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

Investigate and Implement KNN Classifier

Deblina Karmakar
deblina.karmakar@stud.fra-uas.de

Indranil Saha
indranil.saha@stud.fra-uas.de

Riswan Basha Saleem Basha
riswan.saleem-basha2@stud.fra-uas.de

Shivakumar Bailabasappa Veerapur
shivakumar.veerapur@stud.fra-uas.de

Abstract*—Algorithms drive the world of machine learning. They're often praised for their predictive capabilities and spoken of as hard workers that consume huge amounts of data to produce instant results. Among them, there's an algorithm often labeled as lazy, however it's quite a performer when it comes to classifying data points - K-Nearest Neighbor (KNN) algorithm. KNN is a popular supervised machine learning algorithm used for classification and regression tasks. Classification is used to predict the class label of a given input data point, while regression is used to predict the value of a continuous variable for a given input data point. It is also a non-parametric and instance-based algorithm that does not assume any specific probability distribution for data. In KNN, the classification of a new instance is based on the majority class of its K-nearest neighbors in the training dataset. The algorithm relies on finding the K-nearest neighbors of a given data point and assigning a class label based on the majority of those neighbors. In this project, KNN classifier is integrated with Hierarchical temporal memory(HTM) framework to utilize the latter for classification task. Specifically, HTM models are capable of learning to represent input data in a feature space with high dimensionality and consequently KNN classifier has been employed to classify the data by considering the closest neighbors within this feature space. This approach is beneficial where the input data exhibits an intricate temporal structure, requiring a more advanced classification approach. The KNN model, that is being developed is also integrated with the Neocortex API. The input sequence of data is fed into the encoder and further into the spatial pooler to form a Sparse Distributed Representation(SDR) of the data, which is then fed into the KNN model for the final classification. Methodologies followed, challenges faced, and enhancements performed are described in detail in subsequent sections of this paper.

Keywords— Supervised algorithm; K-Nearest Neighbors; Non-parametric; Hierarchical temporal memory(HTM); Temporal structure; Neocortex API; Sparse Distributed Representation(SDR).

I. INTRODUCTION

As a cornerstone in the domain of computer science, machine learning constitutes a pivotal field in it, revolutionizing the way computers comprehend and interpret data. The essence of machine learning lies in the development of statistical models and algorithms that can autonomously learn from input data, providing invaluable insights and predictions. One such supervised machine learning algorithm of ML, predominantly used for classification purposes is K-Nearest Neighbors or KNN.

A study, conducted in 2016, in the field of biology and medicine had showed that researchers used KNN classifier to classify patients as either having heart disease or not based on their medical and clinical features. They experimented with the classifier with different values of K to classify the patients and evaluated the performance of the model using metrics such as accuracy, sensitivity, and specificity. The study found that KNN classifier with K=7 achieved an accuracy of 84.5%, sensitivity of 80.6%, and specificity of 88.4%. The model was also able to identify the most important features for heart disease diagnosis, such as chest pain, electrocardiogram results, and age.

Supervised learning is an approach to creating artificial intelligence where a computer algorithm is trained on input data that has been labeled for a particular output. The model is trained until it can detect the underlying patterns and relationships between the input data and the output labels, enabling it to yield accurate labeling results when presented with never-before-seen data. The aim is to make sense of data within the context of a specific question. Supervised learning is good at classification and regression problems, such as determining what category a news article belongs to or 2predicting the volume of sales for a given future date. Organizations can use supervised learning in processes like anomaly detection, fraud detection, image classification, risk assessment and spam filtering [1].

Unsupervised Machine Learning is a powerful tool for gaining valuable insights from data. Unlike supervised machine learning, it does not require labelled data, but aims to automatically discover patterns, structures, or groupings in the data. Using techniques such as clustering, dimensionality reduction or association analysis, companies can uncover hidden information, gain new insights, and make better decisions. It can help understand customer behavior, detect fraud, identify product segments and much more. An understanding of Unsupervised Machine Learning is therefore important for companies to realize the full potential of their data and gain competitive advantage [2].

In this paper, K-Nearest Neighbor (*KNN*) classifier, which comes under the supervised machine learning algorithm is implemented. The KNN model is being developed primarily, which is afterwards incorporated into the Neocortex API. The Neocortex API, in this project, is utilized in order to integrate HTM cortical Learning Algorithm. HTM models, tailored for learning and recognizing patterns in different time-varying data such as sensor data, audio, and video, serve as the input for the KNN classifier. This collaborative utilization between HTM and KNN holds specific significance in scenarios where input data exhibits intricate temporal structures,

demanding a more sophisticated classification approach beyond conventional threshold-based methods.

II. METHODS

This particular section of the paper covers in detail both the theoretical approach and the practical steps of the project work. It deals majorly with the following things – a thorough literature review, a detailed theoretical background study of KNN and HTM, and a comprehensive coverage of the implementation procedure. In addition to it, the unit testing and the integration process of KNN, along with HTM are also discussed thoroughly.

As part of the project work, detailed research on KNN and its fundamental concept has been conducted, establishing a solid understanding of the working principle of the implemented supervised machine learning algorithm. In addition to it, HTM has been studied in order to have a clear idea of its functionality, resulting in effective collaboration of the same with the developed KNN model.

A. Literature Review

Extensive research on KNN has been carried out by various authors, researchers, and engineers, laying a robust groundwork for understanding the KNN algorithm. Such in-depth investigation significantly aids in the design process of the KNN model's development.

The main objective of the experiment[3] conducted is to utilize the K-Nearest Neighbors (KNN) classifier to categorize unlabeled observations, specifically determining the classification of a sweet potato based on its characteristics. The experiment involves collecting data on crunchiness and sweetness for various labeled examples, such as fruits, vegetables, and grains, and using this information to train the KNN model. By selecting the four nearest labeled examples (apple, green bean, lettuce, and corn) based on the given characteristics, the sweet potato is then assigned to the class with the majority of votes, demonstrating the practical application of the KNN algorithm in classification tasks. It is also noted that average accuracy is the most widely used statistic to reflect the performance of the algorithm. Factors such as K value, distance calculation and choice of appropriate predictors all have significant impact on the model performance.

In the study [4] carried out, classification accuracy (CA) and reduction rate (RR) metrics were employed, where CA represents the average classification accuracy for K values of 1, 3, and 5 in the K-Nearest Neighbors (KNN) model. The experimental outcomes reveal that the proposed KNN model method consistently outperforms C5.0 in 5-fold cross-validation and demonstrates comparable performance to traditional KNN. Notably, the KNN model exhibits a significant enhancement in efficiency by preserving only a limited number of representatives for classification purposes. The results illustrate an average reduction rate of 90.41%. At the end, superior classification accuracy and notable efficiency improvements were achieved through the application of the KNN model method across multiple datasets.

The paper [5] focuses on simplifying the understanding of supervised learning and the KNN classification algorithm through easy examples, emphasizing that supervised learning operates on labelled data. The algorithm is demonstrated using a small student dataset, illustrating its application in predicting the pass or fail outcome for a new student based on marks, with a K size of 3 for easy comprehension.

KNN stands out for its computational intensity, requiring more time compared to other algorithms like Neural Networks, which demand extensive training data for precision. The choice between KNN and SVM depends on the relationship of size between training data and features, with KNN favored when data is larger and SVM when features are more numerous. KNN supports non-linear solutions, in contrast to logistic regression, which only handles linear solutions and is relatively faster. Despite its simplicity, KNN lacks efficiency in complex tasks, as it operates without training, making it faster but requiring proper scaling and not always suitable for solutions demanding more sophisticated models [6].

In a nutshell, the conducted literature review suggests some useful insights into KNN. The main advantages of KNN classifier lies with its adaptability to complex datasets, improved efficiency while maintaining optimum resources and simplicity. Although computational intensity is one unfriendly aspect of the classifier, application of KNN model suits a large spectrum of applications.

B. Theoritical Background - Fundamentals and Design Considerations of K-Nearest Neighbors and its Parameters

KNN algorithm is a versatile and widely used machine learning algorithm that is primarily used for its simplicity and ease of implementation. It does not require any assumptions about the underlying data distribution. It can also handle both numerical and categorical data, making it a flexible choice for various types of datasets in classification and regression tasks. Figure 1(given below) shows the usage of the KNN algorithm in identifying the proper category of the new data point.

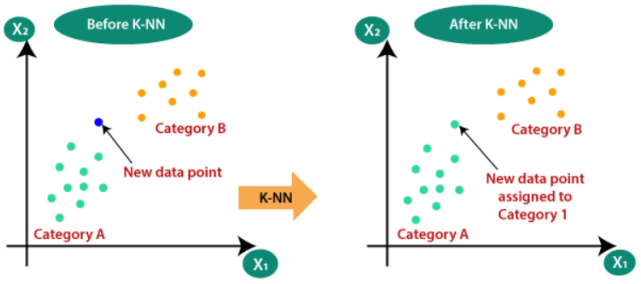

Figure 1: KNN algorithm to determine category of new data point

The working of the KNN can be explained in a rather simpler manner with the below steps –

Step 1 – Select the number K of the neighbors

Step 2 - Calculate the distance of K number of neighbors from the new data point

Step 3 – Take the K nearest neighbors as per the calculated distance

Step 4 - Among these k neighbors, count the number of data points in each category.

Step 5 - Assign the new data points to that category for which the number of the neighbor is maximum, and the model shall be ready.

Underlying this effective and efficient algorithm are some significant parameters as mentioned below –

B.1. Methods of distance calculation:
There are a lot of different distance metrics available, however for the algorithm to work best on a particular dataset, it is important to choose the most appropriate distance metric. Some of the most widely used ones are as below –

Minkowski Distance – It is a metric intended for real-valued vector spaces. We can calculate Minkowski distance only in a space where distances can be represented as a vector that has a length and the lengths cannot be negative. There are a few conditions that the distance metric must satisfy:
Non-negativity: d(x, y) >= 0
Identity: d(x, y) = 0 if and only if x == y
Symmetry: d(x, y) = d(y, x)
Triangle Inequality: d(x, y) + d(y, z) >= d(x, z)

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

The above formula for Minkowski distance is in generalized form. The p value in the formula can be manipulated to give us different distances like:
p = 1, when p is set to 1, we get Manhattan distance
p = 2, when p is set to 2, we get Euclidean distance

Manhattan Distance – This distance is also known as taxicab distance or city block distance, because of the way this distance is calculated. The distance between two points is the sum of the absolute differences of their Cartesian coordinates. As mentioned earlier, the formula for Manhattan distance is achieved by substituting p=1 in the Minkowski distance formula –

$$d = \sum_{i=1}^{n} |x_i - y_i|$$

This distance is preferred over Euclidean distance when we have a case of high dimensionality.

Euclidean Distance – It is a measure of the true straight-line distance between two points in Euclidean space. It can be used by setting the value of p equal to 2 in Minkowski distance metric. Consequently, the distance formula can be provided in the following manner -

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Cosine Distance – This distance metric is used mainly to calculate similarity between two vectors. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in the same direction. Formula for the cosine distance is as follows –

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|}$$

Using this formula, a value can be obtained conveying the similarity between the two vectors. Using this distance, we get values between 0 and 1, where 0 means the vectors are 100% similar to each other and 1 means they are not similar at all. In this project, cosine similarity distance metrics has been utilized for the classification using KNN.

Jaccard Distance – The Jaccard approach [7] looks at two data sets and finds the incident where both values are equal to 1. So, the resulting value reflects how many one-to-one matches occur in comparison to the total number of data points. It is extremely sensitive to small samples sizes and may give erroneous results, especially with very small data sets with missing observations. The formula [8] for Jaccard index is -

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Jaccard distance [9] is the complement of the Jaccard index and can be found by subtracting the Jaccard Index from 100%, thus the formula for Jaccard distance is - D(A,B) = 1 – J(A,B).

Hamming Distance - Hamming distance is a metric for comparing two binary data strings. While comparing two binary strings of equal length, the hamming distance is the number of bit positions in which the two bits are different. The Hamming distance method looks at the whole data and finds when data points are similar or dissimilar one to one. It gives the result of how many attributes were different. This is used mostly when data is one-hot encoded, and it is required to find distances between the two binary vectors. For e.g., if two strings “ABCDE” and “AGDDF” of same length are present and it is required to find the hamming distance between these strings, then letter by letter in each string is checked whether they are similar or not. For example, it verifies like first letters of both strings are similar, then second letters are similar or not and so on. At the end, it is seen that only two letters (in the given example) were similar and three were dissimilar in the given strings. Hence, the Hamming Distance here will be 3. It is to be noted that larger the Hamming Distance between two strings, more dissimilar will be those strings and vice versa.

B.2. Selection of 'K' value:
The choice of the K value in the K-Nearest Neighbors (KNN) classifier is a crucial parameter that significantly influences the model's performance. The value of K represents the number of nearest neighbors considered during the classification process. A smaller K value, such as K=1, leads to a more flexible model that can be sensitive to noise in the data but may overfit. On the other hand, a larger K value, such as K=5 or K=10, provides a smoother decision boundary, offering more robustness to noise. The selection of an optimal K value often involves a trade-off between model sensitivity and generalization. Cross-validation techniques, such as K-fold cross-validation, can be employed to assess the model's performance across different K values and help determine the most suitable one for a given dataset. Experimentation and testing various K values are essential to strike the right balance between bias and variance, ensuring an effective and accurate KNN classification model.

B.3. Softmax algorithm:
The Softmax function, often used in machine learning and deep learning, is a mathematical function that transforms a vector of numerical values into a vector of probabilities, with the probabilities of each value being proportional to the exponent of the original value. This function is particularly useful in classification tasks where we need to determine the probability distribution over a set of potential classes.

For example, given a vector Z of real numbers Z= [z1, z2, z3, …, zn], the softmax function for each element zi of Z is defined as –

$$\sigma(Z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

C. Hierarchical Temporal Memory (HTM)
HTM was developed by Jeff Hawkins, the founder of Numenta, Inc., and his colleagues. Hawkins outlined the foundational concepts of HTM in his book "On Intelligence" (2004). The development of HTM is motivated by the desire to understand the neocortex's workings and to apply these biological principles to computing. The properties of HTM systems are characterized by several key properties as mentioned below –

Hierarchy: HTM structures information in a hierarchical manner, similar to the neocortex. This hierarchy allows the system to understand complex patterns and relationships within the data by processing information at various levels of abstraction.

Temporal Memory: HTM systems have a unique capability to learn and remember sequences of patterns over time, which is critical for understanding temporal contexts and predicting future events.

Sparse Distributed Representations (SDRs): HTM uses SDRs to represent information efficiently. In SDRs, each piece of information is encoded in a way that uses a small percentage of active bits in a large binary vector, providing robustness to noise and enabling the system to handle subtle pattern differences.

Online Learning: HTM systems learn continuously from a stream of data, updating their knowledge without needing to restart the learning process from scratch. This property allows HTM to adapt to changing environments and new information.

The Hierarchical Temporal Memory algorithm is inspired by the neocortex and implements many known features that have roots in neurosciences. Nowadays, many results show that the algorithm is very flexible and can solve different kinds of problems like sequence learning, anomaly detection, object recognition, classification, etc. However, the reverse engineering of the neocortex is still a complex and unsolved task[10].

Hierarchical Temporal Memory (HTM) operates through a biologically inspired process that involves converting data into sparse distributed representations (SDRs) for efficient information encoding. It then engages in pattern recognition to identify spatial relationships within the data. The core of HTM's functionality lies in its ability to learn and remember sequences of patterns over time, incorporating temporal contexts into its processing. This sequential learning enables HTM to make predictions about future events or infer missing information from data streams. Additionally, HTM systems are adept at detecting anomalies by recognizing patterns or sequences that deviate from learned norms. This multi-faceted approach, grounded in principles mimicking the human neocortex, allows HTM to handle complex, time-sensitive data in a robust and adaptive manner, making it particularly suitable for applications involving intricate temporal patterns and prediction tasks. Figure 2 shows the incorporated loops in the architecture of HTM, enabling retention of information. This design feature makes them well-suited for learning from sequences, as they can maintain a memory of previous inputs to inform their understanding and processing of data over time.

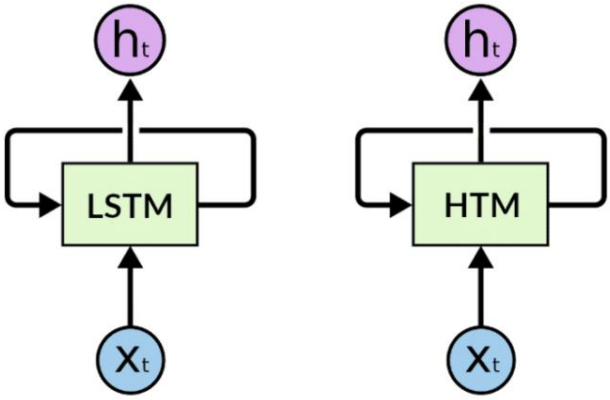

Figure 2: Recurrent Neural Networks as Hierarchical Temporal Memories have loops in them, allowing information to persist and making them suitable for sequence learning problems

D. Implementation

The KNN classifier is used within the experiment to learn sequences and predict future elements based on the learned patterns. KNN is used alongside HTM to aid in predicting values based on the active cells and winners are obtained from HTM components such as spatial pooler and temporal memory. The KNN classifier learns patterns derived from HTM activations and tries to predict the next elements in the sequences. This section of the paper describes in detail the important methods of the code and the implemented logic in the project experiment.

D.1 Integration of model with Neocortex API

The integration between the Neocortex API and the new KNN model involves designing the KNN model to work cohesively with the Neocortex API. In this integration, the Neocortex API provides a continuous stream of sequences as input, while the KNN model predicts predefined labels for the output data sequence. Subsequently, unit test cases are developed to evaluate the robustness and accuracy of the integrated model. The implementation of NuPIC's KNN Classifier for Sparsely Distributed Represented (SDR) datasets has significantly influenced the architecture of this integrated KNN model. As discussed in the theoretical background, these algorithms encompass distance computation and weightage calculation. Therefore, the design of the algorithm incorporates class-based generic containers responsible for storing distance, classification parameters.

In line with the Neocortex API requirements, two essential methods from the IClassifier interface are implemented. The initial method facilitates the model's learning process from the dataset, while the second method retrieves the N values determined by the classifier. Subsequently, all the significant methods and functionalities are discussed in alignment with their application with the code.

D.1.1 Learn

public void Learn (TIN input, Cell[] cells)

The Learn method is a crucial component of the K-nearest neighbors (KNN) classifier, serving to enhance the model's understanding of new input patterns. Initially, it extracts the classification label corresponding to the provided input, utilizing the ***GetClassificationFromDictionary*** method[11]. Subsequently, the method transforms the array of Cell objects (cells) into an array of integer indices, capturing the specific indices associated with each cell. The final step involves updating the internal models of the KNN classifier through the ***UpdateModels*** function. This function, in turn, ensures that the model incorporates the latest information by associating the obtained classification label with the array of cell indices.

D.1.2 ClassificationAndDistance

The ***ClassificationAndDistance*** class is integral to machine learning algorithms like K-Nearest Neighbors (KNN), where it bridges classification labels with their respective distance measures. As a utility class, it supports critical functionalities such as storing the computed distances between a query input and the training dataset, which is pivotal for identifying the nearest neighbors in the classification process. The inclusion of the ***IComparable<ClassificationAndDistance>*** interface allows these instances to be compared and sorted based on the distance metric, streamlining the process of organizing training samples by proximity to enhance the accuracy and efficiency of the KNN classifier.

Furthermore, the ***ClassificationAndDistance*** class facilitates the pairing of distance metrics with classification labels, encapsulating both pieces of information in a singular, manageable object. The ***CompareTo*** method in the ***ClassificationAndDistance*** class is marked as public to implement the ***IComparable<ClassificationAndDistance>*** interface. The purpose of making it public is to allow instances of the ***ClassificationAndDistance*** class to be compared and sorted based on their distance values outside of the class.

While it's true that the ***CompareTo*** method is not explicitly called within the ***ClassificationAndDistance*** class , its public visibility is essential for cases where instances of this class are used in sorting operations or comparisons in other parts of the code.

In the subsequent section, the prediction behavior of the model is discussed. It is dependent on the value of the 'useSoftmax' argument, defined in the instance of the KNNClassifier. If the value of the argument is true, then the ***PredictWithSoftmax*** method will be returned as the output of ***GetPredictedInputValues*** or else it will be returned as the output of ***SelectBestClassification*** method.

D.1.3 GetPredictedInputValues

public List<ClassifierResult<TIN>> GetPredictedInputValues(Cell[] unclassifiedCells, short howMany = 1)

The ***GetPredictedInputValues*** method[11] in the KNN classifier is responsible for predicting input values based on the distances between unclassified cells and the stored sequences in the model. To begin with, it checks if there are any unclassified cells, and if not, it returns an empty list of ClassifierResult<TIN>

```
if (unclassifiedCells.Length == 0)
    return new List<ClassifierResult<TIN>>();
```

The method then extracts the indices of the unclassified cells and initializes a DefaultDictionary named mappedElements to store distances and classifications[11] as evident from the below code snippet.

```
var unclassifiedSequences = unclassifiedCells.Select(cell
=> cell.Index).ToArray();
var mappedElements = new DefaultDictionary<int,
List<ClassificationAndDistance>>();
```

As discussed earlier, depending on the value of the useSoftmax argument, ***GetPredictedInputValues*** method

utilizes either the ***PredictWithSoftmax*** approach or the ***SelectBestClassification*** method. While the value of the argument is true, ***PredictWithSoftmax*** is called upon and the value is returned employing methods such as ***ComputeCosineSimilarity***, ***CalculateSoftmaxWeights*** among others.

D.1.4 PredictWithSoftmax

Public List<ClassifierResult<TIN>> PredictWithSoftmax(Cell[] unclassifiedCells, short howMany = 1)

PredictWithSoftmax method utilizes the softmax function to predict the class of unclassified data points or cells, considering the probabilities of each class. Initially, if there are no unclassified cells, it returns an empty list, indicating that no predictions are needed. The method then extracts the indices of unclassified cells and calculates the distances between these cells and the models (classified sequences) using a modified distance calculation method tailored for cosine similarity, ***GetDistanceTableforCosine***.

```
var distanceTable = GetDistanceTableforCosine(sequence, unclassifiedSequences);

foreach (var kvp in distanceTable)
{
   if (!mappedElements.ContainsKey(kvp.Key))
   {
      mappedElements[kvp.Key] = new List<ClassificationAndDistance>();
   }
```

Following this, the distances in mappedElements are sorted to ensure consistent ordering. Subsequently, Softmax weights are computed for each class based on the distances stored in mappedElements using the ***CalculateSoftmaxWeights*** method[11], introducing a softness parameter of 0.5. The Softmax function is then applied to these weights through the Softmax method, generating probabilities for each class.

```
var softmaxWeights = CalculateSoftmaxWeights(mappedElements, 0.5);
var softmaxProbabilities = Softmax(softmaxWeights);
```

The method creates a list of ClassifierResult<TIN> instances, where each instance contains the predicted input, calculated similarity (using Softmax probability as the similarity score), and any additional relevant information. Finally, the list is limited to the specified number (howMany) of best predictions, and the resulting predictions are returned by the method.

D.1.5 ComputeCosineSimilarity

private double ComputeCosineSimilarity(HashSet<int> classifiedSet, HashSet<int> unclassifiedSet)

The ***ComputeCosineSimilarity*** method[11] calculates the cosine similarity between two sets represented by HashSet<int> instances - classifiedSet and unclassifiedSet. It starts by computing the dot product of the two sets, i.e., the count of common elements. Then, it determines the lengths of both sets. To avoid division by zero, it checks for edge cases where either set has zero length and returns 0.0 in such situations. Finally, it calculates the cosine similarity using the dot product and the lengths of the sets, providing a measure of similarity between the classified and unclassified sets. The result is a double value representing the cosine similarity, with 1.0 indicating perfect similarity and 0.0 indicating no similarity.

On the other hand, While the value of the argument is false, ***SelectBestClassification*** is being utilized and the value is returned employing methods such as ***GetDistanceTable*** computing the ***LeastValue*** among others.

D.1.6 SelectBestClassification

Public List<ClassifierResult<TIN>> SelectBestClassification(Dictionary<int, List<ClassificationAndDistance>> mapping, int howMany, int numberOfNeighbors)

The ***SelectBestClassification*** method[11] operates within a classifier system, tasked with identifying the most appropriate classifications for a set of unclassified data points, drawing on previously classified data and the relationships between them. At its core, this method incorporates both distance metrics and the concept of overlaps - instances where classified and unclassified data points match exactly. To begin, the method initializes various dictionaries to track weighted votes, overlaps, and final similarity scores for each potential classification. These initial steps ensure a comprehensive assessment framework that considers every possible classification equally.

The method then iterates through the provided mapping, which contains distance information between unclassified sequences and stored sequences from various models. For each set of coordinates in mapping, it selects the top 'numberOfNeighbors' neighbors and calculates votes based on distances. If the distance is zero, indicating an exact match, the corresponding overlap count is incremented, otherwise, weighted votes are computed inversely proportional to the distance.

```
    if (value.Distance == 0)
    {
       overlaps[value.Classification]++;
    }
    else
    {
weightedVotes[value.Classification]+=1.0/value.Distance;
    }
```

Following the vote calculation, the method proceeds to normalize the weighted votes to ensure a balanced contribution from each classification. Normalization is performed by dividing each weighted vote by the maximum weighted vote. Additionally, similarity scores based on overlaps are computed, representing the proportion of instances where exact matches occurred.

Finally, the normalized weighted votes and overlap scores are combined to yield the overall similarity scores. The method

orders these scores in descending order to facilitate the final decision-making process. The results are then transformed into a list of ClassifierResult<TIN> instances, each containing the predicted input, similarity score, and the count of overlapping bits. The final list is limited to the specified number (howMany) of best predictions, which is then returned by the method.

D.1.7 GetDistanceTable

private Dictionary<int, List<ClassificationAndDistance>> GetDistanceTable(int[] classifiedSequence, int[] unclassifiedSequence)

The ***GetDistanceTable*** method[11] calculates distances between a classified sequence and an unclassified sequence. For each index in the unclassified sequence, it computes the shortest distance using the ***LeastValue*** method.

```
    foreach (var unclassifiedIdx in unclassifiedSequence) {
        var shortestDistance = LeastValue(classifiedSequence, unclassifiedIdx);
```

The distances are stored in a dictionary where each key represents an index from the unclassified sequence, and the associated value is a list of ClassificationAndDistance objects containing the classification label ("Classification") and the respective shortest distance. The method then returns this distance table.

D.1.8 RunMultiSequenceLearningExperimentWithImage

private static void RunMultiSequenceLearningExperimentWithImage()

The presented method encapsulates the initiation of a multi-sequence learning experiment utilizing image data, primarily focused on MNIST images at the current stage. Figure 3 below shows such an input MNIST image. By locating the target directory through a systematic search within the application's directory structure, the method then identifies and accesses the designated folder containing the input images. Upon successful identification, the method proceeds to convert the images into pixel sequences, storing them in a list for further processing. It iterates through each image file, utilizing the ConvertImageToSequence function to transform the images into lists of pixel values, subsequently compiled into arrays and added to the overall collection of pixel sequences.

Figure 3: Input MINST image 'a'

The method then displays the extracted pixel values for each image sequence to facilitate inspection. Following this data preparation phase, an instance of the MultiSequenceLearning class is instantiated to perform the sequence learning experiment. The image pixel sequences are structured into a dictionary format as required by the experiment's Run method, facilitating the execution of the learning process. Finally, the method concludes by resetting the predictor instance to prepare for subsequent experiments or further analyses within the context of MNIST image sequence learning.

D.1.9 ConvertImageToSequence

private static List<double> ConvertImageToSequence(string imagePath)

The Program.cs file includes a method named **ConvertImageToSequence**. This method is designed to convert an image file into a sequence of grayscale pixel values. It first loads the specified image using the System.Drawing.Bitmap class and locks its data to read the pixel values efficiently. Subsequently, it iterates through each pixel in the image, extracting the red, green, and blue (RGB) colour values, and computes the average grayscale value by taking the average of these RGB values. The resulting grayscale value is then appended to a list representing the sequence of pixel values. Finally, after processing all pixels, the method releases the locked image data and returns the generated pixel sequence as a List<double>. This method employs unsafe code blocks for pointer manipulation to access pixel data directly, optimizing the conversion process.

E. Unit Testing:

Unit testing plays a crucial role in any software development by enabling developers to verify the functionality of individual components or units of code in isolation. It ensures that each part of the code performs as expected, leading to higher quality of software, facilitating easier maintenance, and refactoring.

In this project work, the KNN algorithm was rigorously tested using test cases inspired by the HTM Classifier within the Neocortex API framework. During unit testing, the model exhibited highly encouraging outcomes, covering most of the methods and functionalities implemented with the means of the test cases written. This impressive performance, evident from the results shown in subsequent section, underscores the reliability and robustness of the designed KNN algorithm, showcasing its capability to effectively predict sequences within the experimental setup.

F. Integration of KNN with HTM

The experiment showcased in the code integrating Hierarchical Temporal Memory (HTM) with a K-nearest neighbors (KNN) classifier to facilitate sequence learning. Within the MultiSequenceLearning class, this implementation involves leveraging HTM functionalities, including CortexLayer, TemporalMemory, and related classes from the NeoCortexApi namespace, along with the KNeighborsClassifier class from NeoCortexApi.Classifiers for the KNN.

The process initiates by configuring HTM parameters and setting up the encoder (ScalarEncoder) to transform scalar values into sparse distributed representations (SDRs). The experiment execution comprises several key steps: the Run method commences the sequence learning experiment, initializing HTM components (SpatialPooler, TemporalMemory), and the KNN classifier. It progresses by initially training the Spatial Pooler (SP) without the Temporal Memory (TM) until stability is achieved, followed by training both SP and TM with given sequences to learn patterns.

Notably, the KNN classifier (KNeighborsClassifier) is incorporated alongside HTM within this experiment to learn sequences and predict future elements based on acquired patterns. It operates by utilizing the active and winner cells obtained from HTM components (SP and TM), striving to predict subsequent elements in sequences based on HTM activations. The integration of KNN with HTM aims to complement HTM's sequence learning capabilities by offering an additional mechanism for predicting future elements derived from learned patterns.

The combined approach between HTM and the KNN classifier holds potential advantages in enhancing prediction accuracy and robustness. The integration leverages the strengths of both algorithms; while HTM excels in temporal sequence learning, KNN may capture patterns that HTM might overlook and vice versa. Overall, this integration seeks to improve the accuracy and predictive capabilities of the system when handling sequential data by amalgamating the strengths of HTM and KNN.

III. Results

The utilization of the K Nearest Neighbors (KNN) model within the NeoCortexApi library has proven to be instrumental in facilitating sequence learning and prediction. The model's consistent attainment of an impressive 85.71% accuracy rate across various integer input data sequences attests to its robustness and reliability. Furthermore, in the realm of image classification, the model showcases commendable proficiency, achieving a notable 66% accuracy in processing pixel sequences. This substantiates its competence in complex sequence analysis and underscores its potential in diverse data domains.

Throughout the experiment, the training of the Spatial Pooler (SP) and Temporal Memory (TM) with predefined input patterns facilitated effective pattern recognition and sequence learning. The incorporation of Homeostatic Plasticity Control ensured model stability, guaranteeing a robust initial implementation without encountering failed tests during unit testing. Despite achieving remarkable accuracy, the iterative process highlighted the necessity for further refinement and fine-tuning to elevate the model's reliability and scalability across a broader spectrum of data sequences.

This experiment underscores the imperative of addressing nuanced special conditions that may arise during model development, emphasizing the continuous pursuit of a more dependable and adaptable model. The findings not only reinforce the model's potential but also shed light on the persistent need for ongoing improvements to bolster its performance across diverse data sets, affirming the iterative nature of model development and the quest for enhanced accuracy and adaptability. Figure 4 and figure 5 showcase the predicted output of the KNN model along with the accuracy result for the same.

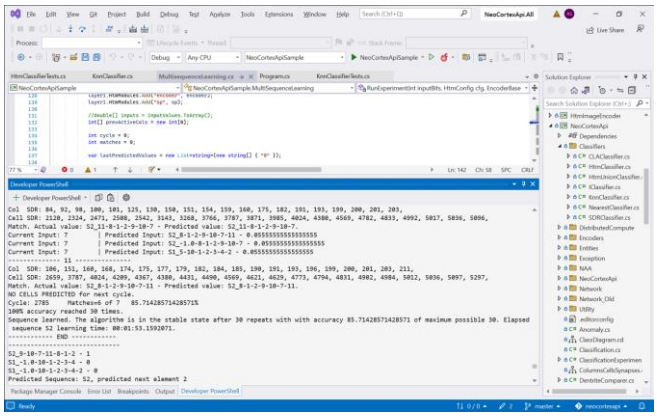

Figure 4: KNN Classification Output with Integers

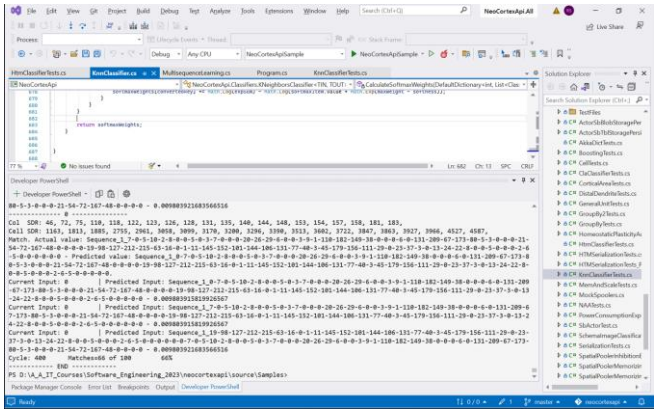

Figure 5: KNN Classification Output with Image sequence

The unit test cases rigorously scrutinize the K-nearest neighbors (KNN) classifier functionalities integrated within the NeoCortexApi library. Covering a spectrum of scenarios, these tests meticulously assess the classifier's behavior across diverse conditions. The test suite encompasses pivotal functionalities, scrutinizing the correctness of predicted input values, the classifier's response to unclassified sequences, and the precise initialization of classifier parameters. It meticulously evaluates the classifier's capability to accurately return the anticipated number of predictions, handle empty scenarios seamlessly, and effectively manage sequences with varying numbers of predicted values.

Furthermore, the test suite, as shown below in figure 6, intricately examines specific cases, delving into the classifier's behavior when dealing with the unclassified index within or outside the classified sequence. It also rigorously evaluates the classifier's proficiency in handling negative values. These comprehensive test cases serve as a robust validation mechanism, affirming the KNN classifier's

reliability, accuracy, and resilience in both learning from and predicting sequences within the NeoCortexApi framework.

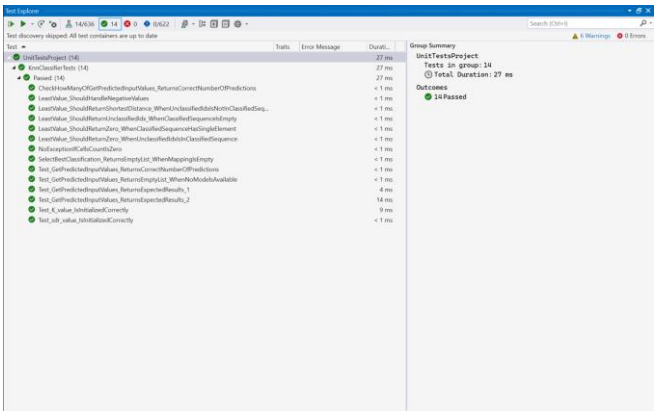

Figure 6: Unit Testing of the KNN Algorithm

IV. DISCUSSION

In this section, The KNN model designing challenges and some of the improvements that can be made to get better accuracy of the model are discussed.

A. Design Challenges:

i. Selection of K-Value:

The K-value selection is pivotal in K-nearest neighbors (KNN). Exploring different K-values and employing cross-validation techniques could enhance the model's reliability and performance by finding an optimal K-value that balances bias and variance.

ii. Distance Metric:

The choice of distance calculation matrices (e.g., Euclidean, Manhattan, Minkowski, cosine) significantly impacts KNN's effectiveness. Experimenting with diverse distance metrics can contribute to building a more stable and adaptable model.

B. Methods to Enhance Accuracy:

Apart from standard practices like normalization, standardization, and scaling, the following methods can potentially enhance the KNN algorithm's performance:

i. Algorithm Variants:

Implementation of KNN algorithm variants such as weighted KNN, instance-based KNN, and kernel-based KNN can assess their efficacy on specific datasets, potentially improving predictions by leveraging different algorithmic approaches.

ii. Ensemble Techniques:

Employing ensemble techniques like Bagging or Boosting by combining multiple KNN models may further enhance accuracy. Ensemble methods often mitigate overfitting and improve generalization by aggregating predictions from multiple models.

C. Handling Image Sequences and Limitations:

When dealing with image sequences sourced from datasets like MNIST, representing a numeral '2' through a sequence of pixels is a useful approach for encapsulating essential handwritten attributes. This method allows the model to capture distinctive patterns inherent in the numeral '2', aiding in its recognition.

i. Varied Writing Styles:

Handwritten digits, such as '2', can exhibit diverse writing styles. Individuals may write the digit '2' with variations in angles, thickness of strokes, or even slight positional differences within the image. These variations result in different pixel sequences, potentially making it challenging for the model to consistently recognize all variations as representing the same numeral.

ii. Stroke Thickness and Positioning:

Differences in stroke thickness or the positioning of the numeral '2' within the image can significantly impact the pixel sequences generated. Even though these variations might represent the same digit conceptually, the resulting sequences could differ substantially, posing a challenge for accurate classification.

iii. Potential Misinterpretation:

Due to the diversity in pixel sequences resulting from variations in writing styles, stroke thickness, or positioning, the model might encounter difficulties in correctly identifying and associating similar pixel patterns. As a result, distinct variations of the numeral '2' might not be appropriately recognized as representing the same digit, potentially affecting the model's classification accuracy.

D. Mitigation Strategies:

To mitigate these limitations, model enhancements or pre-processing steps might be considered. Techniques such as data augmentation, normalization, or transformational adjustments could help standardize the pixel sequences, making the model more robust to variations in writing styles or stroke thickness. Additionally, employing advanced algorithms capable of handling diverse representations or exploring ensemble methods could further improve the model's accuracy when dealing with varied handwritten styles.

V. CONCLUSION

Our journey embarked with the creation of a versatile K-nearest neighbors (KNN) prototype, laying the groundwork for an optimized model that surpassed initial expectations. Leveraging the synergy between the Neocortex API and our refined KNN model, we seamlessly integrated and harnessed the combined power of cosine similarity and SoftMax

functions. This fusion facilitated the proficient categorization of output sequences, effectively distinguishing matches and mismatches against input sequences.

Demonstrating resilience and adaptability, our model consistently achieves an impressive 85.71% accuracy rate when processing diverse normal integer input data sequences. Extending its capabilities to image classification, our model exhibits a commendable 66% accuracy in handling pixel sequences, showcasing its versatility across varied data domains.

Our rigorous approach encompassed comprehensive unit testing, including specialized cases, ensuring consistent and satisfactory outcomes akin to those achieved by the KNN Classifier. This holistic approach, amalgamating cosine similarity and SoftMax functions, underscores the robustness and reliability of our model in predictive tasks, spanning conventional integer sequences to image-based pixel sequences.

The successful fusion of cutting-edge techniques within our model, coupled with its consistent performance, positions it as a reliable and adaptable solution, capable of addressing multifaceted real-world challenges in sequence-based prediction systems.

VI. REFERENCES

[1]Alexander S. Gillis, supervised learning, https://www.techtarget.com/searchenterpriseai/definition/supervised-learning

[2]Patrick, Unsupervised Learning, https://www.alexanderthamm.com/en/blog/this-is-how-unsupervised-machine-learning-works, 12 May 2023

[3]Zhongheng Zhang, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4916348/, June, 2016

[4]Gongde Guo, Hui Wang, David A. Bell, Yaxin Bi, KNN Model-Based Approach in Classification, https://www.researchgate.net/publication/2948052_KNN_Model-Based_Approach_in_Classification, August 2004

[5]Manish Suyal, Piyush Goyal, A Review on Analysis of K-Nearest Neighbor Classification Machine Learning Algorithms based on Supervised learning, Department of CA & IT, SGRR University, Uttarakhand, India, 18 July 2022

[6]Kashvi Taunk; Sanjukta De; Srishti Verma; Aleena Swetapadma, A Brief Review of Nearest Neighbor Algorithm for Learning and Classification, IEEE publications, 16 April 2020

[7]Agresti A, Categorical Data Analysis. John Wiley and Sons, New York, 1990.

[8]Dodge Y, The Concise Encyclopedia of Statistics. Springer, 2008.

[9]Wheelan, C, Naked Statistics. W. W. Norton & Company, 2014

[10] Damir Dobric, Andreas Pech, Bogdan Ghita, Thomas Wennekers, https://link.springer.com/epdf/10.1007/s42979-022-01066-4?sharing_token=yTMMuMLBJcnQyscwzydpave4RwlQNchNByi7wbcMAY4EFDzmnLsjwwWH8OOZoaGew5TrV2QuIwEWhxOXvyyRUPkxKWny6O3F8mTAy1zl_jSsMM7L49uhljaybjTuRgV9iN2dxNy2qFheXRPNqtnEIRQZ4niQGVqnx_TXPRM76vI%3D, 3 March 2022

[11]https://github.com/IndranilSaha09/neocortexapi/blob/master/source/NeoCortexApi/Classifiers/KnnClassifier.cs