Language Independent Text Correction using Finite State Automata

Ahmed Hassan* **Sara Noeman** **Hany Hassan**
IBM Cairo Technology Development Center
Giza, Egypt
hasanah,noemans,hanyh@eg.ibm.com

Abstract

Many natural language applications, like machine translation and information extraction, are required to operate on text with spelling errors. Those spelling mistakes have to be corrected automatically to avoid deteriorating the performance of such applications. In this work, we introduce a novel approach for automatic correction of spelling mistakes by deploying finite state automata to propose candidates corrections within a specified edit distance from the misspelled word. After choosing candidate corrections, a language model is used to assign scores the candidate corrections and choose best correction in the given context. The proposed approach is language independent and requires only a dictionary and text data for building a language model. The approach have been tested on both Arabic and English text and achieved accuracy of 89%.

1 Introduction

The problem of detecting and correcting misspelled words in text has received great attention due to its importance in several applications like text editing systems, optical character recognition systems, and morphological analysis and tagging (Roche and Schabes, 1995). Other applications, like machine translation and information extraction, operate on text that might have spelling errors. The automatic detection, and correction of spelling erros should be of great help to those applications.

The problem of detecting and correcting misspelled words in text is usually solved by checking whether a word already exists in the dictionary or not. If not, we try to extract words from the dictionary that are most similar to the word in question. Those words are reported as candidate corrections for the misspelled word.

Similarity between the misspelled word and dictionary words is measured by the Levenshtein edit distance (Levenshtein, 1966; Wagner and M.Fisher, 1974). The Levenshtein edit distance is usually calculated using a dynamic programming technique with quadratic time complexity (Wagner and M.Fisher, 1974). Hence, it is not reasonable to compare the misspelled word to each word in the dictionary while trying to find candidate corrections.

The proposed approach uses techniques from finite state theory to detect misspelled words and to generate a set of candidate corrections for each misspelled word. It also uses a language model to select the best correction from the set of candidate corrections using the context of the misspelled word. Using techniques from finite state theory, and avoiding calculating edit distances makes the approach very fast and efficient. The approach is completely language independent, and can be used with any language that has a dictionary and text data to building a language model.

The rest of this paper will proceed as follows. Section 2 will present an overview of related work. Section 3 will discuss the different aspects of the proposed approach. Section 4 presents a performance evaluation of the system. Finally a conclusion is presented in section 5.

2 Related Work

Several solutions were suggested to avoid computing the Levenshtein edit distance while finding candidate corrections. Most of those solutions select a number of dictionary words that are supposed to contain the correction, and then measure the distance between the misspelled word and all selected words. The most popular of those methods are the similarity keys methods (Kukich, 1992; Zobel and Dart, 1995; De Beuvron and Trigano, 1995). In

*Now with the University of Michigan Ann Arbor, hassanam@umich.edu

those methods, the dictionary words are divided into classes according to some word features. The input word is compared to words in classes that have similar features only.

In addition to the techniques discussed above, other techniques from finite state automata have been recently proposed. (Oflazer, 1996) suggested a method where all words in a dictionary are treated as a regular language over an alphabet of letters. All the words are represented by a finite state machine automaton. For each garbled input word, an exhaustive traversal of the dictionary automaton is initiated using a variant of Wagner-Fisher algorithm (Wagner and M.Fisher, 1974) to control the traversal of the dictionary. In this approach Levenshtein distance is calculated several times during the traversal. The method carefully traverses the dictionary such that the inspection of most of the dictionary states is avoided. (Schulz and Mihov, 2002) presents a variant of Oflazers's approach where the dictionary is also represented as deterministic finite state automaton. However, they avoid the computation of Levenshtein distance during the traversal of the dictionary automaton. In this technique, a finite state acceptor is constructed for each input word. This acceptor accepts all words that are within an edit distance k from the input word. The dictionary automaton and the Levenshtein-automaton are then traversed in parallel to extract candidate corrections for the misspelled word. The authors present an algorithm that can construct a deterministic Levenshtein-automaton for an arbitrary word of degrees 1, and 2 which corresponds to 1 or 2 errors only. They suggest another algorithm that can construct a non-deterministic Levenshtein-automaton for any other degree. They report results using a Levenshtein-automaton of degree 1(i.e. words having a single insertion, substitution, or deletion) only.

The method we propose in this work also assumes that the dictionary is represented as a deterministic finite state automaton. However, we completely avoid computing the Levenshtein-distance at any step. We also avoid reconstructing a Levenshtein-automaton for each input word. The proposed method does not impose any constraints on the bound k, where k is the edit distance between the input word and the candidate corrections. The approach can adopt several constraints on which characters can substitute certain other characters. Those constraints are obtained from a phonetic and spatial confusion matrix of characters.

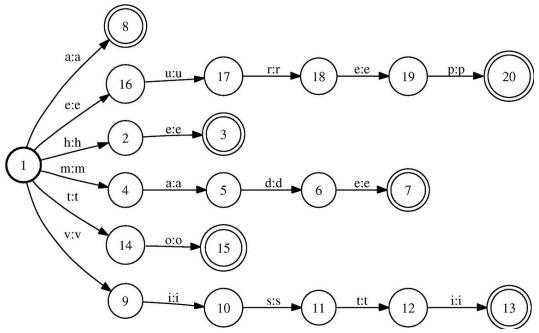

Figure 1: An FSM representation of a word list

The purpose of context-dependent error correction is to rank a set of candidate corrections taking the misspelled word context into account. A number of approaches have been proposed to tackle this problem that use insights from statistical machine learning (Golding and Roth, 1999), lexical semantics (Hirst and Budanitsky, 2005), and web crawls (Ringlstetter et al., 2007).

3 Error Detection and Correction in Text Using FSMs

The approach consists of three main phases: detecting misspelled words, generating candidate corrections for them, and ranking corrections. A detailed description of each phase is given in the following subsections.

3.1 Detecting Misspelled Words

The most direct way for detecting misspelled words is to search the dictionary for each word, and report words not found in the dictionary. However, we can make use of the finite state automaton representation of the dictionary to make this step more efficient. In the proposed method, we build a finite state machine (FSM) that contains a path for each word in the input string. This FSM is then composed with the dictionary FSM. The result of the composition is merely the intersection of the words that exist in both the input string and the dictionary. If we calculated the difference between the FSM containing all words and this FSM, we get an FSM with a path for

each misspelled word. Figure 1 illustrate an FSM that contain all words in an input string.

3.2 Generating Candidate Corrections

The task of generating candidate corrections for misspelled words can be divided into two sub tasks: Generating a list of words that have edit distance less than or equal k to the input word, and selecting a subset of those words that also exist in the dictionary. To accomplish those tasks, we create a single transducer(Levenshtein-transducer) that is when composed with an FSM representing a word, generates all words withing any edit distance k from the input word. After composing the misspelled word with Levenshtein-transducer, we compose the resulting FSM with the dictionary FSM to filter out words that do not exist in the dictionary.

3.2.1 Levenshtein-transducers for primitive edit distances

To generate a finite state automaton that contain all words within some edit distance to the input word, we use a finite state transducer that allows editing its input according to the standard Levenshtein-distance primitive operations: substitution, deletion, and insertion.

A finite-state transducers (FST) is a a 6-tuple $(Q, \Sigma_1, \Sigma_2, \sigma, i, F)$, where Q is a set of states, Σ_1 is the input alphabet, Σ_2 is the output alphabet, i is the initial state, $F \subseteq Q$ is a set of final states, and σ is a transition function (Hopcroft and Ullman, 1979; Roche and Shabes, 1997). A finite state acceptor is a special case of an FST that has the same input/output at each arc.

Figure 2 illustrates the Levenshtein-transducer for edit distance 1 over a limited set of vocabulary (a,b, and c). We can notice that we will stay in state zero as long as the output is identical to the input. On the other hand we can move from state zero, which corresponds to edit distance zero, to state one, which corresponds to edit distance one, with three different ways:

- input is mapped to a different output (input is consumed and a different symbol is emitted) which corresponds to a substitution,
- input is mapped to an epsilon (input is consumed and no output emitted) which corresponds to a deletion, and

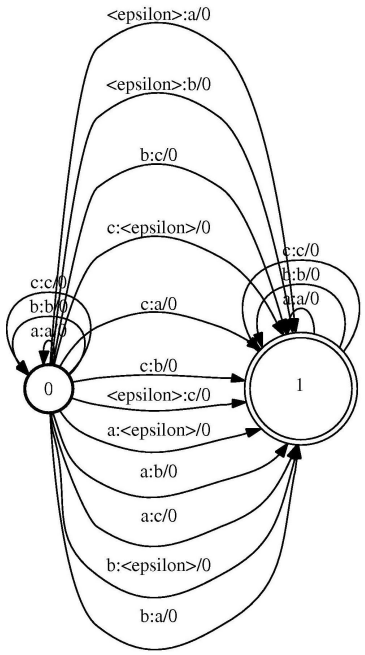

Figure 2: A Levenshtein-transducer (edit distance 1)

- an epsilon is mapped to an output (output is emitted without consuming any input) which corresponds to an insertion.

Once we reach state 1, the only possible transitions are those that consume a symbol and emit the same symbol again and hence allowing only one edit operation to take place.

When we receive a new misspelled word, we represent it with a finite state acceptor that has a single path representing the word, and then compose it with the Levenshtein-transducer. The result of the composition is a new FSM that contains all words with edit distance 1 to the input word.

3.2.2 Adding transposition

Another non-primitive edit distance operation that is frequently seen in misspelled words is transposition. Transposition is the operation of exchanging the order of two consecutive symbols ($ab \rightarrow ba$). Transposition is not a primitive operation because it can be represented by other primitive operations. However, this makes it a second degree operation. As transposition occurs frequently in misspelled words, adding it to the Levenshtein-transducer as a single editing operation would be of great help.

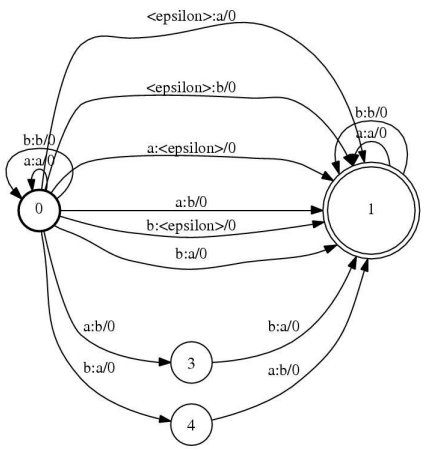

Figure 3: A Levenshtein-transducer for edit distance 1 with transposition

To add transposition, as a single editing operation, to the Levenshtein-transducer we add arcs between states zero and one that can map any symbol sequence xy to the symbol sequence yx, where x, and y are any two symbols in the vocabulary. Figure 3 shows the Levenshtein-transducer with degree 1 with transposition over a limited vocabulary (a and b).

3.2.3 Adding symbol confusion matrices

Adding a symbol confusion matrix can help reduce the number of candidate corrections. The confusion matrix determines for each symbol a set of symbols that may have substituted it in the garbled word. This matrix can be used to reduce the number of candidate corrections if incorporated into the Levenshtein-transducer. For any symbol x, we add an arc $x : y$ between states zero, and one in the transducer where $y \in Confusion\_Matrix(x)$ rather than for all symbols y in the vocabulary.

The confusion matrix can help adopt the methods to different applications. For example, we can build a confusion matrix for use with optical character recognition error correction that captures errors that usually occur with OCRs. When used with a text editing system, we can use a confusion matrix that predicts the confused characters according to their phonetic similarity, and their spatial location on the keyboard.

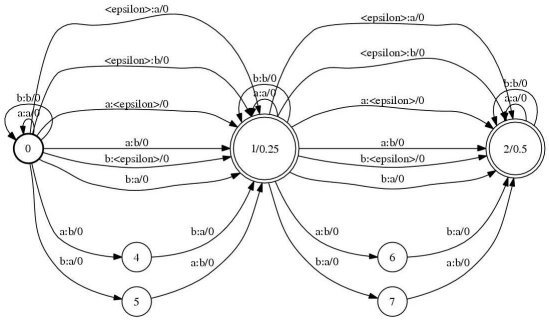

Figure 4: A Levenshtein-transducer for edit distance 2 with transposition

3.2.4 Using degrees greater than one

To create a Levenshtein-transducer that can generate all words within edit distance two of the input word, we create a new state (2) that maps to two edit operations, and repeat all arcs that moves from state 0 to state 1 to move from state 1 to state 2.

To allow the Levenshtein-transducer of degree two to produce words with edit distance 1 and 2 from the input word, we mark both state 1, and 2 as final states. We may also favor corrections with lower edit distances by assigning costs to final states, such that final states with lower number of edit operations get lower costs. A Levenshtein-transducer of degree 2 for the limited vocabulary (a and b) is shown in figure 4.

3.3 Ranking Corrections

To select the best correction from a set of candidate corrections, we use a language model to assign a probability to a sequence of words containing the corrected word. To get that word sequence, we go back to the context where the misspelled word appeared, replace the misspelled word with the candidate correction, and extract n *ngrams* containing the candidate correction word in all possible positions in the *ngram*. We then assign a score to each *ngram* using the language model, and assign a score to the candidate correction that equals the average score of all *ngrams*. Before selecting the best scoring correction, we penalize corrections that resulted from higher edit operations to favor corrections with the minimal number of editing operations.

	Edit 1/with trans.		Edit 1/no trans.		Edit 2/with trans.		Edit 2 / no trans.	
word len.	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**
3	3.373273	18.769	2.983733	18.197	73.143538	532.637	69.709387	514.174
4	3.280419	4.797	2.796275	4.715	67.864291	136.230	66.279842	131.680
5	3.321769	1.858	2.637421	1.838	73.718353	33.434	68.695935	32.461
6	3.590046	1.283	2.877242	1.277	75.465624	11.489	69.246055	11.258
7	3.817453	1.139	2.785156	1.139	78.231015	6.373	72.2057	6.277
8	4.073228	1.063	5.593761	1.062	77.096026	4.127	73.361455	4.066
9	4.321661	1.036	3.124661	1.036	76.991945	3.122	73.058418	3.091
10	4.739503	1.020	3.2084	1.020	75.427416	2.706	72.2143	2.685
11	4.892105	1.007	3.405101	1.007	77.045616	2.287	71.293116	2.281
12	5.052191	0.993	3.505089	0.993	78.616536	1.910	75.709801	1.904
13	5.403557	0.936	3.568391	0.936	81.145124	1.575	78.732955	1.568

Table 1: Results for English

	Edit 1/with trans.		Edit 1/no trans.		Edit 2/with trans.		Edit 2 / no trans.	
word len.	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**	**av. time**	**av. correcs.**
3	5.710543	31.702	4.308018	30.697	83.971263	891.579	75.539547	862.495
4	6.033066	12.555	4.036479	12.196	80.481281	308.910	71.042372	296.776
5	7.060306	6.265	4.360373	6.162	79.320644	104.661	69.71572	100.428
6	9.08935	4.427	4.843784	4.359	79.878962	51.392	74.197127	48.991
7	8.469497	3.348	5.419919	3.329	82.231107	24.663	70.681298	23.781
8	10.078842	2.503	5.593761	2.492	85.32005	13.586	71.557569	13.267
9	10.127946	2.140	6.027077	2.136	83.788916	8.733	76.199034	8.645
10	11.04873	1.653	6.259901	1.653	92.671732	6.142	81.007893	6.089
11	12.060286	1.130	7.327353	1.129	94.726469	4.103	77.464609	4.084
12	13.093397	0.968	7.194902	0.967	95.35985	2.481	82.40306	2.462
13	13.925067	0.924	7.740105	0.921	106.66238	1.123	78.966914	1.109

Table 2: Results for Arabic

4 Experimental Setup

4.1 Time Performance

The proposed method was implemented in C++ on a 2GHz processor machine under Linux. We used 11,000 words of length 3,4,..., and 13, 1,000 word for each word length, that have a single error and computed correction candidates. We report both the average correction time, and the average number of corrections for each word length. The experiment was run twice on different test data, one with considering transposition as primitive operation, and the other without. We also repeated the experiments for edit distance 2 errors, and also considered the two cases where transposition is considered as a primitive operation or not. Table 1 shows the results for an English dictionary of size 225,400 entry, and Table 2 shows the results for an Arabic dictionary that has 526,492. entries.

4.2 Auto-correction accuracy

To measure the accuracy of the auto-correction process, we used a list of 556 words having common spelling errors of both edit distances 1 and 2. We put a threshold on the number of characters per word to decide whether it will be considered for edit distance 1 or 2 errors. When using a threshold of 7, the spell engine managed to correct 87% of the words. This percentage raised to 89% when all words were considered for edit distance 2 errors. The small degradation in the performance occured because in 2% of the cases, the words were checked for edit distance 1 errors although they had edit distance 2 errors. Figure 6 shows the effect of varying the characters limit on the correction accuracy.

Figure 5 shows the effect of varying the weight assigned to corrections with lower edit distances on the accuracy. As indicated in the figure, when we only consider the language model weight, we get accuracies as low as 79%. As we favor corrections with lower edit distances the correction accuracy raises, but occasionally starts to decay again when emphasis on the low edit distance is much larger than that on the language model weights.

Finally, we repeated the experiments but with us-

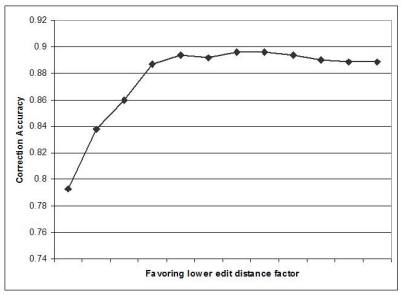

Figure 5: Effect of increasing lower edit distance favoring factor on accuracy

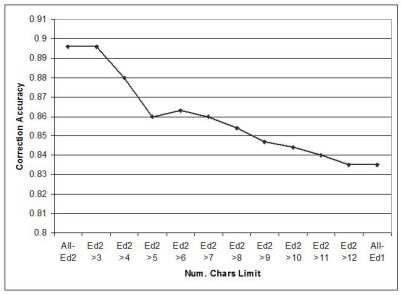

Figure 6: Effect of increasing Ed1/Ed2 char limits on accuracy

ing a confusion matrix, as 3.2.3. We found out that the average computation time dropped by 78% (below 1 ms for edit distance 1 errors) at the price of losing only 8% of the correction accuracy.

5 Conclusion

In this work, we present a finite state automata based spelling errors detection and correction method. The new method avoids calculating the edit distances at all steps of the correction process. It also avoids building a Levenshtein-automata for each input word. The method is multilingual and may work for any language for which we have an electronic dictionary, and a language model to assign probability to word sequences. The preliminary experimental results show that the new method achieves good performance for both correction time and accuracy. The experiments done in this paper can be extended in several directions. First, there is still much room for optimizing the code to make it faster especially the FST composition process. Second, we can allow further editing operations like splitting and merging.

References

Francois De Bertrand De Beuvron and Philippe Trigano. 1995. Hierarchically coded lexicon with variants. *International Journal of Pattern Recognition and Artificial Intelligence*, 9:145–165.

Andrew Golding and Dan Roth. 1999. A winnow-based approach to context-sensitive spelling correction. *Machine learning*, 34:107–130.

Graeme Hirst and Alexander Budanitsky. 2005. Correcting real-word spelling errors by restoring lexical cohesion. *Natural Language Engineering*, 11:87–111.

J.E. Hopcroft and J.D. Ullman. 1979. Introduction to automata theory, languages, and computation. Reading, Massachusetts: Addison-Wesley.

Karen Kukich. 1992. Techniques for automatically correcting words in text. *ACM Computing Surveys*, pages 377–439.

V.I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady*.

Kemal Oflazer. 1996. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22:73–89.

Christoph Ringlstetter, Max Hadersbeck, Klaus U. Schulz, and Stoyan Mihov. 2007. Text correction using domain dependent bigram models from web crawls. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2007) Workshop on Analytics for Noisy Unstructured Text Data*.

Emmanuel Roche and Yves Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 2:227253.

Emmanuel Roche and Yves Shabes. 1997. Finite-state language processing. Cambridge, MA, USA: MIT Press.

Klaus Schulz and Stoyan Mihov. 2002. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition (IJDAR)*, 5:67–85.

R.A. Wagner and M.Fisher. 1974. The string-to-string correction problem. *Journal of the ACM*.

Justin Zobel and Philip Dart. 1995. Finding approximate matches in large lexicons. *Software Practice and Experience*, 25:331–345.