Proceedings of the Federated Conference on Computer Science and Information Systems pp. 527–532
DOI: 10.15439/2016F311
ACSIS, Vol. 8. ISSN 2300-5963

Towards increasing F-measure of approximate string matching in O(1) complexity

Adrian Boguszewski, Julian Szymański, Karol Draszawka
Department of Electronic, Telecommunication and Informatics
Gdańsk University of Technology
Gdańsk, Poland
E-mail: {adrbogus1, julian.szymanski, kadr}@eti.pg.gda.pl

***Abstract*—The paper analyzes existing approaches for approximate string matching based on linear search with Levenshtein distance, AllScan and CPMerge algorithms using cosine, Jaccard and Dice distance measures. The methods are presented and compared to our approach that improves indexing time using Locally Sensitive Hashing. Advantages and drawbacks of the methods are identified based on theoretical considerations as well as empirical evaluations on real-life dictionaries.**

***Index Terms*—approximate string matching, misspelling correction, LSH, CPMerge, AllScan, Levenshtein distance, neural net indexer**

I. Introduction

AN APPROXIMATE string matching is a task of finding a specific word in a given dictionary, that is similar to the word provided by the user to at least a certain degree.

This task is encountered in a wide spectrum of applications, especially for calculating similarity between texts [1]. One of the examples is misspelling detection in a written text and recommendation for a correct word. This solution is used in most interfaces where a user enters a text, e.g.: in web search engines, while she or he enters a query, or in mobile phones, while typing messages. Among other examples, there is plagiarism detection [2], [3] or spam filtering [4], that checks whether a text contains words intentionally modified in order to evade naive spam filters, e.g. vulgar words with added dots instead of some letters in internet posts.

The typical approaches for approximate string matching use some kind of edit distance, such as Levenshtein distance [5]. The two words are matched, if an edit distance between them is below a specified threshold. A useful algorithm should then return a number (possibly all) of words from a predefined dictionary that approximately match a given (input) word in a reasonable time.

In this paper we compare four different approaches to accomplish this, reporting their processing time as well as quality of matched words returned: linear search with Levenshtein distance, AllScan and CPMerge algorithms proposed in [6], and an approach based on Locality Sensitive Hashing. We identify their advantages and drawbacks.

The paper is constructed as follows. Section II describes Levenshtein, AllScan and CPMerge algorithms. Section III presents the approach based on Locally Sensitive Hashing. The experiments and the results of their evaluation in the approximate string matching task are provided in Section IV. Finally, Section V contains conclusions and an idea of further improvements.

II. Approximate string matching algorithms

At the very beginning we define basic symbols we use to describe the algorithms:

1) Σ - an alphabet, a finite set of symbols (letters)
2) Σ^* - a set of all possible words over Σ (e.g. a, b, ..., aa, ab, ...)
3) $V \subset \Sigma^*$ - a finite size dictionary
4) $sim(x, y)$ - a similarity function between words x and y – $f: \Sigma^* \times \Sigma^* \rightarrow [0, 1]$
5) $\alpha \in [0, 1]$ - similarity threshold
6) $|x|$ - length of word x
7) $|X|$ - the number of elements in set X

In general, a search for similar words, can be defined as constructing a set $Y_{x,\alpha}$ of certain words y from dictionary V, for which the similarity to a given word x is greater or equal to α:

$$Y_{x,\alpha} = \{y \in V | sim(x, y) \geq \alpha\}. \tag{1}$$

A. Linear search with Levenshtein distance

The most popular approaches for approximate string matching use edit distance. In general, an edit distance measures two strings dissimilarity by counting how many edit operations are required to change one word into the other. Levenshtein distance considers three single character modifications: insertion, deletion and substitution [7]. Each of them has an equal cost of 1.

Formally, Levenshtein distance $lev(x, y)$ between two words x and y of lengths $|x|$ and $|y|$, is equal to $lev_{x,y}(|x|, |y|)$, where $lev_{x,y}(i, j)$ is a discrete function of two non-negative arguments defined as:

$$lev_{x,y}(i,j) = \min \begin{cases} lev_{x,y}(i-1,j)+1 \\ lev_{x,y}(i,j-1)+1 \\ lev_{x,y}(i-1,j-1)+1_{(x_i \neq y_j)} \end{cases}, \tag{2}$$

if $\min(i, j) \neq 0$,

$$lev_{x,y}(i,j) = \max(i,j), \quad \text{if} \min(i,j) = 0.$$

For example, the distance between words *written* and *writes* equals 2, because we need two modifications to transform first

string into the second one:
written $\rightarrow$ *writen* (deletion of *t*)
writen $\rightarrow$ *writes* (substitution of *s* for *n*)

In order to make use of (1) explicitly, a Levenshtein-derived similarity measure can be defined:

$$sim_{lev}(x, y) = \frac{\max(|x|, |y|) - lev(x, y)}{\max(|x|, |y|)}. \quad (3)$$

Because we need to browse the whole dictionary, the complexity of this algorithm is $O(n)$ where n denotes number of objects in the dictionary in term of dictionary size. This is the main main disadvantage of this straightforward approach. On the other hand, this method provides all matching words in the result, so that no one similar word is ever missed. Another useful property it has is that there is no need for preprocessing of the dictionary, that may require additional computations and storage space, which will be the case for methods described next. The approach finds many modifications e.g.: Damerau-Levenshtein distance [8] or others [9], [10], but the complexity usually remains linear. Thus, in experiments we use the basic Levenshtein implementation as a baseline.

B. Shingle word representation

The rest of compared algorithms for approximate string matching do not use edit distance to compare strings in their raw form. Instead, they all incorporate a preprocessing step, called *shingling* [11], which converts a string into a set of *n*-grams. The similarity of two strings are then determined indirectly by computing the similarity between the corresponding sets of *n*-grams.

In our experiments, we split words into letter trigrams and call them *features* that represent a particular word. For example, a word `rotation` is represented by a set `{$$r, $ro, rot, ota, tat, ati, tio, ion, on$, n$$}`, where `$` sign denotes 'no letter'.

It should be noticed that such a tri-gram representation is chosen arbitrarily and can be further improved [12]. However, to compare the approaches of approximate string matching we decided to stick to one fixed representation.

There is a number of similarity measures between feature sets than can be used for shingle-based strings representation. One of the most popular is cosine similarity, defined by (4).

$$sim_{cos}(x, y) = cos(X, Y) = \frac{|X \cap Y|}{\sqrt{|X||Y|}}, \quad (4)$$

where X and Y are *feature sets* (containing $|X|$ and $|Y|$ elements) of x and y respectively .

Other measures, like Jaccard or Dice distance, can also be successfully applied here [13]. They are defined by (5) and (6) respectively.

$$sim_{jacc}(x, y) = jaccard(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \quad (5)$$

$$sim_{dice}(x, y) = dice(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|} \quad (6)$$

To provide an illustrative example let us assume two words: x = `rotation` and y = `aviation`. We then have X = `{$$r, $ro, rot, ota, tat, ati, tio, ion, on$, n$$}` and Y = `{$$a, $av, avi, via, iat, ati, tio, ion, on$, n$$}`. Hence $|X| = 10, |Y| = 10$. The cosine similarity of these words is equal to $cos(X, Y) = \frac{5}{\sqrt{10*10}} = 0.5$. Jaccard and Dice distances are as follows: $jaccard(X, Y) = \frac{5}{15} = 0.33$, $dice(X, Y) = \frac{2*5}{10+10} = 0.5$

C. AllScan algorithm

The AllScan algorithm first shingles word x obtaining a feature set X. All words in the vocabulary must also be shingled accordingly in a preprocessing phase. Then, AllScan computes lower and upper bounds for *length* of word y potentially similar to x to at least α. For cosine similarity, this bounds are determined by inequality:

$$\lceil \alpha^2 |X| \rceil \leq |Y| \leq \left\lfloor \frac{|X|}{\alpha^2} \right\rfloor. \quad (7)$$

Inequality (7) comes from (1) and (4), after observing that for minimal length of y we have $|X \cap Y| = |Y|$, while for maximal length of y, it is $|X \cap Y| = |X|$.

To give an example, if we assume $\alpha = 0.7$ and x = `rotation`, $|X| = 10$. Therefore Y set size must be between 5 and 20, because $|Y| \geq \lceil 0.7^2 * 10 \rceil \geq 5$ and $|Y| \leq \lceil \frac{10}{0.7^2} \rceil \leq 20$

Additionally, there is a minimum *overlap* τ value defined for each possible word length from (7). The τ means minimal value of the same letter trigrams, which have to occur in both strings to exceed the similarity threshold value α. For cosine similarity, it is therefore:

$$\tau = \left\lceil \alpha \sqrt{|X||Y|} \right\rceil. \quad (8)$$

Let us assume $\alpha = 0.7$ and words `rotation` and `aviation` ($|X| = |Y| = 10$). Equation (8) gives 7. Hence the feature sets of the words have to contain at least seven shingles in common to be similar in 0.7. They have only four such shingles, so they do not satisfy the minimal overlap value.

Based on obtained $|Y|$ and τ values, the algorithm retrieves all words from the dictionary that satisfy a given matching criterion, i.e. it returns all the words, similarities of which exceed the threshold.

The main advantage of this approach is the retrieval of all matching words (that satisfy given similarity measure), but the problem is its performance. Although the search space is reduced due to y length bounds calculation, the overall complexity of AllScan algorithm is still $O(n)$ in term of dictionary size, both preprocessing the dictionary and searching.

D. CPMerge

CPMerge algorithm [6] extends AllScan by reduction of the dictionary that is searched during single retrieval. The improvement limits the size of the dictionary, removing words which certainly are not a result. It uses Property 1 to

determine which words are candidates to be in a result set. The result set is much smaller than the whole dictionary and it causes significant speedup.

Property 1 *Let there be a set X (of size $|X|$) of word x n-grams and a set (of any size) Y of word y n-grams. Consider any subset $Z \subseteq X$ of size $(|X| - \tau + 1)$. If $|X \cap Y| \geq \tau$, then $Z \cap Y \neq \varnothing$.*

Assume $x = \texttt{rotation}$ ($|X| = 10$) and word y with length 6 ($|Y| = 8$). If $\alpha = 0.5$ then $\tau = 5$ (from (8)). Hence, $|Z|$ must be 6 and if $|X \cap Y| \geq 5$ then $Z \cap Y$ must have at least one element.

As previous algorithms, CPMerge returns all matching words as a result. It is faster than AllScan algorithm, due to described improvements, but its complexity is still $O(n)$. Full explanation is available in [6].

III. LSH-BASED APPROACH

To improve the efficiency of dictionary indexing we propose an approach based on special type of constructing the hash indexes. The idea is to construct such an index that works in opposite way to MD5 signature [14]. If the source strings differ slightly, the output hash would be also modified slightly. One way to create such hash function can be based on Locality-Sensitive Hashing [15] used to reduce dimensionality of high-dimensional data. It can be used in many applications where nearest neighbors need to be effectively computed [16], such as in tasks of entity resolution, fingerprint comparison or finding similar documents.

The algorithm takes data and computes a hash, which is a lower-dimensional representation of a given input. The result must preserve similarity, i.e. if words are similar then their hashes must be similar as well. In contrast to conventional hashing functions, LSH tries to maximize probability of a collision for similar items.

The main goal of incorporating LSH idea to approximate string matching is to significantly improve time performance. The LSH-based approach to this task consists of three phases:

1) Shingling (described above)
2) Min-Hashing – converts large sets to short signatures, while preserving similarity
3) Locality-Sensitive Hashing – places similar words into the same bucket

Firstly, an empty set C of shingles is created. During shingling of each word from the dictionary, shingles are added to collection C, so that after this step C is a sorted set of all shingles that occur in the shingled dictionary. Every trigram has a corresponding unique number (index of the shingle in C).

Secondly, we take the indices of a word shingles and store them in a vector. This vector representation is an input for Min-Hash algorithm, which calculates the signature. It should be noticed that different strategies can be used here [17]. We used the most popular version of Min-Hash algorithm.

Min-Hash internally creates occurrence matrix (of size #shingles $\times$ #words) filled with ones at positions where a given string contains a specific shingle. An example of such a matrix is shown in Table I. Then, rows of the matrix are permuted n times and at every permutation, for each word (each column) an index of the first row containing 1 is saved. In result, each word has a signature, which is a vector of these indices.

TABLE I
SAMPLE OCCURRENCE MATRIX

	the	those	these
$$t	1	1	1
$th	1	1	1
the	1	0	1
he$	1	0	0
e$$	1	0	0
tho	0	1	0
hos	0	1	0
ose	0	1	0
se$	0	1	1
e$$	0	1	1
hes	0	0	1
ese	0	0	1

In the third step, we partition these signatures into b bands. Every band is hashed, using locality sensitive hashing, into one of k buckets. In this case, hash function is of the form: $f: \mathbb{Z}^{\lfloor \frac{n}{b} \rfloor} \rightarrow \mathbb{Z}$

We chose $n = 100, b = 20$ and we want each bucket contains about one hundred hashes, so k must be $\frac{dict_size}{100}$.

Candidate words are in buckets, which contain at least one hash from word typed by user. Then, a similarity between given hash and all hashes in bucket is calculated. If the similarity exceeds the threshold, a string associated with the hash is added to the result list.

The complexity of LSH searching is sub-linear, better than that of previous approaches, although worse than O(1). In contrast to previously described algorithms, LSH has a big disadvantage – it does not guarantee that all matching words are in the result set. Bigger dictionary can cause worse results [18].

IV. EXPERIMENTS

In our experiments, we measured processing time depending on various settings of the compared algorithms. In the case of LSH algorithm, we also measured the quality of obtained results in terms of recall and precision.

In every test, we randomly choose words from polish dictionary for games containing over 2,700,000 words, taking into account words with length less or equal to 15. The dictionary is available online [19].

The processing times were measured ten times. Averages of them are reported below.

A. The time of constructing the search structure

At the very beginning, we tested the time of building searching structures needed for algorithms and how it depends on the number of words in a dictionary. The time for Levenshtein and CPMerge algorithms are the same as in the case of AllScan, because data preparation process is exactly the same.

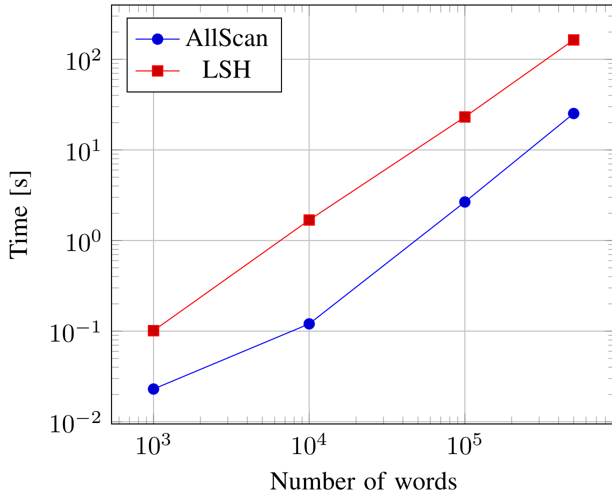

Fig. 1. Building time depending on the number of words

As we can see in Fig. 1, the time grows linearly in relation to the number of words with LSH having bigger coefficient.

B. Searching time

1) Dictionary size: The second test measured the search time of algorithms depending on the number of words in the dictionary. In this experiment we assumed the following values:
distance (allscan, cpmerge) = cosine
length of a given word ($|x|$) = 10
similarity threshold (α) = 0.7

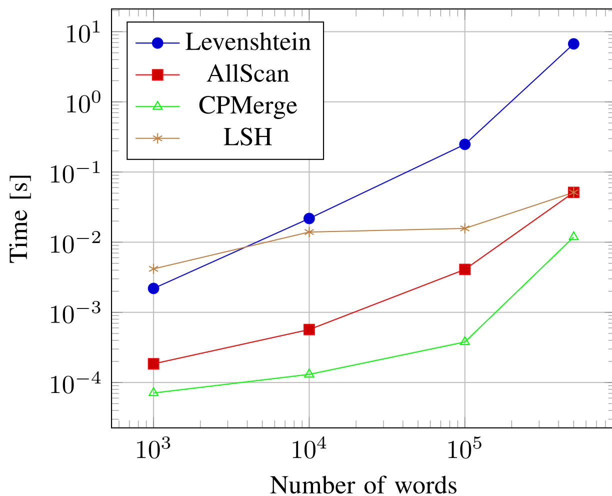

Fig. 2. Searching time depending on the number of words

Fig. 2 shows that the fastest algorithm is CPMerge. However, its search time grows squarely, what in comparison to LSH sub-linear growth allows us to state that LSH should be faster for big data.

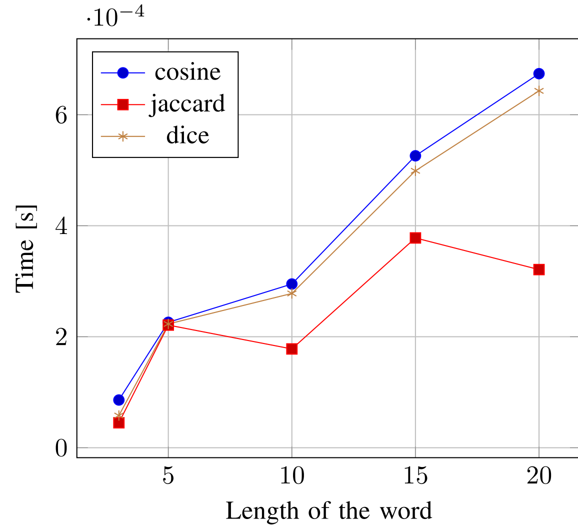

Fig. 3. Searching time depending on the length of a given word

2) Length of given words: To evaluate search time in the function of the length of given words we assumed the following values:
algorithm = cpmerge
number of words (V) = 100 000
similarity threshold (α) = 0.7
We measured the results for all three distance metrics.

Fig. 3 shows that processing time is the smallest for Jaccard distance, as the range of candidate words is the narrowest. All times grows linearly. Subtle deviations are caused by the number of candidate words changing stepwise in relation to word length.

3) Similarity threshold value: To evaluate search time depending on similarity threshold and distance metric we assumed:
searching algorithm = cpmerge
number of words (V) = 100 000
length of given words ($|x|$) = 15

As can be seen in Fig. 4, time falls in relation to similarity. This is because the number of candidate words is decreasing. The algorithm is the fastest for Jaccard distance due to the same reason as above, i.e. the smallest set of candidate words.

C. Number of words found

In this experiment we tested the impact of α similarity threshold on the number of returned similar strings. CPMerge algorithm had the following settings:
number of words ($|V|$) = 100 000
length of given words ($|x|$) = 10
Every given word had a typo in order to search for similar, but not identical, word.

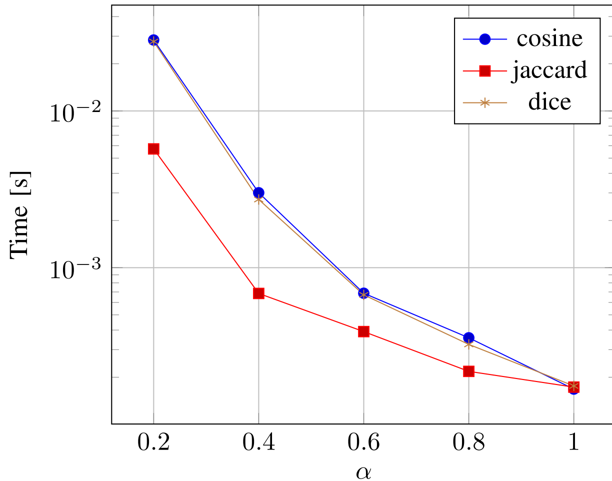

Fig. 4. Searching time depending on similarity threshold

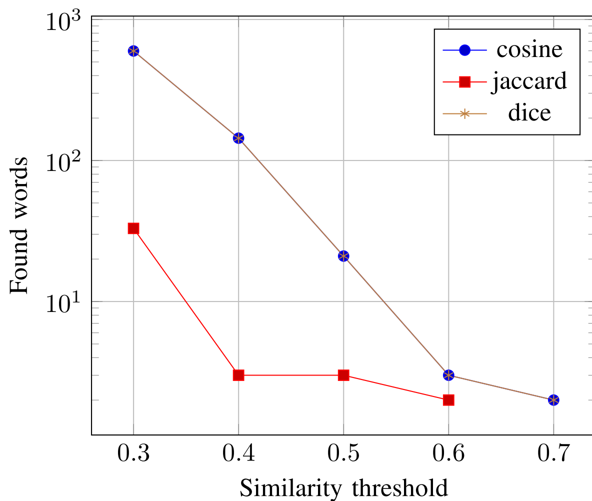

Fig. 5. Number of found word depending on similarity threshold

Fig. 5 shows that the number of found words falls exponentially in relation to assumed similarity threshold. As previously noticed, Jaccard distance returns the smallest set of candidates.

D. Recall and precision of LSH algorithm

We used the following measures for evaluating the quality of the approximate string matching of LSH algorithm: $recall = \frac{found\&relevant}{relevant}$ and $precision = \frac{found\&relevant}{found}$. Recall indicates what fraction of all matching words is returned in the result. Precision indicates what is the fraction of correct matches in the result set. In this task it is beneficial to maximize both recall and precision, therefore we also show their balanced harmonic mean, i.e. F-measure: $F_1 = \frac{2*P*R}{P+R}$.

1) Dictionary size impact: We assumed:
number of hash functions (n) = 100
number of bands (b) = 20
similarity threshold (α) = 0.7
number of buckets (k) = $\frac{number_of_words}{100}$

TABLE II
LSH RECALL AND PRECISION DEPENDING ON DICTIONARY SIZE

	100	1000	10000	100000
relevant	89	93	93	105
found	76	69	48	48
precision	1	1	1	1
recall	0.85	0.74	0.52	0.46

2) Similarity threshold impact: We assumed the following values:
number of words (V) = 100 000
number of hash functions (n) = 100
number of bands (b) = 20
number of buckets (k) = 1000

TABLE III
LSH RECALL AND PRECISION DEPENDING ON SIMILARITY THRESHOLD

	0.4	0.5	0.6	0.7
relevant	10911	1806	261	103
found	453	189	91	46
precision	1	1	1	1
recall	0.04	0.10	0.35	0.45

Table III shows that recall for low threshold is very low, which means LSH algorithm, in this configuration, is almost useless.

E. Wikipedia editors misspellings correction

In the following two tests we used a list of misspellings made by Wikipedia editors [20].

1) Comparison of algorithms: We set: number of words (V) = 1922
number of misspellings = 2455
similarity threshold (α) = 0.5
distance (cpmerge) = cosine
number of hash functions (lsh: n) = 100
number of bands (lsh: b) = 20
number of buckets (lsh: k) = 19

TABLE IV
COMPARED ALGORITHMS AT WIKI TYPOS CORRECTION TASK

	Levenshtein	AllScan & CPMerge	LSH
recall	0.89	0.89	0.74
precision	0.56	0.48	0.64
F-measure	0.69	0.63	0.69

Allscan results are the same like results of CPMerge algorithm, because they differ only in the processing time.

2) Similarity threshold impact on CPMerge: The setting were:

searching algorithm = CPMerge
distance (cpmerge) = cosine
number of words (V) = 1922
number of misspellings = 2455

TABLE V
CPMERGE AT WIKI TYPOS CORRECTION TASK DEPENDING ON SIMILARITY MEASURE

	0.2	0.3	0.4	0.5	0.6	0.7
recall	0.99	0.98	0.94	0.89	0.81	0.67
precision	0.03	0.12	0.29	0.48	0.64	0.63
F-measure	0.06	0.21	0.44	0.63	0.71	0.65

V. CONCLUSIONS

The most straightforward approach to approximate string matching is to incorporate some edit distance, like Levenshtein. Nevertheless, searching time of this method is the highest, because it has to browse the whole dictionary. $O(n)$ complexity disqualifies this approach for big dictionaries. AllScan and CPMerge reduces the size of the set of words to process, so they are faster than Levenshtein-based approach. However, their processing time still very much depends on dictionary size. All three methods have a significant advantage – they correctly (depending on similarity measure in use) return all words similar to a given string within assumed margins.

On the other hand, LSH provides results in acceptable time, but it degrades recall value. This is the main disadvantage of the algorithm, which causes that many similar words are not in the returned list.

We wish to have a solution, that is fast and that provides recall on very high level. To reach this goal we propose and test an initial solution that employs feed-forward neural network as indexer. The network takes a word as input and returns its index in the dictionary. With constant searching time, recall above 97% and high precision, this method can potentially beat all algorithms compared in this article in the task of approximate string matching. The idea extending the approach based on feedforward network is to use a denoising autoencoder. The approach based on on auto-associative network reconstructs its input given at the output layer through a bottleneck hidden layer, while the input additionally contains some noise, i.e. misspelled words in this case. In this approach, we would learn a network using one exact word and some words with typos and expect a correct word to be reconstructed.

REFERENCES

[1] W. H. Gomaa and A. A. Fahmy, "A survey of text similarity approaches," *International Journal of Computer Applications*, vol. 68, no. 13, 2013.
[2] A. Parker *et al.*, "Computer algorithms for plagiarism detection," 1989.
[3] Z. Su, B.-R. Ahn, K.-Y. Eom, M.-K. Kang, J.-P. Kim, and M.-K. Kim, "Plagiarism detection using the levenshtein distance and smith-waterman algorithm," in *Innovative Computing Information and Control, 2008. ICICIC'08. 3rd International Conference on*. IEEE, 2008, pp. 569–569.
[4] F. D. Garcia, J.-H. Hoepman, and J. Van Nieuwenhuizen, "Spam filter analysis," in *Security and Protection in Information Processing Systems*. Springer, 2004, pp. 395–410.
[5] T. Okuda, E. Tanaka, and T. Kasai, "A method for the correction of garbled words based on the levenshtein metric," *Computers, IEEE Transactions on*, vol. 100, no. 2, pp. 172–178, 1976.
[6] N. Okazaki and J. Tsujii, "Simple and efficient algorithm for approximate dictionary matching," *Proceedings of the 23rd International Conference on Computational Linguistics*, pp. 851–859, 2010.
[7] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*. Soviet Physics Doklady, 1966.
[8] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.
[9] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric," in *Proceedings of the fifth Australasian symposium on ACSW frontiers-Volume 68*. Australian Computer Society, Inc., 2007, pp. 117–124.
[10] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.
[11] K. Monostori, R. Finkel, A. Zaslavsky, G. Hodász, and M. Pataki, "Comparison of overlap detection techniques," in *Computational Science—ICCS 2002*. Springer, 2002, pp. 51–60.
[12] C. Badica, N. T. Nguyen, and M. Brezovan, Eds., *Computational Collective Intelligence. Technologies and Applications - 5th International Conference, ICCCI 2013, Craiova, Romania, September 11-13, 2013, Proceedings*, ser. Lecture Notes in Computer Science, vol. 8083. Springer, 2013.
[13] S.-S. Choi, S.-H. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *Journal of Systemics, Cybernetics and Informatics*, vol. 8, no. 1, pp. 43–48, 2010.
[14] B. Kaliski and M. Robshaw, "Message authentication with md5," *CryptoBytes (RSA Labs Technical Newsletter)*, vol. 1, no. 1, 1995.
[15] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, no. 6, 1999, pp. 518–529.
[16] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors [lecture notes]," *Signal Processing Magazine, IEEE*, vol. 25, no. 2, pp. 128–131, 2008.
[17] L. Paulevé, H. Jégou, and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognition Letters*, vol. 31, no. 11, pp. 1348–1358, 2010.
[18] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2014.
[19] SJP, http://sjp.pl/slownik/growy/, 2016, [Online; 01-07-2016].
[20] Wikipedia, https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings, 2016, [Online: 01-07-2016].