Integrating approximate string matching with phonetic string similarity

Junior Ferri[1], Hegler Tissot[1], and Marcos Didonet Del Fabro[1]

C3SL labs, Universidade Federal do Paraná, Curitiba, Brazil
`junior.ferri@ufpr.br, hegler@gmail.com, marcos.ddf@inf.ufpr.br`

Abstract. Well-defined dictionaries of tagged entities are used in many tasks to identify entities where the scope is limited and there is no need to use machine learning. One common solution is to encode the input dictionary into Trie trees to find matches on an input text. However, the size of the dictionary and the presence of spelling errors on the input tokens have a negative influence on such solutions. We present an approach that transforms the dictionary and each input token into a compact well-known phonetic representation. The resulting dictionary is encoded in a Trie that is about 72 percent smaller than a non-phonetic Trie. We perform inexact matching over this representation to filter a set of initial results. Lastly, we apply a second similarity measure to filter the best result to annotate a given entity. The experiments showed that it achieved good F1 results. The solution was developed as an entity recognition plug-in for GATE, a well-known information extraction framework.

Keywords: Entity Recognition, Metaphone, Text Tagging, Trie, Active Nodes, Fast Similarity Search

1 Introduction

An Information Extraction (IE) pipeline is composed by tasks aiming at extracting information from unstructured sources and making it available in specific and structured formats [1,12]. The Named Entity Recognition (NER) task aims at finding and classifying specific entities within a text, such as organizations, cities or drug names [4,14].

Several approaches use well-defined dictionaries as input for NER tasks [16]. The dictionaries contain lists of classified entities. They are appropriate solutions either when the scope is limited and when there is no need to use machine learning techniques, or when they could be used as input for a machine learning solution. Existing solutions often use indexing structures such as suffix trees and arrays, q-grams or q-samples. However, it is not enough to use only index-like structures to support string exact matching algorithms. It is necessary to take into account the existence of spelling/typing errors in the input text, thus supporting Approximate String Matching [10]. The size of the dictionary may also be an issue, specially if there are several dictionaries used by the same task.

Approximate String Matching (ASM) has been widely studied in different contexts, including Information Extraction (IE) and in the Named Entity Resolution (NER) task. One of the central studies was the survey by Navarro et. al [10]. In this paper, however, we do not intend to be exhaustive in this context, but to narrow the scope with recent approaches that provide Trie-based solutions for Approximate String Matching.

Recent studies coupling Tries with inexact matching are the works from [5] and [7]. They introduce the notion of valid (also called active) nodes while searching through the tree nodes. The active nodes have a calculated Edit Distance (ED) value which is lesser or equal than a max allowed ED value. They experimentally showed that these approaches were superior than q-Grams based solutions. In [8], the authors improve the active nodes computation, by incrementally computing them and storing them in a cache, creating the ICAN/ICPAN algorithms. The IncNGTrie [17] algorithm maintains a smaller set of active nodes, improving the performance. The META approach [3] presents a solution based on matching over compact indexes and that supports top-k queries. The most recent work from [15] handles approximate matching using efficient Trie implementations, though in the context of abbreviations, not full words. However, these approaches, or similar solutions, have not experimented using phonetic information from the text and dictionary, such as applying conversion using the Soundex [18] or Metaphone [11] algorithms. There are approaches that combine different similarity measures into a super-metric [6], though using distinct solutions than the Trie-based encoding, which has been shown to be effective.

In this paper we present an approach that couples a phonetic conversion algorithm with a Trie-based encoding in a NER task. First, we transform the given dictionary into a phonetic representation using the Metaphone algorithm using a well-known API. The phonetic representation is encoded in a Trie, and the Trie approximate matching is developed based on the active nodes algorithm from [5]. Each input token is also converted and than matched. The main advantage of this encoding is the reduced size, which may be important when using several dictionaries. It produces a Trie around 72 per cent smaller than a Trie with the complete strings. In order to avoid a low precision, since the representation is smaller, we apply a second string similarity metric to return the best result, this time over the original string, linked by the Trie structure.

We have executed a set of experiments showing the applicability of this two-phase matching solution. The approach is implemented as a Gazeteer plug-in[1] for the GATE suite [2], a known information extraction framework. The implementation enables setting execution parameters, including the string metrics.

This paper is organized as follows. Section 2 presents our solution integrating inexact matching and a Trie-based ASM solution. Section 3 shows the experiments. Section 4 has the final conclusions.

[1] `http://gitlab.c3sl.ufpr.br/faes/asm/tree/master`

2 Inexact matching and phonetic encoding in a NER task

In the following sections we describe the tree major steps on how we integrate inexact matching with phonetic encoding.

2.1 Input tokens extraction

We define the NER task as the following, adapting the definition from [14] to add the input dictionary. Given a document D, composed by a sequence of tokens $\mathbf{T}=\{t_1, ..., t_n\}$, the NER task extracts from D a set of fields $\mathbf{F} = \{f_1, ..., f_k\}$, where each field is an attribute-value pair $f_i = \langle a, v\rangle$. The value v is a token or a set of tokens matched with a key k, which has a corresponding label value lv, both available in entries from an input dictionary $ID = \{\langle k_1, lv_1\rangle, ..., \langle k_j, lv_j\rangle\}$. For instance, we could have a field $f_i = \langle City, London\rangle$, where the *City* label is extracted from a dictionary after matching an input token with the *London* key. In other words, it produces a list with annotated input tokens with labels from the input dictionary.

A token is a finite sequence of characters representing a subset of another finite sequence of characters, both conforming to the same alphabet. A token is extracted through the definition of delimiters d_{start}, d_{stop} to identify its start and end positions within a given sequence. Given a sequence of characters S, with size $|S|$, the size of each *token* t from T, is $0 < |t| \leq |S|$ and the sum of all tokens size is $\sum |T_{1..n}| \leq |S|$. The tokenizing rules depend on the tokenizer chosen. In our case study, we will use an existing tokenizer which is a plugin from the GATE framework.

Before the matching process, each individual token and the keys from the input dictionary are converted into a phonetic representation. We apply a phonetic conversion function:

$$CF(x : String) : String = t_{ph} \tag{1}$$

where t_{ph} is a new phonetic token. In a large part of dictionary-based NER solutions, there is no CF function, so they perform exact or approximate matching. In our solution, the matching process is done using only the newly produced phonetic tokens. The phonetic conversion function could be existing phonetic conversion algorithms, such as the Metaphone algorithm, which provides a compact phonetic representation for the English language. Table 1 presents some examples on how the Metaphone algorithm phonetically represents English words. We included samples in two common utilization of Gazeteers: cities and medication names.

2.2 Phonetic Approximate String Matching

The matching process to find a named entity considers only full tokens or set of full tokens. For example, if we have an input key *Brazil* and the current input

Table 1. Examples of phonetic representations resulting from Metaphone.

Word	Phonetic representation
medroxalol	MTRKSLL
amoxicillin	AMKSSLN
bromfenac	BRMFNK
New York	NYRK
Avondale Estates	AFNTLSTTS
Washington	WXNKTN

token is *Brazilian*, the matching process does not consider this dictionary entry as a valid matching, if it is not within an Edit Distace (ED) limit.

We implement an algorithm that uses a Trie tree to encode the dictionary and to perform the inexact matching with the input tokens, using the phonetic converted versions of the tokens and the dictionary. Our algorithm implements the idea from [5], having a set of active nodes while searching the tree nodes, which have an Edit Distance lesser than a given threshold value. Briefly, the algorithm checks whether each character from an input token matches with a given Trie node and its descendents. If it does not match, the node ED is increased by 1. This search process stops when a node has the ED value larger then the given limit, when it is then deactivated. This need to be done for each previously activated node. The core algorithm is the same, with two main differences. First, the data structure has additional information to support NER matching, containing the following fields:

- **character**: the current character representing the node;
- **child**: an array containing the child nodes;
- **entryEnd**: defines if a given node identifies the end of a dictionary entry. It is used for composed entries;
- **activeNode**: identifies if the node is an active node, i.e., its assigned ED value is within a given threshold;
- **currentEd**: the current ED value;
- **entries**: stores the dictionary entries for a given node. One node may contain more than one dictionary entry. In this case the input token is annotated more than one time;
- **dictionaryEntry**: it contains the complete entry and the label used to annotate the input tokens.

As second difference, we add an extra step to allow the matching with entities composed with more than one token, for instance, a composite city name such as *New York*. This means that the matching is not performed only token by token, but after an unsuccessful token match, the algorithm continues to verify if it could match as a composed token. The result of the matching process is a list of candidate entities to be annotated. Utilizing compact phonetic structures diminishes the size of the Trie, however, it has the drawback of increasing the number of matchings. For this reason, it is necessary to add a filtering step, which is explained in the following section.

2.3 Filtering the results

The results from the matching phase are filtered by applying the following similarity function:

$$Sim(t : String, e : String) : Float = s \tag{2}$$

where t is the input token, e is the dictionary entry and s is a similarity value between both parameters, with $0 < s \leq 1$. However, to avoid any information loss, the parameters t and e are the original input token and entry, not the phonetic representation used along the previous step. The approach does not define a new similarity function, but it uses existing ones, such as the Jaro-Winkler or $String_{Sim}$ [13] metrics.

The result of the similarity function is used in two filtering rules. First, we set up a similarity threshold, called *minimum similarity*, to be considered (e.g., 0.7). Second, we rank the remaining entries and we choose only the best result.

The result of the filtering phase is a list with the annotated entities. Each entity contains the start and end positions in the input text, so it can be integrated with NER frameworks, the annotation labels associated and the final ED value.

3 Experiments

We have implemented a plug-in for the GATE suite to evaluate our approach, which is freely available for download,[2] as well as the dictionary, the input text and the complete raw results. In addition to the matching algorithm, we took special attention on providing a fully configurable plug-in, which means several configuration parameters can be easily modified. The main parameters are the following: *maxEditDistance, minSimilarityAccepted, similarityClass/Method, conversionClass/Method* (their names are self explanatory). We also provide parameters for setting up the tokenizer and the format of the dictionary, though we do not detail them here.

The input dictionary is a list with 76,912 English words obtained from the WordNet [9] database. The input entities with errors are randomly selected from a collection of common misspellings from Wikipedia.[3] It contains the correct word and a version with a misspelling error. We used 1,000 input words, all with at least one misspelling error.

We applied our approach using two main settings. First, we used the Metaphone[4] conversion function prior to the matching. Second, we used the original tokens. We used the Jaro-Winkler metric for filtering the results.[5] We have used

[2] `https://gitlab.c3sl.ufpr.br/faes/asm/tree/master`

[3] `https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings`

[4] Implementation from the commons-codec-1.10.jar library, available at `https://commons.apache.org/proper/commons-codec/download_codec.cgi`

[5] Implementation from lucene-suggest-5.2.1.jar, at `http://lucene.apache.org/`

the $String_{Sim}$ metric as second similarity metric, which keeps the similarity values higher for words with lower differences.

We used three different Edit Distance values: 0, 1 e 2. The 0 value was used to evaluate the effect of the conversion function on already eliminating errors. The limit of 2 was chosen because a higher value would produce a too low recall. For each ED value, we used 3 minimum similarities: 0.7, 0.8 and 0.9. Values smaller than 0.7 returned too many results. We evaluated the results in terms of: a) *precision* - the fraction of the relevant annotations among all the returned ones; b) *recall* - the fraction of relevant annotations among the total; and c) the *F1 measure* - the harmonic average between both. We used the list with the correct words to verify these results.

3.1 Trie construction

Table 2 shows a comparison between the size of the produced Trie using the original text (no phonetic conversion) and using the Metaphone function.

Table 2. Trie size for 76,912 entries

	Original text	Metaphone
Input chars	658.774	372.226
Avg. entries size	8,6	4,8
Trie nodes	240.484	67.602

The utilization of the Metaphone conversion on the list of 76,912 entries diminished the number of the input characters by 43%, from 658,774 to 372,226. After encoding the Trie, the number of nodes was 72% smaller, from 240,484 to 67,602. We do not measure the performance results on the Trie creation, since it is constructed only once. However, the reduced size yields loss of information. The impact of such codification is explained in the next section.

3.2 Phonetic approximate matching

The resulting scores are presented in Tables 3(a) to 4(b). First, we present the results when using the Metaphone conversion. Second, the results without a conversion function.

The choice of good parameters values is important to achieve good results. For instance, choosing a low ED value may minimize the choice of higher similarity thresholds, since it affects a lesser number of entries. Combined variations of ED and minimum similarity results yield results with a large variation. For instance, the recall values vary from 64.2% to 89.4%. The experiments results may be used to guide on the choice of the metrics and this threshold.

Table 3. Metaphone conversion

(a) Jaro-Winkler

ED	Min Sim	Precision	Recall	F1
0	0.7	81.3%	64.2%	71.7%
0	0.8	84.4%	64.2%	72.9%
0	0.9	**87.8%**	62.7%	73.2%
1	0.7	81.5%	**84.4%**	**82.9%**
1	0.8	81.5%	**84.4%**	**82.9%**
1	0.9	82.6%	82.8%	82.7%
2	0.7	78.3%	82.3%	80.3%
2	0.8	78.7%	82.3%	80.4%
2	0.9	79.7%	81.5%	80.6%

(b) $String_{Sim}$

ED	Min Sim	Precision	Recall	F1
0	0.7	86.7%	66.5%	75.3%
0	0.8	90.7%	66.3%	76.6%
0	0.9	**94.9%**	51.9%	67.1%
1	0.7	85.7%	**89.5%**	87.6%
1	0.8	86.4%	89.3%	**87.8%**
1	0.9	89.3%	72.9%	80.3%
2	0.7	84.0%	88.7%	86.3%
2	0.8	84.0%	88.5%	86.2%
2	0.9	87.3%	73.0%	79.5%

We can see that the highest precision (87.8%) from Table 3(a) was obtained with the ED value equals to 0. This means that the choice of this phonetic representation also had an impact and it absorbed partially the misspelling errors, since every word had at least 1 error. However, the recall was smaller, 62.7%, obtaining an F1 score of 73.2%. Increasing the ED by 1 had a positive impact on the recall, since we had 84.4% of recall. This combination yielded the best F1 score, 82.9%. The precision diminished on about 2% when augmenting the ED. In addition, the execution time of the search in the tree started on around 15 ms, raising gradually for higher EDs.

Without the phonetic conversion, as shown in Table 4(a), the best precision was obtained with $ED \leq 1$ (86.6%), with the same result with 3 filtering similarity metrics. The best precision was obtained with $ED \leq 2$. The best recall was obtained with a similar configuration than with the conversion. Finally, the best F1 results were obtained with $ED \leq 2$ and with filtering similarity ≥ 0.7 and ≥ 0.8. A higher filtering value yielded a too low precision. We do not consider ED equals to 0, since the result is an exact matching. The execution times have a similar order of magnitude, independently from using the Metaphone or without conversion. This is an interesting result since we could argue that the phonetic conversion could be chosen without additional time cost.

The results from $String_{Sim}$ were obtained with a very similar choice of parameters, for both cases. However, the filtering metric yielded a better precision in both cases, 87.8% with conversion and 88.2% without conversion. The performance results were slightly worse, though not very significant.

We have also conducted experiments using the Double Metaphone conversion. This method has an even more compact representation: the Trie size is about 95% smaller, with 10,325 nodes, since in this representation the tokens have only 4 characters. However, it increases too much the number of the matched entries, thus the filtering metrics need to be executed over too many entries. For this reason, we discarded such approach.

Table 4. No phonetic conversion

(a) Jaro-Winkler

ED	Min Sim	Precision	Recall	F1
1	0.7	**86.6%**	72.0%	78.6%
1	0.8	**86.6%**	72.0%	78.6%
1	0.9	**86.6%**	70.7%	77.9%
2	0.7	82.6%	**84.4%**	**83.5%**
2	0.8	82.6%	**84.4%**	**83.5%**
2	0.9	82.6%	82.8%	82.7%

(b) $String_{Sim}$

ED	Min Sim	Precision	Recall	F1
1	0.7	89.3%	74.3%	81.1%
1	0.8	89.5%	74.3%	81.2%
1	0.9	**91.8%**	65.3%	76.3%
2	0.7	87.0%	**89.4%**	**88.2%**
2	0.8	86.9%	89.2%	88.1%
2	0.9	88.8%	73.1%	80.2%

To summarize, we can see from these tables that the parameters with better F1 were obtained with ED equals to 1 and 2, and with a small variation in the filtering similarity threshold. Comparing the three methods, the best F1 scores were better with the $String_{Sim}$ metric. This indicates that a phonetic approach with filtering is a valid solution when choosing the right parameters and filtering function. The performance results were similiar from both metrics and conversion methods. Considering the phonetic conversion, the increase on precision was not very high and the performance results were similar. However, the smaller size of the Trie justifies the choice of a conversion method. These findings shows that the initial matching on the phonetic version acts like an initial filter, in order to reduce the size of the entries.

4 Conclusions

We presented an hybrid approach that integrates approximate string matching with phonetic string similarity. We have implemented a version of an existing algorithm to encode a Trie and to apply inexact string matching. We changed the input of the algorithm by using a compact phonetic representation, using the Metaphone algorithm. The size of the Trie was 72% smaller, thus having a large impact on the Trie size. The Trie inexact phonetic matching acts like an initial filter of similar words, and a second similarity metric, applied on the original text, does the final filtering. The best F1 scores were achieved with the $String_{Sim}$ metric. This means it can be used when the size of the Trie is important (for instance, when there are several dictionaries used on a desktop application). We have seen that the choice of the right parameters is also important, otherwise the F1 scores may decrease. The phonetic conversion did not have a significant impact on performance, making it a good choice.

We have implemented the approach as a Gazeteer plug-in for GATE, a well-known information retrieval framework, with setup parameters that can be modified, including the metrics presented. This flexibility can be used to conduct further experiments with different settings, from the initial tokenizer, the execution parameters, up to the phonetic algorithm and the similarity measure, in order to obtain better final F1 scores in a future work. All the results and the plug-in are freely available.

Acknowledgments

This work was partially funded by Project *Sistema de Monitoramento de Políticas de Promoção da Igualdade Racial (SNPPIR).*

References

1. H. Cunningham. Information Extraction, Automatic. *Encyclopedia of Language and Linguistics, 2nd Edition*, 2005.
2. Hamish Cunningham, Valentin Tablan, Angus Roberts, and Kalina Bontcheva. Getting more out of biomedical documents with gate's full lifecycle open source text analytics. *PLOS Computational Biology*, 2013.
3. D. Deng, G. Li, H. Wen, H. V. Jagadish, and J. Feng. Meta: An efficient matching-based method for error-tolerant autocompletion. *Proc. VLDB Endow.*, 9(10):828–839, June 2016.
4. R. Grishman and B. Sundheim. Message understanding conference-6: A brief history. In *COLING*, volume 96, pages 466–471, 1996.
5. S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th WWW*, WWW '09, pages 371–380, Madrid, Spain, 2009. ACM.
6. L. Lamontagne and I. Abi-Zeid. *Advances in Case-Based Reasoning: 8th ECCBR 2006 Fethiye, Turkey, September 4-7*, chapter Combining Multiple Similarity Metrics Using a Multicriteria Approach, pages 415–428. Springer, 2006.
7. G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: A tastier approach. In *Proceedings of the 2009 ACM SIGMOD*, SIGMOD '09, pages 695–706, Providence, Rhode Island, USA, 2009. ACM.
8. G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *The VLDB Journal*, 20(4):617–640, August 2011.
9. George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.
10. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.
11. L. Philips. Hanging on the metaphone. *Computer Language Magazine*, 7(12):38–44, 1990.
12. S. Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, March 2008.
13. H. Tissot, G. Peschl, and M. Didonet Del Fabro. Fast phonetic similarity search over large repositories. In *25th , DEXA 2014, Munich, Germany, September 1-4*, pages 74–81. Springer, 2014.
14. A. Culotta; T. Kristjansson; A. McCallum; P. Viola. Corrective feedback and persistent learning for information extraction. *Artificial Intelligence*, 170(14):1101–1122, 2006.
15. M. Stonebraker W. Tao, D. Deng. Approximate string joins with abbreviations. *Proc. VLDB Endow.*, 11(1).
16. D. C. Wimalasuriya and D. Dou. Ontology-based information extraction: An introduction and a survey of current approaches. *Jour. of Information Science*, 2010.

17. C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *VLDB Endow.*, 6(6):373–384, April 2013.
18. J. Zobel and P. Dart. Phonetic string matching: Lessons from information retrieval. In *the 19th SIGIR*, SIGIR '96, pages 166–172, Zurich, Switzerland, 1996. ACM.