PyPortS

Svetlana Tchistiakova

1 Introduction

For morphemically rich languages, gaining access to morpheme boundaries provides key knowledge for further linguistic analysis. However, hand annotating large text corpora with morpheme boundaries is time consuming. Furthermore, each new language would require its own hand-annotated corpus, which will necessarily become tied to a snapshot of the language at a given time, becoming out of date as the language changes. Deriving morpheme boundaries in an automatic fashion using an unsupervised method (that is, without the need to provide previously annotated data) could save months of work for the linguist, and provide a way to analyze new languages and dialects as they emerge.

As a morphologically rich language, Russian is a prime candidate to work with for unsupervised morpheme induction, because knowledge of morphemic context would provide benefits for many other applications. For example, since the same affixes are applied systematically to the same parts of speech, part of speech tagging is simplified by morphemic knowledge. Phonology modeling is another area where morphemic knowledge can provide insight, since processes such as stress shift occur systematically across certain morphological root-affix boundaries. Finally, the large number and variety of affixes in Russian make it impractical to store each word form in a lexicon, while the process of lemmatization (grouping the various word forms into their intended meaning) requires knowledge of morpheme boundaries.

Keshava and Pitler (2006) implemented an algorithm in Perl for unsupervised morpheme induction. [1] They focused on English in their implementation, and saw good results. They also tested it on Turkish and Finnish, but their results were a little worse. The goal of this work was to re-implement their algorithm in Python3 with Russian, but the final PyPortS program was also tested on English and Japanese.

2 RePortS algorithm

Keshava and Pitler's (K&P) algorithm, which they call "RePortS", was written in Perl and consisted of the following four steps. The steps will be explained further below, in particular in those places where the current implementation deviates from Keshava and Pitler's implementation (the reader is advised to read the original paper for more details about K&S's implementation).

1. Build trees with probabilities based on the corpus
2. Score word fragments using these trees to obtain a large list of morphemes
3. Prune this list of morphemes

4. Segment the test words using the morpheme list and lexicographic trees

The PyPortS implementation breaks the steps down into two Python3 classes: `Tree`, and `Segmenter`. The `Tree` is an implementation of the prefix/suffix tree structure described by Keshava and Pitler in Step 1. The `Segmenter` implements the scoring and pruning of Steps 2-4. There is an additional `Evaluator` class (not described by K&P), that is used to calculate precision/recall/fscore on a development data set, for which a gold standard was manually created. Some additional utility functions for batch segmentation, testing, and argument parsing are also included, and the `pickle` package was also used to save/load trained models.

2.1 Step 1: Building trees

In Step 1, trees are built by traversing all the words in the corpus and building symmetric "forward" (prefix) and "backward" (suffix) trees. This is a b-way tree of depth d, where b are the letters and d is the length of the longest word. A path from the root to some node spells out the starting or ending fragment of some word. The node itself contains the count of the letter, at that depth.

In this implementation, once the tree has build built, the program recursively traverses it again and normalizes the tree at each depth by the count of items at that depth, thus creating transition probabilities for each letter. The chain probability of the string up to that point is also recorded to avoid having to loop again later. The probabilities are converted into log base 2.

2.2 Step 2: Scoring fragments

Scoring word fragments proceeds as follows.

If $aABb$ is a word in the language, then Bb is a suffix if

$$aA \text{ is also a word in the corpus} \tag{1}$$

$$P_f(A|a) \approx 1 \tag{2}$$

$$P_f(B|aA) < 1 \tag{3}$$

where P_f is the probability of this letter given the previous letters as calculated by the forward tree.

Similarly aA is a prefix if

$$Bb \text{ is also a word in the corpus} \tag{4}$$

$$P_b(B|b) \approx 1 \tag{5}$$

$$P_b(A|Bb) < 1 \tag{6}$$

where P_b is the probability of this letter given the following letters as calculated by the backward tree.

For both prefixes and suffixes, the first condition comes from the idea that affixes tend to attach to actual words. For example, in English, "corresponded" may be a word in the corpus, and the expectation is that removing the affix would produce another word "correspond" that is in the corpus.

This condition proves problematic for Russian, in which many stems without an affix are not usually considered grammatical word forms, e.g. nouns must have a case affix (nominative for the citation form), verbs must have a tense (infinitive for the citation form), adjectives occur with noun agreement affixes (singular masculine for the citation form), and so on. Because of this reason, this condition was actually removed from PyPortS.

The second condition corresponds to the intuition that the order of letters inside of a stem is fairly predictable, so the probability within stems should be high.

The third condition corresponds to the intuition that a stem can take many different affixes, hence at stem-affix transitions, there will be a lower probability that any particular letter will occur next.

K&P don't specify how close to 1 is "≈ 1" in the second condition. In addition, although they say that the probability in the third condition should be < 1, most transition probabilities at this point will be < 1 for an alphabet-based language, since it is very rare for no words to share at least some similar endings/beginnings, even if those endings/beginning do not point to the existence of an affix.

As such, in PyPortS, instead of using a probability of 1 in for the second and third conditions, a range of "threshold" probabilities were tested. The threshold probabilities are used when the `Segmenter` is traversing the a word letter-by-letter to decide if a split should be made. The "parent threshold" is the probability that the current letter should stay glued to its parent, and the "child threshold" is the probability that the child of the current letter should break off from the current letter.

Four threshold probabilities were searched for in a word $aABb$, reflecting the four conditions:

1. prefix parent threshold: $P_b(B|b)$
2. prefix child threshold: $P_b(A|Bb)$
3. suffix parent threshold: $P_f(A|a)$
4. suffix child threshold: $P_f(B|aA)$

The threshold testing was performed on a mini training set (for faster runtime), and the best result was chosen to train on with the larger data set. This was done by setting three of the thresholds to 1.0 and varying the fourth threshold in increments of 0.1 until it produced the best F-score on the test set.

For training the `Segmenter`, training corpora were built consisting of one word per line. Words from the training corpus are lowercased as they are read in, and those words that have symbols apart from letters or the dash (-) are ignored.

Each word is traversed letter-by-letter in search of affixes. Once an affix is found that meets the necessary conditions, its score is increased by 19. Each potential affix that does not meet the above conditions has its score decreased by 1. This is to ensure that an affix passes the tests $> 5\%$ of the time. The affixes are added to two different dictionaries: one containing their scores, and another containing their chain probabilities.

2.3 Step 3: Pruning the morpheme list

Pruning the morpheme list first involves removing any affixes that score below 0, since they aren't needed. Next, any prefix/suffix that is a concatenation of two higher scoring prefixes/suffixes gets pruned. This follows from K&P's observation that words such as "throwers" might be incorrectly split as "throw+ers" instead of the correct "throw+er+s", if the "ers" remains in the dictionary.

2.4 Step 4: Segmenting words

After the `Segmenter` has been trained in steps 1-3, it can be used to predict morpheme boundaries in new words.

K&P observe that in this step words such as "politeness" are difficult to segment because it is not immediately clear to the program whether the segmentation should be "polite+ness" or "politenes+s". In addition, "polite+ness" can be contrasted with "activate+s;" the last "s" should be handled differently in these two cases.

PyPortS follows K&P's observation that since a morpheme can attach to many differet roots, the probability between its last letter and the stem's first letter will be low. So the prefix/suffix with the lowest chain probability (as recorded in the dictionaries created during training) is peeled off the end of the word first, and the process is repeated until all letters have been consumed. (Note that longer affixes may be at a disadvantage here due to the chain probability.)

Segmentation returns an ordered list of morphemes in the word, which can then be sent to the `Evaluator` for evaluation.

3 Evaluation

The `Evaluator` is used for comparing the results of the segmentation to a gold standard. This requires a test corpus, consisting of one word per line, and a gold standard file consisting of one word per line, with morpheme breaks marked by a plus sign `+`. The `Evaluator` calculates the precision, recall, and fscore of the morpheme breaks in the corpus as a whole; that is, the results

are a fraction of the morpheme breaks in the entire corpus (not a fraction of the words in the corpus).

4 Data

Although the goal of this project was to extend the RePortS algorithm to Russian, a few other data sets were tested on as well.

The Russian data set was constructed partly from SynTagRus [2] and partly from the Orthographic Russian Dictionary (scraped from the Internet) [3]. The training corpus consisted of 233571 types. The testing corpus only used part of the SynTagRus development set and consisted of 207 hand-annotated words.

An English data set was constructed from data available in `nltk` (also available online). [4] [5] [6] The training data consisted of 316394 types, and the testing corpus consisted of 606 hand-annotated words.

Two Japanese data sets were graciously provided by Dana Ruiter. The first was written in kanji (native character-based script). The second was the same data transcribed into kunrei (a romanization system for Japanese). They were originally built up from Japanese short stories. [7] The training corpus consisted of 10839 lines of text (treated as words), and the testing corpus consisted of 204 words hand-annoted by Ms. Ruiter.

5 Results

K&P report results for precision, recall, and F-score on English, Turkish, and Finish in their original paper:

Language	Precision	Recall	F-score
English	82.84	79.10	80.92
Turkish	72.68	43.01	54.04
Finish	83.76	32.30	46.62

Table 1: Keshava & Pitler's RePortS algorithm performance.

Note the big drop in recall for Turkish and Finish in their results. A lower recall suggests that the RePortS algorithm failed to find many of the morpheme boundaries in the testing data. English is more isolating than Turkish and Finish, which both have more overt morphology. Because of this, it's not surprising that the algorithm would have lower recall, failing to find many of the extra affixes stacked on the Turkish and Finish words.

The performance of PyPortS on the English data set was much worse than RePortS performance. This may partly be because of the removal of the condition that a suffix can be peeled off only if the stem also appears in the corpus. It may also be that K&P had a better way of picking the thresholds, or perhaps some other implementation detail that they did not mention in their paper that would give better results. PyPortS was tested on the datasets described above. The results, as well as the best probability thresholds that PyPortS found are reported in the tables below.

Language	Precision	Recall	F-score
English	20.44	42.23	34.85
Russian	22.99	48.18	39.52
Japanese Kanji	65.71	22.33	25.75
Japanese Kunrei	58.38	29.27	32.51

Table 2: PyPortS performance on data sets.

A high precision suggests that when the algorithm makes a split, it is often in the correct place. A high recall suggests that the algorithm finds all the splits it is supposed to.

It is interesting that the Japanese data sets have a high precision and low recall, while the English and Russian datasets have the opposite. Perhaps Japanese uses less affixes than the other languages but they are highly productive. In this case, it is difficult to make conclusions from the small data set and the small development set available.

On the other hand, for English and Russian, the recall was quite a bit higher than the precision. In general, for these languages datasets, PyPortS decided to split all over the place. Of course it found a lot of the splits it was meant to, hence the higher recall, but it also found a lot of unnecessary splits, hence the low precision.

It's not surprising that the algorithm had trouble making these splits just from letter transition probabilities. It had to make some difficult decisions in both English and Russian. For example, it had to decide which splits would be best in words like "whole+sale+r" vs "whole+sal+er" (it chose "w+holes+al+er"), what to do with stems that have an orthographic change like in "busi+ness" (it chose "busin+ess"), or how to split up a word with more affix than root such as "пут+е+ше+ств+ен+ник+а" (it predicted "п+уте+ше+ств+енни+ка").

During training, PyPortS swept over the different threshold values in search of the ones that maximize the F-score:

Language	prefix parent	prefix child	suffix parent	suffix child
English	0.7	0.7	0.4	0.1
Russian	0.1	0.7	0.1	0.1
Japanese Kanji	0.1	0.6	0.2	0.1
Japanese Kunrei	0.7	0.9	0.7	0.1

Table 3: The best thresholds PyPortS found.

The prefix child threshold is high, suggesting that PyPortS wanted to limit the amount of prefixes that were split off. On the other hand, it was perfectly happy splitting every suffix off that it could, hence the low suffix child threshold.

The parent thresholds varied a lot by language. Where they are low, it suggests that PyPortS couldn't predict that well from just the unigram transition probabilities if a letter should be glued to its parent or not, as is the case for Russian. This is unsurprising, because often times in Russian, even just one letter can be considered an affix, and the root might exhibit changes depending on what affix is attached to it.

6 Conclusion

In general, the PyPortS implementation of Keshava &Pitler's RePortS algorithm did not have very good performance, in particular on English, on which K&P reported very good performance. It's not certain whether this is due to ignoring the condition that an affix cannot be peeled off if the stem is not also in the corpus, or due to a missing implementation detail that was not described in the original paper (perhaps a difference in the data set used or the way in which the probability thresholds are dealt with), or if it is due to the inherent limitation of trying to work with only transition probabilities.

Future work on this algorithm could include extending the threshold search to a wider variety of thresholds (rather than keeping three thresholds fixed while testing the last one) or adding the stem condition back in for more isolating language like English.

References

[1] Samarth Keshava and Emily Pitler. A simpler, intuitive approach to morpheme induction. In *Proceedings of 2nd Pascal Challenges Workshop*, pages 31–35, 2006.

[2] Syntagrus. https://github.com/UniversalDependencies/UD_Russian-SynTagRus.

[3] Russian orthographic dictionary. http://dazor.narod.ru/russkie/slovari/orfograficheskij/orfograficheskij-slovar-a.htm.

[4] 850 english words: C.K. Ogden in the ABC of basic english. http://en.wikipedia.org/wiki/Words_(Unix), 1932.

[5] Conll 2000 chunking data. http://cnts.uia.ac.be/conll2000/chunking/, 2000.

[6] The carnegie mellon pronuncing dicitonary. https://cmusphinx.svn.sourceforge.net/svnroot/cmusphinx/trunk/cmudict/cmudict.0.7a.

[7] Aozora. http://www.aozora.gr.jp.