Sequence matching with subsequence analysis

Marko Ferme, Milan Ojsteršek

Abstract—This article describes an alternative approach for matching user text input in natural language processing against an existing knowledge base, consisting of semantically described words or phrases. Most common methods of natural language processing are overviewed and their main problems are outlined. A sequence matching algorithm is introduced, which deals with some of these problems. First the longest subsequences discovery algorithm is explained. Then the major components of the similarity measure are defined and the computation of concurrence and dispersion measure is presented. Results of the algorithms performance on a test set are then shown. The work is concluded with some ideas for the future and some examples where our approach can be practically used.

Keywords—Sequence matching, subsequence analysis, similarity measure, fuzzy string search.

I. INTRODUCTION

In natural language processing, user input is usually being matched against a knowledge base, which consist of a finite collection of semantically described words or phrases. While trying to classify parts of the user input a comparison between those two must eventually be made. Because of user input being an infinite set, either a reduction of input words or expansion of the knowledge base must be made. Popular approaches that solve this problem include stemming, lemmatization and various distance functions. Stemming is based upon a set of rules, which determine word morphing, and is therefore limited to weakly inflected languages, where such rule collections exist. Lemmatization is used in conjunction with large language specific dictionaries, which are used to expand the knowledge base dictionary. This information is then used to derive morphed words into their lemma. Both approaches are intolerable to user input errors and have a finite set of either rules or words. Distance functions such as Levenshtein distance [1], which are based on the number of changes required to transform one sequence into another, are used to address this issue. While they do offer some level of "fuzzy" sequence matching they lack the information and depth of analysis to determine sufficient sequence similarity. Additionally all of the above methods require text segmentation to a smallest free form of a language (word), which implies the usage of advanced matching algorithms when dealing with phrases or multiple word entities. In our work we present an alternative approach for determining sequence similarity based on subsequence analysis which is language invariant, error tolerable, does not require additional rules or language dictionaries and can be used with same efficiency on single words, phrases or even larger texts. Moreover it can also be used for phrase extraction or detection in sentences or larger quantities of text. In chapter II we describe the problems we have encountered while trying to deploy the more commonly used solutions described above. We also describe the proposed solution to this problems and define the necessary attributes for determining similarity between two sequences and a similarity metric. In chapter III an algorithm for determining the longest subsequences from a sequence pair is described. In chapter IV we describe the computation of similarity. We provide the measured similarity results on a test set in chapter V and conclude our paper with some future work ideas in chapter VI.

II. USER INPUT MATCHING

The problem of user input manipulation, to better match the existing knowledge, is one being solved in different manners, mostly depending on the language in which the knowledge is contained and the user input is submitted.

A. Stemming

One of the most common approaches in natural language processing is stemming. It is very effective in languages that are weakly inflected like modern English, Swedish, Norwegian and Danish and struggles with languages that are moderately inflected such as Spanish, Italian, French, Portuguese and Romanian. Languages being highly inflected are considered inappropriate for efficient stemming [2]. Such languages include all the Slavic languages. Stemmers are also intolerable to user input errors (misspelling), which are bound to occur during natural text processing.

B. Lemmatization

In heavily inflected languages the use of lemmatization is preferred. It offers a fast and accurate way of matching user input to morphed instances of a headword but requires exact dictionaries, which have to be build by language experts. A major problem in the process of lemmatization is disambiguation, which occurs when a word or phrase can be transformed into two or headwords. It is most widely being solved with the usage of tree taggers which require large training corpuses and use probability to determine the most suitable headword, which we call a lemma. Building such large collections is very time consuming and requires the aid

Manuscript received July 21, 2010.
M. Ferme is with the Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia (phone: +386-2-220-7253; fax: +386-2-220-7272; e-mail: marko.ferme@uni-mb.si).
M. Ojsteršek is with the Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova ulica 17, 2000 Maribor, Slovenia (e-mail: ojstersek@uni-mb.si).
of language experts. Such corpuses can also suffer from domain specific language usage and can cripple the tree tagger results on a different domain. A very large portion of misses in lemmatization, when being used on heavily inflected languages, is produced from unknown words [3], such as names, surnames and geographical locations. Those are mostly excluded from dictionaries and tagged corpuses, which makes them nigh on impossible to convert to a lemma. Input error (misspelling) intolerance during lemmatization or tree tagging is in most cases also unaddressed.

C. Edit distance functions

Edit distance function where designed to counter the occurrence of misspelling in sequence matching, which both of the above described approaches cannot resolve. The basic idea is to determine the number of transforms needed to convert one sequence to another. Different variations of edit distance are known and used, one of the most popular being the Levenshtein distance [1]. It is defined as the number of insertions, deletions and substitutions needed to align two sequences, where weights for each transformation can be defined. Its most commonly used in spellchecking and is implemented in various software that contains text processing.

D. Sequence matching

Algorithms for sequence matching exist but are more popular in DNA sequence matching then in string matching. They try to find global [4] or local [5] subsequence alignments and mostly use expert defined tables which determine the distance between single sequence elements. Because our goal was to find a measure of similarity expressed in a way, that would allow effective comparison between sequences and allowed us to find the most similar one, the above described sequence matching algorithms were not sufficient.

We propose a new algorithm that discovers the longest subsequences contained in a pair of sequences and then evaluates them. By doing so we measure similarity instead of difference. We define three main decision factors, that will help us calculate a similarity measure:

- total length of common subsequences
- dispersion of common subsequences
- order of common subsequences

Based on this decision factors, similarity between two sequences is calculated and present as a value between zero, meaning that the sequences are disjoint or completely scrambled, and one, meaning that the sequences are identical.

III. SUBSEQUENCE DISCOVERY

In order to find similarity between two sequences we pursued an idea of finding the longest common subsequences which do not overlap. Our first step was to find all the subsequences contained in a sequence pair. We derived an algorithm from the solution to the longest common substring problem. In Fig 1. we can see how algorithm transverses through the matrix, building a list of subsequences candidates.

![Fig. 1 Subsequences discovery](image_url)
Fig. 2 Matrix division into zones for determining the longest subsequences

Every remaining subsequence is then processed depending on the zone it is residing in. For each subsequence the start and end coordinates are compared with the conditions shown in (3), (4) and (5) and a fitting zone is assigned. Three outcomes are possible:

1. If the subsequence candidate start and end coordinates are both located in zone I or in zone III, we remove that subsequence from the list of subsequences candidates, because it is already contained in the subsequence just added to the final list.

2. If the subsequence candidate start and end coordinates are located in zone II, we leave the subsequence unaltered, because it does not overlap with the subsequence just added to the final list and is still a potential candidate for the final subsequence list.

3. If the subsequence candidate start or end coordinates are located in zone III, subsequence overlapping must be solved. The subsequence cannot start and end in zone III, since that would mean that it is longer than the subsequence we just added to the final list, which cannot be true, since we ordered the list descending by sequence length. We then must either remove the leading or the trailing members of the subsequence depending on where the subsequence starts. If it starts inside zone I, we must remove the leading elements until we move its starting coordinates into zone II. If it starts inside zone III we must remove its trailing elements until we move its end coordinates into zone II. That way we removed the possible overlapping by favoring the longest sequence to retain all its member elements and removing the overlapped elements from the shorter subsequence candidates.

The candidate list is again ordered according to the subsequences length, the longest subsequence added to the final subsequences list and the candidates list is checked for overlapping again. This process is repeated until the candidates list is empty and we are left with the final list of longest subsequences, which do not overlap.

IV. MEASURING SEQUENCE SIMILARITY

The similarity measure between two sequences is composed out of the following three parts:

- concurrence measure,
- dispersion measure and
- ordering measure.

The first part measures the subsequence concurrence in both sequences being matched and is calculated as shown in (6).

\[ f = \frac{\sum_{i=1}^{L} \min(\text{a}_i, \text{b}_i)}{\max(\text{a}, \text{b})} \]  

(6)

\( f \) - length of sequence a
\( b \) - length of sequence b
\( L \) - length of the longest subsequences list
\( L_{\text{max}} \) - length of the i-th subsequence in the longest subsequences list

This part of comparison basically represents the share of elements contained in both sequences. If the sequences are identical the result is one. If the sets are disjoint the result is zero.

The second part measures the subsequences dispersion and is calculated as shown in (7).

\[ g = 1 - \left( \frac{N_p - 1}{\min(\text{a}, \text{b}) - 1} \right)^{\frac{1}{L_{\text{max}}}} \]  

(7)

\( N_p \) - number of longest subsequences found
\( a \) - length of sequence a
\( b \) - length of sequence b
\( L_{\text{max}} \) - length of the longest subsequence

Dispersion measure is designed around the fact that good similarity can be achieved only if there is a small number of found subsequences. Two identical sequences should only have one found subsequence, where the dispersion measure would be one. On the other hand, two sequences that might have the same elements, but their order is completely scrambled, would have n subsequences, where n is the length of the shorter sequence. In that case the dispersion similarity would be zero. The equation constructs a curve between these two values where its shape is affected by the length of the longest subsequence found. The division of the length of the
longest subsequence by 4 makes this function either a convex, concave or a linear function. The higher the value the slower the curve will decline from the value 1, which means that it will be more tolerable to the higher number of subsequences, and the other way around. The value 4 which makes the curve linear was chosen for our test sets under the assumption that when comparing sequence and finding the longest subsequence with its length less than 4 should decline more rapidly. For other test or working sets a different value can be chosen as the convex/concave limit. Fig 3. shows the dispersion measure behavior for different function parameters based on the comparison of two sequences, which contained 15 elements. Axis Z represents the calculated dispersion measure \( g \), axis X represents the length of the longest subsequence found \( \ell_{\text{max}} \) and axis Y represents the number of subsequences found \( d_{p} \). We can see that in cases of longer sequences the dispersion measure is very high when the number of found subsequences (Y) is low. The value declines more rapidly if the maximum subsequence length (X) is low.

The measure of ordering is the last measure included in the measure of similarity. We do recognize its importance in sequence analysis but since our original problem was a sequence set, where ordering was not so important for discovering similarity (our sequence domain was a flexible word order language), we did not develop it further. So in our current solution we used the maximum value of order measure for each case of comparison, which is one. We do plan to address and solve this problem in our future work.

Since all the single measures are normalized the final measure of similarity is computed as a product of sub measures. The result of such a product is also normalized and can be used for simple comparison to find the most similar sequence.

V. RESULTS

For measuring the results we had to create a test set, which would allow us to compare sequences against each other and would at the same time contain the information about the closest match. Therefore we chose to use a Slovene language dictionary, which contains a large number of words in all morphed forms and their relation to the lemma as our test set. We suspected that our algorithm could have troubles on such a set, since it contains a lot of sequences that are very similar, where their distinction could prove to be difficult. The dictionary contains over 3 million distinct words which are linked to over 250 thousand lemmas. We then randomly chose one thousand words out of the whole dictionary creating a test set with an average sequence length of 10. We used the algorithm to compare each of the chosen sequences against the entire lemma collection in order to find the most similar match. That way we calculated all possible similarity measures. Then, we chose the lemma with the highest similarity measure and also noted the similarity measure between the random word and its true lemma. If the lemma with the highest similarity measure is the same as the sequence true lemma, our algorithm performed well. We are aware that better approaches of choosing the most similar subsequence could be made and we discuss them in our plans for future work.

Fig 4. shows the measured result of our comparison. We can see, that the highest similarity between a sequence and its lemma was achieved 80.5 % of the time. In that cases the average similarity was 86.4%. In cases where the algorithm missed the lemma we can see, that the average similarity was 70.5%. From the collection of the missed cases we see that there were in majority considerably different as their lemma. Based on our result we can conclude that this algorithm is successful and suits the needs of natural language processing. Its success rate is slightly worse than those of stemmers and tree taggers when compared on a set that is strictly build out of the dictionary.

Spelling mistakes or flexible sequence order were not taken into account in this test, since they are harder to produce and measure. Unfortunately our test set also did not include phrases where our algorithm would excel. As described in our plans for future work, we attempt to compare our algorithm against other natural language processing techniques on a test set that would include both regular and irregular spelled words and would also include phrases.
VI. FUTURE WORK

The major unaddressed issue in our work is time complexity, which we feel can be improved. Because we developed our algorithm based on a premise of comparing two sequences, we ended up with time complexity $O(n^2)$. When comparing a sequence to a collection of other sequences, trying to find the most similar one, this time complexity can become an issue. In our future work, we propose building a special case of suffix trees best suited for subsequence discovery. Such trees would reduce the time complexity of single sequence comparison against a sequence collection and would allow the development of special algorithm designed to find the most similar match. In course of this work we also plan to test our algorithm against other approaches used in natural language processing. We plan to construct a more suitable test set, which would mimic user input as close as possible. Two major cases that we would like to include in our test set are phrases and spelling errors. We plan to extract phrases out of openly available thesauruses such as EuroVoc[6] and add them to our test set. We also plan to insert custom spelling errors and change the word order in phrases to make the test set more representative. With the help of such a test set we plan to improve our algorithms effectiveness by trying out different algorithm parameter values. We also want to test different ways of determining the best sequence match. In our work we simply chose the sequence with the highest similarity measure, but we think that other factors can have a large effect on the most relevant sequence found. Some of these factors include taking relative distance between the highest similarity measure and its runner up into the account, using information about the sequences difference, which is a side product of the algorithm usage, or applying external knowledge, which holds information about the sequence space we are in, to the similarity calculation.

VII. CONCLUSION

In our work, we presented an alternative approach to string matching. We based our work on sequence analysis and developed measures to evaluate similarity between two sequences. We preformed some test and presented the results on our test set and also pointed out some ideas for our future work. Our current work can be used in natural language processing either as a standalone metric to find the most appropriate matches in corpuses or as an addition to existing comparison techniques in a similar way that spellcheckers are used. It can also be used to perform fuzzy string matching, either in full text searches across databases or as a text search in larger documents. It can also be used to extract phrases or even to compare documents. All of the above can be done with small modifications to the original algorithm and with smart parameters choice. We successfully deployed our algorithm into the advanced search in DKUM [7] making it a fuzzy full text search. We also used the algorithm in the upgraded version of the question answering system described in [8]. We plan to use our algorithm in our next applicative projects and with it's help plan to increase our matching precision and with that improve our users experience.

REFERENCES