Unsupervised and Knowledge-Free Morpheme Segmentation and Analysis

Stefan Bordag

Natural Language Processing Department, University of Leipzig
sbordag@informatik.uni-leipzig.de

Abstract. This paper presents a revised version of an unsupervised and knowledge-free morpheme boundary detection algorithm based on letter successor variety (LSV) and a trie classifier [1]. Additional knowledge about relatedness of the found morphs is obtained from a morphemic analysis based on contextual similarity. For the boundary detection the challenge of increasing recall of found morphs while retaining a high precision is tackled by adding a compound splitter, iterating the LSV analysis and dividing the trie classifier into two distinctly applied clasifiers. The result is a significantly improved overall performance and a decreased reliance on corpus size. Further possible improvements and analyses are discussed.

Keywords: Letter successor variety, morpheme boundary detection, morpheme analysis, distributed similarity.

1 Introduction

The algorithm presented in this paper[1] is a revised version of the *letter successor variety* (LSV) based algorithm [2,3,4] described and implemented previously [5,1]. The additional component of morpheme analysis is based on a prototypical implementation described in [6].

The morpheme segmentation algorithm attempts to find morpheme boundaries within word forms. For a given input word form it results in a segmentation into morphs (as opposed to morphemes). It is based on the assumption that any grammatical function is expressed with only a small amount of different affixes. For example, plural is expressed with only five different morphs in German *-en, -s, -e, -er* (and zero).

In essence, the algorithm measures the amount of various letters occuring after a given substring with respect to some context of other words (in this case semantically similar ones), weighting that value according to bi- and trigram probabilities and comparing the resulting score to a threshold. Hence it is designed to handle concatenative morphology and it is likely to fail in finding morpheme boundaries in languages with other types of morphology. The algorithm is not rooted in any particular (linguistic) theory of morphology, especially

[1] A recent implementation of this algorithm is available at http://wortschatz.uni-leipzig.de/∼sbordag/

© Springer-Verlag Berlin Heidelberg 2008

since such theories tend to omit the fact that morphemes, as their basic units of interest, are not as simply observable as words. The knowledge about where a morph begins and ends is usually assumed to be given a priori.

The present implementation of the morpheme boundary detection consists of three distinct major parts: a compound splitter, a letter successor variety algorithm using contextual similarity of word forms and a trie based machine learning step. Due to the low performance of the LSV based method in splitting longer words, in a pre-processing step a simple compund splitter algorithm is applied. The LSV part is iterated to increase recall with only a moderate loss of precision. The machine learning part (using a trie) is split into two parts, one with high precision and a subsequent one with high recall.

According to an evaluation using the German Celex [7], each change improves the overall performance slightly. Several possibilities of further improvements and analyses are discussed. Any of the major three parts (compound splitter, LSV algorithm, trie classifier) of the described algorithm can be replaced by or merged with a different algorithm, which should facilitate the combination of this algorithm with others.

The morpheme analysis part is based on statistical co-occurrence of the found morphs and subsequent contextual similarity and a basic rule learning algorithm. The rules are then used to find related morphs where groups of related morphs represent a morpheme.

2 Letter Successor Variety

LSV is a measure of the amount of different letters encountered after (or before) a certain substring, given a set of other strings as context. It is possible to use the entire word list as context for each string and its substrings [3,4]. Alternatively, only a specific set of words may be used as context [1], if a method for the selection of relevant words is included. In order to use LSV to find true morpheme boundaries, this set ideally consists of words that share at least one grammatical feature with the input word. For example, if the input word is *hurried*, then relevant words are past tense forms. It is obvious that in such a case the amount of different letters encountered before the substring -*ed* is maximized.

As has been shown earlier [6], using the entire word list for morpheme boundary detection (global LSV) is inferior to using a simulation of semantic similarity (contextual similarity based on comparing statistically significant co-occurrences) of words to find the relevant ones (local LSV). However, the power-law distribution of word frequencies makes it impossible to compute a proper representation of their usage and accordingly compare such words for usage similarity. Hence, local LSV based morpheme boundary detection might have a high precision, but is guaranteed to have a low recall. Another related method, first globally finding the contextually most similar word pairs and then analyzing their differences [8], appears to have even lower recall than the LSV method.

2.1 Trie Classifier

In order to increase the recall of the local LSV method, a machine learning method was proposed. It is based on training a patricia compact trie (PCT) [9] with morpheme segmentations detected by the local LSV methods. The trained trie can then be used to recursively split all words into morphs, irrespective of their frequency.

Training the trie, as depicted in Figure 1, is performed as follows: Each known morpheme boundary is reformulated as a rule: The entire word is the input string, whereas the shorter half of the word (according to the morpheme boundary) is the class to be learned. The trie learns by adding nodes that represent letters of the word to be learned along with increasing the count of the class for each letter (see Figure 1). With multiple boundaries within a single word form, training is applied recursively, taking the outmost and shortest morphs first (from right to left).

Two distinct tries, a **forward-trie** and a **backward-trie** are used to separately learn suffixes and affixes. The decision which trie to use for any given training instance is based on the length of the morphs. The longer half of the word probably contains the stem, whereas the shorter half is used as the class. In the case of the backward-trie, the word itself is reversed.

The classification is applied recursively as well: For an input string both the backward and forward tries are used to obtain the most probable class. This results in up to two identified morpheme boundaries and hence three parts of the original words. Each part is analyzed recursively in the same way as the entire word form until no further classifications can be found.

In the Morpho Challenge 2005 [10], both the local LSV and a subsequent application of the trie learning were submitted separately. As expected, the LSV method had a high precision, but extremely low recall (only 1.9% for Finnish, for example). The application of the trie increased recall, but also lowered precision

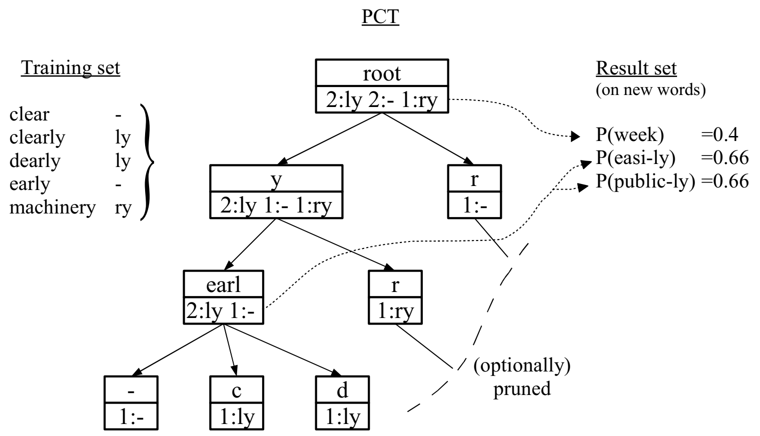

Fig. 1. Illustration of training a PCT and then using it to classify previously unseen words

due to overgeneralization. Overgeneralization occurs mostly because of missing negative examples. Since even for a well-represented input word not all contextually similar words share grammatical information with it, it is impossible to take words without found boundaries as examples of words that in fact do not have any morpheme boundaries.

3 Refined Implementation

The above mentioned weaknesses of the LSV + trie combination hold uniformly over all tested languages. The following modifications attempt to address some of these weaknesses, while trying to avoid language-specific rules or thresholds. The new version contains several changes: a recursive compound identifier, an iteration of the LSV algorithm and splitting the trie classification into two steps.

3.1 Identifying Compounds

The LSV algorithm is based on using contextually similar words. However, compounds usually are less frequent, and words contextually similar to a compound do not necessarily contain other compounds, or compounds sharing parts with the input word. Particularly for semantically opaque compounds this is almost guaranteed to be the case. Therefore it is mostly impossible for the LSV algorithm to find morpheme boundaries between the parts of a compound, unless the compound contains a very productive part.

Since only a small sample set is sufficient for the trie to correctly classify most compounds later, it is not necessary to find all compounds at this point. The compound splitter is therefore based on simply trying to divide a given word $word$ at a position i and testing whether that division seems plausible. The function $testDiv(word, i)$ then tests the plausibility and returns a score. The division is plausible if both parts of the hypothetical decomposition exist as words in the underlying corpus, and reach a threshold of minimum length (4) and a threshold of minimum frequency (20). If that is the case, the score is the sum of the frequencies of the parts assumed to be words.

It is then possible to take the one partition of the input word that maximizes the frequency of the participating parts. This mechanism is applied to recursively divide a long word into shorter units. Table 1 shows that the algorithm (as expected) has a high precision, but it also has a very low recall. In fact, it may have even lower recall for other languages. It also shows that training the trie classifier with this data directly indeed increases recall, but also incurs a rather strong loss in precision. It can be assumed that if compounding exists in a language, then this algorithm in combination with the trie classifier helps to find the parts of a large part of compounds. However, a more elaborate implementation is desirable at this point, especially since this algorithm does not take linking elements into account.

3.2 Iterated LSV Algorithm

For the LSV algorithm, the ideal case is achieved when all contextually similar words to a given input word carry the same grammatical information. However, due to data sparseness, compounds, overly high co-occurrence frequency and other factors, this ideal state is achieved only for few words. In many cases only a few contextually similar words actually share grammatical information with the input word. Running the LSV algorithm may thus find some morpheme boundaries correctly and not find many others. It is important that in this setup (using contextually similar words as context) it nearly never finds wrong morpheme boundaries, if it does find any, see also Table 1 which shows that the first run of the LSV algorithm found very few (but very precisely) morpheme boundaries.

In order to facilitate the boundary identification for some of the remaining words, it is possible to iterate the LSV algorithm by incorporating knowledge produced in earlier iterations. This is done by an additional factor to the computation of the LSV score: Given a substring *-ly* of the input word *clearly*, if the same substring was identified as a morph in any (or all) of the contextually similar words, then increase the LSV score. However, some very frequent words such as *was* or *do-es* are contextually similar to a large amount of words, which in turn means that these frequent words might influence the analyses of many other words adversely, such as *James* to *Jam-es*. Therefore the increase of the LSV score is normalized against the number of words with the same substring and the number of contextually similar words.

To recall from [6], the formula to compute the left LSV score for the word w at the position i (the formula for the right score is likewise) is:

$$lsv_l(w,i) = plsv_l(w,i) \cdot fw_l(w,i) \cdot ib(w,i) \tag{1}$$

This takes anomalies such as phonemes represented my several letters into account. Here $plsv_l(w,i)$ is the plain number of different letters found to the right of the substring between the beginning of the word w and the position i. $fw_l(w,i)$ is the bi- or trigram based frequency weight of the substring, whereas $ib(w,i)$ is the inverse bigram weight. The previously acquired knowledge about morpheme boundaries is used to compute $prev_l(w,i)$ as the number of previously identified morphs $pf_l(w,i)$ divided by 2 and multiplied with the quotient of the number of words containing the same substring $subf_l(w,i)$ and the size of the pruned list of contextually similar words $prune$:

$$prev_l(w,i) = pf_l(w,i) \cdot 0.5 \cdot (subf_l(w,i)/prune) \tag{2}$$

To prevent the previous analyses from overriding the analysis of the present word, the new LSV score is computed as a multiplication of the LSV score with the previous knowledge, which is at most as high as $lsv_l(w,i)$ -1:

$$lsv2_l(w,i) = min(lsv_l(w,i) - 1, prev_l(w,i)) \cdot lsv_l(w,i) \tag{3}$$

The same is reversely applied to the right LSV score $lsv_r(w,i)$ and both $lsv_l(w,i)$ and $lsv_r(w,i)$ are summed to produce the final $lsv2(w,i)$ and compare it to a threshold (for example 6) to obtain a decision whether the position i in the word w is a morpheme boundary.

For example, the analyses of the most similar words of *clear-ly* might result in the following morpheme boundaries: *closely, white, great-ly, legal-ly, clear,* *linear-ly, really, weakly, ...*. Hence, for the position 5 (which corresponds to *-ly*) in *clearly*, the amount of previously identified morphs $pf_r(w, i)$ is 3. The number of such substrings $subf_l(w, i)$ is 5 and the amount of contextually similar words was 150. Hence, $prev_r(clearly, 5) = 3 \cdot 0.5 \cdot (5/150) = 0.05$ and thus the absolute increase of the LSV score is only 0.05 in this case.

Table 1 shows that there are many cases where the influence was sufficiently strong for the resulting LSV score to reach the threshold. It also shows that iterating the LSV algorithm increases Recall. However, it also incurs a certain Precision loss due with words such as *James* being contextually similar to many other words where *-es* is really a suffix.

Table 1. Iterating the LSV algorithm and applying the modified trie classifier increases recall while keeping precision at high levels

				recursive pretree		
	R	P	F	R	P	F
compounds	10.30	88.33	18.44	27.93	66.45	39.33
lsv_iter_1	17.88	88.55	29.76	57.66	71.00	63.64
lsv_iter_3	23.96	84.34	37.31	62.72	68.96	65.69
saveTrie	31.09	82.69	**45.19**	66.10	68.92	**67.48**

3.3 Split Trie Classification

Irrespective of its source, knowledge about boundaries is used to train the trie classifier and then apply the trained classifier to identify more morpheme boundaries. In the original version the trie produces a most probable class for an input string simply by searching for the deepest node in the trie. This mean that often decisions were made without considering further context. For example, the LSV algorithm found the morpheme boundary *drama-tic*. When analyzing *plas-tic*, the trie classifier would find *t* as the deepest matching node. Since that node has only a single class stored with the frequency count of 1, the classifier would decide in favor of *-tic* being a morph with a maximal probability of 1. No further context from the word is considered and the decision is made on grounds of only a single training instance.

However, simply forbidding all decisions that do not take a certain amount of the word into account, would result in extremely low recall, such as 31% for German in Table 1. The trie classification is thus split into two parts, a modified trie classifier and subsequently an original unmodified trie classifier. The modified trie classifier returns a decision only if all of the following conditions are met:

– The deepest matching node must be at least two letters deeper than the class to be returned.
– The matching node must have a minimal distance of three from the root of the trie.
– The total sum of the frequency of all classes stored in the deepest matching node must be larger than 5.

Table 1 shows that applying the modified trie classifier *saveTrie* increases recall by 8% while reducing precision by less than 2%. It also shows that the subsequent application of the original trie classifier further increases recall to a total of 66% while lowering precision to roughly 69%. The table also shows that applying the original trie classifier directly on any of the LSV iterations or even the compound identification algorithm results in lower overall performance.

3.4 Assessing the Improvements

In order to measure the influence of the various improvements proposed, a number of experiments were run on the 3 million sentences German corpus available for the Morpho Challenge 2007. The results of each improvement were measured and are depicted in Table 1. Additionally, the original trie classifier was applied to the results of each modification.

These evaluations show that ultimately, the local LSV implementation could be significantly improved. As such, it reaches similar performance as reported in [1], despite being run on a significantly smaller corpus (3 million sentences vs. 11 million). On the other hand, the relatively small improvements achieved indicate that a significantly better morpheme boundary detection may only be achieved by combining this method with an entirely different approach.

The results of the Morpho Challenge 2007 also show that currently the MDL based approaches to morpheme boundary detection [11,12] mostly outperform the LSV based approach, especially in the more important Information Retrieval task evaluation. The most probable reason is that the LSV algorithm is good at detecting boundaries within high-frequent words, whereas the MDL based algorithms are better at detecting boundaries in longer words. Longer words tend to be less frequent and thus more important for Information Retrieval as opposed to the more frequent words.

A manual analysis of the resulting word list revealed several possible improvements:

- An algorithm specifically designed to identify compounds and take the existence of linking elements into accounts, for example by means of finding reformulations.
- In a post-processing step, an algorithm based on affix signatures such as proposed by [13], might find errors or generalize known morpheme boundaries better than the trie classifiers and ultimately avoid mistakes such as *in-fra-struktur*.
- A global morpheme vocabulary control mechanism, such as the MDL [14,15,11,12] might provide further evidence for or against certain morpheme boundaries and subsequently inhibit mistakes such as *schwa-ech-er*.

4 Morpheme Analysis

Under the assumption that morpheme boundaries were correctly detected, it is possible to treat every single morph separately (similarly to a word) in a statistical co-occurrence analysis. This allows computing contextual similarity between morphs, instead of words. The following algorithm uses this procedure to find rules that relate various morphs to each other and then applies these rules to produce morphemic analyses of the words that originally occurred in the corpus:

```
for each morph m
 for each cont. similar morph s of m
  if LD_Similar(s,m)
   r = makeRule(s,m)
   store(r->s,m)

for each word w
 for each morph m of w
  if in_store(m)
   sig = createSignature(m)
   write sig
  else
   write m
```

For each morph, the function *LD_Similar(s,m)* filters from the contextually most similar morphs those that differ only minimally, based on Levenshtein Distance (LD) [16] and word lengths. This step could be replaced by a more elaborate clustering mechanism. Pairs with short morphs are only accepted if $LD = 1$, pairs with longer morphs may have a larger distance. The function *makeRule(s,m)* creates a hypothetical rule that explains the difference between two contextually similar morphs. For example, the morphs *ion* and *ions* have a Levenshtein Distance of 1 so the function creates a rule *_-s* (or *n_-ns* to take more context into account) which says that *s* can be added to derive the second morph from the first one. This rule is then stored and associated with the pair of morphs that produced it. This allows deciding between probably correct (if many morph pairs are associated with it) and incorrect rules later.

The second part of the morphemic analysis then applies the acquired knowledge to the original word list. The goal is an analysis of the morphemic structure of all words, where a morpheme is represented by all its allomorphs. In the first step, each word is thus split into its morphs, according to the LSV and trie based algorithm described above. In the next step, all related morphs as stored by the first part of the morphemic analysis are retrieved for each morph of the input word. The function *createSignature(m)* produces a representation of each morpheme. For example, the original word *fracturing* was found to have two morphs: *fractur* and *ing*. The first morph is related to two morphs *fracture* and *fractures*. The second morph is related to *inag, ingu* and *iong*. This results in the following analysis:

```
fracturing
 > fractur.fracture.fractures
 > inag.ing.ingu.iong
```

It is noteworthy that this algorithm cannot distinguish between various meanings of a single morph. In English, the suffix *-s* may be a plural marker if used with a noun or the third person singular marker if used with a verb. Given the extremely high frequency of some these ambiguous morphs, the number of (at least partially) wrong analyses produced by the algorithm is likely to be high. Further research may evolve around using an unsupervised POS tag inducer [17] to distinguish between different word classes or using a word sense induction algorithm [18] applied to morphs in order to induce the various meanings.

The results from the Morpho Challenge 2007 are surprising in that the morpheme analysis did not yield any significant changes to the evaluation results. This is despite the fact that on average nearly every single morpheme is represented by several morphs. After exploring the word lists for German, the most probable reasons for this appear to be any of the following:

– During construction of the rules no context is taken into account. This often results in morphs to be found as correlated despite them just incidentally looking similar and sharing some contextual similarity. Hence, benefit of the analysis and error might be cancelling each other out.
– Many of the morphs representing a morpheme are, in fact, only artifacts of the mistakes of the morpheme boundary detection algorithm. Thus, the morpheme analysis appears to be strongly influenced by the quality of the detected boundaries.
– When determing the validity of a rule, the amount of morph pairs is taken into account, but not their frequency. This results in many extremely rare morphs (without any impact on the evaluation) to be merged correctly into morphemes, but many very frequent ones (with actual impact on the evaluation) to be missed.

5 Conclusions

Whereas the changes introduced to the morpheme boundary detection improve the overall performance, they also add several more parameters to the entire process. The paramaters do not have to be set specifically for each language, but a large number of parameters often indicates the possibility of overfitting. Yet, despite the improvements and the possibility of overfitting, the performance of knowledge-free morpheme boundary detection is far below what knowledge-rich systems (i.e. rule-based) achieve. Nevertheless, the significant beneficial effects achieved in the Information Retrieval evaluation task in the Morpho Challenge 2007 sufficiently demonstrate the usefulness of such algorithms even in the current state.

Compared to other knowledge-free morpheme boundary detection algorithms, the version of the LSV algorithm described in this paper produces good results. The modular design of this algorithm allows for a better interoperability with other algorithms. For example, the significant performance boost achieved by adding a compound splitter indicates that combining various underlying hypotheses is more likely to yield significant improvements than changes to any single method. Also, given that the most simple combination of algorithms in the form of a voting algorithm in the Morpho Challenge 2005 demonstrated an extraordinary increase in performance, it is reasonable to assume that more direct combinations should perform even better.

The noise produced during the morpheme boundary detection, the missing method for distinguishing ambiguous affixes and other factors resulted in the subsequent morphemic analysis to produce apparently insignificant results. It becomes obvious that adding further algorithmic solutions representing other hypotheses about morpheme boundaries, as well as a more elaborate morphemic analysis, should be a significant step towards a true morphemic analysis similarily to what can be done manually.

References

1. Bordag, S.: Two-step approach to unsupervised morpheme segmentation. In: Proceedings of the PASCAL Challenges Workshop on Unsupervised Segmentation of Words into Morphemes, Venice, Italy (April 2006)
2. Harris, Z.S.: From phonemes to morphemes. Language 31(2), 190–222 (1955)
3. Hafer, M.A., Weiss, S.F.: Word segmentation by letter successor varieties. Information Storage and Retrieval 10, 371–385 (1974)
4. Déjean, H.: Morphemes as necessary concept for structures discovery from untagged corpora. In: Powers, D. (ed.) Workshop on Paradigms and Grounding in Natural Language Learning at NeMLaP3/CoNLL 1998, Adelaide, Australia, pp. 295–299 (January 1998)
5. Bordag, S.: Unsupervised knowledge-free morpheme boundary detection. In: Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP), Borovets, Bulgaria (September 2005)
6. Bordag, S.: Elements of Knowledge-free and Unsupervised lexical acquisition. PhD thesis, Department of Natural Language Processing, University of Leipzig, Leipzig, Germany (2007)
7. Baayen, R.H., Piepenbrock, R., Gulikers, L.: The CELEX lexical database (CD-ROM). Linguistic Data Consortium, University of Pennsylvania, Philadelphia (1995)
8. Schone, P., Jurafsky, D.: Knowledge-free induction of inflectional morphologies. In: Proceedings of the 2nd Annual Meeting of the North American Chapter of Association for Computational Linguistics, Pittsburgh, PA, USA (2001)
9. Morrison, D.R.: Patricia - practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM 15(4), 514–534 (1968)
10. Kurimo, M., Creutz, M., Varjokallio, M., Arisoy, E., Saraclar, M.: Unsupervised segmentation of words into morphemes - Challenge 2005 An Introduction and Evaluation Report. In: Proceedings of the PASCAL Challenges Workshop on Unsupervised Segmentation of Words into Morphemes, Venice, Italy (2006)
11. Creutz, M., Lagus, K.: Unsupervised morpheme segmentation and morphology induction from text corpora using morfessor 1.0. In: Publications in Computer and Information Science, Report A81, Helsinki, Finland, Helsinki University of Technology (March 2005)
12. Bernhard, D.: Unsupervised morphological segmentation based on segment predictability and word segments alignment. In: Proceedings of the PASCAL Challenges Workshop on Unsupervised Segmentation of Words into Morphemes (2006)

13. Goldsmith, J.: Unsupervised learning of the morphology of a natural language. Computational Linguistics 27(2), 153–198 (2001)
14. de Marcken, C.: The unsupervised acquisition of a lexicon from continuous speech. Memo 1558, MIT Artificial Intelligence Lab (1995)
15. Kazakov, D.: Unsupervised learning of word segmentation rules with genetic algorithms and inductive logic programming. Machine Learning 43, 121–162 (2001)
16. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR 163(4), 845–848 (1965)
17. Biemann, C.: Unsupervised part-of-speech tagging employing efficient graph clustering. In: Proceedings of the Student Research Workshop at the COLING/ACL, Sydney, Australia (July 2006)
18. Bordag, S.: Word sense induction: Triplet-based clustering and automatic evaluation. In: Proceedings of 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL), Trento, Italy (April 2006)