EasyChair Preprint

№ 832

A Combined Semantic Search and Machine Learning Approach for Address Entity Resolution

Anne Moshyedi, Taylor Kramer, Amitava Gangopadhyay and Sujit Pal

March 15, 2019

A Combined Semantic Search and Machine Learning Approach for Address Entity Resolution

Anne Moshyedi
Innovation Labs
SWIFT, Inc.
Manassas, VA 20110
anne.moshyedi@swift.com

Taylor Kramer
Innovation Labs
SWIFT, Inc.
Manassas, VA 20110
taylor.kramer@swift.com

Amitava Gangopadhyay
Innovation Labs
SWIFT, Inc.
Manassas, VA 20110
amitava.gangopadhyay@swift.com

Sujit Pal
Elsevier Labs, Antioch,
CA 94531
sujit.pal@comcast.net

Abstract— ***We have developed a comprehensive prototype solution for a specific use case involving entity resolution for mailing addresses of financial institutions. Our objective was to find matches between user entry of misspelled or inaccurate addresses of business entities and their corresponding entries in a "gold copy" of complete and accurate mailing addresses (dictionary). Three distinct matching methods (PySolr, SoDA and Record Linkage) were used for a preliminary, yet diverse scheme of lookups in finding matches. These lookup processes may optionally be followed by search via a hybrid machine learning (ML) model via regularized logistic regression and hierarchical clustering using Dedupe. Our experimental results of elapsed times for searches using the three lookup methods on a variety of match types suggest that majority of the simpler matches are detected extremely fast (elapsed times: ~ 6 – 48 milliseconds) at the lookup stage, making it suitable for detecting simple and possibly most common errors in user entries for mailing addresses. The performance of ML models, on the other hand, is comparatively slower (elapsed times: ~ 174 – 201 milliseconds). Nevertheless, the hybrid ML model seems most suitable in cases where multiple ambiguities exist in user entry of addresses, and, as a result, the preliminary lookup methods may fail to detect possible matches. The precision and recall of the ML model on a sizeable test dataset are 0.89 and 0.94, respectively. These high scores on model performance suggest that the ML models can be applied successfully to entity resolution of mailing addresses. Our combined solution can be integrated with any enterprise software applications in order to provide both efficient and robust address matching service in cases where users enter mailing addresses as free-form texts that may carry inaccuracies.***

Keywords—semantic search, natural language processing, machine learning, deep learning, entity resolution.

I. INTRODUCTION

Natural language processing (NLP) is a field of Artificial Intelligence (AI) that enables software applications to understand and interpret human languages [1]. The theoretical foundations of a large variety of AI-powered NLP applications have been developed over the last several decades. Examples include speech recognition [2] "semantic" search engines [3], document classification [4], text summarization [5], and record deduplication [6, 7]. "Entity resolution" (also termed 'record linkage") is one such NLP problem where different manifestations of the same real world object ("entity") are linked or grouped together in order to find matches, eliminate duplicates or find relationships among them within single or multiple seemingly disparate datasets [8-10].

In this study, we have built a prototype solution to address a recurring business problem involving financial message transfers. The customers for financial message transfer often enter recipient financial institution or company addresses as free-form texts that, at times, do not match with any entries in the corresponding system of records for exact mailing addresses. Such cases of free-form text entry constitute a sizable portion of financial message transfers. The lack of exact matches in most cases, however, is due to typographical errors, inaccuracies or ambiguities in user entries of institution names and/or addresses rather than actual absence of records for intended recipient institutions. Here we report a comprehensive and robust end-to-end solution to this entity resolution problem to find exact or most likely matches between addresses from users' inputs and those from an existing larger address dataset persisted as a dictionary.

II. METHODOLODIES

There are four stages in the end-to-end workflow for our solution, as shown in *Fig. 1*. Here we give a brief description on each of the stages: input; preliminary lookup; ML model; and output.

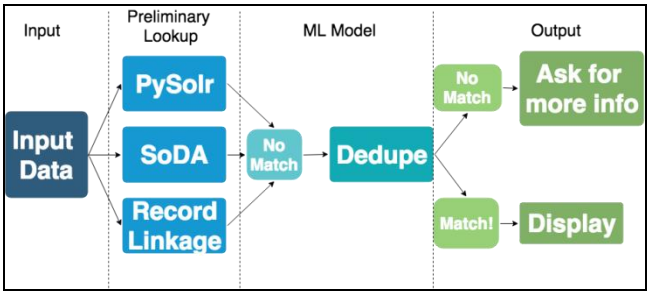

Fig.1 The entity resolution workflow in our solution

Stage I: User Input

Our entity resolution workflow starts with users' input of two datasets in csv format on a web application we have built natively using python and Apache Flask. We chose Flask for the deployment of our web application because it is a simple

and light-weight Python 3 microframe-work that can run on our Apache web server, and, as such, it involves minimal effort on its installation and setup. As part of user entry, the first dataset represents a "clean gold copy" of the data ("dictionary") that is cached into the memory of an index-based search engine (Apache Solr™). This clean copy of data is ideally collected and compiled over time to include all known valid and complete addresses that the users are likely to use. In the real world, these data are collected from various data vendors globally (e.g., United States Postal Service). For the purpose of our current proof-of-concept, however, we have used a small set (1000 records) of publicly available addresses of some financial institutions in the UK [11] as a dictionary. Next, the user uploads a sample dataset of records for analyses in order to find matches with those in the dictionary dataset.

Stage II: Preliminary Lookup

We have implemented three matching methods for this stage: [a] simple index-based query via PySolr (a Python wrapper for Apache Solr [12]); [b] dictionary annotator using SoDA [13] and SolrTextTagger [14]; and [c] Record Linkage (python Toolkit for fast lookup: [15, 16]). See below for more details on each of these lookup methods. The search results, as returned by each of the above three lookup methods, are ranked and sorted based on their matching scores. The highest-ranking match along with a success metric for each method is displayed on the web application user interface (UI).

Here we briefly describe each of the three lookup methods in more detail.

1) Index-based query via PySolr

Apache Solr™ is a widely used index-based search engine in the industry, and it enables advanced search capabilities and high scalability. The PySolr wrapper allows users to query indexes on a Solr server using a python client [12]. The address schema for our dictionary data contains fields for the company name, address, city name etc. (*Table 1*). The addresses data used in are mode dataset are publicly available mailing addresses of 1,000 business entities in the UK [11]. Table 1 displays a subset of five sample records out of which the first four are distinct and the last two records are two variants of the same physical addresses (duplicates). The "Name" field lists the business entity names and the "Code" field is populated with postal codes.

Table 1: Sample address data in the dictionary

ID	Name	Address	City	Country	Code
1	1 MOBILE LIMITED	30 CITY ROAD	LONDON	UK	EC1Y2AB
2	1 TECH LTD	57 CHARTERHOUSE STREET	LONDON	UK	EC1M6HA
3	23SNAPS LIMITED	16 BOWLING GREEN LANE	LONDON	UK	EC1R0BD
4	2E2 SERVICES LIMITED	200 ALDERSGATE STREET	LONDON	UK	EC1A4HD
5	2E2 UK LIMITED	200 ALDERSGATE STREET	LONDON	UK	EC1A4HD

The address data in the dictionary are cached as Solr indexes to enable fast and high-performance queries. Every time the user searches for an address, the Solr index data are queried via HTTP GET method. Each search query is processed with a request handler that calls a query parser, where the parser defines the search strings and parameters in order to specify the query. For example, when searching based on the company name and address fields, the query parser selects only those two fields for query execution.

We have leveraged the built-in support by Solr for both phrase queries and DisMax queries. The phrase queries are particularly helpful in detecting typographical errors, stemming and phonetic spelling. Similarly, Solr's built-in DisMax query parser is designed to process simple phrases entered by users with no need for complex syntax, making it particularly suitable for address entity resolution. Also, different weighting ("boosts") can be assigned based on the significance of each field in search terms, and Solr supports search for individual terms across several fields.

2) Dictionary based annotation via SoDA and SolrTextTagger

Our second matching method, SoDA, is a dictionary-based annotator for Apache Solr that supports both exact as well as fuzzy lookups across multiple lexicons [13]. Architecturally, it is essentially an HTTP REST microservice that enables a client to post a text corpus and retrieve a corresponding set of annotations. Annotations, in this context, refer to structured objects that carry information on entity identifier, matching text, offsets of matching texts within a given input text corpus, and the confidence score of a specific match. SoDA allows the client to specify the desired level of accuracy. Importantly, it performs more efficiently when the user breaks the dictionary down into smaller dictionaries based on the individual fields at sub-entity levels (city, state, country etc.). These new smaller dictionaries are then loaded into SoDA and Solr using a bulk loader, which stores the data in separate lexicons. SoDA can be implemented "on premise" or on the cloud (e.g., Amazon Web Services (AWS)). The AWS implementation is particularly better suited for cases where multiple users need to use SoDA concurrently and yet they are not required to install it separately on each instance. Instead, users can simply connect to the AWS machine specified in the URL of their HTTP requests and run the tests.

In order to query the lexicons, users can send their requests over HTTP POST using Python or Scala as JSON documents. We used a python client that exposes an Application Programming Interface (API) to SoDA. The Python version was chosen in this study because it helps to maintain consistency and seamless integration with all other matching methods we have implemented using tools and technologies within python ecosystem. The HTTP requests are sent through Jetty that serves as an HTTP web server, and processes requests, responses, and stores the lexicon data in a TSV (Tab Separated Values) format. SoDA itself interacts with SolrText-Tagger[3] on the Solr index.

SolrTextTagger is part of Apache Solr (7.4.0 and above), and it leverages Lucene's Finite State Transducers (FST) technology [14]. This tagger is commonly used to find entities

in large text, return pattern matching results in queries or to enhance "query understanding." FST refers to finite state automata consisting of a set of strings with optional edges between the nodes of strings. The connected nodes provide a complete representation of the target entity. The FST structure enables substring tagging on word level rather than at character levels. FST is written as an immutable byte array, and, consequently, it is efficient with respect to memory usage and elapsed time to execute search queries. Further, when new dictionary entries are encountered for matching entities, FST, which is index-based, does not need to be rebuilt; the user only needs to add the new entries.

Two separate FSTs were used in our implementation in order to expand the abilities of the search method. The first FST contains every word in the dictionary with a unique integer id that can be used as a substitution. The second is a word-phrase FST that is used based on those unique ids and allow the tagger to account for prefixes and suffixes in words and word phrases.

Next, SoDA, which is built on top of SolrTextTagger, is used to hold the entity names and unique identifiers. Our approach allows us to perform searches with multiple matching modes and methods. This flexible approach thus enables discovery of varieties of matches. For example, the address dataset can be uploaded to SoDA server in multiple formats, and is queried either concurrently or until a strong match at pre-defined level is found. The TSV files with separate address fields are searched first, followed by searches on a second file in which the entire address is stored together. Additional methods were also used to further strengthen this approach. They include the use of non-streaming matches of phrases against entries, text annotations against specified lexicons and the use of multiple stemming algorithms, such as, Porter stemmer [17, 18] and KStem stemmer [19].

3) Pattern-based matching via Record Linkage:

The third lookup method in our solution was via use of "Record Linkage Toolkit [16]." It is a python implementation widely used in entity resolution problems. It provides the user with capabilities for fast lookup via powerful matching algorithms and also provides an optional capability to build ML models. For the purposes of this study, we have, however, used the toolkit only for its fast lookup capability.

Similar to the other two lookup methods discussed earlier (PySolr and SoDA), the same two separate datasets were also used in Record Linkage as user inputs. The first stage of processing employs block and sorted neighbor indexing methods to reduce the volume of data that is sent to comparison algorithms. Under block indexing, records with exact matches between any or all fields produce outputs that show "matched" records. A sorted neighborhood indexing method, however, combines the two datasets, sorts them alphabetically, and finds the alphabetically closest dictionary record(s) for each user entry. These selected "neighbors" are subsequently sent to the classification stage. In the event that the previous two indexing methods deem insufficient, full indexing may be applied for a more comprehensive scheme of search.

In the classification stage, the indexed results are compared against a wide variety of matching metrics. For string-based data, the Record Linkage program can use a variety of algorithms including *jaro*, *jaro-winkler*, *levenshtein*, *qgram*, and *cosine similarity [16]*. These algorithms compare records and compute their matching status. For our specific datasets, jaro-winkler and levenshtein algorithms yielded the most consistently accurate results. Positive matches are reported when the algorithm finds a match between the values in an input field and the corresponding dictionary field under a specified "similarity threshold." This process is repeated for each input field, and it results in a matrix of record pairs classified into ones and zeros. The 1's in this matrix signify a positive match for the particular field of that given record pair, whereas 0's indicate a negative match status. Once this initial process to find match is complete at individual field levels, the program computes the match status of each record as a whole.

In the next step, the matching process uses the previously created matrix, and, for each record, determines whether or not a sufficient number of fields in the record match in order to be considered a complete and successful match. For our implementation, the program first checks for agreements within all five fields, and if no matches are found, it subsequently checks for agreement within at least four of the five fields in the user entry. The pair of addresses with the highest matching score in the list of search results is returned to the user as the final output.

Stage III: ML Model

Any user entry of address that cannot be matched via the three preceding preliminary lookup methods is subsequently sent to the ML model stage for further processing. The ML model was trained via Dedupe python library that uses a combination of two separate ML models (regularized logistic regression and hierarchical clustering [20]; discussed later). Similar to the lookup methods discussed above, the first step in using ML model via Dedupe is to load two datasets in csv format: the dictionary and the simulated user entry datasets. The dictionary data contain a list of complete and accurate company addresses that potential users are likely to use. The second dataset includes a simulated list of addresses that potential users are likely to spell or type in differently. For example, the user-entered addresses may contain ambiguities, missing values, incorrect fields, improper formatting, abbreviations, truncations, and other possible variations. The ML model compares these two datasets, and determines matches within pairs of records.

The first step to determine matches between strings or text corpora using Dedupe involves calculation of "similarity scores" via different measures. The method in our implementation uses the *Affine Gap Distance [21]*, which is a string metric used to score alignments between strings. The Affine Gap Distance counts the minimum number of changes, such as substitutions, deletions, and additions required to achieve an exact match between the two strings. This numeric value thus represents the pairwise similarity between two strings. Additionally, each address is split into component fields (city, state, postal code etc.), and each individual field is then compared with its corresponding dictionary value.

The gap distance is calculated based on individual fields within a single address (city, state etc.) rather than the entire address as a whole. This is particularly useful in our specific use case because some fields within an address, such as, the company name, may carry higher importance than other fields, such as postal code. Different numeric weights (discussed later) are assigned to each field (company name, address, city, country, and postal code). These weight factors are multiplied by their corresponding individual gap distance. The final gap distance of the entire record is given by the weighted sum of distances for all address fields.

In order to determine the numeric weights for each field and the threshold values for gap distance, a supervised ML method was employed in our model. For our training data, we manually created labeled sample pairs across our address dataset, where each pair was marked either a match or distinct. This is a mode of training method in ML, called supervised "active learning." The labeled sample pairs allow a *regularized logistic regression model* to learn from training data and assign weights for each field. The individual weights, combined with the total gap distance of the entire address, yield an estimate of probability for pairs of records being duplicates, which, in turn, is indicative of the likelihood of a match between them. Thus, the predictive ML model is essentially based upon the binary dependent variable (match or distinct) derived from active learning and on the respective gap penalties associated with each address.

The ML model also uses a method called "blocking," where different addresses are separated into distinct blocks with some common features between them. Because similar entities are likely to share some common feature(s), the search algorithm can limit comparisons among addresses only within the same block and not across separate blocks. This reduces the number of required searches for a match, and, consequently, results in improved efficiency in performance of search algorithm of the ML model. Dedupe uses two sets of blocking rules, namely, predicate and index blocks. Predicate blocks bundle together records that share a similar trait or characteristic feature. One such feature, for example, can be the same few characters that two or more company names start with in the name field of our address dataset. In the case of index blocks, Dedupe creates a data structure, called inverted index, which is populated with all the unique values in a specific field. Records with at least one similar value for its index are grouped together as a single block. Further, Dedupe uses the Greedy Set-Cover algorithm [22] in order to select a set of blocking rules that, on one hand, maximizes the checks for duplicates and, on the other, minimizes the required number of comparisons. This algorithm selects addresses with lowest weights and includes them within the most prospective candidate blocks for match discovery. At the same time, the algorithm does not unnecessarily increase the size of the block and number of comparisons. Combined, these methods provide the algorithm with a fast yet robust search mechanism for pattern matching.

Once the probabilities are calculated for pairs of record being duplicates or not, Dedupe uses a method, known as *hierarchical clustering with centroid linkage*, in order to group potential duplicates [21]. For example, let us assume that one pair of addresses (A and B) and a second pair (B and C) have high probabilities of being within-pair duplicates, 0.7 and 0.8, respectively. While we do not have direct measure of probability of match between A and C in this case, each of them is indirectly linked via high match probability with addresses B. The address B, based on this algorithm, would be considered the centroid for the same cluster in which all three addresses are constituent members. Note that this clustering algorithm relies on a computed value of probability threshold for group membership within a single cluster. The calculation of this threshold involves use of an F-score, which, in turn, is calculated with an optimum tradeoff between precision and recall. Precision, in this context, is a measure of how valid the predictions by the Dedupe model are, whereas recall refers to the sensitivity of the model in detecting true matches. In order to calculate the precision and recall, a random sample of the blocked data is taken and the pairwise probabilities are calculated. In our specific use case, the prediction of false match ("false positive") is less desirable than missing a true match (false negative). This is because a false match could potentially lead to an undesirable consequence of monetary transaction being delivered to an unintended recipient entity. Accordingly, a higher weight was placed on precision than on recall for our model performance. This was accomplished by setting an optimum threshold for match definition.

Finally, in the event no match is found even via the hybrid ML model, the user is prompted to enter additional information and the search process is repeated starting with stage I.

Stage IV: Output

The same Flask web application user interface (UI) that is used for user input is also used to display the results from our lookup methods and the ML model. The user can view all results from the three lookup methods and the hybrid ML model displayed individually on this presentation layer. This provides the user with greater choice in accepting a particular match results from any single matching method. Also, the UI allows the user to run each search method sequentially or all four methods concurrently.

III. RESUTLS AND DISCUSSIONS

Table 2: Time (milliseconds) elapsed for different match types versus different matching methods

Match Types	PySolr	SoDA	Record Linkage	Dedupe
Exact match: 1 MOBILE LIMITED 30 CITY ROAD LONDON	7.46	17.07	47.31	173.64
Incorrect wording: 1 MOBILE LIMITED 30 CITY STREET LONDON	5.67	16.73	44.73	200.72
Multiple ambiguities: 1 mobil lim 30 city rd lon uk	4.78 (No match found)	50.83 (No match found)	135.77 (No match found)	175.53

Our results on three example match types (exact match, incorrect wording and multiple ambiguities) are shown in Table 2 along with their corresponding elapsed times (in milliseconds) in returning results via the three lookup methods and the ML model. These results suggest that PySolr can detect the most common types of inaccuracies in user entries and successfully match them with corresponding accurate addresses. Further, the results returned via PySolr are consistently the fastest among all three preliminary lookup methods. It is also easily scalable to large set of dictionary data, and the lookup works seamlessly when updates are made in the dictionary. Moreover, PySolr is useful in setting up the foundation of Solr, which can also be used with SoDA. Combined, PySolr provides a simple, fast and scalable way to find matches between records when the threshold value for a given match is not critical.

As for the use of SoDA in our solution, one of its advantages is that it places higher importance on recall over precision. As such, it uses a fast and dynamic programming algorithm to calculate the edit distance between two text corpora. For our use case, we estimated an optimum threshold value for this edit distance (25) via multiple trial and errors. This is to ensure that the "gap penalty" between the user entry and the exact address are not exceedingly high for a desired conclusive match. The use of an optimum threshold can help reduce the number of false matches.

Another advantage of using SoDA in our entity resolution problem is its ability to store aliases. When the dictionaries are loaded into lexicons, the user can specify a list of words and their variants that are frequently spelled differently. For example, if the user specifies "RD" or "Rd" as known aliases for "Road," SoDA can identify them as exact matches. These differently spelled variants are evaluated the same way as their primary dictionary records. This capability of SoDA is thus particularly useful in our use case, as abbreviations or acronyms for both company name and street addresses are common in user entries.

Notably, both SoDA and Record Linkage return match results only when the matches satisfy a defined threshold for confidence level in each method. The user sets the threshold values based on trial and error on known test data. Also, these threshold values can be customized based upon desired level of confidence and the data format used in a specific use case.

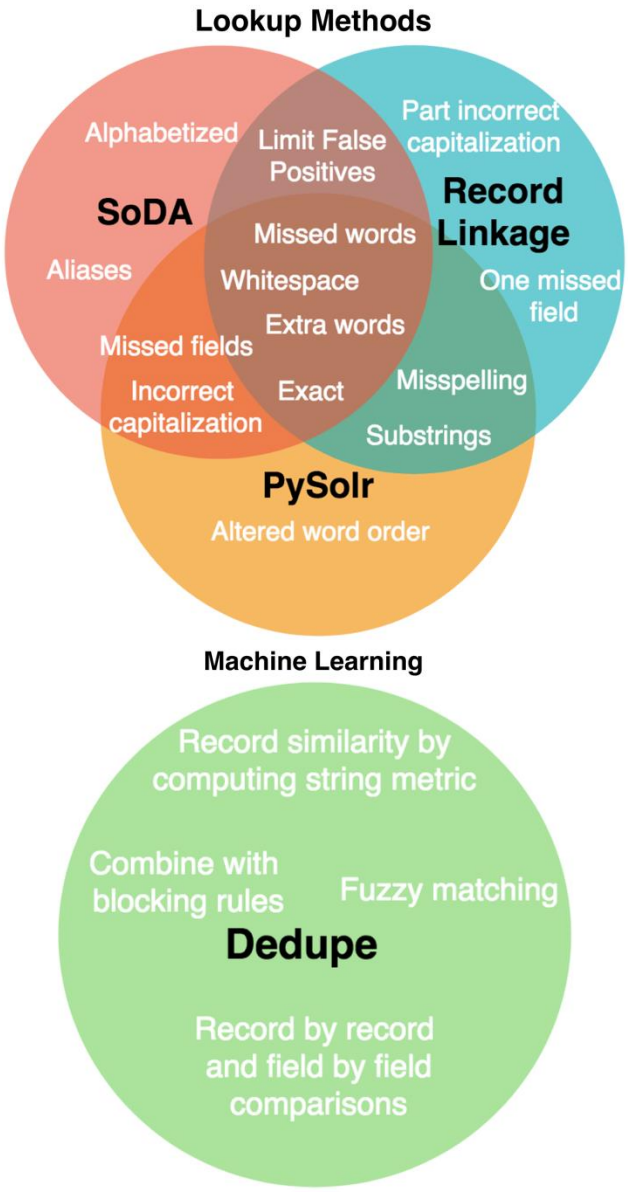

Fig.2 The relative strengths of each method used in our solution for entity resolution

While the three lookup methods are all useful in finding matches, they show different degrees of effectiveness based upon the match types (Fig. 2). For example, they all can correctly detect matches in cases where only the name string in the user input is incorrect, but the address string is exact and accurate (no deviation from the dictionary entry). Similarly, as desired, all lookup methods can ignore extra white spaces and also account for special characters. There are specific match scenarios, however, where these lookup methods show different degrees of relative efficacy. For example, unlike PySolr or SoDA, matching via Record Linkage may fail due to its case sensitivity and sensitivity to the order in which different substrings appear in the input address. Further, there are other specific match types, such as missing substring for which Record Linkage and SoDA fail to find matches, but PySolr succeeds (e.g., input: "1 mobile 30

city London." versus dictionary: "1 MOBILE LIMITED 30 CITY ROAD LONDON UK EC1Y 2AB"). Moreover, we note examples of inaccurate street names that both Record Linkage and SoDA can correctly match, whereas the identical search phrases fail to find matches via PySolr. For example, when the input address is "1 MOBILE LIMITED 400 CENTER POINTE LANE LONDON UK," PySolr incorrectly suggests a completely different address as a possible match. Thus, these three lookup methods display their individual strengths and weaknesses that can have different implications for their suitability in specific use cases.

All three lookup methods are, however, consistently one or two orders of magnitude faster (~ 6 – 48 milliseconds) than the ML model (~173 – 200 milliseconds). Thus, our results suggest that the three lookup methods can provide an efficient solution for simpler match types (discussed in the last paragraph) where they are likely to find consistent and identical search result for a given user entry.

The use of three separate lookup methods reduces the possibility of "false positives" (false matches), and provides a higher degree of confidence on each match. However, in the event that identical matching results are obtained consistently across all three lookup methods for a given user entry, the user may optionally accept the final result from this lookup stage and forego the use of ML models in the next stage (Stage III; as discussed in section 2). Thus, the lookup stage can potentially limit the use of the computationally more expensive ML model to only those cases when either no matches are found at the lookup stage (Stage II) or search results are not consistent among the three lookup methods. Depending on the need and preference of the user, our solution provides options to search each method sequentially one after the other, or run them all concurrently.

As noted, the ML model seems most suitable in cases where the preliminary lookup methods fail to return matches due to multiple ambiguities. For example, the last sample in *Table 2* includes minor ambiguities in all the address fields (dictionary: "1 MOBILE LIMITED, 30 CILTY ROAD, LONDON, UK" versus user entry: "1 mobil lim, 30 city rd, lon, uk"). In this case, all three preliminary lookup methods fail to find a match, whereas the ML model successfully returns a match between the user input and the dictionary entry. Also, the ML model yields overall high values of precision and recall (0.89 and 0.94, respectively) noted over a large number of searches. These high values (~90%) indicate that our ML predictions yield only a limited number of false positives (high precision), while the searches via ML are still very sensitive to all potential matches (high recall). Combined, these results suggest that the Dedupe ML model can provide a reliable solution to the address entity problem.

As noted, the use of the ML model, albeit computationally more expensive, becomes an essential part of our overall solution. It is particularly vital in cases where multiple ambiguities exist in user entries. Because it is virtually impossible to anticipate all possible ambiguities and misspellings in each address that a user may enter, it is impractical to prepare an exhaustive set of rules *a priori* in order to account for all likely match scenarios. Accordingly, the use of the ML model is critical in our overall solution of supervised learning method via Dedupe (*Fig. 2*). This is because the ML model does not require rule-based pattern matching via, for example, regular expression. The model can learn from the labeled training data and detect the patterns for matching that can be applied on new user entries. Furthermore, as additional marginal cases of ambiguities are encountered over time, they can be appended to the existing training dataset, and the ML model can be retrained with the revised dataset in an attempt to improve the accuracy in model predictions.

IV. CONCLUSIONS

We have explored the efficacy of a variety of fast lookup methods and ML model solutions in entity resolution of postal addresses. Our experimental results suggest that the simple and perhaps most common user errors in entering mailing addresses can be rectified via a variety of preliminary lookup methods. These methods in our solution involve the use of fast index-based search and dictionary annotation via several Python software packages (PySolr, SoDA and Record Linkage). Most common user errors are detected extremely fast at this stage (elapsed time ~6 – 48 milliseconds). For the ML model using Dedupe, however, the elapsed times for searches are orders of magnitude higher (~ 174–201 milliseconds) across all match types. Nevertheless, the use of the ML model seems most suitable in detecting more complex match types where multiple ambiguities exist in user entries. The ML model yields consistently high values of precision and recall (0.89 and 0.94, respectively), suggesting its potential use in address entity resolution problems.

Depending on specific use cases, the proportions of false positives and false negatives in the lookups and ML model results may have diverging implications for their use in entity resolution problems involving monetary transactions. Accordingly, our choice of different methods in both lookup and ML models reflects an optimum balance among speed, accuracy and caution. Further, all of the four methods (three lookup and one ML methods) employed in our solution can be customized based on specific user data and requirements for address matches.

Our comprehensive solution is applicable to any entity resolution problems in any area of business where the matching of records among multiple datasets is desired. This flexible yet advanced solution is scalable, easy to integrate with other enterprise applications, and it can potentially reduce transaction processing times significantly.

Finally, as more real data of user entries are collected over time, the precision and recall of the ML model are expected to further improve. Also, our additional experimental work is currently underway that involves use of additional ML models and deep learning (DL) models in order to include more complex match types and leverage a majority voting algorithm to minimize any bias within the ML or DL models.

Disclaimer

The views articulated in this paper are personal to the authors and do not represent the views of their employers or any other organization.

Acknowledgement

This research was funded and supported by the Innovation Labs and Summer Internship Program at SWIFT. We thank Soumitra Dutta (Cornell Univ.) and Uwe Aickelin (Univ. of Melbourne) for their very constructive informal reviews. We also thank Chris Martis, Kevin Dize, Nancy Murphy and Sudhir Pai for their help with different steps in preparation for the manuscript. We appreciate the help and support extended by Peter Ware in this collaborative research work.

REFERENCES

[1] Powers, D. M. W., & Turk, C. C. R., Machine Learning of Natural Language, Springer-Verlag, 1989.

[2] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., & Kingsbury, B., Deep Neural Networks for Accoustic Modeling in Speech Recongnition: The Shared Views of Four Research Group, Signal Processing magazine, IEEE, 29.6, 82-97, 2012.

[3] Dong, Hai & Hussain, Farookh & Chang, Elizabeth., A survey in semantic search technologies. 2008 2nd IEEE International Conference on Digital Ecosystems and Technologies, IEEE-DEST, 403 – 408, 2008.

[4] Yang, Z., Yang, D., Dyer, C., He, X., Smola, A., & Hovy, E., Hierarchial Attention Networks for Document Classification, Proceedings of NAACL-HLT, 1480-1489, 2016.

[5] Brigitte, E-N., Summarizing Informations, Springer, 1998.

[6] Dunn, H. L., Record Linkage, American Journal of Public Health, 36, 12, 1412-1416, 1946

[7] Newcombe, H. B., Kennedy, J. M., Axford, S. J., James, A P., Automatic Linkage of Vital Records, Science, 130, 3381, 954-959, 1959.

[8] Fellegi, L. P. & Sunter, A. B., "A theory for record linkage," Journal of the American Statistical Society, vol. 64, no. 328, 1969.

[9] Christen, P., A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. Z, NO. Y, ZZZZ 2011

[10] Cohen, W. W., Ravikumar, P., & Fienberg, S., "A comparison of string distance metrics for name-matching tasks," in Workshop on Information Integration on the Web, held at IJCAI'03, Acapulco, 2003.

[11] Dictionary data source: Open source data available at: https://github.com/moshyedi/Data-Source/blob/master/companies_final.csv

[12] Kocherhans, J., Kaplan-Moss, J., & Lindsley, D., pysolr, GitHub Inc., 2018, https://github.com/django-haystack/pysolr.

[13] Pal, S., Solr Dictionary Annotator, 2015, https://github.com/elsevierlabs-os/soda.

[14] Smiley. D., and Westenthaler, R., SolrTextTagger, 2013, https://github.com/OpenSextant/SolrTextTagger.

[15] Christen, P., Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media, 2012.

[16] de Bruin, J. d., Record Linkage Toolkit Documentation, Release 0.12., 2018, https://media.readthedocs.org/pdf/recordlinkage/latest/recordlinkage.pdf

[17] M.F. Porter, An algorithm for suffix stripping, Program, 14(3), 130−137, 1980.

[18] Porter, M., The Porter Stemming Algorithm, Revised Version, https://tartarus.org/martin/PorterStemmer/

[19] Krovetz, R., Viewing Morphology as an Inference Process. In: Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, S. 191–203, 1993.

[20] Gregg, F., & Eder, D., DataMade & Contributors, Dedupe 1.9.3., The MIT License (MIT), 2014, https://github.com/dedupeio/dedupe.

[21] Gotoh, O. An improved algorithm for matching biological sequences. Journal of Molecular Biology, 162 (3), 705-708, 1982.

[22] Greedy Set-Cover Algorithms, Neal Young, 2008. Encyclopedia of Algorithms, 379-381.