Postprint of: Olewniczak S., Szymański J. (2021) Fast Approximate String Search for Wikification. In: Paszynski M., Kranzlmüller D., Krzhizhanovskaya V.V., Dongarra J.J., Sloot P.M. (eds) Computational Science – ICCS 2021. ICCS 2021. Lecture Notes in Computer Science, vol. 12744. Springer, Cham. https://doi.org/10.1007/978-3-030-77967-2_29

Fast Approximate String Search for Wikification

Szymon Olewniczak[0000−0002−9387−8546]
Julian Szymański[0000−0001−5029−6768]

Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology, Poland
{szymon.olewniczak,julian.szymanski}@eti.pg.edu.pl

Abstract. The paper presents a novel method for fast approximate string search based on neural distance metrics embeddings. Our research is focused primarily on applying the proposed method for entity retrieval in the Wikification process, which is similar to edit distance-based similarity search on the typical dictionary. The proposed method has been compared with symmetric delete spelling correction algorithm and proven to be more efficient for longer stings and higher distance values, which is a typical case in the Wikification task.

Keywords: information retrieval · neural embeddings · edit distance · convolutional neural networks · approximate matching

1 Introduction

The goal of our research is to build an efficient method for Wikification [12], a process of creating links between arbitrary textual content and Wikipedia articles. The links must be relevant to the context of the processed content what is a non-trivial task. Wikification might be considered as a variant of the more general task of Entity Linking.

The Entity Linking systems usually split the process into two stages [10]: entity retrieval and entity disambiguation. In the first stage, the goal is to extract all candidates from content i.e. spans of text that might be linked to our knowledge-base (Wikipedia article's base in case of Wikification). In the second stage among all candidates only the relevant ones, that make sense in the context of the processed text, are selected.

Wikipedia is composed of many articles and covers a large number of topics from different disciplines. Carefully implemented Wikification tool can support many crucial information retrieval tasks such as text representation that is the basis of achieving good results of text classification. It makes Wikification a very important problem among other NLP tasks.

The first stage of Wikification - an entity retrieval may be considered as a specific variant of the Named Entity Recognition (NER) problem. We called it extended NER because we consider here not only named entities but also some common nouns and phrases. One of the possible approaches to this problem is to extract all links' labels from Wikipedia articles and use them as a dictionary of possible entities.

To further improve the quality of entity retrieval, we can adopt an approximate string matching for our dictionary of possible entities. The goal of approximate string search is for a given query string q, we retrieve a dictionary element or elements that is the most similar to q according to some metric.

There are many reasons to adopt this strategy for entity retrieval. First, the terms in source text might be misspelled what is even more common for rare named entities. Second, the words in a phrase may be compounded (some spaces might be omitted). Third, the words in phrases can have slightly different variants regarding their position in a sentence, which is common in many languages. Fourth, the words in phrase might be reordered, for example, name and surname may be swapped in an entity describing a person, without changing the meaning of the entity.

2 Approximate string search

Formally, approximate string search is defined as a task of retrieving elements from dictionary D that are similar to query string q, according to a given metric $dist(.)$. We will denote the set of retrieved elements as X. The dictionary elements and query string are sequences constructed from some finite alphabet A.

There are two variants of approximate string search. The first is called a radius based nearest neighbors search (rNN). In this variant we receive all the terms X from the D which satisfy the condition for some predefined r:

$$\forall x \in X : dist(q, x) \leq r \tag{1}$$

Second, called k-nearest neighbours search (kNN) retrieves k nearest neighbours for a query q such as:

$$\forall d \in D \setminus X : dist(q, d) \geq max(dist(q, X)) \tag{2}$$

In the entity retrieval task, we are rather interested in the best match rather than all possible matches for two reasons. First, we don't want to create too many possibilities for the entity disambiguation stage. Second, there is usually one correct match that the user really meant. Thus in our research, we use kNN variant for approximate string search.

The most common distance metric for approximate string search is edit distance. Its simplest variant is called Levenshtein distance, where we count the smallest possible number of basic transformations that are required in order to transform one string into another. There are three kinds of transformation defined in Levenshtein distance. The first is insertion which means inserting an additional character on a selected position. The second is deletion which means deleting a selected charter from a string and the last one is substitution which replaces one character with the other.

For misspellings correction, an extended variant of Levenshtein distance called Damerau-Levenshtein is commonly used [3]. In this method, we allow the additional transformation of text called transposition which means swapping two

adjacent characters in a string. This modified metric is known to better represent misspellings occurring on real data.

The problem with edit distance metrics is that calculating them between two strings is computationally costly. The best currently known algorithm of computing the distance between two strings has $O(n^2)/log(n)$ complexity[8]. In addition, if the strong exponential time hypothesis (SETH) is true, the distance cannot be calculated in time lower than $O(D^{2-\gamma})$ for any fixed $\gamma > 0$.

Another problem with edit distance metrics is that they are not perfect when it comes to real data. For example, in our experiments on Wikipedia's List of common misspellings dataset [16] and the dictionary from SymSpell project [4], the Damerau-Levenshtein resulted in 86% of correct matches, while the Levenshtein metric over the same data achieved 79% accuracy. There are several reasons for the mismatches, where among others there is the fact that similarly sounding phones are more easily mistaken.

Additionally, from the entity retrieval point of view, the edit distance metrics are useless when it comes to word rearranging in a phrase or synonyms detection. Nevertheless, they might be a good approximation in many use cases.

3 Related work

Calculating the edit distance between two strings is a computationally costly task. It causes that the most straightforward approach to approximate string search, that is iterating over the entire dictionary D and comparing each element with q is usually too slow. To speed up that process we may use an auxiliary data structure, called index, that is intended to reduce the number of actual commissions that we conduct.

There are many indexing methods proposed both for rNN and kNN approximate searches [17,9]. Most generally we split them into two main categories: exact indexes and approximate indexes. Exact indexes guarantee that if there exists a record satisfying the search criteria, it will be returned. On the other hand, an approximate index might sometimes fail but by relaxing the conditions, it might also work faster.

For exact indexes, most common are solutions based on inverted indexes or trees. In an inverted index approach, we create a data structure that allows us to narrow the set of possible results, before conducting actual edit distance calculations. As an example of this approach, we can give the DevideSkip [7] algorithm or AppGram [15]. The main disadvantage of this approach is that we still need to do the manual purification of candidates, which can make them impractical when the dictionary elements are long or a high edit distance threshold is required.

Another class of exact indexes is trees-based methods. Most generally these methods reduce the time complexity of a dictionary search from linear to logarithmic. As an example for kNN approximate search, we can give a classical BK-Tree data structure or HS-Topk algorithm [14] which uses hierarchical segment tree index together with preliminary purifying. The disadvantage of these

approaches is that the time complexity is also dependant on the dictionary size and the length of a query string.

Another interesting class of solutions is indexes based on all potential misspellings generation. The solution has $O(1)$ time complexity but their spatial complexity is tremendous and grows really fast with the maximum supported edit distance between strings. However, in the case of Levenshtein distance (or Damerau-Levenshtein) the spatial complexity of these methods can be highly improved by generating not all the possible misspellings but only the deletions. This method is called symmetric delete spelling correction algorithm and was utilized by FastSS [13], which is currently state-of-the-art for small edit distance values. The method was future improved by Wolf Garbe in SymSpell [4] where we the all deletions generation can be reduced only to the prefix of selected length. This allows us to establish a compromise between the space and time complexity of the method.

Another category of index structures is approximate indexes. The main idea behind this class of solutions is embedding a costly edit distance metric space in another metric space (for example Euclidean) that will retain the properties of the original metric. After the projection, we can use locality sensitive hashing function to quickly compute kNN for our query term.

The main challenge for approximate indexes is to find a good embedding function. There were several proposed embedding functions for edit distance, for example, [11] or more recently CGK [1]. The main drawback of these embedding functions is that they are data independent. It means they work exactly the same regardless of the dictionary used in the search, which reduces the accuracy of the method. To mitigate the problem, the approach of training embedding functions from data using neural networks (a.k.a. learning to hash) was proposed recently: [2,18]. These approaches turned out to have much better properties than previously used functions, and what is also very important, they offer a more general framework, that can be used to embed many different metrics, not only the edit distance based ones.

4 SimpleWiki labels dataset

Our study aimed to create an efficient index for Wikification. To test the relevance of a proposed method we decided to test it with Wikipedia in Simple English. We called our test dictionary: SimpleWiki labels dataset.

The SimpleWiki labels dataset was created by parsing all the SimpleWiki articles and extracting all the links from them. Then, for each link we got its anchor text (which we called label) and added it to our dictionary. Before storing in dictionary we also removed all the charters from the label that are not English letters, digits or space character.

Generated dataset consists of 227,575 unique labels. Comparing to the dataset of 82,767 English words[1] it has different characteristics which are summarized in Table 1.

[1] We refer here to English words dictionary from SymSpell project [4].

In the Figure 1 we have also presented the differences in distribution of Damerau-Levenshtein metric between elements in both datasets. It is visible that in case of English words the distribution is much more concentrated than in the SimpleWiki labels. This differences in datasets might cause that the methods that were designed to work optimally for English words, might not be optimal for entity retrieval in Wikification process.

Table 1. Comparison of the SimpleWiki labels and English words datasets.

Dataset	SimpleWiki labels	English words
Dataset size	227,575	82,767
Avg. Len.	14.13	8.1
Std. Dev.	7.66	2.55
Min. Len.	1	1
Max. Len.	164	28

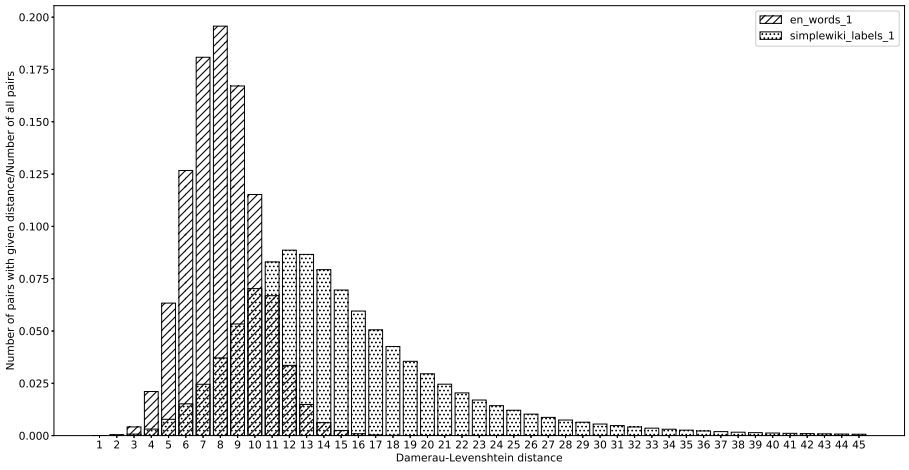

Fig. 1. A compression between distributions of Damerau-Levenshtein distances in English dictionary and SimpleWiki labels dataset.

5 Our method

Proposed index structure for approximate entity retrieval for Wikification uses approximate index with embedding function. Our solution is inspired by CNN-ED [2] but it was improved to better fit the task. First, our embedding function is trained with approximate string search in mind, not the approximate string

join as in the original paper. Second, we trained our function for Damerau-Levenshtein distance, not the Levenshtein, which better reflects the actual misspellings made by people. Finally, we provided a complete end-to-end solution, not only the embedding function itself, which can be compared with other approximate string search methods.

Our index structure consists of three main components. The first is an embedding function, that is trained to map our Damerau-Levenshtein metric space to Euclidean metric space. The second is a training component of the embedding function. The third is an efficient kNN index for Euclidean metric space, which we use to perform the actual search.

5.1 Embedding function

Our embedding function is neural network with one convolutional layer, one pooling layer, and the final dense layer. In the input, the function receives one-hot encoded strings. The maximum length of the input string is M=167 characters (the maximum label length of SimpleWiki labels is 164). The alphabet consists of 37 characters which are: "qwertyuiopasdfghjklzxcvbnm1234567890" and space.

One-hot encoding means that we transform each input string S of length L to the matrix X of size: $|A| \times L$ where $|A|$ is the size of our alphabet. Then for each character in a string:

$$\sum_{i=1}^{|A|}\sum_{j=1}^{L} X_{ij} = \begin{cases} 1 & \text{if } S_j = A_i \\ 0 & \text{if } S_j \neq A_i \end{cases} \tag{3}$$

If the string length is lower than the maximum string input, we fill the rest of the input matrix with zeros.

The convolutional layer uses 1D convolutional with 64 output channels, the kernel of size 3, stride 1, and padding 1, without a bias. As a result the network transforms the input of size $N \times |A| \times M$ (N is a batch size) to matrix of size $N \times 64 \times M$. The results of convolution are further passed to ReLU for non-linearity. The convolutional step is crucial in our function because it detects the local modifications in the input, without being sensitive to the modification position, which would be a case in deeply connected layers.

After the convolutional layer comes the pooling layer. We decided to use a max-pooling function with kernel size 2, which was inspired by the CNN-ED model. We tested our model both with and without the pooling layer and it turned out that the pooling significantly reduces the size of the network without a negative impact on the predictions. The max-pooling reduces the convolutional layer output from $N \times 64 \times M$ to $N \times 64 \times M/2$.

The output is constructed from the dense layer that maps its input to the vector of floats of size 100. This vector forms our final embeddings. The network has 538,404 trainable parameters and takes 2,05 MB of memory. Figure 2 shows the network architecture.

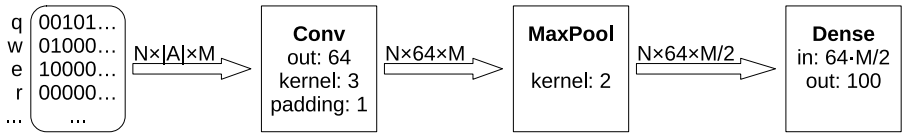

Fig. 2. The architecture of neural embedding function.

5.2 Training component

Loss function. The proposed embedding function was trained using triplet loss [5], together with mean square error:

$$\mathcal{L}(s_{acr}, s_{pos}, s_{neg}) = \mathcal{L}_t(s_{acr}, s_{pos}, s_{neg}) + \alpha\mathcal{L}_m(s_{acr}, s_{pos}, s_{neg}) \tag{4}$$

where α is the scaling factor, set to 0.1.

In the triplet loss approach, we train our network by sampling and comparing triplets from our training set. The first element of triplet is called anchor, the second is a positive example and the third is a negative example. In Damerau-Levenshtein metric space the distance between s_{acr} and s_{pos} is smaller than the distance between s_{acr} and s_{neg}. In a triplet loss we want to move the relation from Damerau-Levenshtein metric space to Euclidean space using a vector representations from embedding function: y_{acr}, y_{pos}, y_{neg}. The triplet loss is formally defined as follows:

$$\mathcal{L}_t(s_{acr}, s_{pos}, s_{neg}) = \max(0, \|y_{acr} - y_{pos}\| - \|y_{acr} - y_{neg}\| + \eta) \tag{5}$$

where $\eta = \overline{dist}(s_{acr}, s_{neg}) - \overline{dist}(s_{acr}, s_{pos})$ and $\overline{dist}(.)$ is a function that returns a Damerau-Levenshtein distance between its arguments, divided by the average distance between all pairs in the dictionary:

$$\overline{dist}(s_1, s_2) = \frac{dist(s_1, s_2)}{\frac{1}{|D|^2}\sum_{i=1}^{|D|}\sum_{j=1}^{|D|} dist(d_i, d_j)} \tag{6}$$

The triplet loss function pushes $\|y_{acr} - y_{pos}\|$ to 0 and $\|y_{acr} - y_{neg}\|$ to be greater than $\|y_{acr} - y_{pos}\| + \eta$. Where the η is in fact the actual distance between the negative and the positive examples.

The triplet loss itself is relative positioning loss function, which means that it only positions the learning set elements in a correct order, without preserving the absolute distance values between them. To mitigate the issue the additional mean square error loss is introduced. The $\mathcal{L}_m$ is formally defined as:

$$\begin{aligned}\mathcal{L}_m(s_{acr}, s_{pos}, s_{neg}) = (\|y_{acr} - y_{pos}\| - \overline{dist}(s_{acr}, s_{pos}))^2 + \\ (\|y_{acr} - y_{neg}\| - \overline{dist}(s_{acr}, s_{neg}))^2 + \\ (\|y_{pos} - y_{neg}\| - \overline{dist}(s_{pos}, s_{neg}))^2\end{aligned} \tag{7}$$

This loss component aims to make the Euclidean distance between embedding vectors the same as the averaged Damerau-Levenshtein distance between strings. Figure 3 presents a visual summary of the proposed learning architecture.

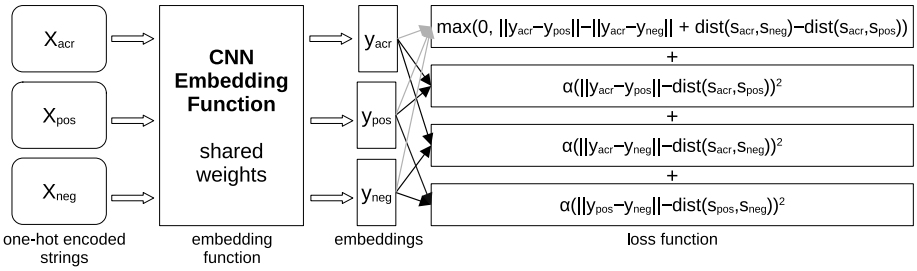

Fig. 3. Triplet Loss Network architecture.

Training samples. During each epoch of the training process, we iterate over all elements of the dictionary in random order, considering them our anchor elements. Then for each anchor, we select one positive and one negative example. The examples are selected either from the other dictionary elements or generated by corrupting the anchor word. In the first case, we consider the top 100 kNN of the anchor word and choose the positive and negative examples at random from them. In the second case, the positive and the negative examples are generated by corrupting the anchor, such that the positive example is at Damerau-Levenshtein distance 1 from the anchor and the negative at distance 2.

We train our Triplet Loss Network with a batch size of 64 and a learning rate of 0.001 until the epoch loss stabilizes.

5.3 Index

In order to achieve the full potential of our solution, in addition to a good hashing function, we need an efficient method to retrieve near neighbors from the dictionary. In our solution, we decided to use a faiss library [6], which is currently state-of-the-art for kNN search in Euclidean distance, using GPU.

We construct our index by creating a hash for every element in our dictionary. These hashes are then used to create the faiss index structure. Then for every incoming query string, we calculating the hash for it and look up its nearest neighbors in the index structure. Figure 4 summarizes this process.

6 Results

To test our method for different dictionary elements' lengths and edit distances, we prepared three different test cases. For the first test case, we took only the

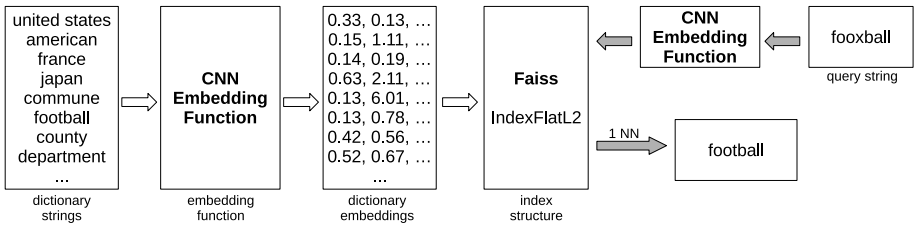

Fig. 4. The index construction process (white arrows) and approximate string search using index structure (gray arrows).

SimpleWiki labels that are no longer than 10 characters. For the second test case, we took the labels that are longer than 20 characters. For the last one we took all the SimpleWiki labels. The distances' distribution of the first and second test case labels is presented in Figure 5. The distribution of distances over all SimpleWiki labels was already presented in Figure 1.

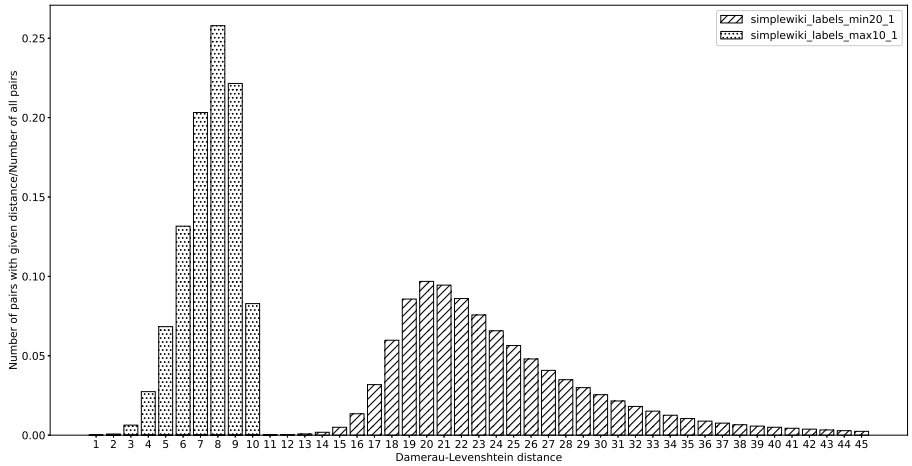

Fig. 5. Comparison of distributions of Damerau-Levenshtein distances between SimpleWiki labels of maximum length 10 and minimum length 20.

To test the performance of our method, we decided to compare it with SymSpell 6.7, which is currently state-of-the-art for small edit distances and short dictionary elements [4]. All of our tests were 1NN searches. When there were several possibilities (several words with an identical distance to the query), any of them was considered a correct result.

Additionally, we tested the neural hashing method in two variants. In the first valiant, we retrieved the 1NN for the query string hash and returned it as the

result. In the second variant, we retrieved 10NN for the query hash and returned the dictionary element with the smallest Damerau-Levenshtein distance to the query string.

All benchmarks were performed on Ubuntu 20.04, with Intel Core i7-8700 CPU, 16 GB RAM, and NVIDIA GeForce GTX 1660 GPU. For running SymSpell, we used .NET Core SDK 2.1.811 and for distance embeddings, Python 3.8.5, PyTorch 1.6.0, and Faiss 1.6.3.

For the first test case, we prepared three test sets. Every set contained 10 misspellings per word in the dictionary, which gave us 766,370 examples per set. The first set contained misspellings generated at Damerau-Levenshtein distance 1 from the original word. The second contained misspellings at distance 2. The third contained misspellings at distance 3.

Our hashing function was separately trained for the SimpleWiki labels subset used in this test case. In order to better fit the new dictionary statistics, we changed the input size of the function to 15 and increased the number of convolution output channels to 4096. Table 2 summarizes the running time of the SymSpell and both variants of the neural hashing method, for processing all test sets. Figure 6 shows the dependency between the maximum allowed Damerau-Levenshtein distance between the correct and misspelled word and running time of each procedure. As we can observe, the execution time of the SymSpell method grows within the maximum allowed distance between a misspelled and correct version of a label, while the execution time of the neural hashing method remains constant.

Table 2. The execution times (in seconds) of processing test sets for the first test case. "Ed" is the maximum allowed Damerau-Levenshtein distance between the correct and misspelled label for the test set.

Ed	Symspell, prefixLength=7	Neural Embeddings, 1NN	Neural Embeddings, 10NN
1	2.508	74.784	74.264
2	11.834	75.909	75.913
3	48.143	74.146	80.055

Table 3 shows the percent of correct results for the neural hashing method in its 1NN and 10NN variants. We consider the result correct, when the returned dictionary element is the correct form of the misspelled word or any other dictionary element with the same distance to the query string as the misspelled word.

Our second test case also contained the three test sets. Same as in the previous case, each set contained 10 misspellings per every word in the dictionary, which gave us 344,250 examples per test set. Because the second test case was built from longer labels, we also decided to test it against higher Damerau-Levenshtein distances. The first test set in the case contained the misspelled labels with Damerau-Levenshtein distance 3 to the correct form. The second set

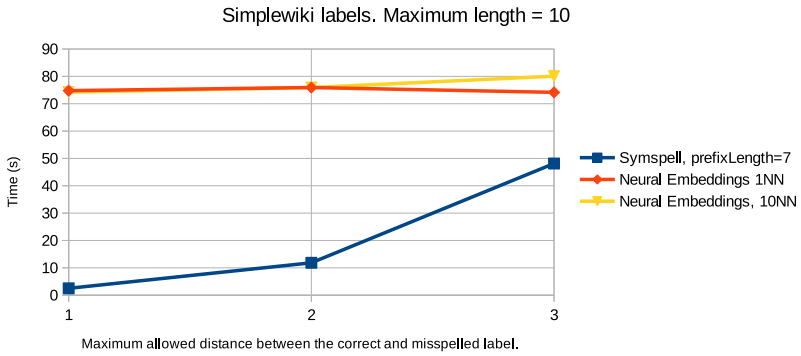

Fig. 6. The execution time compared to growing maximum allowed distance between correct and misspelled labels.

Table 3. The percent of correct results for the two variants of the neural hashing method, for the first test case. "Ed" is the maximum allowed Damerau-Levenshtein distance between the correct and misspelled label for the test set.

Ed	Neural Embeddings, 1NN	Neural Embeddings, 10NN
1	94.993 %	99.700 %
2	84.169 %	97.133%
3	77.515 %	94.329 %

contained the misspelled labels with Damerau-Levenshtein distance 4 and the final set with the distance 5.

For the neural hashing method, we used here the convolutional network with the same architecture as for the full labels set but trained only on the subset examples. Table 4 shows the execution time for SymSpell and both variants of the neural method for the test sets. Figure 7 plots the execution time of all the methods according to the growing maximum distance of misspelled words. As we can see, the neural method outperforms the SymSpell starting from edit distance = 4 and is much faster for the edit distance = 5.

Table 4. The execution times (in seconds) of processing test sets for the second test case. "Ed" is the maximum allowed Damerau-Levenshtein distance between the correct and misspelled label for the test set.

Ed	Symspell, prefixLength=7	Neural Embeddings, k=1	Neural Embeddings, k=10
3	45.727	70.367	72.145
4	193.047	72.668	74.922
5	1057.782	74.066	76.962

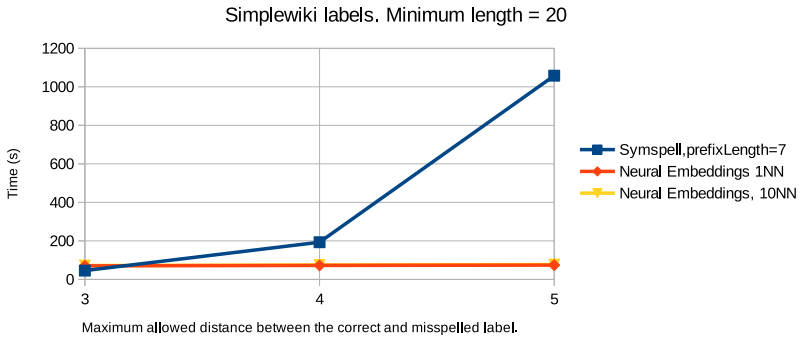

Fig. 7. The execution time compared to growing maximum allowed distance between correct and misspelled labels.

Table 5 shows the precision of the neural method for both variants. As we can see the results are very precise, even for 1NN. This shows that the neural method is well suited for the dictionaries with more sparse distance distributions.

Table 5. The percent of correct results for the two variants of the neural hashing method, for the second test case. "Ed" is the maximum allowed Damerau-Levenshtein distance between the correct and misspelled label for the test set.

Ed	Neural Embeddings, 1NN	Neural Embeddings, 10NN
3	97.599 %	98.865 %
4	96.403 %	98.05 %
5	95.395 %	97.423 %

Our final test case had only one test set that contained 10 misspellings per each label in the SimpleWiki labels dataset, which gave us 2,275,750 examples. The misspellings were introduced here in a progressive manner, which means that the maximum allowed Damerau-Levenshtein distance between the correct and the misspelled word grew with the label's length. We were calculating the Maximum Damerau-Levenshtein distance between the correct word w of length L and the misspelled word using the following formula:

$$max_{ed} = \begin{cases} \lceil \frac{L}{5} \rceil & \text{if } L <= 40 \\ 8 & \text{if } L > 40 \end{cases} \tag{8}$$

The progressive error rate was meant to reflect the real-world cases, where the probability of the typo, grows accordingly to the phrase length. The execution times for the test set and the accuracy of the neural algorithm are presented in Table 6. As we can see here, the performance of the neural hashing method is superior to the SymSpell algorithm, while the correctness is still on a high level.

Table 6. The execution times (in seconds) of processing test set and the percent of correct results for the two variants of the neural hashing method, for the third test case.

Method	Symspell, prefixLength=9	Neural Emb.,1NN	Neural Emb., 10NN
Execution time (s)	4933.292	508.678	556.225
Correctness	100%	88.851%	96.933%

7 Conclusions and Future works

The paper presented a novelty solution for an approximate string matching index, that is based on the latest research in the field of neural network metrics embeddings. The work has two important contributions. First is the convolution neural network-based embedding function optimized directly for Wikification. Secondly, according to our knowledge, this is the first attempt to combine the embedding function with an efficient Euclidean space search algorithm for the approximate string search task.

Our results show that the neural hashing method might be a good alternative to the other approximate string matching indexes, for the entity retrieval in the Wikification process. However, there are still many areas that might be further explored.

Firstly, we want to test our method for different distance metrics, e.g. weighted edit distance or phonetic algorithms. Thanks to the generality of our solution, it can be used with any distance metric. Particularly, we want to test the method for learning metrics from the data approach, where the metric is learned from the real user misspellings.

Secondly, we want to find the correlation between the dictionary and the actual size of the embedding neural network. We should consider here not only the dictionary size but also the element lengths and used metric. The correlation would be very important in the adaptation of the method for different use cases.

Finally, we want to future investigate the possibilities of joining the neural method with the traditional ones, to get the best from both worlds. We think that using the SymSpell for the shorter strings and the neural hashing method for longer ones might be the best solution for practical applications.

Acknowledgments

The work was supported by funds of Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology.

References

1. Chakraborty, D., Goldenberg, E., Koucký, M.: Streaming algorithms for embedding and computing edit distance in the low distance regime. In: Proceedings of

the Forty-Eighth Annual ACM Symposium on Theory of Computing. p. 712–725. STOC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2897518.2897577
2. DAI, X., Yan, X., Zhou, K., Wang, Y., Yang, H., Cheng, J.: Convolutional embedding for edit distance. Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (Jul 2020). https://doi.org/10.1145/3397271.3401045
3. Damerau, F.J.: A technique for computer detection and correction of spelling errors. Commun. ACM **7**(3), 171–176 (Mar 1964). https://doi.org/10.1145/363958.363994
4. Garbe, W.: Symspell, https://github.com/wolfgarbe/symspell, last accessed 18 Dec 2020
5. Hermans, A., Beyer, L., Leibe, B.: In Defense of the Triplet Loss for Person Re-Identification. arXiv preprint arXiv:1703.07737 (2017)
6. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with gpus. IEEE Transactions on Big Data pp. 1–1 (2019). https://doi.org/10.1109/TBDATA.2019.2921572
7. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico. pp. 257–266. IEEE Computer Society (2008). https://doi.org/10.1109/ICDE.2008.4497434
8. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. Journal of Computer and System Sciences **20**(1), 18 – 31 (1980). https://doi.org/https://doi.org/10.1016/0022-0000(80)90002-1
9. Rachkovskij, D.: Index structures for fast similarity search for symbol strings. Cybernetics and Systems Analysis **55**(5), 860–878 (2019)
10. Sevgili, O., Shelmanov, A., Arkhipov, M., Panchenko, A., Biemann, C.: Neural entity linking: A survey of models based on deep learning (2020)
11. Sokolov, A.: Vector representations for efficient comparison and search for similar strings. Cybernetics and Systems Analysis **43**(4), 484–498 (2007)
12. Szymański, J., Naruszewicz, M.: Review on wikification methods. AI Communications **32**(3), 235–251 (2019)
13. T. Bocek, E. Hunt, B.S.: Fast Similarity Search in Large Dictionaries. Tech. Rep. ifi-2007.02, Department of Informatics, University of Zurich (April 2007)
14. Wang, J., Li, G., Deng, D., Zhang, Y., Feng, J.: Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In: Gehrke, J., Lehner, W., Shim, K., Cha, S.K., Lohman, G.M. (eds.) 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015. pp. 519–530. IEEE Computer Society (2015). https://doi.org/10.1109/ICDE.2015.7113311
15. Wang, X., Ding, X., Tung, A.K.H., Zhang, Z.: Efficient and effective knn sequence search with approximate n-grams. Proc. VLDB Endow. **7**(1), 1–12 (Sep 2013). https://doi.org/10.14778/2732219.2732220
16. Wikipedia:Lists of common misspellings/For machines, https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines: Last accessed 18 Dec 2020
17. Yu, M., Li, G., Deng, D., Feng, J.: String similarity search and join: a survey. Frontiers of Computer Science **10**(3), 399–417 (2016)
18. Zhang, X., Yuan, Y., Indyk, P.: Neural embeddings for nearest neighbor search under edit distance (2020), `https://openreview.net/forum?id=HJlWIANtPH`