Fast Phonetic Similarity Search over Large Repositories

Hegler Tissot, Gabriel Peschl, and Marcos Didonet Del Fabro

Federal University of Parana, C3SL Labs, Curitiba, Brazil
{hctissot,gpeschl,marcos.ddf}@inf.ufpr.br

Abstract. Today there is a large amount of unstructured data produced by information systems from different domains. These sources may be analyzed for different purposes. Existing approaches use string similarity methods to search for valid words within a text, with a supporting dictionary. However, they have two main drawbacks. First, they are not rich enough to encode phonetic information to assist the search. Second, the solutions may be inefficient in the presence of spelling errors. In this paper, we present a novel approach for efficiently perform phonetic similarity search over large data sources. We present a data structure called *PhoneticMap*, which encodes language-specific phonetic information. The phonetic maps are used by a novel fast similarity search algorithm to find words with spelling errors. We validate our approach through an experiment over a data set using a Portuguese variant of a well-known repository, to automatically correct words with spelling errors.

Keywords: Phonetic Similarity, String Similarity, Fast Search

1 Introduction

Today there is a large amount of unstructured data being produced by different kinds of information systems, in a variety of formats, due to the advancement of communication and information technologies [12, 13]. One important kind of unstructured data is free text of a particular domain, for instance, records from the medical or avionic industries.

The extraction of information from these data sources is often performed using string similarity comparison algorithms to identify concepts from the free text [8] when text is loaded with misspellings. String similarity metrics can measure similarity between two text strings. Edit Distance (ED) [14] and Jaro-Winkler (JW) distance [25] are two well known functions found in literature. These algorithms can be used to compare the elements from the input data source with an existing dictionary to identify a possible valid word for a mispelling.

One of the most commonly used digital dictionaries is the Princeton WordNet (PWN). PWN is a lexical database of the English language that provides a more intuitive combination of dictionary and thesaurus, to support automatic text analysis coupled with artificial intelligence applications [15]. However, PWN

should be modified in the following aspects: (a) PWN contains a wide range of common words and it was designed to be an underlying database for different applications, not to cover domain-specific vocabulary; (b) PWN does not include information about derivative words and the forms of irregular verbs; this problem is even greater when considering the variation of verb conjugation in different tenses, e.g., 67 variations for each verb in Brazilian Portuguese [6]; (c) PWN does not offer a repository structure to support string similarity search.

The existing string similarity algorithms coupled with a supporting dictionary may be very inefficient, in particular when the analyzed text has spelling errors [16], because they do not necessarily handle specific application aspects related to spelling errors. In these cases, it is necessary to use phonetic similarity metrics. Phonetics are language-dependent [17] and solutions for this sort of problems must be specially designed for each specific language.

In addition, similarity algorithms are often slow when executed over large databases, although fast search algorithms have been implemented. Fast Similarity Search (FastSS) [2] is an algorithm based on Edit distance (ED) designed to find strings similarities in a large database. It finds similar words from an input word with misspelling errors. However, the results are based only in the ED metric, and it does not consider phonetic similarity.

In this paper, we present an approach of fast phonetic similarity search (FPSS) over large repositories, coupled with a dictionary. Our solution has three main contributions. First, we present an indexed data structure called *PhoneticMap*, which is used by our novel fast similarity search algorithm. Second, we define a string similarity method that keeps the similarity higher for words with low differences. In contrast, it adds the notion of penalty, in a way the similarity value drops faster when the words have several differences. Finally, we integrate the previous contributions with PWN to implement the fast phonetic search. We validate our approach through an experiment using an extended version of PWN for the Brazilian Portuguese language, over a large repository, in which we try to find correct words given words that have errors, to promote an automatic correction of spelling errors.

This article is organized as follows: Section 2 proposes a string similarity function, a phonetic similarity function, and a method for searching for phonetic similarities over PWN-based repositories; Section 3 describes the experiment where the proposed methods was applied and shows the results obtained using words from Portuguese language; Section 4 refers to the related work and Section 5 concludes with final considerations and future work.

2 Fast Phonetic Similarity Search

String similarity search methods are essential to be able to extract words from repositories with spelling errors. We conducted a simple experiment over 126,812 words from a set of 4,748 medical records, which were submitted to an exact match over a dictionary. Only 40,212 words (31,71%) were found. This small value of exact matches took place due to spelling errors, and the use of abbre-

viations and acronyms. Efficient inexact methods are important to be able to improve such results.

However, when dealing with a large repositories, it is also required to support a fast similarity search, i.e., for a given possible not well-written word, we want to find phonetically similar words, but not performing a full search in the repository.

In this section we present our approach to perform fast similarity search using phonetic information. First, we present a novel string similarity function. Then, we describe a phonetic similarity function using $PhoneticMaps$, and a PWN repository extension to support finding phonetically similar words.

2.1 String Similarity

We present a novel algorithm to calculate string similarity. The $String_{sim}$ function illustrated in Figure 1 measures similarity based on the percentage of characters of one word that can be found in the other one, also considering the position of matching characters and the difference in string sizes.

```
in:  w1 (String), w2 (String)
out: similarity (Number)
       0 = completely different
       1 = exactly equal
 1: g1 ← CharsFound(w1, w2);
 2: g2 ← CharsFound(w2, w1);
 3: Ω ← 0.975;
 4: p1 ← Ω^PositionPenalty(w1,w2);
 5: p2 ← Ω^PositionPenalty(w2,w1);
 6: similarity ← avg(g1 × p1, g2 × p2);
 7: ϒ ← 0.005;
 8: S_MAX ← MAX(length(w1), length(w2));
 9: s_min ← min(length(w1), length(w2));
10: if (S_MAX > s_min) then
11:    b ← 1 + (S_MAX − s_min) × ϒ;
12:    f ← ln(S_MAX − s_min + 1);
13:    c ← (S_MAX − s_min) / 2;
14:    similarity ← similarity × (1 / (b^f)^c);
15: end if;
16: return similarity;
```

Fig. 1. String_{sim}: A proposed string similarity function pseudocode

$String_{sim}$ function calculates the average between the percentage of w_1 characters found in w_2 and the percentage of w_2 characters found in w_1 (lines 1–6). *CharsFound* return the number of characters of first parameter found in the second one, not taking into account the position in which the characters are

found. For each character found, but not in the same string position, a reduction penalty is calculated based on the constant Ω (lines 3–5). *PositionPenalty* returns the number of characters of first parameter found in the second one but not in the same string position. Penalty calculated based on Ω guarantees, for example, that strings “ba” and “baba” will NOT result a $similarity = 100\%$.

When the lengths of both strings (S_{MAX} and s_{min}) are different, there is a result adjustment in order to provide another penalty in the similarity level, based on the difference on the length of words and the factor Υ (lines 7–15).

Ω (=0.975) and Υ (=0.005) were empirically defined after testing the proposed function in an application that searches for similar names of people and companies. Ω and Υ were manually adjusted based on a list of known pairs of names that should or should not be considered similar in the result of each search.

We compared $String_{sim}$ against normalized versions of Edit Distance (by Levenshtein [14]) and Jaro-Winkler distance [3]. In Figure 2 we illustrate how each function decreases the similarity as the words become more different from each other (some similarity results are detailed in Table 1). In this case we used only a combination of letters, for didactic purposes. $String_{sim}$ keeps the similarity higher as there are more characters in common between strings with small differences in size. Comparing strings “ab” and “abab”, Edit Distance results in a similarity of 50% while $String_{sim}$ results in 96.9%. As the difference in length of strings becomes larger, $String_{sim}$ tends to reduce the similarity more sharply, getting close to 0% faster. $String_{sim}$ has also the ability to distinguish different characters when comparing strings. For example, Edit Distance results the same similarity (50%) when comparing (“ab” × “abab”) or (“ab” × “abcd”). The same is not true when using $String_{sim}$: $String_{sim}(\text{“}ab\text{”}, \text{“}abab\text{”}) = 96.9\%$ and $String_{sim}(\text{“}ab\text{”}, \text{“}abcd\text{”}) = 74.2\%$.

Table 1. Comparing similarity functions

#	Word$_1$	Word$_2$	Levenshtein	Jaro-Winkler	String$_{sim}$
1	ab	ab	100.0%	100.0%	100.0%
2	ab	abab	50.0%	86,0%	96.9%
3	ab	ababab	33.3%	82.0%	91.0%
...	...	...	...	...	...
10	ab	ab ... ab (10x)	10.0%	76.0%	9.1%
...	...	...	...	...	...
20	ab	ab ... ab (20x)	5.0%	74.0%	0.0%

= number of repetitions of “ab” in Word$_2$

2.2 Phonetic Similarity

When considering phonemes, a straightforward string comparison of characters may not be enough. In order to support indexing phonemes for a fast search, we

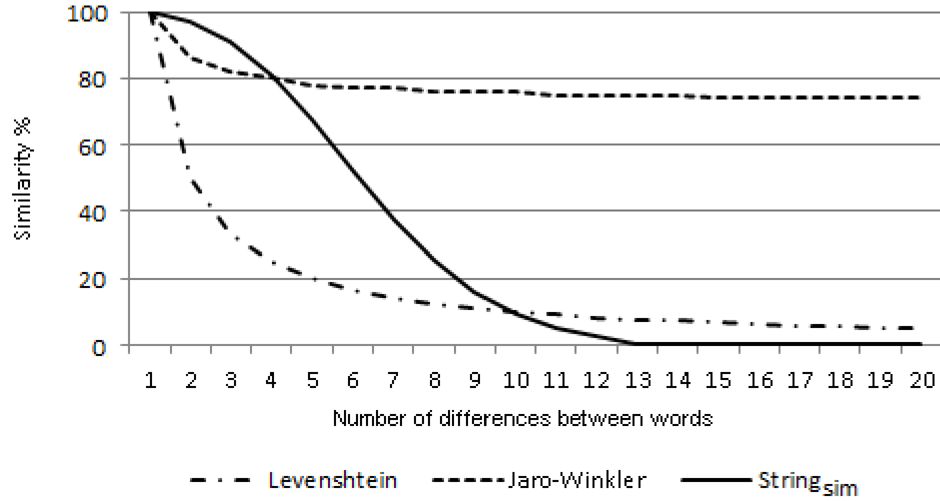

Fig. 2. Similarity functions behaviour

present a structure called *PhoneticMap* and we define the *PhoneticMap Similarity.*

Definition 1 (PhoneticMap). *Given a word (string) w consisting of a sequence of letters (as symbols and digits usually fall outside this scope), the generic function $PhoneticMap(w)$ is a function that results a PhoneticMap tuple $M = (w, P, D)$, where: w is the word itself, $P = \{p_1, p_2, \dots, p_n\}$ is a set of n phonetic variations of word w, and $D = \{d_1, d_2, \dots, d_n\}$ is a set of n definitions, where d_i is the definition of variation p_i.*

Definition 2 (PhoneticMap Similarity). *Given two $PhoneticMaps$ M_1 and M_2, $PhoneticMapSim(M_1, M_2)$ is a generic function that results a similarity value (ranging from 0=different to 1=equal) between $PhoneticMaps$ M_1 and M_2.*

As $PhoneticMap(w)$ and $PhoneticMapSim(M_1, M_2)$ are language-dependent, it is also possible to create more than one instance to each function for different languages.

Brazilian Portuguese consonant phonemes can be classified according to the articulation manner and the articulation point of vocal chords. Articulation manner codes comprise: A for {/m/, /n/, /nh/}; B for {/b/, /k/, /d/, /g (gue)/, /p/, /t/}; C for {/s/, /f/, /j/, /v/, /x/, /z/}; D for {/l/, /lh/}; and E for {/r/, /R/}. Articulation point codes comprise: A for {/m/, /p/, /b/}; B for {/f/, /v/}; C for {/t/, /d/}; D for {/l/, /lh/, /z/, /s/, /n/, /nh/, /r/}; E for {/j/, /x/}; and F for {/k/, /R/, /g (gue)/}. Such codes were used to generate phonetic variations 7–10 based on variation 5 (Table 2).

We develop two variations for the *PhoneticMap* and *PhoneticMap Similarity* functions to support the Brazilian Portuguese: $PhoneticMap_{PT}(w)$ and $PhoneticMapSim_{PT}(M_1, M_2)$.

The function $PhoneticMap_{PT}(w)$ returns a map of 11 entries. Table 2 describes each entry and shows an example generated for a Brazilian Portuguese word. This structure can be adapted for different languages.

Table 2. A result sample for the $PhoneticMap_{PT}$ function

$PhoneticMap_{PT}$("arrematação")

Entry i	**Definition** d_i	**Phonetic variation** p_i
w	Word	*arrematação*
1	Word with no accents	*arrematacao*
2	Word phonemes	*aRematasao*
3	Vowel phonemes only	*aeaaao*
4	Vowel phonemes (reverse)	*oaaaea*
5	Consonant phonemes	*Rmts*
6	Consonant phonemes (reverse)	*stmR*
7	Articulation manner	*EABC*
8	Articulation manner (reverse)	*CBAE*
9	Articulation point	*FACD*
10	Articulation point (reverse)	*DCAF*

The function $PhoneticMapSim_{PT}(M_1, M_2)$, as defined in Formula 1, calculates the phonetic similarity between PhoneticMaps M_1 and M_2 as the string similarity weighted average between some phonetic variations of M_1 and M_2. We defined weights used in $PhoneticMapSim_{PT}$ empirically, in order to give more importance to similarities of consonant phonemes.

$$PhoneticMapSim_{PT}(M_1, M_2) =$$

$$\frac{1 \times S_w + 2 \times S_{(1)} + 5 \times S_{(2)} + 1 \times S_{(3)} + 3 \times S_{(5)} + 2 \times S_{(7)} + 2 \times S_{(9)}}{1 + 2 + 5 + 1 + 3 + 2 + 2} \quad (1)$$

where:

$S_w = String_{sim}((M_1.w, M_2.w)$

$S_{(i)} = String_{sim}((M_1.p_i, M_2.p_i)$

2.3 Phonetic Search

To perform a fast phonetic similarity search (FPSS), we propose a method for indexing PhoneticMaps (using single column indexes in a relational database) and phonetically searching the words. FPSS must locate phonetically similar words in the repositories based on the indexed phoneme variations, returning not only similar words but also the similarity level of each one.

Definition 3 (Fast Phonetic Similarity Search). *Given a word w and a minimum desirable similarity level l,* $PhoneticSearch(w, l)$ *is a generic function that results a set of tuples* (r, s)*, where r is a phonetically similar word, and s is the similarity level resulted between* $PhoneticMap(w)$ *and* $PhoneticMap(r)$*, where* $s \geq l$*. Similarity level ranges from 0 to 1.*

To support FPSS, we extended the PWN repository with the *PhoneticMap* table that stores PhoneticMap entries for a specific language (Figure 3). Each *Phonetic Variation* (1–10) is indexed, to support a fast search over each column.

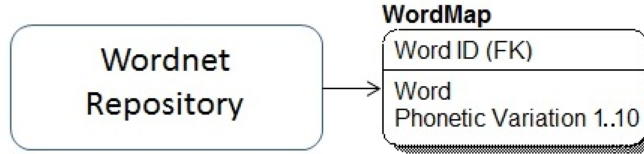

Fig. 3. PWN repository extended to support fast phonetic similarity search

FPSS is performed using the pseudocode described in Figure 4. We develop $PhoneticSearch_{PT}(w, l, p, s)$ function, an extended version of the $PhoneticSearch$ function that returns a set of similar words in Portuguese, considering the minimum desirable similarity level l. Additional parameters p and s set the number of extended consonant phonemes that can be considered as prefix and suffix when searching for similar words. p and s have default values 0 (zero). When $p > 0$, then $PhoneticSearch_{PT}$ uses the reverse indexed PhoneticMaps entries to locate similar words (entries 4, 6, 8 and 10 described in Table 2). Figure 4 shows $PhoneticSearch_{PT}$ pseudocode.

Function $DBPhoneticMapSearch(column, value, extension)$ finds records in the *PhoneticMap* table, searching for *column* equals to *value* (exact match), or *column* like *value* with up to *extension* characters added ("like" match), when *extension* is set > 0. $PhoneticSearch_{PT}$ results a exact match when *minSimLevel = 1* (lines 2–3). Otherwise, it creates a dataset combining results of different $DBPhoneticMapSearch$ executions (line 5). In lines 6–8, phonetic variations 4, 6, 8, and 10 are used whether it is necessary to perform search over the reverse PhoneticMap entries ($prefix > 0$). After creating a result set of candidate words, the phonetic similarity between each found word and the search word is calculated (line 10). Words that does not satisfy the *minSimLevel* are removed from the result set (lines 10-11).

Even it is presented as an instance for Brazilian Portuguese language, this approach is tailored to adapt phonetic matching to use over large repositories and for different languages, as English and Spanish, since one can define a new *PhoneticMap* structure for an specific language, and instantiate the $PhoneticMapSim$ and $PhoneticSearch$ functions for such language.

in:	$word$ String, $minSimLevel$ Number, $prefix$ Integer default 0, $suffix$ Integer default 0
out:	$result$ Dataset
1 :	$wm \leftarrow PhoneticMap_{PT}(word)$;
2 :	if $minSimLevel = 1$ then
3 :	$result \leftarrow DBPhoneticMapSearch(0, wm.w)$;
4 :	else;
5 :	$result \leftarrow$ $DBPhoneticMapSearch(1, wm.p_1) \cup$ $DBPhoneticMapSearch(2, wm.p_2) \cup$ $DBPhoneticMapSearch(3, wm.p_3, suffix) \cup$ $DBPhoneticMapSearch(5, wm.p_5, suffix) \cup$ $DBPhoneticMapSearch(7, wm.p_7, suffix) \cup$ $DBPhoneticMapSearch(9, wm.p_9, suffix)$;
6 :	if $p > 0$ then
7 :	$result \leftarrow result \cup$ $DBPhoneticMapSearch(4, wm.p_4) \cup$ $DBPhoneticMapSearch(6, wm.p_6, prefix) \cup$ $DBPhoneticMapSearch(8, wm.p_8, prefix) \cup$ $DBPhoneticMapSearch(10, wm.p_{10}, prefix)$;
8 :	end if;
9 :	foreach ($fWord$ in $result$)
10 :	if $PhoneticMapSim_{PT}(wm, PhoneticMap_{PT}(fWord))$ $< minSimLevel$ then
11 :	$result.remove(fWord)$;
12 :	end if;
13 :	end if;
14 :	return $result$;

Fig. 4. $PhoneticSearch_{PT}$ pseudocode

3 Experiments

In this section we describe the experiments conducted to validate our approach. First, we compare our String Similarity algorithms with two well-known ones. Second, we compare the performance of our full search method with a search using the indexed *PhoneticMap*s. Finally, we describe precision and recall results of our fast phonetic search solution. We implemented our solution using an instance of Oracle database version 11g running over a Intel(R) Core(TM) i5 2.50 GHz with 8GB RAM.

3.1 String Similarity

We performed an experiment to verify the efficiency of $String_{sim}$ in automatic error correction compared with other functions, comprising the following steps:

– We extracted a set of 3,933 words containing spelling errors from a sample of medical record texts in Portuguese. Each word was manually annotated with the correct spelling form. We call these words *reference* words.
– We used the $String_{sim}$ function to search for the 10 most similar words for each incorrect word, based on the returned similarity values. We used a Brazilian Portuguese version of PWN dictionary (containing 798,750 distinct words, verb conjugation derivatives). The resultsets for each word were ranked from 1 (most similar) to 10 (less similar).
– We store the rank in which each *reference* word is found in each resultset — $Rank = NotFound$ when the *reference* word is not in the resulted recordset.
– The two previous steps were repeated using Edit distance and Jaro-Winkler functions — thus, for each misspelled word, we store the rank of the *reference* word for each one of the functions.
– Lastly, we compared the results of $String_{sim}$ against Edit distance and Jaro-Winkler, as shown in Tables 3 and 4. These tables show that $String_{sim}$ had more reference words with top-1 ranking, which is the objective of the approach.

Table 3. $String_{sim}$ (SS) x Edit Distance (ED)

SS	**ED Rank**					***Not***
Rank	**1**	**2**	**3**	**4-5**	**6-10**	***Found***
1	2970	420	51	30	25	26
2	127	51	37	15	18	13
3	32	12	8	8	3	7
4-5	17	8	7	4	6	3
6-10	14	1	2	4	4	0
Not Found	2	0	0	1	1	6

Table 3 compares $String_{sim}$ and Edit Distance. In 75.5% of cases (2,970 words), both functions find the reference word in the dictionary as a top-1 ranking (the most similar). For the remaining cases, $String_{sim}$ performs better (finds the reference word in a better rank) than Edit Distance in 16.9% of cases (666 searches with better ranking) while Edit Distance is better than $String_{sim}$ in only 5.8% (230 searches).

Table 4 compares $String_{sim}$ and Jaro-Winkler distance. In 68.0% of cases (2,675 words), both functions find the reference word as a top-1 ranking. For other cases, $String_{sim}$ performs better than Jaro-Winkler in 23.7% of cases (934 searches) while Jaro-Winkler is better than $String_{sim}$ in only 4.9% (193 searches).

3.2 Full and Fast Similarity Search

We compared the performance of full and fast similarity search methods, using a set of words extracted from *Aurélio* [6] (a Brazilian Portuguese dictionary).

Table 4. $String_{sim}$ (SS) x Jaro-Winkler (JW)

SS Rank	**JW Rank 1**	**2**	**3**	**4-5**	**6-10**	***Not Found***
1	2675	360	119	107	92	169
2	102	49	25	19	22	44
3	20	19	11	8	4	8
4-5	14	8	4	8	4	7
6-10	4	9	3	1	3	5
Not Found	1	1	1	1	2	4

The steps to perform the experiment are described as follows: (1) A WordNet repository was created in a relational database and it was populated with a total amount of 798,750 distinct words and verb conjugation derivatives; (2) one PhoneticMap for each dictionary entry was created with function $PhoneticMap_{PT}$, populating the *PhoneticMap* table. The table was indexed with 11 single-column indexes – one for the *Word* column and 10 for each *Phonetic Variation*; (3) The same set of 3,933 words containing spelling errors used in the previous experiment was applied in the search methods; (4) A *Full Search* was executed – each input word was compared with each dictionary entry using the $String_{sim}$ (Figure 1), searching for words with a similarity level ≥ 0.8; the spent search time and the number of found words were computed in the result – $PhoneticMapSim_{PT}$ function was not used in the *Full Search* due to its high processing time (60 seconds in average to perform each search). (5) A *Fast Search* was executed – each input word was submitted twice to $PhoneticSearch_{PT}$ (Figure 4), with two different set of parameters: 5a) similarity level ≥ 0.9, and parameters p and s both equal to 0 (similar words might have the same number of consonant phonemes); and 5b) similarity level ≥ 0.8, and parameters p and s both equal to 1 (similar words could have one additional consonant phonemes as prefix or suffix); (6) *Full Search* and *Fast Search* results were compared based on (6a) the total amount of spent time to execute each search, and (6b) the number of words obtained as the search result.

Table 5 shows that *Fast Search* can be 10-30 times faster than *Full Search*. However, it should be clear that *Fast Search* does not return the same number of similar words. Although a *Full Search* is complete in terms of the resulting words, both search methods did not use the same similarity function — *Full Search* was performed with $String_{sim}$, and *Fast Search* used the phonetic similarity metric $PhoneticMapSim_{PT}$, i.e., using the *PhoneticMap*s.

Even with a different number of words in the fast method result, $PhoneticSearch_{PT}$ is able to find the reference word for each spelling error. Table 6 compares accuracy between $String_{sim}$ (SS) and $PhoneticSearch_{PT}$ (PS). In 80.6% of cases (3.170 words), both functions find the reference word as a top-1 ranking. $String_{sim}$ performs better in 10.2% of cases (402 searches) while PS is better in 8.1% (317 searches). Tables 6 shows in bold the amount of cases in which $PhoneticSearch_{PT}$ result was better than $String_{sim}$.

Table 5. Average spent time (in secods) to execute word search

Method	Similarity Level	Spent Time (average)	Words Found (average)
Full	≥ 0.80	4.92 seconds	29.95
Fast	≥ 0.80	0.49 seconds	62.31
Fast	≥ 0.90	0.17 seconds	6.28

Table 6. $PhoneticSearch_{PT}$ x $String_{sim}$

PS Rank	SS Rank 1	2	3	4-5	6-10	***Not Found***
1	3170	189	50	31	14	5
2	143	37	10	7	1	0
3	57	13	1	2	0	1
4-5	46	9	6	4	5	0
6-10	47	6	0	0	2	1
Not Found	59	7	3	1	3	3

3.3 Comparing Results: Precision and Recall

We analyze our results through precision and recall relevance measures. Precision is equivalent to the amount of retrieved instances that are relevant, while recall is equivalent to the amount of relevant instances that are retrieved. For classification and search tasks, the terms *true positives* (TP) and *true negatives* (TN) represent the correct result and the correct absence of result respectively, while the terms *false positives* (FP) and *false negatives* (FN) correspond to the unexpected result and the missing result respectively. These terms are used to define: $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$. In addition, we also present the result of F-measure (or F_1-score) which is a measure of accuracy that considers both the precision and the recall to compute the score: $F_1 = 2 \times \frac{Precision \times Recall}{Precision+Recall}$. F-measure result can be interpreted as a weighted average of the precision and recall, where an score reaches its best value at 1 and worst score at 0 [4].

In our experiment, the concepts TP, FP e FN can be defined according to the results obtained by the *Fast Search* with respect to the results previously performed by the *Full Search*. It means that, for each word w submitted to the $PhoneticSearch_{PT}$, we define:

$$TP(w) = count\big(FastSearch(w, \alpha) \cap FullSearch(w, \beta)\big) \tag{2}$$

$$FP(w) = count\big(FastSearch(w, \alpha) \notin FullSearch(w, \beta)\big) \tag{3}$$

$$FN(w) = count\big(FullSearch(w, \beta) \notin FastSearch(w, \alpha)\big) \tag{4}$$

where $FastSearch(W, \alpha)$ represents the list of words returned by a *Fast Search* method performed to a given word w using a similarity level α, and

$FullSearch(W, \beta)$ is equivalent to the list of words returned by a *Full Search* method performed to a given word w using a similarity level β.

Table 7 shows the average precision and recall, calculated based on the precision and recall of each word submitted to the *Fast Search*, compared to the result of *Full Search* with $\beta = 0.9$. Lower values are observed in column *Avg Precision* for $\alpha < 0.95$ (value of FP is high). It means that $FastSearch$ results more similar words then $FullSearch$, even considering $\alpha > \beta$. Column *Avg Recall* highlights the fact of FN is, in average, around 25% of TP, what means that 25% of words returned by $FullSearch$ do not appear in the $FastSearch$ result. This loss also draws that not all words returned as "similar" by the $String_{sim}$ function in the *Full Search* method are considered phonetically similar by the $PhoneticMapSim_{PT}$ function in the *Fast Search* method.

Table 7. Precision, Recall and F-measure Results ($\beta = 0.9$)

α	**Avg Precision**	**Avg Recall**	**F-measure**
0.80	0.1838%	0.8616%	0.3029
0.85	0.3014%	0.8529%	0.4454
0.90	0.4836%	0.7923%	0.6006
0.95	0.7342%	0.7028%	0.7182

The results obtained with full (string similarity) and fast (phonetic similarity) methods are complementary. A hybrid approach can generate better results than using one single alternative.

4 Related Work

Edit Distance (ED) (or Levenshtein Distance) [14] – $ED(w_1, w_2)$ – calculates the minimum number of operations (single-character edits) required to transform string w_1 into w_2. ED can be also normalized to calculate a percentage similarity instead of the number of operations needed to transform one string to another.

[9] presents a survey with the existing works on text similarity through partitioning them into three approaches. In addition, examples of string similarity applications can be found in the literature:

- [18] used the Levenshtein distance to measuring differences on dialects pronunciations over 27 Dutch dialects in a database.
- Hamming Distance calculates the number of bits (or characters) that are different between two vectors (or strings) [11].
- Longest Commom Subsequence (LCS) finds the longest subsequence of two strings that is as long as any other common subsequence [19, 1]. ROUGE-L is an automatic method for machine translation evaluation based on LCS, which empirical results showed that method is correlated with human judgments [20]. AckSeer is "a search engine and a repository for automatically

extracted acknowledgments in the CiteSeerX digital library", using LCS to evaluate disambiguation of abbreviations in the proposed dataset [21].

- Smith-Waterman distance was originally designed to identify similarities between linked DNA and protein sequencies [23]. The Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. In [24] we can find an application that uses the Smtih-Waterman algorithm and Levenshtein Distance to detect plagiarism in academic papers. Monge-Elkan distance [22] is a recursive variant of the Smith-Waterman distance function which assigns a relatively lower cost to a sequence of insertions and deletions to identify equivalent data in multiple sources ("field matching problem").
- Jaro-Winkler distance is generally used to compare prefix of strings [3]. For example, [7] adopted the Jaro-Winkler distance to compare the similarities of Geography Markup Language (GML) nodes and ontology tree node.
- [3] compares different string distance metrics for name-matching tasks, including edit-distance like functions, token-based distance functions and hybrid methods, concluding Monge-Elkan distance performed best among several metrics.

Soundex is a phonetic matching scheme initially designed for English that uses codes based on the sound of each letter to translate a string into a canonical form of at most four characters, preserving the first letter [26]. For example, "reynold" and "renauld" are both reduced to "r543". As the result, phonetically similar entries will have the same keys and they can be indexed for efficient search using some hashing method. However, Soundex fails to consider only the initial portion of a string to generate the phonetic representation, which impairs the phonetic comparison when words have more than 4-5 consonant phonemes [10]. More commonly, Soundex also makes the error of transforming dissimilar-sounding strings such as "catherine" and "cotroneo" to the same code, and of transforming similar-sounding strings to diffrent codes.

State Set Index (SSI) [5] as an efficient solution for finding strings in a string set that are similar to the query string. SSI is based on a trie (prefix index) that is interpreted as a nondeterministic finite automaton and it implements a novel state labeling strategy making the index highly space-efficient.

Fast Similarity Search (FastSS) [2] is an algorithm designed to find strings similarities in a large database. This algorithm is based on ED metric. According to the authors, in a dictionary that cointais n words, and given a maximum number e of spelling errors, the FastSS algorithm creates an index of all n words containing up to e deletions. Each query is mutated, at search time, to generate a deletion neighborhood, which is compared to the indexed deletion dictionary. The algorithm was tested and compared with NR-grep, a keyword tree, dynamic programming, n-grams, and neighborhood generation using entries of the English Dictionary, English Wikipedia and a chapter from the book Moby Dick.

The more important difference of our approach from these solutions is the utilization of phonetic information in an indexed structure. This structure is well adapted for calculating string similarity between misspelled words, since not all

the characters may be relevant. In addition, non phonetic approaches may result in high similarity values, but for different words with distinct phonetic.

5 Conclusions and Future Work

We presented an approach of fast phonetic similarity search coupled with an extended version of the WordNet dictionary. Our main contribution is the definition of an index structured, called *PhoneticMap* that stores phonetic information to be used by a novel string similarity search algorithm, which is described in detail. We adapted the Wordnet dictionary to support this structure. We implemented our approach using a relational database. The experiments showed that the algorithm has good precision results, and that it executes faster than one version of the algorithm that does not use the $PhoneticMap$. We also presented a string similarity algorithm based on the notion of penalty. We compared this algorithm with other string similarity solutions: it keeps the similarity values higher for words with less than 4–6 differences. In contrast, it decreases and converges to zero the similarity values for strings with more than 10 differences.

As future work, we plan to use our solution as the initial component of Medical Records Information Extraction System to address the problem of dealing with spelling errors in such extracting process. The approach is tailored to adapt phonetic matching to use over large repositories and easily adaptable to different languages, as English and Spanish, to create new instances for the $PhoneticMap$ structure, and the $PhoneticMapSim$ and $PhoneticSearch$ function. We plan to explore the usage of machine learning methods to optimally tune the parameters involved in the proposed hybrid similarity metrics, and to compare our proposed solution with other phonetic search approaches.

References

1. Allison, L., Dix, T.I.: A Bit-String Longest-Common-Subsequence Algorithm. Inf. Process. Lett., vol. 26, pp. 305–310 (1986)
2. Bocek, T., Hunt, E., Stiller, B., Hecht, F.: Fast similarity search in large dictionaries. Department of Informatics, University of Zurich (2007)
3. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: IIWeb, pp. 73–78 (2003)
4. Davis, J., Goadrich, M.: The relationship between Precision-Recall and ROC curves. In: Proceedings of the 23rd international conference on Machine learning, ACM. ICML '06, pp. 233–240. New York (2006)
5. Fenz, D., Lange, D., Rheinlander, A., Naumann, F., Leser, U.: Efficient similarity search in very large string sets. In: Proceedings of the 24th international conference on Scientific and Statistical Database Management, Springer-Verlag, SSDBM'12, pp. 262–279. Berlin, Heidelberg (2012)
6. Ferreira, A.B.: Novo dicionario Aurelio da lingua portuguesa. Positivo (2004)
7. Frozza, A.A., Mello, R.S.: Um metodo para determinar a equivalencia semantica entre esquemas GML. GeoInfo, 283–294 (2006)

8. Godbole, S., Bhattacharya, I., Gupta, A., Verma, A.: Building re-usable dictionary repositories for real-world text mining. In: Huang, J., Koudas, N., Jones, G. J. F., Wu, X., Collins-Thompson, K., An, A. (eds.). CIKM, pp. 1189-1198. ACM (2010)
9. Gomaa, W.H., Fahmy, A.A.: A Survey of Text Similarity Approaches. International Journal of Computer Applications, vol. 68, pp. 13–18. Foundation of Computer Science, New York (2013)
10. Hall, P.A.V., Dowling, G.R.: Approximate String Matching. ACM Comput. Surv., vol. 12, pp. 381–402. ACM, New York (1980)
11. Hamming, R.: Error Detecting and Error Correcting Codes. Bell System Technical Journal, vol. 26, pp. 147–160 (1950)
12. Jellouli, I., Mohajir, M.E.: An ontology-based approach for web information extraction. In: Information Science and Technology (CIST), 2011 Colloquium, pp. 5 (2011)
13. Pavel, S., Euzenat, J.: Ontology Matching: State of the Art and Future Challenges. IEEE Transactions on Knowledge and Data Engineering, 99 (2011)
14. Levenshtein, V.I.: Binary codes capable of correcting insertions and reversals. Soviet Physics Doklady, vol. 10, pp. 707–710 (1966)
15. Miller, G.A.: WordNet: a lexical database for English. Commun. ACM, vol. 38, pp. 39–41. ACM, New York (1995)
16. Stvilia, B.: A model for ontology quality evaluation. First Monday, vol. 12 (2007)
17. Mann, V.A.: Distinguishing universal and language-dependent levels of speech perception: Evidence from Japanese listeners' perception of English. Cognition, vol. 24, pp. 169 - 196 (1986)
18. Heeringa, W.: Measuring Dialect Pronunciation Differences Using Levenshtein Distance. Groningen dissertations in linguistics. University Library Groningen (2004)
19. Paterson, M., Dancik, V.: Longest Common Subsequences. In: Proc. of 19th MFCS, number 841 in LNCS, pp. 127–142. Springer (1994)
20. Lin, C.Y., Och, F.J.: Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics (2004)
21. Khabsa, M., Treeratpituk, P., Giles, C.L.: Ackseer: a repository and search engine for automatically extracted acknowledgments from digital libraries. In: Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries, pp. 185–194. ACM (2012)
22. Monge, A., Elkan, C.: The field matching problem: Algorithms and applications. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, pp. 267–270 (1996)
23. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of molecular biology, vol. 147, pp. 195–197. Academic Press (1981)
24. Su, Z., Ahn, B.R., Eom, K.Y., Kang, M.K., Kim, J.P., Kim, M.K.: Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm. Innovative Computing Information and Control, 2008. ICICIC '08. 3rd International Conference, pp. 569-569 (2008)
25. Winkler, William E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In: Proceedings of the Section on Survey Research, 1990, S. 354–359
26. Zobel, J., Dart, P.W.: Phonetic String Matching: Lessons from Information Retrieval. In: Frei, H.P., Harman, D., Schauble, P., Wilkinson, R. (eds.) ACM. SIGIR, pp. 166-172 (1996)