Chapter 4

Proposed Algorithm and NMRS Model for Mentor-Mentee Matching Process

4.1 Proposed Mentor-Mentee Matching Process

4.1.1 Information Collection and Input Plan

The foundation of the coordinating preparation is gathering nitty gritty, organized, and noteworthy data from coaches and mentees through mindfully created studies or shapes. This step includes:

- **Planning Surveys:** Consolidate a mix of subjective and quantitative questions to capture inclinations, ability, and desires.
- **For Coaches:** Zones of mastery, a long time of involvement, industry information, mentoring fashion, and accessibility.
- **For Mentees:** Career objectives, learning styles, zones for advancement, accessibility, and inclinations for mentoring approaches.
- **Energetic Areas:** Incorporate versatile addressing based on the introductory answers to form the shapes more natural and diminish reaction weariness.
- **Organized Capacity:** Utilize social databases, such as SQL, to store and categorize the reactions for simple recovery and preparing.

4.2 Progressed Information Pre-processing

Planning the crude information is basic to guaranteeing the machine learning algorithm's viability. This stage incorporates:

- **Comprehensive Cleaning:** Address lost values utilizing domain-relevant ascription procedures, such as mean/mode ascription for numerical information and prescient ascription for categorical information.
- **Standardization and Normalization:** Apply highlight scaling procedures like MinMaxScaler or Z-Score normalization to harmonize the information run and decrease change affect.
- **Exception Location and Evacuation:** Utilize factual strategies (e.g., boxplots) or calculations like Confinement Timberlands to distinguish and evacuate inconsistencies.

- **Include Designing:** Improve the dataset with determined properties, such as mentor-mentee accessibility cover or calculated expertise crevice.

4.3 Training and Model Design

In order to process the pre-processed data and forecast mentor-mentee compatibility, this stage entails choosing the proper machine learning techniques:

4.3.1 Selection of Similarity Metrics:

For textual data (such as descriptions of mentoring styles), use cosine similarity. For numerical data, such years of experience or availability overlap, use Euclidean distance. Try using hybrid metrics to integrate different kinds of data.

4.3.2 Choosing an Algorithm:

- **Clustering Algorithms:** To create groups with a high degree of internal similarity between mentors and mentees, use hierarchical clustering or K-Means.
- **Recommendation systems:** Use collaborative filtering or matrix factorization (like Singular Value Decomposition) to get individualized matching.
- **Hybrid Models:** To improve match accuracy, combine collaborative and content-based filtering.
- **Model Training:** Divide the dataset into subgroups for testing, validation, and training. Use strategies such as cross-validation to avoid overfitting.

4.4 Evaluation and Suggestions

The program rates mentors for each mentee, producing the top two matches, after determining similarity scores for each mentor-mentee pair. Improvements in this stage consist of: Determine the relative value of particular traits using weighted similarity scoring; for example, alignment with professional goals may be given greater weight than availability.

4.4.1 Diversity Consideration:

Put restrictions in place to guarantee a variety of mentor choices and prevent suggesting mentors to mentees more than once.

4.4.2 Explainable AI (XAI):

Offer comprehensible explanations for the recommendations made for particular matches (e.g., a high similarity score because of common data science knowledge).

4.5 User Engagement and Results

The output is intended to be actionable and easy to use:

4.5.1 Dashboard for Mentees:

Show the similarity scores and biographies of the top two suggested mentors.
For every match, include important characteristics and a succinct explanation.
Give mentees the choice to select one of the two mentors.

4.5.2 Optional Administrator Panel:

Give program managers the option to manually match mentors and mentees or to override automated recommendations.

4.6 Feedback Loop and Improvement of the Model

Feedback is included into the system as it develops to continuously improve performance:

4.6.1 Feedback Collection:

After the match, get comments about results and satisfaction from mentors and mentees.

4.6.2 Model tuning:

To improve the matching model, use feedback data. Depending on feedback trends, change the weights of specific features or investigate different algorithms.

4.6.3 Data Augmentation:

Add new mentor and mentee entries to the dataset on a regular basis, or modify the questions to take into account shifting program goals.

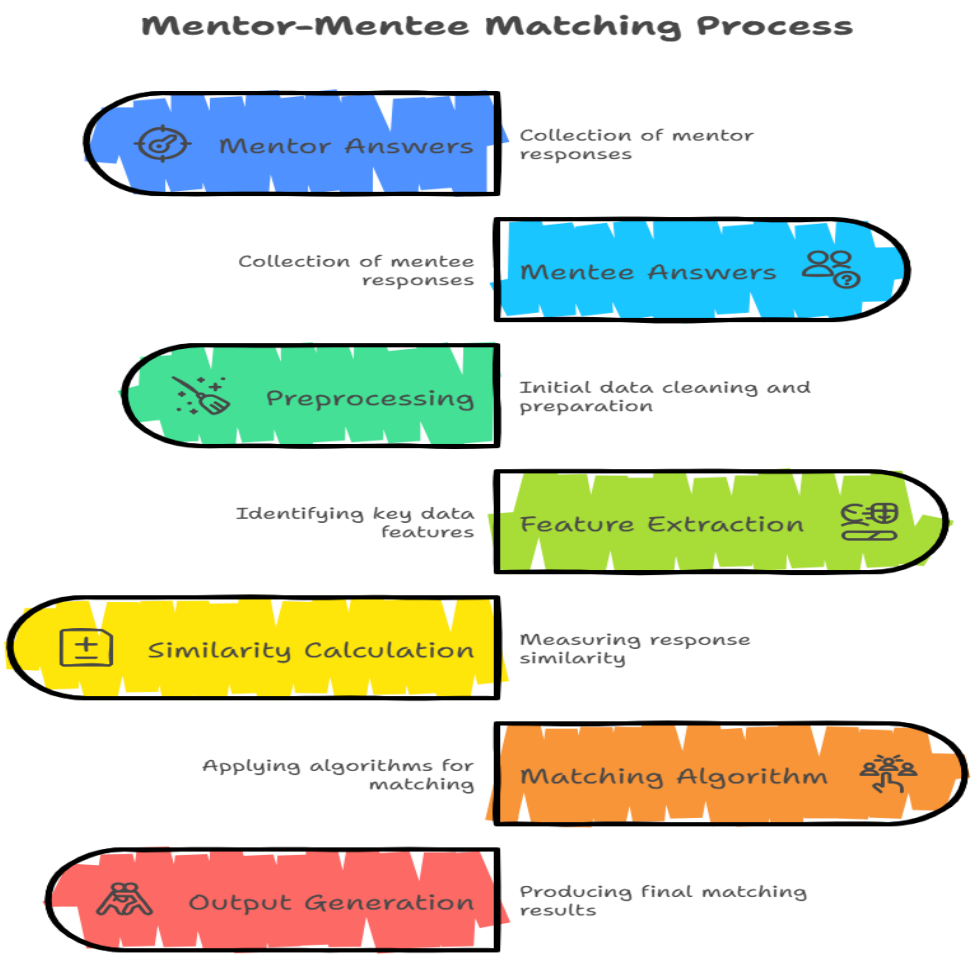


Figure 4. 1: Mentor – Mentee Matching Process

4.7 Machine Learning in Mentor Mentee Match making Process

Machine learning (ML) recommendation systems, which leverage individual preferences, data-driven insights, and complex algorithms to optimize the pairing process, can significantly help with successful mentor-mentee matching. The primary goal of these systems is to pair mentees with mentors who can provide the best guidance, creating a powerful and advantageous relationship. Making sure that mentors and mentees are a good fit in terms of abilities, objectives, and values is a crucial component of a successful partnership. By evaluating how well mentor and mentee profiles overlap based on answers to a predetermined set of questions, machine learning—specifically, cosine similarity—can greatly enhance the matching process.

By analysing past data, patterns, and preferences to create better, more educated matches, machine learning (ML) can automate and improve the mentor-mentee matching process. The compatibility of a mentor and a mentee is assessed in a typical mentor-mentee matching situation based on a number of qualities or attributes, including communication preferences, career goals, interests, and areas of competence.

Large data sets can be processed using machine learning techniques to uncover hidden trends in the profiles of mentors and mentees. An intelligent matchmaking system that learns from previous fruitful mentorship relationships and uses this information for upcoming matching jobs can be developed by utilizing these patterns.

4.8 Cosine Similarity

An efficient mathematical method for calculating the similarity between two vectors—in this example, the mentors' and mentees' response sets—is cosine similarity. The best matches between mentors and mentees are determined by converting each questionnaire response into a numerical vector and comparing their alignment using cosine similarity.

The cosine of the angle between two vectors is used to calculate cosine similarity. On a scale of 0 (totally dissimilar) to 1 (identical), the similarity score is displayed.

This is the cosine similarity formula:

$$\text{Similarity}(A, B) = A \cdot B \;/\; \|A\|\|B\|$$

Where, the mentor's and mentee's response vectors are denoted by A and B, respectively.

$A \cdot B$ is the vectors' dot product.

These are the vector magnitudes: $\|A\|$ and $\|B\|$.

4.9 Steps for Algorithm Formation

4.9.1 Data Preparation

- **Representation in the Questionnaire:** There are several questions in each questionnaire, categorized into several groups. Numerical encoding of the responses is used, such as weighted scores for specific responses or Likert scale values like 1–5.
- **Formation of Vectors:** Every individual involved (mentor or mentee) is depicted as a vector of numerical answers: V=[v1,v2,v3,…,vn], where vi is the numerical value of the i-th question in the questionnaire.
- **Normalization:** Vectors are normalized (if required) to provide uniformity and prevent similarity calculations from being skewed by variations in question scales.

4.10 Procedure for the Algorithm of Matching

4.10.1 Step 1: Input Data:

4.10.1.1 Gather Responses to the Questionnaire:

- Collect questionnaire answers from mentees and mentors.
- With online tools (like Google Forms) or in-person surveys, make sure that every question is answered precisely and consistently.
- To get rid of any erroneous or incomplete responses, validate and clean the data.

4.10.1.2 Use Vectors to Show Responses:

- Create a numerical vector from each participant's responses:
- For questions that are binary, Yes = 1 and No = 0.
- Use the scale values (e.g., 1 to 5) directly for Likert scale inquiries.
- If the question is weighted, give the options predetermined weights and incorporate them into the vectors.

Example:

- Mentor A’s responses: [1,4,1,3,5]
- Mentee X’s responses: [1,5,1,4,5]

Each vector represents a participant's preferences, values, and personality traits based on the questionnaire.

4.10.1.3 Save Information for Effective Processing:

- Vectors for each mentor and mentee should be kept in a structured database or matrix.

Mentors:

Mentor A: [1, 4, 1, 3, 5]

Mentor B: [0, 5, 1, 4, 4]

Mentees:

Mentee X: [1, 5, 1, 4, 5]

Mentee Y: [1, 3, 1, 2, 4]

4.10.2 Step 2: Cosine Similarity Calculation in Pairs

4.10.2.1 Determine the Similarity between Pairs:

Example:

Mentor A's vector: [1,4,1,3,5]

Mentee X's vector: [1,5,1,4,5]

Dot product: (1×1)+(4×5)+(1×1)+(3×4)+(5×5)=1+20+1+12+25=59

Magnitudes:

$$\|\mathbf{A}\| = \sqrt{(1^2 + 4^2 + 1^2 + 3^2 + 5^2)} = \sqrt{1 + 16 + 1 + 9 + 25} = \sqrt{52}$$

$$\|\mathbf{B}\| = \sqrt{(1^2 + 5^2 + 1^2 + 4^2 + 5^2)} = \sqrt{1 + 25 + 1 + 16 + 25} = \sqrt{68}$$

Cosine Similarity:

$$\frac{59}{\sqrt{52} \cdot \sqrt{68}} \approx 0.98$$

4.10.2.2 Store Results:

- Save similarity scores in a similarity matrix.

4.11 Proposed Algorithm

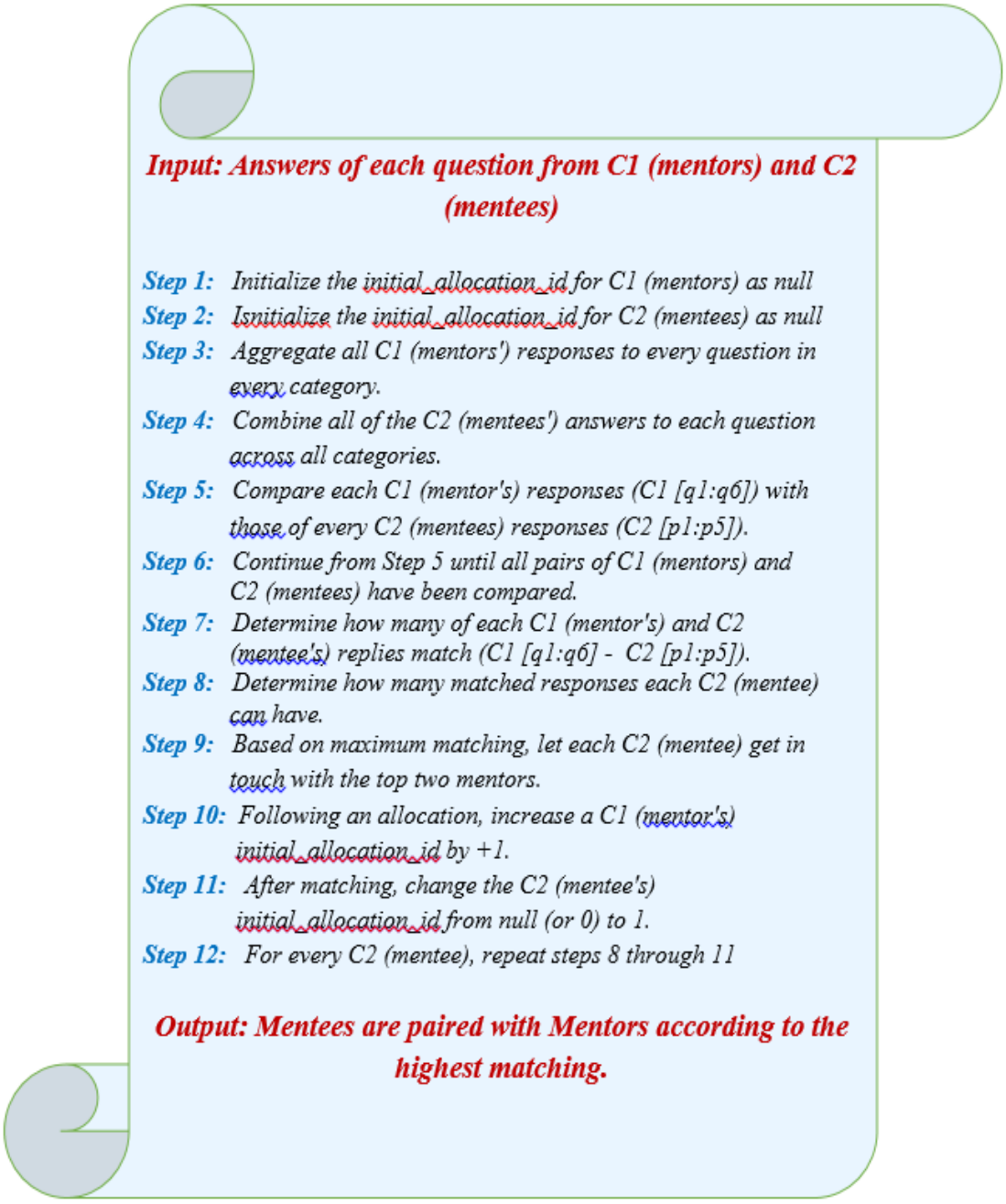


Figure 4. 2: Proposed Algorithm - NMRS

Explanation:

Input: Answers of each question from mentors and students / mentees.

Explanation: As an input we will take the answers from each question from each category from mentors and mentees.

.

Step 1: Initialize the initial_allocation_id for mentors as null

Every mentor's initial_allocation_id is set to null (or any other unassigned state, like 0). Afterwards, this variable will be used to monitor if the mentor has been assigned a student or not. As students are matched with mentors, we will update this variable to reflect the allocation status of each mentor.

Step 2: Initialize the initial_allocation_id for students as null. In a similar manner, each student's initial_allocation_id is set to null. This suggests that no mentor has been assigned to the student as of yet. As soon as the student and mentor are matched, this variable will be updated.

Step 3: Aggregate all mentors' responses to every question in every category. Answers to a series of pre-established questions are given by each mentor, perhaps spanning many categories (such as abilities, interests, objectives, etc.). These responses are saved, and subsequently, each mentor's response and the students' responses are compared to identify matches.

Step 4: Combine all of the mentees' answers to each question across all categories. Each student responds to an identical series of questions, which might cover multiple categories. Their responses, like those of the mentors, will be kept on record for comparison purposes. The students' interests or preferences are reflected in these responses.

Step 5: Compare each mentor's responses with those of every student. Examine each mentor's responses, question by question and category by category, to those of every student. A match counter may be increased for each time a student and mentor provide a matched response. This will provide an initial sense of how well a mentor fits with a given student.

Step 6: Continue from Step 5 until all pairs of mentors and students have been compared. For each potential mentor-student relationship, the matching procedure from Step 5 is carried out over and over. We will have a record of the number of answers that match between each mentor and each student at the end of this stage.

Step 7: Determine how many of each mentor's and student's replies match. The total number of matched answers is computed and recorded for each mentor-student pair.

Based on the students' answers, this stage basically measures how compatible each mentor and student are.

Step 8: Determine how many matched responses each student can have. Ascertain which mentor(s) best fit each student based on the greatest number of matched responses. This is an important step since it will assist in determining which mentors are best suited for each individual student.

Step 9: Based on maximum matching, let each student get in touch with the top two mentors. The student is given the chance to get in touch with their top two mentors once the mentor(s) with the highest match scores for them have been identified. In order to prevent overstuffing any one mentor, the algorithm can give preference to mentors with the highest match scores and let students select from the top two. This method gives students a feeling of independence by letting them choose from the options that best suit them.

Step 10: Following an allocation, increase a mentor's initial_allocation_id by +1. When a student chooses a mentor, that mentor's initial_allocation_id is adjusted (for example, from null or 0 to 1). The number of students who have been paired with a mentor is tracked in this phase. The system can use this data to restrict the amount of students that are paired with each mentor, if necessary.

Step 11: After matching, change the student's initial_allocation_id from null (or 0) to 1. Similar to this, a student's initial_allocation_id is changed from null (or 0) to 1 when they choose a mentor. This modification shows that the student and mentor match went well.

Step 12: For every student, repeat steps 8 through 11 For every student, the algorithm keeps up the matching and allocation procedure. Every student has the opportunity to meet with their top two mentors; pairs of students and mentors are then selected based on the highest matching scores. This iteration guarantees that a mentor is allocated to each student

Output:

Students are paired with mentors according to the highest matching. Upon the completion of the algorithm, every student will have been paired with a mentor (or

two mentors, contingent on the methodology), and each mentor will have been given students based on the greatest degree of similarity between their responses. To ensure the best possible mentor-student match, the allocation is based on the highest level of compatibility between mentors and students.

Through a series of question comparisons, the aforementioned algorithm systematically assigns mentors to students / mentees. It makes sure that every student is paired with mentors who best fit their responses in terms of goals, interests, and skill sets. By giving each student / mentee a choice between their top two matching mentors, the method guarantees equity.

4.12 Pseudocode used to build NMRS – Mentor Buddy Matching Model

Table 4. 1: Load Excel File

Procedure Name : load_excel
Action : pseudo code to load excel file
Purpose: The goal is to load a user-selected Excel file, store its contents in a global DataFrame, and then show the data along with the names of its columns.
Input: A user-selected Excel file via a file dialog box.
Output: • The loaded Excel data is contained in a DataFrame df. Column names were shown and a loading success confirmation was printed.
Variables Used: • df (Global): The DataFrame that contains the contents of the Excel file. • file_path: The string pointing to the Excel file that was chosen.
Pseudocode: BEGIN FUNCTION load_excel DECLARE a global variable `df` to store the data from the Excel file. BEGIN PROMPT the user to select an Excel file using a file dialog. STORE the selected file path in the variable `file_path`. END

```
    BEGIN
      LOAD the Excel file from `file_path` into a DataFrame using `pandas.read_excel`.
      ASSIGN the loaded data to `df`.
    END

    BEGIN
      DISPLAY the content of `df` by calling the `display_data` function.
    END

    BEGIN
      PRINT "Excel file loaded."
      PRINT the column names of `df` using its `columns` attribute.
    END

  END FUNCTION
```

Table 4. 2: Classify Roles

Procedure Name : classify_roles
Action : pseudo code to **classify_roles Function**
Purpose: To use a machine learning model to categorize the dataset's members as mentors or pupils, compute classification metrics, and present pertinent findings.
Input: • df: The loaded dataset is thought to have additional feature columns for classification as well as a column called student_mentor that indicates roles.

Output:
• The GUI shows the following metrics: F1 Score, Accuracy, Precision, and Recall.
• Classification Outcomes:
  • Students: Rows with student_mentor = 1 are categorized as students.
  • Mentors: Rows with student_mentor = 0 are designated as mentors.
Execution Time: The amount of time needed to finish classifying.

.

Variables Used:

- **students**: Filtered DataFrame containing rows where student_mentor = 1.
- **mentors**: Filtered DataFrame containing rows where student_mentor = 0.
- **model**: RandomForestClassifier object used for classification.
- **question_columns**: List of feature column names used for classification.
- **classification_time**: Time taken to complete the classification process.
- **X, y**: Feature matrix and target variable array.
- **X_train, X_test, y_train, y_test**: Split datasets for training and testing.

y_pred: Predicted labels from the model.

Pseudocode:

```
BEGIN FUNCTION classify_roles
  BEGIN
    RECORD the current time as `start_time` to track execution time.
  END

  BEGIN
    DECLARE global variables `students`, `mentors`, `model`, `question_columns`, and `classification_time`.
  END

  BEGIN
    IDENTIFY all columns in the DataFrame `df` except 'sr_no' and 'student_mentor', and STORE them in `question_columns`.
    EXTRACT values from `df` for the identified `question_columns` and STORE them in `X`.
    EXTRACT values from `df` for the column 'student_mentor' and STORE them in `y`.
  END

  BEGIN
    SPLIT the data in `X` and `y` into training and testing sets:
      - Use 70% of the data for training and 30% for testing.
      - SET random state to 42 for reproducibility.
```

```
        STORE the results as `X_train`, `X_test`, `y_train`, and `y_test`.
    END

    BEGIN
        INITIALIZE a RandomForestClassifier with 100 estimators and random state 42.
        TRAIN the `model` using `X_train` and `y_train`.
    END

    BEGIN
        USE the trained `model` to PREDICT labels for `X_test` and STORE the results in
`y_pred`.
    END

    BEGIN
        CALCULATE the following metrics based on `y_test` and `y_pred`:
            - ACCURACY as `accuracy_score(y_test, y_pred) * 100`.
            - PRECISION as `precision_score(y_test, y_pred, average='binary') * 100`.
            - RECALL as `recall_score(y_test, y_pred, average='binary') * 100`.
            - F1 SCORE as `f1_score(y_test, y_pred, average='binary') * 100`.
        UPDATE the GUI label `accuracy_label` to display these metrics.
    END

    BEGIN
        FILTER rows in `df` where 'student_mentor' equals 1 and STORE them in `students`.
        FILTER rows in `df` where 'student_mentor' equals 0 and STORE them in `mentors`.
    END

    BEGIN
        UPDATE the GUI label `students_count` to show the total number of students.
        UPDATE the GUI label `mentors_count` to show the total number of mentors.
    END
```

.

BEGIN RECORD the current time as `end_time`. CALCULATE the total classification time as the difference between `end_time` and `start_time`. END BEGIN PRINT the message: "Classification completed in {classification_time:.4f} seconds". UPDATE the GUI label `speed_label` to display `classification_time`. END END FUNCTION

Table 4. 3: Match Similarity

Procedure Name : match_similarity
Action : pseudo code **for match_similarity Function**
Purpose: To calculate and display the similarity scores between students and mentors using cosine similarity, identifying the top matches for each student.
Input: • **students**: DataFrame containing rows classified as students. • **mentors**: DataFrame containing rows classified as mentors. • **question_columns**: List of columns representing the feature set for similarity calculations.
Output: • **match_result**: A list containing student-mentor matches with similarity scores and ranks. • **Similarity Results Display**: Matches displayed in the GUI, including student ID, mentor ID, similarity score, and rank. • **Execution Time**: Time taken to complete the matching process displayed in the GUI.
Variables Used:

.

- **students**: Filtered DataFrame containing student rows.
- **mentors**: Filtered DataFrame containing mentor rows.
- **question_columns**: List of columns used for similarity calculation.
- **students_data**: Array of feature values from students.
- **mentors_data**: Array of feature values from mentors.
- **match_result**: List to store the results of matches.
- **similarity_scores**: Array of cosine similarity scores for a student against all mentors.
- **top_matches**: Indices of the top two mentors with the highest similarity scores for a student.
- **matching_time**: Time taken for the matching process.

Pseudocode:

```
BEGIN FUNCTION match_similarity
  BEGIN
    RECORD the current time as `start_time` to track execution time.
  END

  BEGIN
    DECLARE global variables `students`, `mentors`, `model`, `question_columns`, and `classification_time`.
  END

  BEGIN
    IF `students` is empty OR `mentors` is empty THEN
      PRINT "Please load and classify data first."
      RETURN from the function.
    END IF
  END

  BEGIN
    EXTRACT feature values from `students` for `question_columns` and STORE them in `students_data`.
```

```
    EXTRACT feature values from `mentors` for `question_columns` and STORE them in `mentors_data`.
  END

  BEGIN
    INITIALIZE an empty list `match_result` to store match results.
  END

  BEGIN
    FOR EACH student index `i` and `student` in the 'sr_no' column of `students`:
      CALCULATE `similarity_scores` as the cosine similarity between `students_data[i]` and `mentors_data`.
      FIND the indexes of the top 2 matches in `similarity_scores`, sorted in descending order, and STORE them in `top_matches`.

      BEGIN
        FOR EACH `rank` and `match_idx` in `top_matches`:
          APPEND the tuple (`student`, `mentors[match_idx]['sr_no']`, `similarity_scores[match_idx]`, `rank + 1`) to `match_result`.
        END FOR
      END
    END FOR
  END

  BEGIN
    CALL the `display_matches` function with `match_result` to display the matching results.
  END

  BEGIN
    RECORD the current time as `end_time`.
```

.

CALCULATE `matching_time` as the difference between `end_time` and `start_time`. END BEGIN PRINT the message: "Matching completed in {matching_time:.4f} seconds." UPDATE the GUI label `speed_label` to display both `classification_time` and `matching_time`. END END FUNCTION

Table 4. 4: Classify Roles

Procedure Name : classify_roles
Action : pseudo code for the Program
Purpose: The program provides a GUI-based solution for role classification and similarity matching between students and mentors based on Excel data. It performs machine learning classification using a Random Forest model and calculates cosine similarity for matching purposes.
Input: 1. **Excel File**: Loaded by the user, containing data with features and labels. 2. **Feature Columns**: Columns used for training the classification model and similarity matching.
Output: 1. **Metrics**: Accuracy, Precision, Recall, and F1 Score of the classification model. 2. **Classified Data**: • **Students**: Rows classified as students (student_mentor = 1). • **Mentors**: Rows classified as mentors (student_mentor = 0). 2. **Matching Results**: Top mentor matches for each student, displayed in the GUI.
Variables Used:

.

- **df**: Global variable to store the loaded Excel data.
- **students**: Global variable to store rows classified as students.
- **mentors**: Global variable to store rows classified as mentors.
- **model**: Random Forest classifier used for role classification.
- **question_columns**: List of columns used for classification and similarity matching.
- **classification_time**: Time taken to perform classification.
- **match_result**: List to store similarity match details.
- **X, y**: Feature matrix and target labels for classification.
- **X_train, X_test, y_train, y_test**: Split datasets for training and testing.
- **y_pred**: Predicted labels from the model.

Pseudocode:

```
BEGIN PROGRAM
  BEGIN IMPORT NECESSARY LIBRARIES
    IMPORT pandas AS pd for data manipulation.
    IMPORT tkinter for creating the GUI.
    IMPORT sklearn for machine learning models and metrics.
    IMPORT time for measuring execution time.
  END IMPORT

  BEGIN DECLARE GLOBAL VARIABLES
    DECLARE `df` for storing the loaded dataset.
    DECLARE `students` and `mentors` for classified data.
    DECLARE `model` for the machine learning classifier.
    DECLARE `question_columns` for relevant feature columns.
    DECLARE `classification_time` for recording execution time.
  END DECLARE

  BEGIN DEFINE FUNCTION load_excel
    PROMPT the user to select an Excel file using a file dialog.
    LOAD the selected Excel file into `df` using `pandas.read_excel`.
    CALL the `display_data` function to display the contents of `df`.
    PRINT "Excel file loaded successfully."
```

```
    PRINT the column names of `df`.
  END DEFINE FUNCTION

  BEGIN DEFINE FUNCTION display_data
    CLEAR any existing content in the tree view display.
    SET the tree view headings to match the columns in `df`.
    INSERT each row of `df` into the tree view for display.
  END DEFINE FUNCTION

  BEGIN DEFINE FUNCTION classify_roles
    RECORD the current time as `start_time` to measure execution time.
    IDENTIFY relevant columns in `df` (excluding 'sr_no' and 'student_mentor') and STORE them in `question_columns`.
    EXTRACT values for `question_columns` into `X` and values for 'student_mentor' into `y`.
    SPLIT `X` and `y` into training and test datasets.
    INITIALIZE a `RandomForestClassifier`.
    TRAIN the `model` using the training data.
    PREDICT the labels for the test data using the `model`.
    CALCULATE metrics: accuracy, precision, recall, and F1 score.
    DISPLAY these metrics on the GUI.
    FILTER rows where 'student_mentor' equals 1 into `students` and 0 into `mentors`.
    DISPLAY the total number of students and mentors on the GUI.
    CALCULATE the classification time and display it.
  END DEFINE FUNCTION

  BEGIN DEFINE FUNCTION match_similarity
    RECORD the current time as `start_time` to measure execution time.
    IF `students` or `mentors` are empty:
      PRINT "Please load and classify data first."
      RETURN from the function.
    END IF
```

```
    EXTRACT values for `question_columns` from `students` into `students_data`.
    EXTRACT values for `question_columns` from `mentors` into `mentors_data`.
    INITIALIZE an empty list `match_result` to store match details.
    FOR EACH student in `students`:
      CALCULATE cosine similarity scores with all mentors.
      IDENTIFY the top 2 matches based on similarity scores.
      ADD the student ID, mentor ID, similarity score, and rank to `match_result`.
    END FOR
    CALL the `display_matches` function to display match results.
    CALCULATE and DISPLAY the matching time.
  END DEFINE FUNCTION

  BEGIN DEFINE FUNCTION display_matches
    CLEAR any existing content in the match results tree view.
    INSERT each entry from `match_result` into the tree view.
  END DEFINE FUNCTION

  BEGIN INITIALIZE GUI
    CREATE the main application window.
    ADD buttons for "Load Excel File", "Classify Roles", and "Match Similarity".
    ADD labels to display accuracy, classification time, and student/mentor counts.
    ADD a tree view for displaying dataset contents.
    ADD another tree view for displaying match results.
    ASSIGN the appropriate functions to button actions.
    START the GUI's main event loop to run the application.
  END INITIALIZE

END PROGRAM
```

Table 4. 5: Train Dataset

Procedure Name : train_dataset

.

Action : pseudo code **train_dataset Function**

Purpose:

To prepare the dataset, train a Random Forest classification model using labeled data, and return the trained model along with the test dataset for evaluation.

Input:

1. **df**: DataFrame containing the dataset with features and target labels.
    - **Relevant Columns**: Feature columns (excluding sr_no and student_mentor).

Target Column: student_mentor (binary classification target).

Output:

1. **model**: A trained Random Forest model.
2. **X_test**: Test set features for evaluation.
3. **y_test**: Test set labels for evaluation.

Variables Used:

- **model**: Random Forest classifier instance.
- **question_columns**: List of relevant feature columns in the dataset.
- **X, y**: Feature matrix and target variable array.
- **X_train, X_test, y_train, y_test**: Training and test sets split from the dataset.

Key Steps in the Process:

1. **Data Preparation**: The function identifies the features and target columns, ensuring only relevant data is used for training.
2. **Splitting Data**: The dataset is divided into training and testing subsets for model evaluation.
3. **Model Training**: A Random Forest classifier is created and trained on the training data.
4. **Returning Results**: The function returns the trained model and test data for evaluation or further processing.

Pseudocode:

```
BEGIN FUNCTION train_dataset
  BEGIN
    DECLARE global variables `model` and `question_columns`.
```

```
END

BEGIN
  IDENTIFY all columns in the DataFrame `df` except 'sr_no' and 'student_mentor'.
  STORE these identified columns in `question_columns`.
END

BEGIN
  EXTRACT values from `df` for the columns in `question_columns` and STORE them in `X`.
  EXTRACT values from `df` for the column 'student_mentor' and STORE them in `y`.
END

BEGIN
  SPLIT the data in `X` and `y` into training and test sets:
    - USE 70% of the data for training and 30% for testing.
    - SET random state to 42 for reproducibility.
  STORE the results in `X_train`, `X_test`, `y_train`, and `y_test`.
END

BEGIN
  INITIALIZE `model` as a RandomForestClassifier with the following parameters:
    - NUMBER OF ESTIMATORS set to 100.
    - RANDOM STATE set to 42 for reproducibility.
END

BEGIN
  TRAIN the `model` using the training data (`X_train` and `y_train`).
  PRINT "Model training complete" to indicate the training process has finished.
END

BEGIN
```

.

RETURN the trained `model` along with `X_test` and `y_test` for evaluation. END END FUNCTION

Table 4. 6: Test Dataset

Procedure Name : test_dataset(model, X_test, y_test)
Action : pseudo code for **test_dataset Function**
Purpose: To evaluate the performance of a trained model on a test dataset by calculating key metrics: accuracy, precision, recall, and F1-score.
Input: 1. **model**: The trained Random Forest classification model. 2. **X_test**: Test set features used for predictions. 3. **y_test**: Test set labels (ground truth) for evaluation.
Output: 1. **Metrics**: • **accuracy**: Percentage of correct predictions. • **precision**: Proportion of positive predictions that were correct. • **recall**: Proportion of actual positives correctly identified. • **f1_score**: Harmonic mean of precision and recall. 2. **Display**: • Printed metrics for user visibility.
Variables Used: • **y_pred**: Predicted labels from the model for X_test. • **accuracy**: Calculated accuracy of the model. • **precision**: Calculated precision of the model. • **recall**: Calculated recall of the model. • **f1_score**: Calculated F1-score of the model.
Key Steps in the Process: 1. **Prediction**: Use the trained model to predict test labels.

.

2. **Metric Calculation**: Compute accuracy, precision, recall, and F1-score to evaluate the model's performance.
3. **Display Results**: Print the metrics to the console for user understanding.
4. **Return Metrics**: Provide the calculated metrics for further use or integration.

Pseudocode:

```
BEGIN FUNCTION test_dataset(model, X_test, y_test)
  BEGIN
    PREDICT the labels for the test data `X_test` using the provided `model`.
    STORE the predicted labels in `y_pred`.
  END

  BEGIN
    CALCULATE the accuracy as the percentage of correct predictions:
      - USE `accuracy_score(y_test, y_pred)` and MULTIPLY by 100.
    CALCULATE the precision as the percentage of true positive predictions:
      - USE `precision_score(y_test, y_pred, average='binary')` and MULTIPLY by 100.
    CALCULATE the recall as the percentage of actual positives correctly identified:
      - USE `recall_score(y_test, y_pred, average='binary')` and MULTIPLY by 100.
    CALCULATE the F1 score as the harmonic mean of precision and recall:
      - USE `f1_score(y_test, y_pred, average='binary')` and MULTIPLY by 100.
  END

  BEGIN
    DISPLAY the calculated metrics:
      PRINT "Accuracy: {accuracy:.2f}%"
      PRINT "Precision: {precision:.2f}%"
      PRINT "Recall: {recall:.2f}%"
      PRINT "F1 Score: {f1_score:.2f}%"
  END

  BEGIN
    RETURN the calculated metrics:
```

```
        - RETURN `accuracy`, `precision`, `recall`, and `f1_score`.
    END


END FUNCTION
```

4.13 Generating the User Interface for study of different models

I have created a graphical user interface application called Nehal's Mentor Recommendation System (NMRS) to recommend mentors based on the responses provided by mentors and mentees in the form. This user interface advises the mentee to select the most suitable mentor depending on his or her routine habits and psychological thinking.

The efficacy, advantages, disadvantages, and applications of classification methods such as K-Nearest Centroid Classifier, Naive Bayes, One-Class SVM, and K-Means Classification are assessed in a comparative analysis using my datasets.

4.13.1 Comparative Analysis

- Z Algorithm
- Rabin Karp
- Naïve
- Knuth-Morris-Pratt (KMP) algorithm
- Brute-Force
- Boyer-Moore

.

- Open NMRS - Mentor Matching Model

Figure 4. 3: NMRS – Mentor Matching Model

- Load Excel File

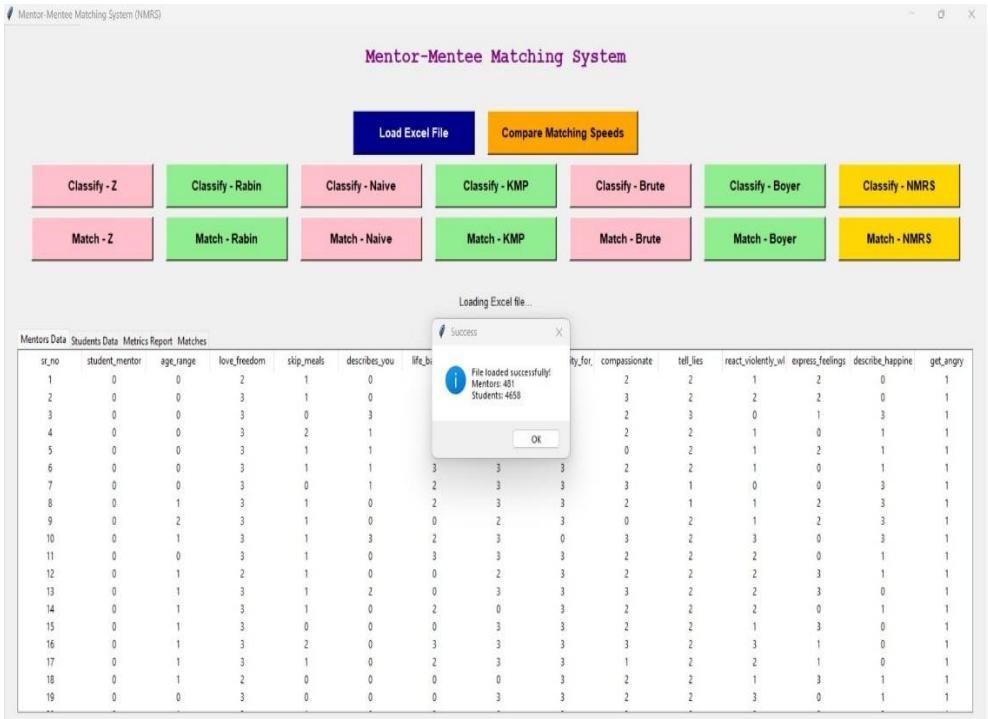


Figure 4. 4: Load Excel File

.

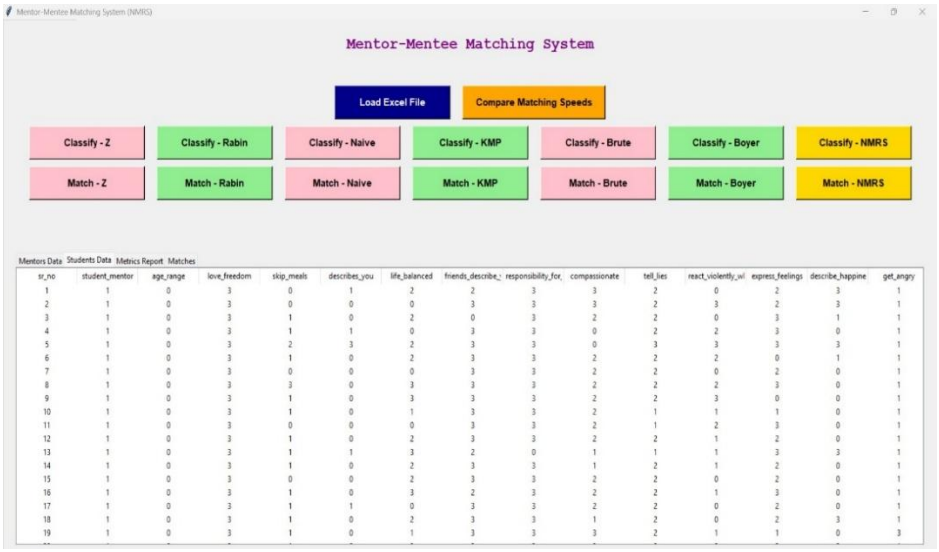


Figure 4. 5: Load Excel File

❖ Z-Algorithm

An effective string-matching method for locating every instance of a pattern in a text is the Z-Algorithm. The concatenated string created by connecting the pattern and text, separated by a special delimiter (such as "P$T"), is preprocessed. The Z-array is calculated during preprocessing, and each element Z[i] denotes the length of the largest substring beginning at position i that corresponds to the string's prefix. The algorithm then looks for values in the Z-array that match the pattern's length, suggesting precise textual matches.

Classification

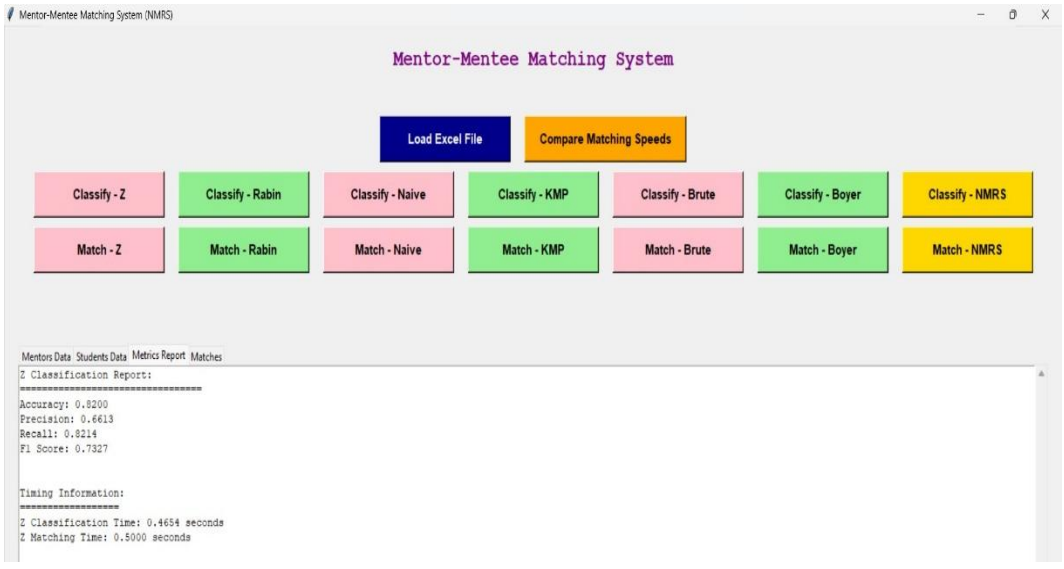


Figure 4. 6: Classification of Z Algorithm

.

Matching Similarity

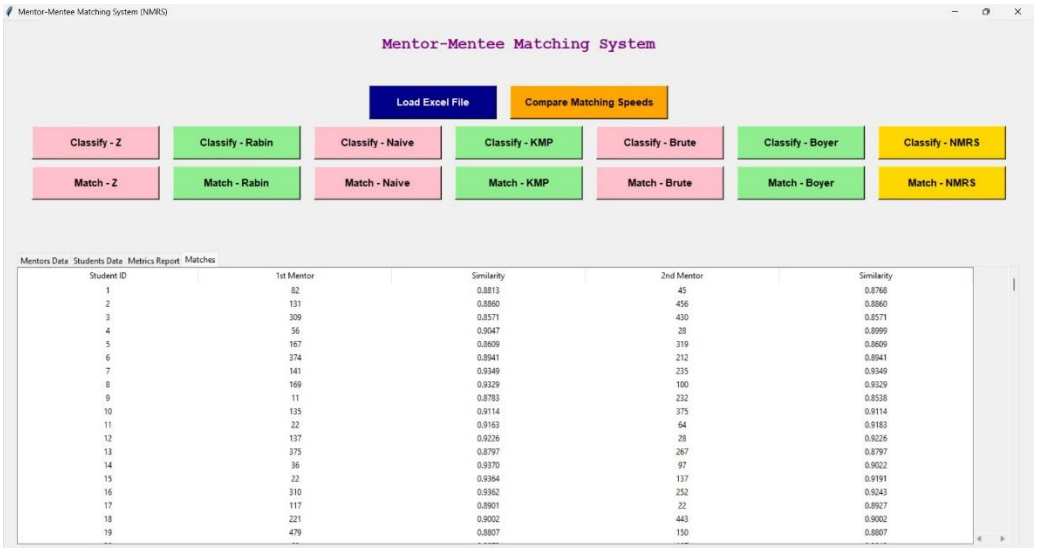


Figure 4. 7: Matching Similarity of Z Algorithm

❖ Rabin Karp

A string-matching method called the Rabin-Karp algorithm effectively finds a pattern in a text by using hashing. For the pattern and every text substring of the same length as the pattern, it calculates a hash value. The technique swiftly finds possible matches by comparing the hash values. Because hash collisions can happen, it confirms the match by directly comparing the hash values character by character. By reusing the previous hash, the next substring's hash can be computed in constant time thanks to the optimization of the hash computation through the use of a rolling hash function.

Classification

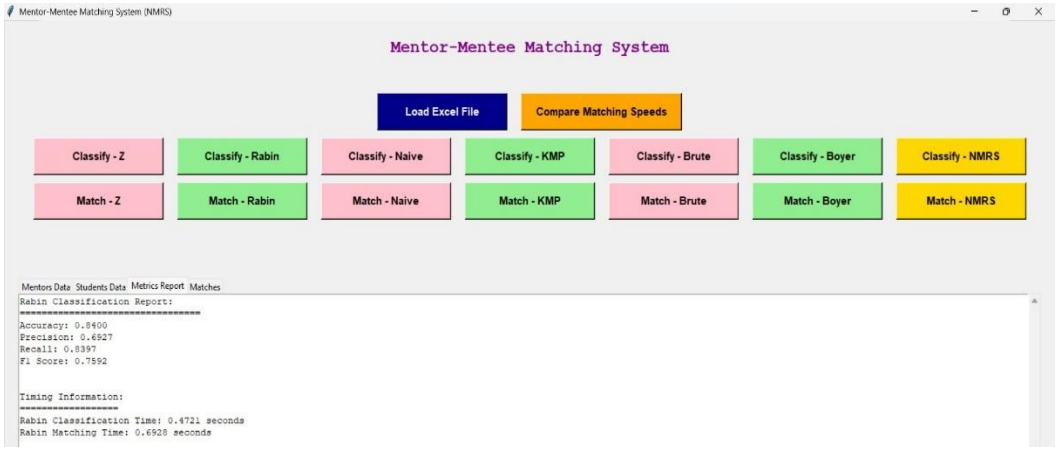


Figure 4. 8: Classification of Rabin Karp

.

Matching Similarity

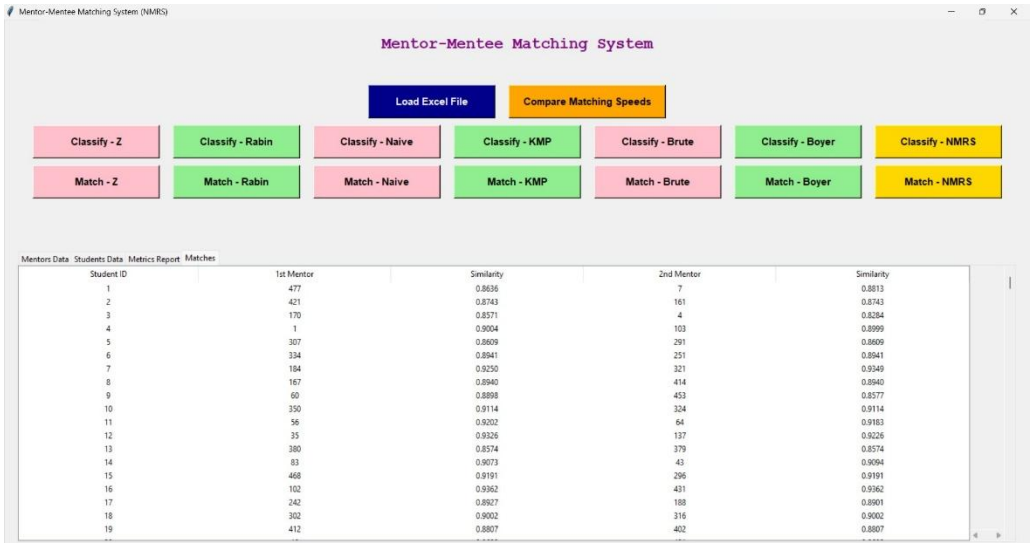


Figure 4. 9: Matching Similarity of Rabin Karp

❖ Naïve

The simplest method for identifying a pattern in a text is the Naïve string-matching algorithm. By iterating over every potential starting point in the text and comparing each pattern character with its associated textual character, it looks for the pattern. The pattern is deemed to have been discovered at that location if every character matches. The program continues the comparison after shifting the pattern by one location if there is a discrepancy.

Classification

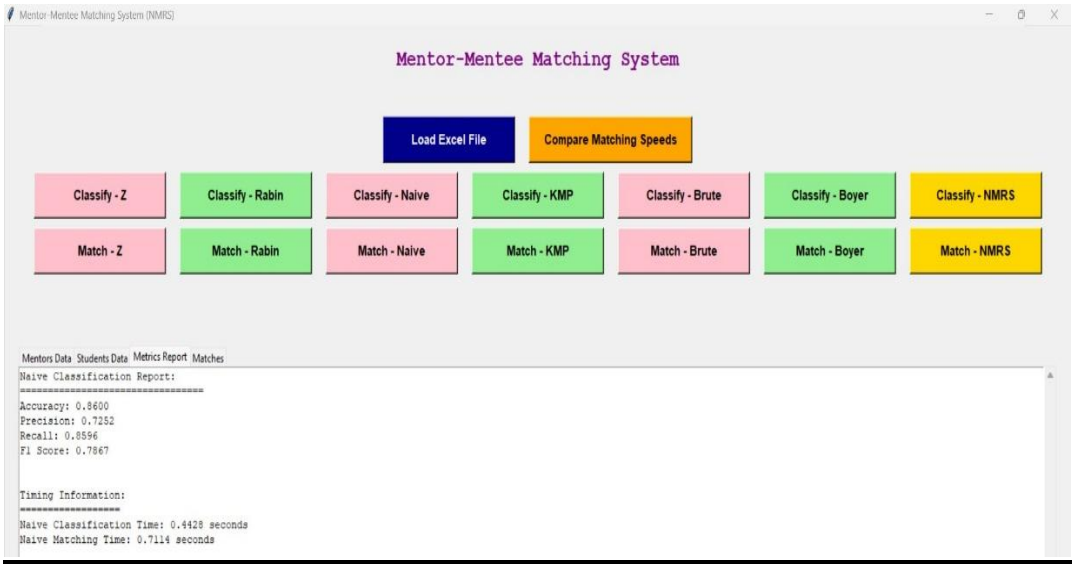
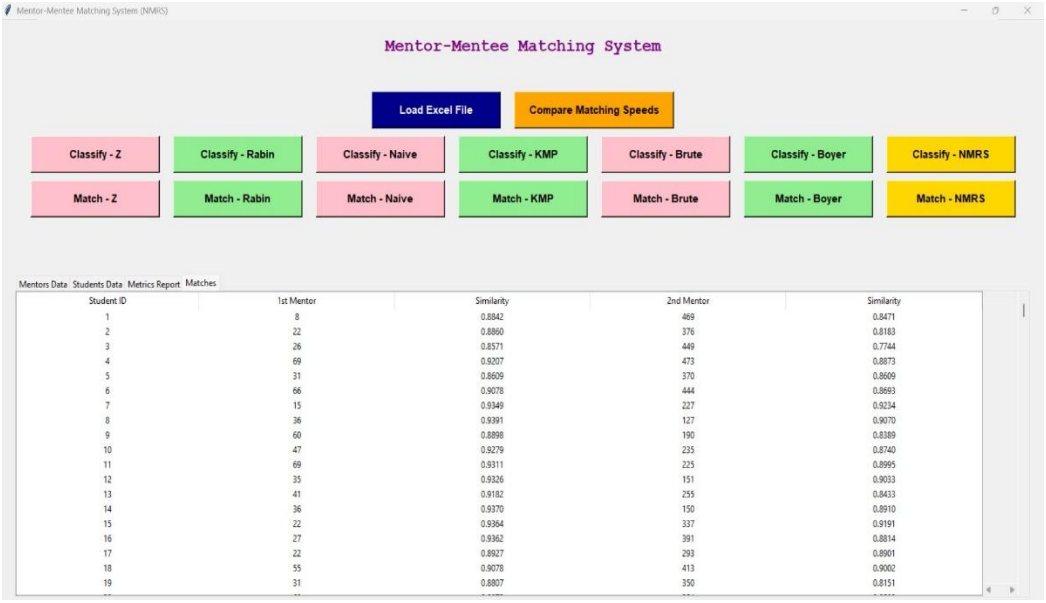


Figure 4. 10: Classification of Naive

.

Matching Similarity



Student ID	1st Mentor	Similarity	2nd Mentor	Similarity
1	8	0.8842	469	0.8471
2	22	0.8860	376	0.8183
3	26	0.8571	449	0.7744
4	69	0.9207	473	0.8873
5	31	0.8609	370	0.8609
6	66	0.9078	444	0.8693
7	15	0.9349	227	0.9234
8	36	0.9391	127	0.9070
9	60	0.8898	190	0.8389
10	47	0.9279	235	0.8740
11	69	0.9311	225	0.8995
12	35	0.9326	151	0.9033
13	41	0.9182	255	0.8433
14	36	0.9370	150	0.8910
15	22	0.9364	337	0.9191
16	27	0.9362	391	0.8814
17	22	0.8927	293	0.8901
18	55	0.9078	413	0.9002
19	31	0.8807	350	0.8151

Figure 4. 11: Matching Similarity of Naive

❖ Knuth-Morris-Pratt (KMP) algorithm

By pre-processing the pattern to produce a Partial Match Table (or LPS array), the Knuth-Morris-Pratt (KMP) algorithm is an effective string-matching method that eliminates the need for duplicate comparisons. For every place, the length of the pattern's longest proper prefix that doubles as a suffix is stored in the LPS array. If a mismatch arises during the search, the algorithm moves the pattern to the correct location using the LPS array without going back to look at previously matched characters.

Classification

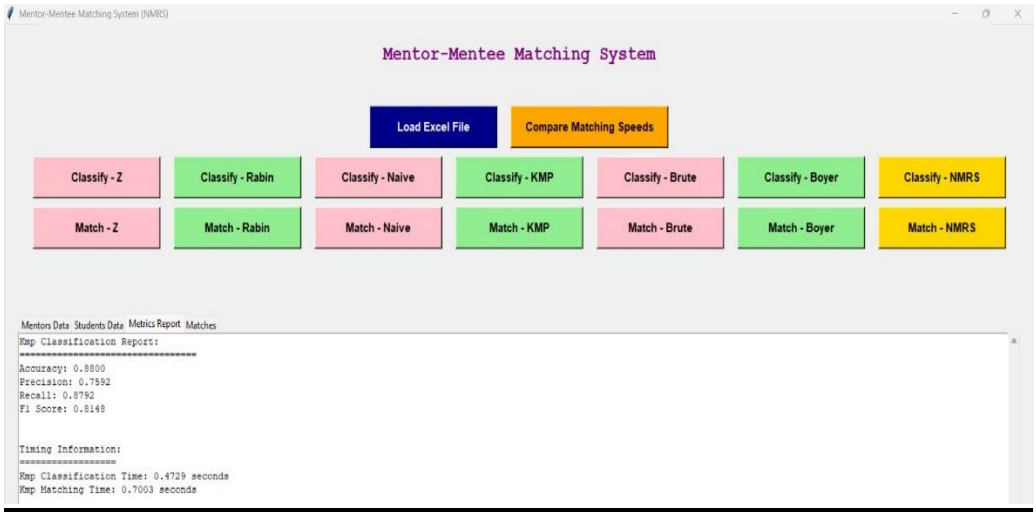


Figure 4. 12: Classification of KMP

.

Matching Similarity

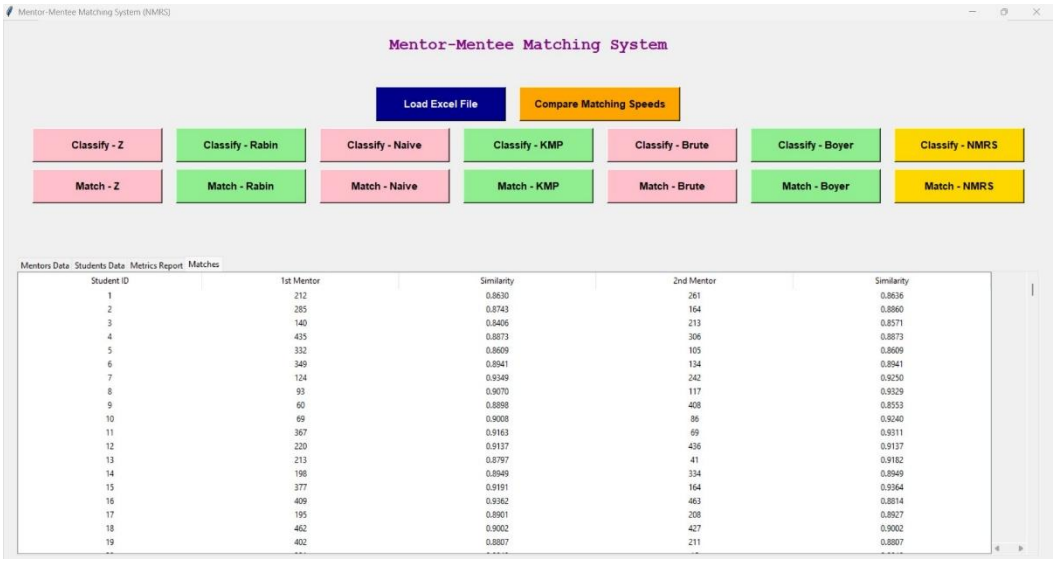


Figure 4. 13: Matching Similarity of KMP

❖ Brute-Force

By methodically comparing the pattern with every potential starting point in the text, the Brute Force algorithm finds two strings that match. It sequentially checks each pattern character with its matching textual characters for every place. The pattern is deemed to have been discovered at that location if every character matches. The method restarts the comparison after shifting the pattern by one location if there is a discrepancy.

Classification

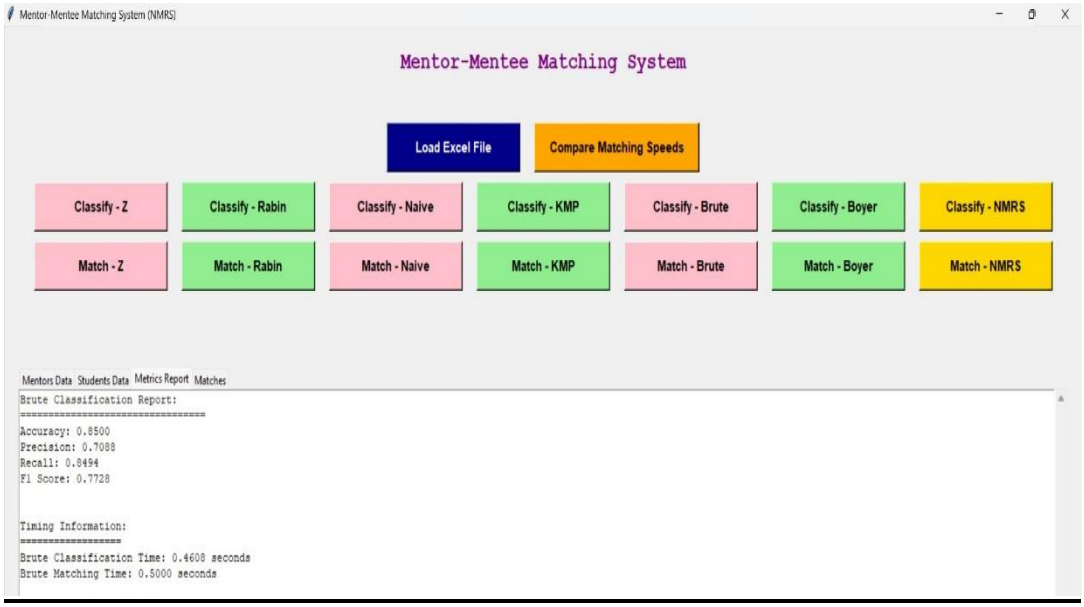


Figure 4. 14: Classification of Brute Force

.

Matching Similarity

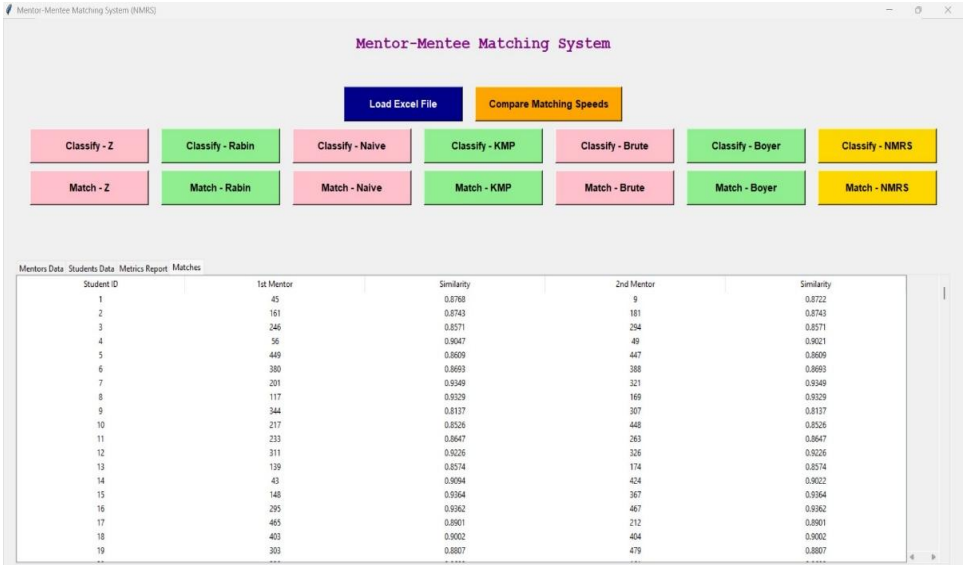


Figure 4. 15: Matching Similarity of Brute Force

❖ Boyer-Moore

The Boyer-Moore algorithm is a very effective string-matching method that skips textual passages during comparisons by using preprocessing. The Good Suffix Rule and the Bad Character Rule are the two heuristics it uses. The Bad Character Rule either skips the entire pattern if the character is not in it or moves the pattern to match the last instance of a mismatched character in the text. Based on the pattern's matched suffix and its repetitions elsewhere in the pattern, the Good Suffix Rule modifies the pattern. By letting the pattern skip over portions of the text instead of inspecting each character, these heuristics reduce the number of pointless comparisons.

Classification

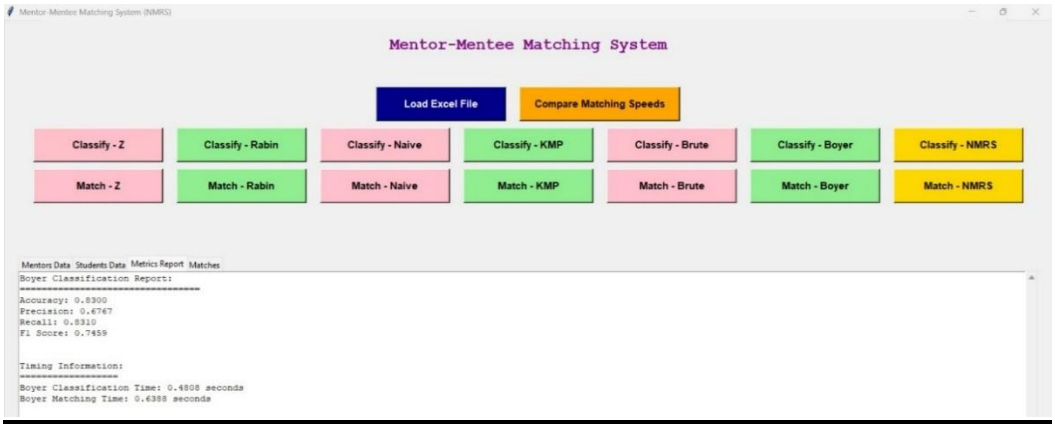


Figure 4. 16: Classification of Boyer Moore

.

Matching Similarity

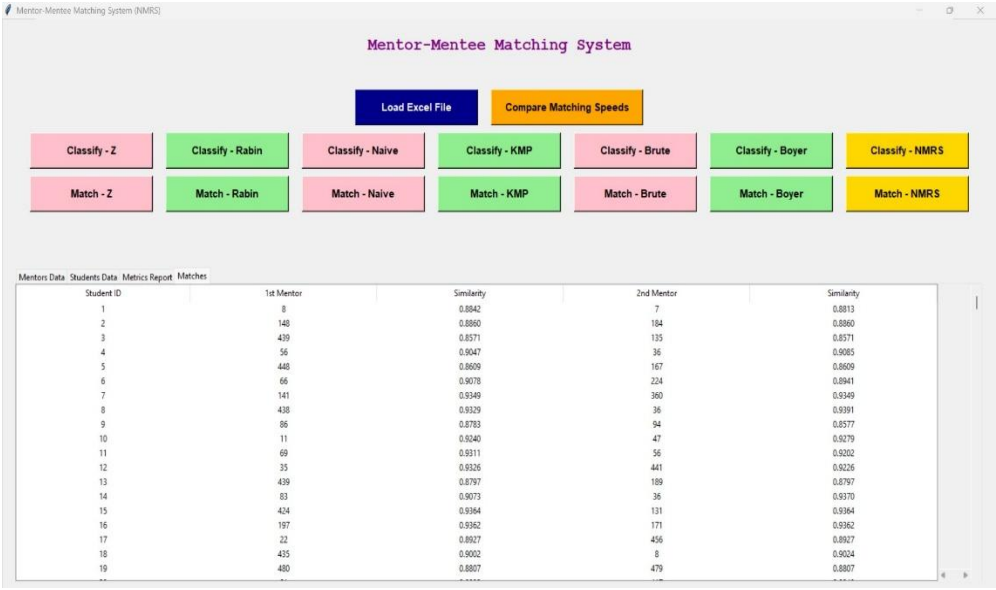


Figure 4. 17: Matching Similarity of Boyer Moore

4.13.2 NMRS Model

Features of NMRS Model

- Response Combination: Combines and classifies mentors' and mentees' answers to enable organized comparisons across a variety of topics and categories.
- Pairwise Correspondence: Uses a methodical comparison of mentors' (C1) and mentees' (C2) responses to assess compatibility.
- Comprehensive Pairwise Analysis: Guarantees a thorough matching procedure by comparing each mentor with each mentee.
- Compatibility Rating: Provides a mathematical foundation for pairing by calculating a match score based on the amount of matching responses between each mentor and mentee.
- Allocation Based on Priority: Ensures optimal compatibility by linking mentees with their top two mentors based on highest matching scores.

.

Classify Rates

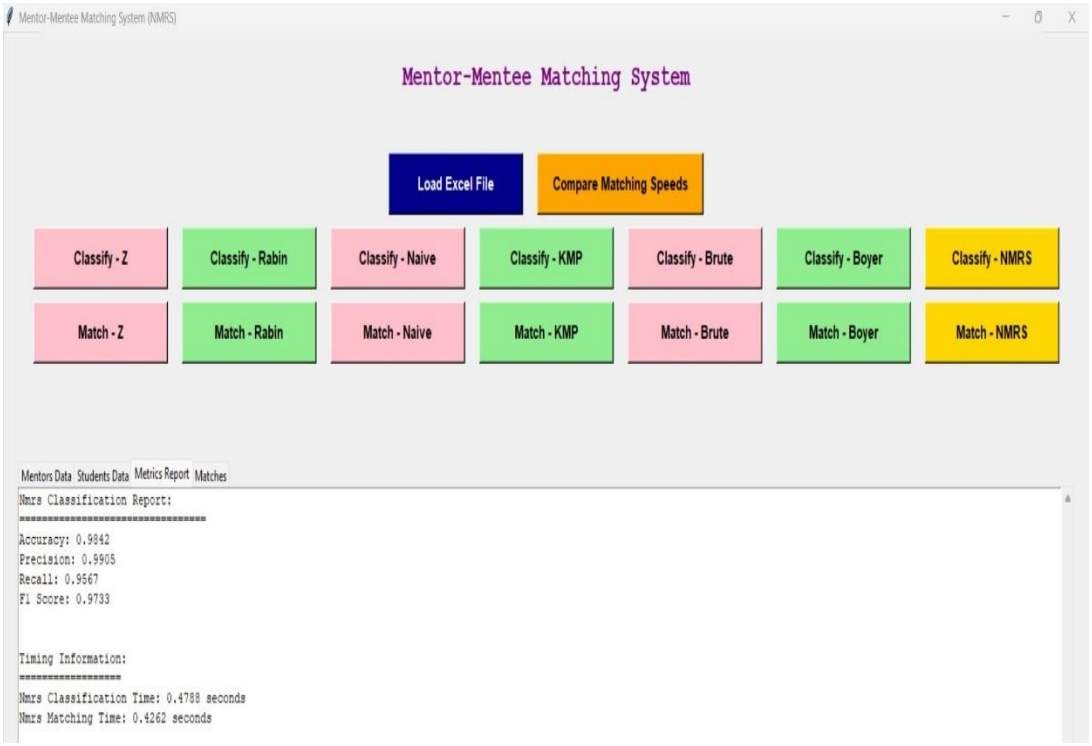


Figure 4. 18: Classification of NMRS

Matching Similarity

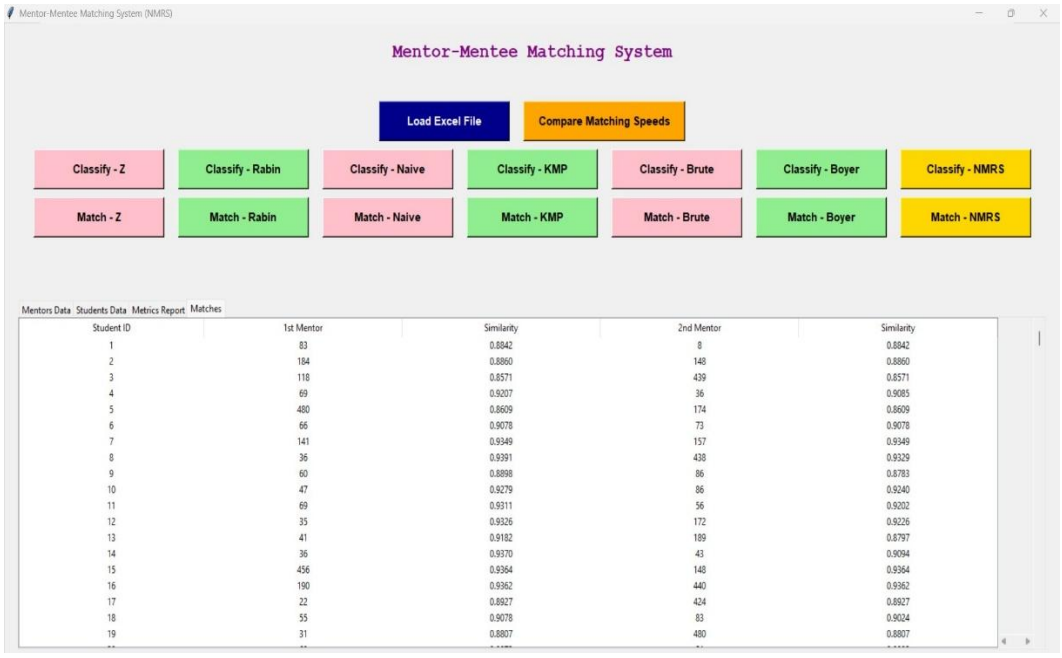


Figure 4. 19: Matching Similarity of NMRS

.

Comparative Analysis

Table 4. 7: Comparative Analysis

		Classify Rate			Matching Similarity
	Accuracy	**Precision**	**Recall**	**F1 Score**	**Speed in Seconds**
Boyer-Moore	83%	0.6767	0.8310	0.7459	0.6388
Knuth-Morris-Pratt (KMP)	88%	0.7592	0.8792	0.8148	0.7003
Naive Bayes	86%	0.7252	0.8596	0.7867	0.7114
Brute Force	85%	0.7088	0.8494	0.7728	0.5000
Rabin karp	84%	0.6927	0.8397	0.7592	0.6928
Z Algorithm	82%	0.6613	0.8214	0.7327	0.5000
NMRS	98%	0.9905	0.9567	0.9733	0.4262