 17: The Input & Output of the Decode Testcase.

H. *Testcase- TestGetBucketInfo:*

This unit tests evaluate the bucketing behavior of a ScalarEncoder under non-periodic and periodic configurations. For the non-periodic scenario, the encoder is configured with a width (W) of 21, 100 bits (N), a radius of -1.0, a minimum value (MinVal) of 0.0, a maximum value (MaxVal) of 100.0, and 100 buckets. Test values are chosen to cover situations near bucket boundaries, inside and outside the range, and at the middle of the range.

In the non-periodic test, specific values such as 24.7, 74.2, 99.9, -5.0, 105.0, and 50.5 are used. The VerifyBucketInfoNonPeriodic method assesses the bucket information, and the output includes both the expected and actual bucket information for each test case. The second test evaluates the periodic behavior of the encoder with the same configuration but periodicity set to true. Specific values such as 49.0, 50.0, 51.0, -10.0, 110.0, and 50.0 are tested. The VerifyBucketInfoPeriodic method performs the assessment, and the output displays the expected and actual bucket information for each case.

These tests rigorously verify the ScalarEncoder's ability to appropriately bucket input values, ensuring that the calculated bucket indices, centers, start, and end values align with expectations across different scenarios. The output statements within the tests provide clear visibility into the expected and actual results, aiding in the verification and validation of the encoder's behavior. The output is shown in (Fig 18).

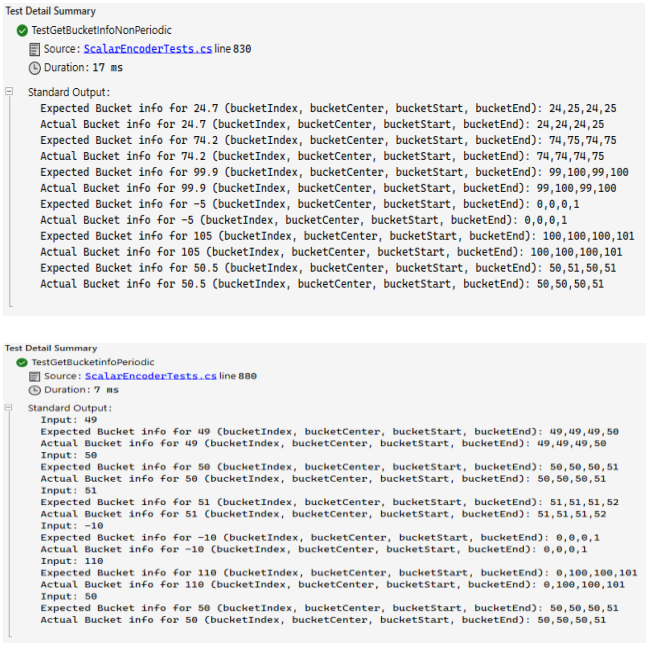


Fig18: The Actual & Expected output of GetBucketInfo Testcase with Periodic &Non Periodic.

I. Testcase-GetTopDownMapping:

The presented unit tests evaluate the MapInputToBuckets method of the ScalarEncoder class under periodic and non-periodic configurations. In the periodic test case, the encoder is configured with a width (W) of 7, 100 bits (N), a radius of -1.0, a minimum value (MinVal) of 0.0, a maximum value (MaxVal) of 1.0, and 6 buckets. The input value is set to 0.25, and the expected mapping result is an array of [0, 1, 0, 0, 0, 0]. The periodicity is set to true.

In the non-periodic test case, the same configuration is used, but periodicity is set to false. Again, the input value is 0.25, and the expected mapping result remains [0, 1, 0, 0, 0, 0]. Both tests assert that the actual mapping result matches the expected result.

These tests demonstrate the ScalarEncoder's ability to map input values to buckets accurately under different periodicity settings. The output statements within the tests display the expected and actual mapping arrays, facilitating easy verification and validation. The specific input value of 0.25 is chosen to showcase the encoder's behavior with a fractional input within the specified range, providing a representative test scenario. The output is shown in (Fig 19).

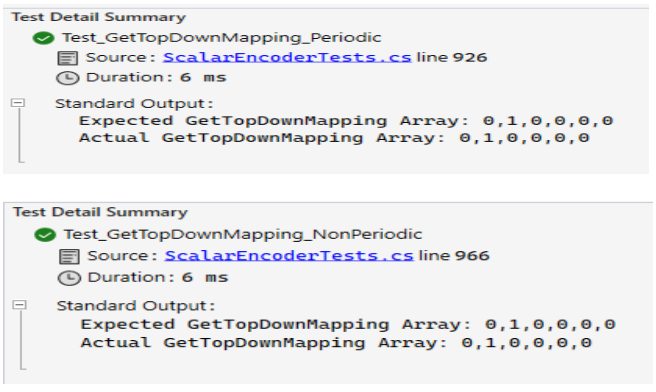


Fig19: Expected Output & Actual Output of the GetTopDownMapping

VI. Applications of Scalar Encoder with Buckets

The Scalar Encoder with Buckets finds versatile applications across various domains, leveraging its unique features and capabilities. Below are notable applications showcasing its adaptability and utility:

Sensor Data Processing:
Utilizing the Scalar Encoder with Buckets proves invaluable in processing sensor data where numerical values need to be efficiently encoded for further analysis. The bucketing mechanism enhances precision and flexibility in handling diverse sensor inputs.

Anomaly Detection:
In anomaly detection systems, the encoder can be employed to convert numerical data into a format suitable for anomaly analysis. The ability to customize bucket ranges and efficiently encode values contributes to the accuracy of anomaly detection algorithms.

Time Series Analysis:
The encoder's periodic encoding feature is particularly beneficial for time series data, such as stock prices or weather patterns. It allows for capturing cyclical patterns and trends, enabling more accurate analysis of temporal data.

Control Systems and Robotics:
Scalar Encoder with Buckets can be applied in control systems and robotics to encode numerical inputs, facilitating precise control and decision-making processes. The enhanced flexibility in defining encoding parameters adds to its utility in these applications.

Natural Language Processing:
In certain natural language processing tasks, numerical values may need to be encoded for computational analysis. The encoder's ability to handle a wide range of numeric inputs and provide detailed bucket information can be advantageous in such scenarios.

Machine Learning Feature Engineering:
Feature engineering is a critical aspect of machine learning. The Scalar Encoder with Buckets can be employed in preparing input features, providing a structured representation of numerical data for machine learning models.

Environmental Monitoring:
When dealing with environmental sensor data, the encoder aids in transforming diverse numeric inputs into a standardized format. This is particularly useful for monitoring and analyzing environmental parameters over time.

Signal Processing:
Scalar Encoder with Buckets can play a role in signal processing applications, where numerical signals need to be encoded and analyzed efficiently. The precision and adaptability of the encoding scheme contribute to better signal representation.

In essence, the Scalar Encoder with Buckets emerges as a versatile tool with wide-ranging applications, showcasing its effectiveness in diverse fields that involve numerical data processing and analysis.
Overall, the scalar encoder with buckets is a versatile technique that may be used in a variety of applications requiring the encoding of continuous data into sparse distributed representations.

VII. Overcoming Limitations

The methods mentioned serve to address several issues inherent in the standard scalar encoder, offering solutions for the scalar encoder with buckets.

Enhanced Precision:
The *BucketMatchScores()* method enhances the precision of the scalar encoder by evaluating proximity scores derived from the difference between expected and actual values. This scoring mechanism facilitates encoding values with superior precision compared to the conventional scalar encoder.

Augmented Flexibility:
The *GenerateNumericRangeDescription()* method provides users with the authority to specify minimum and maximum values, in addition to determining the number of bits allocated for encoding.

This confers an elevated level of control to users over the encoding framework, thereby enhancing overall flexibility.

Periodic Encoding:
The integration of the Periodic option in the *BucketMatchScores()* method introduces the capacity for periodic encoding of values. This proves particularly advantageous for handling values with cyclical patterns, such as angles or temporal data. The feature accommodates scenarios where values exhibit periodicity or wrap around.

Non-Periodic Encoding:
The *BucketMatchScores()* method, without the Periodic option, is adept at encoding values without imposing periodic constraints. This is particularly useful for handling non-cyclical data, ensuring effective encoding without the consideration of wrapping around or cyclical patterns.

Efficient Encoding Scheme: By employing the *EncodedArray()* method, values are transformed into an array of binary digits, optimizing the encoding scheme. This method proves to be more effective than the conventional scalar encoder, which relies on setting one bit high and the rest low for value encoding.

Bucket_Indexing:
The *ScalarEncoderDetermineBucketIndex()* method functions as a valuable tool, mapping values to their respective bucket indices. This functionality is essential for pinpointing the appropriate bucket to encode a given value accurately.

Top-Down Mapping:
Utilizing the *MapInputToBuckets()* method, values are systematically mapped to a range of bucket indices, commencing from the highest point in the hierarchy. This functionality plays a pivotal role in identifying the most general bucket to which a given value belongs.

Comprehensive Bucket Information:
The *GetBucketValue()* and *ScalarEncoderAnalyzeInfo()* methods offer comprehensive information about the buckets utilized in the encoding scheme. This includes details such as boundaries and center points, providing valuable insights for understanding the encoding scheme and streamlining the debugging process.

VIII. Conclusion

The Scalar Encoder with Buckets represents an enhanced iteration of the Scalar Encoder, addressing its limitations. This improved version achieves superior precision in encoding data by mapping input values to continuous ranges of buckets. The automatic parameter setting, including the number of buckets and bucket size based on input data, along with the generation of a descriptive bucket range, enhances the adaptability of the Scalar Encoder with Buckets across diverse datasets. Additionally, its support for periodic encoding enables more effective handling of cyclical data.

The various methods employed in the Scalar Encoder with Buckets, such as EncodedArray, DetermineBucketIndex, BucketMatchScore, Decode, ScalarEncoderAnalyzeinfo, MapInputToBuckets, GetBucketValue, and GenerateNumericRangeDescription, collectively contribute to the encoder's improved performance. Leveraging continuous bucket ranges and an enhanced encoding scheme ensures greater precision in data encoding. The automated parameter configuration and provision of a comprehensive bucket range description further augment the encoder's flexibility. The added capability for periodic encoding enhances its effectiveness in dealing with cyclical data. In conclusion, the Scalar Encoder with Buckets emerges as a potent tool for data encoding, offering a more adaptable and precise alternative compared to the original Scalar Encoder.

IX. References

[1] S. &. H. .. Ahmad, " "Properties of sparse distributed representations and their application to hierarchical temporal memory.,"," 2011. [Online]. Available: doi: 10.1371/journal.pone.0022149.

[2] S. A. J. Hawkins, ""Why neurons have thousands of synapses, a theory of sequence memory in neocortex,"," 2016. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fncir.2016.00023/full.

[3] S. Purdy, "Encoding Data for HTM Systems," [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1602/1602.05925.pdf.

[4] "Scalar Encoders," [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1602/1602.05925.pdf.

[5] S. P. S. Ahmad, "Spatial Pooling with Hierarchical Temporal Memory," [Online]. Available: https://arxiv.org/abs/1705.05363.