doi: 10.17234/INFUTURE.2015.4 Keynote paper

Approximate sentence matching and its applications in corpus-based research

Rafał Jaworski
Adam Mickiewicz University in Poznań, Poland
rjawor@amu.edu.pl

Summary

This article presents the technique of approximate sentence matching. It is known to perform well as an aid in linguistics-related tasks, among others in translation. The article presents two different applications of this technique in author's algorithms for concordance searching and sentence clustering.
Technical part of this article is succeeded by conclusions regarding the synergy between computer scientists and linguists. It is necessary for these two groups to overcome communication obstacles and work together to create linguistic tools of the future.

Keywords: approximate matching, corpus, searching, clustering, synergy

Introduction

Approximate sentence matching (ASM) is a technique of retrieving sentences from a large corpus that are similar to a given pattern sentence. One of the most important challenges in ASM is finding a good sentence similarity measure, which would reflect human intuition of sentence resemblance. Another challenge is designing a robust and scalable algorithm, capable of retrieving similar sentences from a large text corpus, using a given similarity measure. This article presents an author's algorithm for approximate sentence matching and sentence similarity computation. Its implementation is based on suffix arrays augmented by custom designed auxiliary data structures. In order to measure the sentence similarity it uses an improved version of Jaccard index (see [Jaccard, 1901]).
The second part of the article focuses on applying the sentence similarity measure in the task of building parallel corpora for under-resourced languages. This is achieved by selecting the most valuable sentences of a monolingual corpus by the means of cluster analysis.
Work on the development and testing of the above algorithms required extensive cooperation with linguists. The last part of this article lists observations and conclusions regarding this work.

Approximate sentence matching in CAT

As fully automatic translation often falls short of expectations, people turn to other methods of using computers in the translation process. One of such methods is Computer-Aided Translation (CAT). CAT systems use sets of previously translated sentences, called the translation memories. For a given sentence, CAT system searches for a similar sentence in the translation memory. If such a sentence is found, its translation is used to produce the output sentence. This output sentence is then used as a suggestion for translation. A human translator carries out the post-editing. This technique is used in many CAT platforms, such as SDL Trados [SDL, 2015] or Kilgray Memoq [Kilgray Translation Studios, 2015].

It is crucial for a CAT system that the translation memory contain translations of frequently used sentences. With such a translation memory, a CAT system generates good suggestions for translations, reducing the amount of human work on post-editing.

Author's approximate sentence matching algorithm

This section presents the author's solution for approximate sentence matching. It differs significantly from the approach described in the previous section. The most important difference is that author's solution allows to cover the searched sentence with more than one example sentence from the memory.

In order to carry out the search procedures efficiently, an offline index, based on the suffix array (see [Manber et. al., 1990], [Makinen et al., 2004]) and other auxiliary data structures was used.

Operations on index

Main operations performed on the index are the following:

- *void addToIndex(string sentence, int id)* This method is used to add a sentence to the index along with its unique id. The id is treated as additional information about the sentence and is then retrieved from the index by the search algorithm. This is useful in a standard scenario, where sentences are stored in a database or a text file, where the id is the line number. Within the *addToIndex* method the sentence is tokenized and from this point forward treated as a word sequence.
- *void generateIndex()* After adding all the sentences to the index the *generateIndex* method should be called in order to compute the suffix array for the needs of the fast lookup index. This operation may take some time depending on the number of sentences in the index. Nevertheless, its length rarely exceeds one minute (in reported experiments with 2 million sentences the index generation took 6-7 seconds).
- *simpleSearch(string pattern)* Basic search method takes a text fragment, tokenizes it and tries to locate all its occurrences in the index. The return

value is a list of matches, each holding information about the id of the sentence containing the fragment and an offset of this fragment in this sentence.

- *concordiaSearch(string pattern)* The main similarity search method returns the longest fragments from the index that cover the search pattern. This technique can be used for concordance searching.

Simple searching example

Functioning of the index generation and simple search algorithm is best illustrated by the following example. Suppose that the index contains the sentences presented in Table 1. Note that the id's of the sentences are not consecutive as there is no such requirement.

Table 1: Example index.

Sentence	ID
Novel methods were used to measure the system success rates.	23
Various statistics, including the school success rate, were reported.	12
The research is still ongoing	259

Let us now perform simple search for the fragment "success rate" in the example index. The expected results (in the form [sentence id, offset]) are *(23, 8)* and *(12, 5)*. Returned results allow for quick location of the contexts in which the phrase "success rate" appeared. Note also that the system is expected to return the result *(23,8)* ("the system **success rates**") even though the word "rate" was in plural in the index.

Index construction

Author's index incorporates the idea of a suffix array and is aided by two auxiliary data structures – the **hashed index** and **markers array**. During the operation of the system, i.e. when the searches are performed, all three structures are loaded in RAM. For performance reasons, hashed index and markers array are backed up on the hard disk.

When a new sentence is added to the index via the aforementioned *addToIndex* method, the following operations are performed:

- tokenize the sentence
- lemmatize each token
- convert each token to numeric value according to a dynamically created map (called dictionary)

Lemmatizing each word and replacing it with a code results in a situation, where even large text corpora require relatively few codes. For example, research of this phenomenon presented in [Jaworski, 2013] reported that a corpus of 3 593 227 tokens required only 17 001 codes. In this situation each word could be stored in just 2 bytes, which significantly reduces space complexity.

The following sections will explain in detail the data structures used by the index.

Index – hashed index

Hashed index is an array of sentences in the index. The sentences are stored as code lists. For example, let us compute the hashed index for the sentences of the example index shown in Table 1 ("Novel methods were used to measure the system success rates." "Various statistics, including the school success rate, were reported." and "The research is still ongoing"). First, the sentences are tokenized, then the tokens are lemmatized and mapped into integers. Results of this process for all three example sentences are shown in Table 2.

Table 2: Hashed sentences.

Novel	methods	were	used	to	measure	the	system	success	Rates
novel	method	be	use	to	measure	the	system	success	Rate
1	2	3	4	5	6	7	8	9	10

Various	statistics	including	the	school	success	rate	were	reported
Various	statistic	include	the	school	success	rate	be	report
11	12	13	7	14	9	10	3	15

The	research	is	still	ongoing
The	research	be	still	ongoing
7	16	3	17	18

The dictionary created during this process is shown in Table 3.

Table 3: The dictionary

Lemma	**code**	**lemma**	**code**	**lemma**	**code**
Be	3	rate	10	success	9
Include	13	report	15	system	8
measure	6	research	16	the	7
Method	2	school	14	to	5
Novel	1	statistic	12	use	4
ongoing	18	still	17	various	11

Code lists obtained from the sentences are then concatenated in order to form the hashed index. A special code (referred to as EOS – end of sentence) is used as sentence separator.

The hashed index is used as the "text" (denoted *T*) for the suffix array.

Index – markers array

When a fragment is retrieved from the index with the help of a suffix array, its position is returned as the search result. However, this position is relative to the "text", stored as the hashed index. For example, if we searched for the fragment "success rate", as in previous examples, we would obtain the information about

two hits: one at position 8, and the other at position 16 (mind that the positions are 0-based and EOS characters count as single text positions).
This result does not contain information about the id of the sentence where the fragment was found nor the offset of the fragment in this sentence. Naturally, this information is retrievable from the hash index alone. However, operation of that kind would require searching the hashed index in at least *O(n)* time in order to determine which sentence contains the given position. In addition, this would only return the ordinal number of the sentence, not its id, since this information is not stored in the hashed index.
In order to overcome these difficulties, a simple, yet effective data structure was introduced. Markers array is used to store information about the sentence id and offset of each word. Technically it is an array of integers of the length equal to the length of the hashed index. Each integer in the markers array stores both the sentence id and the offset of the corresponding word in the hashed index. Current implementation uses 4-byte integers, where 3 bytes are assigned to store the sentence id and 1 byte is used for the offset. This means the index can store up to 16 777 216 sentences, each no longer than 255 characters (one position is reserved for the EOS character). For example, the pair: $id = 342_{10} = 101010110_2$, $offset = 27_{10}=11011_2$ is stored as the integer: $10101011011011_2=10971_{10}$.
Even though the markers array is not free from redundancy, the cost space occupied by this data structure is affordable on modern computers. The benefits of its influence on speeding up the search process are much more significant.

Index – suffix array
The last element of the index is a generated suffix array. It is constructed after the hashed index is complete. Technically, this structure is a classic suffix array for the hashed index. As stated in Section *Index – hashed index*, hashed index plays the role of the "text" (*T*), whose "letters" are dictionary codes of words.
Algorithm used for construction of the suffix array is an implementation of classic construction algorithm proposed by Manber and Myers [Manber, Myers; 1990], running in *O (n log(n))* time. It differs significantly from the naive approach (generating suffixes, sorting them and reading their positions) which runs in $O (n^2 log(n))$.
Sorted suffixes for the example hashed index result in the following suffix array: [0, 1, 2, 18, 23, 3, 4, 5, 6, 14, 21, 7, 16, 8, 17, 9, 11, 12, 13, 15, 19, 22, 24, 25, 26, 20, 10].

Simple searching
Searching of the index is done according to the classic suffix array search procedure. In order to make this possible, an input search phrase must first undergo the same procedure as every sentence being added to the index (lemmatizing and coding).

Let us demonstrate the search on the same example search pattern presented in Section *Simple searching example*. We are searching for the pattern "success rate" in the example index. The pattern is tokenized and lemmatized, thus transformed into a sequence of lemmas: 'success' 'rate'. These lemmas are then encoded using the dictionary generated during the creation of the hashed index (see Table 3). As a result, the search pattern has the form '9 10'.
By searching in the suffix array with the help of the hashed index we know that the phrase "success rate" can be found in the source text at positions 16 and 8. In fact, we also know that this phrase is present only at these positions, as follows from suffix array properties. However, we expect that the final search results will be given in the form [sentence id, offset]. For that we need to check the markers array. In this example *M[8]=(23,8)* and *M[16]=(12,5)*. This corresponds to the expected final results.

Concordia searching
Concordia search internally uses the simple search procedure but serves for a more complicated purpose. It is aimed at finding the longest matches from the index that cover the search pattern. Such match is called "matched pattern fragment". Then, out of all matched pattern fragments, the best pattern overlay is computed.
Pattern overlay is a set of matched pattern fragments which do not intersect with each other. Best pattern overlay is an overlay that matches the most of the pattern with the fewest number of fragments.
Additionally, the score for this best overlay is computed. The score is a real number between 0 and 1, where 0 indicates, that the pattern is not covered at all (i.e. not a single word from this pattern is found in the index). The score 1 represents the perfect match - pattern is covered completely by just one fragment, which means that the pattern is found in the index as one of the examples. The formula used to compute the best overlay score is shown below:

$$score = \sum_{fragment \in overlay} \frac{length(fragment)}{length(pattern)} \cdot \frac{\log(length(fragment)+1)}{\log(length(pattern)+1)}$$

According to the above formula, each fragment covering the pattern is assigned base score equalling the relation of its length to the length of the whole pattern. This concept is taken from the classic Jaccard index (see [Jaccard, 1901]). However, this base score is modified by the second factor, which assumes the value 1 when the fragment covers the pattern completely, but decreases significantly, when the fragment is shorter. For that reason, if we consider a situation where the whole pattern is covered with two continuous fragments, such overlay is not given the score 1.

Let us consider an example illustrating the Concordia search procedure. Let the index contain the sentences from Table 4:

Table 4: Example sentences for Concordia searching:

Sentence	ID
Alice has a cat	56
Alice has a dog	23
New test product has a mistake	321
This is just testing and it has nothing to do with the above	14

Here are the results of Concordia searching for pattern: "Our new test product has nothing to do with computers":

Table 5: Concordia search results.

Pattern interval	Example id	Example offset
[4,9]	14	6
[1,5]	321	0
[5,9]	14	7
[2,5]	321	1
[6,9]	14	8
[3,5]	321	2
[7,9]	14	9
[8,9]	14	10

Best overlay: [1,5] [5,9], score = 0.53695

These results list all the longest matched pattern fragments. The longest is [4,9] (length 5, as the end index is exclusive) which corresponds to the pattern fragment "has nothing to do with", found in the sentence 14 at offset 7. However, this longest fragment was not chosen to the best overlay. The best overlay are two fragments of length 4: [1,5] "new test product has" and [5,9] "nothing to do with". Notice that if the fragment [4,9] was chosen to the overlay, it would eliminate the [1,5] fragment.
The score of such overlay is 0.53695, which can be considered as quite satisfactory to serve as an aid for a translator.

Sentence clustering

The idea of sentence clustering is applied to the task of preparing a specialized translation memory for a given purpose. It is based on the assumption that the most useful sentences that might appear in translation memory are those which occur most frequently in texts in the source language. A clustering algorithm classifies sentences from a monolingual corpus into clusters of similar sentences and selects one representative for each cluster. The translation for each representative is then produced manually by human specialists. The database prepared in such a manner forms a high-quality specialized translation memory,

covering a wide range of sentences intended for translation. This technique can be applied to prepare parallel corpora for under-resourced languages.

Clustering algorithm

Clustering algorithms work on sets of objects. They use a measure of distance between these objects in order to divide sets into smaller chunks containing objects which are "close" to each other (in terms of the sentence similarity measure). In our case, the set of objects is the monolingual corpus, the objects are sentences. In order to determine the distance between sentences we use two distance measures: "cheap" and "expensive" (the idea of using two distance measures is inspired by [McCallum et. al., 2000]). The "cheap" and "expensive" terms refer to complexity of calculations.

Cheap sentence distance measure

The cheap sentence distance measure is expected to work fast. It is based only on the lengths of sentences. The formula for computing the value of the measure is the following:

$$d_C(S_1, S_2) = 2^{-\frac{|len(S_1) - len(S_2)|}{10}}$$

where *len(S)* is the length of the sentence in characters.

Expensive sentence distance measure

The expensive sentence distance measure is expected to give more accurate assessment of sentence similarity. The first step of computing the distance between sentences S_1 and S_2 in this measure is to recognize named entities of the two sentences. Then, so called "penalties" are imposed for discrepancies between the sentences. The penalty values have been based on human translators' intuition, e.g. if one sentence contains a named entity and the other does not, then the sentences are not likely to be similar. (The values for penalties are bound to be calculated by self-learning techniques in future experiments).
Let us define:

- p – the sum of penalties imposed on sentences S_1 and S_2
- L_{S1} – the number of words and Named Entities in S_1
- L_{S2} – the number of words and Named Entities in S_2

$$d_E(S_1, S_2) = 1 - \frac{2p}{L_{S1} + L_{S2}}$$

The clustering procedure

The detailed clustering algorithm is:

IN: Set (S) of strings (representing sentences)
OUT: List (C) of clusters (representing clusters of sentences)
1. Divide set S into clusters using the QT algorithm with the "cheap" sentence distance measure (the measure is based only on sentence length).
2. Sort the clusters in the descending order by number of elements resulting in the sorted list of clusters, C.
3. For each cluster cL in list C:
(a) Apply the QT algorithm with the "expensive" distance measure (based on sentences contents) to cL, resulting in subclusters.
(b) Sort the subclusters in cL in descending order by number of elements.
(c) Copy the sentences from the cL into C

The QT algorithm is described in [Heyer et al., 1999]. The final step is performed by humans. They manually select the most valuable sentence from each clusters. The task is facilitated by a frequency prompt (the most frequent sentences always appear at the beginning of the cluster).

Evaluation

The clustering method presented above was applied in an experiment described in [Jaworski et al.; 2010]. The authors prepared a translation memory of about 500 sentences from Polish law texts and used it to augment existing memory of 20 000 sentences. The augmentation in size of the memory by merely 2.3% resulted in increasing the percentage of sentences found in the memory from 9% to 31% and decreasing the overall human effort in the process of translation by 12.5%.

Conclusion – need for synergy

The techniques presented above proved useful in experiments. Importantly, they were developed with the help of professional linguists. During this process, some subjects became clear. Firstly, linguists do not seem to know much about how computer software is created and which techniques are easy to implement and which are not. However, to be fair, computer scientists probably know even less about the translation process (see [Zetzsche, 2014]).

Moreover, the two groups are motivated differently – translators are primarily focused on the quality of their translation. They are, naturally, interested in optimizing their effort to achieve desired quality, but this is not their main concern. Computer scientists, on the other hand, are focused on the performance of their software and optimization of the efficiency of translator's work. Because of these differences, software developers and translators need to communicate with each other as often as possible.

The work model where a computer scientist comes up with ideas of improvement, implements them and presents to the linguist is actually one of the worst imaginable. Because of developer's lack of knowledge concerning the translation process and the different goals described above, solutions he or she sug-

gests are almost never perfectly useful for the linguist (which is what the developer hopes for). Instead, the linguist works hard to modify the developer's idea and the developer feels that his work has gone in vain, which leads to frustration on both sides.

Ideally, they should spend about 1-2 hours a week working together. They should exchange concepts and educate each other in their fields. The computer scientist should translate a document under supervision of the linguist. The translator should get accustomed with the architecture of the system he or she is using for their work. Ideas for new features in the software should be a result of their mutual thinking process. Only with this approach one can establish true synergy.

References

Heyer, Laurie; Kruglyak, Semyon; Yooseph, Shibu. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. // *Genome Research*. 9 (1999); 1106-1115

Jaccard, Paul. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. // *Bulletin de la Société Vaudoise des Sciences Naturelles*. 57 (1901); 547-579

Jaworski, Rafał; Jassem, Krzysztof. Building high quality translation memories acquired from monolingual corpora. *// Proceedings of the IIS 2010 Conference/* (2010)

Jaworski, Rafał. Anubis – speeding up Computer-Aided Translation. // *Computational Linguistics – Applications, Studies in Computational Intelligence*. Springer-Verlag 458 (2013)

Kilgray Translation Studios. Memoq Translator Pro. https://www.memoq.com/memoq_translator_pro (8.05.2015)

Makinen, Veli; Navarro, Gonzalo. Compressed compact suffix arrays. // *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM) /: LNCS 3109 (2004), 420-433*

McCallum, Andrew; Nigam, Kamal; Ungar, Lyle. Efficient clustering of highdimensional data sets with application to reference matching. // *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (2000), 169-178*

Manber, Udi; Myers, Gene. Suffix arrays: a new method for on-line string searches. // *First Annual ACM-SIAM Symposium on Discrete Algorithms(1990)*. 319–327

SDL. Trados Studio. http://www.sdl.com/cxc/language/translation-productivity/trados-studio/ (8.05.2015)

Zetzsche, Jost. Encountering the Unknown. Part 2. // *Proceedings of the 17th Annual Conference of the European Association for Machine Translation EAMT2014*/ 2014, xvii