An efficient comparison algorithm for natural language sentences

Dario Chiappetta

May 2, 2013

Abstract

Computing a **similarity score** between two natural language **sentences** is not a trivial problem. Many features can be taken into account to solve it, such as the syntactic structure of the sentence, or the meaning of the single words, or their order; all of this features are informative respect to the meaning of the sentence, and it is reasonable to assume they are all considered by humans solving the same task. Here I describe a recursive, dynamic programming **algorithm** for sentence comparison aimed to exploit word-based information, as well as the syntactic one, expressed in the form of unsupervised learned binary trees.

1 Introduction

In the context of sentences **classification**, where the task is to label an unknown input sentence with the correct meaning, and each meaning is defined by a set of representative natural language sentences, one crucial point is to measure the similarity of the input sentence with each of the sentences defining each meaning.

The problem of scoring the similarity between two sentences is not new in the **literature**, and a number of different approaches already exist to tackle it. Achananuparp et al. in [1] classify these approaches in *word overlap* measures (based on the number of words shared by the two sentences), *TF-IDF* measures (based on term frequency-inverse document frequency) and

linguistic measures (based on semantic relations between words and their syntactic composition).

Another source of inspiration for this work is represented by **statistic**, corpus-based methods in Computational Linguistics; a significant example comes from The IBM models for Statistical **Machine Translation** [3], that first introduced the idea of feeding statistically intensive **Machine Learning** algorithms with big data from corpora, which nowadays is the dominant paradigm in MT; insightful is also the work on Data Oriented Parsing, and particularly the U-DOP model for **Unsupervised Language Learning** [2], which core idea is to initially assume all the possible syntax trees for a set of sentences as equally possible, and then use all the possible sub-trees of them to compute the most probable parse trees, letting the structure of the language emerge from the data.

The algorithm presented in this paper aims to provide a similarity measure for two sentences s_1 and s_2 featuring many of the ideas that have proven to be successful in recent developments of Computational Linguistics. The following are some of the key insights of the algorithm:

- **Every possible** sub-sentence of s_1 is compared with every possible sub-sentence of s_2.
- A score between two chunks of text is build **recursively** on the best matches of smaller chunks, and always brought down to scores between single words.
- **Dynamic programming** is used to increase the efficiency of the algorithm, preventing it from computing the same result two times.
- Other than the similarity score, the algorithm outputs the most probable chunking of each sentence in the form of an unlabeled binary **parse tree**.
- Other than the similarity score and the parse trees, the algorithm outputs the most likely **alignment** between the chunks of the two sentences.
- Every score is expressed as a linear combination of **features**, designed to be independent and extensible.
- The partial computations of the algorithm can be stored and further used to implement a learning model.

This paper is **structured** as follows. Section 2 describes the algorithm theoretically. Section 3 walks through a toy example. Section 4 describes possible extensions and improvements. Section 5 draws the conclusion of the paper.

2 The algorithm

Algorithm 1: The main algorithm

input : c_1, c_2 chunks of text; T partial results table
output: $\sigma(c_1, c_2)$ Similarity score between c_1 e c_2

```
1  if c1,c2 are words then
2  |  return τ(c1, c2)
3  S ← [ ];
4  for i ← 1 to Length(c1) do
5  |  C1 ← Split(c1,i);
6  |  for j ← 1 to Length(c2) do
7  |  |  if i=Length(c1) and j=Length(c2) then break;
8  |  |  C2 ← Split(c2,j);
9  |  |  foreach sc1 in C1 do
10 |  |  |  foreach sc2 in C2 do
11 |  |  |  |  if T[sc1, sc2] = ∅ then
12 |  |  |  |  |  T[sc1, sc2] ← σ(sc1, sc2, T)
13 |  |  |  end
14 |  |  end
15 |  |  Append(S,σ̃(C1, C2, T))
16 |  end
17 end
18 return Max(R)
```

Algorithm 1 contains the main loop of the sentence scoring algorithm. Before entering the details of it, it is necessary to say that:

- The input c_1, c_2 can be any **chunks** of text because, while the algorithm is called with two sentences in input, it will recursively call itself on every possible chunk extracted from the two sentences.

- The input T (T for Table) is a data structure holding the intermediate results of the algorithm. As we can infer from line 12 of Algorithm 1, $T[c_1, c_2] = \sigma(c_1, c_2)$
- $\tau(c_1, c_2)$ returns a similarity score between two words. This score is a linear combination of independent features. At the moment, the following features are taken into account:
  - Equality (boolean)
  - Edit distance
  - Difference in position
  - Wordnet Path Length similarity

  The weights of the single features, now static, are meant to be trained through Machine Learning
- `Length(`c`)` returns the number of words contained in the chunk c
- `Split(`c, i`)` returns a list of two chunks, one containing the words of c from the first to the ith, and the other containing the words of c form the $(i+1)$th to the last.

  If the ith word is the last word of c, it returns a list containing the only element c.
- `Append(`L, x`)` is the standard *append* operation for lists, adding the element x in the last position of the list L.
- $\hat{\sigma}(C_1, C_2, T)$ returns the score of c_1, c_2 **given** a particular 2-split of the two chunks. Such a score is computed combining the ones of the smaller constituents of c_1, c_2, which are required to be already present in the table T.

  The score is again the linear combination of independent features, leaving its implementation open and extensible.

  At the moment the only feature being employed is the average of the scores of the best alignments of the sub-chunks in C_1 with the ones in C_2.

  For example, given C_1 = ["increase","the volume"] and C_2 = ["raise", "the volume"], "increase" is likely to achieve the best score with "raise" (eg. T["increase","raise"] = 0.8), as well as "the volume" will be aligned with "the volume" (T["the volume","the volume"] = 1)). The

score of "increase the volume" and "raise the volume", in this particular splitting, will thus be the average of the two, 0.9.

After these premises it is relatively easy to walk through the pseudo-code of Algorithm 1. Lines 1 and 2 handle the base case, that is, when the algorithm is run on **two words**; in this case the result of the Word Similarity function, which is described above, is returned.

At line 3 the list of **candidate results** is initialized as an empty list. This is because more than one score will be computed for c_1 and c_2, the maximum of which will be returned as a result.

Lines 4 and 6 define a nested loop structure, which purpose is to update the two indexes i (from the first to the last word of c_1) and j (from the first to the last word of c_2) to cover **every possible combination** of word positions in the two input strings. These indexes are used to **split** the input chunks in smaller parts (lines 5 and 8). As an example, if the input is given by c_1 ="increase the volume", c_2 ="raise the volume", the outer loop will produce the three possible values of C_1: ["increase","the volume"], ["increase the","volume"] and ["increase the volume"]; each of these values will be combined, in the inner loop, with the three possible values of C_2: ["raise","the volume"], ["raise the","volume"] and ["raise the volume"].

Note that, while it is reasonable to consider the entire c_1 and c_2 in the comparisons (eg. if c_1 ="quit",c_2 ="quit please", we'd want the whole c_1 to be associated with the "quit" sub-part of c_2), it is necessary to prevent the two entire c_1 and c_2 to be associated at the same time, thus producing an **infinite loop**; this is done in line 7.

The next part of the inner loop, from line 9 to line 10 combines every sub-chunk of c_1 ($sc_1 \in C_1$) with every sub-chunk of c_2 ($sc_2 \in C_2$); note that C_1 and C_2 contain either one or two elements. Every combination that is not present in the table is scored with a **recursive** call to the algorithm itself, and the resulting score is saved in the table (line 12).

The last operation done in the loop, at line 15, is to combine the sub-scores that have just been scored in the table to compute the final score of this particular subdivision of c_1 and c_2. This is a **candidate score** for c_1 and c_2, and thus is appended to the list of candidate scores. As we already said, the definition of this combination is open; however it is most reasonable for the $\hat{\sigma}$ function to find the **best alignment** between the input sub-chunks,

and provide a score based on it. It is plain that the definition of $\hat{\sigma}$ is crucial for getting sensible results from the algorithm. Furthermore, $\hat{\sigma}$ is the point where the algorithm can be extended with Machine Learning capabilities, that will be discussed in Section 4.

At line 18 the function returns the **maximum of the candidate** score, that is, the split that produced the best score (eg. for "increase the volume" vs "raise the volume", the score of "increase | the volume" vs "raise | the volume" is likely to produce a better score than "increase | the volume" vs. "raise the | volume"). Note that this decision determines the branching of one level of the recursion tree; once the algorithm is done computing the score between two sentences, a traceback of the selected splits can be interpreted as a parse tree for the two input sentences.

3 An example

4 Machine Learning Possibilities

5 conclusions

References

[1] Palakorn Achananuparp, Xiaohua Hu, Xiajiong Shen. The Evaluation of Sentence Similarity Measures. Lecture Notes in Computer Science Volume 5182, 2008, pp 305-316

[2] Rens Bod. Unsupervised parsing with U-DOP. CoNLL-X '06 Proceedings of the Tenth Conference on Computational Natural Language Learning, 2006, pp 85-92.

[3] P. Brown, S. Della Pietra, V. Della Pietra, and R. Mercer (1993). The mathematics of statistical machine translation: parameter estimation. Computational Linguistics, 19(2), 263-311.