PostMatch: A Framework for Efficient Address Matching

Darren Yates[1(✉)], Md Zahidul Islam[1], Yanchang Zhao[2], Richi Nayak[3], Vladimir Estivill-Castro[4], and Salil Kanhere[5]

[1] School of Computing and Mathematics, Charles Sturt University, Bathurst, Australia
{dyates,zislam}@csu.edu.au
[2] Data61, CSIRO, Canberra, ACT 2601, Australia
yanchang.zhao@csiro.au
[3] School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia
r.nayak@qut.edu.au
[4] Universitat Pompeu Fabra, Barcelona, Spain
vladimir.estivill@upf.edu
[5] University of New South Wales, Sydney, NSW 2052, Australia
salil.kanhere@unsw.edu.au

Abstract. Matching lists of addresses is an increasingly common task executed by business and governments alike. However, due to security issues, this task cannot always be performed using cloud computing. Moreover, addresses can arrive with spelling errors that can cause non-matches or 'false negatives' to occur. Our proposed framework, PostMatch, provides a locally-executed method for address-matching that combines the open-source 'Libpostal' address-parsing library with our 'postparse' post-processor code and machine-learning. PostMatch provides improved parsing accuracy compared with Libpostal alone, approaching 96.9%. The matching process features the Jaro-Winkler edit distance algorithm together with XGBoost machine-learning to achieve very high accuracy on public data. PostMatch is open-source (GPL3 licensed) and available as R script code on Github.

Keywords: Address matching · Data matching · XGBoost

1 Introduction

The need to match two lists of addresses can be a common occurrence in many organisations. For example, it could be to link or merge two customer databases from different retail store records or to identify fraud by matching addresses entered into application forms with those on a known 'black list'. However, address matching is often more complex due to the nature of the data itself. Address components, such as street name and town/suburb name, can be swapped or misaligned. This complexity can be the result of clients or customers

Y. Xu et al. (Eds.): AusDM 2021, CCIS 1504, pp. 136–151, 2021.
https://doi.org/10.1007/978-981-16-8531-6_10

from different regional, cultural and/or language backgrounds entering address components differently into the same form. Added complexity can also arise from address components that have been misspelt or are incomplete. Moreover, the larger the address lists that are to be matched, the more system resources and search performance become a concern. Cloud computing services can be used to overcome many of these issues, however, there are many applications, such as law enforcement, where a locally-executed solution is required. This local-processing requirement could be due to the nature of the address lists or cyber security concerns with using online resources. As a result, matching the addresses from two large lists both accurately and efficiently can create significant challenges.

Other methods have been suggested previously to tackle some of these issues. For example, Christen and Belacic [6] developed an automated probabilistic solution employing Hidden Markov Models (HMMs) to guide the normalisation and parsing process. Address matching is a specific application of 'record linkage' theory pioneered by Fellegi and Sunter [8]. More recently, Koumarelas, Kroschk, Mosley and Naumann [11] investigated methods for combining address geocoding with a similarity measure to maximise address matching accuracy.

This paper introduces a machine-learning based address-matching framework called 'PostMatch'[1] to address these issues using locally-executed methods. The framework, shown in Fig. 1, consists of three key sections: parsing, normalisation and matching. The parsing process identifies and separates various address components, normalisation modifies those address components to fit in with agreed standard identifiers and features, and matching detects addresses common to both lists.

The research contributions of this paper include:

- identification of eight key address fields that summarise Australian addresses,
- a post-processing method called 'postparse' that improves the accuracy of the Libpostal open-source address parsing and normalisation library,
- the combination of machine-learning and string edit-distance measures to deliver high address-matching accuracy, and
- experimental evaluation of the proposed framework on public address data.

Although Australia is the example focus of this work, the methods used here could be applied to other national address formats following similar principles.

2 Related Work

Modern address matching can be traced back to work into 'record linkage' theory by Fellegi and Sunter [8], who applied a mathematical approach to the problem of matching two records using the records' components by comparing them as a vector of element pairs. The approach determined three possible matching labels. The first two are 'link' (indicating the two records match) and 'non-link' (do not match). Both of these were considered as definitive or 'positive' decisions. The

[1] https://github.com/darrenyatesau/postmatch.

third level, 'possible link', occurs when the match fails to meet the preconditions for either of the two positive outcomes. The aim is to increase the likelihood of a 'link' or 'no link' result, whilst minimising a 'possible link' outcome, which would require more expensive resources to link manually.

While record linkage has been well researched since then [9,10,14,16], Fellegi and Sunter identified initially that the accuracy of record matching would be directly affected by the completeness and accuracy of the initial records [8]. Within the specific area of address matching, these two issues noted by Fellegi and Sunter appear as missing or misspelt address fields. Thus, research has focused on achieving high matching accuracy in applications where address data is incomplete or contains errors that would ordinarily compromise that accuracy.

Christen and Belacic [6] tackled these issues of address cleaning and standardisation or 'normalisation' through hidden Markov Models (HMMs). They used this type of finite state machine in an attempt to identify various components from address strings. It also featured the Australian Geocoded-National Address File (G-NAF) database [6] as a verification source to determine that addresses being matched were genuine. More broadly, their approach features a four-step process involving 1) address cleaning or normalisation, 2) tagging of address component fields, such as street, town, state and so on, 3) segmenting the tag lists into appropriate output fields using the HMM and 4) verification against the G-NAF database.

A popular open-source software solution for address parsing and normalisation on a global scale is 'Libpostal' [1]. This library executes within Linux-based computer systems and supports application programming interfaces (APIs) for several programming languages, including Python and R. Libpostal authors claim it can support addresses from around the world. The Libpostal library offers two essential functions: 1) normalisation, to standardise an address using commonly-agreed and region-specific descriptors and features, and 2) parsing, to separate the address into individual components.

Even with address records parsed and normalised, the issues remain of how to accurately determine whether a pair of addresses are 'matched', 'not matched' or 'maybe matched'. In particular, there has been notable research into quantitative methods for determining the extent of matching between pairs of text-based records or 'strings', with numerous string-similarity measures developed, including Levenshtein [12] and Jaro-Winkler [15]. String-similarity measures quantify the steps or 'cost' required to transform one text string to another and are commonly used in record linkage applications.

By contrast, Arasu, Ganti and Kaushik developed a more generalised alternative [2]. Their work builds upon a concept of 'similarity join', whereby two databases are tested by each combination of record pairs against a similarity measure function, with those pairs that exceed a preset threshold being recorded. They acknowledged that despite the availability of numerous similarity or 'distance' functions, no one measure excels in every application. Their solution expanded the similarity join into a more extensive set of joins featuring multiple similarity functions to boost accuracy.

Practical applications of similarity measures have included matching health records. Bell and Sethi [3] developed a matching framework that also covers parsing and normalisation. A crucial part of the normalisation process was incorporating the ethnic origins of patients, to account for how patients from different backgrounds enter their name details. Their record-matching process features a composite matching formula that included string similarity and phonetic matching.

Cohen, Ravikumar and Fienburg [7] experimentally tested a number of common string distance measures available in an open-source Java programming language toolkit called 'SecondString'. One of these distance measures tested was the Levenshtein string distance function [12]. In broad terms, the function counts up the number of character transitions to move from one string to another. These transition options include insertion, deletion and substitution. Unlike the standard Hamming distance measure, the Levenshtein function handles strings of different length, an important factor for address matching.

We also draw attention to other experimental results in [7]. A test of 11 datasets comparing a series of distance measures revealed that, on average, the standard Levenshtein distance measure performed lower than the other measures on ten of those datasets. However, the one dataset it did excel on among these tests was a synthetic 'Census' dataset, consisting of names and addresses. Moreover, the results were further improved by applying the Winkler variation [15] to the Levenshtein measure. This variation aims to take into account common prefixes in strings by weighting them more favourably.

A more common non-scaled variation of the Levenshtein measure is the Damerau-Levenshtein (DL) function [13], which, in addition to insertion, deletion and substitution, includes the option of character transposition, or swapping of two adjacent characters, as part of the edit distance calculation. Each transformation carries an equal penalty or weight of one and the series of transformations that converts a string to the other string with the fewest steps becomes the edit distance for those two strings. However, work by Christen [5] on matching personal names showed that while the DL measure generally performed well in testing, alternatives, including the Jaro-Winkler (JW) algorithm [15], outperformed DL, confirming earlier results [7]. The JW measure has similarities with DL in that it accounts for insertion, deletion and transposition on characters. However, it differs by taking into account the number of characters in common between the two strings in addition to the number of transpositions relative to the length of the longer string. The tests [5] show that not only did the JW algorithm outperform DL in terms of matching accuracy in all tests, JW also processed the same number of tests at approximately twice to three times the speed of DL.

Thus, as noted, there may well be numerous distance measures, but it is apparent that no one measure is perfect for every application. However, while this might appear to strengthen the case for incorporating multiple edit measures, such as a set-similarity join, to cover all eventualities, the extra cost in time required to process these measures all but ensures there can be no gains without losses.

The research reported here shows that machine-learning techniques can be used to either enhance or replace the traditional rule-based solutions that are commonly applied to record linkage. One of the more recent machine-learning developments is XGBoost [4]. It is a gradient tree-boosting algorithm boasting high scalability, making it well suited to tasks such as address matching, where the number of addresses required to be checked may be considerable. It also offers high performance in single-machine use. To our knowledge, no research has yet been published into the efficacy of XGBoost in address-matching applications. Moreover, the combination of distance measures and XGBoost machine-learning forms one of the novel contributions of our PostMatch framework, which we will now cover in depth in the next section.

3 The PostMatch Framework

Our proposed PostMatch framework for efficient address matching is shown in Fig. 1. It consists of three key sections: 1) parsing, using the Libpostal open-source library, 2) normalisation, provided by our 'postparse' post-processing method, and 3) list-matching, involving set-similarity calculations based on edit distance and machine-learning algorithms.

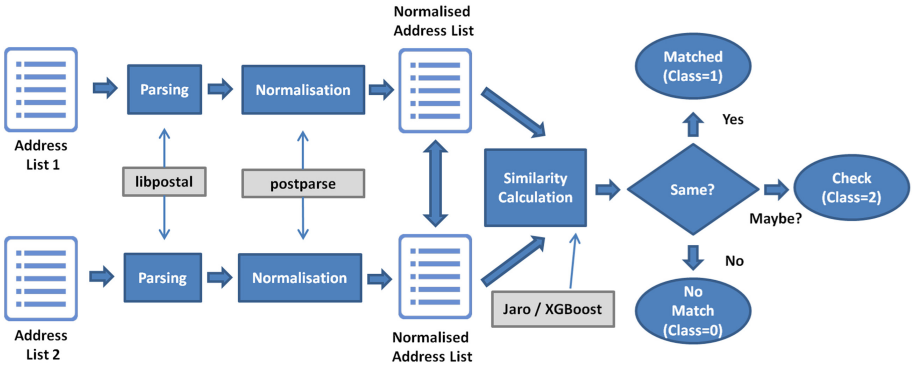

Fig. 1. The PostMatch address-matching framework

The framework begins with two separate address lists in the form of text files, where each address is a single string of text, one address per line. At first, the two lists are processed separately by parsing each address into component fields and then by normalising to reduce variations of values within fields. After that, address pairs from the two lists are compared using a combination of Jaro-Winkler edit-distance [15] and XGBoost machine-learning [4] algorithms. If an address pair is classed as 'matched' (class value of '1') or 'maybe-matched' (2),

the class value and two addresses are combined to form a new record in the results matrix. After all address pair combinations have been tested, the results matrix can be displayed to the user.

3.1 'Site' and 'Locality' Fields in Addresses

With Australian postal addresses as a focus, our research has identified eight descriptor fields that can summarise all postal addresses within Australia's states and territories. These descriptors and their abbreviations are shown in Table 1. Moreover, these eight fields can be divided into two groups. The 'site' group, consisting of Fields 1 to 4, indicate an individual site, such as a building or subsection of a building (as in a level or unit/apartment, or even a post office box). The 'locality' group, featuring the four remaining fields, covers progressively larger groups of sites, from streets to towns and regions. A fundamental differentiator between the two groups is that while the 'site' group fields are, in some respects, optional (and generally mutually exclusive, in the case of PBox and HseNo fields in Australian addresses), the final three fields of the 'locality' group - Town, State and PCode - are essential to ensure accurate postal location identification. This can be seen in the three fictitious but standards-compliant Australian addresses examples shown in Fig. 2.

Table 1. The eight descriptor fields for identifying all Australian postal addresses, including abbreviations, content type and assigned group in our research.

Field no	Address descriptor	Abbrev'n	Content type	Group
1	Post Office (PO) Box	PBox	Alpha-numeric	Site
2	Apartment/Unit number	Unit	Alpha-numeric	Site
3	Building Level number	Level	Alpha-numeric	Site
4	House/Site number	HseNo	Alpha-numeric	Site
5	Street name	Street	Alpha-numeric	Locality
6	Suburb/Town name	Town	Alpha-numeric	Locality
7	State name	State	Alpha-numeric	Locality
8	Postal code	PCode	Numeric (4-digit)	Locality

3.2 Address Parsing

Addresses can be described as region-specific location descriptors that follow local conventions or customs. Thus, it is not only important to identify the key address components, but also to identify where those components occur within an address. The general order for Australian addresses is in increasing order of scale, that is, building or 'site', street, town, region/state. The 'postal code' field is usually associated with the 'town/suburb' field, though in Australia, these two

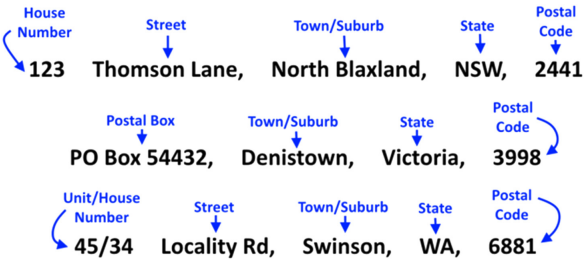

Fig. 2. An example of three (fictitious) compliant Australian-standard addresses. Each line in black is an address and the text in blue shows various fields in addresses.eps

do not always form exclusively distinct pairs and it is not uncommon for smaller localities to share a common postal code. By contrast, some towns may also have more than one post code. Examples of valid (albeit fictitious) addresses and their fields are shown in Fig. 2. The task of separating these components into identifiable fields is referred to as 'parsing'. The complexity of this task arises from the need to be able to recognise the various fields simply from the address format itself. The role of parsing within the PostMatch framework is to separate the address components appropriately into the eight component address fields (see Table 1), using Libpostal [1], an open-source address normalisation and parsing library.

3.3 Normalisation

Once the address fields have been identified and separated, each field has to be normalised for common extensions and abbreviations to improve matching accuracy. The normalisation process firstly reduces redundant information, such as 'PO Box 123' to simply the PO Box number itself. This removes the potential for post office prefixes to causes non-matches for matched fields. However, in Australia, normalisation should also handle a common forward-slash '/' abbreviation typically used to separate apartment/unit from building site descriptors. For example, '12/34 qwerty st' is a common short-form representing 'Unit 12, site or building number 34, Qwerty Street'. Another example is the 'street' field, where the word 'street' itself is often abbreviated to 'st', 'avenue' to 'ave' or 'av', and 'road' to 'rd'. Moreover, Australia's eight states also have common abbreviations, such as 'NSW' for New South Wales, 'QLD' for Queensland and so on. A common approach is to turn the name-based fields into lower-case and expand all abbreviations to their root form. This normalisation can be achieved through look-up tables or simple rule-based substitutions.

Thus, the task of normalisation is to process abbreviations and other common short-forms back into a set of consistent alphanumeric site descriptors to simplify the matching process and improve matching accuracy. This is the task of our 'postparse' post-processing method (see Sect. 3.4) that is applied to the address records after parsing is complete.

3.4 The 'Postparse' Post-Processing Method

While Libpostal is capable of handling a wide array of address formats from countries around the world, a number of issues were observed during testing with regards to incorrect normalisation of certain Australian-specific address components. For example, the Australian state abbreviation 'WA', commonly used for 'Western Australia', would expand to 'Washington', one of the United States. It also incorrectly handled addresses for Northern Territory, the Libpostal normalisation expanding the short-form 'NT' to 'Intermediary', as well as the 'street' reduction, 'st', to 'saint'. Further, it did not recognise the forward-slash '/' short-form for unit or apartment, instead replacing it with a space character. Similar issues with Libpostal have also been identified by Koumarelas et al. [11].

Our method for overcoming parsing and normalisation issues in Libpostal is to identify patterns of inconsistency and to correct them through a separate process after the event, commonly referred to as 'post-processing'. The PostMatch framework achieves this through an R script called 'postparse' that slots into the PostMatch framework following completion of address parsing by Libpostal.

In general, the traditional workflow process would be to use Libpostal to first normalise the address to ensure consistent field values, then parse the address into its component fields. However, due to the normalisation inconsistencies with Libpostal, experiments in Sect. 4 will show that it is possible to achieve greater levels of matching accuracy by using Libpostal as a parsing agent only and applying a targeted normalisation technique afterwards. As a result of those experiments, the 'postparse' post-processing method applies a number of corrective procedures to a parsed address, as follows.

Searching Libpostal 'suburb' and 'city' Output Fields. During our experiments on the parsing process, it was noted that Libpostal outputs the parsed Australian town/suburb value in either the 'suburb' or 'city' fields, but not in both. The results of experiments in Sect. 4 show that overall parsing accuracy is improved by searching both of these output fields for the PostMatch 'Town' field value, regardless of whether the value is misspelt or not.

Unit/Building Number Deciphering. On testing with addresses featuring the forward-slash unit/building number abbreviation, it was also noted that Libpostal's normalisation process would simply replace the forward-slash with a space, for example, '12/34' (i.e., unit 12 of building/house no. 34) would become '12 34' and this value would be incorrectly assigned to the Libpostal 'house_number' output field. However, in contrast, bypassing Libpostal normalisation for parsing only resulted in the original forward-slash notation value being assigned to the 'house_number' output field. As a result of this behaviour, 'postparse' checks the house-number field for a forward-slash character, splitting the values either side and assigning them to the appropriate fields. In this way, both unit number and house/building number can be identified correctly.

Misspelt Town Field-Parsing Correction. As will also be noted in Sect. 4, Libpostal loses accuracy when the 'town' name in an address is misspelt. Further, if the 'town' name is misspelt and contains two words, such as 'North Strathfield', Libpostal can mis-parse both the town and street fields, adding all but the last word of the 'town' name to the end of the 'street' field value. For example, '123 railway parade clareview heghts' would be parsed with {street = 'railway parade clareview'} and {suburb = 'heghts'}. The postparse post-processor handles this by determining if a known street suffix (e.g. street, parade, drive etc.) is the last word in the 'street' field value. If not, the 'street' field is searched for a known suffix, with all words following the suffix removed and added to the start of the 'suburb' field.

Normalising State Names. It has been observed that Libpostal can randomly alternate between the full Australian state names and their common abbreviations depending on the condition of the address, for example, 'victoria' becomes 'vic' or vice versa. However, in the many examples seen, the state is always correct, even if the value is not consistent between full and short names. Nevertheless, the difference between full and short names technically constitutes different strings and would be seen as 'not matched' by strict edit distance measures. To cater for this behaviour, PostMatch checks for inconsistencies in parsing and reverts any full state names back to their three-letter standardised abbreviations for Australian states and territories.

State Field Value Spelling Correction. Given that Australia only has eight states and territories, PostMatch provides spelling correction using a combination of n-grams and the Damerau-Levenshtein (DL) edit distance [13]. If Libpostal fails to provide a valid 'state' output value, the entire address string is searched as a series of n-grams (n = 1,2,3). The state name that has the lowest edit distance to one of the address n-grams is then chosen. The need to allow for values up to n = 3 in n-grams is to allow for the state names 'australian capital territory' and 'new south wales' as possible address 'state' field values. An example of this is shown in Fig. 3.

3.5 Address Pair Matching

Once the addresses have been parsed and normalised, they are ready to be match-checked. Taking a leaf from Fellegi and Sunter [8], PostMatch employs a combination of Jaro-Winkler edit distance [15] and XGBoost machine-learning [4] to compare addresses in two lists, identifying addresses that appear in both lists as 'matched' or 'maybe matched'. The process involves first taking an address pair to be checked and converting it into an edit difference record. This is done by taking a corresponding field from each address and calculating the edit distance or cost for transitioning from one field value to the other. The edit distance becomes the attribute value for that field pair in the new record. Thus, the eight fields of each address are combined to form eight edit-distance attributes

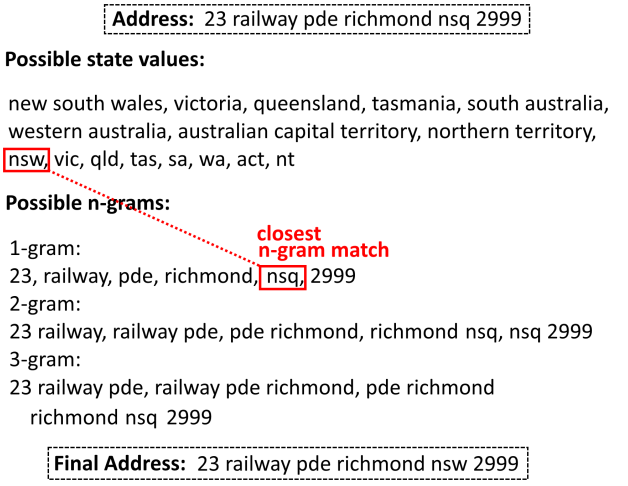

Fig. 3. Using n-grams and DL edit distance to correct state spelling errors.

in the new record. Once all eight address field pairs have been processed and the new record completed, it is tested against a machine-learning model previously learned from a training dataset of known address pair records. The predicted result of the model determines whether the current address pair is matched, not-matched or maybe-matched.

A training dataset containing class-labelled records is required to train the machine-learning model. In practice, this would be obtained through expert domain knowledge or from an existing reference dataset, depending on the application. However, as an existing labelled training dataset was not available during research, Sect. 4 will now describe the experimental phase, including the development of a synthetic training dataset from a public address data source.

4 Experimental Evaluation

4.1 Data and Experimental Environment

A public dataset, the Australian Geocoded-National Address File (G-NAF) [6], is used to evaluate the performance of our proposed PostMatch framework. G-NAF is an Australian government initiative to provide an open database of all physical addresses in all eight Australian states and territories[2]. It contains over 15 million addresses and Table 2 shows its statistics.

All experiments reported in this work were conducted on a desktop PC with a 3GHz Intel quad-core Core i5 CPU and 16GB of RAM, running Xubuntu 16.04.4 operating system and RStudio development environment. The Libpostal library was locally compiled from source code downloaded in March 2019.

[2] G-NAF: https://data.gov.au/data/dataset/19432f89-dc3a-4ef3-b943-5326ef1dbecc.

Table 2. Details of the Australian Geocoded-National Address File (G-NAF) version 201908 (August 2019).

Australian state	Address count	Street count	Town count
New South Wales	4,703,553	180,594	4,716
Victoria	3,838,006	190,472	2,997
Queensland	3,188,961	157,900	3,529
Australian Capital Territory	241,000	8,251	142
Tasmania	344,231	18,952	781
Western Australia	1,532,860	82,245	2,041
South Australia	1,153,802	72,590	2,716
Northern Territory	112,877	8,204	1,096
TOTAL	15,115,290	719,208	18,018

4.2 Experiments on Address Parsing

To begin, a set of complete addresses were derived from G-NAF, totalling a maximum of 200,000 for each state, for a total in excess of 1.5 million. From this, a test dataset of 100,000 addresses were randomly selected. Excess information was removed and each address was formed into a record of eight fields noted in Table 1. After that, eight fields of each address were concatenated into a single string of text. To test PostMatch's parsing ability in the face of addresses with spelling errors, we created a second dataset by injecting spelling errors, e.g., by swapping a vowel in town name.

We conducted a series of seven tests to compare PostMatch with various parsing and normalisation combinations of Libpostal. The tests are:

- Test 1: libpostal parsing. This involved using only Libpostal's parsing routine, with all suburb/town values being selected from Libpostal's 'Suburb' output field only.
- Test 2: libpostal parsing with 'town' search (libpostal-STC). During testing, Libpostal was identified to split parsing of suburbs and towns between its 'Suburb' and 'City' fields. This test is the same as Test 1, except that the final PostMatch 'town' value is selected from either Libpostal's 'Suburb' or 'City' output fields.
- Test 3: libpostal normalisation and parsing. This test first normalises each address using Libpostal's normalisation routine, then parses it with Libpostal. This test most mirrors standard Libpostal usage.
- Test 4: libpostal-State normalisation and parsing (libpostal-STATE). This test is the same as Test 3, but allows state values to be either in the full or short forms. This is to address the issue that, when normalising, Libpostal inconsistently converts some state abbreviations to their full names.
- Test 5: combines Tests 2 and 3
- Test 6: combines Tests 2 and 4

– Test 7: PostMatch (our method). This test runs our PostMatch framework, combining Libpostal parsing with 'postparse' post-processing.

Results on Addresses Without Spelling Errors. The results of the seven tests on the address dataset without spelling errors are shown in Table 3, where our proposed method is of the highest accuracy (highlighted in bold). As previously noted, common usage of Libpostal is to enact its normalisation process on an address first, then apply the results to its parsing engine. Comparing the results of Test 1 and Test 3, it appears that for Australian addresses at least, Libpostal works best when its normalisation process is not used and the addresses are parsed directly. Moreover, the 97.9% accuracy result of Test 2 for Libpostal-STC (searching 'suburb' and 'city' fields for the suburb value) shows that Libpostal is capable of high-accuracy parsing. However, applying Libpostal parsing followed by our 'postparse' post-processing improves the parsing accuracy further to 99.4%, albeit at the cost of 1:20 (i.e., 1 min 20 secs) of extra processing time (3:05 vs 1:45).

Table 3. Experimental results on address data with no spelling errors. The best method is highlighted in bold.

Test no	Parsing	Normalisation	Records matched (out of 100,000)	Percent matched (%)	Process time (mins:secs)
1	Libpostal	–	61,687	61.6	1:42
2	Libpostal-STC	–	97,957	97.9	1:45
3	Libpostal	Libpostal	31,966	31.9	2:08
4	Libpostal-STATE	Libpostal-STATE	47,076	47.0	N/A
5	Libpostal-STC	Libpostal-STC	47,810	47.8	2:09
6	Libpostal-STC-STATE	Libpostal-STC-STATE	72,958	72.9	N/A
7	**Libpostal**	**postparse (our method)**	**99,418**	**99.4**	**3:05**

Results on Addresses with Spelling Errors. The results of the seven tests on address data with spelling errors are shown in Table 4. These accuracy levels are well down compared with those shown in Table 3. Moreover, introducing the Libpostal normalisation before parsing mis-spelt addresses reduces parsing accuracy further still, with all results well below those of the correctly-spelt addresses tested in Table 3 and at best, reaching only 56.1%. The most successful technique tested is PostMatch, our combination of Libpostal parsing and 'postparse' post-processing, where parsing accuracy was largely maintained at just under 97% (Table 4, Test 7), a fall of less than 3% compared with the correctly-spelt result of Table 3. This is in comparison to the 'Libpostal-STC' results, where the fall

Table 4. Experimental results on address data with spelling errors. The best method is highlighted in bold.

Test no	Parsing	Normalisation	Records matched (out of 100,000)	Percent matched (%)	Process time (mins:secs)
1	Libpostal	–	13632	13.6	1:50
2	Libpostal-STC	–	74477	74.4	1:52
3	Libpostal	Libpostal	11388	11.3	2:19
4	Libpostal-STATE	Libpostal-STATE	12487	12.4	N/A
5	Libpostal-STC	Libpostal-STC	38303	38.3	2:19
6	Libpostal-STC-STATE	Libpostal-STC-STATE	56126	56.1	N/A
7	**Libpostal**	**postparse (our method)**	**96920**	**96.9**	**3:30**

Table 5. Experimental results on address data with field swaps. The best method is highlighted in bold.

Test no	Parsing	Normalisation	Records matched (out of 100,000)	Percent matched (%)	Process time (mins:secs)
1	Libpostal-STC	–	59030	59.0	1:49
2	Libpostal-STC	Libpostal-STC	26793	26.7	2:16
3	Libpostal-STC-STATE	Libpostal-STC-STATE	40896	40.8	N/A
4	**Libpostal**	**postparse (our method)**	**59833**	**59.8**	**6:02**

in accuracy from correctly-spelt (Table 3, 97.9%) to misspelt (Table 4, 74.4%) is in excess of 23%. Thus, the small fall of PostMatch from 99.4% to 96.9% shows the efficacy of this approach with addresses containing spelling errors.

Results on Addresses with Field Swaps. To test the robustness of different methods, we randomly changed the order of some fields within addresses and conducted another experiment. An example of field order change is to push the 'Town' field to the end of the string and move the 'State' and 'PCode' fields closer to the front of the string. The results are shown in Table 5, which suggests that in its current form, Libpostal (together with our postparse post-processing) is only able to correctly parse six out of ten Australian addresses (59.8%) at best. Moreover, PostMatch (Test 4) shows only a minor improvement in parsing accuracy, whilst processing time has increased significantly (6:02 vs 3:30 in Test 7, Table 4) as a result of the change in field order.

4.3 Experiments on Address Matching

Following the same approach used in Sect. 4.2, we created a dataset containing 200,000 address pairs for address matching evaluation. The address pairs are of three classes:

- Class '0': unmatched address pairs,
- Class '1': matched address pairs, and
- Class '2': matched address pairs with some fields changed using the same perturbation method used in Sect. 4.2, to test the model's ability of dealing with misspellings and field misalignment.

Table 6. Experimental results on address matching.

	Actual		
Predicted	Class = 0	Class = 1	Class = 2
Class = 0	33361	0	6
Class = 1	0	33300	0
Class = 2	0	55	33278
Sensitivity	1.0	0.9984	0.9998
Specifivity	0.9999	1.0	0.9992
Balanced accuracy	1.0	0.9992	0.9995

Each class covers 1/3 of the data. We split the above data into training and test datasets in a 50:50 ratio.

With our PostMatch framework (see Fig. 1), each record of above data was first parsed with Libpostal and processed with our postparse processing method. After that, the Jaro-Winkler (JW) algorithm [15] was used to calculate the edit distance between the corresponding fields of each address pair. Finally, the edit distances of the eight address fields, together with the class label, were fed into a XGBoost [4] model for prediction.

Results of Address Matching. The results of address matching are shown in Table 6. The speed of the XGBoost algorithm allowed the 100,000 record-pairs to be processed in 12.91 ms. This would enable one million record-pairs to be processed every 130 ms, or 0.13 µs per record-pair. However, this is one million address comparisons required to compare two normalised addresses containing only 1,000 addresses each. Thus, since the computational load increases by $O(n^2)$, where n is the number of addresses per list, this task requires reasonable levels of computing power to execute locally. Still, the XGBoost matching component of this framework would be able to compare two normalised address lists, each containing 100,000 records, in less than 22 min on our test quad-core computer system.

5 Conclusions

This paper introduced the PostMatch framework for efficient address matching using a combination of the Libpostal open-source address parsing software and Jaro-Winkler/XGBoost algorithms. We have identified the issues of Libpostal and introduced a 'postparse' post-processing routine to improve it. Experimental results show that the combination of Libpostal and 'postparse' routines can achieve parsing accuracy of 99.4% with correctly-spelt addresses and even 96.9% with addresses featuring misspelt components. Our PostMatch framework built with a Jaro-Winkler similarity measure and an XGBoost model achieved near-100% matching accuracy. Nevertheless, we must also outline the areas for improvement, including greater understanding of Libpostal's ability to parse addresses with varying field order, as well as addresses with missing values.

The PostMatch framework is generic, although the proposed method was evaluated with Australia address data. We plan to confirm that our generic method can be adapted and applied to addresses from other counties or for other languages in future work.

References

1. Libpostal: international street address NLP, March 2019. https://github.com/openvenues/libpostal
2. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 918–929. VLDB Endowment (2006)
3. Bell, G.B., Sethi, A.: Matching records in a national medical patient index. Association for Computing Machinery. Commun. ACM **44**(9), 83 (2001)
4. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016)
5. Christen, P.: A comparison of personal name matching: techniques and practical issues. In: Sixth IEEE International Conference on Data Mining-Workshops (ICDMW 2006), pp. 290–294. IEEE (2006)
6. Christen, P., Belacic, D.: Automated probabilistic address standardisation and verification. In: Australasian Data Mining Conference (2005)
7. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string metrics for matching names and records. In: KDD Workshop on Data Cleaning and Object Consolidation, vol. 3, pp. 73–78 (2003)
8. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. J. Am. Stat. Assoc. **64**(328), 1183–1210 (1969)
9. Hand, D., Christen, P.: A note on using the f-measure for evaluating record linkage algorithms. Stat. Comput. **28**(3), 539–547 (2018)
10. Koudas, N., Sarawagi, S., Srivastava, D.: Record linkage: similarity measures and algorithms. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 802–803. ACM (2006)
11. Koumarelas, I., Kroschk, A., Mosley, C., Naumann, F.: Experience: enhancing address matching with geocoding and similarity measure selection. J. Data Inf. Qual. (JDIQ) **10**(2), 8 (2018)

12. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics Doklady, vol. 10, pp. 707–710 (1966)
13. Van der Loo, M.P.: The stringdist package for approximate string matching. R J. **6**(1), 111–122 (2014)
14. Vatsalan, D., Sehili, Z., Christen, P., Rahm, E.: Privacy-preserving record linkage for big data: current approaches and research challenges. In: Zomaya, A.Y., Sakr, S. (eds.) Handbook of Big Data Technologies, pp. 851–895. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-49340-4_25
15. Winkler, W.E.: String comparator metrics and enhanced decision rules in the Fellegi-Sunter model of record linkage (1990)
16. Winkler, W.E.: Matching and record linkage. Bus. Survey Meth. **1**, 355–384 (1995)