# **CREATING MASTER-DATA FROM DATASET USING SIMILARITY-SCORES OF TEXT-FIELDS**

|  |  |  |
| --- | --- | --- |
| Rohan Gursale  Deloitte-USI, A&C  Mumbai, India  @gmail.com | Vikrant Deshpande  Deloitte-USI, A&C  Mumbai, India  vikrant.deshpande09876@gmail.com | Roopal Gupta  Deloitte-USI, A&C  Mumbai, India  @gmail.com |
|  |  |  |

## **INTRODUCTION**

Most conglomerates today, have many separate applications and systems (ie. ERP, CRM) where data that crosses organizational departments or divisions, can easily become fragmented, or duplicated. Reporting critical KPI (Key-Performance-Indicators) for a business accurately becomes difficult. Questions like “How many entities do we govern?”, “Which of them are most profitable?”, require a cleaned, accurate, master dataset. The data-capturing phase itself might lack a standardized approach, resulting in fundamental discrepancies rendering the data unusable for reporting. An incorrect address in the customer-master might mean orders, bills and marketing literature are all sent to the wrong address; an incorrect account number in an account master could lead to huge fines!

**MOTIVATION FOR THE USE-CASE**

This paper focuses on *masterizing* clinical-data in terms of hospitals/sites, that the pharmaceutical-client manages. For example- the same site “Kadlec Regional Medical Center”, might be reported differently as “Kadlec Clinic Hematology and Oncology” but with the same address, across the client’s source systems. Our goal is to identify a *golden entity* (Master Record) to which other duplicate records can be matched, and maintain their *source-to-master* *linkage* (Cross-Reference). Although industry-standard tools are available (Informatica, Oracle, SAP, etc.) that can be used with third party collaborators like *Address-Doctor-Service*, or *Dun&Bradstreet* to retrieve the standardized asset-data, this case-study was intended to prove that open-source code and libraries could produce near-standardized results.

## **LITERATURE SURVEY & IMPLEMENTATION CHOICES**

The intuition behind identifying unique entities within a dataset is as follows:

* Within a dataset of *n* records, we’d have to compare the 1st record with the remaining (*n - 1*) records, the 2nd record with the remaining (*n - 2*) records, and so on. Thus, there would be unique combinations to be considered.
* Between 2 different datasets of *m* and *n* records each, there would similarly be unique combinations to be considered. These *n* records can be thought of as the previously identified master records.

At each individual-combination level i.e. for the participating records, a string-comparison algorithm [a] will be used to compute a match-score of the relevant feature-strings. Let *str1*=“Kadlec Regional Medical Center” and *str2*="Kadlec Clinic Hematology and Oncology".

1. **Edit-distance based** **algorithms** (ex- Levenshtein) compute the number of character-level operations needed to transform one string to another. More the number of these character addition/subtraction/replacement operations, less is the similarity between the two strings. For example- the Levenshtein distance will be 25, and the normalized-similarity will be:

*0.325*

Jaro-Winkler is a similar directional-algorithm which checks for characters of str1 occurring in a window of some size within str2.

1. **Token based** **algorithms** (ex- Jaccard-index) will find the similar tokens in both string-sets. More the number of common tokens (words or n-gram characters), more is the similarity between the sets.

=

Using words as tokens, the score *0.125*, while individual character-tokens gives *0.558*.

1. **Sequence based algorithms** (ex- Ratcliff-Obershelp similarity) try to find the longest sequences present in both strings. First, remove the longest common substring from both strings, and split the originals into the left and right parts of the common substring. Repeat this recursively for both the left and right parts, until the size of any broken part is less than a default value. The score is twice the number of characters found in common divided by the total number of characters in the two strings.

*0.45*

1. **Cosine-similarity** can be summarized as a widely used NLP technique that uses a matrix of word-embeddings: where each cell in a column, represents the weight by which the word associates to that row/attribute. Two words *x* and *y*, are first converted to their word-vectors from this word-embedding matrix, and the cosine formula is applied to identify semantic similarity:

Subadhra Parthasarathy et al. [b] concluded that Levenshtein produces results on-par with cosine-similarity, when matching potential duplicate organization-names against a master list. Bearing in mind that most of our dataset contained junk characters, spelling errors, and the anagram-possibility scenario of Jaccard-measure, coupled with this comparison against cosine-similarity, it made sense to use the Levenshtein algorithm. In contrast to their approach, we wanted to identify the unique entities in our dataset since there was no standard set available, deduplicating the input by comparing it against itself was vital for our process.

## The RecordLinkage library [c] in R, provides two main functions to generate candidates for deduplication within a single dataset (hereafter called the *dedup* function), or candidates for identifying duplicates between two different datasets (hereafter called the *linkage* function). The Python equivalent library [d] is limited by the array-size that Pandas can hold when the number of candidate-pairs is ginormous. Python’s easy to use data-wrangling libraries, ability to invoke a child-subprocess including R-scripts, topped with some deployment-server versioning limitations, led to developing the end-to-end structural pipeline in Python. R would be used only for generating match-scores (indirectly by using a pre-compiled C function), since in-memory statistical computations are much faster.

Since the deployment server supported only Python 2.5x and R 3.4.4, we had to refactor the Python code, and reverse-engineer the RecordLinkage library in R (since it requires R >= 3.5.0). The original core capabilities of the *dedup* and *linkage* functions were maintained, but the cursory code involving phonetic algorithms, blocking datasets, etc. were removed to speed up the algorithm. The Levenshtein function written in C by Joe Conway, Murat Sariyar and Andreas Borg was pre-compiled into its binaries, and reloaded into R using the following commands:

* R CMD SHLIB levenshtein.c
* dyn.load("levenshtein.dll")

Binaries generated in Windows have the .*dll* (Dynamically Linked Libraries) extension, while in Unix they have the .*so* (Shared Object) extension.

Once the *dyn.load*() function loads the binaries and the symbols within, the Levenshtein function can be invoked to return the number of characters replaced/added/removed to make the strings match.

## **ARCHITECTURE**

1. **PREPROCESSING THE RAW-DATA**

The incoming dataset for this pharmaceutical client was a set of 4+ source-systems, with more than 30 countries due to a global presence.

1. Country-level batching is performed; primarily since the country field was standardized in the preprocessing ETL phase (using ISO-standard translation tables), to act as the most reliable field amongst all others, and secondly 2 or more duplicate Site records would implicitly belong to the same country.
2. Features relevant to calculating a match-score are used to sort the data into minibatches of size less than or equal to some threshold: *Site-Name*, *Postal-Code*, *State*, *City*, *Address-Line*

This is to ensure that even though the data might contain junk characters, or spell errors, each minibatch itself could contain a high volume of the duplicates already, leading to higher compression.

An ID is assigned to records using Row-Number function, which will be used in the cross-reference table for backtracking.

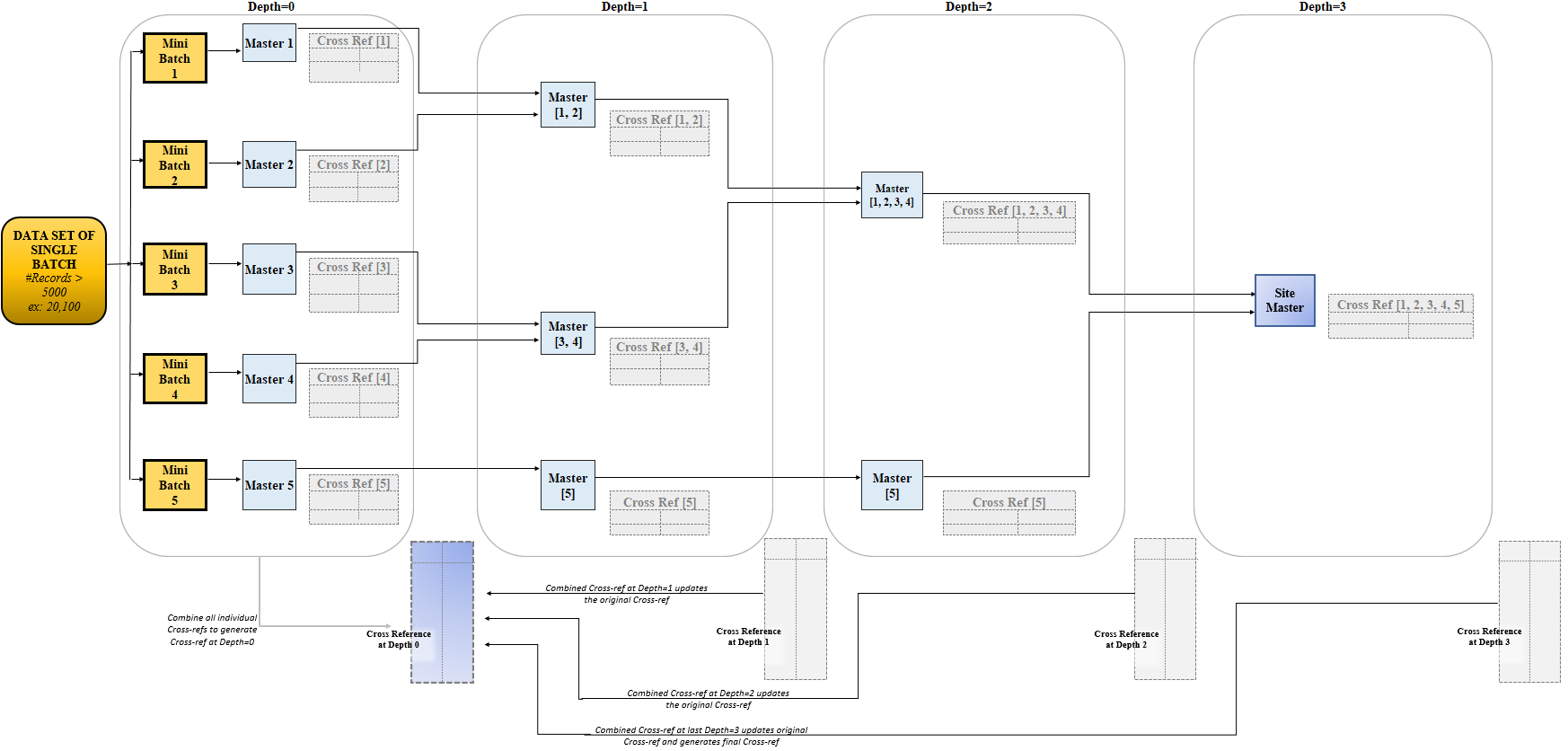
1. The raw csv is ingested into a pandas DataFrame in UTF-8 encoding to ensure Non-Latin scripts are handled. It is then cleaned using simple ETL functions to replace punctuation marks, replacing “NULL” with blanks, and ensuring the index of the DataFrame is the row-number column generated in previous step. Address-Fields 1 to 3, are all concatenated into a single column, and individually dropped.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Sr.**  **No.** | **Country** | **Site-Name** | **State** | **City** | **Addr-Line-1** | **Addr-Line-2** | **Addr-Line-3** | **Zip**  **code** |
| 1001 | Algeria | Centre Hospitalo Universitaire de Batna | Batna | Batna | Allées Mohamed Boudiaf | NULL | NULL | 05000 |
| 1002 | Algeria | Centre Hospitalier Universitaire Tlemcen | Tlemcen | Tlemcen | Boulevard Mohamed V | NULL | NULL | 13000 |
| 1003 | Algeria | Centre Hospitalo Universitaire de Constantine | Alger | Alger | 11 BP, Colonel Amirouche | NULL | NULL | 16000 |
| 1004 | Algeria | Centre Pierre et Marie Curie | Alger | Sidi M'Hamed | Place du 1er Mai 1945 | Centre Hospitalier Universitaire Mustapha Pacha | NULL | 16000 |
| 1005 | Algeria | EPH Mascara | Mascara | Mascara | Mascara | NULL | NULL | 29000 |
| 1006 | Argentina | Hospital Universitario Austral | Buenos Aires | Buenos Aires | Avenida Juan D. Peron 1500 | NULL | NULL | 01629 |
| 1007 | Argentina | FUNDALEU - Fundacion para combatir la Leucemia | Ciudad Autonoma Buenos Aires | Ciudad Autonoma Buenos Aires | José E.Uriburu 1450 | NULL | NULL | 1114 |
| **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** | **. . .** |
| 1499 | Argentina | Hospital Italiano de Buenos Aires | Ciudad Autónoma de BuenosAires | Ciudad Autonoma de Buenos Aires | Calle Tte Gral Juan Domingo Peron 4190 | Department of Oncology | NULL | 1199 |
| 1500 | Argentina | Hospital Britanico de Buenos Aires | Ciudad Autonoma Buenos Aires | Ciudad Autonoma Buenos Aires | Perdriel 74 | NULL | NULL | 1280 |
| 1501 | Argentina | Hospital Britanico de Buenos Aires | Ciudad Autonoma Buenos Aires | Ciudad Autonoma Buenos Aires | Perdriel 74 | NULL | NULL | 1280 |

*Fig. ABC1- Sorted dataset splits into minibatches with high volume of potential duplicates*

## **RECURSIVE PROCESSING**

This algorithmic approach will first pass minibatches of a fixed size into the *dedup* R-function and generate deduplicated master-datasets. These deduplicated master-datasets would be compared against each other using the *linkage* R-function. This is like the conventional level-order traversal of a binary tree using a queue in reverse, until each record is compared against every other. The motivation here is to prevent overuse of RAM, due to in-memory candidate pair computations.



*Fig. ABC3- Pipeline for the Recursive approach*

At Depth=0, the number of minibatches will be:

For each of these m iterations, a csv file will be generated after the *dedup* function, and added to a queue of CSV file names. A cross-reference DataFrame will be maintained for the entire batch that will keep getting updated during each step of the process.

For example- for an incoming batch of 20,100 records:

1. We’ll have 5 minibatches considering each minibatch-size of 5,000.
2. After the 5 iterations, the cross-reference has 20,100 entries (source-to-master linkages).
3. Rather than computing match-scores for 201,994,950 candidate pairs\* in a single go, the algorithm deduplicates 4 minibatches of 5,000 records, and 1 minibatch of 100 records, at this Depth=0.
4. The already sorted data in each minibatch leads to high volume-compressions; on average 80% are identified as duplicates of the remaining 20% master records.
5. These 20% master records are written to a CSV file and the file name is added to a queue of CSVs.

For the subsequent Depths = 1, 2 … (*m* + 1)/2 , we pop 2 CSV file names at a time from the queue and process them using the *linkage* R-function. The output of each pair is written as a new CSV, and the file name is appended to the same queue. If no CSV is present in the current queue to compare against the first CSV, simply write the first CSV dataset as the output of the comparison.

At each depth, we maintain a cross-reference of that depth which will be used to update the cross-reference of the entire batch with what has newly been observed as source-to-master linkage. Essentially, after comparing a set of masters amongst each other, update these newly identified masters into the existing cross-reference of the entire batch.

The time taken for recursively processing a large batch, is significantly lower than the time that would’ve been required to process it in a single go. The following observations were taken on an AWS EC2 instance \*\*\*\*\*\*\*\* by considering minibatches of size 5,000:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input batch-size** | **Candidate-pairs** *n\*(n-1)/2* | **Minibatches**  *(BatchSize/5000)+1* | **Time required** | **Comment** |
| 735 | 269,745 | 1 | 5 sec | Single minibatch |
| 3,500 | 6,123,250 | 1 | 2 min | Single minibatch |
| 5,000 | 12,497,500 | 1 | 5 min | Single minibatch |
| 22,882 | 261,781,521 | 5 | 35 min | 5 mastered minibatches created, and recursively processed |
| 22,882 | 261,781,521 | 1 | N/A | Single-shot processing would theoretically require 90 min. However, the child process itself gets killed (RAM usage exceeds limit) |

*Fig. ABC2- Execution stats of different volumes of input batch*

## **INTERPRETING SCORES AND IDENTIFYING MASTERS**

For both, the *dedup* and *linkage* R*-*functions, we use thresholds to convert the normalized Levenshtein-similarity score into a binary-values to indicate if the feature matches for a candidate pair or not. The address-match score is scaled up by a factor, since in many cases, the state, city and postal code were empty/different but showed up in the address.

We sum up all comparison outputs to produce a total-score of that candidate pair. If total-score is greater than or equal to TOTAL\_MATCHES\_THRESHOLD, this candidate pair is considered for further processing.

|  |  |  |
| --- | --- | --- |
| **Features** | **Match-score Threshold** | **Scaling Factor** |
| SITE\_NAME | 0.85 | 1 |
| STATE | 0.85 | 1 |
| CITY | 0.85 | 1 |
| CONCAT\_ADDRESS | 0.75 | 3 |
| POSTAL\_CODE | 0.85 | 1 |
| TOTAL\_MATCHES\_THRESHOLD | 4 |  |

*Fig. ABC2- Thresholds/parameters for the match-score computations*

1. The output of these R-functions can be interpreted as *the raw universe of potential duplicates* for that minibatch; a DataFrame of ( Source-Record-Id, Master-Record-Id, Site-Name-Comparison-Score, State-Comparison-Score, City-Comparison-Score, Address-Comparison-Score, Postal-Code-Comparison-Score ) .
2. The source-record can match against multiple master-records with total match-score >= 4. We choose the best match for incoming source-records based on highest total-score.
3. There are also cyclic cases in the score outputs like-

Record B matches against Record A

Record C matches against Record B

Ideally, we should transitively maintain:

Record C matches against Record A

These cyclic occurrences can extend up to 10-15 such transitive linkages, so handling them efficiently was crucial.

1. Finally, from this list of cleaned-normalized-score-features, we use basic set-theory to find the unique list of masters.

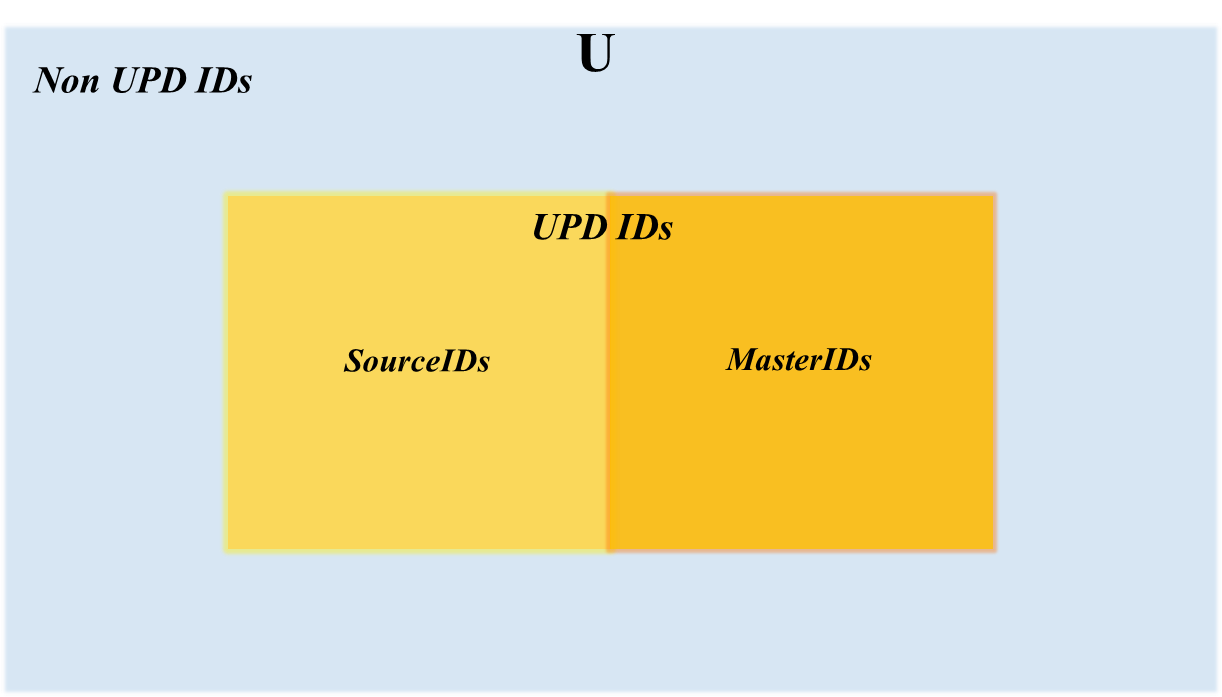
‘SR\_NUM’ of the entire minibatch, will be the universe of records.

Consider ‘SR\_NUM\_1' as the list of incoming Source-Ids, and 'SR\_NUM\_2' as the Master-Id to which it should be mapped based on match-score.

Union of 'SR\_NUM\_1' & 'SR\_NUM\_2' will be *the universe of potential duplicates* (UPD).

Stand-alone records in the current minibatch, are those which do not fall in this *universe of potential duplicates* (Non-UPD).

The final Master-records will be the union of Master-Ids and the Stand-alone Ids identified above.



*Fig. ABC4- Identifying master records by interpreting score output*

## **FUTURE SCOPE**

## Lemmatization is the grouping together of a word’s different inflected forms to a single item, i.e. it links words having similar meaning, to one word. Lemmatizing each word in the features, during the pre-processing step itself, *might* improve match-score computations (data has spelling errors, so the root word outputted would be the same as input since no root-word would be found). Python has a library called NLTK (Natural Language Processing Tool Kit) for this.

## Implement an incremental approach to match: incoming dataset of delta-records vs the already identified master-records. The few records with country=”NULL” should also be handled in this case.

* 1. The score output (*raw* *universe of potential duplicates* within a minibatch) generated by computing each candidate-pair’s match-scores can have cyclic occurrences. For now, this is our strategy, but it has scope for improvement:
     1. If *1548* matches with *1543* with total score of 7 (max possible score)
     2. But *1543* itself matches with *1541* probably with a lesser score of 4 (lower score)
     3. If site-names are different, remove this candidate-pair [*1543* vs *1541*] from this universe of potential duplicates, thus making ***1543*** and ***1541*** as 2 separate master records.
     4. If site-names are same for this candidate-pair [*1543* vs *1541*], ***1541*** would be the final parent record for *1548* and *1543* both.
  2. Fine-tune the process of master-record selection; instead of just choosing the very first record amongst the universe of potential duplicates within a country, scan through this universe and check for max-occurrences of features within that subset. The best-candidate for the golden master-record, would be the one having the highest overall combined-similarity score. For ex, for a subset of interlinked candidate-masters: *1501, 1502, 1503,* and *1504*, consider that *1501* matches against *1502* with total score of 4, against *1503* with total score of 5, against *1504* with total score of 7 and so on:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Candidates** | **1501** | **1502** | **1503** | **1504** | **Total Preference** |
| **1501** | - | 4 | 5 | 7 | ***16*** |
| **1502** | 4 | - | 6 | 7 | ***17*** |
| **1503** | 5 | 6 | - | 4 | ***15*** |
| **1504** | 7 | 7 | 4 | - | ***18*** |

*Fig. ABC5- Better identification of master amongst a set of duplicates*

*1504* could be the best candidate here for the golden master-record since it has highest overall combined-similarity score.

* 1. Address-Fields 1, 2, and 3 can have human errors, ex: addr2 of first record might be same as addr3 of second record, or addr3 may not be present for one record, but could be a huge string for second record, leading to address-comparison mismatch. Compare each combination of addresses for each address fields, viz. [a.addr1 vs b.addr1], [a.addr1 vs b.addr2], [a.addr1 vs b.addr3], [a.addr2 vs b.addr1], [a.addr2 vs b.addr2], and so on.

## Develop a front-end application to enable business-users to look at merge-scenarios and take actions themselves: Merge or unmerge 2 different records.

## **REFERENCES**

## \*\* the Medium blog that summarizes these algorithms (<https://itnext.io/string-similarity-the-basic-know-your-algorithms-guide-3de3d7346227>)

## \*\* the other research paper stating advantages of Levenshtein (<https://amedeloitte.sharepoint.com/:f:/r/sites/AIIEEmergentCapabilities/Shared%20Documents/04.%20NLP%20-%20NUG/Text%20Search?csf=1&web=1&e=DFImLT>)

## \*\* the RecordLinkage R library (<https://CRAN.R-project.org/package=RecordLinkage>)

1. \*\* the RecordLinkage Python library (<https://recordlinkage.readthedocs.io/en/latest/ref-index.html>)