**Fuzzy Matching Project**

***Python Code Results:***

* **In the Python code I used the fuzzywuzzy and rapidfuzz packages for 1k, 10k and 100k rows on 2 syntethic generated dataframes.**
* The code needed the following amount of time to run for different methods/packages and amounts of data:
* **1k rows of synthetic generated first and last names:**
  + Elapsed time fuzzy simple ratio: 15.73 seconds
  + Elapsed time fuzzy partial ratio: 52.58 seconds
  + Elapsed time fuzzy token sort ratio: 29.77 seconds
  + Elapsed time fuzzy token set ratio: 40.71 seconds
  + Elapsed time rapidfuzz simple ratio: 5.11 seconds
  + Elapsed time rapidfuzz partial ratio: 11.63 seconds
  + Elapsed time rapidfuzz token sort ratio: 7.82 seconds
  + Elapsed time rapidfuzz token set ratio: 23.06 seconds
  + Elapsed time optimized rapidfuzz simple ratio: 0.10 seconds
  + Elapsed time optimized rapidfuzz partial ratio: 2.54 seconds
  + Elapsed time optimized rapidfuzz token sort ratio: 0.13 seconds
  + Elapsed time optimized rapidfuzz token set ratio: 18.86 seconds
  + Elapsed time parallel&optimized rapidfuzz simple ratio: 4.27 seconds
  + Elapsed time parallel&optimized rapidfuzz partial ratio: 1.04 seconds
  + Elapsed time parallel&optimized rapidfuzz token sort ratio: 0.17 seconds
  + Elapsed time parallel&optimized rapidfuzz token set ratio: 1.12 seconds
* **10k rows (ran only the simple ratio() function for both packages):**
  + fuzzywuzzy: 16min
  + rapidfuzz: 4min
  + optimized rapidfuzz: 3sec
  + optimized & parallelized rapidfuzz: -
* **100k rows (ran only the simple ratio() function for both packages):**
  + fuzzywuzzy: - (could take entire hours to run because of the exhaustive search between the strings)
  + rapidfuzz: - (could take entire hours to run because of the exhaustive search between the strings)
  + optimized rapidfuzz: 13min
  + optimized & parallelized rapidfuzz: 6min
* **1mil rows: couldn’t ran the code (the pc was blocking/freezing and overheating)**

***INSTRUCTIONS/QUESTIONS ANSWERED***

***- Describe the algorithm or method you would use for fuzzy matching.***

The algorithm would have to follow/contain the following steps:

**1. Preprocessing**

Preprocessing is crucial to standardize the data and reduce noise. The steps include:

* **Lowercasing**: Convert all names to lowercase to ensure case-insensitivity.
* **Remove Special Characters**: Strip special characters and punctuation.
* **Standardize Suffixes and Prefixes**: Handle common suffixes (e.g., Jr., Sr., III) and prefixes (e.g., Dr., Mr.).

**\* Optional: Blocking Keys**

Blocking Keys are used to reduce the number of comparisons by grouping similar records.

* **Blocking Key Creation**: Create blocking keys using the first few characters of the last name and the first name initials.

**2. Fuzzy Matching**

Using similarity metrics and different packages to find potential matches (and fast) between the first, last and full names.

* **String Similarity Metrics** (e.g. Levenshtein Distance or) Jaro-Winkler) to measure the similarity between names.

*- Discuss the trade-offs and efficiency of the chosen method.*

#### Trade-offs:

* **Accuracy vs. Speed**: Increasing the threshold for similarity metrics improves accuracy but increases computational cost.
* **Blocking Complexity**: More sophisticated blocking methods reduce the number of comparisons but might miss some potential matches.
* **Preprocessing Time**: Extensive preprocessing ensures better matching accuracy but adds to the initial computational load (multiple checks for special chars, suffixes/prefixes and so on).

#### Efficiency:

* **Parallel Processing**: Utilizes multi-core processors to handle large datasets efficiently.
* **Threshold Tuning**: Balances precision and recall by adjusting similarity thresholds based on validation results and could make the running of the pipeline more faster.

*- Explain any pre-processing/cleaning steps you would take.*

**Preprocessing/Cleaning Steps of the data**

1. **Lowercasing**: Convert all characters to lowercase.
2. **Remove Special Characters**: Strip out punctuation and special characters.
3. **Standardize Suffixes and Prefixes**: Normalize or remove common suffixes (Jr., Sr.) and prefixes (Dr., Mr.).
4. **Tokenization**: Split names into first and last names, handle common nicknames.

*- Explain how you would optimize the matching process to handle the large dataset size efficiently.*

**Optimizing for Large Datase**

* **Parallel Processing**: Use multi-threading or distributed computing frameworks to process the data in parallel (even frameworks/tools like Spark or Dask).
* **Efficient Data Structures**: Utilize efficient data structures (e.g., hash maps) to speed up lookups.
* **Incremental Processing**: Process data in chunks to manage memory usage and improve performance.

*- Provide a sample code in Python that is able to identify possible matches from large data sets in an efficient manner.*

-> Attached in the email, and contains:

* The virtual environment (can be used for replicating the used env for developing/testing)
* The jupyter notebooks that were used for testing and developing the code, and also for generating the artificial data.
* A sample of data (1k rows for both df1 & df2).
* The modularized code that can be run in terminal with: python main.py and it shows the elapsed time for each method/algo (also for the optimized and parallel optimized ones), which saves in the end a result dataframe from a merge between preprocessed dataframe with rapidfuzz parallel matching and client\_df to get the info from client\_df based on the matchedfullnames.