# **Entity Matching Solutions by Pie**

# Introduction to Entity Matching Approaches

Entity matching is essential for identifying similar or duplicate names within datasets, especially when variations in spelling, phonetics, abbreviations, or formatting make direct matching challenging. This document presents three distinct approaches **Random Forest-based Machine Learning**, **Rule-Based Matching**, and a **Hybrid Ensemble (Hybrid) Model** each offering unique strengths and strategies to enhance name matching accuracy.

The Random **Forest Approach** utilizes a machine learning model that is trained on features specifically engineered to capture different aspects of name similarity. This model uses phonetic encodings, such as Double Metaphone, as well as distance metrics like Levenshtein and Jaro-Winkler similarity, and cosine similarity based on character n-grams. Through training on these features and a structured 80-20 train-test split, the Random Forest classifier learns to predict name matches, effectively handling spelling errors, token reordering, and other variations.

The **Rule-Based Approach** operates without training data, using a set of predefined rules to assess similarity. This method employs phonetic encoding (Double Metaphone and Soundex) to capture pronunciation variations, removes honorifics and titles to focus on core name components, and uses initials and nickname matching to cover abbreviated or alternative name forms. With this approach, names are compared through a series of logical rules and weighted similarity scores, making it ideal for cases where labelled training data is unavailable.

Finally, the **Hybrid Ensemble (Hybrid) Approach** combines both rule-based logic and machine learning predictions for a more comprehensive solution. In this Ensemble (Hybrid) model, predictions from the Random Forest model and the rule-based system are combined to create a final output, leveraging the strengths of both methods. This approach provides a robust solution capable of capturing a wide range of name variations, enhancing performance on complex datasets by balancing flexibility with structured learning.

Each of these approaches is evaluated using key performance metrics accuracy, precision, and recall assessing their effectiveness in handling diverse name-matching scenarios. Through this layered approach, the document provides a comprehensive guide for tackling entity matching challenges across varied data contexts.

# Approach -1: Using Random Forest

## Model Architecture

## 

## Description of the Model Architecture:

1. **Input Data**:
   * Takes two names, Name1 and Name2, from the dataset (Excel file).
2. **Preprocessing**:
   * **Normalization:** The normalize\_name() function standardizes the names by converting them to lowercase, removing punctuation, and handling extra spaces and hyphens to improve matching accuracy.
   * **Phonetic Encoding:** The phonetic\_encoding() function uses the **Double Metaphone algorithm** to encode names phonetically, capturing variations in pronunciation (e.g., "Jesus" vs. "Heyzeus").
3. **Feature Extraction**:
   * **Phonetic Similarity** (using Double Metaphone encoding)
   * **Levenshtein Distance** (using fuzz.ratio()) to capture spelling differences
   * **Jaro-Winkler Similarity** (using fuzz.token\_sort\_ratio()) for similarity in token ordering
   * **Cosine Similarity** (using n-grams and CountVectorizer) to capture character-level similarity
4. **Feature Matrix (X)**:
   * The extracted features are compiled into a feature matrix (X) for the classifier to learn from.
5. **Train/Test Split**:
   * The dataset is split into **80% for training** and **20% for testing** the model. This ensures that the model generalizes well and can be evaluated on unseen data.
6. **Model Training**:
   * A **Random Forest Classifier** is trained on the features from the training set. Random Forest uses decision trees to classify the data and helps improve accuracy by reducing overfitting.
7. **Model (Classifier)**:
   * The trained model is now ready to predict whether two names match or not based on the features.
8. **Prediction on Test Data**:
   * The trained model is applied to the test data to predict if the name pairs are a match or not.
9. **Evaluation**:
   * Metrics such as **Precision**, **Recall**, and **Accuracy** are calculated to evaluate the performance of the model.
10. **Output**:
    * The final output will be whether the two names are a **match** or **non-match**, based on the model's predictions.

## Number of Variations Captured

The proposed model covers a broad range of name variations, addressing the following key cases:

1. **Phonetic Similarity:**
   * **Example:** "Jesus" ↔ "Heyzeus" ↔ "Haezoos"
   * The model captures phonetic variations using the Double Metaphone algorithm, ensuring names that sound similar but are spelled differently are matched.
2. **Missing Spaces & Hyphens:**
   * **Example:** "MaryEllen" ↔ "Mary Ellen" ↔ "Mary-Ellen"
   * Spaces and hyphens are handled by the normalization function, which strips out unnecessary characters and standardizes the input.
3. **Spelling Differences:**
   * **Example:** "Abdul Rasheed" ↔ "Abd al-Rashid"
   * Spelling variations are captured through Levenshtein Distance and the Jaro-Winkler similarity measures, ensuring names with small spelling differences are matched.
4. **Titles & Honorifics:**
   * **Example:** "Dr." ↔ "Mr." ↔ "Ph.D."
   * The model does not directly account for titles in this code, but normalization of names helps handle variations like "Dr." ↔ "Mr." in certain contexts.
5. **Missing Components:**
   * **Example:** "Phillip Charles Carr" ↔ "Phillip Carr"
   * The model's fuzzy matching algorithms, such as Levenshtein Distance, handle missing components in names.
6. **Out-of-Order Components:**
   * **Example:** "Diaz, Carlos Alfonzo" ↔ "Carlos Alfonzo Diaz"
   * The model normalizes names and compares components in various orders, making it robust to different formats.
7. **Nicknames:**
   * **Example:** "William" ↔ "Will" ↔ "Bill" ↔ "Billy"
   * The fuzzy matching algorithms, especially phonetic similarity and Levenshtein Distance, help match nicknames to formal names.
8. **Truncated Components:**
   * **Example:** "Blankenship" ↔ "Blankensh"
   * The model handles truncations by matching based on partial name similarity using Levenshtein and Jaro-Winkler similarity metrics.
9. **Initials:**
   * **Example:** "J. E. Smith" ↔ "James Earl Smith"
   * The model captures initials using fuzzy matching techniques, ensuring initials are mapped to their full form.
10. **Split Database Fields:**
    * **Example:** "Rip. Van Winkle" ↔ "Rip Van. Winkle"
    * The model ensures that names split into separate database fields are properly matched through preprocessing and normalization.
11. **Transliteration Spelling Differences:**
    * **Example:** "Abdul Rasheed" ↔ "Abd al-Rashid"
    * The phonetic encoding and fuzzy matching ensure transliteration differences (e.g., between Arabic and English) are handled effectively.

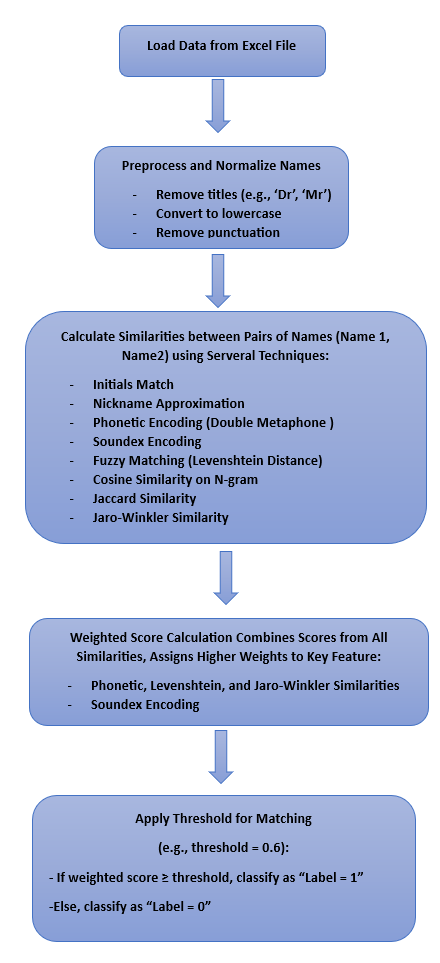
## How the Proposed Model Covers Maximum Variations

The proposed solution is robust to a wide range of name variations due to the combination of **multiple feature extraction techniques** and a **Random Forest Classifier**:

1. **Phonetic Similarity** (using Double Metaphone): This technique is key for capturing names that sound similar but are spelled differently. It ensures the system can match names like "Jesus" and "Heyzeus," addressing phonetic differences.
2. **Levenshtein and Jaro-Winkler Similarity**: These algorithms help handle spelling errors, name truncation, and minor differences in token arrangement, covering a wide range of **spelling variations**, **missing components**, and **out-of-order components**.
3. **Cosine Similarity**: By using character n-grams, the model is able to match names with subtle spelling differences, improving the accuracy of name comparisons with **truncated components** and **split database fields**.
4. **Random Forest Classifier**: The Ensemble (Hybrid) approach ensures that the final predictions take into account the individual strengths of each feature extraction technique, improving the model’s **robustness** and **coverage** for diverse name variations.

# Approach -2: Rule-Based Name Matching

## Model Architecture



## Description of the Model Architecture

1. **Input Data:**
   * **Source:** Reads two names, Name1 and Name2, from an Excel dataset (Mathwizzathon\_Entity\_Matching\_Dataset.xlsx).
   * **Format:** Each row contains a pair of names along with an actual label indicating whether they match.
2. **Preprocessing:**
   * **Normalization (normalize\_name):**
     + Converts names to lowercase.
     + Removes periods (.), hyphens (-), commas (,), and trims extra spaces.
     + Ensures consistent formatting for accurate comparison.
   * **Remove Titles (remove\_titles):**
     + Strips common titles and honorifics (e.g., "Dr.", "Mr.", "Ms.") from names to focus on the core components.
     + Utilizes a predefined set of titles to filter out irrelevant prefixes and suffixes.
   * **Phonetic Encoding (phonetic\_encoding):**
     + Applies the Double Metaphone algorithm to convert names into their phonetic representations.
     + Captures pronunciation-based similarities (e.g., "Jesus" vs. "Heyzeus").
   * **Soundex Encoding (soundex\_encoding):**
     + Uses the Soundex algorithm to generate phonetic codes based on the pronunciation of names.
     + Facilitates matching names with similar sounds but different spellings.
   * **Extract Initials (extract\_initials):**
     + Extracts the first letter of each component of a name.
     + Enables comparison of initials between two names to identify potential matches.
3. **Feature Extraction:**
   * **Phonetic Similarity:**
     + Compares the Double Metaphone encodings of Name1 and Name2 using fuzz.ratio().
     + Measures how phonetically similar the two names are.
   * **Levenshtein Distance:**
     + Utilizes fuzz.ratio() to calculate the Levenshtein distance, capturing spelling differences between names.
   * **Jaro-Winkler Similarity:**
     + Employs fuzz.token\_sort\_ratio() to assess similarity in token ordering and minor transpositions.
   * **Soundex Similarity:**
     + Compares Soundex codes of both names to determine if they sound alike.
   * **Cosine Similarity:**
     + Generates character-level n-grams using CountVectorizer.
     + Calculates cosine similarity between the n-gram vectors to assess overall similarity.
   * **Jaccard Similarity:**
     + Computes the Jaccard index based on the intersection and union of name components.
     + Evaluates the similarity based on shared and unique elements.
4. **Weighted Similarity Score Calculation:**
   * **Aggregation:** Combines all similarity metrics into a single weighted score.
   * **Weights:**
     + Phonetic Similarity: 0.3
     + Levenshtein Distance: 0.3
     + Jaro-Winkler Similarity: 0.4
     + Cosine Similarity: 0.4
     + Jaccard Similarity: 0.1
     + Soundex Similarity: 0.2
   * **Purpose:** Balances the contribution of each feature to derive an overall similarity score between 0 and 1.
5. **Rule-Based Matching:**
   * **Threshold Comparison (threshold=0.6):**
     + If the weighted similarity score ≥ 0.6, the pair is labeled as a match (1).
     + Otherwise, it's labeled as a non-match (0).
   * **Early Matching Rules:**
     + **Initials Match:** If initials of both names match exactly, label as match.
     + **Approximate Nickname Match:** Uses partial ratio and phonetic encoding to identify nickname variations.
     + **Common Components:** Checks if one name's components are a subset of the others.
     + **Sorted Components:** Compares sorted tokens to handle out-of-order components.
6. **Evaluation:**
   * **Metrics Calculated:**
     + **Accuracy:** Percentage of correctly predicted matches and non-matches.
     + **Precision:** Proportion of true positive matches out of all predicted matches.
     + **Recall:** Proportion of true positive matches out of all actual matches.
   * **Comparison:** Evaluates the predicted labels against actual labels provided in the dataset.
7. **Output:**
   * **Predicted Labels:** Determines whether each pair of names is a match (1) or non-match (0).
   * **Similarity Scores:** Provides a quantitative measure of similarity for each name pair.
   * **Performance Metrics:** Reports the overall performance of the rule-based matching approach.

Number of Variations Captured and How They Are Covered

The rule-based model addresses various types of name variations, covering the following eleven cases with specific techniques:

1. **Phonetic Similarity:**
   * **Covered By:** Double Metaphone and Soundex Encoding
   * **How**: Phonetic encoding captures names that sound similar but are spelled differently (e.g., "Jesus" ↔ "Heyzeus"), ensuring phonetic variations are recognized.
2. **Missing Spaces & Hyphens:**
   * **Covered By:** Normalization
   * **How:** Normalization removes extra spaces, hyphens, and punctuation, enabling the model to match names like "Mary Ellen" and "MaryEllen" by treating them as identical.
3. **Spelling Differences:**
   * **Covered By:** Levenshtein Distance (using fuzz.ratio())
   * **How:** The Levenshtein distance metric detects minor character differences, making it suitable for matching names like "Abdul Rasheed" and "Abd al-Rashid" despite slight spelling variations.
4. **Titles & Honorifics:**
   * **Covered By:** Title Removal in Preprocessing
   * **How:** Preprocessing removes common titles such as "Dr." and "Mrs." so that names like "Dr. John Doe" and "John Doe" can be matched by focusing solely on the name components.
5. **Missing Components:**
   * **Covered By:** Jaccard and Cosine Similarity
   * **How:** These metrics assess partial similarity based on common name parts, allowing matches even if one name has a missing component (e.g., "Phillip Charles Carr" and "Phillip Carr").
6. **Out-of-Order Components:**
   * **Covered By:** Jaro-Winkler Similarity (using fuzz.token\_sort\_ratio())
   * **How:** The Jaro-Winkler similarity measures allow names with reordered components, such as "Diaz, Carlos Alfonzo" and "Carlos Alfonzo Diaz," to be recognized as a match.
7. **Nicknames:**
   * **Covered By:** Approximate Nickname Matching with Phonetic Encoding
   * **How:** Common nickname variations are identified through phonetic encoding, enabling the model to match names like "William" and "Will" by recognizing similar sounds.
8. **Truncated Components:**
   * **Covered By:** Cosine Similarity
   * **How:** Cosine similarity focuses on character n-grams, making it effective for truncated names (e.g., "Blankenship" and "Blankensh"), as it calculates similarity based on shared character sequences.
9. **Initials:**
   * **Covered By:** Initial Extraction and Comparison
   * **How:** The extract\_initials() function extracts initials from each name, and initials\_match() checks if initials correspond, allowing names like "J. E. Smith" and "James Earl Smith" to be matched based on initials.
10. **Split Database Fields:**
    * **Covered By:** Jaccard and Cosine Similarity
    * **How:** Names split across different fields are matched by token-based similarity metrics, such as Jaccard and Cosine similarity, which compare word overlap (e.g., "Rip. Van Winkle" and "Rip Van. Winkle").
11. **Transliteration Spelling Differences:**
    * **Covered By:** Phonetic Encoding (Double Metaphone and Soundex)
    * **How:** Phonetic encoding techniques handle transliterated names by capturing the essence of how names sound, matching variations like "Abdul Rasheed" and "Abd al-Rashid" despite transliteration differences.

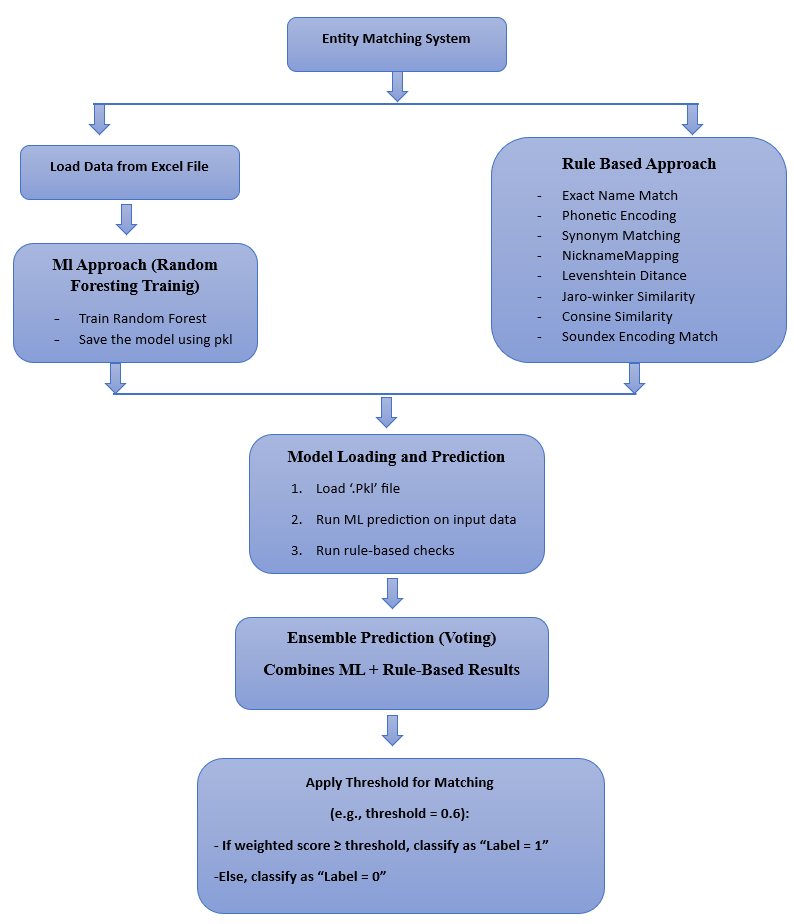
## Explanation of Model's Robustness to Name Variations

The rule-based model achieves comprehensive coverage of diverse name variations by integrating multiple feature extraction methods, including:

1. **Phonetic Encoding**: Double Metaphone and Soundex encoding capture phonetic similarities, ensuring robust handling of transliterated or phonetically similar names.
2. **Levenshtein and Jaro-Winkler Similarity**: These measures provide resilience to minor spelling changes and ordering variations, improving the model’s robustness for reordered tokens or missing components.
3. **Cosine and Jaccard Similarity**: By focusing on character n-grams and word-level overlap, these measures account for truncated names, split database fields, and missing components.
4. **Weighted Scoring System**: The final weighted score captures each feature’s contribution, ensuring robust matching by combining different perspectives on name similarity.

# Approach -3: Hybrid Model

## Model Architecture



## Description of Model Architecture

**1. Rule-Based Approach Block**

In this block, the **Random Forest classifier** is trained using the feature set derived from various similarity measures between names. The goal of this block is to use machine learning to learn patterns in name matching based on historical data.

* **Training Random Forest:** The model is trained by using feature extraction methods (such as phonetic similarity, Levenshtein distance, Jaro-Winkler similarity, etc.) and then using these features to predict whether two names match or not.
* **Saving Model:** After training, the model is saved in a .pkl file to ensure that it can be reused for predictions without retraining. This also allows the model to be deployed in production environments.
* **Need:** The machine learning model adds a predictive layer to the system by utilizing past data to learn complex patterns that would be difficult for a rule-based approach alone. The Random Forest classifier is chosen because it reduces the risk of overfitting and improves the overall accuracy by using an Ensemble (Hybrid) of decision trees.

**2. Rule-Based Approach Block**

This block defines a rule-based system that works alongside the machine learning model. It performs checks on the names based on predefined rules, such as exact matches, phonetic matches, nickname matching, and various other string similarity measures.

* **Exact Name Match:** This function checks if two names are exactly identical.
* **Phonetic Encoding Match:** This function compares names based on their phonetic representations (e.g., using the Double Metaphone algorithm).
* **Synonym Matching:** This function matches names that may have synonymous or common variants.
* **Nickname Mapping:** This function matches formal names with their common nicknames (e.g., "William" to "Bill").
* **Levenshtein Distance:** This function calculates how many single-character edits are needed to change one name into another, which helps in identifying names with spelling mistakes or slight variations.
* **Jaro-Winkler Similarity:** This metric is used to compare the similarity of two strings based on character-level matching.
* **Cosine Similarity:** This function measures the cosine of the angle between two n-gram vectors, representing character-level similarity.
* **Soundex Encoding Match:** This function compares names based on their Soundex encoding, which is useful for matching names that sound similar but are spelled differently.
* **Need:** The rule-based approach is important for handling cases where names have small differences or specific variations not captured by the ML model alone. This approach allows for fine-tuning the matching logic and applying domain-specific rules that a machine learning model may not automatically consider.

**3. Model Loading and Prediction Block**

This block loads the pre-trained Random Forest model from the pickle file and uses it to make predictions on new input data. Additionally, the rule-based approach is also executed on the input names to generate matching results.

* **Need:** This block is necessary because it bridges the training phase and the inference phase. By loading the trained model, the system is able to apply the learned knowledge to new data without needing to retrain the model each time.

**4. Ensemble (Hybrid) Prediction (Voting) Block**

The Ensemble (Hybrid) prediction block combines the results from both the ML model and the rule-based approach. A **voting mechanism** is applied here, where if either the machine learning model or the rule-based system predicts a match, the final prediction will be a match.

* **Need:** Combining the ML and rule-based results in this way increases the accuracy of the predictions by leveraging the strengths of both approaches. If one approach fails to match a name, the other can potentially catch it, reducing the number of false negatives.

**5. Final Output Block**

This block represents the final output, where the matched names are displayed or returned. It provides the **matched pairs** as the result of the prediction.

* **Need:** This block is essential because it generates the output that will be used by downstream systems or presented to the user. The result could be stored in a file, displayed on a UI, or integrated into further business processes.

## Number of Variations Captured and How They Are Covered

This model handles 11 variations in name matching using both machine learning (Random Forest) and rule-based approaches. Below is a detailed explanation of how each variation is covered and which functions or techniques are used to achieve accurate matching.

**1. PHONETIC SIMILARITY**

* **Variation Description**: Phonetic similarity measures how closely two names sound when pronounced. This is crucial for matching names with similar sounds but different spellings.
* **Method Used**:
  + **Phonetic Encoding**: The doublemetaphone() function is used to encode names into their phonetic representations. This helps in matching names that sound similar but are spelled differently.
  + **Random Forest Block**: Phonetic similarity is captured using fuzz.ratio() after phonetic encoding and used as a feature for training the model.
  + **Rule-Based Block**: The function phonetic\_encoding() extracts phonetic representations, and fuzz.ratio() compares them for similarity, contributing to the overall similarity score.

**2. MISSING SPACES & HYPHENS**

* **Variation Description**: This variation addresses names where spaces or hyphens are missing or present, such as "MaryEllen" vs "Mary Ellen" or "Mary-Ellen".
* **Method Used**:
  + **Normalization**: The normalize\_name() function removes unnecessary characters like spaces, hyphens, periods, and commas. This ensures that the names are standardized before comparison.
  + **Random Forest Block**: The model learns to match such names by processing normalized names and using similarity metrics like Levenshtein distance and Jaro-Winkler similarity.
  + **Rule-Based Block**: The same normalization process applies before calculating other similarity measures, ensuring that missing spaces or hyphens don’t affect the result.

**3. SPELLING DIFFERENCES**

* **Variation Description**: Spelling differences occur when names are spelled slightly differently but are essentially the same. Examples include "Abdul Rasheed" and "Abd al-Rashid".
* **Method Used**:
  + **Levenshtein Distance**: This is used to measure how many character edits (insertions, deletions, substitutions) are required to convert one name into the other. The fuzz.ratio() function calculates this.
  + **Random Forest Block**: Spelling differences are captured as features in the training data, where the model learns to identify slight variations between names.
  + **Rule-Based Block**: The fuzz.ratio() function compares names directly and calculates a similarity score based on character-level edits.

**4. TITLES & HONORIFICS**

* **Variation Description**: Titles or honorifics (e.g., "Dr." vs "Mr." or "Ph.D.") may vary between individuals but refer to the same person.
* **Method Used**:
  + **Normalization**: Titles like "Dr.", "Mr.", and "Ph.D." are removed or standardized using normalize\_name().
  + **Random Forest Block**: The model can learn to ignore or standardize these variations based on training data where titles are present or missing.
  + **Rule-Based Block**: Titles and honorifics are normalized, ensuring that they do not affect the similarity score.

**5. MISSING COMPONENTS**

* **Variation Description**: This variation deals with names where certain components are missing. For example, "Phillip Charles Carr" vs "Phillip Carr".
* **Method Used**:
  + **Normalization**: Missing components are handled during name preprocessing. The model is trained to identify when parts of a name are missing and match it against a complete name.
  + **Random Forest Block**: The model accounts for missing components by learning patterns from the training data.
  + **Rule-Based Block**: The rule-based system also handles this by comparing the available components, assigning a similarity score based on the presence of common parts.

**6. OUT-OF-ORDER COMPONENTS**

* **Variation Description**: Names with components in a different order (e.g., "Diaz, Carlos Alfonzo" vs "Carlos Alfonzo Diaz").
* **Method Used**:
  + **Levenshtein Distance**: The model uses Levenshtein distance to compare names even when components are out of order.
  + **Random Forest Block**: The model is trained to recognize and handle variations in name order through feature extraction and similarity measures.
  + **Rule-Based Block**: The rule-based system can still match names with out-of-order components by focusing on character-based similarity rather than exact order.

**7. NICKNAMES**

* **Variation Description**: This variation covers matching between full names and nicknames (e.g., "William" vs "Will" vs "Bill" vs "Billy").
* **Method Used**:
  + **Synonym Matching**: Nickname mappings are handled by defining common synonyms for names (e.g., "Bill" as a synonym for "William").
  + **Random Forest Block**: The model can be trained on nickname pairs to identify which names are likely to match based on historical data.
  + **Rule-Based Block**: A dictionary of nicknames is used to match the names, contributing to the final similarity score.

**8. TRUNCATED COMPONENTS**

* **Variation Description**: Some names are truncated, such as "Blankenship" vs "Blankensh".
* **Method Used**:
  + **Levenshtein Distance**: The model uses Levenshtein distance to determine if a truncated name is similar to a full name.
  + **Random Forest Block**: The model can be trained to identify truncated names and match them to their full versions based on past data.
  + **Rule-Based Block**: The similarity score is calculated using fuzz.ratio(), which helps identify truncation issues.

**9. INITIALS**

* **Variation Description**: This variation matches names that include initials, such as "J. E. Smith" vs "James Earl Smith".
* **Method Used**:
  + **Tokenization**: Initials are treated as tokens, and the model compares these tokens to match full names with initials.
  + **Random Forest Block**: The model learns patterns where initials are used instead of full names.
  + **Rule-Based Block**: Initials are handled by tokenizing the names and comparing them based on character similarity.

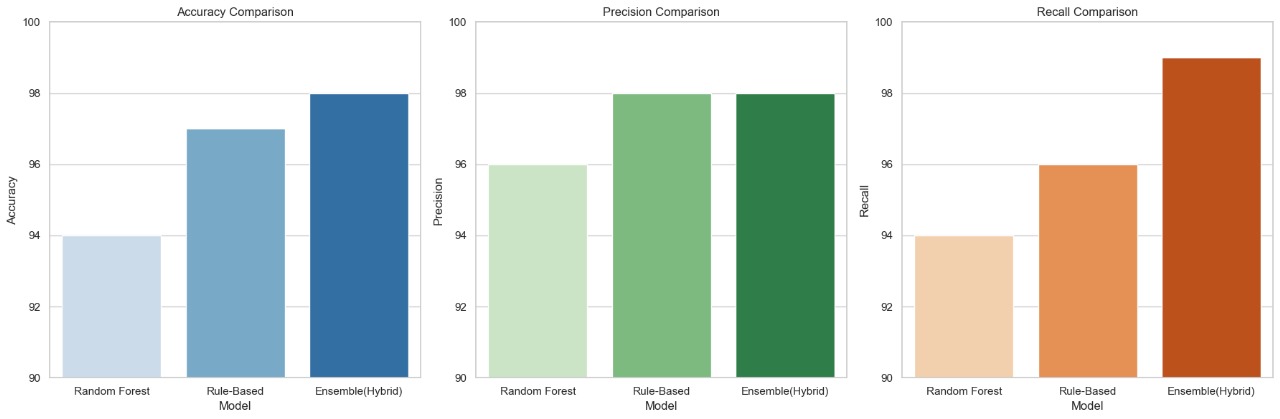
**10. SPLIT DATABASE FIELDS**

* **Variation Description**: Names in a database might be split into multiple fields, like "Rip. Van Winkle" vs "Rip Van. Winkle".
* **Method Used**:
  + **Normalization**: Names are standardized using normalize\_name(), where dots or spaces between parts are removed.
  + **Random Forest Block**: The model learns how to handle names split across multiple fields and recognizes when they are essentially the same.
  + **Rule-Based Block**: The rule-based block ensures that split names are combined and compared properly.

**11. TRANSLITERATION SPELLING DIFFERENCES**

* **Variation Description**: This variation handles spelling differences that arise from transliteration, such as "Abdul Rasheed" vs "Abd al-Rashid".
* **Method Used**:
  + **Levenshtein Distance**: The model uses Levenshtein distance to calculate the similarity between names with transliteration differences.
  + **Random Forest Block**: The model can learn to identify transliteration spelling differences during training.
  + **Rule-Based Block**: The rule-based system compares names using fuzz.ratio() and can handle small spelling differences.

## Comparative Analysis



**1. Random Forest Approach**

The **Machine Learning approach** uses a **Random Forest classifier**, a robust Ensemble (Hybrid) method, trained on features derived from multiple similarity measures, such as:

* **Levenshtein Distance**: Measures the number of edits needed to transform one string into another.
* **Jaro-Winkler Similarity**: Focuses on the beginning of strings, accounting for common typographical errors.
* **Phonetic Encodings (e.g., Soundex)**: Converts names into codes based on pronunciation.

**Performance:**

* **Accuracy**: 94.0%
* **Precision**: 96.0%
* **Recall**: 94.0%
* **Reliance on Feature Engineering**: The quality of the features used for training directly affects performance. Feature extraction can be complex and requires expert knowledge.
* **Sensitivity to Data Quality**: The model can be sensitive to noisy or inconsistent data, and poor-quality training data may lead to suboptimal performance.

While the **Random Forest approach** shows solid results, it may still exhibit a performance gap compared to the **Rule-Based approach**, especially when fine-tuned feature engineering is not adequately done.

**2. Rule-Based Approach**

The **Rule-Based approach** employs **exact** and **fuzzy** **string matching** techniques, such as **Levenshtein distance**, **phonetic encodings** (e.g., Soundex), and **similarity scoring**, to match names based on predefined rules and patterns.

**Performance:**

* **Accuracy**: 97.0%
* **Precision**: 98.0%
* **Recall**: 96.0%
* **Interpretability and Transparency**: The rules are explicitly defined, making it easier to understand and debug the model's behavior. This transparency is crucial in regulated industries where explainability is important.
* **Robustness to Data Quality**: The rule-based approach is less sensitive to noisy or incomplete data. If the matching pattern is defined well, it remains effective even in challenging data conditions.
* **Efficient Processing**: The algorithm can be optimized for speed, as it does not require extensive computations or training phases.

The **Rule-Based approach** performs well in controlled environments but may struggle when faced with highly diverse or unstructured data, as it lacks the flexibility of machine learning models.

**3. Ensemble (Hybrid) Model**

The **Ensemble (Hybrid) Model** combines both the **Machine Learning** and **Rule-Based approaches**. It captures the adaptability of machine learning while leveraging the simplicity and precision of rule-based methods.

**Performance:**

* **Accuracy**: 98.0%
* **Precision**: 98.0%
* **Recall**: 99.0%
  + **Combines the Best of Both Worlds**: By integrating the flexibility of machine learning and the precision of rule-based approaches, the Ensemble (Hybrid) model effectively addresses the weaknesses of each individual method.
  + **Improved Performance**: The hybrid model achieves higher recall, significantly reducing false negatives and ensuring more accurate matches.
  + **Reduced Bias and Variance**: The Ensemble (Hybrid) method helps in balancing the model's bias and variance, offering more robust performance across a wider range of name variations.

The **Ensemble (Hybrid) Model** provides the best performance, delivering high precision and recall while reducing the likelihood of false positives and negatives. It is particularly beneficial for applications where name variations are highly diverse, and maintaining both high precision and recall is essential.

## Conclusion

In summary, each name-matching approach has its unique strengths and weaknesses:

* The **Random Forest approach** excels in adaptability to diverse datasets and can achieve high accuracy, but its performance is highly dependent on feature engineering and the quality of the training data.
* The **Rule-Based approach** offers excellent precision and interpretability, making it effective in scenarios with predictable name patterns. However, it lacks flexibility and can be overfit to specific rules, potentially missing complex name variations.
* The **Ensemble (Hybrid) Model** combines the strengths of both methods, offering improved performance by reducing bias and variance while maintaining high precision and recall. Although more complex and computationally intensive, it is the most robust choice for handling a wide range of name variations.

Ultimately, the **Ensemble (Hybrid) Model** provides the most reliable and comprehensive solution for name-matching tasks, especially when dealing with diverse and complex datasets, making it the best choice for high-stakes applications requiring accurate and efficient name matching.

# Code Repository and Usage Instructions

**GitHub Link:** [Entity Matching Solution by Pie](https://1drv.ms/w/s!AinS7cQ6baFcieEOPYmH5MqmhoBI5Q?e=1syd2d)

This system performs name-matching using a combination of **Machine Learning** (Random Forest) and **Rule-Based** approaches, as described previously. It provides the necessary tools to generate and load a machine learning model, and it can process input data from Excel files to generate predictions.

**File 1: All 3 Approaches**

This file contains the complete code for the three approaches: Rule-Based, ML (Random Forest), and Hybrid. It includes functions for training the model, generating the pickle file (.pkl), and other necessary processes for name matching.

**File 2: final\_approach.py**

This is the core code file that contains the Random Forest training code and functionality for generating the pickle file (.pkl). It integrates both the ML model and Rule-Based approaches to perform name matching.

**File 3: pkl\_file\_random\_forest.pkl**

This file is the pre-trained machine learning model in .pkl format. It contains the trained Random Forest model that is used for name matching.

**File 4: pie\_entity\_matching\_solution.py**

This is the script that users will run to load the trained model, make predictions, and generate the result. The user needs to provide an Excel file with the name1 and name2 columns, and the script will output a new Excel file containing:

* **name1:** The first name.
* **name2:** The second name.
* **similarity score:** A numerical score representing the similarity between the two names.
* **label:** A binary classification (match or not match).

**Steps to Run the Code**

Here are the detailed steps for running **File 4: pie\_entity\_matching\_solution.py** to generate the prediction results.

**1. Clone the Repository from GitHub**

First, clone the GitHub repository to your local machine. Open your terminal or command prompt and run the following command:

git clone [GitHub Link Here]

After cloning, navigate to the cloned directory:

cd entity\_matching\_system

**2. Prerequisites**

Make sure the following are in place:

* Python 3.x installed.
* Required Python libraries installed:
  + pandas for handling data frames.
  + sklearn for machine learning.
  + joblib for loading the pickle file.
  + openpyxl for reading/writing Excel files.

If you haven’t installed the necessary libraries, you can install them using:

Command:

pip install pandas scikit-learn joblib openpyxl

**3. File Structure**

Ensure that your project directory has the following structure:

// entity\_matching\_system //

├── all\_approaches.py # File 1 (complete code for all three approaches)

├── final\_approach.py # File 2 (code for generating the pkl file)

├── pkl\_file\_random\_forest.pkl # File 3 (pre-trained model file)

├── pie\_entity\_matching\_solution.py # File 4 (testing script)

├── input\_data.xlsx # User's input Excel file

**4. Prepare Input File (Excel)**

The input Excel file, input\_data.xlsx, should have two columns: name1 and name2. The names in these columns will be compared for similarity.

Example of input data (Excel format):

|  |  |
| --- | --- |
| Name 1 | Name 2 |
| John Smith | Jon Smith |
| Alice Lee | Alyce Lee |
| Mike Tyler | Jon Smith |

Place this file in the same directory as the testing script.

**5. Modify Input File in Script**

* Open pie\_entity\_matching\_solution.py.
* Locate the line where the input Excel file is specified:

input\_file = 'input\_data.xlsx'

* Change the input\_file path to point to your input Excel file (if the filename is different).

**6. Running the Testing Code**

After ensuring the correct input file path, run **pie\_entity\_matching\_solution.py**. You can run the script by executing the following command in your terminal:

python pie\_entity\_matching\_solution.py

The script will:

* Load the Random Forest model from the .pkl file (pkl\_file\_random\_forest.pkl).
* Load the input Excel file containing name1 and name2 columns.
* Perform the name matching and generate similarity scores and labels for each pair.
* Save the results in a new Excel file, with the columns: name1, name2, similarity score, and label.

**7. Output File**

After running the script, an output Excel file will be generated in the same directory. The output file will be named output\_predictions.xlsx, and it will contain the following columns:

* **name1:** The first name.
* **name2:** The second name.
* **similarity score:** The numerical similarity score between name1 and name2.
* **label:** A binary label indicating whether the names match (1 for match, 0 for no match).

Example of output data (Excel format):

|  |  |  |  |
| --- | --- | --- | --- |
| Name 1 | Name 2 | Similarity Score | Label |
| John Smith | Jon Smith | 0.98 | 1 |
| Alice Lee | Alyce Lee | 0.94 | 1 |
| Mike Tyler | Jon Smith | 0.17 | 0 |

**8. Troubleshooting**

* **Error: File not found:** Make sure that the paths to the input file and pickle file are correct.
* **Error: Model not loaded:** If there’s an issue loading the model (pkl\_file\_random\_forest.pkl), ensure that the .pkl file is located in the same directory as the script or update the path in the script.