# **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  roopalgupta123@gmail.com |
|  |  |  |

## **INTRODUCTION**

Most conglomerates today have many separate applications and systems (viz. 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 and 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 CASE-STUDY**

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 must 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 n2 unique combinations to be considered.
* Between 2 different datasets of *m* and *n* records each, there would similarly be unique combinations to be considered.

At an individual combination level i.e. for the participating records, a string-comparison algorithm [1] 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 between *str1* and *str2* will be 25, and the normalized-similarity will be:

*0.325*

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

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

=

For *str1* and *str2,* using words as tokens the score *0.125*, while using 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 [2]: 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:

Kaitlin Coltin et al. observed that Levenshtein produces results on par with Cosine-similarity, when matching potential duplicate organization names against a master list [4]. Bearing in mind that a high volume of our dataset contained junk characters and spelling errors, and considering the anagram-possibility scenario of Jaccard-measure, coupled with their 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 with no standard set available, deduplicating the input by comparing it against itself was vital for our process. Similarly, applying machine-learning techniques like a clustering or classification algorithm wasn’t possible since there isn’t a target variable/list to train or test on.

The **RecordLinkage** library 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) [5]. The Python equivalent library is limited by the array-size that Pandas can hold when the number of candidate-pairs is ginormous [6]. However, Python’s easy-to-use data-wrangling features, ability to invoke a child-subprocess like R-scripts, topped with some deployment-server versioning limitations, led to developing the end-to-end structural pipeline in Python. R is used only for generating match-scores (indirectly by using a pre-compiled C function), since in-memory statistical computations are much faster, and Pandas cannot generate such massive sized-DataFrames in-memory [7].

Since the deployment server supported only Python 2.5x and R 3.4x, 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 supporting phonetic algorithms, blocking datasets, etc. was removed to speed up the algorithm. The Levenshtein function can be implemented in multiple ways [3], but we picked the source code written in C by Joe Conway, Murat Sariyar, and Andreas Borg [5] since it is already used as part of the RecordLinkage package. The source code in C was pre-compiled into its binaries, and reloaded into R using the following commands:

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

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

Once the *dyn.load*() function loads the binaries and the symbols within, the Levenshtein function can be invoked from within R, 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 using a wrapper for-loop; firstly because 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 spelling errors, each minibatch itself can contain a high volume of duplicates already, leading to higher compression.

A Row-Number ID is assigned to records, to be used in the cross-reference table for backtracking. (Refer Fig. 1)

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 like removing punctuation marks, replacing “NULL” with blanks, and ensuring the index of the DataFrame is the integer row-number column generated in the previous step.

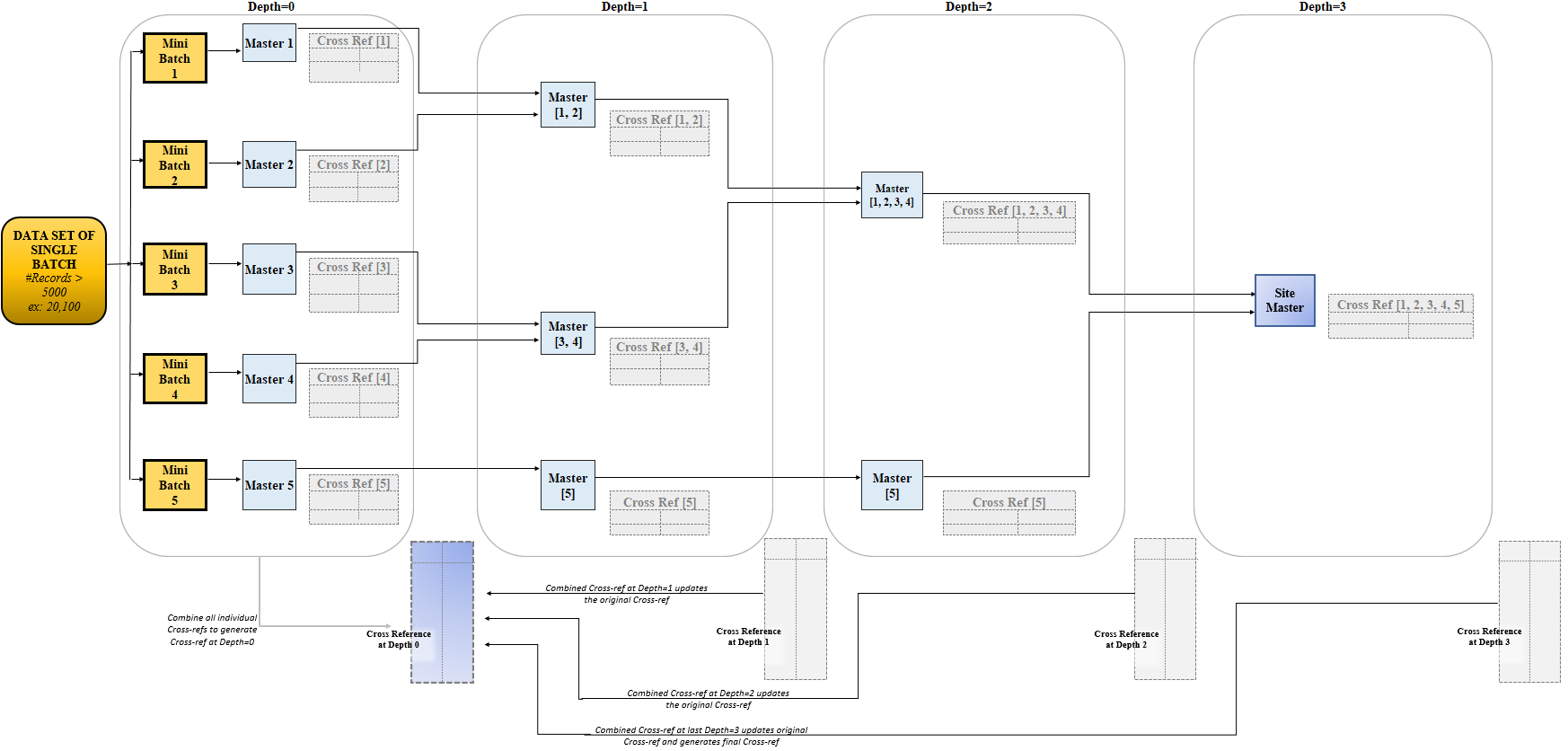
Address-Fields 1 to 3, are all concatenated into a single column, and individually dropped.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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. 1- Sorted dataset splits into minibatches with a 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, but 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. 2- 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, i.e. on average we’d identified 1000 unique masters.
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 at depth *d*, 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 by considering minibatches of size 5,000, on an AWS EC2 instance m5.4xlarge (64 GB RAM, and 16 vCPUs- each a single thread on a 3.1 GHz Intel Xeon Platinum 8175M processor):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input batch-size** | **Candidate-pairs** *n(n-1)/2* | **Minibatches**  *(BatchSize/5000)+1* | **Time required** | **Comment** |
| 735 | 269,745 | 1 | 5 sec | Single minibatch processed |
| 3,500 | 6,123,250 | 1 | 2 min | Single minibatch processed |
| 5,000 | 12,497,500 | 1 | 5 min | Single minibatch 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) |
| 22,882 | 261,781,521 | 5 | 35 min | 5 mastered minibatches created, and recursively processed faster than the theoretical expected time |

*Fig. 3- 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 the 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. 4- 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 containing [ 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 a total match-score >= 4. We choose the best match for incoming source-records based on the highest total score for all its potential master-records.
3. There are cyclic cases in these 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 may extend to upwards of 10-15 such transitive linkages, so handling them efficiently is crucial.

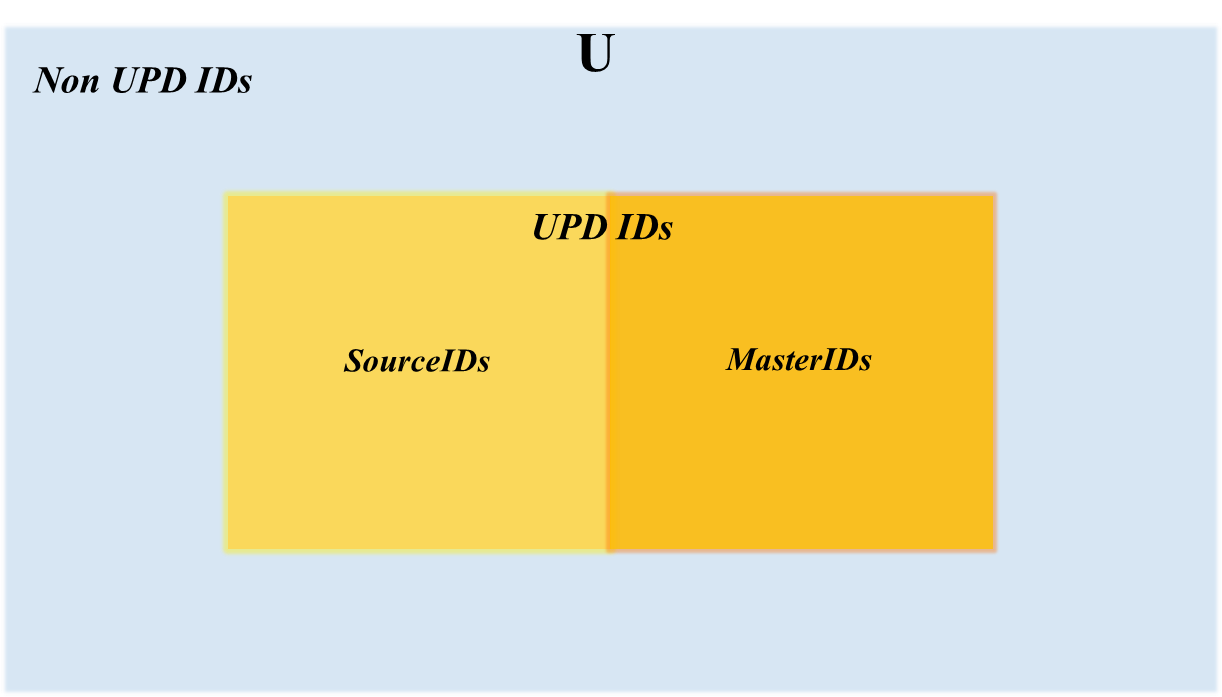
1. Finally, from this list of cleaned-normalized-score-features, using basic set-theory we find the unique list of masters. Consider ‘SR\_NUM\_1' as the list of incoming Source-Ids, and 'SR\_NUM\_2' as the Master-Ids to which it should be mapped based on match-score.

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

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. 5- Identifying master records by interpreting score output*

## **OUTCOMES**

## The *masterization* process for a batch finally produces the Master file and a Cross-reference file which contains a translation of each record against either: a parent Master-record, or itself (in case of the record itself being a Master-record). The following table concisely summarizes the functionality of the existing algorithm based on a few scenarios covered for testing:

|  |  |  |  |
| --- | --- | --- | --- |
| **Country** | **Test case description** | **Observation** | **Status** |
| USA | Batch has size>mini-batch size, data should be processed in recursive minibatches | Data processed in recursive minibatches | Pass |
| Turkey | Batch has special characters in UTF-8 encoding. Master data should be generated by considering special chars. | Comparison can handle special chars | Pass |
| UK and Northern Ireland | NULL valued features should match with each other with score=0 | Comparisons against NULL values are given a score of 0 | Pass |
| France | Batch with potential duplicates. Master data should look unique. | There are still some potential duplicates due to data-scenarios  (refer to Future Scope in the next section) | Fail |
| Paraguay | Batch has a single record, should be marked as master itself | The single record is marked as master itself | Pass |

*Fig. 6- Test cases and scenarios covered*

The following table gives the compression statistics for a few batches monitored. Compression percent of Raw-data to Master-data is just the percent of duplicates found.

|  |  |  |  |
| --- | --- | --- | --- |
| **Country** | **Input Dataset size** | **Unique Masters identified** | **Data Compression %** |
| USA | 22,881 | 8,818 | 61.46 |
| Turkey | 2,275 | 764 | 66.42 |
| UK and Northern Ireland | 1,885 | 631 | 66.53 |
| France | 1,628 | 639 | 60.75 |
| .  .  . | **. . .** | **. . .** | **. . .** |
| Paraguay | 1 | 1 | 0 |
| **Overall** | **50,937** | **18,695** | **63.29** |

*Fig. 7- Compression stats for input datasets*

In the absence of a reference Master-list of site information that the client governs, it wasn’t possible to compute the accuracy/precision of the output. However, the next section will dig into the current shortcomings, which were fairly noticeable.

## **FUTURE SCOPE**

## Lemmatization is the grouping together of a word’s different inflected forms to a single item, i.e. it links words having a similar meaning, to one word [8]. Lemmatizing each word in the features, during the pre-processing step itself, can improve match-score computations. However, in case of spelling errors (a high number in our dataset), the output word would be the same as input since no root word would be found and overall performance *might not* improve. Python has a library called NLTK (Natural Language Processing Tool Kit) for this.

## Implement an incremental approach to match: an 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 scoring 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 a 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 the same for this candidate-pair [*1543* vs *1541*], ***1541*** would be the final parent record for both *1548* and *1543*.
  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 a total score of 4, against *1503* with a total score of 5, against *1504* with a 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. 8- Better identification of master amongst a set of duplicates*

*1504* could be the best candidate here for the golden Master-record since it has the 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. Improved approach could be to 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.
  2. 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**

## Mohit Mayank, “String similarity- the basic know your algorithms guide” (February 3, 2019): <https://itnext.io/string-similarity-the-basic-know-your-algorithms-guide-3de3d7346227>

## “Deep Learning for NLP: Word Embeddings” (June 13, 2020): https://towardsdatascience.com/deep-learning-for-nlp-word-embeddings-4f5c90bcdab5

## Michael Gilleland, “Levenshtein Distance, in Three Flavors”: https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

## Kaitlin Coltin, Sumeet Siddhartha, Subadhra Parthasarathy, “Using NLP Enabled Text Search for Entity Disambiguation and Data Cleanup” (November 17, 2020): <https://amedeloitte.sharepoint.com/:f:/r/sites/AIIEEmergentCapabilities/Shared%20Documents/04.%20NLP%20-%20NUG/Text%20Search?csf=1&web=1&e=DFImLT>

## Murat Sariyar, Andreas Borg, “RecordLinkage: Functions for Linking and Deduplicating Data Sets” (August 25, 2020): <https://CRAN.R-project.org/package=RecordLinkage>

1. Jonathan De Bruin, “Python RecordLinkage Toolkit” (December 4, 2019):  
   <https://recordlinkage.readthedocs.io/en/latest/ref-index.html>)
2. “Scaling to large datasets”:  
   <https://pandas.pydata.org/pandas-docs/stable/user_guide/scale.html>)
3. Sunny Srinidhi, “Lemmatization in Natural Language Processing (NLP) and Machine Learning” (February 26, 2020):

https://towardsdatascience.com/lemmatization-in-natural-language-processing-nlp-and-machine-learning-a4416f69a7b6