# **Implementation Report on Custom Fuzzy Data Matching Module**

## **1. Introduction**

In modern data-driven applications, handling and deduplicating large datasets containing potentially duplicate or slightly mismatched records can be challenging. This project focuses on developing a custom fuzzy matching and merging module for address and personal information datasets. The module aims to identify and merge records from two datasets based on defined fuzzy criteria, handling partial matches where exact comparisons may fall short. The solution proposed here provides a flexible, scalable alternative to off-the-shelf libraries such as Splink, allowing for adaptability in data comparison and merging rules.

The purpose of this report is to document the implementation, design decisions, and methodologies used to create a custom fuzzy data matching and merging module, fuzzy\_merge. This module is built in Python and leverages configurable fuzzy logic rules to handle inexact matches, an essential feature for many data integration projects.

## **2. Project Requirements and Objectives**

The primary objectives of this project were:

1. To design a modular solution capable of matching and merging records from two datasets based on approximate or fuzzy criteria.
2. To implement a Python-based module that can replace or serve as an alternative to the Splink library, which is widely used for similar tasks.
3. To ensure configurability in the matching process so that tolerance levels for matches can be easily adjusted.

Given these objectives, the project required an approach that balances accuracy in matching records with flexibility in configuration. This approach includes defining the tolerance for fuzzy matching to avoid false positives while ensuring that slightly mismatched entries are correctly identified and merged.

## **3. Implementation Overview**

The solution is structured as a Python package with four main components:

1. **Configuration File (config.py)**: Defines settings such as the matching threshold and tolerance level for Levenshtein distance in fuzzy comparisons.
2. **Fuzzy Matcher Module (fuzzy\_matcher.py)**: Implements the core fuzzy matching logic by calculating similarity scores between corresponding fields in two datasets.
3. **Data Merger Module (data\_merger.py)**: Contains logic to merge records from two datasets based on fuzzy matching results.
4. **Main Script (main.py)**: Runs the solution with sample data, demonstrating the module's capabilities and providing a reference for integrating the solution into broader data processing pipelines.

### **3.1 Configuration Settings**

The config.py file provides essential settings, such as the similarity threshold required for a match to be considered valid (set to 0.95) and the columns to be compared (email, firstname, lastname, street, zip, and city). Additionally, the tolerance for Levenshtein distance, a common metric for fuzzy string matching, is set to allow a maximum difference of two characters between matched fields.

The settings in this file enable flexibility, as adjustments to matching criteria can be made without changing the core codebase, promoting maintainability and customization.

### **3.2 Fuzzy Matching Logic**

The fuzzy\_matcher.py module performs fuzzy comparisons between corresponding fields in two datasets. It uses the Levenshtein similarity ratio to determine the degree of match between string fields, especially for first names, last names, and street addresses, which are more susceptible to typographical errors or variations.

The Levenshtein ratio (Brill, 1995) calculates the similarity between two strings based on the minimum number of single-character edits needed to transform one string into another. Using a tolerance level of two character differences allows for minor typographical errors, improving match accuracy for real-world datasets where perfect data entry cannot be assumed.

### **3.3 Merging Logic**

The data\_merger.py module manages to merge records from one dataset into another. For each record in the first dataset, it iterates through the records in the second dataset, looking for matches that meet the criteria defined in the fuzzy\_matcher.py module. If a match is found, the data is updated, with missing fields in the existing record being populated from the new record. If no match is found, the new record is appended to the second dataset as a unique entry.

This approach not only ensures data completeness but also reduces redundancy by avoiding duplicate records.

### **3.4 Sample Execution and Testing**

The main.py script demonstrates the module’s functionality. Hypothetical data was created for testing purposes, consisting of variations in names, emails, and addresses. The script runs through the merging process and outputs an updated DataFrame, showcasing the successful integration and deduplication of the two datasets.

Sample data includes names, addresses, and email variations that are realistic in terms of potential data entry errors. The module correctly identifies similar records across both datasets, validates matches based on the defined similarity threshold, and merges records as expected.

## **4. Summary of Results**

The fuzzy\_merge module successfully identified and merged records based on the fuzzy matching criteria, effectively handling minor variations in address and name fields. The module’s modular design also allows it to be easily configured for datasets with different structures or matching requirements. Overall, this custom solution meets the project’s objectives, providing a robust, adaptable, and transparent method for deduplicating and merging address records.

## **5. References**

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational linguistics*, *21*(4), 543-565.