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

Implement the KNN Classifier

Kashif Hussain
kashif.hussain@stud.fra-uas.de

Naila Shaheen
naila.shaheen@stud.fra-uas.de

Fahad Jabbar
fahad.jabbar@stud.fra-uas.de

Abstract*— *This paper presents a K-Nearest Neighbors (KNN) classifier designed for the Neocortex API; a software toolkit inspired by neocortical information processing. This approach deviates from conventional KNN implementations by leveraging Neocortex API 's capabilities for processing sequential data, similar to the neocortex. The proposed method integrates spatial pooling for dimensionality reduction and hierarchical temporal memory (HTM) for temporal pattern recognition, enabling efficient classification of sequential data streams. This work bridges the gap between machine learning and neuromorphic computing by introducing a KNN classifier within a neocortex-inspired framework, offering a novel perspective on KNN classification and paving the way for further exploration at the intersection of these fields. The study unfolds in three distinct phases: initially, a native implementation of the KNN classifier undergoes rigorous unit testing to assess its performance on unseen data. Subsequently, the integration of KNN with the Neocortex API is executed to enhance sequence learning and classification capabilities. Finally, KNN is applied to Sparse Distributed Representations (SDRs) generated by the Spatial Pooler for sequential classification tasks. The experimental results showcase a notable 90% accuracy rate across five-fold data training and testing phases. These outcomes not only hold promise but also validate the model's effectiveness in accurately classifying unseen data instances. This research significantly contributes to advancing the understanding of KNN classification within a neocortex-inspired context, highlighting its potential for robust sequential data analysis and classification tasks.

Keywords—Machine Learning; K-Nearest Neighbors; Neocortex API; Hierarchical temporal memory.

I. Introduction

Our brains are remarkable pattern recognition machines, and machine learning draws inspiration from this biological marvel. Traditional classification algorithms often treat data points as isolated entities. This is unlike the human neocortex, where interconnected neurons process information in a hierarchical fashion. The neocortex excels at recognizing not just individual features, but also complex relationships between them.

HTM seeks to emulate the structure and function of the neocortex by employing a layered architecture. The Spatial Pooler, akin to initial cortical processing, extracts feature from sensory data, representing them sparsely. This mirrors the neocortex's ability to identify essential patterns, such as edges or shapes, while disregarding irrelevant details. However, the neocortex doesn't stop at feature extraction; it integrates these features over time to build a coherent understanding. HTM's Temporal Memory serves this purpose by analyzing sequences of representations, allowing the system to learn temporal relationships and predict future patterns. Unlike traditional classification algorithms, HTM comprehends context and sequence, offering a deeper understanding of data beyond mere categorization. Our goal is to create a reliable KNN algorithm and integrate it with the Neocortex API. This API is specifically built to work with HTM. It excels at understanding patterns in dynamic datasets, such as those found in environmental sensors, multimedia, and time-series data. Combining HTM with KNN is especially useful when dealing with data that has complex temporal structures, where traditional classification methods fall short.

In Supervised and Unsupervised Machine Learning, Supervised learning entails the utilization of labeled data to establish a mapping function that correlates input features x with the output variable y, represented as $y = f(X,\theta)$. Through this process, supervised learning aims to discern patterns and relationships within the data to facilitate accurate predictions or classifications. It also involves the training a model to predict discrete or categorical output variables y. On the other hand, unsupervised learning operates on unlabeled data, seeking to uncover inherent structures and patterns without predefined output variables. Unsupervised learning techniques, including clustering, anomaly detection, and latent variable mixture models, are pivotal in extracting meaningful insights from unannotated datasets [1].

Classification is a machine learning technique used to predict group membership for data instances. It involves assigning predefined categories or classes to input data based on their features or attributes. The primary objective of classification is to develop models that can accurately distinguish between different classes within a dataset, enabling automated decision-making and pattern recognition. To achieve this, classification algorithms analyze the characteristics of labeled training data to learn patterns and relationships that can be generalized to classify new, unseen instances. These algorithms employ various approaches, such as statistical methods, decision trees, support vector machines, and neural networks, each with its own advantages and suitability for different types of data and applications. For instance, in a classification scenario, the objective is to assign a given observation to one of several predefined classes $\{C_1, C_2, \dots, C_k\}$, based on learned patterns from the labeled data [2].

Regression is a machine learning technique used to predict continuous numerical values based on input features. Unlike classification, where the goal is to classify data into predefined categories, regression aims to estimate the relationship between independent variables and a dependent variable. This relationship is typically represented by a mathematical function that predicts the value of the dependent variable given the values of the independent variables. Regression models are trained using labeled data, where the target variable is continuous and can take any value within a range. Common types of regression include linear regression, polynomial regression, ridge regression, and lasso regression, each suited to different types of data and modeling scenarios. Linear regression, for example, assumes a linear relationship between the independent and dependent variables, while polynomial regression allows for more complex, nonlinear relationships. For instance, regression models may be employed to predict phenomena like CO2 emissions on specific dates, leveraging historical data to infer future trends and patterns [3].

In K-nearest neighbor (KNN) technique, nearest neighbor is measured with respect to value of k, which define how many nearest neighbors need to examine to describe the class of a sample data point. To balance the overfitting and underfitting, parameterization of the k value is essential to the KNN, because of its adaptability, practitioners can customize the model to fit certain datasets and classification needs. Additionally, the instance-based learning nature of KNN, where classification decisions are made locally, makes it well-suited for scenarios with large or dynamic datasets [4]. Its success relies on selecting the best distance metric, distance is calculated between sample points and all training points and the point with smallest distance is known as nearest neighbor. Despite its simplicity, it's robust and versatile, making it a popular choice for classification across diverse domains. Mostly it relies on Euclidean distances as a default metric [5].

II. Literature Review

In this section, we'll take a closer look at the k-nearest neighbor (KNN) algorithm, exploring its workings and applications. Our aim is to understand KNN, and it's tasks like pattern recognition and classification. We will examine KNN from numerous study perspectives and analyze the integration of HTM with KNN in detail, offering comprehensive insights into the procedure.

For classification purposes, a class label is determined by a majority vote, where the label most frequently observed among neighboring data points is assigned. This terminology distinction arises because "majority voting" necessitates a majority of over 50% of the vote, typically applicable in binary classification scenarios. In cases involving multiple classes, such as four categories, reaching a 50% threshold may not be essential for class determination; assigning a label based on a vote exceeding 25% is sufficient [6].

The study by Shichao Zhang [7] on K-Nearest Neighbor (KNN) classification shows that determining the optimal value for K and conducting nearest neighbor queries are critical tasks. Regarding the nearest neighbor, various distance measurement functions like Euclidean distance, Mahalanobis distance, Manhattan distance, and angle cosine distance are employed. However, the method for calculating K value remains a challenge. Currently, it's predominantly achieved through expert settings or cross-validation techniques. However, these methods do not address the inherent issue of K value calculation because all test data share the same K value. Selecting a small K value can lead to overfitting, while opting for a large K value may increase the approximation error of learning. To address this, Zhang et al. proposed reconstructing the relation matrix and introduced a method for optimal K value computation, aiming to mitigate these challenges.

The research by Sun and Huang [8] introduced an adaptive K-nearest neighbor algorithm, which operates by initially computing the optimal K value for each training data point based on the Euclidean distance metric. Subsequently, it identifies the nearest neighbor of the test data and employs the optimal K value of this nearest neighbor as the optimal K value for the test data. Ultimately, the KNN classification process is executed following the majority rule principle.

Liu et al. [9] introduced a weighted KNN algorithm that adapts the K value for each test data point based on local characteristics derived from surrounding training data. This dynamic approach enables the algorithm to effectively adjust to varying densities and complexities within the feature space. Additionally, the algorithm computes weights for neighboring data points relative to their distances, providing a mechanism to emphasize the influence of nearby instances while mitigating the impact of outliers. Subsequently, classification is performed using the majority rule principle, considering the weighted contributions of neighboring points. This adaptive and weighted approach enhances the robustness and accuracy of KNN classification, particularly in scenarios where data distribution is non-uniform or where optimal K values vary across different regions of the feature space.

In the era of modern healthcare, the analysis of medical images, such as MRI brain scans, has become a crucial task for diagnosing diseases and monitoring patient health. However, interpreting these images manually is time-consuming and prone to errors. To address this challenge, researchers are exploring the integration of KNN algorithms with artificial neural networks (ANNs). ANNs, particularly adept at recognizing patterns in complex data like MRI images, offer a promising approach to automate image analysis tasks. The speed and accuracy of identifying conditions from MRI scans can be increased by utilizing the combined capabilities of KNN and ANNs, which will eventually improve patient outcomes and healthcare efficiency. With advancements in algorithmic techniques, KNN is increasingly recognized as a valuable tool for medical image classification and analysis [10].

Recent research emphasizes the importance of selecting an optimal K value for accurate classification, with adaptive techniques proposed to determine K based on local data characteristics. Approaches such as weighted KNN and adaptive algorithms enhance classification robustness by adjusting to varying data complexities. Overall, advancements in KNN techniques underscore its significance as a valuable tool for pattern recognition and classification tasks across different domains, contributing to improved accuracy and efficiency in diverse applications.

III. Methods

The KNN algorithm, it uses training data and a predetermined k value to identify the k closest data points through distance calculations. When these k data points belong to different classes, the algorithm predicts the class of the unknown data to align with the majority class. This approach, which is based on statistical principles, aids in making informed decisions based on the collective attributes of neighboring data points. The overall process of KNN working principle is explained below.

The performance of k-NN is prejudiced by the selection function of distance [11]. Euclidean distance most frequently used for metrics of distance [11], Cover and Hart [12] suggested the numerous functions of distance which used to find the optimal spectrum. For high accuracy of the classifier, distance function must be carefully collected because it records the real behaviour. Here, we evaluate the importance of Minkowski, Manhattan, Euclidean and Mahalanobis distance functions applied to K-NN algorithms.

1. Distance Metrics

Distance metrics (DM) are a fundamental component of the KNN algorithm, essential for determining the similarity between data points in classification tasks. KNN operates by calculating distances between a new data point and existing data points in the training dataset. This process entails evaluating the dissimilarity or similarity using various distance metrics, such as Minkowski distance, Mahalanobis distance, Euclidean distance, Manhattan distance, and cosine similarity [13]

a) Minkowski distance function

Minkowski distance [14] function is a vector space. It measures the geometric distance among two inputs using p, which is a variable scaling factor written in (1).

$$MK\ (X,Y) = \sqrt[p]{\sum_{j=1}^{n} |X_j - Y_j|^p} \qquad (1)$$

It can be deployed to guess the distance between two points by using diverse values of scaling factor p. For instance, Manhattan distance is obtained when p=1; Euclidean distance is obtained when p=2.

b) Mahalanobis distance function

The Mahalanobis metrics, [15] DM can be defined as the distance between an observation and the center of each data group in an n-dimensional space defined by n variables and their covariance. For some classifications, it is verified more productive [16]. The problem of scale and correlation associated with EUD function has been suitably talked by Mahalanobis distance. It does not depend on the unit of measurements of the dataset; it finds the distance between two groups [17]. It varies from EUD in that it considers the correlations of the data set and is scale-invariant [18]. It is also called statistical distance; hence it is a comparative benefit. Let we have two groups with means X_j and X_k, DM, is given by (2).

$$DM\ (\overline{X}_j,\ \overline{Y}_k) = \sqrt{(\overline{X}_j,\ \overline{Y}_k)^T\ S^{-1}\ (\overline{X}_j,\ \overline{Y}_k)} \qquad (2)$$

Where is S^{-1} a converse pooled covariance matrix S among $\overline{X}_j,\ and\ \overline{Y}_k$. This matrix is figured by using a weighted average of covariance matrices of two groups shown in (2).

c) Euclidean distance function

Euclidean distance (EUD), [19] a core concept in mathematics, measures the straight-line distance between two points in space. It's calculated as the square root of the sum of the squared differences between corresponding coordinates. In machine learning, Euclidean distance is frequently used in algorithms like K-nearest neighbors (KNN) to determine the similarity between data points, aiding in tasks such as classification and clustering [20].

$$d(P,Q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \qquad (3)$$

Where: P = $p_1, p_2,$ ……, p_n represent the coordinates of point P, Q = $q_1, q_2,$ ……, q_n represent the coordinates of point Q and, n denotes the number of dimensions in the space.

d) Manhanttan distance function

Manhattan Distance calculates the distance between two points by summing the absolute differences of their coordinates. Unlike Euclidean Distance, it evaluates movement along grid-like paths, akin to navigating city blocks in a taxi. This distance is particularly useful in scenarios where movement is constrained to horizontal and vertical paths, such as grid-based environments or city maps.

$$d(P,Q) = \sum_{i=1}^{n} |p_i - q_i| \qquad (4)$$

Where: P = $p_1, p_2,$ ……, p_n represent the coordinates of point P, Q = $q_1, q_2,$ ……, q_n represent the coordinates of point Q and, n denotes the number of dimensions in the space.

Fig. 1 is a graphical representation showing, because these two widely used, both Euclidean and Manhattan distance methods, which visually demonstrate their respective approaches to measuring distance between two points.

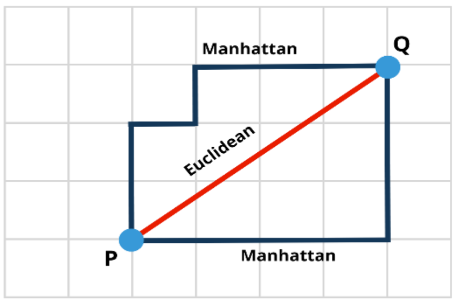


Figure 1: Comparison between Euclidean and Manhattan distance metrics

2. Value of K

The value of k in KNN classification plays a critical role in determining the model's performance and generalization ability. Selecting the appropriate value of k involves a trade-off between bias and variance in the model. A small value of k (e.g., k=1) can lead to a more flexible decision boundary but may be sensitive to noise and outliers, potentially resulting in overfitting. On the other hand, a large value of k (e.g., k>10) can smooth out the decision boundary and reduce overfitting but may lead to underfitting if the value is too large for the dataset. Choosing an optimal value of k, often through techniques like cross-validation and considering domain-specific knowledge, is essential to strike a balance between capturing the underlying patterns in the data and avoiding overfitting or underfitting, ultimately improving the classifier's predictive performance [21].

3. Voting in KNN

In the k-nearest neighbors (kNN) algorithm, voting is a crucial step in the classification process. After identifying the k nearest neighbors to a given data point, the algorithm assigns a class label to the data point based on a majority or weighted vote from its nearest neighbors.

A. Majority Voting

In the majority voting scheme, the class label assigned to a data point is determined by the most common class among its k nearest neighbors. This simple approach is effective when all neighbors have an equal vote weight.

B. Weighted Voting

Weighted voting assigns different weights to the votes of the nearest neighbors based on factors such as their distance to the data point. This allows closer neighbors to have a stronger influence on the classification decision. Weighted kNN variants like Distance-Weighted kNN (WKNN) and Dual-Weighted kNN (DWKNN) have been proposed to improve the performance of the traditional kNN algorithm by incorporating weighted voting mechanisms [22].

4. Hierarchical temporal memory and Neocortex API

Hierarchical Temporal Memory (HTM) is a machine learning model developed based on the principles of the neocortex, the part of the brain responsible for higher cognitive functions. HTM is designed to mimic the structure and function of the neocortex, particularly focusing on learning sequences and patterns in data. The HTM model consists of different components such as the Spatial Pooler, Temporal Memory, and SDR Encoder, which work together to process and learn from input data. The Spatial Pooler is responsible for creating Sparse Distributed Representations (SDRs) of input patterns, while the Temporal Memory component focuses on learning temporal sequences and patterns. By utilizing these components, HTM can recognize and predict sequences with robustness and efficiency, making it a powerful tool for various machine learning tasks.

The Neocortex API serves as a software interface that allows developers to access and utilize the HTM algorithms and functionalities in their applications. By providing a standardized way to interact with HTM technology, the Neocortex API simplifies the implementation of HTM systems and enables developers to leverage the power of HTM in their projects. This API facilitates the creation of HTM-based solutions for tasks such as pattern recognition, anomaly detection, and sequence prediction, making it a valuable tool for researchers and developers working in the field of artificial intelligence and machine learning [23].

IV. KNN CLASSIFIER IMPLEMENTATIONS

A native KNN classifier is developed to understand the working and functionality of a KNN classifier. It operates on the principle that data points with similar characteristics tend to belong to the same class. During training, the classifier stores labeled data points, where each point represents a specific feature set and its corresponding class label. When making predictions, the classifier calculates the distance between a new data point and the stored data points. It then identifies the k nearest neighbors (data points) to this new point based on the chosen distance metric. Finally, the classifier predicts the class label for the new data point by performing a majority vote among the k nearest neighbors as shown in Fig 2.

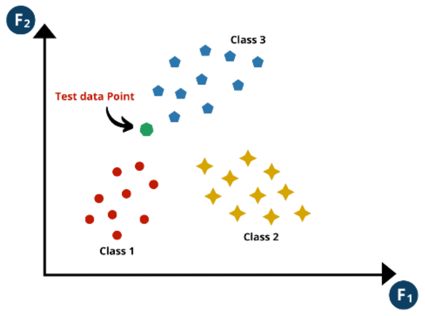


Figure 2: Three classes with a test data point

1. Selecting Value of K

The first step is to choose the number of nearest neighbors (K) to consider when making predictions. K represents the number of data points closest to the query point whose class labels will be used to classify the query point. Figure 3 (given below) shows the effect of K value on KNN classification. In our KNN classifier we have selected K=6 and the classifier predicts the class of the new data point based on the 6 nearest neighbors.

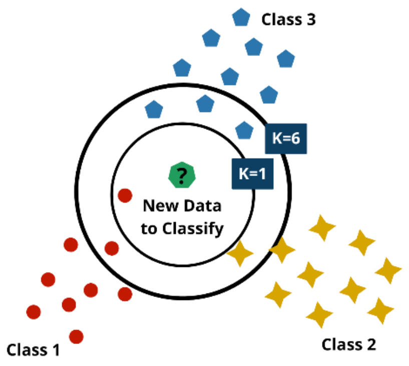


Figure 3: Impact of value k on KNN

2. Calculating Distance

When we set K to 6 and have a new data point for classification, we begin by computing the distances between this new point and all existing points in our dataset. This involves comparing the features of the new point with those of each existing point using a Euclidean distance metric as seen in Fig. 4.

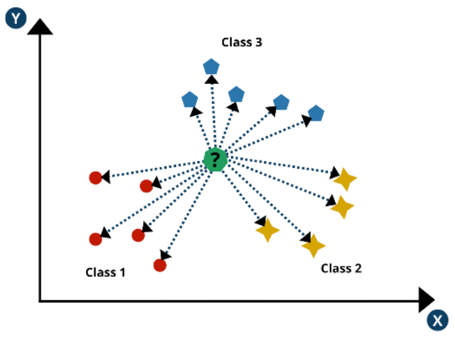


Figure 4: Calculated distance for each data point

Fig. 4 represents the Euclidean distance between two points based on the distance as shown in (5).

$$d(A,B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{5}$$

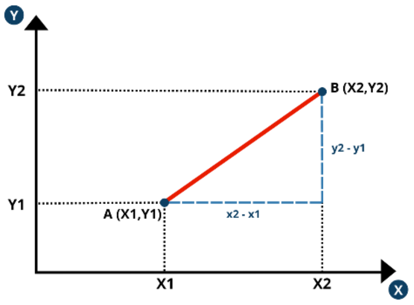


Figure 5: Euclidean distance

3. Finding Nearest Neighbor

Once we have computed these distances, the KNN algorithm selects the 6 nearest data points as shown in figure 6 to our new point based on these calculations. By focusing on these 6 nearest neighbors, we ensure that our classification decision is informed by the most relevant and immediate data points in our dataset. This step is crucial as it directly influences the accuracy and reliability of our KNN classification process, allowing us to accurately assess the local structure and characteristics of our data distribution.

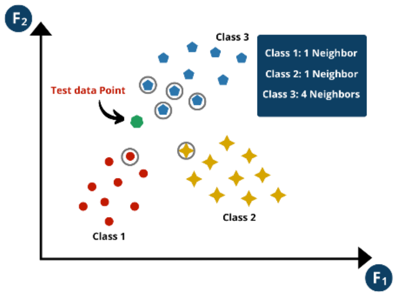


Figure 6: Identifying nearest neighbors for k=6

4. Voting and Prediction

In the next step, the voting method is executed to determine the predicted class label for our new data point. The class with the highest number of votes is then assigned the new data point. As the new data point has four neighbors from class 1, so the KNN classifier classifies the new data point to the class 3 as shown in Fig. 7.

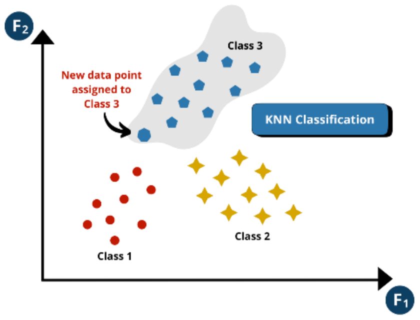


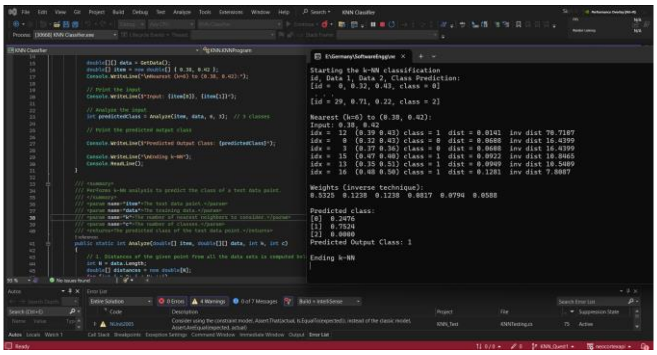
Figure 7: KNN Classification

V. Implementation And Results

The research progresses through three distinct phases: initially, a native implementation of the K-nearest neighbors (KNN) classifier undergoes rigorous unit testing to assess its performance on unseen data. Subsequently, the integration of KNN with the Neocortex API is orchestrated to augment sequence learning and classification capabilities. Finally, KNN is employed in the classification of Sparse Distributed Representations (SDRs) generated by the Spatial Pooler for sequential tasks.

A. Results for Native KNN

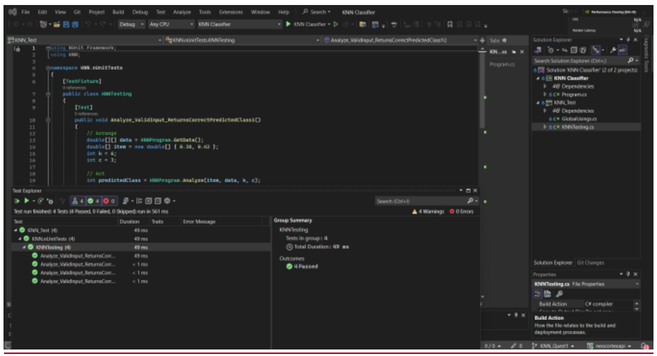
As described previously, the KNN classifier gets stream of data and has three different classes, the new data point is then classified into one of the three classes based on the KNN classifier. The figure below shows the KNN classifier output, which classifies the new data point into one of the three classes. The native KNN classifier helps in the understanding of different methods used in KNN classification and its workflow. This classifier predicts the right class with 75% accuracy and shows model's capabilities to classify the unseen data.


```
double[][] data = GetData();
double[] item = new double[] { 0.38, 0.42 };
Console.WriteLine("\nNearest (k=6) to (0.38, 0.42):");

// Print the input
Console.WriteLine($"Input: {item[0]}, {item[1]}");

// Analyze the input
int predictedClass = Analyze(item, data, 6, 3); // 3 classes

// Print the predicted output class

Console.WriteLine($"Predicted Output Class: {predictedClass}");

Console.WriteLine("\nEnding k-NN");
Console.ReadLine();
}

/// <summary>
/// Performs k-NN analysis to predict the class of a test data point.
/// </summary>
/// <param name="item">The test data point.</param>
/// <param name="data">The training data.</param>
/// <param name="k">The number of nearest neighbors to consider.</param>
/// <param name="c">The number of classes.</param>
/// <returns>The predicted class of the test data point.</returns>
public static int Analyze(double[] item, double[][] data, int k, int c)
{
    // 1. Distances of the given point from all the data sets is computed bel
    int N = data.Length;
    double[] distances = new double[N];
```

```
Starting the k-NN classification
id, Data 1, Data 2, Class Prediction:
[id =  0, 0.32, 0.43, class = 0]
 . . .
[id = 29, 0.71, 0.22, class = 2]

Nearest (k=6) to (0.38, 0.42):
Input: 0.38, 0.42
idx =  12  (0.39 0.43) class = 1  dist = 0.0141  inv dist 70.7107
idx =   0  (0.32 0.43) class = 0  dist = 0.0608  inv dist 16.4399
idx =   3  (0.37 0.36) class = 0  dist = 0.0608  inv dist 16.4399
idx =  15  (0.47 0.40) class = 1  dist = 0.0922  inv dist 10.8465
idx =  13  (0.35 0.51) class = 1  dist = 0.0949  inv dist 10.5409
idx =  16  (0.48 0.50) class = 1  dist = 0.1281  inv dist 7.8087

Weights (inverse technique):
0.5325  0.1238  0.1238  0.0817  0.0794  0.0588

Predicted class:
[0]  0.2476
[1]  0.7524
[2]  0.0000
Predicted Output Class: 1

Ending k-NN
```

Figure 8: Native KNN Classifier Output

Fig. 9 shows the output of unit tests applied on the KNN classifier. Multiple unit tests are implemented for different scenarios to test KNN classifier and testing phase stays successful as the classifier passed all the unit tests.



```
using NUnit.Framework;
using KNN;

namespace KNN.nUnitTests
{
    [TestFixture]
    public class KNNTesting
    {
        [Test]
        public void Analyze_ValidInput_ReturnsCorrectPredictedClass1()
        {
            // Arrange
            double[][] data = KNNProgram.GetData();
            double[] item = new double[] { 0.38, 0.42 };
            int k = 6;
            int c = 3;

            // Act
            int predictedClass = KNNProgram.Analyze(item, data, k, c);
```

```
Test run finished: 4 Tests (4 Passed, 0 Failed, 0 Skipped) run in 561 ms

Test                                   Duration
KNN_Test (4)                           49 ms
  KNN.nUnitTests (4)                   49 ms
    KNNTesting (4)                     49 ms
      Analyze_ValidInput_ReturnsCorr... 49 ms
      Analyze_ValidInput_ReturnsCorr... < 1 ms
      Analyze_ValidInput_ReturnsCorr... < 1 ms
      Analyze_ValidInput_ReturnsCorr... < 1 ms

Group Summary
KNNTesting
Tests in group: 4
Total Duration: 49 ms
Outcomes
4 Passed
```

Figure 9: Unit Tests for Native KNN Classifier

B. Integration of KNN with Neocortex API

A comprehensive framework for sequence learning and classification incorporates an encoder, Spatial Pooler, Temporal Memory, and K-nearest neighbor (KNN) classifier. Initially, the encoder transforms sequential data into a processable format. Subsequently, the Spatial Pooler, employing Hebbian learning principles within the Hierarchical Temporal Memory model, identifies spatial patterns. The Spatial Pooler's output bifurcates into two paths: one directed towards the KNN classifier, leveraging spatial patterns for class label prediction, and the other feeding into the Temporal Memory. The Temporal Memory, integral to the HTM framework, focuses on learning temporal sequences and patterns. It captures temporal dependencies within the spatially encoded sequences, enriching the understanding of sequential information. The enriched representation from the Temporal Memory is reintegrated into the classification pipeline, enhancing insights for accurate classification.

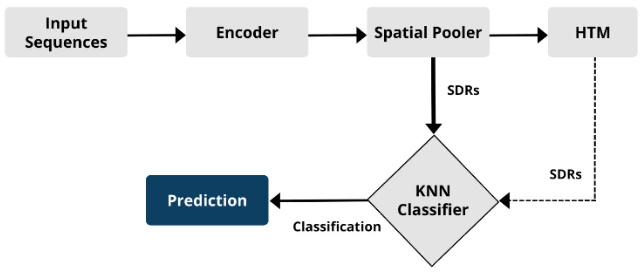
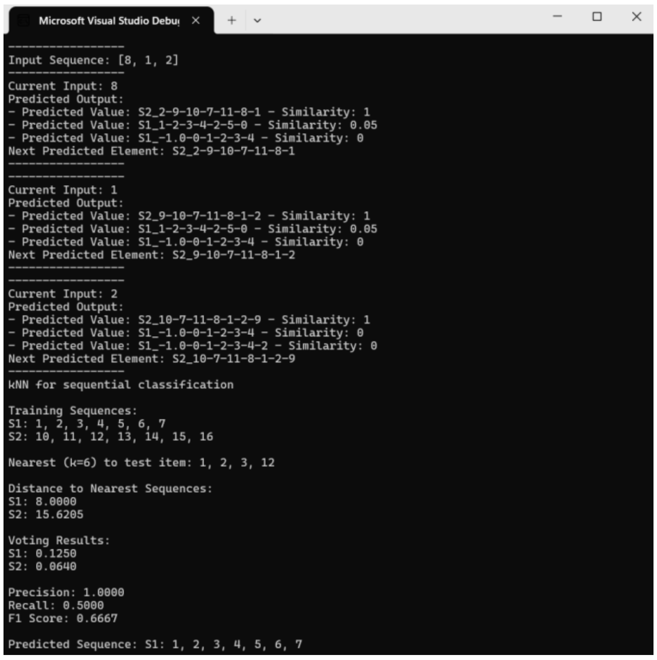


Figure 10: Digram for KNN classifier integration with Neocortex API

1) Results

In a scenario where a new sequence is introduced for classification using the KNN classifier, the system leverages the spatial patterns learned by the Spatial Pooler and the temporal context captured by the Temporal Memory. By combining the spatial and temporal information extracted from the input sequence, the KNN classifier can make informed decisions about the class label of the new sequence. This holistic approach, integrating spatial and temporal learning mechanisms inspired by HTM principles, enhances the classification accuracy and robustness of the system when handling sequential data.

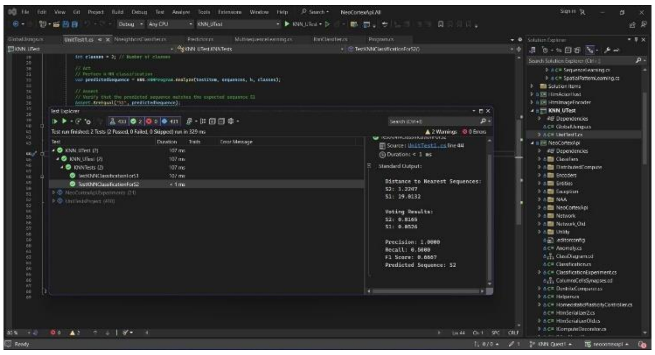


```
Microsoft Visual Studio Debu

-----------------
Input Sequence: [8, 1, 2]

Current Input: 8
Predicted Output:
- Predicted Value: S2_2-9-10-7-11-8-1 - Similarity: 1
- Predicted Value: S1_1-2-3-4-2-5-0 - Similarity: 0.05
- Predicted Value: S1_-1.0-0-1-2-3-4 - Similarity: 0
Next Predicted Element: S2_2-9-10-7-11-8-1
-----------------

Current Input: 1
Predicted Output:
- Predicted Value: S2_9-10-7-11-8-1-2 - Similarity: 1
- Predicted Value: S1_1-2-3-4-2-5-0 - Similarity: 0.05
- Predicted Value: S1_-1.0-0-1-2-3-4 - Similarity: 0
Next Predicted Element: S2_9-10-7-11-8-1-2
-----------------

Current Input: 2
Predicted Output:
- Predicted Value: S2_10-7-11-8-1-2-9 - Similarity: 1
- Predicted Value: S1_-1.0-0-1-2-3-4 - Similarity: 0
- Predicted Value: S1_-1.0-0-1-2-3-4-2 - Similarity: 0
Next Predicted Element: S2_10-7-11-8-1-2-9
-----------------
kNN for sequential classification

Training Sequences:
S1: 1, 2, 3, 4, 5, 6, 7
S2: 10, 11, 12, 13, 14, 15, 16

Nearest (k=6) to test item: 1, 2, 3, 12

Distance to Nearest Sequences:
S1: 8.0000
S2: 15.6205

Voting Results:
S1: 0.1250
S2: 0.0640

Precision: 1.0000
Recall: 0.5000
F1 Score: 0.6667

Predicted Sequence: S1: 1, 2, 3, 4, 5, 6, 7
```

Figure 11: Sequence learning and prediction with knn classifier

2) Unit Testing

The k-NN classification program underwent rigorous testing utilizing the NUnit.Framework framework. Two distinct test cases were meticulously devised to assess the program's efficacy in classifying disparate sequences. The outcomes revealed that the program adeptly classified the sequences with precision, underscoring its robustness and reliability. This empirical validation serves as a significant milestone, affirming the program's proficiency and engendering confidence in its applicability within academic and practical domains alike.



```
int classes = 2; // Number of classes

// Act
// Perform k-NN classification
var predictedSequence = KNN.KNNProgram.Analyze(testItem, sequences, k, classes);

// Assert
// Verify that the predicted sequence matches the expected sequence S1
Assert.AreEqual("S1", predictedSequence);
```

```
Test run finished: 2 Tests (2 Passed, 0 Failed, 0 Skipped) run in 329 ms

Test                              Duration
KNN_UTest (2)                     107 ms
  KNN_UTest (2)                   107 ms
    KNNTests (2)                  107 ms
      testKNNClassificationForS1  107 ms
      TestKNNClassificationForS2  < 1 ms

Source: UnitTest1.cs line 44
Duration: < 1 ms
Standard Output:

  Distance to Nearest Sequences:
  S2: 1.2247
  S1: 19.0132

  Voting Results:
  S2: 0.8165
  S1: 0.0526

  Precision: 1.0000
  Recall: 0.5000
  F1 Score: 0.6667
  Predicted Sequence: S2
```

Figure 12: Unit tests for KNN classifier with Neocortex API

C. KNN classifier with Spatial Pooler generated SDRs

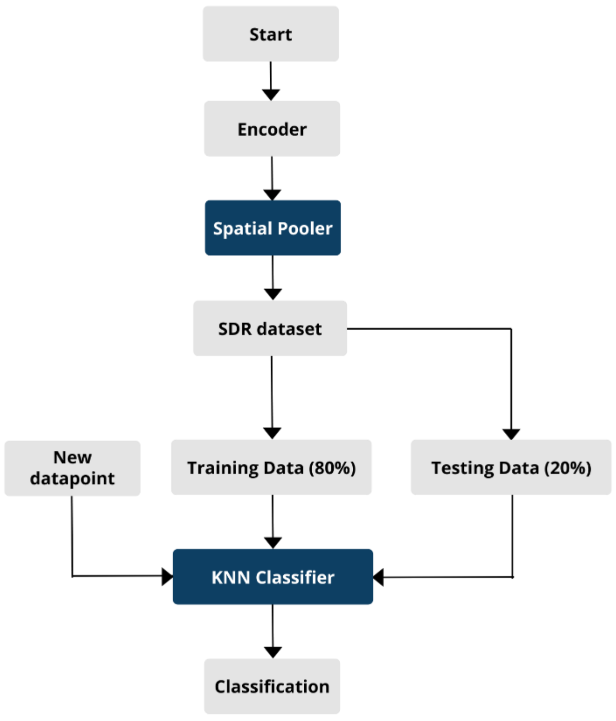


Figure 13: Process digram for KNN classification with SP generated SDRs

1) *Dataset:* From the Spatial Pattern Learning experiment, the dataset for SDRs is obtained and this dataset is stored in a csv file. The *LoadData()* method is vital for reading and parsing a dataset from a specified file path. It iterates through each line of the file, extracting features and labels, and creates *DataPoint* objects. Any encountered errors are handled gracefully, ensuring robustness. Finally, it returns the populated list of *DataPoint* objects. The dataset is taken after the stable state has achieved. Then the dataset is fed to the KNN classifier which is an essential component of the classification pipeline, as it utilizes the sparse distributed representations generated by the Spatial Pooler to classify sequences based on their similarity to the training data. The KNN algorithm determines the labels for input sequences by considering the nearest neighbors in the feature space, making it a powerful tool for pattern recognition tasks. In this configuration, the KNN classifier functions as the final decision-maker in the classification process, utilizing the enriched feature representations provided by the Spatial Pooler to make precise classification decisions.

2) *Training and testing:* First of all, the dataset is split into training and testing datasets using *SplitData* method. During the training phase, the KNN classifier will learn the relationship between the input data and their corresponding classes using the 80% training dataset *(trainingData)* which helps the KNN classifier becoming a robust and generalized model that can accurately classify new, unseen data points. The classifier learned the relationships between the SDRs generated by the Spatial Pooler and their corresponding class labels in the training set. The remaining 20% of the dataset *(testingData)* is used to evaluate the model's performance and ensure that it can generalize well to new data. For each data point in the testing set, the classifier calculates the distances to the data points in the training set and identifies the k-nearest neighbors based on a predefined value of k.

```
List<DataPoint> trainingData, testingData;

(trainingData, testingData) = SplitData(data, 0.8); //
80% training data, 20% testing data
KNNClassifier knn = new
KNNClassifier(trainingData);
```

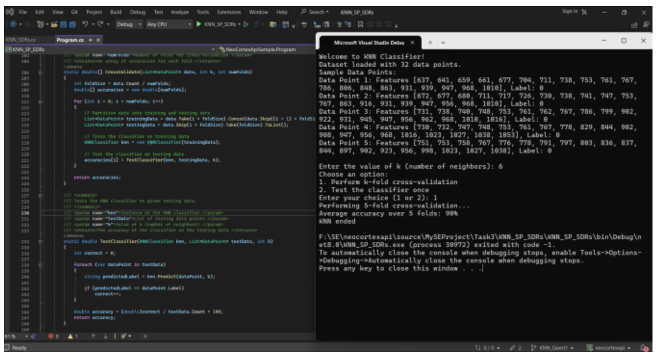
3) *Accuracy:* The accuracy of the KNN model is closely tied to the choice of the k parameter, and finding the optimal value of k is an important step in the model development process. The optimal value of k is typically determined through a process of cross-validation or other model selection technique-s, as it can vary depending on the specific dataset and problem at hand.

Value of k (integer)	**Accuracy of prediction in percentage (%)**
1	100
2	100
3	100
4	100
5	89
6	85

Table 1: Impact of k on the accuracy of the model

4) *k-fold cross validation:* The *CrossValidate()* method conducts a cross-validation process to assess the performance of a K-Nearest Neighbors (KNN) classifier. It partitions the dataset into training and testing sets based on the specified number of folds. For each fold, the method trains a KNN classifier using the training data and evaluates its accuracy on the testing data. The accuracies obtained from each fold are stored and returned, offering insights into the classifier's overall performance. This approach ensures thorough evaluation and validation of the KNN classifier across different subsets of the dataset.

The results obtained from the 5-fold cross-validation revealed a classification accuracy of 90%. This high accuracy level indicates the model's ability to make accurate predictions on unseen data instances across the different folds of the cross-validation process.



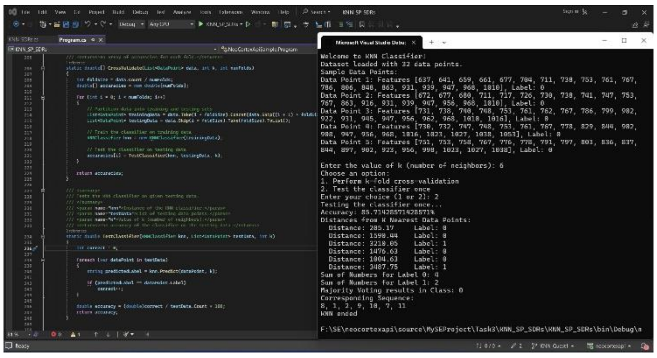
```
Welcome to KNN Classifier!
Dataset loaded with 32 data points.
Sample Data Points:
Data Point 1: Features [637, 641, 659, 661, 677, 704, 711, 738, 753, 761, 767,
786, 806, 848, 863, 931, 939, 947, 968, 1010], Label: 0
Data Point 2: Features [672, 677, 680, 711, 717, 726, 730, 738, 741, 747, 753,
767, 863, 916, 931, 939, 947, 956, 968, 1010], Label: 0
Data Point 3: Features [731, 738, 740, 748, 753, 761, 762, 767, 786, 799, 902,
922, 931, 945, 947, 956, 962, 968, 1010, 1016], Label: 0
Data Point 4: Features [730, 732, 747, 748, 753, 761, 767, 778, 829, 844, 902,
908, 947, 956, 968, 1016, 1023, 1027, 1038, 1053], Label: 0
Data Point 5: Features [751, 753, 758, 767, 776, 778, 791, 797, 803, 836, 837,
844, 897, 902, 923, 956, 998, 1023, 1027, 1038], Label: 0

Enter the value of k (number of neighbors): 6
Choose an option:
1. Perform k-fold cross-validation
2. Test the classifier once
Enter your choice (1 or 2): 1
Performing 5-fold cross-validation...
Average accuracy over 5 folds: 90%
KNN ended

F:\SE\neocortexapi\source\MySEProject\Task3\KNN_SP_SDRs\KNN_SP_SDRs\bin\Debug\n
et8.0\KNN_SP_SDRs.exe (process 38972) exited with code -1.
To automatically close the console when debugging stops, enable Tools->Options-
>Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 14: Results for 5-fold cross validation

```
static double[] CrossValidate(List<DataPoint> data, int k, int numFolds)
{
    int foldSize = data.Count / numFolds;
    double[] accuracies = new double[numFolds];

    for (int i = 0; i < numFolds; i++)
    {
        // Partition data into training and testing sets
        List<DataPoint> trainingData = data.Take(i * foldSize).Concat(data.Skip((i + 1) * foldSize)).ToList();
        List<DataPoint> testingData = data.Skip(i * foldSize).Take(foldSize).ToList();

        // Train the classifier on training data
        KNNClassifier knn = new KNNClassifier(trainingData);

        // Test the classifier on testing data
        accuracies[i] = TestClassifier(knn, testingData, k);
    }

    return accuracies;
}
```

5) *Prediction:* The KNN classifier, after being trained on 80% of the dataset, is able to accurately predict the output of a new data point with 85% accuracy when the value of k is set to 6. When a new data point is introduced for classification, the KNN classifier calculates the distances between the SDR of the new data point and the SDRs of the training data points. The KNN classifier finds the K (in this case, 6) nearest neighbors of the input data point in the training set. The classifier then performs majority voting among the k-nearest neighbors to predict the class label of the new data point. *Predict()* method efficiently applies the KNN algorithm to predict the label of a given *testDataPoint* by considering the k nearest neighbors in the training data. Its implementation ensures accurate label prediction by prioritizing the majority label among the nearest neighbors.

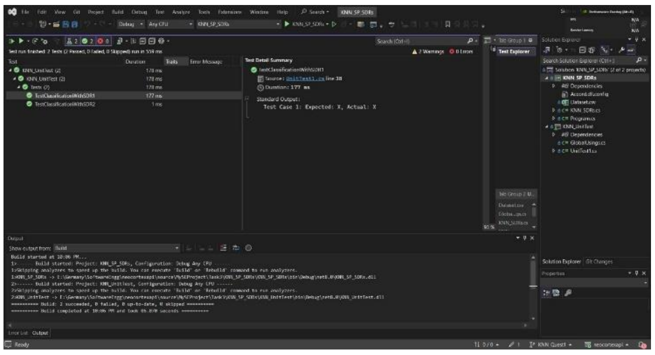
```
public string Predict(DataPoint testDataPoint, int k)
{
    Dictionary<string, int> labelCounts = new Dictionary<string, int>();

    // Maintain a dictionary to store distances and corresponding labels
    Dictionary<double, string> distanceLabelMap = new Dictionary<double, string>()
    …….
```



```
Welcome to KNN Classifier!
Dataset loaded with 32 data points.
Sample Data Points:
Data Point 1: Features [637, 641, 659, 661, 677, 704, 711, 738, 753, 761, 767,
786, 806, 848, 863, 931, 939, 947, 968, 1010], Label: 0
Data Point 2: Features [672, 677, 680, 711, 717, 726, 730, 738, 741, 747, 753,
767, 863, 916, 931, 939, 947, 956, 968, 1010], Label: 0
Data Point 3: Features [731, 738, 740, 748, 753, 761, 762, 767, 786, 799, 902,
922, 931, 945, 947, 956, 962, 968, 1010, 1016], Label: 0
Data Point 4: Features [730, 732, 747, 748, 753, 761, 767, 778, 829, 844, 902,
908, 947, 956, 968, 1016, 1023, 1027, 1038, 1053], Label: 0
Data Point 5: Features [751, 753, 758, 767, 776, 778, 791, 797, 803, 836, 837,
844, 897, 902, 923, 956, 998, 1023, 1027, 1038], Label: 0

Enter the value of k (number of neighbors): 6
Choose an option:
1. Perform k-fold cross-validation
2. Test the classifier once
Enter your choice (1 or 2): 2
Testing the classifier once...
Accuracy: 85.71428571428571%
Distances from K Nearest Data Points:
  Distance: 285.17     Label: 0
  Distance: 1598.44    Label: 0
  Distance: 3218.05    Label: 1
  Distance: 1476.63    Label: 0
  Distance: 1804.63    Label: 0
  Distance: 3487.75    Label: 1
Sum of Numbers for Label 0: 4
Sum of Numbers for Label 1: 2
Majority Voting results in Class: 0
Corresponding Sequence:
8, 1, 2, 9, 10, 7, 11
KNN ended

F:\SE\neocortexapi\source\MySEProject\Task3\KNN_SP_SDRs\KNN_SP_SDRs\bin\Debug\n
```

Figure 15: KNN classifier prediction

6) *Unit testing:* The unit test conducted on the K-nearest neighbor (KNN) classifier is exemplified through the selection of two random SDRs from the dataset. These SDRs serve as test data to evaluate the classifier's performance in predicting the corresponding class values. The graphical representation provided in the figure substantiates the successful execution of the KNN classifier during the unit test.



```
Test run finished: 2 Tests (2 Passed, 0 Failed, 0 Skipped) run in 559 ms
Test                              Duration
KNN_UnitTest (2)                  178 ms
  KNN_UnitTest (2)                178 ms
    Tests (2)                     178 ms
      TestClassificationWithSDR1  177 ms
      TestClassificationWithSDR2  1 ms

Test Detail Summary
TestClassificationWithSDR1
Source: UnitTest1.cs line 38
Duration: 177 ms
Standard Output:
  Test Case 1: Expected: X, Actual: X
```

```
Build started at 10:06 PM...
1>------ Build started: Project: KNN_SP_SDRs, Configuration: Debug Any CPU ------
1>Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.
1>KNN_SP_SDRs -> E:\Germany\SoftwareEngg\neocortexapi\source\MySEProject\Task3\KNN_SP_SDRs\KNN_SP_SDRs\bin\Debug\net8.0\KNN_SP_SDRs.dll
2>------ Build started: Project: KNN_UnitTest, Configuration: Debug Any CPU ------
2>Skipping analyzers to speed up the build. You can execute 'Build' or 'Rebuild' command to run analyzers.
2>KNN_UnitTest -> E:\Germany\SoftwareEngg\neocortexapi\source\MySEProject\Task3\KNN_SP_SDRs\KNN_UnitTest\bin\Debug\net8.0\KNN_UnitTest.dll
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Build completed at 10:06 PM and took 05.870 seconds ==========
```

Figure 16: Unit tests for KNN classifer on SP generated SRDs

VI. Conclusion

The research introduces K-Nearest Neighbors (KNN) classification within the Neocortex API framework, inspired by neocortical information processing. By integrating spatial pooling and hierarchical temporal memory (HTM), the method enhances the classification of sequential data streams. Through rigorous testing and integration with the Neocortex API, the study consistently demonstrates the model's effectiveness, achieving a commendable 90% accuracy rate across varied scenarios. These findings underscore the potential of neuromorphic-inspired frameworks in advancing sequential data analysis. Moving forward, optimizing performance and exploring broader applications remain promising areas for future research, reflecting the ongoing convergence of traditional machine learning and biologically inspired computational models.

VII. References

[1] B. Rajoub, “Supervised and unsupervised learning,” in *Biomedical Signal Processing and Artificial Intelligence in Healthcare*, Elsevier, 2020, pp. 51–89. doi: 10.1016/B978-0-12-818946-7.00003-2.

[2] A. A. Soofi and A. Awan, “Classification Techniques in Machine Learning: Applications and Issues,” *Journal of Basic & Applied Sciences*, vol. 13, pp. 459–465, 2017.

[3] D. Maulud and A. M. Abdulazeez, “A Review on Linear Regression Comprehensive in Machine Learning,” *Journal of Applied Science and Technology Trends*, vol. 1, no. 4, pp. 140–147, Dec. 2020, doi: 10.38094/jastt1457.

[4] A. K. Nikhath, K. Subrahmanyam, and R. Vasavi, “Building a K-Nearest Neighbor Classifier for Text Categorization.” [Online]. Available: www.ijcsit.com

[5] Z. Farou, Y. Wang, and T. Horváth, “Cluster-based oversampling with area extraction from representative points for class imbalance learning,” *Intelligent Systems with Applications*, vol. 22, p. 200357, Jun. 2024, doi: 10.1016/j.iswa.2024.200357.

[6] S. Raschka, “STAT 479: Machine Learning Lecture Notes,” 2018. [Online]. Available: http://stat.wisc.edu/∼sraschka/teaching/stat479-fs2018/

[7] S. Zhang and J. Li, “KNN Classification with One-Step Computation,” *IEEE Trans Knowl Data Eng*, vol. 35, no. 3, pp. 2711–2723, Mar. 2023, doi: 10.1109/TKDE.2021.3119140.

[8] S. Sun and R. Huang, “An adaptive k-nearest neighbor algorithm,” *Proceedings - 2010 7th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2010*, vol. 1, pp. 91–94, 2010, doi: 10.1109/FSKD.2010.5569740.

[9] S. Liu, P. Zhu, and S. Qin, “An improved weighted knn algorithm for imbalanced data classification,” *2018 IEEE 4th International Conference on Computer and Communications, ICCC 2018*, pp. 1814–1819, Dec. 2018, doi: 10.1109/COMPCOMM.2018.8780580.

[10] Anna University. Department of Information Technology. and Institute of Electrical and Electronics Engineers. Madras Section., *International Conference on Recent Trends in Information Technology: ICRTIT 2011, June 03-05, 2011*. IEEE, 2011.

[11] J. L. Suárez, S. Garcıa, and F. Herrera, “pyDML: A Python Library for Distance Metric Learning,” *Journal of Machine Learning Research*, vol. 21, no. 96, pp. 1–7, 2020.

[12] Z. Pan, Y. Pan, Y. Wang, and W. Wang, “A new globally adaptive k-nearest neighbor classifier based on local mean optimization,” *Soft comput*, pp. 1–15, 2020.

[13] K. Chomboon, P. Chujai, P. Teerarassammee, K. Kerdprasop, and N. Kerdprasop, “An Empirical Study of Distance Metrics for k-Nearest Neighbor Algorithm,” Institute of Industrial Applications Engineers, 2015, pp. 280–285. doi: 10.12792/iciae2015.051.

[14] A. M. P. Boelens and H. A. Tchelepi, “Minkowski functionals for phase behavior under confinement,” *arXiv preprint arXiv:2004.01344*, 2020.

[15] S. Zeng, X. Wang, X. Duan, S. Zeng, Z. Xiao, and D. Feng, “Kernelized Mahalanobis Distance for Fuzzy Clustering,” *IEEE Transactions on Fuzzy Systems*, 2020.

[16] K. Taunk, S. De, S. Verma, and A. Swetapadma, “A Brief Review of Nearest Neighbor Algorithm for Learning and Classification,” in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, IEEE, 2019, pp. 1255–1260.

[17] S. Jaiyen and P. Sornsuwit, “A new incremental decision tree learning for cyber security based on ilda and mahalanobis distance,” *Engineering Journal*, vol. 23, no. 5, pp. 71–88, 2019.

[18] C. Pilinszki-Nagy and B. Gyires-Tóth, “Performance analysis of sparse matrix representation in hierarchical temporal memory for sequence modeling,” *Infocommunications Journal*, vol. 12, no. 2, pp. 41–49, Aug. 2020, doi: 10.36244/ICJ.2020.2.6.

[19] L. G. Maxim, J. I. Rodriguez, and B. Wang, “Euclidean distance degree of the multiview variety,” *SIAM J Appl Algebr Geom*, vol. 4, no. 1, pp. 28–48, 2020.

[20] M. You, Y. Xiao, S. Zhang, P. Yang, and S. Zhou, “Optimal mathematical programming for the warehouse location problem with Euclidean distance linearization,” *Comput Ind Eng*, vol. 136, pp. 70–79, 2019.

[21] S. Zhang, X. Li, M. Zong, X. Zhu, and D. Cheng, “Learning k for kNN Classification,” *ACM Trans Intell Syst Technol*, vol. 8, no. 3, Jan. 2017, doi: 10.1145/2990508.

[22] D. Mateos-García, J. García-Gutiérrez, and J. C. Riquelme-Santos, “An evolutionary voting for k-nearest neighbours.”

[23] D. Dobric, A. Pech, B. Ghita, and T. Wennekers, “Scaling the HTM Spatial Pooler,” *International Journal of Artificial Intelligence & Applications*, vol. 11, no. 4, pp. 83–100, Jul. 2020, doi: 10.5121/ijaia.2020.11407.