Tuning the Selection of Correction Candidates for Garbled Tokens using Error Dictionaries

Stoyan Mihov Petar Mitankin Annette Gotscharek Ulrich Reffle
Klaus U. Schulz Christoph Ringlstetter

Abstract

In previous work, we introduced a method for efficiently selecting from a background dictionary suitable correction candidates for an malformed token of a given input text. In order to select small and meaningful candidate sets, refinements of the Levenshtein distance with restricted sets of substitutions, merges and splits *were used. In these experiments, the subset of possible substitutions, merges and splits was determined via training, using ground truth data representing corrected parts of the input text. Here we show that an appropriate set of possible substitutions, merges and splits for the input text can be retrieved without any ground truth data. In the new approach, we compute an error profile of the erroneous input text in a fully automated way, using error dictionaries. From this profile, suitable sets of substitutions, merges and splits are derived. Error profiling with error dictionaries is simple and very fast. We obtain an adaptive form of candidate selection which is very efficient, does not need ground truth data and leads to small candidate sets with high recall.*

1 Introduction

When faced with an ill-formed input token, lexical text correction systems typically compute a fine-ranked list of correction suggestions from the dictionary. The ranking takes word similarity, word frequencies and other aspects into account [2]. Until today, no efficient methods are known for computing such a ranked list in a direct, one-step manner. Hence, actual systems are typically based on a first step where a set of correction candidates for the ill-formed token is selected from the given background dictionary. At this point, in order to guarantee maximal efficiency, a coarse similarity measure is used. In a second step, candidates are ranked, now using a fine-grained word similarity measure and other scores [2, 5, 12, 10].

In earlier work [8, 4], we introduced a direct method for efficiently selecting for a given token V all entries W of the background dictionary where the Levenshtein distance d_L (cf. [3]) between V and W does not exceed a given bound n. The method strongly depends on the concept of a universal Levenshtein automaton for fixed bound n. This automaton, only computed once in an offline step, may be used to decide for arbitrary input words V and W if $d_L(V, W) \leq n$. For lexical text correction, the dictionary is represented as a finite state automaton. The universal Levenshtein automaton is used to control a traversal of the dictionary automaton in a way that paths not leading to suitable correction candidates are recognized immediately. In this way, all correction candidates V with $d_L(V, W) \leq n$ are found while visiting only a small part of the dictionary automaton.

While this method is very fast, the resulting sets of correction candidates are often much larger than needed. More recently, we introduced a similar method [9] which is based on a refinement of the Levenshtein distance with *restricted sets of substitutions, merges and splits* as edit operations. The new method selects very small candidate sets which nevertheless contain the proper correction in all cases but a few exceptions. The efficiency of candidate selection is further improved. Hence, the method is well-suited both for automated and interactive text correction. As an input for the new method, restricted sets of substitutions, merges and splits have to be specified. In [9], these sets are obtained via training, aligning ground truth data with the erroneous text to recognize error types.

In many practical application scenarios, ground truth data are not available. This explains why adaptive techniques become more and more relevant that automatically compute a kind of profile of the given input text. These profiles can be used to improve the behaviour of text correction systems in many ways. In this paper, we show how to efficiently compute restricted sets of substitutions, merges and splits without analyzing any ground truth data. We use error dictionaries of a special form to compute an error profile of the given erroneous input text. This form of error profiling is simple and very efficient. From the profile of a text, suitable restricted sets of edit operations are obtained immediately. In our experiments we found that the new method for candidate selection leads to results directly comparable to those obtained from supervised training.

The paper, which combines and refines methods presented in [9, 6], has the following structure. In Section 2 we briefly recall the idea of a universal Levenshtein automaton and indicate how these automata are used for efficiently selecting correction candidates for a garbled input word in a dictionary. In Section 3 we summarize techniques presented in [9]: we introduce the new distance measures based on restricted sets of edit operations that yield refined ranking orders. We then describe a new type of universal automaton, $\mathcal{A}_n$ and show how to use it for refined selection of correction candidates. Section 4 describes the construction of error dictionaries. We show how to compute an error profile for a given input text and how to obtain restricted sets of edit operations from these profiles. In Section 5 we present our evaluation results.

In what follows, we assume that the reader is familiar with the theory of finite state automata. The symbol d_L denotes the Levenshtein distance. Strings are built over a fixed alphabet Σ.

2 Universal Levenshtein automata and fast approximate search in dictionaries

Universal Levenshtein automata [4] can be considered as derivatives of a well-known family of nondeterministic automata $\mathcal{A}_n^W$. For a given word W, $\mathcal{A}_n^W$ accepts all strings V where $d_L(V, W) \leq n$. The automaton $\mathcal{A}_2^{\text{“}chold''}$ is shown in Figure 1. The states of $\mathcal{A}_n^W$ have the form b^i, $0 \leq b \leq |W|$, $0 \leq i \leq n$. If we traverse $\mathcal{A}_n^W$ with the input word V starting from the initial state 0^0, the state b^i indicates that i edit operations have been used for the alignment of $W_1W_2...W_b$ and the prefix of V consumed to reach b^i. Upward transitions represent insertions, empty (non-empty) diagonal transitions represent deletions (substitutions). The universal Levenshtein automaton of degree n, $\mathcal{A}_n^\forall$, represents a *deterministic* version of $\mathcal{A}_n^W$ which at the same time *abstracts from the specific input word* W. For the construction of $\mathcal{A}_n^\forall$, the following observations are crucial:

(1) The set of states of $\mathcal{A}_n^W$ that can be reached from the start with an input of length k (i.e., the set of active states for this input) is always a subset of a “triangular sliding window” of $\leq (n+1)^2$ states anchored in the $k+1$-st state (fltr) of the bottom layer, as indicated in Figure 1. (2) Given any set of active states reached with input of length $k-1$ and a new input symbol σ_k, the next set of active states only depends on the distribution of the letter σ in the subword $w_l \dots w_k \dots w_r$ of $W = w_1 \dots w_h$ where $l = \max\{1, k-n\}$ and $r = \min\{k+n, h\}$. In what follows, $w_l \dots w_r$ is called the *relevant subword* of W for the k-th input symbol.

Ignoring details that arise for triangular windows which contain final states, the automaton $\mathcal{A}_n^\forall$ uses just one “generic” triangular window G instead of the sliding sequence of “positioned” triangular windows of $\mathcal{A}_n^W$. Nonfinal states of $\mathcal{A}_n^\forall$ then correspond to subsets of G.[1] The input of $\mathcal{A}_n^\forall$ are sequences of bitvectors, the k-th bitvector χ_k encoding the distribution of the k-th input letter in an imaginary relevant subword. In this way, any sequence of sets of active states reached with input letters $\sigma_1, \dots, \sigma_k$ in $\mathcal{A}_n^W$ is represented as a sequence of states of $\mathcal{A}_n^\forall$ reached with the bitvectors $\chi_1, \dots, \chi_k$ that encode the distribution of the letters $\sigma_1, \dots, \sigma_k$ in the relevant subwords of W.

The automaton $\mathcal{A}_n^\forall$ is “universal” in the sense that it can be applied to any pair of words V, W. One of the words, say V, is treated as input. The distributions of the letters of V in the relevant subwords of W, formalized as bitvectors, then represent the input for $\mathcal{A}_n^\forall$. The sequence of bitvectors is accepted if and only if $d_L(V, W) \leq n$. More details about the construction of the universal Levenshtein automaton can be found in [4].

Fast approximate search in dictionaries. Assume now we have a dictionary D, encoded as a deterministic finite state automaton $\mathcal{A}_D$. Given an input string W, we want to use $\mathcal{A}_n^\forall$ to find all entries $V \in D$ such that $d_L(V, W) \leq n$. The task is solved by a simple parallel backtracking procedure. Beginning at the initial state we traverse $\mathcal{A}_D$, visiting all prefixes $P = \sigma_1 \dots \sigma_{k-1}$ of dictionary words. At each forward step, the label σ_k of the transition is matched against the relevant subword of W to produce a bitvector χ_k which serves as input for $\mathcal{A}_n^\forall$. Note that in general there are distinct options for selecting σ_k. If the transition in the “control” automaton $\mathcal{A}_n^\forall$ fails, we choose another possible transition in $\mathcal{A}_D$ with label σ'_k. If no choices are left, we go back to the previous state and prefix $P = \sigma_1 \dots \sigma_{k-2}$. We output the prefix P once we reach a final state in both automata.

Forward-backward method. It should be mentioned that the evaluation results for approximate search in dictionaries presented below make use of a highly effective optimization ([4]) in order to speed up the search. Typically, automata encoding a dictionary D for a given natural language have a characteristic structure. The branching degree of nodes is very high in the initial part of the automaton close to the start state (“initial wall”). A huge number of distinct states can be reached with the first two, three transitions. Deeper inside, when 3, 4 nodes are traversed, most nodes only have a very small number of successors. The “forward-backward method” ([4]) uses this observation to speed up approximate search. Given D, two automata $\mathcal{A}_D$ and $\mathcal{A}_D^{rev}$ are used. The former (latter) encodes all words of D in the usual, left-to-right (reverse, right-to-left) order. An erroneous token W given as input is split into two parts W_1, W_2 of approx. the same length. In order to find all lexical neighbours of $W = W_1W_2$ in a given edit distance, we first assume that the ma-

[1] The construction of final states is similar, using another generic window. For simplicity, we also ignore other details of the construction.

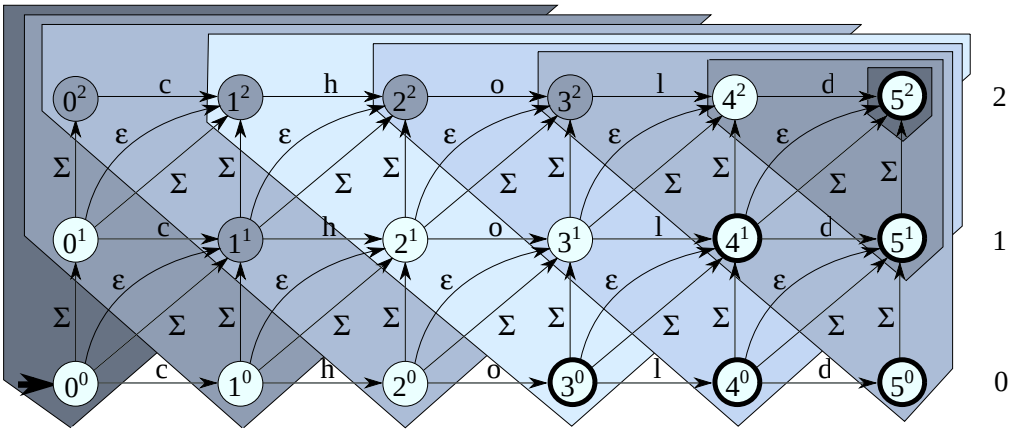

Fig. 1. Nondeterministic Levenshtein automaton $\mathcal{A}_2^{\text{“}chold''}$, triangular sliding windows highlighted.

jority (at least half) of all errors is found in W_2. Since in W_1 the number of possible errors is reduced, not too much backtracking is necessary when traversing the initial wall of $\mathcal{A}_D$, looking for possible variants of W_1. Variants of W_2 are also found quickly since in this part of the automaton $\mathcal{A}_D$ the branching degree of nodes is very small. The second subsearch is dual. We assume that most errors are concentrated on W_1. We now traverse $\mathcal{A}_D^{rev}$, using the reversed input string $W^{rev} = W_2^{rev}W_1^{rev}$. Again we quickly pass through the initial wall of $\mathcal{A}_D^{rev}$, and possible variations of W_1^{rev} are found easily. The speed-up of this method is striking, see [4].

3 Universal Levenshtein automata for refined Levenshtein distances

Definition 3.1 Let three sets $S_{1,1} \subseteq \Sigma \times \Sigma$, $M_{2,1} \subseteq \Sigma^2 \times \Sigma$, $S_{1,2} \subseteq \Sigma \times \Sigma^2$ be given. A *restricted substitution* is an operation where we replace a letter σ in a word W by another letter σ' where $\langle \sigma, \sigma' \rangle \in S_{1,1}$. A *restricted merge* is defined similarly, replacing a bigram $\sigma\sigma'$ of consecutive letters in W by σ'' for $\langle \sigma\sigma', \sigma'' \rangle \in M_{2,1}$. By a *restricted split* we mean an operation where a letter σ of W is replaced by a bigram $\sigma'\sigma''$ s.th. $\langle \sigma, \sigma'\sigma'' \rangle \in S_{1,2}$. Insertions and deletions are defined as usual. Each restricted substitution, merge, split, and each insertion and deletion is called a *refined edit operation.*

Definition 3.2 Let $S_{1,1}$, $M_{2,1}$, and $S_{1,2}$ as above. Let $V, W \in \Sigma^*$. The *refined Levenshtein distance* between V and W is the minimal number of non-overlapping refined edit operations that are needed to transform V into W.

In what follows, by $d_{S_{1,1},M_{2,1},S_{1,2}}$ we denote the refined Levenshtein distance for the sets $S_{1,1}$, $M_{2,1}$, and $S_{1,2}$. The index is omitted if the sets are clear from the context. It is trivial to see that the refined Levenshtein distance between two words can be computed via dynamic programming, using a variant of the well-known Wagner-Fischer scheme [11].

By $d_{S_{1,1}}$ we denote the simplified version of the refined Levenshtein distance where $M_{2,1} = S_{1,2} = \emptyset$. In what follows we illustrate the universal Levenshtein automata $\mathcal{A}_n$ for the special metrics $d_{S_{1,1}}$, and we show how these automata are applied. Due to space limitations, the very technical computation of the automata $\mathcal{A}_n$ cannot be described here. At the end of the section we indicate the modifications that are necessary to cope with the general situation where the sets $M_{2,1}$ and $S_{1,2}$ may be non-empty.

$\mathcal{A}_n$ only depends on the maximum number of refined edit operations n that we want to tolerate, not on the choice of $S_{1,1}$. The automaton $\mathcal{A}_1$ for $d_{S_{1,1}}$ is shown in Figure 2. Given two words V and W, the set $S_{1,1}$ is used to calculate the actual input $i(V, W)$ for the automaton $\mathcal{A}_n$. $\mathcal{A}_n$ accepts $i(V, W)$ only when $d_{S_{1,1}}(V, W) \leq n$. We build $i(V, W)$ using the so-called characteristic vectors.

Definition 3.3 For each letter $c \in \Sigma$ and each word $a_1a_2...a_r \in \Sigma^*$ we define the *characteristic vectors*

1. $\chi(c, a_1a_2...a_r) = b_1b_2...b_r$, where $b_j = 1$ if $c = a_j$ and $b_j = 0$ otherwise,
2. $\chi_{sub}(c, a_1a_2...a_r) = f_1f_2...f_r$, where $f_j = 1$ if $\langle a_j, c \rangle \in S_{1,1}$ and $f_j = 0$ otherwise.

Definition 3.4 Given the two words $V, W \in \Sigma^*$ we define the input $i(V, W)$ for $\mathcal{A}_n$ as the sequence of pairs of characteristic vectors $\alpha_1\alpha_2...\alpha_{|W|}$ where $\alpha_i = \langle \beta_i, \beta_i^{sub} \rangle$ and

1. $\beta_i = \chi(W_i, V_{i-n}V_{i-n+1}...V_k)$, where $k = min(|V|, i + n + 1)$, $V_{-n+1} = \ldots = V_0 = \$$ for $n > 0$. Here $\$ \notin \Sigma$ is a new symbol.

2. $\beta_i^{sub} = \chi_s(W_i, V_{i-n+1}V_{i-n+1}...V_k)$, where $k = min(|V|, i+n-1)$, $V_{-n+2} = \ldots = V_0 = \$$ for $n > 1$.

For example, let $\Sigma = \{a, b, c, ..., z\}$ and $S_{1,1} = \{\langle a, d\rangle, \langle d, a\rangle, \langle h, k\rangle, \langle h, n\rangle\}$. Let $V = hahd$ and $W = hand$. Then $i(V, W) = \alpha_1\alpha_2\alpha_3\alpha_4$ where

$\alpha_1 = \langle \beta_1, \beta_1^{sub}\rangle = \langle \chi(h, \$hah), \chi_s(h, h)\rangle = \langle 0101, 0\rangle$,
$\alpha_2 = \langle \beta_2, \beta_2^{sub}\rangle = \langle \chi(a, hahd), \chi_s(a, a)\rangle = \langle 0100, 0\rangle$,
$\alpha_3 = \langle \beta_3, \beta_3^{sub}\rangle = \langle \chi(n, ahd), \chi_s(n, h)\rangle = \langle 000, 1\rangle$,
$\alpha_4 = \langle \beta_4, \beta_4^{sub}\rangle = \langle \chi(d, hd), \chi_s(d, d)\rangle = \langle 01, 0\rangle$.

The automaton $\mathcal{A}_1$ is shown in Fig. 2. In the figure, any x represents a don't care symbol that can be interpreted as 0 or 1, and expressions in round brackets are optional. For instance, from the state $\{I{-}1^{\#1}, I^{\#1}, I{+}1^{\#1}\}$ with $\langle 010(x), x\rangle$ we can reach the state $\{I^{\#1}\}$. This means that from $\{I{-}1^{\#1}, I^{\#1}, I{+}1^{\#1}\}$ we can reach $\{I^{\#1}\}$ with $\langle 010, 0\rangle$, $\langle 010, 1\rangle$, $\langle 0100, 0\rangle$, $\langle 0100, 1\rangle$, $\langle 0101, 0\rangle$ and $\langle 0101, 1\rangle$. In the above example we start from the initial state $\{I^{\#0}\}$; with input $\langle 0101, 0\rangle$, $\langle 0100, 0\rangle$, $\langle 000, 1\rangle$ and $\langle 01, 0\rangle$ we visit the states $\{I^{\#0}\}$, $\{I^{\#0}\}$, $\{I{-}1^{\#1}, I^{\#1}\}$, and $\{M^{\#1}\}$. $\{M^{\#1}\}$ is a final state. Hence $d_{S_{1,1}}(V, W) \leq 1$.

The universal automaton $\mathcal{A}'_n$ for the refined Levenshtein distance $d_{S_{1,1},M_{2,1},S_{1,2}}$ is similar to $\mathcal{A}_n$. The difference is that the input $i'(V, W)$ is a sequence of quadruples of bitvectors, because in addition to χ and χ_{sub} we use two other characteristic vectors - one for the restricted merge and one for the restricted split. The sets $M_{2,1}$ and $S_{1,2}$ are used for the computation of these two additional vectors respectively.

4 Using annotated error dictionaries

By an *error dictionary* ([1, 7]), we mean a collection $\mathcal{E}$ of strings that is generated from a set of correct words $\mathcal{D}$, systematically applying error patterns of a particular form. In an *annotated error dictionary* we store with each erroneous token the corresponding correct word and the transformation (here: edit operation) that was used to generate the entry. Since the same erroneous token might be produced in distinct ways the mapping from errors to correct words in general gives rise to ambiguities.

Generating annotated error dictionaries. The design of an error dictionary for estimating frequencies of edit operations in an erroneous text is a delicate matter. We first used a simple brute-force construction, based on a set of typical OCR errors. For some test sets, satisfactory results were obtained. For others, it turned out that some of the relevant error patterns were new, and results were disappointing. After a series of other tests we arrived at the following construction. (a) To the 100,000 most frequent words of our English dictionary we applied all substitutions, a selected list of "typical" merges and splits, and deletion of a letter i, l, t, or f. In addition, (b) to the 25,000 most frequent words we applied all (other) merges, and (c) to the 5,000 most frequent words we applied all (other) splits. Entries of the error dictionary $\mathcal{E}$ were produced by applying only one error transformation at one position. Hence, entries contain just one error. For each error W' obtained in this way, we only stored the retranslation that leads to the *most frequent correct* word W. At this point, when garbling a word with one of the non-typical merges and splits (cases b, c), we reduce its frequency by a penalty factor of 1/100. We deleted all erroneous tokens with a length ≤ 3 since acronyms and special names of length ≤ 3 can easily be misinterpreted as errors. We also excluded all errors that correspond to some word in a large collection of standard dictionaries with an overall sum of 3.2 million entries of various languages as well as of person and geographic names. Since each correct word that is left in the error dictionary may lead to a misclassification, it is important to use a collection of dictionaries with very high coverage. The error dictionary $\mathcal{E}$ produced in this way contains $38,794,294$ entries.

Applying annotated error dictionaries. Given the annotated error dictionary, $\mathcal{E}$, it can be used as follows to produce an "alignment list" $\mathcal{L}_{al}$ containing triples (W, W', op) from an erroneous text. The strings W' are the normal tokens found in the input text. A token is called "normal" if it is composed of standard letters only. If token W' represents an error found in $\mathcal{E}$, then its correction W and the applied edit operation $op = (\sigma \mapsto \sigma')$ are specified in $\mathcal{E}$ - (W, W', op) is added to $\mathcal{L}_{al}$. If W' is not found in $\mathcal{E}$ but in the base lexicon $\mathcal{D}$, W' is considered to be a correct token: we add $(W', W', \emptyset)$ to $\mathcal{L}_{al}$. Tokens that are found neither in $\mathcal{E}$ nor in $\mathcal{D}$ are discarded. This lexicon constraint balances the only partial lexical coverage of the error dictionary and also prevents erroneous tokens that are missing in $\mathcal{E}$ from being misinterpreted as correct words.

It is important to note that no groundtruth data is involved in this procedure. Furthermore the error dictionary is made for the purpose of OCR post-correction but is in no way adjusted to a certain OCR engine or document type.

Obtaining the set of restricted edit operations. From $\mathcal{L}_{al}$ we can easily compute estimated frequencies for all symbol dependent substitutions, merges and splits $f(\sigma \mapsto \sigma')$ as well as $f(\sigma)$ for all symbols and 2-grams. The score for each edit operation is its relative frequency: $f_{rel}(\sigma \mapsto \sigma') = f(\sigma \mapsto \sigma')/f(\sigma)$. By $f(\sigma)$, we denote the number of occurrences of σ in the left components W of entries (W, W', op) or $(W, W, \emptyset)$ of $\mathcal{L}_{al}$. Thresholds *subs*, *merge*, and *split* define the set of restricted substitutions, merges, and splits. A substitution $\sigma \mapsto \sigma'$ is only used as a restricted substitution during the selection of answer candidates if the estimated relative frequency of $\sigma \mapsto \sigma'$ is larger than *subs*. *merge* is the corresponding threshold for merges, *split* is the threshold for splits.

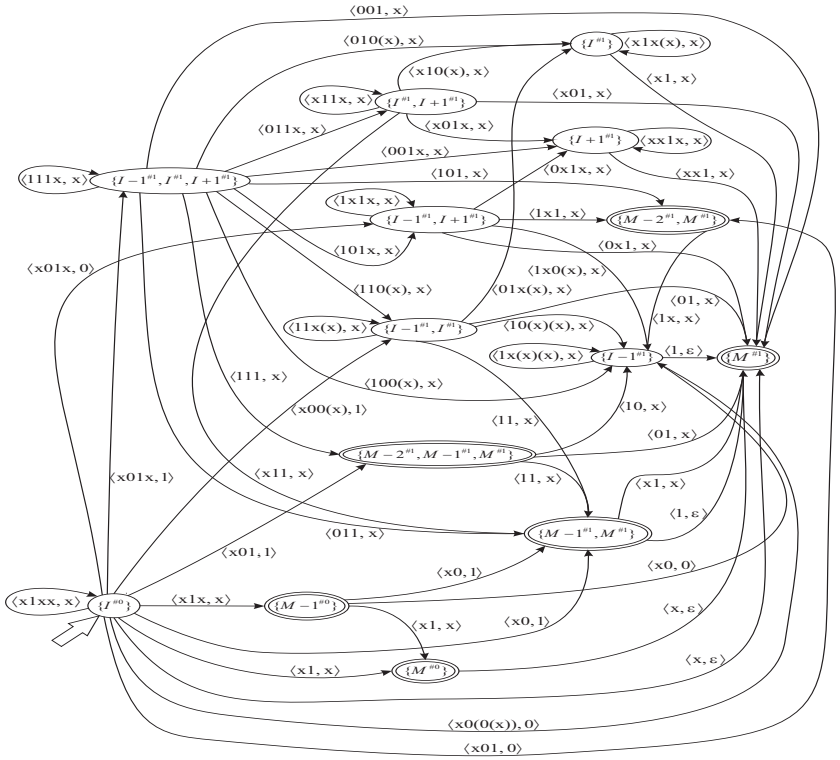

Fig. 2. Universal Levenshtein automaton $\mathcal{A}_1$ for refined Levenshtein metrics.

Implementation and efficiency aspects. The just under 40 million entries of the error dictionary are stored in a minimal deterministic finite-state automaton to grant a compact representation and very fast lookup times. In an offline step, it takes about 3 minutes and 30 seconds to compute the automaton with approx. 1.5 million states, its size in memory (including the annotations) is 513 MB. The estimation at runtime is basically a sequence of lookups for all normal tokens of the text and thus very fast. After loading the automaton, 1,000 tokens of text are processed in $\sim$ 35ms.

5 Evaluation results

As a test set for the new method we used the TREC-5 corpus with character error rate 5% (http://trec.nist.gov/data/t5_confusion.html) and an English dictionary with 264,061 words. Via dynamic programming we extracted from the corpus couples of the type $\langle pattern, original \rangle$, where *pattern* is an OCRed word and *original* is its corresponding original. The problem of case errors was ignored for these tests.
From one half of the test set, we extracted the couples $\langle pattern, original \rangle$ where $pattern \neq original$ and where *original* was found in the dictionary. This subset represents all recognition errors that can possibly be corrected with the applied dictionary. For each *pattern* we selected a set of correction candidates from the dictionary.
To have a baseline comparison, we first made experiments with Levenshtein automata for the standard Levenshtein distance (lines (a) in Table 1) and automata allowing unrestricted substitutions, merges and splits (lines (b)). We then repeated the tests with restricted edit operations. The sets of allowed operations were determined by the thresholds [2] given in Table 1 and relative frequencies derived either from supervised training (lines (c)) or from an error profile obtained with error dictionaries as described above (lines (d)). Only for (c) we used the remaining half of the test set for supervised training.

In Table 1, n denotes the maximal number of (restricted) edit operations. For $n = 1$, we used $100,000$ test patterns for evaluation. For $n = 2$, the number of test patterns is $50,000$. For $n = 3$, the number of test patterns is $23,000$. With *len*, we denote the possible length for an input pattern. In practice, a distance bound $n = 1$ ($n = 2$, $n = 3$) is mainly interesting for words of length $l \leq 6$ ($7 \leq l \leq 12$, $13 \leq l$), which explains our focus. The number *cand* gives the average number of correction candidates per pattern

[2] The thresholds are standard values that proved suitable for the corpus. Future experiments must show if the values are stable for texts of a different kind.

		subs	*merge*	*split*	*cand*	*recall*	*time*
$n = 1$, $len = 1 - 6$							
(a)	standard	0	1	1	7.78	70.565%	0.032 ms
(b)	unrestricted	0	0	0	45.73	94.52%	0.107 ms
(c)	training	0.0006	0.0325	0.0005	5.48	94.519%	0.049 ms
(d)	err.dics.	0.0005	0.01	0.0005	6.85	94.519%	0.068 ms
$n = 2$, $len = 7 - 12$							
(a)	standard	0	1	1	12.24	95.178%	0.37 ms
(b)	unrestricted	0	0	0	96.88	99.988%	2.448 ms
(c)	training	0.01	0.0002	0.0004	3.8	99.986%	0.487 ms
(d)	err.dics.	0.01	0.02	0	18.73	99.966%	1.67 ms
$n = 3$, $len > 12$							
(a)	standard	0	1	1	4.1	99.81%	1.214 ms
(b)	unrestricted	0	0	0	43.971	100%	6.623 ms
(c)	training	0.04	0.03	1	4.029	100%	1.08 ms
(d)	err.dics.	0.01	0.01	1	4.029	100%	1.08 ms

Table 1. Test for distance bounds $n = 1, 2, 3$.

found in the dictionary. The value of *cand* depends on the distance bound, the length of the input pattern and the thresholds for the substitutions, merges and splits. With *time*, we denote the average time per pattern needed to find and output all correction candidates. With *recall*, we denote the percentage of patterns where the correct *original* is found in the selected set of answer candidates.

As lines (d) show, the new metrics based on the estimation with error dictionaries lead to substantially improved results: For $n = 1$ and also for $n = 3$, the candidate sets obtained with standard Levenshtein distance were larger but lead to poorer recall. When we allowed unrestricted substitutions, merges and splits, the recall was basically the same, but the extraction is much slower and the size of the candidate sets is unacceptably high. For $n = 2$, a significantly higher recall (99.97%, compared to 95.18% with standard Levenshtein distance) is obtained at the cost of larger candidate sets: 18.73 (as compared to 12.24) candidates on average. Only for $n = 2$ the estimation with error dictionaries was clearly outperformed by supervised training with ground truth data where the candidate sets are much smaller (3.8 on average). This might be due to an important edit operation that went unnoticed by the automatic error profiling.

The CPU of the machine used for the experiments is Pentium 4, 2.4 GHz with 1GB RAM.

6 Conclusion and future work

In this paper we showed that error dictionaries may be used to compute profiles for erroneous input texts that help to fine-tune the selection of correction candidates for malformed tokens from a dictionary. The computation of the profile for a given input text is simple, fast, and completely automatic. Combining this technique with a recently introduced method for accessing dictionaries, we obtain an automated procedure for candidate selection which is adaptive in the sense that the specifics of the input text are taken into account. The method leads to very small candidate sets with high recall.

Acknowledgements. This work was supported by AICML, iCore, DFG and VolkswagenStiftung.

References

[1] A. Arning. *Fehlersuche in großen Datenmengen unter Verwendung der in den Daten vorhandenen Redundanz.* PhD thesis, University of Osnabrück, 1995.

[2] K. Kukich. Techniques for automatically correcting words in texts. *ACM Computing Surveys*, pages 377–439, 1992.

[3] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.*, 1966.

[4] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, December 2004.

[5] O. Owolabi and D. McGregor. Fast approximate string matching. *Software Practice and Experience*, 18(4):387–393, 1988.

[6] C. Ringlstetter, U. Reffle, A. Gotscharek, and K. U. Schulz. Deriving symbol dependent edit weights for text correction - the use of error dictionaries. In *Proceedings of the ninth International Conference on Document Analysis and Recognition (ICDAR)*, page to appear, 2007.

[7] C. Ringlstetter, K. U. Schulz, and S. Mihov. Orthographic errors in web pages: Towards cleaner web corpora. *Computational Linguistics*, 32(3):295–340, September 2006.

[8] K. U. Schulz and S. Mihov. Fast String Correction with Levenshtein-Automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85, 2002.

[9] K. U. Schulz, S. Mihov, and P. Mitankin. Fast selection of small and precise candidate sets from dictionaries for text correction tasks. In *Proceedings of the ninth International Conference on Document Analysis and Recognition (ICDAR)*, page to appear, 2007.

[10] C. Strohmaier, C. Ringlstetter, K. U. Schulz, and . Mihov. A visual and interactive tool for optimizing lexical postcorrection of OCR results. In *Proceedings of the IEEE Workshop on Document Image Analysis and Recognition, DIAR'03*, 2003.

[11] R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[12] J. Zobel and P. Dart. Finding approximate matches in large lexicons. *Software Practice and Experience*, 25(3):331–345, 1995.